7 Event Recommendation System

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

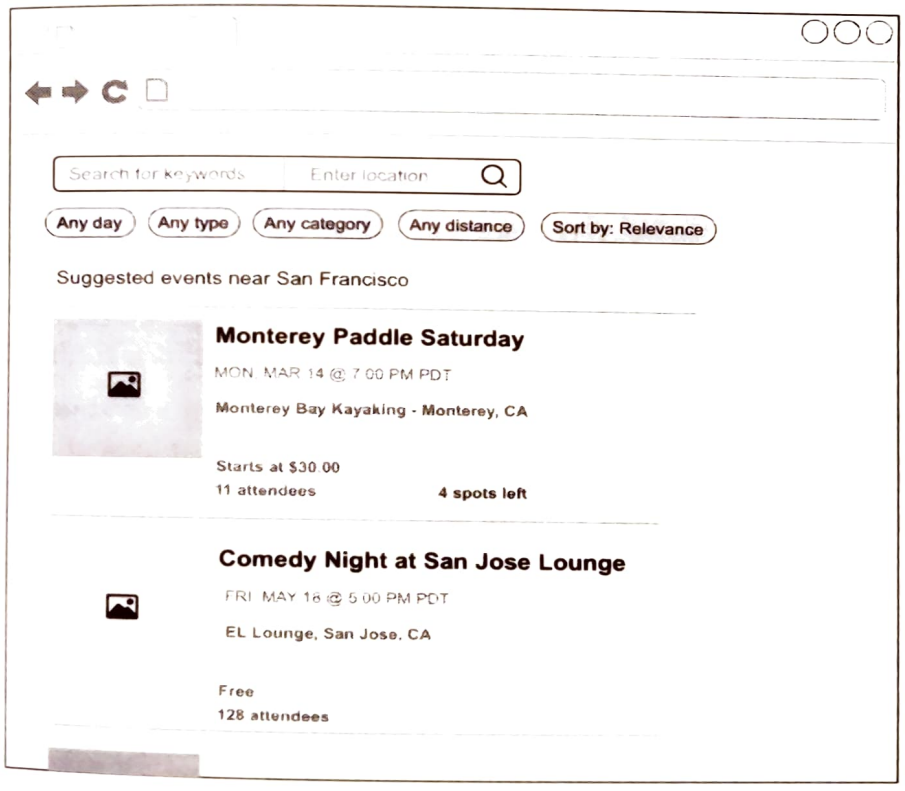


Figure 7.1: Recommended events

Clarifying Requirements

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

Candidate: What is the business objective? Can I assume the main business objective is to increase ticket sales?
Interview: Yes, that sounds good.

Candidate: Besides attending an event, can users book hotels or restaurants on the platform?
Interviewer: For simplicity, let's assume only events are supported.

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

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

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

Candidate: Do we have access to the user's current location?
Interviewer: Yes. Since this problem focuses on a location-based recommendation system, let's assume users agree to share their location data.

Candidate: Can users become friends on the platform? Friendship information is valuable for building a personalized event recommendation system.
Interviewer: Good question. Yes, let's assume users can form friendships on our platform. A friendship is bidirectional, meaning if A is a friend of B, then B is also a friend of A.

Candidate: Can users invite others to events?
Interviewer: Yes.

Candidate: Can a user RSVP to an event?
Interviewer: For simplicity, let's assume only a registration option is available for an event.

Candidate: Are the events free or paid?
Interviewer: We need to support both.

Candidate: How many users and events are available?
Interviewer: We host around 1 million total events every month.

Candidate: How many daily active users visit the website/app?
Interviewer: Assume we have one million unique users per day.

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

Let's summarize the problem statement. We are asked to design an event recommendation system, which displays a personalized list of events to users. When an event is finished, users can no longer register for it. In addition to registering for events, users can invite others to events and form friendships. The training data should be constructed online from user interactions. The primary goal of this system is to increase total ticket sales.

Frame the Problem as an ML Task

Defining the ML objective

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

Specifying the system's input and output

The input to the system is a user, and the output is the top k events ranked by relevance to the user.

Choosing the right ML category

There are different ways to solve a recommendation problem:

- Simple rules, such as recommending popular events
- Embedding-based models which rely on content-based or collaborative filtering
- Reformulating it into a ranking problem

Rule-based methods are good starting points to form a baseline. However, ML-based approaches usually lead to better outcomes. In this chapter, we reformulate the task into a ranking problem and use Learning to Rank (LTR) to solve it.

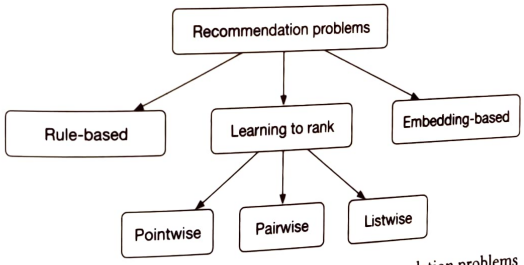


Figure 7.2: Different approaches to solving recommendation problems

LTR is a class of algorithmic techniques that apply supervised machine learning to solve ranking problems. The ranking problem can be formally defined as: "having a query and a list of items, what is the optimal ordering of the items from most relevant to least relevant to the query?" There are generally three LTR approaches: pointwise, pairwise, and listwise. Let's briefly examine each. Note that a detailed explanation of these approaches is beyond the scope of this book. If you're interested in learning more about LTR, refer to [1].

Pointwise LTR

In this approach, we go over each item and predict the relevance between the query and the item, using classification or regression methods. Note that the score of one item is predicted independently of other items.

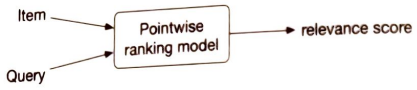


Figure 7.3: Pointwise ranking model

The final ranking is achieved by sorting the predicted relevance scores.

Pairwise LTR

In this approach, the model takes two items and predicts which item is more relevant to the query.

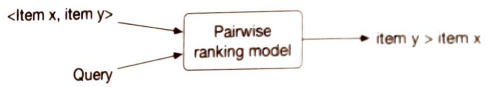


Figure 7.4: Pairwise ranking model

Some of the most popular pairwise LTR algorithms are RankNet [2], LambdaRank [3], and LambdaMART [4].

Listwise LTR

Listwise approaches predict the optimal ordering of an entire list of items, given the query.

Figure 7.5: Listwise ranking model

Some popular listwise LTR algorithms are SoftRank [5], ListNet [6], and AdaRank [7].

In general, pairwise and listwise approaches produce more accurate results, but they are more difficult to implement and train. For simplicity, we use the pointwise approach for this problem. In particular, we employ a binary classification model which takes a single event at a time and predicts the probability that the user will register for it. This approach is shown in Figure 7.6.

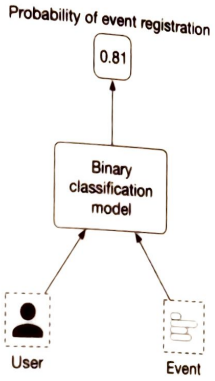


Figure 7.6: Binary classification model

Data Preparation

Data engineering

To engineer good features, we need first to understand the raw data available in the system. Since an event management platform is mainly centered around users and events, we assume the following data are available:

- Users
- Events
- Friendship
- Interactions

Users

The user data schema is shown below.

ID	Username	Age	Gender	City	Country	Language	Time zone

Table 7.1: User data schema

Events

Table 7.2 shows what the event data might look like.

ID	Host User ID	Category/ Subcategory	Description	Price	Location	Date/Time
1	5	Music Concert	Dua Lipa Tour in Miami	200-900	American Airlines Arena Miami, FL	09/18/2022 19:00-24:00
2	11	Sports Basketball	Golden State Warriors vs. Milwaukee Bucks	140-2500	Chase Center SF, CA	09/22/2022 17:00-19:00
3	7	Art Theater	The Comedy and Magic of Robert Hall	Free	San Jose Improv San Jose, CA	09/06/2022 18:00-19:30

Table 7.2: Event data

Friendship

In Table 7.3, each row represents a friendship formed between two users, along with the timestamp of when it was formed

User ID 1	User ID 2	Timestamp when friendship was formed
28	3	1658451341
7	39	1659281720
11	25	1659312942

Table 7.3: Friendship data

Interactions

Table 7.4 stores user interaction data, such as event registrations, invitations, and impressions. In practice, we may store interaction data in different databases, but for simplicity, we include them in a single table.

User ID	Event ID	Interaction type	Interaction value	Location (lat, long)	Timestamp
4	18	Impression	-	38.8951 -77.0364	1658450539
4	18	Register	Confirmation number	38.8951 -77.0364	1658451341
4	18	Invite	User 9	41.9241 -89.0389	1658451365

Table 7.4: Interaction data

Feature engineering

Event-based recommendations are more challenging than traditional recommendations. An event is fundamentally different from a movie or a book, as there is no consumption after the event ends. Events are typically short-lived, meaning the time is short between event creation and when it finishes. As a result, there are not many historical interactions available for a given event. For this reason, event-based recommendations are intrinsically cold-start and suffer from a constant new-item problem.

To overcome those issues, we put more effort into feature engineering to create as many meaningful features as possible. Due to space constraints, we will only discuss some of the most important features. In practice, the number of predictive features can be much higher.

In this section, we create features related to each of the following categories:

- Location-related features
- Time-related features
- Social-related features
- User-related features
- Event-related features.

Location-related features

How accessible is the event's location?

The accessibility of an event's location is an important factor. For example, if an event is high up in hills far from public transportation, the commute may discourage users from attending. Let's create the following features to capture accessibility:

- Walk score: Walk score is a number between 0 and 100, which measures how walkable an address is, based on the distance to nearby amenities. It is computed by analyzing various factors such as distance to amenities, pedestrian friendliness, population density, etc. We assume walk scores can be obtained from external data sources such as Google Maps, Open Street Map, etc. Table 7.5 shows walk scores bucketized into 5 categories.

Category	Walk score	Description
1	90-100	No car needed
2	70-89	Very walkable
3	50-69	Somewhat walkable
4	25-49	Car-dependent
5	0-24	Requires a car

Table 7.5: Walk score categories

- Walk score similarity: The difference between the event's walk score and the user's average walk score of previous events registered by the user.
- Transit score, transit score similarity, bike score, bike score similarity.

Is the event in the same country and city as the user?

A very important deciding factor for a user is whether the event is in the same country and city where they are located. The following two features can be created:

- If the user's country is the same as the event's country, this feature is 1, otherwise 0

- If the user's city is the same as the event's city, this feature is 1, otherwise 0

Is the user comfortable with the distance?
Some users may prefer events that are very close to their location, while others prefer events that are further away. We use the following features to capture this:

- The distance between the user's location and the event's location. This value can be obtained from external APIs and bucketized into a few categories. For example:
  - 0: less than a mile
  - 1: 1-5 miles
  - 2: 5-20 miles
  - 3: 20-50 miles
  - 4: 50-100 miles
  - 5: +100 miles
- Distance similarity: Difference between the distance to an event and the average distance (in reality, the median or percentile range can be used) to events previously registered by the user.

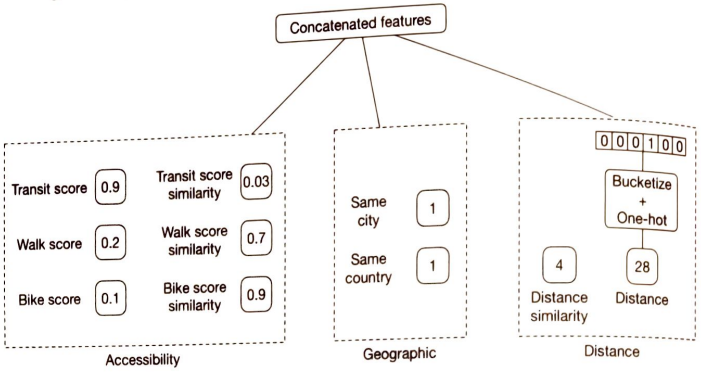


Figure 7.7: Location-related features

Time-related features

How convenient is the time remaining until an event?
Some users may plan events a few days in advance, while others don't. Let's create the following features to capture this:

- The remaining time until the event begins. This feature can be bucketized into different categories and one-hot encoded. For example:
  - 0: less than 1 hour left until the event starts
  - 1: 1-2 hours
  - 2: 2-4 hours
  - 3: 4-6 hours
  - 4: 6-12 hours
  - 5: 12-24 hours
  - 6: 1-3 days
  - 7: 3-7 days
  - 8: +7 days
- Remaining time similarity: Difference between "remaining time" and average "remaining time" of events previously registered by the user.
- The estimated travel time from the user's location to the event's location. This value will be obtained from external services and bucketized into categories.
- Estimated travel time similarity: The difference between the estimated travel time to the event in question, and the average estimated travel time of events previously registered by the user.

Are the date and time convenient for the user?
Some users may prefer events that occur at weekends, while others prefer weekdays. Some users prefer events in the morning, while others may prefer evening events. To capture a user's historical preferences for days of the week, we create a user profile. This user profile is a vector of size 7, and each value counts the number of events the user attended on a particular day. By dividing these values by the total number of attended events, we get the historical rate of event attendance for each day of the week. Figure 7.8 shows the per-day distribution of a user's previously attended events. As we can see, this user has never attended an event on Monday or Wednesday, so displaying an event that occurs on Wednesday may not be a good recommendation for this user. Per-hour user profiles can be created using a similar approach. Similarly, we add day and hour similarity.

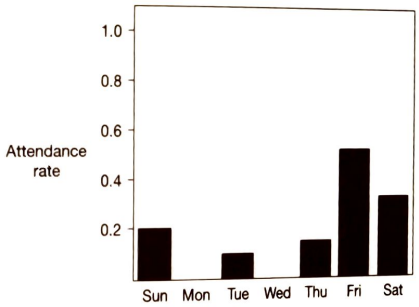


Figure 7.8: Per-day distribution of the event data

A summary of time-related features is shown in Figure 7.9.

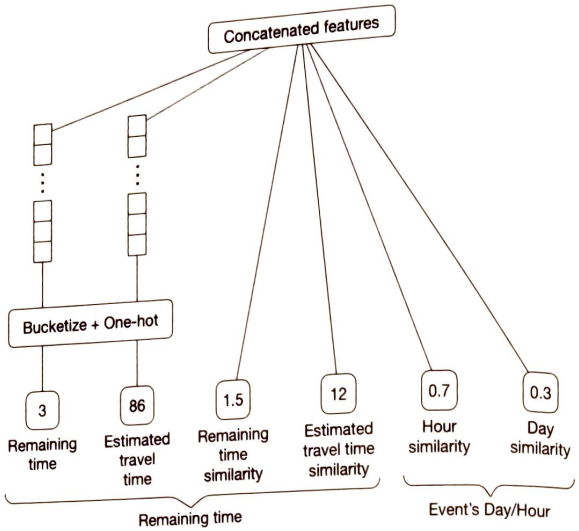


Figure 7.9: Time-related features overview

Social-related features

How many people are attending this event?
In general, users are more likely to register for an event if there are a lot of other attendees. Let's extract the following features to capture this:

- Number of users registered for this event
- The ratio of the total number of registered users to the number of impressions
- Registered user similarity: The difference between the number of registered users for the event in question and previously registered events

Features related to attendance by friends
A user is more likely to register for an event if their friends are attending it. Here are some of the features we can use:

- Number of the user's friends who registered for this event
- The ratio of the number of registered friends to the total number of friends
- Registered friend similarity: Difference between the number of registered friends for the event in question and previously registered events

Is the user invited to this event by others?
Users are more likely to attend events to which they are invited. Some features that might be helpful are:

- The number of friends who invited this user to the event
- The number of fellow users who invited this person to the event

Is the event's host a friend of the user?
Users tend to attend events created by their friends. We create a binary feature to reflect this: if the event's host is the user's friend, this value is 1, otherwise, 0.

How often has the user attended previous events created by this host?
Some users are interested in following a particular host's events.

User-related features

Age and gender
Some events are geared toward specific ages and genders. For example, "Women in Tech" and "Life lessons to excel in your 30s" are examples of events that may be specific to certain demographic groups. We create two features to capture this:

- User's gender, encoded with one-hot encoding
- User's age, bucketized into multiple categories and encoded with one-hot encoding

Event-related features

Price of event:
The price of an event might affect the user's decision to register for it. Some features to use are:

- Event's price, bucketized into a few categories. For example:
  - 0: Free
  - 1: $1-$99
  - 2: $100-$499
  - 3: $500-$1,999
  - 4: +$2,000
- **Price similarity:** Difference between the price of the event in question and the average price of events previously registered for by the user.

How similar is this event's description to previously registered descriptions?
This indicates the user's interests, based on previously registered events. For example, if the word "concert" repeatedly appears in the descriptions of previous events, it may indicate the user is interested in concert events. To capture this, we create a feature that represents the similarity between the event's description and the descriptions of previously registered events by the user. To compute the similarity, the description is converted into a numerical vector using TF-ID, and similarity is calculated using cosine distance.

Note, this feature might be noisy as descriptions are manually provided by hosts. We can experiment by training our model with and without this feature, to measure its importance.

Figure 7.10 shows an overview of user features, event features, and social-related features.

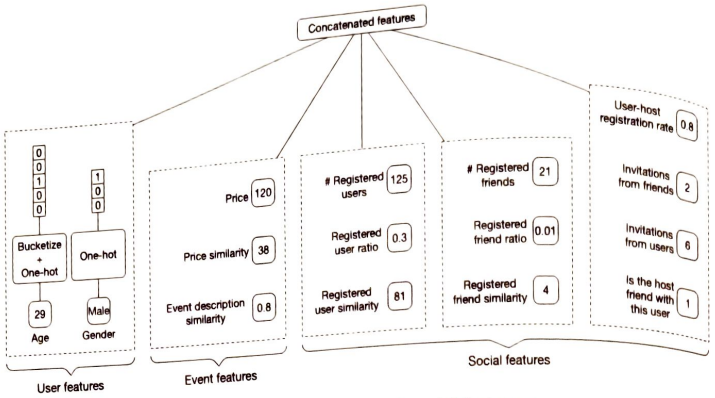


Figure 7.10: User, event, and social features

The features listed above are not exhaustive. There are lots of other predictive features that can be created in practice. For example, host-related features such as the host's popularity, user's search history, event's category, auto-generated event tags, etc. At an interview, it's not necessary to follow this section strictly. You can use it as a starting point and then discuss topics that are more relevant to the interviewer. Here are some potential talking points you might want to elaborate on:

- **Batch vs. streaming features:** Batch (static) features refer to features that change less frequently, such as age, gender, and event description. These features can be computed periodically using batch processing and stored in a feature store. In contrast, streaming (aka dynamic) features change quickly. For example, the number of users registered for an event and the remaining time until an event, are dynamic features. The interviewer may want you to dive deeper into this topic and discuss batch vs. online processing in ML. If you're interested to learn more, refer to [8].
- **Feature computation efficiency.** Computing features in real-time is not efficient. You may want to discuss this issue and possible ways to avoid it. For example, instead of computing the distance between the user's current location and the event's location as a feature, we can pass both locations to the model as two separate features, and rely on the model to implicitly compute useful information from the two locations. To learn more about how to prepare location data for ML models, refer to [9].
- **Using a decay factor** for features that rely on the user's last X interactions. A decay factor gives more weight to the user's recent interactions/behaviors.
- **Using embedding learning** to convert each event and user into an embedding vector. Those embedding vectors are used as the features representing the events and users.
- **Creating features from users' attributes may create bias**. For example, relying on age or gender to decide if an applicant is a good match for a job, may lead to discrimination. Since we create features from users' attributes, it's important to be aware of potential bias issues.

Model Development

Model selection

Binary classification problems can be solved by various ML methods. Let's take a look at the following:

- Logistic regression
- Decision tree
- Gradient-boosted decision tree (GBDT)
- Neural network

Logistic regression (LR)

LR models the probability of a binary outcome by using a linear combination of one or multiple features. For the details of LR, refer to [10].

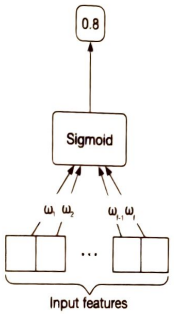


Figure 7.11: Logistic regression

Let's see the pros and cons of LR.

Pros:

- **Fast inference speed.** Computing a weighted combination of input features is fast.
- **Efficient training.** Given the simple architecture, it's easy to implement, interpret, and train quickly.
- Works well when the data is linearly separable (Figure 7.12).
- **Interpretable and easy to understand.** The weights assigned to each feature indicate the importance of different features, which gives us insight into why a decision was made.

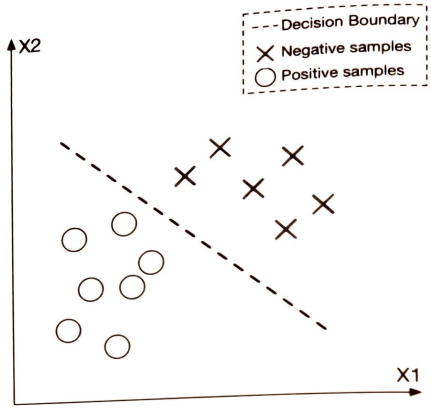


Figure 7.12: A linearly separable data with LR's decision boundary

Cons:

- **Non-linear problems can't be solved** with LR, since it uses a linear combination of input features.
- **Multicollinearity** occurs when two or more features are highly correlated. One of the known limitations of LR is that it cannot learn the task well when multicollinearity is present in the input features.

In our system, the number of input features can be very large. Often, these features have complex and non-linear relations with the target variable (binary outcome). This complexity might be hard for LR to learn.

Decision tree

Decision trees are another class of learning methods that use a tree-like model of decisions and their possible consequences to make predictions. Figure 7.13 shows a simple decision tree with two features: age and gender. It also shows the corresponding decision boundary. Each leaf node in the decision tree indicates a binary outcome where "+" indicates the given input is classified as positive, and "-" means negative. To learn more about decision trees, refer to [11].

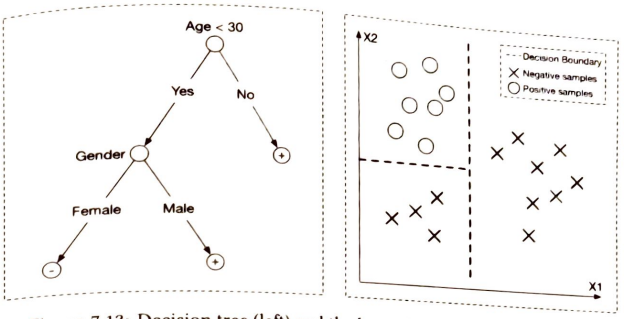


Figure 7.13: Decision tree (left) and the learned decision boundary (right)

Pros:

- **Fast training:** Decision trees are quick to train.
- **Fast inference:** Decision trees make predictions quickly at inference time.
- **Little to no data preparation:** Decision tree models don't require data normalization or scaling, since the algorithm does not depend on the distribution of the input features.
- **Interpretable** and easy to understand. Visualizing the tree provides good insights into why a decision was made and what the important decision factors are.

Cons:

- **Non-optimal decision boundary:** decision tree models produce decision boundaries that are parallel to the axes in the feature space (Figure 7.13). This may not be the optimal way to find a decision boundary for certain data distributions.
- **Overfitting:** Decision trees are very sensitive to small variations in data. A small change in input data may lead to different outcomes at serving time. Similarly, a small change in training data can lead to a totally different tree structure. This is a major issue and makes predictions less reliable.

In practice, naive decision trees are rarely used. The reason is that they are too sensitive to variations of input data. To reduce the sensitivity of decision trees, two techniques are commonly used:

- Bootstrap aggregation (Bagging)
- Boosting

These two techniques are widely used across the tech industry. It's essential to understand how they work. Let's take a closer look.

Bagging

Bagging is the ensemble learning method that trains a set of ML models in parallel, on multiple subsets of the training data. In bagging, the predictions of all these trained models are combined to make a final prediction. This significantly reduces the model's sensitivity to the change in data (variance).

One example of bagging is the commonly used "random forest" model [12]. Random forest builds multiple decision trees in parallel during training, to reduce the model's sensitivity. To make a prediction, each decision tree independently predicts the output class (positive or negative) of the given input, and then a voting mechanism is used to combine these predictions to make a final prediction. Figure 7.14 shows a random forest with three decision trees.

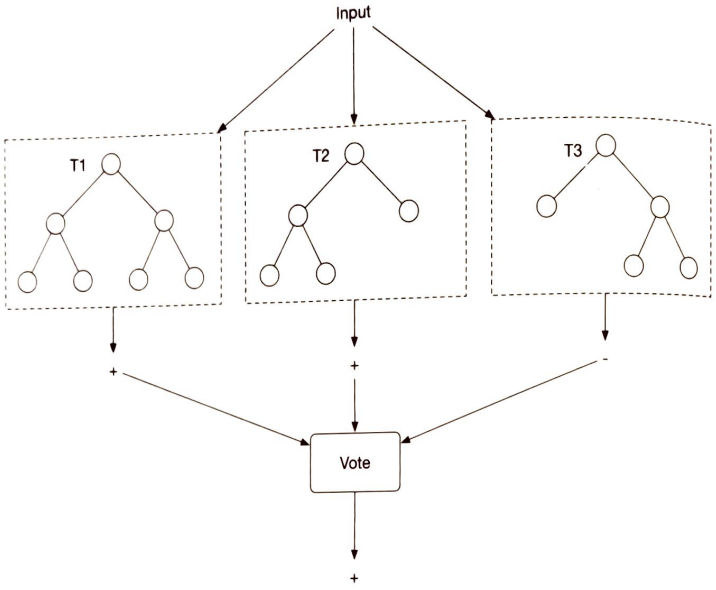


Figure 7.14: Random forest

The bagging technique has the following advantages:

- Reduces the effect of overfitting (high variance).
- Does not significantly increase training time because the decision trees can be trained in parallel.
- Does not add much latency at the inference time because decision trees can process the input in parallel.

Despite its advantages, bagging is not helpful when the model faces underfitting (high bias). To overcome bagging's drawbacks, let's discuss another technique called boosting.

Boosting

In ML, boosting involves training several weak classifiers sequentially to reduce prediction errors. The phrase "weak classifier" refers to a simple classifier that performs slightly better than random guesses. In boosting, multiple weak classifiers are converted into a single strong learning model. Figure 7.15 shows an example of boosting.

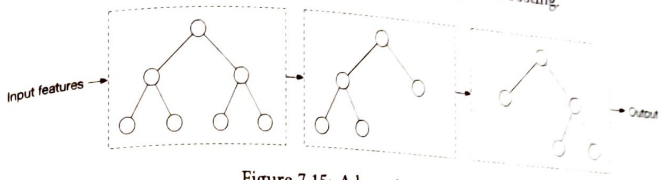


Figure 7.15: A boosting example

Pros:

- **Boosting reduces bias and variance.** Combining weak classifiers leads to a strong model less sensitive to the change in data. To learn more about bias/variance trade-offs, refer to [13].

Cons:

- **Slower training and inference.** Given the classifiers are trained based on the mistakes of the previous classifiers, they work sequentially. This adds to the serving time due to the sequential nature of boosting.

The boosting method is usually preferred over bagging in practice because bagging is not helpful in cases of bias, whereas boosting reduces the effect of both bias and variance.

Typical boosting-based decision trees are Adaboost [14], XGBoost [15], and Gradient boost [16]. They are commonly employed to train classification models.

GBDT

GBDT is a commonly used tree-based model, utilizing GradientBoost to improve decision trees. Some variants of GBDT, such as XGBoost [15], have demonstrated strong performance in various ML competitions [17]. If you're interested in learning more about GBDT, refer to [18] [19].

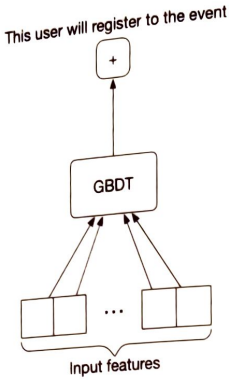


Figure 7.16: A GBDT model with a binary output

Here are the pros and cons of the GBDT model.

Pros:

- **Easy data preparation:** Similar to decision trees, it does not require data preparation.
- **Reduces variance:** GBDT reduces variance as it uses the boosting technique.
- **Reduces bias:** GBDT reduces the prediction error by leveraging several weak classifiers, iteratively improving upon the misclassified data points from the previous classifiers.
- Works well with structured data.

Cons:

- **Lots of hyperparameters to tune**, such as the number of iterations, tree depth, regularization parameters, etc.
- GBDT does not work well on **unstructured data** such as images, videos, audio, etc.
- **Unsuitable for continual learning** from streaming data.

In our case, since the created features are structured data, GBDT or one of its variants – such as XGBoost – is a good choice to experiment with.

A major drawback of GBDT is that it is unsuitable for continual learning. In an event recommendation system, new data continuously becomes available to the system, such as recent user interactions, registrations, new events, and even new users. In addition, users' tastes and interests may change over time. It is vital for a good event recommendation system to adapt itself to new data, continuously. Without the possibility of continual learning, it is very costly to retrain GBDT from scratch regularly. Next, we explore neural networks which overcome this limitation.

Neural network (NN)

In an event recommendation system, we have many features that might not correlate linearly with the outcome. Learning these complex relationships is difficult. In addition, continual learning is necessary for adapting the model to new data.

NNs are great at solving those challenges. They are capable of learning complex tasks with non-linear decision boundaries. Additionally, NN models can be fine-tuned on new data very easily, making them ideal for continual learning.

If you are unfamiliar with the details of NNs, you are encouraged to read [20].

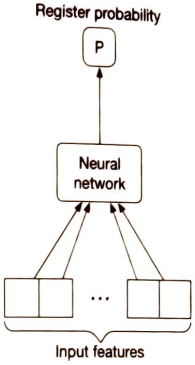


Figure 7.17: Neural network input-output

Let's see its pros and cons.

Pros

- **Continual learning:** NNs are designed to learn from data and improve themselves continually.
- **Works well with unstructured data** such as text, image, video, or audio.
- **Expressiveness:** NNs have expressive power due to their high number of learning parameters. They can learn very complex tasks and non-linear decision boundaries.

Cons

- **Computationally expensive** to train.
- **The quality of input data strongly influences the outcome:** NNs are sensitive to input data. For example, if input features are in very different ranges, the model may converge slowly during the training phase. An important step for NNs is data preparation, such as normalization, log-scaling, one-hot encoding, etc.
- **Large training data** is required to train NNs.
- **Black-box nature:** NNs are not interpretable, meaning it's not easy to understand

the influence of each feature upon the outcome, as the input features go through multiple layers of non-linear transformations.

Which model should we select?

Picking the right model is challenging. We often need to experiment with different models to determine which works best. We can choose the right model based on various factors:

- Complexity of the task
- Data distribution and data type
- Product requirements or constraints, such as training cost, speed, model size, etc

In this problem, both GBDTs and NNs are good candidates for experimentation. We start with the GBDT variant, XGBoost, since it is fast to implement and train. The result can be used as an initial baseline.

Once we have a baseline, we explore the possibility of building a better model with NNs. Neural networks are expected to work well here for the following reasons:

- Massive training data is available in our system. Users continuously interact with the system by registering for events, inviting friends, publishing new events, etc. Given the number of users, this creates a massive amount of data available for training.
- Data may not be linearly separable, and neural networks can learn non-linear data.

When designing a NN architecture, several hyperparameters must be considered, including the number of hidden layers, neurons in each layer, activation function, etc. These can be determined by employing hyperparameter tuning techniques. NN architectural details are not typically the main focus of ML system design interviews, since there is no systematic way to choose the right architecture.

Model training

Constructing the dataset

Building training and evaluation datasets is an essential step in developing a model. For example, let's look at how we compute features and their labels.

To construct a single data point, we extract a ⟨user, event⟩ pair from the interaction data and compute the input features from the pair. We then label the data point with 1 if the user has registered for the event, and 0 if not.

#	Extracted ⟨user, event⟩ features							Label
1		1	0	1	1	0	1	1
2		0	0	0	1	1	0	0

Figure 7.18: Constructed dataset

One issue we may face after constructing the dataset is class imbalance. The reason is that users may explore tens or hundreds of events before registering for one. Therefore, the number of negative ⟨user, event⟩ pairs is significantly higher than positive data points. We can use one of the following techniques to address the class imbalance issue:

- Use focal loss or class-balanced loss to train the classifier
- Undersample the majority class

Choosing the loss function

Since the model is a binary classification model, we use a typical classification loss function such as binary cross-entropy to optimize the neural network model.

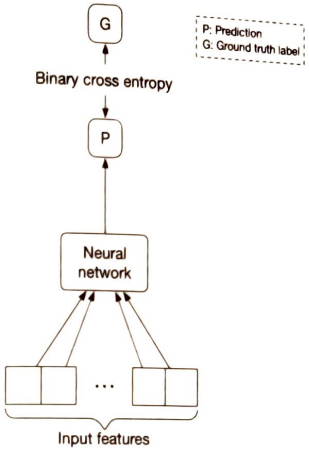


Figure 7.19: Loss between the prediction and the label

Evaluation

Offline metrics

To evaluate the ranking system, we consider the following options.

Recall@k or Precision@k. These metrics are not good fits because they do not consider the ranking quality of the output.

MRR, nDCG, or mAP. These three metrics are commonly used to measure ranking quality. But which one is best?

MRR focuses on the rank of the first relevant item in the list, which is suitable in systems where only one relevant item is expected to be retrieved. However, in an event recommendation system, several recommended events may be relevant to the user. MRR is not a good fit.

nDCG works well when the relevance score between a user and an item is non-binary. In

contrast, mAP works only when the relevance scores are binary. Since events are either relevant (a user registered for it) or irrelevant (a user saw the event but did not register), mAP is a better fit.

Online metrics

In our case, the business objective is to increase revenue by increasing ticket sales. To measure the impact of the system on revenue, let's explore the following metrics:

- Click-through rate (CTR)
- Conversion rate
- Bookmark rate
- Revenue lift

CTR. A ratio showing how often users who see recommended events go on to click on an event.

$$CTR = \frac{\text{total number of clicked events}}{\text{total number of impressions}}$$

A high CTR shows our system is good at recommending events that users click on. Having more clicks generally means more event registrations.

However, relying only on CTR as the online metric may be insufficient. Some events are clickbait. Ideally, we would like to measure how relevant recommended events are for the user. This metric is called the conversion rate, which we discuss now.

Conversion rate. A ratio showing how often users who see recommended events go on to register for them. The formula is:

$$\text{Conversion rate} = \frac{\text{total number of event registrations}}{\text{total number of impressions}}$$

A high conversion rate indicates users register for recommended events more often. For example, a conversion rate of 0.3 means that users, on average, register for 3 events out of every 10 recommended events.

Bookmark rate. A ratio showing how often users bookmark recommended events. This is based on the assumption that the platform allows users to save or bookmark an event.

Revenue lift. This is the increase in revenue as a result of event recommendations.

Serving

In this section, we propose an ML system design that can be used to serve requests. As Figure 7.20 shows, there are two main pipelines in the design:

- Online learning pipeline
- Prediction pipeline

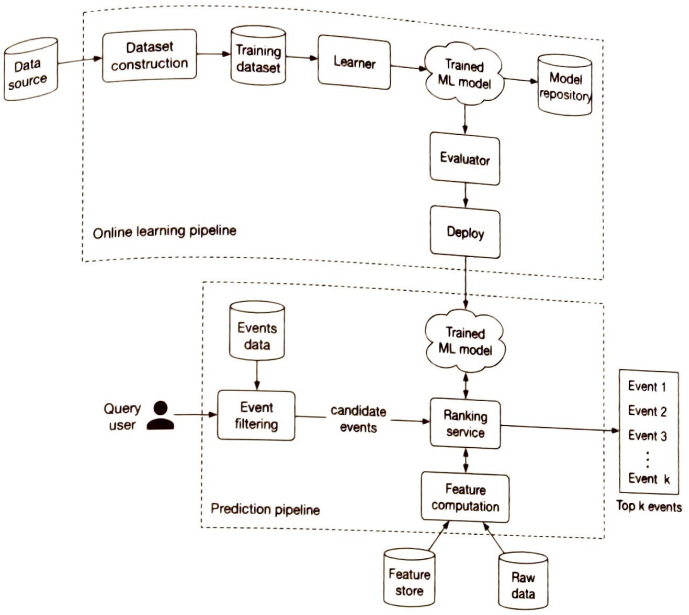


Figure 7.20: ML system design

Online learning pipeline

As described earlier, event recommendations are intrinsically cold-start and suffer from a constant new-item problem. Consequently, the model must be continuously fine-tuned to adapt to new data. This pipeline is responsible for continuously training new models by incorporating new data, evaluating the trained models, and deploying them.

Prediction pipeline

The prediction pipeline is responsible for predicting the top k most relevant events to a given user. Let's discuss some of the most important components of the prediction pipeline.

Event filtering

The event filtering component takes the query user as input and narrows down the events from 1 million to a small subset of events. This is based upon simple rules, such as event locations, or other types of user filters. For example, if a user adds a "concerts only" filter, the component quickly narrows down the list to a subset of candidate events. Since these types of filters are common in event recommendation systems, they can be used to

significantly reduce our search space from potentially millions of events, to hundreds of candidate events.

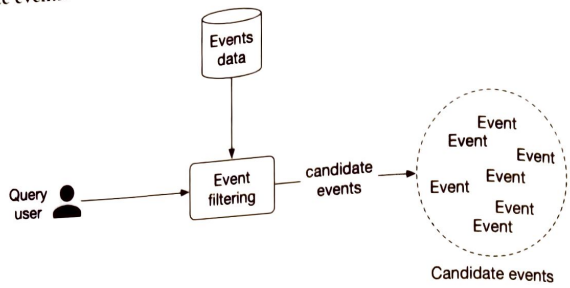


Figure 7.21: Event filtering input-output

Ranking service

This service takes the user and candidate events produced by the filtering component as input, computes features for each ⟨user, event⟩ pair, sorts the events based on the probabilities predicted by the model, and outputs a ranked list of top k most relevant events to the user.

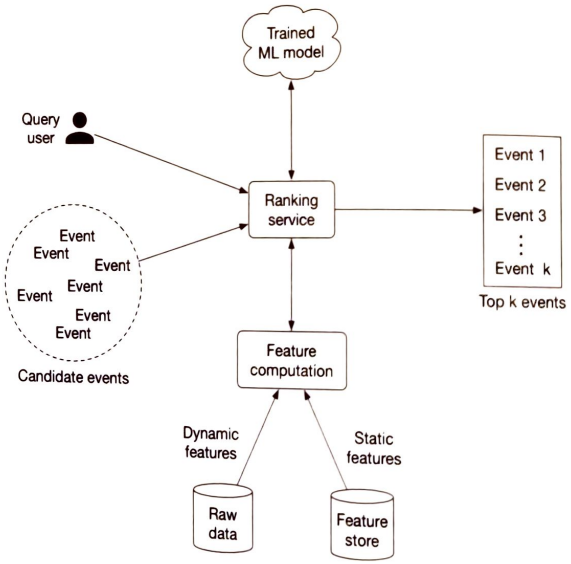


Figure 7.22: Ranking service workflow

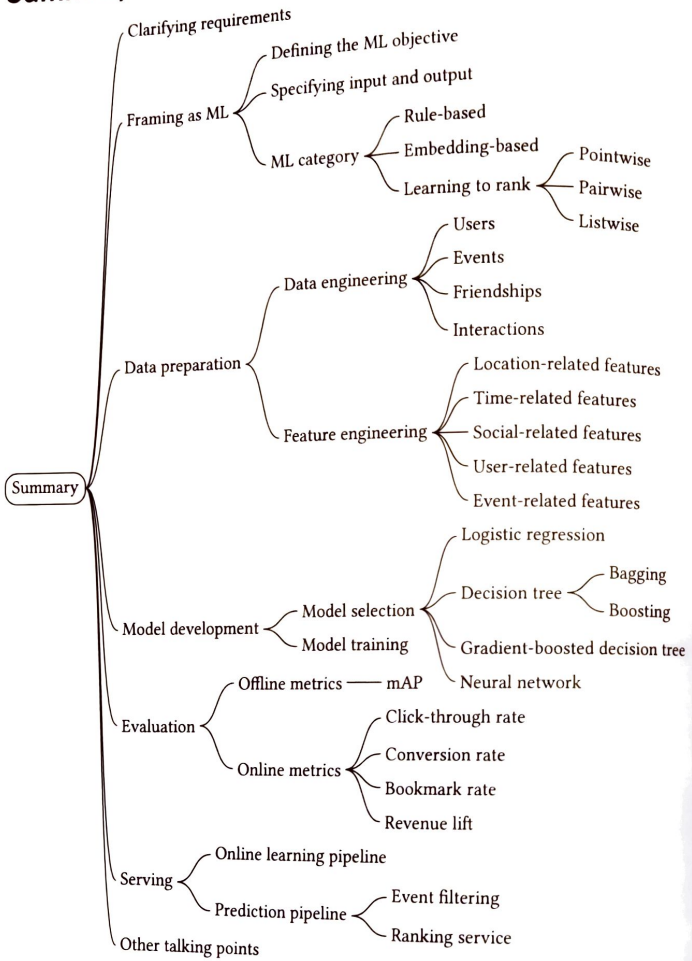
Ranking service interacts with the feature computation component responsible for computing features that the model expects. Static features are obtained from a feature store, while dynamic features are computed in real-time from the raw data.

Other Talking Points

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

- What are the different types of bias we may observe in this system [21].
- How to utilize feature crossing to achieve more expressiveness [22].
- Some users like to see a diverse list of events. How to ensure the recommended events are diverse and fresh [23]?
- We utilize the user's attributes to train a model. We also rely on users' live locations. What are additional considerations related to privacy and security [24]?
- Event management platforms are usually two-sided marketplaces, where event hosts are the suppliers and users fulfill the demand side. How to ensure the system is not optimized for one side only? Additionally, how to keep the platform fair for different hosts? To learn more about unique challenges in two-sided marketplaces, refer to [25].
- How to avoid data leakage when constructing the dataset [26].
- How to determine the right frequency to update the models [27].

Summary



Reference Material

[1] Learning to rank methods. https://livebook.manning.com/book/practical-recommender-systems/chapter-13/53.

[2] RankNet paper. https://icml.cc/2015/wp-content/uploads/2015/06/icml_ranking.pdf.

[3] LambdaRank paper. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/lambdarank.pdf.

[4] LambdaMART paper. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/MSR-TR-2010-82.pdf.

[5] SoftRank paper. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/SoftRankWsdm08Submitted.pdf.

[6] ListNet paper. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2007-40.pdf.

[7] AdaRank paper. https://dl.acm.org/doi/10.1145/1277741.1277809.

[8] Batch processing vs stream processing. https://www.confluent.io/learn/batch-vs-real-time-data-processing/#:~:text=Batch%20processing%20is%20when%20the,data%20flows%20through%20a%20system.

[9] Leveraging location data in ML systems. https://towardsdatascience.com/leveraging-geolocation-data-for-machine-learning-essential-techniques-192ce3a969bc#:~:text=Location%20data%20is%20an%20important,based%20on%20your%20customer%20data.

[10] Logistic regression. https://www.youtube.com/watch?v=yIYKR4sgzI8.

[11] Decision tree. https://careerfoundry.com/en/blog/data-analytics/what-is-a-decision-tree/.

[12] Random forests. https://en.wikipedia.org/wiki/Random_forest.

[13] Bias/variance trade-off. http://www.cs.cornell.edu/courses/cs578/2005fa/CS578.bagging.boosting.lecture.pdf.

[14] AdaBoost. https://en.wikipedia.org/wiki/AdaBoost.

[15] XGBoost. https://xgboost.readthedocs.io/en/stable/.

[16] Gradient boosting. https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/.

[17] XGBoost in Kaggle competitions. https://www.kaggle.com/getting-started/145362.

[18] GBDT. https://blog.paperspace.com/gradient-boosting-for-classification/.

[19] An introduction to GBDT. https://www.machinelearningplus.com/machine-learning/an-introduction-to-gradient-boosting-decision-trees/.

[20] Introduction to neural networks. https://www.youtube.com/watch?v=0twSSFZN9Mc.

[21] Bias issues and solutions in recommendation systems. https://www.youtube.com/watch?v=pPq9iyGIZZ8.

[22] Feature crossing to encode non-linearity. https://developers.google.com/machine-learning/crash-course/feature-crosses/encoding-nonlinearity.

[23] Freshness and diversity in recommendation systems. https://developers.google.com/machine-learning/recommendation/dnn/re-ranking.

[24] Privacy and security in ML. https://www.microsoft.com/en-us/research/blog/privacy-preserving-machine-learning-maintaining-confidentiality-and-preserving-trust/.

[25] Two-sides marketplace unique challenges. https://www.uber.com/blog/uber-eats-recommending-marketplace/.

[26] Data leakage. https://machinelearningmastery.com/data-leakage-machine-learning/.

[27] Online training frequency. https://huyenchip.com/2022/01/02/real-time-machine-learning-challenges-and-solutions.html#towards-continual-learning.

8 Ad Click Prediction on Social Platforms

Introduction

Online advertising allows advertisers to bid and place their advertisements (ads) on a platform for measurable responses such as impressions, clicks, and conversions. Displaying relevant ads to users is a fundamental for many online platforms such as Google, Facebook, and Instagram.

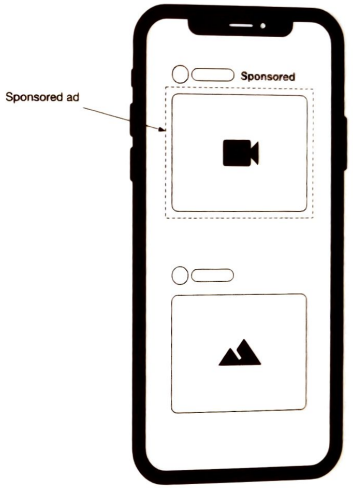


Figure 8.1: Sponsored ads placed on the user's timeline

In this chapter, we design an ad click prediction (also known as click-through rate pre-

diction) system similar to what popular social media platforms use.

Clarifying Requirements

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

Candidate: Can I assume the business objective of building an ad prediction system is to maximize revenue?
Interviewer: Yes, that's correct.

Candidate: There are different types of ads, such as video and image ads. In addition, ads can be displayed in different sizes and formats, like users' timelines, pop-up ads, etc. For simplicity, can I assume ads are placed on users' timelines only, and every click generates the same revenue?
Interviewer: That sounds good.

Candidate: Can the system show the same ad to the same user more than once?
Interviewer: Yes, we can show an ad more than once. Sometimes, an ad turns into a click after multiple impressions. In reality, companies have a "fatigue period", that is, they don't show the same ad to the same user for X days if the user repeatedly ignores it. For simplicity, assume we have no fatigue period.

Candidate: Do we support the "hide this ad" feature? How about "block this advertiser"? These kinds of negative feedback help us to detect irrelevant ads.
Interviewer: Good question. Let's assume users can hide an ad they don't like. "Block this advertiser" is an interesting feature, but we don't need to support it for now.

Candidate: Would it be okay to assume that the training dataset should be constructed using user and ad data, and the labels should be based on user-ad interactions?
Interviewer: Sure.

Candidate: We can construct positive training data points via user clicks, but how do we generate negative data points? Can we assume any impression that is not clicked is a negative data point? What if the user scrolls fast and doesn't spend time seeing the ad? What if we count an impression as negative, but eventually, the user clicks on it?
Interviewer: These are excellent questions. What are your thoughts?

Candidate: If an ad is visible on a user's screen for a certain duration but not clicked, we can count it as a negative data point. An alternative approach would be to assume impressions are negative until a click is observed. In addition, we can rely on negative feedback such as "hide this ad" to label negative data points.
Interviewer: Makes sense! In practice, we might use other complex techniques to label negative data points [1]. For this interview, let's proceed with your suggestions.

Candidate: In ad click prediction systems, it's critical for the model to learn from new interactions continuously. Is it fair to assume continual learning is a necessity here?
Interviewer: Great point. Experiments have shown that even a 5-minute delay in updating models can damage performance [1].

Let's summarize the problem statement. We are asked to design an ad click prediction system. The business objective of the system is to maximize revenue. The ads are placed only on users' timelines, and each click generates the same revenue. It is necessary to train the model on new interactions continually. We construct the dataset from the user and ad data, and label them based on interactions. In this chapter, we will not discuss AdTech-specific topics as they are not relevant to ML interviews. To learn more about AdTech, refer to [2].

Frame the Problem as an ML Task

Defining the ML objective

The goal of the ad click prediction system is to increase revenue by showing users ads they are more likely to click on. This can be converted into the following ML objective: predicting if an ad will be clicked. This is due to the fact that by correctly predicting click probabilities, the system can display relevant ads to users, which leads to an increase in revenue.

Specifying the system's input and output

The ad click prediction system takes a user as input, and outputs a ranked list of ads based on click probabilities.

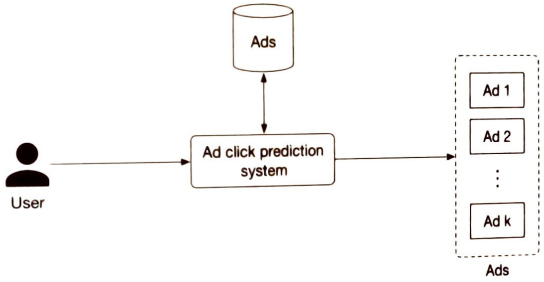


Figure 8.2: Ad click prediction system's input and output

Choosing the right ML category

Figure 8.2 illustrates how ad prediction can be framed as a ranking problem. As described in Chapter 7, Event Recommendation System, the pointwise Learning to Rank (LTR) is a great starting point for solving ranking problems. The pointwise LTR employs a binary classification model that takes a ⟨user, ad⟩ pair as input and predicts whether the user will click on the ad. Figure 8.3 shows the model's input and output.

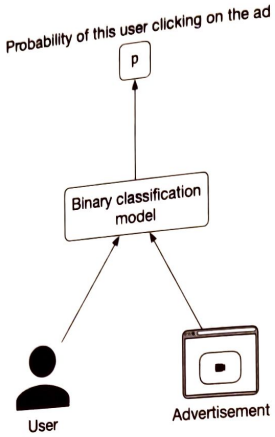


Figure 8.3: Binary classification model's input-output

Data Preparation

Data engineering

Here is some raw data available in this system:

- Ads
- Users
- User-ad interactions

Ads

Ad data is shown in Table 8.1. In practice, we may have hundreds of attributes associated with each ad. For simplicity, we only list important ones.

Ad ID	Advertiser ID	Ad group ID	Campaign ID	Category	Subcategory	Images or Videos
1	1	4	7	travel	hotel	http://cdn.mysite.com/u1.jpg
2	7	2	9	insurance	car	http://cdn.mysite.com/t3.mp4
3	9	6	28	travel	airline	http://cdn.mysite.com/t5.jpg

Table 8.1: Ad data

Users

The schema for user data is shown below.

ID	Username	Age	Gender	City	Country	Language	Time zone

Table 8.2: Schema for user data

User-ad interactions

This table stores user-ad interactions such as impressions, clicks, and conversions.

User ID	Ad ID	Interaction type	Dwell time[1]	Location (lat, long)	Timestamp
11	6	Impression	5 sec	38.8951 -77.0364	165845053
11	7	Impression	0.4 sec	41.9241 -89.0389	1658451365
4	20	Click	-	22.7531 47.9642	1658435948
11	6	Conversion	-	22.7531 47.9642	1658451849

Table 8.3: User-ad interaction data

Feature engineering

Our aim in this section is to engineer features that will assist us in predicting user clicks.

Ad features

Ad features include the following:

- IDs
- Image/video
- Category and subcategory
- Impression and click numbers

Let's examine each in more detail.

IDs

These are advertiser ID, campaign ID, ad group ID, ad ID, etc.

Why is it important? The IDs represent the advertiser, the campaign, the ad group, and the ad itself. These IDs are used as predictive features to capture the unique characteristics of different advertisers, campaigns, ad groups, and ads.

How to prepare it? The embedding layer converts sparse features, such as IDs, into dense feature vectors. Each ID type has its own embedding layer.

[1]Dwell time refers to the total time an ad is present on a user's screen

Image/video

Why is it important? A video or image in a post is another signal that can help us predict what the ad is about. For example, an image of an airplane may indicate the ad is related to travel.

How to prepare it? The images or videos are first preprocessed. After that, we use a pre-trained model such as SimCLR [3] to convert unstructured data into a feature vector.

Ad category and subcategory

As provided by the advertiser, this is the ad's category and subcategory. For example, here is a list of broad types of categories that are targetable: Arts & Entertainment, Autos & Vehicles, Beauty & Fitness, etc.

Why is it important? It helps the model to understand which category the ad belongs to.

How to prepare it? These are manually provided by the advertiser based on a predefined list of categories and subcategories. To learn more about preparing textual data, read Chapter 4, YouTube Video Search.

Impressions and click numbers

- Total impression/clicks on the ad
- Total impressions/clicks on ads supplied by an advertiser
- Total impressions of the campaign

Why is it important? These numbers indicate how other users reacted to this ad. For example, users are more likely to click on an ad with a high click-through rate (CTR).

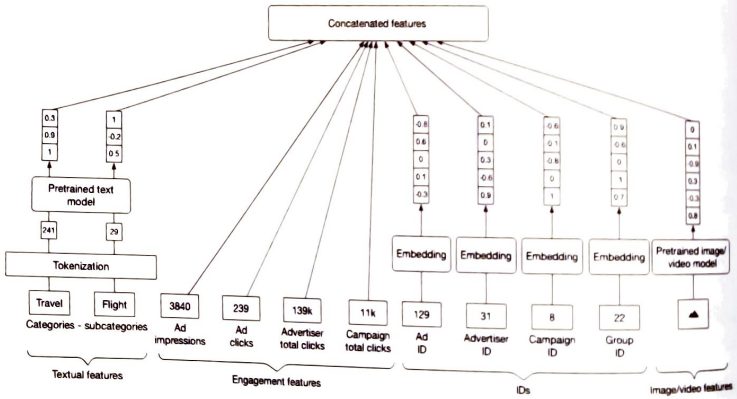


Figure 8.4: Summary of ad-related feature preparation

User features

Similar to previous chapters, we choose the following features:

- Demographics: age, gender, city, county, etc
- Contextual information: device, time of the day, etc
- Interaction-related features: clicked ads, user's historical engagement statistics, etc

Let's take a closer look at interaction-related features.

Clicked ads

Ads previously clicked by the user.

Why is it important? Previous clicks indicate a user's interests. For example, when a user clicks on lots of insurance-related ads, it suggests they are likely to click on a similar ad again.

How to prepare it? In the same way as described in "Ad features".

User's historical engagement statistics

These are the user's historical engagement numbers, such as their total ad views and ad click rate.

Why is it important? An individual's historical engagement is a good predictor of future engagement. In general, users are more likely to click on ads in the future, if they clicked on ads frequently in the past.

How to prepare it? Engagement statistics are represented as numerical values. To prepare them, we scale their values into a similar range.

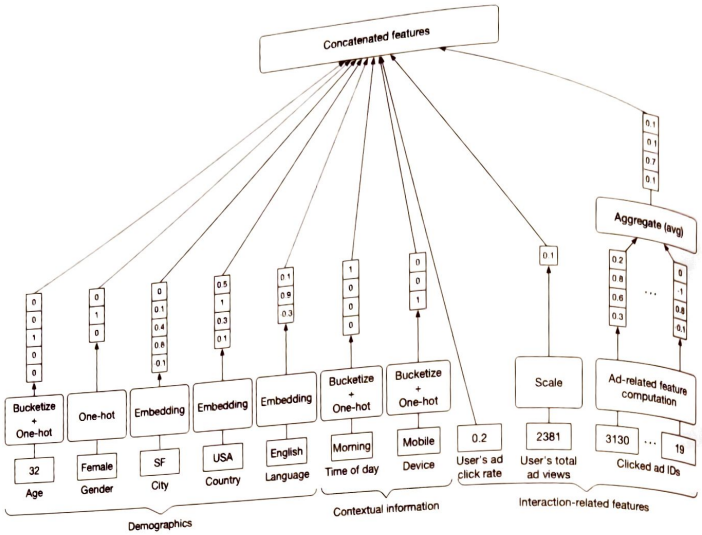


Figure 8.5: Feature preparation for user metadata and interactions

Before concluding the data preparation section, let's examine a common challenge in ad click prediction systems. In most cases, these systems deal with lots of high cardinality categorical features. For example, "ad category" takes values from a huge list of all possible categories. Similarly, "advertiser ID" and "user ID" take potentially millions of unique values, depending on how many users or advertisers are active on the platform. Given the huge feature space which often exists, it is common to have thousands or millions of features mostly filled with zeroes. In the model selection section, we will cover techniques to overcome these unique challenges.

Model Development

Model selection

As described in the section "Frame the Problem as an ML Task", the binary classification model is chosen to solve the ranking problem. Binary classifications can be modeled in several different ways. The following are common choices in ad click prediction systems:

- Logistic regression
- Feature crossing + logistic regression
- Gradient boosted decision trees
- Gradient boosted decision trees + logistic regression
- Neural networks
- Deep & Cross networks
- Factorization Machines
- Deep Factorization Machines

Logistic regression (LR)

LR models the probability of a binary outcome using a linear combination of one or multiple features. LR is fast to train and easy to implement. An LR-based ad click prediction system, however, does have the following drawbacks:

- **Non-linear problems can't be solved with LR.** LR solves the task using a linear combination of input features, which leads to a linear decision boundary. In ad click prediction systems, data is usually not linearly separable, so LR may perform poorly.
- **Inability to capture feature interactions.** LR is not capable of capturing feature interactions. In ad prediction systems, it is very common to have various interactions between features. When features interact with each other, the output probability cannot be expressed as the sum of the feature effects, since the effect of one feature depends on the value of the other feature.

Given these two drawbacks, LR is not the best choice for the ad prediction system. However, because it is fast to implement and easy to train, many companies use it to create a baseline model.

Feature crossing + LR

To capture feature interactions better, we use a technique called feature crossing.

What is feature crossing?

Feature crossing is a technique used in ML to create new features from existing features. It involves combining two or more existing features into one new feature by taking their product, sum, or another combination. It is possible to capture nonlinear interactions between the original features in this way, which can improve the performance of ML models. For example, interactions such as "young and basketball" or "USA and football" may positively impact a model's ability to predict click probability.

How to create feature crosses?

In feature crossing, we manually add new features to the existing features based on prior knowledge. As Figure 8.6 shows, crossing two features, such as "country" and "language" adds six new features to the existing feature space. To learn more about crossing, refer to [4].

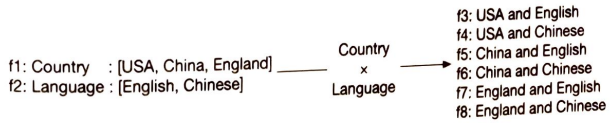


Figure 8.6: Crossing two features: country and language

How to use feature crossing + LR?

As shown in Figure 8.7, feature crossing + LR works as follows:

1. Use feature crossing on the original set of features to extract new features (crossed features)
2. Use the original and the crossed features as input for the LR model

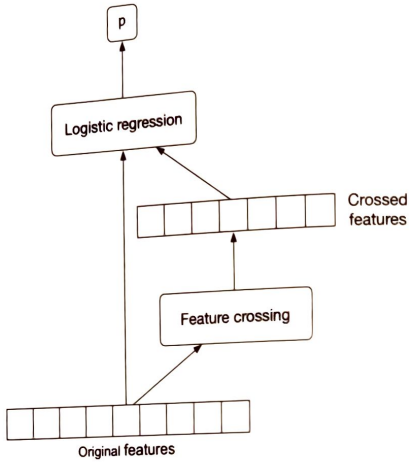


Figure 8.7: Feature crossing performed on original features

This method allows the model to capture certain pairwise (second-order) feature interactions. However, it has three shortcomings:

- **Manual process:** Human involvement is required to choose features to be crossed, which is time-consuming and expensive.
- **Requires domain knowledge:** Feature crossing requires domain expertise. To determine which interactions between features are predictive signals for the model, we need to understand the problem and the feature space in advance.
- **Cannot capture complex interactions:** Crossed features may not be sufficient to capture all the complex interactions from thousands of sparse features.
- **Sparsity:** The original features can be sparse. With feature crossing, the cardinality of the crossed features can become much larger, leading to more sparsity.

Given the drawbacks, this method is not an ideal solution for the ad prediction system.

Gradient-boosted decision trees (GBDT)

We examined GBDT in Chapter 7, Event Recommendation System. Here, we will only explore the pros and cons of GBDT when applied to the ad click prediction system.

Pros

- GBDT is interpretable and easy to understand

Cons

- **Inefficient for continual learning.** In ad click prediction systems, we continuously collect new data such as user, ad, and interaction data. To continually train a model on new data, we generally have two options: 1) training from scratch, or 2) fine-tuning the model on new data. GBDT is not designed to be fine-tuned with new data. So we usually need to train the model from scratch, which is inefficient at a large scale.
- **Cannot train embedding layers.** In ad prediction systems, it's common to have many sparse categorical features, and the embedding layer is an effective way to represent these features. However, GBDT cannot benefit from embedding layers.

GBDT + LR

There are two steps in this approach:

1. Train the GBDT model to learn the task.
2. Instead of using the trained model to make predictions, use it to select and extract new predictive features. The newly generated features and the original features are used as input into the LR model for predicting clicks.

Use GBDT for feature selection

Feature selection is intended to reduce the number of input features to only those most useful and informative. Using decision trees, we can select a subset of features based on their importance. To better understand how decision trees are used for feature generation, refer to [5].

Use GBDT for feature extraction

The purpose of feature extraction is to reduce the number of features by creating new features from existing ones. The newly extracted features are expected to have better predictive power. Figure 8.8 explains how to extract features using GBDT.

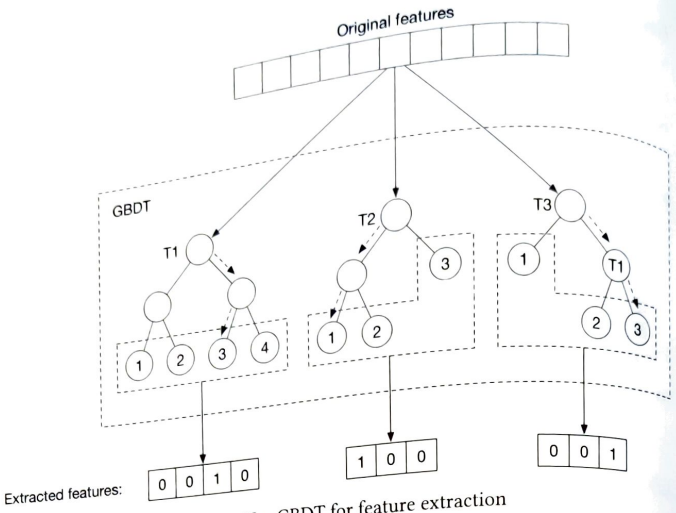


Figure 8.8: Use GBDT for feature extraction

An overview of GBDT in use followed by LR, is shown in Figure 8.9.

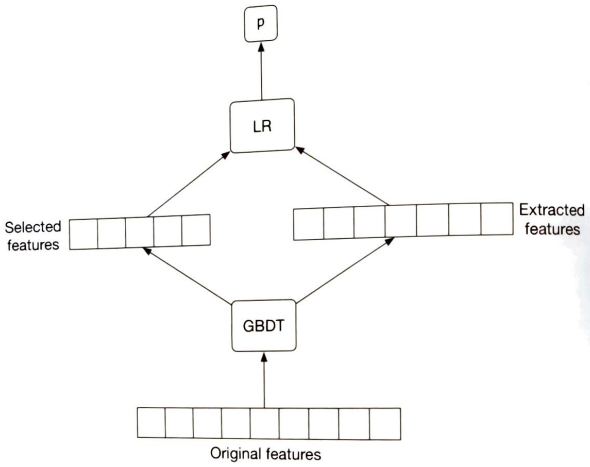


Figure 8.9: GBDT + LR overview

Let's explore the pros and cons of this approach.

Pros:

- In contrast to the existing features, the newly created ones produced by GBDT are more predictive, making it easier for the LR model to learn the task.

Cons:

- **Cannot capture complex interactions.** Similar to LR, this approach cannot learn pairwise feature interactions.
- **Continual learning is slow.** Fine-tuning GBDT models on new data takes time, which slows down continual learning overall.

Neural network (NN)

NN is another candidate for building the ad click prediction system. To predict click probabilities using a NN, we have two architectural options:

- Single NN
- Two-tower architecture

Single NN: Using the original features as input, a neural network outputs the click probability (Figure 8.10).

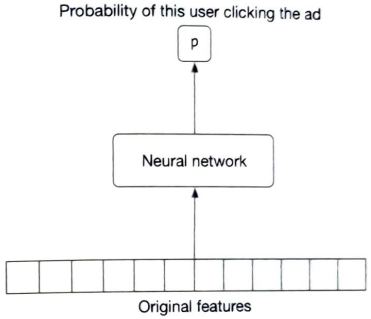


Figure 8.10: NN architecture

Two-tower architecture: In this option, we use two encoders: user encoder and ad encoder. The similarity between ad and user embeddings is used to determine the relevance, that is, the click probability. Figure 8.11 shows an overview of this architecture.

to understand it better:

$$\hat{y}(x) = w_0 + \sum_i w_i x_i + \sum_i \sum_j \langle v_i, v_j \rangle x_i x_j$$

Where x_i refers to the i-th feature, w_i is the learned weight, and v_i represents the embedding of the i-th feature. $\langle v_i, v_j \rangle$ denotes the dot product between two embeddings.

This formula may look complex, but it's actually easy to understand. The first two terms compute a linear combination of the features, similar to how logistic regression works. The third term models pairwise feature interactions. Figure 8.13 shows a high-level overview of FM. Refer to [9] to learn the details of FM.

$$\hat{y}(x) = \underbrace{w_0 + \sum_i w_i x_i}_{\text{Logistic regression}} + \underbrace{\sum_i \sum_j \langle v_i, v_j \rangle x_i x_j}_{\text{Pairwise interactions}}$$

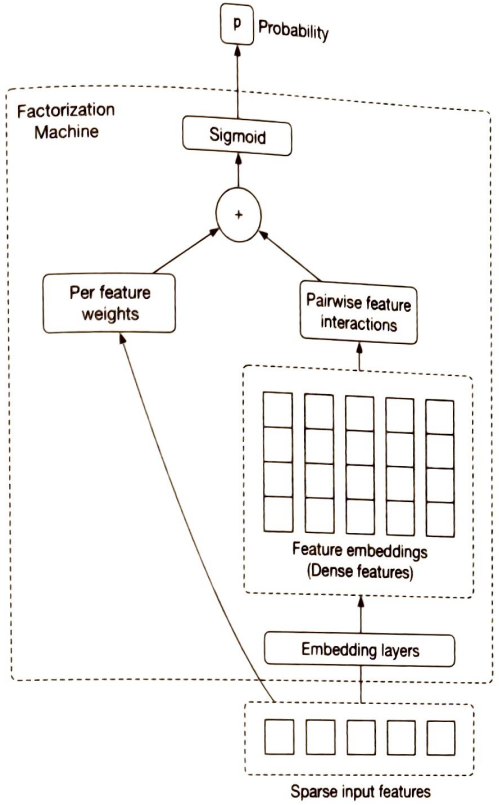


Figure 8.13: Factorization Machine architecture

FM and its variants, such as FFM, effectively capture pairwise interactions between features. FM cannot learn sophisticated higher-order interactions from features, unlike neural networks, which can. In the next method, we combine FM and DNN to overcome this.

Deep Factorization Machines (DeepFM)

DeepFM is an ML model that combines the strengths of both NN and FM. A DNN network captures sophisticated higher-order features, and an FM captures low-level pairwise feature interactions. Figure 8.14 shows the high-level architecture of DeepFM. If you are interested to learn more about DeepFM, refer to [10].

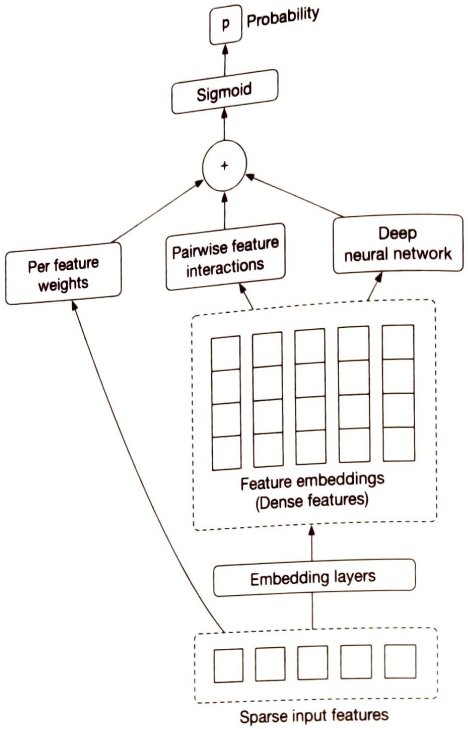


Figure 8.14: DeepFM overview

One potential improvement is to combine GBDT and DeepFM. A GBDT converts the original features into more predictive features, while DeepFM operates on the new features. This method has won various ad prediction system competitions [11]. However, adding GBDT to DeepFM negatively affects the training and inference speed, and slows down the continual learning process.

In practice, we usually choose the correct model by running experiments. In our case, we start with a simple LR to create a baseline. Next, we experiment with DCN and DeepFM, as both are widely used in the tech industry.

Model training

Constructing the dataset

For every ad impression, we construct a new data point. The input features are computed from the user and the ad. A label is assigned to the data point, based on the following strategy:

- **Positive label:** if the user clicks the ad in less than t seconds after the ad is shown, we label the data point as "positive". Note that t is a hyperparameter and can be tuned via experimentation.
- **Negative label:** if the user does not click the ad in less than t seconds, we label the data point as "negative".

In practice, companies use more complex methods to find the optimal strategy for labeling negative data points. To learn more, refer to [1].

#	User and interaction features	Ad features	Label
1	1 0 1 0.8 0.1 1 0	0 1 1 0.4 0.9 0	Positive
2	1 1 0 -0.6 0.9 1 1	1 1 0 0.2 0.7 1	Negative

Figure 8.15: Constructed dataset

To keep the model adaptive to new data, it must continuously be trained. As a result, new training data points should be continuously generated using new interactions. We will discuss continual learning further in the serving section.

Choosing the loss function

Since we are training a binary classification model, we choose cross-entropy as a classification loss function.

Evaluation

Offline metrics

Two metrics are typically used to evaluate an ad click prediction system:

- Cross-entropy (CE)
- Normalized cross-entropy (NCE)

CE. This metric measures how close the model's predicted probabilities are to the ground truth labels. CE is zero if we have an ideal system that predicts a 0 for the negative classes and 1 for the positive classes. The lower the CE, the higher the accuracy of the prediction. The formula is:

$$H(p,q) = -\sum_{c=1}^{C} p_c \log q_c$$

where p is the ground truth, q is the predicted probability, and C is the total number of classes.

For binary classification, the CE formula can be rewritten as:

$$H(p,q) = -\sum_i p_i \log q_i = -\sum_i \Big(y_i \log \hat{y}_i + (1-y_i) \log (1-\hat{y}_i) \Big)$$

where y_i is the ground truth label of the i-th data point and $\hat{y}_i$ is the predicted probability of i-th data point.

Let's take a look at a concrete example, as shown in Figure 8.16.

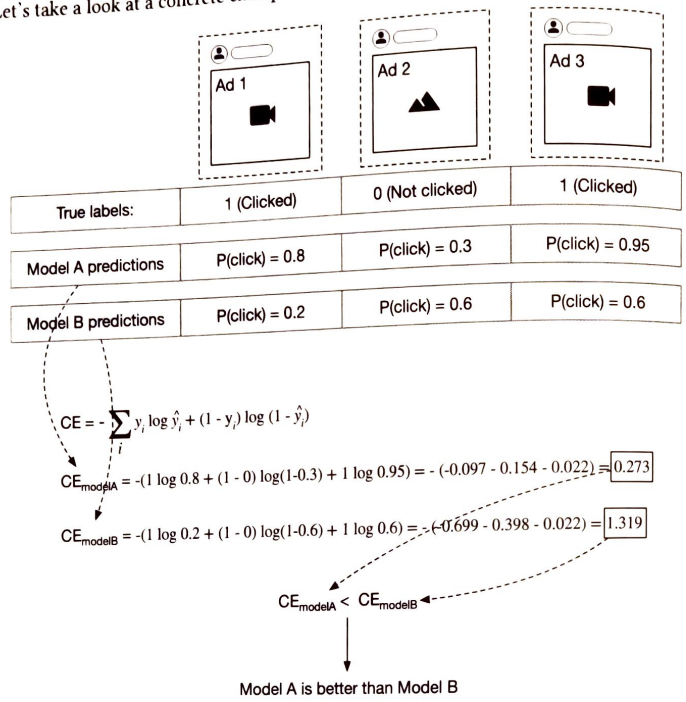


Figure 8.16: CE of two ML models

Note that aside from using CE as a metric, it is also commonly used as a standard loss function in classification tasks during model training.

Normalized cross-entropy (NCE). NCE is the ratio between our model's CE and the CE of the background CTR (average CTR in the training data). In other words, NCE compares the model with a simple baseline which always predicts the background CTR. A low NCE indicates the model outperforms the simple baseline. $NCE \geq 1$ indicates that the model is not performing better than the simple baseline.

$$\text{Normalized cross entropy} = \frac{CE(\text{ ML model })}{CE(\text{ Simple baseline })}$$

Let's take a look at a concrete example to understand better how NCE is calculated. As shown in Figure 8.17, a simple baseline model always predicts 0.6 (CTR in the training data). In this case, the NCE value is 0.324 (less than 1), indicating model A outperforms the simple baseline.

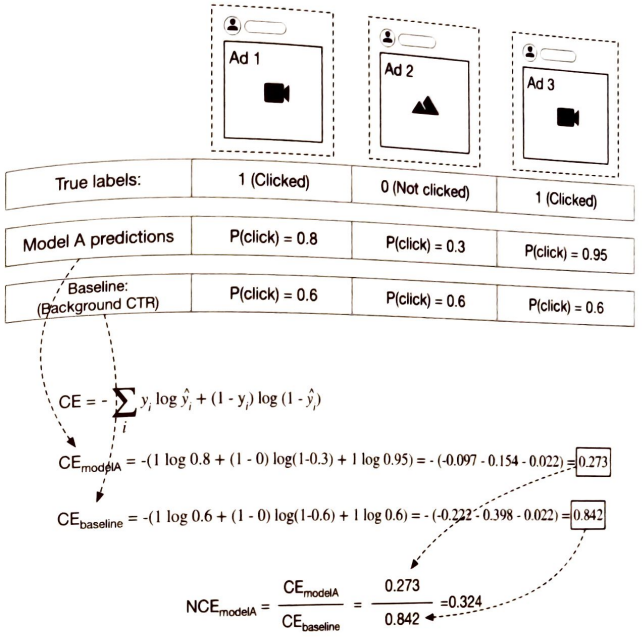


Figure 8.17: NCE calculations for model A

Online metrics

Let's examine some metrics we may use during online evaluation.

- CTR
- Conversion rate
- Revenue lift
- Hide rate

CTR. This metric measures the ratio between clicked ads and the total number of shown

ads.

$$CTR = \frac{\text{Number of clicked ads}}{\text{Number of shown ads}}$$

CTR is a great online metric for ad click prediction systems, as maximizing user clicks on ads is directly related to an increase in revenue.

Conversion rate. This metric measures the ratio between the number of conversions and the total number of ads shown.

$$\text{Conversion rate} = \frac{\text{Number of conversions}}{\text{Number of impressions}}$$

This metric is important to track as it indicates how many times advertisers actually benefited from the system. This matters because advertisers will eventually lose interest and cease spending on ads if their ads do not lead to conversions.

Revenue lift. This measures the percentage of revenue increase over time.

Hide rate. This metric measures the ratio between the number of ads hidden by users and the number of shown ads.

$$\text{Hide rate} = \frac{\text{Number of ads hidden by users}}{\text{Number of shown ads}}$$

This metric is helpful for understanding how many irrelevant ads the system displayed to users, also known as false positives.

Serving

At serving time, the system is responsible for outputting a list of ads ranked by their click probabilities. The proposed ML system design is shown in Figure 8.18. Let's examine each of the following pipelines:

- Data preparation pipeline
- Continual learning pipeline
- Prediction pipeline

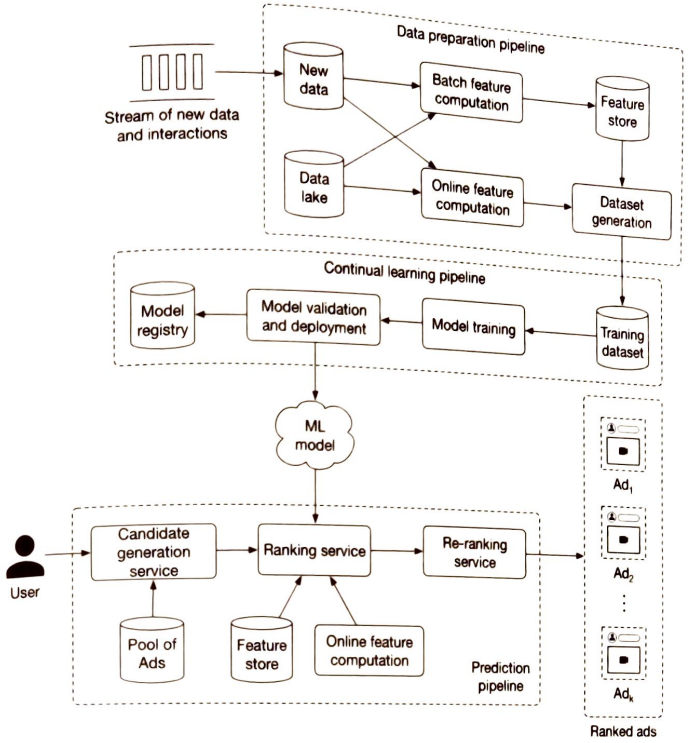


Figure 8.18: ML system design

Data preparation pipeline

The data preparation pipeline performs the following two tasks:

1. Compute online and batch features
2. Continuously generate training data from new ads and interactions

To compute features, the following two options are used: batch feature computation and online feature computation. Let's see how they differ from each other.

Batch feature computation

Some of the features we chose are static, which means they change very rarely. For example, an ad's image and category are static features. This component computes static features periodically (e.g., every few days or weeks) with batch jobs and then stores the features in a feature store. This improves the system's performance during serving because the features are precomputed.

Online feature computation
Some features are dynamic as they change frequently. For example, the numbers of ad impressions and clicks are examples of dynamic features. These features need to be computed at query time, and this component is used to compute dynamic features.

Continual learning pipeline

Based on the requirements, continually learning the model is critical. This pipeline is responsible for fine-tuning the model on new training data, evaluating the new model, and deploying the model if it improves the metrics. It ensures the prediction pipeline always uses a model adapted to the most recent data.

Prediction pipeline

The prediction pipeline takes a query user as input and outputs a list of ads ranked by their click probabilities. Since some of the features which the model relies upon are dynamic, we cannot use batch prediction. Instead, requests are served as they arrive using online prediction.

As we've seen in previous chapters, a two-stage architecture is used in the prediction pipeline. First, we employ a candidate generation service to efficiently narrow down the available pool of ads to a small subset of ads. In this case, we use the ad targeting criteria often provided by advertisers, such as target age, gender, and country.

Next, we employ a ranking model which fetches the candidate ads from the candidate generation service, ranks them based on click probability, and outputs the top ads. This component interacts with the same feature store and online feature computation component. Once the static and dynamic features are obtained, the ranking service uses the model to get a predicted click probability for each candidate ad. These probabilities are used to rank the ads and to output those with the highest click probability.

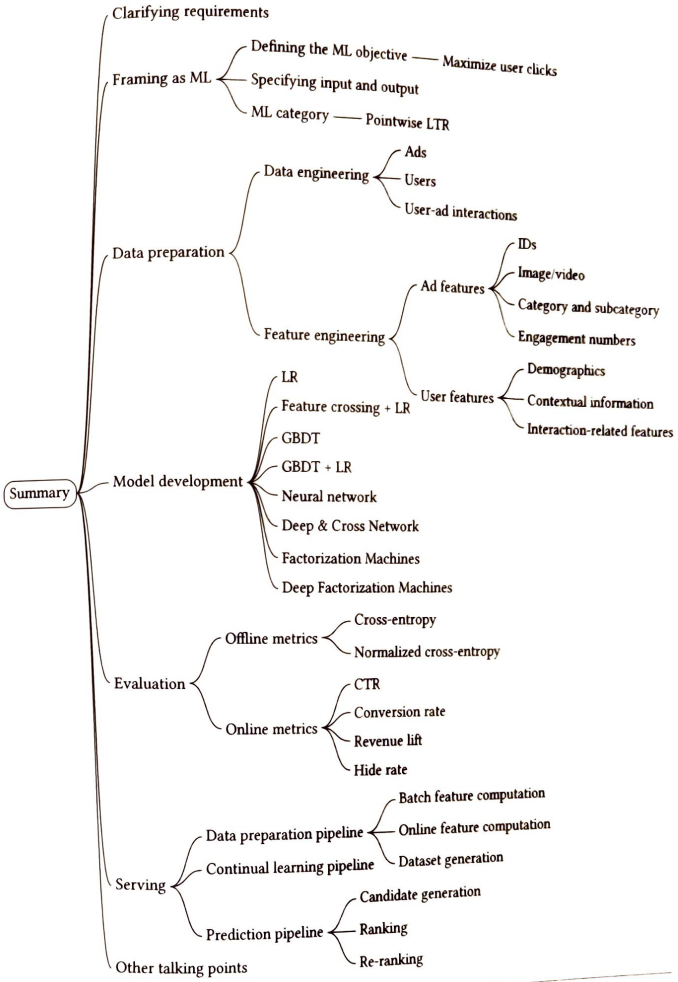
Finally, a re-ranking service modifies the list of ads by incorporating additional logic and heuristics. For example, we can increase the diversity of ads by removing very similar ads from the list.

Other Talking Points

If there is time left at the end of the interview, here are some potential talking points you might discuss with the interviewer:

- In ranking and recommendation systems, it's important to avoid data leakage [12][13].
- The model needs to be calibrated in ad click prediction systems. Discuss model calibration and techniques for calibrating a model [14].
- A common variant of FM is a field-aware Factorization Machine (FFM). It's good to talk about FFM and how it differs from FM [15].
- A common variant of DeepFM is XDeepFM. Talk about XDeepFM and how it differs from DeepFM [10].
- We've described why continuous learning is necessary for ad click prediction systems. However, continual learning on new data may lead to catastrophic forgetting. Discuss what catastrophic forgetting is and what common solutions are [16].

Summary



Reference Material

[1] Addressing delayed feedback. https://arxiv.org/pdf/1907.06558.pdf.

[2] AdTech basics. https://advertising.amazon.com/library/guides/what-is-adtech.

[3] SimCLR paper. https://arxiv.org/pdf/2002.05709.pdf.

[4] Feature crossing. https://developers.google.com/machine-learning/crash-course/feature-crosses/video-lecture.

[5] Feature extraction with GBDT. https://towardsdatascience.com/feature-generation-with-gradient-boosted-decision-trees-21d4946d6ab5.

[6] DCN paper. https://arxiv.org/pdf/1708.05123.pdf.

[7] DCN V2 paper. https://arxiv.org/pdf/2008.13535.pdf.

[8] Microsoft's deep crossing network paper. https://www.kdd.org/kdd2016/papers/files/adf0975-shanA.pdf.

[9] Factorization Machines. https://www.jefkine.com/recsys/2017/03/27/factorization-machines/.

[10] Deep Factorization Machines. https://d2l.ai/chapter_recommender-systems/deepfm.html.

[11] Kaggle's winning solution in ad click prediction. https://www.youtube.com/watch?v=4Go5crRVyuU.

[12] Data leakage in ML systems. https://machinelearningmastery.com/data-leakage-machine-learning/.

[13] Time-based dataset splitting. https://www.linkedin.com/pulse/time-based-splitting-determining-train-test-data-come-manraj-chalokia/?trk=public_profile_article_view.

[14] Model calibration. https://machinelearningmastery.com/calibrated-classification-model-in-scikit-learn/.

[15] Field-aware Factorization Machines. https://www.csie.ntu.edu.tw/~cjlin/papers/ffm.pdf.

[16] Catastrophic forgetting problem in continual learning. https://www.cs.uic.edu/~liub/lifelong-learning/continual-learning.pdf.

9 Similar Listings on Vacation Rental Platforms

Recommending items similar to those a user is currently viewing, is a key technology that allows people to discover potentially relevant content on large platforms. For example, Airbnb recommends similar accommodation listings, Amazon recommends similar products, and Expedia recommends similar experiences to users.

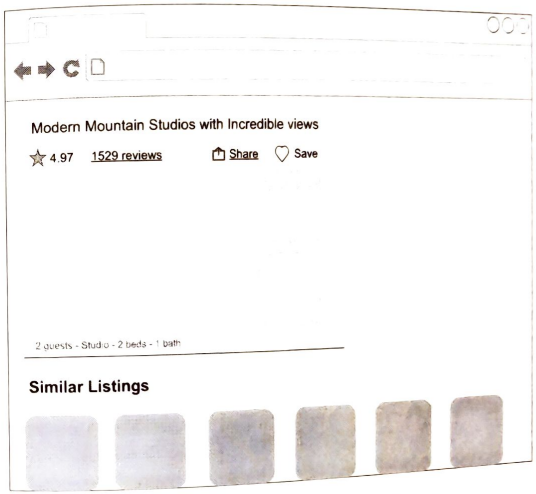


Figure 9.1: Recommended similar listings

In this chapter, we design a "similar listings" feature which resembles those used by vacation rental websites such as Airbnb and Vrbo. When a user clicks a specific listing, a list of similar listings is recommended to them.

Clarifying Requirements

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

Candidate: Can I assume the business objective is to increase the number of bookings?
Interviewer: Yes.

Candidate: What is the definition of "similarity"? Are the recommended listings expected to be similar to the listing that a user is currently viewing?
Interviewer: Yes, that's correct. Two listings are defined as similar when they are in the same neighborhood, city, price range, etc.

Candidate: Are the recommended listings personalized to users?
Interviewer: We want this feature to work for both logged-in and anonymous users. In practice, we treat the two groups differently and apply personalization to logged-in users. However, for simplicity, let's assume we treat logged-in and anonymous users equally.

Candidate: How many listings are available on the platform?
Interviewer: 5 million listings.

Candidate: How do we construct the training dataset?
Interviewer: Good question. For this interview, let's assume we use user-listing interactions only. The model doesn't leverage users' attributes, such as age or location, or a listing's attributes, like price and location, at all.

Candidate: How long does it take for new listings to appear in the similar listings result?
Interviewer: Let's assume new listings are okay to appear as recommendations one day after being posted. During this time, the system collects interaction data for new listings.

Let's summarize the problem statement. We are asked to design a "similar listings" feature for vacation rental platforms. The input is a specific listing that a user is currently viewing, and the output is a ranked list of similar listings the user is likely to click on next. The recommended listings should be the same for both anonymous and logged-in users. There are around 5 million listings on the platform, and new listings can appear in recommendations after one day. The business objective of the system is to increase the number of bookings.

Frame the Problem as an ML Task

Defining the ML objective

The sequence of listings that a user clicks on usually have similar characteristics, such as being in the same city or having similar price ranges. We rely on this observation to define the ML objective as accurately predicting which listing the user will click next, given the listing the user is currently viewing.

Specifying the system's input and output

As shown in Figure 9.2, the "similar listings" system takes a listing a user is currently viewing as input and then outputs a ranked list of listings, sorted by the probability of this user clicking on them.

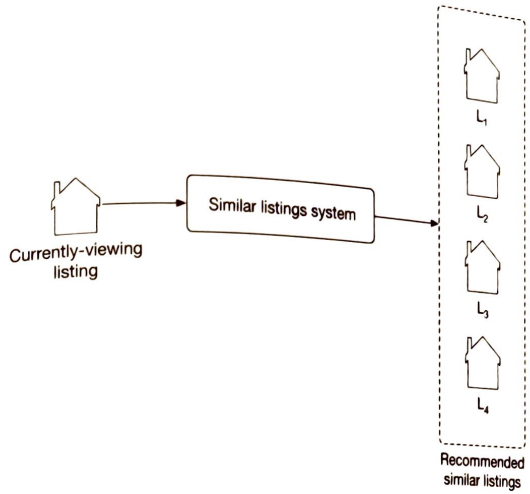


Figure 9.2: A similar listings system's input-output

Choosing the right ML category

Most recommendation systems rely on users' historical interactions to understand their long-term interests. However, such recommendation systems may not be good at solving the similar listings problem. In our situation, recently viewed listings are more informative than those viewed a long time ago. In this case, a session-based recommendation system is commonly used.

Like Airbnb, many e-commerce and travel booking platforms rely more on short-term interests to make recommendations. In systems where high-quality recommendations depend more on recent interactions than long-term interests, a session-based recommendation is often a substitute for traditional recommendation systems. A session-based recommendation makes recommendations based on the user's current browsing session. Now, let's take a closer look at session-based recommendation systems.

Session-based recommendation systems

A session-based recommendation system aims to predict the next item, given a sequence of recent items browsed by a user. In the system, users' interests are context-dependent and evolve fast. A good recommendation heavily depends on the user's most recent

interactions, not their generic interests.

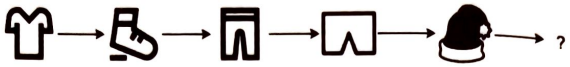


Figure 9.3: A browsing session of products

How do session-based and traditional recommendation systems compare? In traditional recommendation systems, users' interests are context-independent and won't change too frequently. In session-based recommendations, users' interests are dynamic and evolve fast. The goal of a traditional recommendation system is to learn users' generic interests. In contrast, session-based recommendation systems aim to understand users' short-term interests, based on their recent browsing history.

A widely-used technique to build session-based recommendation systems is to learn item embeddings using co-occurrences of items in users' browsing histories. For example, Instagram learns account embeddings to power its "Explore" feature [1], Airbnb learns listing embeddings to power its similar listings feature [2], and word2vec [3] uses a similar approach to learn meaningful word embeddings.

In this chapter, we frame the "similar listings" problem as a session-based recommendation task. We build the system by training a model which maps each listing into an embedding vector, so that if two listings frequently co-occur in users' browsing history, their embedding vectors are in close proximity in the embedding space.

To recommend similar listings, we search the embedding space for listings closest to the one currently being viewed. Let's take a look at an example of this. In Figure 9.4, each listing is mapped into a 2D space. To recommend similar listings to L_t , we choose the top 3 listings with the closest embeddings.

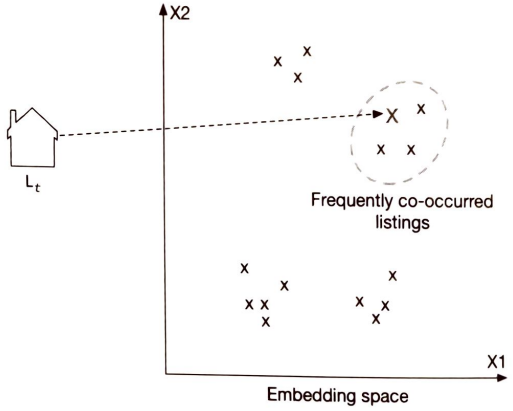


Figure 9.4: Similar listings in the embedding space

Data Preparation

Data engineering

The following data are available:

- Users
- Listings
- User-listing interactions

Users

A simplified user data schema is shown below.

ID	Username	Age	Gender	City	Country	Language	Time zone

Table 9.1: User data schema

Listings

Listing data contains attributes related to each listing, such as price, number of beds, host ID, etc. Table 9.2 shows a simple example of what the listing data might look like.

ID	Host ID	Price	Sq ft	Rate	Type	City	Beds	Max guests
1	135	135	1060	4.97	Entire place	NYC	3	4
2	81	80	830	4.6	Private room	SF	1	2
3	64	65	2540	5.0	Shared room	Boston	4	6

Table 9.2: Listing data

User-listing interactions

Table 9.3 stores user-listing interactions such as impressions, clicks, and bookings.

ID	User ID	Listing ID	Position of the listing in the displayed list	Interaction type	Source	Timestamp
2	18	26	2	Click	Search feature	1655121925
3	5	18	5	Book	Similar listing feature	1655135257

Table 9.3: User-listing interaction data

Feature engineering

As described in the "Frame the Problem as ML Task" section, the model only utilizes users' browsing history during training. Other information is not used, such as listing price, user's age, etc.

In this chapter, the browsing histories are called "search sessions". A search session is a sequence of clicked listing IDs, followed by an eventually booked listing, without interruption . Figure 9.5 shows an example of a search session, where the user's session started when the user clicked on L_1, and ended when the user eventually booked L_{20}.

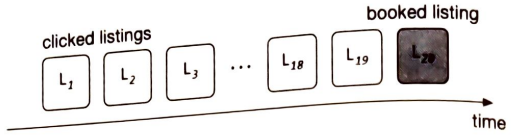


Figure 9.5: A search session

In the feature engineering step, we extract search sessions from the interaction data. Table 9.4 shows a simple example of the search sessions.

Session ID	Clicked listing IDs	Eventually booked listing ID
1	1, 5, 4, 9	26
2	6, 8, 9, 21, 6, 13, 6	5
3	5, 9	11

Table 9.4: Search session data

Model Development

Model selection

A neural network is the standard method to learn embeddings. Choosing a good architecture for the neural network depends upon various factors, such as the complexity of the task, the amount of training data, etc. One common way to choose the hyperparameters associated with neural network architectures – such as the number of neurons, layers, activation function, etc. – is to run experiments and choose the architecture that performs best. In our case, we choose a shallow neural network architecture to learn listing embeddings.

Model training

As shown in Figure 9.6, for a given input listing, the model's job is to predict listings within the input listing's context.

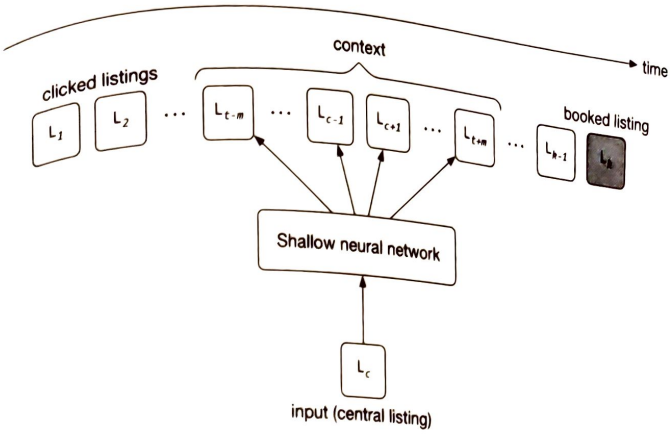


Figure 9.6: Predicting neighboring listings

The training process starts by initializing listing embeddings to random vectors. These embeddings are learned gradually by reading through search sessions, using the sliding window method. As the window slides, the embedding of the central listing in the window is updated to be similar to the embedding of other listings in the window, and dissimilar from listings outside the window. The model then uses these embeddings to predict the context of a given listing.

To adapt the model to new listings, we train it daily on the newly constructed training data.

Constructing the dataset

There are different ways to construct a dataset. In our case, we choose a technique called "negative sampling" [4], commonly used to learn embeddings.

To construct the training data, we create positive pairs and negative pairs from search sessions. Positive pairs are listings that are expected to have similar embeddings, while negative pairs are expected to have dissimilar embeddings.

More precisely, for each session, we read through the listings with the sliding window method. As the window slides, we use the central listing in the window and its context listings to create positive pairs. We use the central listing and randomly sampled listings to form negative pairs. Positive pairs have a ground truth label of 1, and negative pairs are given a label of 0.

Figure 9.7 shows how positive and negative pairs are generated by sliding through a search session.

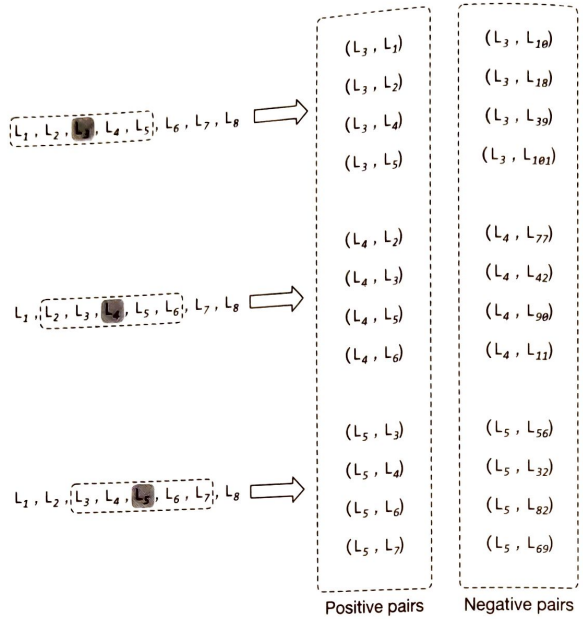


Figure 9.7: Constructed positive and negative listing pairs

Choosing the loss function

Loss function measures the agreement between the ground truth label and the predicted probability. If two listings form a positive pair, the embeddings should be close, and if the two listings form a negative pair, the embeddings should be far apart. More formally, here are the steps to calculate loss:

1. Compute the distance (e.g., dot product) between two embeddings.
2. Use the Sigmoid function to convert the computed distance to a probability value between 0 and 1.
3. Use cross-entropy as a standard classification loss to measure the loss between the predicted probability and the ground truth label.

Figure 9.8 shows the loss calculation steps.

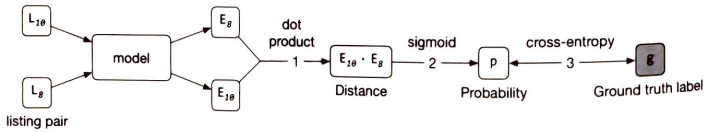


Figure 9.8: Loss calculation steps

The loss can be represented by the following formula:

$$\text{loss} = \sum_{(c,p)\in D_p} \log \frac{1}{1+e^{-E_p \cdot E_c}} + \sum_{(c,n)\in D_n} \log \frac{1}{1+e^{E_n \cdot E_c}}$$

Where:

- c is a central listing, p is a positive listing (co-occurred with c in a context), and n is a negative listing (did not co-occur with c)
- E_c represents the embedding vector of the central listing c
- E_n represents the embedding vector of negative listing n
- E_p represents the embedding vector of positive listing p
- D_p is a positive set of pairs $\langle c, p \rangle$ which represents (central listing, context listing) tuples whose vectors are being pushed toward one another
- D_n is a negative set of pairs $\langle c, n \rangle$ which represents (central listing, random listing) tuples whose vectors are being pushed away from each other

The first summation computes the loss over positive pairs and the second summation computes the loss over negative pairs.

Can we improve the loss function to learn better embeddings?

The loss function described earlier is a good starting point, but it has two shortcomings. First, during training, the embedding of the central listing is pushed closer to the embeddings in its context, but not towards the embedding of the eventually booked listing. This leads to embeddings that are good at predicting neighboring clicked listings, but not at predicting eventually booked listings. This is not optimal for helping users discover a listing that leads to a booking.

Second, the negative pairs generated earlier mainly comprise listings from different regions, since they are sampled randomly. However, users typically search only within a certain region, e.g., San Francisco. This may lead to embeddings that do not work well for same-region listings; those that have not co-occurred in context, but are from the same region.

Let's address these shortcomings.

Using the eventually booked listing as a global context

To learn embeddings that are good at predicting eventually booked listings, we treat the eventually booked listing as a global context during the training phase. As the window slides, some listings fall in or out of the context set, while the eventually booked listing always remains in the global context, and is used to update the central listing vector.

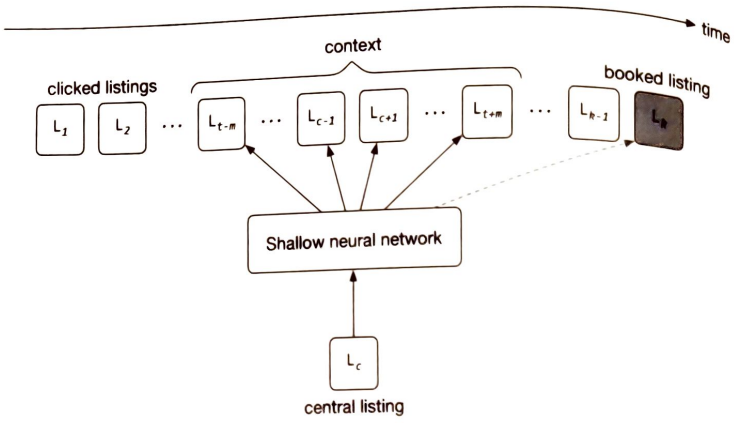


Figure 9.9: Adding the eventually booked listing to the positive pairs

To use the eventually booked listing as a global context during training, we add pairs of ⟨central listing, eventually booked listing⟩ to our training data and label them as positive. This drives the model to push the embedding of the eventually booked listing close to each of the clicked listings in the session during training, as shown in Figure 9.9.

Add negative pairs from the same region to the training data

As the window slides, we choose a listing from the same neighborhood as the central listing, which is not within the central listing's context. We label the pair as negative and add it to our training data.

Let's see the updated loss function that considers newly added training data.

$$\text{loss} = \sum_{(c,p)\in D_p} \log \frac{1}{1+e^{-E_c E_p}} + \sum_{(c,n)\in D_n} \log \frac{1}{1+e^{E_c E_n}} + \sum_{(c,b)\in D_{\text{booked}}} \log \frac{1}{1+e^{-E_c E_b}} + \sum_{(c,n)\in D_{\text{hard}}} \log \frac{1}{1+e^{E_c E_n}}$$

Where:

- E_b represents the embedding vector of the eventually booked listing b
- D_{booked} are pairs of $\langle c, b \rangle$ that represent (central listing, booked listing) tuples whose vectors are being pushed close to each other
- D_{hard} are hard negative pairs $\langle c, n \rangle$ that represent (central listing, same-region negative listing) tuples whose vectors are being pushed away from each other

We explained the first two summations earlier. The third summation computes the loss over newly added positive pairs which contain the global context. It helps the model to push the central listings' embeddings close to eventually booked listings' embeddings.

The fourth summation computes the loss over newly added negative pairs from the same region. It enforces the model to push their embeddings away from each other.

Evaluation

Offline metrics

During the model development phase, we use offline metrics to measure the output quality of the model and compare the newly developed models with older ones. One way to evaluate learned embeddings is to test how good they are at predicting the eventually-booked listing, based on the latest user click. Let's create a metric called "average rank of eventually booked listing" and discuss this in more detail.

The average rank of the eventually-booked listing. Let's look at an example to understand this metric. Figure 9.10 shows a user's search session. As you can see, the search session consists of seven listings in total. The first listing is what the user viewed first (L_0). The next five are listings the user clicked on, sequentially. The last one (L_6) is the listing that the user eventually booked.

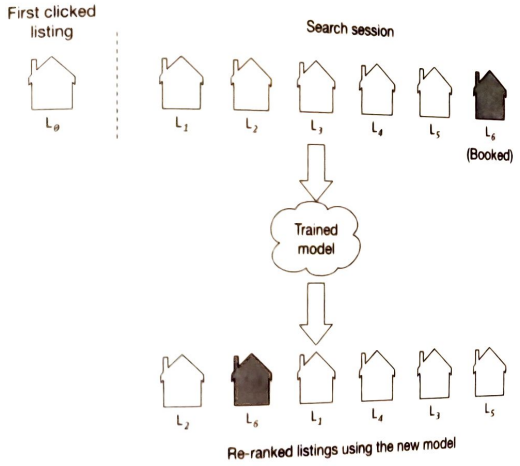


Figure 9.10: A session re-ranked by a model

We use the model to compute the similarities between the first clicked listing and other listings in the embedding space. Once similarities are computed, the listings are ranked. The position of the eventually booked listing indicates how high in the ranking we could have recommended it (L_6) by employing the new model. As you can see in Figure 9.10, the new model (second row) was able to rank the eventually booked listing (L_6) in second place.

If the model ranks the eventually booked listing highly, it indicates the learned embeddings can place the eventually booked listing earlier in the recommended list. We average the rank of the eventually booked listings across all the sessions in the validation dataset, to compute the value of this metric.

Online metrics

According to the requirements, the business objective is to increase the number of bookings. Here are some options for online metrics:

- Click-through rate (CTR)
- Session book rate

CTR. A ratio showing how often people who see the recommended listings, end up clicking them.

$$CTR = \frac{\text{Number of clicked listings}}{\text{Number of recommended listings}}$$

This metric is used to measure user engagement. For example, when users click on listings more frequently, there is a higher likelihood that some of the clicked listings become a booking. But since CTR does not measure the actual number of bookings made on the platform, we use the "session book rate" metric to supplement CTR.

Session book rate. A ratio showing how many search sessions turn into a booking.

$$\text{Session book Rate} = \frac{\text{Number of sessions turned into booking}}{\text{Total number of sessions}}$$

This metric is directly related to our business objective, which is to increase the number of bookings. The higher the "session book rate" is, the more revenue the platform generates.

Serving

At serving time, the system recommends listings similar to that the user is currently viewing. Figure 9.11 shows an overview of the ML system design.

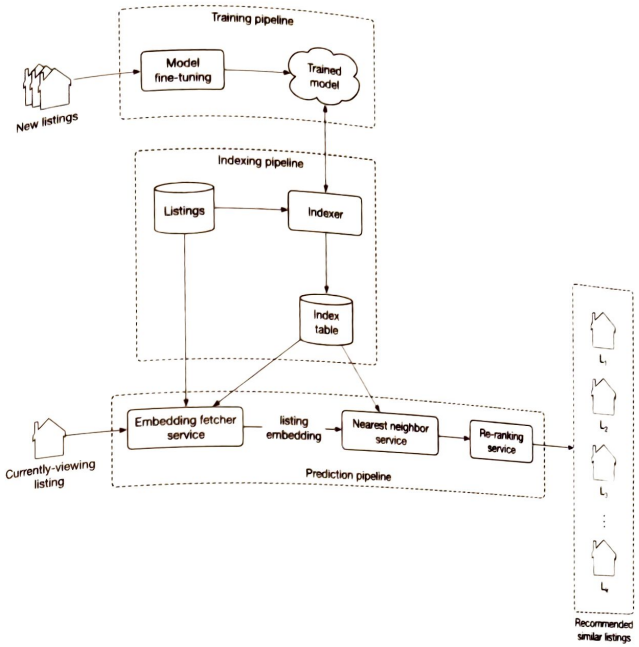


Figure 9.11: ML system design

Let's examine the main components in detail.

Training pipeline

The training pipeline fine-tunes the model using new listings and user-listing interactions. This ensures the model is always adapted to new interactions and listings.

Indexing pipeline

With a trained model, the embeddings of all listings on the platform can be pre-computed and stored in the index table. This significantly speeds up the prediction pipeline.

The indexing pipeline creates and maintains the index table. For example, when a new listing embedding becomes available, the pipeline adds its embedding to the index table. In addition, when a newly trained model becomes available, the pipeline re-computes all the embeddings using the new model and updates the index table.

Prediction pipeline

The prediction pipeline recommends similar listings to what a user is currently viewing. The prediction pipeline, as shown in Figure 9.11, consists of:

- Embedding fetcher service

- Nearest neighbor service
- Re-ranking service

Let's inspect each component.

Embedding fetcher service

This service takes the currently viewing listing as input and acts differently depending on whether or not the listing has been seen by the model during training.

The input listing has been seen by the model during training

If a listing was seen during training, its embedding vector has already been learned and is available in the index table. In this case, the embedding fetcher service directly fetches the listing embedding from the index table.

The input listing has not been seen by the model during training

If the input listing is new, the model hasn't seen it during training. This is problematic since we cannot find similar listings if we do not have the embedding of the given listing.

To solve this issue, the embedding fetcher uses heuristics to handle new listings. For example, we can use the embedding of a geographically nearby listing when the listing is new. When enough interaction data is gathered for the new listing, the training pipeline learns the embedding by fine-tuning the model.

Nearest neighbor service

To recommend similar listings, we need to compute the similarity between the embedding of the currently-viewing listing and the embeddings of other listings on the platform. This is where the nearest neighbor service comes into play. This service computes these similarities and outputs the nearest neighbor listings in the embedding space.

Remember from the requirements that we have five million listings on the platform. Computing similarities for this many listings takes time and may slow down serving. Therefore, we use an approximate nearest neighbor method to speed up the search.

Re-ranking service

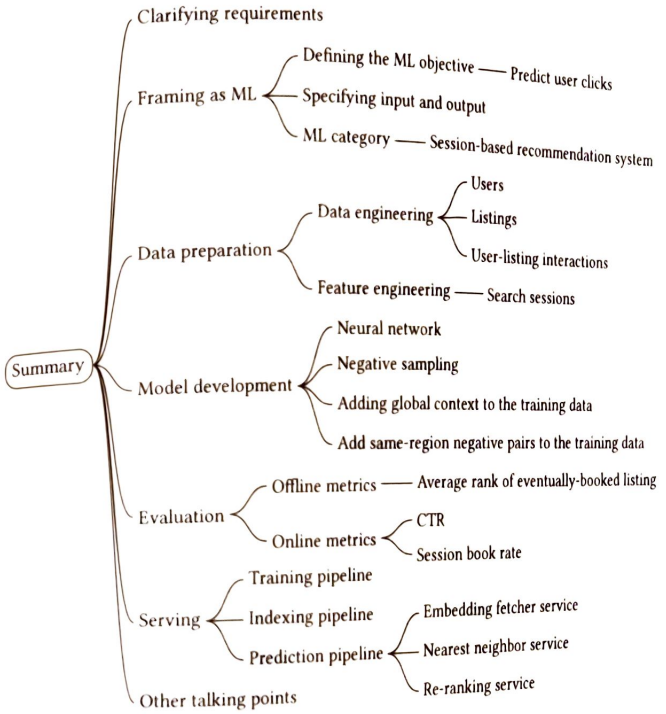
This service modifies the listings by applying user filters and certain constraints. For example, if a listing is above a certain price filter set by the user, this layer removes it. In addition, listings in cities other than the currently viewed listing can be removed from the list before being displayed to the user.

Other Talking Points

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

- What is positional bias, and how to address it [5].
- How does a session-based approach compare to random walk [6], and how random walks with restart (RWR) can be used to recommend similar listings [7].
- How to personalize the results of a session-based recommendation system by considering users' longer-term interests (in-session personalization) [2].
- Given that seasonality greatly affects vacation rentals, how should we incorporate seasonality into our similar listings system [8].

Summary



Reference Material

[1] Instagram's Explore recommender system. https://ai.facebook.com/blog/powered-by-ai-instagrams-explore-recommender-system.

[2] Listing embeddings in search ranking. https://medium.com/airbnb-engineering/listing-embeddings-for-similar-listing-recommendations-and-real-time-personalization-in-search-601172f7603e.

[3] Word2vec. https://en.wikipedia.org/wiki/Word2vec.

[4] Negative sampling technique. https://www.baeldung.com/cs/nlps-word2vec-negative-sampling.

[5] Positional bias. https://eugeneyan.com/writing/position-bias/.

[6] Random walk. https://en.wikipedia.org/wiki/Random_walk.

[7] Random walk with restarts. https://www.youtube.com/watch?v=HbzQzUaJ_9I.

[8] Seasonality in recommendation systems. https://www.computer.org/csdl/proceedings-article/big-data/2019/09005954/1hJsfgT0qL6.

10 Personalized News Feed

Introduction

A news feed is a feature of social network platforms that keeps users engaged by showing friends' recent activities on their timelines. Most social networks such as Facebook [1], Twitter [2], and LinkedIn [3] personalize news feed to maintain user engagement.

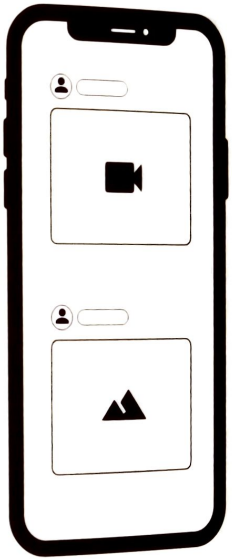


Figure 10.1: User timeline with personalized feeds

In this chapter, we are asked to design a personalized news feed system.

Clarifying Requirements

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

Candidate: Can I assume the motivation for a personalized news feed is to keep users engaged with the platform?
Interviewer: Yes, we display sponsored ads between posts, and more engagement leads to increased revenue.

Candidate: When a user refreshes their timeline, we display posts with new activities to the user. Can I assume this activity consists of both unseen posts and posts with unseen comments?
Interviewer: That is a fair assumption.

Candidate: Can a post contain textual content, images, video, or any combination?
Interviewer: It can be any combination.

Candidate: To keep users engaged, the system should place the most engaging content at the top of timelines, as people are more likely to interact with the first few posts. Does that sound right?
Interviewer: Yes, that's correct.

Candidate: Is there a specific type of engagement we are optimizing for? I assume there are different types of engagement, such as clicks, likes, and shares.
Interviewer: Great question. Different reactions have different values on our platform. For example, liking a post is more valuable than only clicking it. Ideally, our system should consider major reactions when ranking posts. With that, I'll leave you to define "engagement" and choose what your model should optimize for.

Candidate: What are the major reactions available on the platform? I assume users can click, like, share, comment, hide, block another user, and send connection requests. Are there other reactions we should consider?
Interviewer: You mentioned the major ones. Let's keep our focus on those.

Candidate: How fast is the system supposed to work?
Interviewer: We expect the system to display the ranked posts quickly after users refresh their timelines or open the application. If it takes too long, users will get bored and leave. Let's assume the system should display the ranked posts in less than 200 milliseconds (ms).

Candidate: How many daily active users do we have? How many timeline updates do we expect each day?
Interviewer: We have almost 3 billion users in total. Around 2 billion are daily active users who check their feeds twice a day.

Let's summarize the problem statement. We are asked to design a personalized news feed system. The system retrieves unseen posts or posts with unseen comments, and ranks them based on how engaging they are to the user. This should take no longer than 200ms. The objective of the system is to increase user engagement.

Frame the problem as an ML task

Defining the ML objective

Let's examine the following three possible ML objectives:

- Maximize the number of specific implicit reactions, such as dwell time or user clicks
- Maximize the number of specific explicit reactions, such as likes or shares
- Maximize a weighted score based on both implicit and explicit reactions

Let's discuss each option in more detail.

Option 1: Maximize the number of specific implicit reactions, such as dwell time or user clicks
In this option, we choose implicit signals as a proxy for user engagement. For example, we optimize the ML system to maximize user clicks.

The advantage is that we have more data about implicit reactions than explicit ones. More training data usually leads to more accurate models.

The disadvantage is that implicit reactions do not always reflect a user's true opinion about a post. For example, a user may click on a post, but find it is not worth reading.

Option 2: Maximize the number of specific explicit reactions, such as likes, shares, and hides
With this option, we choose explicit reactions as a proxy for user opinions about a post.

The advantage of this approach is that explicit signals usually carry more weight than implicit signals. For example, a user liking a post sends a stronger engagement signal than if they simply click it.

The main disadvantage is that very few users actually express their opinions with explicit reactions. For example, a user may find a post engaging but not react to it. In this scenario, it's hard for the model to make an accurate prediction given the limited training data.

Option 3: Maximize a weighted score based on both implicit and explicit reactions
In this option, we use both implicit and explicit reactions to determine how engaged a user is with a post. In particular, we assign a weight to each reaction, based on how valuable the reaction is to us. We then optimize the ML system to maximize the weighted score of reactions.

Table 10.1 shows the mapping between different reactions and weights. As you can see, pressing the "like" button has more weight than a click, while a share is more valuable than a like. In addition, negative reactions such as hide and block have a negative weight. Note that these weights can be chosen based on business needs.

Reaction	Click	Like	Comment	Share	Friendship request	Hide	Block
Weight	1	5	10	20	30	-20	-50

Table 10.1: Weights of different reactions

Which option to choose?
We choose the final blended option because it allows us to assign different weights to different reactions. This is important because we can optimize the system based on what's important to the business.

Specifying the system's input and output

As Figure 10.2 shows, the personalized news feed system takes a user as input and outputs a ranked list of unseen posts or posts with unseen comments sorted by engagement score.

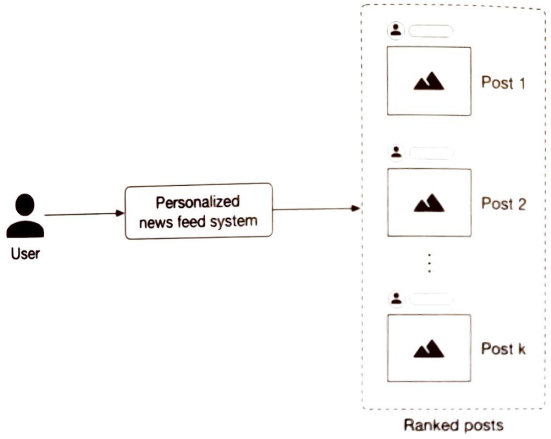


Figure 10.2: A personalized news feed system's input-output

Choosing the right ML category

A personalized news feed system produces a ranked list of posts based on how engaging the posts are to a user. Pointwise Learning to Rank (LTR) is a simple yet effective approach that personalizes news feeds by ranking posts based on engagement scores. To understand how to compute engagement scores between users and posts, let's examine a concrete example.

As Figure 10.3 shows, we employ several binary classifiers to predict the probabilities of various implicit and explicit reactions for a ⟨user, post⟩ pair.

Reaction	Click	Like	Comment	Share	Friendship request	Hide	Block
Predicted probability	23%	48%	12%	4%	0.1%	0.005%	0.0003%

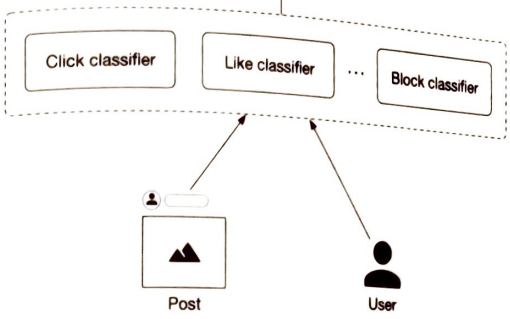


Figure 10.3: Predicted probabilities of various reactions

Once these probabilities are predicted, we compute the engagement score. Figure 10.4 shows an example of how the engagement score is calculated.

Reaction	Click	Like	Comment	Share	Friendship request	Hide	Block
Predicted probability	23%	48%	12%	4%	0.1%	0.005%	0.0003%
Value	1	5	10	20	30	−20	−50
Score	0.23	2.4	1.2	0.8	0.03	−0.001	−0.00015
Engagement score = 4.65885							

Figure 10.4: Calculate the engagement score

Data Preparation

Data engineering

It is generally helpful to understand what raw data is available before going on to engineer predictive features. Here, we assume the following types of raw data are available:

- Users
- Posts
- User-post interactions
- Friendship

Users

The user data schema is shown below.

ID	Username	Age	Gender	City	Country	Language	Time zone

Table 10.2: User data schema

Posts

Table 10.3 shows post data.

Author ID	Textual Content	Hashtags	Mentions	Images or videos	Timestamp
5	Today at our fav place with my best friend	life_is_good, happy	hs2008	-	1658450539
1	It was the best trip we ever had	Travel, Maldives	Alexish, shan.tony	http://cdn.mysite.com/maldives.jpg	1658451341
29	Today I had a bad experience I would like to tell you about. I went...	-	-	-	1658451365

Table 10.3: Post data

User-post interactions

Table 10.4 shows user-post interaction data.

User ID	Post ID	Interaction type	Interaction value	Location (lat, long)	Timestamp
4	18	Like	-	38.8951 -77.0364	1658450539
4	18	Share	User 9	41.9241 -89.0389	1658451365
9	18	Comment	You look amazing	22.7531 47.9642	1658435948
9	18	Block	-	22.7531 47.9642	1658451849
6	9	Impression		37.5189 122.6405	1658821820

Table 10.4: User-post interaction data

Friendship

The friendship table stores data of connections between users. We assume users can specify their close friends and family members. Table 10.5 shows examples of friendship data.

User ID 1	User ID 2	Time when friendship was formed	Close friend	Family member
28	3	1558451341	True	False
7	39	1559281720	False	False
11	25	1559312942	False	True
			False	False

Table 10.5: Friendship data

Feature engineering

In this section, we engineer predictive features and prepare them for the model. In particular, we engineer features from each of the following categories:

- Post features
- User features
- User-author affinities

Post features

In practice, each post has many attributes. We cannot cover everything, so only discuss the most important ones.

- Textual content
- Images or videos
- Reactions
- Hashtags
- Post's age

Textual content

What is it? This is the textual content – the main body – of a post.

Why is it important? Textual content helps determine what the post is about.

How to prepare it? We preprocess textual content and use a pre-trained language model to convert the text into a numerical vector. Since the textual content is usually in the form of sentences and not a single word, we use a context-aware language model such as BERT [4].

Images or videos

What is it? A post may contain images or videos.

Why is it important? We can extract important signals from images. For example, an image of a gun may indicate that a post is unsafe for children.

How to prepare it? First, preprocess the images or videos. Next, use a pre-trained model to convert the unstructured image/video data into an embedding vector. For example, we can use ResNet, [5] or the recently introduced CLIP model [6] as the pre-trained model.

Reactions

What is it? This refers to the number of likes, shares, replies, etc., of a post.

Why is it important? The number of likes, shares, hides, etc., indicates how engaging users find a post to be. A user is more likely to engage with a post with thousands of likes than a post with ten likes.

How to prepare it? These values are represented by numerical values. We scale these numerical values to bring them into a similar range.

Hashtags

Why is it important? Users use hashtags to group content around a certain topic. These hashtags represent the topics to which a post relates. For example, a post with the hashtag "#women_in_tech" indicates the content relates to technology and females, so the model may decide to rank it higher for people who are interested in technology.

How to prepare it? The detailed steps to preprocess text are already explained in Chapter 4, YouTube Video Search, so we will only focus on the unique steps for preparing hashtags.

- **Tokenization:** Hashtags like "lifeisgood" or "programmer_lifestyle" contain multiple words. We use algorithms such as Viterbi [7] to tokenize the hashtags. For instance, "lifeisgood" becomes 3 words: "life" "is" "good".
- **Tokens to IDs:** Hashtags evolve quickly on social media platforms and change as trends come and go. A feature hashing technique is a good fit because it is capable of assigning indexes to unseen hashtags.
- **Vectorization:** We use simple text representation methods such as TF-IDF [8] or word2vec [9], instead of Transformer-based models, to vectorize hashtags. Let's take a look at why. Transformer-based models are useful when the context of the data is essential. In the case of hashtags, each one is usually a single word or a phrase, and often no context is necessary to understand what the hashtag means. Therefore, faster and lighter text representation methods are preferred.

Post's age

What is it? This feature shows how much time has passed since the author posted the content.

Why is it important? Users tend to engage with newer content.

How to prepare it? We bucketize the post's age into a few categories and use one-hot encoding to represent it. For example, we use the following buckets:

- 0: less than 1 hour
- 1: between 1 to 5 hours
- 2: between 5 to 24 hours
- 3: between 1-7 days

- 4: between 7-30 days
- 5: more than a month

Figure 10.5 summarizes all post-related features.

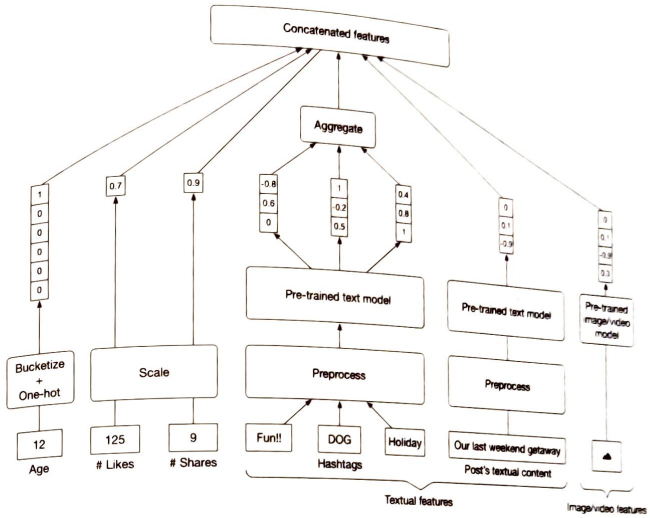


Figure 10.5: Post-related feature preparation

User features

Some of the most important user-related features are:

- Demographics: age, gender, country, etc
- Contextual information: device, time of the day, etc
- User-post historical interactions
- Being mentioned in the post

Since we have already discussed user demographic and contextual information in previous chapters, here we only examine the remaining two features.

User-post historical interactions

All posts liked by a user are represented by a list of post IDs. The same logic applies to shares and comments.

Why is it important? Users' previous engagements are usually helpful in determining their future engagements.

How to prepare it?
Extract features from each post that the user interacted with.

Being mentioned in a post

What is it? This means whether or not the user is mentioned in a post.

Why is it important? Users usually pay more attention to posts that mention them.

How to prepare it? This feature is represented by a binary value. If a user is mentioned in the post, this feature is 1, otherwise 0.

Figure 10.6 summarizes feature preparation for users.

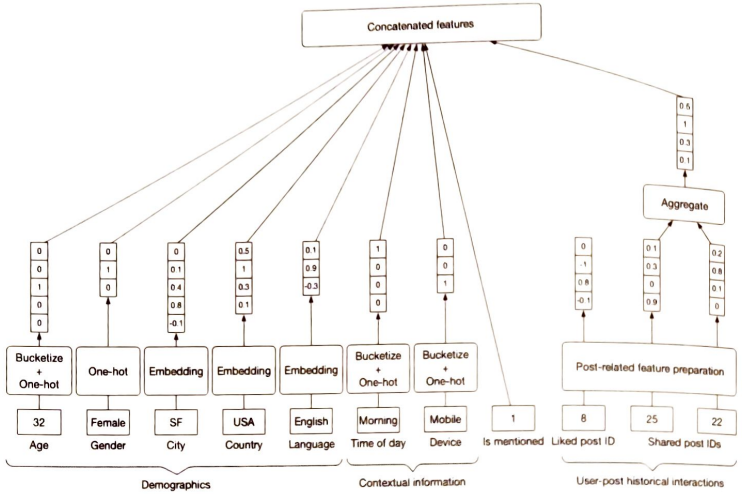


Figure 10.6: Feature preparation for user-related data

User-author affinities

According to studies, affinity features, such as the connection between the user and the author, are among the most important factors in predicting a user's engagement on Facebook [10]. Let's engineer some features to capture user-author affinities.

Like/click/comment/share rate

This is the rate at which a user reacted to previous posts by an author. For example, a like rate of 0.95 indicates that a user liked the posts from that author 95 percent of the time.

Length of friendship

The number of days the user and the author have been friends on the platform. This feature can be obtained from the friendship data.

Why is it important? Users tend to engage more with their friends.

Close friends and family

A binary value representing whether the user and the author have included each other in their close friends and family list.

Why is it important? Users pay more attention to posts by close friends and family members. Figure 10.7 summarizes features related to user-author affinities.

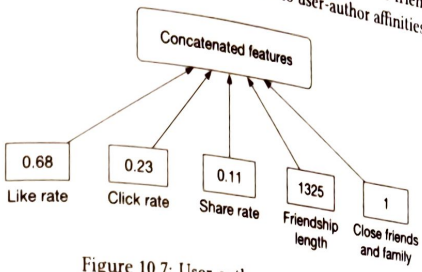


Figure 10.7: User-author affinity features

Model Development

Model selection

We choose neural networks for the following reasons:

- Neural networks work well with unstructured data, such as text and images.
- Neural networks allow us to use embedding layers to represent categorical features.
- With a neural network architecture, we can fine-tune pre-trained models employed during feature engineering. This is not possible with other models.

Before training a neural network, we need to choose its architecture. There are two architectural options for building and training our neural networks:

- N independent DNNs
- A multi-task DNN

Let's explore each one.

Option 1: N independent DNNs
In this option, we use N independent deep neural networks (DNN), one for each reaction. This is shown in Figure 10.8.

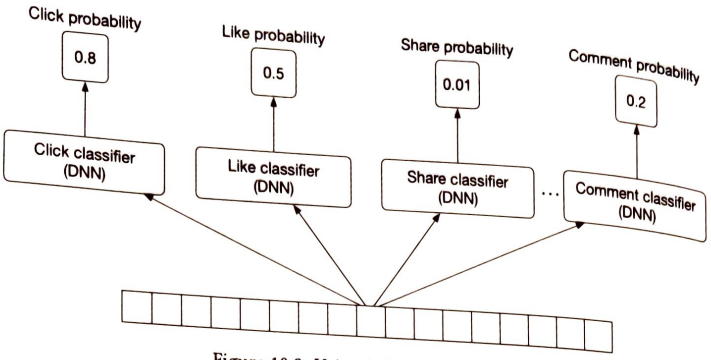


Figure 10.8: Using independent DNNs

This option has two drawbacks:

- **Expensive to train.** Training several independent DNNs is compute-intensive and time-consuming.
- **For less frequent reactions, there might not be enough training data.** This means our system is not able to predict the probabilities of infrequent reactions accurately.

Option 2: Multi-task DNN

To overcome these issues, we use a multi-task learning approach (Figure 10.9).

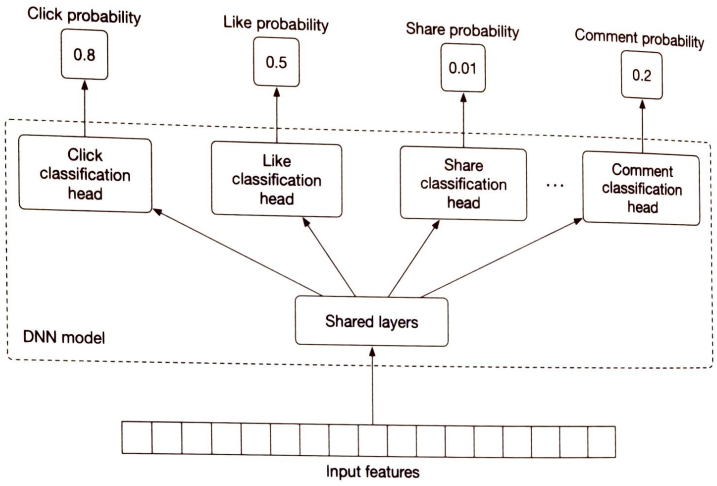


Figure 10.9: Multi-task DNN

We explained multi-task learning in Chapter 5, Harmful Content Detection, so we only briefly discuss it here. In summary, multi-task learning refers to the process of learning multiple tasks simultaneously. This allows the model to learn the similarities between tasks and avoid unnecessary computations. For a multi-task neural network model, it's essential to choose an appropriate architecture. The choice of architecture and the associated hyperparameters are usually determined by running experiments. That means training and evaluating the model on different architectures and choosing the one which leads to the best result.

Improving the DNN architecture for passive users

So far, we have employed a DNN to predict reactions such as shares, likes, clicks, and comments. However, many users use the platform passively, meaning they do not interact much with the content on their timelines. For such users, the current DNN model will predict very low probabilities for all reactions, since they rarely react to posts. Therefore, we need to change the DNN architecture to consider passive users.

For this to work, we add two implicit reactions to the list of tasks:

- Dwell-time: the time a user spends on a post.
- Skip: if a user spends less than t seconds (e.g., 0.5 seconds) on a post, then that post can be assumed to have been skipped by the user.

Figure 10.10 shows the multi-task DNN model with the additional tasks.

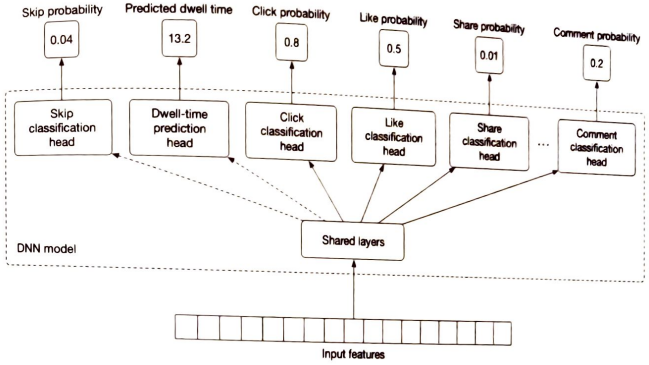


Figure 10.10: Multi-task DNN model with two new tasks

Model training

Constructing the dataset

In this step, we construct the dataset from raw data. Since the DNN model needs to learn multiple tasks, positive and negative data points are created for each (e.g., click, like, etc.).

We will use the reaction type of "like" as an example to explain how to create positive/negative data points. Each time a user likes a post, we add a data point to our dataset, compute ⟨user, post⟩ features, and then label it as positive.

To create negative data points, we chose impressions that didn't lead to a "like" reaction. Note that the number of negative data points is usually much higher than positive data points. To avoid having an imbalanced dataset, we create negative data points to equal the number of positive data points. Figure 10.11 shows positive and negative data points for the "like" reaction.

#	User features	Post features	Affinity features	Label
1	1 0 1 0.8 0.1 1	0 1 1 0.4 0	0.9 0.6 0.3 8 0	Positive
2	0 0 0 0.4 0.9 0	1 1 0 0.3 1	1 0.9 0.8 120 1	Positive
3	1 1 0 0.1 0.5 0	0 1 0 0.9 1	0.1 0 0 2 0	Negative

Figure 10.11: Training data for like classification task

This same process can be used to create positive and negative labels for other reactions. However, because dwell-time is a regression task, we construct it differently. As shown in Figure 10.12, the ground truth label is the dwell-time of the impression.

#	User features	Post features	Affinity features	Dwell-time
1	0 0 0 0.1 0.9 1	1 1 0 0.1 1	0.6 0.6 0.3 0.2 5 0	8.1
2	1 1 1 0.9 0.1 0	1 1 0 0.8 0	0.1 0.9 0.3 0.1 3 1	11.5

Figure 10.12: Training data for dwell-time task

Choosing the loss function

Multi-task models are trained to learn multiple tasks simultaneously. This means we need to compute the loss for each task separately and then combine them to get an overall loss. Typically, we define a loss function for each task depending on the ML category of the task. In our case, we use a binary cross-entropy loss for each binary classification task, and a regression loss such as MAE [11], MSE [12], or Huber loss [13] for the regression task (dwell-time prediction). The overall loss is computed by combining task-specific losses, as shown in Figure 10.13.

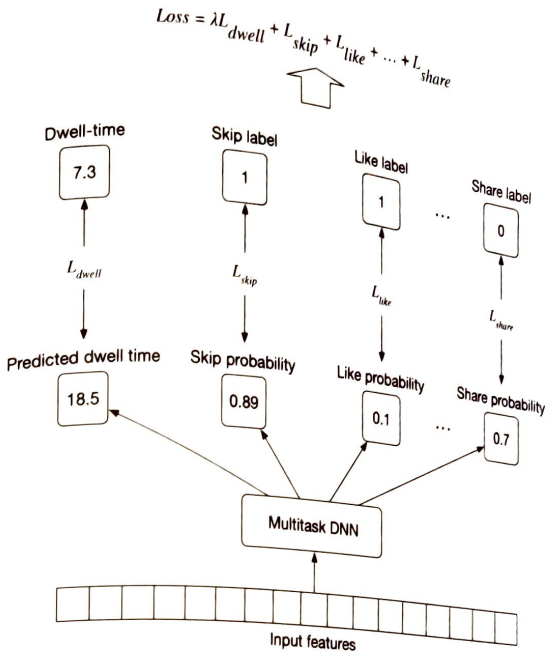


Figure 10.13: Training workflow

Evaluation

Offline metrics

During the offline evaluation, we measure the performance of our model in predicting different reactions. To evaluate the performance of an individual type of reaction, we can use binary classification metrics, such as precision and recall. However, these metrics alone may not be sufficient to understand the overall performance of a binary classification model. So, we use the ROC curve to understand the trade-off between the true positive rate and false positive rate. In addition, we compute the area under the ROC curve (ROC-AUC) to summarize the performance of the binary classification with a numerical value.

Online metrics

We use the following metrics to measure user engagement from various angles:

- Click-through rate (CTR)
- Reaction rate

- Total time spent
- User satisfaction rate found in a user survey

CTR. The ratio between the number of clicks and impressions.

$$CTR = \frac{\text{number of clicked posts}}{\text{number of impressions}}$$

A high CTR does not always indicate more user engagement. For example, users may click on a low-value clickbait post, and quickly realize it is not worth reading. Despite this limitation, it is an important metric to track.

Reaction rates. These are a set of metrics that reflect user reactions. For example, a like rate measures the ratio between posts liked and the total number of posts displayed in users' feeds.

$$\text{Like rate} = \frac{\text{number of liked posts}}{\text{number of impressions}}$$

Similarly, we track other reactions such as "share rate", "comment rate", "hide rate", "block rate", and "skip rate". These are stronger signals than CTR, as users have explicitly expressed a preference.

The metrics we discussed so far are based on user reactions. But what about passive users? These are users who tend not to react at all to the majority of posts. To capture the effectiveness of our personalized news feed system for passive users, we add the following two metrics.

Total time spent. This is the total time users spend on the timeline during a fixed period, such as 1 week. This metric measures the overall engagement of both passive and active users.

User satisfaction rate found in a user survey. Another way to measure the effectiveness of our personalized news feed system is to explicitly ask users for their opinion about the feed, or how engaging they find the posts. Since we seek explicit feedback, this is an accurate way to measure the system's effectiveness.

Serving

At serving time, the system serves requests by outputting a ranked list of posts. Figure 10.14 shows the architectural diagram of the personalized news feed system. The system comprises the following pipelines:

- Data preparation pipeline
- Prediction pipeline

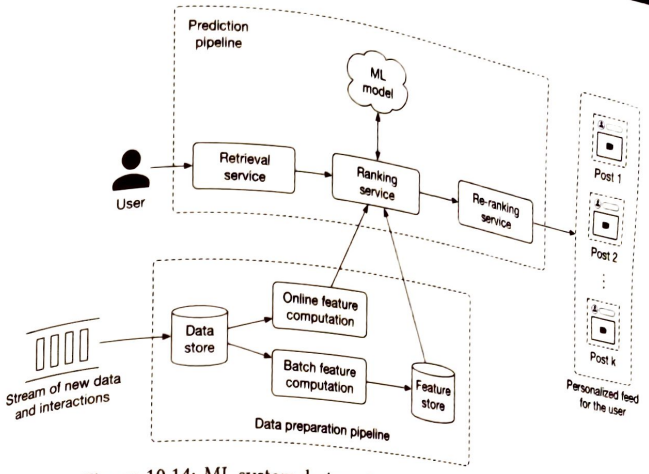


Figure 10.14: ML system design of a personalized news feed system

We do not go into detail about the data preparation pipeline because it is very similar to that described in Chapter 8, Ad Click Prediction in Social Platforms. Let's examine the prediction pipeline.

Prediction pipeline

The prediction pipeline consists of the following components: retrieval service, ranking service, and re-ranking service.

Retrieval service

This component retrieves posts that a user has not seen, or which have comments also unseen by them. To learn more about efficiently fetching unseen posts, read [14].

Ranking service

This component ranks the retrieved posts by assigning an engagement score to each one.

Re-ranking service

This service modifies the list of posts by incorporating additional logic and user filters. For example, if a user has explicitly expressed interest in a certain topic, such as soccer, this service assigns a higher rank to the post.

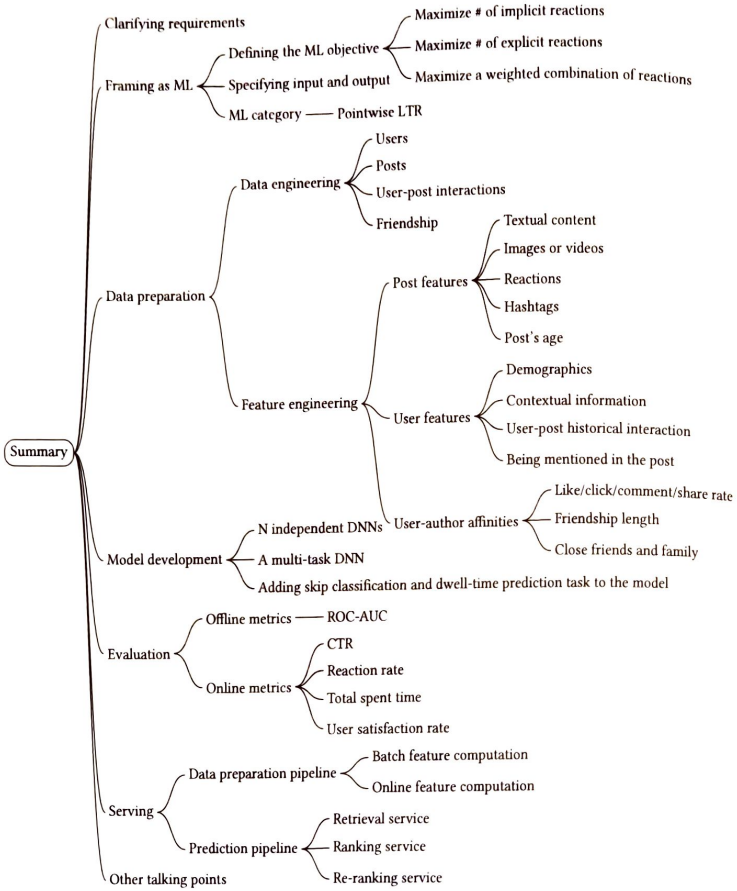
Other Talking Points

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

- How to handle posts that are going viral [15].

- How to personalize the news feed for new users [16].
- How to mitigate the positional bias present in the system [17].
- How to determine a proper retraining frequency [18].

Summary



Reference Material

[1] News Feed ranking in Facebook. https://engineering.fb.com/2021/01/26/ml-applications/news-feed-ranking/.

[2] Twitter's news feed system. https://blog.twitter.com/engineering/en_us/topics/insights/2017/using-deep-learning-at-scale-in-twitters-timelines.

[3] LinkedIn's News Feed system LinkedIn. https://engineering.linkedin.com/blog/2020/understanding-feed-dwell-time.

[4] BERT paper. https://arxiv.org/pdf/1810.04805.pdf.

[5] ResNet model. https://arxiv.org/pdf/1512.03385.pdf.

[6] CLIP model. https://openai.com/blog/clip/.

[7] Viterbi algorithm. https://en.wikipedia.org/wiki/Viterbi_algorithm.

[8] TF-IDF. https://en.wikipedia.org/wiki/Tf%E2%80%93idf.

[9] Word2vec. https://en.wikipedia.org/wiki/Word2vec.

[10] Serving a billion personalized news feed. https://www.youtube.com/watch?v=Xpx5RYNTQvg.

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

[12] Means squared error loss. https://en.wikipedia.org/wiki/Mean_squared_error.

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

[14] A news feed system design. https://liuzhenglaichn.gitbook.io/system-design/news-feed/design-a-news-feed-system.

[15] Predict viral tweets. https://towardsdatascience.com/using-data-science-to-predict-viral-tweets-615b0acc2e1e.

[16] Cold start problem in recommendation systems. https://en.wikipedia.org/wiki/Cold_start_(recommender_systems).

[17] Positional bias. https://eugeneyan.com/writing/position-bias/.

[18] Determine retraining frequency. https://huyenchip.com/2022/01/02/real-time-machine-learning-challenges-and-solutions.html#towards-continual-learning.

11 People You May Know

Introduction

People You May Know (PYMK) is a list of users with whom you may want to connect based on things you have in common, such as a mutual friend, school, or workplace. Many social networks, such as Facebook, LinkedIn, and Twitter, utilize ML to power PYMK functionality.

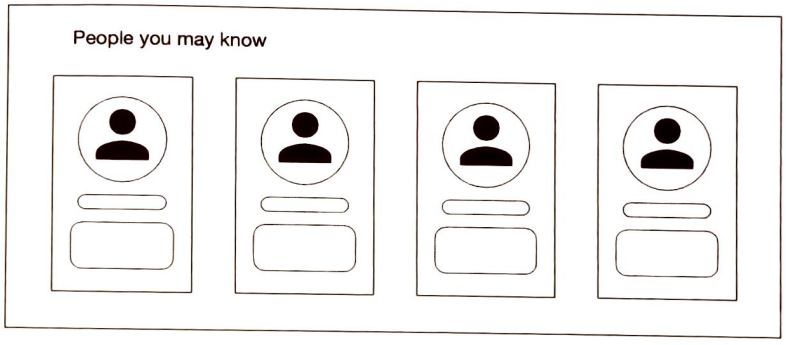


Figure 11.1: The PYMK feature

In this chapter, we will design a PYMK feature similar to LinkedIn's. The system takes a user as input and recommends a list of potential connections as output.

Clarifying Requirements

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

Candidate: Can I assume the motivation for building the PYMK feature is to help users discover potential connections and grow their network?
Interviewer: Yes, that's a good assumption.

Candidate: To recommend potential connections, a huge list of factors must be considered, such as location, educational background, work experience, existing connections, previous activities, etc. Should I focus on the most important factors, such as educational

background, work experience, and the user's social context?
Interviewer: That sounds good.

Candidate: On LinkedIn, two people are friends if – and only if – each is a friend of the other. Is that correct?
Interviewer: Yes, friendship is symmetrical. When someone sends a connection request to another user, the recipient needs to accept the request for the connection to be made.

Candidate: What's the total number of users on the platform? How many of them are daily active users?
Interviewer: We have nearly 1 billion users and 300 million daily active users.

Candidate: How many connections does an average user have?
Interviewer: 1,000 connections.

Candidate: The social graph of most users is not very dynamic, meaning their connections don't change significantly over a short period. Can I make this assumption when designing PYMK?
Interviewer: That's an excellent point. Yes, it's a reasonable assumption.

Let's summarize the problem statement. We are asked to design a PYMK system similar to LinkedIn's. The system takes a user as input and recommends a ranked list of potential connections as output. The motivation for building the system is to enable users to discover new connections more easily and grow their networks. There are 1 billion total users on the platform, and a user has 1,000 connections on average.

Frame the problem as an ML task

Defining the ML objective

A common ML objective in PYMK systems is to maximize the number of formed connections between users. This helps users to grow their networks quickly.

Specifying the system's input and output

The input to the PYMK system is a user, and the outputs are a list of connections ranked by relevance to the user. This is shown in Figure 11.2.

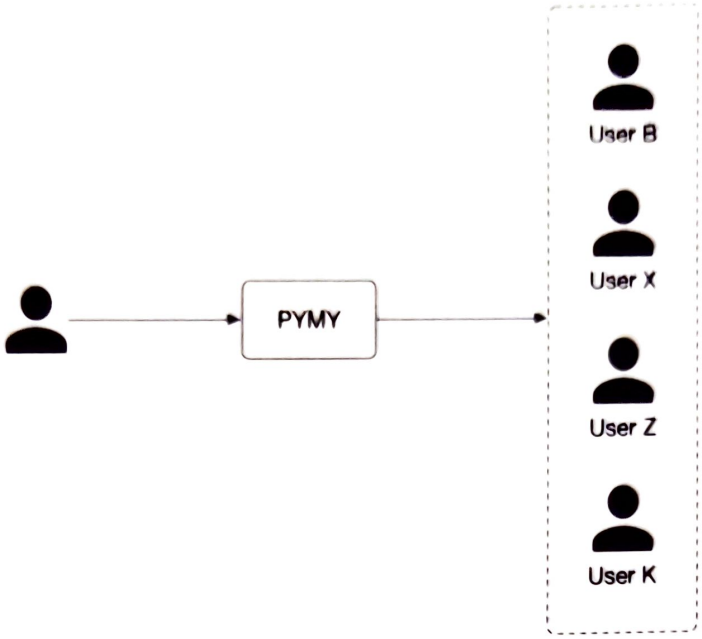


Figure 11.2: PYMK system's input-output

Choosing the right ML category

Let's examine two approaches commonly used to build PYMK: pointwise Learning to Rank (LTR) and edge prediction.

Pointwise LTR

In this approach, we frame PYMK as a ranking problem and use a pointwise LTR to rank users. In pointwise LTR, as Figure 11.3 shows, we employ a binary classification model which takes two users as input and outputs the probability of the given pair forming a connection.

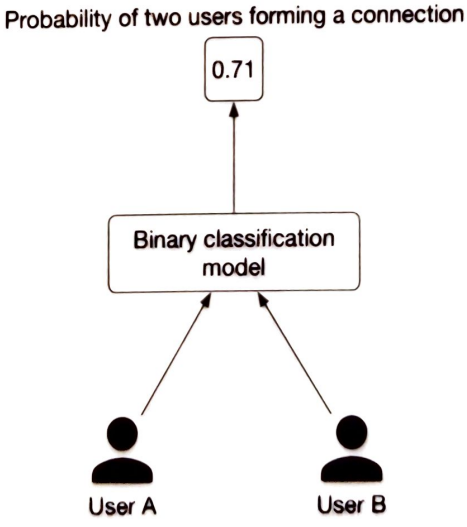


Figure 11.3: Binary classification with two input users

However, this approach has a major drawback; since the model's inputs are two distinct users, it doesn't consider the available social context. While this does simplify things, leaving out information about a user's connections might make predictions less accurate.

Let's analyze an example to understand how social context can provide very important insights. Imagine we want to predict whether or not ⟨user A, user B⟩ is a potential connection.

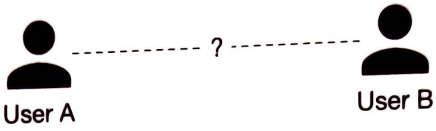


Figure 11.4: Can user A and user B form a potential connection?

By looking at their one-hop neighborhood (connections of user A or user B), we gain more information to determine if ⟨user A, user B⟩ is a potential connection. As shown in Figure 11.5, consider two different scenarios.

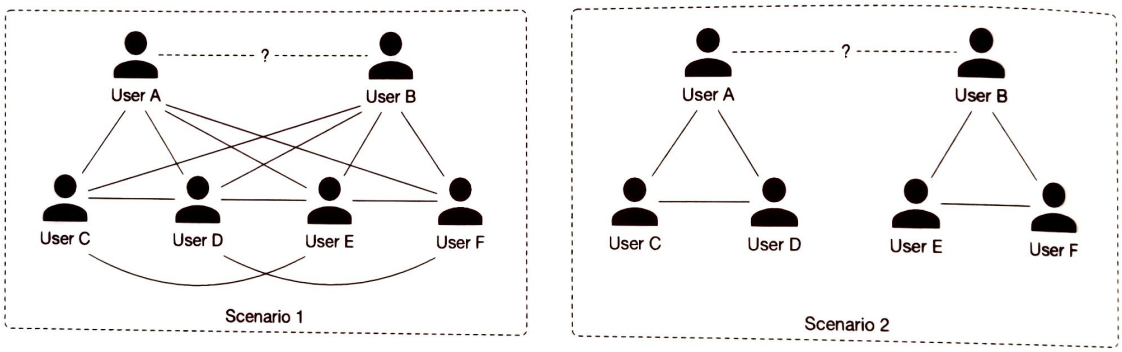


Figure 11.5: Two different scenarios of one-hop neighborhoods

In scenario 1, user A and user B each have four mutual connections, and there are mutual connections between users C, D, E, and F.

In scenario 2, user A and user B each have two friends, and there's no connection between user A and user B's connections.

By looking at their one-hop neighborhood, you might expect that ⟨user A, user B⟩ is more likely to form a connection in scenario 1 rather than in scenario 2. In practice, we can even leverage two-hop or three-hop neighborhoods to capture more useful information from the social context.

Before discussing the second approach, let's understand how graphs store structural data, such as the social context, and which machine learning tasks can be performed on graphs.

In general, a graph represents relations (edges) between a collection of entities (nodes). The entire social context can be represented by a graph, where each node represents a user, and an edge between two nodes indicates a formed connection between two users. Figure 11.6 shows a simple graph with four nodes and three edges.

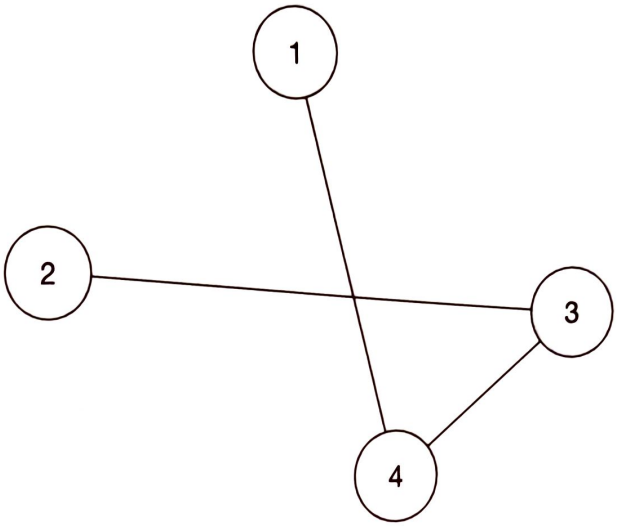


Figure 11.6: A simple graph

There are three general types of prediction tasks that can be performed on structured data represented by graphs:

- **Graph-level prediction.** For example, given a chemical compound as a graph, we predict whether the chemical compound is an enzyme or not.
- **Node-level prediction.** For example, given a social network graph, we predict if a specific user (node) is a spammer.
- **Edge-level prediction.** Predict if an edge is present between two nodes. For example, given a social network graph, we predict if two users are likely to connect.

Let's look at the edge prediction approach for building the PYMK system.

Edge prediction

In this approach, we supplement the model with graph information. This enables the model to rely on the additional knowledge extracted from the social graph, to predict whether an edge exists between two nodes.

More formally, we use a model that takes the entire social graph as input, and predicts the probability of an edge existing between two specific nodes. To rank potential connections for user A, we compute the edge probabilities between user A and other users, and use these probabilities as the ranking criteria.

In addition to the typical features that the model utilizes, the model also relies on additional knowledge extracted from the social graph to predict whether an edge exists between two nodes.

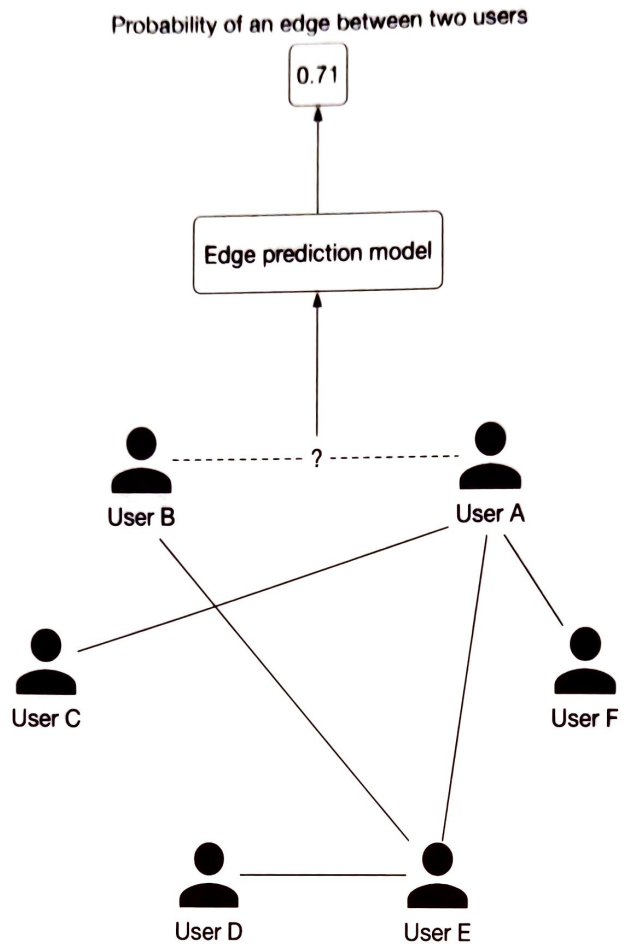


Figure 11.7: Binary classification with graph input

Data Preparation

Data engineering

In this section, we discuss the raw data available:

- Users
- Connections
- Interactions

Users

In addition to users' demographic data, we have information about their educational and work backgrounds, skills, etc. Table 11.1 shows an example of a user's educational background data. There might be similar tables to store work experiences, skills, etc.

User ID	School	Degree	Major	Start date	End date
11	Waterloo	M.Sc	Computer Science	August 2015	May 2017
11	Harvard	M.Sc	Physics	May 2004	August 2006
11	UCLA	Bachelors	Electrical Engineering	Sep 2022	-

Table 11.1: Users' educational background data

One challenge with this type of raw data is that a specific attribute can be represented in different forms. For example, "computer science" and "CS" have the same meaning, but the text differs. So, it's important to standardize the raw data during the data engineering step so we don't treat different forms of a single attribute differently.

There are various approaches to standardizing the raw data. For example:

- Force users to select attributes from a predefined list.
- Use heuristics to group different representations of an attribute.
- Use ML-based methods such as clustering [1] or language models to group similar attributes.

Connections

A simplified example of connection data is shown in Table 11.2. Each row represents a connection between two users and when the connection was formed.

User ID 1	User ID 2	Timestamp when the connection was formed
28	3	1658451341
7	39	1659281720
11	25	1659312942

Table 11.2: Connection data

Interactions

There are different types of interactions: a user sends a connection request, accepts a request, follows another user, searches for an entity, views a profile, likes or reacts to a post, etc. Note, in practice, we may store interaction data in different databases, but for simplicity, here, we include everything in a single table.

User ID	Interaction type	Interaction value	Timestamp
11	Connection request	user_id_8	1658450539
8	Accepted connection	user_id_11	1658451341
11	Comment	[usser_id_4, Very insightful]	1658451365
4	Search	"Scott Belsky"	1658435948
11	Profile view	user_id_21	1658451849

Table 11.3: Interaction data

Feature engineering

To determine potential connections for a user (e.g., user A), the model needs to utilize user A's information, such as age, gender, etc. In addition, the affinities between user A and other users are useful. In this section, we discuss some of the most important features.

User features

Demographics: age, gender, city, country, etc.

Demographic data helps determine if two users are likely to form a connection. Users tend to connect with others who have similar demographics.

It's common to have missing values in demographic data. To learn more about how to handle missing values, refer to the "Introduction and Overview" chapter.

The numbers of connections, followers, following, and pending requests

This information is important as users are more likely to connect with someone with lots of followers or connections, compared to a user with few connections.

Account's age

Accounts created very recently are less reliable than those that have existed for longer. For example, if an account was created yesterday, it's more likely to be a spam account. So, it may not be a good idea to recommend it to users.

The number of received reactions

These are numerical values representing the total number of reactions received, such as likes, shares, and comments over a certain period, like one week. Users tend to connect with more active users on the platform, who receive more interactions from other users.

User-user affinities

The affinity between two users is a good signal to predict if they will connect. Let's look at some important features which capture user-user affinities.

Education and work affinity

- **Schools in common:** Users tend to connect with others who attended the same school.
- **Contemporaries at school:** Overlapping years at school increases the likelihood of two users connecting. For example, users might want to connect with someone who attended school X the same time they did.
- **Same major:** A binary feature representing whether two users had the same major in school.
- **Number of companies in common:** Users may connect with people who have worked at the same companies.

- **Same industry:** A binary feature representing whether the two users work in the same industry.

Social affinity

- **Profile visits:** The number of times a user looks at the profile of another user.
- **Number of connections in common, aka mutual connections:** If two users have many common connections, they are more likely to connect. This feature is one of the most important predictive features [2].
- **Time discounted mutual connections:** This feature weighs mutual connections by how long they have existed. Let's go through an example to understand the reasoning behind this feature.

Imagine we want to determine whether user B is a potential connection for user A. Consider two scenarios: in scenario 1, user A's connections were formed very recently, whereas in scenario 2, the connections were formed a long time ago. This is shown in Figure 11.8.

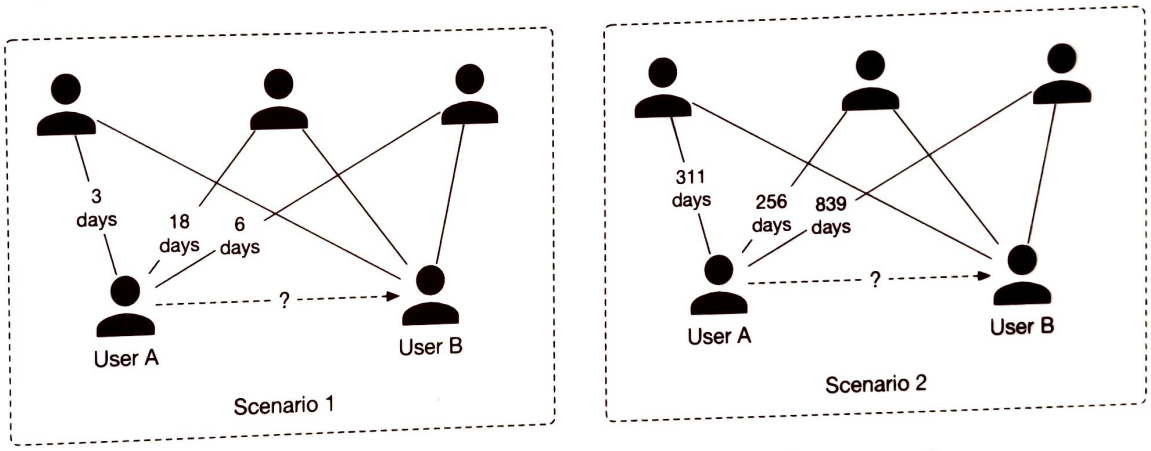


Figure 11.8: Comparing recent connections to old connections

In scenario 1, user A's network has grown recently, meaning it's more likely user A will connect with user B. Meanwhile, in scenario 2, the chances are that user A is aware of user B but has decided not to connect.

Model Development

Model selection

Earlier, we formulated the PYMK problem as an edge prediction task, where a model takes the social graph as input and predicts the probability of an edge existing between two users. To handle the edge prediction task, we choose a model that can process graph inputs. Graph neural networks (GNNs) are designed to operate on graph data. Let's take a closer look.

GNNs

GNNs are neural networks that can be directly applied to graphs. They provide an easy way to perform graph-level, node-level, and edge-level prediction tasks.

As shown in Figure 11.9, GNN takes a graph as input. This input graph contains attributes associated with nodes and edges. For example, the nodes can store information such as age, gender, etc., while the edges can store user-user characteristics, such as the number of common schools and workplaces, connection age, etc. Given the input graph and associated attributes, the GNN produces node embeddings for each node.

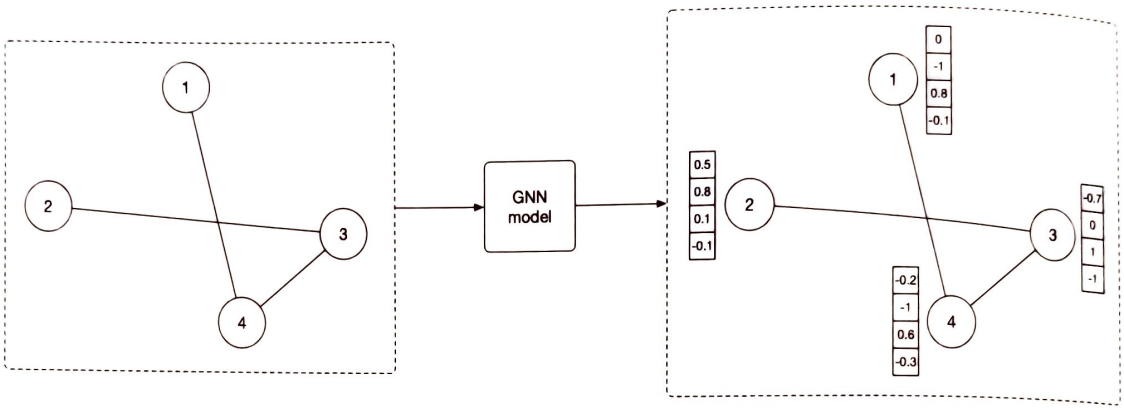


Figure 11.9: GNN model produces node embeddings for each graph node

Once the node embeddings are produced, they are used to predict how likely two nodes will form a connection using a similarity measure, such as dot product. For example, as shown in Figure 11.10, we compute the dot product between the embeddings of node 2 and node 4 to predict whether there is an edge between them.

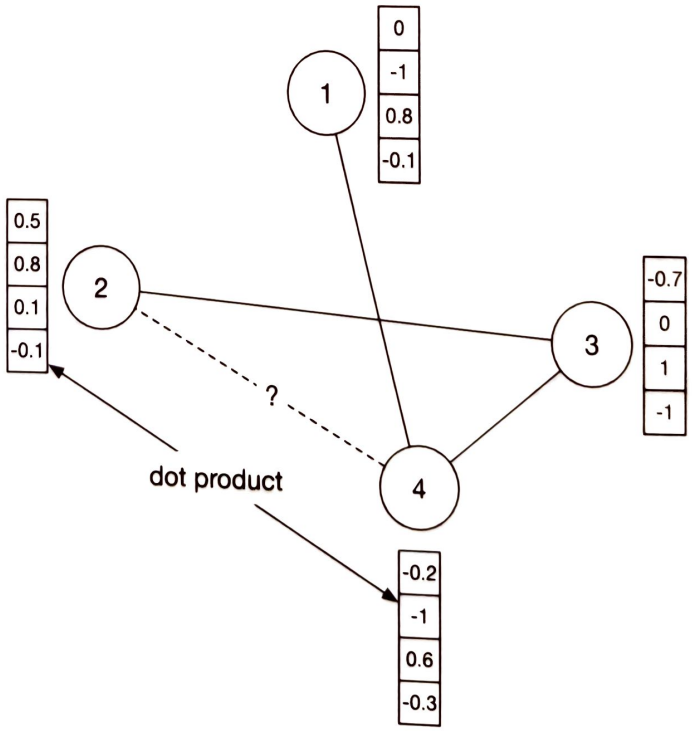


Figure 11.10: Predict how likely there is an edge between nodes 2 and 4

Many GNN-based architectures, such as GCN [3], GraphSAGE [4], GAT [5], and GIT [6], have been developed in recent years. These variants have different architectures and different levels of complexity. To determine which architecture works best, extensive experimentation is required. To gain a deeper understanding of GNN-based architectures, refer to [7].

Model training

To train a GNN model, we provide the model with a snapshot of the social graph at time t. The model predicts the connections which will form at time $t + 1$. Let's examine how to construct the training data.

Constructing the dataset

To construct the dataset, we do the following:

1. Create a snapshot of the graph at time t
2. Compute initial node features and edge features of the graph
3. Create labels

1. Create a snapshot of the graph at time t. The first step in constructing training data is to create input for the model. Since a GNN model expects a social graph as input, we create a snapshot of the social graph at time t using the available raw data. Figure 11.11 shows an example of the graph at time t.

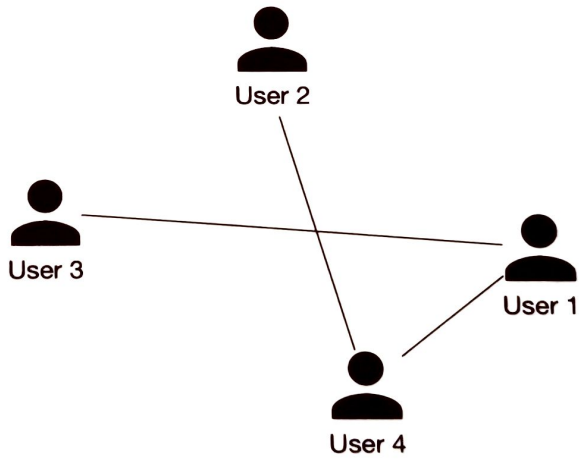


Figure 11.11: A snapshot of the social graph at time t

2. Compute initial node features and edge features of the graph. As shown in Figure 11.12, we extract the user's features, such as age, gender, account age, number of connections, etc. These are used as the nodes' initial feature vectors.

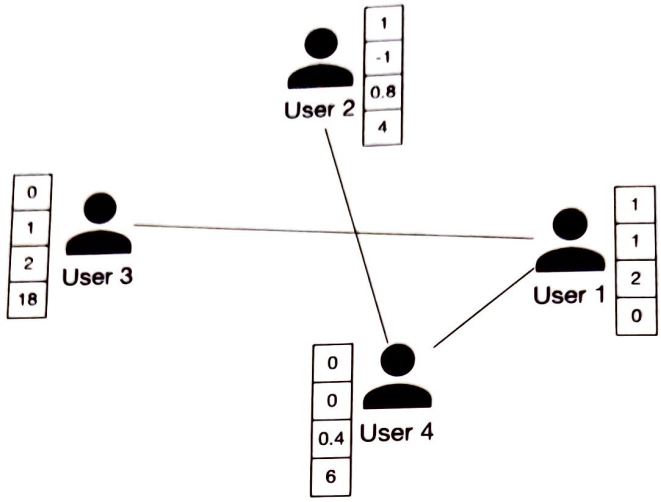


Figure 11.12: Initial edge features

Similarly, we extract user-user affinity features and employ them as the initial feature vectors of the edges. As shown in Figure 11.13, there is an edge between user 2 and user 4. $E_{2,4}$ represents the initial feature vector which captures information such as the number of mutual connections, profile visits, overlapping time at schools in common, etc.

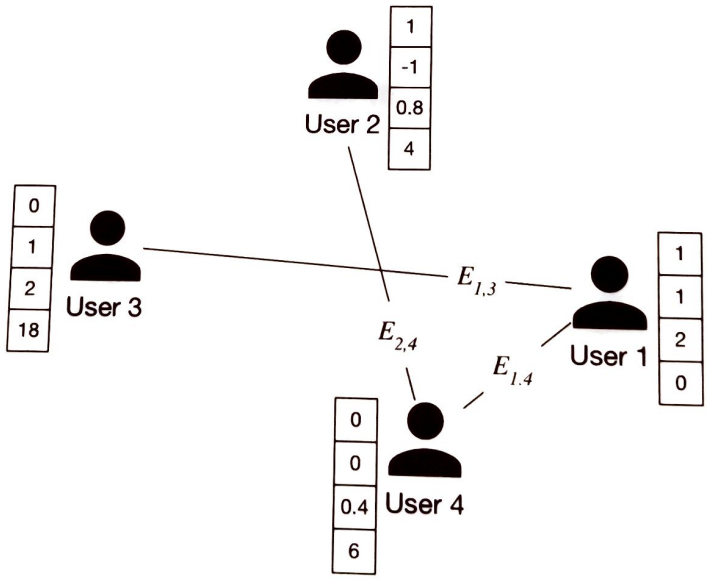


Figure 11.13: Initial node features

3. Create labels

In this step, we create labels that the model is expected to predict. We use the graph snapshot at time $t + 1$ to determine positive or negative labels. Let's take a look at a concrete example.

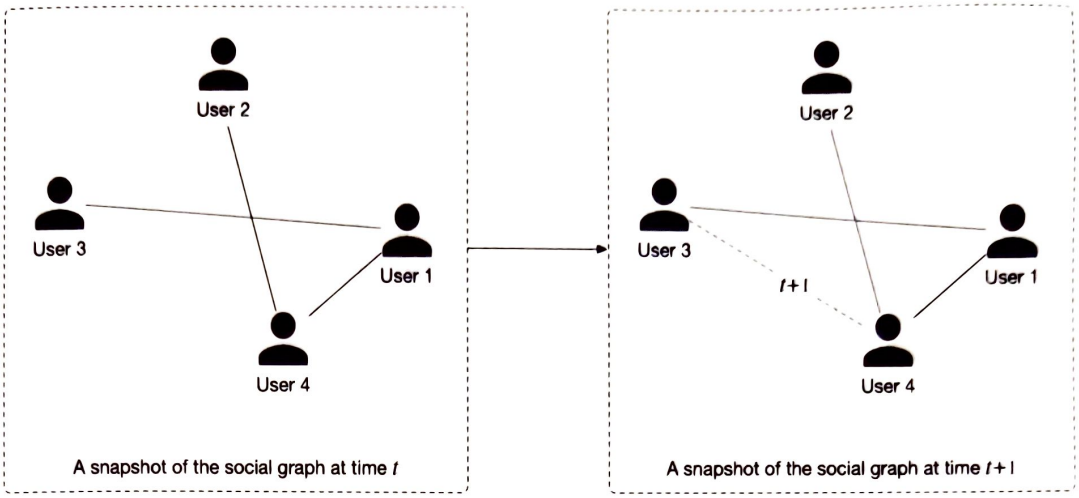


Figure 11.14: Newly formed edges from time t to $t + 1$

As shown in Figure 11.14, positive and negative labels are created depending on whether a new edge forms at $t + 1$. In particular, we label a pair of nodes as positive when they connect at $t + 1$. Otherwise, they are labeled as negative.

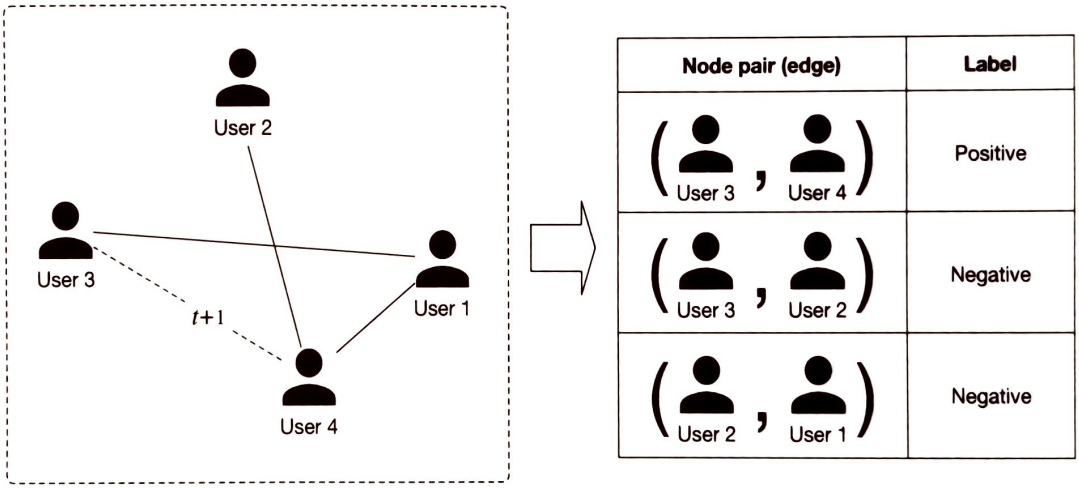


Figure 11.15: Creating positive and negative labels

Choosing the loss function

Once the input graph and labels are created, we are ready to train the GNN model. A detailed explanation of how GNN training works and which loss functions to employ is beyond the scope of this book. To learn more about these, see [7].

Evaluation

Offline metrics

During the offline evaluation, we evaluate the performance of the GNN model and the PYMK system.

GNN model

Since the GNN model predicts the presence of edges, we can think of it as a binary classification model. ROC-AUC metric is used to measure the performance of the model.

PYMK system

We extensively discuss choosing the right offline metrics for ranking and recommendation systems in previous chapters, so don't go into detail here. In our system, a user will either connect with a recommended connection or discard it. Due to this binary nature (connect or not), mAP is a good choice.

Online metrics

In practice, companies track lots of online metrics to measure the impact of PYMK systems. Let's explore two of the most important metrics:

- The total number of connection requests sent in the last X days
- The total number of connection requests accepted in the last X days

The total number of connection requests sent in the last X days. This metric helps us understand if the model increases or decreases the number of connection requests. For example, if a model leads to a 5% increase in the total number of sent connection requests, we can assume the model has a positive impact on the business objective.

However, this metric has a major drawback. A new connection forms between two users only when the recipient accepts a request to connect. For example, a user may send 1,000 connection requests, but recipients accept only a small percentage. This metric might not correctly reflect the actual growth of the users' network. Now, let's address this drawback with the next metric.

The total number of connection requests accepted in the last X days. As a new connection forms only when the recipient accepts the sender's request, this metric accurately reflects the real growth of the users' network.

Serving

At serving time, the PYMK system efficiently recommends a list of potential connections to a given user. In this section, we explain why speed optimization is needed and introduce some techniques to make PYMK efficient. Then, we propose a design in which different components work together to serve requests.

Efficiency

As discussed in the requirement gathering section, the total number of users on the platform is 1 billion, which indicates we need to sort through 1 billion embeddings to find potential connections for a single user. To make things even more challenging, the algorithm needs to be run for each user. Unsurprisingly, this is impractical at our scale. To mitigate the issue, two common techniques are used: 1) utilizing friends of friends (FoF) and 2) pre-compute PYMK.

Utilizing FoF

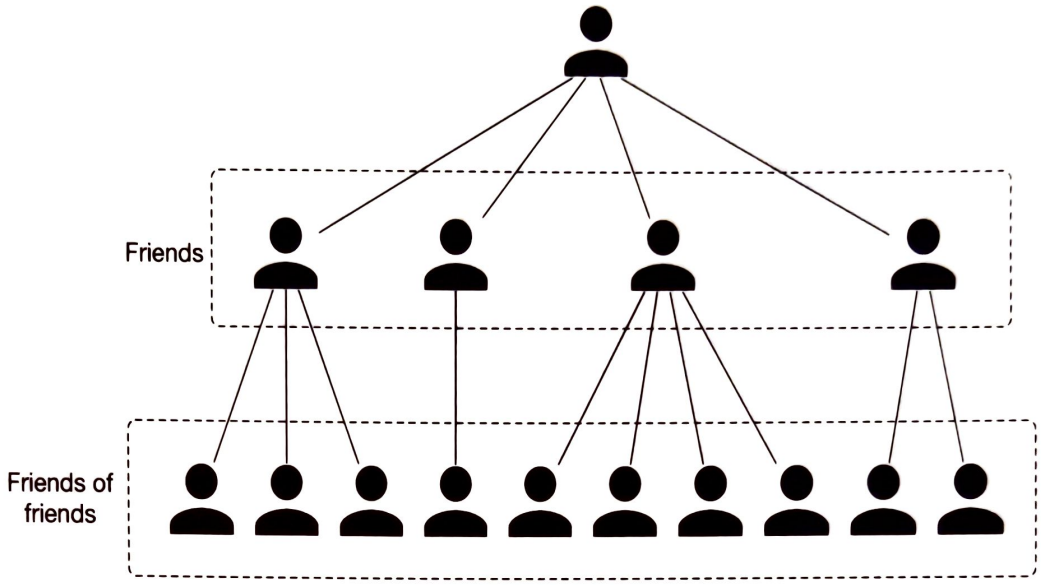


Figure 11.16: FoF of a user

According to a Meta study [2], 92% of new friendships are formed via FoF. This technique uses a user's FoF to narrow down the search space.

As previously mentioned, a user has 1,000 friends on average. That means a user has 1 million (1000×1000) FoF, on average. This reduces the search space from 1 billion to 1 million.

Pre-compute PYMK

Let's take a step back and consider adopting online or batch predictions.

Online prediction

In PYMK, online prediction refers to generating potential connections in real-time when a user loads the homepage. In this approach, we don't generate recommendations for inactive users. Since recommendations are calculated "on the fly", if computing the recommendations takes a long time, it creates a poor user experience.

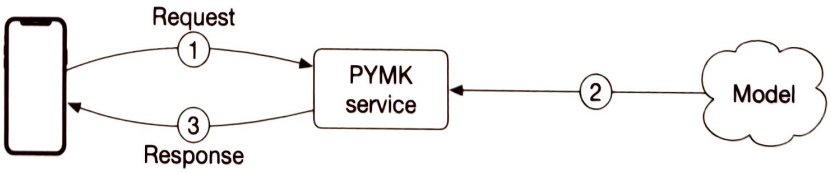


Figure 11.17: Online prediction in PYMK

Batch prediction

Batch prediction means the system pre-computes potential connections for all users and stores them in a database. When a user loads the homepage, we fetch pre-computed recommendations directly, so from the end user's standpoint, the recommendation is

instantaneous. The downside of batch prediction is that we may end up with unnecessary computations. Imagine 20% of users log in daily. If we generate recommendations for every user daily, then the computing power used to generate 80% of recommendations will be wasted.

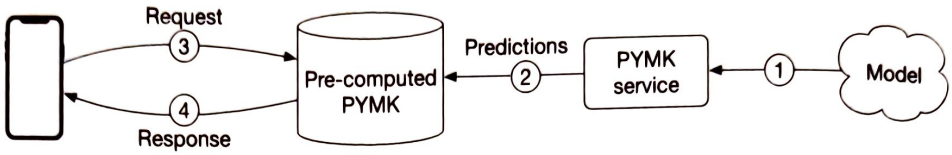


Figure 11.18: Batch prediction in PYMK

Which option do we choose: online or batch?
We recommend batch prediction for two reasons. First, based on the requirements gathered, there are 300 million daily active users. Computing PYMK for all 300 million users on the fly may be too slow for a quality user experience.

Second, as the social graph in PYMK does not evolve quickly, the pre-computed recommendations remain relevant for an extended period. For example, we can keep PYMK recommendations for seven days and then re-compute them. The time window can be shortened (for instance, by one day) for newer users because their networks tend to grow faster.

In a social network, a user may not want to see the same set of recommended connections repeatedly. To support this, we can pre-compute more connections than needed and only display those a user hasn't seen before.

ML system design

Figure 11.19 shows the PYMK ML system design. The design comprises two pipelines:

- PYMK generation pipeline
- Prediction pipeline

Let's inspect each.

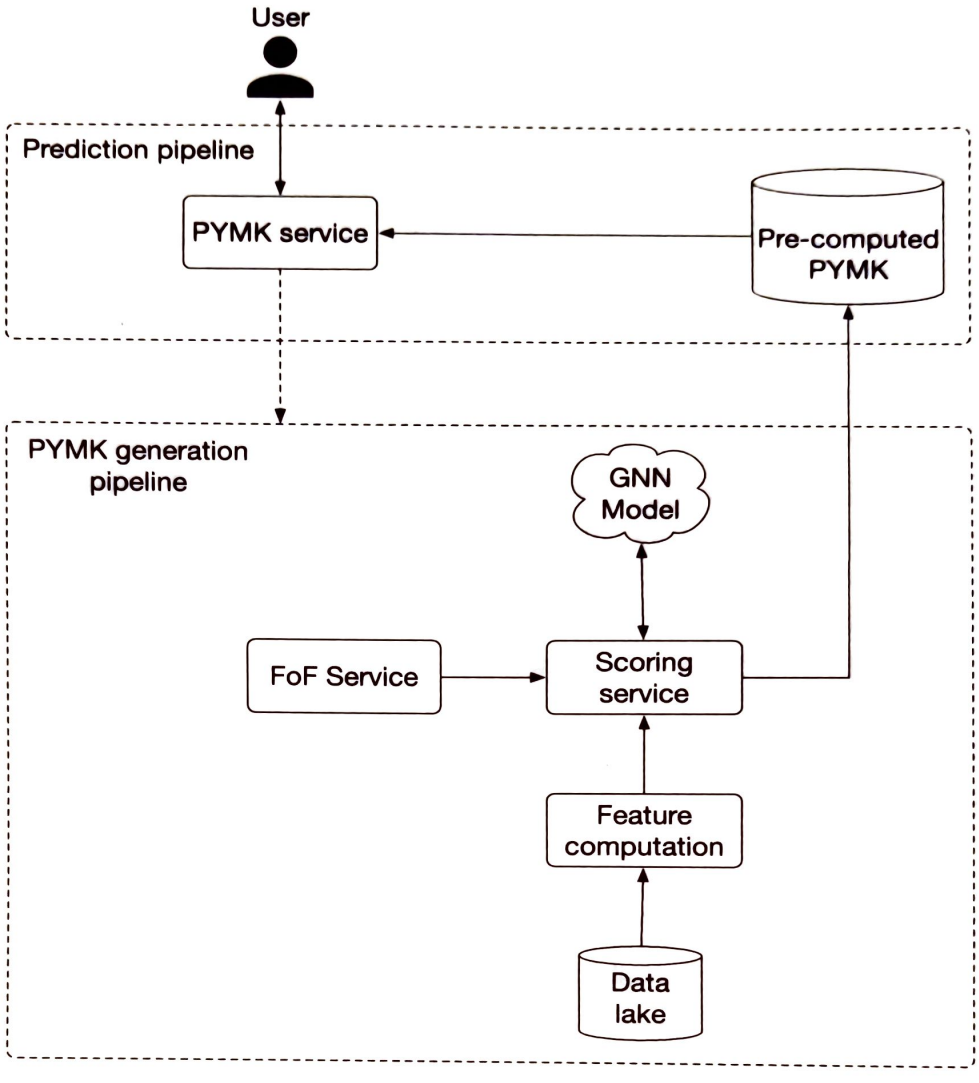


Figure 11.19: PYMK ML system design

PYMK generation pipeline

This pipeline is responsible for generating PYMK for all users and storing the results in a database. Let's take a closer look at this pipeline.

First, for a specific user, the FoF service narrows down the connections into a subset of candidate connections (2-hop neighbors). This is shown in Figure 11.20.

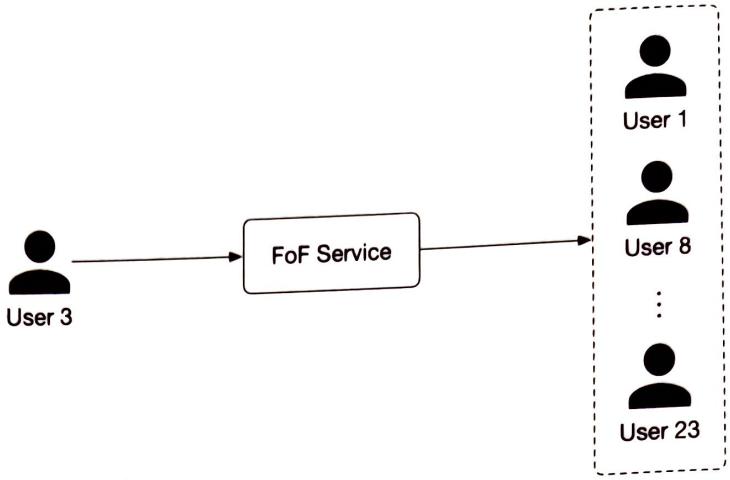


Figure 11.20: FoF service input-output

Next, the scoring service takes the candidate connections produced by the FoF service, scores each of them using the GNN model, then generates a ranked list of PYMK for the user. The PYMK is stored in a database. When a user request is made, we can simply pull their individual PYMK list directly from the database. This flow is shown in Figure 11.21.

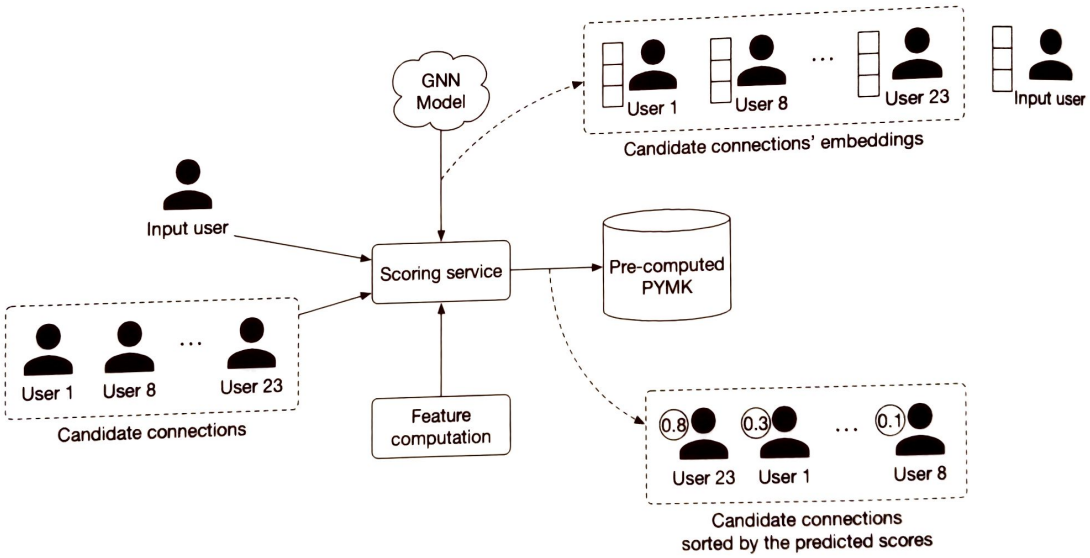


Figure 11.21: Scoring service input-output

Prediction pipeline

When a request arrives, the PYMK service first looks at the pre-computed PYMKs to see if recommendations exist. If they do, recommendations are fetched directly. If not, it sends a one-time request to the PYMK generation pipeline.

Note that what we have proposed is a simplified system. If you asked during an interview to optimize it, here are a few potential talking points:

- Pre-computing PYMK only for active users.

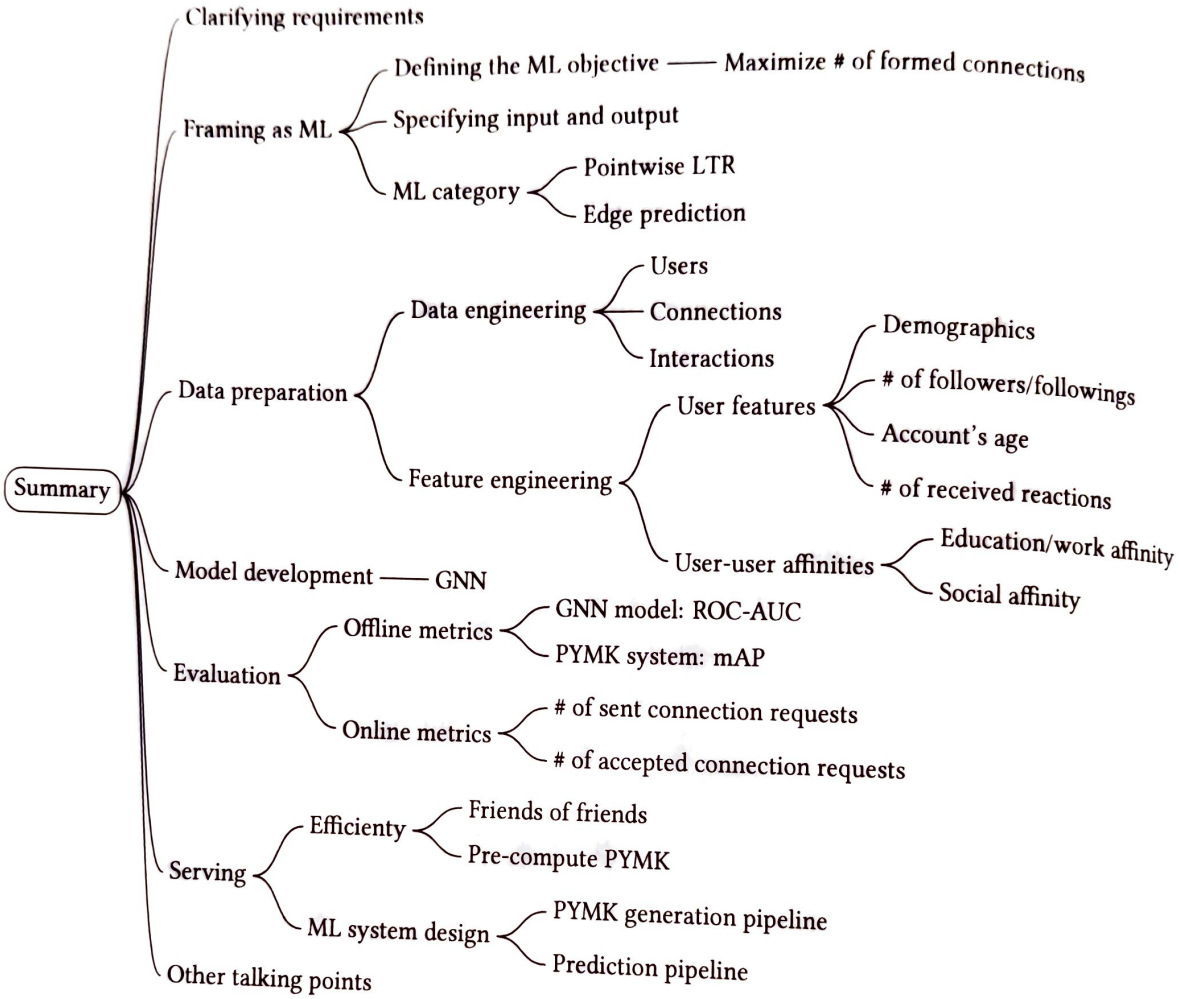
- Using a lightweight ranker to reduce the number of generated candidates into a smaller set before the scoring service assigns them a score.
- Using a re-ranking service to add diversity to the final PYMK list.

Other Talking Points

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

- Personalized random walk [8] is another method often used to make recommendations. Since it's efficient, it is a helpful way to establish a baseline.
- Bias issue. Frequent users tend to have greater representation in the training data than occasional users. The model can become biased towards some groups and against others due to uneven representation in the training data. For example, in the PYMK list, frequent users might be recommended to other users at a higher rate. Subsequently, these users can make even more connections, making them even more represented in the training data [9].
- When a user ignores recommended connections repeatedly, the question arises of how to take them into account in future re-ranks. Ideally, ignored recommendations should have a lower ranking [9].
- A user may not send a connection request immediately when we recommend it to them. It may take a few days or weeks. So, when should we label a recommended connection as negative? In general, how would we deal with delayed feedback in recommendation systems [10]?

Summary


Summary
Clarifying requirements
Framing as ML
Defining the ML objective — Maximize # of formed connections
Specifying input and output
ML category
Pointwise LTR
Edge prediction
Data preparation
Data engineering
Users
Connections
Interactions
Feature engineering
User features
Demographics
# of followers/followings
Account's age
# of received reactions
User-user affinities
Education/work affinity
Social affinity
Model development — GNN
Evaluation
Offline metrics
GNN model: ROC-AUC
PYMK system: mAP
Online metrics
# of sent connection requests
# of accepted connection requests
Serving
Efficienty
Friends of friends
Pre-compute PYMK
ML system design
PYMK generation pipeline
Prediction pipeline
Other talking points

Reference Material

[1] Clustering in ML. https://developers.google.com/machine-learning/clustering/overview.

[2] PYMK on Facebook. https://youtu.be/Xpx5RYNTQvg?t=1823.

[3] Graph convolutional neural networks. http://tkipf.github.io/graph-convolutional-networks/.

[4] GraphSage paper. https://cs.stanford.edu/people/jure/pubs/graphsage-nips17.pdf.

[5] Graph attention networks. https://arxiv.org/pdf/1710.10903.pdf.

[6] Graph isomorphism network. https://arxiv.org/pdf/1810.00826.pdf.

[7] Graph neural networks. https://distill.pub/2021/gnn-intro/.

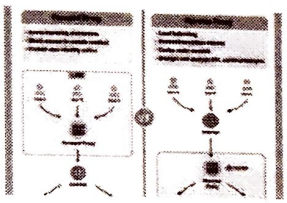
[8] Personalized random walk. https://www.youtube.com/watch?v=HbzQzUaJ_9I.

[9] LinkedIn's PYMK system. https://engineering.linkedin.com/blog/2021/optimizing-pymk-for-equity-in-network-creation.

[10] Addressing delayed feedback. https://arxiv.org/pdf/1907.06558.pdf.

System Design Newsletter

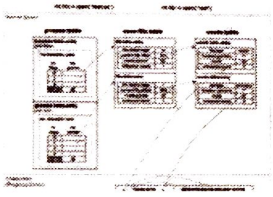
Subscribe to our weekly newsletter: blog.bytebytego.com



EP26: Proxy vs reverse proxy

In this issue, we will cover: Why is Nginx called a "reverse" proxy? CAP theorem How Does Live Streaming Platform Work? CDN Postman the API platform for...

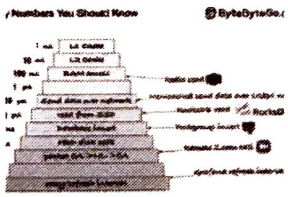
ALEX XU OCT 1 ♡ 225 💬 6



EP17: Design patterns cheat sheet. Also...

For this week's newsletter, we will cover: Design patterns cheat sheet 6 ways to turn code into beautiful architecture diagrams What is a File...

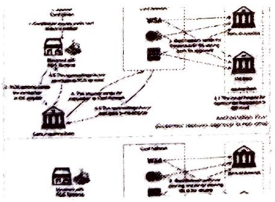
ALEX XU JUL 30 ♡ 166 💬 7



EP22: Latency numbers you should know. Also...

In this newsletter, we'll cover the following topics: Latency numbers you should know Microservice architecture Handling hotspot accounts E-commerce...

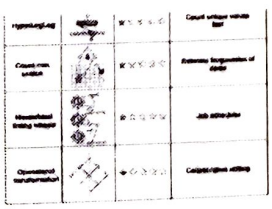
ALEX XU SEP 3 ♡ 153 💬 9



EP15: What happens when you swipe a credit card? Also...

For this week's newsletter, we will cover: How does VISA work when we swipe a credit card at a merchant's shop? What are the differences between bare...

ALEX XU JUL 16 ♡ 141 💬 8



EP14: Algorithms you should know for System Design. Also...

In this newsletter, we'll cover the following topics: Algorithms you should know before taking System Design Interviews How to store passwords safely in...

ALEX XU JUL 9 ♡ 185 💬 2

YouTube Channel

Check us out on YouTube: https://www.youtube.com/@ByteByteGo