

Machine Learning System Design Interview

Ali Aminian · Alex Xu

Contents

01	Introduction and Overview	1
02	Visual Search System	31
03	Google Street View Blurring System	53
04	YouTube Video Search	71
05	Harmful Content Detection	93
06	Video Recommendation System	115
07	Event Recommendation System	145
08	Ad Click Prediction on Social Platforms	171
09	Similar Listings on Vacation Rental Platforms	195
10	Personalized News Feed	211
11	People You May Know	231

01

Introduction and Overview

We have written this book because we want to help machine learning (ML) engineers and data scientists to succeed at ML system design interviews. This book can also be helpful for those who want to achieve a high-level conception of how ML is applied in the real world.

Many engineers think of ML algorithms such as logistic regression or neural networks as the entirety of an ML system. However, an ML system in production involves much more than just model development. ML systems are typically complex, consisting of a number of components, including data stacks to manage data, serving infrastructure to make the system available to millions of users, an evaluation pipeline to measure the performance of the proposed system, and monitoring to ensure the model's performance doesn't degrade over time.

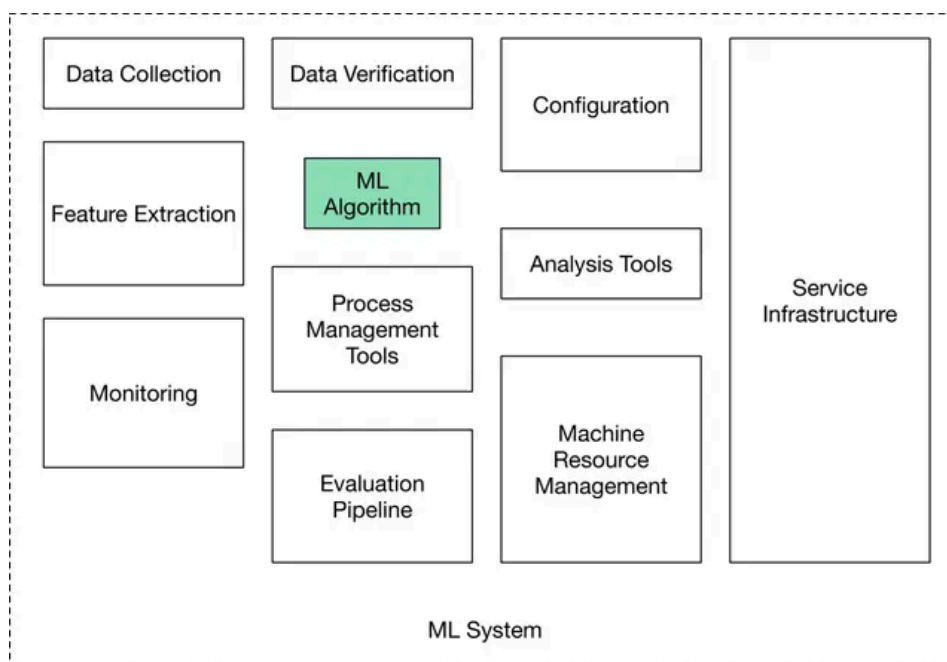


Figure 1.1 Components of a production-ready ML system

At an ML system design interview, you are expected to answer open-ended questions. For example, you might be asked to design a movie recommendation system or a video search engine. There is no single correct answer. The interviewer wants to evaluate your thought process, your in-depth understanding of various ML topics, your ability to design an end-to-end system, and your design choices based on the trade-offs of various options.

To succeed in designing complex ML systems, it is very important to follow a framework. Unstructured answers make the flow difficult to follow. In this introduction, we propose a framework that is used throughout this book to

tackle questions about ML system design. The framework consists of the following key steps:

1. Clarifying requirements
2. Framing the problem as an ML task
3. Data preparation
4. Model development
5. Evaluation
6. Deployment and serving
7. Monitoring and infrastructure

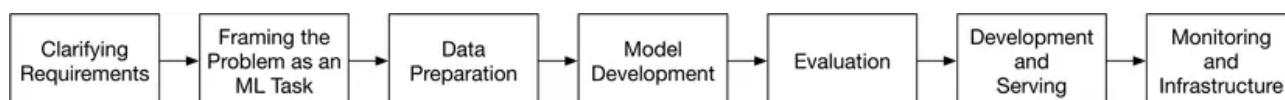


Figure 1.2: ML system design steps

Every ML system design interview is different because the problem is open-ended, and there's no one-size-fits-all way to succeed. The framework is intended to help you structure your thoughts, but there's no need to follow it strictly. Be flexible. If an interviewer is mainly interested in model development, you should almost always adhere to their focus.

Let's start by going over each step in the framework.

Clarifying Requirements

ML system design questions are usually intentionally vague, with the bare minimum of information. For example, an interview question could be: "design an event recommendation system". The first step is to ask clarifying questions. But what kind of questions to ask? Well, we should ask questions to understand the exact requirements. Here is a list of categorized questions to help us get started:

- **Business objective.** If we are asked to create a system to recommend vacation rentals, two possible motivations are to increase the number of bookings and increase the revenue.
- **Features the system needs to support.** What are some of the features that the system is expected to support which could affect our ML system design? For example, let's assume we're asked to design a video recommendation system. We might want to know if users can "like" or "dislike" recommended videos, as those interactions could be used to label training data.
- **Data.** What are the data sources? How large is the dataset? Is the data labeled?
- **Constraints.** How much computing power is available? Is it a cloud-based system, or should the system work on a device? Is the model expected to improve automatically over time?
- **Scale of the system.** How many users do we have? How many items, such as videos, are we dealing with? What's the rate of growth of these metrics?
- **Performance.** How fast must prediction be? Is a real-time solution expected? Does accuracy have more priority or latency?

This list is not exhaustive, but you can use it as a starting point. Remember that other topics, such as privacy and

ethics, can also be important. At the end of this step, we're expected to reach an agreement with the interviewer about the scope and requirements of the system. It's generally a good idea to write down the list of requirements and constraints we gather. By doing so, we ensure everyone is on the same page.

Frame the Problem as an ML Task

Effective problem framing plays a critical role in solving ML problems. Suppose an interviewer asks you to increase the user engagement of a video streaming platform. Lack of user engagement is certainly a problem, but it's not an ML task. So, we should frame it as an ML task in order to solve it. In reality, we should first determine whether or not ML is necessary for solving a given problem. In an ML system design interview, it's safe to assume that ML is helpful. So, we can frame the problem as an ML task by doing the following:

- Defining the ML objective
- Specifying the system's input and output
- Choosing the right ML category

Defining the ML objective

A business objective can be to increase sales by 20% or to improve user retention. But the objective may not be well defined, and we cannot train a model simply by telling it, "increase sales by 20%". For an ML system to solve a task, we need to translate the business objective into a well-defined ML objective. A good ML objective is one that ML models can solve. Let's look at some examples as shown in Table 1.1. In later chapters, we will see more examples.

Application	Business objective	ML objective
Event ticket selling app	Increase ticket sales	Maximize the number of event registrations
Video streaming app	Increase user engagement	Maximize the time users spend watching videos
Ad click prediction system	Increase user clicks	Maximize click-through rate
Harmful content detection in a social media platform	Improve the platform's safety	Accurately predict if a given content is harmful
Friend recommendation system	Increase the rate at which users grow their network	Maximize the number of formed connections

Table 1.1: Translate the business objective to an ML objective

Specifying the system's input and output

Once we decide on the ML objective, we need to define the system's inputs and outputs. For example, for a harmful content detection system on a social media platform, the input is a post, and the output is whether this post is considered harmful or not.

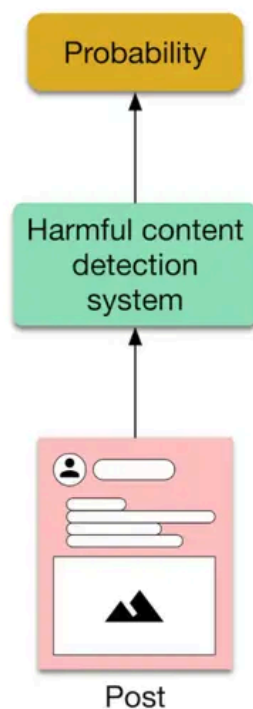


Figure 1.3: Harmful content detection system input-output

In some cases, the system may consist of more than one ML model. If so, we need to specify the input and output of each ML model. For example, for harmful content detection, we may want to use one model to predict violence and another model to predict nudity. The system depends on these two models to determine if a post is harmful or not.

Another important consideration is that there might be multiple ways to specify each model's input-output. Figure 1.4 shows an example.

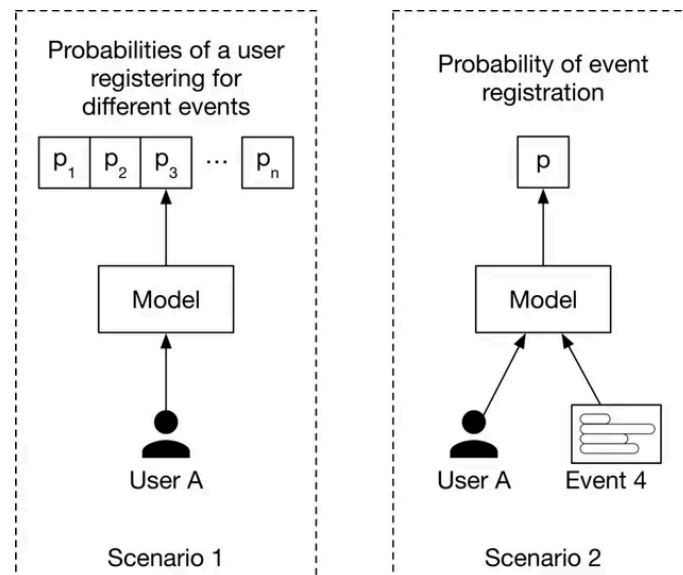


Figure 1.4: Different ways to specify the model's input-output

Choosing the right ML category

There are many different ways to frame a problem as an ML task. Most problems can be framed as one of the ML categories (leaf nodes) shown in Figure 1.5. As most readers are likely already familiar with them, we only provide an outline here.

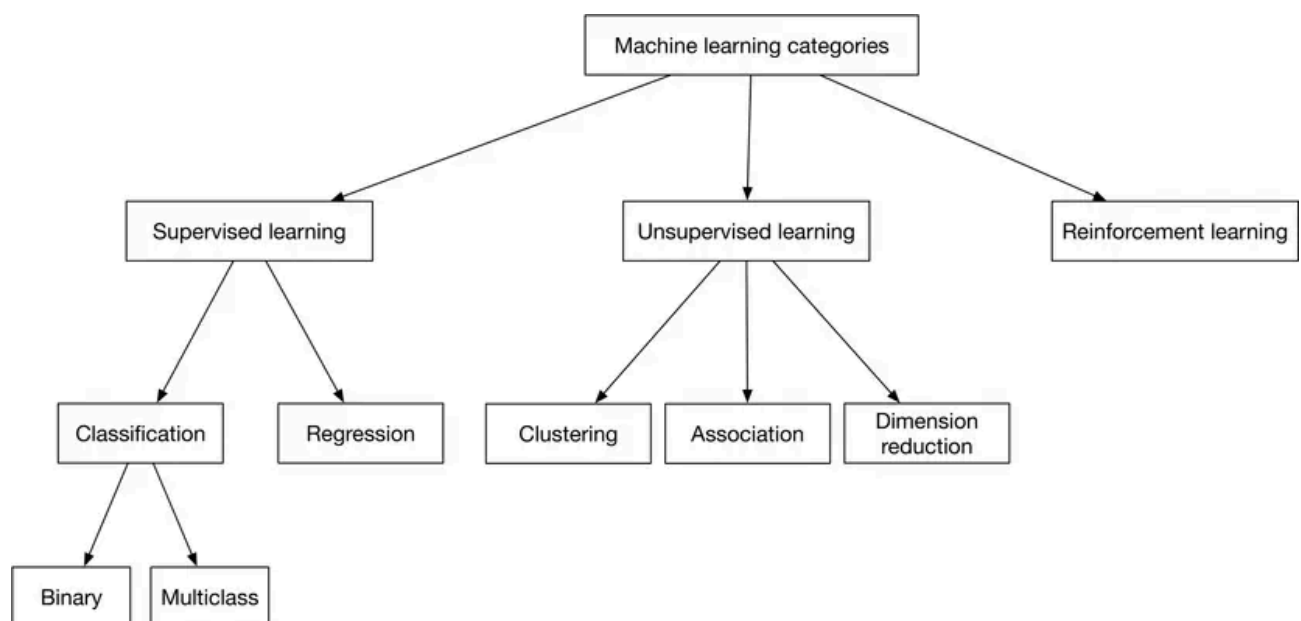


Figure 1.5: Common ML categories

Supervised learning. A supervised learning model learns a task by using a training dataset. In reality, many problems fall into this category, since learning from a labeled dataset usually leads to better results.

Unsupervised learning. Unsupervised learning models make predictions by processing data that contain no

correct answers. The goal of an unsupervised learning model

Reinforcement learning. In reinforcement learning, a computer agent learns to perform a task through repeated trial-and-error interactions with the environment. For example, robots can be trained to walk around a room using reinforcement learning, and software programs like AlphaGo can compete in the game of Go by using reinforcement learning.

Compared to supervised learning, unsupervised learning and reinforcement learning are less popular in real-world systems, as ML models usually learn a specific task better when training data is available. As a result, the majority of problems we tackle in this book rely on supervised learning. Let's take a closer look at the different categories of supervised learning.

Classification model. Classification is the task of predicting a discrete class label; for example, whether an input image should be classified as "dog", "cat", or "rabbit". Classification models can be divided into two groups:

- **Binary classification** models predict a binary outcome. For example, the model predicts whether an image contains a dog or not
- **Multiclass classification** models classify the input into more than one class. For example, we can classify an image as a dog, cat, or rabbit

In this step, you are expected to choose the correct ML category. Later chapters provide examples of how to choose the right category during an interview.

Talking points

Here are some topics we might want to talk about during an interview:

- What is a good ML objective? How do different ML objectives compare? What are the pros and cons?
- What are the inputs and outputs of the system, given the ML objective?
- If more than one model is involved in the ML system, what are the inputs and outputs of each model?
- Does the task need to be learned in a supervised or unsupervised way?
- Is it better to solve the problem using a regression or classification model? In the case of classification, is it binary or multiclass? In the case of regression, what is the output range?

Data Preparation

ML models learn directly from data, meaning that data with predictive power is essential for training an ML model. This section aims to prepare high-quality inputs for ML models via two essential processes: data engineering and feature engineering. We will cover the important aspects of each process.

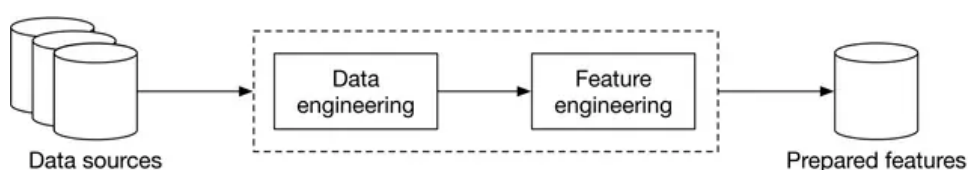


Figure 1.6: Data preparation process

Data engineering

Data engineering is the practice of designing and building pipelines for collecting, storing, retrieving, and processing data. Let's briefly review data engineering fundamentals to understand the core components we may need.

Data sources

An ML system can work with data from many different sources. Knowing the data sources is a good way to answer many context questions, which may include: who collected it? How clean is the data? Can the data source be trusted? Is the data user-generated or system generated?

Data storage

Data storage, also called a database, is a repository for persistently storing and managing collections of data. Different databases are built to satisfy different use cases, so it's important to understand at a high level how different databases work. You are usually not expected to know database internals during ML system design interviews.

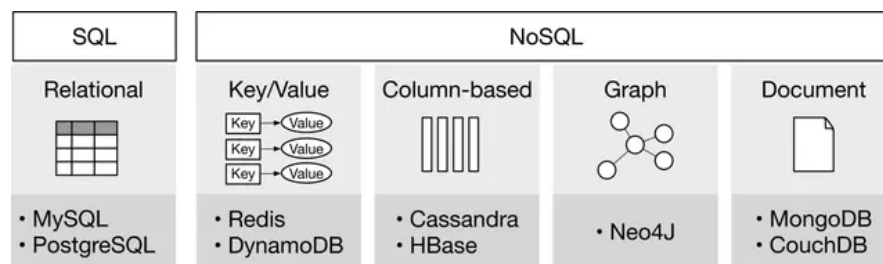


Figure 1.7: Different types of databases

Extract, transform, and load (ETL)

ETL consists of three phases:

- **Extract.** This process extracts data from different data sources.
- **Transform.** In this phase, data is often cleansed, mapped, and transformed into a specific format to meet operational needs.
- **Load.** The transformed data is loaded into the target destination, which can be a file, a database, or a data warehouse [1].

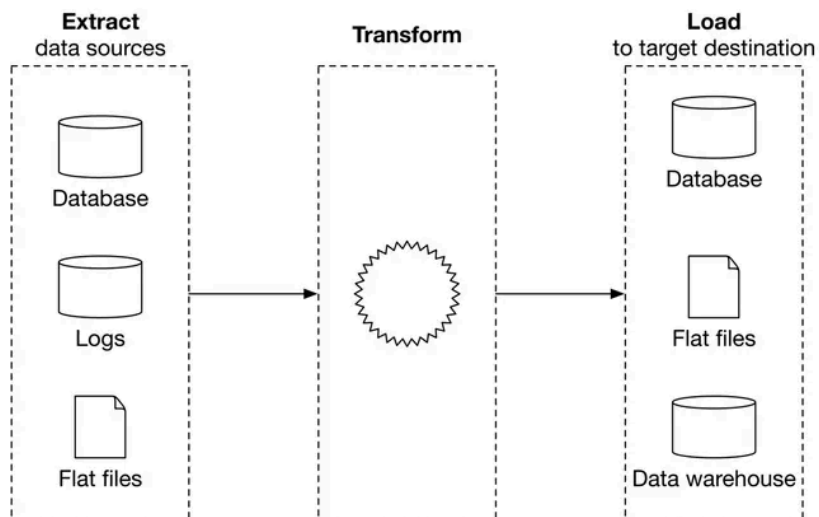


Figure 1.8: An overview of the ETL process

Data types

In ML, data types differ from those in programming languages, such as int, float, string, etc. At the high level, data types can be broken down into two types: structured and unstructured data, as shown in Figure 1.9.

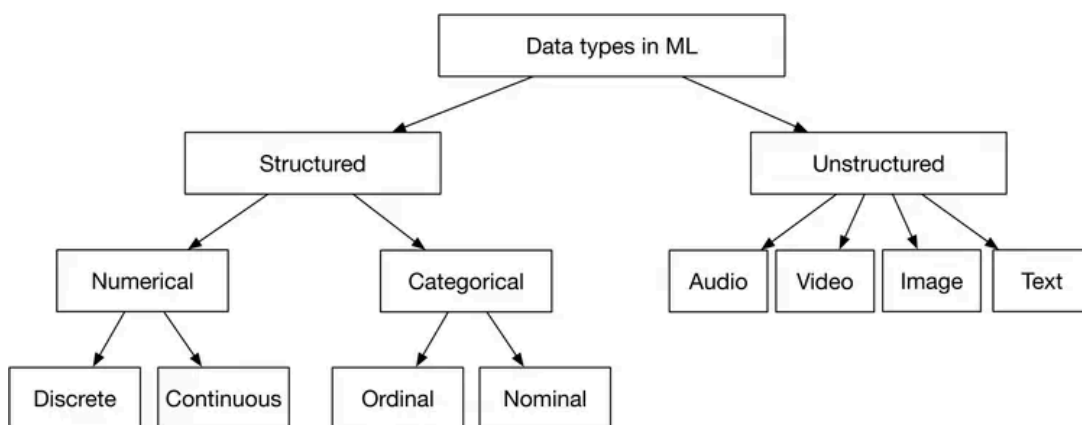


Figure 1.9: Data types in ML

Structured data follows a predefined data schema, whereas unstructured data does not. For example, dates, names, addresses, credit card numbers, and anything that can be represented in a tabular format with rows and columns, can be considered structured data. Unstructured data refers to data with no underlying data schema, such as images, audio files, videos, and text. Table 1.2 summarizes the key differences between structured and unstructured data.

	Structured data	Unstructured Data
Characteristics	•Predefined schema •Easy to search	•No schema •Difficult to search
Resides in	•Relational databases •Many NoSQL databases can store structured data •Data warehouses	•NoSQL databases •Data lakes
Examples	•Dates •Phone numbers •Credit card numbers •Addresses •Names	•Text files •Audio files •Images •Videos

Table 1.2: Summary of structured and unstructured data

As shown in Figure 1.10, ML models perform differently based on the type of data. Understanding and clarifying whether the data is structured or unstructured helps us choose the appropriate ML model during the model development step.

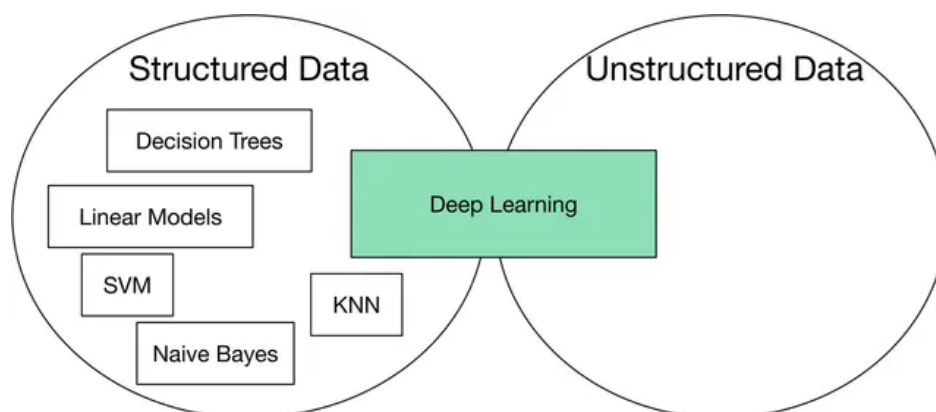


Figure 1.10: Models for structured and unstructured data

Numerical data

Numerical data are any data points represented by numbers. As shown in Figure 1.9, numerical data is divided into continuous numerical data and discrete numerical data. For example, house prices can be considered as a continuous numerical value since a house price can take any value within a range. In contrast, the number of

houses sold in the past year can be considered discrete numerical data, since it takes distinct values only.

Categorical data

Categorical data refers to data that can be stored and identified based on their assigned names or labels. For example, gender is categorical data since its value is from a limited set of age ranges. Categorical data can be divided into two groups: nominal and ordinal.

Nominal data refers to data with no numerical relationship between its categories. For example, gender is nominal data since there is no relationship between "male" and "female". Ordinal data refers to data with a predetermined or sequential order. For example, rating data which takes three unique values from "not happy", "neutral", and "happy" is an example of ordinal data.

Feature Engineering

Feature engineering contains two processes:

- Using domain knowledge to select and extract predictive features from raw data
- Transforming predictive features into a format usable by the model

Choosing the appropriate features is one of the most important decisions when developing and training ML models. It's essential to choose features that bring the most value, and this feature engineering process requires subject matter expertise and is also highly dependent upon the task at hand. To help you master this process, we give many examples throughout this book.

Once the predictive features are chosen, they need to be transformed into suitable formats using feature engineering operations, which we examine next.

Feature engineering operations

It's quite common for some of the selected features to not be in a format the model can use. Feature engineering operations transform the selected features into a format the model can use. Techniques include handling missing values, scaling values that have skewed distributions, and encoding categorical features. The following list is not comprehensive, but it contains some of the most common operations for structured data.

Handling missing values

Data in production often has missing values, which can generally be addressed in two ways: deletion or imputation.

Deletion. This method removes any records with a missing value in any of the features. Deletion can be divided into row deletion and column deletion. In column deletion, we remove the whole column representing a feature, if the feature has too many missing values. In row deletion, we remove a row representing a data point if the data point has many missing values.



Data				
ID	Feature 1	Feature 2	Feature 3	Feature 4
1	2	N/A	6	6
2	9	N/A	8	7
3	18	N/A	11	21
4	2	11	5	6

Data			
ID	Feature 1	Feature 3	Feature 4
1	2	6	6
2	9	8	7
3	18	11	21
4	2	5	6

Figure 1.11: Column deletion

The drawback of deletion is that it reduces the quantity of data the model can potentially use for training. This is not ideal since ML models tend to work better when they are exposed to more data.

Imputation. Alternatively, we can impute the missing values by filling them with certain values. Some common practices include:

- Filling in missing values with their defaults
- Filling in missing values with the mean, median, or mode (the most common value)

The drawback of imputation is it may introduce noise to the data. It's important to note that no technique is perfect for handling missing values, as each has its trade-offs

Feature scaling

Feature scaling refers to the process of scaling features to have a standard range and distribution. Let's first look at why feature scaling might be needed.

Many ML models struggle to learn a task when the features of the dataset are in different ranges. For example, features such as age and income might have different value ranges. In addition, some models may struggle to learn the task when a feature has a skewed distribution. What are some of the feature scaling techniques? Let's take a look.

Normalization (min-max scaling). In this approach, the features are scaled, so all values are within the range $[0, 1]$ using the following formula:

$$z = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

Note that normalization does not change the distribution of the feature. In order to change the distribution of a feature to follow a standard distribution, standardization is used.

Standardization (Z-score normalization). Standardization is the process of changing the distribution of a feature to have a 0 mean and a standard deviation of 1. The following formula is used to standardize a feature:

$$z = \frac{x - \mu}{\sigma}$$

Where μ is the feature's mean and σ is the standard deviation.

Log scaling. To mitigate the skewness of a feature, a common technique called log scaling can be used, with the following formula:

$$z = \log(x)$$

Log transformation can make data distribution less skewed, and enable the optimization algorithm to converge faster.

Discretization (Bucketing)

Discretization is the process of converting a continuous feature into a categorical feature. For example, instead of representing height as a continuous feature, we can divide heights into discrete buckets and represent each height by the bucket to which it belongs. This allows the model to focus on learning only a few categories instead of attempting to learn an infinite number of possibilities.

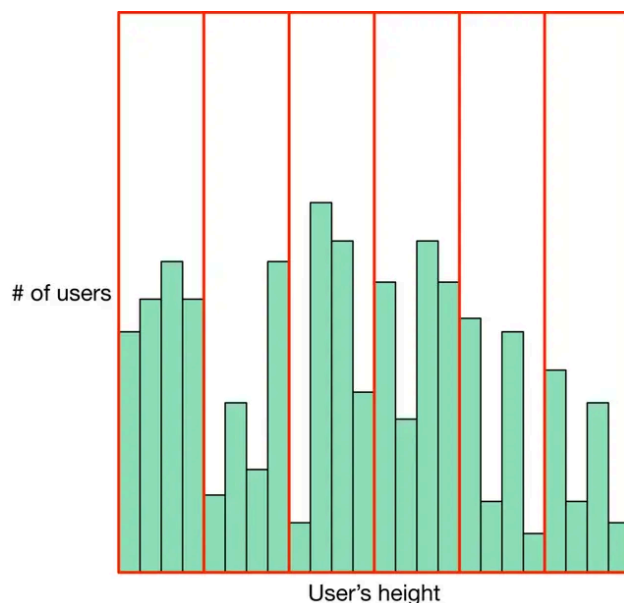


Figure 1.12: Discretizing user's height into 6 buckets

Discretization can also be applied to discrete features. For example, a user's age is a discrete feature, but discretizing it reduces the number of categories, as shown in Table 1.3.

Bucket	Age range
1	0-9
2	10-19
3	20-39
4	40-59
5	60+

Table 1.3: Discretizing numeric age attributes

Encoding categorical features

In most ML models, all inputs and outputs must be numerical. This means if a feature is categorical, we should encode it into numbers before sending it to the model. There are three common methods for converting categorical features into numeric representations: integer encoding, one-hot encoding, and embedding learning.

Integer encoding. An integer value is assigned to each unique category value. For example, “Excellent” is 1, “Good” is 2, and “Bad” is 3. This method is useful if the integer values have a natural relationship with each other.

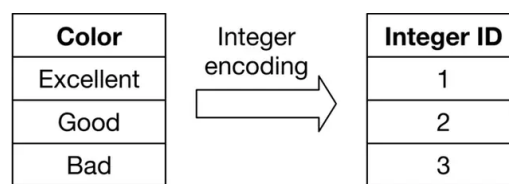


Figure 1.13: Integer encoding

However, when there is no ordinal relationship between categorical features, integer encoding is not a good choice. One-hot encoding, which we will examine next, addresses this issue.

One-hot encoding. With this technique, a new binary feature is created for each unique value. As shown in Figure 1.14, we replace the original feature (color) with three new binary features (red, green, and blue). For example, if a data point has a “red” color, we replace it with “1, 0, 0”.

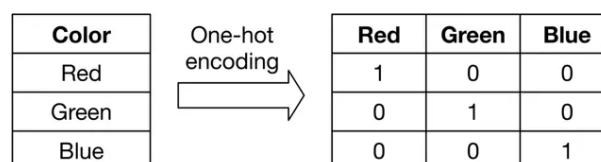


Figure 1.14: One-hot encoding

Embedding learning. Another way to encode a categorical feature is to use embedding learning. An embedding

is a mapping of a categorical feature into an N -dimensional vector. Embedding learning is the process of learning an N -dimensional vector for each unique value that the categorical feature may take. This approach is useful when the number of unique values the feature takes is very large. In this case, one-hot encoding is not a good option because it leads to very large vector sizes. We will see more examples in later chapters.

Talking points

Here are some topics we might want to discuss during the interview:

- **Data availability and data collection:** What are the data sources? What data is available to us, and how do we collect it? How large is the data size? How often do new data come in?
- **Data storage:** Where is the data currently stored? Is it on the cloud or on user devices? Which data format is appropriate for storing the data? How do we store multimodal data, e.g., a data point that might contain both images and texts?
- **Feature engineering:** How do we process raw data into a form that's useful for the models? What should we do about missing data? Is feature engineering required for this task? Which operations do we use to transform the raw data into a format usable by the ML model? Do we need to normalize the features? Which features should we construct from the raw data? How do we plan to combine data of different types, such as texts, numbers, and images?
- **Privacy:** How sensitive are the available data? Are users concerned about the privacy of their data? Is anonymization of user data necessary? Is it possible to store users' data on our servers, or is it only possible to access their data on their devices?
- **Biases:** Are there any biases in the data? If yes, what kinds of biases are present, and how do we correct them?

Model Development

Model development refers to the process of selecting an appropriate ML model and training it to solve the task at hand.

Model selection

Model selection is the process of choosing the best ML algorithm and architecture for a predictive modeling problem. In practice, a typical process for selecting a model is to:

- **Establish a simple baseline.** For example, in a video recommendation system, the baseline can be obtained by recommending the most popular videos.
- **Experiment with simple models.** After we have a baseline, a good practice is to explore ML algorithms that are quick to train, such as logistic regression.
- **Switch to more complex models.** If simple models cannot deliver satisfactory results, we can then consider more complex models, such as deep neural networks.
- **Use an ensemble of models if we want more accurate predictions.** Using an ensemble of multiple models

instead of only one may improve the quality of predictions. Creating an ensemble can be accomplished in three ways: bagging [2], boosting [3], and stacking [4], which will be discussed in later chapters.

In an interview setting, it's important to explore various model options and discuss their pros and cons. Some typical model options include:

- Logistic regression
- Linear regression
- Decision trees
- Gradient boosted decision trees and random forests
- Support vector machines
- Naive Bayes
- Factorization Machines (FM)
- Neural networks

When examining different options, it's good to briefly explain the algorithm and discuss the trade-offs. For example, logistic regression may be a good option for learning a linear task, but if the task is complex, we may need to choose a different model. When choosing an ML algorithm, it's important to consider different aspects of a model. For example:

- The amount of data the model needs to train on
- Training speed
- Hyperparameters to choose and hyperparameter tuning techniques
- Possibility of continual learning
- Compute requirements. A more complex model might deliver higher accuracy, but might require more computing power, such as a GPU instead of a CPU
- Model's interpretability [5]. A more complex model can give better performance, but its results may be less interpretable

There is no single best algorithm that solves all problems. The interviewer wants to see if you have a good understanding of different ML algorithms, their pros and cons, and your ability to choose a model based on requirements and constraints. To help you improve model selection, this book contains a variety of model selection examples. This book assumes you are familiar with common ML algorithms. To refresh your memory, read [6].

Model training

Once the model selection is complete, it's time to train the model. During this step, there are various topics you may want to discuss at an interview, such as:

- Constructing the dataset
- Choosing the loss function
- Training from scratch vs. fine-tuning
- Distributed training

Let's look at each one.

Constructing the dataset

At an interview, it's usually a good idea to talk about constructing the dataset for model training and evaluation. As Figure 1.15 illustrates, there are 5 steps in constructing the dataset.

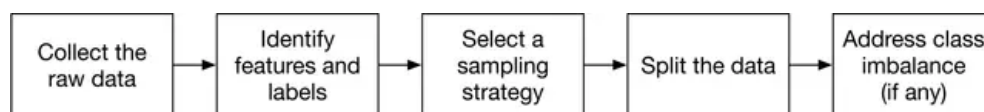


Figure 1.15: Dataset construction steps

All the steps except “identify features and labels” are generic operations, which can be applied to any ML system design task. In this chapter, we will take a close look at each step, but in later chapters, we will mainly focus on “identify features and labels,” which is task-specific.

Collect the raw data

This is extensively discussed in the data preparation step, so we do not repeat it here.

Identify features and labels

During the feature engineering step, we have already discussed which features to use. So, let's focus on creating labels for the data. There are two common ways to get labels: hand labeling and natural labeling.

Hand labeling. This means individual annotators label the data by hand. For example, an individual annotator labels whether a post contains misinformation or not. Hand labeling leads to accurate labels since a human is involved in the process. However, hand labeling has many drawbacks; it is expensive and slow, introduces bias, requires domain knowledge, and is a threat to data privacy.

Natural labeling. In natural labeling, the ground truth labels are inferred automatically without human annotations. Let's see an example to better understand natural labels.

Suppose we want to design an ML system that ranks news feeds based on relevance. One possible way to solve this task is to train a model which takes a user and a post as input, and outputs the probability that the user will press the “like” button after seeing this post. In this case, the training data are pairs of $\langle \text{user}, \text{post} \rangle$ and their corresponding label, which is 1 if the user liked the post, and 0 if they did not. This way, we are able to naturally label the training data without relying on human annotations.

In this step, it is important to clearly communicate how we obtain training labels and what the training data looks like.

Select a sampling strategy

It is often impractical to collect all the data, so sampling is an efficient way to reduce the amount of data in the system. Common sampling strategies include convenience sampling, snowball sampling, stratified sampling, reservoir sampling, and importance sampling. To learn more about sampling methods, you can refer to [7].

Split the data

Data splitting refers to the process of dividing the dataset into training, evaluation (validation), and test dataset. To learn more about data splitting techniques, refer to [8].

Address any class imbalances

A dataset with skewed class labels is called an imbalanced dataset. The class that makes up a larger proportion of the dataset is called the majority class, and the class which comprises a smaller proportion of the dataset is known as the minority class.

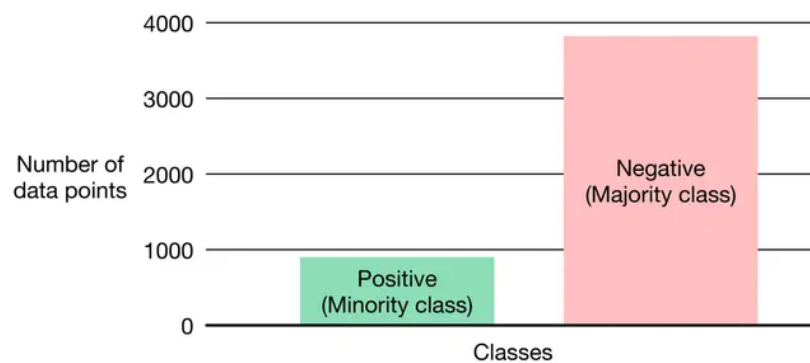


Figure 1.16: Imbalanced dataset with two classes

An imbalanced dataset is a serious issue in model training, as it means a model may not have enough data to learn the minority class. There are different techniques to mitigate this issue. Let's take a look at two commonly used approaches: resampling training data and altering the loss function.

Resampling training data

Resampling refers to the process of adjusting the ratio between different classes, making the data more balanced. For example, we can oversample the minority class (Figure 1.17) or undersample the majority class (Figure 1.18)

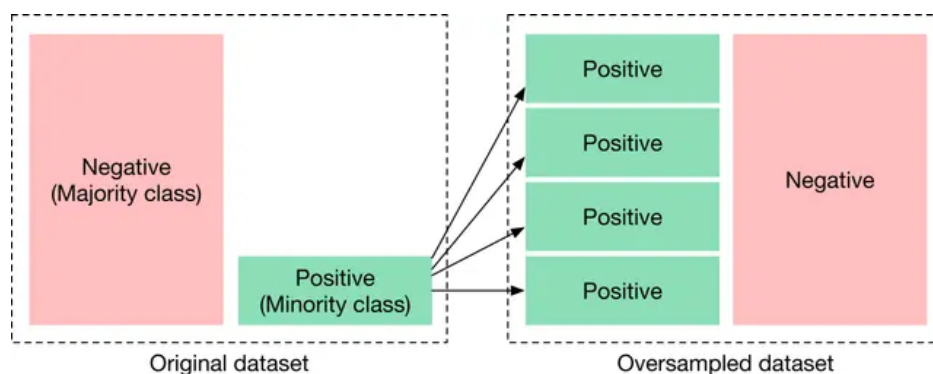


Figure 1.17: Oversampling the minority class

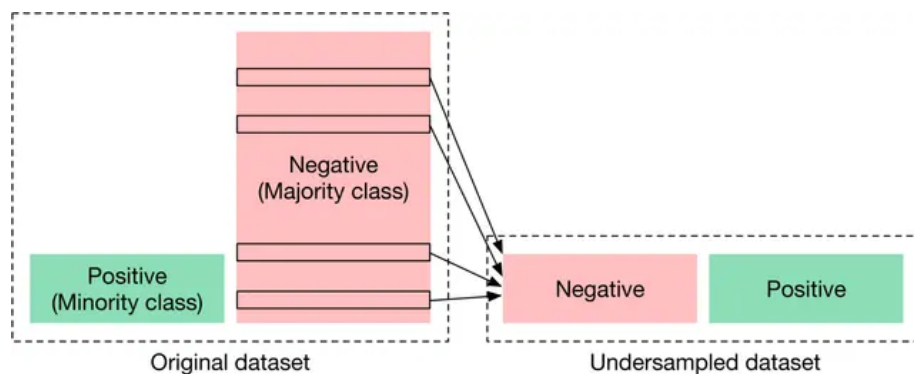


Figure 1.18: Undersampling the majority class

Altering the loss function

This technique alters the loss function to make it more robust against class imbalance. The high-level idea is to give more weight to data points from the minority class. A higher weight in the loss function penalizes the model more when it makes a wrong prediction about a minority class. This forces the model to learn minority classes more effectively. Two commonly used loss functions to mitigate class imbalance are class-balanced loss [9] and focal loss [10][11].

Choosing the loss function

Once the dataset is constructed, we need to choose a proper loss function to train the model. A loss function is a measurement of how accurate the model is at predicting an expected outcome. The loss function allows the optimization algorithm to update the model's parameters during the training process in order to minimize the loss.

Designing a novel loss function is not easy. In ML interviews, you are usually expected to select a loss function from a list of existing loss functions based on how you framed the problem. Sometimes, you might need to make minor changes to the loss function to make it specific to the problem. We will provide more examples in later chapters.

Training from scratch vs. fine-tuning

One topic that might be useful to discuss briefly is training from scratch vs. fine-tuning. Fine-tuning means continuing to train the model on new data by making small changes to its learned parameters. This is a design decision you may need to discuss with the interviewer.

Distributed training

Training at scale becomes increasingly important because models grow bigger over time, and the size of the dataset also increases dramatically. Distributed training is commonly used to train the model, by dividing the work among multiple worker nodes. These worker nodes operate in parallel in order to speed up model training. There are two main types of distributed training: data parallelism [12] and model parallelism [13].

Depending on which task we are solving, employing distributed training may be necessary. In such cases, it's important to discuss this topic with the interviewer. Note that distributed training is a generic topic you can talk

about irrespective of the specific task.

Talking points

Some talking points are listed below:

- **Model selection:** Which ML models are suitable for the task, and what are their pros and cons. Here's a list of topics to consider during model selection:
 - The time it takes to train
 - The amount of training data the model expects
 - The computing resources the model may need
 - Latency of the model at inference time
 - Can the model be deployed on a user's device?
 - Model's interpretability. Making a model more complex may increase its performance, but the results might be harder to interpret
 - Can we leverage continual training, or should we train from scratch?
 - How many parameters does the model have? How much memory is needed?
 - For neural networks, you might want to discuss typical architectures/blocks, such as ResNet or Transformer-based architectures. You can also discuss the choice of hyperparameters, such as the number of hidden layers, the number of neurons, activation functions, etc.
- **Dataset labels:** How should we obtain the labels? Is the data annotated, and if so, how good are the annotations? If natural labels are available, how do we get them? How do we receive user feedback on the system? How long does it take to get natural labels?
- **Model training.**
 - What loss function should we choose? (e.g., Cross-entropy [14], MSE [15], MAE [16], Huber loss [17], etc.)
 - What regularization should we use? (e.g., L1 [19], L2 [18], Entropy Regularization [19], K-fold CV [20], or dropout [21])
 - What is backpropagation?
 - You may need to describe common optimization methods such as SGD [22], AdaGrad [23], Momentum [24], and RMSProp [25].
 - What activation functions do we want to use and why? (e.g., ELU [26], ReLU [27], Tanh [28], Sigmoid [29]).
 - How to handle an imbalanced dataset?
 - What is the bias/variance trade-off?
 - What are the possible causes of overfitting and underfitting? How to address them?
- **Continual learning:** Do we want to train the model online with each new data point? Do we need to personalize the model to each user? How often do we retrain the model? Some models need to be retrained daily or weekly, and others monthly or yearly.

Evaluation

The next step after the model development is evaluation, which is the process of using different metrics to

understand an ML model's performance. In this section, we examine two evaluation methods, offline and online.

Offline evaluation

Offline evaluation refers to evaluating the performance of ML models during the model development phase. To evaluate a model, we usually first make predictions using the evaluation dataset. Next, we use various offline metrics to measure how close the predictions are to the ground truth values. Table 1.4 shows some of the commonly used metrics for different tasks.

Task	Offline metrics
Classification	Precision, recall, F1 score, accuracy, ROC-AUC, PR-AUC, confusion matrix
Regression	MSE, MAE, RMSE
Ranking	Precision@k, recall@k, MRR, mAP, nDCG
Image generation	FID [30], Inception score [31]
Natural language processing	BLEU [32], METEOR [33], ROUGE [34], CIDEr [35], SPICE [36]

Table 1.4: Popular metrics in offline evaluation

During an interview, it's important to identify suitable metrics for offline evaluation. This depends on the task at hand and how we have framed it. For example, if we try to solve a ranking problem, we might need to discuss ranking metrics and their trade-offs.

Online evaluation

Online evaluation refers to the process of evaluating how the model performs in production after deployment. To measure the impact of the model, we need to define different metrics. Online metrics refer to those we use during an online evaluation and are usually tied to business objectives. Table 1.5 shows various metrics for different problems.

Problem	Online metrics
Ad click prediction	Click-through rate, revenue lift, etc.
Harmful content detection	Prevalence, valid appeals, etc.
Video recommendation	Click-through rate, total watch time, number of completed videos, etc.
Friend recommendation	Number of requests sent per day, number of requests accepted per day, etc.

Table 1.5: Possible metrics in online evaluation

In practice, companies usually track many online metrics. At an interview, we need to select some of the most important ones to measure the impact of the system. As opposed to offline metrics, choosing online metrics Online metrics is subjective and depends upon product owners and business stakeholders.

In this step, the interviewer gauges your business sense. So, it is helpful to communicate your thought process and your reasons for choosing certain metrics.

Talking points

Here are some talking points for the evaluation step:

- **Online metrics:** Which metrics are important for measuring the effectiveness of the ML system online? How do these metrics relate to the business objective?
- **Offline metrics:** Which offline metrics are good at evaluating the model's predictions during the development phase?
- **Fairness and bias:** Does the model have the potential for bias across different attributes such as age, gender, race, etc.? How would you fix this? What happens if someone with malicious intent gets access to your system?

Deployment and Serving

A natural next step after choosing the appropriate metrics for online and offline evaluations is to deploy the model to production and serve millions of users. Some important topics to cover include:

- Cloud vs. on-device deployment
- Model compression
- Testing in production
- Prediction pipeline

Let's take a look at each.

Cloud vs. on-device deployment

Deploying a model on the cloud is different from deploying it on a mobile device. Table 1.6 summarizes the main differences between on-device deployment and cloud deployment.

	Cloud	On-device
Simplicity	✓ Simple to deploy and manage using cloud-based services	✗ Deploying models on a device is not straightforward
Cost	✗ Cloud costs might be high	✓ No cloud cost when computations are performed on-device
Network latency	✗ Network latency is present	✓ No network latency
Inference latency	✓ Usually faster inference due to more powerful machines	✗ ML models run slower
Hardware constraints	✓ Fewer constraints	✗ More constraints, such as limited memory, battery consumption, etc.
Privacy	✗ Less privacy as user data is transferred to the cloud	✓ More privacy since data never leaves the device
Dependency on internet connection	✗ Internet connection needed to send and receive data to the cloud	✓ No internet connection needed

Table 1.6: Trade-offs between cloud and on-device deployment

Model compression

Model compression refers to the process of making a model smaller. This is necessary to reduce the inference latency and model size. Three techniques are commonly used to compress models:

- **Knowledge distillation:** The goal of knowledge distillation is to train a small model (student) to mimic a larger model (teacher).
- **Pruning:** Pruning refers to the process of finding the least useful parameters and setting them to zero. This leads to sparser models which can be stored more efficiently.
- **Quantization:** Model parameters are often represented with 32-bit floating numbers. In quantization, we use fewer bits to represent the parameters, which reduces the model's size. Quantization can happen during training or post-training [37].

To learn more about model compression, you are encouraged to read [38].

Test in production

The only way to ensure the model will perform well in production is to test it with real traffic. Commonly used techniques to test models include shadow deployment [39], A/B testing [40], canary release [41], interleaving experiments [42], bandits [43], etc.

To demonstrate that we understand how to test in production, it's a good idea to mention at least one of these methods. Let's briefly review shadow deployment and A/B testing.

Shadow deployment

In this method, we deploy the new model in parallel with the existing model. Each incoming request is routed to both models, but only the existing model's prediction is served to the user.

By shadow deploying the model, we minimize the risk of unreliable predictions until the newly developed model has been thoroughly tested. However, this is a costly approach that doubles the number of predictions.

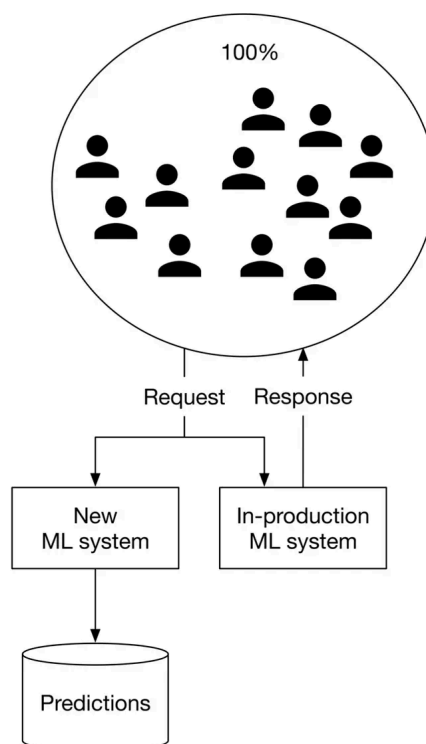


Figure 1.19: Shadow deployment

A/B testing

With this method, we deploy the new model in parallel with the existing model. A portion of the traffic is routed to the newly developed model, while the remaining requests are routed to the existing model.

In order to execute A/B testing correctly, there are two important factors to consider. First, the traffic routed to each

model has to be random. Second, A/B tests should be run on a sufficient number of data points in order for the results to be legitimate.

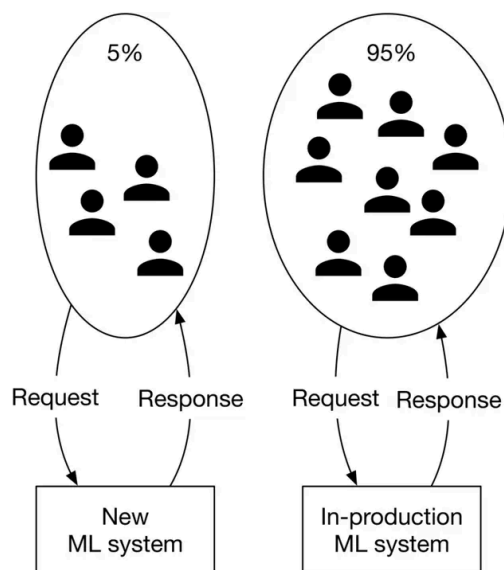


Figure 1.20: A/B testing

Prediction pipeline

To serve requests in production, we need a prediction pipeline. An important design decision to make is the choice between online prediction and batch prediction.

Batch prediction. With batch prediction, the model makes predictions periodically. Because predictions are pre-computed, we don't need to worry about how long it takes the model to generate predictions once they are pre-computed.

However, the batch prediction has two major drawbacks. First, the model becomes less responsive to the changing preferences of users. Secondly, batch prediction is only possible if we know in advance what needs to be pre-computed. For example, in a language translation system, we are not able to make translations in advance as it entirely depends on the user's input.

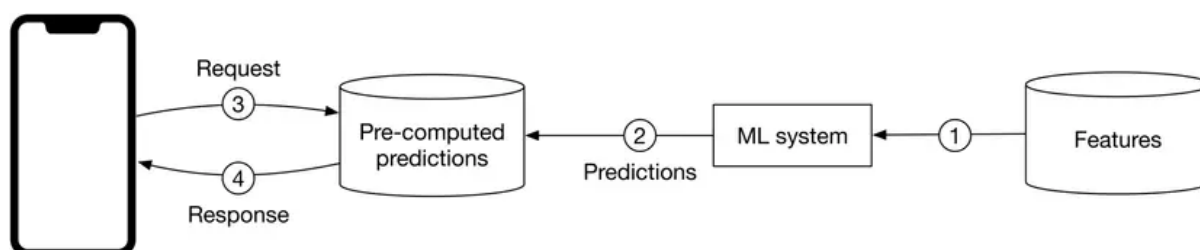


Figure 1.21: Batch prediction workflow

Online prediction. In online prediction, predictions are generated and returned as soon as requests arrive. The main problem with online prediction is that the model might take too long to generate predictions.

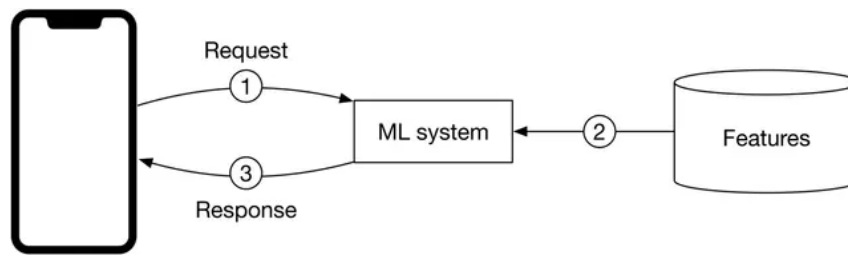


Figure 1.22: Online prediction workflow

This choice of batch prediction or online prediction is mainly driven by product requirements. Online prediction is generally preferred in situations where we do not know what needs to be computed in advance. Batch prediction is ideal when the system processes a high volume of data, and the results are not needed in real time.

As previously discussed, ML system development involves more than just ML modeling. Proposing an overall ML system design in an interview demonstrates a deep understanding of how different components work together as a whole. Interviewers often take this as a critical signal.

Figure 1.23 shows an example of the ML system design for a personalized news feed system. We will examine it in more depth in Chapter 10.

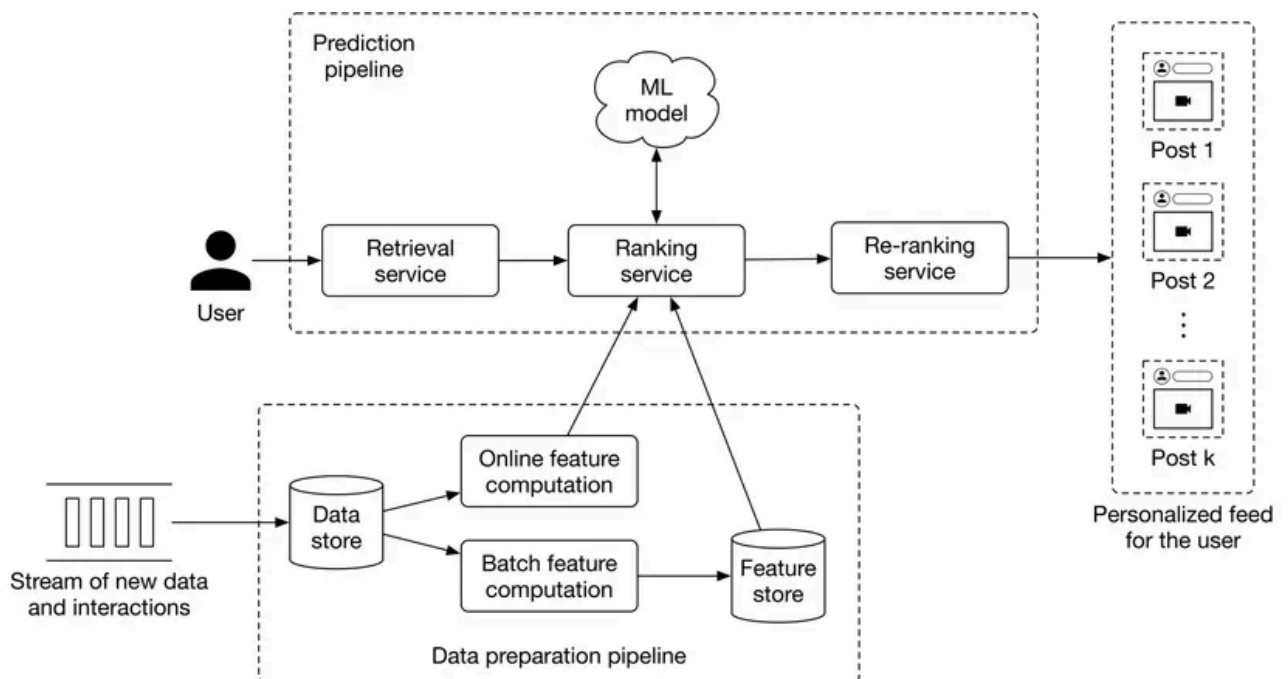


Figure 1.23: ML system design for a personalized news feed system

Talking points

- Is model compression needed? What are some commonly used compression techniques?
- Is online prediction or batch prediction more suitable? What are the trade-offs?
- Is real-time access to features possible? What are the challenges?
- How should we test the deployed model in production?

- An ML system consists of various components working together to serve requests. What are the responsibilities of each component in the proposed design?
- What technologies should we use to ensure that serving is fast and scalable?

Monitoring

Monitoring refers to the task of tracking, measuring, and logging different metrics. An ML system in production can fail for many reasons. Monitoring helps to detect system failures when they occur, so they can be fixed as quickly as possible.

Monitoring is an important topic to discuss in ML system design interviews. Two primary areas you may want to discuss are:

- Why a system fails in production
- What to monitor

Why a system fails in production

There are various reasons why an ML system may fail after it is deployed to production. One of the most common reasons is data distribution shift.

Data distribution shift refers to the scenario in which the data a model encounters in production differs from that it encountered during training. Figure 1.24 shows an example where the training data comprised images of cups seen from the front view, but at serving time, an image of a cup at a different angle is passed to the ML model.



Figure 1.24: Data distribution shift

Data distribution in the real world constantly changes. In other words, the data used for training is likely to become less relevant as time passes. This leads to a stale model with deteriorating performance over time. Therefore, we should continuously monitor the system to detect this problem. Two common approaches for dealing with data distribution shifts are:

- Train on large datasets. A big enough training dataset enables the model to learn a comprehensive distribution, so any data points encountered in production likely come from this learned distribution.
- Regularly retrain the model using labeled data from the new distribution.

What to monitor

This is a broad topic, and here we focus on post-production monitoring techniques. Our goal is to detect failures

and identify shifts in the ML system. Broadly speaking, we can categorize monitoring techniques in ML systems into two buckets: operation-related and ML-specific metrics.

Operation-related metrics: Those metrics ensure the system is up and running. They include average serving times, throughput, the number of prediction requests, CPU/GPU utilization, etc.

ML-specific metrics:

- Monitoring inputs/outputs. Models are only as good as the data they consume, so monitoring the model's input and outputs is vital.
- Drifts. Inputs to the system and the model's outputs are monitored to detect changes in their underlying distribution.
- Model accuracy. For example, we expect the accuracy to be within a specific range.
- Model versions. Monitor which version of the model is deployed.

Infrastructure

Infrastructure is the foundation for training, deploying, and maintaining ML systems.

In many ML interviews, you won't be asked infrastructure-related questions. However, some ML roles, such as DevOps and MLOps may require infrastructure knowledge. So, it's important to be clear about the interviewer's expectations for this topic.

Infrastructure is a very broad subject and cannot be summarized in just a few lines. If you're interested in learning more about ML infrastructure, refer to [44][45][46].

Summary

In this chapter, we proposed a framework for an ML system design interview. While many topics discussed in this chapter are task-specific, some are generic and applicable to a wide range of tasks. Throughout this book, we only focus on unique talking points specific to the problem at hand, in order to avoid repetition. For example, topics related to deployment, monitoring, and infrastructure are often similar, regardless of the task. Therefore, we do not repeat generic topics in later chapters, but you are usually expected to talk about them during an interview.

Finally, no engineer can be an expert in every aspect of the ML lifecycle. Some engineers specialize in deployment and production, while others specialize in model development. Some companies may not care about infrastructure, while others may focus heavily on monitoring and infrastructure. Data science roles generally require more data engineering, while applied ML roles focus more on model development and productionization. Depending on the role and the interviewer's preference, some steps may be discussed in more detail, while others may be discussed briefly or even skipped. In general, a candidate should seek to drive the conversation, while being ready to go with the interviewer's flow, if they raise a question.

Now you understand these fundamentals, we're ready to tackle some of the most common ML system design interview questions.

References

1. Data warehouse. <https://cloud.google.com/learn/what-is-a-data-warehouse>.

2. Bagging technique in ensemble learning. https://en.wikipedia.org/wiki/Bootstrap_aggregating.
3. Boosting technique in ensemble learning. <https://aws.amazon.com/what-is/boosting/>.
4. Stacking technique in ensemble learning. <https://machinelearningmastery.com/stacking-ensemble-machine-learning-with-python/>.
5. Interpretability in Machine Learning. <https://blog.ml.cmu.edu/2020/08/31/6-interpretability/>.
6. Traditional machine learning algorithms. <https://machinelearningmastery.com/a-tour-of-machine-learning-algorithms/>.
7. Sampling strategies. <https://www.scribbr.com/methodology/sampling-methods/>.
8. Data splitting techniques. <https://machinelearningmastery.com/train-test-split-for-evaluating-machine-learning-algorithms/>.
9. Class-balanced loss. <https://arxiv.org/pdf/1901.05555.pdf>.
10. Focal loss paper. <https://arxiv.org/pdf/1708.02002.pdf>.
11. Focal loss. <https://medium.com/swlh/focal-loss-an-efficient-way-of-handling-class-imbalance-4855ae1db4cb>.
12. Data parallelism. <https://www.telesens.co/2017/12/25/understanding-data-parallelism-in-machine-learning/>.
13. Model parallelism. <https://docs.aws.amazon.com/sagemaker/latest/dg/model-parallel-intro.html>.
14. Cross entropy loss. https://en.wikipedia.org/wiki/Cross_entropy.
15. Mean squared error loss. https://en.wikipedia.org/wiki/Mean_squared_error.
16. Mean absolute error loss. https://en.wikipedia.org/wiki/Mean_absolute_error.
17. Huber loss. https://en.wikipedia.org/wiki/Huber_loss.
18. L1 and L2 regularization. <https://www.analyticssteps.com/blogs/l2-and-l1-regularization-machine-learning>.
19. Entropy regularization. <https://paperswithcode.com/method/entropy-regularization>.
20. K-fold cross validation. [https://en.wikipedia.org/wiki/Cross-validation_\(statistics\)](https://en.wikipedia.org/wiki/Cross-validation_(statistics)).
21. Dropout paper. <https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf>.
22. Stochastic gradient descent. https://en.wikipedia.org/wiki/Stochastic_gradient_descent.
23. AdaGrad optimization algorithm. <https://optimization.cbe.cornell.edu/index.php?title=AdaGrad>.
24. Momentum optimization algorithm. <https://optimization.cbe.cornell.edu/index.php?title=Momentum>.
25. RMSProp optimization algorithm. <https://optimization.cbe.cornell.edu/index.php?title=RMSProp>.
26. ELU activation function. https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#elu.
27. ReLU activation function. https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#relu.
28. Tanh activation function. https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#tanh.
29. Sigmoid activation function. https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#softmax.
30. FID score. https://en.wikipedia.org/wiki/Fr%C3%A9chet_inception_distance.
31. Inception score. https://en.wikipedia.org/wiki/Inception_score.
32. BLEU metrics. <https://en.wikipedia.org/wiki/BLEU>.
33. METEOR metrics. <https://en.wikipedia.org/wiki/METEOR>.
34. ROUGE score. [https://en.wikipedia.org/wiki/ROUGE_\(metric\)](https://en.wikipedia.org/wiki/ROUGE_(metric)).
35. CIDEr score. <https://arxiv.org/pdf/1411.5726.pdf>.
36. SPICE score. <https://arxiv.org/pdf/1607.08822.pdf>.
37. Quantization-aware training. <https://pytorch.org/docs/stable/quantization.html>.
38. Model compression survey. <https://arxiv.org/pdf/1710.09282.pdf>.
39. Shadow deployment. <https://christophergs.com/machine%20learning/2019/03/30/deploying-machine-learning->

[applications-in-shadow-mode/](#).

40. A/B testing. https://en.wikipedia.org/wiki/A/B_testing.
41. Canary release. <https://blog.getambassador.io/cloud-native-patterns-canary-release-1cb8f82d371a>.
42. Interleaving experiment. <https://netflixtechblog.com/interleaving-in-online-experiments-at-netflix-a04ee392ec55>.
43. Multi-armed bandit. <https://vwo.com/blog/multi-armed-bandit-algorithm/>.
44. ML infrastructure. <https://www.run.ai/guides/machine-learning-engineering/machine-learning-infrastructure>.
45. Interpretability in ML. <https://fullstackdeeplearning.com/spring2021/lecture-6/>.
46. Chip Huyen. *Designing Machine Learning Systems: An Iterative Process for Production-Ready Application*. "O'Reilly Media, Inc.", 2022.

02

Visual Search System

A system helps users discover images that are visually similar to a selected image. In this chapter, we design a visual search system similar to Pinterest's [1] [2].

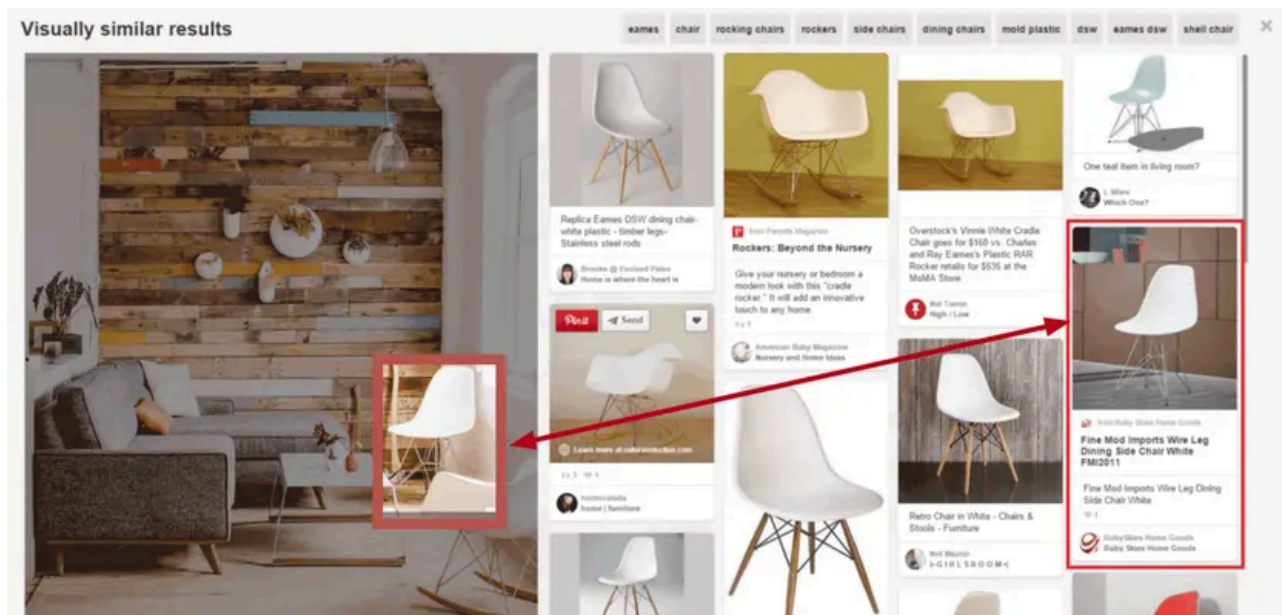


Figure 2.1: Retrieved images that are visually similar to the selected crop

Clarifying Requirements

Here's a typical interaction between a candidate and an interviewer.

Candidate: Should we rank the results from most similar to least similar?

Interviewer: Images that appear first in the list should be more similar to the query image.

Candidate: Should the system support videos, too?

Interviewer: Let's focus only on images.

Candidate: A platform like Pinterest allows users to select an image crop and retrieve similar images. Should we support that functionality?

Interviewer: Yes.

Candidate: Are the displayed images personalized to the user?

Interviewer: For simplicity, let's not focus on personalization. A query image yields the same results, regardless of who searches for it.

Candidate: Can the model use the metadata of the query image, such as image tags?

Interviewer: In practice, the model uses image metadata. But for simplicity, let's assume we don't rely on the metadata, but only on the image pixels.

Candidate: Can users perform other actions such as save, share, or like? These actions can help label training data.

Interviewer: Great point. For simplicity, let's assume the only supported action is image clicks.

Candidate: Should we moderate the images?

Interviewer: It's important to keep the platform safe, but content moderation is out of scope.

Candidate: We can construct training data online and label them based on user interactions. Is this the expected way to construct training data?

Interviewer: Yes, that sounds reasonable.

Candidate: How fast should the search be? Assuming we have 100-200 billion images on the platform, the system should be able to retrieve similar images quickly. Is that a reasonable assumption?

Interviewer: Yes, that is a reasonable assumption.

Let's summarize the problem statement. We are asked to design a visual search system. The system retrieves images similar to the query image provided by the user, ranks them based on their similarities to the query image, and then displays them to the user. The platform only supports images, with no video or text queries allowed. For simplicity, no personalization is required.

Frame the Problem as an ML Task

In this section, we choose a well-defined ML objective and frame the visual search problem as an ML task.

Defining the ML objective

In order to solve this problem using an ML model, we need to create a well-defined ML objective. A potential ML objective is to accurately retrieve images that are visually similar to the image the user is searching for.

Specifying the system's input and output

The input of a visual search system is a query image provided by the user. The system outputs images that are visually similar to the query image, and the output images are ranked by similarities. Figure 2.2 shows the input and output of a visual search system.

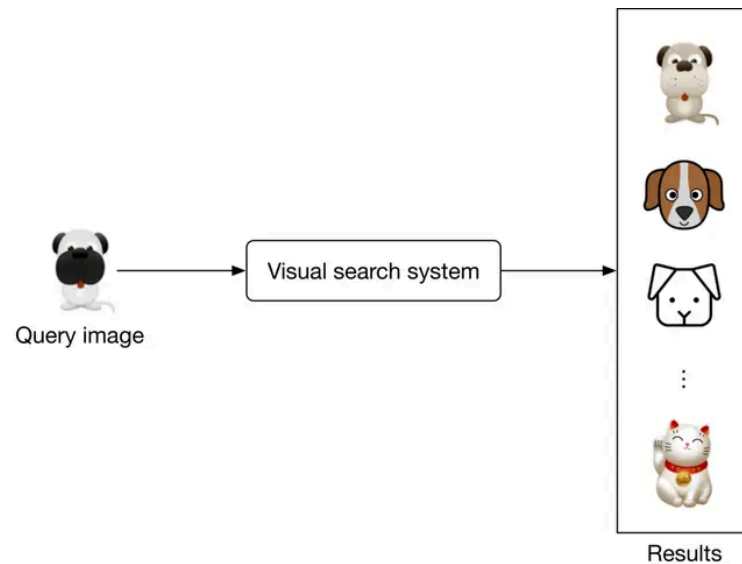


Figure 2.2: A visual search system's input-output

Choosing the right ML category

The output of the model is a set of ranked images that are similar to the query image. As a result, visual search systems can be framed as a ranking problem. In general, the goal of ranking problems is to rank a collection of items, such as images, websites, products, etc., based on their relevance to a query, so that more relevant items appear higher in the search results. Many ML applications, such as recommendation systems, search engines, document retrieval, and online advertising, can be framed as ranking problems. In this chapter, we will use a widely-used approach called representation learning. Let's examine this in more detail.

Representation learning. In representation learning [3], a model is trained to transform input data, such as images, into representations called embeddings. Another way of describing this is that the model maps input images to points in an N -dimensional space called embedding space. These embeddings are learned so that similar images have embeddings that are in close proximity to each other, in the space. Figure 2.3 illustrates how two similar images are mapped onto two points in close proximity within the embedding space. To demonstrate, we visualize image embeddings (denoted by ' x ') in a 2-dimensional space. In reality, this space is N -dimensional, where N is the size of the embedding vector.

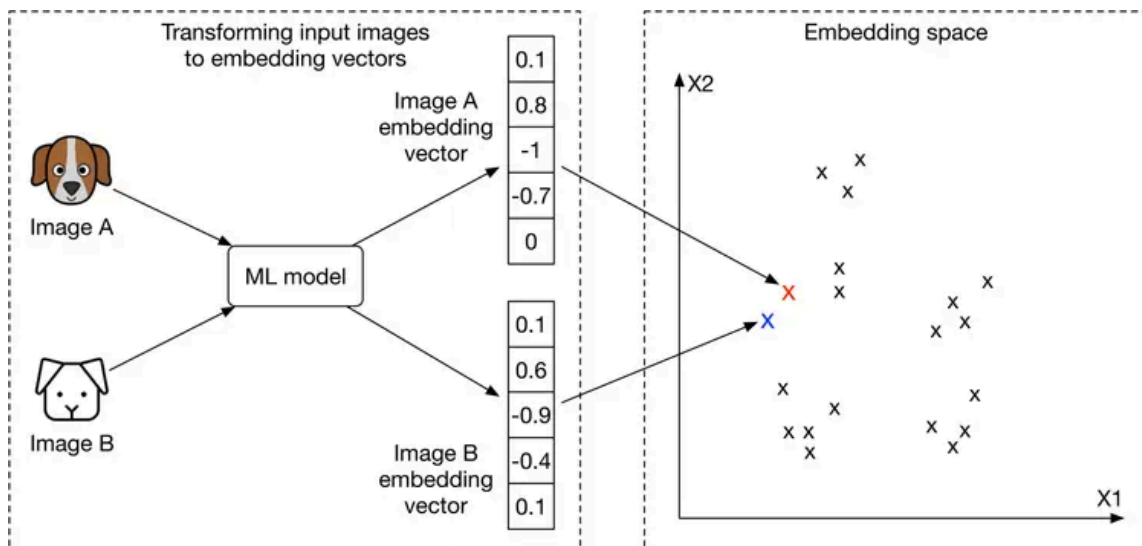


Figure 2.3: Similar images in the embedding space

How to rank images using representation learning?

First, the input images are transformed into embedding vectors. Next, we calculate the similarity scores between the query image and other images on the platform by measuring their distances in the embedding space. The images are ranked by similarity scores, as shown in Figure 2.4.

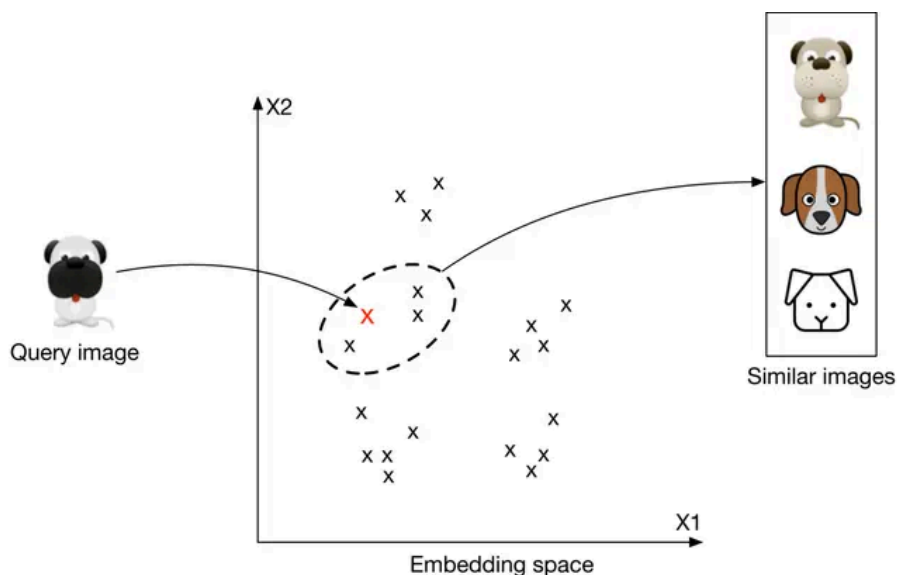


Figure 2.4: Top 3 images similar to the query image

At this point, you may have many questions, including how to ensure similar images are placed close to each other in the embedding space, how to define the similarity, and how to train such a model. We will talk more about these in the model development section.

Data Preparation

Data engineering

Aside from generic data engineering fundamentals, it's important to understand what data is available. As a visual search system mainly focuses on users and images, we have the following data available:

- Images
- Users
- User-image interactions

Images

Creators upload images, and the system stores the images and their metadata, such as owner id, contextual information (e.g., upload time), tags, etc. Table 2.1 shows a simplified example of image metadata.

ID	Owner ID	Upload time	Manual tags
1	8	1658451341	Zebra
2	5	1658451841	Pasta, Food, Kitchen
3	19	1658821820	Children, Family, Party

Table 2.1: Image metadata

Users

User data contains demographic attributes associated with users, such as age, gender, etc. Table 2.2 shows an example of user data.

ID	Username	Age	Gender	City	Country	Email
1	johnduo	26	M	San Jose	USA	john@gmail.com
2	hs2008	49	M	Paris	France	hsieh@gmail.com
3	alexish	16	F	Rio	Brazil	alexh@yahoo.com

Table 2.2: User data

User-image interactions

Interaction data contains different types of user interactions. Based on the requirements gathered, the primary types of interactions are impressions and clicks. Table 2.3 shows an overview of interaction data.

User ID	Query image ID	Displayed image ID	Position in the displayed list	Interaction type	Location (lat, long)	Timestamp
8	2	6	1	Click	38.8951 -77.0364	1658450539
6	3	9	2	Click	38.8951 -77.0364	1658451341
91	5	1	2	Impression	41.9241 -89.0389	1658451365

Table 2.3: User-image interaction data

Feature engineering

In this section, you are expected to talk about engineering great features and preparing them as model inputs. This usually depends on how we framed the task and what the model's inputs are. In the earlier "Framing the problem as an ML task" section, we framed the visual search system as a ranking problem and used representation learning to solve it. In particular, we employed a model which expects an image as input. The image needs to be preprocessed before being passed to the model. Let's take a look at common image preprocessing operations:

- **Resizing:** Models usually require fixed image sizes (e.g., 224×224)
- **Scaling:** Scale pixel values of the image to the range of 0 and 1
- **Z-score normalization:** Scale pixel values to have a mean of 0 and variance of 1
- **Consistent :** Ensuring images have a consistent color mode (e.g., RGB or CMYK)

Model Development

Model selection

We choose neural networks because:

- Neural networks are good at handling unstructured data, such as images and text
- Unlike many traditional machine learning models, neural networks are able to produce the embeddings we need for representation learning

What types of neural network architectures should we use? It is essential that the architecture works with images. CNN-based architectures such as ResNet [4] or more recent Transformer-based architectures [5] such as ViT [6] perform well with image inputs. Figure 2.5 shows a simplified model architecture that transforms the input image into an embedding vector. The number of convolution layers, the number of neurons in fullyconnected layers, and the size of the embedding vector are hyperparameters typically chosen via experimentation.

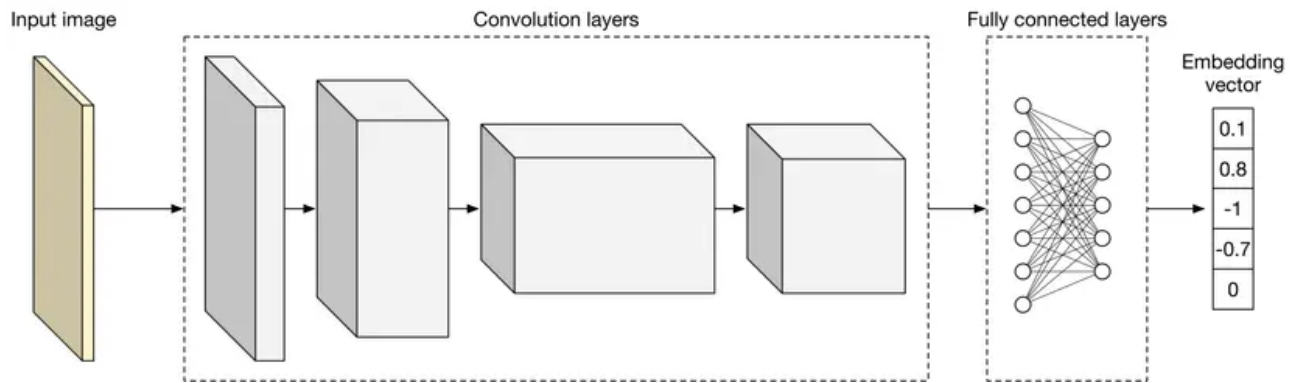


Figure 2.5: A simplified model architecture

Model training

In order to retrieve visually similar images, a model must learn representations (embeddings) during training. In this section, we discuss how to train a model to learn image representations.

A common technique for learning image representations is contrastive training [7]. With this technique, we train the model to distinguish similar and dissimilar images. As Figure 2.6 shows, we provide the model with a query image (left), one similar image to the query image (highlighted dog image on the right), and a few dissimilar images (also right.) During training, the model learns to produce representations in which the similar image more closely resembles the query image, than do other images on the right side of Figure 2.6.

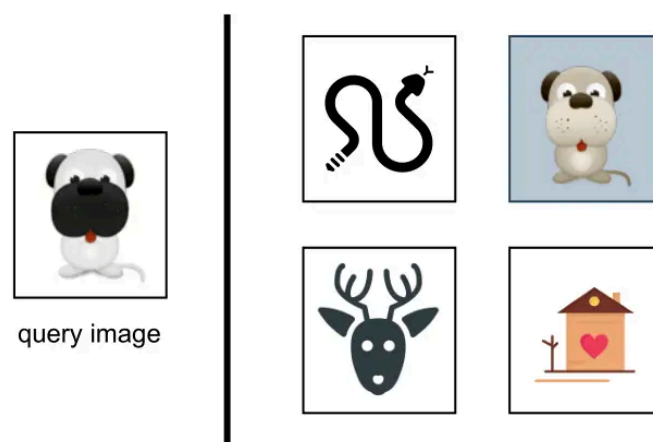


Figure 2.6: Contrastive training

To train the model using the contrastive training technique, we first need to construct training data.

Constructing the dataset

As described earlier, each data point used for training contains a query image, a positive image that's similar to the query image, and $n - 1$ negative images that are dissimilar to the query image. The ground truth label of the data point is the index of the positive image. As Figure 2.7 shows, along with the query image (q), we have n other images, of which one is similar to the q (image of the dog), and the other $n - 1$ images are dissimilar. The ground truth label for this data point is the index of the positive image, which is 2 (the second image among the n images in Figure 2.7).

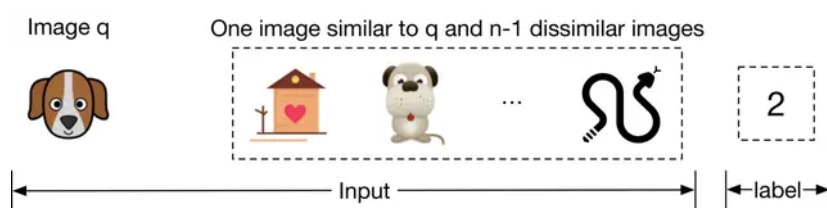


Figure 2.7: Training data point

To construct a training data point, we randomly choose a query image and $n - 1$ images as negative images. To select a positive image, we have the following three options:

- Use human judgment
- Use interactions such as user clicks as a proxy for similarity
- Artificially create a similar image from the query image, known as self-supervision

Let's evaluate each option.

Use human judgments

This approach relies on human contractors manually finding similar images. Human involvement produces

accurate training data, but using human annotators is expensive and time-consuming.

Use interactions such as user click as a proxy for similarity

In this approach, we measure similarity based on interaction data. As an example, when a user clicks on an image, the clicked image is considered to be similar to the query image q .

This approach does not require manual work and can generate training data automatically. However, the click signal is usually very noisy. Users sometimes click on images even when the image is not similar to the query image. Additionally, this data is very sparse, and we may not have click data available for lots of the images. The use of noisy and sparse training data leads to poor performance.

Artificially create a similar image from the query image

In this approach, we artificially create a similar image from the query image. For example, we can augment the query image by rotating it and using the newly generated image as a similar image. Recently developed frameworks such as SimCLR [7] and MoCo [8] use the same approach.

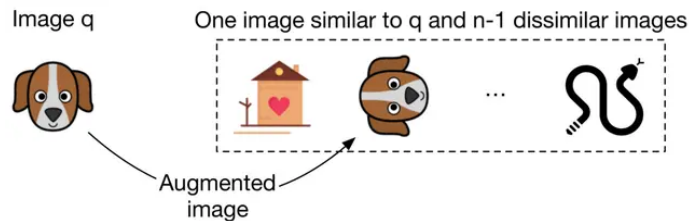


Figure 2.8: Use data augmentation to create a similar image

An advantage of this method is that no manual work is required. We can implement simple data augmentation logic to create similar images. In addition, the constructed training data is not noisy, since augmenting an image always results in a similar image. The major drawback of this approach is that the constructed training data differs from the real data. In practice, similar images are not augmented versions of the query image; they are visually and semantically similar, but are distinct.

Which approach works best in our case?

In an interview setting, it's critical you propose various options and discuss their tradeoffs. There is usually not a single best solution that always works. Here, we use the self-supervision option for two reasons. Firstly, there is no upfront cost associated with it, since the process can be automated. Secondly, various frameworks such as SimCLR [7] have shown promising results when trained on a large dataset. Since we have access to billions of images on the platform, this approach might be a good fit.

We can always switch to other labeling methods if the experiment results are unsatisfactory. For example, we can start with the self-supervision option and later use click data for labeling. We can also combine the options. For example, we may use clicks to build our initial training data and rely on human annotators to identify and remove noisy data points. Discussing different options and trade-offs with the interviewer is critical to make good design decisions.

Once we construct the dataset, it's time to train the model using a proper loss function.

Choosing the loss function

As Figure 2.9 shows, the model takes images as input and produces an embedding for each input image. E_x denotes the embedding of the image x .

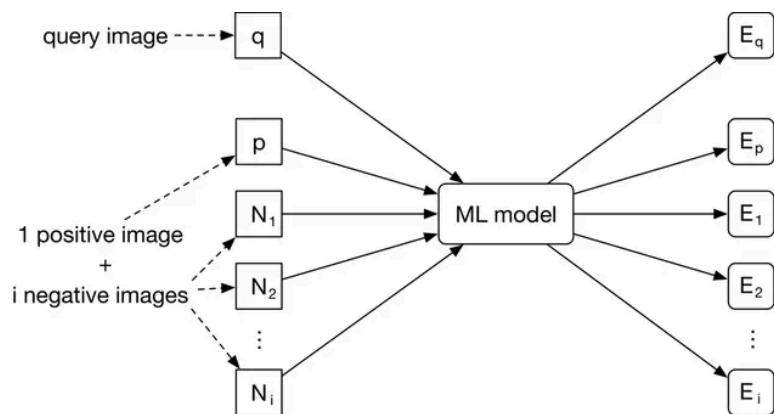


Figure 2.9: Model input-output

The goal of the training is to optimize the model parameters so that similar images have embeddings close to each other in the embedding space. As Figure 2.10 shows, the positive image and the query image should become closer during training.

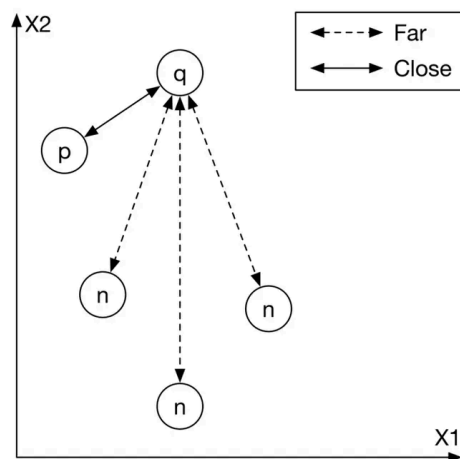


Figure 2.10: Input images mapped into the embedding space

To achieve this goal, we need to use a loss function to measure the quality of the produced embeddings. Different loss functions are designed for contrastive training, and interviewers don't usually expect you to hold an in-depth discussion. However, it is crucial to have a high-level understanding of how contrastive loss functions work.

We are going to briefly discuss how a simplified contrastive loss operates. If you are interested in learning more about contrastive losses, refer to [9].

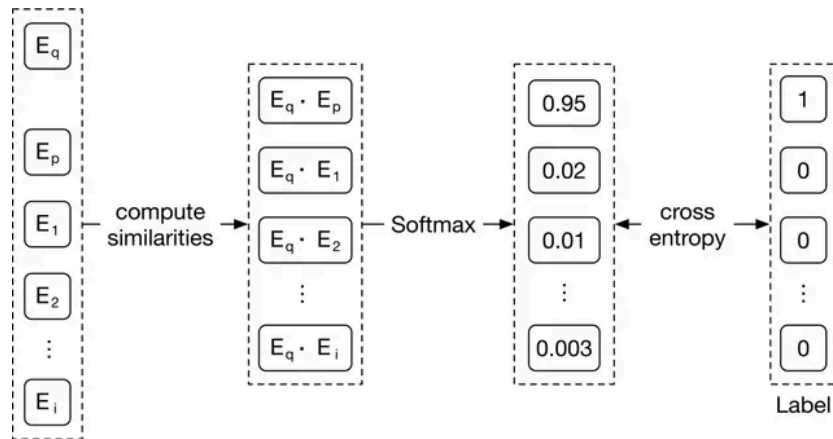


Figure 2.11: A simplified contrastive loss

As Figure 2.11 shows, we compute the contrastive loss in three steps.

Compute similarities. First, we compute the similarities between the query image and the embeddings of other images. Dot product [10] and cosine similarity [11] are widely used to measure the similarity between points in the embedding space. Euclidean distance [12] can also measure the similarity. However, Euclidean distance usually performs poorly in high dimensions because of the curse of dimensionality [13]. To learn more about the curse of dimensionality issues, read [14].

Softmax. A softmax function is applied over the computed distances. This ensures the values sum up to one, which allows the values to be interpreted as probabilities.

Cross-entropy. Cross-entropy [15] measures how close the predicted probabilities are to the ground truth labels. When the predicted probabilities are close to the ground truth, it shows that the embeddings are good enough to distinguish the positive image from the negative ones.

In the interview, you can also discuss the possibility of using a pre-trained model. For example, we could leverage a pre-trained contrastive model and fine-tune it using the training data. These pre-trained models have already been trained on large datasets, and therefore they have learned good image representations. This significantly reduces the training time compared to training a model from scratch.

Evaluation

After we develop the model, we can discuss the evaluation. In this section, we cover important metrics for offline and online evaluations.

Offline metrics

Based on the given requirements, an evaluation dataset is available for offline evaluation. Let's assume each data point has a query image, a few candidate images, and a similarity score for each candidate image and the query image pair. A similarity score is an integer number between 0 to 5, where 0 indicates no similarity and 5 indicates two images are visually and semantically very similar. For each data point in the evaluation dataset, we compare the ranking produced by the model with the ideal ranking, based on the ground truth scores.

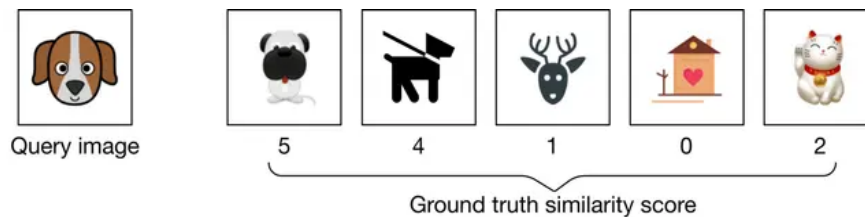


Figure 2.12: A data point from the evaluation dataset

Now, let's examine offline metrics that are commonly used in search systems. Note that search, information retrieval, and recommendation systems usually share the same offline metrics.

- Mean reciprocal rank (MRR)
- Recall@k
- Precision@k
- Mean average precision (mAP)
- Normalized discounted cumulative gain (nDCG)

MRR. This metric measures the quality of the model by considering the rank of the first relevant item in each output list produced by the model, and then averaging them. The formula is:

$$MRR = \frac{1}{m} \sum_{i=1}^m \frac{1}{rank_i}$$

Where m is the total number of output lists and $rank_i$ refers to the rank of the first relevant item in the i th output list.

Figure 2.13 illustrates how this works. For each of the 4 ranked lists, we compute the reciprocal rank (RR) and then calculate the average value of the RRs to get the MRR.

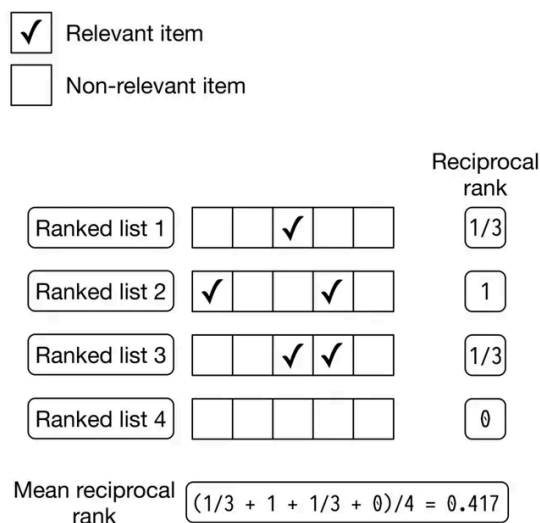


Figure 2.13: MRR calculation example

Let's examine the shortcoming of this metric. Since MRR considers only the first relevant item and ignores other relevant items in the list, it does not measure the precision and ranking quality of a ranked list. For example, Figure 2.14 shows the outputs of two different models. The output of model 1 has 3 relevant items, while the output of model 2 has 1 relevant item. However, the reciprocal rank of both models is 0.5. Given this shortcoming, we will not use this metric.

						MRR
Model 1 output		✓	✓	✓		$\frac{1}{2}$
Model 2 output		✓				$\frac{1}{2}$

Figure 2.14: MRR of two different models

Recall@k. This metric measures the ratio between the number of relevant items in the output list and the total number of relevant items available in the entire dataset. The formula is:

$$\text{recall@k} = \frac{\text{number of relevant items among the top } k \text{ items in the output list}}{\text{total relevant items}}$$

Even though recall@k measures how many relevant items the model failed to include in the output list, this isn't always a good metric. Let's understand why not. In some systems, such as search engines, the total number of relevant items can be very high. This negatively affects the recall as the denominator is very large. For example, when the query image is an image of a dog, the database may contain millions of dog images. The goal is not to return every dog image but to retrieve a handful of the most similar dog images.

Given recall@k doesn't measure the ranking quality of the model, we will not use it.

Precision@k. This metric measures the proportion of relevant items among the top k items in the output list. The formula is:

$$\text{precision@k} = \frac{\text{number of relevant items among the top } k \text{ items in the output list}}{k}$$

This metric measures how precise the output lists are, but it doesn't consider the ranking quality. For example, in Figure 2.15, if we rank more relevant items higher in the list, the precision won't change. This metric is not ideal for our use case, since we need to measure both the precision and ranking quality of the results.

						Precision@5
Model 1 output	✓	✓				$\frac{2}{5}$
Model 2 output		✓			✓	$\frac{2}{5}$

Figure 2.15: Precision@5 of two different models

mAP. This metric first computes the average precision (AP) for each output list, and then averages AP values.

Let's first understand what AP is. It takes a list of k items, such as images, and averages the precision@k at

different values of k . AP is high if more relevant items are located at the top of the list. For a list of size k , the AP formula is:

$$AP = \frac{\sum_{i=1}^k \text{Precision@}i \text{ if } i\text{'th item is relevant to the user}}{\text{total relevant items}}$$

Let's look at an example to better understand the metric. Figure 2.16 shows AP calculations for each of the 4 output lists produced by the model.

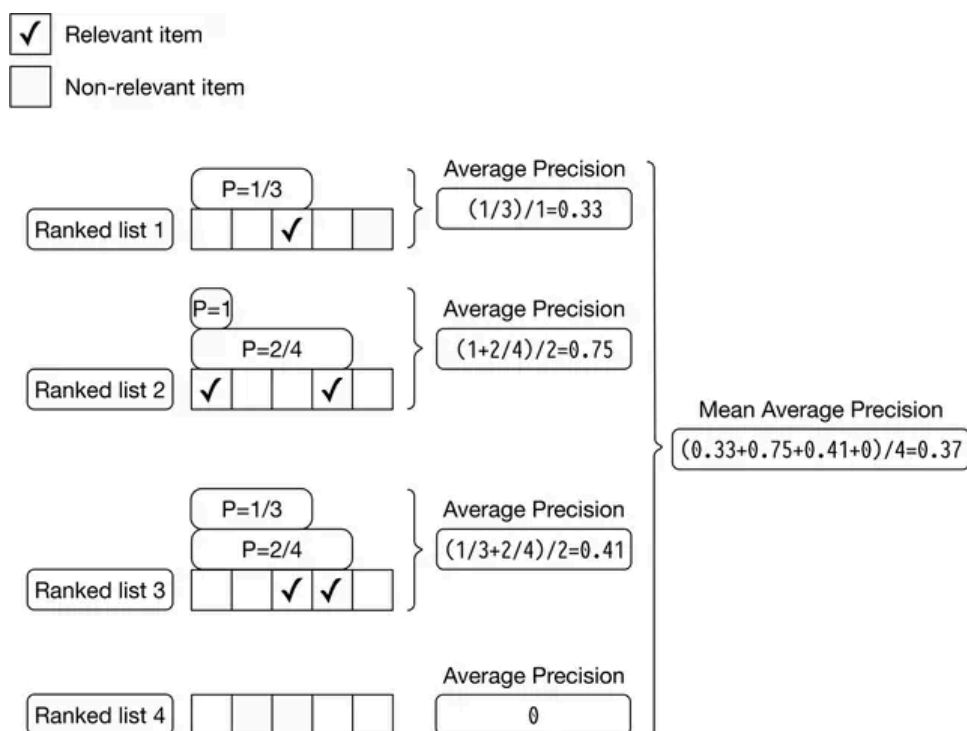


Figure 2.16: mAP calculations

Since we average precisions, the overall ranking quality of the list is considered. However, mAP is designed for binary relevances; in other words, it works well when each item is either relevant or irrelevant. For continuous relevance scores, nDCG is a better choice.

nDCG. This metric measures the ranking quality of an output list and shows how good the ranking is, compared to the ideal ranking. First, let's explain DCG and then discuss nDCG.

What is DCG?

DCG calculates the cumulative gain of items in a list by summing up the relevance score of each item. Then the score is accumulated from the top of the output list to the bottom, with the score of each result discounted at lower ranks. The formula is:

$$DCG_p = \sum_{i=1}^p \frac{rel_i}{\log_2(i+1)}$$

Where rel_i is the ground truth relevance score of the image ranked at location i .

What is nDCG?

Because DCG sums up the relevance scores of items and discounts their positions, the result of DCG could be any value. In order to get a more meaningful score, we need to normalize DCG. For this, nDCG divides the DCG by the DCG of an ideal ranking. The formula is:

$$nDCG_p = \frac{DCG_p}{IDCG_p}$$

Where $IDCG_p$ is the DCG of the ideal ranking (a ranking ordered by the relevance scores of items). Note that in a perfect ranking system, the DCG is equal to IDCG.

Let's use an example to better understand nDCG. In Figure 2.17, we can see a list of output images and their associated ground truth relevance scores produced by a search system.

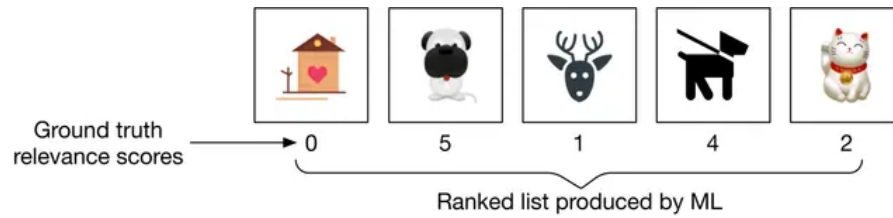


Figure 2.17: A ranked list produced by a search system

We can compute nDCG in 3 steps:

1. Compute DCG
2. Compute IDCG
3. Divide DCG by IDCG

Compute DCG: The DCG for the current ranking produced by the model is:

$$DCG_p = \sum_{i=1}^p \frac{rel_i}{\log_2(i+1)} = \frac{0}{\log_2(2)} + \frac{5}{\log_2(3)} + \frac{1}{\log_2(4)} + \frac{4}{\log_2(5)} + \frac{2}{\log_2(6)} = 6.151$$

Compute IDCG: The ideal ranking calculation is the same as the DCG calculation, except that it recommends the most relevant items first (Figure 2.18).

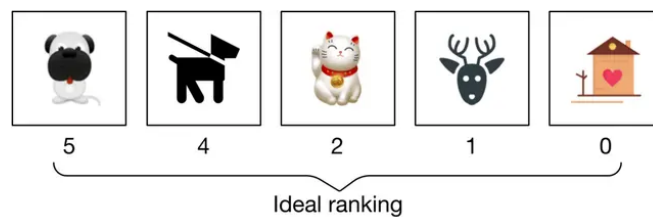


Figure 2.18: Ideal ranking list

The IDCG for the ideal ranking is:

$$IDCG_p = \sum_{i=1}^{\nu} \frac{rel_i}{\log_2(i+1)} = \frac{5}{\log_2(2)} + \frac{4}{\log_2(3)} + \frac{2}{\log_2(4)} + \frac{1}{\log_2(5)} + \frac{0}{\log_2(6)} = 8.9543$$

Divide DCG by IDCG:

$$nDCG_p = \frac{DCG_p}{IDCG_p} = \frac{6.151}{8.9543} = 0.6869$$

nDCG works well most times. Its primary shortcoming is that deriving ground truth relevance scores is not always possible. In our case, since the evaluation dataset contains similarity scores, we can use nDCG to measure the performance of the model during the offline evaluation.

Online metrics

In this section, we explore a few commonly used online metrics for measuring how quickly users can discover images they like.

Click-through rate (CTR). This metric shows how often users click on the displayed items. CTR can be calculated using the following formula:

$$CTR = \frac{\text{Number of clicked images}}{\text{Total number of suggested images}}$$

A high CTR indicates that users click on the displayed items often. CTR is commonly used as an online metric in search and recommendation systems, as we will see in later chapters.

Average daily, weekly, and monthly time spent on the suggested images. This metric shows how engaged users are with the suggested images. When the search system is accurate, we expect this metric to increase.

Serving

At serving time, the system returns a ranked list of similar images based on a query image. Figure 2.19 shows the prediction pipeline and an indexing pipeline. Let's look closer at each pipeline.

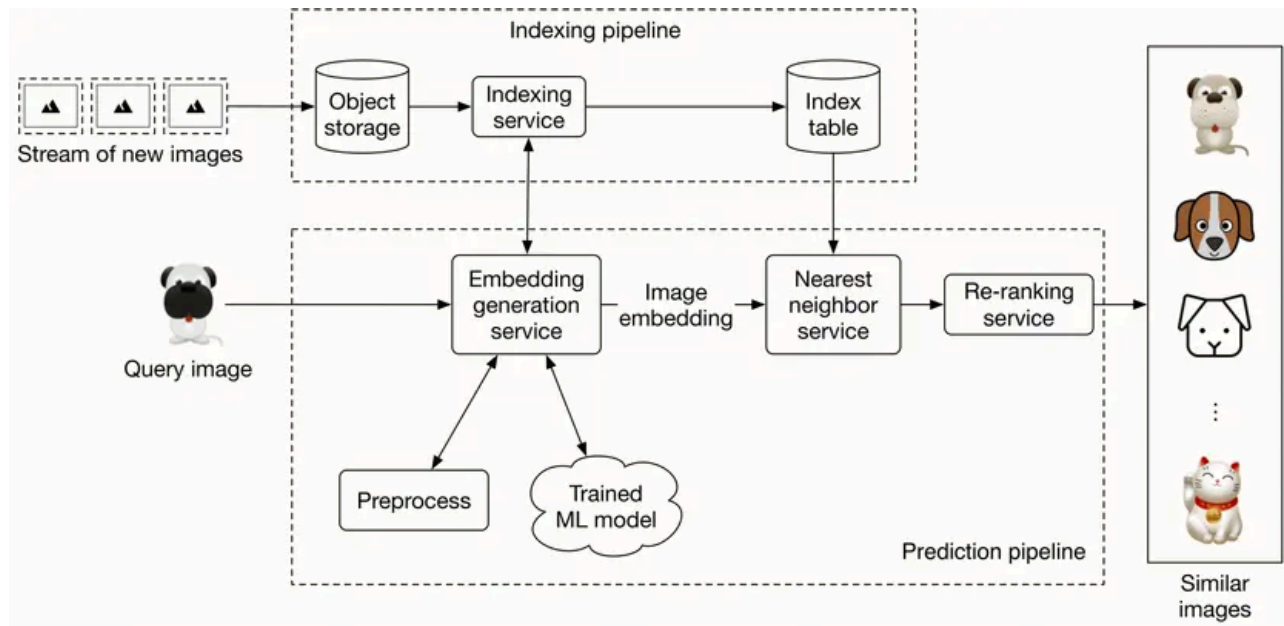


Figure 2.19: Prediction and indexing pipeline

Prediction pipeline

Embedding generation service

This service computes the embedding of the input query image. As Figure 2.20 shows, it preprocesses the image and uses the trained model to determine the embedding.

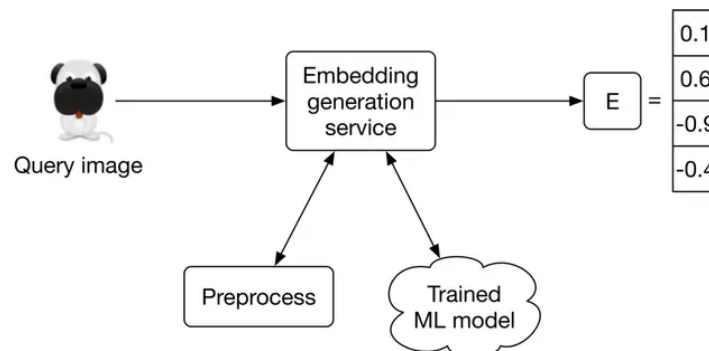


Figure 2.20: Embedding generation service

Nearest neighbor service

Once we get the embedding of the query image, we need to retrieve similar images from the embedding space. The nearest neighbor service does this.

Let's define the nearest neighbor search more formally. Given a query point "q" and a set of other points S, it finds the closest points to "q" in set S. Note that an image embedding is a point in N -dimensional space, where N is the size of the embedding vector. Figure 2.21 shows the top 3 nearest neighbors of image q. We denote the query image as q, and other images as x .

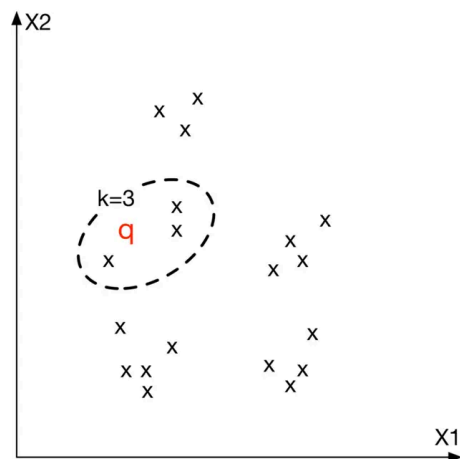


Figure 2.21: Top 3 nearest neighbors to image q in the embedding space

Re-ranking service

This service incorporates business-level logic and policies. For example, it filters inappropriate results, ensures we don't include private images, removes duplicates or near-duplicate results, and enforces other similar logic before displaying the final results to the user.

Indexing pipeline

Indexing service

All images on the platform are indexed by this service to improve search performance.

Another responsibility of the indexing service is to keep the index table updated. For example, when a creator adds a new image to the platform, the service indexes the embedding of the new image to make it discoverable by the nearest neighbor search.

Indexing increases memory usage because we store the embeddings of the entire images in an index table. Various optimizations are available to reduce memory usages, such as vector quantization [16] and product quantization [17].

Performance of nearest neighbor (NN) algorithms

Nearest neighbor search is a core component of information retrieval, search, and recommendation systems. A slight improvement in its efficiency leads to significant overall performance improvement. Given how critical this component is, the interviewer may want you to deep dive into this topic.

NN algorithms can be divided into two categories: exact and approximate. Let's examine each in more detail.

Exact nearest neighbor

Exact nearest neighbor, also called linear search, is the simplest form of NN. It works by searching the entire index table, calculating the distance of each point with the query point q , and retrieving the k nearest points. The time complexity is $O(N \times D)$, where N is the total number of points and D is the point dimension.

In a large-scale system in which N may easily run into the billions, the linear time complexity is too slow.

Approximate nearest neighbor (ANN)

In many applications, showing users similar enough items is sufficient, and there is no need to perform an exact nearest neighbor search.

In ANN algorithms, a particular data structure is used to reduce the time complexity of NN search to sublinear (e.g., $O(D \times \log N)$). They usually require up-front preprocessing or additional space.

ANN algorithms can be divided into the following three categories:

- Tree-based ANN
- Locality-sensitive hashing (LSH)-based ANN
- Clustering-based ANN

There are various algorithms within each category, and interviewers typically do not expect you to know every detail. It's adequate to have a high-level understanding of them. So, let's briefly cover each category.

Tree-based ANN

Tree-based algorithms form a tree by splitting the space into multiple partitions. Then, they leverage the characteristics of the tree to perform a faster search.

We form the tree by iteratively adding new criteria to each node. For instance, one criterion for the root node can be: gender = male. This means any point with a female attribute belongs to the left sub-tree.

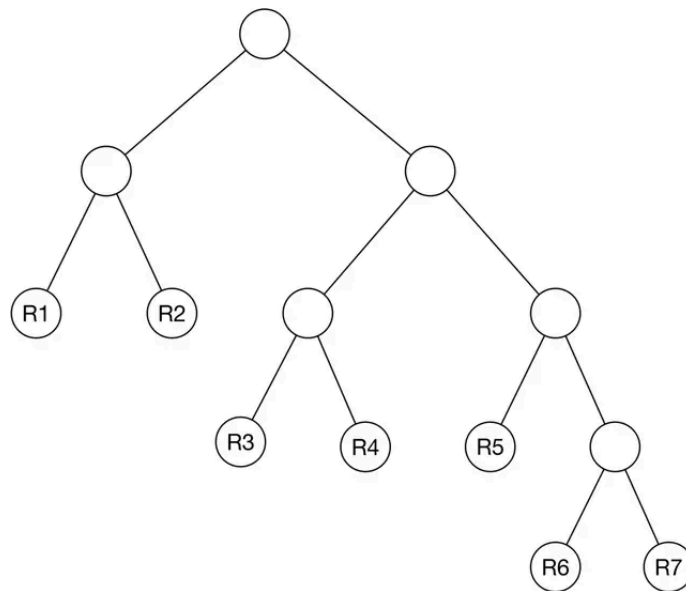


Figure 2.22: A formed tree from the points

In the tree, non-leaf nodes split the space into two partitions given the criterion. Leaf nodes indicate a particular region in space. Figure 2.23 shows an example of the space divided into 7 regions. The algorithm only searches the partition that the query point belongs to.

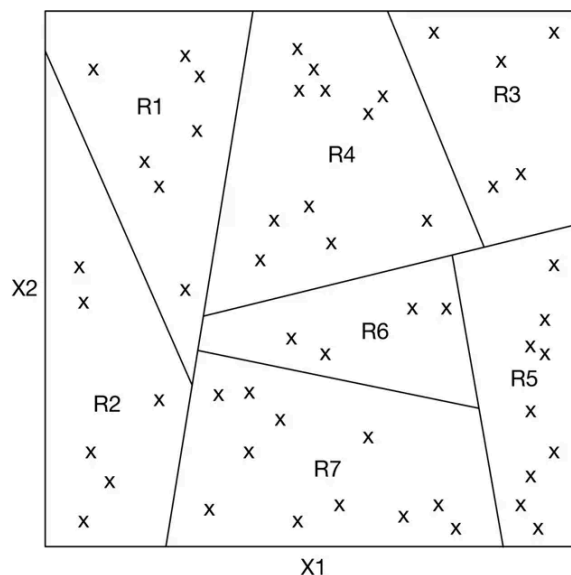


Figure 2.23: Partitioned space by the tree

Typical tree-based methods are R-trees [18], Kd-trees [19], and Annoy (Approximate Nearest Neighbor Oh Yeah) [20].

Locality sensitive hashing (LSH):

LSH uses particular hash functions to reduce the dimensions of points and group them into buckets. These hash functions map points in close proximity to each other into the same bucket. LSH searches only those points belonging to the same bucket as the query point q . You can learn more about LSH by reading [21].

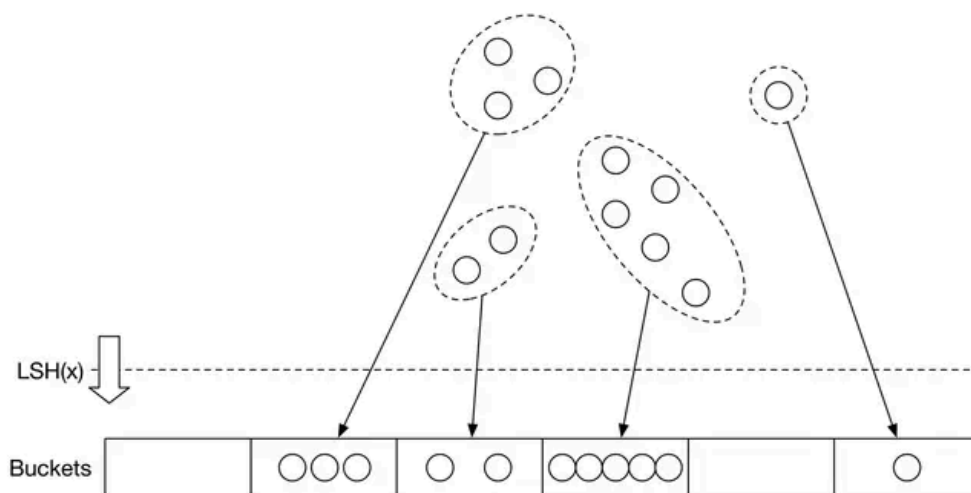


Figure 2.24: Use LSH to group the data points into buckets

Clustering-based ANN

These algorithms form clusters by grouping the points based on similarities. Once the clusters are formed, the algorithms search only the subset of points in the cluster to which the query point belongs.

Which algorithm should we use?

Results from the exact nearest neighbor method are guaranteed to be accurate. This makes it a good option when we have limited data points, or if it's required to have the exact nearest neighbors. However, when there are a large number of points, it's impractical to run the algorithm efficiently. In this case, ANN methods are commonly used. While they may not return the exact points, they are more efficient in finding the nearest points.

Given the amount of data available in today's systems, the ANN method is a more pragmatic solution. In our visual search system, we use ANN to find similar image embeddings.

For an applied ML role, the interviewer may ask you to implement ANN. Two widely used libraries are Faiss [22] (developed by Meta) and ScaNN [23] (developed by Google). Each supports the majority of methods we have described in this chapter. You are encouraged to familiarize yourself with at least one of these libraries to better understand the concepts and to gain the confidence with which to implement the nearest neighbor search in an

ML coding interview.

Other Talking Points

If there is extra time at the end of the interview, you might be asked follow-up questions or challenged to discuss advanced topics, depending on various factors such as the interviewer's preference, the candidate's expertise, role requirements, etc. Some topics to prepare for, especially for senior roles, are listed below.

- Moderate content in the system by identifying and blocking inappropriate images [24].
- Different biases present in the system, such as positional bias [25][26].
- How to use image metadata such as tags to improve search results. This is covered in Chapter 3 Google Street View Blurring System.
- Smart crop using object detection [27].
- How to use graph neural networks to learn better representations [28].
- Support the ability to search images by a textual query. We examine this in Chapter 4.
- How to use active learning [29] or human-in-the-loop [30] ML to annotate data more efficiently.

References

1. Visual search at pinterest. <https://arxiv.org/pdf/1505.07647.pdf>.
2. Visual embeddings for search at Pinterest. <https://medium.com/pinterest-engineering/unifying-visual-embeddings-for-visual-search-at-pinterest-74ea7ea103f0>.
3. Representation learning. https://en.wikipedia.org/wiki/Feature_learning.
4. ResNet paper. <https://arxiv.org/pdf/1512.03385.pdf>.
5. Transformer paper. <https://arxiv.org/pdf/1706.03762.pdf>.
6. Vision Transformer paper. <https://arxiv.org/pdf/2010.11929.pdf>.
7. SimCLR paper. <https://arxiv.org/pdf/2002.05709.pdf>.
8. MoCo paper.
9. Contrastive representation learning methods. <https://lilianweng.github.io/posts/2019-11-10-self-supervised/>.
10. Dot product. https://en.wikipedia.org/wiki/Dot_product.
11. Cosine similarity. https://en.wikipedia.org/wiki/Cosine_similarity.
12. Euclidean distance. https://en.wikipedia.org/wiki/Euclidean_distance.

13. Curse of dimensionality. https://en.wikipedia.org/wiki/Curse_of_dimensionality.
14. Curse of dimensionality issues in ML. <https://www.mygreatlearning.com/blog/understanding-curse-of-dimensionality/>.
15. Cross-entropy loss. https://en.wikipedia.org/wiki/Cross_entropy.
16. Vector quantization. [http://ws.binghamton.edu/fowler/fowler%20personal%20page/EE523_files/Ch_10_1%20VQ%20Description%20\(PPT\).pdf](http://ws.binghamton.edu/fowler/fowler%20personal%20page/EE523_files/Ch_10_1%20VQ%20Description%20(PPT).pdf).
17. Product quantization. <https://towardsdatascience.com/product-quantization-for-similarity-search-2f1f67c5fddd>.
18. R-Trees. <https://en.wikipedia.org/wiki/R-tree>.
19. KD-Tree. <https://kanoki.org/2020/08/05/find-nearest-neighbor-using-kd-tree/>.
20. Annoy. <https://towardsdatascience.com/comprehensive-guide-to-approximate-nearest-neighbors-algorithms-8b94f057d6b6>.
21. Locality-sensitive hashing. <https://web.stanford.edu/class/cs246/slides/03-lsh.pdf>.
22. Faiss library. <https://github.com/facebookresearch/faiss/wiki>.
23. ScaNN library. <https://github.com/google-research/google-research/tree/master/scann>.
24. Content moderation with ML. <https://appen.com/blog/content-moderation/>.
25. Bias in AI and recommendation systems. <https://www.searchenginejournal.com/biases-search-recommender-systems/339319/#close>.
26. Positional bias. <https://eugeneyan.com/writing/position-bias/>.
27. Smart crop. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2018/Smart-Auto-Cropping-of-Images.
28. Better search with gnns. <https://arxiv.org/pdf/2010.01666.pdf>.
29. Active learning. [https://en.wikipedia.org/wiki/Active_learning_\(machine_learning\)](https://en.wikipedia.org/wiki/Active_learning_(machine_learning)).
30. Human-in-the-loop ML. <https://arxiv.org/pdf/2108.00941.pdf>.

03

Google Street View Blurring System

Google Street View [1] is a technology in Google Maps that provides street-level interactive panoramas of many public road networks around the world. In 2008, Google created a system that automatically blurs human faces and license plates to protect user privacy. In this chapter, we design a blurring system similar to Google Street View.

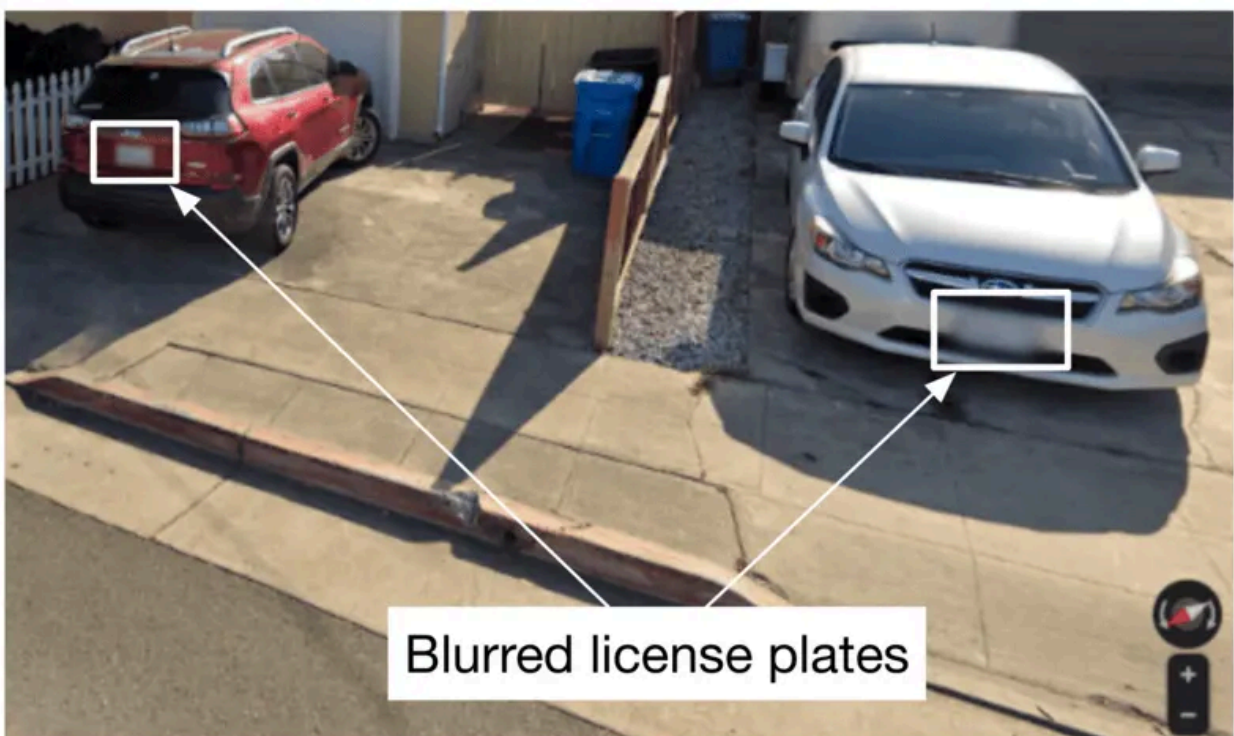


Figure 3.1: A Street View image with blurred license plates

Clarifying Requirements

Here's a typical conversation between a candidate and the interviewer.

Candidate: Is it fair to say the business objective of the system is to protect user privacy?

Interviewer: Yes.

Candidate: We want to design a system that detects all human faces and license plates in Street View images and blurs them before displaying them to users. Is that correct? Can I assume users can report images that are not correctly blurred?

Interviewer: Yes, those are fair assumptions.

Candidate: Do we have an annotated dataset for this task?

Interviewer: Let's assume we have sampled 1 million images. Human faces and license plates are manually annotated in those images.

Candidate: The dataset may not contain faces from certain racial profiles, which may cause a bias towards certain human attributes such as race, age, gender, etc. Is that a fair assumption?

Interviewer: Great point. For simplicity, let's not focus on fairness and bias today.

Candidate: My understanding is that latency is not a big concern, as the system can detect objects and blur them offline. Is that correct?

Interviewer: Yes. We can display existing images to users while new ones are being processed offline.

Let's summarize the problem statement. We want to design a Street View blurring system that automatically blurs license plates and human faces. We are given a training dataset of 1 million images with annotated human faces and license plates. The business objective of the system is to protect user privacy.

Frame the Problem as an ML Task

In this section, we frame the problem as an ML task.

Defining the ML objective

The business objective of this system is to protect user privacy by blurring visible license plates and human faces in Street View images. But protecting user privacy is not an ML objective, so we need to translate it into an ML objective that an ML system can solve. One possible ML objective is to accurately detect objects of interest in an image. If an ML system can detect those objects accurately, then we can blur the objects before displaying the images to users.

Throughout this chapter, we use "objects" instead of "human faces and license plates" for conciseness.

Specifying the system's input and output

The input of an object detection model is an image with zero or multiple objects at different locations within it. The model detects those objects and outputs their locations. Figure 3.2 shows an object detection system, along with its input and output.

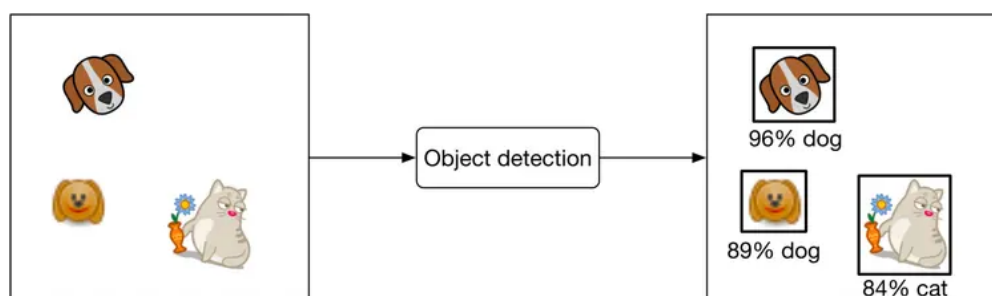


Figure 3.2: Object detection system's input-output

Choosing the right ML category

In general, an object detection system has two responsibilities:

- Predicting the location of each object in the image
- Predicting the class of each bounding box (e.g., dog, cat, etc.)

The first task is a regression problem since the location can be represented by (x, y) coordinates, which are numeric values. The second task can be framed as a multi-class classification problem.

Traditionally, object detection architectures are divided into one-stage and two-stage networks. Recently, Transformer-based architectures such as DETR [2] have shown promising results, but in this chapter, we mainly explore two-stage and one-stage architectures.

Two-stage networks

As the name implies, two separate models are used in two-stage networks:

1. **Region proposal network (RPN):** scans an image and proposes candidate regions that are likely to be objects.
2. **Classifier:** processes each proposed region and classifies it into an object class.

Figure 3.3 shows these two stages.



Figure 3.3: Two-stage network

Commonly used two-stage networks include: R-CNN [3], Fast R-CNN [4], and FasterRCNN [5].

One-stage networks

In these networks, both stages are combined. Using a single network, bounding boxes and object classes are generated simultaneously, without explicit detection of region proposals. Figure 3.4 shows a one-stage network.

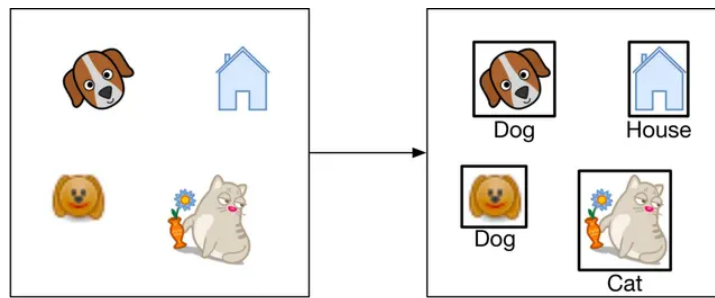


Figure 3.4: One-stage network

Commonly used one-stage networks include: YOLO [6] and SSD [7] architectures.

One-stage vs. two-stage

Two-stage networks comprise two components that run sequentially, so they are usually slower, but more accurate.

In our case, the dataset contains 1 million images, which is not huge by modern standards. This indicates that using a two-stage network doesn't increase the training cost excessively. So, for this exercise, we start with a two-stage network. When training data increases or predictions need to be made faster, we can switch to one-stage networks.

Data Preparation

Data engineering

In the Introduction chapter, we discussed data engineering fundamentals. Additionally, it's usually a good idea to discuss the specific data available for the task at hand. For this problem, we have the following data available:

- Annotated dataset
- Street View images

Let's discuss each in more detail.

Annotated dataset

Based on the requirements, we have 1 million annotated images. Each image has a list of bounding boxes and associated object classes. Table 3.1 shows data points from the dataset:

Image path	Objects	Bounding boxes
dataset/image1.jpg	human face	[10,10,25,50]
	human face	[120,180,40,70]
	license plate	[80,95,35,10]
dataset/image2.jpg	human face	[170,190,30,80]
dataset/image3.jpg	license plate	[25,30,210,220]
	human face	[30,40,30,60]

Table 3.1: A few data points from the annotated dataset

Each bounding box is a list of 4 numbers: top left X and Y coordinates, followed by the width and height of the object.

Street View images

These are the Street View images collected by the data sourcing team. The ML system processes these images to detect human faces and license plates. Table 3.2 shows the metadata of the images.

Image path	Location (lat, lng)	Pitch, Yaw, Roll	Timestamp
tmp/image1.jpg	(37.432567, -122.143993)	(0,10,20)	1646276421
tmp/image2.jpg	(37.387843, -122.091086)	(0,10,-10)	1646276539
tmp/image3.jpg	(37.542081, -121.997640)	(10,-20,45)	1646276752

Table 3.2: Metadata of Street View images

Feature engineering

During feature engineering, we first apply standard , such as resizing and normalization. After that, we increase the size of the dataset by using a data augmentation technique. Let's take a closer look at this.

Data augmentation

A technique called data augmentation involves adding slightly modified copies of original data, or creating new

data artificially from the original. As the dataset size increases, the model is able to learn more complex patterns. This technique is especially useful when the dataset is imbalanced, as it increases the number of data points in minority classes.

A special type of data augmentation is image augmentation. Among the commonly used augmentation techniques are:

- Random crop
- Random saturation
- Vertical or horizontal flip
- Rotation and/or translation
- Affine transformations
- Changing brightness, saturation, or contrast

Figure 3.5 shows an image with various data augmentation techniques applied to it.



Figure 3.5: Augmented images (source [8])

It is important to note that with certain types of augmentations, the ground truth bounding boxes also need to be transformed. For example, when rotating or flipping the original image, the ground truth bounding boxes must also be transformed.

Data augmentation is used in offline or online forms.

- **Offline:** Augment images before training
- **Online:** Augment images on the fly during training

Online vs. offline: In offline data augmentation, training is faster since no additional augmentation is needed.

However, it requires additional storage to store all the augmented images. While online data augmentation slows down training, it does not consume additional storage.

The choice between online and offline data augmentation depends upon the storage and computing power constraints. What is more important in an interview is that you talk about different options and discuss trade-offs. In our case, we perform offline data augmentation.

Figure 3.6 shows the dataset preparation flow. With preprocessing, images are resized, scaled, and normalized. With image augmentation, the number of images is increased. Let's say the number increases from 1 million to 10 million.

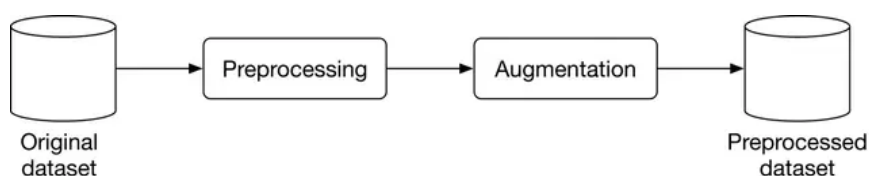


Figure 3.6: Dataset preparation workflow

Model Development

Model selection

As mentioned in the "Frame the Problem as an ML Task" section, we opt for two-stage networks. Figure 3.7 shows a typical two-stage architecture.

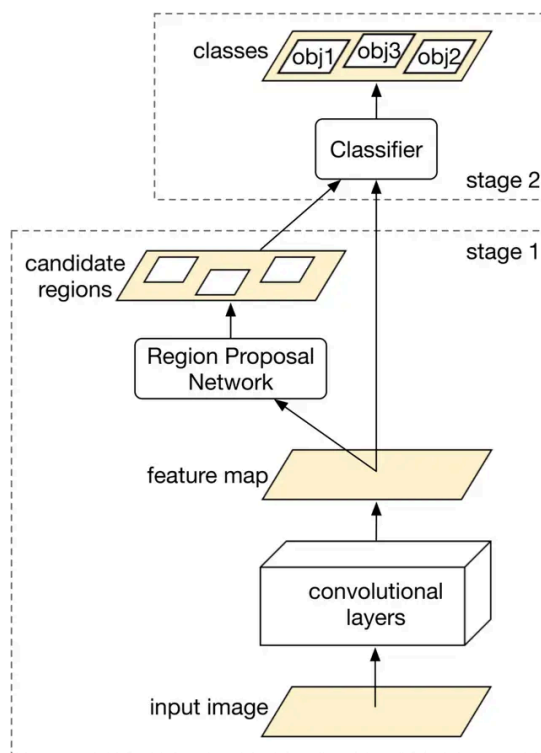


Figure 3.7: Two-stage object detection network

Let's examine each component.

Convolutional layers

Convolutional layers [9] process the input image and output a feature map.

Region Proposal Network (RPN)

RPN proposes candidate regions that may contain objects. It uses neural networks as its architecture and takes the feature map produced by convolutional layers as input and outputs candidate regions in the image.

Classifier

The classifier determines the object class of each candidate region. It takes the feature map and the proposed candidate regions as input, and assigns an object class to each region. This classifier is usually based on neural networks.

In ML system design interviews, you are generally not expected to discuss the architecture of these neural networks.

Model training

The process of training a neural network usually involves three steps: forward propagation, loss calculation, and backward propagation. Readers are expected to be familiar with these steps, but for more information, see [10]. In this section, we discuss the loss functions commonly used to detect objects.

An object detection model is expected to perform two tasks well. First, the bounding boxes of the objects predicted should have a high overlap with the ground truth bounding boxes. This is a regression task. Second, the predicted probabilities for each object class should be accurate. This is a classification task. Let's define a loss function for each.

Regression loss: This loss measures how aligned the predicted bounding boxes are with the ground truth bounding boxes. We use standard regression loss functions, such as Mean Squared Error (MSE) [11], and denote it by L_{reg} :

$$L_{reg} = \frac{1}{M} \sum_{i=1}^M \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + (w_i - \hat{w}_i)^2 + (h_i - \hat{h}_i)^2 \right]$$

Where:

- M : total number of predictions
- x_i : ground truth top left x coordinate
- $\hat{x}_i$: predicted top left x coordinate
- y_i : ground truth top left y coordinate
- $\hat{y}_i$: predicted top left y coordinate
- w_i : ground truth width
- $\hat{w}_i$: predicted width

- h_i : ground truth height
- $\hat{h}_i$: predicted height

Classification loss: This measures how accurate the predicted probabilities are for each detected object. Here, we use a standard classification loss, such as log loss (crossentropy) [12] and denote it by L_{cls} :

$$L_{cls} = -\frac{1}{M} \sum_{i=1}^M \sum_{c=1}^C y_c \log \hat{y}_c$$

Where:

- M : total number of detected bounding boxes
- C : total number of classes
- y_i : ground truth label for detection i
- $\hat{y}_i$: predicted class label for detection i

To define a final loss that measures the model's overall performance, we combine the classification loss and regression loss weighted by a balancing parameter λ :

$$L = L_{cls} + \lambda L_{reg}$$

Evaluation

During an interview, it is crucial to discuss how to evaluate an ML system. The interviewer usually wants to know which metrics you'd choose and why. This section describes how object detection systems are usually evaluated, and then selects important metrics for offline and online evaluations.

An object detection model usually needs to detect N different objects in an image. To measure the overall performance of the model, we evaluate each object separately and then average the results.

Figure 3.8 shows the output of an object detection model. It shows both the ground truth and detected bounding boxes. As shown, the model detected 6 bounding boxes, while we only have two instances of the object.

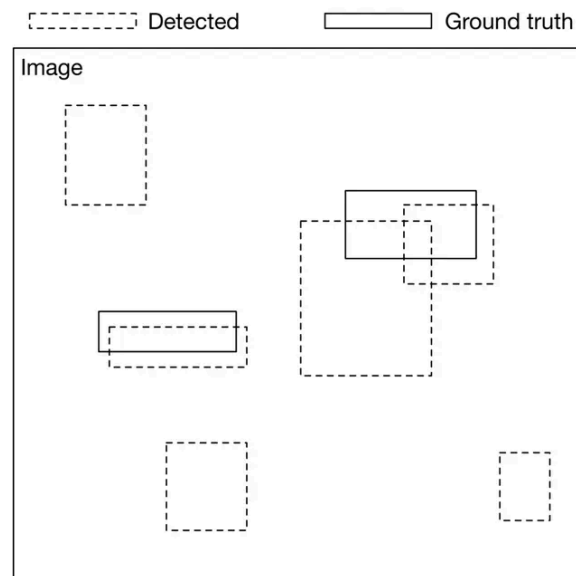


Figure 3.8: Ground truth and detected bounding boxes

When is a predicted bounding box considered correct? To answer this question, we need to understand the definition of Intersection Over Union.

Intersection Over Union (IOU): IOU measures the overlap between two bounding boxes. Figure 3.9 shows a visual representation of IOU.

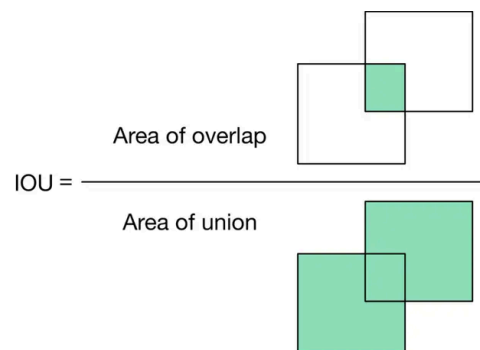


Figure 3.9: IOU formula

IOU determines whether a detected bounding box is correct. An IOU of 1 is ideal, indicating the detected bounding box and the ground truth bounding box are fully aligned. In practice, it's rare to see an IOU of 1. A higher IOU means the predicted bounding box is more accurate. An IOU threshold is usually used to determine whether a detected bounding box is correct (true positive) or incorrect (false positive). For example, an IOU threshold of 0.7 means any detection that has an overlap of 0.7 or higher with a ground truth bounding box, is a correct detection.

Now we know what IOU is and how to determine correct and incorrect bounding box predictions, let's discuss metrics for offline evaluation.

Offline metrics

Model development is an iterative process. We use offline metrics to quickly evaluate the performance of newly developed models. Here are some metrics that might be useful for the object detection system:

- Precision
- Average precision
- Mean average precision

Precision

This is the fraction of correct detections among all detections across all images. A high precision value shows the system's detections are more reliable.

$$\text{Precision} = \frac{\text{Correct detections}}{\text{Total detections}}$$

In order to calculate precision, we need to pick an IOU threshold. Let's use an example to better understand this.

Figure 3.10 shows a set of ground truth bounding boxes and detected bounding boxes, with their respective IOUs.

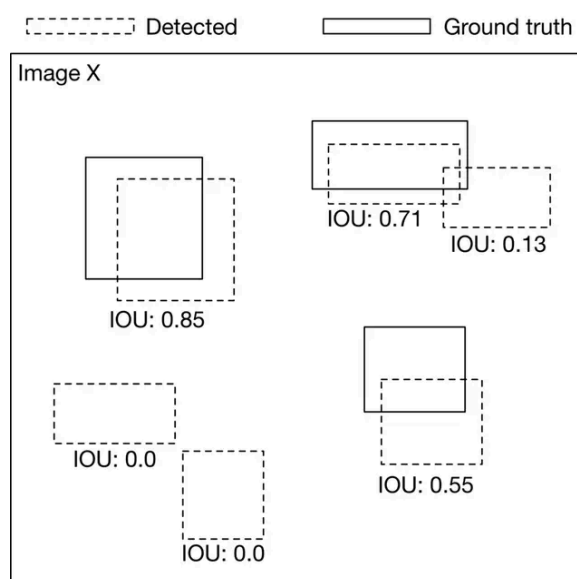


Figure 3.10: Ground truth bounding boxes and detected bounding boxes

Let's calculate precision for three different IOU thresholds: 0.7, 0.5, and 0.1.

- **IOU threshold = 0.7** Out of the six total detections, two have an IOU above 0.7. Therefore, we have two correct predictions at this threshold.

$$\text{Precision}_{0.7} = \frac{\text{Correct detections}}{\text{Total detections}} = \frac{2}{6} = 0.33$$

- **IOU threshold = 0.5** At this threshold, we have three detections with IOU above 0.5 :

$$\text{Precision}_{0.5} = \frac{\text{Correct detections}}{\text{Total detections}} = \frac{3}{6} = 0.5$$

- **IOU threshold = 0.1** This time, we have four correct detections:

$$\text{Precision}_{0.1} = \frac{\text{Correct detections}}{\text{Total detections}} = \frac{4}{6} = 0.67$$

As you may have noticed, the primary disadvantage of this metric is that precision varies with different IOU thresholds. Therefore, it's difficult to understand the model's overall performance by looking at a precision score with a particular IOU threshold. Average precision addresses this limitation.

Average Precision (AP) This metric computes precision across various IOU thresholds and calculates their average. The AP formula is:

$$AP = \int_0^1 P(r)dr$$

Where $P(r)$ is the precision at IOU threshold r .

The above formula can be approximated by a discrete summation over a predefined list of thresholds. For example, in the pascal VOC2008 benchmark [13], the AP is calculated across 11 evenly-spaced threshold values.

$$AP = \frac{1}{11} \sum_{n=0}^{n=10} P(n)$$

AP summarizes the model's overall precision for a specific object class (e.g., human faces). To measure the model's overall precision across all object classes (e.g., human faces and license plates), we need to use mean average precision.

Mean average precision (mAP) This is the average of AP across all object classes. This metric summarizes the model's overall performance. Here is the formula:

$$mAP = \frac{1}{C} \sum_{c=1}^C AP_c$$

Where C is the total number of object classes the model detects.

The mAP metric is commonly used to evaluate object detection systems. To find out which thresholds are used in standard benchmarks, refer to [14] [15].

Online metrics

According to the requirements, the system needs to protect the privacy of individuals. One way to measure this is to count the number of user reports and complaints. We can also rely on human annotators to spot-check the percentage of incorrectly blurred images. Other metrics that measure bias and fairness are also critical. For example, we want to blur human faces equally well across different races and age groups. But measuring bias, as

stated in the requirements, is out of the scope.

To conclude the evaluation section, we use mAP and AP as our offline metrics. mAP measures the overall precision of the model, while AP gives us insight into the precision of the model in particular classes. The main metric of the online evaluation is "user reports."

Serving

In this section, we first talk about a common problem that may occur in object detection systems: overlapping bounding boxes. Next, we propose an overall ML system design.

Overlapping bounding boxes

When running an object detection algorithm on an image, it is very common to see bounding boxes overlap. This is because the RPN network proposes various highly overlapping bounding boxes around each object. It is important to narrow down these bounding boxes to a single bounding box per object during inference.

A widely used solution is an algorithm called "Non-maximum suppression" (NMS) [16]. Let's examine how it works.

NMS

NMS is a post-processing algorithm designed to select the most appropriate bounding boxes. It keeps highly confident bounding boxes and removes overlapping bounding boxes. Figure 3.11 shows an example.

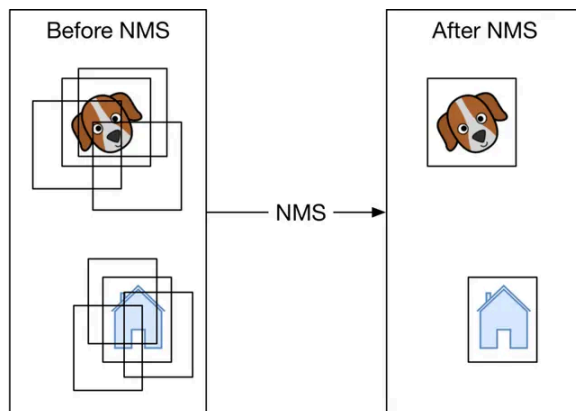


Figure 3.11: Before and after applying NMS

NMS is a commonly asked algorithm in ML system design interviews, so you're encouraged to have a good understanding of it [17].

ML system design

As illustrated in Figure 3.12, we propose an ML system design for the blurring system.

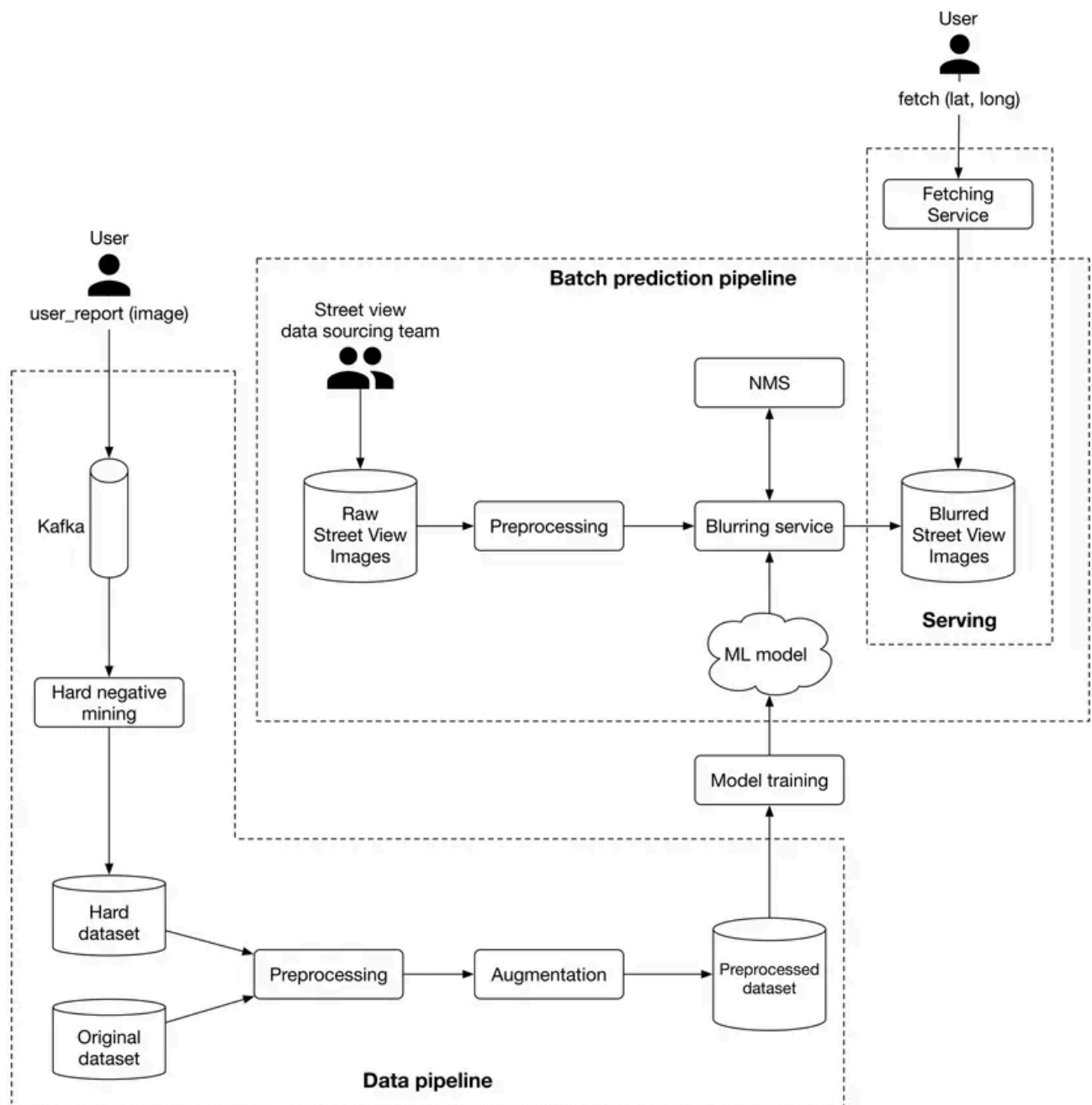


Figure 3.12: ML system design

Let's examine each pipeline in more detail.

Batch prediction pipeline

Based on the requirements gathered, latency is not a big concern because we can display existing images to users while new ones are being processed. Since instant results are not required, we can utilize batch prediction and precompute the object detection results.

Preprocessing Raw images are preprocessed by this component. This section does not discuss the preprocess operations as we have already discussed them in the feature engineering section.

Blurring service This performs the following operations on a Street View image:

1. Provides a list of objects detected in the image.
2. Refines the list of detected objects using the NMS component.
3. Blurs detected objects.
4. Stores the blurred image in object storage (Blurred Street View images).

Note that the preprocessing and blurring services are separate in the design. The reason is preprocessing images tends to be a CPU-bound process, whereas blurring service relies on GPU. Separating these services has two benefits:

- Scale the services independently based on the workload each receives.
- Better utilization of CPU and GPU resources.

Data pipeline

This pipeline is responsible for processing users' reports, generating new training data, and preparing training data to be used by the model. Data pipeline components are mostly self-explanatory. Hard negative mining is the only component that needs more explanation.

Hard negative mining. Hard negatives are examples that are explicitly created as negatives out of incorrectly predicted examples, and then added to the training dataset. When we retrain the model on the updated training dataset, it should perform better.

Other Talking Points

If time allows, here are some additional points to discuss:

- How Transformer-based object detection architectures differ from one-stage or twostage models, and what are their pros and cons [18].
- Distributed training techniques to improve object detection on a larger dataset [19] [20].
- How General Data Protection Regulation (GDPR) in Europe may affect our system [21].
- Evaluate bias in face detection systems [22] [23].
- How to continuously fine-tune the model [24].
- How to use active learning [25] or human-in-the-loop ML [26] to select data points for training.

References

1. Google Street View. <https://www.google.com/streetview>.
2. DETR. <https://github.com/facebookresearch/detr>.
3. RCNN family. <https://lilianweng.github.io/posts/2017-12-31-object-recognition-part-3>.
4. Fast R-CNN paper. <https://arxiv.org/pdf/1504.08083.pdf>.
5. Faster R-CNN paper. <https://arxiv.org/pdf/1506.01497.pdf>.
6. YOLO family. <https://pyimagesearch.com/2022/04/04/introduction-to-the-yolo-family>.
7. SSD. <https://jonathan-hui.medium.com/ssd-object-detection-single-shot-multibox-detector-for-real-time-processing-9bd8deac0e06>.
8. Data augmentation techniques. <https://www.kaggle.com/getting-started/190280>.

9. CNN. https://en.wikipedia.org/wiki/Convolutional_neural_network.
10. Forward pass and backward pass. <https://www.youtube.com/watch?v=qzPQ8cEsVK8>.
11. MSE. https://en.wikipedia.org/wiki/Mean_squared_error.
12. Log loss. https://en.wikipedia.org/wiki/Cross_entropy.
13. Pascal VOC. <http://host.robots.ox.ac.uk/pascal/VOC/voc2008/index.html>.
14. COCO dataset evaluation. <https://cocodataset.org/#detection-eval>.
15. Object detection evaluation. <https://github.com/rafaelpadilla/Object-Detection-Metrics>.
16. NMS. <https://en.wikipedia.org/wiki/NMS>.
17. Pytorch implementation of NMS. <https://learnopencv.com/non-maximum-suppression-theory-and-implementation-in-pytorch/>.
18. Recent object detection models. <https://viso.ai/deep-learning/object-detection/>.
19. Distributed training in Tensorflow. https://www.tensorflow.org/guide/distributed_training.
20. Distributed training in Pytorch. https://pytorch.org/tutorials/beginner/dist_overview.html.
21. GDPR and ML. <https://www.oreilly.com/radar/how-will-the-gdpr-impact-machine-learning>.
22. Bias and fairness in face detection. <http://sibgrapi.sid.inpe.br/col/sid.inpe.br/sibgrapi/2021/09.04.19.00/doc/103.pdf>.
23. AI fairness. <https://www.kaggle.com/code/alexisbcook/ai-fairness>.
24. Continual learning. <https://towardsdatascience.com/tag/fine-tuning/>.
25. Active learning. [https://en.wikipedia.org/wiki/Active_learning_\(machine_learning\)](https://en.wikipedia.org/wiki/Active_learning_(machine_learning)).
26. Human-in-the-loop ML. <https://arxiv.org/pdf/2108.00941.pdf>.

04

YouTube Video Search

On video-sharing platforms such as YouTube, the number of videos can quickly grow into the billions. In this chapter, we design a video search system that can efficiently handle this volume of content. As shown in Figure 4.1, the user enters text into the search box, and the system displays the most relevant videos for the given text.

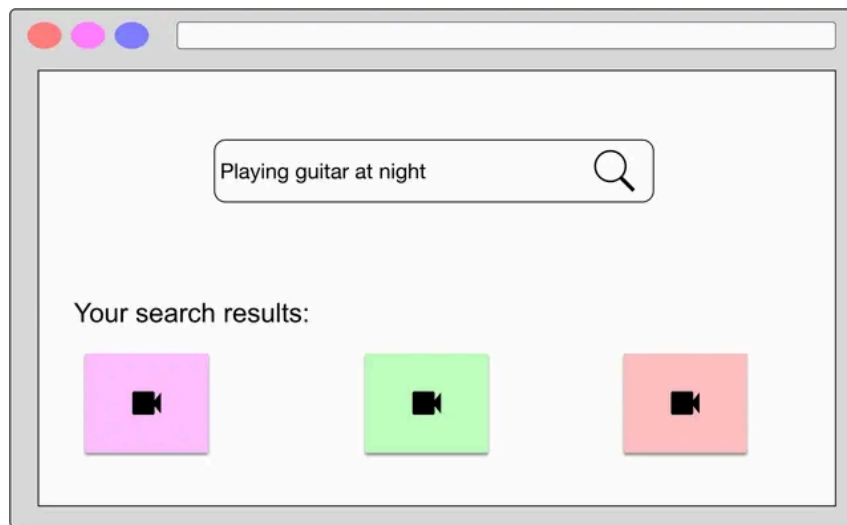


Figure 4.1: Searching videos with a text query

Clarifying Requirements

Here is a typical interaction between a candidate and an interviewer.

Candidate: Is the input query text-only, or can users search with an image or video?

Interviewer: Text queries only.

Candidate: Is the content on the platform only in video form? How about images or audio files?

Interviewer: The platform only serves videos.

Candidate: The YouTube search system is very complex. Can I assume the relevancy of a video is determined solely by its visual content and the textual data associated with the video, such as the title and description?

Interviewer: Yes, that's a fair assumption.

Candidate: Is there any training data available?

Interviewer: Yes, let's assume we have ten million pairs of (video, text query).

Candidate: Do we need to support other languages in the search system?

Interviewer: For simplicity, let's assume only English is supported.

Candidate: How many videos are available on the platform?

Interviewer: One billion videos.

Candidate: Do we need to personalize the results? Should we rank the results differently for different users, based on their past interactions?

Interviewer: As opposed to recommendation systems where personalization is essential, we do not necessarily have to personalize results in search systems. To simplify the problem, let's assume no personalization is required.

Let's summarize the problem statement. We are asked to design a search system for videos. The input is a text query, and the output is a list of videos that are relevant to the text query. To search for relevant videos, we leverage both the videos' visual content and textual data. We are given a dataset of ten million $\langle \text{video}, \text{text query} \rangle$ pairs for model training.

Frame the Problem as an ML Task

Defining the ML objective

Users expect search systems to provide relevant and useful results. One way to translate this into an ML objective is to rank videos based on their relevance to the text query.

Specifying the system's input and output

As shown in Figure 4.2, the search system takes a text query as input and outputs a ranked list of videos sorted by their relevance to the text query.

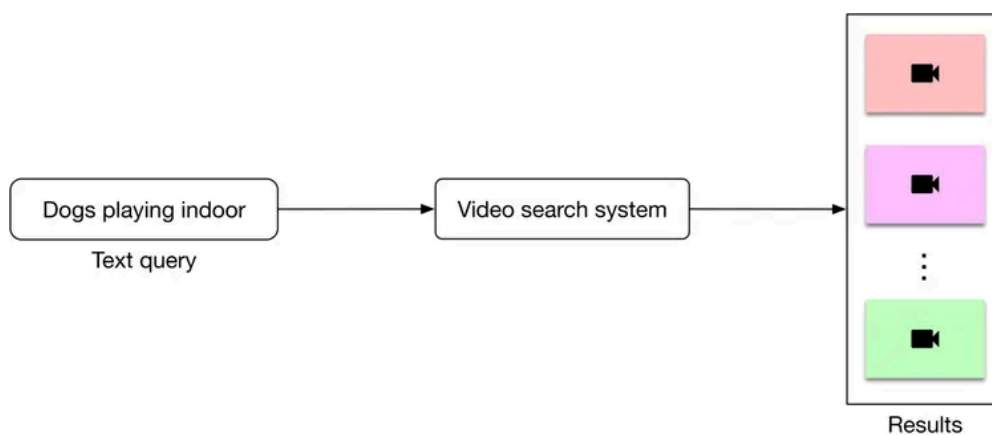


Figure 4.2: Video search system's input-output

Choosing the right ML category

In order to determine the relevance between a video and a text query, we utilize both visual content and the video's textual data. An overview of the design can be seen in Figure 4.3.

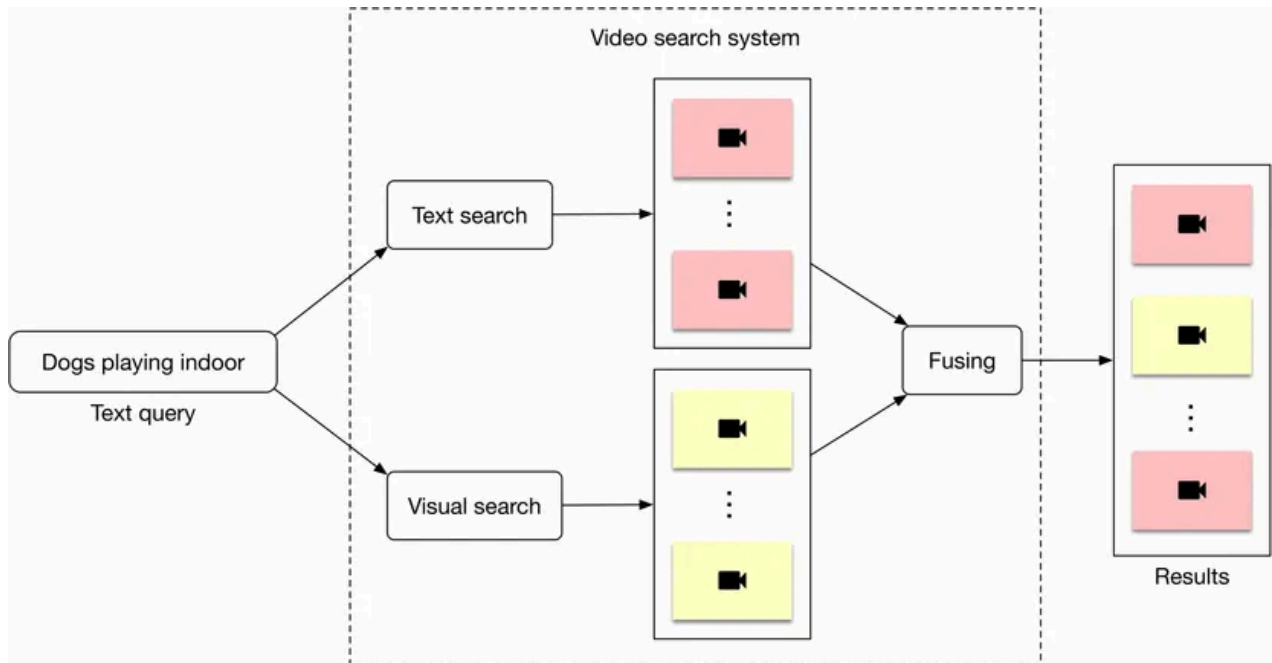


Figure 4.3: High-level overview of the search system

Let's briefly discuss each component.

Visual search

This component takes a text query as input and outputs a list of videos. The videos are ranked based on the similarity between the text query and the videos' visual content.

Representation learning is a commonly used approach to search for videos by processing their visual content. In this approach, text query and video are encoded separately using two encoders. As shown in Figure 4.4, the ML model contains a video encoder that generates an embedding vector from the video, and a text encoder that generates an embedding vector from the text. The similarity score between the video and the text is calculated using the dot product of their representations.

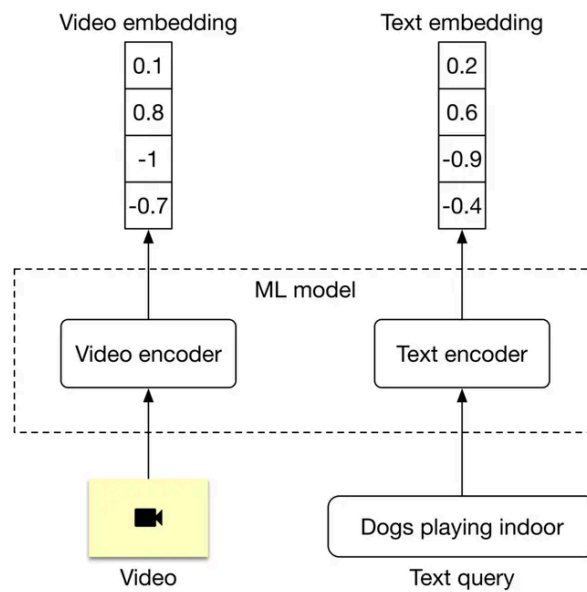


Figure 4.4: ML model's input-output

In order to rank videos that are visually and semantically similar to the text query, we compute the dot product between the text and each video in the embedding space, then rank the videos based on their similarity scores.

Text search

Figure 4.5 shows how text search works when a user types in a text query: "dogs playing indoor". Videos with the most similar titles, descriptions, or tags to the text query are shown as the output.

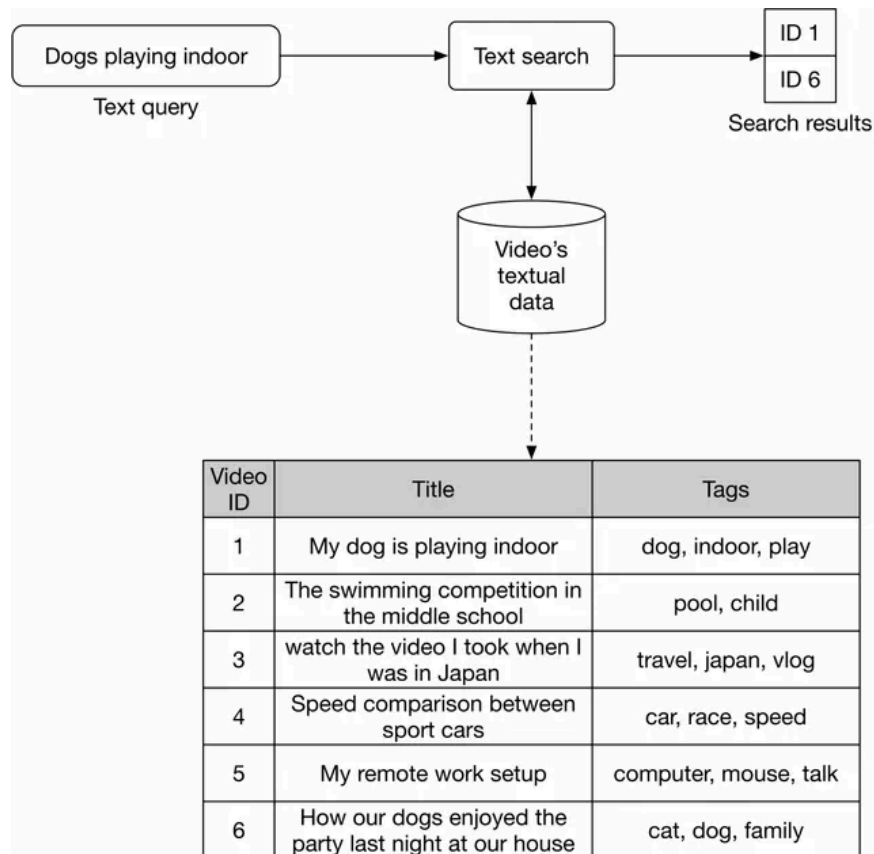


Figure 4.5: Text search

The inverted index is a common technique for creating the text-based search component, allowing efficient full-text search in databases. Since inverted indexes aren't based on machine learning, there is no training cost. A popular search engine companies often use is Elasticsearch, which is a scalable search engine and document store. For more details and a deeper understanding of Elasticsearch, refer to [1].

Data Preparation

Data engineering

Since we are given an annotated dataset to train and evaluate the model, it's not necessary to perform any data engineering. Table 4.1 shows what the annotated dataset might look like.

Video name	Query	Split type
76134.mp4	Kids swimming in a pool!	Training
92167.mp4	Celebrating graduation	Training
2867.mp4	A group of teenagers playing soccer	Validation
28543.mp4	How Tensorboard works	Validation
70310.mp4	Road trip in winter	Test

Table 4.1: Annotated dataset

Feature engineering

Almost all ML algorithms accept only numeric input values. Unstructured data such as texts and videos need to be converted into a numerical representation during this step. Let's take a look at how to prepare the text and video data for the model.

Preparing text data

As shown in Figure 4.6, text is typically represented as a numerical vector using three steps: text normalization, tokenization, and tokens to IDs [2].

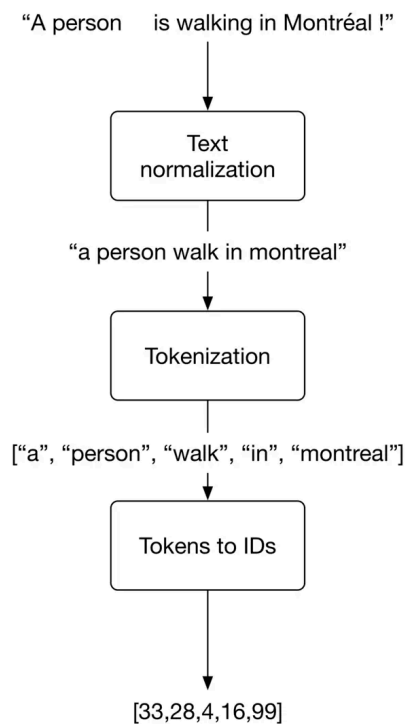


Figure 4.6: Represent a text with a numerical vector

Let's take a look at each step in more detail.

Text normalization

Text normalization - also known as text cleanup - ensures words and sentences are consistent. For example, the same word may be spelled slightly differently; as in "dog", "dogs", and "DOG!" all refer to the same thing but are spelled in different ways. The same is true for sentences. Take these two sentences, for example:

- "A person walking with his dog in Montréal !"
- "a person walks with his dog, in Montreal."

Both sentences mean the same, but have differing punctuation and verb forms. Here are some typical methods for text normalization:

- Lowercasing: make all letters lowercase, as this does not change the meaning of words or sentences
- Punctuation removal: remove punctuation from the text. Common punctuation marks are the period, comma, question mark, exclamation point, etc.
- Trim whitespaces: trim leading, trailing, and multiple whitespaces
- Normalization Form KD (NFKD) [3]: decompose combined graphemes into a combination of simple ones
- Strip accents: remove accent marks from words. For example: Màlaga → Malaga, Noël → Noel
- Lemmatization and stemming: identify a canonical representative for a set of related word forms. For example: walking, walks, walked → walk

Tokenization

Tokenization is the process of breaking down a piece of text into smaller units called tokens. Generally, there are three types of tokenization:

- Word tokenization: split the text into individual words based on specific delimiters. For example, a phrase like "I have an interview tomorrow" becomes ["I", "have", "an", "interview", "tomorrow"]
- Subword tokenization: split text into subwords (or n-gram characters)
- Character tokenization: split text into a set of characters The details of different tokenization algorithms are not usually a strong focus in ML system design interviews. If you are interested to learn more, refer to [4].

Tokens to IDs

Once we have the tokens, we need to convert them to numerical values (IDs). The representation of tokens with numerical values can be done in two ways:

- Lookup table
- Hashing

Lookup table. In this method, each unique token is mapped to an ID. Next, a lookup table is created to store these 1 : 1 mappings. Figure 4.7 shows what the mapping table might look like.

Word	ID
⋮	
animals	18
⋮	
art	35
⋮	
car	128
⋮	
insurance	426
⋮	
travel	1239
⋮	

Figure 4.7: A lookup table

Hashing. Hashing, also called "feature hashing" or "hashing trick," is a memory-efficient method that uses a hash function to obtain IDs, without keeping a lookup table. Figure 4.8 shows how a hash function is used to convert words to IDs.

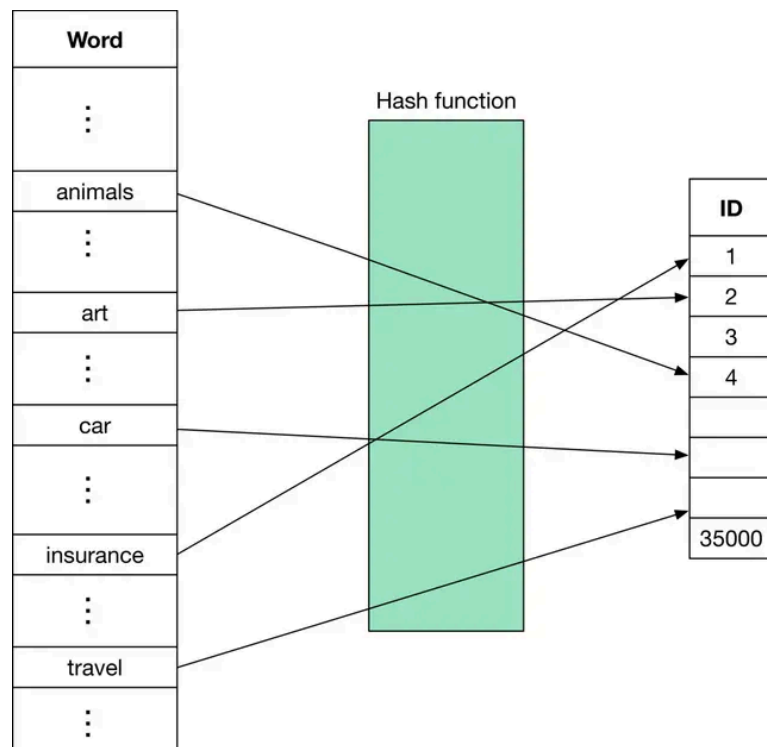


Figure 4.8: Use hashing to obtain word IDs

Let's compare the lookup table with the hashing method.

	Lookup table	Hashing
Speed	✓ Quick to convert tokens to IDs	✗ Need to compute hash function to convert tokens to IDs
ID to token	✓ Easy to convert IDs to tokens using a reverse index table	✗ Not possible to convert IDs to tokens
Memory	✗ The table is stored in memory. A large number of tokens will result in an increase in memory required	✓ The hash function is sufficient to convert any token to its ID
Unseen tokens	✗ New or unseen words cannot be properly handled	✓ Easily handles new or unseen words by applying the hash function to any word
Collisions [5]	✓ No collision issue	✗ Collisions are a potential problem

Table 4.2: Lookup table vs. feature hashing

Preparing video data

Figure 4.9 shows a typical workflow for preprocessing a raw video.

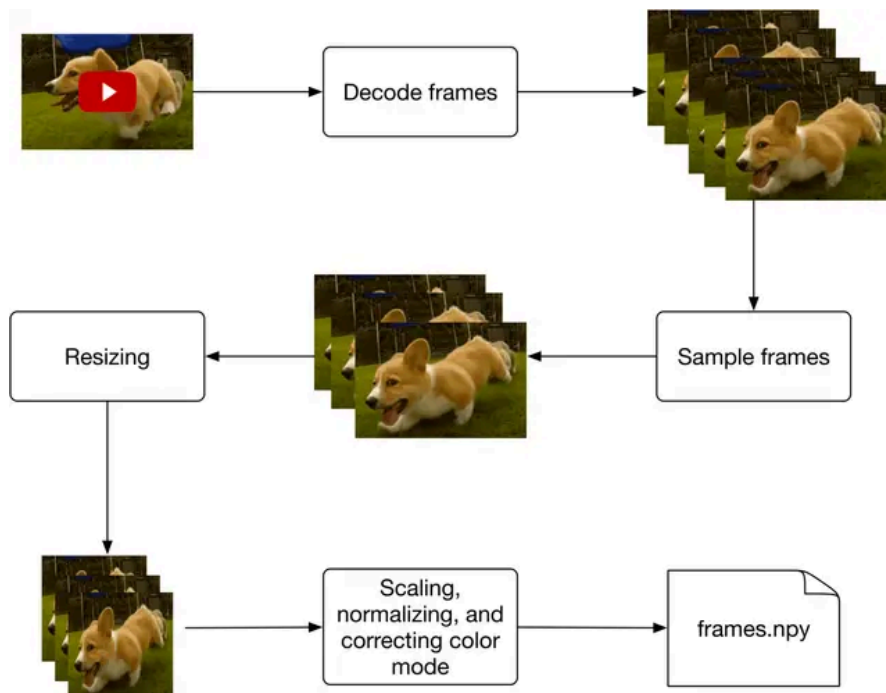


Figure 4.9: Video preprocessing workflow

Model Development

Model selection

As discussed in the "Framing the problem as an ML task" section, text queries are converted into embeddings by a text encoder, and videos are converted into embeddings by a video encoder. In this section, we examine possible model architectures for each encoder.

A typical text encoder's input and output are shown in Figure 4.10.

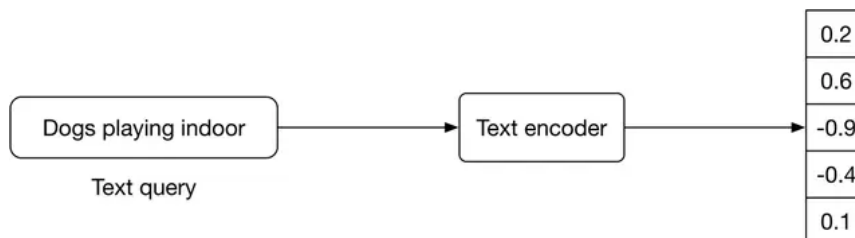


Figure 4.10: Text encoder's input-output

The text encoder converts text into a vector representation [6]. For example, if two sentences have similar meanings, their embeddings are more similar. To build the text encoder, two broad categories are available: statistical methods and ML-based methods. Let's examine each.

Statistical methods

Those methods rely on statistics to convert a sentence into a feature vector. Two popular statistical methods are:

- Bag of Words (BoW)
- Term Frequency Inverse Document Frequency (TF-IDF)

BoW. This method converts a sentence into a fixed-length vector. It models sentence word occurrences by creating a matrix with rows representing sentences, and columns representing word indices. An example of BoW is shown in Figure 4.11.

	best	holiday	is	nice	person	this	today	trip	very	with
this person is nice very nice	0	0	1	2	1	1	0	0	1	0
today is holiday	0	1	1	0	0	0	1	0	0	0
this trip with best person is best	2	0	1	0	1	1	0	1	0	1

Figure 4.11: BoW representations of different sentences

BoW is a simple method that computes sentence representations fast, but has the following limitations:

- It does not consider the order of words in a sentence. For example, "let's watch TV after work" and "let's work after watch TV" would have the same BoW representation.
- The obtained representation does not capture the semantic and contextual meaning of the sentence. For example, two sentences with the same meaning but different words have a totally different representation.
- The representation vector is sparse. The size of the representation vector is equal to the total number of unique tokens we have. This number is usually very large, so each sentence representation is mostly filled with zeros.

TF-IDF. This is a numerical statistic intended to reflect how important a word is to a document in a collection or corpus. TF-IDF creates the same sentence-word matrix as in BoW, but it normalizes the matrix based on the frequency of words. To learn more about the mathematics behind this, refer to [7].

Since TF-IDF gives less weight to frequent words, its representations are usually better than BoW. However, it has the following limitations:

- A normalization step is needed to recompute term frequencies when a new sentence is added.
- It does not consider the order of words in a sentence.
- The obtained representation does not capture the semantic meaning of the sentence.
- The representations are sparse.

In summary, statistical methods are usually fast. However, they do not capture the contextual meaning of sentences, and the representations are sparse. ML-based methods address those issues.

ML-based methods

In these methods, an ML model converts sentences into meaningful word embeddings so that the distance between two embeddings reflects the semantic similarity of the corresponding words. For example, if two words, such as "rich" and "wealth" are semantically similar, their embeddings are close in the embedding space. Figure 4.12 shows a simple visualization of word embeddings in the 2D embedding space. As you can see, similar words are grouped together.

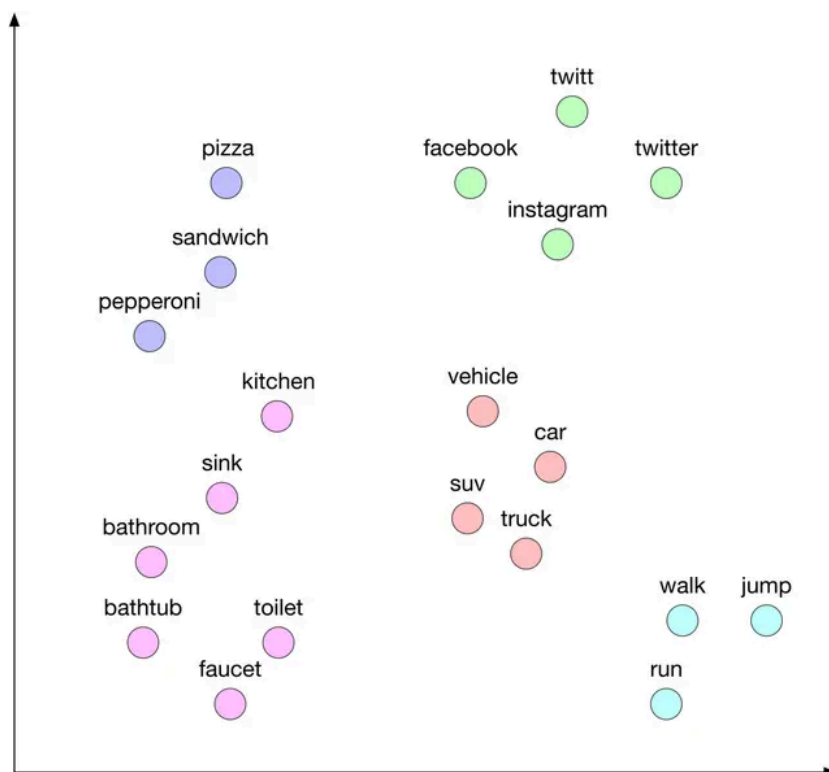


Figure 4.12: Words in the 2D embedding space

There are three common ML-based approaches for transforming texts into embeddings:

- Embedding (lookup) layer
- Word2vec
- Transformer-based architectures

Embedding (lookup) layer In this approach, an embedding layer is employed to map each ID to an embedding vector. Figure 4.13 shows an example.

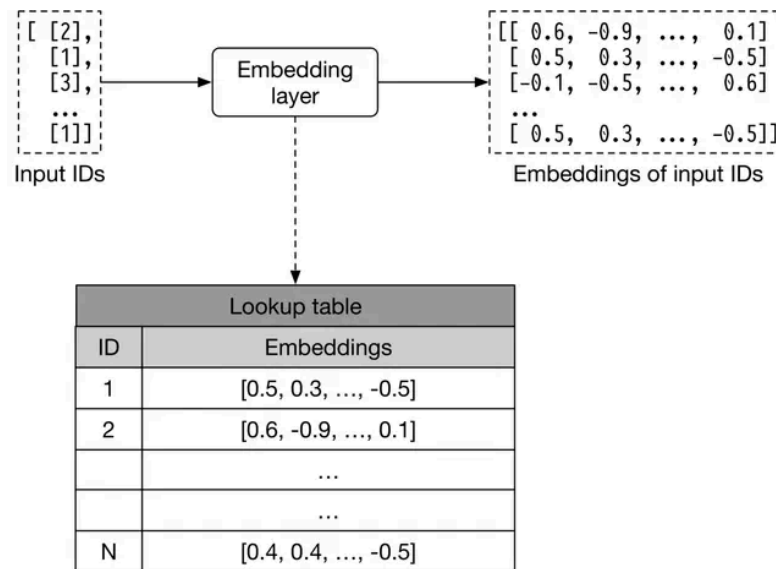


Figure 4.13: Embedding lookup method

Employing an embedding layer is a simple and effective solution to convert sparse features, such as IDs, into a fixed-size embedding. We will see more examples of its usage in later chapters.

Word2vec Word2vec [8] is a family of related models used to produce word embeddings. These models use a shallow neural network architecture and utilize the co-occurrences of words in a local context to learn word embeddings. In particular, the model learns to predict a center word from its surrounding words during the training phase. After the training phase, the model is capable of converting words into meaningful embeddings.

There are two main models based on word2vec: Continuous Bag of Words (CBOW) [9] and Skip-gram [10].

Figure 4.14 shows how CBOW works at a high level. If you are interested to learn about these models, refer to [8].

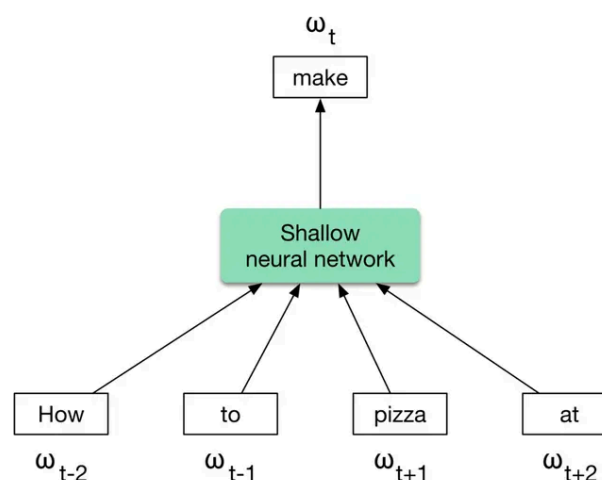


Figure 4.14: CBOW approach

Even though word2vec and embedding layers are simple and effective, recent architectures based upon Transformers have shown promising results.

Transformer-based models

These models consider the context of the words in a sentence when converting them into embeddings. As opposed to word2vec models, they produce different embeddings for the same word depending on the context.

Figure 4.15 shows a Transformer-based model which takes a sentence - a set of words as input, and produces an embedding for each word.

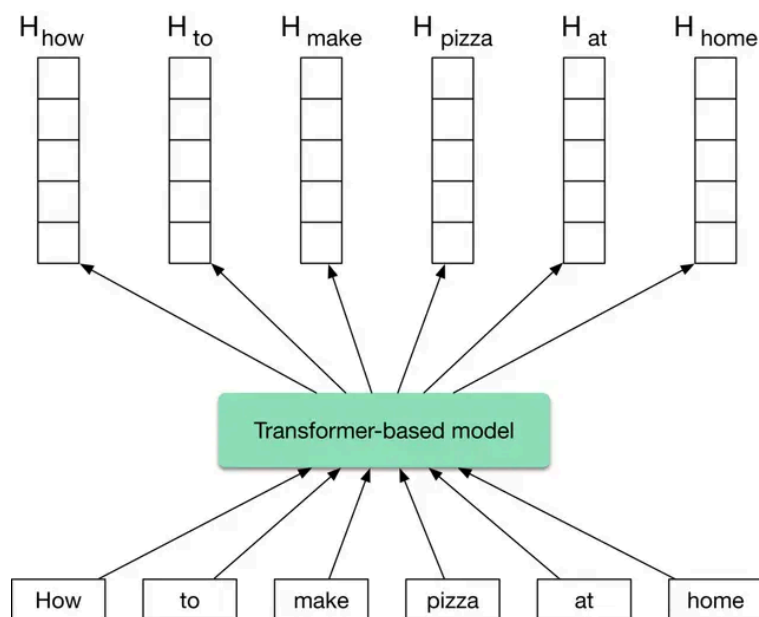


Figure 4.15: Transformer-based model's input-output

Transformers are very powerful at understanding the context and producing meaningful embeddings. Several models, such as BERT [11], GPT3 [12], and BLOOM [13], have demonstrated Transformers' potential to perform a wide variety of Natural Language Processing (NLP) tasks. In our case, we choose a Transformer-based architecture such as BERT as our text encoder.

In some interviews, the interviewer may want you to dive deeper into the details of the Transformer-based model. To learn more, refer to [14].

Video encoder

We have two architectural options for encoding videos: We have two architectural options for encoding videos:

- Video-level models
- Frame-level models

Video-level models process a whole video to create an embedding, as shown in Figure 4.16. The model architecture is usually based on 3D convolutions [15] or Transformers. Since the model processes the whole video, it is computationally expensive.

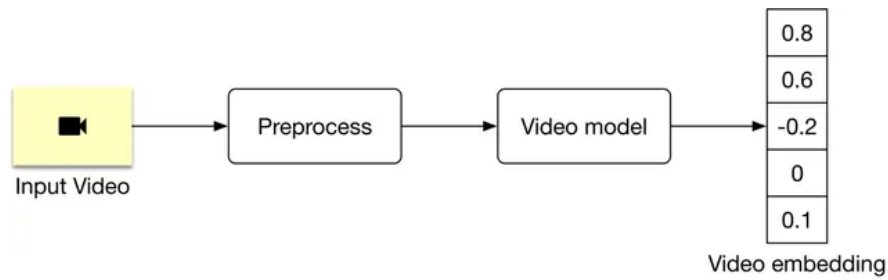


Figure 4.16: A video-level model

Frame-level models work differently. It is possible to extract the embedding from a video using a frame-level model by breaking it down into three steps:

- Preprocess a video and sample frames.
- Run the model on the sampled frames to create frame embeddings.
- Aggregate (e.g., average) frame embeddings to generate the video embedding.

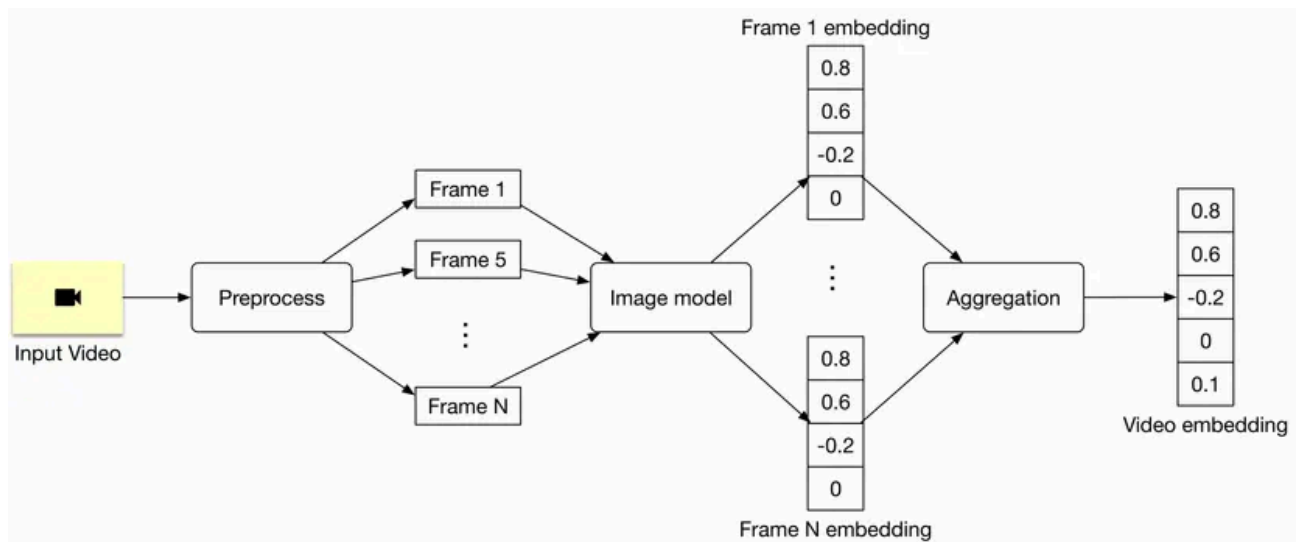


Figure 4.17: A frame-level model

Since this model works at the frame level, it is often faster and computationally less expensive. However, frame-level models are usually not able to understand the temporal aspects of the video, such as actions and motions. In practice, frame-level models are preferred in many cases where a temporal understanding of the video is not crucial. Here, we employ a frame-level model such as ViT [16] for two reasons:

- Improve the training and serving speed
- Reduce the number of computations

Model training

To train the text encoder and video encoder, we use a contrastive learning approach. If you are interested in learning more about this, see the "Model training" section in Chapter 2, Visual Search System.

An explanation of how to compute the loss during model training is shown in Figure 4.18.

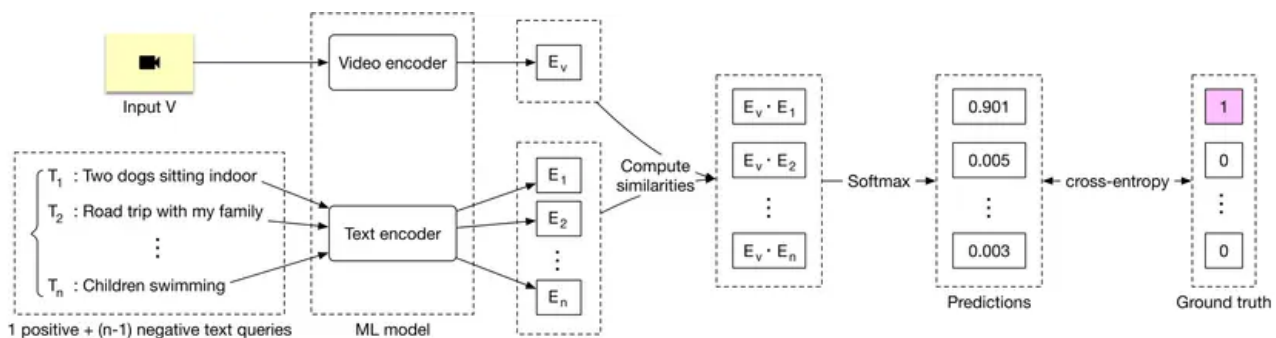


Figure 4.18: Loss computation

Evaluation

Offline metrics

Here are some offline metrics that are typically used in search systems. Let's examine which are the most relevant.

Precision@k and mAP

$$\text{precision@k} = \frac{\text{number of relevant items among the top } k \text{ items in the ranked list}}{k}$$

In the evaluation dataset, a given text query is associated with only one video. That means the numerator of the precision@k formula is at most 1. This leads to low precision@k values. For example, for a given text query, even if we rank its associated video at the top of the list, the precision@10 is only 0.1. Due to this limitation, precision metrics, such as precision@k and mAP, are not very helpful.

Recall@k. This measures the ratio between the number of relevant videos in the search results and the total number of relevant videos.

$$\text{recall@k} = \frac{\text{Number of relevant videos among the top } k \text{ videos}}{\text{Total number of relevant videos}}$$

As described earlier, the "total number of relevant videos" is always 1. With that, we can translate the recall@k formula to the following:

$\text{recall@k} = 1$ if the relevant video is among the top k videos, 0 otherwise

What are the pros and cons of this metric?

Pros

- It effectively measures a model's ability to find the associated video for a given text query.

Cons

- It depends on k . Choosing the right k could be challenging.
- When the relevant video is not among the k videos in the output list, $\text{recall}@k$ is always 0. For example, consider the case where model A ranks a relevant video at place 15, and model B ranks the same video at place 50. If we use $\text{recall}@10$ to measure the quality of these two models, both would have $\text{recall}@10=0$, even though model A is better than model B.

Mean Reciprocal Rank (MRR). This metric measures the quality of the model by averaging the rank of the first relevant item in each search result. The formula is:

$$MRR = \frac{1}{m} \sum_{i=1}^m \frac{1}{\text{rank}_i}$$

This metric addresses the shortcomings of $\text{recall}@k$ and can be used as our offline metric.

Online metrics

As part of online evaluation, companies track a wide variety of metrics. Let's take a look at some of the most important ones:

- Click-through rate (CTR)
- Video completion rate
- Total watch time of search results

CTR. This metric shows how often users click on retrieved videos. The main problem with CTR is that it does not track whether the clicked videos are relevant to the user. In spite of this issue, CTR is still a good metric to track because it shows how many people clicked on search results.

Video completion rate. A metric measuring how many videos appear in search results and are watched by users until the end. The problem with this metric is that a user may watch a video only partially, but still find it relevant. The video completion rate alone cannot reflect the relevance of search results.

Total watch time of search results. This metric tracks the total time users spent watching the videos returned by

the search results. Users tend to spend more time watching if the search results are relevant. This metric is a good indication of how relevant the search results are.

Serving

At serving time, the system displays a ranked list of videos relevant to a given text query. Figure 4.19 shows a simplified ML system design.

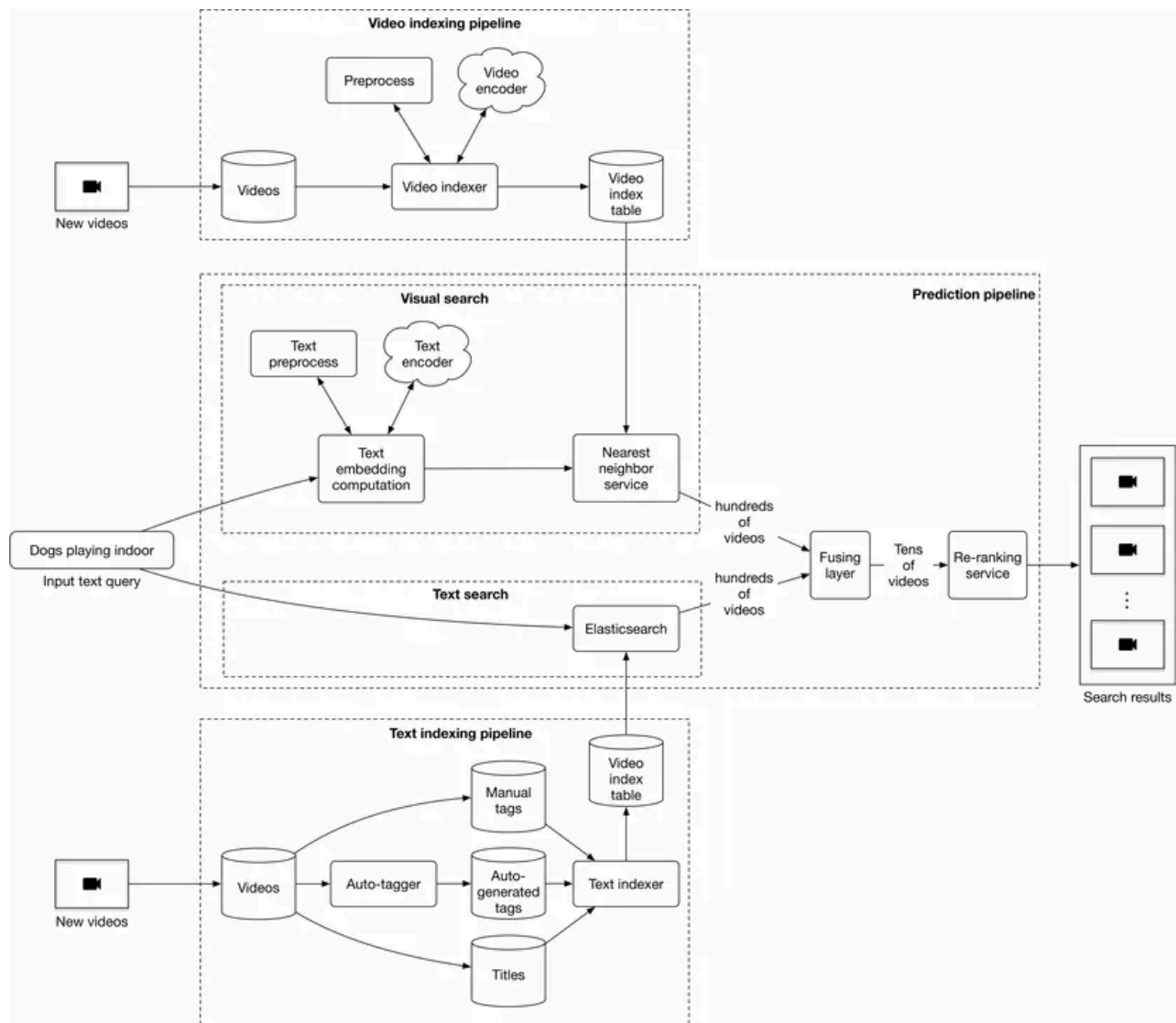


Figure 4.19: ML system design

Let's discuss each pipeline in more detail.

Prediction pipeline

This pipeline consists of:

- Visual search
- Text search
- Fusing layer
- Re-ranking service

Visual search. This component encodes the text query and uses the nearest neighbor service to find the most similar video embeddings to the text embedding. To accelerate the NN search, we use approximate nearest neighbor (ANN) algorithms, as described in Chapter 2, Visual Search System.

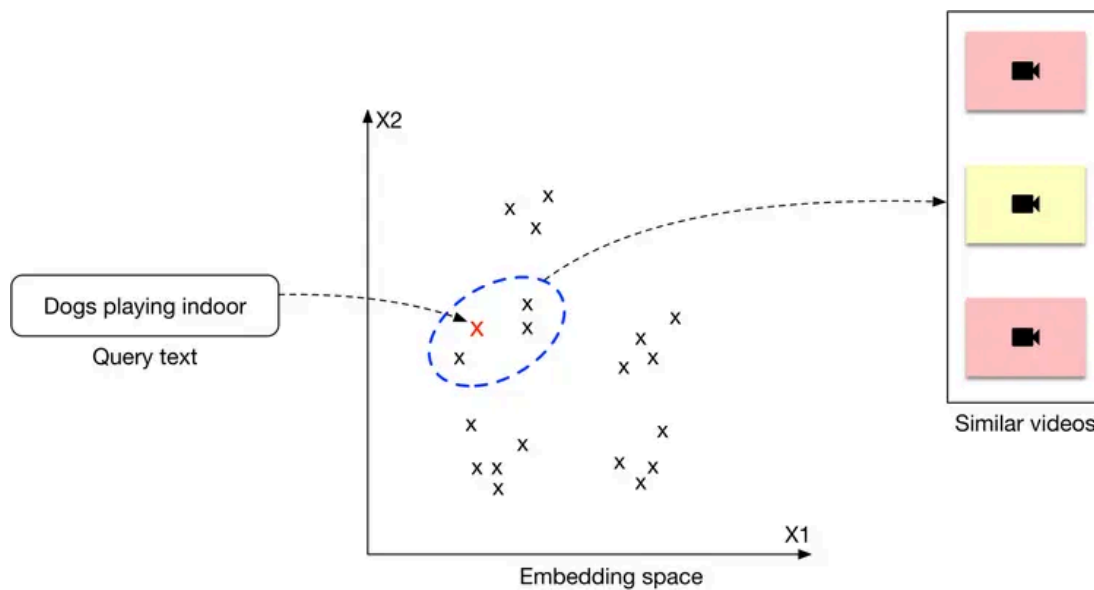


Figure 4.20: Retrieving the top 3 results for a given text query

Text search. Using Elasticsearch, this component finds videos with titles and tags that overlap the text query.

Fusing layer. This component takes two different lists of relevant videos from the previous step, and combines them into a new list of videos.

The fusing layer can be implemented in two ways, the easiest of which is to re-rank videos based on the weighted sum of their predicted relevance scores. A more complex approach is to adopt an additional model to re-rank the videos, which is more expensive because it requires model training. Additionally, it's slower at serving. As a result, we use the former approach.

Re-ranking service. This service modifies the ranked list of videos by incorporating business-level logic and policies.

Video indexing pipeline

A trained video encoder is used to compute video embeddings, which are then indexed. These indexed video embeddings are used by the nearest neighbor service.

Text indexing pipeline

This uses Elasticsearch for indexing titles, manual tags, and auto-generated tags.

Usually, when a user uploads a video, they provide tags to help better identify the video. But what if they do not manually enter tags? One option is to use a standalone model to generate tags. We name this component the auto-tagger and it is especially valuable in cases where a video has no manual tags. These tags may be noisier than manual tags, but they are still valuable.

Other Talking Points

Before concluding this chapter, it's important to note we have simplified the system design of the video search

system. In practice, it is much more complex. Some improvements may include:

- Use a multi-stage design (candidate generation + ranking).
- Use more video features such as video length, video popularity, etc.
- Instead of relying on annotated data, use interactions (e.g., clicks, likes, etc.) to construct and label data. This allows us to continuously train the model.
- Use an ML model to find titles and tags which are semantically similar to the text query. This model can be combined with Elasticsearch to improve search quality.

If there's time left at the end of the interview, here are some additional talking points:

- An important topic in search systems is query understanding, such as spelling correction, query category identification, and entity recognition. How to build a query understanding component? [17].
- How to build a multi-modal system that processes speech and audio to improve search results [18].
- How to extend this work to support other languages [19].
- Near-duplicate videos in the final output may negatively impact user experience. How to detect near-duplicate videos so we can remove them before displaying the results [20] ?
- Text queries can be divided into head, torso, and tail queries. What are the different approaches commonly used in each case [21]?
- How to consider popularity and freshness when producing the output list [22]?
- How real-world search systems work [23][24][25].

References

1. Elasticsearch. https://www.tutorialspoint.com/elasticsearch/elasticsearch_query_dsl.htm.
2. Preprocessing text data. <https://huggingface.co/docs/transformers/v4.42.0/preprocessing>.
3. NFKD normalization. <https://unicode.org/reports/tr15/>.
4. What is Tokenization summary. https://huggingface.co/docs/transformers/tokenizer_summary.
5. Hash collision. https://en.wikipedia.org/wiki/Hash_collision.
6. Deep learning for NLP. http://cs224d.stanford.edu/lecture_notes/notes1.pdf.
7. TF-IDF. <https://en.wikipedia.org/wiki/Tf%E2%80%93idf>.
8. Word2Vec models. <https://www.tensorflow.org/tutorials/text/word2vec>.
9. Continuous bag of words. <https://www.kdnuggets.com/2018/04/implementing-deep-learning-methods-feature-engineering-text-data-cbow.html>.
10. Skip-gram model. <http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/>.
11. BERT model. <https://arxiv.org/pdf/1810.04805.pdf>.
12. GPT3 model. <https://arxiv.org/pdf/2005.14165.pdf>.
13. BLOOM model. <https://bigscience.huggingface.co/blog/bloom>.
14. Transformer implementation from scratch. <https://peterbloem.nl/blog/transformers>.
15. 3D convolutions. <https://www.kaggle.com/code/shivamb/3d-convolutions-understanding-use-case/notebook>.
16. Vision Transformer. <https://arxiv.org/pdf/2010.11929.pdf>.
17. Query understanding for search engines. <https://www.linkedin.com/pulse/ai-query-understanding-daniel-tunkel-ang/>.
18. Multimodal video representation learning. <https://arxiv.org/pdf/2012.04124.pdf>.
19. Multilingual language models. <https://arxiv.org/pdf/2107.00676.pdf>.

20. Near-duplicate video detection. <https://arxiv.org/pdf/2005.07356.pdf>.
21. Generalizable search relevance. <https://livebook.manning.com/book/ai-powered-search/chapter-10/v-10/20>.
22. Freshness in search and recommendation systems. <https://developers.google.com/machine-learning/recommendation/dnn/re-ranking>.
23. Semantic product search by Amazon. <https://arxiv.org/pdf/1907.00937.pdf>.
24. Ranking relevance in Yahoo search. <https://www.kdd.org/kdd2016/papers/files/adf0361-yinA.pdf>.
25. Semantic product search in E-Commerce. <https://arxiv.org/pdf/2008.08180.pdf>.

05

Harmful Content Detection

Many social media platforms such as Facebook [1], LinkedIn [2], and Twitter [3] have standard guidelines to enforce integrity and make their platforms safe for users. These guidelines prohibit certain user behaviors, activities, and content that are harmful to the community. It is essential to have technologies and resources in place to identify harmful content and bad actors. We can divide the focus of integrity enforcement into two categories:

- Harmful content: Posts that contain violence, nudity, self-harm, hate speech, etc.
- Bad acts/bad actors: Fake accounts, spam, phishing, organized unethical activities, and other unsafe behaviors.

In this chapter, we focus on detecting posts that might contain harmful content. In particular, we design a system that proactively monitors new posts, detects harmful content, and removes or demotes them if the content violates the platform's guidelines. To understand how companies build a harmful content detection system in practice, refer to [4] [5] [6].

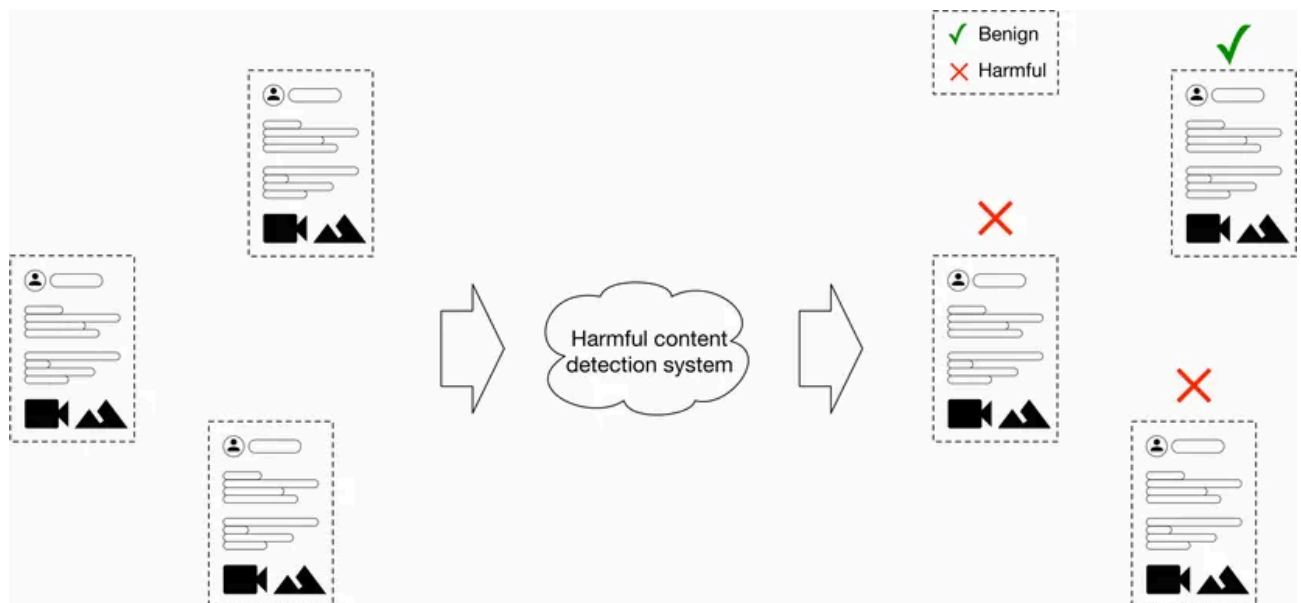


Figure 5.1: Harmful content detection system

Clarifying Requirements

Here is a typical interaction between a candidate and an interviewer.

Candidate: Does the system detect both harmful content and bad actors?

Interviewer: Both are equally important. For simplicity, let's focus on detecting harmful content only.

Candidate: Should a post only contain text, or are images and videos allowed?

Interviewer: The content of a post can be text, image, video, or any combination of these.

Candidate: What languages are supported? Is it English only?

Interviewer: The system should detect harmful content in various languages. For simplicity, assume we can use a pre-trained multilingual model to embed the textual content.

Candidate: Which specific categories of harmful content are we looking to identify? I can think of violence, nudity, hate speech, misinformation, etc. Are there other harmful categories to consider?

Interviewer: Great, you brought up the major ones. Misinformation is more complex and controversial. For simplicity, let's not focus on misinformation.

Candidate: Are there any human annotators available to label posts manually?

Interviewer: The platform receives more than 500 million posts each day. Asking humans to label all of them would be very expensive and time-consuming. However, you can assume human annotation is available to label a limited number of posts, say 10,000 per day.

Candidate: Allowing users to report harmful content is beneficial for understanding where the system is failing. Can I assume the system has that feature?

Interviewer: Good point. Yes, users can report harmful posts.

Candidate: Should we explain why a post is deemed harmful and removed?

Interviewer: Yes. Explaining to users why we remove a post is essential. It helps users to ensure they align their future posts with the guidelines.

Candidate: What is the system's latency requirement? Do we need a real-time prediction, i.e., the system detects harmful content immediately and blocks it, or can we rely on batch prediction, i.e., detecting harmful content offline hourly or daily?

Interviewer: This is a very important question. What are your thoughts?

Candidate: In my opinion, the requirements for different harmful content might vary. For example, violent content may require real-time solutions, while for others, late detection may work.

Interviewer: Those are fair assumptions.

So, let's summarize the problem statement. We will design a harmful content detection system, which identifies harmful posts, then deletes or demotes them and informs the user why the post was identified as harmful. A post's content can be text, image, video, or any combination of these, and the content can be in different languages. Users can report harmful posts.

Frame the Problem as an ML Task

Defining the ML objective

We define our ML objective as accurately predicting harmful posts. The reason is that if we can accurately detect harmful posts, we can remove or demote them, leading to a safer platform.

Specifying the system's input and output

The system receives a post as input, and outputs the probability that the post is harmful.

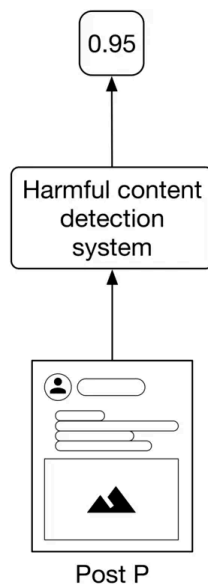


Figure 5.2: A harmful content detection system's input-output

Let's dive deeper into the input post. As shown in Figure 5.3, a post can be heterogeneous and potentially multimodal.

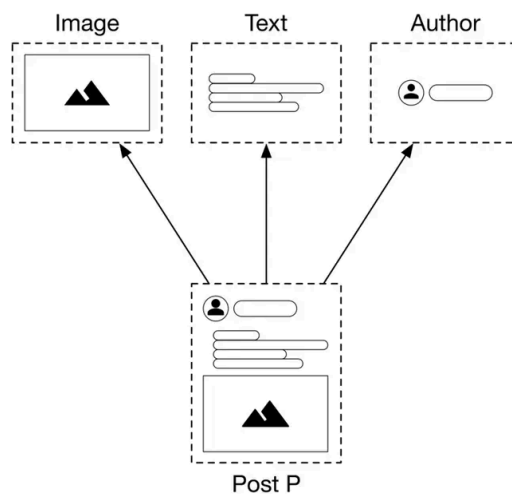


Figure 5.3: Heterogeneous post data

To make accurate predictions, the system should consider all modalities. Let's discuss two commonly used fusing methods to combine heterogeneous data: late fusion and early fusion.

Late fusion

With late fusion, ML models process different modalities independently, then combine their predictions to make a

final prediction. The figure below illustrates how late fusion works.

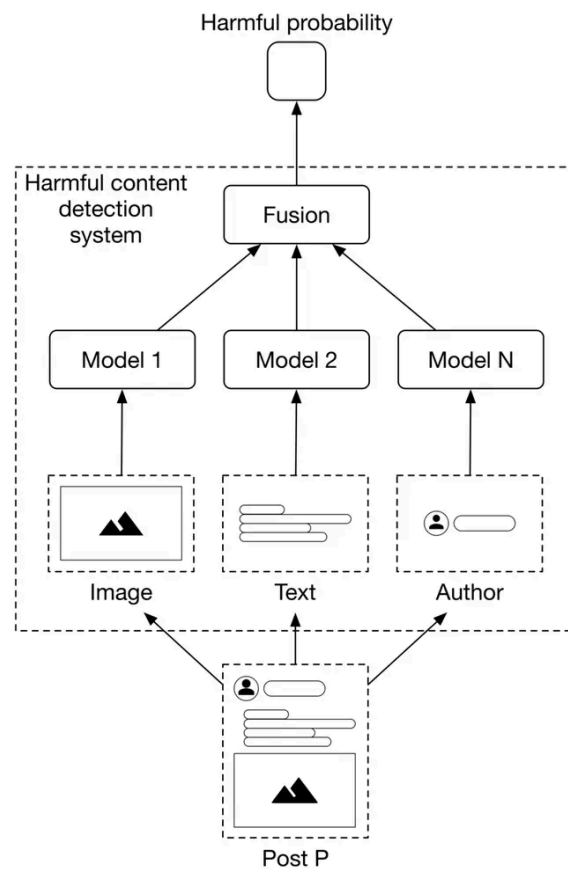


Figure 5.4: Late fusion

The advantage of late fusion is that we can train, evaluate, and improve each model independently.

However, late fusion has two major disadvantages. First, to train these individual models, we need to have separate training data for each modality, which can be time-consuming and expensive.

Second, the combination of the modalities might be harmful, even if each is benign in isolation. This is frequently the case with memes that combine images and text. In these cases, late fusion fails to predict whether the content is harmful. This is because each modality is benign, so the models predict benignity when processing each modality. The fusion layer output is benign because the output of each separate modality is benign. But this is incorrect, as the combination of modalities can be harmful.

Early fusion

With early fusion, the modalities are combined first, and then the model makes a prediction. Figure 5.5 illustrates how early fusion works.

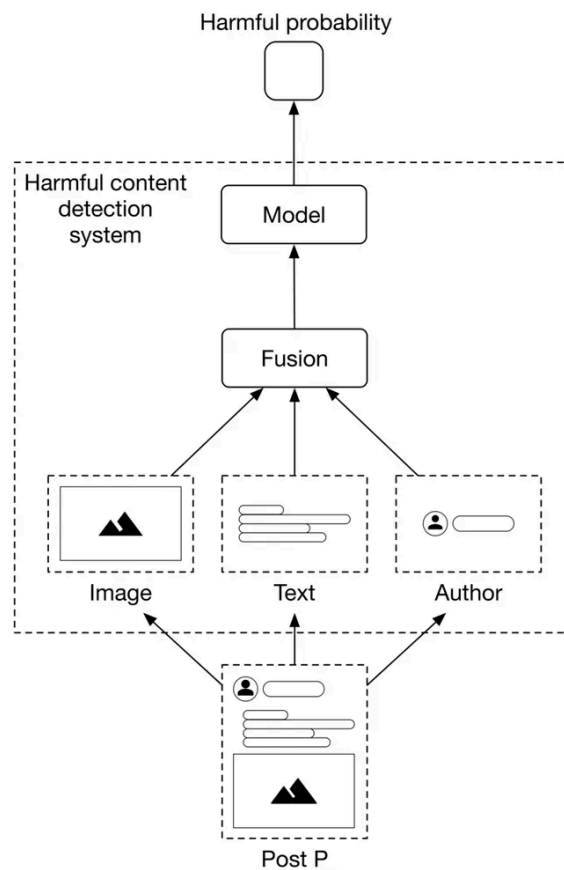


Figure 5.5: Early fusion

Early fusion has two major advantages. First, it is unnecessary to collect training data separately for each modality. Since there is a single model to train, we only need to collect training data for that model. Second, the model considers all the modalities, so if each modality is benign, but their combination is harmful, then the model can potentially capture this in the unified feature vector.

However, learning this task is more difficult for the model due to the complex relationships between modalities. In the absence of sufficient training data, it is challenging for the model to learn complex relationships and make good predictions.

Which fusion method should we use?

The early fusion method is used because it allows us to capture posts that may be harmful overall, even if each modality is benign on its own. Additionally, with around 500 million posts being published every day, the model has enough data to learn the task.

Choosing the right ML category

In this section, we examine the following ML category options:

- Single binary classifier
- One binary classifier per harmful class
- Multi-label classifier
- Multi-task classifier

Single binary classifier

In this option, a model takes the fused features as input and predicts the probability of the post being harmful (Figure 5.6). Since the output is a binary outcome, the model is a binary classifier.

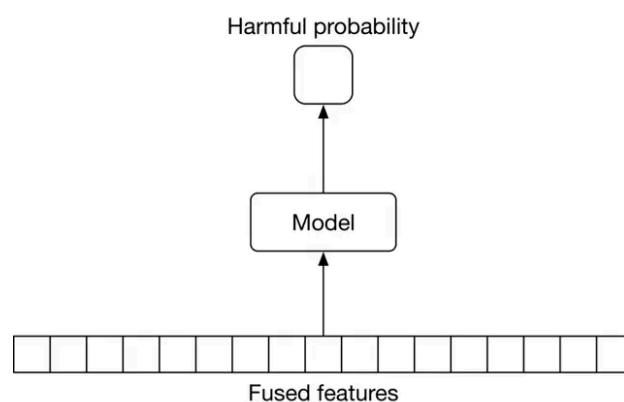


Figure 5.6: A single binary classifier

The shortcoming of this option is that it's difficult to determine which class of harm, such as violence, a post belongs to. This limitation causes two main issues:

- It is not easy to inform users why we take down a post as the system only outputs a binary value indicating whether the post as a whole is harmful or not. We have no clue which particular class of harm the post belongs to.
- It is not easy to identify harmful classes in which the system doesn't perform well, meaning we cannot improve the system for underperforming classes.

Since it is essential to explain why a post is removed, a single binary classifier is not a good option.

One binary classifier per harmful class

In this option, we adopt one binary classifier for each harmful class. As Figure 5.7 shows, each model determines if a post belongs to a specific harmful class or not. Each model takes fused features as input and predicts the probability of the post being classified as a harmful class.

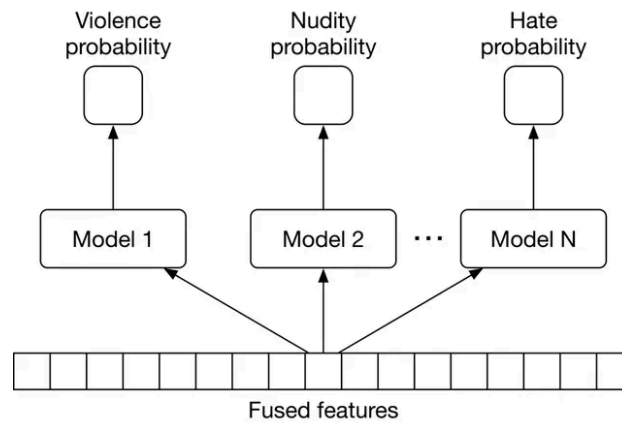


Figure 5.7: One binary classifier per harmful class

The advantage of this option is that we can explain to users why a post was taken down. In addition, we can monitor different models and improve them independently.

However, this option has one major drawback. Since we have multiple models, they must be trained and maintained separately. Training these models separately is time-consuming and expensive.

A multi-label classifier

In multi-label classification, the data point we want to classify may belong to an arbitrary number of classes. In this option, a single model is used as a multi-label classifier. As Figure 5.8 shows, the input to the model is the fused features, and the model predicts probabilities for each harmful class.

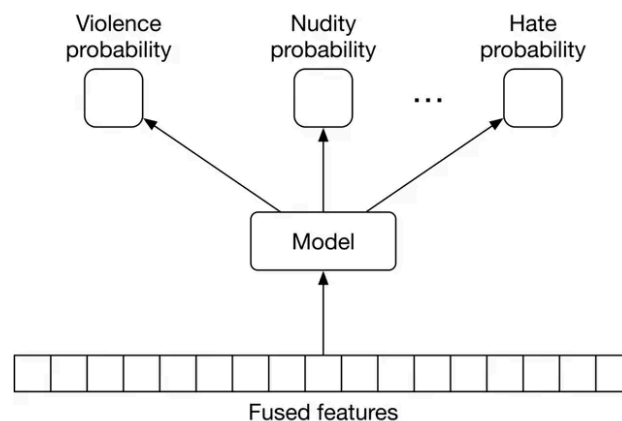


Figure 5.8: A multi-label classifier

By using a shared model for all harmful classes, training and maintaining the model is less costly. If you would like to learn more about this method, refer to the approach called WPIE [7].

However, predicting the probabilities of each harmful class using a shared model isn't ideal, as the input features may need to be transformed differently.

Multi-task classifier

Multi-task learning refers to the process in which a model learns multiple tasks simultaneously. This allows the model to learn similarities between tasks. By doing so, we avoid unnecessary computations when a certain input transformation is beneficial for multiple tasks.

In our case, we treat different classes of harm, such as violence and nudity, as different tasks and use a multi-task classification model to learn each task. As Figure 5.9 shows, multi-task classification has two stages: shared layers and task-specific layers.

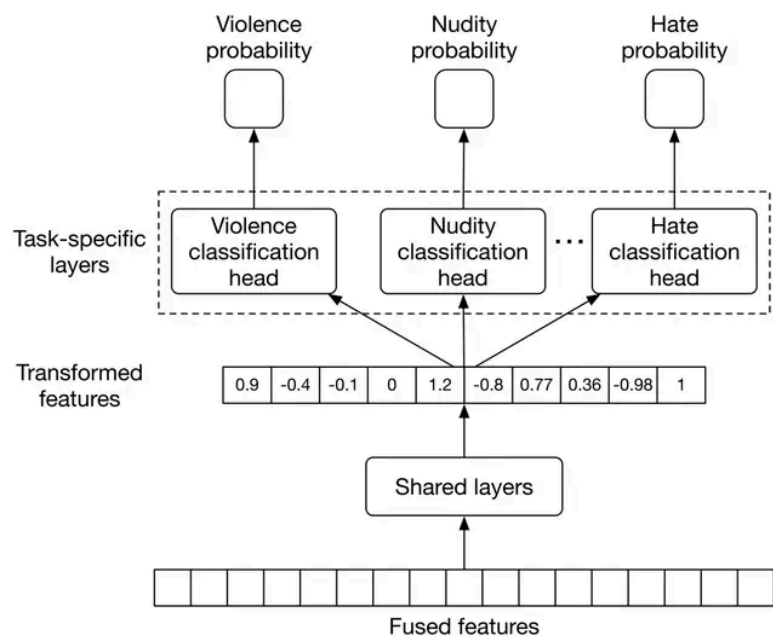


Figure 5.9: Multi-task classification overview

Shared layers A shared layer, as shown in Figure 5.10, is a set of hidden layers that transform input features into new ones. These newly transformed features are used to make predictions for each of the harmful classes.

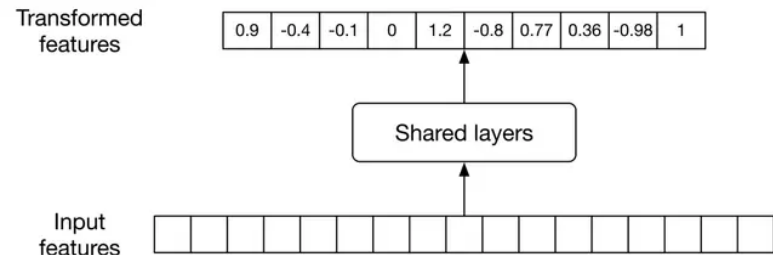


Figure 5.10: Shared layers

Task-specific layers Task-specific layers are a set of independent ML layers (also called classification heads).

Each classification head transforms features in a way that is optimal for predicting a specific harm probability.

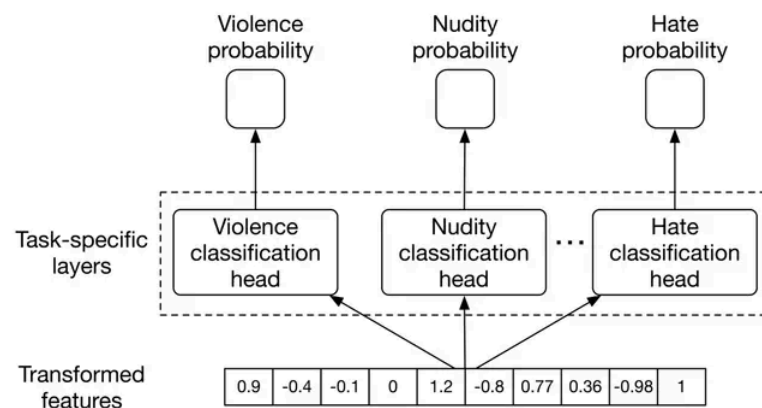


Figure 5.11: Task-specific layers

Multi-task classification has three advantages. First, it is not expensive to train or maintain since we use a single model. Second, the shared layers transform the features in a way that is beneficial for each task. This prevents redundant computations and makes multi-task classification efficient. Lastly, the training data for each task contributes to the learning of other tasks. This is especially helpful when limited data is available for a particular task.

Because of these advantages, we employ a multi-task classification method. Figure 5.12 shows how we frame the problem.

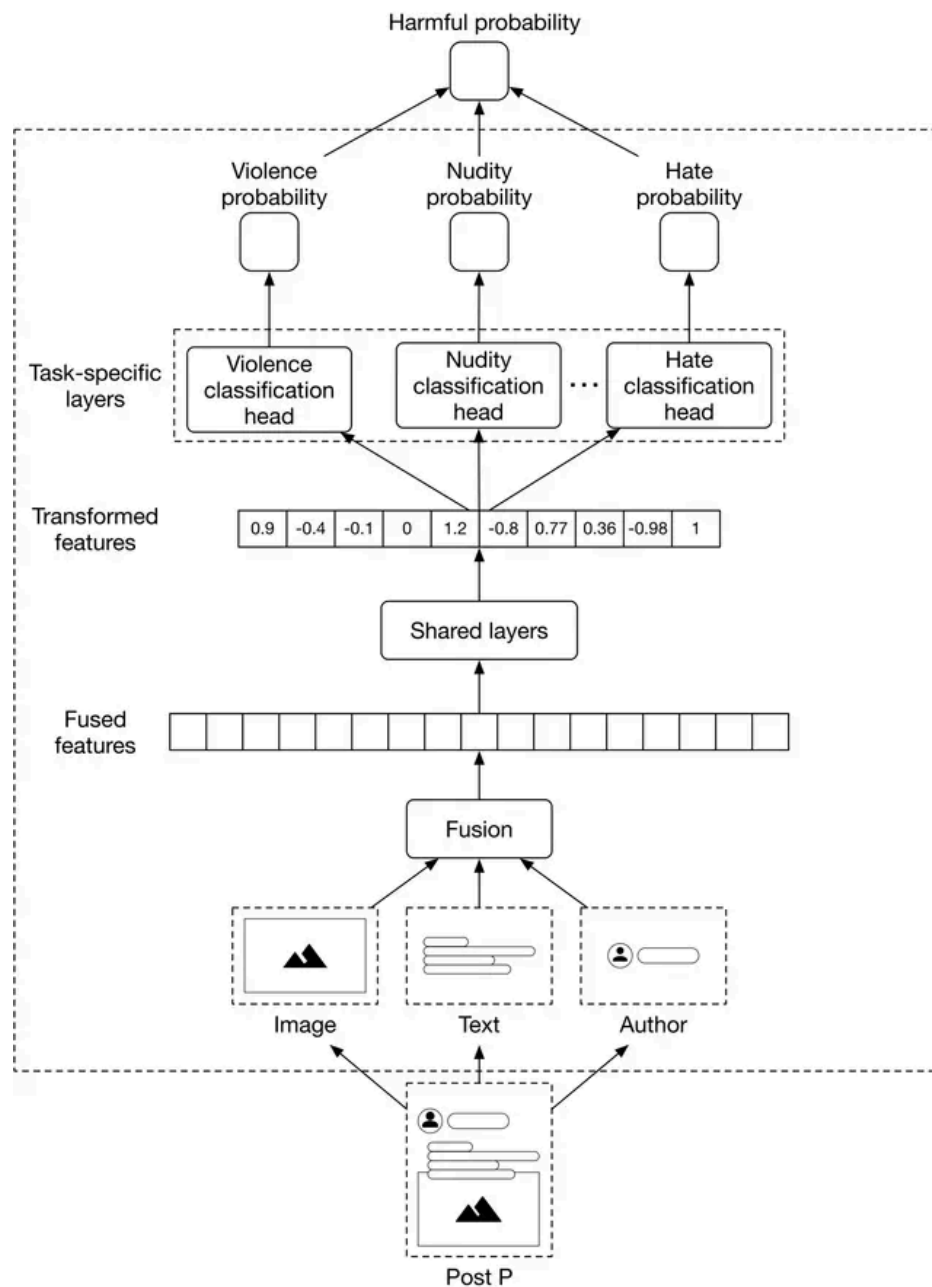


Figure 5.12: Frame the problem as an ML task

Data Preparation

Data engineering

We have the following data available:

- Users
- Posts
- User-post interactions

Users

A user data schema is shown below.

ID	Username	Age	Gender	City	Country	Email
----	----------	-----	--------	------	---------	-------

Table 5.1: User data schema

Posts

Post data contain fields such as author, time of upload, etc. Table 5.2 shows some of the most important attributes. In practice, we typically have hundreds of attributes associated with each post.

Post ID	Author ID	On-device	Timestamp	Textual content	Images or videos	Links
1	1	73.93.220.240	1658469431	Today, I am starting my diet.	http://cdn.mysite.com/u1.jpg	-
2	11	89.42.110.250	1658471428	The video amazed me! Please donate	http://cdn.mysite.com/t3.mp4	gofundme.com/f/3u1njd32
3	4	39.55.180.020	1658489233	What is a good restaurant in the Bay area?	http://cdn.mysite.com/t5.jpg	-

Table 5.2: Post data

User-post interactions

User-post interaction data primarily contain users' reactions to posts, such as likes, comments, saves, shares, etc.

Users can also report a post as harmful or request an appeal. Table 5.3 shows what the data might look like.

User ID	Post ID	Interaction type	Interaction value	Timestamp
11	6	Impression	-	1658450539
4	20	Like	-	1658451341
11	7	Comment	This is disgusting	1658451365
4	20	Share	-	1658435948
11	7	Report	violence	1658451849

Table 5.3: User-post interaction data

Feature engineering

In the "Frame the problem as an ML task" section, we framed the problem as a multitask classification task where the input is the post. In this section, we explore predictive features that can be derived from a post. A post might comprise the following elements:

- Textual content
- Image or video
- User reactions to the post
- Author
- Contextual information

Let's look at each element.

Textual content

The textual content of a post can be used to determine whether a post is harmful or not. As described in Chapter 4 YouTube Video Search, text data is usually prepared in two steps:

- Text preprocessing (e.g., normalization, tokenization)
- Vectorization: convert the preprocessed text into a meaningful feature vector Let's focus on vectorization since it is unique to this chapter. To vectorize the text and extract a feature vector, statistical or ML-based methods can be used. Statistical methods such as BoW or TF-IDF are easy to implement and fast to compute. However, they cannot encode the semantics of the text. For our system, understanding the semantics of the textual content is important for determining harm, so we adopt the ML-based method. To convert text into a feature vector, we use a pre-trained Transformer-based language model such as BERT [8]. However, the original BERT has two issues:
 - Producing the text embedding takes a long time due to the large size of the model. Because this is a slow process, using it for online predictions is not ideal.
 - BERT was trained on English-only data. Thus, it does not produce meaningful embeddings for texts in other languages.

DistilmBERT [9], a more efficient variant of BERT, addresses those two issues. If two sentences have the same meaning but are in two different languages, their embeddings are very similar. If you are interested to learn more about multilingual language models, refer to [10].

Image or video

You can usually find out what a post is about by looking at the image or video within it. The following two steps are commonly used to prepare unstructured data such as images or videos.

- **Preprocessing:** decode, resize, and normalize the data.
- **Feature extraction:** after preprocessing, we use a pre-trained model to convert unstructured data to a feature vector. This allows us to represent an image or video by a feature vector. For images, a pre-trained image model such as CLIP's visual encoder [11] or SimCLR [12] are viable options. For videos, pre-trained models like VideoMoCo [13] might work well.

User reactions to the post

It is also possible to determine whether a post is harmful based on user reactions, especially when the content is ambiguous. As shown in Figure 5.13, with more comments, it becomes increasingly evident the post contains content related to self-harm.

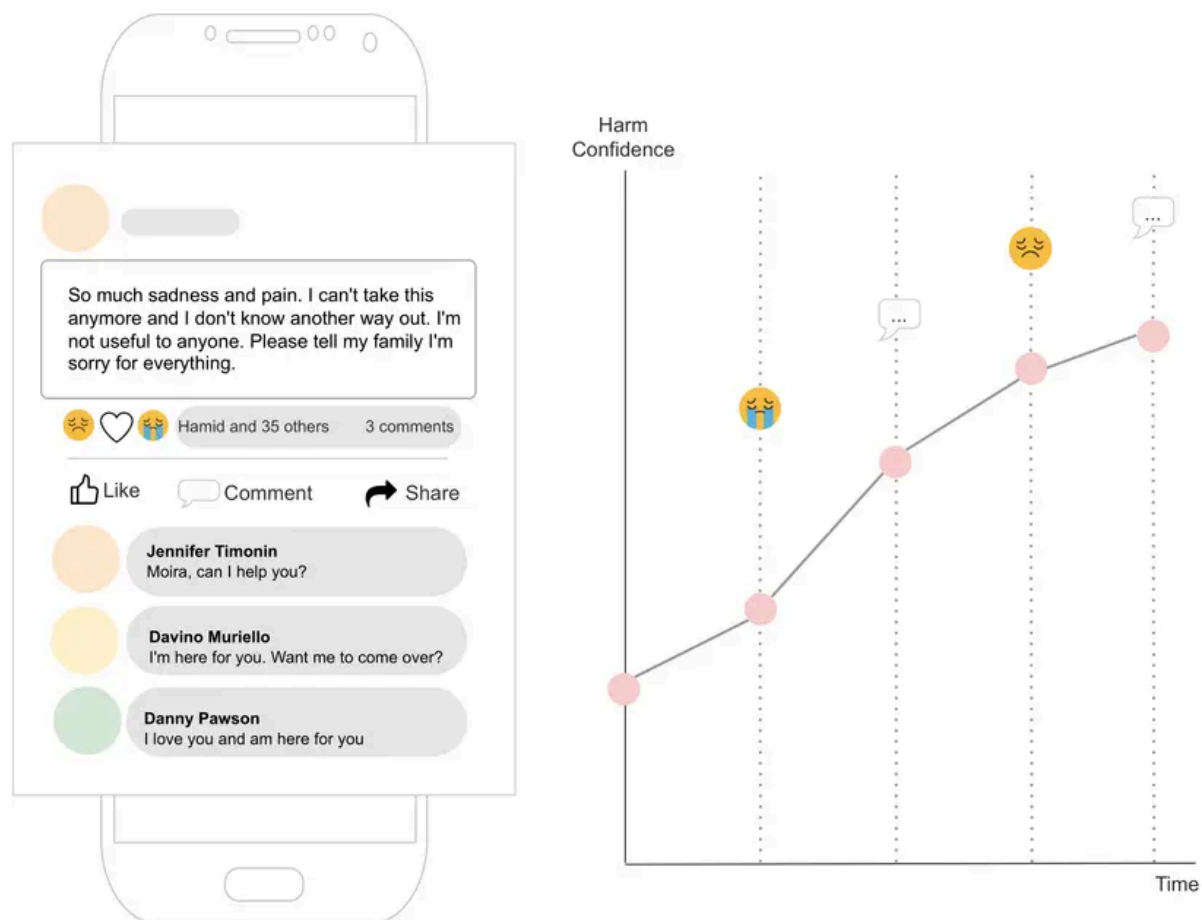


Figure 5.13: Post with self-harm concern

Since user reactions are crucial in determining harmful content, let's examine some of the features we can engineer based on them.

The number of likes, shares, comments, and reports: We usually scale these numerical values to speed up convergence during model training.

Comments: As demonstrated in Figure 5.13, comments can help us identify harmful content. For feature preparation, we convert comments into numerical representations by doing the following:

- Use the same pre-trained model we employed earlier to obtain the embedding of each comment.
- Aggregate (e.g., average) the embeddings to obtain a final embedding.

A summary of the features we have described so far can be found in Figure 5.14.

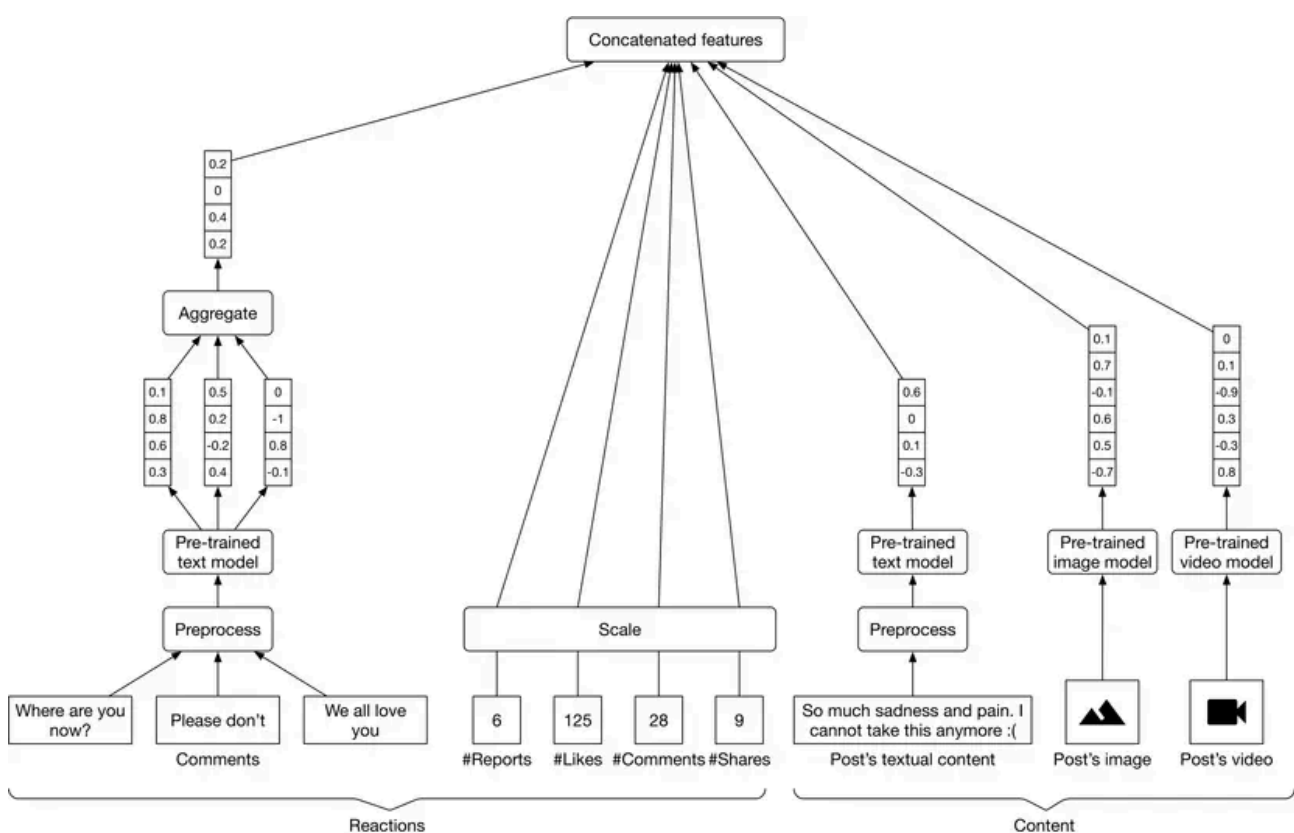


Figure 5.14: Feature engineering for reactions and content

Author features

The author's past interactions can be used to determine if the post is harmful or not. Let's engineer features related to the post author.

Author's violation history

- **Number of violations:** This is a numerical value representing the number of times the author violated the guidelines in the past.
- **Total user reports:** A numerical value representing the number of times users reported the author's posts.

- **Profane words rate:** This is a numerical value representing the rate of profane words used in the author's previous posts and comments. A predefined list of profane words is used to determine whether a word is profane.

Author's demographics

- **Age:** A user's age is one of the most important predictive features.
- **Gender:** This categorical feature represents the user's gender. We use one-hot encoding to represent gender.
- **City and country:** Both the city and country take many distinct values. To represent the features, we use an embedding layer to convert city and country into feature vectors. Note that one-hot encoding is not an efficient method to represent the city and country because their representations would be long and sparse.

Account information

- **Number of followers and followings**
- **Account age:** This is a numerical value representing the age of the author's account. This is a predictive feature as accounts with a lower age are more likely to be spam or to violate integrity.

Contextual information

- **Time of day:** This is the time of day when the author made a post. We bucketize this into multiple categories, such as morning, noon, afternoon, evening or night. We use one-hot encoding to represent this feature.
- **Device:** The device the author uses, such as a smartphone or desktop computer. One-hot encoding is used to represent this feature.

Figure 5.15 summarizes some of the most important features of the harmful content detection system.

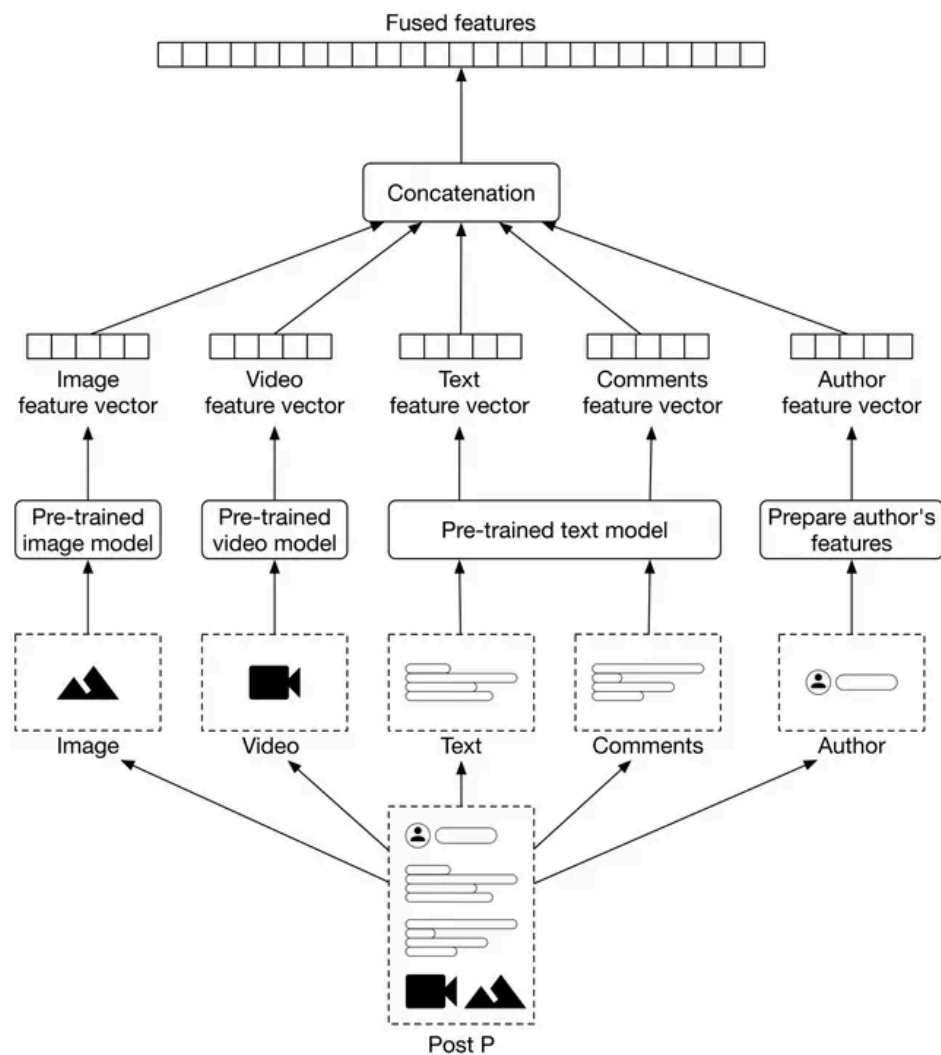


Figure 5.15: Summary of feature engineering

Model Development

Model selection

A neural network is the most common model used for multi-task learning. In developing our model, we employ neural networks.

When choosing a neural network, what factors should be considered? It is necessary to determine the neural network's architectural design and the optimal hyperparameter selection, such as its hidden layers, activation function, learning rate, etc. The optimal choice of hyperparameters is usually determined by hyperparameter tuning. Let's briefly go over it.

Hyperparameter tuning is the process of finding the best values for hyperparameters in order to produce the best performance for a model. To tune hyperparameters, grid search is commonly used. The procedure involves training a new model for each combination of hyperparameter values, evaluating each model, then selecting the hyperparameters that lead to the best model. If you are interested in learning more about hyperparameter tuning, refer to [14].

Model training

Constructing the dataset

To train the multi-task classification model, we first need to construct the dataset. The dataset comprises model inputs (features) and outputs (labels) that the model is expected to predict. To construct inputs, we process posts offline in batches and compute fused features as described earlier. These features can be stored in a feature store for future training. In order to create labels for each input, we have two options:

- Hand labeling
- Natural labeling

With hand labeling, human contractors label posts manually. This option produces accurate labels, but it is expensive and time-consuming. With natural labeling, we rely on user reports to label posts automatically. While this option results in noisier labels, labels are produced more quickly. For the evaluation dataset, we use hand labeling to prioritize the accuracy of labels, and for the training dataset, we use natural labeling to prioritize labeling speed. A data point from the constructed dataset is shown in Figure 5.16.

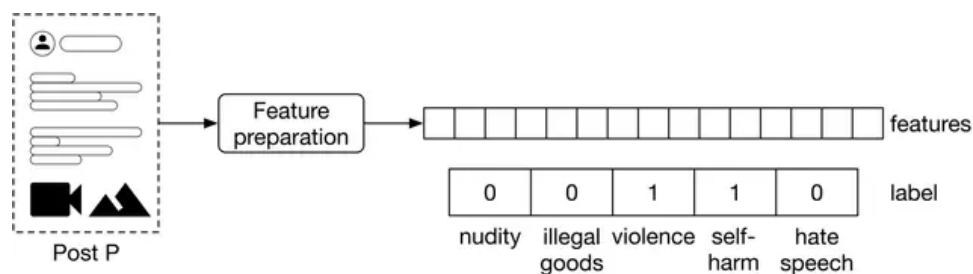


Figure 5.16: A constructed data point

Choosing the loss function

Training a multi-task neural network is very similar to how we typically train neural network models. A forward propagation performs computations to make a prediction, a loss function measures the correctness of the prediction, and a backward propagation optimizes the model's parameters to reduce the loss in the next iteration. Let's examine the loss function. In multi-task training, each task is assigned a loss function based on its ML category. In our case, each task is framed as a binary classification, so we adopt a standard binary classification loss such as cross-entropy for each task. The overall loss is computed by combining task-specific losses, as shown in Figure 5.17.

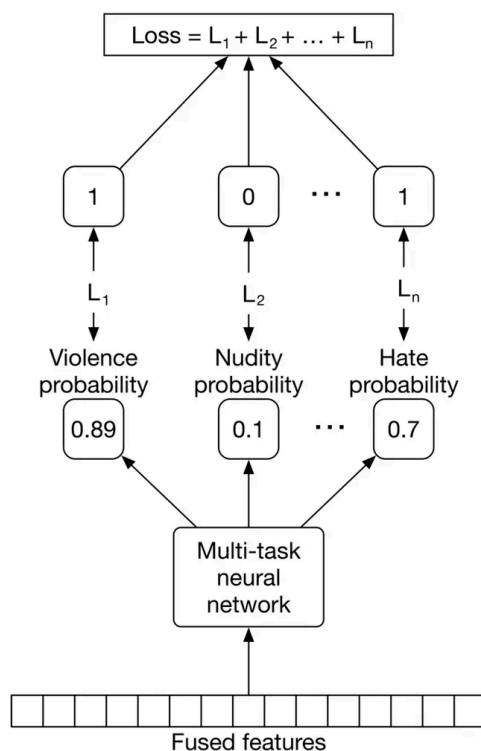


Figure 5.17: Model training

A common challenge in training multimodal systems is overfitting [15]. For example, when the learning speed varies across different modalities, one modality (e.g., image) can dominate the learning process. Two techniques to address this issue are gradient blending and focal loss. If you are interested in learning more about these

techniques, refer to [16] [17].

Evaluation

Offline metrics

To evaluate the performance of a binary classification model, offline metrics such as precision, recall, and f1 score are commonly used. However, precision or recall alone is not sufficient to understand the overall performance. For example, a model with high precision might have a very low recall. The precision-recall (PR) curve and receiver operating characteristic (ROC) curve address those limitations. Let's explore each one.

PR-curve. PR curve shows the trade-off between precision and recall of the model. As Figure 5.18 shows, we obtain a PR curve by plotting the precision of the model using different probability thresholds, ranging from 0 to 1. To summarize the trade-offs between precision and recall, PR-AUC (the area under the precision-recall curve) calculates the area beneath the PR curve. In general, a high PR-AUC indicates a more accurate model.

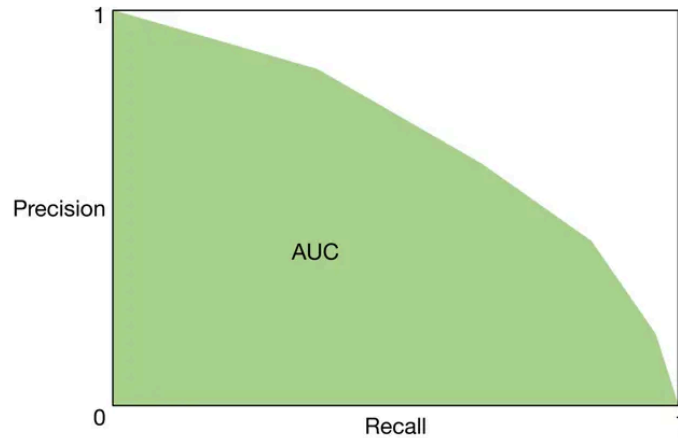


Figure 5.18: PR curve

ROC curve. The ROC curve shows the trade-offs between the true positive rate (recall) and the false positive rate. Similar to the PR curve, ROC-AUC summarizes the model's performance by calculating the area under the ROC curve.

ROC and PR curves are two different ways to summarize the performance of a classification model. To learn about the differences between the PR curve and the ROC curve, read [18].

In our case, we use both ROC-AUC and PR-AUC as our offline metrics.

Online metrics

Let's explore a few important metrics to capture how safe the platform is.

Prevalence. This metric measures the ratio of harmful posts which we didn't prevent and all posts on the platform.

$$\text{Prevalence} = \frac{\text{Number of harmful posts we didn't prevent}}{\text{Total number of posts on the platform}}$$

The shortcoming of this metric is that it treats harmful posts equally. For example, one harmful post with 100 K views or impressions is more harmful than two posts with 10 views each.

Harmful impressions. We prefer this metric over prevalence. The reason is that the number of harmful posts on the platform does not show how many people were affected by those posts, whereas the number of harmful impressions does capture this information.

Valid appeals. Percentage of posts that were deemed harmful, but appealed and reversed.

$$\text{Appeals} = \frac{\text{Number of reversed appeals}}{\text{Number of harmful posts detected by the system}}$$

Proactive rate. Percentage of harmful posts found and deleted by the system before users report it.

$$\text{Proactive rate} = \frac{\text{Number of harmful posts detected by the system}}{\text{Number of harmful posts detected by the system} + \text{reported by users}}$$

User reports per harmful class. This metric measures the system's performance by looking into user reports for each harmful class.

Serving

Figure 5.19 shows the high-level ML system design. Let's take a closer look at each component.

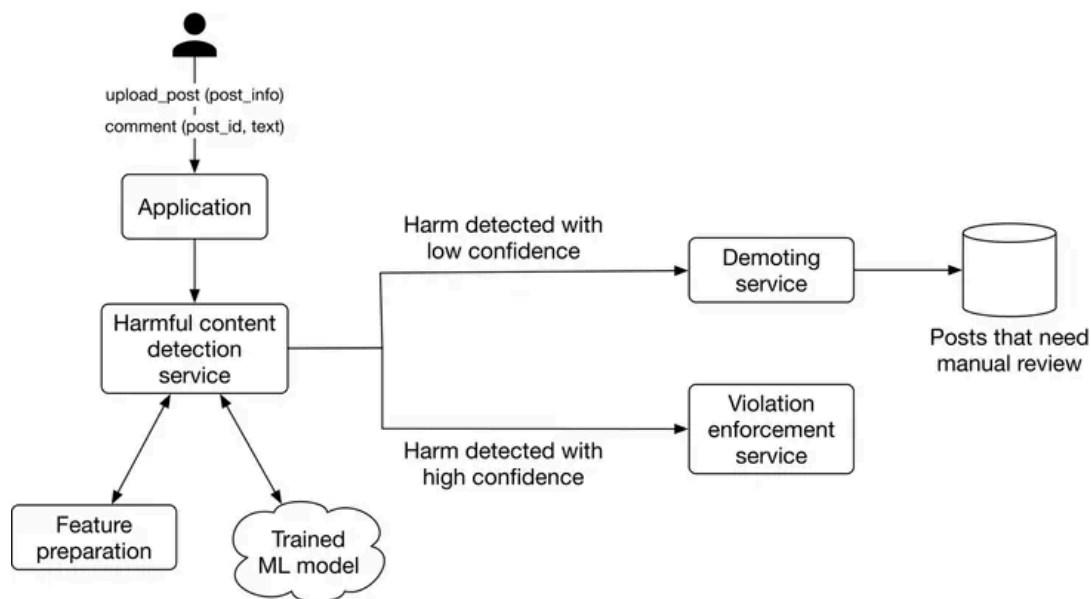


Figure 5.19: ML system design

Harmful content detection service

Given a new post, this service predicts the probability of harm. According to the requirements, some types of harm should be handled immediately due to their sensitivity. When this happens, the violation enforcement service removes the post immediately.

Violation enforcement service

The violation enforcement service immediately takes down a post if the harmful content detection service predicts harm with high confidence. It also notifies the user why the post was removed.

Demoting service

If the harmful content detection service predicts harm with low confidence, the demoting service temporarily demotes the post in order to decrease the chance of it spreading among users.

Then, the post is stored in storage for manual review by humans. The review team manually reviews the post and assigns a label from one of the predefined classes of harm. We will use these labeled posts in future training iterations to improve the model.

Other Talking Points

- Handle biases introduced by human labeling [19].
- Adapt the system to detect trending harmful classes (e.g., Covid-19, elections) [20].
- How to build a harmful content detection system that leverages temporal information such as users' sequence of actions [21][22].
- How to effectively select post samples for human review [23].
- How to detect authentic and fake accounts [24].
- How to deal with borderline contents [25], i.e., types of content that are not prohibited by guidelines, but come close to the red lines drawn by those policies.
- How to make the harmful content detection system efficient, so we can deploy it on-device [26].
- How to substitute Transformer-based architectures with linear Transformers to create a more efficient system [27] [28].

References

1. Facebook's inauthentic behavior. <https://transparency.fb.com/policies/community-standards/inauthentic-behavior/>.
2. LinkedIn's professional community policies. <https://www.linkedin.com/legal/professional-community-policies>.
3. Twitter's civic integrity policy. <https://help.twitter.com/en/rules-and-policies/election-integrity-policy>.
4. Facebook's integrity survey. <https://arxiv.org/pdf/2009.10311.pdf>.
5. Pinterest's violation detection system. <https://medium.com/pinterest-engineering/how-pinterest-fights-misinformation-hate-speech-and-self-harm-content-with-machine-learning-1806b73b40ef>.
6. Abusive detection at LinkedIn. <https://engineering.linkedin.com/blog/2019/isolation-forest>.
7. WPIE method. <https://ai.facebook.com/blog/community-standards-report/>.
8. BERT paper. <https://arxiv.org/pdf/1810.04805.pdf>.
9. Multilingual DistilBERT. <https://huggingface.co/distilbert-base-multilingual-cased>.
10. Multilingual language models. <https://arxiv.org/pdf/2107.00676.pdf>.
11. CLIP model. <https://openai.com/blog/clip/>.
12. SimCLR paper. <https://arxiv.org/pdf/2002.05709.pdf>.
13. VideoMoCo paper. <https://arxiv.org/pdf/2103.05905.pdf>.
14. Hyperparameter tuning. <https://cloud.google.com/ai-platform/training/docs/hyperparameter-tuning-overview>.
15. Overfitting. <https://en.wikipedia.org/wiki/Overfitting>.
16. Focal loss. <https://amaarora.github.io/posts/2020-06-29-FocalLoss.html>.
17. Gradient blending in multimodal systems. <https://arxiv.org/pdf/1905.12681.pdf>.
18. ROC curve vs precision-recall curve. <https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/>.
19. Introduced bias by human labeling. <https://labeledyourdata.com/articles/bias-in-machine-learning>.
20. Facebook's approach to quickly tackling trending harmful content. <https://ai.facebook.com/blog/harmful-content-can-evolve-quickly-our-new-ai-system-adapts-to-tackle-it/>.
21. Facebook's TIES approach. <https://arxiv.org/pdf/2002.07917.pdf>.
22. Temporal interaction embedding. <https://www.facebook.com/atscaleevents/videos/730968530723238/>.
23. Building and scaling human review system. <https://www.facebook.com/atscaleevents/videos/12017518833286>.

95/.

24. Abusive account detection framework. <https://www.youtube.com/watch?v=YeX4MdU0JNk>.
25. Borderline contents. <https://transparency.fb.com/features/approach-to-ranking/content-distribution-guidelines/content-borderline-to-the-community-standards/>.
26. Efficient harmful content detection. <https://about.fb.com/news/2021/12/metis-new-ai-system-tackles-harmful-content/>.
27. Linear Transformer paper. <https://arxiv.org/pdf/2006.04768.pdf>.
28. Efficient AI models to detect hate speech. <https://ai.facebook.com/blog/how-facebook-uses-super-efficient-ai-models-to-detect-hate-speech/>.

06

Video Recommendation System

Recommendation systems play a key role in video and music streaming services. For example, YouTube recommends videos a user may like, Netflix recommends movies a user may enjoy watching, and Spotify recommends music to users.

In this chapter, we design a video recommendation system similar to YouTube's [1]. The system recommends videos on the user's homepage based on their profile, previous interactions, etc.

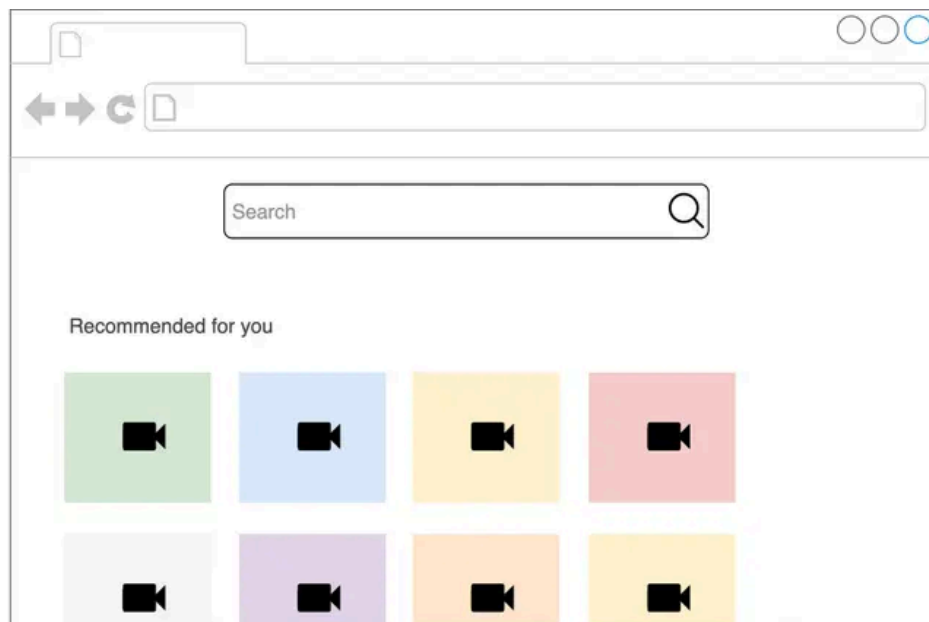


Figure 6.1: Homepage video recommendation

Recommendation systems are often very complex in design, and a good amount of engineering effort is required to develop an efficient and scalable system. Don't worry, though; no one expects you to build the perfect system in a 45 -minute interview. The interviewer is primarily interested in observing your thought process, communication skills, ability to design ML systems, and ability to discuss trade-offs.

Clarifying Requirements

Here is a typical interaction between a candidate and an interviewer.

Candidate: Can I assume the business objective of building a video recommendation system is to increase user engagement?

Interview: That's correct.

Candidate: Does the system recommend similar videos to a video a user is watching right now? Or does it show a personalized list of videos on the user's homepage?

Interviewer: This is a homepage video recommendation system, which recommends personalized videos to users when they load the homepage.

Candidate: Since YouTube is a global service, can I assume users are located worldwide and videos are in different languages?

Interviewer: That's a fair assumption.

Candidate: Can I assume we can construct the dataset based on user interactions with video content?

Interviewer: Yes, that sounds good.

Candidate: Can a user group videos together by creating playlists? Playlists can be informative for the ML model during the learning phase.

Interviewer: For the sake of simplicity, let's assume the playlist feature does not exist.

Candidate: How many videos are available on the platform?

Interviewer: We have about 10 billion videos.

Candidate: How fast should the system recommend videos to a user? Can I assume the recommendation should not take more than 200 milliseconds?

Interviewer: That sounds good.

Let's summarize the problem statement. We are asked to design a homepage video recommendation system. The business objective is to increase user engagement. Each time a user loads the homepage, the system recommends the most engaging videos. Users are located worldwide, and videos can be in different languages. There are approximately 10 billion videos on the platform, and recommendations should be served quickly.

Frame the Problem as an ML Task

Defining the ML objective

The business objective of the system is to increase user engagement. There are several options available for translating business objectives into well-defined ML objectives. We will examine some of them and discuss their trade-offs.

Maximize the number of user clicks. A video recommendation system can be designed to maximize user clicks. However, this objective has one major drawback. The model may recommend videos that are so-called "clickbait", meaning the title and thumbnail image look compelling, but the video's content may be boring, irrelevant, or even misleading. Clickbait videos reduce user satisfaction and engagement over time.

Maximize the number of completed videos. The system could also recommend videos users will likely watch to completion. A major problem with this objective is that the model may recommend shorter videos that are quicker to watch.

Maximize total watch time. This objective produces recommendations that users spend more time watching.

Maximize the number of relevant videos. This objective produces recommendations that are relevant to users.

Engineers or product managers can define relevance based on some rules. Such rules can be based on implicit and explicit user reactions. For example, one definition could state a video is relevant if a user explicitly presses the "like" button or watches at least half of it. Once we define relevance, we can construct a dataset and train a model to predict the relevance score between a user and a video.

In this system, we choose the final objective as the ML objective because we have more control over what signals to use. In addition, it does not have the shortcomings of the other options described earlier.

Specifying the system's input and output

As Figure 6.2 shows, a video recommendation system takes a user as input and outputs a ranked list of videos sorted by their relevance scores.

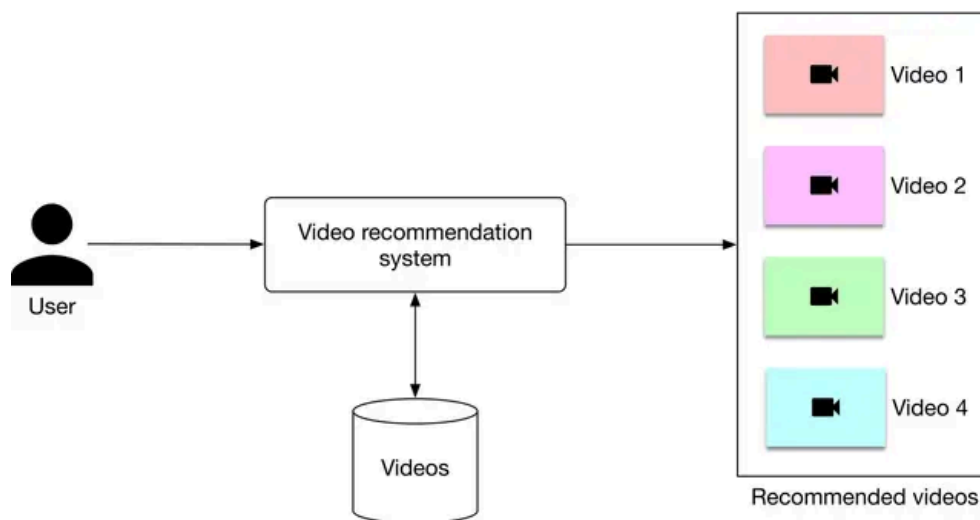


Figure 6.2: A video recommendation system's input-output

Choosing the right ML category

In this section, we examine three common types of personalized recommendation systems.

- Content-based filtering
- Collaborative filtering
- Hybrid filtering

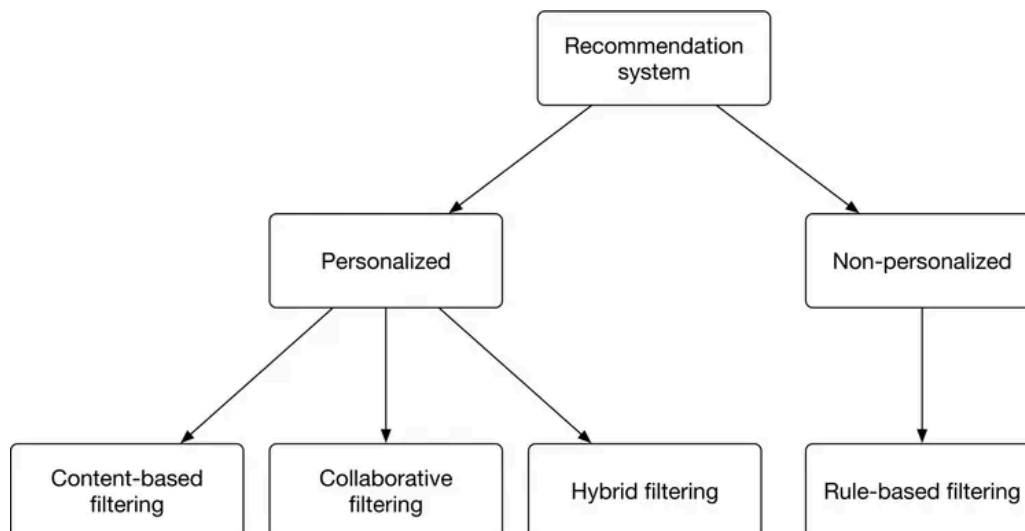


Figure 6.3: Common types of recommendation systems

Let's examine each type in more detail.

Content-based filtering

This technique uses video features to recommend new videos similar to those a user found relevant in the past. For example, if a user previously engaged with many ski videos, this method will suggest more ski videos. Figure 6.4 shows an example.

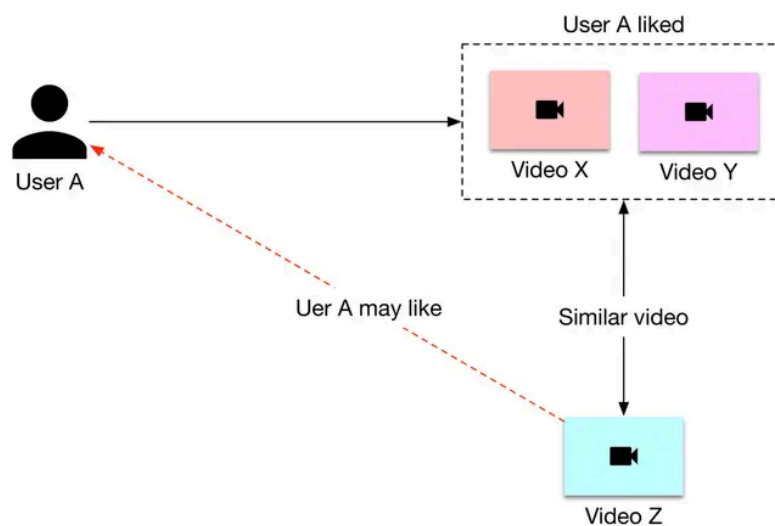


Figure 6.4: Content-based filtering

Here is an explanation of the diagram.

1. User A engaged with videos X and Y in the past
2. Video Z is similar to video X and video Y
3. The system recommends video Z to user A

Content-based filtering has pros and cons.

Pros:

- **Ability to recommend new videos.** With this method, we don't need to wait for interaction data from users to build video profiles for new videos. The video profile depends entirely upon its features.
- **Ability to capture the unique interests of users.** This is because we recommend videos based on users' previous engagements.

Cons:

- **Difficult to discover a user's new interests.**
- The method requires **domain knowledge**. We often need to engineer video features manually.

Collaborative filtering (CF)

CF uses user-user similarities (user-based CF) or video-video similarities (item-based CF) to recommend new videos. CF works with the intuitive idea that similar users are interested in similar videos. You can see a user-based CF example in Figure 6.5.

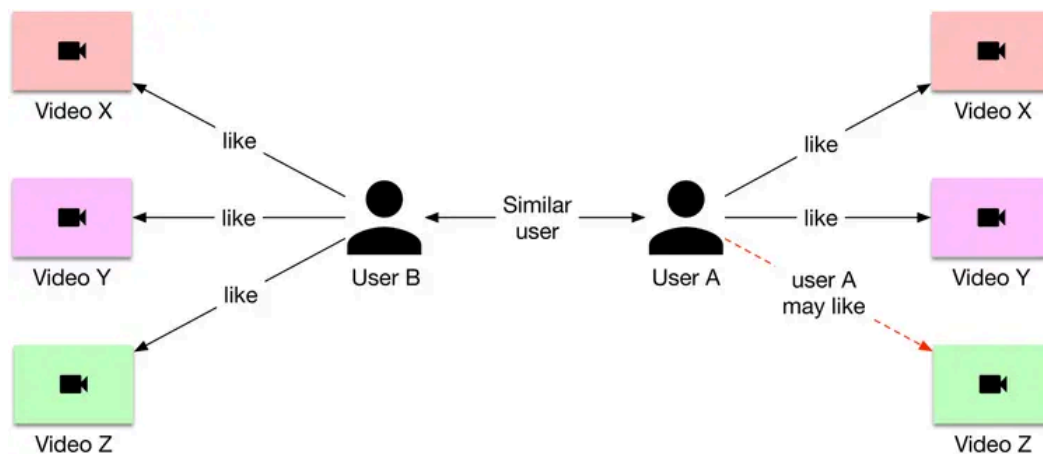


Figure 6.5: User-based collaborative filtering

Let's explain the diagram. The goal is to recommend a new video to user A.

1. Find a similar user to A based on their previous interactions; say user B
2. Find a video that user B engaged with but which user A has not seen yet; say video Z
3. Recommend video Z to user A

A major difference between content-based filtering and CF filtering is that CF filtering does not use video features and relies exclusively upon users' historical interactions to make recommendations. Let's see the pros and cons of CF filtering.

Pros:

- **No domain knowledge needed.** CF does not rely on video features, which means no domain knowledge is needed to engineer features from videos.
- **Easy to discover users' new areas of interest.** The system can recommend videos about new topics that

other similar users engaged with in the past.

- **Efficient.** Models based on CF are usually faster and less compute-intensive than content-based filtering, as they do not rely on video features.

Cons:

- **Cold-start problem.** This refers to a situation when limited data is available for a new video or user, meaning the system cannot make accurate recommendations. CF suffers from a cold-start problem due to the lack of historical interaction data for new users or videos. This lack of interactions prevents CF from finding similar users or videos. We will discuss later in the serving section how our system handles the cold-start problem.
- **Cannot handle niche interests.** It's difficult for CF to handle users with specialized or niche interests. CF relies upon similar users to make recommendations, and it might be difficult to find similar users with niche interests.

	Contentbased filtering	Collaborative filtering
Handle new videos	✓	✗
Discover new interest areas	✗	✓
No domain knowledge necessary	✗	✓
Efficiency	✗	✓

Table 6.1: Comparison between content-based filtering and CF

A comparison of the two types of filtering is shown in Table 6.1. As you see, the two methods are complementary.

Hybrid filtering

Hybrid filtering uses both CF and content-based filtering. As Figure 6.6 shows, hybrid filtering combines CF-based and content-based recommenders sequentially, or in parallel. In practice, companies usually use sequential hybrid filtering [2].

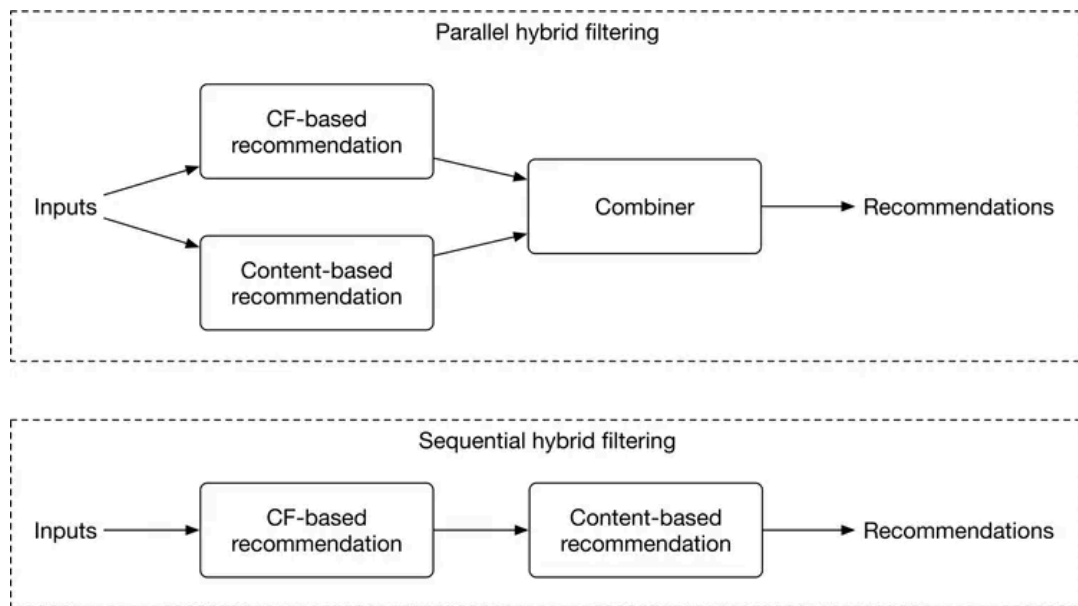


Figure 6.6: Hybrid filtering method

This approach leads to better recommendations because it uses two data sources: the user's historical interactions and video features. Video features allow the system to recommend relevant videos based on videos the user engaged with in the past, and CF-based filtering helps users to discover new areas of interest.

Which method should we choose?

Many companies use hybrid filtering to make better recommendations. For example, a paper published by Google [2] describes how YouTube employs a CF-based model as the first stage (candidate generator), followed by a content-based model as the second stage, to recommend videos. Due to the advantages of hybrid filtering, we choose this option.

Data Preparation

Data engineering

We have the following data available:

- Videos
- Users
- User-video interactions

Videos

Video data contains raw video files and their associated metadata, such as video ID, video length, video title, etc. Some of these attributes are provided explicitly by video uploaders, and others can be implicitly determined by the system, such as the video length.

Video ID	Length	Manual tags	Manual title	Likes	Views	Language
1	28	Dog, Family	Our lovely dog playing!	138	5300	English
2	300	Car, Oil	How to change your car oil?	5	250	Spanish
3	3600	Ouli, Vlog	Honeymoon to Bali	2200	255K	Arabic

Table 6.2: Video metadata

Users

The following simple schema represents user data.

ID	Username	Age	Gender	City	Country	Language	Time zone
----	----------	-----	--------	------	---------	----------	-----------

Table 6.3: User data schema

User-video interactions

The user-video interaction data consists of various user interactions with the videos, including likes, clicks, impressions, and past searches. Interactions are recorded along with other contextual information, such as location and timestamp. The following table shows how user-video interactions are stored.

User ID	Video ID	Interaction type	Interaction value	Location (lat, long)	Timestamp
4	18	Like	-	38.8951 -77.0364	1658451361
2	18	Impression	8 seconds	38.8951 -77.0364	1658451841
2	6	Watch	46 minutes	41.9241 -89.0389	1658822820
6	9	Click	-	22.7531 47.9642	1658832118
9	-	Search	Basics of clustering	22.7531 47.9642	1659259402
8	6	Comment	Amazing video. Thanks	37.5189 122.6405	1659244197

Table 6.4: User-video interaction data

Feature engineering

The ML system is required to predict videos that are relevant to users. Let's engineer features to help the system make informed predictions.

Video features

Some important video features include:

- Video ID
- Duration
- Language
- Titles and tags

Video ID

The IDs are categorical data. To represent them by numerical vectors, we use an embedding layer, and the embedding layer is learned during model training.

Duration

This defines approximately how long the video lasts from start to finish. This information is important since some users may prefer shorter videos, while others prefer longer videos.

Language

The language used in a video is an important feature. This is because users naturally prefer particular languages. Since language is a categorical variable and takes on a finite set of discrete values, we use an embedding layer to represent it.

Titles and tags

Titles and tags are used to describe a video. They are either provided manually by the uploader or are implicitly predicted by standalone ML models. The titles and tags of a video are valuable predictors. For example, a video titled "how to make pizza" indicates the video is related to pizza and cooking.

How to prepare it? For tags, we use a lightweight pre-trained model, such as CBOW [3], to map them into feature vectors.

For the title, we map it into a feature vector using a context-aware word embedding model, such as a pre-trained BERT [4].

Figure 6.7 shows an overview of video feature preparation.

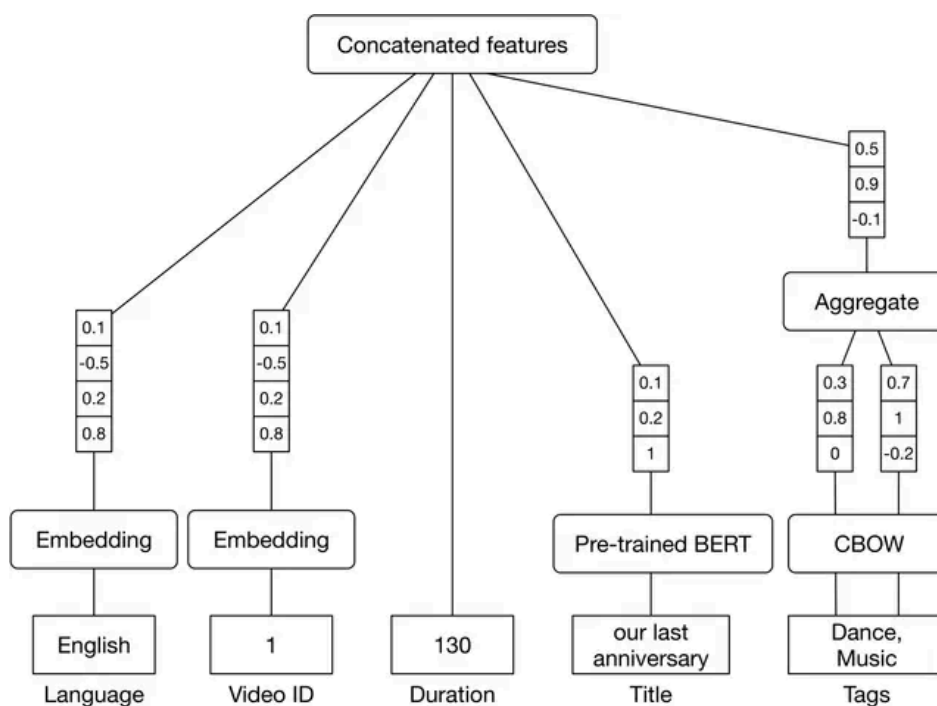


Figure 6.7: Video feature preparation

User features

We categorize user features into the following buckets:

- User demographics
- Contextual information
- User historical interactions

User demographics

An overview of user demographic features is shown in Figure 6.8.

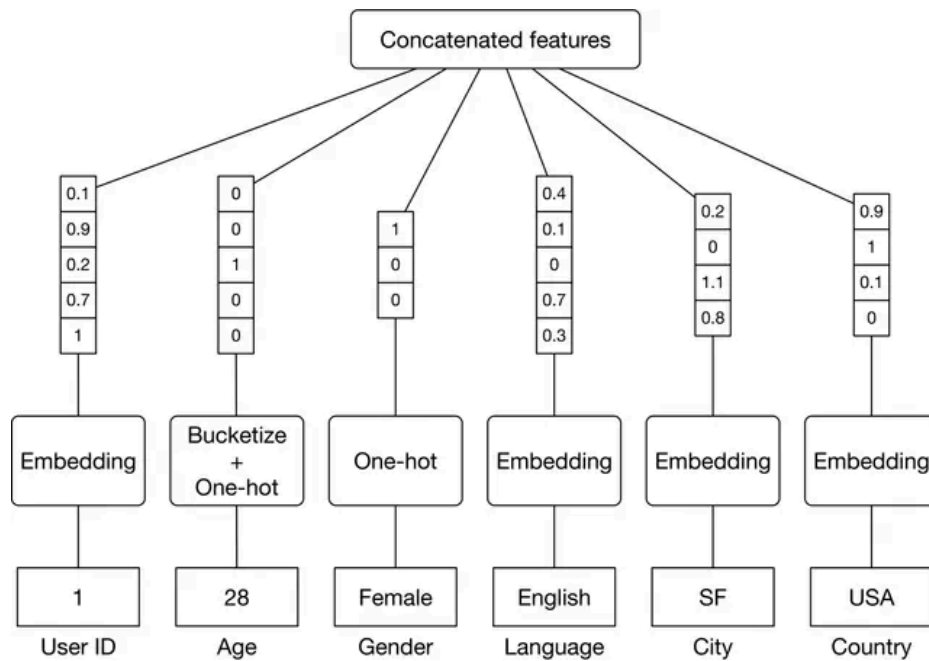


Figure 6.8: Features based on a user's demographics

Contextual information

Here are a few important features for capturing contextual information:

- **Time of day.** A user may watch different videos at different times of day. For example, a software engineer may watch more educational videos during the evening.
- **Device.** On mobile devices, users may prefer shorter videos.
- **Day of the week.** Depending on the day of the week, users may have different preferences for videos.

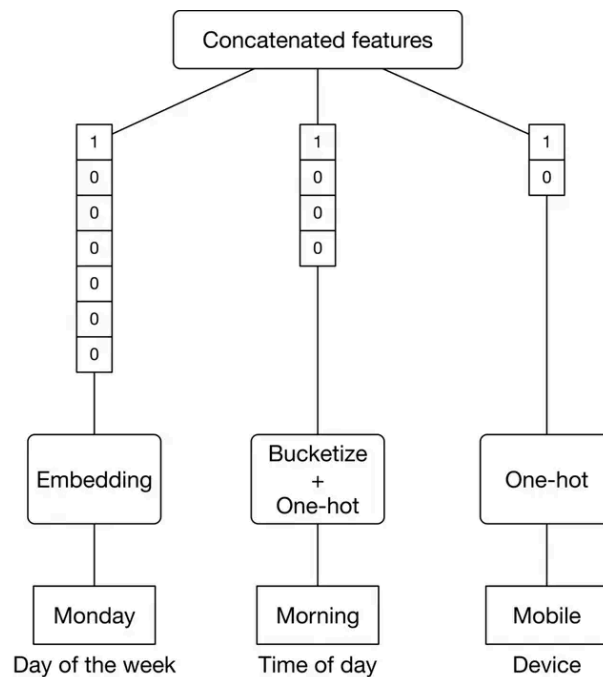


Figure 6.9: Features related to contextual information

User historical interactions

User historical interactions play an important role in understanding user interests. A few features related to historical interactions are:

- Search history
- Liked videos
- Watched videos and impressions

Search history

Why is it important?

Previous searches indicate what the user looked for in the past, and past behavior is often an indicator of future behavior.

How to prepare it?

Use a pre-trained word embedding model, such as BERT, to map each search query into an embedding vector. Note that a user's search history is a variable-sized list of textual queries. To create a fixed-size feature vector summarizing all the search queries, we average the query embeddings.

Liked videos

Why is it important?

The videos a user liked previously can be helpful in determining which type of content they're interested in.

How to prepare it?

Video IDs are mapped into embedding vectors using the embedding layer. Similarly to search history, we average liked embeddings to get a fixed-size vector of liked videos.

Watched videos and impressions

The feature engineering process for “watched videos” and “impressions” is very similar to what we did for liked videos. So, we won't repeat it.

Figure 6.10 summarizes features related to user-video interactions.

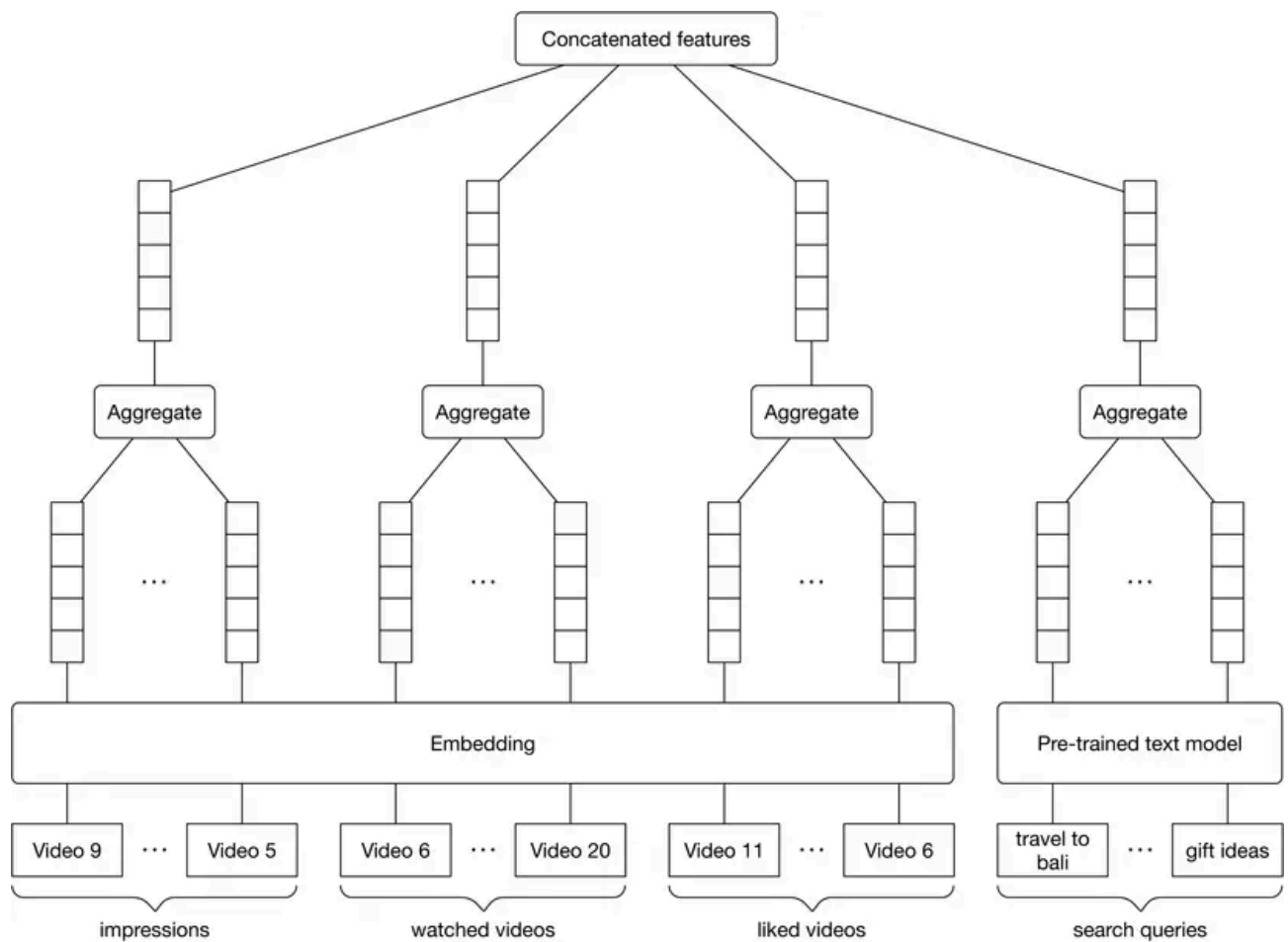


Figure 6.10: Features related to user-video interactions

Model Development

In this section, we examine two embedding-based models that are typically employed in CF-based or content-based recommenders:

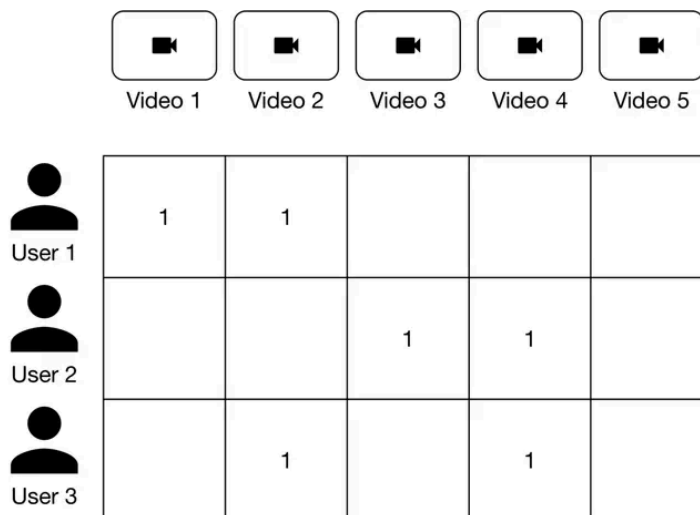
- Matrix factorization
- Two-tower neural network

Matrix factorization

To understand the matrix factorization model, it is important to know what a feedback matrix is.

Feedback matrix

Also called a utility matrix, this is a matrix that represents users' opinions about videos. Figure 6.11 shows a binary user-video feedback matrix where each row represents a user, and each column represents a video. The entries in the matrix specify the user's opinion to 1 as "observed" or "positive."



	Video 1	Video 2	Video 3	Video 4	Video 5
User 1	1	1			
User 2			1	1	
User 3		1		1	

Figure 6.11: User-video feedback matrix

How can we determine whether a user finds a recommended video relevant? We have three options:

- Explicit feedback
- Implicit feedback
- Combination of explicit and implicit feedback

Explicit feedback. A feedback matrix is built based on interactions that explicitly indicate a user's opinion about a video, such as likes and shares. Explicit feedback reflects a user's opinion accurately as users explicitly expressed their interest in a video. This option, however, has one major drawback: the matrix is sparse since only a small fraction of users provide explicit feedback. Sparsity makes ML models difficult to train.

Implicit feedback. This option uses interactions that implicitly indicate a user's opinion about a video, such as "clicks" or "watch time". With implicit feedback, more data points are available, resulting in a better model after training. Its main disadvantage is that it does not directly reflect users' opinions and might be noisy.

Combination of explicit and implicit feedback. This option combines explicit and implicit feedback using heuristics.

What is the best option for building our feedback matrix?

Since the model needs to learn the values of the feedback matrix, it's important to build the matrix that aligns well with the ML objective we chose earlier.

In our case, the ML objective is to maximize relevancy, where relevancy is defined as the combination of explicit and implicit feedback. As such, the final option of combining explicit and implicit feedback is the best choice.

Matrix factorization model

Matrix factorization is a simple embedding model. The algorithm decomposes the user-video feedback matrix into the product of two lower-dimensional matrices. One lower-dimensional matrix represents user embeddings, and the other represents video embeddings. In other words, the model learns to map each user into an embedding vector and each video into an embedding vector, such that their distance represents their relevance. Figure 6.12 shows how a feedback matrix is decomposed into user and video embeddings.

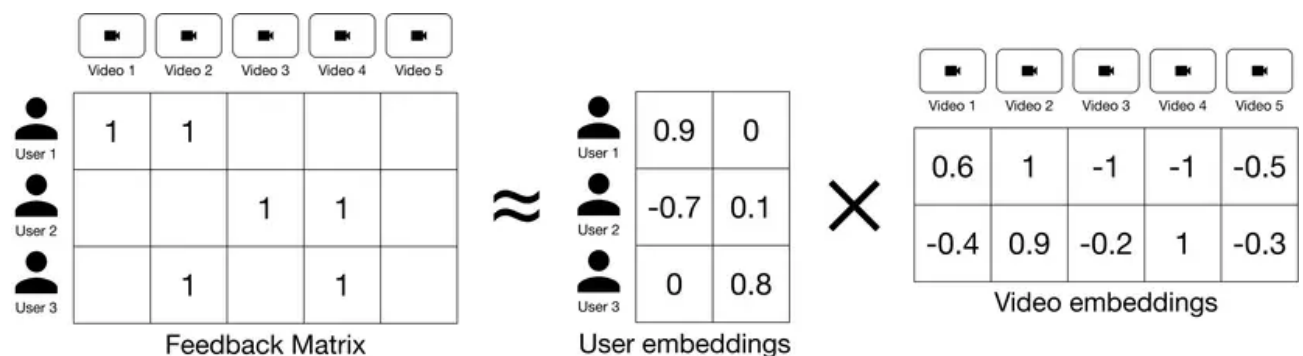


Figure 6.12: Decompose the feedback matrix into two matrices

Matrix factorization training

As part of training, we aim to produce user and video embedding matrices so that their product is a good approximation of the feedback matrix (Figure 6.13.)

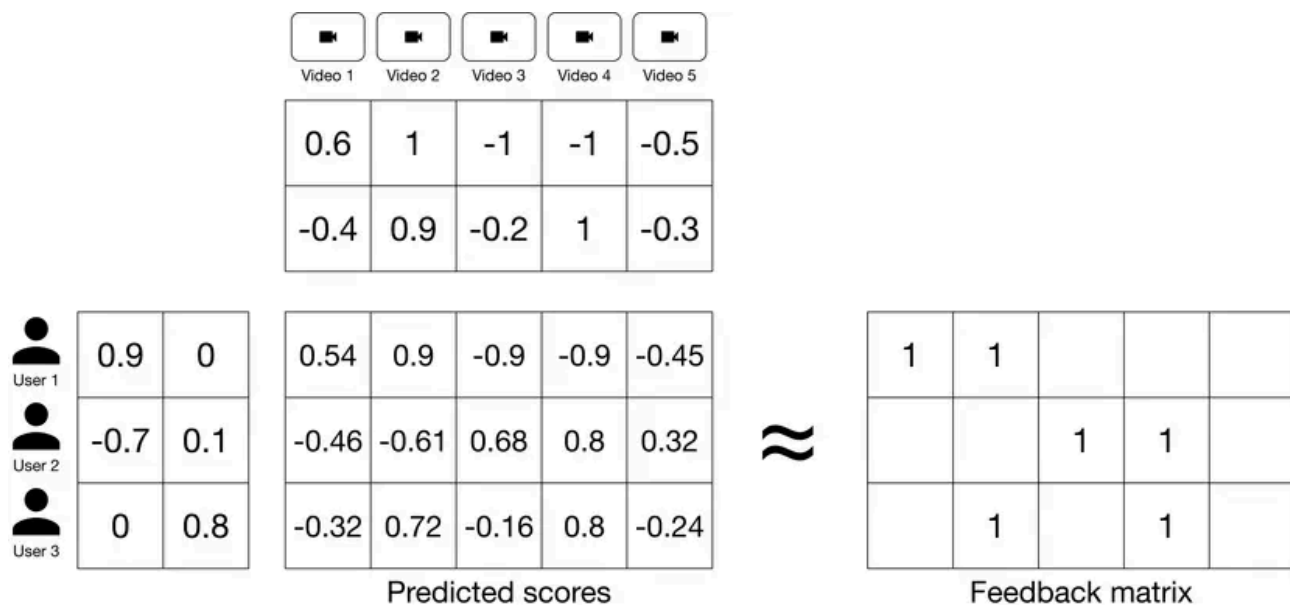


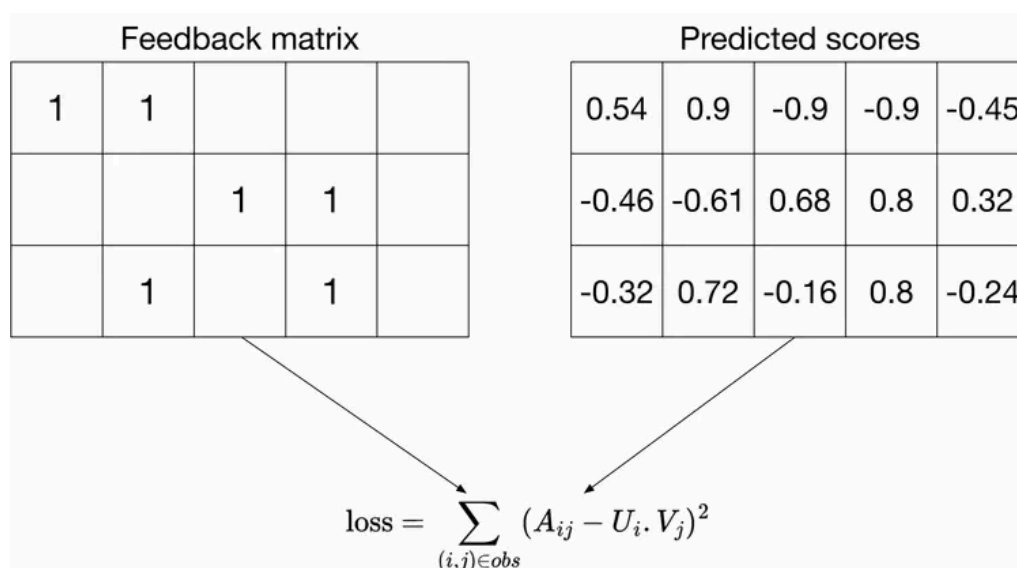
Figure 6.13: The product of embeddings should approximate the feedback matrix

To learn these embeddings, matrix factorization first randomly initializes two embedding matrices, then iteratively optimizes the embeddings to decrease the loss between the "Predicted scores matrix" and the "Feedback matrix". Loss function selection is an important consideration. Let's explore a few options:

- Squared distance over observed $\langle \text{user}, \text{video} \rangle$ pairs
- A weighted combination of squared distance over observed pairs and unobserved pairs

Squared distance over observed $\langle \text{user}, \text{video} \rangle$ pairs

This loss function measures the sum of the squared distances over all pairs of observed (non-zero values) entries in the feedback matrix. This is shown in Figure 6.14.

Figure 6.14: Squared distance over observed $\langle \text{user}, \text{video} \rangle$ pairs

A_{ij} refers to the entry with row i and column j in the feedback matrix, U_i is the embedding of user i , V_j is the embedding of video j , and the summation is over the observed pairs only.

Only summing over observed pairs leads to poor embeddings because the loss function doesn't penalize the model for bad predictions on unobserved pairs. For example, embedding matrices of all ones would have a zero loss on the training data. However, those embeddings may not work well for unseen $\langle \text{user}, \text{video} \rangle$ pairs.

Squared distance over both observed and unobserved $\langle \text{user}, \text{video} \rangle$ pairs This loss function treats unobserved pairs as negative data points and assigns a zero to them in the feedback matrix. As Figure 6.15 shows, the loss computes the sum of the squared distances over all entries in the feedback matrix.

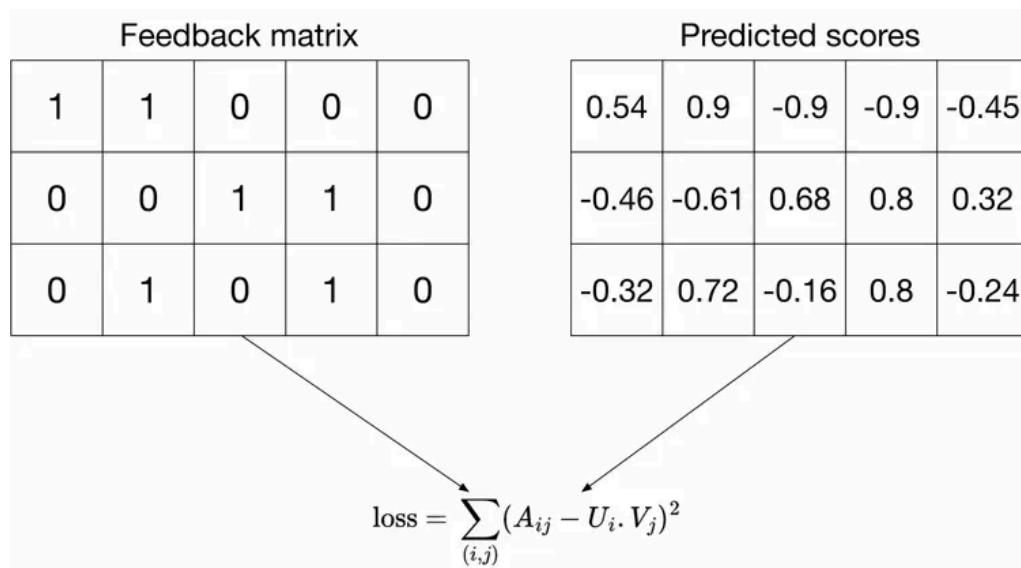


Figure 6.15: Squared distance over all $\langle \text{user}, \text{video} \rangle$ pairs

This loss function addresses the previous issue by penalizing bad predictions for unobserved entries. However, this loss has a major drawback. The feedback matrix is usually sparse (lots of unobserved pairs), so unobserved pairs dominate observed pairs during training. This results in predictions that are mostly close to zero. This is not desirable and leads to poor generalization performance on unseen $\langle \text{user}, \text{video} \rangle$ pairs.

A weighted combination of squared distance over observed and unobserved pairs

To overcome the drawbacks of the loss functions described earlier, we opt for weighted combinations of both.

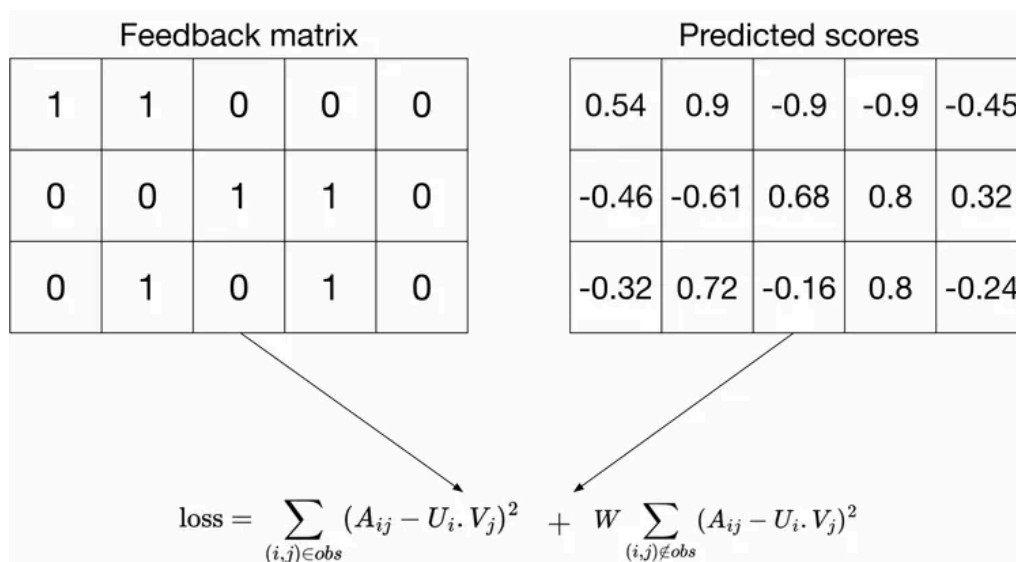


Figure 6.16: Combined loss

The first summation in the loss formula calculates the loss on the observed pairs, and the second summation calculates the loss on unobserved pairs. W is a hyperparameter that weighs the two summations. It ensures one does not dominate the other in the training phase. This loss function with a properly tuned W works well in practice [5]. We choose this loss function for the system.

Matrix factorization optimization

To train an ML model, an optimization algorithm is required. Two commonly used optimization algorithms in matrix factorization are:

- Stochastic Gradient Descent (SGD): This optimization algorithm is used to minimize losses [6].
- Weighted Alternating Least Squares (WALS): This optimization algorithm is specific to matrix factorization.

The process in WALS is:

1. Fix one embedding matrix (U), and optimize the other embedding (V)
2. Fix the other embedding matrix (V), and optimize the embedding matrix (U)
3. Repeat.

WALS usually converges faster and is parallelizable. To learn more about WALS, read [7]. Here, we use WALS because it converges faster.

Matrix factorization inference

To predict the relevance between an arbitrary user and a candidate video, we calculate the similarity between their embeddings using a similarity measure, such as a dot product. For example, as shown in Figure 6.17, the relevance score between user 2 and video 5 is 0.32.

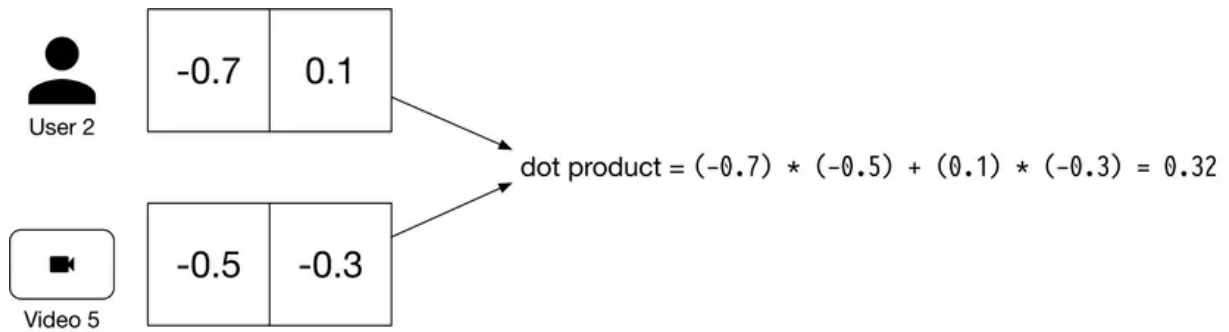
Figure 6.17: Relevance score for $\langle \text{user 2, video 5} \rangle$ pair

Figure 6.18 shows the predicted scores for all the $\langle \text{user, video} \rangle$ pairs. The system returns recommended videos based on relevance scores.

		Video 1	Video 2	Video 3	Video 4	Video 5
		0.6	1	-1	-1	-0.5
		-0.4	0.9	-0.2	1	-0.3
	User 1	0.9	0	0.54	0.9	-0.9
	User 2	-0.7	0.1	-0.46	-0.61	0.68
	User 3	0	0.8	-0.32	0.72	-0.16
					0.8	-0.24

Figure 6.18: Predicted pairwise relevance scores

Reminder

Since matrix factorization uses user-video interactions only, it is commonly used in collaborative filtering.

Before wrapping up matrix factorization, let's discuss the pros and cons of this model.

Pros:

- Training speed: Matrix factorization is efficient during the training phase. This is because there are only two embedding matrices to learn.
- Serving speed: Matrix factorization is fast at serving time. The learned embeddings are static, meaning that once we learn them, we can reuse them without having to transform the input at query time.

Cons:

- Matrix factorization only relies on user-video interactions. It does not use other features, such as the user's age or language. This limits the predictive capability of the model because features like language are useful to improve the quality of recommendations.
- Handling new users is difficult. For new users, there are not enough interactions for the model to produce meaningful embeddings. Therefore, matrix factorization cannot determine whether a video is relevant to a user by computing the dot product between their embeddings.

Let's see how two-tower neural networks address the shortcomings of matrix factorization.

Two-tower neural network

A two-tower neural network comprises two encoder towers: the user tower and the video tower. The user encoder takes user features as input and maps them to an embedding vector (user embedding). The video encoder takes video features as input and maps them into an embedding vector (video embedding). The distance between their embeddings in the shared embedding space represents their relevance.

Figure 6.19 shows the two-tower architecture. In contrast to matrix factorization, two-tower architectures are flexible enough to incorporate all kinds of features to better capture the user's specific interests.

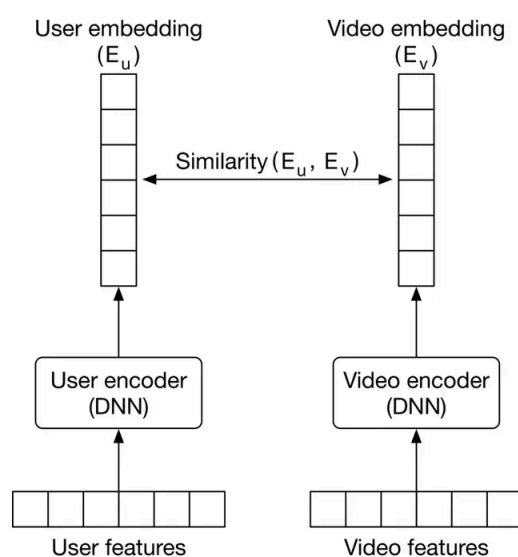


Figure 6.19: Two-tower neural network

Constructing the dataset

We construct the dataset by extracting features from different $\langle \text{user}, \text{video} \rangle$ pairs and labeling them as positive or negative based on the user's feedback. For example, we label a pair as "positive" if the user explicitly liked the video, or watched at least half of it.

To construct negative data points, we can either choose random videos which are not relevant or choose those the user explicitly disliked by pressing the dislike button. Figure 6.20 shows an example of the constructed data

points.

#	User-related features	Video-related features	Label													
1	<table><tr><td>0</td><td>0</td><td>1</td><td>0.7</td><td>-0.6</td><td>0</td><td>0</td></tr></table>	0	0	1	0.7	-0.6	0	0	<table><tr><td>0</td><td>1</td><td>0</td><td>0.9</td><td>0.9</td><td>1</td></tr></table>	0	1	0	0.9	0.9	1	1 (positive)
0	0	1	0.7	-0.6	0	0										
0	1	0	0.9	0.9	1											
2	<table><tr><td>0</td><td>1</td><td>1</td><td>0.2</td><td>0.1</td><td>1</td><td>0</td></tr></table>	0	1	1	0.2	0.1	1	0	<table><tr><td>0</td><td>1</td><td>0</td><td>-0.1</td><td>0.3</td><td>1</td></tr></table>	0	1	0	-0.1	0.3	1	0 (negative)
0	1	1	0.2	0.1	1	0										
0	1	0	-0.1	0.3	1											

Figure 6.20: Two constructed data points

Note, users usually only find a small fraction of videos relevant. While constructing training data, this leads to an imbalanced dataset where there are many more negative than positive pairs. Training a model on an imbalanced dataset is problematic. We can use the techniques described in Chapter 1 Introduction and Overview, to address the data imbalance issue.

Choosing the loss function

Since the two-tower neural network is trained to predict binary labels, the problem can be categorized as a classification task. We use a typical classification loss function, such as cross-entropy, to optimize the encoders during training. This process is shown in Figure 6.21

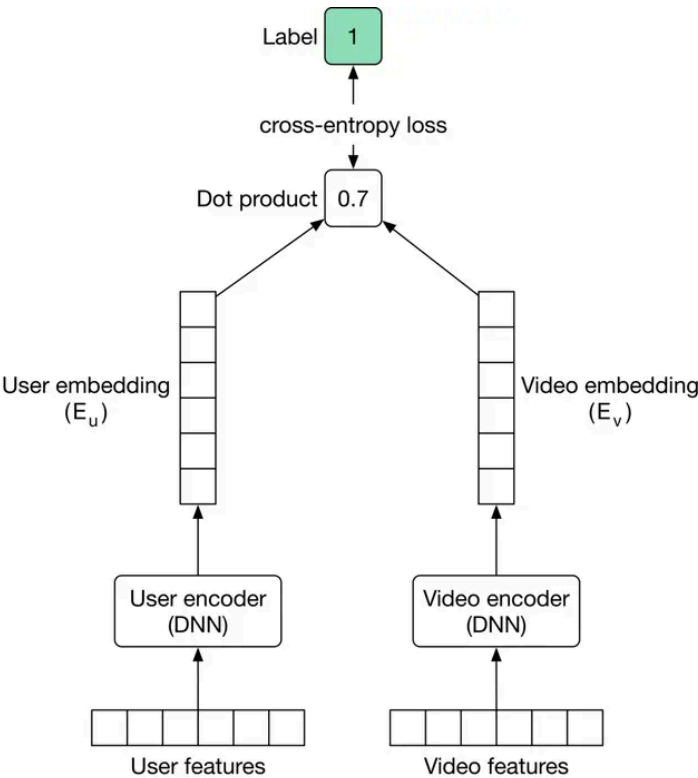


Figure 6.21: Two-tower neural network training workflow

Two-tower neural network inference

At inference time, the system uses the embeddings to find the most relevant videos for a given user. This is a classic "nearest neighbor" problem. We use approximate nearest neighbor methods to find the top k most similar video embeddings efficiently.

Two-tower neural networks are used for both content-based filtering and collaborative filtering. When a two-tower architecture is used for collaborative filtering, as shown in Figure 6.22, the video encoder is nothing but an embedding layer that converts the video ID into an embedding vector. This way, the model doesn't rely on other video features.

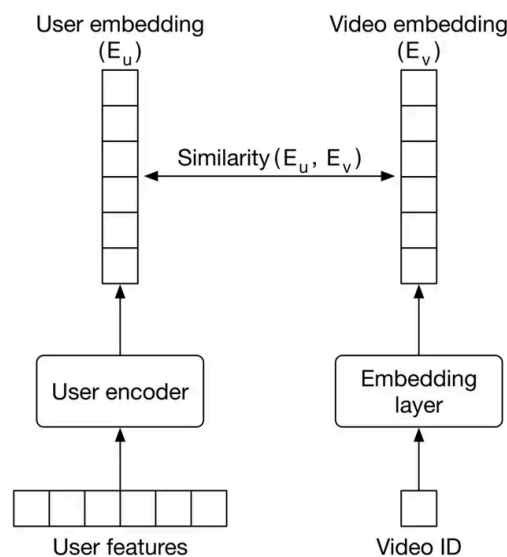


Figure 6.22: Two-tower neural network used for collaborative filtering

Let's see the pros and cons of a two-tower neural network model.

Pros:

- **Utilizes user features.** The model accepts user features, such as age and gender, as input. These predictive features help the model make better recommendations.
- **Handles new users.** The model easily handles new users as it relies on user features (e.g., age, gender, etc.).

Cons:

- **Slower serving.** The model needs to compute the user embedding at query time. This makes the model slower to serve requests. In addition, if we use the model for content-based filtering, the model needs to transform video features into video embedding, which increases the inference time.
- **Training is more expensive.** Two-tower neural networks have more learning parameters than matrix factorization. Therefore, the training is more compute-intensive.

Matrix factorization vs. two-tower neural network

Table 6.5 summarizes the differences between matrix factorization and two-tower neural network architecture.

	Matrix factorization	Two-tower neural network
Training cost	✓ More efficient to train	✗ More costly to train
Inference speed	✓ Faster as embeddings are static and can be precomputed	✗ User features should be transformed into embeddings at query time
Cold-start problem	✗ Cannot handle new users easily	✓ Handles new users as it relies on user features
Quality of recommendations	✗ Not ideal since the model does not use user/video features	✓ Better recommendations since it relies on more features

Table 6.5: Matrix factorization vs. two-tower neural networks

Evaluation

The system's performance can be evaluated with offline and online metrics.

Offline metrics

We evaluate the following offline metrics commonly used in recommendation systems.

Precision@k. This metric measures the proportion of relevant videos among the top k recommended videos. Multiple k values (e.g., 1, 5, 10) can be used.

mAP. This metric measures the ranking quality of recommended videos. It is a good fit because the relevance scores are binary in our system.

Diversity. This metric measures how dissimilar recommended videos are to each other. This metric is important to track, as users are more interested in diversified videos. To measure diversity, we calculate the average pairwise similarity (e.g., cosine similarity or dot product) between videos in the list. A low average pairwise similarity score indicates the list is diverse.

Note that using diversity as the sole measure of quality can result in misleading interpretations. For example, if the recommended videos are diverse but irrelevant to the user, they may not find the recommendations helpful. Therefore, we should use diversity with other offline metrics to ensure both relevance and diversity.

Online metrics

In practice, companies track many metrics during online evaluation. Let's examine some of the most important ones:

- Click-through rate (CTR)

- The number of completed videos
- Total watch time
- Explicit user feedback

CTR. The ratio between clicked videos and the total number of recommended videos. The formula is:

$$CTR = \frac{\text{number of clicked videos}}{\text{total number of recommended videos}}$$

CTR is an insightful metric to track user engagement, but the drawback of CTR is that we cannot capture or measure clickbait videos.

The number of completed videos. The total number of recommended videos that users watch until the end. By tracking this metric, we can understand how often the system recommends videos that users watch.

Total watch time. The total time users spent watching the recommended videos. When recommendations interest users, they spend more time watching videos, overall.

Explicit user feedback. The total number of videos that users explicitly liked or disliked. The metric accurately reflects users' opinions of recommended videos.

Serving

At serving time, the system recommends the most relevant videos to a given user by narrowing the selection down from billions of videos. In this section, we will propose a prediction pipeline that's both efficient and accurate at serving requests.

Given we have billions of videos available, the serving speed would be slow if we choose a heavy model which takes lots of features as input. On the other hand, if we choose a lightweight model, it may not produce high-quality recommendations. So, what to do? A natural decision is to use more than one model in a multi-stage design. For example, in a two-stage design, a lightweight model quickly narrows down the videos during the first stage, called candidate generation. The second stage uses a heavier model that accurately scores and ranks the videos, called scoring. Figure 6.23 shows how candidate generation and scoring work together to produce relevant videos.

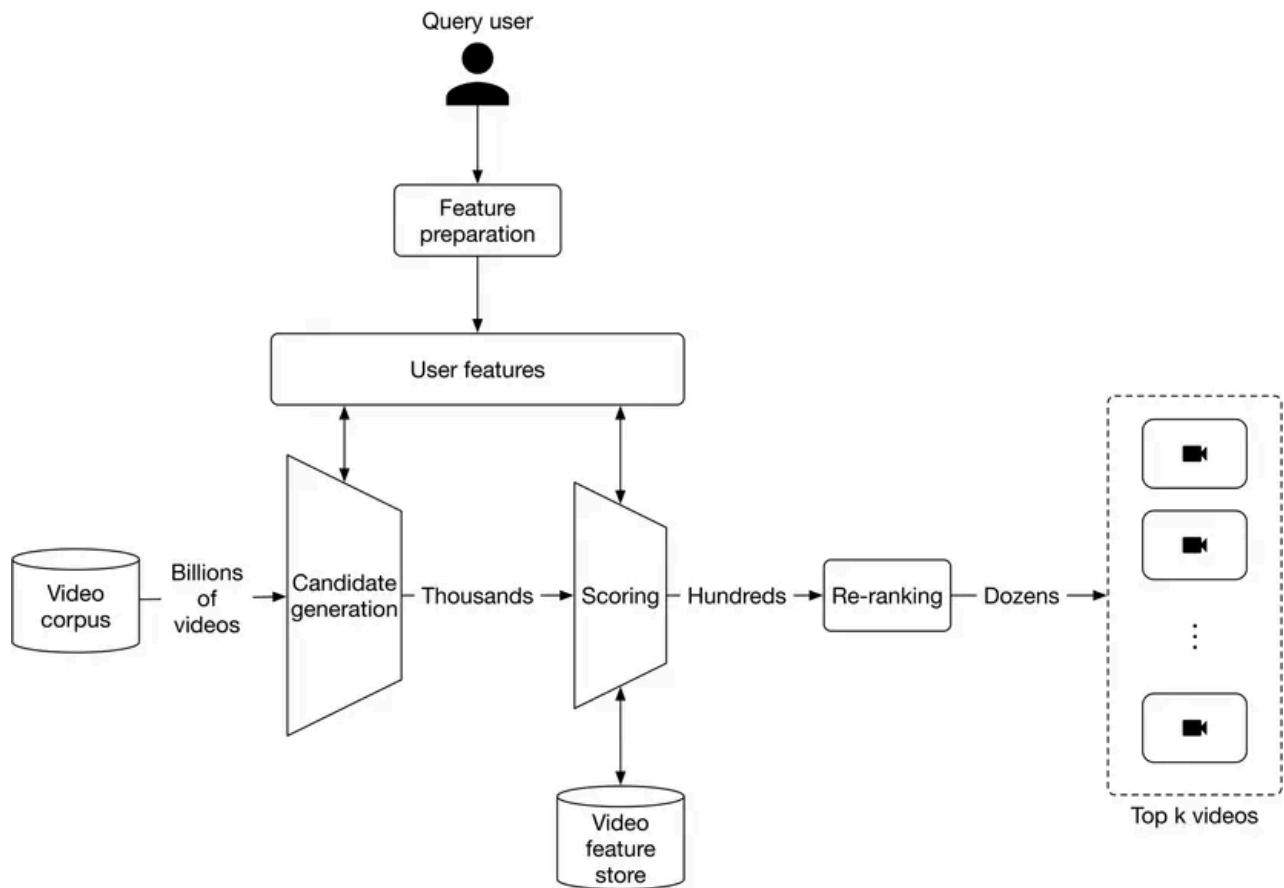


Figure 6.23: Prediction pipeline

Let's take a closer look at the components of the prediction pipeline.

- Candidate generation
- Scoring
- Re-ranking

Candidate generation

The goal of candidate generation is to narrow down the videos from potentially billions, to thousands. We prioritize efficiency over accuracy at this stage and are not concerned about false positives.

To keep candidate generation fast, we choose a model which doesn't rely on video features. In addition, this model should be able to handle new users. A two-tower neural network is a good fit for this stage.

Figure 6.24 shows the candidate generation workflow. The candidate generation obtains a user's embedding from the user encoder. Once the computation is complete, it retrieves the most similar videos from the approximate nearest neighbor service. These videos are ranked based on similarity in the embedding space and are returned as the output.

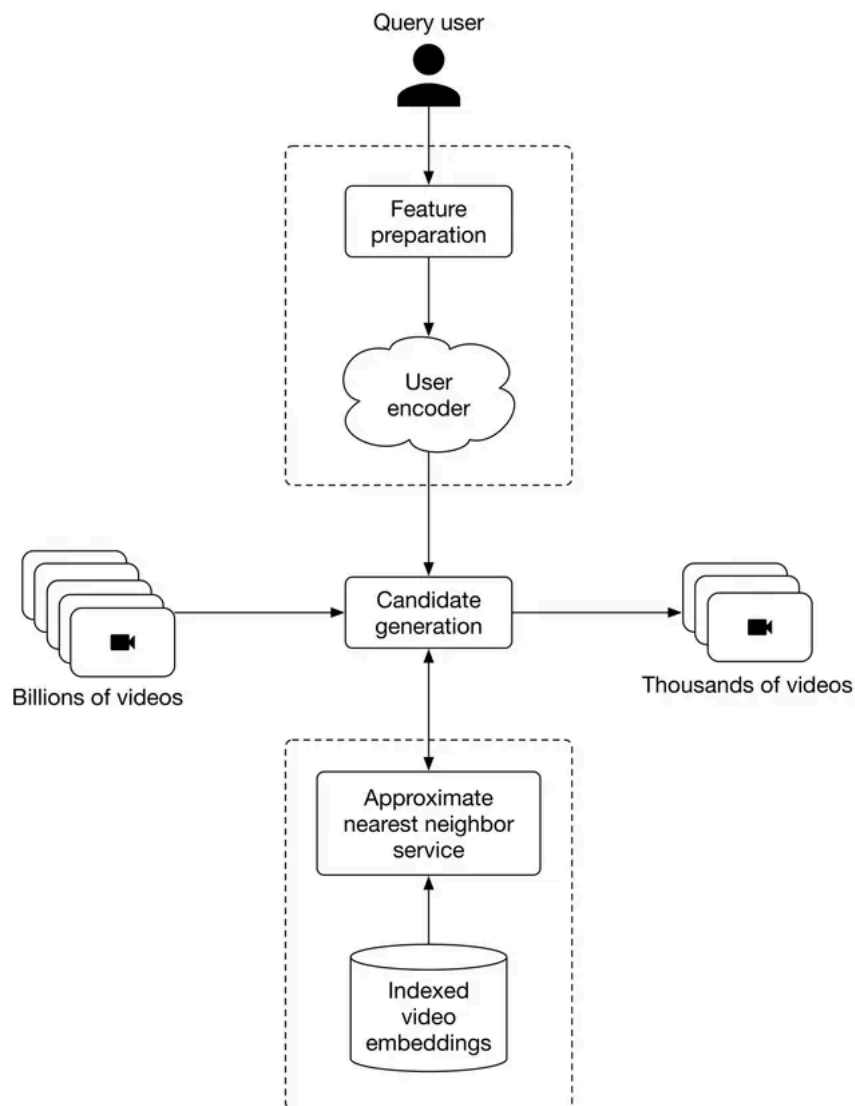


Figure 6.24: Candidate generation workflow

In practice, companies may choose to use more than one candidate generation because it could improve the performance of the recommendation. Let's take a look at why.

Users may be interested in videos for many reasons. For example, a user may choose to watch a video because it's popular, trending, or relevant to their location. To include those videos in the recommendations, it is common to use more than one candidate generation, as shown in Figure 6.25.

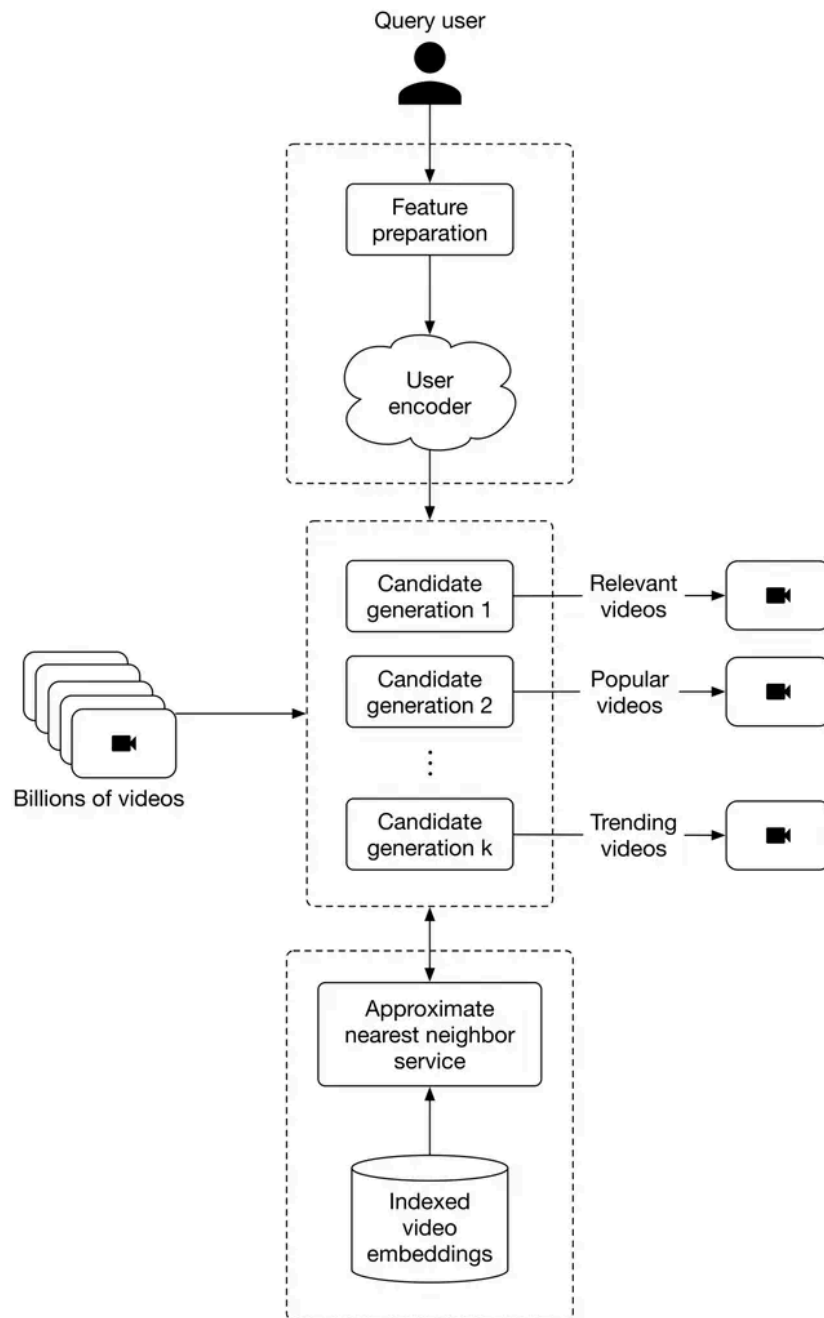


Figure 6.25: Use k candidate generations to diversify recommended videos

As soon as we have narrowed down potential videos from billions to thousands, we can use a scoring component to rank these videos before they are displayed.

Scoring

Also known as ranking, scoring takes the user and candidate videos as input, scores each video, and outputs a ranked list of videos.

At this stage, we prioritize accuracy over efficiency. To do so, we choose content-based filtering and pick a model which relies on video features. A two-tower neural network is a common choice for this stage. Since there

are only a handful of videos to rank in the scoring stage, we can employ a heavier model with more parameters. Figure 6.26 shows an overview of the scoring component.

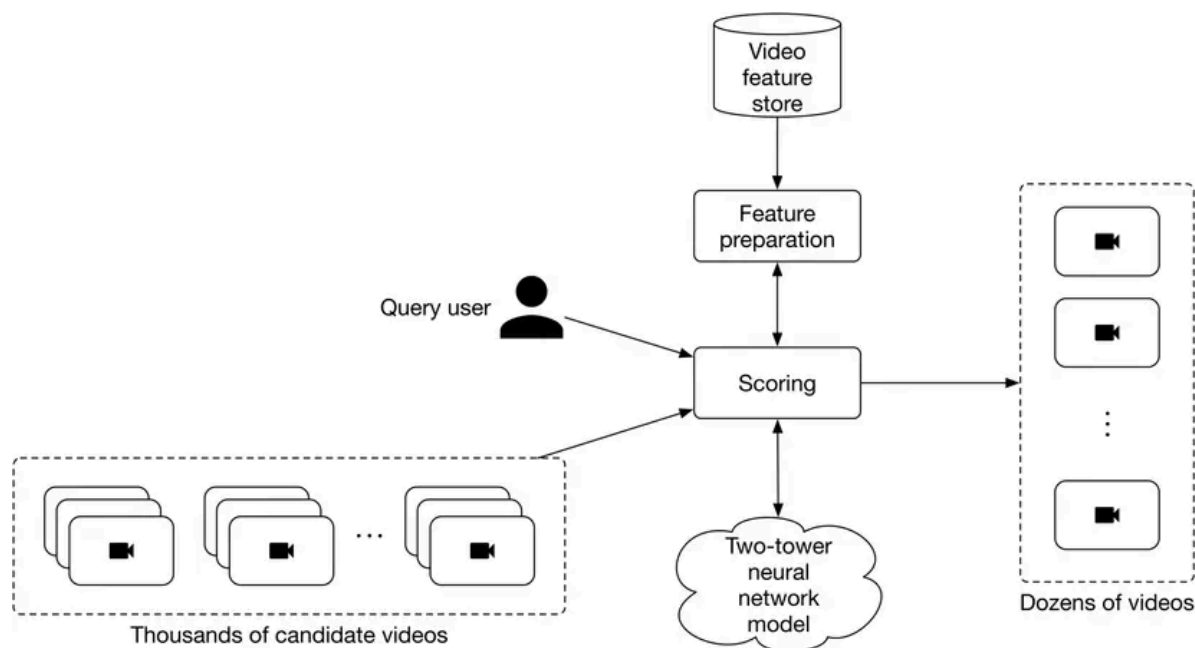


Figure 6.26: Overview of the scoring component

Re-ranking

This component re-ranks the videos by adding additional criteria or constraints. For example, we may use standalone ML models to determine if a video is clickbait. Here are a few important things to consider when building the re-ranking component:

- Region-restricted videos
- Video freshness
- Videos spreading misinformation
- Duplicate or near-duplicate videos
- Fairness and bias

Challenges of video recommendation systems

Before wrapping up this chapter, let's see how our design addresses typical challenges in video recommendation systems.

Serving speed

It is vital to recommend videos fast. However, as we have billions of videos in this system, recommending them efficiently and accurately is challenging. To address this issue, we used a two-stage design.

Specifically, we use a lightweight model in the first stage to quickly narrow down candidate videos from billions to thousands. YouTube uses a similar approach [2], and Instagram adopts a multi-stage design [8].

Precision

To ensure precision, we employ a scoring component that ranks videos using a powerful model, which relies on more features, including video features. Using a more powerful model doesn't affect serving speed because only a small subset of videos is selected after the candidate generation phase.

Diversity

Most users prefer to see a diverse selection of videos in their recommendations. To ensure our system produces a diverse set of videos, we adopt multiple candidate generators, as explained in the candidate generation section.

Cold-start problem

How does our system handle the cold-start problem?

For new users: We don't have any interaction data about new users when they begin using our platform.

In this case, predictions are made using two-tower neural networks based on features such as age, gender, language, location, etc. The recommended videos are personalized to some extent, even for new users. As the user interacts with more videos, we are able to make better predictions based on new interactions.

For new videos: When a new video is added to the system, the video metadata and content are available, but no interactions are present. One way to handle this is to use heuristics. We can display videos to random users and collect interaction data. Once we gather enough interactions, we fine-tune the two-tower neural network using the new interactions.

Training scalability

It's challenging to train models on large datasets in a cost-effective manner. In recommendation systems, new interactions are continuously added, and the models need to quickly adapt to make accurate recommendations. To quickly adapt to new data, the models should be able to be fine-tuned.

In our case, the models are based on neural networks and designed to be easily fine-tuned.

Other Talking Points

If there is time left at the end of the interview, here are some additional talking points:

- The exploration-exploitation trade-off in recommendation systems [9].
- Different types of biases may be present in recommendation systems [10].
- Important considerations related to ethics when building recommendation systems [11].
- Consider the effect of seasonality - changes in users' behaviors during different seasons - in a recommendation system [12].
- Optimize the system for multiple objectives, instead of a single objective [13].
- How to benefit from negative feedback such as dislikes [14].
- Leverage the sequence of videos in a user's search history or watch history [2].

References

1. YouTube recommendation system. <https://blog.youtube/inside-youtube/on-youtubes-recommendation-system>.
2. DNN for YouTube recommendation. <https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45530.pdf>.
3. CBOW paper. <https://arxiv.org/pdf/1301.3781.pdf>.
4. BERT paper. <https://arxiv.org/pdf/1810.04805.pdf>.
5. Matrix factorization. <https://developers.google.com/machine-learning/recommendation/collaborative/matrix>.
6. Stochastic gradient descent. https://en.wikipedia.org/wiki/Stochastic_gradient_descent.
7. WALS optimization. <https://fairyonice.github.io/Learn-about-collaborative-filtering-and-weighted-alternating-least-square-with-tensorflow.html>.
8. Instagram multi-stage recommendation system. <https://ai.facebook.com/blog/powered-by-ai-instagrams-explore-recommender-system/>.
9. Exploration and exploitation trade-offs. https://en.wikipedia.org/wiki/Multi-armed_bandit.
10. Bias in AI and recommendation systems. <https://www.searchenginejournal.com/biases-search-recommender-systems/339319/#close>.
11. Ethical concerns in recommendation systems. <https://link.springer.com/article/10.1007/s00146-020-00950-y>.
12. Seasonality in recommendation systems. https://www.computer.org/csdl/proceedings-article/big-data/2019/09_005954/1hJsfgT0qL6.
13. A multitask ranking system. <https://daiwk.github.io/assets/youtube-multitask.pdf>.
14. Benefit from a negative feedback. <https://arxiv.org/abs/1607.04228?context=cs>.

07

Event Recommendation System

In this chapter, we design an event recommendation system similar to Eventbrite's. Eventbrite is a popular event management and ticketing marketplace which allows users to create, browse, and register events. A recommendation system personalizes the experience and displays events relevant to users.

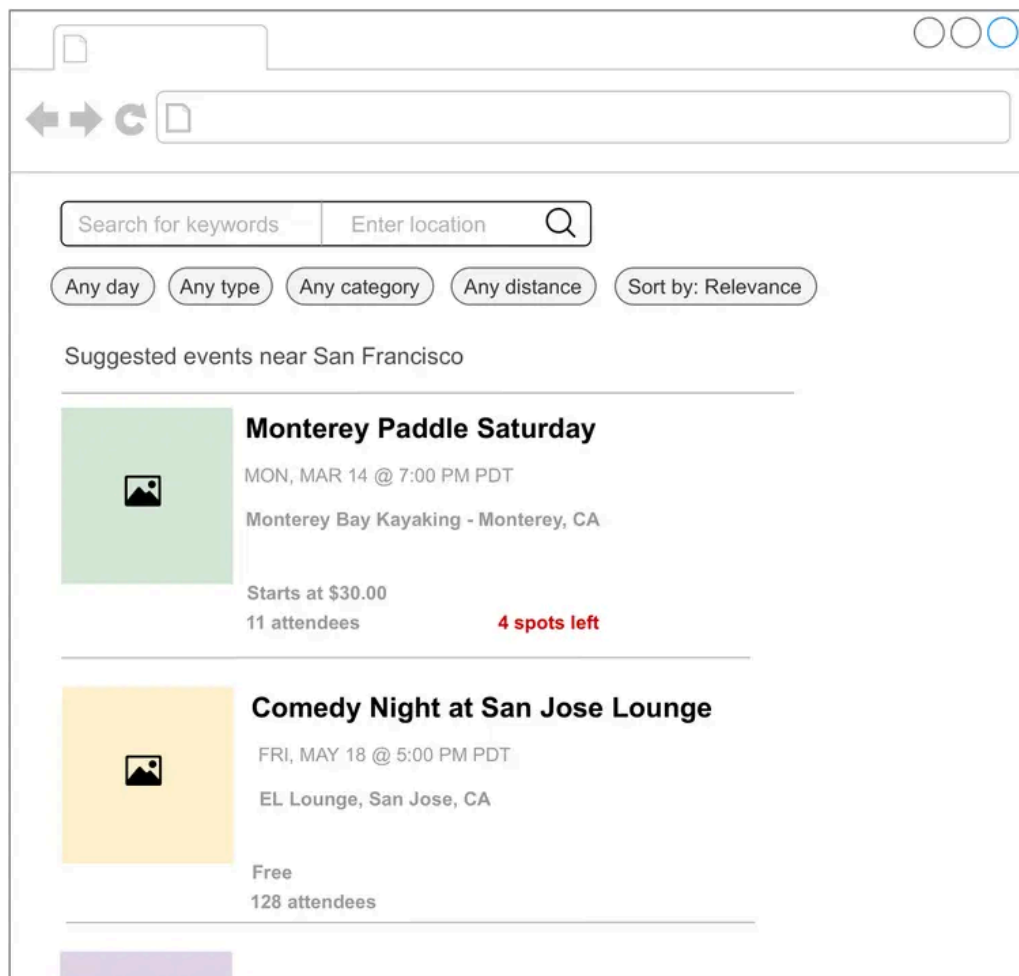


Figure 7.1: Recommended events

Clarifying Requirements

Here is a typical interaction between a candidate and an interviewer.

Candidate: What is the business objective? Can I assume the main business objective is to increase ticket sales?

Interviewer: Yes, that sounds good.

Candidate: Besides attending an event, can users book hotels or restaurants on the platform?

Interviewer: For simplicity, let's assume only events are supported.

Candidate: An event is considered an ephemeral one-time occurrence item that only happens once, and then expires. Is this assumption correct?

Interviewer: That's an excellent observation.

Candidate: What event attributes are available? Can I assume we have access to the textual description of the event, price range, location, date and time, etc.?

Interviewer: Sure, those are fair assumptions.

Candidate: Do we have any annotated data?

Interviewer: We don't have a hand-labeled dataset. You can use event and user interaction data to construct the training dataset.

Candidate: Do we have access to the user's current location?

Interviewer: Yes. Since this problem focuses on a location-based recommendation system, let's assume users agree to share their location data.

Candidate: Can users become friends on the platform? Friendship information is valuable for building a personalized event recommendation system.

Interviewer: Good question. Yes, let's assume users can form friendships on our platform. A friendship is bidirectional, meaning if A is a friend of B, then B is also a friend of A.

Candidate: Can users invite others to events?

Interviewer: Yes.

Candidate: Can a user RSVP to an event?

Interviewer: For simplicity, let's assume only a registration option is available for an event.

Candidate: Are the events free or paid?

Interviewer: We need to support both.

Candidate: How many users and events are available?

Interviewer: We host around 1 million total events every month.

Candidate: How many daily active users visit the website/app?

Interviewer: Assume we have one million unique users per day.

Candidate: Since we are building a location-based event recommendation system, it's important to calculate the distance and travel time between two locations efficiently. Can we assume external APIs such as Google Maps API or other map services can be used to obtain such data?

Interviewer: Good point. Assume we can use third-party services to obtain location data.

Let's summarize the problem statement. We are asked to design an event recommendation system, which displays a personalized list of events to users. When an event is finished, users can no longer register for it. In

addition to registering for events, users can invite others to events and form friendships. The training data should be constructed online from user interactions. The primary goal of this system is to increase total ticket sales.

Frame the Problem as an ML Task

Defining the ML objective

Based on the requirements, the business objective is to increase ticket sales. One way to translate this into a well-defined ML objective is to maximize the number of event registrations.

Specifying the system's input and output

The input to the system is a user, and the output is the top k events ranked by relevance to the user.

Choosing the right ML category

There are different ways to solve a recommendation problem:

- Simple rules, such as recommending popular events
- Embedding-based models which rely on content-based or collaborative filtering
- Reformulating it into a ranking problem

Rule-based methods are good starting points to form a baseline. However, ML-based approaches usually lead to better outcomes. In this chapter, we reformulate the task into a ranking problem and use Learning to Rank (LTR) to solve it.

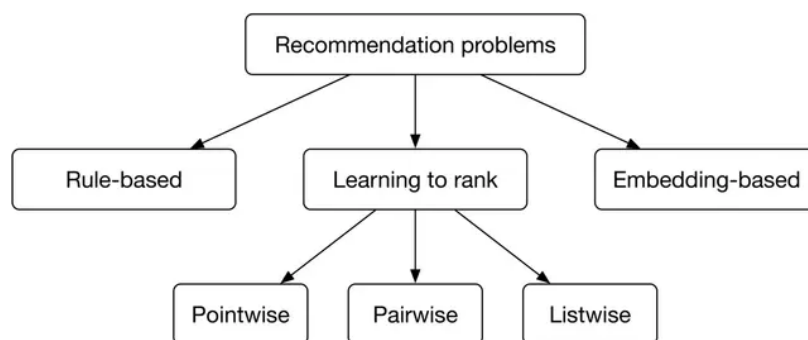


Figure 7.2: Different approaches to solving recommendation problems

LTR is a class of algorithmic techniques that apply supervised machine learning to solve ranking problems. The ranking problem can be formally defined as: "having a query and a list of items, what is the optimal ordering of the items from most relevant to least relevant to the query?" There are generally three LTR approaches: pointwise, pairwise, and listwise. Let's briefly examine each. Note that a detailed explanation of these approaches is beyond the scope of this book. If you're interested in learning more about LTR, refer to [1] .

Pointwise LTR

In this approach, we go over each item and predict the relevance between the query and the item, using classification or regression methods. Note that the score of one item is predicted independently of other items.

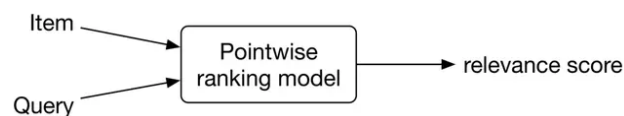


Figure 7.3: Pointwise ranking model

The final ranking is achieved by sorting the predicted relevance scores.

Pairwise LTR

In this approach, the model takes two items and predicts which item is more relevant to the query.

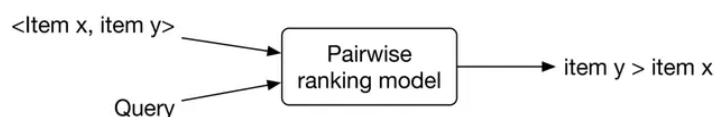


Figure 7.4: Pairwise ranking model

Some of the most popular pairwise LTR algorithms are RankNet [2], LambdaRank [3], and LambdaMART [4].

Listwise LTR

Listwise approaches predict the optimal ordering of an entire list of items, given the query.

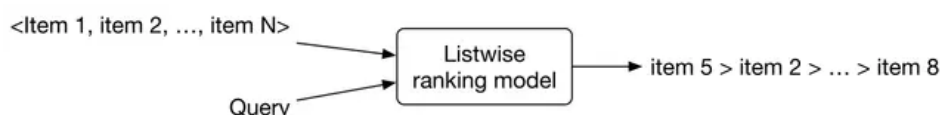


Figure 7.5: Listwise ranking model

Some popular listwise LTR algorithms are SoftRank [5], ListNet [6], and AdaRank [7].

In general, pairwise and listwise approaches produce more accurate results, but they are more difficult to implement and train. For simplicity, we use the pointwise approach for this problem. In particular, we employ a binary classification model which takes a single event at a time and predicts the probability that the user will register for it. This approach is shown in Figure 7.6.

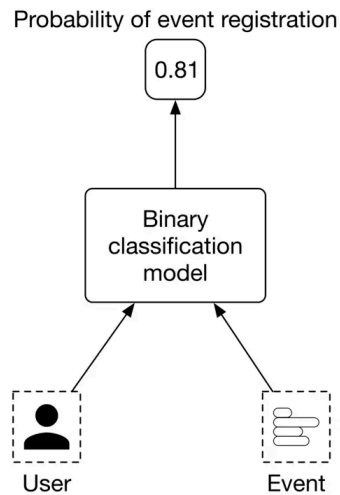


Figure 7.6: Binary classification model

Data Preparation

Data engineering

To engineer good features, we need first to understand the raw data available in the system. Since an event management platform is mainly centered around users and events, we assume the following data are available:

- Users
- Events
- Friendship
- Interactions

Users

The user data schema is shown below.

ID	Username	Age	Gender	City	Country	Language	Time zone
----	----------	-----	--------	------	---------	----------	-----------

Table 7.1: User data schema

Events

Table 7.2 shows what the event data might look like.

ID	Host User ID	Category/ Subcategory	Description	Price	Location	Date/Time
1	5	Music Concert	Dua Lipa Tour in Miami	200-900	American Airlines Arena Miami, FL	09/18/2022 19:00-24:00
2	11	Sports Basketball	Golden State Warriors vs. Milwaukee Bucks	140-2500	Chase Center SF, CA	09/22/2022 17:00-19:00
3	7	Art Theater	The Comedy and Magic of Robert Hall	Free	San Jose Improv San Jose, CA	09/06/2022 18:00-19:30

Table 7.2: Event data

Friendship

In Table 7.3, each row represents a friendship formed between two users, along with the timestamp of when it was formed

User ID 1	User ID 2	Timestamp when friendship was formed
28	3	1658451341
7	39	1659281720
11	25	1659312942

Table 7.3: Friendship data

Interactions

Table 7.4 stores user interaction data, such as event registrations, invitations, and impressions. In practice, we may store interaction data in different databases, but for simplicity, we include them in a single table.

User ID	Event ID	Interaction type	Interaction value	Location (lat, long)	Timestamp
4	18	Impression	-	38.8951 -77.0364	1658450539
4	18	Register	Confirmation number	38.8951 -77.0364	1658451341
4	18	Invite	User 9	41.9241 -89.0389	1658451365

Table 7.4: Interaction data

Feature engineering

Event-based recommendations are more challenging than traditional recommendations. An event is fundamentally different from a movie or a book, as there is no consumption after the event ends. Events are typically short-lived, meaning the time is short between event creation and when it finishes. As a result, there are not many historical interactions available for a given event. For this reason, event-based recommendations are intrinsically cold-start and suffer from a constant new-item problem.

To overcome those issues, we put more effort into feature engineering to create as many meaningful features as possible. Due to space constraints, we will only discuss some of the most important features. In practice, the number of predictive features can be much higher. In this section, we create features related to each of the following categories:

- Location-related features
- Time-related features
- Social-related features
- User-related features
- Event-related features.

Location-related features

How accessible is the event's location?

The accessibility of an event's location is an important factor. For example, if an event is high up in hills far from public transportation, the commute may discourage users from attending. Let's create the following features to capture accessibility:

- Walk score: Walk score is a number between 0 and 100, which measures how walkable an address is, based on the distance to nearby amenities. It is computed by analyzing various factors such as distance to amenities, pedestrian friendliness, population density, etc. We assume walk scores can be obtained from external data sources such as Google Maps, Open Street Map, etc. Table 7.5 shows walk scores bucketized into 5 categories.

Category	Walk score	Description
1	90-100	No car needed
2	70-89	Very walkable
3	50-69	Somewhat walkable
4	25-49	Car-dependent
5	0-24	Requires a car

Table 7.5: Walk score categories

- Walk score similarity: The difference between the event's walk score and the user's average walk score of

previous events registered by the user.

- Transit score, transit score similarity, bike score, bike score similarity.

Is the event in the same country and city as the user?

A very important deciding factor for a user is whether the event is in the same country and city where they are located. The following two features can be created:

- If the user's country is the same as the event's country, this feature is 1, otherwise 0
- If the user's city is the same as the event's city, this feature is 1, otherwise 0

Is the user comfortable with the distance?

Some users may prefer events that are very close to their location, while others prefer events that are further away. We use the following features to capture this:

- The distance between the user's location and the event's location. This value can be obtained from external APIs and bucketized into a few categories. For example:
 - 0: less than a mile
 - 1: 1-5 miles
 - 2: 5-20 miles
 - 3: 20-50 miles
 - 4: 50-100 miles
 - 5: +100 miles
- Distance similarity: Difference between the distance to an event and the average distance (in reality, the median or percentile range can be used) to events previously registered by the user.

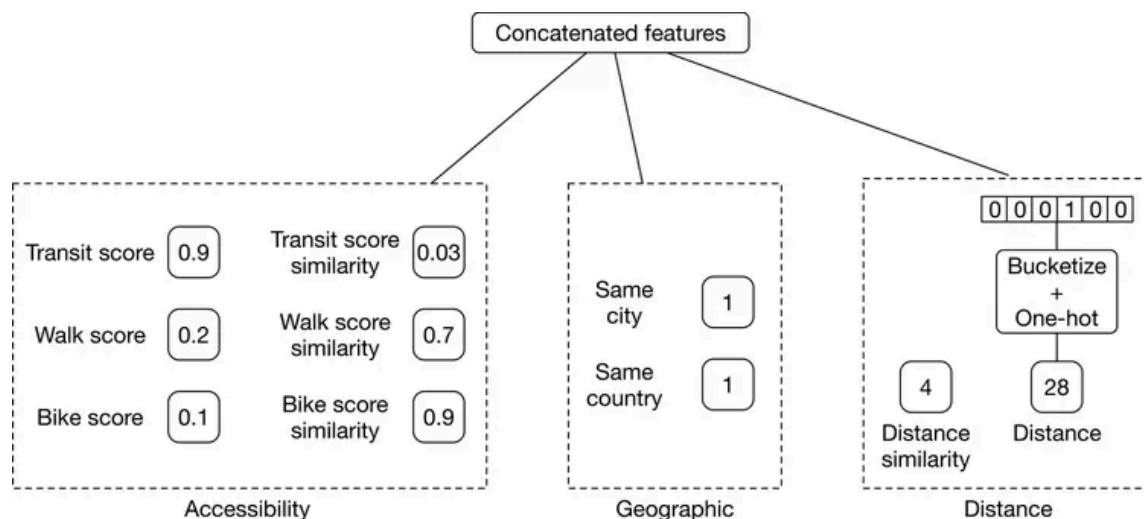


Figure 7.7: Location-related features

Time-related features

How convenient is the time remaining until an event?

Some users may plan events a few days in advance, while others don't. Let's create the following features to

capture this:

- The remaining time until the event begins. This feature can be bucketized into different categories and one-hot encoded. For example:
 - 0: less than 1 hour left until the event starts
 - 1: 1-2 hours
 - 2: 2-4 hours
 - 3: 4-6 hours
 - 4: 6-12 hours
 - 5: 12-24 hours
 - 6: 1-3 days
 - 7: 3-7 days
 - 8: +7 days
- Remaining time similarity: Difference between "remaining time" and average "remaining time" of events previously registered by the user.
- The estimated travel time from the user's location to the event's location. This value will be obtained from external services and bucketized into categories.
- Estimated travel time similarity: The difference between the estimated travel time to the event in question, and the average estimated travel time of events previously registered by the user.

Are the date and time convenient for the user?

Some users may prefer events that occur at weekends, while others prefer weekdays. Some users prefer events in the morning, while others may prefer evening events. To capture a user's historical preferences for days of the week, we create a user profile. This user profile is a vector of size 7, and each value counts the number of events the user attended on a particular day. By dividing these values by the total number of attended events, we get the historical rate of event attendance for each day of the week. Figure 7.8 shows the per-day distribution of a user's previously attended events. As we can see, this user has never attended an event on Monday or Wednesday, so displaying an event that occurs on Wednesday may not be a good recommendation for this user. Per-hour user profiles can be created using a similar approach. Similarly, we add day and hour similarity.

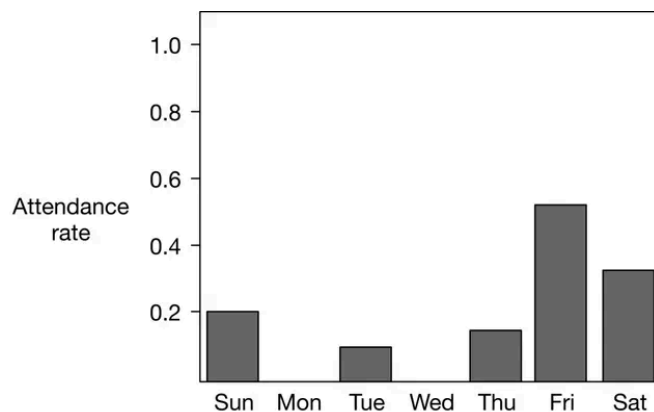


Figure 7.8: Per-day distribution of the event data

A summary of time-related features is shown in Figure 7.9

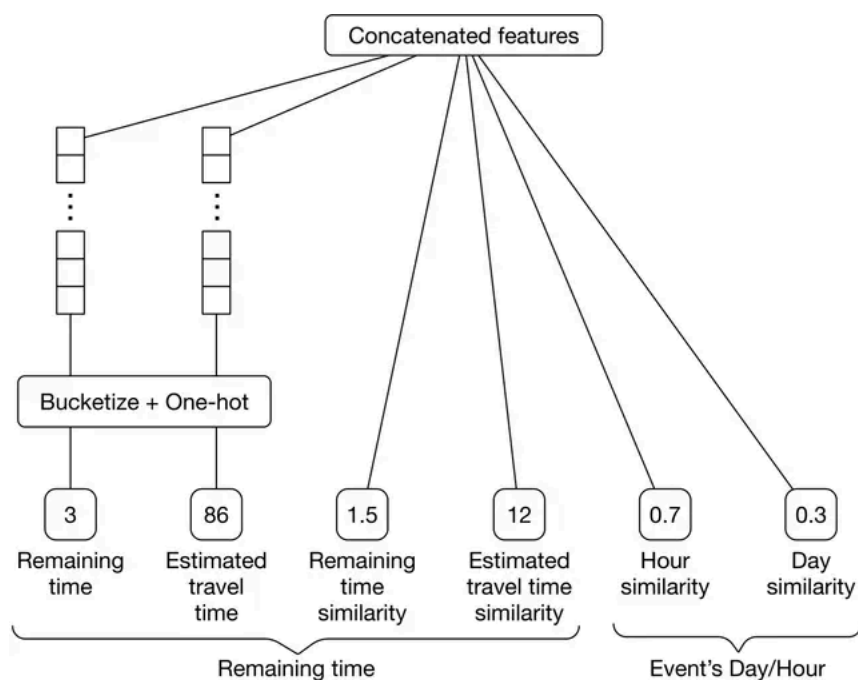


Figure 7.9: Time-related features overview

Social-related features

How many people are attending this event?

In general, users are more likely to register for an event if there are a lot of other attendees. Let's extract the following features to capture this:

- Number of users registered for this event
- The ratio of the total number of registered users to the number of impressions
- Registered user similarity: The difference between the number of registered users for the event in question and previously registered events

Features related to attendance by friends

A user is more likely to register for an event if their friends are attending it. Here are some of the features we can use:

- Number of the user's friends who registered for this event
- The ratio of the number of registered friends to the total number of friends
- Registered friend similarity: Difference between the number of registered friends for the event in question and previously registered events

Is the user invited to this event by others?

Users are more likely to attend events to which they are invited. Some features that might be helpful are:

- The number of friends who invited this user to the event

- The number of fellow users who invited this person to the event

Is the event's host a friend of the user?

Users tend to attend events created by their friends. We create a binary feature to reflect this: if the event's host is the user's friend, this value is 1, otherwise, 0.

How often has the user attended previous events created by this host?

Some users are interested in following a particular host's events.

User-related features

Age and gender

Some events are geared toward specific ages and genders. For example, "Women in Tech" and "Life lessons to excel in your 30 s" are examples of events that may be specific to certain demographic groups. We create two features to capture this:

- User's gender, encoded with one-hot encoding
- User's age, bucketized into multiple categories and encoded with one-hot encoding

Event-related features

Price of event:

The price of an event might affect the user's decision to register for it. Some features to use are:

- Event's price, bucketized into a few categories. For example:
 - 0: Free
 - 1: \$1-\$99
 - 2: \$100-\$499
 - 3: \$500-\$1,999
 - 4: +\$2,000
- **Price similarity:** Difference between the price of the event in question and the average price of events previously registered for by the user.

How similar is this event's description to previously registered descriptions?

This indicates the user's interests, based on previously registered events. For example, if the word "concert" repeatedly appears in the descriptions of previous events, it may indicate the user is interested in concert events. To capture this, we create a feature that represents the similarity between the event's description and the descriptions of previously registered events by the user. To compute the similarity, the description is converted into a numerical vector using TF-IDF, and similarity is calculated using cosine distance.

Note, this feature might be noisy as descriptions are manually provided by hosts. We can experiment by training our model with and without this feature, to measure its importance.

Figure 7.10 shows an overview of user features, event features, and social-related features.

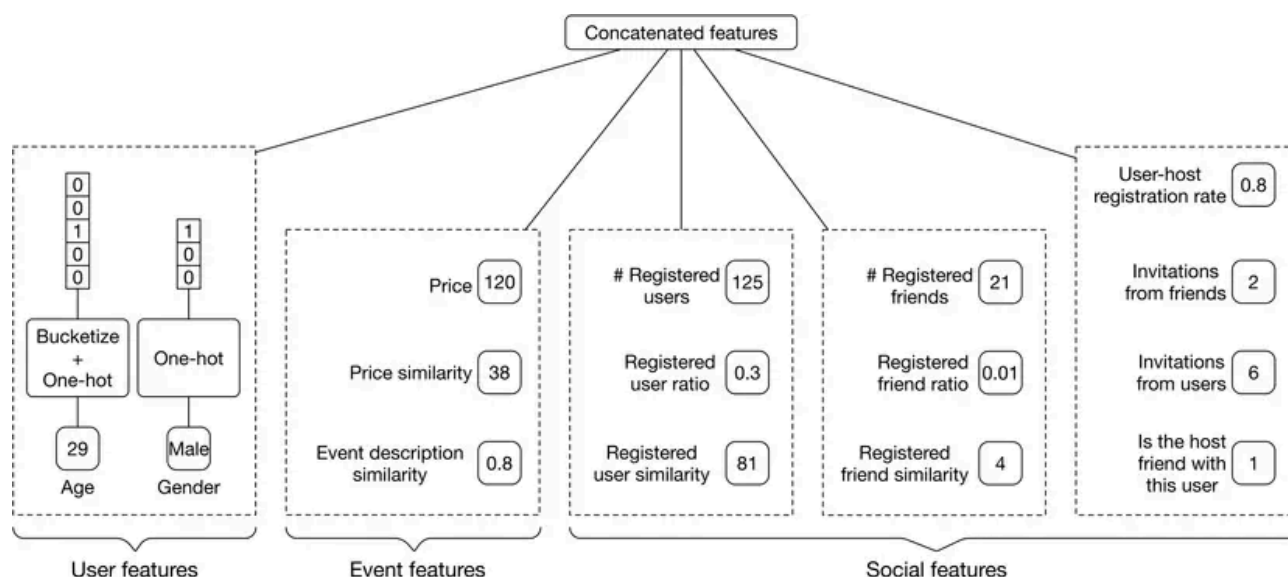


Figure 7.10: User, event, and social features

The features listed above are not exhaustive. There are lots of other predictive features that can be created in practice. For example, host-related features such as the host's popularity, user's search history, event's category, auto-generated event tags, etc. At an interview, it's not necessary to follow this section strictly. You can use it as a starting point and then discuss topics that are more relevant to the interviewer. Here are some potential talking points you might want to elaborate on:

- Batch vs. streaming features:** Batch (static) features refer to features that change less frequently, such as age, gender, and event description. These features can be computed periodically using batch processing and stored in a feature store. In contrast, streaming (aka dynamic) features change quickly. For example, the number of users registered for an event and the remaining time until an event, are dynamic features. The interviewer may want you to dive deeper into this topic and discuss batch vs. online processing in ML. If you're interested to learn more, refer to [8].
- Feature computation efficiency.** Computing features in real-time is not efficient. You may want to discuss this issue and possible ways to avoid it. For example, instead of computing the distance between the user's current location and the event's location as a feature, we can pass both locations to the model as two separate features, and rely on the model to implicitly compute useful information from the two locations. To learn more about how to prepare location data for ML models, refer to [9].
- Using a decay factor** for features that rely on the user's last X interactions. A decay factor gives more weight to the user's recent interactions/behaviors.
- Using embedding learning** to convert each event and user into an embedding vector. Those embedding vectors are used as the features representing the events and users.
- Creating features from users' attributes may create bias.** For example, relying on age or gender to decide if an applicant is a good match for a job, may lead to discrimination. Since we create features from users' attributes, it's important to be aware of potential bias issues.

Model Development

Model selection

Binary classification problems can be solved by various ML methods. Let's take a look at the following:

- Logistic regression
- Decision tree
- Gradient-boosted decision tree (GBDT)
- Neural network

Logistic regression (LR)

LR models the probability of a binary outcome by using a linear combination of one or multiple features. For the details of LR, refer to [10].

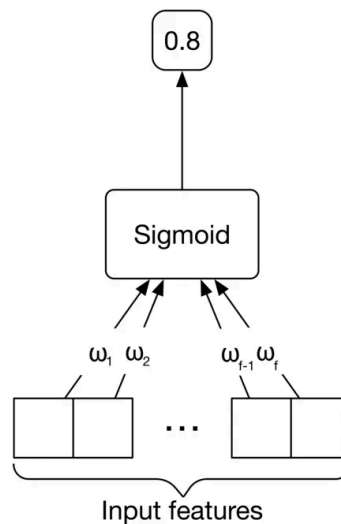


Figure 7.11: Logistic regression

Let's see the pros and cons of LR.

Pros:

- **Fast inference speed.** Computing a weighted combination of input features is fast.
- **Efficient training.** Given the simple architecture, it's easy to implement, interpret, and train quickly.
- Works well when the data is linearly separable (Figure 7.12).
- **Interpretable and easy to understand.** The weights assigned to each feature indicate the importance of different features, which gives us insight into why a decision was made.

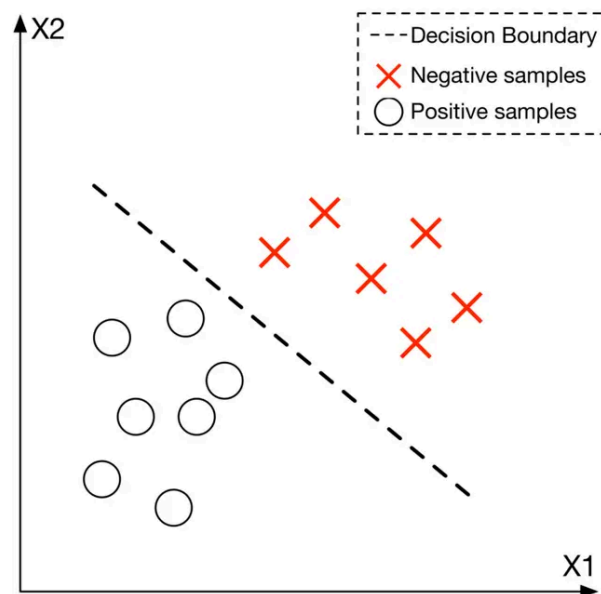


Figure 7.12: A linearly separable data with LR's decision boundary

Cons:

- **Non-linear problems can't be solved** with LR, since it uses a linear combination of input features.
- **Multicollinearity** occurs when two or more features are highly correlated. One of the known limitations of LR is that it cannot learn the task well when multicollinearity is present in the input features.

In our system, the number of input features can be very large. Often, these features have complex and non-linear relations with the target variable (binary outcome). This complexity might be hard for LR to learn.

Decision tree

Decision trees are another class of learning methods that use a tree-like model of decisions and their possible consequences to make predictions. Figure 7.13 shows a simple decision tree with two features: age and gender. It also shows the corresponding decision boundary. Each leaf node in the decision tree indicates a binary outcome where "+" indicates the given input is classified as positive, and "-" means negative. To learn more about decision trees, refer to [11].

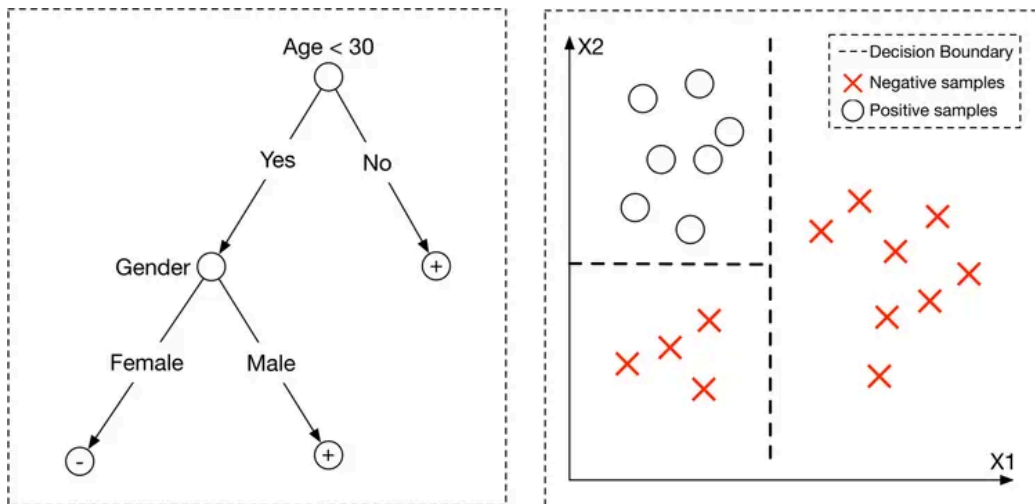


Figure 7.13: Decision tree (left) and the learned decision boundary (right)

Pros:

- **Fast training:** Decision trees are quick to train.
- **Fast inference:** Decision trees make predictions quickly at inference time.
- **Little to no data preparation:** Decision tree models don't require data normalization or scaling, since the algorithm does not depend on the distribution of the input features.
- **Interpretable and easy to understand.** Visualizing the tree provides good insights into why a decision was made and what the important decision factors are.

Cons:

- **Non-optimal decision boundary:** decision tree models produce decision boundaries that are parallel to the axes in the feature space (Figure 7.13). This may not be the optimal way to find a decision boundary for certain data distributions.
- **Overfitting:** Decision trees are very sensitive to small variations in data. A small change in input data may lead to different outcomes at serving time. Similarly, a small change in training data can lead to a totally different tree structure. This is a major issue and makes predictions less reliable.

In practice, naive decision trees are rarely used. The reason is that they are too sensitive to variations of input data. To reduce the sensitivity of decision trees, two techniques are commonly used:

- Bootstrap aggregation (Bagging)
- Boosting

These two techniques are widely used across the tech industry. It's essential to understand how they work. Let's take a closer look.

Bagging

Bagging is the ensemble learning method that trains a set of ML models in parallel, on multiple subsets of the training data. In bagging, the predictions of all these trained models are combined to make a final prediction. This

significantly reduces the model's sensitivity to the change in data (variance).

One example of bagging is the commonly used "random forest" model [12]. Random forest builds multiple decision trees in parallel during training, to reduce the model's sensitivity. To make a prediction, each decision tree independently predicts the output class (positive or negative) of the given input, and then a voting mechanism is used to combine these predictions to make a final prediction. Figure 7.14 shows a random forest with three decision trees.

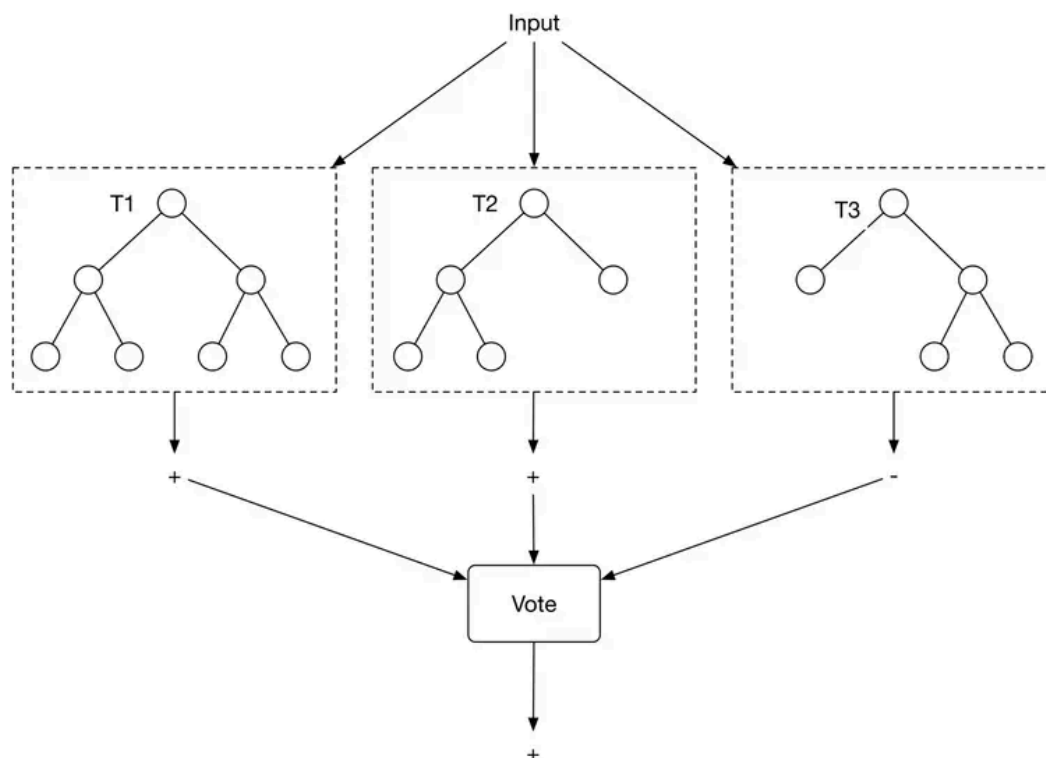


Figure 7.14: Random forest

The bagging technique has the following advantages:

- Reduces the effect of overfitting (high variance).
- Does not significantly increase training time because the decision trees can be trained in parallel.
- Does not add much latency at the inference time because decision trees can process the input in parallel.

Despite its advantages, bagging is not helpful when the model faces underfitting (high bias). To overcome bagging's drawbacks, let's discuss another technique called boosting.

Boosting

In ML, boosting involves training several weak classifiers sequentially to reduce prediction errors. The phrase "weak classifier" refers to a simple classifier that performs slightly better than random guesses. In boosting, multiple weak classifiers are converted into a single strong learning model. Figure 7.15 shows an example of boosting.

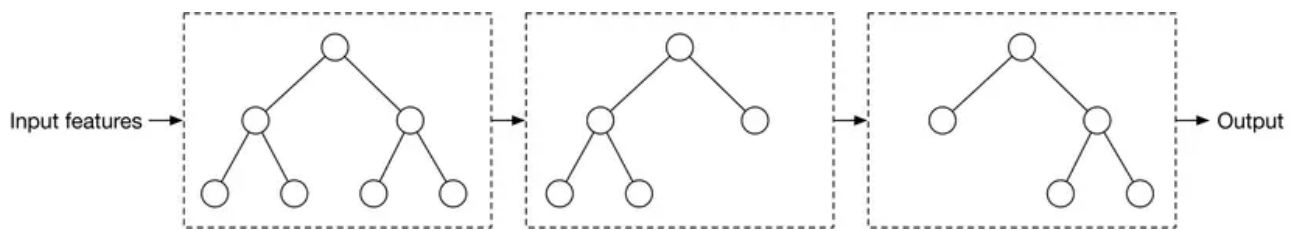


Figure 7.15: A boosting example

Pros:

- **Boosting reduces bias and variance.** Combining weak classifiers leads to a strong model less sensitive to the change in data. To learn more about bias/variance tradeoffs, refer to [13].
- **Slower training and inference.** Given the classifiers are trained based on the mistakes of the previous classifiers, they work sequentially. This adds to the serving time due to the sequential nature of boosting.

The boosting method is usually preferred over bagging in practice because bagging is not helpful in cases of bias, whereas boosting reduces the effect of both bias and variance.

Typical boosting-based decision trees are Adaboost [14], XGBoost [15], and Gradient boost [16]. They are commonly employed to train classification models.

GBDT

GBDT is a commonly used tree-based model, utilizing GradientBoost to improve decision trees. Some variants of GBDT, such as XGBoost [15], have demonstrated strong performance in various ML competitions [17]. If you're interested in learning more about GBDT, refer to [18] [19].

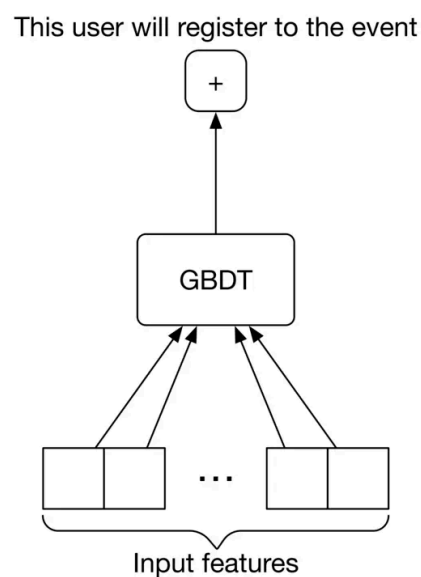


Figure 7.16: A GBDT model with a binary output

Here are the pros and cons of the GBDT model.

Pros:

- **Easy data preparation:** Similar to decision trees, it does not require data preparation.
- **Reduces variance:** GBDT reduces variance as it uses the boosting technique.
- **Reduces bias:** GBDT reduces the prediction error by leveraging several weak classifiers, iteratively improving upon the misclassified data points from the previous classifiers.
- Works well with structured data.

Cons:

- **Lots of hyperparameters to tune**, such as the number of iterations, tree depth, regularization parameters, etc.
- GBDT does not work well on unstructured data such as images, videos, audio, etc.
- **Unsuitable for continual learning** from streaming data.

In our case, since the created features are structured data, GBDT or one of its variants such as XGBoost - is a good choice to experiment with.

A major drawback of GBDT is that it is unsuitable for continual learning. In an event recommendation system, new data continuously becomes available to the system, such as recent user interactions, registrations, new events, and even new users. In addition, users' tastes and interests may change over time. It is vital for a good event recommendation system to adapt itself to new data, continuously. Without the possibility of continual learning, it is very costly to retrain GBDT from scratch regularly. Next, we explore neural networks which overcome this limitation.

Neural network (NN)

In an event recommendation system, we have many features that might not correlate linearly with the outcome. Learning these complex relationships is difficult. In addition, continual learning is necessary for adapting the model to new data.

NNs are great at solving those challenges. They are capable of learning complex tasks with non-linear decision boundaries. Additionally, NN models can be fine-tuned on new data very easily, making them ideal for continual learning. If you are unfamiliar with the details of NNs, you are encouraged to read [20].

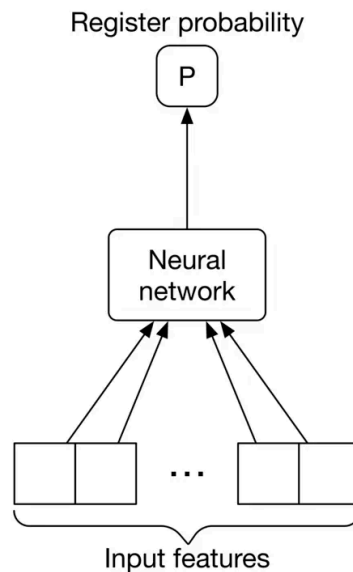


Figure 7.17: Neural network input-output

Let's see its pros and cons.

Pros

- **Continual learning:** NNs are designed to learn from data and improve themselves continually.
- Works well with unstructured data such as text, image, video, or audio.
- **Expressiveness:** NNs have expressive power due to their high number of learning parameters. They can learn very complex tasks and non-linear decision boundaries.

Cons

- **Computationally expensive** to train.
- **The quality of input data strongly influences the outcome:** NNs are sensitive to input data. For example, if input features are in very different ranges, the model may converge slowly during the training phase. An important step for NNs is data preparation, such as normalization, log-scaling, one-hot encoding, etc.
- **Large training data** is required to train NNs.
- **Black-box nature:** NNs are not interpretable, meaning it's not easy to understand the influence of each feature upon the outcome, as the input features go through multiple layers of non-linear transformations.

Which model should we select?

Picking the right model is challenging. We often need to experiment with different models to determine which works best. We can choose the right model based on various factors:

- Complexity of the task
- Data distribution and data type
- Product requirements or constraints, such as training cost, speed, model size, etc In this problem, both GBDTs and NNs are good candidates for experimentation. We start with the GBDT variant, XGBoost, since it is fast to implement and train. The result can be used as an initial baseline.

Once we have a baseline, we explore the possibility of building a better model with NNs. Neural networks are expected to work well here for the following reasons:

- Massive training data is available in our system. Users continuously interact with the system by registering for events, inviting friends, publishing new events, etc. Given the number of users, this creates a massive amount of data available for training.
- Data may not be linearly separable, and neural networks can learn non-linear data.

When designing a NN architecture, several hyperparameters must be considered, including the number of hidden layers, neurons in each layer, activation function, etc. These can be determined by employing hyperparameter tuning techniques. NN architectural details are not typically the main focus of ML system design interviews, since there is no systematic way to choose the right architecture.

Model training

Constructing the dataset

Building training and evaluation datasets is an essential step in developing a model. For example, let's look at how we compute features and their labels.

To construct a single data point, we extract a $\langle \text{user}, \text{event} \rangle$ pair from the interaction data and compute the input features from the pair. We then label the data point with 1 if the user has registered for the event, and 0 if not.

#	Extracted $\langle \text{user}, \text{event} \rangle$ features							Label
1		1	0	1	1	0	1	1
2		0	0	0	1	1	0	0

Figure 7.18: Constructed dataset

One issue we may face after constructing the dataset is class imbalance. The reason is that users may explore tens or hundreds of events before registering for one. Therefore, the number of negative $\langle \text{user}, \text{event} \rangle$ pairs is significantly higher than positive data points. We can use one of the following techniques to address the class imbalance issue:

- Use focal loss or class-balanced loss to train the classifier
- Undersample the majority class

Choosing the loss function

Since the model is a binary classification model, we use a typical classification loss function such as binary cross-entropy to optimize the neural network model.

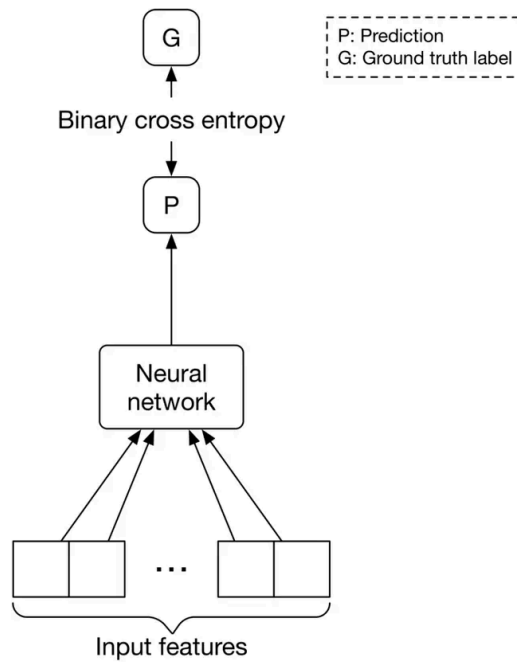


Figure 7.19: Loss between the prediction and the label

Evaluation

Offline metrics

To evaluate the ranking system, we consider the following options.

Recall@k or Precision@k. These metrics are not good fits because they do not consider the ranking quality of the output.

MRR, nDCG, or mAP. These three metrics are commonly used to measure ranking quality. But which one is best?

MRR focuses on the rank of the first relevant item in the list, which is suitable in systems where only one relevant item is expected to be retrieved. However, in an event recommendation system, several recommended events may be relevant to the user. MRR is not a good fit.

nDCG works well when the relevance score between a user and an item is non-binary. In contrast, mAP works only when the relevance scores are binary. Since events are either relevant (a user registered for it) or irrelevant (a user saw the event but did not register), mAP is a better fit.

Online metrics

In our case, the business objective is to increase revenue by increasing ticket sales. To measure the impact of the system on revenue, let's explore the following metrics:

- Click-through rate (CTR)
- Conversion rate
- Bookmark rate
- Revenue lift

CTR. A ratio showing how often users who see recommended events go on to click on an event.

$$CTR = \frac{\text{total number of clicked events}}{\text{total number of impressions}}$$

A high CTR shows our system is good at recommending events that users click on. Having more clicks generally means more event registrations.

However, relying only on CTR as the online metric may be insufficient. Some events are clickbait. Ideally, we would like to measure how relevant recommended events are for the user. This metric is called the conversion rate, which we discuss now.

Conversion rate. A ratio showing how often users who see recommended events go on to register for them. The formula is:

$$\text{Conversion rate} = \frac{\text{total number of event registrations}}{\text{total number of impressions}}$$

A high conversion rate indicates users register for recommended events more often. For example, a conversion rate of 0.3 means that users, on average, register for 3 events out of every 10 recommended events.

Bookmark rate. A ratio showing how often users bookmark recommended events. This is based on the assumption that the platform allows users to save or bookmark an event.

Revenue lift. This is the increase in revenue as a result of event recommendations.

Serving

In this section, we propose an ML system design that can be used to serve requests. As Figure 7.20 shows, there are two main pipelines in the design:

- Online learning pipeline
- Prediction pipeline

Figure 7.20: ML system design

Online learning pipeline

As described earlier, event recommendations are intrinsically cold-start and suffer from a constant new-item problem. Consequently, the model must be continuously fine-tuned to adapt to new data. This pipeline is

responsible for continuously training new models by incorporating new data, evaluating the trained models, and deploying them.

Prediction pipeline

The prediction pipeline is responsible for predicting the top k most relevant events to a given user. Let's discuss some of the most important components of the prediction pipeline.

Event filtering

The event filtering component takes the query user as input and narrows down the events from 1 million to a small subset of events. This is based upon simple rules, such as event locations, or other types of user filters. For example, if a user adds a “concerts only” filter, the component quickly narrows down the list to a subset of candidate events. Since these types of filters are common in event recommendation systems, they can be used to significantly reduce our search space from potentially millions of events, to hundreds of candidate events.

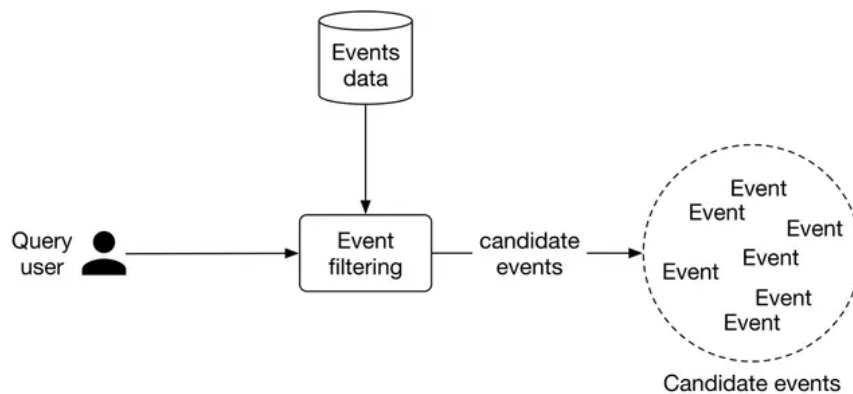


Figure 7.21: Event filtering input-output

Ranking service

This service takes the user and candidate events produced by the filtering component as input, computes features for each $\langle \text{user}, \text{event} \rangle$ pair, sorts the events based on the probabilities predicted by the model, and outputs a ranked list of top k most relevant events to the user.

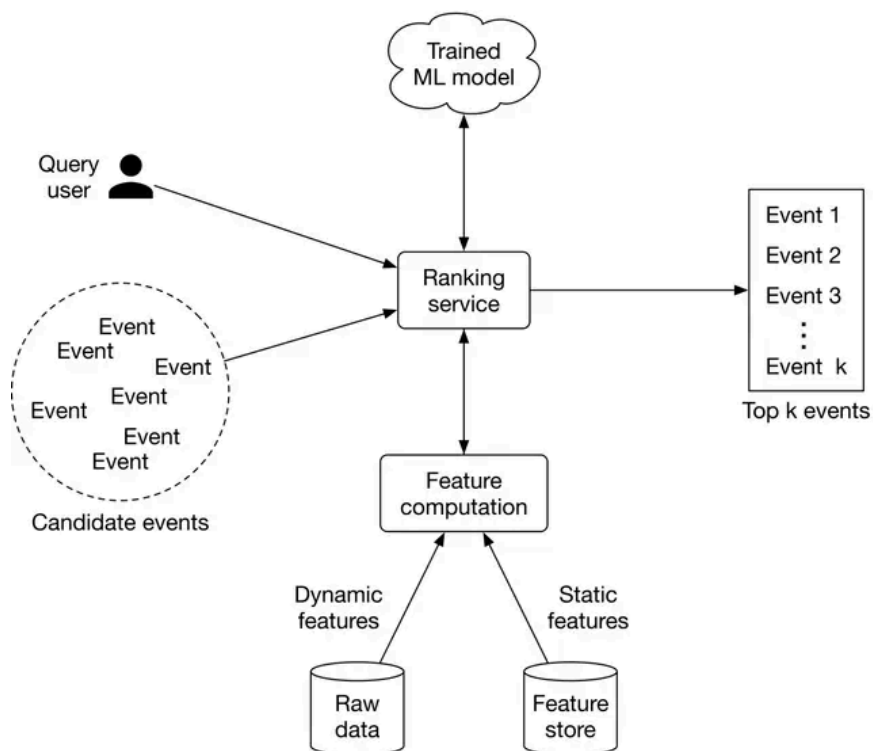


Figure 7.22: Ranking service workflow

Ranking service interacts with the feature computation component responsible for computing features that the model expects. Static features are obtained from a feature store, while dynamic features are computed in real-time from the raw data.

Other Talking Points

If there is extra time at the end of the interview, here are some additional talking points:

- What are the different types of bias we may observe in this system [21].
- How to utilize feature crossing to achieve more expressiveness [22].
- Some users like to see a diverse list of events. How to ensure the recommended events are diverse and fresh [23]?
- We utilize the user's attributes to train a model. We also rely on users' live locations. What are additional considerations related to privacy and security [24]?
- Event management platforms are usually two-sided marketplaces, where event hosts are the suppliers and users fulfill the demand side. How to ensure the system is not optimized for one side only? Additionally, how to keep the platform fair for different hosts? To learn more about unique challenges in two-sided marketplaces, refer to [25].
- How to avoid data leakage when constructing the dataset [26].
- How to determine the right frequency to update the models [27].

References

1. Learning to rank methods. <https://livebook.manning.com/book/practical-recommender-systems/chapter-13/53>.
2. RankNet paper. https://icml.cc/2015/wp-content/uploads/2015/06/icml_ranking.pdf.
3. LambdaRank paper. <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/lambdarank.pdf>.
4. LambdaMART paper. <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/MSR-TR-2010-82.pdf>.
5. SoftRank paper. <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/SoftRankWsdm08Submitted.pdf>.
6. ListNet paper. <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2007-40.pdf>.
7. AdaRank paper. <https://dl.acm.org/doi/10.1145/1277741.1277809>.
8. Batch processing vs stream processing. <https://www.confluent.io/learn/batch-vs-real-time-data-processing/#:~:text=Batch%20processing%20is%20when%20the,data%20flows%20through%20a%20system>.
9. Leveraging location data in ML systems. <https://towardsdatascience.com/leveraging-geolocation-data-for-machine-learning-essential-techniques-192ce3a969bc#:~:text=Location%20data%20is%20an%20important,base,d%20on%20your%20customer%20data>.
10. Logistic regression. <https://www.youtube.com/watch?v=yIYKR4sgzI8>.
11. Decision tree. <https://careerfoundry.com/en/blog/data-analytics/what-is-a-decision-tree/>.
12. Random forests. https://en.wikipedia.org/wiki/Random_forest.
13. Bias/variance trade-off. <http://www.cs.cornell.edu/courses/cs578/2005fa/CS578.bagging.boosting.lecture.pdf>.
14. AdaBoost. <https://en.wikipedia.org/wiki/AdaBoost>.
15. XGBoost. <https://xgboost.readthedocs.io/en/stable/>.
16. Gradient boosting. <https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/>.
17. XGBoost in Kaggle competitions. <https://www.kaggle.com/getting-started/145362>.
18. GBDT. <https://blog.paperspace.com/gradient-boosting-for-classification/>.
19. An introduction to GBDT. <https://www.machinelearningplus.com/machine-learning/an-introduction-to-gradient-boosting-decision-trees/>.

20. Introduction to neural networks. <https://www.youtube.com/watch?v=i2fmaabls5w>.
21. Bias issues and solutions in recommendation systems. <https://www.youtube.com/watch?v=pPq9iyGIZZ8>.
22. Feature crossing to encode non-linearity. <https://developers.google.com/machine-learning/crash-course/feature-crosses/encoding-nonlinearity>.
23. Freshness and diversity in recommendation systems. <https://developers.google.com/machine-learning/recommendation/dnn/re-ranking>.
24. Privacy and security in ML. <https://www.microsoft.com/en-us/research/blog/privacy-preserving-machine-learning-maintaining-confidentiality-and-preserving-trust/>.
25. Two-sides marketplace unique challenges. <https://www.uber.com/blog/uber-eats-recommending-marketplace/>.
26. Data leakage. <https://machinelearningmastery.com/data-leakage-machine-learning/>.
27. Online training frequency. <https://huyenchip.com/2022/01/02/real-time-machine-learning-challenges-and-solutions.html#towards-continual-learning>.

08

Ad Click Prediction on Social Platforms

Introduction

Online advertising allows advertisers to bid and place their advertisements (ads) on a platform for measurable responses such as impressions, clicks, and conversions. Displaying relevant ads to users is a fundamental for many online platforms such as Google, Facebook, and Instagram.

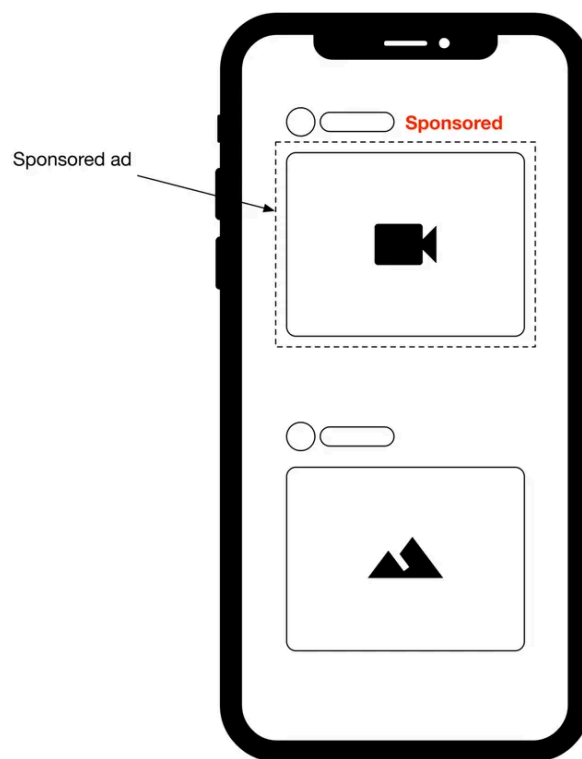


Figure 8.1: Sponsored ads placed on the user's timeline

In this chapter, we design an (also known as) system similar to what popular social media platforms use.

Clarifying Requirements

Here is a typical interaction between a candidate and an interviewer.

Candidate: Can I assume the business objective of building an ad prediction system is to maximize revenue?

Interviewer: Yes, that's correct.

Candidate: There are different types of ads, such as video and image ads. In addition, ads can be displayed in different sizes and formats, like users' timelines, pop-up ads, etc. For simplicity, can I assume ads are placed on

users' timelines only, and every click generates the same revenue?

Interviewer: That sounds good.

Candidate: Can the system show the same ad to the same user more than once?

Interviewer: Yes, we can show an ad more than once. Sometimes, an ad turns into a click after multiple impressions. In reality, companies have a "fatigue period", that is, they don't show the same ad to the same user for X days if the user repeatedly ignores it. For simplicity, assume we have no fatigue period.

Candidate: Do we support the "hide this ad" feature? How about "block this advertiser"? These kinds of negative feedback help us to detect irrelevant ads.

Interviewer: Good question. Let's assume users can hide an ad they don't like. "Block this advertiser" is an interesting feature, but we don't need to support it for now.

Candidate: Would it be okay to assume that the training dataset should be constructed using user and ad data, and the labels should be based on user-ad interactions?

Interviewer: Sure.

Candidate: We can construct positive training data points via user clicks, but how do we generate negative data points? Can we assume any impression that is not clicked is a negative data point? What if the user scrolls fast and doesn't spend time seeing the ad? What if we count an impression as negative, but eventually, the user clicks on it?

Interviewer: These are excellent questions. What are your thoughts?

Candidate: If an ad is visible on a user's screen for a certain duration but not clicked, we can count it as a negative data point. An alternative approach would be to assume impressions are negative until a click is observed. In addition, we can rely on negative feedback such as "hide this ad" to label negative data points.

Interviewer: Makes sense! In practice, we might use other complex techniques to label negative data points. For this interview, let's proceed with your suggestions.

Candidate: In ad click prediction systems, it's critical for the model to learn from new interactions continuously. Is it fair to assume continual learning is a necessity here?

Interviewer: Great point. Experiments have shown that even a 5-minute delay in updating models can damage performance [1].

Let's summarize the problem statement. We are asked to design an ad click prediction system. The business objective of the system is to maximize revenue. The ads are placed only on users' timelines, and each click generates the same revenue. It is necessary to train the model on new interactions continually. We construct the dataset from the user and ad data, and label them based on interactions. In this chapter, we will not discuss AdTech-specific topics as they are not relevant to ML interviews. To learn more about AdTech, refer to [2].

Frame the Problem as an ML Task

Defining the ML objective

The goal of the ad click prediction system is to increase revenue by showing users ads they are more likely to click on. This can be converted into the following ML objective: predicting if an ad will be clicked. This is due to the

fact that by correctly predicting click probabilities, the system can display relevant ads to users, which leads to an increase in revenue.

Specifying the system's input and output

The ad click prediction system takes a user as input, and outputs a ranked list of ads based on click probabilities.

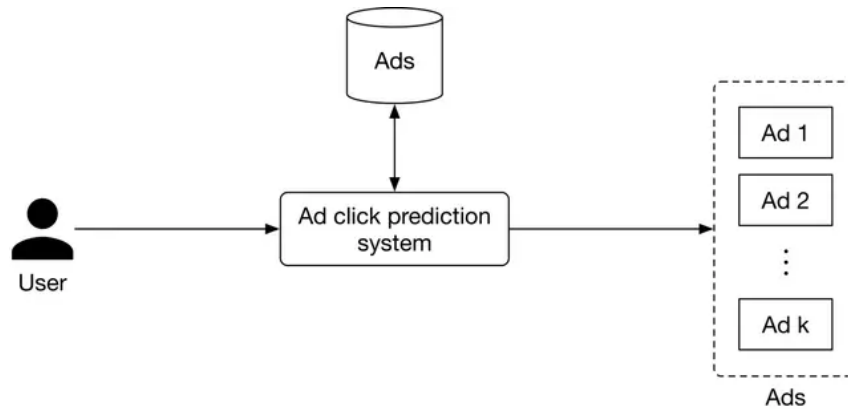


Figure 8.2: Ad click prediction system's input and output

Choosing the right ML category

Figure 8.2 illustrates how ad prediction can be framed as a ranking problem. As described in Chapter 7, Event Recommendation System, the pointwise Learning to Rank (LTR) is a great starting point for solving ranking problems. The pointwise LTR employs a binary classification model that takes a $\langle \text{user}, \text{ad} \rangle$ pair as input and predicts whether the user will click on the ad. Figure 8.3 shows the model's input and output.

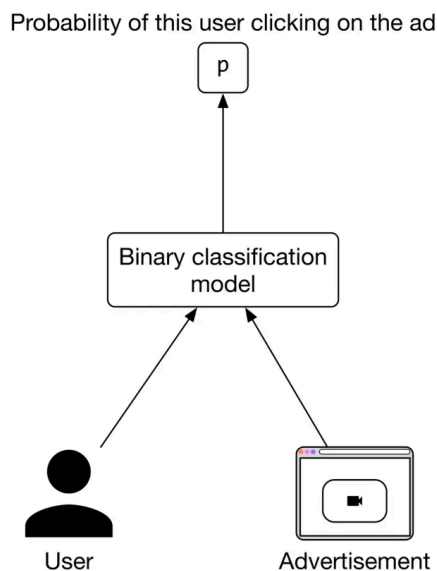


Figure 8.3: Binary classification model's input-output

Data Preparation

Data engineering

Here is some raw data available in this system:

- Ads
- Users
- User-ad interactions

Ads

Ad data is shown in Table 8.1. In practice, we may have hundreds of attributes associated with each ad. For simplicity, we only list important ones.

Ad ID	Advertiser ID	Ad group ID	Campaign ID	Category	Subcategory	Images or Videos
1	1	4	7	travel	hotel	http: //cdn.mysite.com/u1.jpg
2	7	2	9	insurance	car	http: //cdn.mysite.com/t3.mp4
3	9	6	28	travel	airline	http: //cdn.mysite.com/t5.jpg

Table 8.1: Ad data

Users

The schema for user data is shown below.

ID	Username	Age	Gender	City	Country	Language	Time zone
----	----------	-----	--------	------	---------	----------	-----------

Table 8.2: Schema for user data

User-ad interactions

This table stores user-ad interactions such as impressions, clicks, and conversions.

User ID	Ad ID	Interaction type	Dwell time	Location (lat, long)	Timestamp
11	6	Impression	5sec	38.8951 -77.0364	165845053
11	7	Impression	0.4 sec	41.9241 -89.0389	1658451365
4	20	Click	-	22.7531 47.9642	1658435948
11	6	Conversion	-	22.7531 47.9642	1658451849

Table 8.3: User-ad interaction data

Feature engineering

Our aim in this section is to engineer features that will assist us in predicting user clicks.

Ad features

Ad features include the following:

- IDs
- Image/video
- Category and subcategory
- Impression and click numbers

Let's examine each in more detail.

IDs

These are advertiser ID, campaign ID, ad group ID, ad ID, etc.

Why is it important? The IDs represent the advertiser, the campaign, the ad group, and the ad itself. These IDs are used as predictive features to capture the unique characteristics of different advertisers, campaigns, ad groups, and ads.

How to prepare it? The embedding layer converts sparse features, such as IDs, into dense feature vectors. Each ID type has its own embedding layer.

Image/video

Why is it important? A video or image in a post is another signal that can help us predict what the ad is about. For example, an image of an airplane may indicate the ad is related to travel.

How to prepare it? The images or videos are first preprocessed. After that, we use a pre-trained model such as SimCLR [3] to convert unstructured data into a feature vector.

Ad category and subcategory

As provided by the advertiser, this is the ad's category and subcategory. For example, here is a list of broad types of categories that are targetable: Arts & Entertainment, Autos & Vehicles, Beauty & Fitness, etc.

Why is it important? It helps the model to understand which category the ad belongs to.

How to prepare it? These are manually provided by the advertiser based on a predefined list of categories and subcategories. To learn more about preparing textual data, read Chapter 4, YouTube Video Search.

Impressions and click numbers

- Total impression/clicks on the ad
- Total impressions/clicks on ads supplied by an advertiser
- Total impressions of the campaign

Why is it important? These numbers indicate how other users reacted to this ad. For example, users are more

likely to click on an ad with a high click-through rate (CTR).

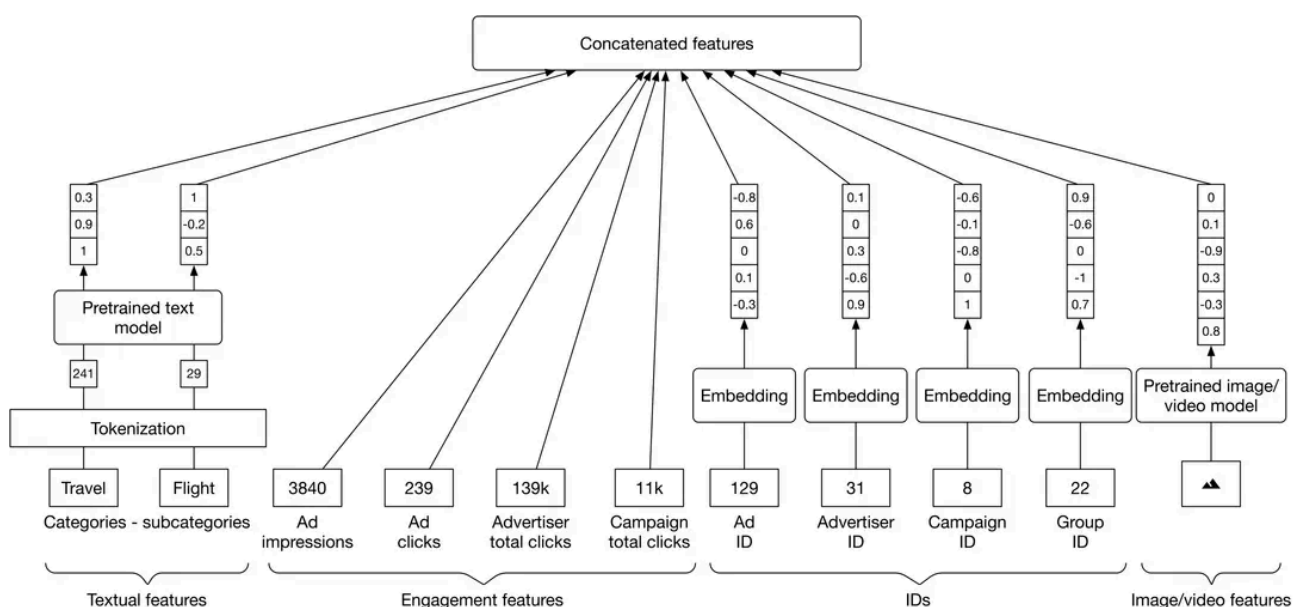


Figure 8.4: Summary of ad-related feature preparation

User features

Similar to previous chapters, we choose the following features:

- Demographics: age, gender, city, country, etc
- Contextual information: device, time of the day, etc
- Interaction-related features: clicked ads, user's historical engagement statistics, etc

Let's take a closer look at interaction-related features.

Clicked ads

Ads previously clicked by the user.

Why is it important? Previous clicks indicate a user's interests. For example, when a user clicks on lots of insurance-related ads, it suggests they are likely to click on a similar ad again.

How to prepare it? In the same way as described in "Ad features".

User's historical engagement statistics

These are the user's historical engagement numbers, such as their total ad views and ad click rate.

Why is it important? An individual's historical engagement is a good predictor of future engagement. In general, users are more likely to click on ads in the future, if they clicked on ads frequently in the past.

How to prepare it? Engagement statistics are represented as numerical values. To prepare them, we scale their values into a similar range.

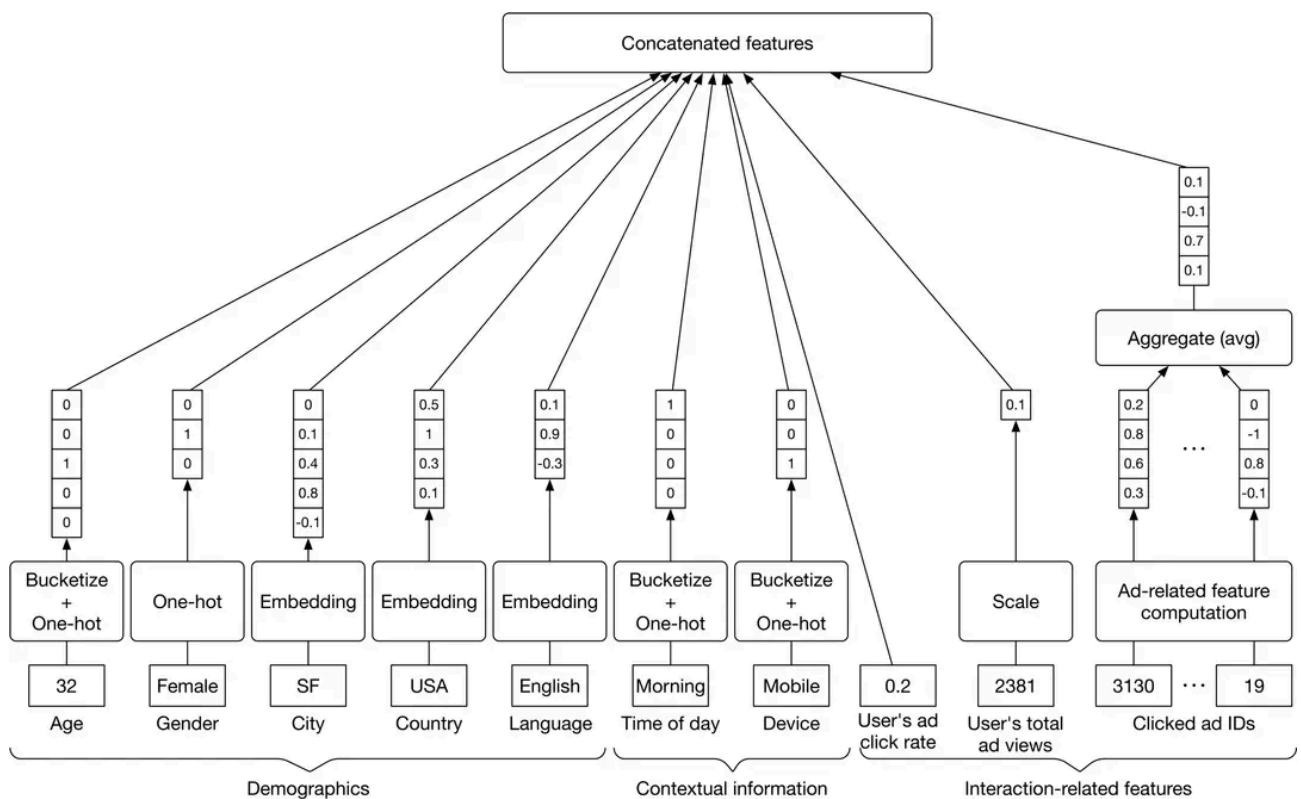


Figure 8.5: Feature preparation for user metadata and interactions

Before concluding the data preparation section, let's examine a common challenge in ad click prediction systems. In most cases, these systems deal with lots of high cardinality categorical features. For example, "ad category" takes values from a huge list of all possible categories. Similarly, "advertiser ID" and "user ID" take potentially millions of unique values, depending on how many users or advertisers are active on the platform. Given the huge feature space which often exists, it is common to have thousands or millions of features mostly filled with zeroes.

In the model selection section, we will cover techniques to overcome these unique challenges.

Model Development

Model selection

As described in the section "Frame the Problem as an ML Task", the binary classification model is chosen to solve the ranking problem. Binary classifications can be modeled in several different ways. The following are common choices in ad click prediction systems:

- Logistic regression
- Feature crossing + logistic regression
- Gradient boosted decision trees
- Gradient boosted decision trees + logistic regression
- Neural networks
- Deep & Cross networks
- Factorization Machines
- Deep Factorization Machines

Logistic regression (LR)

LR models the probability of a binary outcome using a linear combination of one or multiple features. LR is fast to train and easy to implement. An LR-based ad click prediction system, however, does have the following drawbacks:

- **Non-linear problems can't be solved with LR.** LR solves the task using a linear combination of input features, which leads to a linear decision boundary. In ad click prediction systems, data is usually not linearly separable, so LR may perform poorly.
- **Inability to capture feature interactions.** LR is not capable of capturing feature interactions. In ad prediction systems, it is very common to have various interactions between features. When features interact with each other, the output probability cannot be expressed as the sum of the feature effects, since the effect of one feature depends on the value of the other feature.

Given these two drawbacks, LR is not the best choice for the ad prediction system. However, because it is fast to implement and easy to train, many companies use it to create a baseline model.

Feature crossing + LR

To capture feature interactions better, we use a technique called feature crossing

What is feature crossing?

Feature crossing is a technique used in ML to create new features from existing features. It involves combining two or more existing features into one new feature by taking their product, sum, or another combination. It is possible to capture nonlinear interactions between the original features in this way, which can improve the performance of ML models. For example, interactions such as "young and basketball" or "USA and football" may positively impact a model's ability to predict click probability.

How to create feature crosses?

In feature crossing, we manually add new features to the existing features based on prior knowledge. As Figure 8.6 shows, crossing two features, such as "country" and "language" adds six new features to the existing feature space. To learn more about crossing, refer to [4].

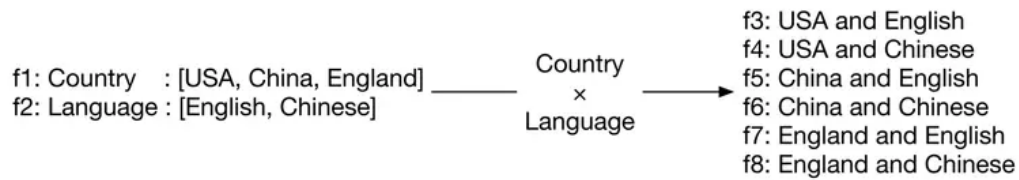


Figure 8.6: Crossing two features: country and language

How to use feature crossing + LR?

As shown in Figure 8.7, feature crossing + LR works as follows:

1. Use feature crossing on the original set of features to extract new features (crossed features)
2. Use the original and the crossed features as input for the LR model

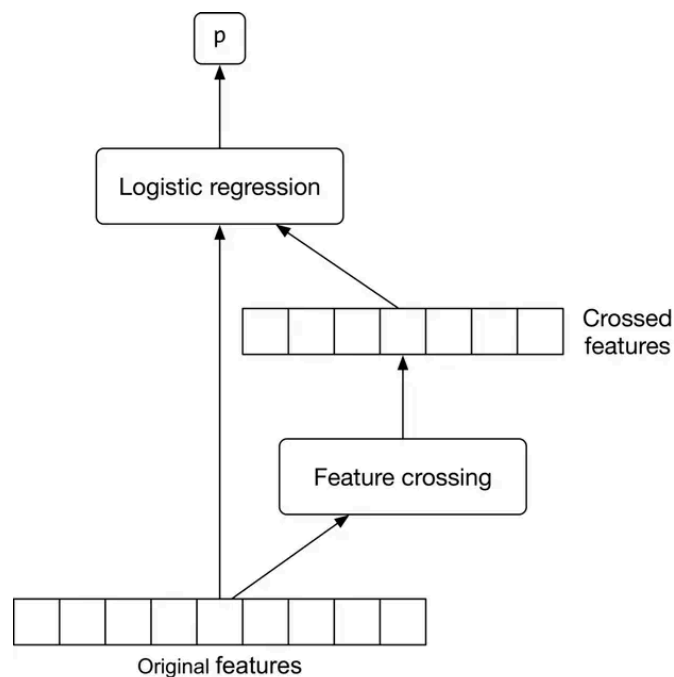


Figure 8.7: Feature crossing performed on original features

This method allows the model to capture certain pairwise (second-order) feature interactions. However, it has three shortcomings:

- **Manual process:** Human involvement is required to choose features to be crossed, which is time-consuming and expensive.
- **Requires domain knowledge:** Feature crossing requires domain expertise. To determine which interactions

between features are predictive signals for the model, we need to understand the problem and the feature space in advance.

- **Cannot capture complex interactions:** Crossed features may not be sufficient to capture all the complex interactions from thousands of sparse features.
- **Sparsity:** The original features can be sparse. With feature crossing, the cardinality of the crossed features can become much larger, leading to more sparsity.

Given the drawbacks, this method is not an ideal solution for the ad prediction system.

Gradient-boosted decision trees (GBDT)

We examined GBDT in Chapter 7, Event Recommendation System. Here, we will only explore the pros and cons of GBDT when applied to the ad click prediction system.

Pros

- GBDT is interpretable and easy to understand

Cons

- **Inefficient for continual learning.** In ad click prediction systems, we continuously collect new data such as user, ad, and interaction data. To continually train a model on new data, we generally have two options: 1) training from scratch, or 2) fine-tuning the model on new data. GBDT is not designed to be fine-tuned with new data. So we usually need to train the model from scratch, which is inefficient at a large scale.
- **Cannot train embedding layers.** In ad prediction systems, it's common to have many sparse categorical features, and the embedding layer is an effective way to represent these features. However, GBDT cannot benefit from embedding layers.

GBDT + LR

There are two steps in this approach:

1. Train the GBDT model to learn the task.
2. Instead of using the trained model to make predictions, use it to select and extract new predictive features. The newly generated features and the original features are used as input into the LR model for predicting clicks.

Use GBDT for feature selection Feature selection is intended to reduce the number of input features to only those most useful and informative. Using decision trees, we can select a subset of features based on their importance. To better understand how decision trees are used for feature generation, refer to [5].

Use GBDT for feature extraction The purpose of feature extraction is to reduce the number of features by creating new features from existing ones. The newly extracted features are expected to have better predictive power. Figure 8.8 explains how to extract features using GBDT.

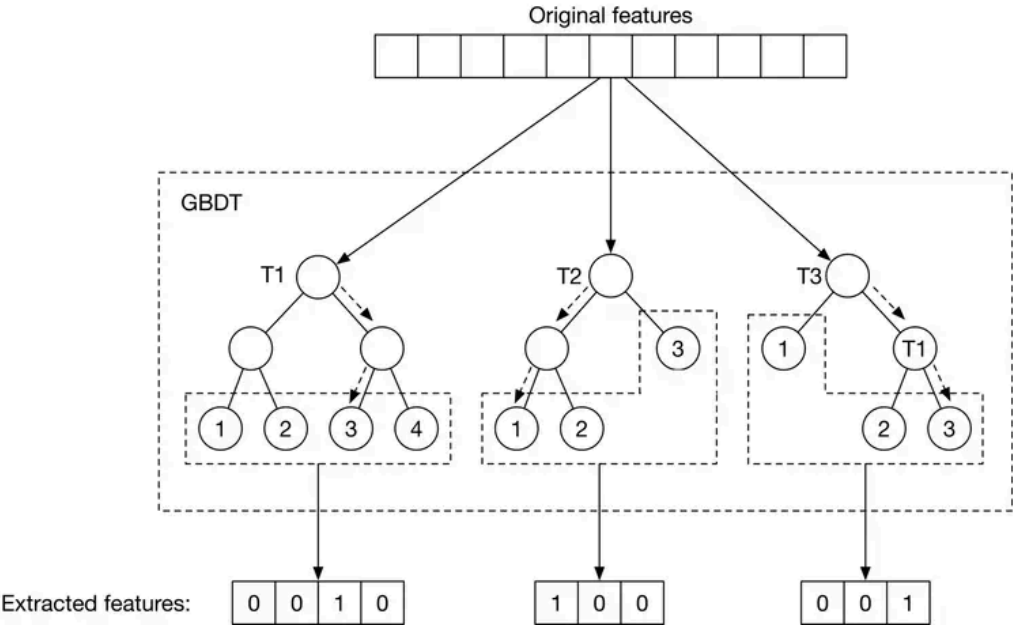


Figure 8.8: Use GBDT for feature extraction

An overview of GBDT in use followed by LR, is shown in Figure 8.9.

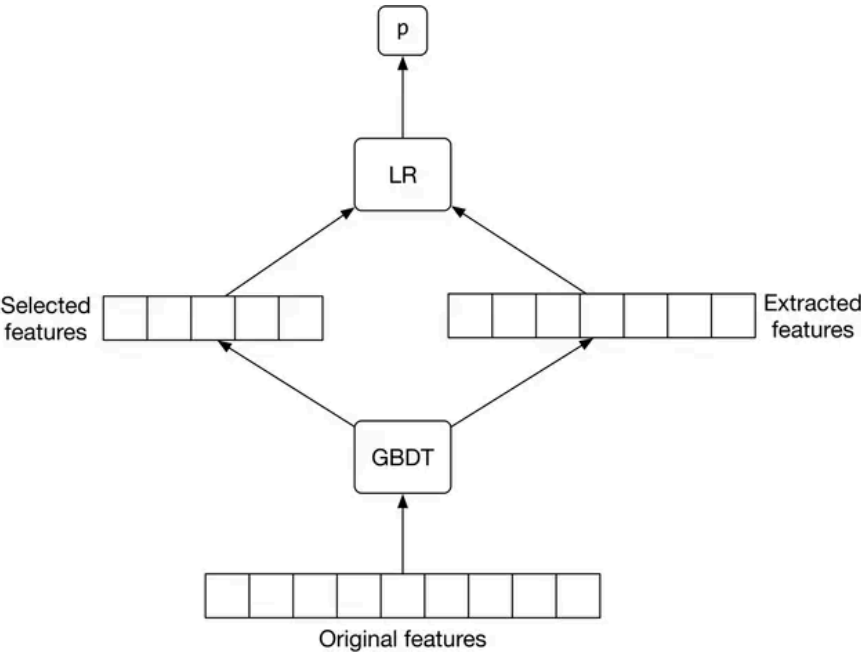


Figure 8.9: GBDT + LR overview

Let's explore the pros and cons of this approach.

Pros:

- In contrast to the existing features, the newly created ones produced by GBDT are more predictive, making it

easier for the LR model to learn the task.

Cons:

- **Cannot capture complex interactions.** Similar to LR, this approach cannot learn pairwise feature interactions.
- **Continual learning is slow.** Fine-tuning GBDT models on new data takes time, which slows down continual learning overall.

Neural network (NN)

NN is another candidate for building the ad click prediction system. To predict click probabilities using a NN, we have two architectural options:

- Single NN
- Two-tower architecture

Single NN: Using the original features as input, a neural network outputs the click probability (Figure 8.10).

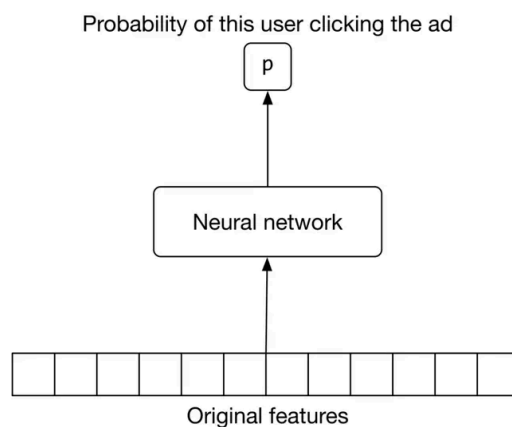


Figure 8.10: NN architecture

Two-tower architecture: In this option, we use two encoders: user encoder and ad encoder. The similarity between ad and user embeddings is used to determine the relevance, that is, the click probability. Figure 8.11 shows an overview of this architecture.

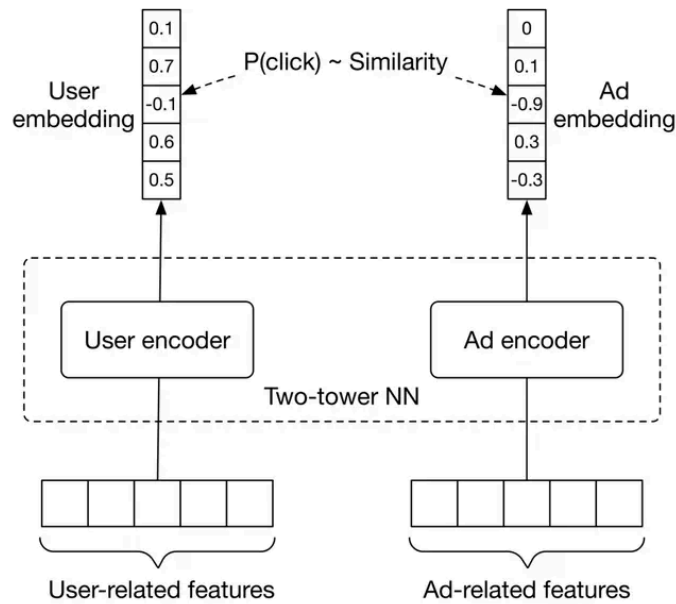


Figure 8.11: Embedding-based NN

Although NNs have many benefits, they may not be the best choice for ad click prediction systems because:

- **Sparsity:** Given that the feature space is usually huge and sparse, most features are filled with zeroes. It may not be possible for NN to learn the task effectively because it does not have access to enough data points.
- **Difficult to capture all pairwise feature interactions** due to the large number of features.

Given these limitations, we will not use NNs.

Deep & Cross Network (DCN)

In 2017, Google proposed an architecture named DCN [6] to find feature interactions automatically. This addresses the challenges of the manual feature crossing method. The following two parallel networks are used in this method:

- **Deep network:** Learns complex and generalizable features using a Deep Neural Network (DNN) architecture.
- **Cross network:** Automatically captures feature interactions and learns good feature crosses.

The outputs of deep network and cross network are concatenated to make a final prediction.

There are two types of DCN architectures: stacked and parallel. Figure 8.12 shows the architecture of a parallel DCN. To learn more about stacked architecture, refer to [7]. Note that it's not usually expected to provide details of DCN during ML system design interviews. If you are interested in learning more about DCN networks, refer to [7] [8].

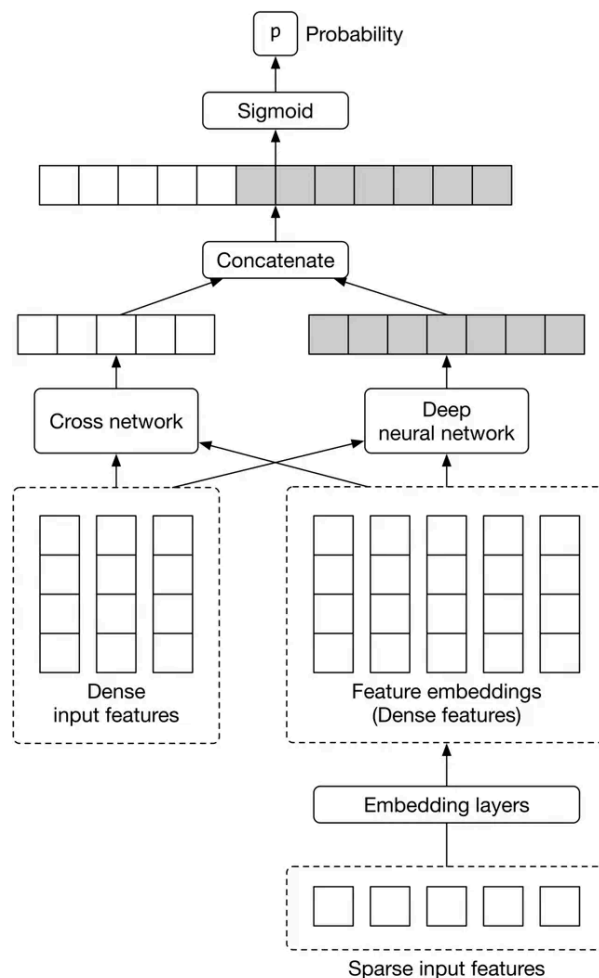


Figure 8.12: DCN architecture

DCN architecture is more effective than neural networks because it implicitly learns feature crosses. However, the cross network only models certain feature interactions, which may negatively affect the performance of the cross network model.

Factorization Machines (FM)

FM is an embedding-based model which improves logistic regression by automatically modeling all pairwise feature interactions. In ad click prediction systems, FM is widely used because it can efficiently model complex interactions between features.

So, let's understand how FM works. It automatically models all pairwise feature interactions by learning an embedding vector for each feature. The interaction between two features is determined by the dot product of their embeddings. Let's look at its formula to understand it better:

$$g(x) = w_0 + \sum_i w_i x_i + \sum_i \sum_j \langle v_i, v_j \rangle x_i x_j$$

Where x_i refers to the i -th feature, w_i is the learned weight, and v_i represents the embedding of the i -th feature. $\langle v_i, v_j \rangle$ denotes the dot product between two embeddings.

This formula may look complex, but it's actually easy to understand. The first two terms compute a linear combination of the features, similar to how logistic regression works. The third term models pairwise feature interactions. Figure 8.13 shows a high-level overview of FM. Refer to [9] to learn the details of FM.

$$\hat{y}(x) = w_0 + \underbrace{\sum_i w_i x_i}_{\text{Logistic regression}} + \underbrace{\sum_i \sum_j \langle v_i, v_j \rangle x_i x_j}_{\text{Pairwise interactions}}$$

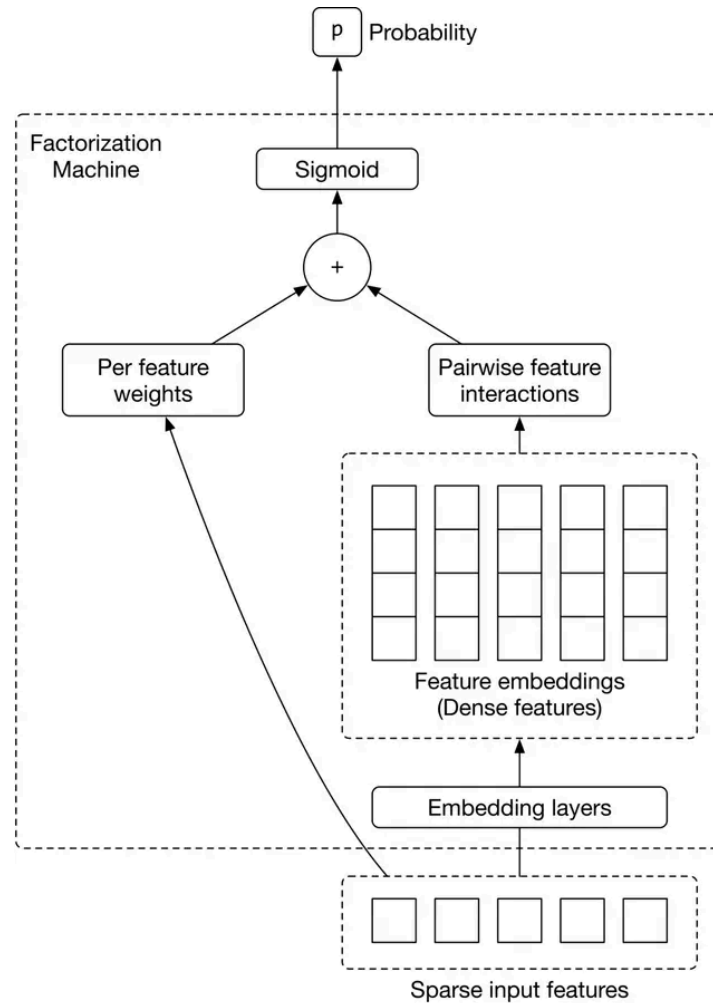


Figure 8.13: Factorization Machine architecture

FM and its variants, such as FFM, effectively capture pairwise interactions between features. FM cannot learn sophisticated higher-order interactions from features, unlike neural networks, which can. In the next method, we combine FM and DNN to overcome this.

Deep Factorization Machines (DeepFM)

DeepFM is an ML model that combines the strengths of both NN and FM. A DNN network captures sophisticated higher-order features, and an FM captures low-level pairwise feature interactions. Figure 8.14 shows the high-

level architecture of DeepFM. If you are interested to learn more about DeepFM, refer to [10].

Figure 8.14: DeepFM overview

One potential improvement is to combine GBDT and DeepFM. A GBDT converts the original features into more predictive features, while DeepFM operates on the new features. This method has won various ad prediction system competitions [11]. However, adding GBDT to DeepFM negatively affects the training and inference speed, and slows down the continual learning process.

In practice, we usually choose the correct model by running experiments. In our case, we start with a simple LR to create a baseline. Next, we experiment with DCN and DeepFM, as both are widely used in the tech industry.

Model training

Constructing the dataset

For every ad impression, we construct a new data point. The input features are computed from the user and the ad. A label is assigned to the data point, based on the following strategy:

- **Positive label:** if the user clicks the ad in less than t seconds after the ad is shown, we label the data point as "positive". Note that t is a hyperparameter and can be tuned via experimentation.

- **Negative label:** if the user does not click the ad in less than t seconds, we label the data point as "negative".

In practice, companies use more complex methods to find the optimal strategy for labeling negative data points. To learn more, refer to [1].

#	User and interaction features	Ad features	Label													
1	<table><tr><td>1</td><td>0</td><td>1</td><td>0.8</td><td>0.1</td><td>1</td><td>0</td></tr></table>	1	0	1	0.8	0.1	1	0	<table><tr><td>0</td><td>1</td><td>1</td><td>0.4</td><td>0.9</td><td>0</td></tr></table>	0	1	1	0.4	0.9	0	Positive
1	0	1	0.8	0.1	1	0										
0	1	1	0.4	0.9	0											
2	<table><tr><td>1</td><td>1</td><td>0</td><td>-0.6</td><td>0.9</td><td>1</td><td>1</td></tr></table>	1	1	0	-0.6	0.9	1	1	<table><tr><td>1</td><td>1</td><td>0</td><td>0.2</td><td>0.7</td><td>1</td></tr></table>	1	1	0	0.2	0.7	1	Negative
1	1	0	-0.6	0.9	1	1										
1	1	0	0.2	0.7	1											

Figure 8.15: Constructed dataset

To keep the model adaptive to new data, it must continuously be trained. As a result, new training data points should be continuously generated using new interactions. We will discuss continual learning further in the serving section.

Choosing the loss function

Since we are training a binary classification model, we choose cross-entropy as a classification loss function.

Evaluation

Offline metrics

Two metrics are typically used to evaluate an ad click prediction system:

- Cross-entropy (CE)
- Normalized cross-entropy (NCE)

CE. This metric measures how close the model's predicted probabilities are to the ground truth labels. CE is zero if we have an ideal system that predicts a 0 for the negative classes and 1 for the positive classes. The lower the CE, the higher the accuracy of the prediction. The formula is:

$$H(p, q) = - \sum_{c=1}^C p_c \log q_c$$

where p is the ground truth, q is the predicted probability, and C is the total number of classes.

For binary classification, the CE formula can be rewritten as:

$$H(p, q) = - \sum_i p_i \log q_i = - \sum_i (y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i))$$

where y_i is the ground truth label of the i -th data point and $\hat{y}_i$ is the predicted probability of i -th data point.

Let's take a look at a concrete example, as shown in Figure 8.16.

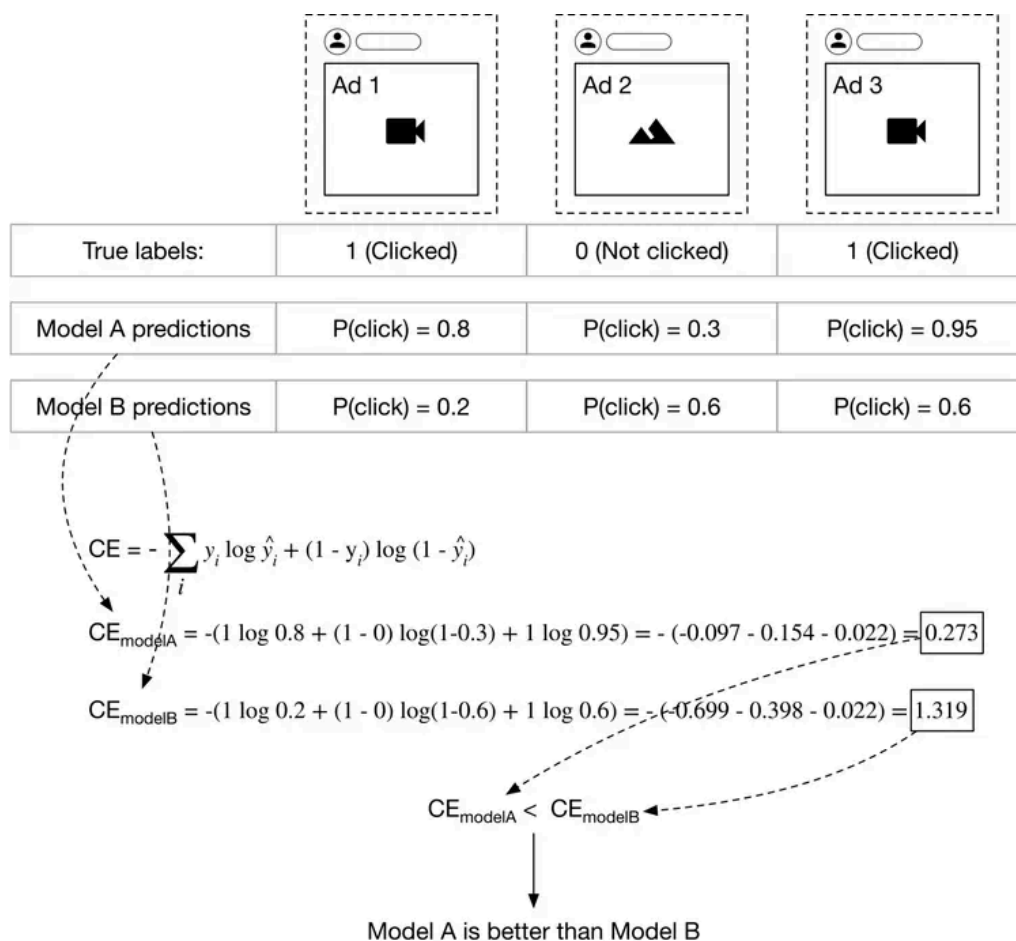


Figure 8.16: CE of two ML models

Note that aside from using CE as a metric, it is also commonly used as a standard loss function in classification tasks during model training.

Normalized cross-entropy (NCE). NCE is the ratio between our model's CE and the CE of the background CTR (average CTR in the training data). In other words, NCE compares the model with a simple baseline which always predicts the background CTR. A low NCE indicates the model outperforms the simple baseline. $NCE \geq 1$ indicates that the model is not performing better than the simple baseline.

$$\text{Normalized cross entropy} = \frac{CE(\text{ML model})}{CE(\text{Simple baseline})}$$

Let's take a look at a concrete example to understand better how NCE is calculated. As shown in Figure 8.17, a simple baseline model always predicts 0.6 (CTR in the training data). In this case, the NCE value is 0.324 (less than 1), indicating model A outperforms the simple baseline.

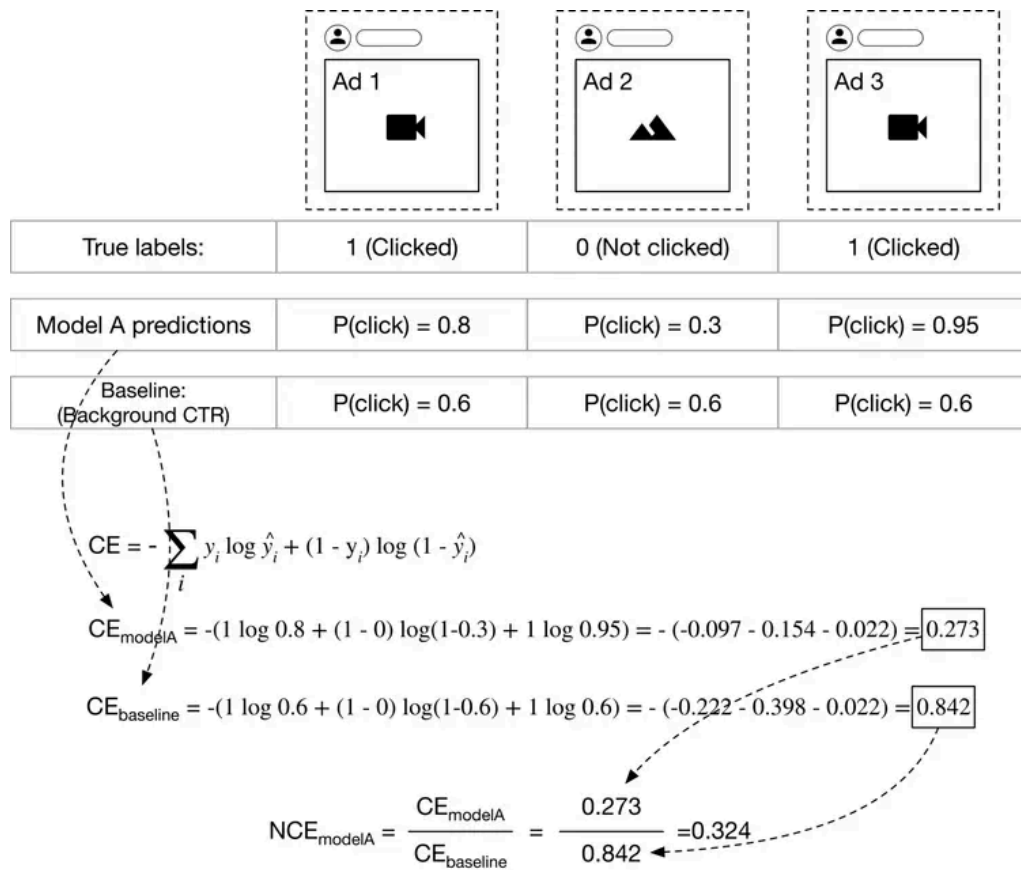


Figure 8.17: NCE calculations for model A

Online metrics

Let's examine some metrics we may use during online evaluation.

- CTR
- Conversion rate
- Revenue lift
- Hide rate

CTR. This metric measures the ratio between clicked ads and the total number of shown ads.

$$CTR = \frac{\text{Number of clicked ads}}{\text{Number of shown ads}}$$

CTR is a great online metric for ad click prediction systems, as maximizing user clicks on ads is directly related to an increase in revenue.

Conversion rate. This metric measures the ratio between the number of conversions and the total number of ads shown.

$$\text{Conversion rate} = \frac{\text{Number of conversions}}{\text{Number of impressions}}$$

This metric is important to track as it indicates how many times advertisers actually benefited from the system. This matters because advertisers will eventually lose interest and cease spending on ads if their ads do not lead to conversions.

Revenue lift. This measures the percentage of revenue increase over time.

Hide rate. This metric measures the ratio between the number of ads hidden by users and the number of shown ads.

$$\text{Hide rate} = \frac{\text{Number of ads hidden by users}}{\text{Number of shown ads}}$$

This metric is helpful for understanding how many irrelevant ads the system displayed to users, also known as false positives.

Serving

At serving time, the system is responsible for outputting a list of ads ranked by their click probabilities. The proposed ML system design is shown in Figure 8.18. Let's examine each of the following pipelines:

- Data preparation pipeline
- Continual learning pipeline
- Prediction pipeline



Figure 8.18: ML system design

Data preparation pipeline

The data preparation pipeline performs the following two tasks:

1. Compute online and batch features
2. Continuously generate training data from new ads and interactions

To compute features, the following two options are used: batch feature computation and online feature computation. Let's see how they differ from each other.

Batch feature computation Some of the features we chose are static, which means they change very rarely. For example, an ad's image and category are static features. This component computes static features periodically (e.g., every few days or weeks) with batch jobs and then stores the features in a feature store. This improves the

system's performance during serving because the features are precomputed.

Online feature computation Some features are dynamic as they change frequently. For example, the numbers of ad impressions and clicks are examples of dynamic features. These features need to be computed at query time, and this component is used to compute dynamic features.

Continual learning pipeline

Based on the requirements, continually learning the model is critical. This pipeline is responsible for fine-tuning the model on new training data, evaluating the new model, and deploying the model if it improves the metrics. It ensures the prediction pipeline always uses a model adapted to the most recent data.

Prediction pipeline

The prediction pipeline takes a query user as input and outputs a list of ads ranked by their click probabilities. Since some of the features which the model relies upon are dynamic, we cannot use batch prediction. Instead, requests are served as they arrive using online prediction.

As we've seen in previous chapters, a two-stage architecture is used in the prediction pipeline. First, we employ a candidate generation service to efficiently narrow down the available pool of ads to a small subset of ads. In this

case, we use the ad targeting criteria often provided by advertisers, such as target age, gender, and country.

Next, we employ a ranking model which fetches the candidate ads from the candidate generation service, ranks them based on click probability, and outputs the top ads. This component interacts with the same feature store and online feature computation component. Once the static and dynamic features are obtained, the ranking service uses the model to get a predicted click probability for each candidate ad. These probabilities are used to rank the ads and to output those with the highest click probability.

Finally, a re-ranking service modifies the list of ads by incorporating additional logic and heuristics. For example, we can increase the diversity of ads by removing very similar ads from the list.

Other Talking Points

If there is time left at the end of the interview, here are some potential talking points you might discuss with the interviewer:

- In ranking and recommendation systems, it's important to avoid data leakage [12][13]
- The model needs to be calibrated in ad click prediction systems. Discuss model calibration and techniques for calibrating a model [14].
- A common variant of FM is a field-aware Factorization Machine (FFM). It's good to talk about FFM and how it differs from FM [15].
- A common variant of DeepFM is XDeepFM. Talk about XDeepFM and how it differs from DeepFM [10].
- We've described why continuous learning is necessary for ad click prediction systems. However, continual learning on new data may lead to catastrophic forgetting. Discuss what catastrophic forgetting is and what common solutions are [16].

References

1. Addressing delayed feedback. <https://arxiv.org/pdf/1907.06558.pdf>.
2. AdTech basics. <https://advertising.amazon.com/library/guides/what-is-adtech>.
3. SimCLR paper. <https://arxiv.org/pdf/2002.05709.pdf>.
4. Feature crossing. <https://developers.google.com/machine-learning/crash-course/feature-crosses/video-lecture>.
5. Feature extraction with GBDT. <https://towardsdatascience.com/gradient-boosted-decision-trees-explained-9259bd8205af/>.
6. DCN paper. <https://arxiv.org/pdf/1708.05123.pdf>.
7. DCN V2 paper. <https://arxiv.org/pdf/2008.13535.pdf>.
8. Microsoft's deep crossing network paper. <https://www.kdd.org/kdd2016/papers/files/adv0975-shanA.pdf>.
9. Factorization Machines. <https://www.jefkine.com/recsys/2017/03/27/factorization-machines/>.
10. Deep Factorization Machines. https://d2l.ai/chapter_recommender-systems/deepfm.html.
11. Kaggle's winning solution in ad click prediction. <https://www.youtube.com/watch?v=4Go5crRVyuU>.
12. Data leakage in ML systems. <https://machinelearningmastery.com/data-leakage-machine-learning/>.
13. Time-based dataset splitting. https://www.linkedin.com/pulse/time-based-splitting-determining-train-test-data-come-manraj-chalokia/?trk=public_profile_article_view.
14. Model calibration. <https://machinelearningmastery.com/calibrated-classification-model-in-scikit-learn/>.
15. Field-aware Factorization Machines. <https://www.csie.ntu.edu.tw/~cjlin/papers/ffm.pdf>.
16. Catastrophic forgetting problem in continual learning. <https://www.cs.uic.edu/~liub/lifelong-learning/continual-learning.pdf>.

09

Similar Listings on Vacation Rental Platforms

Recommending items similar to those a user is currently viewing, is a key technology that allows people to discover potentially relevant content on large platforms. For example, Airbnb recommends similar accommodation listings, Amazon recommends similar products, and Expedia recommends similar experiences to users.

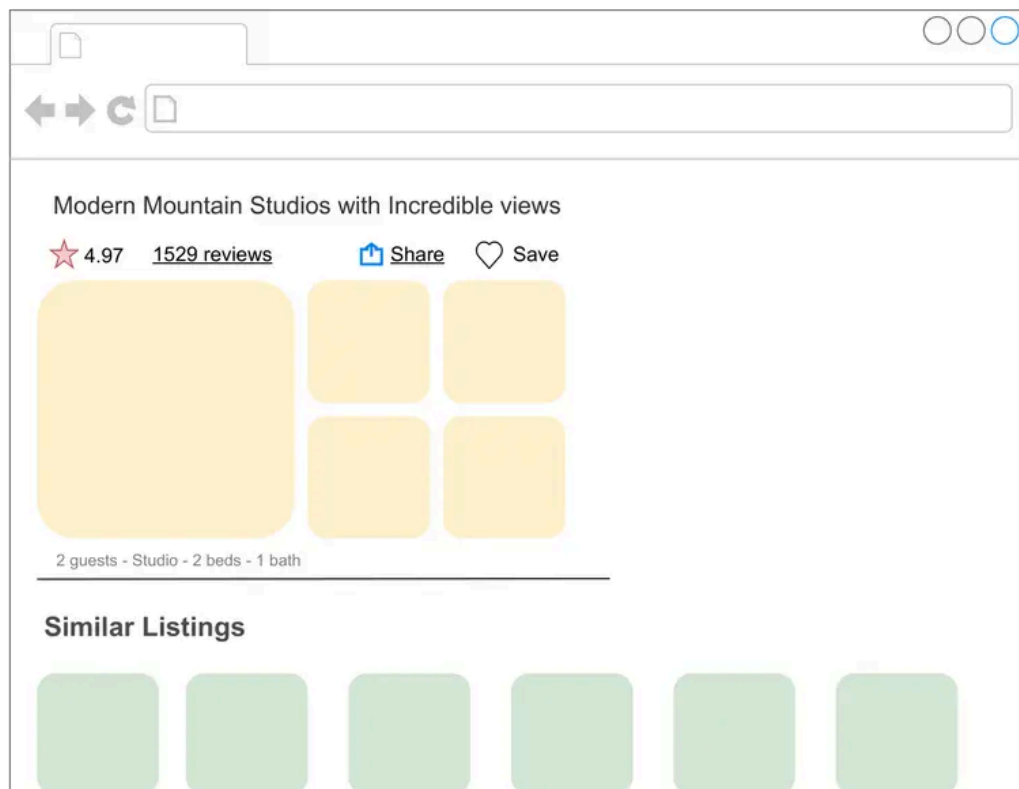


Figure 9.1: Recommended similar listings

In this chapter, we design a "similar listings" feature which resembles those used by vacation rental websites such as Airbnb and Vrbo. When a user clicks a specific listing, a list of similar listings is recommended to them.

Clarifying Requirements

Here is a typical interaction between a candidate and an interviewer.

Candidate: Can I assume the business objective is to increase the number of bookings?

Interviewer: Yes.

Candidate: What is the definition of "similarity"? Are the recommended listings expected to be similar to the listing that a user is currently viewing?

Interviewer: Yes, that's correct. Two listings are defined as similar when they are in the same neighborhood, city, price range, etc.

Candidate: Are the recommended listings personalized to users?

Interviewer: We want this feature to work for both logged-in and anonymous users. In practice, we treat the two groups differently and apply personalization to logged-in users. However, for simplicity, let's assume we treat logged-in and anonymous users equally.

Candidate: How many listings are available on the platform?

Interviewer: 5 million listings.

Candidate: How do we construct the training dataset?

Interviewer: Good question. For this interview, let's assume we use user-listing interactions only. The model doesn't leverage users' attributes, such as age or location, or a listing's attributes, like price and location, at all.

Candidate: How long does it take for new listings to appear in the similar listings result?

Interviewer: Let's assume new listings are okay to appear as recommendations one day after being posted. During this time, the system collects interaction data for new listings.

Let's summarize the problem statement. We are asked to design a "similar listings" feature for vacation rental platforms. The input is a specific listing that a user is currently viewing, and the output is a ranked list of similar listings the user is likely to click on next. The recommended listings should be the same for both anonymous and logged-in users. There are around 5 million listings on the platform, and new listings can appear in recommendations after one day. The business objective of the system is to increase the number of bookings.

Frame the Problem as an ML Task

Defining the ML objective

The sequence of listings that a user clicks on usually have similar characteristics, such as being in the same city or having similar price ranges. We rely on this observation to define the ML objective as accurately predicting which listing the user will click next, given the listing the user is currently viewing.

Specifying the system's input and output

As shown in Figure 9.2, the "similar listings" system takes a listing a user is currently viewing as input and then outputs a ranked list of listings, sorted by the probability of this user clicking on them.

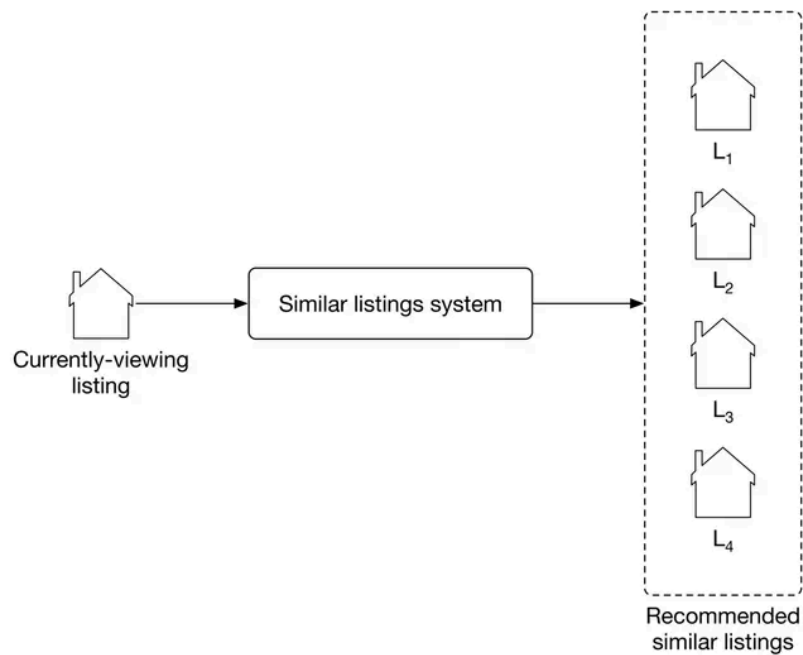


Figure 9.2: A similar listings system's input-output

Choosing the right ML category

Most recommendation systems rely on users' historical interactions to understand their long-term interests. However, such recommendation systems may not be good at solving the similar listings problem. In our situation, recently viewed listings are more informative than those viewed a long time ago. In this case, a session-based recommendation system is commonly used.

Like Airbnb, many e-commerce and travel booking platforms rely more on short-term interests to make recommendations. In systems where high-quality recommendations depend more on recent interactions than long-term interests, a session-based recommendation is often a substitute for traditional recommendation systems. A session-based recommendation makes recommendations based on the user's current browsing session. Now, let's take a closer look at session-based recommendation systems.

Session-based recommendation systems

A session-based recommendation system aims to predict the next item, given a sequence of recent items browsed by a user. In the system, users' interests are context-dependent and evolve fast. A good recommendation heavily depends on the user's most recent interactions, not their generic interests.



Figure 9.3: A browsing session of products

How do session-based and traditional recommendation systems compare?

In traditional recommendation systems, users' interests are context-independent and won't change too frequently. In session-based recommendations, users' interests are dynamic and evolve fast. The goal of a traditional recommendation system is to learn users' generic interests. In contrast, session-based recommendation systems aim to understand users' short-term interests, based on their recent browsing history.

A widely-used technique to build session-based recommendation systems is to learn item embeddings using co-occurrences of items in users' browsing histories. For example, Instagram learns account embeddings to power its "Explore" feature [1], Airbnb learns listing embeddings to power its similar listings feature [2], and word2vec [3] uses a similar approach to learn meaningful word embeddings.

In this chapter, we frame the "similar listings" problem as a session-based recommendation task. We build the system by training a model which maps each listing into an embedding vector, so that if two listings frequently co-occur in users' browsing history, their embedding vectors are in close proximity in the embedding space.

To recommend similar listings, we search the embedding space for listings closest to the one currently being viewed. Let's take a look at an example of this. In Figure 9.4, each listing is mapped into a 2D space. To recommend similar listings to L_t , we choose the top 3 listings with the closest embeddings.

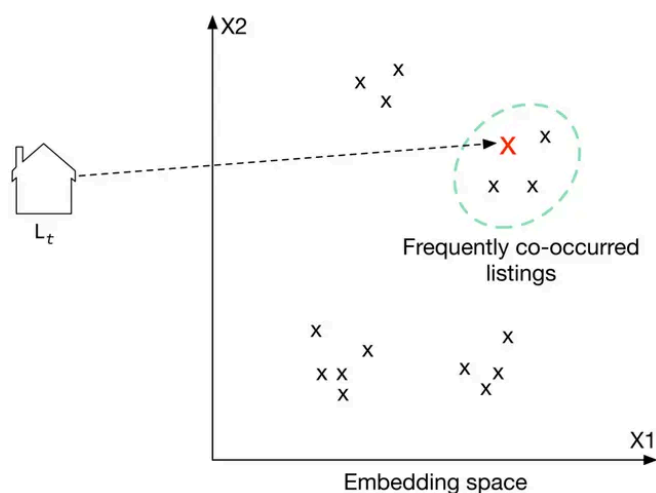


Figure 9.4: Similar listings in the embedding space

Data Preparation

Data engineering

The following data are available:

- Users
- Listings
- User-listing interactions

Users

A simplified user data schema is shown below.

ID	Username	Age	Gender	City	Country	Language	Time zone
----	----------	-----	--------	------	---------	----------	-----------

Table 9.1: User data schema

Listings

Listing data contains attributes related to each listing, such as price, number of beds, host ID, etc. Table 9.2 shows a simple example of what the listing data might look like.

ID	Host ID	Price	Sq ft	Rate	Type	City	Beds	Max guests
1	135	135	1060	4.97	Entire place	NYC	3	4
2	81	80	830	4.6	Private room	SF	1	2
3	64	65	2540	5.0	Shared room	Boston	4	6

Table 9.2: Listing data

User-listing interactions

Table 9.3 stores user-listing interactions such as impressions, clicks, and bookings.

ID	User ID	Listing ID	Position of the listing in the displayed list	Interaction type	Source	Timestamp
2	18	26	2	Click	Search feature	1655121925
3	5	18	5	Book	Similar listing feature	1655135257

Table 9.3: User-listing interaction data

Feature engineering

As described in the "Frame the Problem as ML Task" section, the model only utilizes users' browsing history

during training. Other information is not used, such as listing price, user's age, etc.

In this chapter, the browsing histories are called "search sessions". A search session is a sequence of clicked listing IDs, followed by an eventually booked listing, without interruption. Figure 9.5 shows an example of a search session, where the user's session started when the user clicked on L_1 , and ended when the user eventually booked L_{20} .

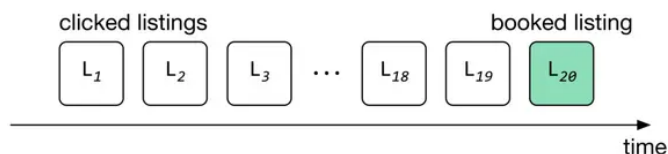


Figure 9.5: A search session

In the feature engineering step, we extract search sessions from the interaction data. Table 9.4 shows a simple example of the search sessions.

Session ID	Clicked listing IDs	Eventually booked listing ID
1	1,5,4,9	26
2	6,8,9,21,6,13,6	5
3	5,9	11

Table 9.4: Search session data

Model Development

Model selection

A neural network is the standard method to learn embeddings. Choosing a good architecture for the neural network depends upon various factors, such as the complexity of the task, the amount of training data, etc. One common way to choose the hyperparameters associated with neural network architectures - such as the number of neurons, layers, activation function, etc. - is to run experiments and choose the architecture that performs best. In our case, we choose a shallow neural network architecture to learn listing embeddings.

Model training

As shown in Figure 9.6, for a given input listing, the model's job is to predict listings within the input listing's context.

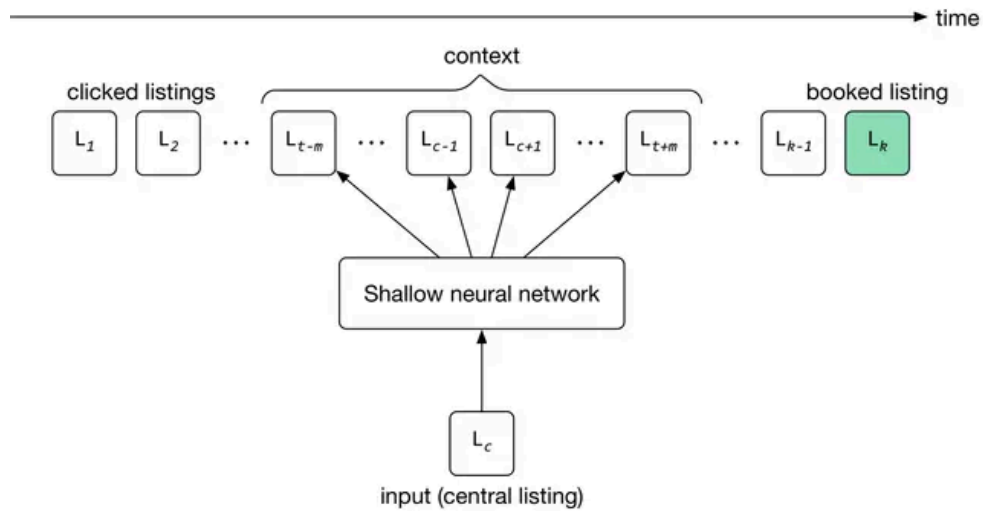


Figure 9.6: Predicting neighboring listings

The training process starts by initializing listing embeddings to random vectors. These embeddings are learned gradually by reading through search sessions, using the sliding window method. As the window slides, the embedding of the central listing in the window is updated to be similar to the embedding of other listings in the window, and dissimilar from listings outside the window. The model then uses these embeddings to predict the context of a given listing.

To adapt the model to new listings, we train it daily on the newly constructed training data.

Constructing the dataset

There are different ways to construct a dataset. In our case, we choose a technique called "negative sampling" [4], commonly used to learn embeddings.

To construct the training data, we create positive pairs and negative pairs from search sessions. Positive pairs are listings that are expected to have similar embeddings, while negative pairs are expected to have dissimilar embeddings.

More precisely, for each session, we read through the listings with the sliding window method. As the window slides, we use the central listing in the window and its context listings to create positive pairs. We use the central listing and randomly sampled listings to form negative pairs. Positive pairs have a ground truth label of 1, and negative pairs are given a label of 0.

Figure 9.7 shows how positive and negative pairs are generated by sliding through a search session.

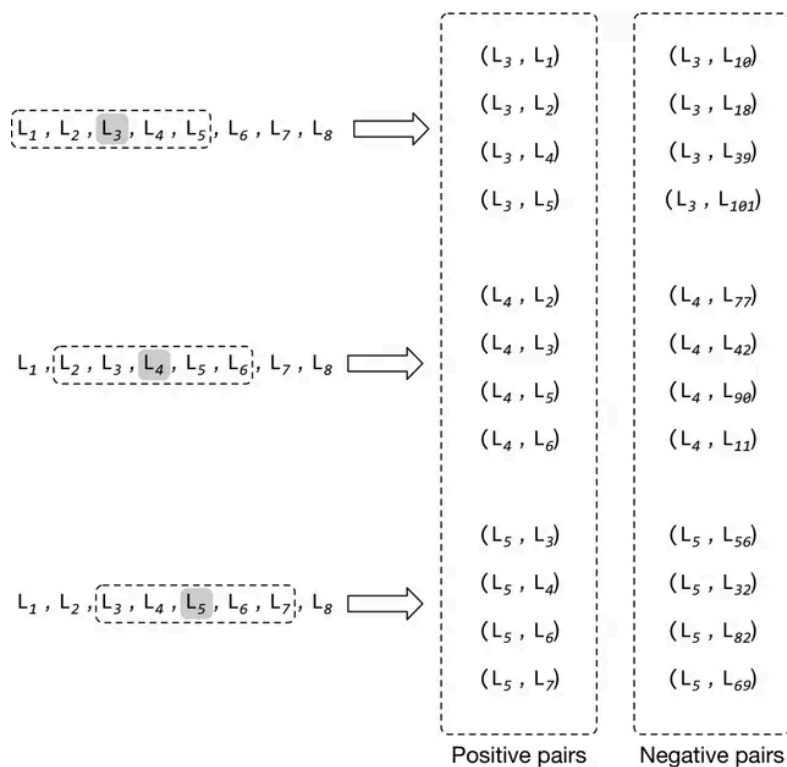


Figure 9.7: Constructed positive and negative listing pairs

Choosing the loss function

Loss function measures the agreement between the ground truth label and the predicted probability. If two listings form a positive pair, the embeddings should be close, and if the two listings form a negative pair, the embeddings should be far apart. More formally, here are the steps to calculate loss:

1. Compute the distance (e.g., dot product) between two embeddings.
2. Use the Sigmoid function to convert the computed distance to a probability value between 0 and 1 .
3. Use cross-entropy as a standard classification loss to measure the loss between the predicted probability and the ground truth label.

Figure 9.8 shows the loss calculation steps.

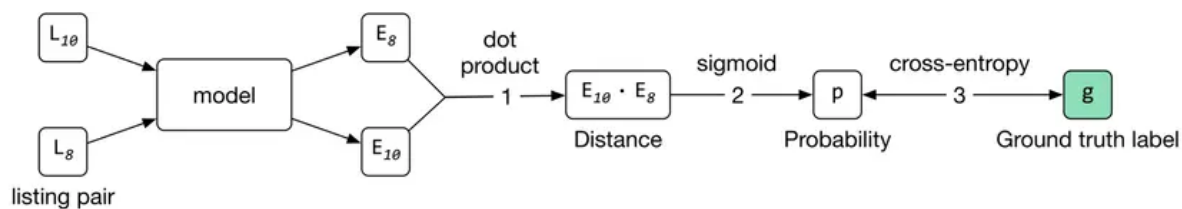


Figure 9.8: Loss calculation steps

The loss can be represented by the following formula:

$$\text{loss} = \sum_{(c,p) \in D_p} \log \frac{1}{1 + e^{-E_p \cdot E_c}} + \sum_{(c,n) \in D_n} \log \frac{1}{1 + e^{E_n \cdot E_c}}$$

Where:

- c is a central listing, p is a positive listing (co-occurred with c in a context), and n is a negative listing (did not co-occur with c)
- E_c represents the embedding vector of the central listing c
- E_n represents the embedding vector of negative listing n
- E_p represents the embedding vector of positive listing p
- D_p is a positive set of pairs $\langle c, p \rangle$ which represents (central listing, context listing) tuples whose vectors are being pushed toward one another
- D_n is a negative set of pairs $\langle c, n \rangle$ which represents (central listing, random listing) tuples whose vectors are being pushed away from each other

The first summation computes the loss over positive pairs and the second summation computes the loss over negative pairs.

Can we improve the loss function to learn better embeddings?

The loss function described earlier is a good starting point, but it has two shortcomings. First, during training, the embedding of the central listing is pushed closer to the embeddings in its context, but not towards the embedding of the eventually booked listing. This leads to embeddings that are good at predicting neighboring clicked listings, but not at predicting eventually booked listings. This is not optimal for helping users discover a listing that leads to a booking.

Second, the negative pairs generated earlier mainly comprise listings from different regions, since they are sampled randomly. However, users typically search only within a certain region, e.g., San Francisco. This may lead to embeddings that do not work well for same-region listings; those that have not co-occurred in context, but are from the same region.

Let's address these shortcomings.

Using the eventually booked listing as a global context

To learn embeddings that are good at predicting eventually booked listings, we treat the eventually booked listing as a global context during the training phase. As the window slides, some listings fall in or out of the context set, while the eventually booked listing always remains in the global context, and is used to update the central listing vector.

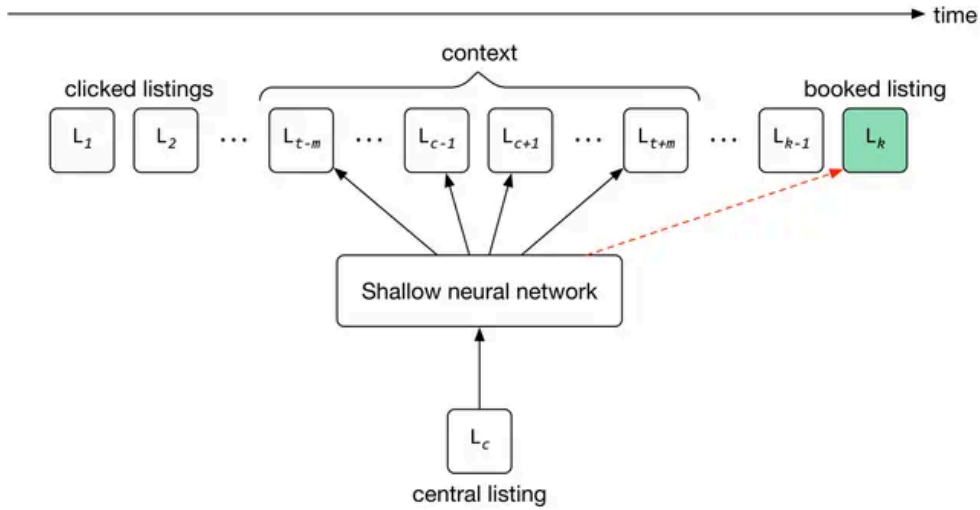


Figure 9.9: Adding the eventually booked listing to the positive pairs

To use the eventually booked listing as a global context during training, we add pairs of $\langle \text{central listing, eventually booked listing} \rangle$ to our training data and label them as positive. This drives the model to push the embedding of the eventually booked listing close to each of the clicked listings in the session during training, as shown in Figure 9.9.

Add negative pairs from the same region to the training data

As the window slides, we choose a listing from the same neighborhood as the central listing, which is not within the central listing's context. We label the pair as negative and add it to our training data. Let's see the updated loss function that considers newly added training data.

$$\text{loss} = \sum_{(c,p) \in D_p} \log \frac{1}{1 + e^{-E_c \cdot E_p}} + \sum_{(c,n) \in D_n} \log \frac{1}{1 + e^{E_c \cdot E_n}} + \sum_{(c,b) \in D_{\text{booked}}} \log \frac{1}{1 + e^{-E_c \cdot E_b}} + \sum_{(c,n) \in D_{\text{hard}}} \log \frac{1}{1 + e^{E_c \cdot E_n}}$$

Where:

- E_b represents the embedding vector of the eventually booked listing b
- D_{booked} are pairs of $\langle c, b \rangle$ that represent (central listing, booked listing) tuples whose vectors are being pushed close to each other
- D_{hard} are hard negative pairs $\langle c, n \rangle$ that represent (central listing, same-region negative listing) tuples whose vectors are being pushed away from each other

We explained the first two summations earlier. The third summation computes the loss over newly added positive pairs which contain the global context. It helps the model to push the central listings' embeddings close to eventually booked listings' embeddings.

The fourth summation computes the loss over newly added negative pairs from the same region. It enforces the model to push their embeddings away from each other.

Evaluation

Offline metrics

During the model development phase, we use offline metrics to measure the output quality of the model and compare the newly developed models with older ones. One way to evaluate learned embeddings is to test how good they are at predicting the eventually booked listing, based on the latest user click. Let's create a metric called "average rank of eventually booked listing" and discuss this in more detail.

The average rank of the eventually-booked listing. Let's look at an example to understand this metric. Figure 9.10 shows a user's search session. As you can see, the search session consists of seven listings in total. The first listing is what the user viewed first (L_0). The next five are listings the user clicked on, sequentially. The last one (L_6) is the listing that the user eventually booked.

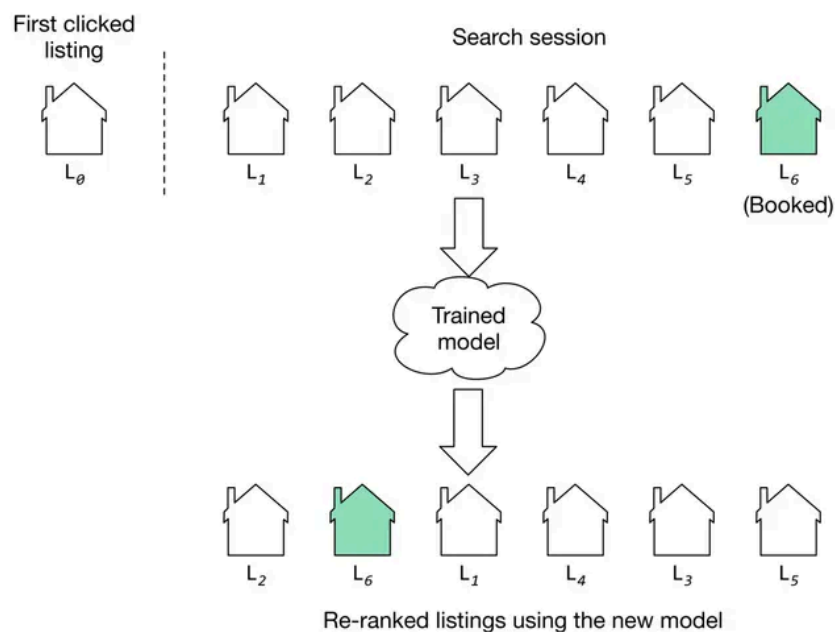


Figure 9.10: A session re-ranked by a model

We use the model to compute the similarities between the first clicked listing and other listings in the embedding space. Once similarities are computed, the listings are ranked. The position of the eventually booked listing indicates how high in the ranking we could have recommended it (L_6) by employing the new model. As you can see in Figure 9.10, the new model (second row) was able to rank the eventually booked listing (L_6) in second place.

If the model ranks the eventually booked listing highly, it indicates the learned embeddings can place the eventually booked listing earlier in the recommended list. We average the rank of the eventually booked listings across all the sessions in the validation dataset, to compute the value of this metric.

Online metrics

According to the requirements, the business objective is to increase the number of bookings. Here are some

options for online metrics:

- Click-through rate (CTR)
- Session book rate

CTR. A ratio showing how often people who see the recommended listings, end up clicking them.

$$CTR = \frac{\text{Number of clicked listings}}{\text{Number of recommended listings}}$$

This metric is used to measure user engagement. For example, when users click on listings more frequently, there is a higher likelihood that some of the clicked listings become a booking. But since CTR does not measure the actual number of bookings made on the platform, we use the "session book rate" metric to supplement CTR.

Session book rate. A ratio showing how many search sessions turn into a booking.

$$\text{Session book Rate} = \frac{\text{Number of sessions turned into booking}}{\text{Total number of sessions}}$$

This metric is directly related to our business objective, which is to increase the number of bookings. The higher the "session book rate" is, the more revenue the platform generates.

Serving

At serving time, the system recommends listings similar to that the user is currently viewing. Figure 9.11 shows an overview of the ML system design.

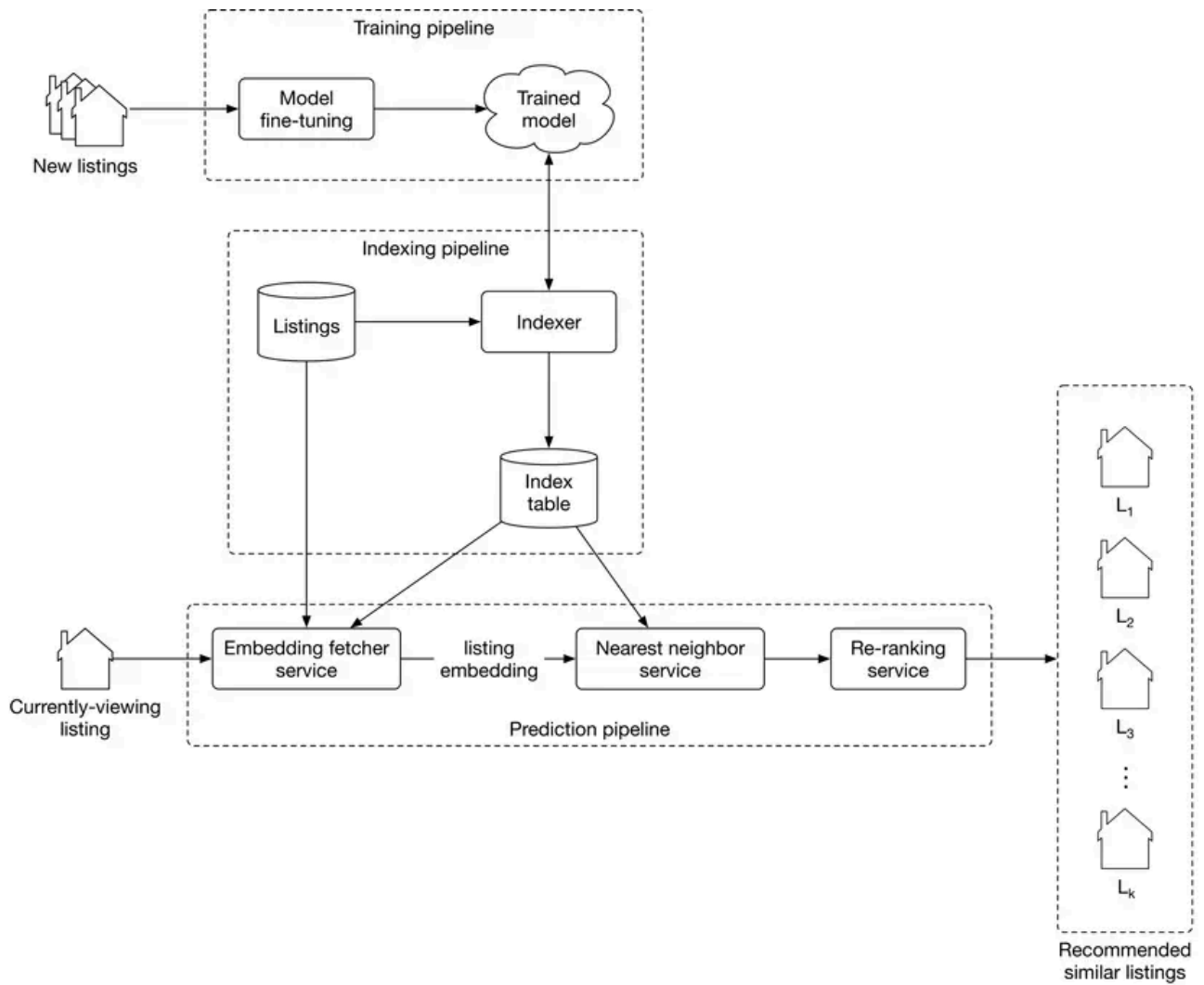


Figure 9.11: ML system design

Let's examine the main components in detail.

Training pipeline

The training pipeline fine-tunes the model using new listings and user-listing interactions. This ensures the model is always adapted to new interactions and listings.

Indexing pipeline

With a trained model, the embeddings of all listings on the platform can be pre-computed and stored in the index table. This significantly speeds up the prediction pipeline.

The indexing pipeline creates and maintains the index table. For example, when a new listing embedding becomes available, the pipeline adds its embedding to the index table. In addition, when a newly trained model becomes available, the pipeline re-computes all the embeddings using the new model and updates the index table.

Prediction pipeline

The prediction pipeline recommends similar listings to what a user is currently viewing. The prediction pipeline, as shown in Figure 9.11, consists of:

- Embedding fetcher service
- Nearest neighbor service
- Re-ranking service

Let's inspect each component.

Embedding fetcher service

This service takes the currently viewing listing as input and acts differently depending on whether or not the listing has been seen by the model during training.

The input listing has been seen by the model during training

If a listing was seen during training, its embedding vector has already been learned and is available in the index table. In this case, the embedding fetcher service directly fetches the listing embedding from the index table.

The input listing has not been seen by the model during training

If the input listing is new, the model hasn't seen it during training. This is problematic since we cannot find similar listings if we do not have the embedding of the given listing.

To solve this issue, the embedding fetcher uses heuristics to handle new listings. For example, we can use the embedding of a geographically nearby listing when the listing is new. When enough interaction data is gathered for the new listing, the training pipeline learns the embedding by fine-tuning the model.

Nearest neighbor service

To recommend similar listings, we need to compute the similarity between the embedding of the currently-viewing listing and the embeddings of other listings on the platform. This is where the nearest neighbor service comes into play. This service computes these similarities and outputs the nearest neighbor listings in the embedding space.

Remember from the requirements that we have five million listings on the platform. Computing similarities for this many listings takes time and may slow down serving. Therefore, we use an approximate nearest neighbor method to speed up the search.

Re-ranking service

This service modifies the listings by applying user filters and certain constraints. For example, if a listing is above a certain price filter set by the user, this layer removes it. In addition, listings in cities other than the currently viewed listing can be removed from the list before being displayed to the user.

Other Talking Points

If there is time left at the end of the interview, here are some additional talking points:

- What is positional bias, and how to address it [5].

- How does a session-based approach compare to random walk [6], and how random walks with restart (RWR) can be used to recommend similar listings [7].
- How to personalize the results of a session-based recommendation system by considering users' longer-term interests (in-session personalization) [2].
- Given that seasonality greatly affects vacation rentals, how should we incorporate seasonality into our similar listings system [8].

References

1. Instagram's Explore recommender system. <https://ai.facebook.com/blog/powered-by-ai-instagrams-explore-recommender-system>.
2. Listing embeddings in search ranking. <https://medium.com/airbnb-engineering/listing-embeddings-for-similar-listing-recommendations-and-real-time-personalization-in-search-601172f7603e>.
3. Word2vec. <https://en.wikipedia.org/wiki/Word2vec>.
4. Negative sampling technique. <https://www.baeldung.com/cs/nlps-word2vec-negative-sampling>.
5. Positional bias. <https://eugeneyan.com/writing/position-bias/>.
6. Random walk. https://en.wikipedia.org/wiki/Random_walk.
7. Random walk with restarts. https://www.youtube.com/watch?v=HbzQzUaJ_9I.
8. Seasonality in recommendation systems. <https://www.computer.org/csdl/proceedings-article/big-data/2019/09/005954/1hJsfgT0qL6>.

10

Personalized News Feed

Introduction

A news feed is a feature of social network platforms that keeps users engaged by showing friends' recent activities on their timelines. Most social networks such as Facebook [1], Twitter [2], and LinkedIn [3] personalize news feed to maintain user engagement.

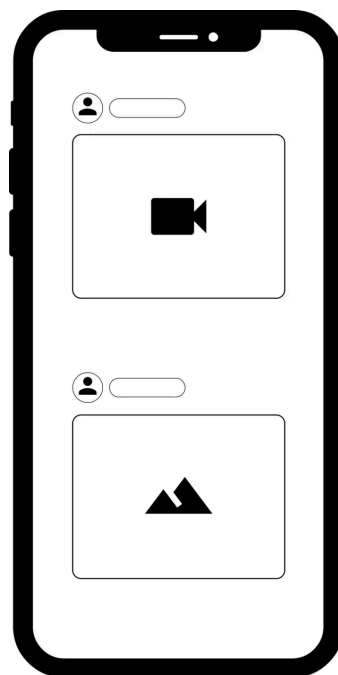


Figure 10.1: User timeline with personalized feeds

In this chapter, we are asked to design a personalized news feed system.

Clarifying Requirements

Here is a typical interaction between a candidate and an interviewer.

Candidate: Can I assume the motivation for a personalized news feed is to keep users engaged with the platform?

Interviewer: Yes, we display sponsored ads between posts, and more engagement leads to increased revenue.

Candidate: When a user refreshes their timeline, we display posts with new activities to the user. Can I assume this activity consists of both unseen posts and posts with unseen comments?

Interviewer: That is a fair assumption.

Candidate: Can a post contain textual content, images, video, or any combination?

Interviewer: It can be any combination.

Candidate: To keep users engaged, the system should place the most engaging content at the top of timelines, as people are more likely to interact with the first few posts. Does that sound right?

Interviewer: Yes, that's correct.

Candidate: Is there a specific type of engagement we are optimizing for? I assume there are different types of engagement, such as clicks, likes, and shares.

Interviewer: Great question. Different reactions have different values on our platform. For example, liking a post is more valuable than only clicking it. Ideally, our system should consider major reactions when ranking posts. With that, I'll leave you to define "engagement" and choose what your model should optimize for.

Candidate: What are the major reactions available on the platform? I assume users can click, like, share, comment, hide, block another user, and send connection requests. Are there other reactions we should consider?

Interviewer: You mentioned the major ones. Let's keep our focus on those.

Candidate: How fast is the system supposed to work?

Interviewer: We expect the system to display the ranked posts quickly after users refresh their timelines or open the application. If it takes too long, users will get bored and leave. Let's assume the system should display the ranked posts in less than 200 milliseconds (ms).

Candidate: How many daily active users do we have? How many timeline updates do we expect each day?

Interviewer: We have almost 3 billion users in total. Around 2 billion are daily active users who check their feeds twice a day.

Let's summarize the problem statement. We are asked to design a personalized news feed system. The system retrieves unseen posts or posts with unseen comments, and ranks them based on how engaging they are to the user. This should take no longer than 200 ms. The objective of the system is to increase user engagement.

Frame the problem as an ML task

Defining the ML objective

Let's examine the following three possible ML objectives:

- Maximize the number of specific implicit reactions, such as dwell time or user clicks
- Maximize the number of specific explicit reactions, such as likes or shares
- Maximize a weighted score based on both implicit and explicit reactions

Let's discuss each option in more detail.

Option 1: Maximize the number of specific implicit reactions, such as dwell time or user clicks In this option, we choose implicit signals as a proxy for user engagement. For example, we optimize the ML system to maximize user clicks.

The advantage is that we have more data about implicit reactions than explicit ones. More training data usually leads to more accurate models.

The disadvantage is that implicit reactions do not always reflect a user's true opinion about a post. For example, a user may click on a post, but find it is not worth reading.

Option 2: Maximize the number of specific explicit reactions, such as likes, shares, and hides

With this option, we choose explicit reactions as a proxy for user opinions about a post.

The advantage of this approach is that explicit signals usually carry more weight than implicit signals. For example, a user liking a post sends a stronger engagement signal than if they simply click it.

The main disadvantage is that very few users actually express their opinions with explicit reactions. For example, a user may find a post engaging but not react to it. In this scenario, it's hard for the model to make an accurate prediction given the limited training data.

Option 3: Maximize a weighted score based on both implicit and explicit reactions

In this option, we use both implicit and explicit reactions to determine how engaged a user is with a post. In particular, we assign a weight to each reaction, based on how valuable the reaction is to us. We then optimize the ML system to maximize the weighted score of reactions.

Table 10.1 shows the mapping between different reactions and weights. As you can see, pressing the "like" button has more weight than a click, while a share is more valuable than a like. In addition, negative reactions such as hide and block have a negative weight. Note that these weights can be chosen based on business needs.

Reaction	Click	Like	Comment	Share	Friendship request	Hide	Block
Weight	1	5	10	20	30	-20	-50

Table 10.1: Weights of different reactions

Which option to choose?

We choose the final blended option because it allows us to assign different weights to different reactions. This is important because we can optimize the system based on what's important to the business.

Specifying the system's input and output

As Figure 10.2 shows, the personalized news feed system takes a user as input and outputs a ranked list of unseen posts or posts with unseen comments sorted by engagement score.

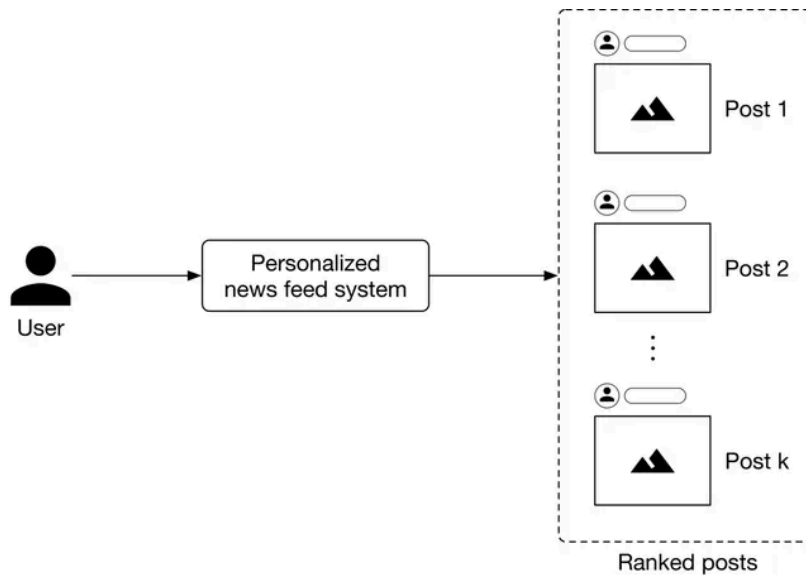


Figure 10.2: A personalized news feed system's input-output

Choosing the right ML category

A personalized news feed system produces a ranked list of posts based on how engaging the posts are to a user. Pointwise Learning to Rank (LTR) is a simple yet effective approach that personalizes news feeds by ranking posts based on engagement scores. To understand how to compute engagement scores between users and posts, let's examine a concrete example.

As Figure 10.3 shows, we employ several binary classifiers to predict the probabilities of various implicit and explicit reactions for a $\langle \text{user}, \text{post} \rangle$ pair.

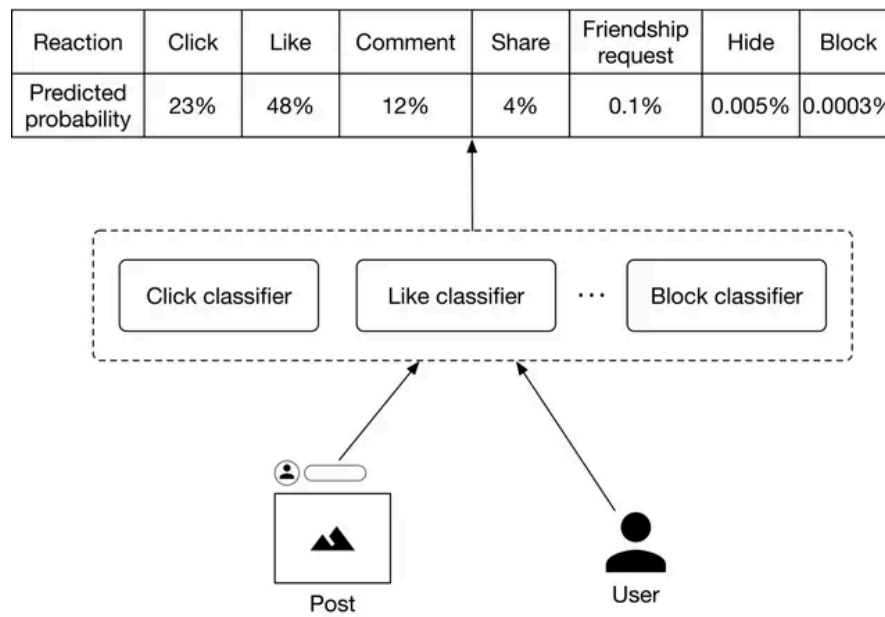


Figure 10.3: Predicted probabilities of various reactions

Once these probabilities are predicted, we compute the engagement score. Figure 10.4 shows an example of how the engagement score is calculated.

Reaction	Click	Like	Comment	Share	Friendship request	Hide	Block
Predicted probability	23%	48%	12%	4%	0.1%	0.005%	0.0003%
Value	1	5	10	20	30	-20	-50
Score	0.23	2.4	1.2	0.8	0.03	-0.001	-0.00015
Engagement score = 4.65885							

Figure 10.4: Calculate the engagement score

Data Preparation

Data engineering

It is generally helpful to understand what raw data is available before going on to engineer predictive features. Here, we assume the following types of raw data are available:

- Users
- Posts
- User-post interactions
- Friendship

Users

The user data schema is shown below.

ID	Username	Age	Gender	City	Country	Language	Time zone
----	----------	-----	--------	------	---------	----------	-----------

Table 10.2: User data schema

Posts

Table 10.3 shows post data.

Author ID	Textual Content	Hashtags	Mentions	Images or videos	Timestamp
5	Today at our fav place with my best friend	life_is_good, happy	hs2008	-	1658450539
1	It was the best trip we ever had	Travel, Maldives	Alexish, shan.tony	htcdn.mysite.com/maldives.jpg	1658451341
29	Today I had a bad experience I would like to tell you about. I went...	-	-	-	1658451365

Table 10.3: Post data

User-post interactions

Table 10.4 shows user-post interaction data.

User ID	Post ID	Interaction type	Interaction value	Location (lat, long)	Timestamp
4	18	Like	-	38.8951 -77.0364	1658450539
4	18	Share	User 9	41.9241 -89.0389	1658451365
9	18	Comment	You look amazing	22.7531 47.9642	1658435948
9	18	Block	-	22.7531 47.9642	1658451849
6	9	Impression		37.5189 122.6405	1658821820

Table 10.4: User-post interaction data

Friendship

The friendship table stores data of connections between users. We assume users can specify their close friends and family members. Table 10.5 shows examples of friendship data.

User ID 1	User ID 2	Time when friendship was formed	Close friend	Family member
28	3	1558451341	True	False
7	39	1559281720	False	True
11	25	1559312942	False	False

Table 10.5: Friendship data

Feature engineering

In this section, we engineer predictive features and prepare them for the model. In particular, we engineer features from each of the following categories:

- Post features
- User features
- User-author affinities

Post features

In practice, each post has many attributes. We cannot cover everything, so only discuss the most important ones.

- Textual content
- Images or videos
- Reactions
- Hashtags
- Post's age

Textual content

What is it? This is the textual content - the main body - of a post.

Why is it important? Textual content helps determine what the post is about.

How to prepare it? We preprocess textual content and use a pre-trained language model to convert the text into a numerical vector. Since the textual content is usually in the form of sentences and not a single word, we use a context-aware language model such as BERT [4].

Images or videos

What is it? A post may contain images or videos.

Why is it important? We can extract important signals from images. For example, an image of a gun may indicate that a post is unsafe for children.

How to prepare it? First, preprocess the images or videos. Next, use a pre-trained model to convert the unstructured image/video data into an embedding vector. For example, we can use ResNet, [5] or the recently introduced CLIP model [6] as the pre-trained model.

Reactions

What is it? This refers to the number of likes, shares, replies, etc., of a post.

Why is it important? The number of likes, shares, hides, etc., indicates how engaging users find a post to be. A user is more likely to engage with a post with thousands of likes than a post with ten likes.

How to prepare it? These values are represented by numerical values. We scale these numerical values to bring them into a similar range.

Hashtags

Why is it important? Users use hashtags to group content around a certain topic. These hashtags represent the topics to which a post relates. For example, a post with the hashtag "#women_in_tech" indicates the content relates to technology and females, so the model may decide to rank it higher for people who are interested in technology.

How to prepare it? The detailed steps to preprocess text are already explained in Chapter 4, YouTube Video Search, so we will only focus on the unique steps for preparing hashtags.

- **Tokenization:** Hashtags like "lifeisgood" or "programmer_lifestyle" contain multiple words. We use algorithms such as Viterbi [7] to tokenize the hashtags. For instance, "lifeisgood" becomes 3 words: "life" "is" "good".
- **Tokens to IDs:** Hashtags evolve quickly on social media platforms and change as trends come and go. A feature hashing technique is a good fit because it is capable of assigning indexes to unseen hashtags.
- **Vectorization:** We use simple text representation methods such as TF-IDF [8] or word2vec [9], instead of Transformer-based models, to vectorize hashtags. Let's take a look at why. Transformer-based models are useful when the context of the data is essential. In the case of hashtags, each one is usually a single word or a phrase, and often no context is necessary to understand what the hashtag means. Therefore, faster and lighter text representation methods are preferred.

Post's age

What is it? This feature shows how much time has passed since the author posted the content.

Why is it important? Users tend to engage with newer content.

How to prepare it? We bucketize the post's age into a few categories and use one-hot encoding to represent it. For example, we use the following buckets:

- 0: less than 1 hour
- 1 : between 1 to 5 hours
- 2: between 5 to 24 hours
- 3: between 1-7 days
- 4: between 7-30 days
- 5: more than a month

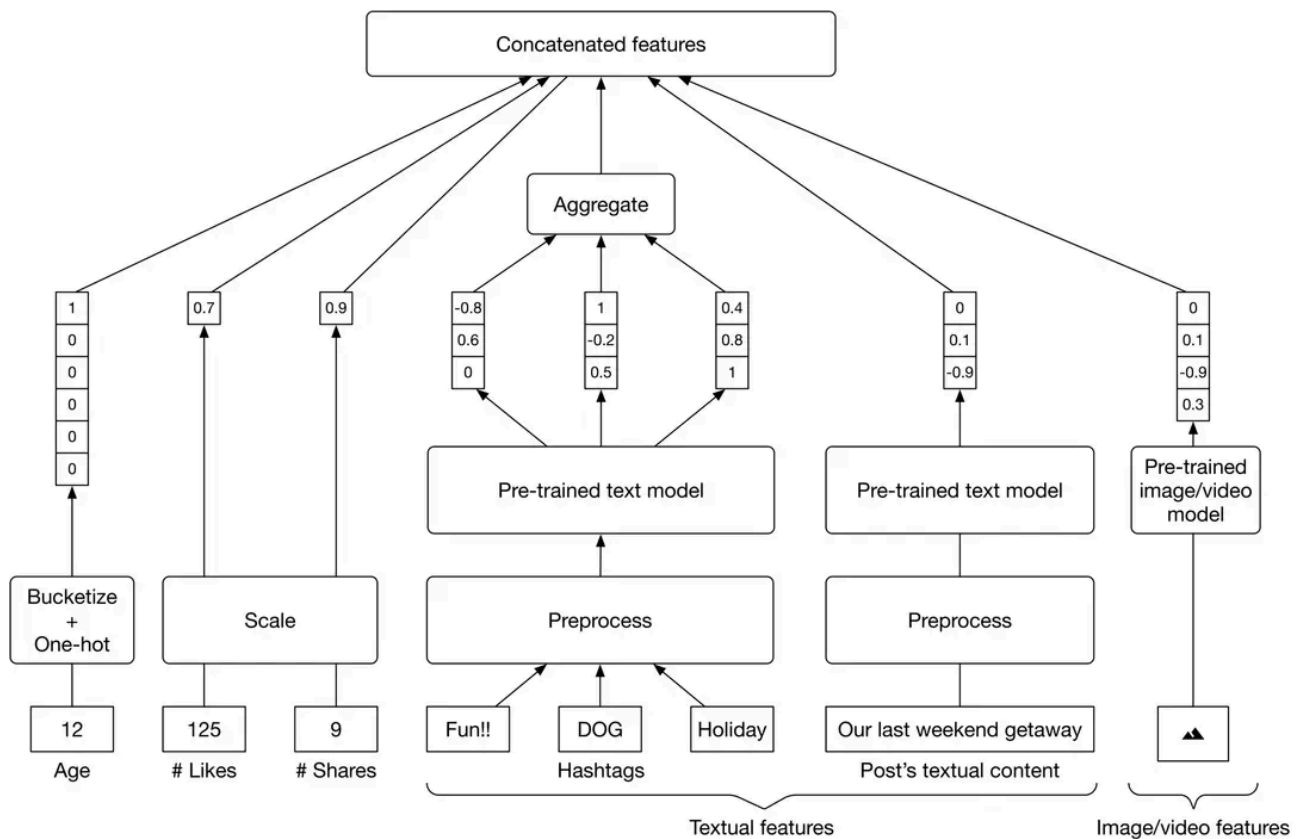


Figure 10.5: Post-related feature preparation

User features

Some of the most important user-related features are:

- Demographics: age, gender, country, etc
- Contextual information: device, time of the day, etc
- User-post historical interactions
- Being mentioned in the post

Since we have already discussed user demographic and contextual information in previous chapters, here we only examine the remaining two features.

User-post historical interactions

All posts liked by a user are represented by a list of post IDs. The same logic applies to shares and comments.

Why is it important? Users' previous engagements are usually helpful in determining their future engagements.

How to prepare it? Extract features from each post that the user interacted with.

Being mentioned in a post

What is it? This means whether or not the user is mentioned in a post. **Why is it important?** Users usually pay

more attention to posts that mention them.

How to prepare it? This feature is represented by a binary value. If a user is mentioned in the post, this feature is 1, otherwise 0.

Figure 10.6 summarizes feature preparation for users.

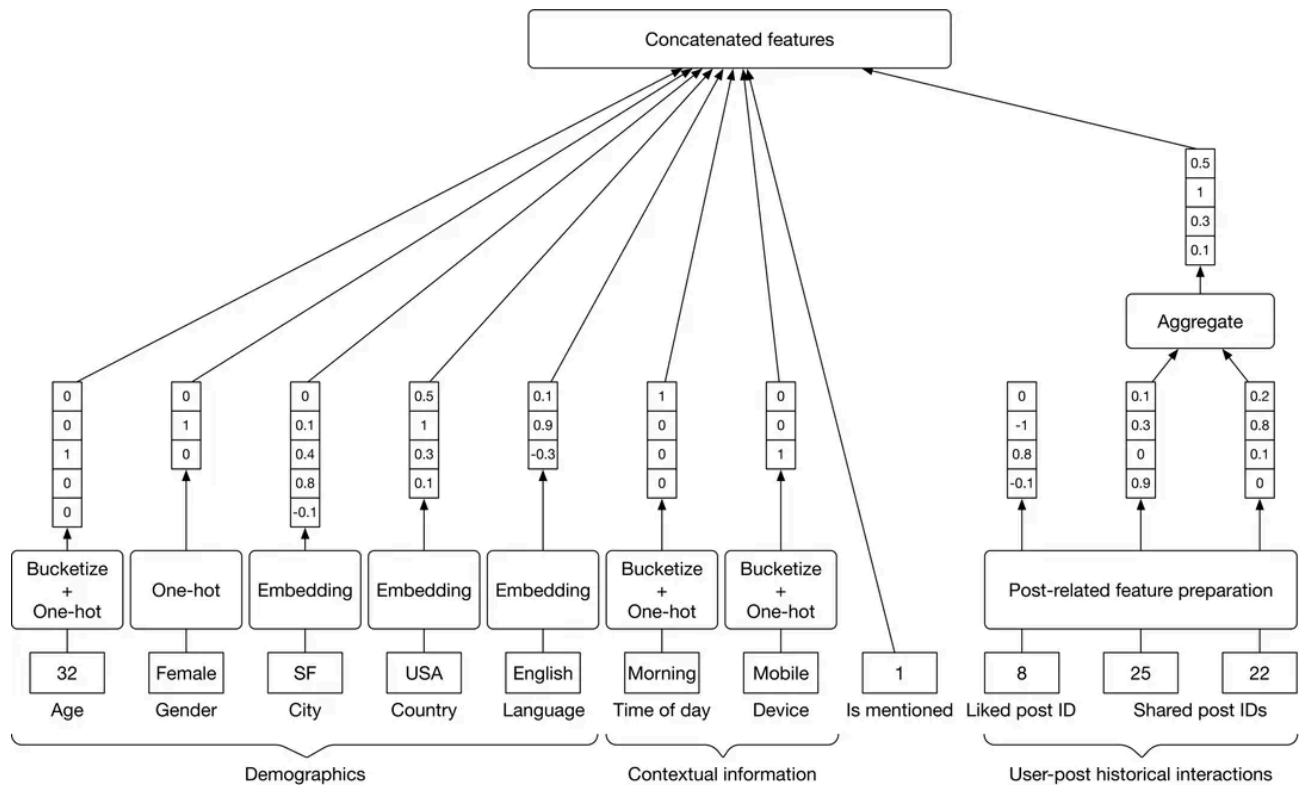


Figure 10.6: Feature preparation for user-related data

User-author affinities

According to studies, affinity features, such as the connection between the user and the author, are among the most important factors in predicting a user's engagement on Facebook [10]. Let's engineer some features to capture user-author affinities.

Like/click/comment/share rate

This is the rate at which a user reacted to previous posts by an author. For example, a like rate of 0.95 indicates that a user liked the posts from that author 95 percent of the time.

Length of friendship

The number of days the user and the author have been friends on the platform. This feature can be obtained from the friendship data.

Why is it important? Users tend to engage more with their friends.

Close friends and family

A binary value representing whether the user and the author have included each other in their close friends and family list.

Why is it important? Users pay more attention to posts by close friends and family members. Figure 10.7 summarizes features related to user-author affinities.

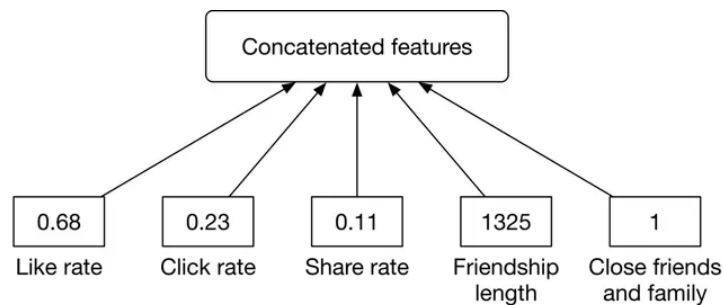


Figure 10.7: User-author affinity features

Model Development

Model selection

We choose neural networks for the following reasons:

- Neural networks work well with unstructured data, such as text and images.
- Neural networks allow us to use embedding layers to represent categorical features.
- With a neural network architecture, we can fine-tune pre-trained models employed during feature engineering. This is not possible with other models.

Before training a neural network, we need to choose its architecture. There are two architectural options for building and training our neural networks:

- N independent DNNs
- A multi-task DNN

Let's explore each one.

Option 1: N independent DNNs

In this option, we use N independent deep neural networks (DNN), one for each reaction. This is shown in Figure 10.8.

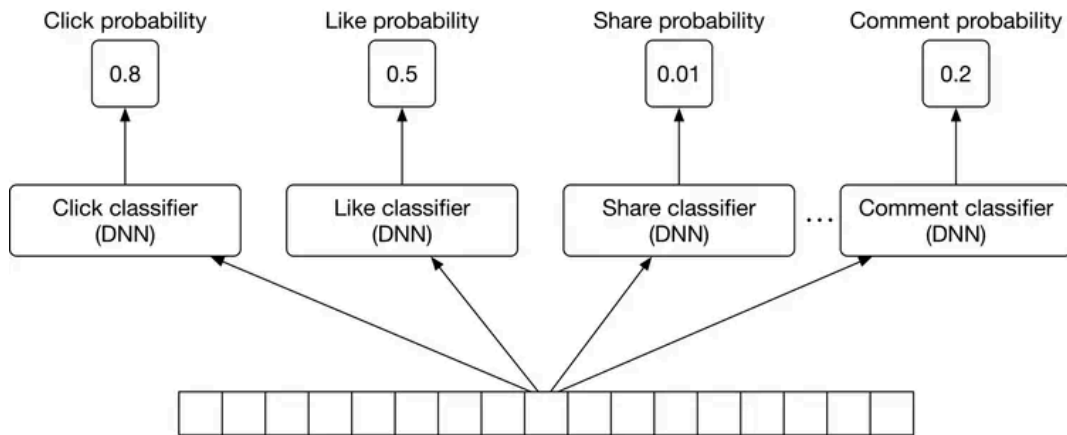


Figure 10.8: Using independent DNNs

This option has two drawbacks:

- **Expensive to train.** Training several independent DNNs is compute-intensive and time-consuming.
- **For less frequent reactions, there might not be enough training data.** This means our system is not able to predict the probabilities of infrequent reactions accurately.

Option 2: Multi-task DNN

To overcome these issues, we use a multi-task learning approach (Figure 10.9).

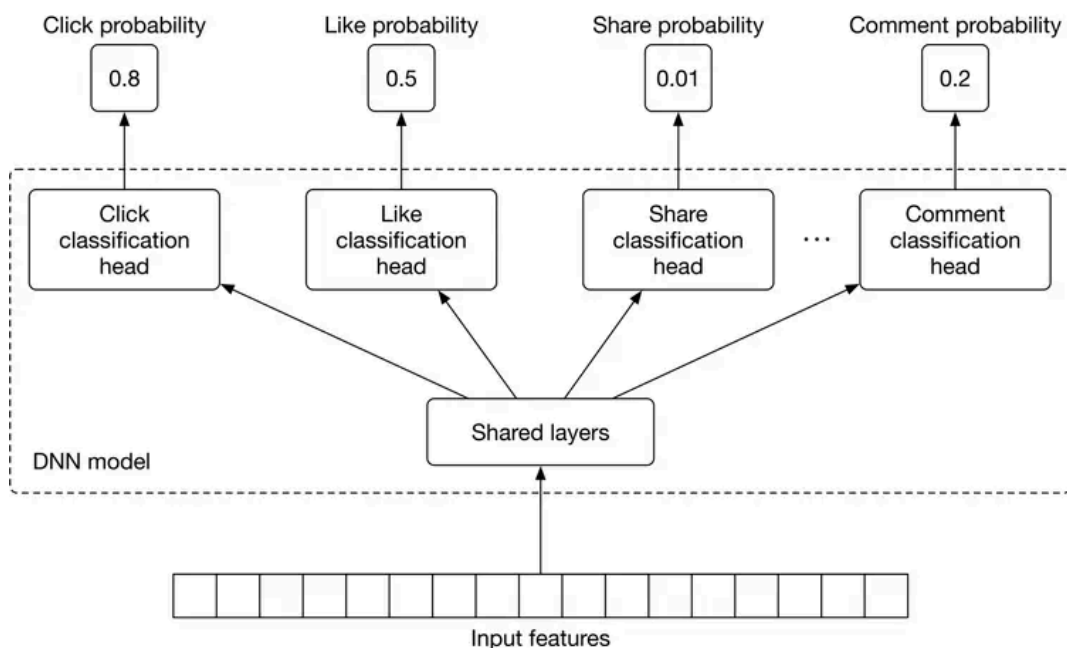


Figure 10.9: Multi-task DNN

We explained multi-task learning in Chapter 5, Harmful Content Detection, so we only briefly discuss it here. In summary, multi-task learning refers to the process of learning multiple tasks simultaneously. This allows the model to learn the similarities between tasks and avoid unnecessary computations. For a multi-task neural network model, it's essential to choose an appropriate architecture. The choice of architecture and the associated

hyperparameters are usually determined by running experiments. That means training and evaluating the model on different architectures and choosing the one which leads to the best result.

Improving the DNN architecture for passive users

So far, we have employed a DNN to predict reactions such as shares, likes, clicks, and comments. However, many users use the platform passively, meaning they do not interact much with the content on their timelines. For such users, the current DNN model will predict very low probabilities for all reactions, since they rarely react to posts. Therefore, we need to change the DNN architecture to consider passive users.

For this to work, we add two implicit reactions to the list of tasks:

- Dwell-time: the time a user spends on a post.
- Skip: if a user spends less than t seconds (e.g., 0.5 seconds) on a post, then that post can be assumed to have been skipped by the user.

Figure 10.10 shows the multi-task DNN model with the additional tasks.

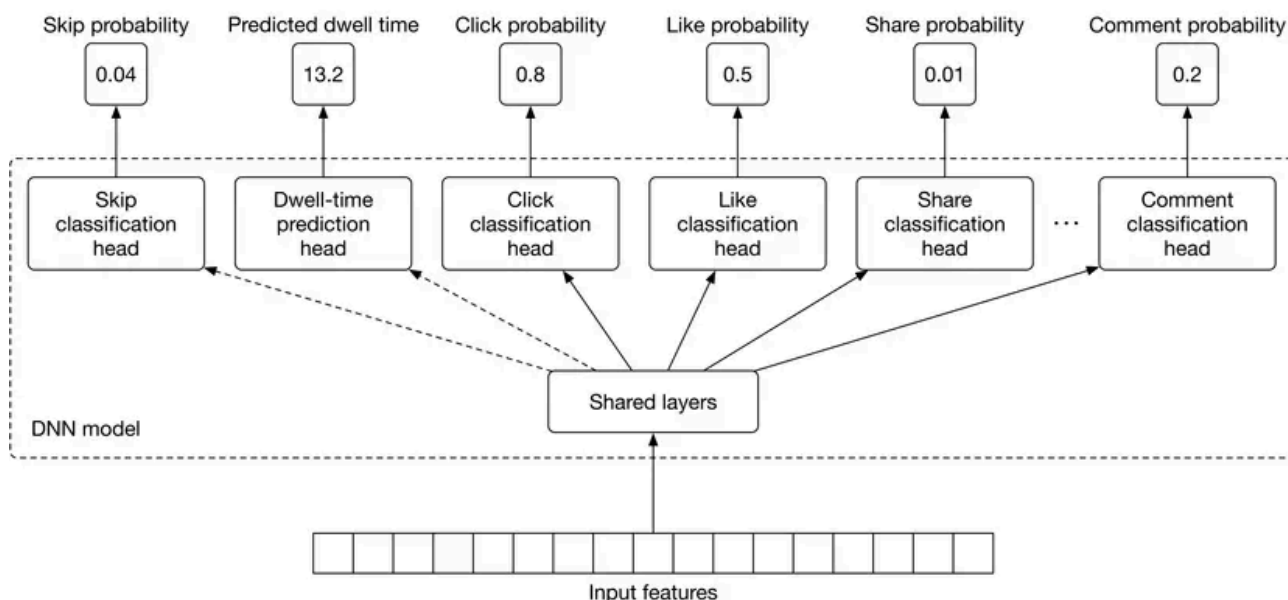


Figure 10.10: Multi-task DNN model with two new tasks

Model training

Constructing the dataset

In this step, we construct the dataset from raw data. Since the DNN model needs to learn multiple tasks, positive and negative data points are created for each (e.g., click, like, etc.)

We will use the reaction type of "like" as an example to explain how to create positive/negative data points. Each time a user likes a post, we add a data point to our dataset, compute $\langle \text{user}, \text{post} \rangle$ features, and then label it as positive.

To create negative data points, we chose impressions that didn't lead to a "like" reaction. Note that the number of

negative data points is usually much higher than positive data points. To avoid having an imbalanced dataset, we create negative data points to equal the number of positive data points. Figure 10.11 shows positive and negative data points for the "like" reaction.

#	User features	Post features	Affinity features	Label
1	1 0 1 0.8 0.1 1	0 1 1 0.4 0	0.9 0.6 0.3 8 0	Positive
2	0 0 0 0.4 0.9 0	1 1 0 0.3 1	1 0.9 0.8 120 1	Positive
3	1 1 0 0.1 0.5 0	0 1 0 0.9 1	0.1 0 0 2 0	Negative

Figure 10.11: Training data for like classification task

This same process can be used to create positive and negative labels for other reactions. However, because dwell-time is a regression task, we construct it differently. As shown in Figure 10.12, the ground truth label is the dwell-time of the impression.

#	User features	Post features	Affinity features	Dwell-time
1	0 0 0 0.1 0.9 1	1 1 0 0.1 1	0.6 0.6 0.3 0.2 5 0	8.1
2	1 1 1 0.9 0.1 0	1 1 0 0.8 0	0.1 0.9 0.3 0.1 3 1	11.5

Figure 10.12: Training data for dwell-time task

Choosing the loss function

Multi-task models are trained to learn multiple tasks simultaneously. This means we need to compute the loss for each task separately and then combine them to get an overall loss. Typically, we define a loss function for each task depending on the ML category of the task. In our case, we use a binary cross-entropy loss for each binary classification task, and a regression loss such as MAE [11], MSE [12], or Huber loss [13] for the regression task (dwell-time prediction). The overall loss is computed by combining task-specific losses, as shown in Figure 10.13.

$$Loss = \lambda L_{dwell} + L_{skip} + L_{like} + \dots + L_{share}$$

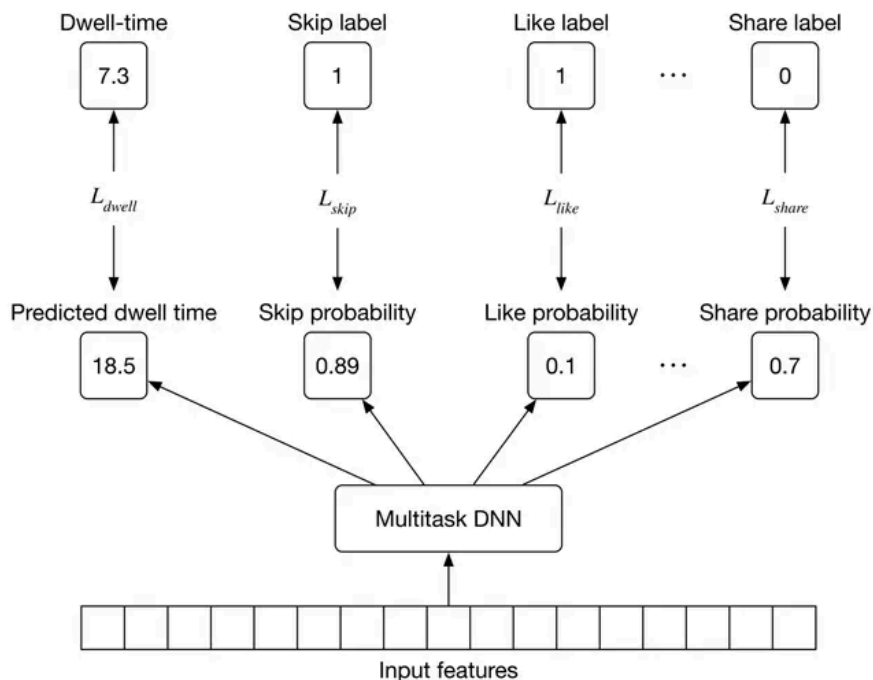



Figure 10.13: Training workflow

Evaluation

Offline metrics

During the offline evaluation, we measure the performance of our model in predicting different reactions. To evaluate the performance of an individual type of reaction, we can use binary classification metrics, such as precision and recall. However, these metrics alone may not be sufficient to understand the overall performance of a binary classification model. So, we use the ROC curve to understand the trade-off between the true positive rate and false positive rate. In addition, we compute the area under the ROC curve (ROC-AUC) to summarize the performance of the binary classification with a numerical value.

Online metrics

We use the following metrics to measure user engagement from various angles:

- Click-through rate (CTR)
- Reaction rate

- Total time spent
- User satisfaction rate found in a user survey

CTR. The ratio between the number of clicks and impressions.

$$CTR = \frac{\text{number of clicked posts}}{\text{number of impressions}}$$

A high CTR does not always indicate more user engagement. For example, users may click on a low-value clickbait post, and quickly realize it is not worth reading. Despite this limitation, it is an important metric to track.

Reaction rates. These are a set of metrics that reflect user reactions. For example, a like rate measures the ratio between posts liked and the total number of posts displayed in users' feeds.

$$\text{Like rate} = \frac{\text{number of liked posts}}{\text{number of impressions}}$$

Similarly, we track other reactions such as "share rate", "comment rate", "hide rate", "block rate", and "skip rate". These are stronger signals than CTR, as users have explicitly expressed a preference.

The metrics we discussed so far are based on user reactions. But what about passive users? These are users who tend not to react at all to the majority of posts. To capture the effectiveness of our personalized news feed system for passive users, we add the following two metrics.

Total time spent. This is the total time users spend on the timeline during a fixed period, such as 1 week. This metric measures the overall engagement of both passive and active users.

User satisfaction rate found in a user survey. Another way to measure the effectiveness of our personalized news feed system is to explicitly ask users for their opinion about the feed, or how engaging they find the posts. Since we seek explicit feedback, this is an accurate way to measure the system's effectiveness.

Serving

At serving time, the system serves requests by outputting a ranked list of posts. Figure 10.14 shows the architectural diagram of the personalized news feed system. The system comprises the following pipelines:

- Data preparation pipeline
- Prediction pipeline

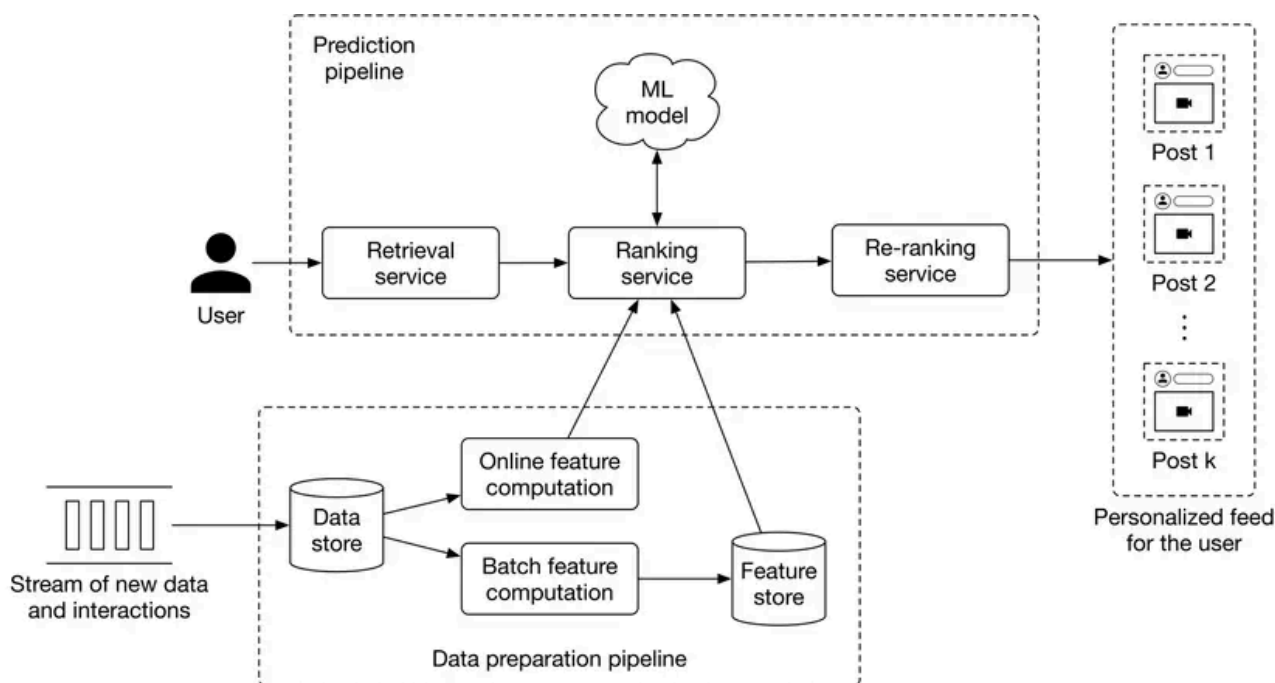


Figure 10.14: ML system design of a personalized news feed system

We do not go into detail about the data preparation pipeline because it is very similar to that described in Chapter 8, Ad Click Prediction in Social Platforms. Let's examine the prediction pipeline.

Prediction pipeline

The prediction pipeline consists of the following components: retrieval service, ranking service, and re-ranking service.

Retrieval service

This component retrieves posts that a user has not seen, or which have comments also unseen by them. To learn more about efficiently fetching unseen posts, read [14].

Ranking service

This component ranks the retrieved posts by assigning an engagement score to each one.

Re-ranking service

This service modifies the list of posts by incorporating additional logic and user filters. For example, if a user has explicitly expressed interest in a certain topic, such as soccer, this service assigns a higher rank to the post.

Other Talking Points

If there is time left at the end of the interview, here are some additional talking points:

- How to handle posts that are going viral [15].
- How to personalize the news feed for new users [16].

- How to mitigate the positional bias present in the system [17].
- How to determine a proper retraining frequency [18].

References

1. News Feed ranking in Facebook. <https://engineering.fb.com/2021/01/26/ml-applications/news-feed-ranking/>.
2. Twitter's news feed system. https://blog.twitter.com/engineering/en_us/topics/insights/2017/using-deep-learning-at-scale-in-twitters-timelines.
3. LinkedIn's News Feed system LinkedIn. <https://engineering.linkedin.com/blog/2020/understanding-feed-dwell-time>.
4. BERT paper. <https://arxiv.org/pdf/1810.04805.pdf>.
5. ResNet model. <https://arxiv.org/pdf/1512.03385.pdf>.
6. CLIP model. <https://openai.com/blog/clip/>.
7. Viterbi algorithm. https://en.wikipedia.org/wiki/Viterbi_algorithm.
8. TF-IDF. <https://en.wikipedia.org/wiki/Tf%E2%80%93idf>.
9. Word2vec. <https://en.wikipedia.org/wiki/Word2vec>.
10. Serving a billion personalized news feed. <https://www.youtube.com/watch?v=Xpx5RYNTQvg>.
11. Mean absolute error loss. https://en.wikipedia.org/wiki/Mean_absolute_error.
12. Means squared error loss. https://en.wikipedia.org/wiki/Mean_squared_error.
13. Huber loss. https://en.wikipedia.org/wiki/Huber_loss.
14. A news feed system design. <https://liuzhenglaichn.gitbook.io/system-design/news-feed/design-a-news-feed-system>.
15. Predict viral tweets. <https://towardsdatascience.com/using-data-science-to-predict-viral-tweets-615b0acc2e1e>.
16. Cold start problem in recommendation systems. [https://en.wikipedia.org/wiki/Cold_start_\(recommender_systems\)](https://en.wikipedia.org/wiki/Cold_start_(recommender_systems)).
17. Positional bias. <https://eugeneyan.com/writing/position-bias/>.
18. Determine retraining frequency. <https://huyenchip.com/2022/01/02/real-time-machine-learning-challenges-and-solutions.html#towards-continual-learning>.

11

People You May Know

Introduction

People You May Know (PYMK) is a list of users with whom you may want to connect based on things you have in common, such as a mutual friend, school, or workplace. Many social networks, such as Facebook, LinkedIn, and Twitter, utilize ML to power PYMK functionality.

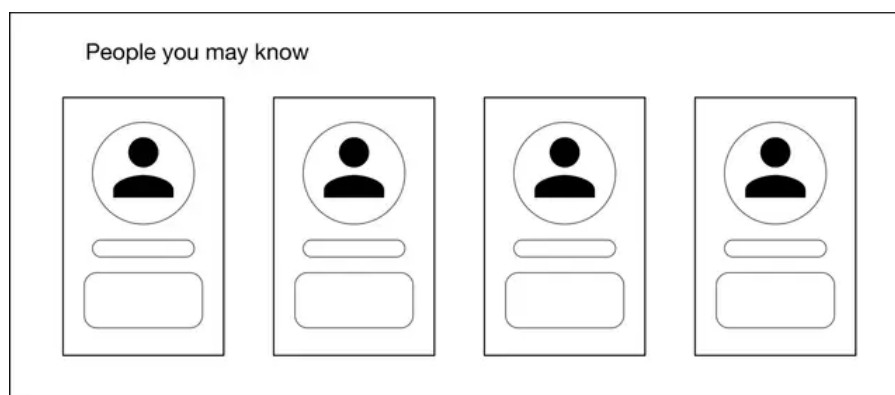


Figure 11.1: The PYMK feature

In this chapter, we will design a PYMK feature similar to LinkedIn's. The system takes a user as input and recommends a list of potential connections as output.

Clarifying Requirements

Here is a typical interaction between a candidate and an interviewer.

Candidate: Can I assume the motivation for building the PYMK feature is to help users discover potential connections and grow their network?

Interviewer: Yes, that's a good assumption.

Candidate: To recommend potential connections, a huge list of factors must be considered, such as location, educational background, work experience, existing connections, previous activities, etc. Should I focus on the most important factors, such as educational background, work experience, and the user's social context?

Interviewer: That sounds good.

Candidate: On LinkedIn, two people are friends if – and only if – each is a friend of the other. Is that correct?

Interviewer: Yes, friendship is symmetrical. When someone sends a connection request to another user, the recipient needs to accept the request for the connection to be made.

Candidate: What's the total number of users on the platform? How many of them are daily active users?

Interviewer: We have nearly 1 billion users and 300 million daily active users.

Candidate: How many connections does an average user have?

Interviewer: 1,000 connections.

Candidate: The social graph of most users is not very dynamic, meaning their connections don't change significantly over a short period. Can I make this assumption when designing PYMK?

Interviewer: That's an excellent point. Yes, it's a reasonable assumption.

Let's summarize the problem statement. We are asked to design a PYMK system similar to LinkedIn's. The system takes a user as input and recommends a ranked list of potential connections as output. The motivation for building the system is to enable users to discover new connections more easily and grow their networks. There are 1 billion total users on the platform, and a user has 1,000 connections on average.

Frame the problem as an ML task

Defining the ML objective

A common ML objective in PYMK systems is to maximize the number of formed connections between users. This helps users to grow their networks quickly.

Specifying the system's input and output

The input to the PYMK system is a user, and the outputs are a list of connections ranked by relevance to the user. This is shown in Figure 11.2.

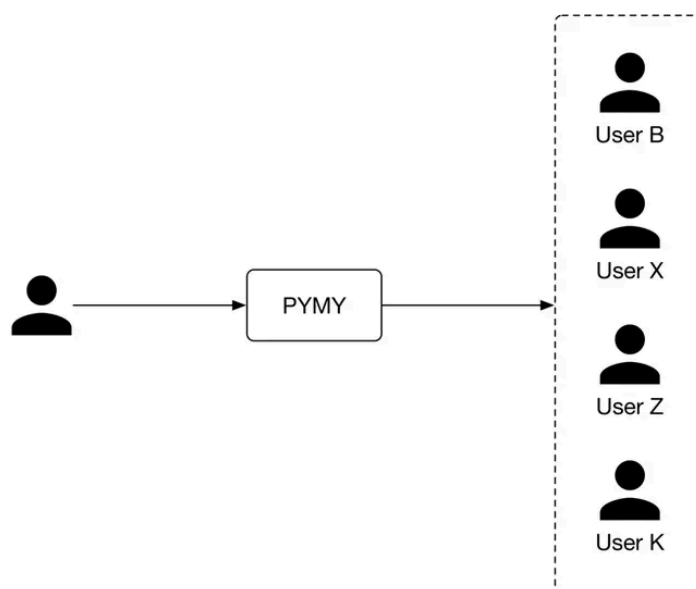


Figure 11.2: PYMK system's input-output

Choosing the right ML category

Let's examine two approaches commonly used to build PYMK: pointwise Learning to Rank (LTR) and edge prediction.

Pointwise LTR

In this approach, we frame PYMK as a ranking problem and use a pointwise LTR to rank users. In pointwise LTR, as Figure 11.3 shows, we employ a binary classification model which takes two users as input and outputs the probability of the given pair forming a connection.

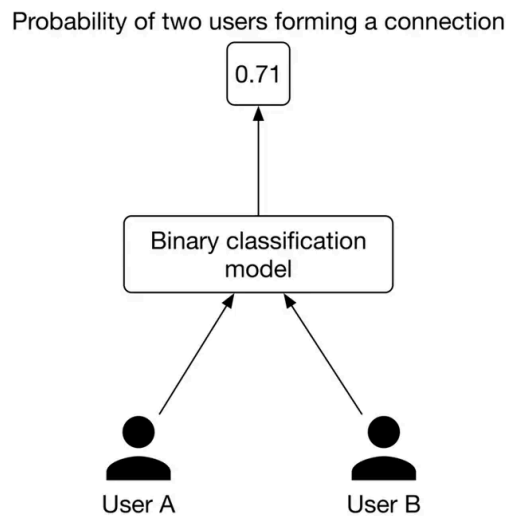


Figure 11.3: Binary classification with two input users

However, this approach has a major drawback; since the model's inputs are two distinct users, it doesn't consider the available social context. While this does simplify things, leaving out information about a user's connections might make predictions less accurate.

Let's analyze an example to understand how social context can provide very important insights. Imagine we want to predict whether or not $\langle \text{user A}, \text{user B} \rangle$ is a potential connection.

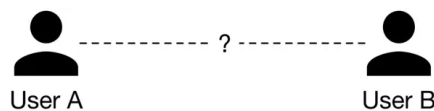


Figure 11.4: Can user A and user B form a potential connection?

By looking at their one-hop neighborhood (connections of user A or user B), we gain more information to determine if $\langle \text{user A}, \text{user B} \rangle$ is a potential connection. As shown in Figure 11.5, consider two different scenarios.

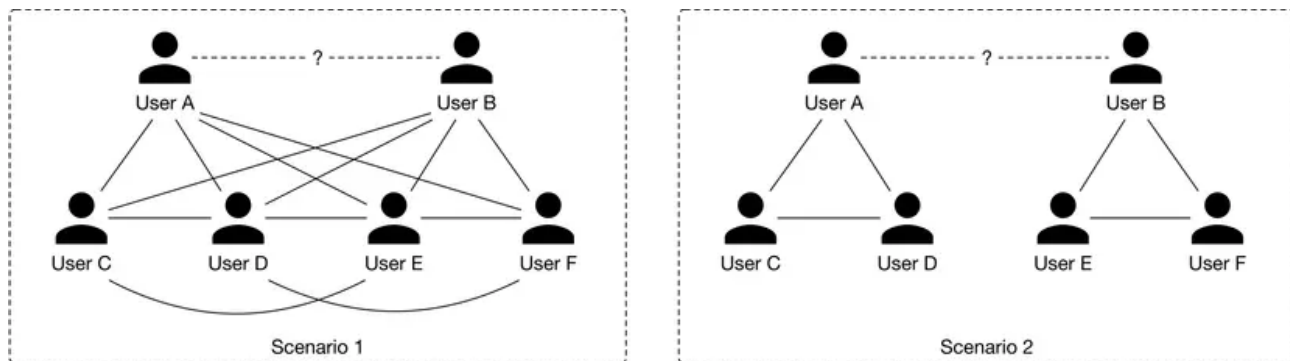


Figure 11.5: Two different scenarios of one-hop neighborhoods

In scenario 1, user A and user B each have four mutual connections, and there are mutual connections between users C, D, E, and F.

In scenario 2, user A and user B each have two friends, and there's no connection between user A and user B's connections.

By looking at their one-hop neighborhood, you might expect that $\langle \text{user } A, \text{user } B \rangle$ is more likely to form a connection in scenario 1 rather than in scenario 2. In practice, we can even leverage two-hop or three-hop neighborhoods to capture more useful information from the social context.

Before discussing the second approach, let's understand how graphs store structural data, such as the social context, and which machine learning tasks can be performed on graphs.

In general, a graph represents relations (edges) between a collection of entities (nodes). The entire social context can be represented by a graph, where each node represents a user, and an edge between two nodes indicates a formed connection between two users. Figure 11.6 shows a simple graph with four nodes and three edges.

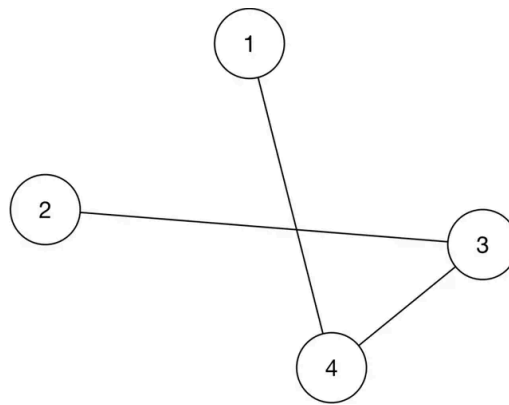


Figure 11.6: A simple graph

There are three general types of prediction tasks that can be performed on structured data represented by graphs:

- **Graph-level prediction.** For example, given a chemical compound as a graph, we predict whether the chemical compound is an enzyme or not.
- **Node-level prediction.** For example, given a social network graph, we predict if a specific user (node) is a spammer.
- **Edge-level prediction.** Predict if an edge is present between two nodes. For example, given a social network graph, we predict if two users are likely to connect.

Let's look at the edge prediction approach for building the PYMK system.

Edge prediction

In this approach, we supplement the model with graph information. This enables the model to rely on the additional knowledge extracted from the social graph, to predict whether an edge exists between two nodes.

More formally, we use a model that takes the entire social graph as input, and predicts the probability of an edge existing between two specific nodes. To rank potential connections for user A, we compute the edge probabilities between user A and other users, and use these probabilities as the ranking criteria.

In addition to the typical features that the model utilizes, the model also relies on additional knowledge extracted from the social graph to predict whether an edge exists between two nodes.

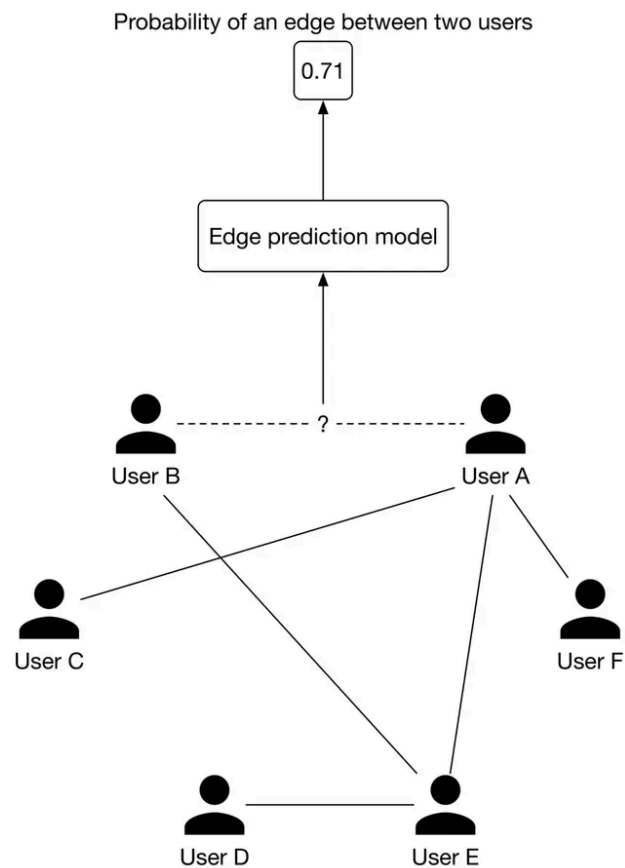


Figure 11.7: Binary classification with graph input

Data Preparation

Data engineering

In this section, we discuss the raw data available:

- Users
- Connections
- Interactions

Users

In addition to users' demographic data, we have information about their educational and work backgrounds, skills, etc. Table 11.1 shows an example of a user's educational background data. There might be similar tables to store work experiences, skills, etc.

User ID	School	Degree	Major	Start date	End date
11	Waterloo	M.Sc	Computer Science	August 2015	May 2017
11	Harvard	M.Sc	Physics	May 2004	August 2006
11	UCLA	Bachelors	Electrical Engineering	Sep 2022	-

Table 11.1: Users' educational background data

One challenge with this type of raw data is that a specific attribute can be represented in different forms. For example, "computer science" and "CS" have the same meaning, but the text differs. So, it's important to standardize the raw data during the data engineering step so we don't treat different forms of a single attribute differently. There are various approaches to standardizing the raw data. For example:

- Force users to select attributes from a predefined list.
- Use heuristics to group different representations of an attribute.
- Use ML-based methods such as clustering [1] or language models to group similar attributes.

Connections

A simplified example of connection data is shown in Table 11.2. Each row represents a connection between two users and when the connection was formed.

User ID 1	User ID 2	Timestamp when the connection was formed
28	3	1658451341
7	39	1659281720
11	25	1659312942

Table 11.2: Connection data

Interactions

There are different types of interactions: a user sends a connection request, accepts a request, follows another user, searches for an entity, views a profile, likes or reacts to a post, etc. Note, in practice, we may store interaction data in different databases, but for simplicity, here, we include everything in a single table.

User ID	Interaction type	Interaction value	Timestamp
11	Connection request	user_id_8	1658450539
8	Accepted connection	user_id_11	1658451341
11	Comment	[user_id_4, Very insightful]	1658451365
4	Search	"Scott Belsky"	1658435948
11	Profile view	user_id_21	1658451849

Table 11.3: Interaction data

Feature engineering

To determine potential connections for a user (e.g., user A), the model needs to utilize user A's information, such as age, gender, etc. In addition, the affinities between user A and other users are useful. In this section, we discuss some of the most important features.

User features

Demographics: age, gender, city, country, etc.

Demographic data helps determine if two users are likely to form a connection. Users tend to connect with others who have similar demographics.

It's common to have missing values in demographic data. To learn more about how to handle missing values, refer to the "Introduction and Overview" chapter.

The numbers of connections, followers, following, and pending requests

This information is important as users are more likely to connect with someone with lots of followers or connections, compared to a user with few connections.

Account's age

Accounts created very recently are less reliable than those that have existed for longer. For example, if an account was created yesterday, it's more likely to be a spam account. So, it may not be a good idea to recommend it to users.

The number of received reactions

These are numerical values representing the total number of reactions received, such as likes, shares, and comments over a certain period, like one week. Users tend to connect with more active users on the platform, who receive more interactions from other users.

User-user affinities

The affinity between two users is a good signal to predict if they will connect. Let's look at some important features which capture user-user affinities.

Education and work affinity

- **Schools in common:** Users tend to connect with others who attended the same school.
- **Contemporaries at school:** Overlapping years at school increases the likelihood of two users connecting. For example, users might want to connect with someone who attended school X the same time they did.
- **Same major:** A binary feature representing whether two users had the same major in school.
- **Number of companies in common:** Users may connect with people who have worked at the same companies.
- **Same industry:** A binary feature representing whether the two users work in the same industry.

Social affinity

- **Profile visits:** The number of times a user looks at the profile of another user.
- **Number of connections in common, aka mutual connections:** If two users have many common connections, they are more likely to connect. This feature is one of the most important predictive features [2].
- **Time discounted mutual connections:** This feature weighs mutual connections by how long they have existed. Let's go through an example to understand the reasoning behind this feature.

Imagine we want to determine whether user B is a potential connection for user A. Consider two scenarios: in scenario 1, user A's connections were formed very recently, whereas in scenario 2, the connections were formed a long time ago. This is shown in Figure 11.8.

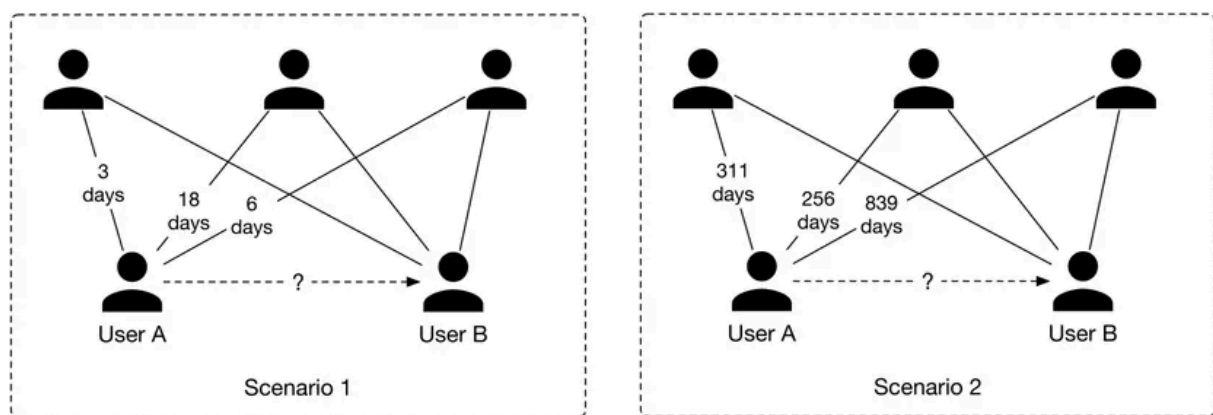


Figure 11.8: Comparing recent connections to old connections

In scenario 1, user A's network has grown recently, meaning it's more likely user A will connect with user B. Meanwhile, in scenario 2, the chances are that user A is aware of user B but has decided not to connect.

Model Development

Model selection

Earlier, we formulated the PYMK problem as an edge prediction task, where a model takes the social graph as input and predicts the probability of an edge existing between two users. To handle the edge prediction task, we choose a model that can process graph inputs. Graph neural networks (GNNs) are designed to operate on graph data. Let's take a closer look.

GNNs

GNNs are neural networks that can be directly applied to graphs. They provide an easy way to perform graph-level, node-level, and edge-level prediction tasks.

As shown in Figure 11.9, GNN takes a graph as input. This input graph contains attributes associated with nodes and edges. For example, the nodes can store information such as age, gender, etc., while the edges can store user-user characteristics, such as the number of common schools and workplaces, connection age, etc. Given the input graph and associated attributes, the GNN produces node embeddings for each node.

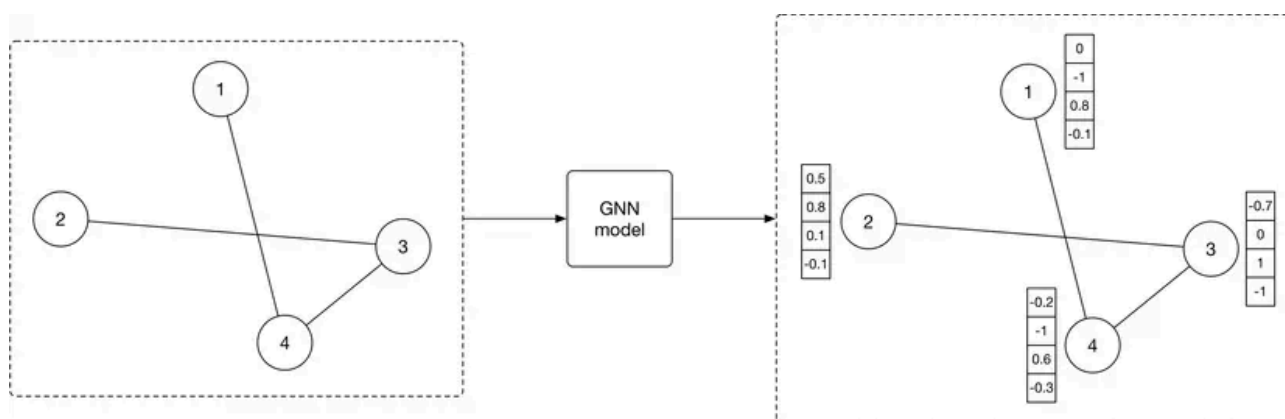


Figure 11.9: GNN model produces node embeddings for each graph node

Once the node embeddings are produced, they are used to predict how likely two nodes will form a connection using a similarity measure, such as dot product. For example, as shown in Figure 11.10, we compute the dot product between the embeddings of node 2 and node 4 to predict whether there is an edge between them.

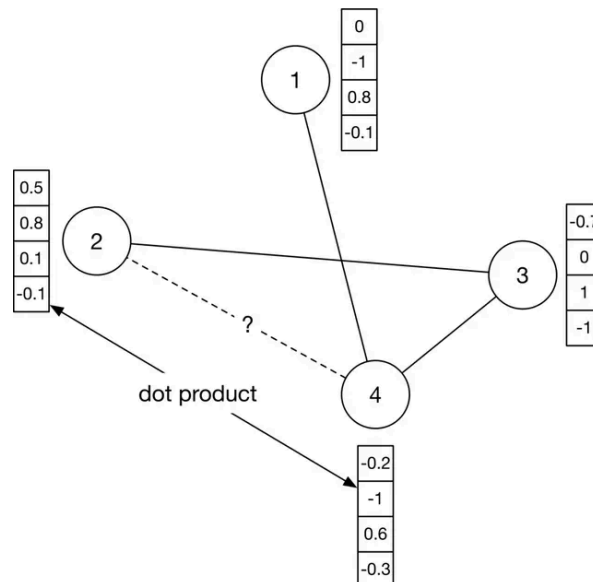


Figure 11.10: Predict how likely there is an edge between nodes 2 and 4

Many GNN-based architectures, such as GCN [3], GraphSAGE [4], GAT [5], and GIT [6], have been developed in recent years. These variants have different architectures and different levels of complexity. To determine which architecture works best, extensive experimentation is required. To gain a deeper understanding of GNN-based architectures, refer to [7]

Model training

To train a GNN model, we provide the model with a snapshot of the social graph at time t . The model predicts the connections which will form at time $t + 1$. Let's examine how to construct the training data.

Constructing the dataset

To construct the dataset, we do the following:

1. Create a snapshot of the graph at time t

2. Compute initial node features and edge features of the graph
3. Create labels

1. Create a snapshot of the graph at time t . The first step in constructing training data is to create input for the model. Since a GNN model expects a social graph as input, we create a snapshot of the social graph at time t using the available raw data. Figure 11.11 shows an example of the graph at time t .

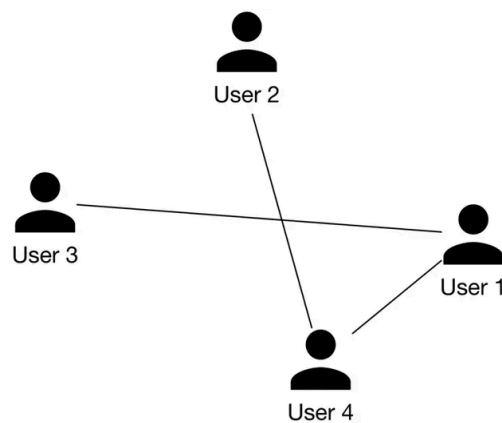


Figure 11.11: A snapshot of the social graph at time t

2. Compute initial node features and edge features of the graph. As shown in Figure 11.12, we extract the user's features, such as age, gender, account age, number of connections, etc. These are used as the nodes' initial feature vectors.

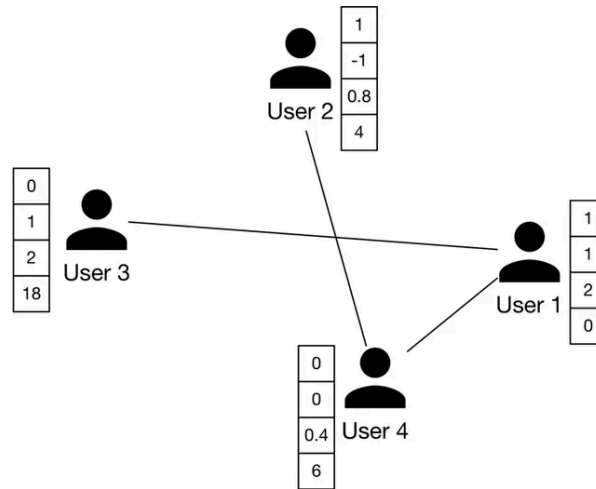


Figure 11.12: Initial node features

Similarly, we extract user-user affinity features and employ them as the initial feature vectors of the edges. As shown in Figure 11.13, there is an edge between user 2 and user 4. $E_{2,4}$ represents the initial feature vector which captures information such as the number of mutual connections, profile visits, overlapping time at schools in common, etc.

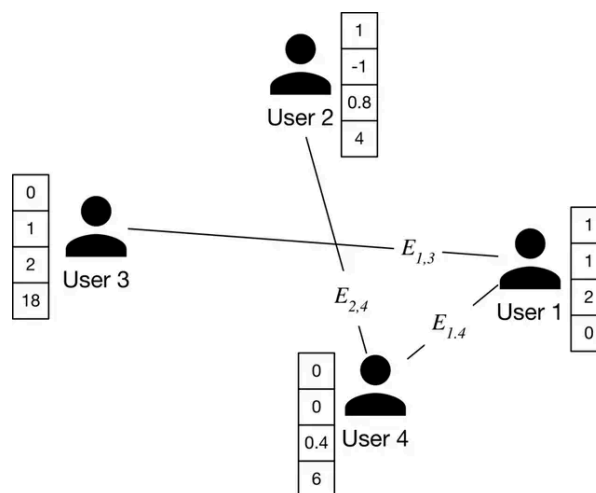


Figure 11.13: Initial edge features

3. Create labels

In this step, we create labels that the model is expected to predict. We use the graph snapshot at time $t + 1$ to

determine positive or negative labels. Let's take a look at a concrete example.

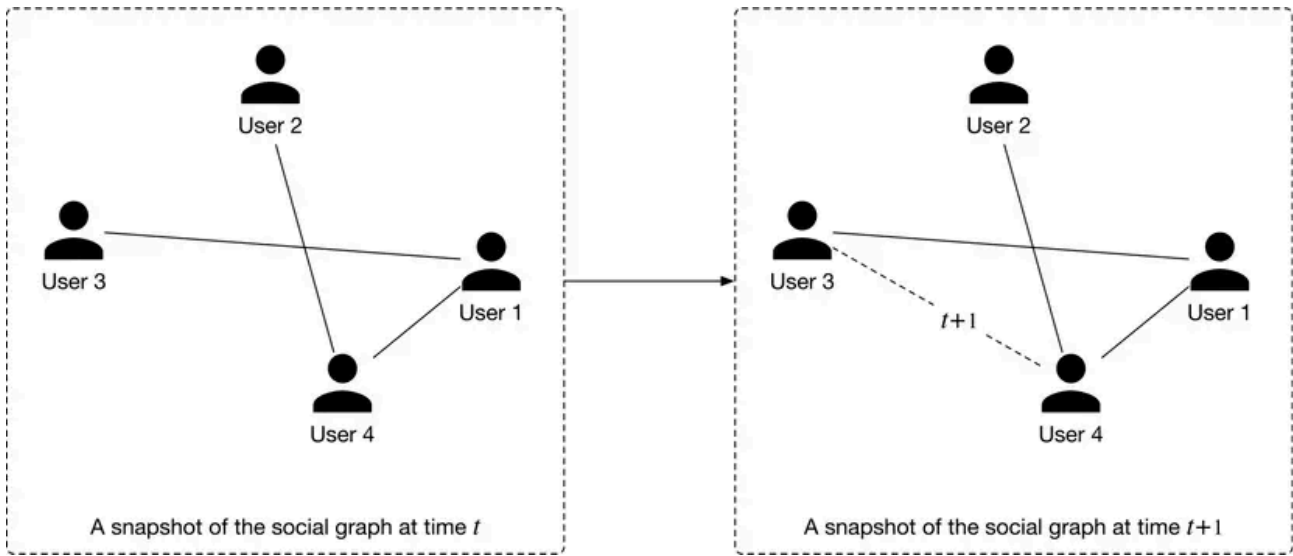


Figure 11.14: Newly formed edges from time t to $t + 1$

As shown in Figure 11.14, positive and negative labels are created depending on whether a new edge forms at $t + 1$. In particular, we label a pair of nodes as positive when they connect at $t + 1$. Otherwise, they are labeled as negative.

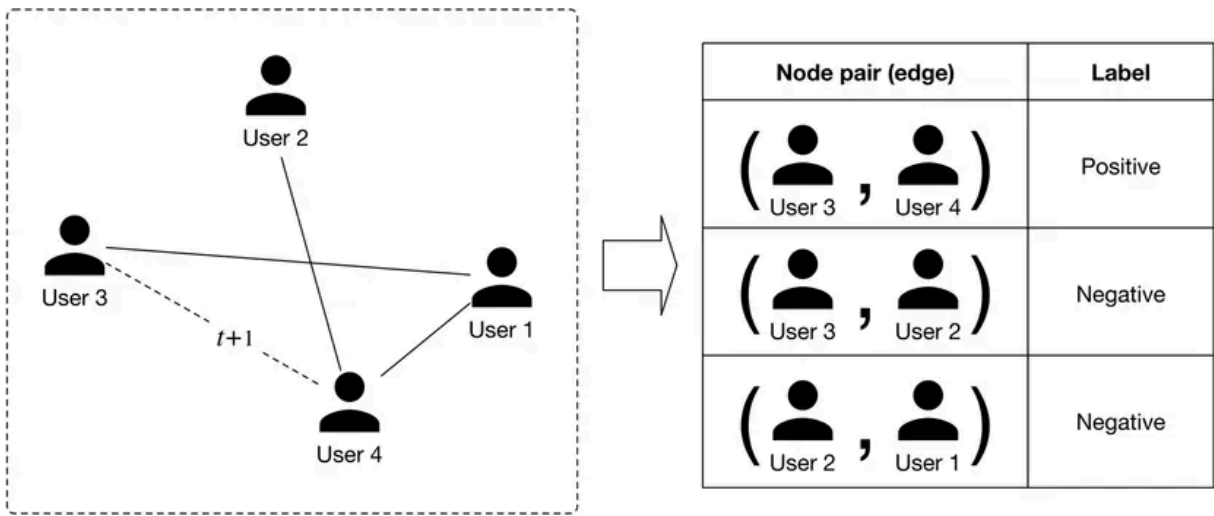


Figure 11.15: Creating positive and negative labels

Choosing the loss function

Once the input graph and labels are created, we are ready to train the GNN model. A detailed explanation of how GNN training works and which loss functions to employ is beyond the scope of this book. To learn more about these, see [7].

Evaluation

Offline metrics

During the offline evaluation, we evaluate the performance of the GNN model and the PYMK system.

GNN model

Since the GNN model predicts the presence of edges, we can think of it as a binary classification model. ROC-AUC metric is used to measure the performance of the model.

PYMK system

We extensively discuss choosing the right offline metrics for ranking and recommendation systems in previous chapters, so don't go into detail here. In our system, a user will either connect with a recommended connection or discard it. Due to this binary nature (connect or not), mAP is a good choice.

Online metrics

In practice, companies track lots of online metrics to measure the impact of PYMK systems. Let's explore two of the most important metrics:

- The total number of connection requests sent in the last X days
- The total number of connection requests accepted in the last X days

The total number of connection requests sent in the last X days. This metric helps us understand if the

model increases or decreases the number of connection requests. For example, if a model leads to a 5% increase in the total number of sent connection requests, we can assume the model has a positive impact on the business objective.

However, this metric has a major drawback. A new connection forms between two users only when the recipient accepts a request to connect. For example, a user may send 1,000 connection requests, but recipients accept only a small percentage. This metric might not correctly reflect the actual growth of the users' network. Now, let's address this drawback with the next metric.

The total number of connection requests accepted in the last X days. As a new connection forms only when the recipient accepts the sender's request, this metric accurately reflects the real growth of the users' network.

Serving

At serving time, the PYMK system efficiently recommends a list of potential connections to a given user. In this section, we explain why speed optimization is needed and introduce some techniques to make PYMK efficient.

Then, we propose a design in which different components work together to serve requests.

Efficiency

As discussed in the requirement gathering section, the total number of users on the platform is 1 billion, which indicates we need to sort through 1 billion embeddings to find potential connections for a single user. To make things even more challenging, the algorithm needs to be run for each user. Unsurprisingly, this is impractical at our scale. To mitigate the issue, two common techniques are used: 1) utilizing friends of friends (FoF) and 2) pre-compute PYMK.

Utilizing FoF

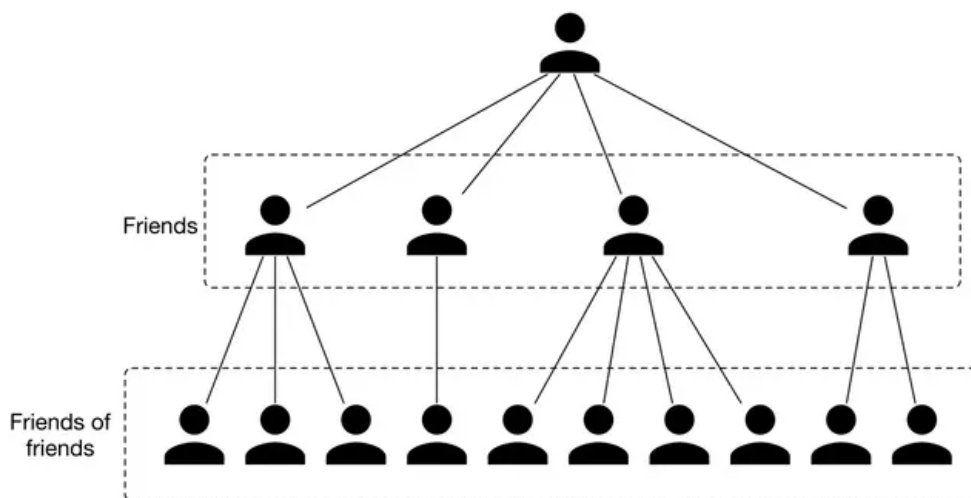


Figure 11.16: FoF of a user

According to a Meta study [2], 92% of new friendships are formed via FoF. This technique uses a user's FoF to narrow down the search space.

As previously mentioned, a user has 1,000 friends on average. That means a user has 1 million (1000×1000)

FoF, on average. This reduces the search space from 1 billion to 1 million.

Pre-compute PYMK

Let's take a step back and consider adopting online or batch predictions.

Online prediction In PYMK, online prediction refers to generating potential connections in real-time when a user loads the homepage. In this approach, we don't generate recommendations for inactive users. Since recommendations are calculated "on the fly", if computing the recommendations takes a long time, it creates a poor user experience.

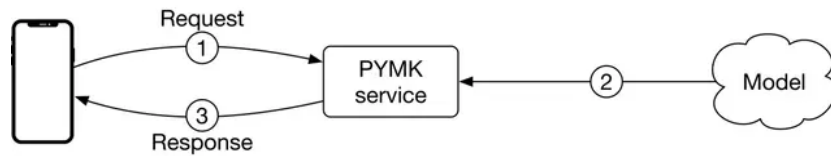


Figure 11.17: Online prediction in PYMK

Batch prediction Batch prediction means the system pre-computes potential connections for all users and stores them in a database. When a user loads the homepage, we fetch pre-computed recommendations directly, so from the end user's standpoint, the recommendation is instantaneous. The downside of batch prediction is that we may end up with unnecessary computations. Imagine 20% of users log in daily. If we generate recommendations for every user daily, then the computing power used to generate 80% of recommendations will be wasted.

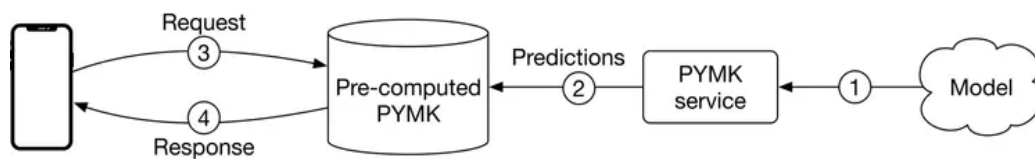


Figure 11.18: Batch prediction in PYMK

Which option do we choose: online or batch? We recommend batch prediction for two reasons. First, based on the requirements gathered, there are 300 million daily active users. Computing PYMK for all 300 million users on the fly may be too slow for a quality user experience.

Second, as the social graph in PYMK does not evolve quickly, the pre-computed recommendations remain relevant for an extended period. For example, we can keep PYMK recommendations for seven days and then re-compute them. The time window can be shortened (for instance, by one day) for newer users because their networks tend to grow faster.

In a social network, a user may not want to see the same set of recommended connections repeatedly. To support this, we can pre-compute more connections than needed and only display those a user hasn't seen before.

ML system design

Figure 11.19 shows the PYMK ML system design. The design comprises two pipelines:

- PYMK generation pipeline
- Prediction pipeline

Let's inspect each.

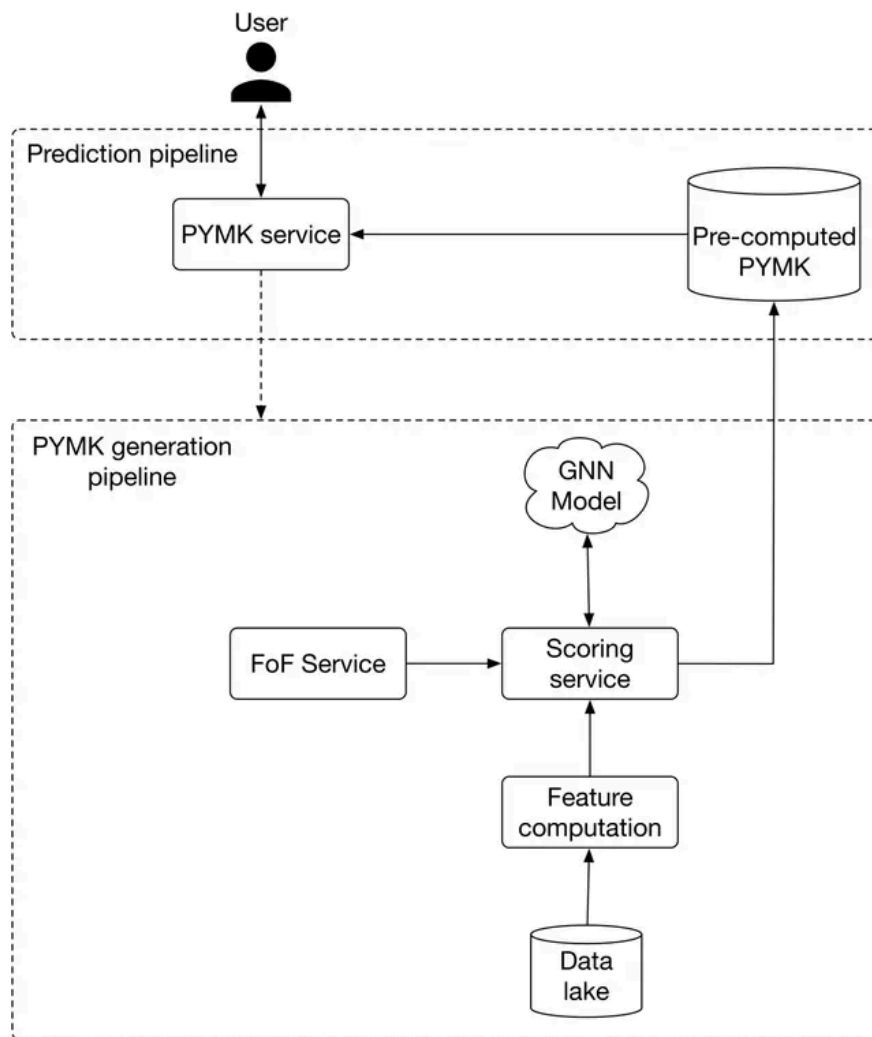


Figure 11.19: PYMK ML system design

PYMK generation pipeline

This pipeline is responsible for generating PYMK for all users and storing the results in a database. Let's take a closer look at this pipeline.

First, for a specific user, the FoF service narrows down the connections into a subset of candidate connections (2-hop neighbors). This is shown in Figure 11.20.

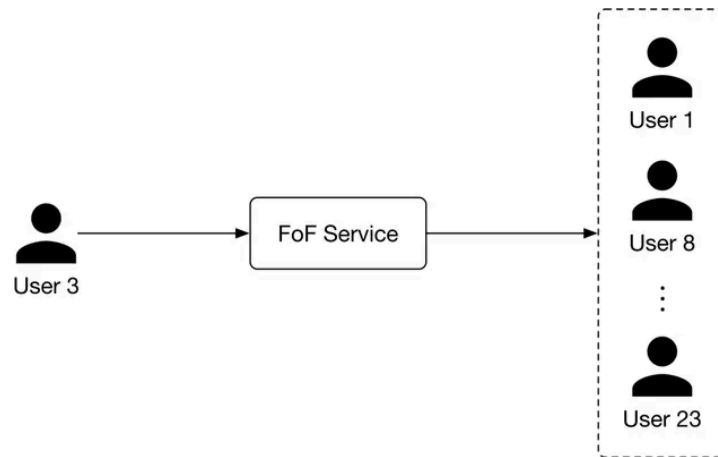


Figure 11.20: FoF service input-output

Next, the scoring service takes the candidate connections produced by the FoF service, scores each of them using the GNN model, then generates a ranked list of PYMK for the user. The PYMK is stored in a database. When a user request is made, we can simply pull their individual PYMK list directly from the database. This flow is shown in Figure 11.21.

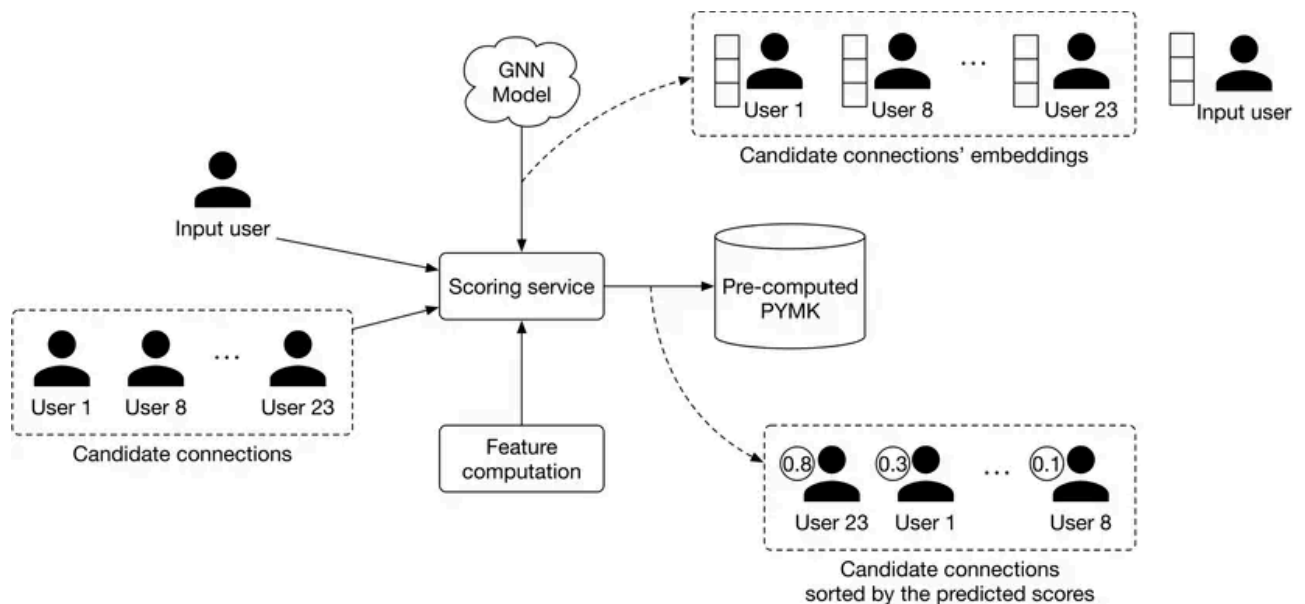


Figure 11.21: Scoring service input-output

Prediction pipeline

When a request arrives, the PYMK service first looks at the pre-computed PYMKs to see if recommendations exist. If they do, recommendations are fetched directly. If not, it sends a one-time request to the PYMK generation pipeline.

Note that what we have proposed is a simplified system. If you asked during an interview to optimize it, here are a few potential talking points:

- Pre-computing PYMK only for active users.
- Using a lightweight ranker to reduce the number of generated candidates into a smaller set before the scoring service assigns them a score.
- Using a re-ranking service to add diversity to the final PYMK list.

Other Talking Points

If there's time left at the end of the interview, here are some additional talking points:

- Personalized random walk [8] is another method often used to make recommendations. Since it's efficient, it is a helpful way to establish a baseline.
- Bias issue. Frequent users tend to have greater representation in the training data than occasional users. The model can become biased towards some groups and against others due to uneven representation in the training data. For example, in the PYMK list, frequent users might be recommended to other users at a higher rate. Subsequently, these users can make even more connections, making them even more represented in the training data [9].
- When a user ignores recommended connections repeatedly, the question arises of how to take them into account in future re-ranks. Ideally, ignored recommendations should have a lower ranking [9].
- A user may not send a connection request immediately when we recommend it to them. It may take a few days or weeks. So, when should we label a recommended connection as negative? In general, how would we deal with delayed feedback in recommendation systems [10]?

References

1. Clustering in ML. <https://developers.google.com/machine-learning/clustering/overview>.
2. PYMK on Facebook. <https://youtu.be/Xpx5RYNTQvg?t=1823>.
3. Graph convolutional neural networks. <http://tkipf.github.io/graph-convolutional-networks/>.
4. GraphSage paper. <https://cs.stanford.edu/people/jure/pubs/graphsage-nips17.pdf>.
5. Graph attention networks. <https://arxiv.org/pdf/1710.10903.pdf>.
6. Graph isomorphism network. <https://arxiv.org/pdf/1810.00826.pdf>.
7. Graph neural networks. <https://distill.pub/2021/gnn-intro/>.
8. Personalized random walk. https://www.youtube.com/watch?v=HbzQzUaJ_9I.
9. LinkedIn's PYMK system. <https://engineering.linkedin.com/blog/2021/optimizing-pymk-for-equity-in-network-creation>.
10. Addressing delayed feedback. <https://arxiv.org/pdf/1907.06558.pdf>.