1 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.

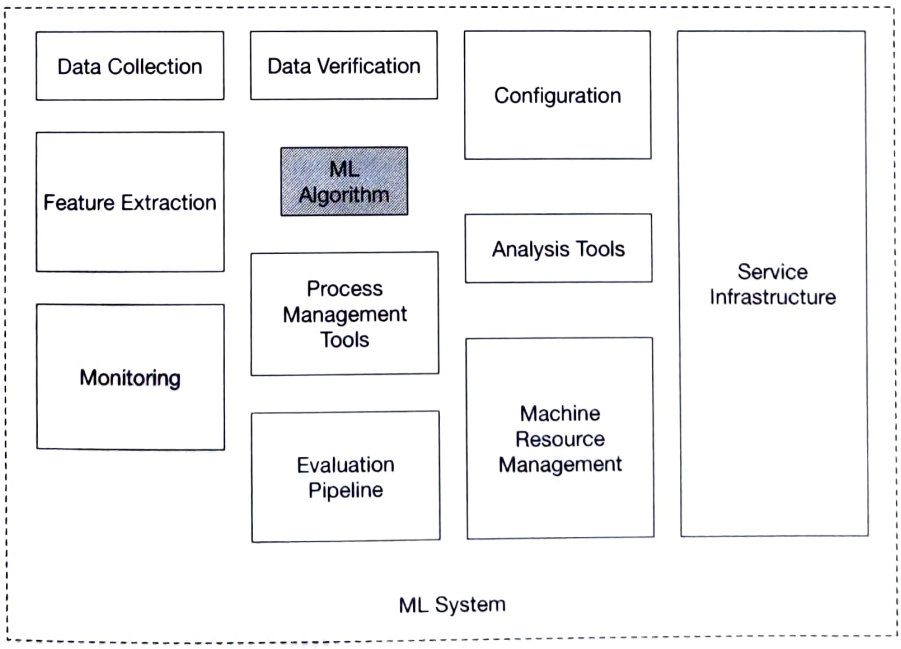


Figure 1.1: Components of a production-ready ML system

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

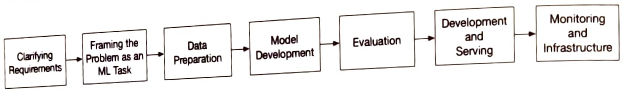


Figure 1.2: ML system design steps

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.

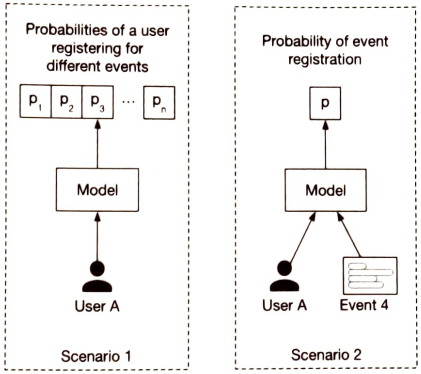


Figure 1.4: Different ways to specify the model's input-output

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.

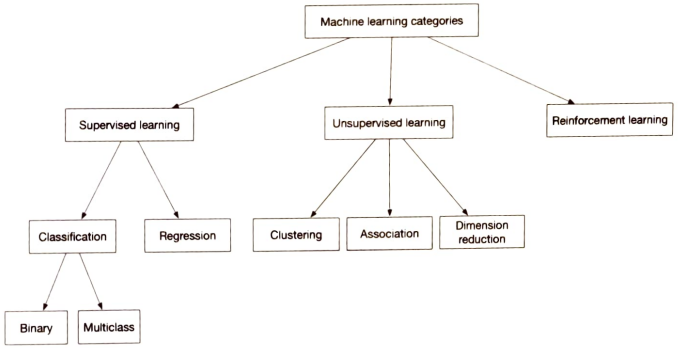


Figure 1.5: Common ML categories

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

is to identify meaningful patterns among data. Commonly used unsupervised learning algorithms are clustering, association, and dimensionality reduction.

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.

Regression model. Regression is the task of predicting a continuous numeric value. For example, the model that predicts the expected value of a house is a regression model.

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.

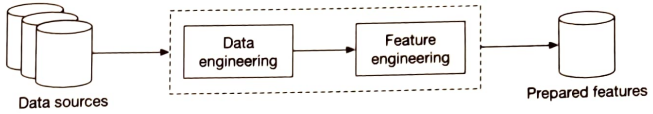


Figure 1.6: Data preparation process

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.

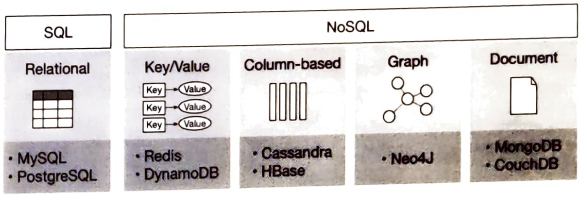


Figure 1.7: Different types of databases

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].

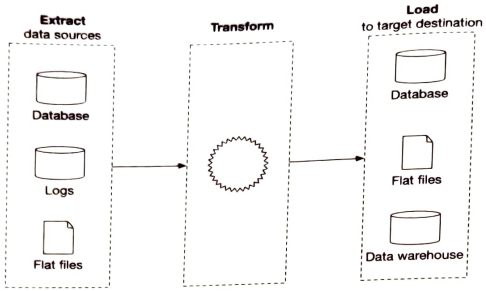


Figure 1.8: An overview of the ETL process

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.

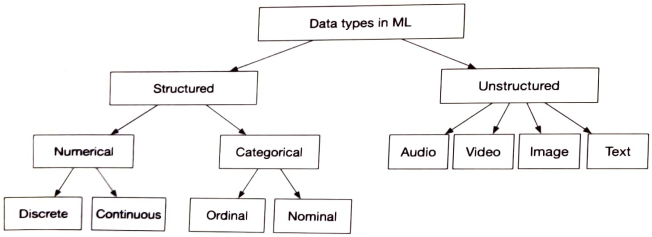


Figure 1.9: Data types in ML

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.

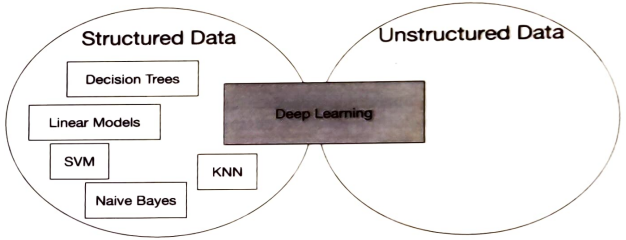


Figure 1.10: Models for structured and unstructured data (source [2])

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.

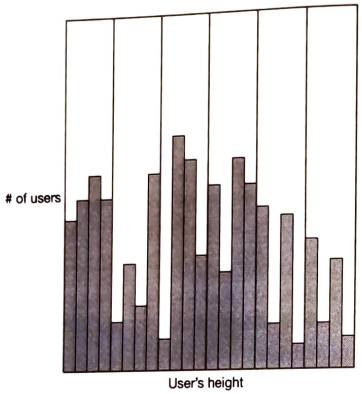


Figure 1.12: Discretizing user's height into 6 buckets

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.

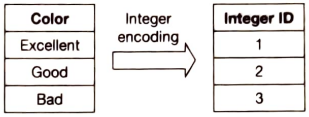


Figure 1.13: Integer encoding

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".

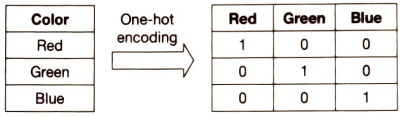


Figure 1.14: One-hot encoding

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 [3], boosting [4], and stacking [5], 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 [6]. 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 [7].

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 ⟨user, post⟩ 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 [8].

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 [9].

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.

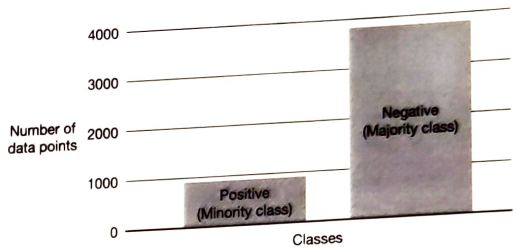


Figure 1.16: Imbalanced dataset with two classes

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).

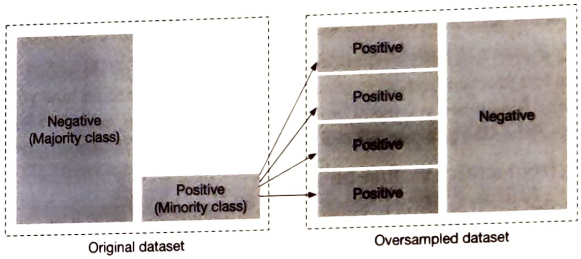


Figure 1.17: Oversampling the minority class

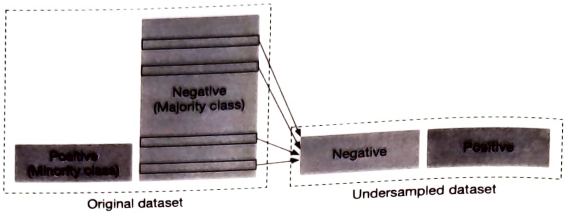


Figure 1.18: Undersampling the majority class

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 [10] and focal loss [11][12].

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 [13] and model parallelism [14].

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 [15], MSE [16], MAE [17], Huber loss [18], etc.)
  - What regularization should we use? (e.g., L1 [19], L2 [19], Entropy Regularization [20], K-fold CV [21], or dropout [22])
  - What is backpropagation?
  - You may need to describe common optimization methods [23] such as SGD [24], AdaGrad [25], Momentum [26], and RMSProp [27].
  - What activation functions do we want to use and why? (e.g., ELU [28], ReLU [29], Tanh [30], Sigmoid [31]).
  - 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 [32], Inception score [33]
Natural language processing	BLEU [34], METEOR [35], ROUGE [36], CIDEr [37], SPICE [38]

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 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 [39].

To learn more about model compression, you are encouraged to read [40].

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 [41], A/B testing [42], canary release [43], interleaving experiments [44], bandits [45], 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.

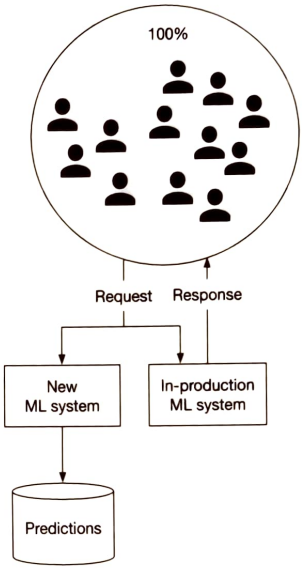


Figure 1.19: Shadow deployment

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.

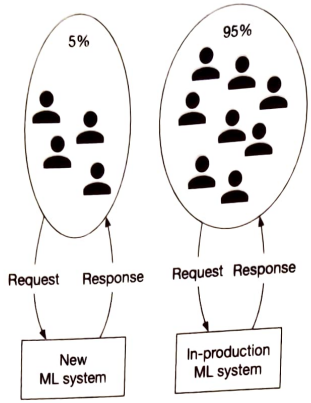


Figure 1.20: A/B testing

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.

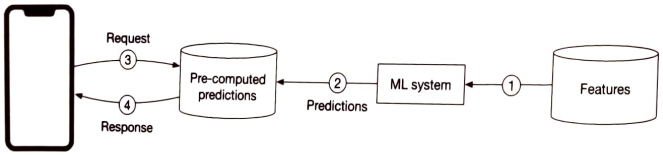


Figure 1.21: Batch prediction workflow

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.

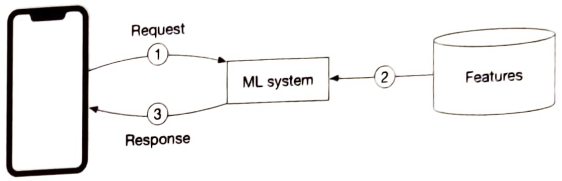


Figure 1.22: Online prediction workflow

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.

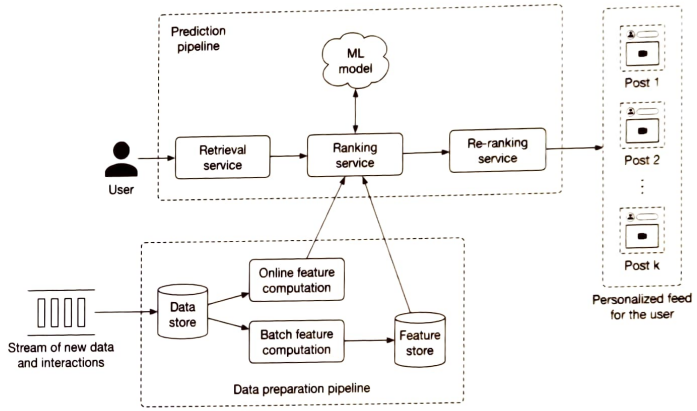


Figure 1.23: ML system design for a personalized news feed system

Talking points

- Where should the computations take place during the inference: in the cloud or on-device?
- 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.

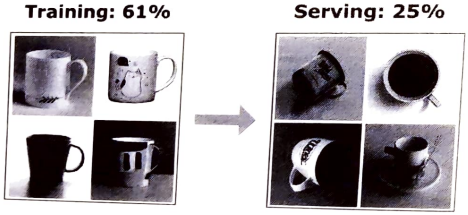


Figure 1.24: Data distribution shift

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 [46][47][48].

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.

Reference Material

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

[2] Structured vs. unstructured data. https://signal.onepointltd.com/post/102gjab/machine-learning-libraries-for-tabular-data-problems.

[3] Bagging technique in ensemble learning. https://en.wikipedia.org/wiki/Bootstrap_aggregating.

[4] Boosting technique in ensemble learning. https://aws.amazon.com/what-is/boosting/.

[5] Stacking technique in ensemble learning. https://machinelearningmastery.com/stacking-ensemble-machine-learning-with-python/.

[6] Interpretability in Machine Mearning. https://blog.ml.cmu.edu/2020/08/31/6-interpretability/.

[7] Traditional machine learning algorithms. https://machinelearningmastery.com/a-tour-of-machine-learning-algorithms/.

[8] Sampling strategies. https://www.scribbr.com/methodology/sampling-methods/.

[9] Data splitting techniques. https://machinelearningmastery.com/train-test-split-for-evaluating-machine-learning-algorithms/.

[10] Class-balanced loss. https://arxiv.org/pdf/1901.05555.pdf.

[11] Focal loss paper. https://arxiv.org/pdf/1708.02002.pdf.

[12] Focal loss. https://medium.com/swlh/focal-loss-an-efficient-way-of-handling-class-imbalance-4855ae1db4cb.

[13] Data parallelism. https://www.telesens.co/2017/12/25/understanding-data-parallelism-in-machine-learning/.

[14] Model parallelism. https://docs.aws.amazon.com/sagemaker/latest/dg/model-parallel-intro.html.

[15] Cross entropy loss. https://en.wikipedia.org/wiki/Cross_entropy.

[16] Mean squared error loss. https://en.wikipedia.org/wiki/Mean_squared_error.

[17] Mean absolute error loss. https://en.wikipedia.org/wiki/Mean_absolute_error.

[18] Huber loss. https://en.wikipedia.org/wiki/Huber_loss.

[19] L1 and l2 regularization. https://www.analyticssteps.com/blogs/l2-and-l1-regularization-machine-learning.

[20] Entropy regularization. https://paperswithcode.com/method/entropy-regularization.

[21] K-fold cross validation. https://en.wikipedia.org/wiki/Cross-validation_(statistics).

[22] Dropout paper. https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf.

[23] Overview of optimization algorithm. https://ruder.io/optimizing-gradient-descent/.

[24] Stochastic gradient descent. https://en.wikipedia.org/wiki/Stochastic_gradient_descent.

[25] AdaGrad optimization algorithm. https://optimization.cbe.cornell.edu/index.php?title=AdaGrad.

[26] Momentum optimization algorithm. https://optimization.cbe.cornell.edu/index.php?title=Momentum.

[27] RMSProp optimization algorithm. https://optimization.cbe.cornell.edu/index.php?title=RMSProp.

[28] ELU activation function. https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#elu.

[29] ReLU activation function. https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#relu.

[30] Tanh activation function. https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#tanh.

[31] Sigmoid activation function. https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#softmax.

[32] FID score. https://en.wikipedia.org/wiki/Fr%C3%A9chet_inception_distance.

[33] Inception score. https://en.wikipedia.org/wiki/Inception_score.

[34] BLEU metrics. https://en.wikipedia.org/wiki/BLEU.

[35] METEOR metrics. https://en.wikipedia.org/wiki/METEOR.

[36] ROUGE score. https://en.wikipedia.org/wiki/ROUGE_(metric).

[37] CIDEr score. https://arxiv.org/pdf/1411.5726.pdf.

[38] SPICE score. https://arxiv.org/pdf/1607.08822.pdf.

[39] Quantization-aware training. https://pytorch.org/docs/stable/quantization.html.

[40] Model compression survey. https://arxiv.org/pdf/1710.09282.pdf.

[41] Shadow deployment. https://christophergs.com/machine%20learning/2019/03/30/deploying-machine-learning-applications-in-shadow-mode/.

[42] A/B testing. https://en.wikipedia.org/wiki/A/B_testing.

[43] Canary release. https://blog.getambassador.io/cloud-native-patterns-canary-release-1cb8f82d371a.

[44] Interleaving experiment. https://netflixtechblog.com/interleaving-in-online-experiments-at-netflix-a04ee392ec55.

[45] Multi-armed bandit. https://vwo.com/blog/multi-armed-bandit-algorithm/.

[46] ML infrastructure. https://www.run.ai/guides/machine-learning-engineering/machine-learning-infrastructure.

[47] Interpretability in ML. https://fullstackdeeplearning.com/spring2021/lecture-6/.

[48] Chip Huyen. *Designing Machine Learning Systems: An Iterative Process for Production-Ready Application.* " O'Reilly Media, Inc.", 2022.

2 Visual Search System

A visual search 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].

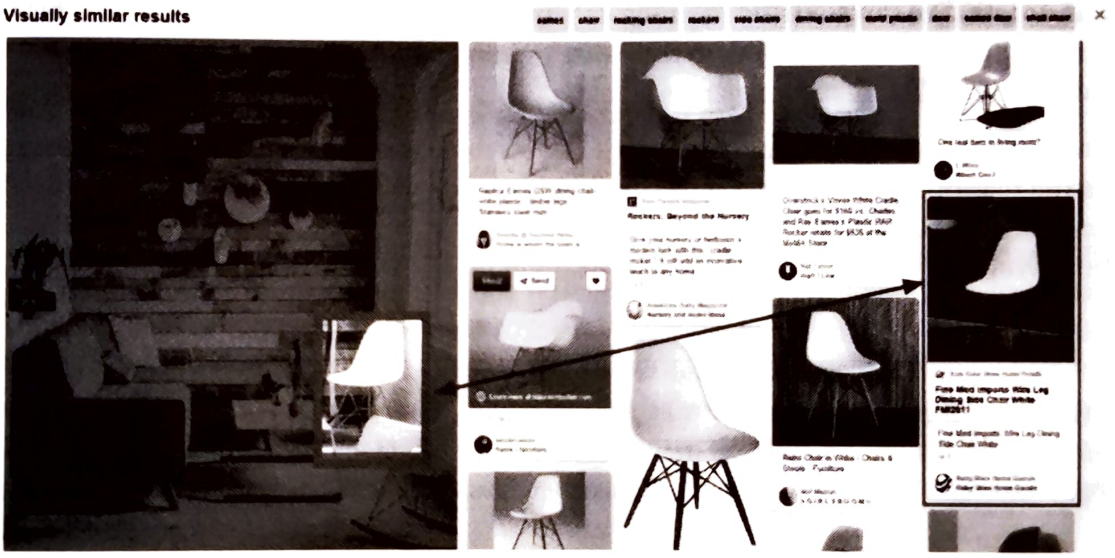


Figure 2.1: Retrieved images that are visually similar to the selected crop

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.

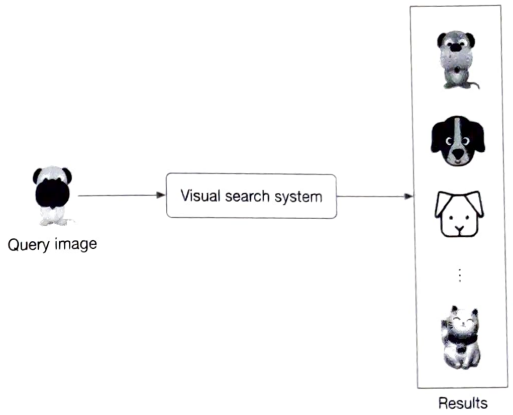


Figure 2.2: A visual search system's input-output

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.

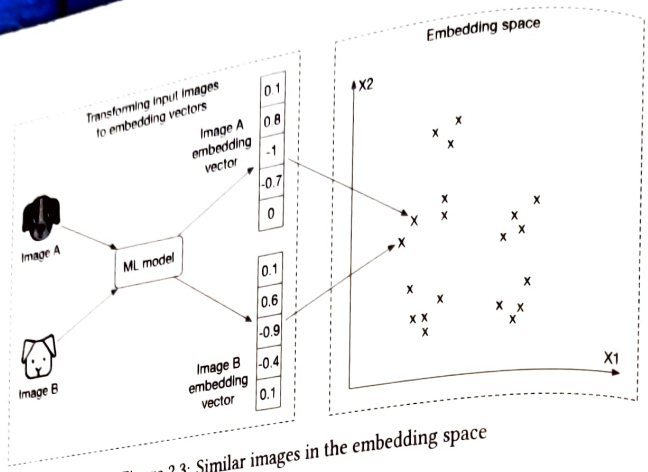


Figure 2.3: Similar images in the embedding space

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.

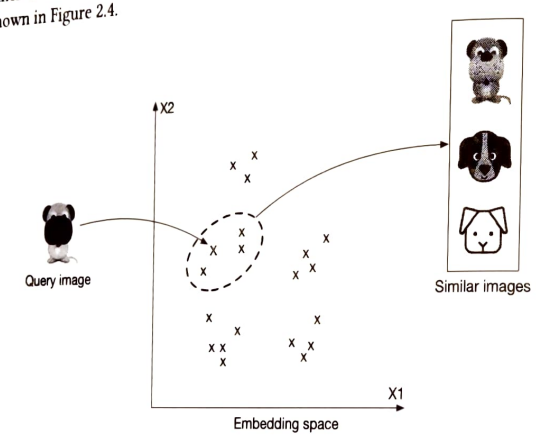


Figure 2.4: Top 3 images similar to the query image

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.

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).

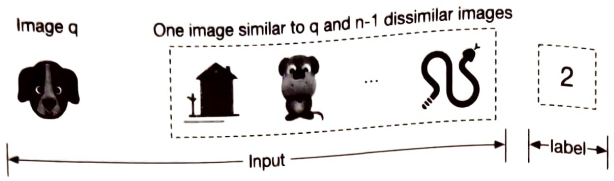


Figure 2.7: Training data point

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.

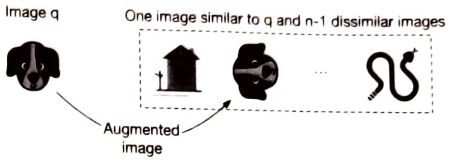


Figure 2.8: Use data augmentation to create a similar image

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 trade-offs. 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.

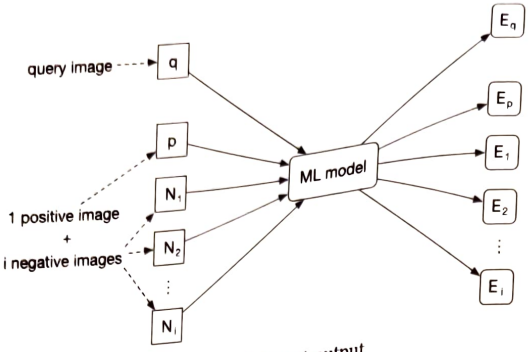


Figure 2.9: Model input-output

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.

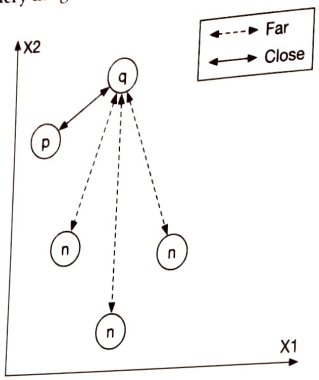


Figure 2.10: Input images mapped into the embedding space

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].

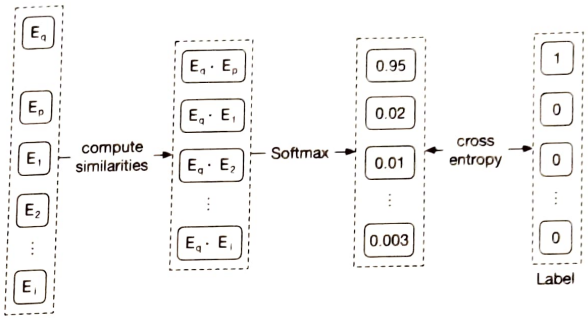


Figure 2.11: A simplified contrastive loss

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.

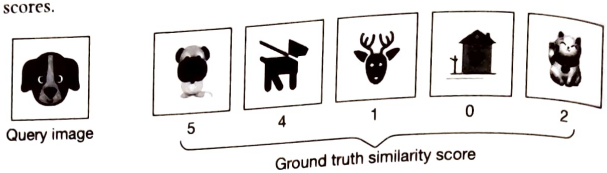


Figure 2.12: A data point from the evaluation dataset

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 ith 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.

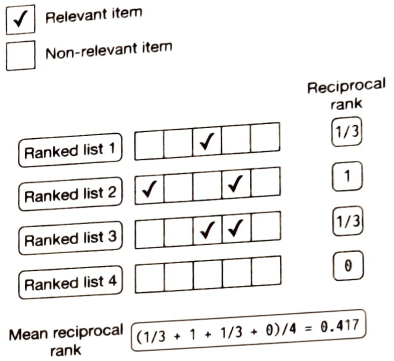


Figure 2.13: MRR calculation example

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.

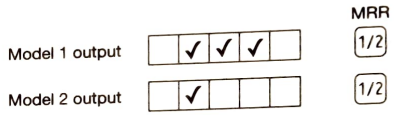


Figure 2.14: MRR of two different models

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.

normalize DCG. For this, nDCG divides the DCG by the DCG of an ideal ranking. The formula is:

$$\text{nDCG}_p = \frac{DCG_p}{IDCG_p}$$

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.

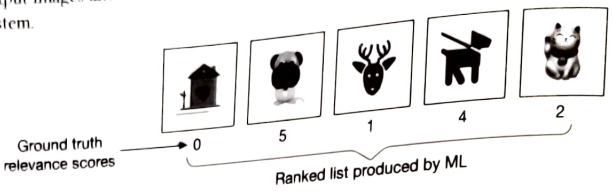


Figure 2.17: A ranked list produced by a search system

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:

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

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).

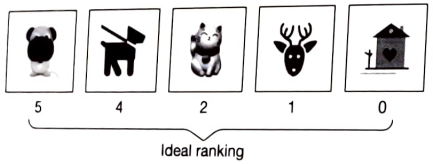


Figure 2.18: Ideal ranking list

The IDCG for the ideal ranking is:

$$\text{IDCG}_p = \sum_{i=1}^{p} \frac{rel_i}{\log_2(i+1)} = \frac{5}{\log_2(2)} + \frac{4}{\log_2(3)} + \frac{2}{\log_2(4)} + \frac{1}{\log_2(5)} + \frac{0}{\log_2(6)} = 8.9543$$

Divide DCG by IDCG:

$$\text{nDCG}_p = \frac{DCG_p}{IDCG_p} = \frac{6.151}{8.9543} = 0.6869$$

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:

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

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.

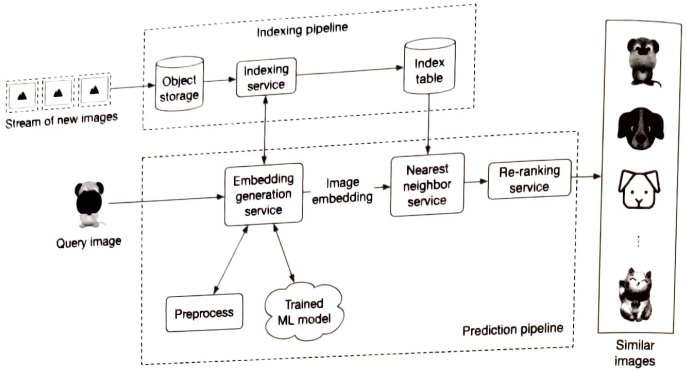


Figure 2.19: Prediction and indexing pipeline

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.

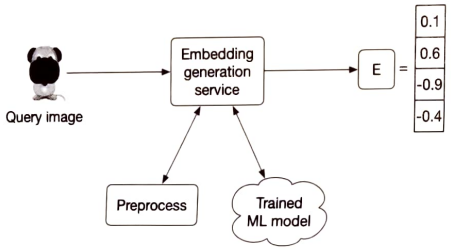


Figure 2.20: Embedding generation service

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.

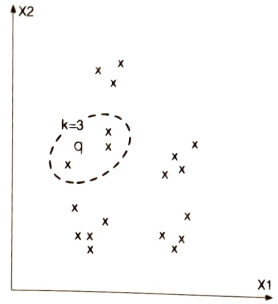


Figure 2.21: Top 3 nearest neighbors to image q in the embedding space

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 diver 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.

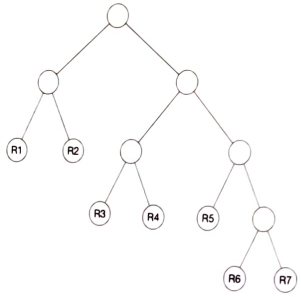


Figure 2.22: A formed tree from the points

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.

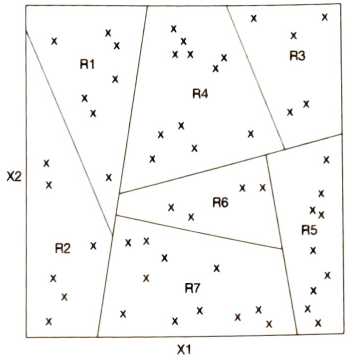


Figure 2.23: Partitioned space by the tree

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].

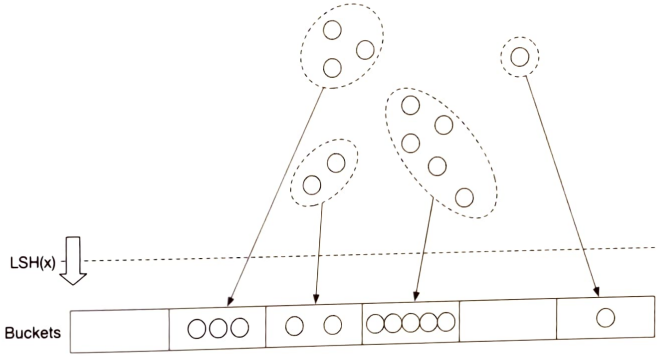


Figure 2.24: Use LSH to group the data points into buckets

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.

Summary

Reference Material

[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. https://openaccess.thecvf.com/content_CVPR_2020/papers/He_Momentum_Contrast_for_Unsupervised_Visual_Representation_Learning_CVPR_2020_paper.pdf.

[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.

[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).

[30] Human-in-the-loop ML. https://arxiv.org/pdf/2108.00941.pdf.

3 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.

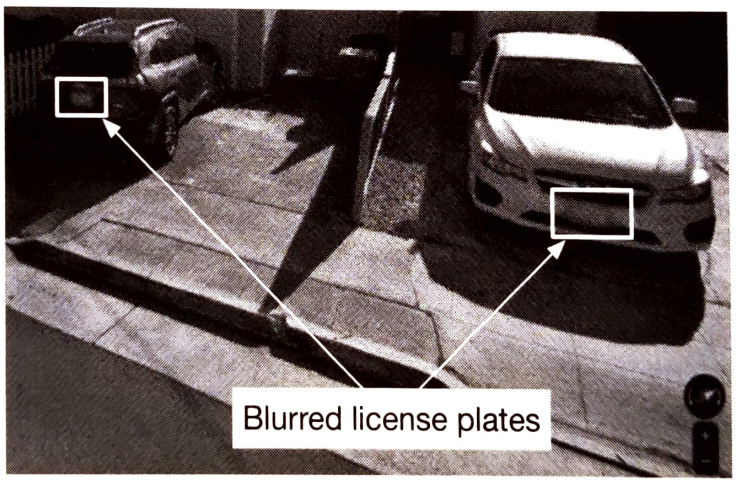


Figure 3.1: A Street View image with blurred license plates

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.

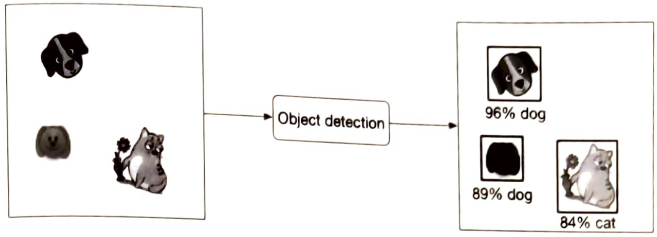


Figure 3.2: Object detection system's input-output

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.

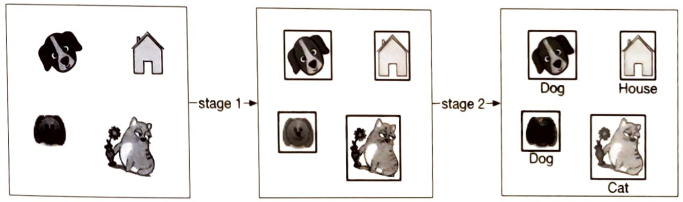


Figure 3.3: Two-stage network

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

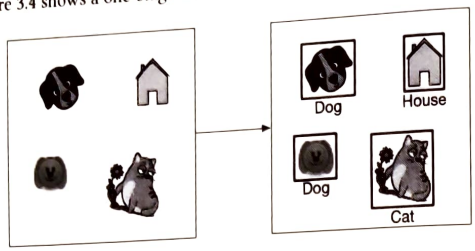


Figure 3.4: One-stage network

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 human face license plate	[10, 10, 25, 50] [120, 180, 40, 70] [80, 95, 35, 10]
dataset/image2.jpg	human face	[170, 190, 30, 80]
dataset/image3.jpg	license plate human face	[25, 30, 210, 220] [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 image preprocessing operations, 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.

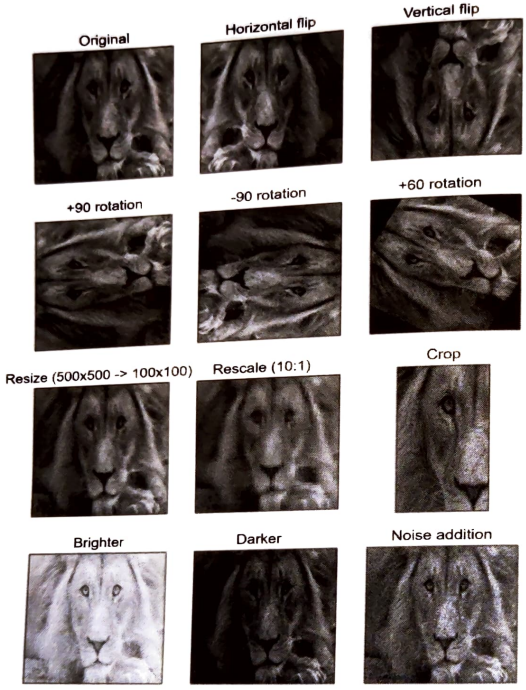


Figure 3.5: Augmented images (source [8])

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.

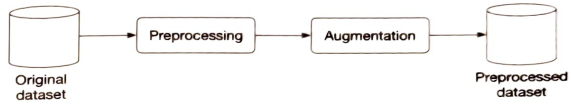


Figure 3.6: Dataset preparation workflow

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.

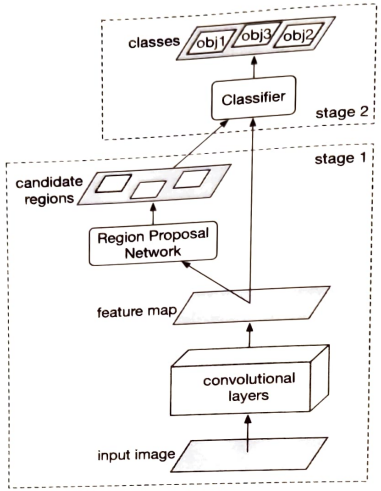


Figure 3.7: Two-stage object detection network

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. For more information, please see [10].

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 [11]. 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) [12], and denote it by L_{reg}:

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

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 (cross-entropy) [13] 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.

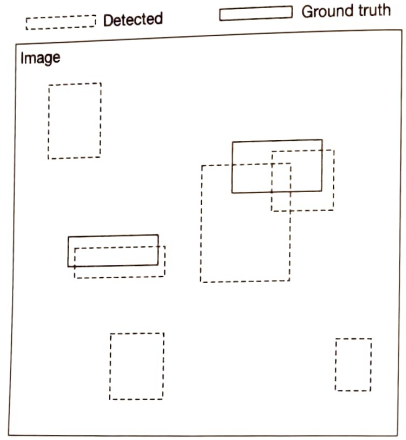


Figure 3.8: Ground truth and detected bounding boxes

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.

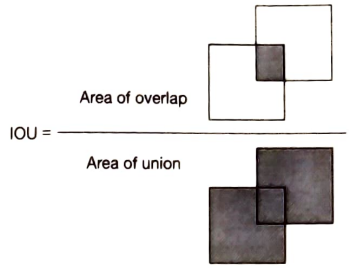


Figure 3.9: IOU formula

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.

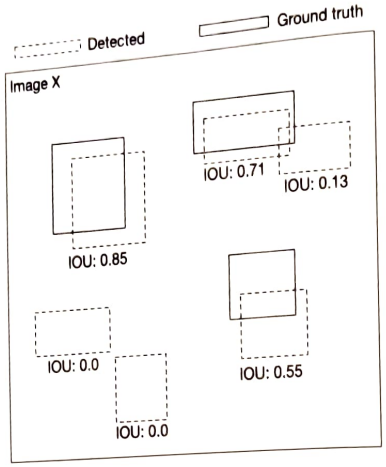


Figure 3.10: Ground truth bounding boxes and detected bounding boxes

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 [14], 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 [15] [16].

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 over-

lapping 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) [17]. 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.

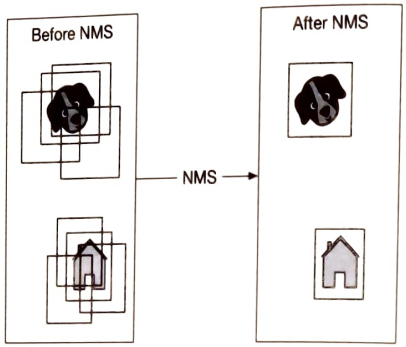


Figure 3.11: Before and after applying NMS

NMS is a commonly asked algorithm in ML system design interviews, so you're encouraged to have a good understanding of it [18].

ML system design

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

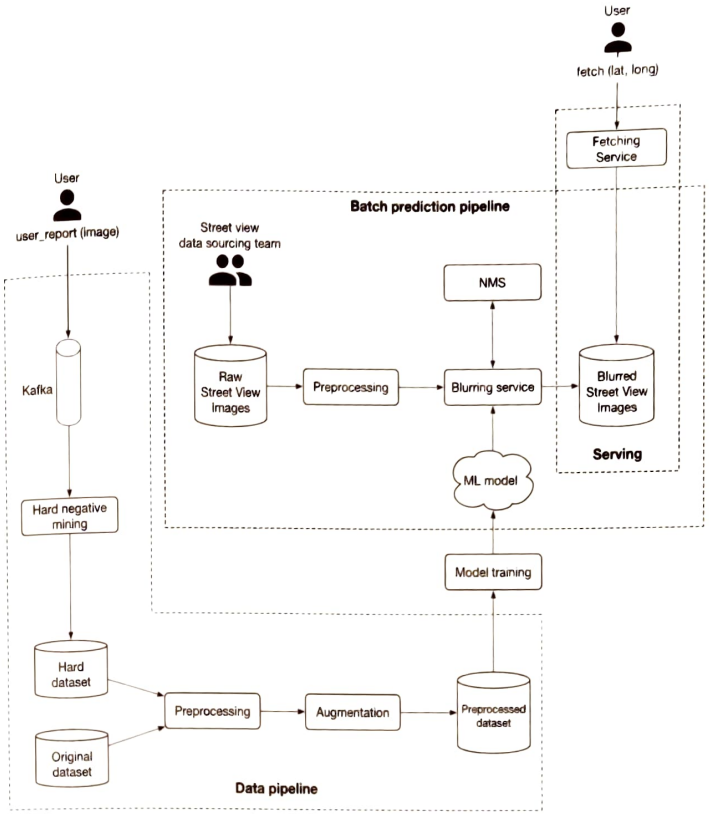


Figure 3.12: ML system design

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.

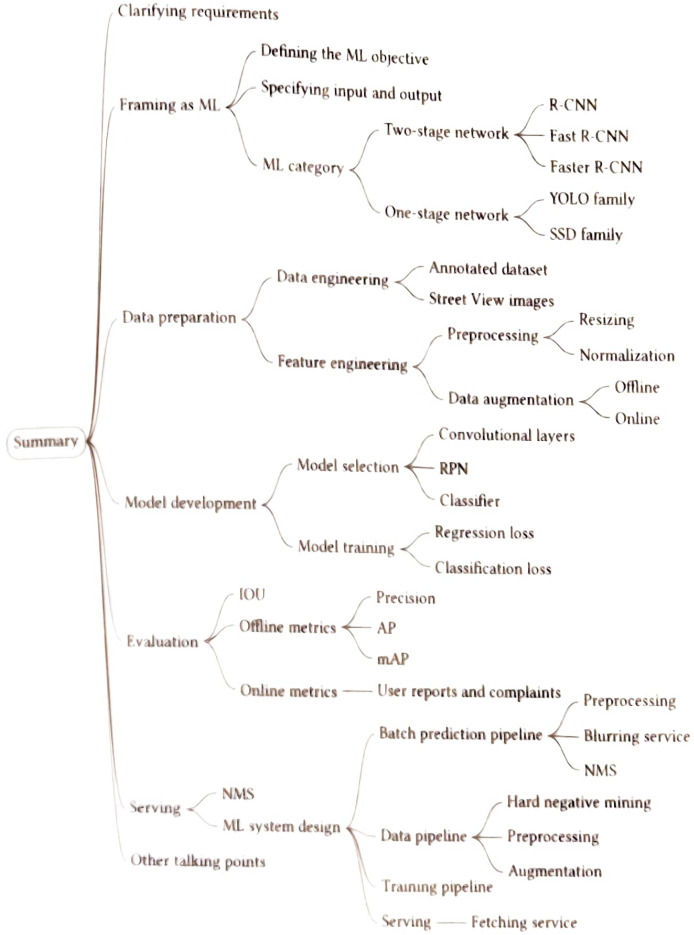
Hard negative mining. Hard negatives are examples that are explicitly created as negatives out of incorrectly predicted examples, and then added to the training dataset. When we retrain the model on the updated training dataset, it should perform better.

Other Talking Points

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

- How Transformer-based object detection architectures differ from one-stage or two-stage models, and what are their pros and cons [19].
- Distributed training techniques to improve object detection on a larger dataset [20] [21].
- How General Data Protection Regulation (GDPR) in Europe may affect our system [22].
- Evaluate bias in face detection systems [23] [24].
- How to continuously fine-tune the model [25].
- How to use active learning [26] or human-in-the-loop ML [27] to select data points for training.

Summary



Reference Material

[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] Object detection details. https://dudeperf3ct.github.io/object/detection/2019/01/07/Mystery-of-Object-Detection.

[11] Forward pass and backward pass. https://www.youtube.com/watch?v=qzPQ8cEsVK8.

[12] MSE. https://en.wikipedia.org/wiki/Mean_squared_error.

[13] Log loss. https://en.wikipedia.org/wiki/Cross_entropy.

[14] Pascal VOC. http://host.robots.ox.ac.uk/pascal/VOC/voc2008/index.html.

[15] COCO dataset evaluation. https://cocodataset.org/#detection-eval.

[16] Object detection evaluation. https://github.com/rafaelpadilla/Object-Detection-Metrics.

[17] NMS. https://en.wikipedia.org/wiki/NMS.

[18] Pytorch implementation of NMS. https://learnopencv.com/non-maximum-suppression-theory-and-implementation-in-pytorch/.

[19] Recent object detection models. https://viso.ai/deep-learning/object-detection/.

[20] Distributed training in Tensorflow. https://www.tensorflow.org/guide/distributed_training.

[21] Distributed training in Pytorch. https://pytorch.org/tutorials/beginner/dist_overview.html.

[22] GDPR and ML. https://www.oreilly.com/radar/how-will-the-gdpr-impact-machine-learning.

[23] Bias and fairness in face detection. http://sibgrapi.sid.inpe.br/col/sid.inpe.br/sibgrapi/2021/09.04.19.00/doc/103.pdf.

[24] AI fairness. https://www.kaggle.com/code/alexisbcook/ai-fairness.

[25] Continual learning. https://towardsdatascience.com/how-to-apply-continual-learning-to-your-machine-learning-models-4754adcd7f7f.

[26] Active learning. https://en.wikipedia.org/wiki/Active_learning_(machine_learning).

[27] Human-in-the-loop ML. https://arxiv.org/pdf/2108.00941.pdf.

4 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.

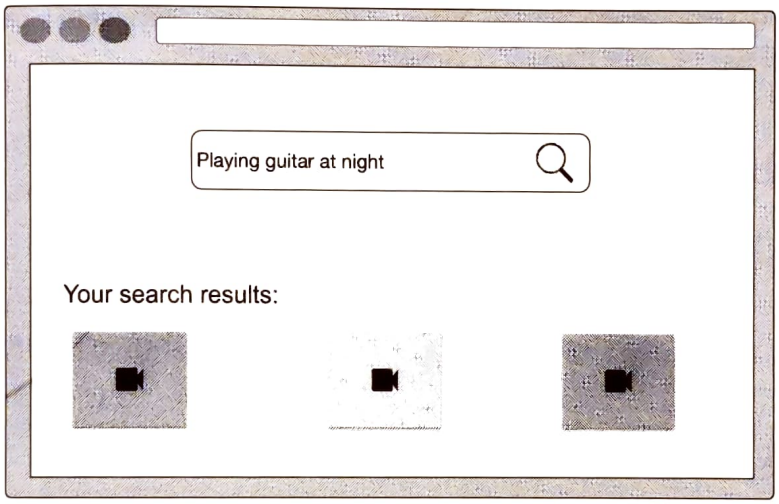


Figure 4.1: Searching videos with a text query

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 <video, text query> 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.

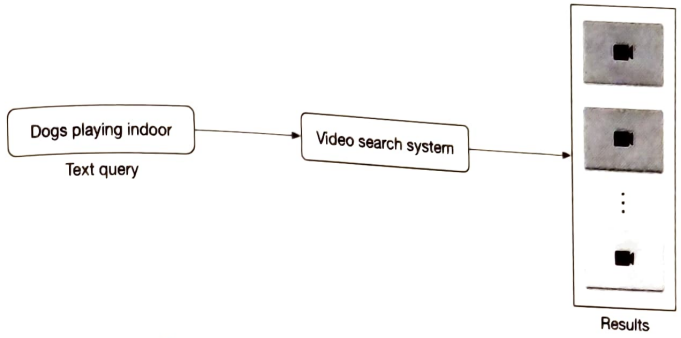


Figure 4.2: Video search system's input-output

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.

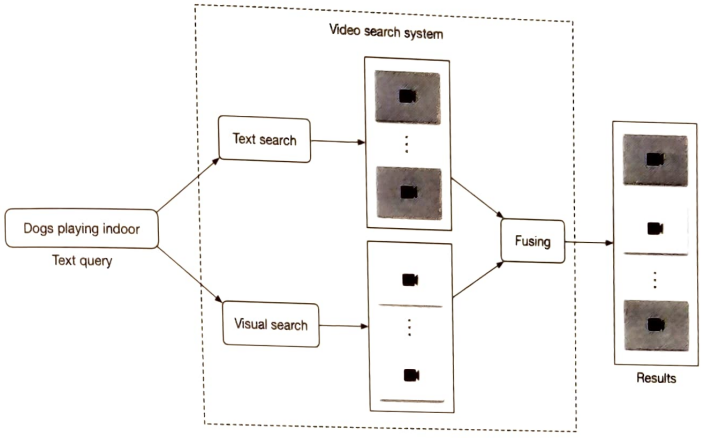


Figure 4.3: High-level overview of the search system

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.

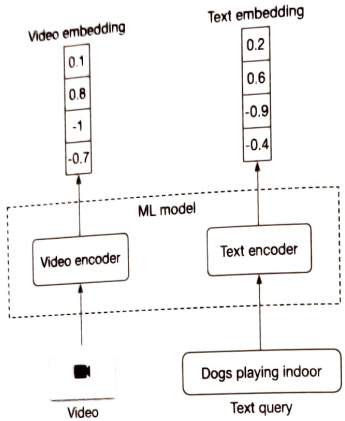
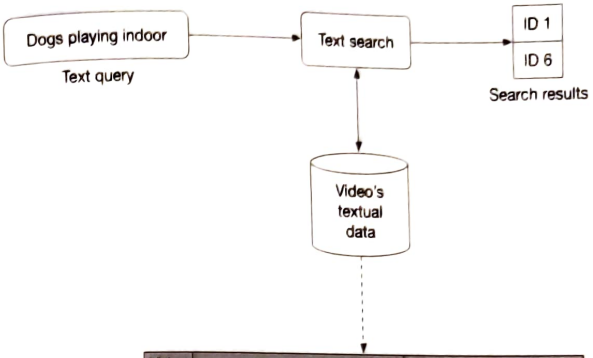


Figure 4.4: ML model's input-output

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.



Video ID	Title	Tags
1	My dog is playing indoor	dog, indoor, play
2	The swimming competition in the middle school	pool, child
3	watch the video I took when I was in Japan	travel, japan, vlog
4	Speed comparison between sport cars	car, race, speed
5	My remote work setup	computer, mouse, talk
6	How our dogs enjoyed the party last night at our house	cat, dog, family

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].

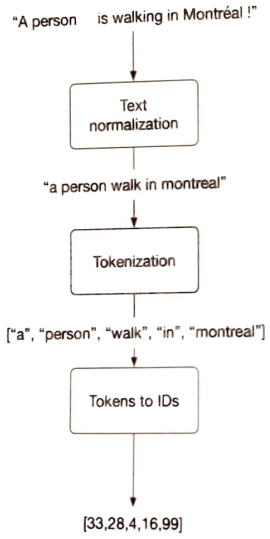


Figure 4.6: Represent a text with a numerical vector

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.

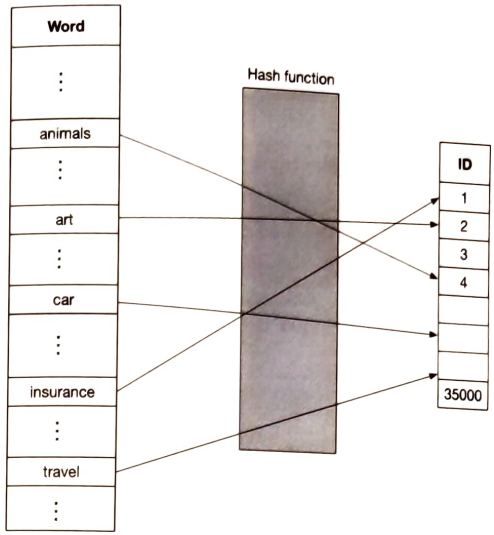


Figure 4.8: Use hashing to obtain word IDs

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.

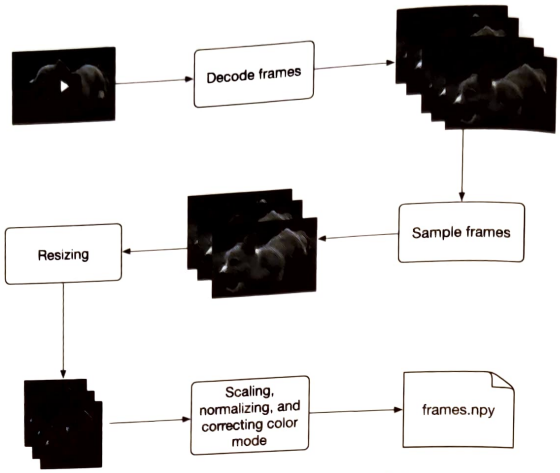


Figure 4.9: Video preprocessing workflow

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.

Text encoder

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

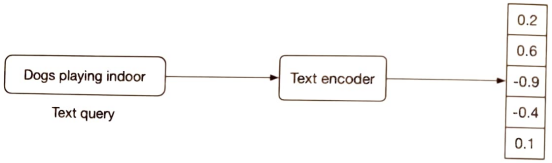


Figure 4.10: Text encoder's input-output

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 con-

textual 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.

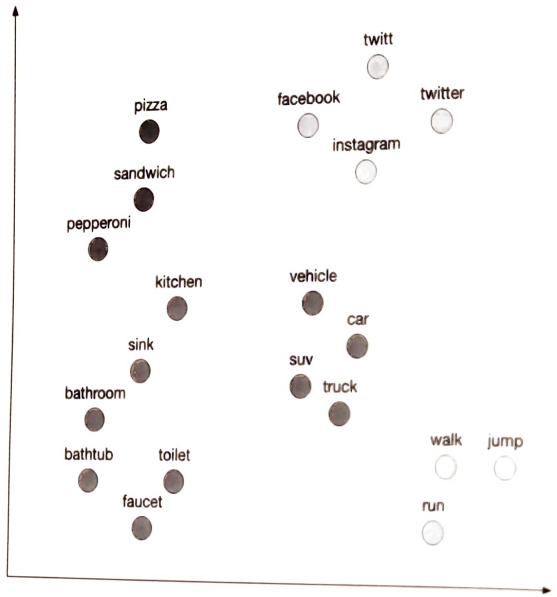


Figure 4.12: Words in the 2D embedding space

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.

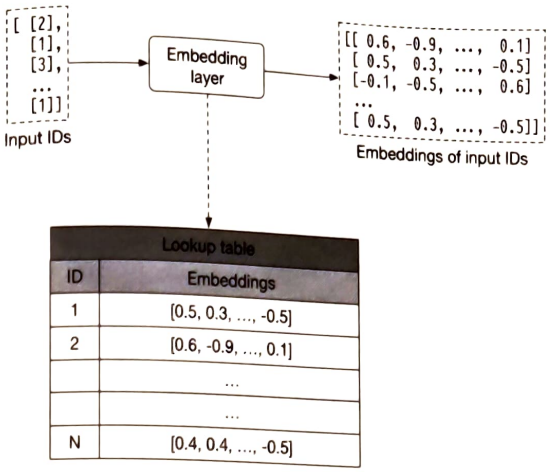


Figure 4.13: Embedding lookup method

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].

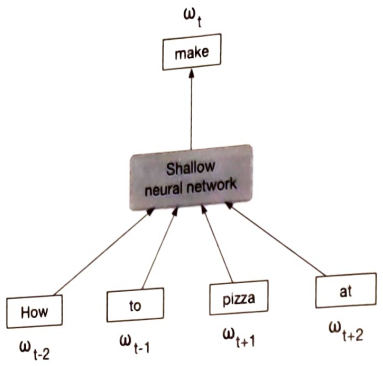


Figure 4.14: CBOW approach

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.

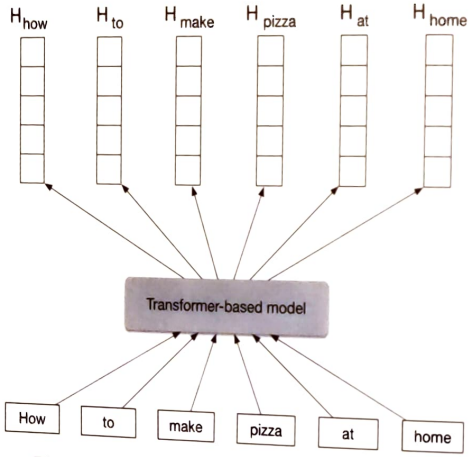


Figure 4.15: Transformer-based model's input-output

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:

- 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.

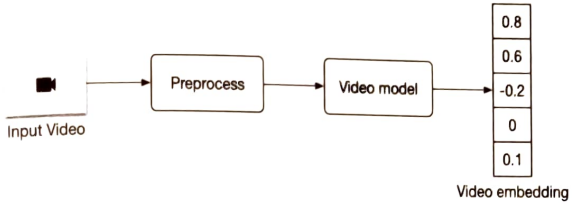


Figure 4.16: A video-level model

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.

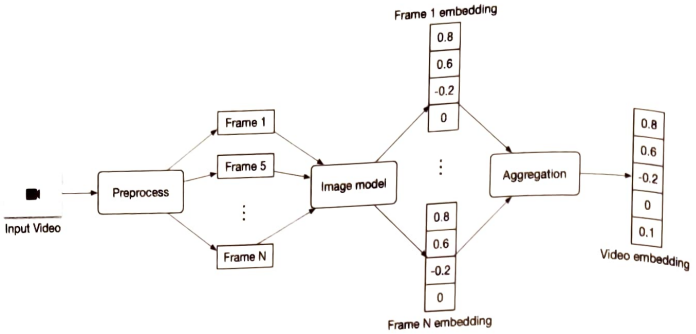


Figure 4.17: A frame-level model

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.

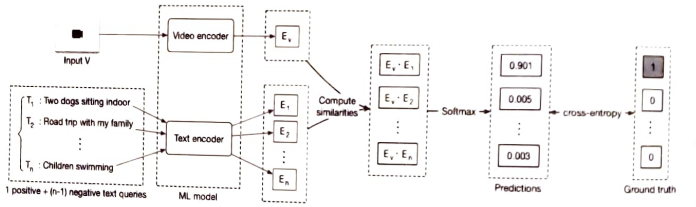


Figure 4.18: Loss computation

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 precison@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.

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

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

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, 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 recall@10 to measure the quality of these two models, both would have 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 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.

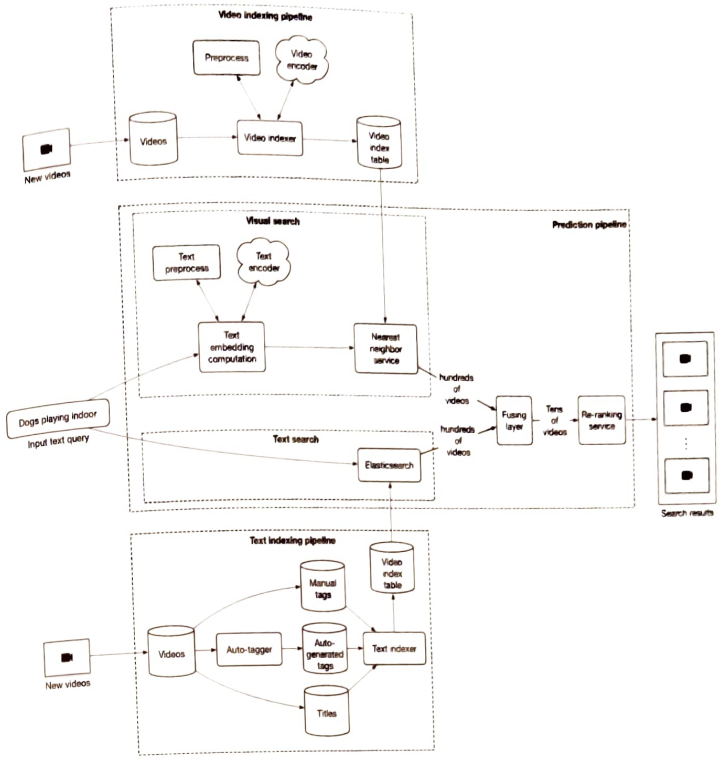


Figure 4.19: ML system design

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.

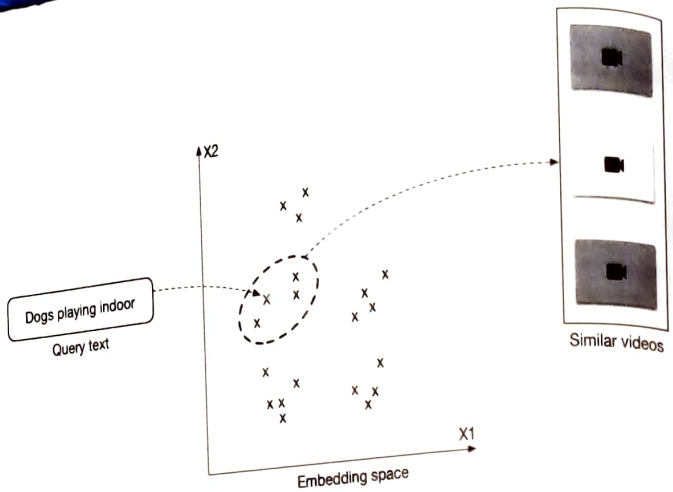


Figure 4.20: Retrieving the top 3 results for a given text query

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

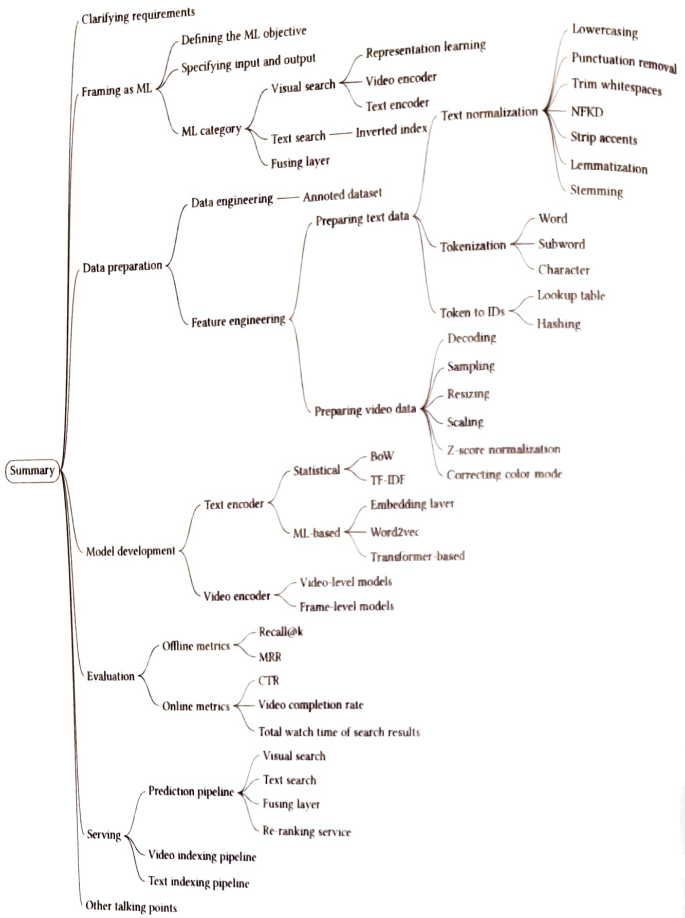
Before concluding this chapter, it's important to note we have simplified the system design of the video search system. In practice, it is much more complex. Some improvements may include:

- 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].

Summary



Reference Material

[1] Elasticsearch. https://www.tutorialspoint.com/elasticsearch/elasticsearch_query_dsl.htm.

[2] Preprocessing text data. https://huggingface.co/docs/transformers/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-tunkelang/.

[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.

5 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].

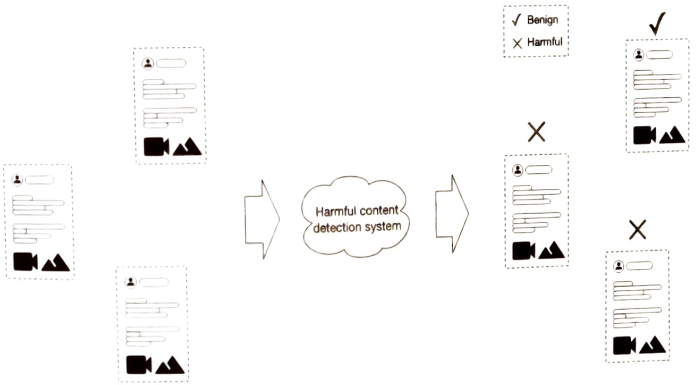


Figure 5.1: Harmful content detection system

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.

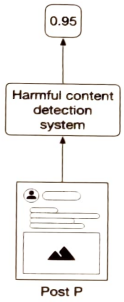


Figure 5.2: A harmful content detection system's input-output

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

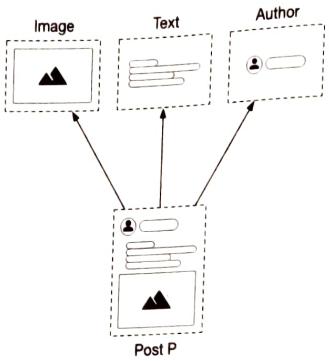


Figure 5.3: Heterogeneous post data

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.

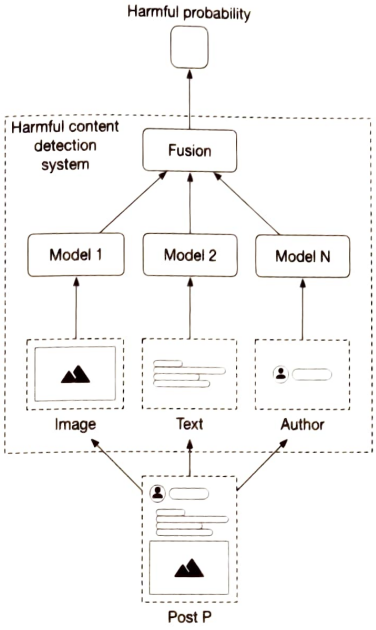


Figure 5.4: Late fusion

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.

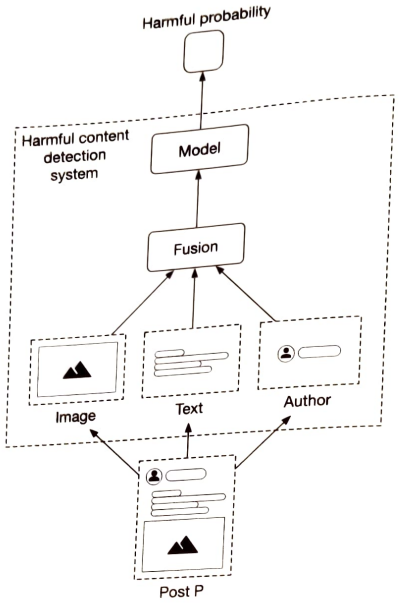


Figure 5.5: Early fusion

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.

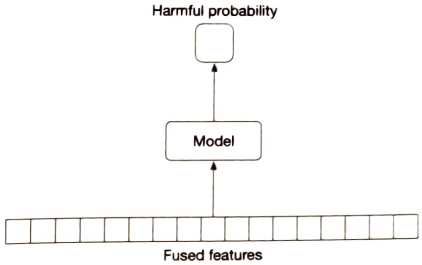


Figure 5.6: A single binary classifier

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.

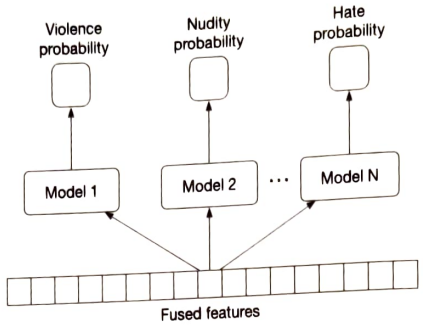


Figure 5.7: One binary classifier per harmful class

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.

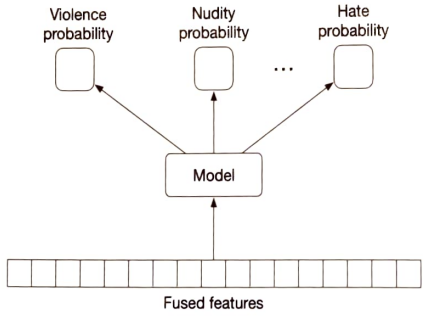


Figure 5.8: A multi-label classifier

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.

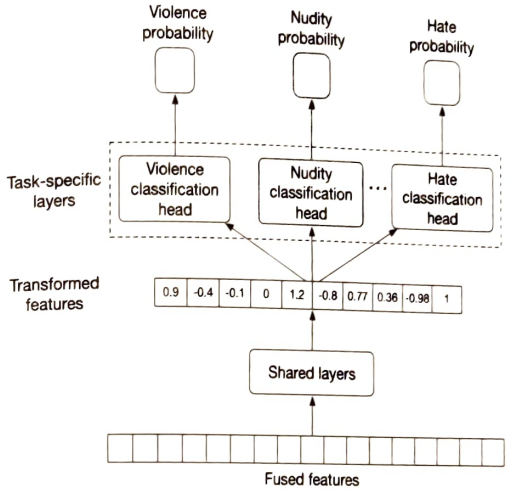


Figure 5.9: Multi-task classification overview

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.

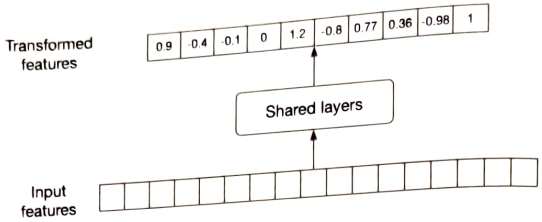


Figure 5.10: Shared layers

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.

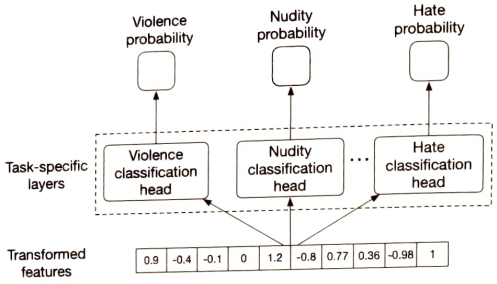


Figure 5.11: Task-specific layers

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.

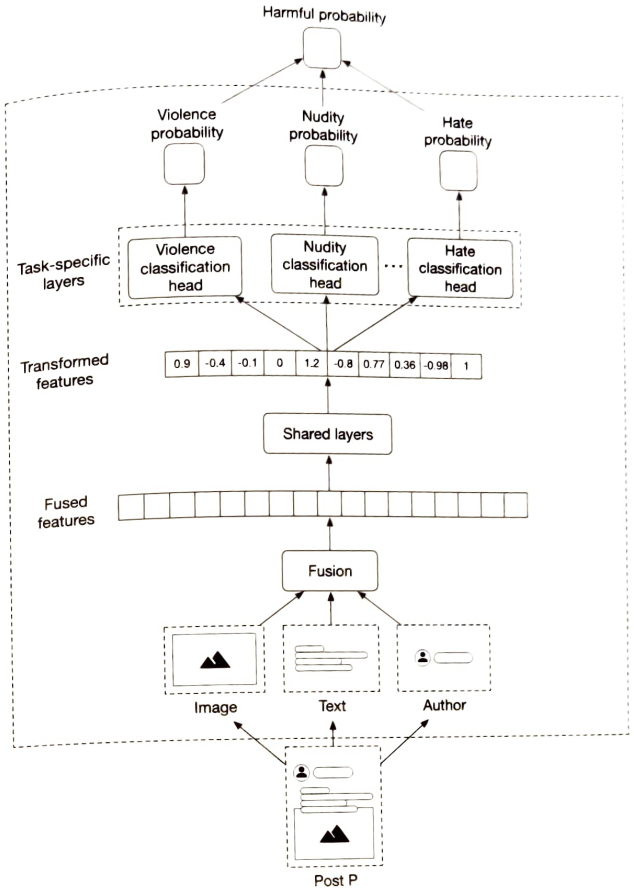


Figure 5.12: Frame the problem as an ML task

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	Author's IP	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 multi-task 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.

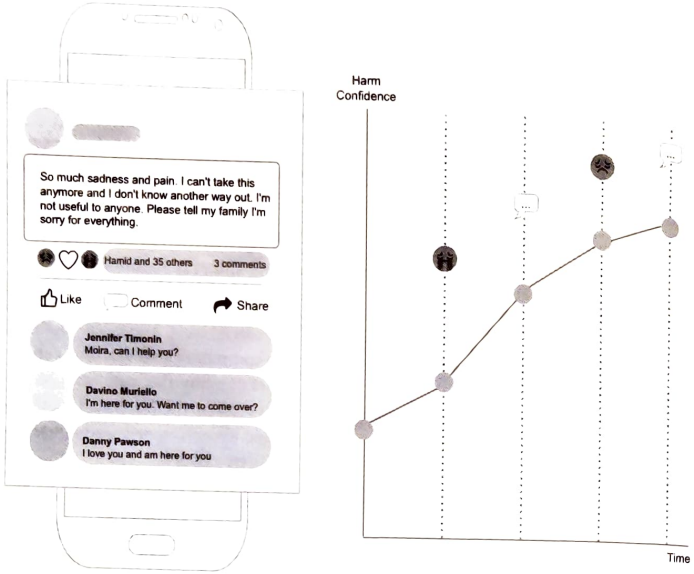


Figure 5.13: Post with self-harm concern

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.

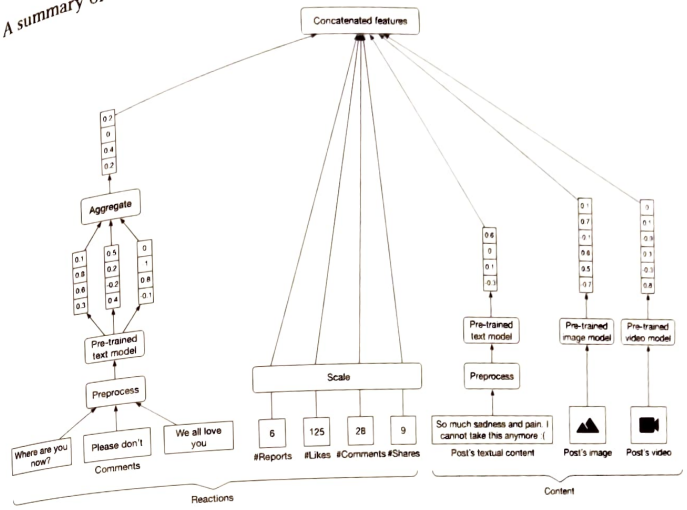


Figure 5.14: Feature engineering for reactions and content

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 repre-

sent 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.

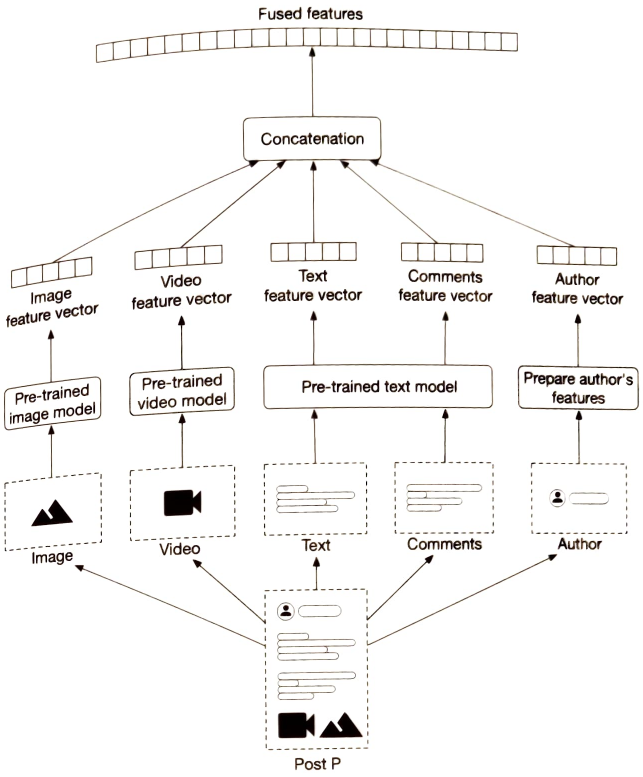


Figure 5.15: Summary of feature engineering

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.

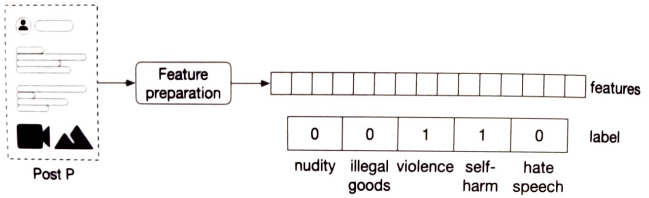


Figure 5.16: A constructed data point

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.

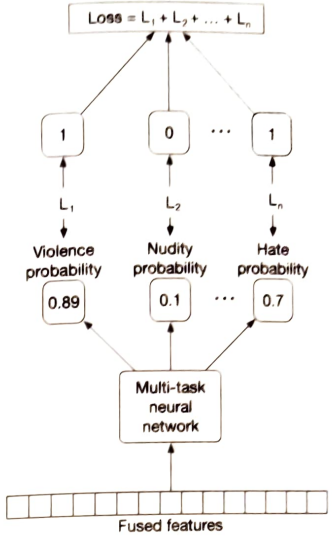


Figure 5.17: Model training

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.

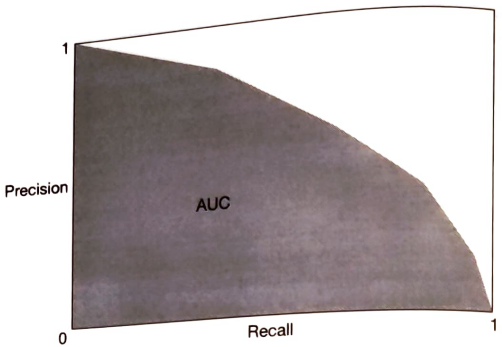


Figure 5.18: PR curve

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 100K 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.

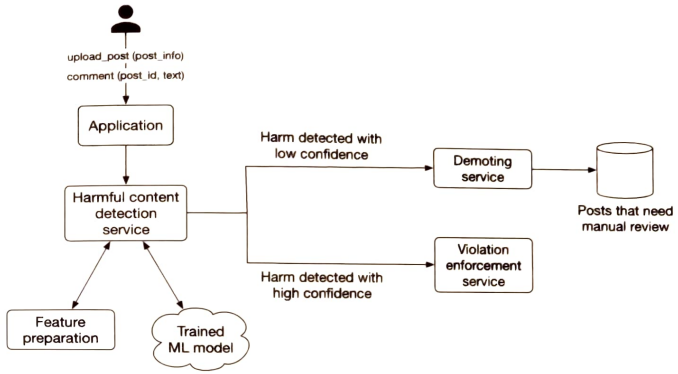


Figure 5.19: ML system design

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

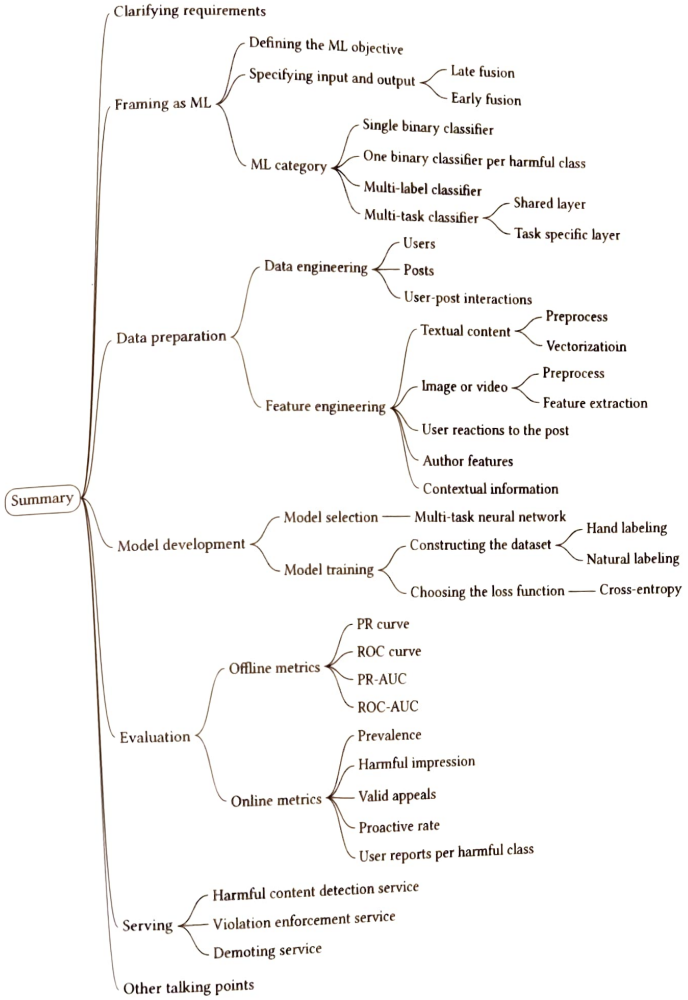
among users.

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].

Summary



Reference Material

[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/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://labelyourdata.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/1201751883328695/.

[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/metas-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/.

6 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.

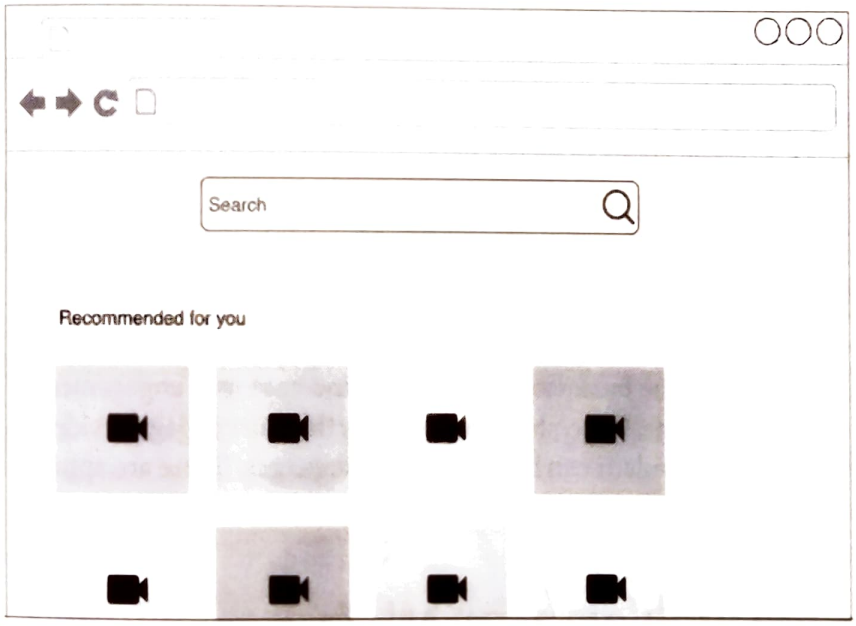


Figure 6.1: Homepage video recommendation

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.

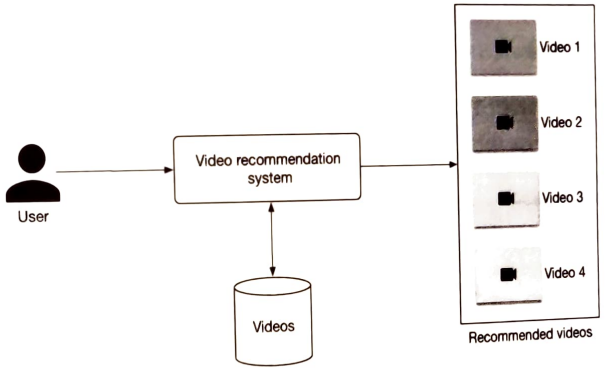


Figure 6.2: A video recommendation system's input-output

Choosing the right ML category

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

- Content-based filtering
- Collaborative filtering
- Hybrid filtering

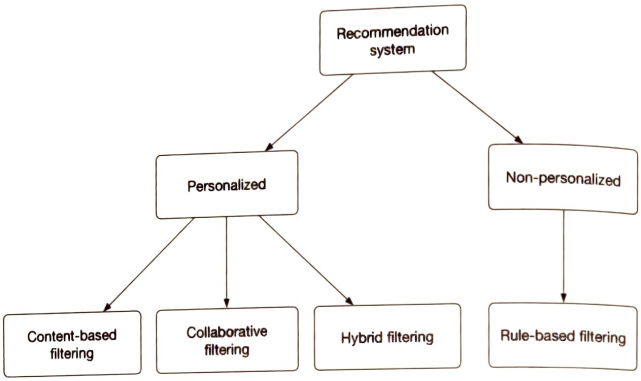


Figure 6.3: Common types of recommendation systems

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.

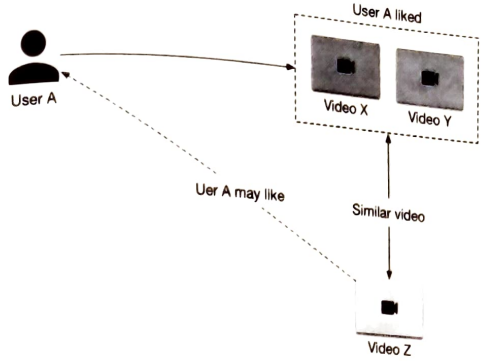


Figure 6.4: Content-based filtering

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.

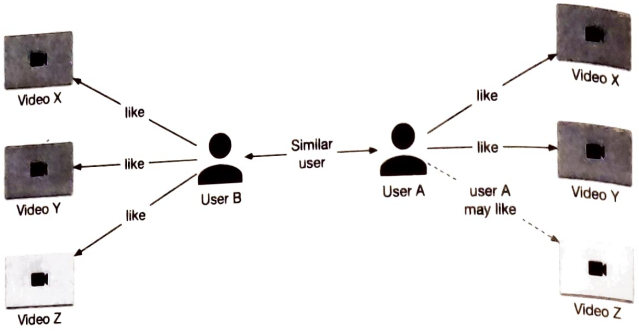


Figure 6.5: User-based collaborative filtering

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.

	Content-based 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].

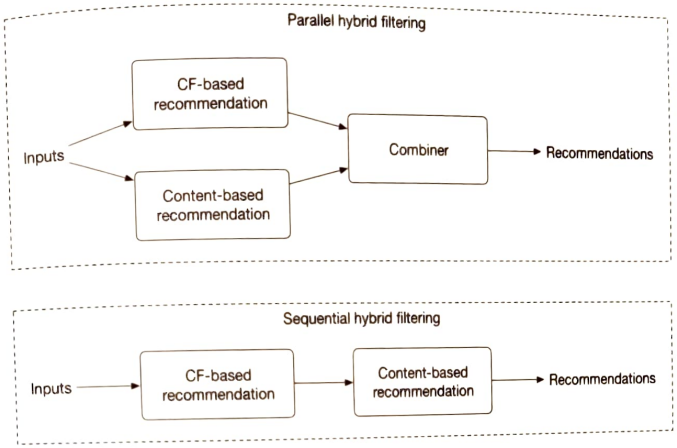


Figure 6.6: Hybrid filtering method

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	Bali, Vlog	Our 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.

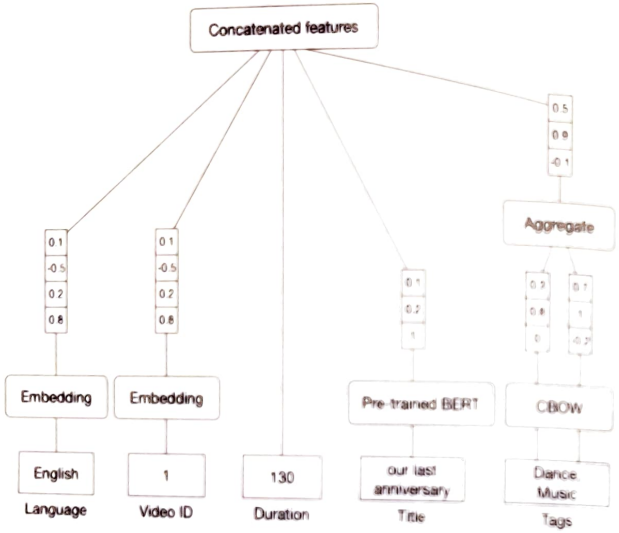


Figure 6.7: Video feature preparation

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.

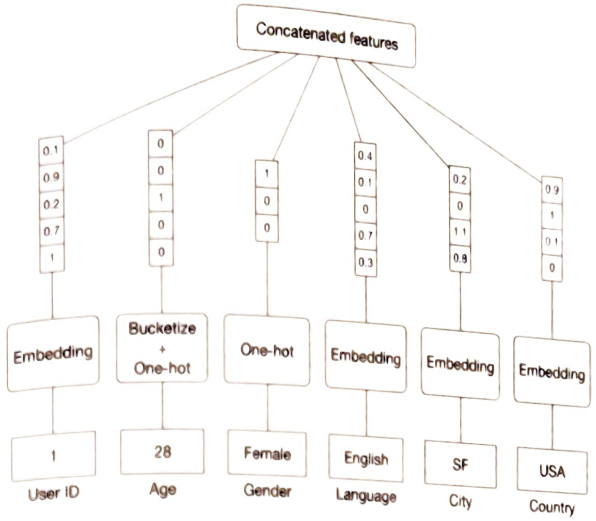


Figure 6.8: Features based on a user's demographics

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.

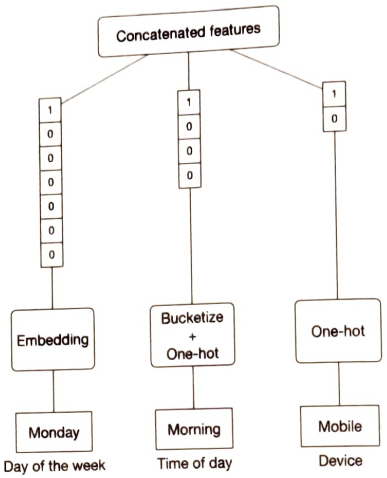


Figure 6.9: Features related to contextual information

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.

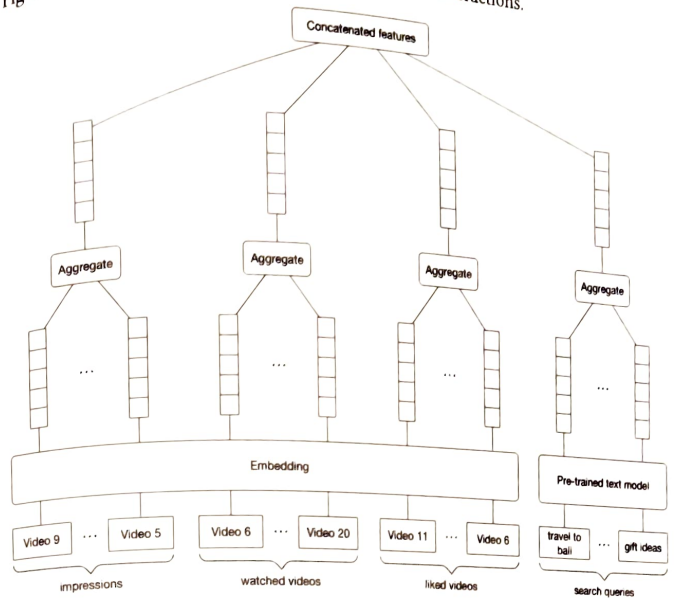


Figure 6.10: Features related to user-video interactions

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 about a video. Throughout this chapter, we refer to ⟨user, video⟩ pairs with values equal to 1 as "observed" or "positive."

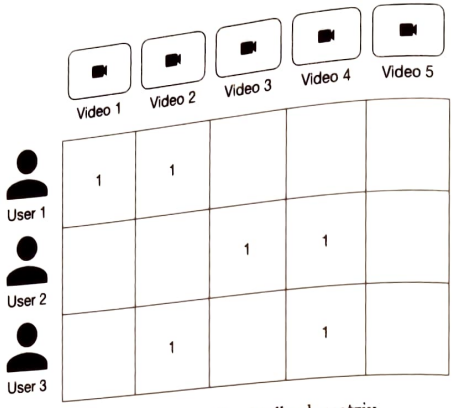


Figure 6.11: User-video feedback matrix

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.

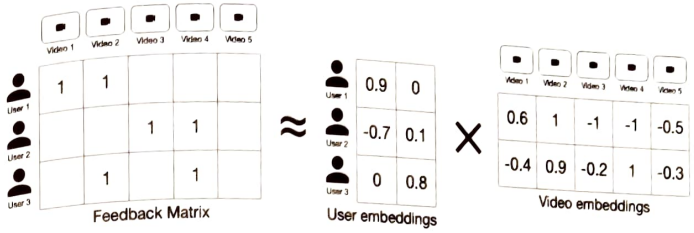


Figure 6.12: Decompose the feedback matrix into two matrices

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.)

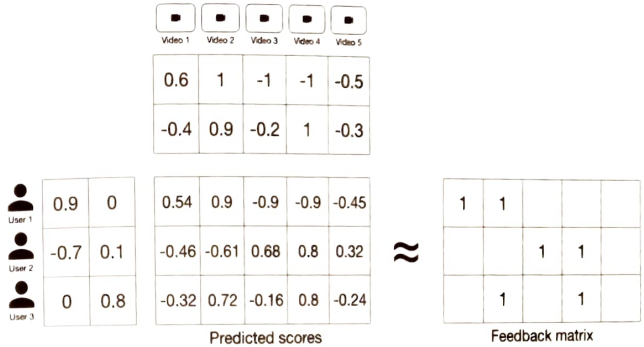


Figure 6.13: The product of embeddings should approximate the feedback matrix

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 ⟨user, video⟩ pairs
- Squared distance over observed and unobserved ⟨user, video⟩ pairs
- A weighted combination of squared distance over observed pairs and unobserved pairs

Squared distance over observed ⟨user, video⟩ 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.

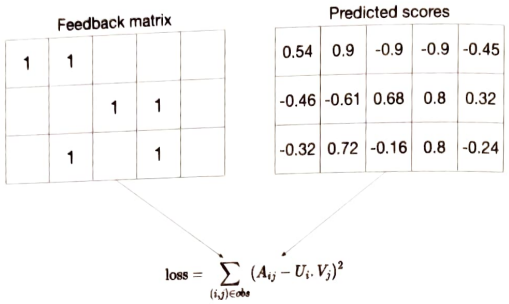


Figure 6.14: Squared distance over observed ⟨ user, video⟩ pairs

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 ⟨user, video⟩ pairs.

Squared distance over both observed and unobserved ⟨user, video⟩ 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.

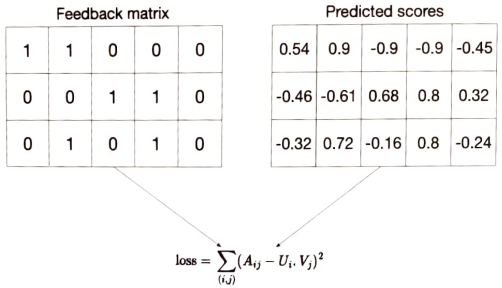


Figure 6.15: Squared distance over all ⟨ user, video⟩ pairs

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 ⟨user, video⟩ 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.

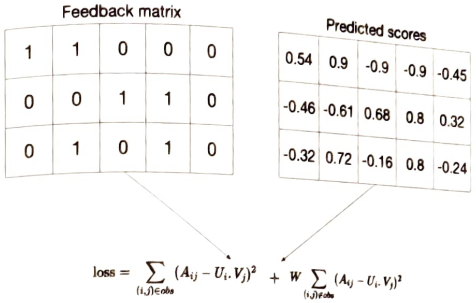


Figure 6.16: Combined loss

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:
  (a) Fix one embedding matrix (U), and optimize the other embedding (V)
  (b) Fix the other embedding matrix (V), and optimize the embedding matrix (U)
  (c) 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.

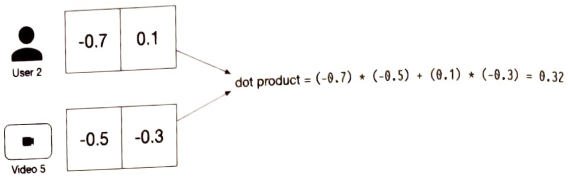


Figure 6.17: Relevance score for ⟨user 2, video 5⟩ pair

Figure 6.18 shows the predicted scores for all the ⟨user, video⟩ pairs. The system returns recommended videos based on relevance scores.

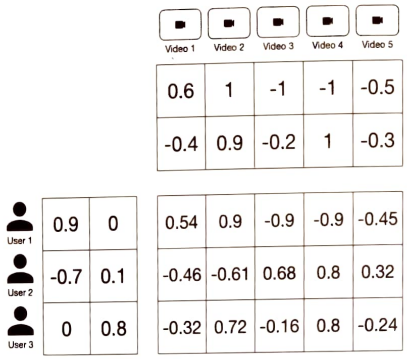


Figure 6.18: Predicted pairwise relevance scores

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.

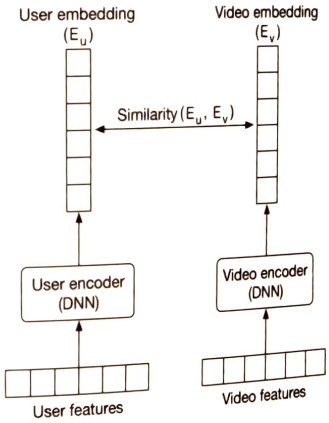


Figure 6.19: Two-tower neural network

Constructing the dataset

We construct the dataset by extracting features from different ⟨user, video⟩ 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	0, 0, 1, 0.7, -0.6, 0, 0	0, 1, 0, 0.9, 0.9, 1	1 (positive)
2	0, 1, 1, 0.2, 0.1, 1, 0	0, 1, 0, -0.1, 0.3, 1	0 (negative)

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.

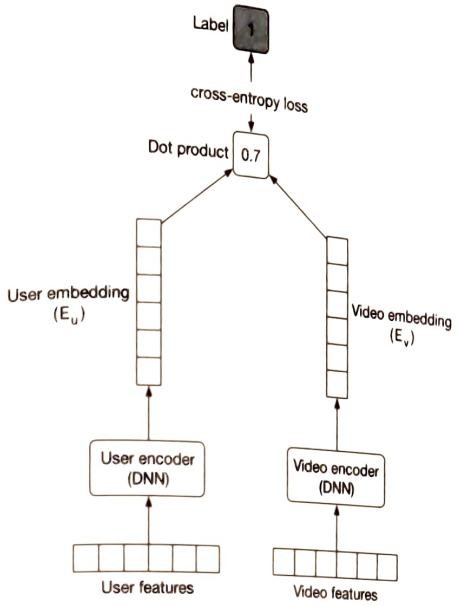


Figure 6.21: Two-tower neural network training workflow

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.

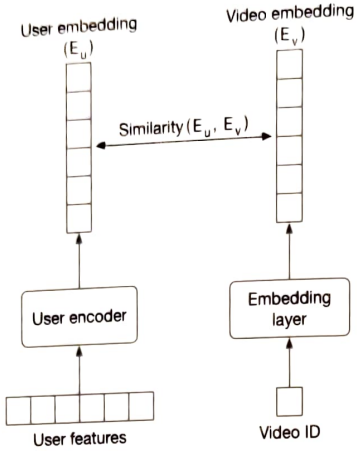


Figure 6.22: Two-tower neural network used for collaborative filtering

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.

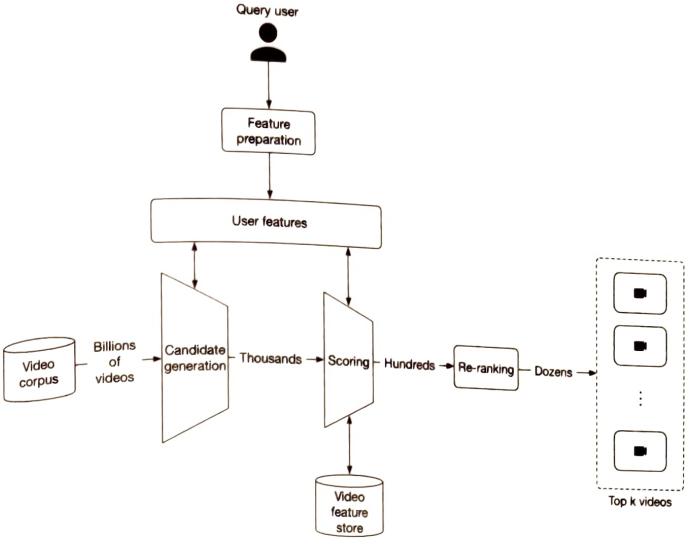


Figure 6.23: Prediction pipeline

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.

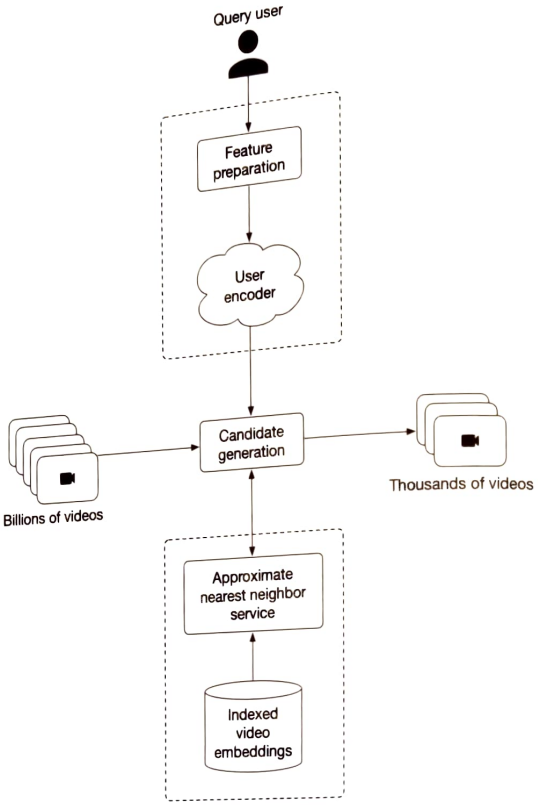


Figure 6.24: Candidate generation workflow

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.

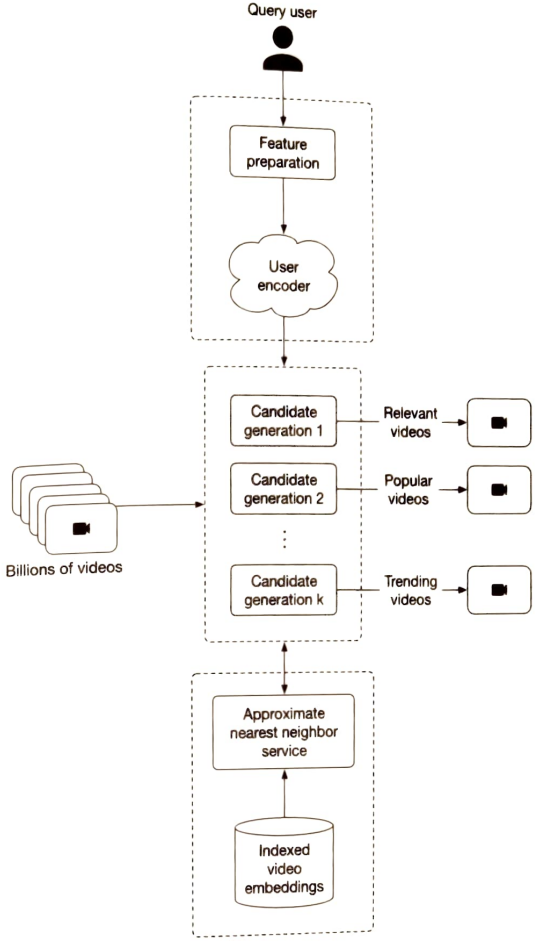


Figure 6.25: Use k candidate generations to diversify recommended videos

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.

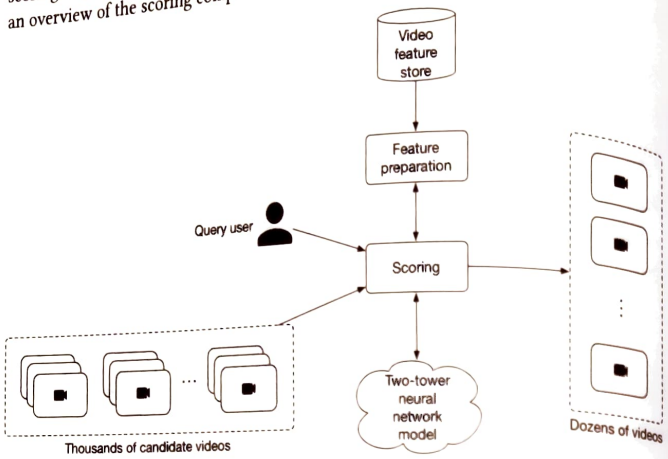


Figure 6.26: Overview of the scoring component

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.

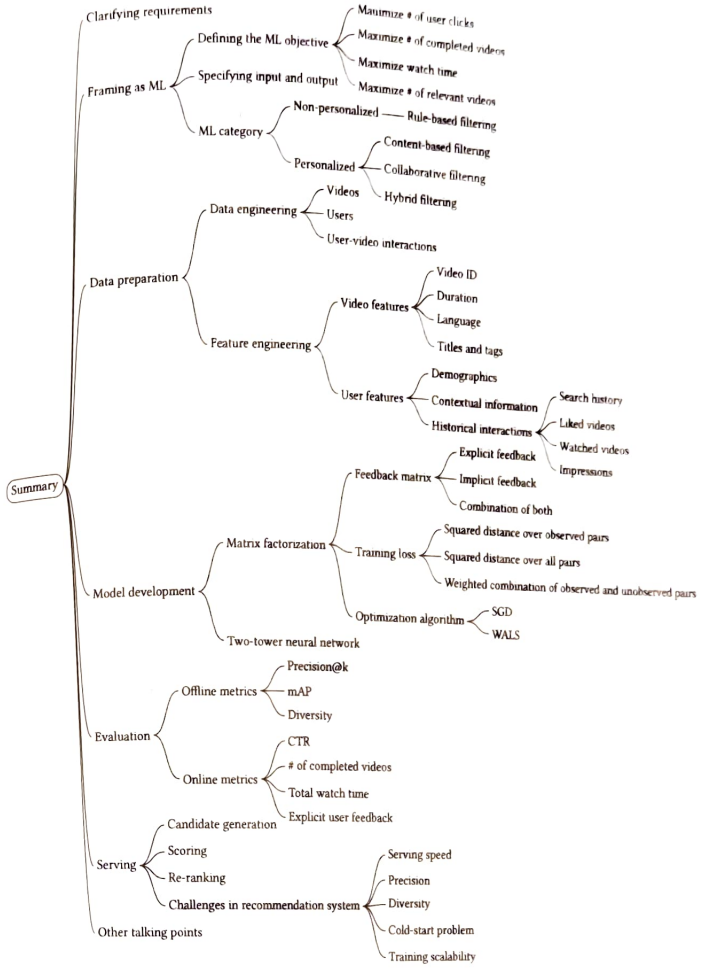
Other Talking Points

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].

Summary



Reference Material

[1] YouTube recommendation system. https://blog.youtube/inside-youtube/on-youtu
bes-recommendation-system.

[2] DNN for YouTube recommendation. https://static.googleusercontent.com/media/r
esearch.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/recommen
dation/collaborative/matrix.

[6] Stochastic gradient descent. https://en.wikipedia.org/wiki/Stochastic_gradient_de
scent.

[7] WALS optimization. https://fairyonice.github.io/Learn-about-collaborative-filteri
ng-and-weighted-alternating-least-square-with-tensorflow.html.

[8] Instagram multi-stage recommendation system. https://ai.facebook.com/blog/pow
ered-by-ai-instagrams-explore-recommender-system/.

[9] Exploration and exploitation trade-offs. https://en.wikipedia.org/wiki/Multi-arm
ed_bandit.

[10] Bias in AI and recommendation systems. https://www.searchenginejournal.com/bi
ases-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/proceedi
ngs-article/big-data/2019/09005954/1hJsfgT0qL6.

[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.