Educative.io Cracking the Machine Learning Engineer Interview

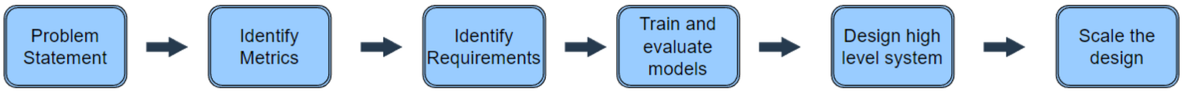
Machine Learning Primer

1. What should you expect in a Machine Learning Interview?

- Most major companies, i.e. Facebook, LinkedIn, Google, Amazon, and Snapchat, expect Machine Learning engineers to have solid engineering foundations and hands-on Machine Learning experiences. This is why interviews for Machine Learning positions share similar components with interviews for traditional software engineering positions. The candidates go through a similar method of problem solving (Leetcode style), system design, knowledge of machine learning and machine learning system design.
- The standard development cycle of machine learning includes data collection, problem formulation, model creation, implementation of models, and enhancement of models. It is in the company's best interest throughout the interview to gather as much information as possible about the competence of applicants in these fields. There are plenty of resources on how to train machine learning models and how to deploy models with different tools. However, there are no common guidelines for approaching machine learning system design from end to end. This was one major reason for designing this course.

2. How will this course help you?

- In this course, we will learn how to approach machine learning system design from a top-down view. It's important for candidates to realize the challenges early on and address them at a structural level. Here is one example of the thinking flow.



The 6 basic steps to approach Machine Learning System Design

Problem statement

It’s important to state the correct problems. It is the candidates job to understand the intention of the design and why it is being optimized. It’s important to make the right assumptions and discuss them explicitly with interviewers. For example, in a LinkedIn feed design interview, the interviewer might ask broad questions:

Design LinkedIn Feed Ranking.

Asking questions is crucial to filling in any gaps and agreeing on goals. The candidate should begin by asking follow-up questions to clarify the problem statement. For example:

Is the output of the feed in chronological order?

How do we want to balance feeds versus sponsored ads, etc.?

If we are clear on the problem statement of designing a Feed Ranking system, we can then start talking about relevant metrics like user agreements.

Identify metrics

During the development phase, we need to quickly test model performance using offline metrics. You can start with the popular metrics like logloss and AUC for binary classification, or RMSE and MAPE for forecast.

Identify requirements

Training requirements

There are many components required to train a model from end to end. These components include the data collection, feature engineering, feature selection, and loss function. For example, if we want to design a YouTube video recommendations model, it’s natural that the user doesn’t watch a lot of recommended videos. Because of this, we have a lot of negative examples. The question is asked:

How do we train models to handle an imbalance class?

Once we deploy models in production, we will have feedback in real time.

How do we monitor and make sure models don’t go stale?

Inference requirements

Once models are deployed, we want to run inference with low latency (<100ms) and scale our system to serve millions of users.

How do we design inference components to provide high availability and low latency?

Train and evaluate model

There are usually three components: feature engineering, feature selection, and models. We will use all the modern techniques for each component.

For example, in Rental Search Ranking, we will discuss if we should use ListingID as embedding features. In Estimate Food Delivery Time, we will discuss how to handle the latitude and longitude features efficiently.

Design high level system

In this stage, we need to think about the system components and how data flows through each of them. The goal of this section is to identify a minimal, viable design to demonstrate a working system. We need to explain why we decided to have these components and what their roles are.

For example, when designing Video Recommendation systems, we would need two separate components: the Video Candidate Generation Service and the Ranking Model Service.

Scale the design

In this stage, it’s crucial to understand system bottlenecks and how to address these bottlenecks. You can start by identifying:

Which components are likely to be overloaded?

How can we scale the overloaded components?

Is the system good enough to serve millions of users?

How we would handle some components becoming unavailable, etc.

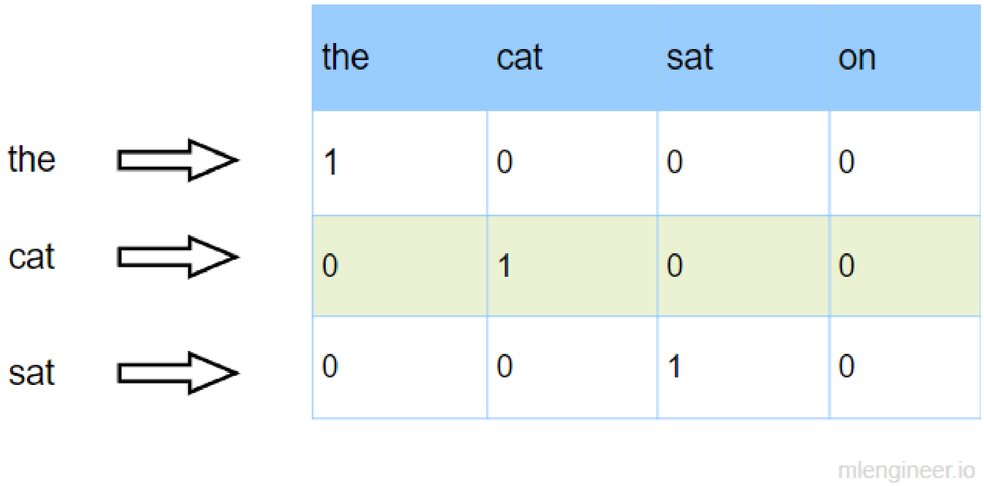
You can also learn more about how companies scale there design here.

Feature Engineering and Feature Selection

Learn how tech companies like Facebook, Twitter, and Airbnb design their feature selection and feature engineering to serve billions of users.

1. One hot encoding

- One hot encoding is a very common technique in feature engineering. It converts categorical variables into a one-hot numeric array.
- One hot encoding is very popular when you have to deal with categorical features that have medium cardinality.



	the	cat	sat	on
the	1	0	0	0
cat	0	1	0	0
sat	0	0	1	0

mlengineer.io

One hot encoding example

Common problems

- Expansive computation and high memory consumption are major problems with one hot encoding. High numbers of values will create high-dimensional feature vectors. For example, if there are one million unique values in a column, it will produce feature vectors that have a dimensionality of one million.
- One hot encoding is not suitable for Natural Language Processing tasks. Microsoft Word's dictionary is usually large, and we can't use one hot encoding to represent each word as the vector is too big to store in memory.

Best practices

- Depending on the application, some levels/categories that are not important, can be grouped together in the "Other" class.
- Make sure that the pipeline can handle unseen data in the test set.
- In Python, there are many ways to do one hot encoding, for example, pandas.get_dummies and sklearn OneHotEncoder pandas.get_dummies does not "remember" the encoding during training, and if testing data has new values, it can lead to inconsistent mapping. OneHotEncoder is a Scikitlearn Transformer; therefore, you can use it consistently during training and predicting.

One hot encoding in tech companies

- It's not practical to use one hot encoding to handle large cardinality features, i.e., features that have hundreds or thousands of unique values. Companies like Instacart and DoorDash use more advanced techniques to handle large cardinality features.

2. Feature hashing

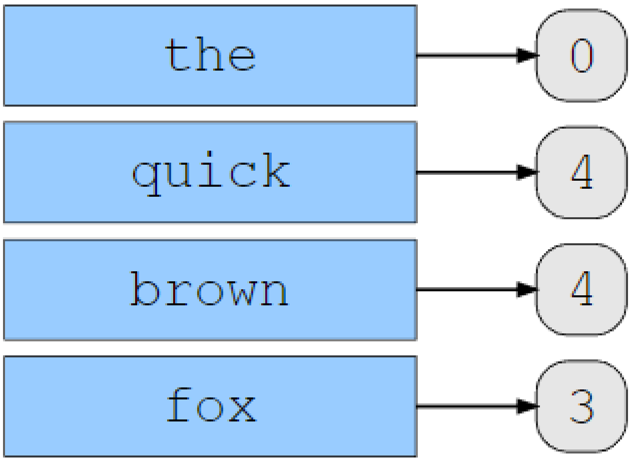
- Feature hashing, called the hashing trick, converts text data or categorical attributes with high cardinalities into a feature vector of arbitrary dimensionality.

Benefits

Feature hashing is very useful for features that have high cardinality with hundreds and thousands of unique values. Hashing trick is a way to reduce the increase in dimension and memory by allowing multiple values to be present/encoded as the same value.

Feature hashing example

First, you chose the dimensionality of your feature vectors. Then, using a hash function, you convert all values of your categorical attribute (or all tokens in your collection of documents) into a number. Then you convert this number into an index of your feature vector. The process is illustrated in the diagram below.



Let’s illustrate what it would look like to convert the text “The quick brown fox” into a feature vector. The values for each word in the phrase are:

```
the = 5
quick = 4
brown = 4
fox = 3
```

Let define a hash function, h*h*, that takes a string as input and outputs a non-negative integer. Let the desired dimensionality be 5. By applying the hash function to each word and applying the modulo of 5 to obtain the index of the word, we get:

```
h(the) mod 5 = 0
h(quick) mod 5 = 4
h(brown) mod 5 = 4
h(fox) mod 5 = 3
```

- In this example:
    - `h(the) mod 5 = 0` means that we have one word in dimension **0** of the feature vector.
    - `h(quick) mod 5 = 4` and `h(brown) mod 5 = 4` means that we have two words in dimension **4** of the feature vector.
    - `h(fox) mod 5 = 3` means that we have one word in dimension **3** of the feature vector.
    - As you can see, that there are no words in dimensions 1 or 2 of the vector, so we keep them as 0.
- Finally, we have the feature vector as: `[1, 0, 0, 1, 2]`.
- As you can see, there is a collision between words “quick” and “brown.” They are both represented by dimension 4. The lower the desired dimensionality, the higher the chances of collision. To reduce the probability of collision, we can increase the desired dimensions. This is the trade-off between speed and quality of learning.

Commonly used hash functions are MurmurHash3, Jenkins, CityHash, and MD5.

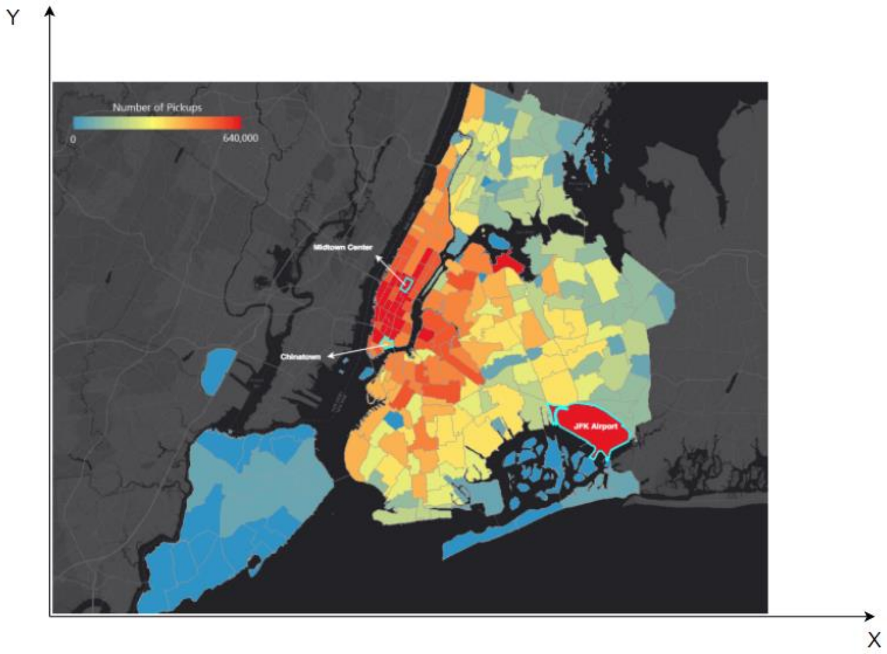
Feature hashing in tech companies

- Feature hashing is popular in many tech companies like Booking, Facebook, Yahoo, Yandex, Avazu and Criteo.
- One problem with hashing is collision. If the hash size is too small, more collisions will happen and negatively affect model performance. On the other hand, the larger the hash size, the more it will consume memory.
- Collisions also affect model performance. With high collisions, a model won’t be able to differentiate coefficients between feature values. For example, the coefficient for “User login/ User logout” might end up the same, which makes no sense.

Depending on application, you can choose the number of bits for feature hashing that provide the correct balance between model accuracy and computing cost during model training. For example, by increasing hash size we can improve performance, but the training time will increase as well as memory consumption.

3. Crossed feature

- Crossed features, or **conjunction**, between two categorical variables of cardinality `c1` and `c2` is just another categorical variable of cardinality `c1` × `c2`. If `c1` and `c2` are large, the conjunction feature has high cardinality, and the use of the hashing trick is even more critical in this case. Crossed feature is usually used with a hashing trick to reduce high dimensions.
- As an example, suppose we have Uber pick-up data with latitude and longitude stored in the database, and we want to predict demand at a certain location. If we just use the feature latitude for learning, the model might learn that a city block at a particular latitude is more likely to have a higher demand than others. This is similar for the feature longitude. However, a feature cross of longitude by latitude would represent a well-defined city block. Consequently, the model will learn more accurately.



Uber Pickups Map in 2015 (source Uber)

Read more about different techniques in handling latitude/longitude here: Haversine distance, Manhattan distance.

Crossed feature in tech companies

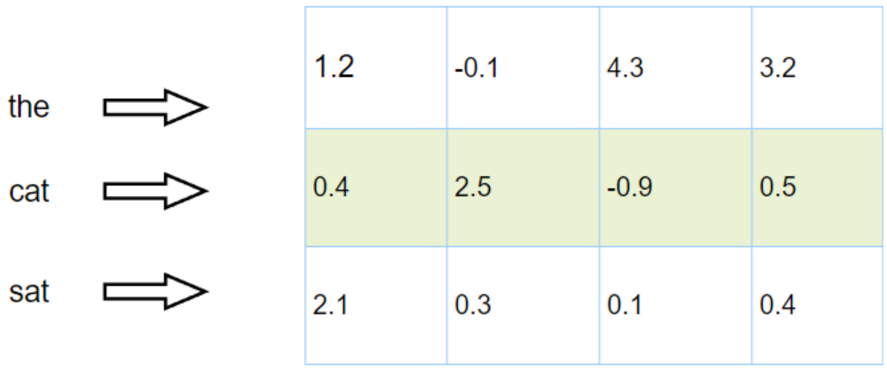
This technique is commonly applied at many tech companies. LinkedIn uses crossed features between user location and user job title for their Job recommendation model. Airbnb also uses cross features for their Search Ranking model.

4. Embedding#

Feature embedding is an emerging technique that aims to transform features from the original space into a new space to support effective machine learning. The purpose of embedding is to capture semantic meaning of features; for example, similar features will be close to each other in the embedding vector space.

Benefits#

- Both one hot encoding and feature hashing can represent features in another dimension. However, these representations do not usually preserve the semantic meaning of each feature. For example, in the **Word2Vector** representation, embedding words into dense multidimensional representation helps to improve the prediction of the next words significantly.



the	1.2	-0.1	4.3	3.2
cat	0.4	2.5	-0.9	0.5
sat	2.1	0.3	0.1	0.4

A 4-dimensional embedding

- As an example, each word can be represented as a `d` dimension vector, and we can train our supervised model. We then use the output of one of the fully connected layers of the neural network model as embeddings on the input object. For example, `cat` embedding can be represented as a `[1.2, -0.1, 4.3, 3.2]` vector.

How to generate/learn embedding vector?#

- For popular deep learning frameworks like TensorFlow, you need to define the dimension of embedding and network architecture. Once defined, the network can learn embedding automatically. For example:

```
# Embed a 1,000 word vocabulary into 5
dimensions.
embedding_layer =
tf.keras.layers.Embedding(1000, 5)
```

Embedding in tech companies#

This technique is commonly applied at many tech companies.

- Twitter uses Embedding for UserIDs and cases like recommendations, nearest neighbor searches, and transfer learning.

- Doordash uses Store Embedding (store2vec) to personalize the store feed. Similar to word2vec, each store is one word and each sentence is one user session. Then, to generate vectors for a consumer, we sum the vectors for each store they ordered from in the past 6 months or a total of 100 orders. Then, the distance between a store and a consumer is determined by taking the cosine distance between the store's vector and the consumer's vector.

- Similarly, Instagram uses account embedding to recommend relevant content (photos, videos, and Stories) to users.

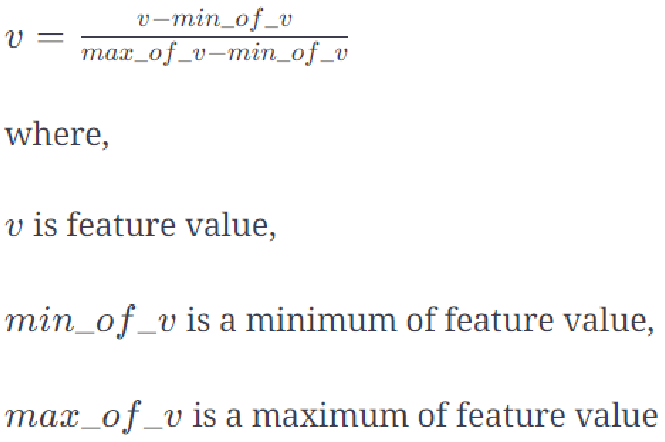
The embedding dimensionality is usually determined experimentally or from experience. In TensorFlow documentation, they recommend: d = fourth root of D. Where D is the number of categories. Another way is to treat d*d* as a hyperparameter, and we can tune on a downstream task.

In large scale production, embedding features are usually pre-computed and stored in key/value storage to reduce inference latency.

5. Numeric features#

Normalization#

- For numeric features, normalization can be done to make the mean equal 0, and the values be in the range [-1, 1]. There are some cases where we want to normalize data between the range [0, 1].



$$v = \frac{v - min\_of\_v}{max\_of\_v - min\_of\_v}$$

where,

v is feature value,

min_of_v is a minimum of feature value,

max_of_v is a maximum of feature value

Standardization#

- If features distribution resembles a normal distribution, then we can apply a standardized transformation.

$$v = \frac{v - mean\_of\_v}{std\_of\_v}$$

where,

v is feature value,

$mean\_of\_v$ is a mean of feature value,

std_of_v is the standard deviation of feature value

If feature distribution resembles power laws, then we can transform it by using the formula:

$$log(\frac{1+v}{1+median\_of\_v})$$

In practice, normalization can cause an issue as the values of `min` and `max` are usually outliers. One possible solution is “clipping”, where we choose a “reasonable” value for `min` and `max`. You can also learn more about how companies apply feature engineering here.

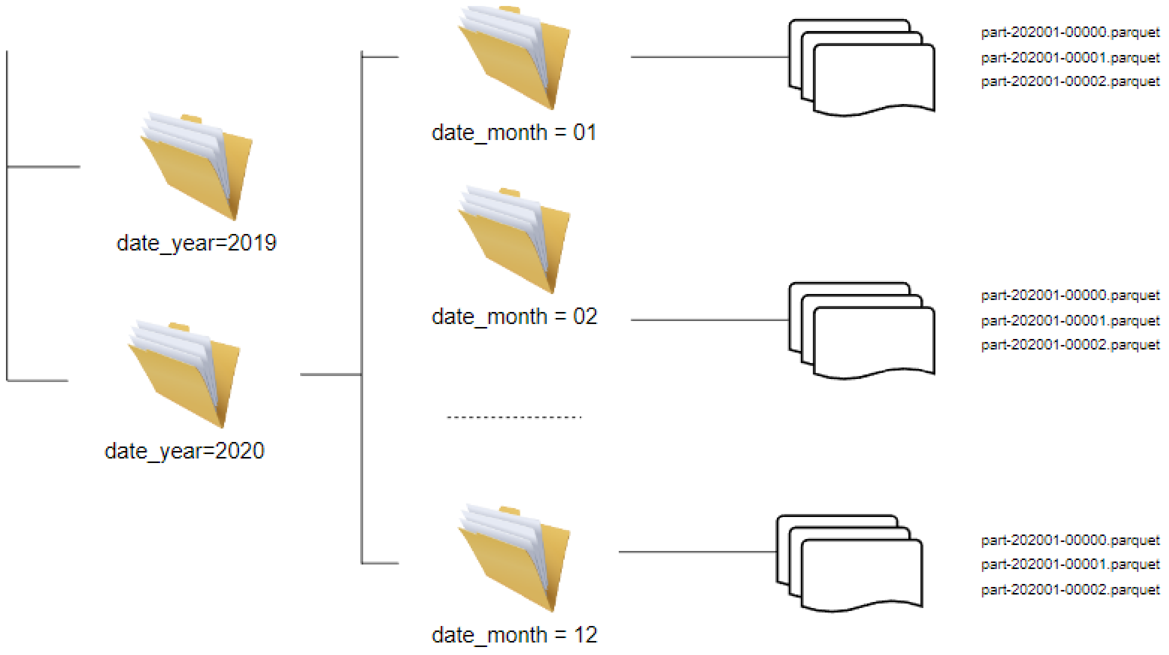
Training Pipeline

Learn common requirements and patterns in building training pipelines.

- A training pipeline needs to handle a large volume of data with low costs. One common solution is to store data in a column-oriented format like Parquet or ORC. These data formats enable high throughput for ML and analytics use cases. In other use cases, the tfrecord data format is widely used in the TensorFlow ecosystem.

Data partitioning#

- Parquet and ORC files usually get partitioned by time for efficiency as we can avoid scanning through the whole dataset. In this example, we partition data by year then by month. In practice, most common services on AWS, RedShift, and Athena support Parquet and ORC. In comparison to other formats like csv, Parquet can speed up the query times to be 30x faster, save 99% of the cost, and reduce the data that is scanned by 99%.


date_year=2019
date_year=2020
date_month = 01
date_month = 02
date_month = 12
part-202001-00000.parquet
part-202001-00001.parquet
part-202001-00002.parquet
part-202001-00000.parquet
part-202001-00001.parquet
part-202001-00002.parquet
part-202001-00000.parquet
part-202001-00001.parquet
part-202001-00002.parquet

Partition training data in Parquet format

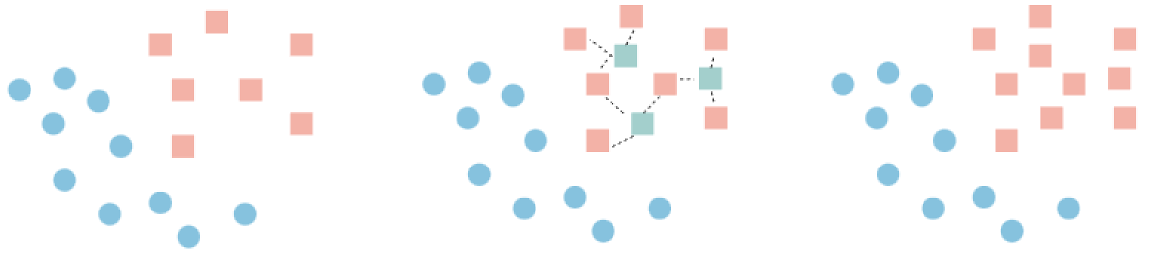
Handle imbalance class distribution#

In ML use cases like Fraud Detection, Click Prediction, or Spam Detection, it’s common to have imbalance labels. There are few strategies to handle them, i.e, you can use any of these strategies depend on your use case.

- Use **class weights in loss function**: For example, in a spam detection problem where non-spam data has 95% data compare to other spam data which has only 5%. We want to penalize more in the non-spam class. In this case, we can modify the entropy loss function using weight.

```
//w0 is weight for class 0,
w1 is weight for class 1
loss_function = -w0 *
ylog(p) - w1*(1-y)*log(1-p)
```

- Use **naive resampling**: Resample the non-spam class at a certain rate to reduce the imbalance in the training set. It’s important to have validation data and test data intact (no resampling).
- Use **synthetic resampling**: The Synthetic Minority Oversampling Technique (SMOTE) consists of synthesizing elements for the minority class, based on those that already exist. It works by randomly picking a point from the minority class and computing the k-nearest neighbors for that point. The synthetic points are added between the chosen point and its neighbors. For practical reasons, SMOTE is not as widely used as other methods.

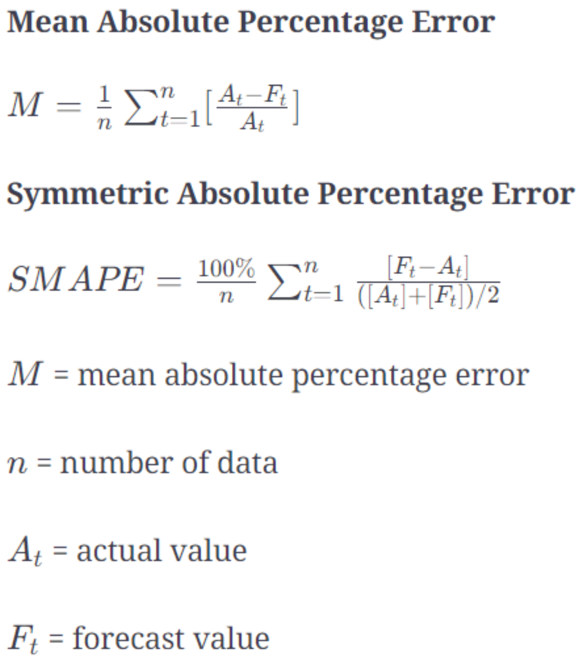


Orignal data Add synthetic samples Final data with new synthetic data

Choose the right loss function#

- It depends on the use case when deciding which loss function to use. For binary classification, the most popular is `cross-entropy`. In the Click Through Rate **(CTR)** prediction, Facebook uses Normalized Cross Entropy loss (a.k.a. `logloss`) to make the loss less sensitive to the background conversion rate.

- In a forecast problem, the most common metrics are the Mean Absolute Percentage Error (MAPE) and the Symmetric Absolute Percentage Error (SMAPE). For MAPE, you need to pay attention to whether or not your target value is skew, i.e., either too big or too small. On the other hand, SMAPE is not symmetric, as it treats under-forecast and over-forecast differently.



Mean Absolute Percentage Error

$$M = \frac{1}{n}\sum_{t=1}^{n}[\frac{A_t - F_t}{A_t}]$$

Symmetric Absolute Percentage Error

$$SMAPE = \frac{100\%}{n}\sum_{t=1}^{n}\frac{[F_t - A_t]}{([A_t]+[F_t])/2}$$

M = mean absolute percentage error

n = number of data

A_t = actual value

F_t = forecast value

- Other companies also use Machine Learning and Deep Learning for forecast problems. For example, Uber uses many different algorithms like Recurrent Neural Networks(RNNs), Gradient Boosting Trees, and Support Vector Regressors for various problems.

Some of the problems include Marketplace forecasting, Hardware capacity planning, and Marketing.

- For the regression problem, DoorDash used Quantile Loss to forecast Food Delivery demand.
- The Quantile Loss is given by:

$$L(\hat{y}, y) = max(\alpha(\hat{y} - y), (1 - \alpha)(y - \hat{y}))$$

Retraining requirements#

- Retraining is a requirement in many tech companies. In practice, the data distribution is a non-stationary process, so the model does not perform well without retraining.
- In AdTech and recommendation/personalization use cases, it's important to retrain models to capture changes in user's behavior and trending topics. So, machine learning engineers need to make the training pipeline run fast and scale well with big data. When you design such a system, you need to balance between model complexity and training time.
- A common design pattern is to use a **scheduler** to retrain models on a regular basis, usually many times per day.

There are two common schedulers: Airflow and Luigi.

Airflow

- Pros: good GUI, strong support community, independent scheduler
- Cons: less flexibility, not easy to manage massive pipeline

Luigi

- Pros: has a lot of libraries (hive, pig, google big query, redshift, etc)
- Cons: not very scalable (tight with cron), not easy to create/test tasks

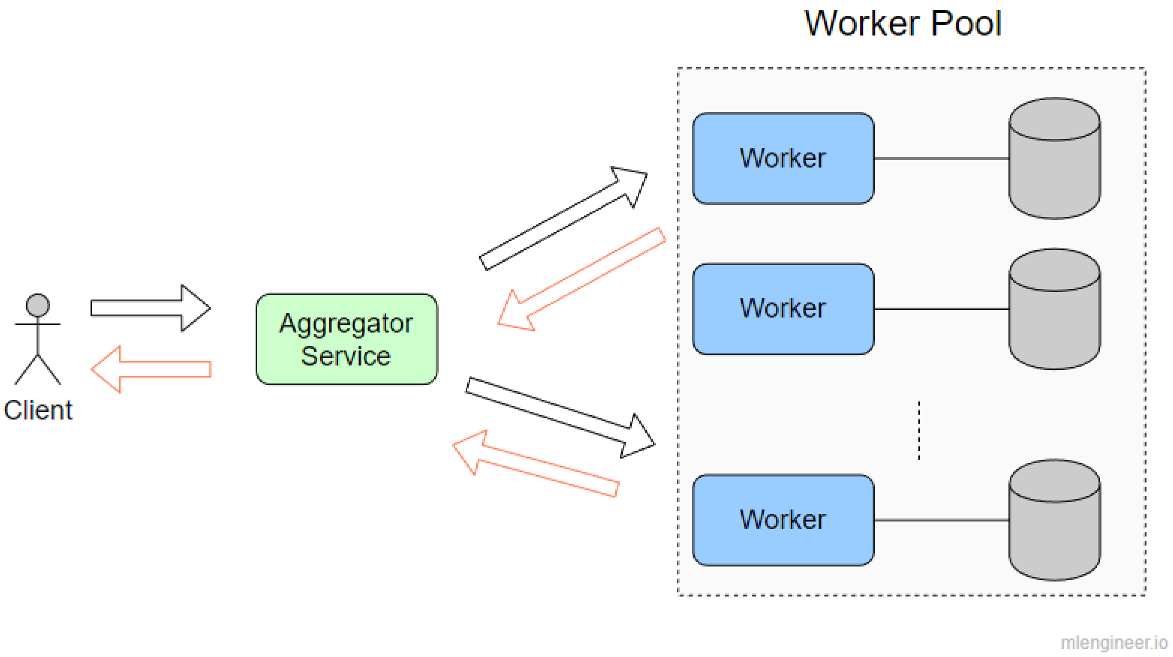
Inference

Learn common techniques to scale inference in production environments.

Inference is the process of using a trained machine learning model to make a prediction. Below are some of the techniques to scale inference in the production environment.

1. Imbalance workload#

- During inference, one common pattern is to split workloads onto multiple inference servers. We use similar architecture in Load Balancers. It is also sometimes called an Aggregator Service.



Dispatcher diagram

1. Clients (upstream process) send requests to the Aggregator Service. If the workload is too high, the Aggregator Service splits the workload and sends it to workers in the Worker pool. Aggregator Service can pick workers through one of the following ways:

   a) Work load

   b) Round Robin

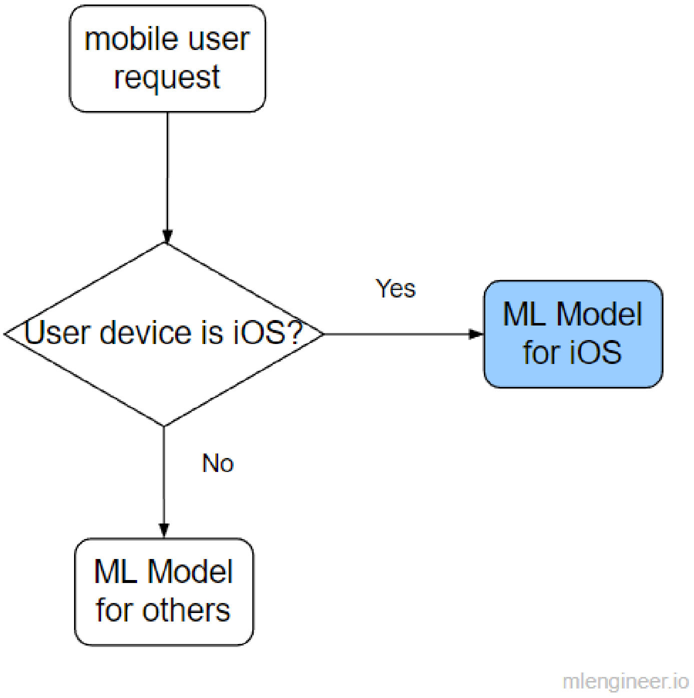
   c) Request parameter

2. Wait for response from workers.

3. Forward response to client.

Serving logics and multiple models#

- For any business-driven system, it’s important to be able to change logic in serving models. For example, in an Ad Prediction system, depending on the type of ad candidates, we will route to a different model to get a score.



Combine multiple models in Inference

2. Non-stationary problem#

- In an online setting, data is always changing. Therefore, the data distribution shift is common. So, keeping the models fresh is crucial to achieving sustained performance. Based on how frequently the model performance degrades, we can then decide how often models need to update/retrain. One common algorithm that can be used is the Bayesian Logistic Regression.

3. Exploration vs. exploitation: Thompson Sampling#

- In an Ad Click prediction use case, it’s beneficial to allow some exploration when recommending new ads. However, if there are too few ad conversions, it can reduce company revenue. This is a well-known exploration-exploitation trade-off. One common technique is Thompson Sampling where at a time, tt, we need to decide which action to take based on the reward.

Metrics Evaluation

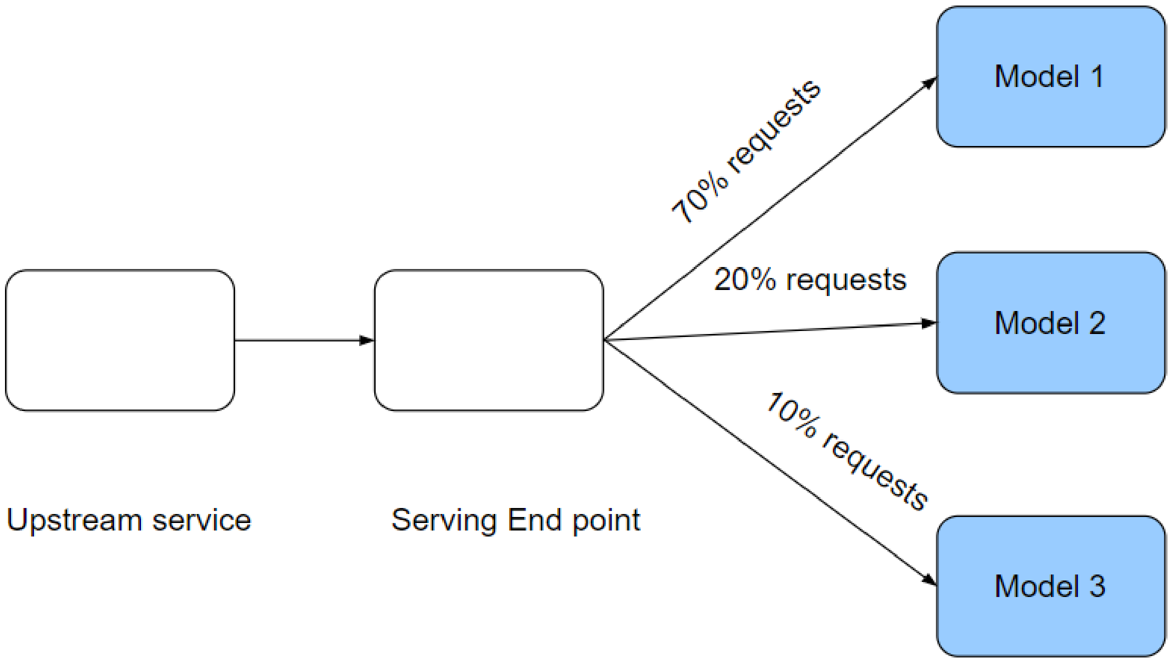
In practice, it’s common that the model performs well during offline evaluation but does not perform well when in production. Therefore, it is important to measure model performance in both on and offline environments.

Offline metrics#

- During offline training and evaluating, we use metrics like logloss, MAE, and R2 to measure the goodness of fit. Once the model shows improvement, the next step would be to move to the staging/sandbox environment to test for a small percentage of real traffic.

Online metrics#

- During the staging phase, we measure certain metrics, such as Lift in revenue or click through rate, to evaluate how well the model recommends relevant content to users. Consequently, we evaluate the impact on business metrics. If the observed revenue-related metrics show consistent improvement, then it is safe to gradually expose the model to a larger percentage of real traffic. Finally, when we have enough evidence that new models have improved revenue metrics, we can replace the current production models with new models. For further reading, explore how Sage Maker enables A/B testing or LinkedIn A/B testing.
- This diagram shows one way to allocate traffic to different models in production. In reality, there will be few a dozen models, each getting real traffic to serve online requests. This is one way to verify whether or not a model actually generates lift in the production environment.

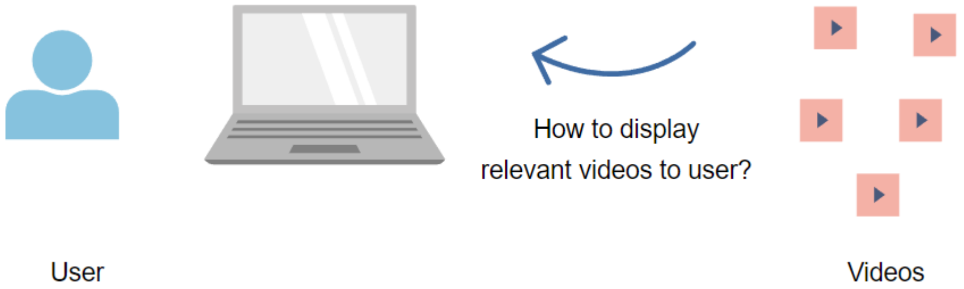


- AB testing is an extensive subject and is use case specific. Read more about A/B testing here.

Video recommendations

1. Problem statement

Build a video recommendation system for YouTube users. We want to maximize users' engagement and recommend new types of content to users.



Video recommendation system

2. Metrics design and requirements

Metrics

Offline metrics

- Use precision, recall, ranking loss, and logloss.

Online metrics

- Use A/B testing to compare Click Through Rates, watch time, and Conversion rates.

Requirements#

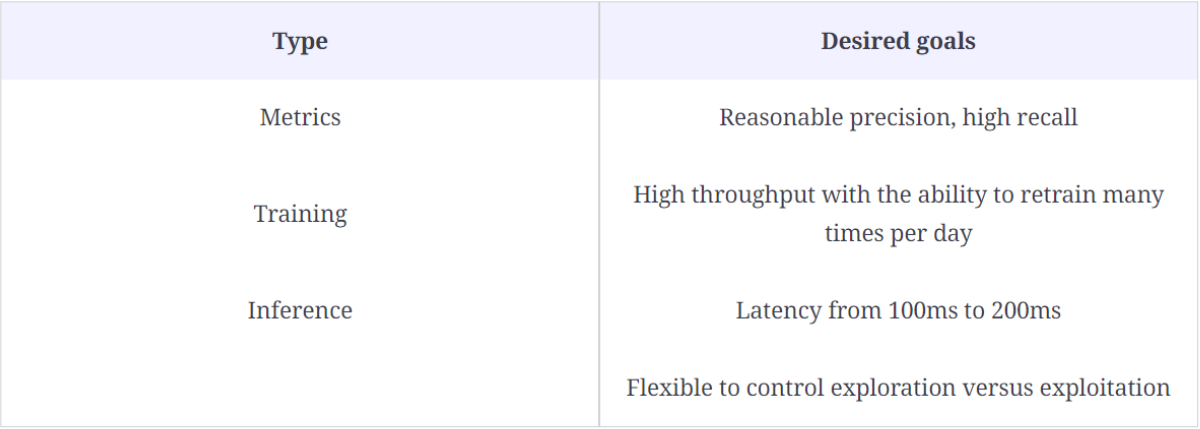
Training

- User behavior is generally unpredictable, and videos can become viral during the day. Ideally, we want to train many times during the day to capture temporal changes.

Inference

- For every user to visit the homepage, the system will have to recommend 100 videos for them. The latency needs to be under 200ms, ideally sub 100ms.
- For online recommendations, it's important to find the balance between exploration vs. exploitation. If the model over-exploits historical data, new videos might not get exposed to users. We want to balance between relevancy and fresh new content.

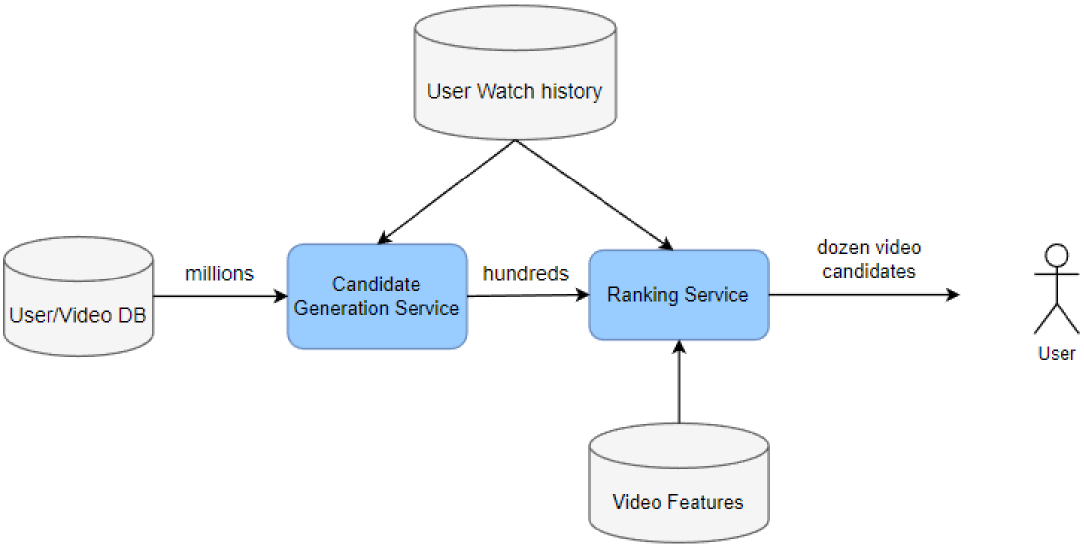
Summary



Type	Desired goals
Metrics	Reasonable precision, high recall
Training	High throughput with the ability to retrain many times per day
Inference	Latency from 100ms to 200ms Flexible to control exploration versus exploitation

Candidate Generation and Ranking Model

3. Multi-stage models



Architecture diagram for the video recommendation system

There are two stages, candidate generation, and ranking. The reason for two stages is to make the system scale.

It's a common pattern that you will see in many ML systems.

We will explore the two stages in the section below.

- The candidate model will find the relevant videos based on user watch history and the type of videos the user has watched.
- The ranking model will optimize for the view likelihood, i.e., videos that have high a watch possibility should be ranked high. It's a natural fit for the logistic regression algorithm.

Feature engineering

- Each user has a list of video watches (videos, minutes_watched).

Training data

- For generating training data, we can make a user-video watch space. We can start by selecting a period of data like last month, last 6 months, etc. This should find a balance between training time and model accuracy.

Model

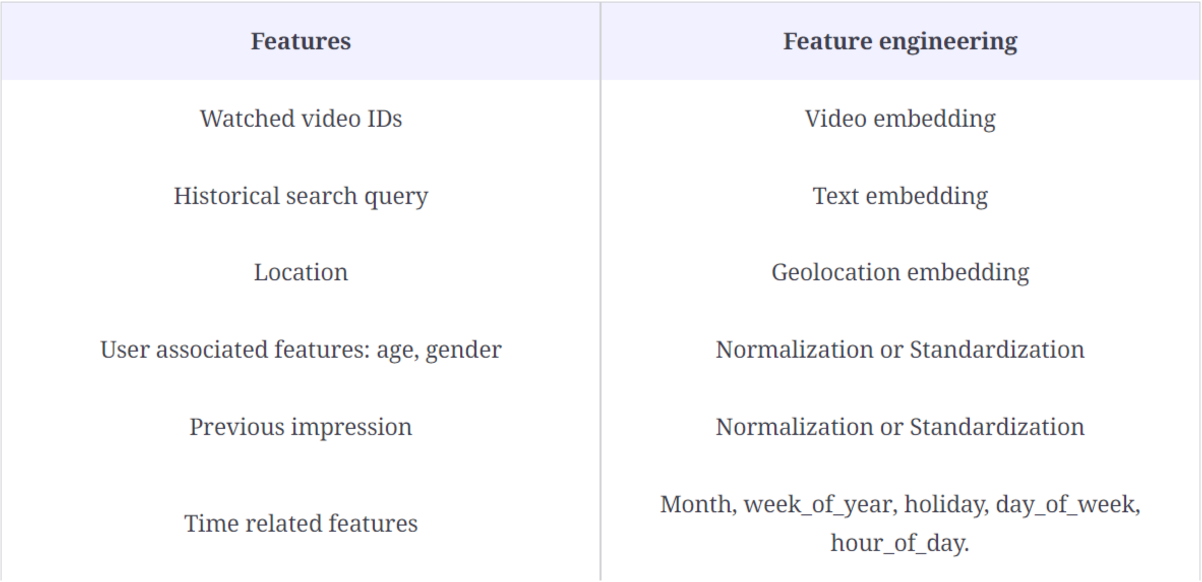
- The candidate generation can be done by Matrix factorization. The purpose of candidate generation is to generate "somewhat" relevant content to users based on their watched history. The candidate list needs to be big enough to capture potential matches for the model to perform well with desired latency.

- One solution is to use collaborative algorithms because the inference time is fast, and it can capture the similarity between user taste in the user-video space.

In practice, for large scale system (Facebook, Google), we don't use Collaborative Filtering and prefer low latency method to get candidate. One example is to leverage Inverted Index (commonly used in Lucene, Elastic Search). Another powerful technique can be found FAISS or Google ScaNN.

Ranking model#

During inference, the ranking model receives a list of video candidates given by the Candidate Generation model. For each candidate, the ranking model estimates the probability of that video being watched. It then sorts the video candidates based on that probability and returns the list to the upstream process.



Features	Feature engineering
Watched video IDs	Video embedding
Historical search query	Text embedding
Location	Geolocation embedding
User associated features: age, gender	Normalization or Standardization
Previous impression	Normalization or Standardization
Time related features	Month, week_of_year, holiday, day_of_week, hour_of_day.

Training data

- We can use User Watched History data. Normally, the ratio between watched vs. not-watched is 2/98. So, for the majority of the time, the user does not watch a video.

Model

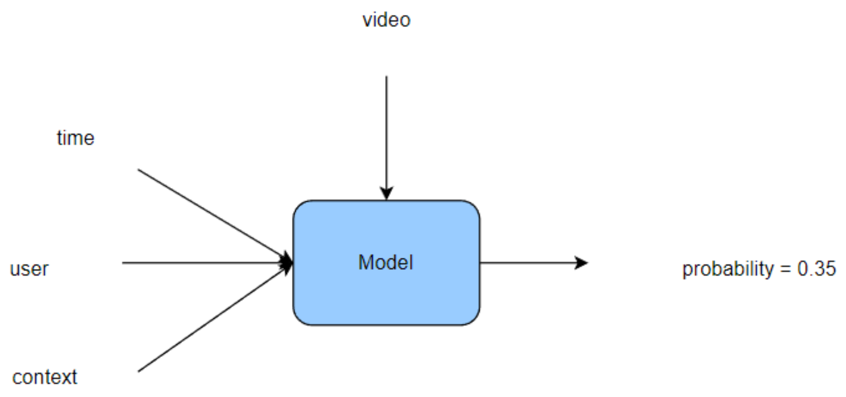
At the beginning, it’s important that we started with a simple model, as we can add complexity later.

- A fully connected neural network is simple yet powerful for representing non-linear relationships, and it can handle big data.
- We start with a fully connected neural network with **sigmoid** activation at the last layer. The reason for this is that

the Sigmoid function returns value in the range [0, 1]; therefore it's a natural fit for estimating probability.

For deep learning architecture, we can use **relu**, (Rectified Linear Unit), as an activation function for hidden layers. It's very effective in practice.

- The loss function can be cross-entropy loss.



Model prediction

Video Recommendation System Design

4. Calculation & estimation#

Assumptions#

For the sake of simplicity, we can make these assumptions:

- Video views per month are 150 billion.
- 10% of videos watched are from recommendations, a total of 15 billion videos.
- On the homepage, a user sees 100 video recommendations.
- On average, a user watches two videos out of 100 video recommendations.
- If users do not click or watch some video within a given time frame, i.e., 10 minutes, then it is a missed recommendation.
- The total number of users is 1.3 billion.

Data size#

- For 1 month, we collected 15 billion positive labels and 750 billion negative labels.
- Generally, we can assume that for every data point we collect, we also collect hundreds of features. For simplicity, each row takes 500 bytes to store. In one month, we need 800 billion rows.
- Total size 500 * 800 * 10^9 = 0.4 Petabytes. To save costs, we can keep the last six months or one year of data in the data lake, and archive old data in cold storage.

Bandwidth#

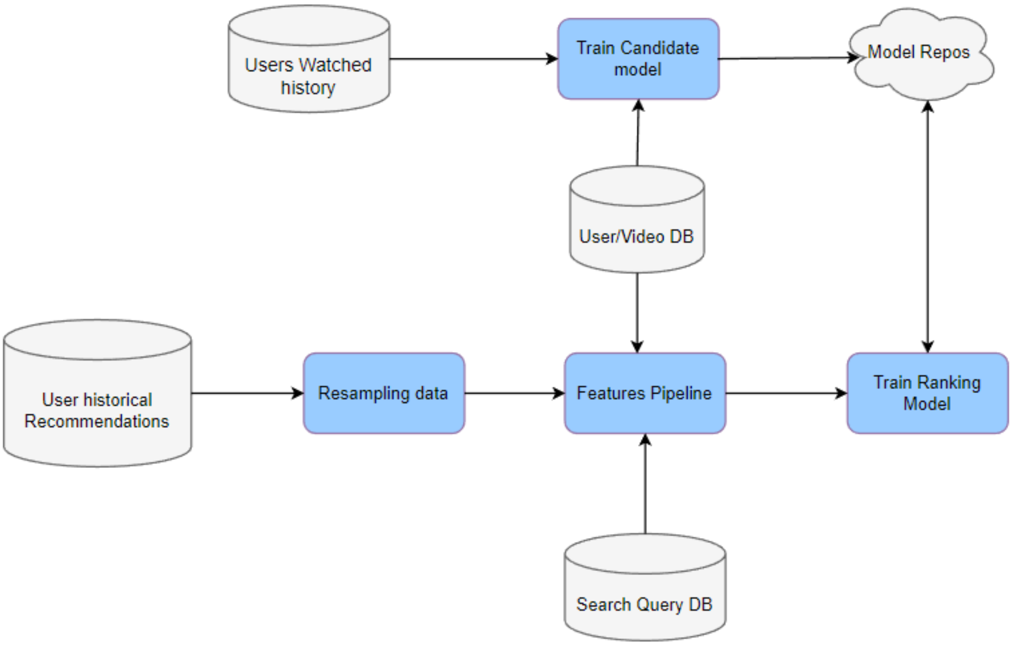
- Assume that every second we have to generate a recommendation request for 10 million users. Each request will generate ranks for 1k-10k videos.

Scale#

- Support 1.3 billion users

5. System design

High-level system design



High level system design

- Database
    - User Watched history stores which videos are watched by a particular user overtime.
    - Search Query DB stores ahistorical queries that users have searched in the past. User/Video DB stores a list of Users and their profiles along with Video metadata.
    - User historical recommendations stores past recommendations for a particular user.

- Resampling data: It's part of the pipeline to help scale the training process by down-sampling negative samples.
- Feature pipeline: A pipeline program to generate all required features for training a model. It's important for feature pipelines to provide high throughput, as we require this to retrain models multiple times. We can use Spark or Elastic MapReduce or Google DataProc.
- Model Repos: Storage to store all models, using AWS S3 is a popular option.

In practice, during inference, it's desirable to be able to get the latest model near real-time. One common pattern for the inference component is to frequently pull the latest models from Model Repos based on timestamp.

Challenges

Huge data size

- Solution: Pick 1 month or 6 months of recent data.

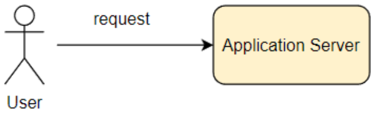
Imbalance data

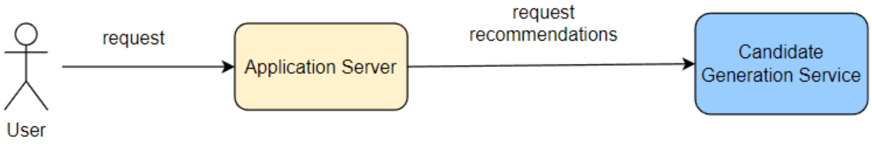
- Solution: Perform random negative down-sampling.

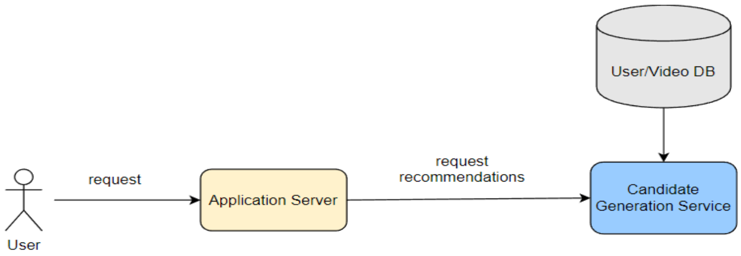
High availability

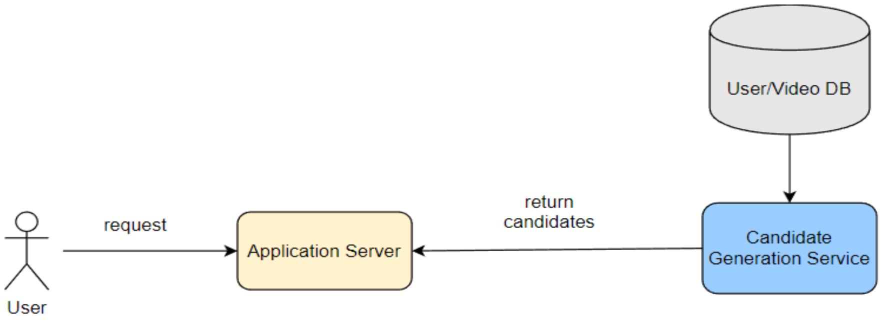
- Solution 1: Use model-as-a-service, each model will run in Docker containers.
- Solution 2: We can use Kubernetes to auto-scale the number of pods.

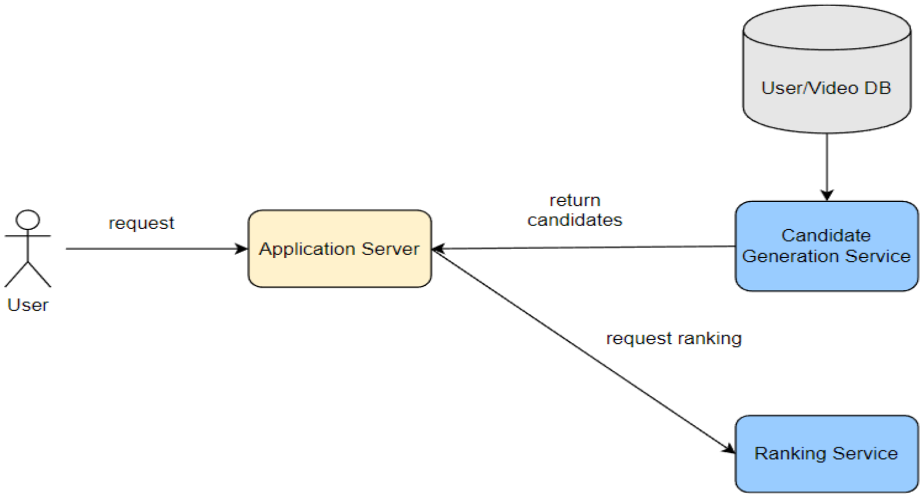
Let's examine the flow of the system:

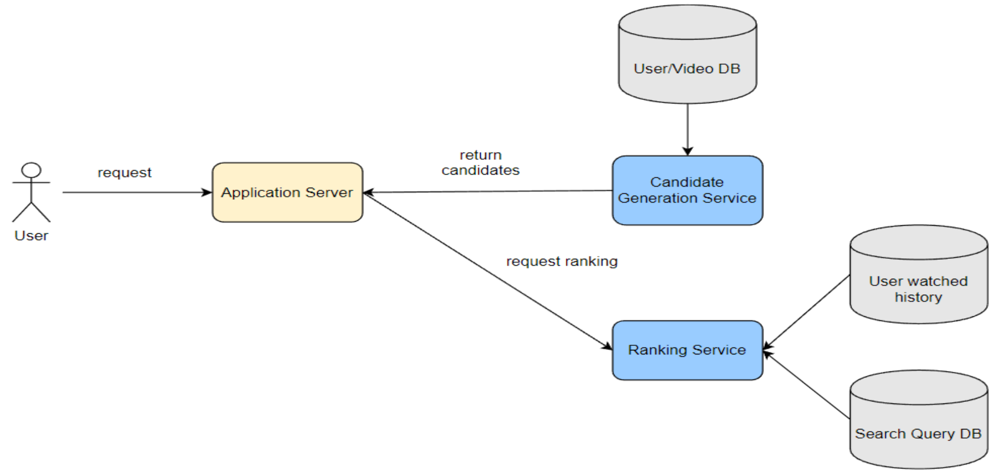

User
request
Application Server

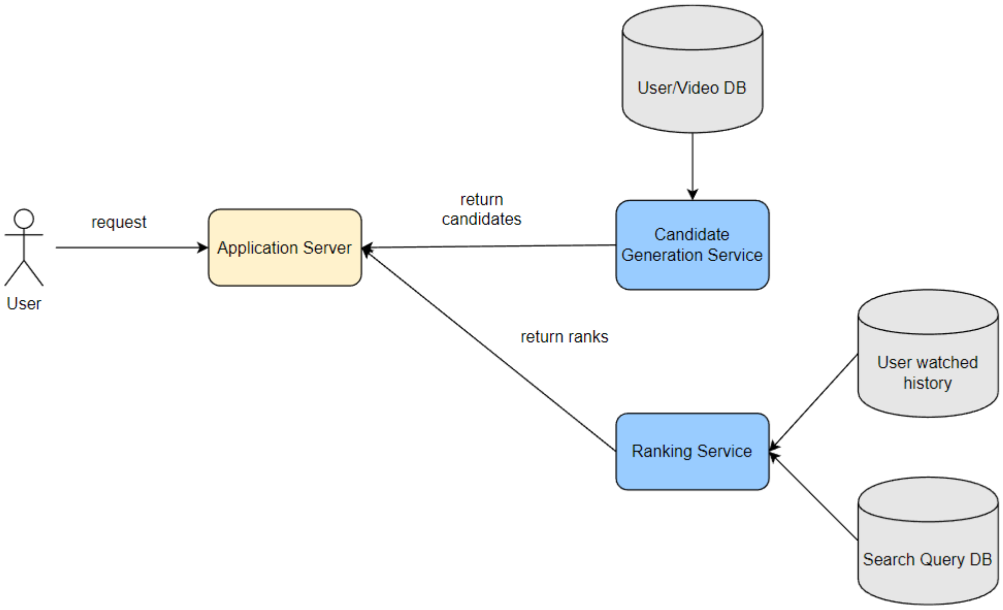

User
request
Application Server
request recommendations
Candidate Generation Service

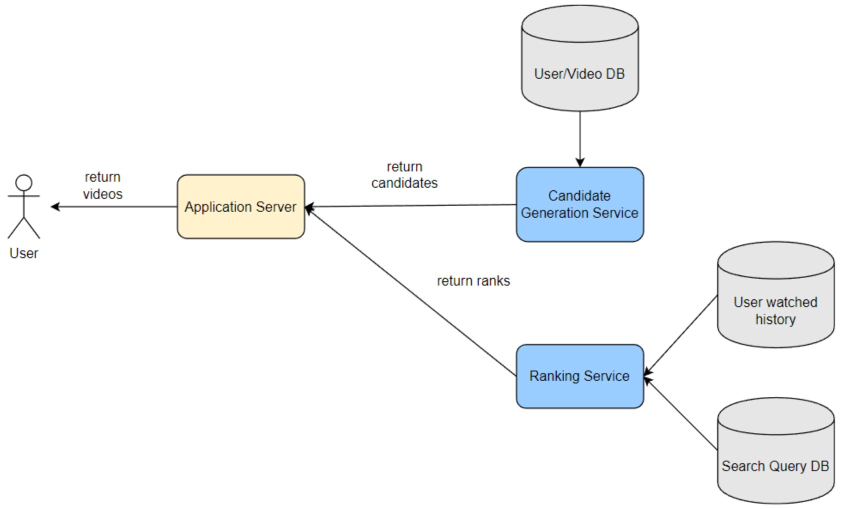

User/Video DB
User
request
Application Server
request recommendations
Candidate Generation Service


User/Video DB
User
request
Application Server
return candidates
Candidate Generation Service


User/Video DB
User
request
Application Server
return candidates
Candidate Generation Service
request ranking
Ranking Service


User/Video DB
User
request
Application Server
return candidates
Candidate Generation Service
request ranking
User watched history
Ranking Service
Search Query DB


User/Video DB
User
request
Application Server
return candidates
Candidate Generation Service
return ranks
User watched history
Ranking Service
Search Query DB


User/Video DB
User
return videos
Application Server
return candidates
Candidate Generation Service
return ranks
User watched history
Ranking Service
Search Query DB

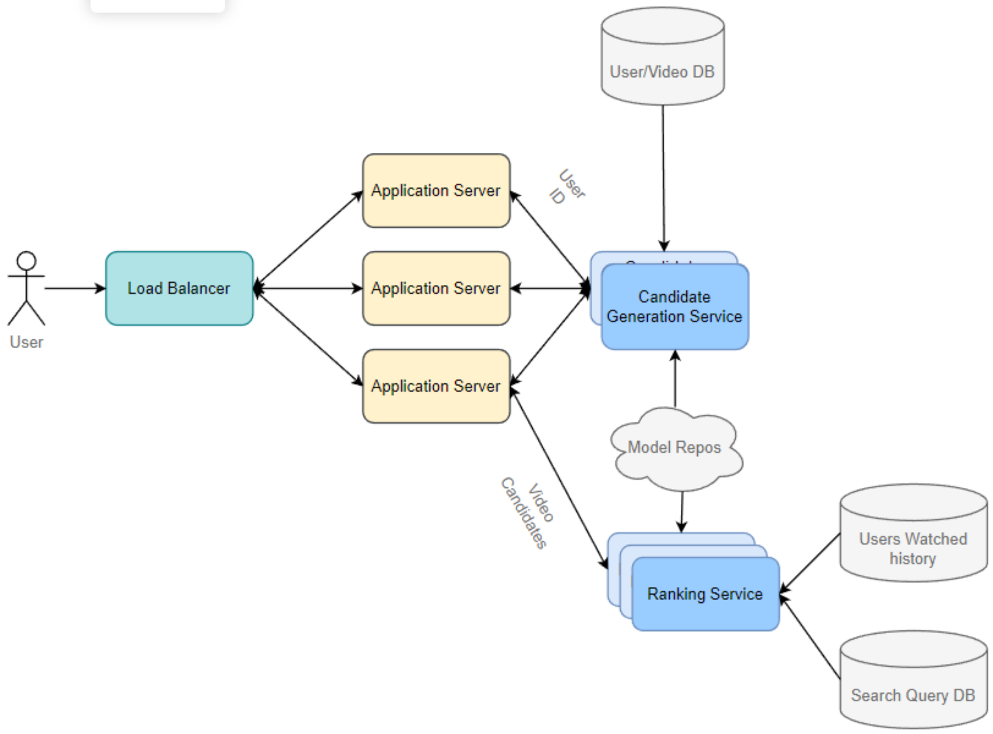
When a user requests a video recommendation, the Application Server requests Video candidates from the Candidate Generation Model. Once it receives the candidates, it then passes the candidate list to the ranking model to get the sorting order. The ranking model estimates the watch probability and returns the sorted list to the Application Server. The Application Server then returns the top videos that the user should watch.

6. Scale the design#

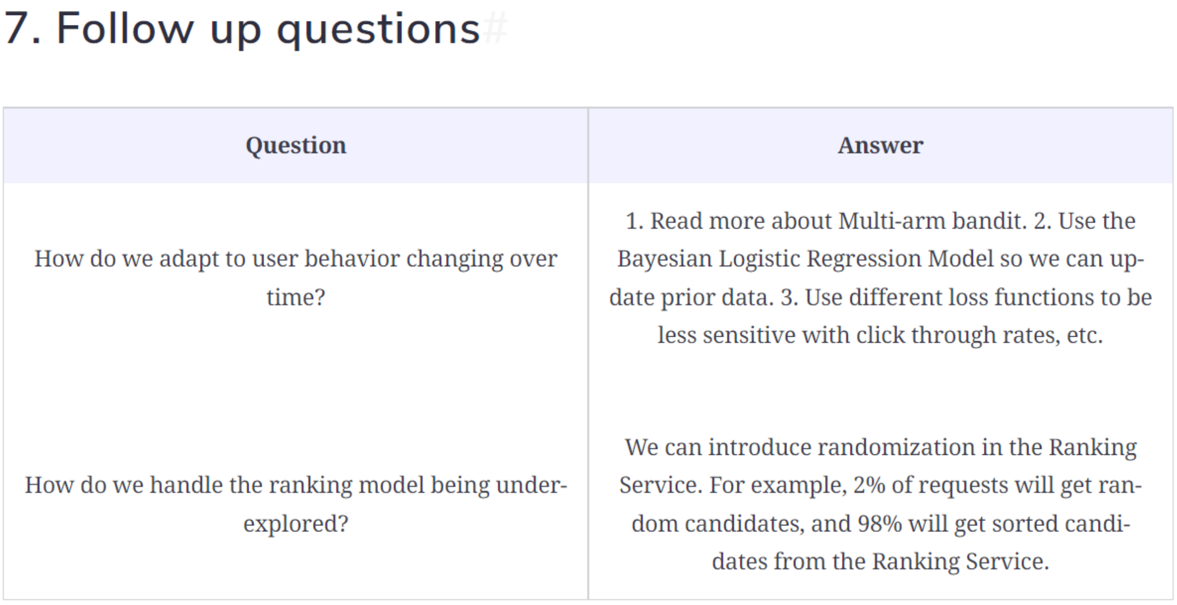
- Scale out (horizontal) multiple Application Servers and use Load Balancers to balance loads.
- Scale out (horizontal) multiple Candidate Generation Services and Ranking Services.

It's common to deploy these services in a Kubernetes Pod and take advantage of the Kubernetes Pod Autoscaler to scale out these services automatically.

In practice, we can also use `Kube-proxy` so the Candidate Generation Service can call Ranking Service directly, reducing latency even further.



Design system at scale



7. Follow up questions#

Question	Answer
How do we adapt to user behavior changing over time?	1. Read more about Multi-arm bandit. 2. Use the Bayesian Logistic Regression Model so we can update prior data. 3. Use different loss functions to be less sensitive with click through rates, etc.
How do we handle the ranking model being under-explored?	We can introduce randomization in the Ranking Service. For example, 2% of requests will get random candidates, and 98% will get sorted candidates from the Ranking Service.

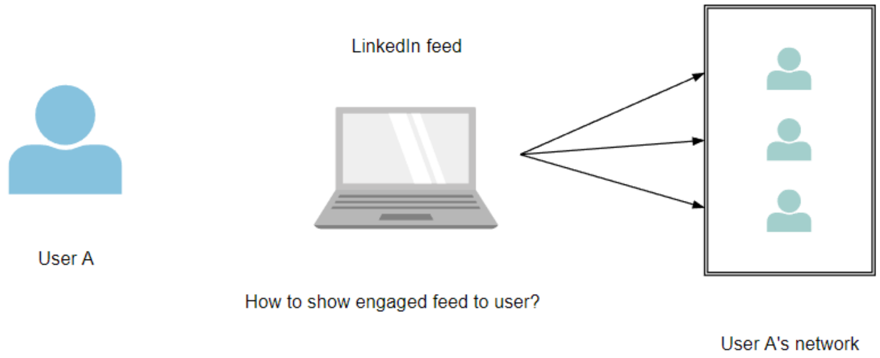
8. Summary#

- We first learned to separate Recommendations into two services: Candidate Generation Service and Ranking Service.
- We also learned about using a Deep learning fully connected layers as a baseline model and how to handle feature engineering.
- To scale the system and reduce latency, we can use `kube-flow` so that the Candidate Generation Service can communicate with the Ranking Service directly.
  -
    - You can also learn more about how companies scale their design here.

LinkedIn feed ranking

1. Problem statement

Design a personalized LinkedIn feed to maximize long-term user engagement. One way to measure engagement is user frequency, i.e, measure the number of engagements per user, but it's very difficult in practice. Another way is to measure the click probability or **Click Through Rate** (CTR).



LinkedIn feed ranking

On the LinkedIn feed, there are five major activity types:

- Connections (A connects with B)
- Informational
- Profile
- Opinion
- Site-specific
- Intuitively different activities have very different CTR. This is important when we decide to build models and generate training data.

Category	Example
Connection	Member connector follows member/company, member joins group
Informational	Member or company shares article/picture/message
Profile	Member updates profile, i.e., picture, job-change, etc.
Opinion	Member likes or comments on articles, pictures, job-changes, etc.
Site-Specific	Member endorses member, etc.

2. Metrics design and requirements#

Metrics#

Offline metrics#

- The Click Through Rate (CTR) for one specific feed is the number of clicks that feed receives, divided by the number of times the feed is shown.

$$CTR = \frac{number\_of\_clicks}{number\_of\_shown\_times}$$

- Maximizing CTR can be formalized as training a supervised binary classification model. For offline metrics, we normalize cross-entropy and AUC.
- Normalizing cross-entropy (NCE) helps the model be less sensitive to background CTR.

$$NCE = \frac{-\frac{1}{N}\sum_{i=1}^{n}(\frac{1+y_i}{2}log(p_i))+\frac{1-y_i}{2}log(1-p_i))}{-(p*log(p)+(1-p)*log(1-p))}$$

Online metrics#

- For non-stationary data, offline metrics are not usually a good indicator of performance. Online metrics need to reflect the level of engagement from users once the model has deployed, i.e., **Conversion rate** (ratio of clicks with number of feeds).

Requirements#

Training#

- We need to handle large volumes of data during training. Ideally, the models are trained in distributed settings. In social network settings, it's common to have online data distribution shift from offline training data distribution. One way to address this issue is to retrain the models (incrementally) multiple times per day.
- Personalization: Support is needed for a high level of personalization since different users have different tastes and styles for consuming their feed.
- Data freshness: Avoid showing repetitive feed on the user's home feed.

Inference#

- Scalability: The volume of users' activities are large and the LinkedIn system needs to handle 300 million users.
- Latency: When a user goes to LinkedIn, there are multiple pipelines and services that will pull data from multiple sources before feeding activities into the ranking model. All of these steps need to be done within 200ms. As a result, the Feed Ranking needs to return within **50ms**.
- Data freshness: Feed Ranking needs to be fully aware of whether or not a user has already seen any particular activity. Otherwise, seeing repetitive activity will compromise the user experience. Therefore, data pipelines need to run really fast.

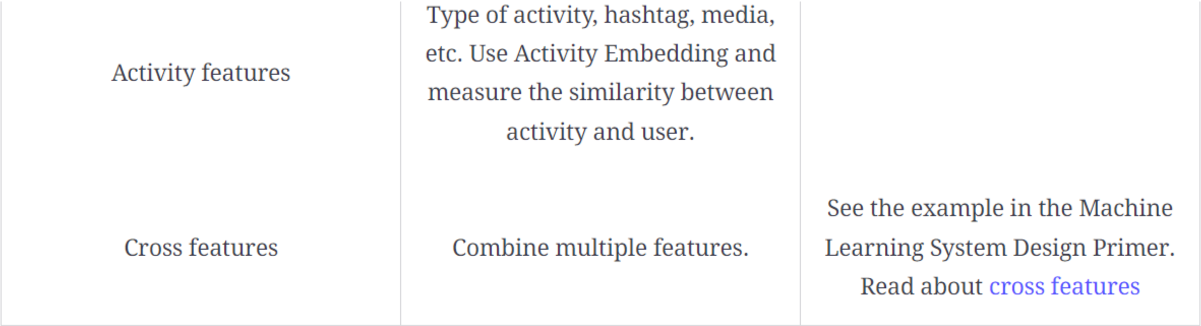
Summary#

Type	Desired goals
Metrics	Reasonable normalized cross-entropy
Training	High throughput with the ability to retrain many times per day
Inference	Supports high level of personalization
	Latency from 100ms to 200ms
	Provides a high level of data freshness and avoids showing the same feeds multiple times

3. Model#

Feature engineering#

Features	Feature engineering	Description
User profile: job title, industry, demographic, etc.	For low cardinality: Use one hot encoding. Higher cardinality: use Embedding.	
Connection strength between users		Represented by the similarity between users. We can also use Embedding for users and measure the distance vector.
Age of activity	Considered as a continuous feature or a binning value depending on the sensitivity of the Click target.	



Activity features	Type of activity, hashtag, media, etc. Use Activity Embedding and measure the similarity between activity and user.	
Cross features	Combine multiple features.	See the example in the Machine Learning System Design Primer. Read about cross features

Training data#

Before building any ML models we need to collect training data. The goal is to collect data across different types of posts, while simultaneously improving user experience. Below are some of the ways we can collect training data:

- Rank by chronicle order: This approach ranks each post in chronological order. Use this approach to collect click/not-click data. The trade-off here is serving bias because of the user's attention on the first few posts. Also, there is a data sparsity problem because different activities, such as job changes, rarely happen compared to other activities on LinkedIn.
- Random serving: This approach ranks post by random order. This may lead to a bad user experience. It also does not help with sparsity, as there is a lack of training data about rare activities.
- Use a Feed Ranking algorithm: This would rank the top feeds. Within the top feeds, you would permute randomly. Then, use the clicks for data collection. This approach provides some randomness and is helpful for models to learn and explore more activities.

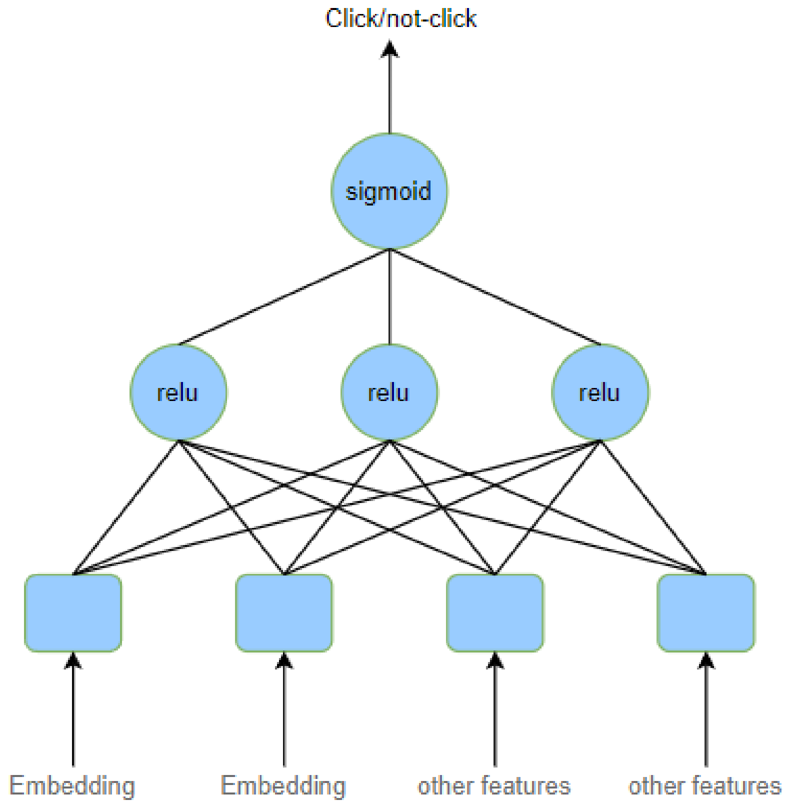
Based on this analysis, we will use an algorithm to generate training data so that we can later train a machine learning model.

We can start to use data for training by selecting a period of data: last month, last 6 months, etc. In practice, we want to find a balance between training time and model accuracy. We also downsample the negative data to handle the imbalanced data.

Model#

Selection#

- We can use a probabilistic sparse linear classifier (logistic regression). This is a popular method because of the computation efficiency that allows it to work well with sparse features.
- With the large volume of data, we need to use distributed training: Logistic Regression in Spark or Alternating Direction Method of Multipliers.
- We can also use deep learning in distributed settings. We can start with the fully connected layers with the Sigmoid activation function applied to the final layer. Because the CTR is usually very small (less than 1%), we would need to resample the training data set to make the data less imbalanced. It's important to leave the validation set and test set intact to have accurate estimations about model performance.



Multilayer perceptron

Evaluation#

- One approach is to split the data into training data and validation data. Another approach is to replay the evaluation to avoid biased offline evaluation. We use data until time t for training the model. We use test data from time t+1 and reorder their ranking based on our model during inference. If there is an accurate click prediction at the correct position, then we record a match. The total match will be considered as total clicks.
- During evaluation we will also evaluate how big our training data set should be, and how frequently we should retrain the model, among many other hyperparameters.

4. Calculation & estimation#

Assumptions#

- 300 million monthly active users
- On average, a user sees 40 activities per visit. Each user visits 10 times per month.
- We have 12 * 10^10 or 120 billion observations/samples

Data size#

- Assume the click through rate is about 1% for 1 month. We collected 1 billion positive labels and about 110 billion negative labels. This is a huge dataset.

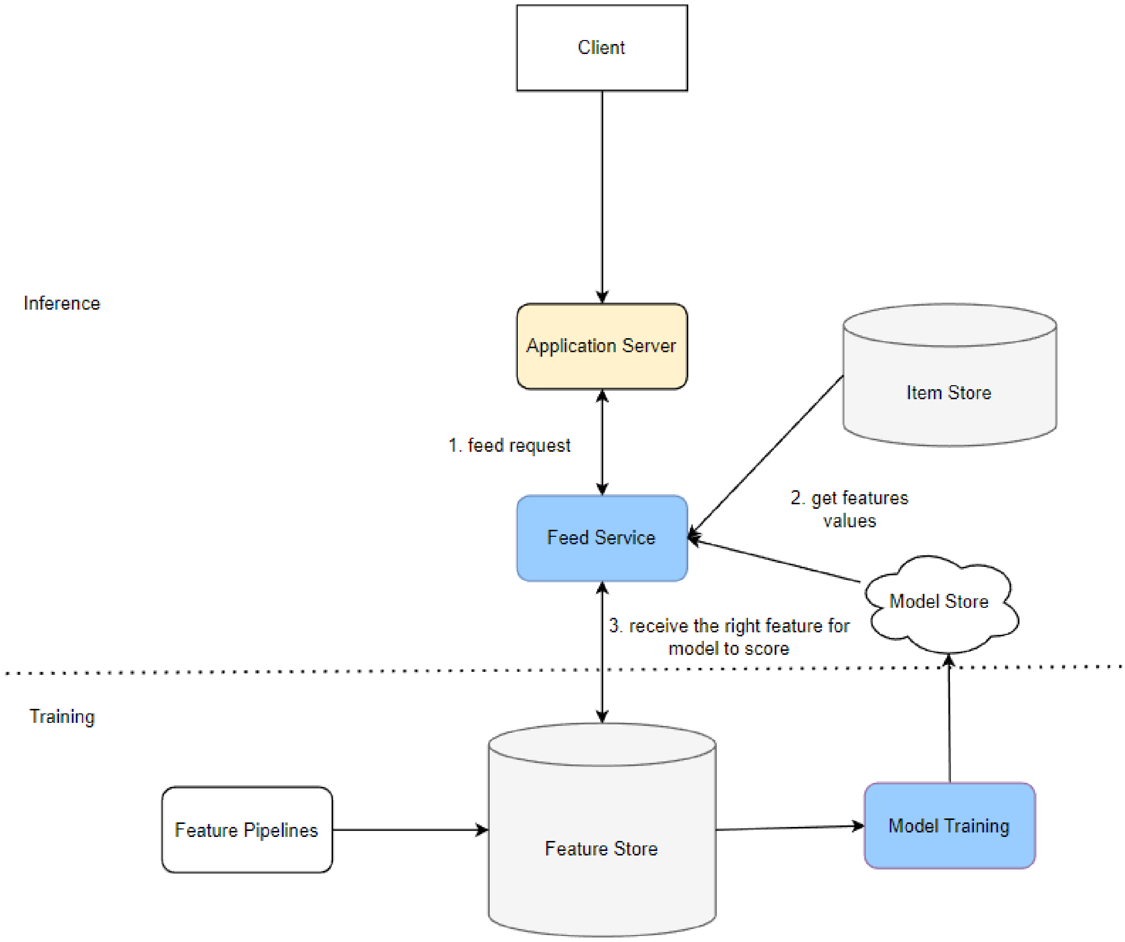
- Generally, we can assume that for every data point, we collect hundreds of features. For simplicity, each row takes 500 bytes to store.

- In one month, we need 120 billion rows. Total size: 500 * 120 * 10^{9} = 60 * 10^{12} bytes = 60 Terabytes. To save costs we can keep the last 6 months or 1 year of data in the data lake and archive old data in cold storage.

Scale#

- Supports 300 million users

5. High-level design#

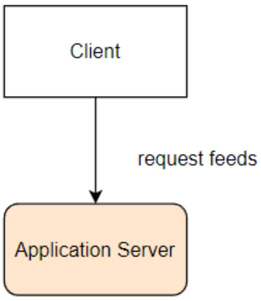


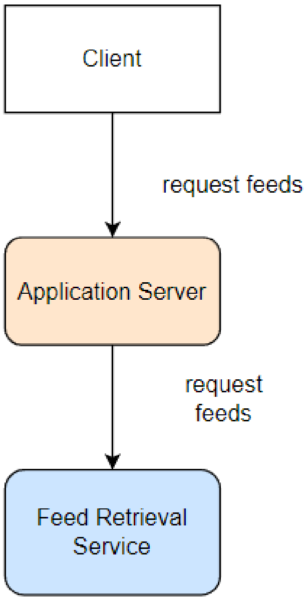
Feed Ranking High level design

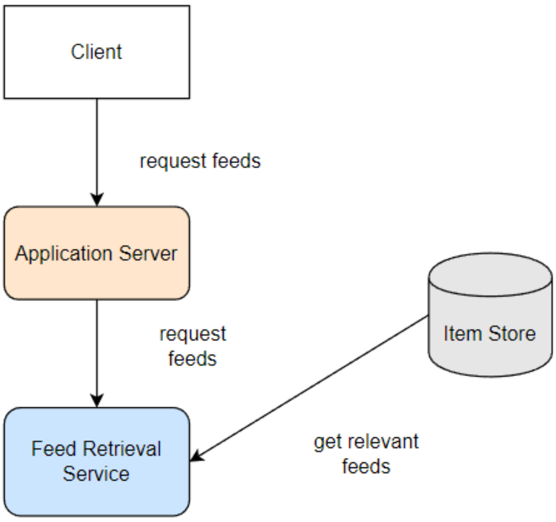
- **Feature store** is feature values storage. During inference, we need low latency (<10ms) to access features before scoring. Examples of feature stores include MySQL Cluster, Redis, and DynamoDB.

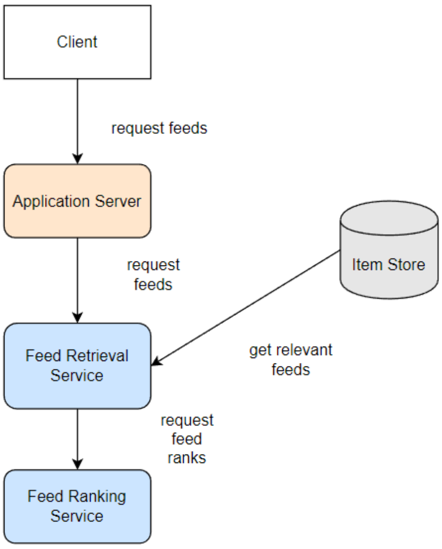
- **Item store** stores all activities generated by users. It also stores models for the corresponding users. One goal is to maintain the consistent user experience, i.e., to use the same feed ranking method for any particular user. Item store provides the correct model for the corresponding users.

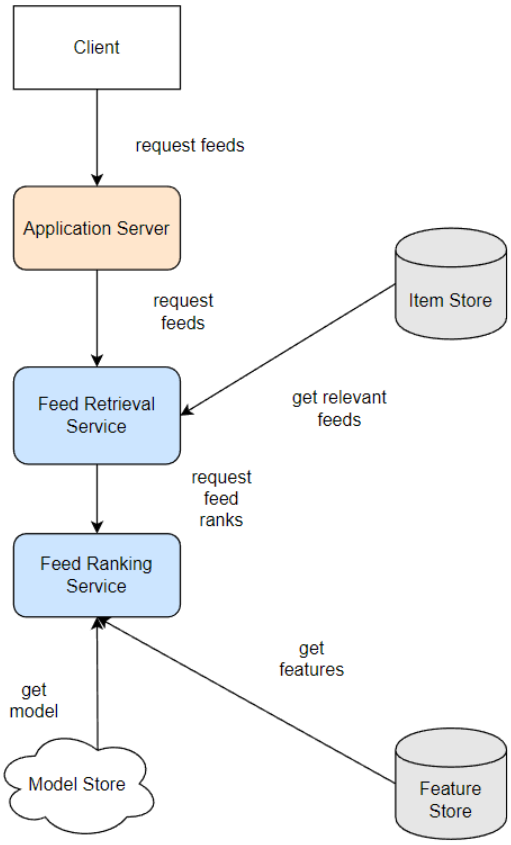
Let’s examine the flow of the system:

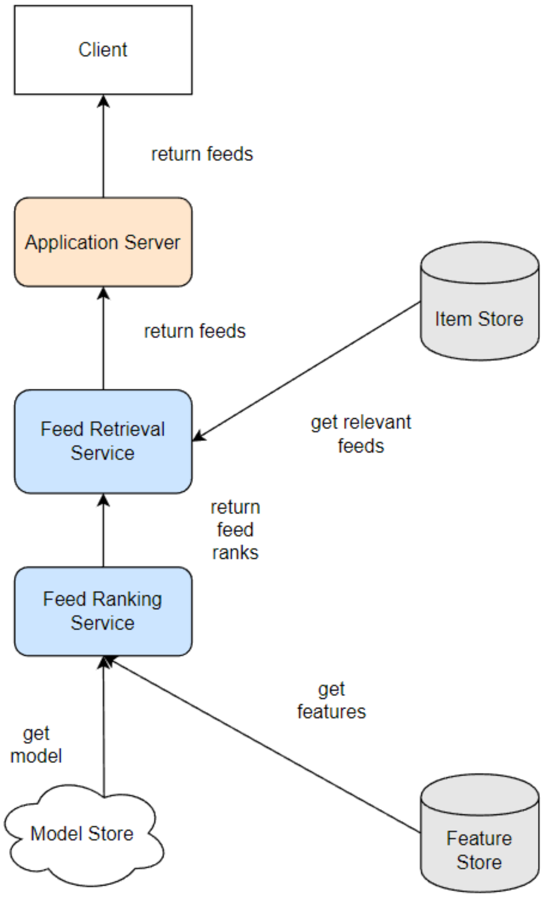







Client
request feeds
Application Server
request feeds
Item Store
get relevant feeds
Feed Retrieval Service
request feed ranks
Feed Ranking Service


Client
request feeds
Application Server
request feeds
Item Store
get relevant feeds
Feed Retrieval Service
request feed ranks
Feed Ranking Service
get features
get model
Model Store
Feature Store



Step by step:

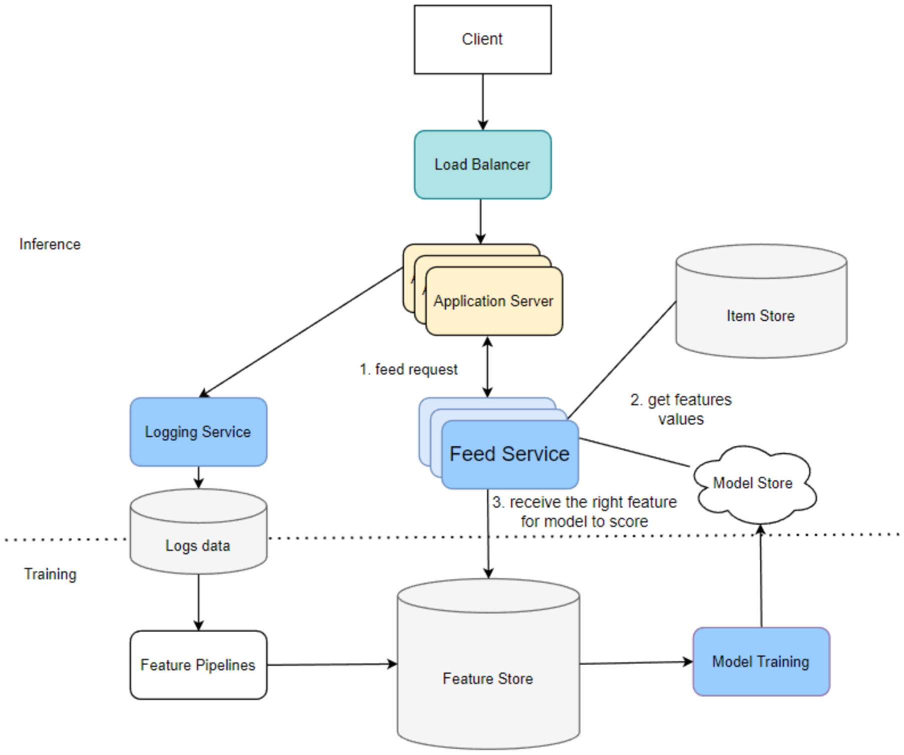
- Client sends feed request to Application Server
- Application Server sends feed request to Feed Retrieval Service
- Feed Retrieval Service selects most relevant feeds from Item Store
- Feed Retrieval Service sends feed ranks request to Feed Ranking Service
- Feed Ranking Service gets the latest ML model, gets the right features from Feature Store.
- Feed Ranking Service scores each feeds and returns to Feed Retrieval Service and Application Server. Application server sorts feeds by rankings and return to client.

- A user visits the LinkedIn homepage and requests an Application Server for feeds. The Application Server sends feed requests to the Feed Service.

- Feed Service gets the latest model from Model Repos, gets the correct features from the Feature Store, and all the feeds from the ItemStore. Feed Service will provide features for the Model to get predictions.
- The Model returns recommended feeds sorted by click through rate likelihood.

6. Scale the design#

- Scale out the Feed Service module as it represents both Retrieval Service and Ranking Service. This provides better visualization.
- Scale out the Application Server and put the Load Balancer in front of the Application Server to balance load.



Feed Ranking Service at scale

7. Summary#

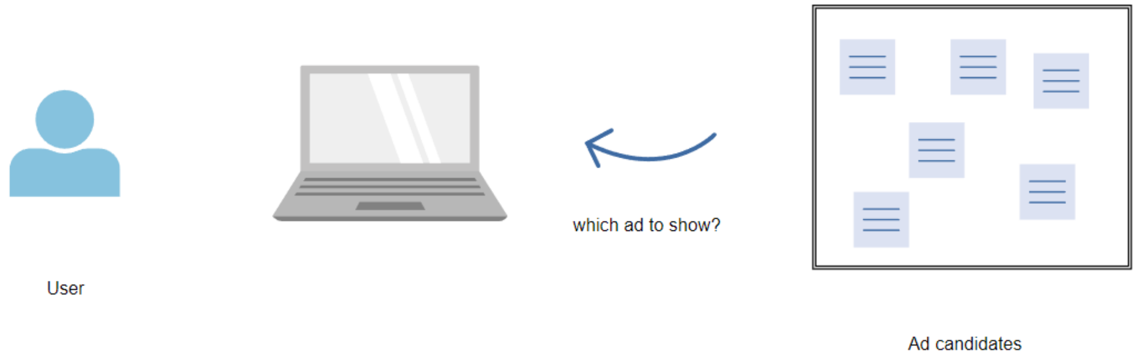
- We learned how to build Machine Learning models to rank feeds. The binary classification model with custom loss function helps the model be less sensitive to background click through rate.
- We learned how to create the process to generate training data for the Machine Learning Model.
- We learned how to scale training and inference by scaling out the Application Server and Feed Services.
- You can also learn more about how companies scale there design here.

Ad click prediction

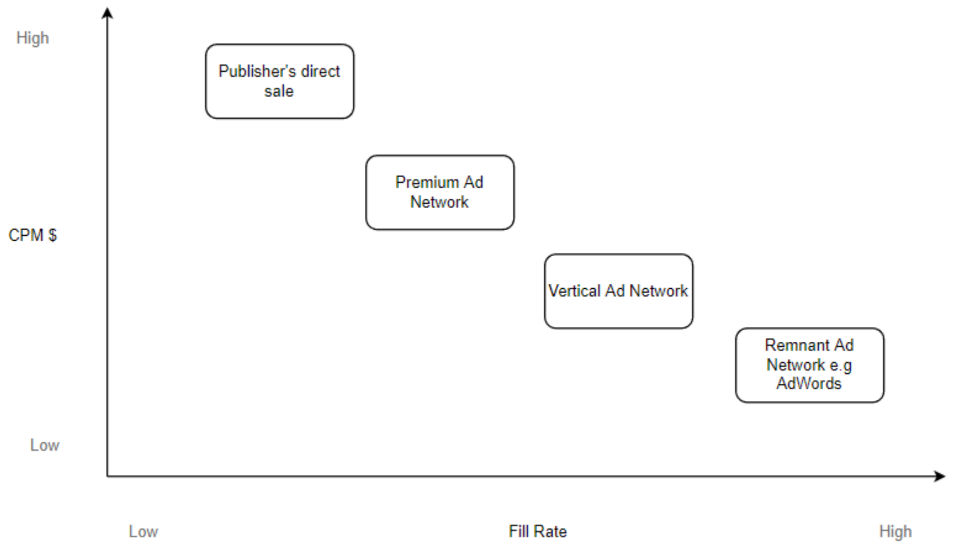
1. Problem statement

Build a machine learning model to predict if an ad will be clicked.

For the sake of simplicity, we will not focus on the cascade of classifiers that is commonly used in AdTech.



- Let's understand the ad serving background before moving forward. The ad request goes through a waterfall model where publishers try to sell its inventory through direct sales with high CPM (Cost Per Million). If it is unable to do so, the publishers pass the impression to other networks until it is sold.



Waterfall Revenue Model

2. Metrics design and requirements#

Metrics#

During the training phase, we can focus on machine learning metrics instead of revenue metrics or CTR metrics. Below are the two metrics:

Offline metrics#

- Normalized Cross-Entropy (NCE): NCE is the predictive `logloss` divided by the `cross-entropy` of the background CTR. This way NCE is insensitive to background CTR. This is the NCE formula:

$$NCE = \frac{-\frac{1}{N}\sum_{i=1}^{n}(\frac{1+y_i}{2}log(p_i))+\frac{1-y_i}{2}log(1-p_i))}{-(p*log(p)+(1-p)*log(1-p))}$$

Online metrics#

- Revenue Lift: Percentage of revenue changes over a period of time. Upon deployment, a new model is deployed on a small percentage of traffic. The key decision is to balance between percentage traffic and the duration of the A/B testing phase.

Requirements#

Training#

- Imbalance data: The Click Through Rate (CTR) is very small in practice (1%-2%), which makes supervised training difficult. We need a way to train the model that can handle highly imbalanced data.
- Retraining frequency: The ability to retrain models many times within one day to capture the data distribution shift in the production environment.
- Train/validation data split: To simulate a production system, the training data and validation data is partitioned by time.

Inference#

- Serving: Low latency (50ms - 100ms) for ad prediction.
- Latency: Ad requests go through a waterfall model, therefore, recommendation latency for ML model needs to be fast.
- Overspent: If the ad serving model repeatedly serves the same ads, it might end up over-spending the campaign budget and publishers lose money.

Summary#

Type	Desired goals
Metrics	Reasonable normalized cross-entropy and click through rate
Training	Ability to handle imbalance data
	High throughput with the ability to retrain many times per day
Inference	Latency from 50 to 100ms
	Ability to control or avoid overspent campaign budget while serving ads

3. Model#

Feature engineering#

Features	Feature engineering	Description
AdvertiserID	Use Embedding or feature hashing	It's easy to have millions of advertisers
User's historical behavior, i.e., numbers of clicks on ads over a period of time.	Feature scaling, i.e., normalization	
Temporal: time_of_day, day_of_week etc	One hot encoding	
Cross features	Combine multiple features	See example in the Machine Learning System Design Primer

Training data#

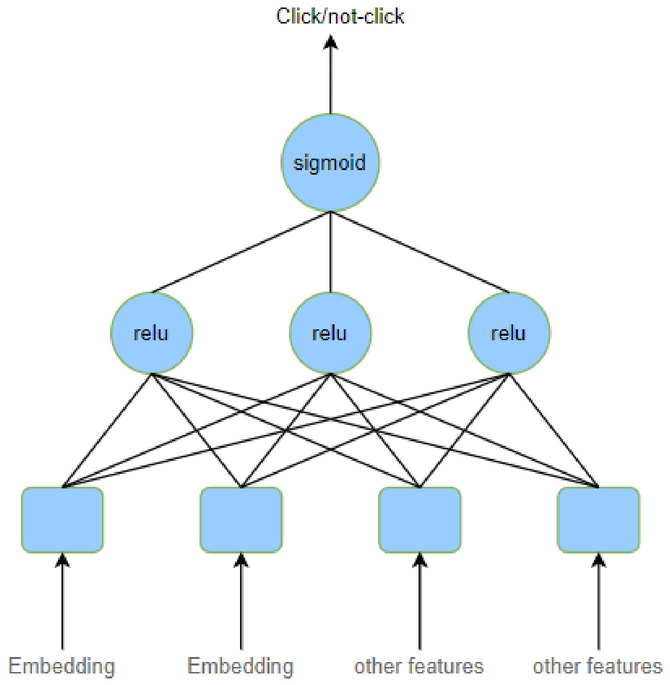
Before building any ML models we need to collect training data. The goal here is to collect data across different types of posts while simultaneously improving the user experience. As you recall from the previous lesson about the waterfall model, we can collect a lot of data about ad clicks. We can use this data for training the Ad Click model.

We can start to use data for training by selecting a period of data: last month, last six months, etc. In practice, we want to find a balance between training time and model accuracy. We also downsample the negative data to handle the imbalanced data.

Model#

Selection#

- We can use deep learning in distributed settings. We can start with fully connected layers with the Sigmoid activation function applied to the final layer. Because the CTR is usually very small (less than 1%), we would need to resample the training data set to make the data less imbalanced. It's important to leave the validation and test sets intact to have accurate estimations about model performance.



DL fully connected model

Evaluation#

- One approach is to split the data into training data and validation data. Another approach is to replay evaluation to avoid biased offline evaluation. Assume the training data we have up until time `t`. We use test data from time `t+1` and reorder their ranking based on our model during inference. If there is an accurate click prediction, we record a match. The total match will be considered as total clicks.
- During evaluation we will also evaluate how big our training data set should be and how frequently we need to retrain the model among many other hyperparameters.

4. Calculation and estimation#

Assumptions#

- 40K ad requests per second or 100 billion ad requests per month
- Each observation (record) has hundreds of features, and it takes 500 bytes to store.

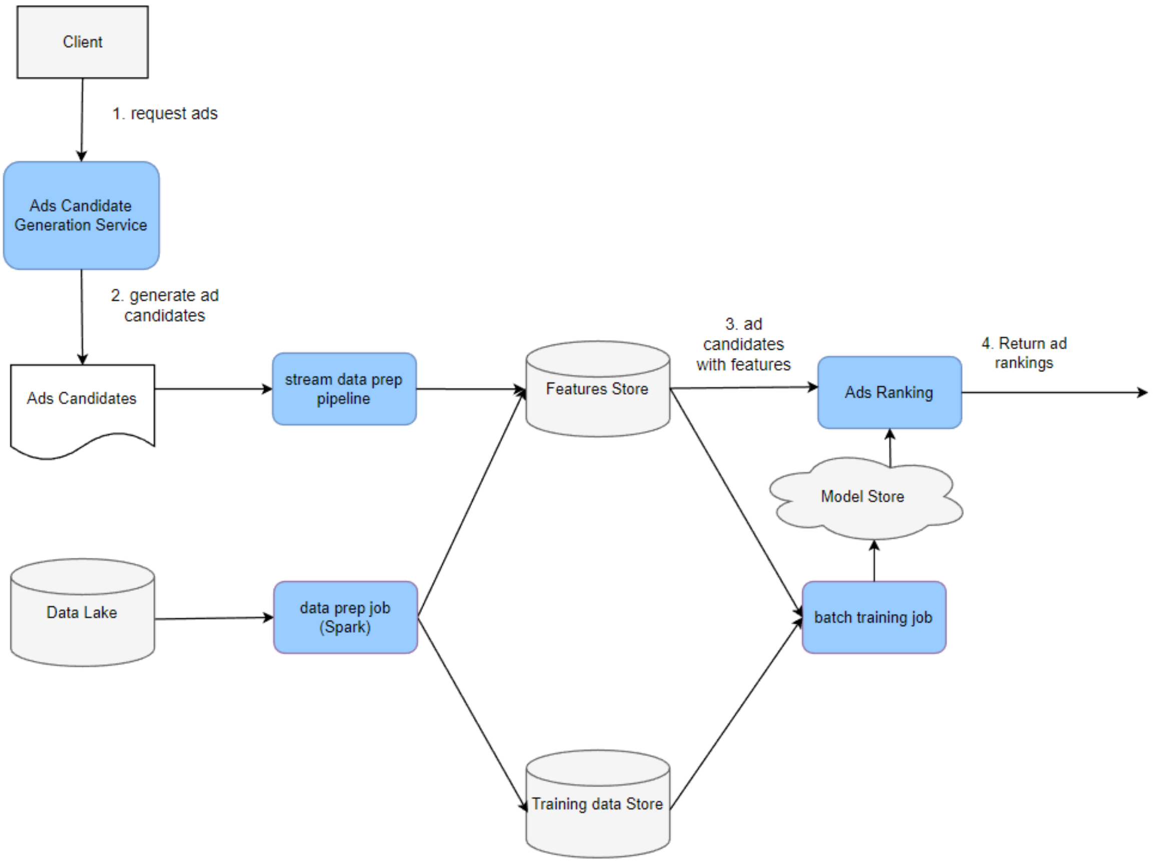
Data size#

- Data: historical ad click data includes [user, ads, click_or_not]. With an estimated 1% CTR, it has 1 billion clicked ads. We can start with 1 month of data for training and validation. Within a month we have, 100 * 10^{12} * 500 = 5 * 10^{16} bytes or 50 PB. One way to make it more manageable is to downsample the data, i.e., keep only 1%-10% or use 1 week of data for training data and use the next day for validation data.

Scale#

- Supports 100 million users

4. High level design#

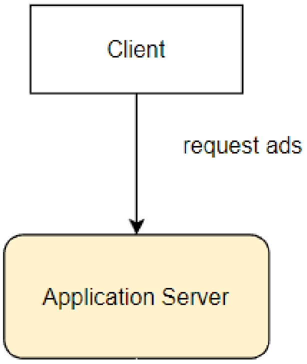


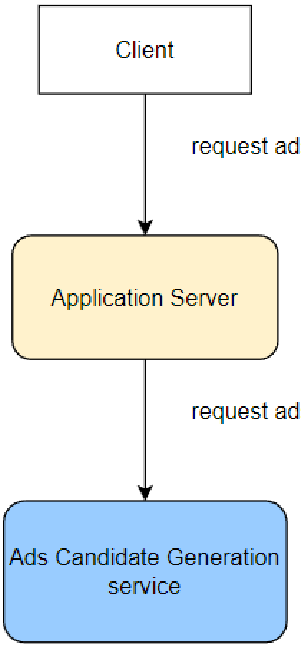
AdClick Prediction high level design

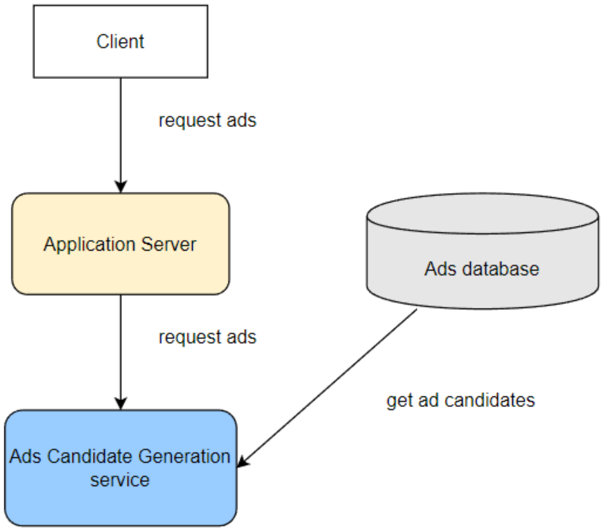
- Data lake: Store data that is collected from multiple sources, i.e., logs data or event-driven data (Kafka)
- Batch data prep: Collections of ETL (Extract, Transform, and Load) jobs that store data in Training data Store.
- Batch training jobs organize scheduled jobs as well as on-demand jobs to retrain new models based on training data storage.
- Model Store: Distributed storage like S3 to store models.
- Ad Candidates: Set of Ad candidates provided by upstream services (refer back to waterfall model).
- Stream data prep pipeline: Processes online features and stores features in key-value storage for low latency down-stream processing.

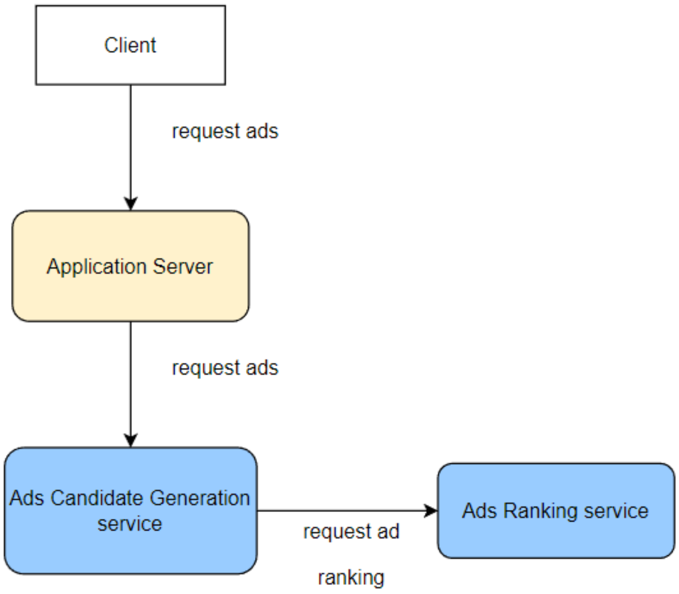
- Model Serving: Standalone service that loads different models and provides Ad Click probability.

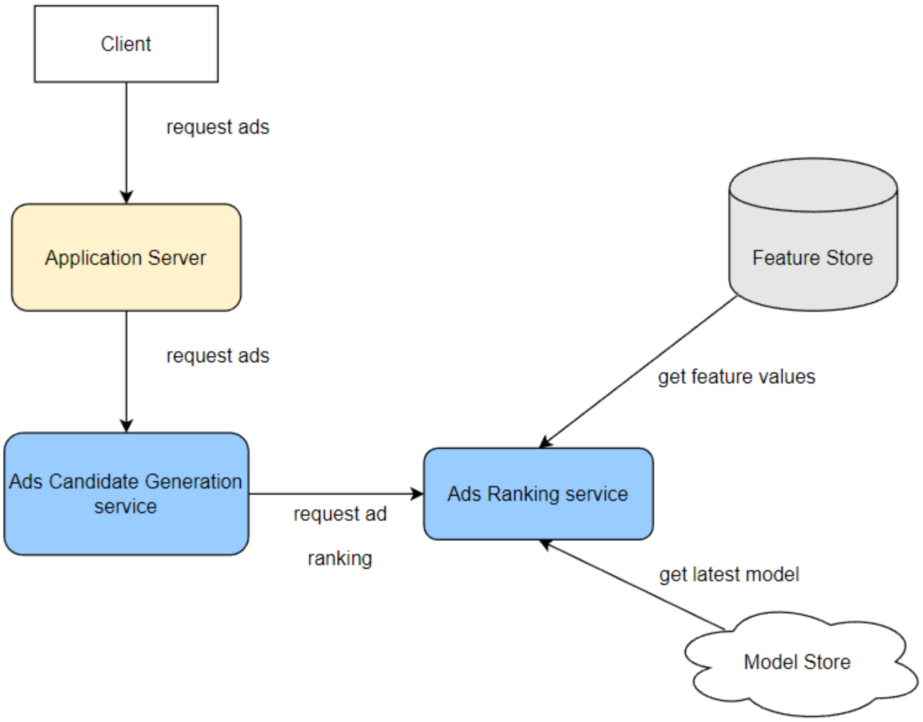
Let’s examine the flow of the system:

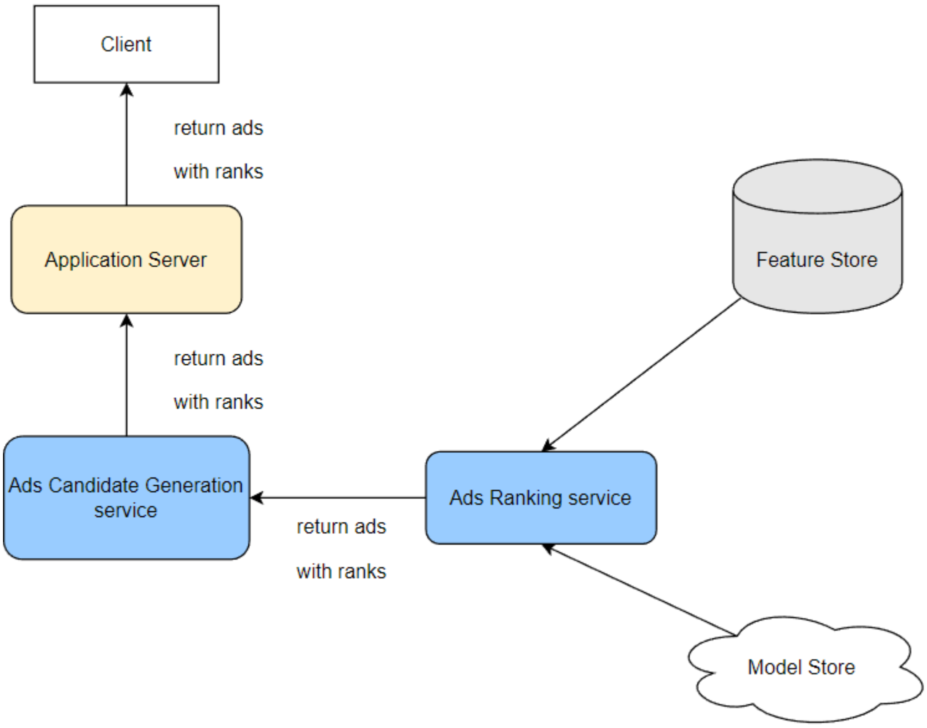





Client
request ads
Application Server
Ads database
request ads
get ad candidates
Ads Candidate Generation service


Client
request ads
Application Server
request ads
Ads Candidate Generation service
request ad ranking
Ads Ranking service


Client
request ads
Application Server
request ads
Feature Store
get feature values
Ads Candidate Generation service
request ad ranking
Ads Ranking service
get latest model
Model Store

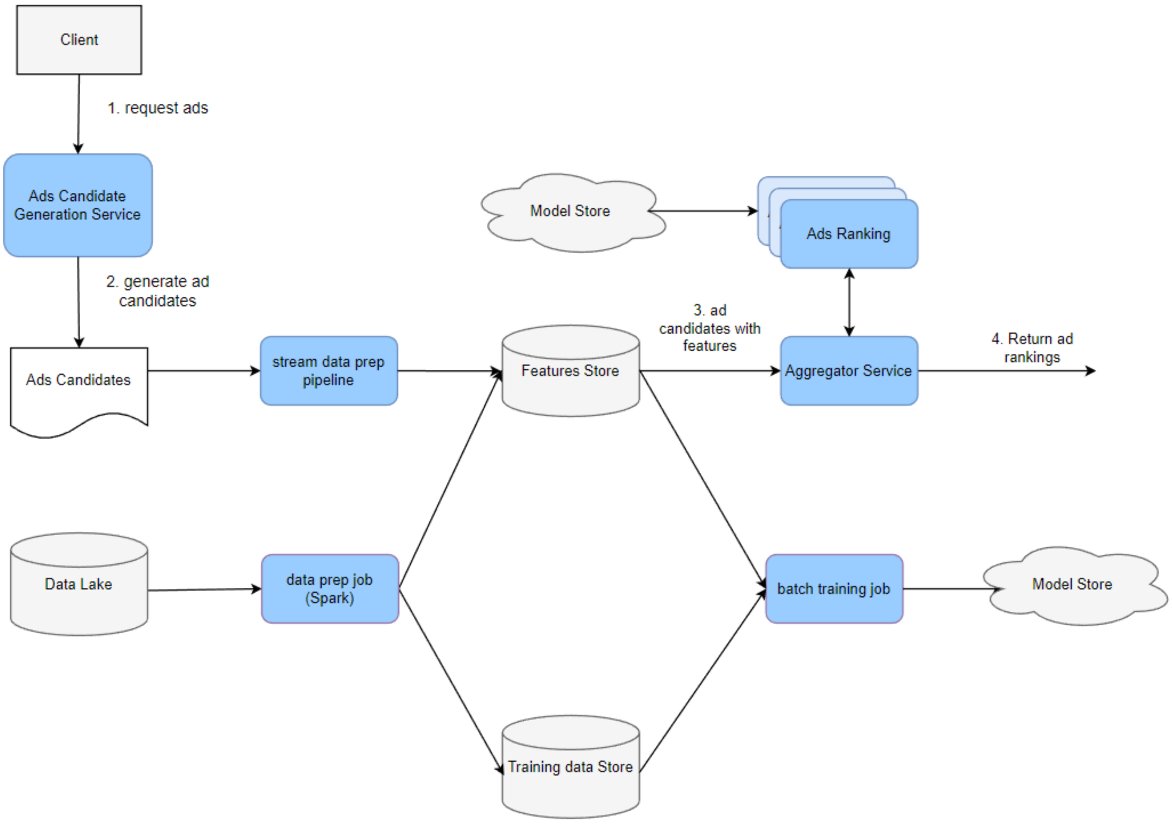

Client
return ads with ranks
Application Server
Feature Store
return ads with ranks
Ads Candidate Generation service
return ads with ranks
Ads Ranking service
Model Store

Step by Step

- Client sends an ad request to Application Server
- Application Server sends ad request to Ads Candidate Generation service
- Ads Candidate Generation Service generates ad candidates from database
- Ads Candidate Generation Service sends ad ranking request to Ads Ranking service
- Ads Ranking service gets feature values from Feature Store, gets latest model
- Ads Candidate Generation Service score ads and return ads with ranking to Ads Candidate Generation service. Ads Candidate Generation service return result to Application server and client.

- User visits the homepage and sends an Ad request to the Candidate Generation Service. Candidate Generation Service generates a list of Ads Candidates and sends them to the Aggregator Service.
- The Aggregator Service splits the list of candidates and sends it to the Ad Ranking workers to score.
- Ad Ranking Service gets the latest model from Model Repos, gets the correct features from the Feature Store, produces ad scores, then returns the list of ads with scores to the Aggregator Service.
- The Aggregator Service selects top K ads (For example, k = 10, 100, etc.) and returns to upstream services.

6. Scale the design#

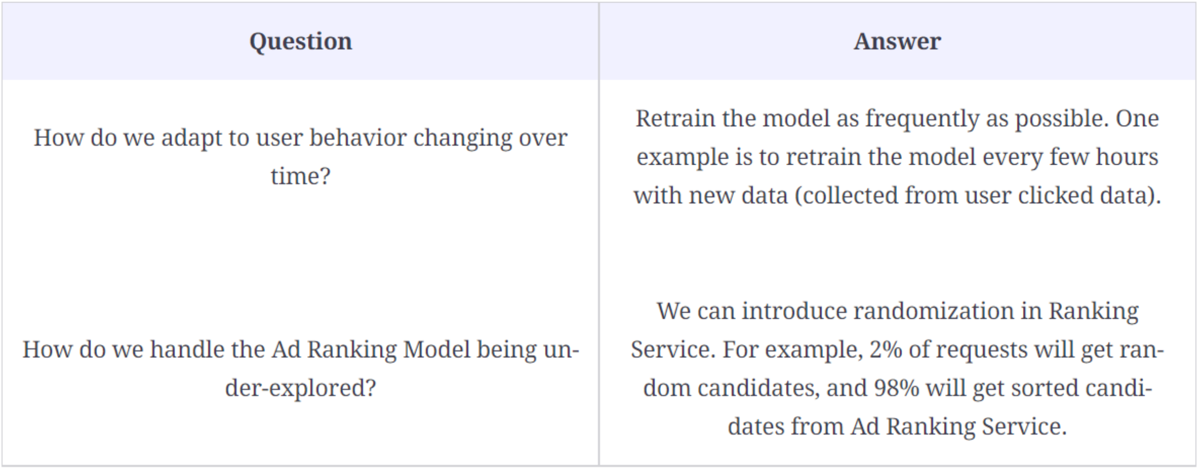


High-level design of ads recommendation system

- Given a latency requirement of 50ms-100ms for a large volume of Ad Candidates (50k-100k), if we partition one serving instance per request we might not achieve Service Level Agreement (SLA). For this, we scale out Model Serving and put Aggregator Service to spread the load for Model Serving components.

One common pattern is to have the Aggregator Service. It distributes the candidate list to multiple serving instances and collects results. Read more about it here.

7. Follow up questions#



Question	Answer
How do we adapt to user behavior changing over time?	Retrain the model as frequently as possible. One example is to retrain the model every few hours with new data (collected from user clicked data).
How do we handle the Ad Ranking Model being under-explored?	We can introduce randomization in Ranking Service. For example, 2% of requests will get random candidates, and 98% will get sorted candidates from Ad Ranking Service.

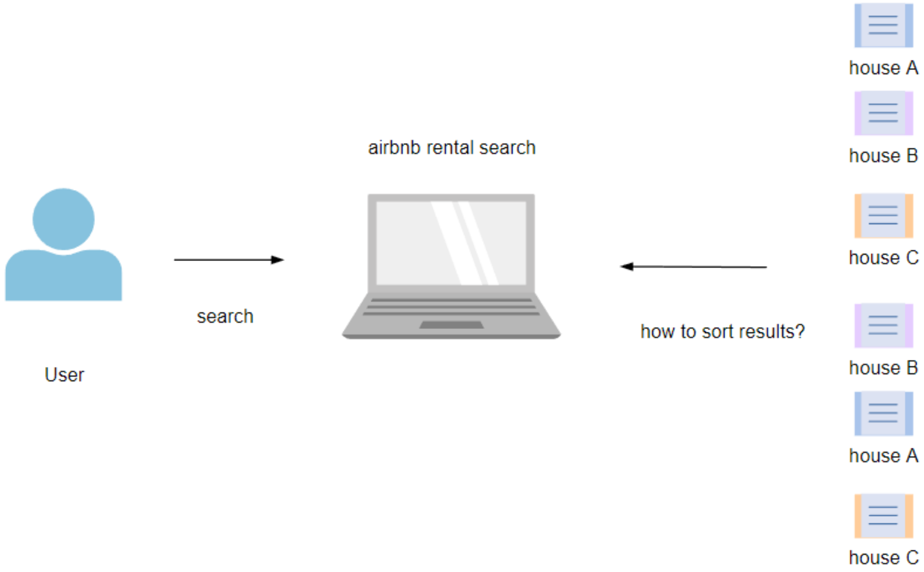
8. Summary#

- We first learned to choose Normalize Entropy as the metric for the Ad Click Prediction Model.
- We learn how to apply the Aggregator Service to achieve low latency and overcome imbalance workloads.
- To scale the system and reduce latency, we can use `kube-flow` so that Ad Generation services can directly communicate with Ad Ranking services.
- We can also learn more about how companies scale there design here.

Airbnb rental search ranking

1. Problem statement

Airbnb users search for available homes at a particular location. The system should sort stays from multiple homes in the search result so that the most frequently booked homes appear on top.



Search ranking system for Airbnb

- The naive approach would be to craft a custom score ranking function. For example, a score based on text similarity given a query. This wouldn’t work well because similarity doesn’t guarantee a booking.
- The better approach would be to sort results based on the likelihood of booking. We can build a supervised ML model to predict booking likelihood. This is a binary classification model, i.e., classify booking and not-booking.

2. Metrics design and requirements#

Metrics#

Offline metrics#

- Discounted Cumulative Gain

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

  where rel_i stands for relevance of result at position i.

- Normalized discounted Cumulative Gain:

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

- IDCG is ideal discounted cumulative gain:

  $$IDCG_p = \sum_{i=1}^{|REL_p|} \frac{2^{rel_i} - 1}{log_2(i+1)}$$

Online metrics#

- Conversion rate and revenue lift: This measures the number of bookings per number of search results in a user session.

  $$conversion\_rate = \frac{number\_of\_bookings}{number\_of\_search\_results}$$

Requirements#

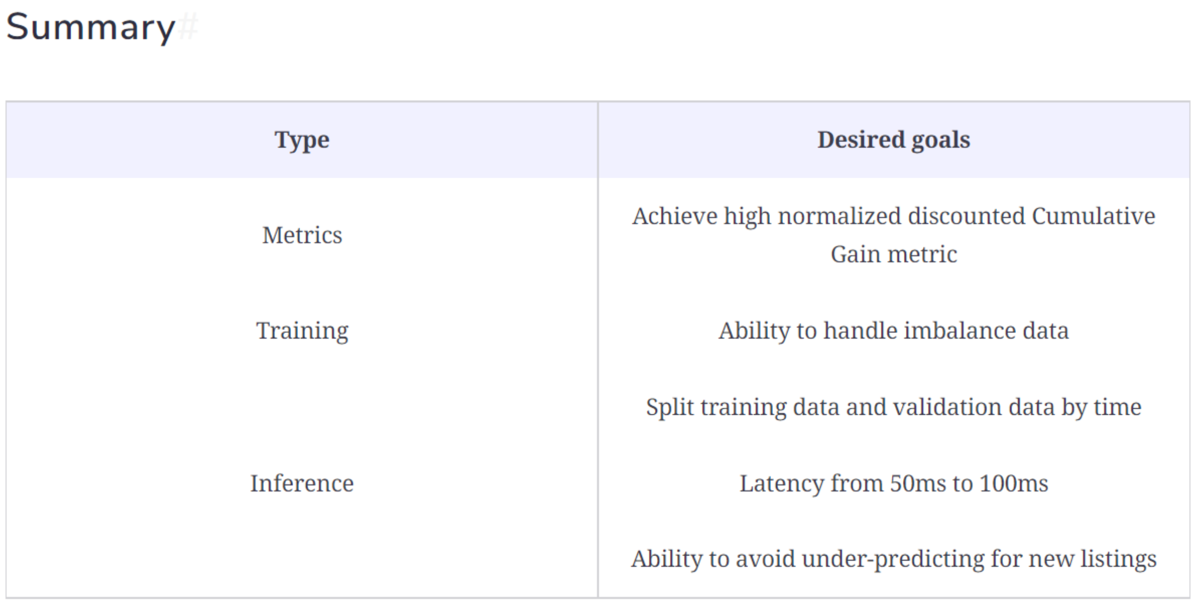
Training#

- Imbalanced data and clear-cut session: An average user might do extensive research before deciding on a booking. As a result, the number of non-booking labels has a higher magnitude than booking labels.

- Train/validation data split: Split data by time to mimic production traffic, for example, we can select one specific date to split training and validation data. We then select a few weeks of data before that date as training data and a few days of data after that date as validation data.

Inference#

- Serving: Low latency (50ms - 100ms) for search ranking
- Under-predicting for new listings: Brand new listings might not have enough data for the model to estimate likelihood. As a result, the model might end up under-predicting for new listings.



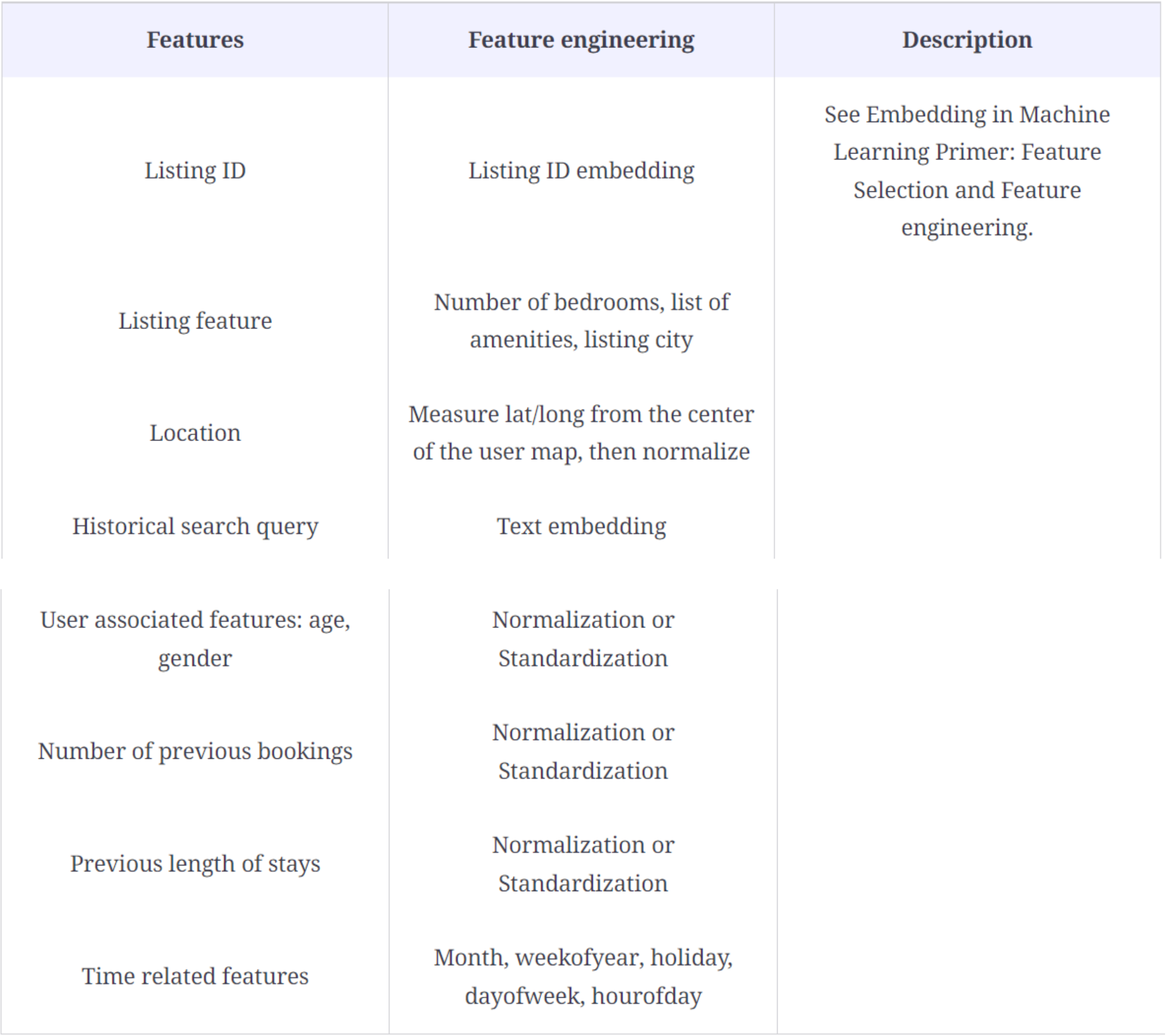
Summary#

Type	Desired goals
Metrics	Achieve high normalized discounted Cumulative Gain metric
Training	Ability to handle imbalance data
	Split training data and validation data by time
Inference	Latency from 50ms to 100ms
	Ability to avoid under-predicting for new listings

3. Model#

Feature engineering#

- Geolocation of listing (latitude/longitude): Taking raw latitude and raw longitude features is very tough to model as feature distribution is not smooth. One way around this is to take a log of distance from the center of the map for latitude and longitude separately.
- Favourite place: store user's favorite neighborhood place in 2 dimensions grid. For example, users add Pier 39 as their favorite place, we encode this place into a specific cell, then use embedding before training/serving.



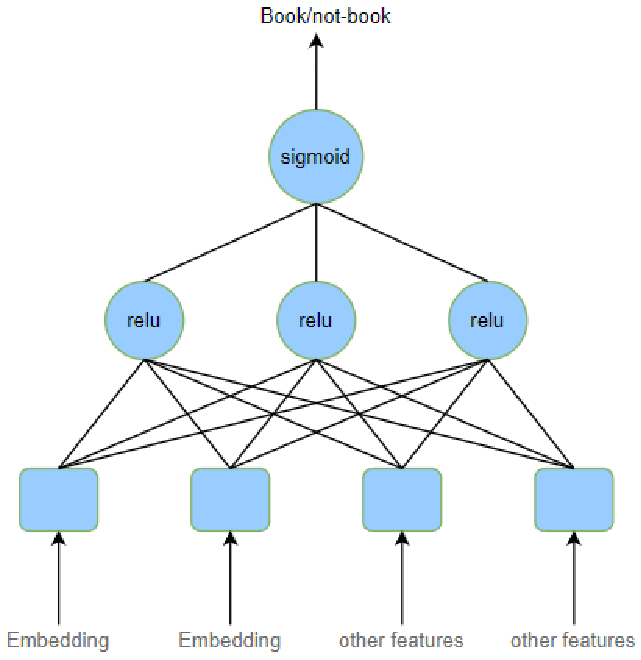
Features	Feature engineering	Description
Listing ID	Listing ID embedding	See Embedding in Machine Learning Primer: Feature Selection and Feature engineering.
Listing feature	Number of bedrooms, list of amenities, listing city	
Location	Measure lat/long from the center of the user map, then normalize	
Historical search query	Text embedding	
User associated features: age, gender	Normalization or Standardization	
Number of previous bookings	Normalization or Standardization	
Previous length of stays	Normalization or Standardization	
Time related features	Month, weekofyear, holiday, dayofweek, hourofday	

Training data#

- User search history, view history, and bookings. We can start by selecting a period of data: last month, last 6 months, etc., to find the balance between training time and model accuracy.

In practice, we decide the length of training data by running multiple experiments. Each experiment will pick a certain time period to train data. We then compare model accuracy and training time across different experimentations.

Model architecture



DL fully connected model

- Input: User data, search query, and Listing data.
- Output: This is a binary classification model, i.e., user books a rental or not.
- We can start with the deep learning with fully connected layers as a baseline. Model outputs a number within [0, 1] and presents the likelihood of booking.
- To further improve the model, we can also use other more modern network architecture, i.e., Variational AutoEncoder or Denoising AutoEncoder. Read more about Variational Autoencoder.

4. Calculation & estimation#

Assumptions#

- 100 million monthly active users
- On average, users book rental homes 5 times per year. Users see about 30 rentals from the search result before booking.

- There are 5 * 30 * 10 ^ 8 or 15 billion observations/samples per year or 1.25 billion samples per month.

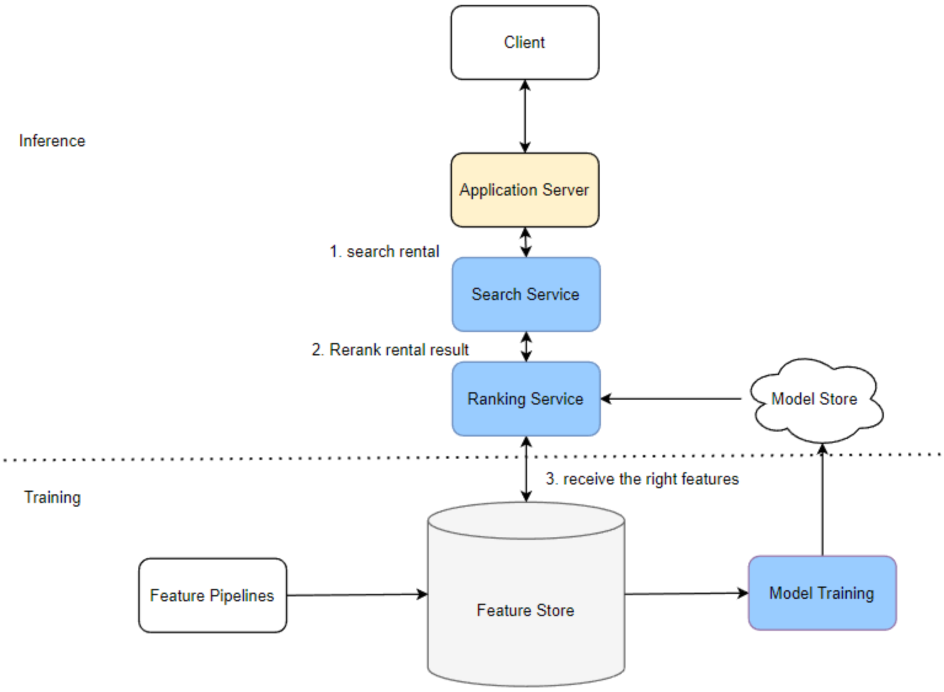
Data size#

- Assume there are hundreds of features per sample. For the sake of simplicity, each row takes 500 bytes to store.
- Total data size: 500 * 1.25 * 10^9= 625 * 10^9 bytes = 625 GB. To save costs, we can keep the last 6 months or 1 year of data in the data lake, and archive old data in cold storage.

Scale#

- Support 150 million users

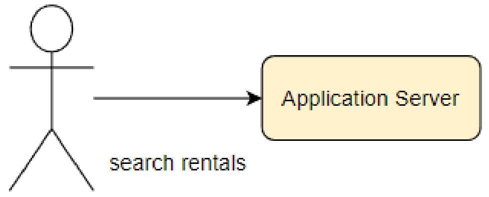
5. High-level design#

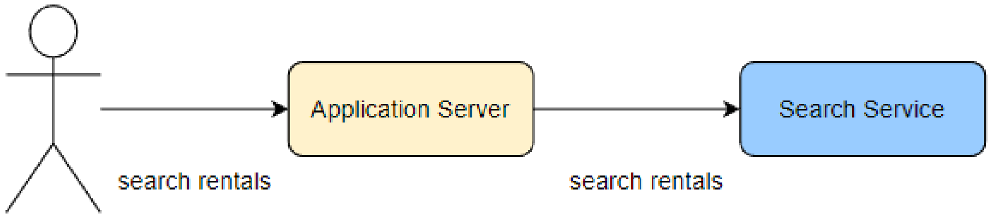


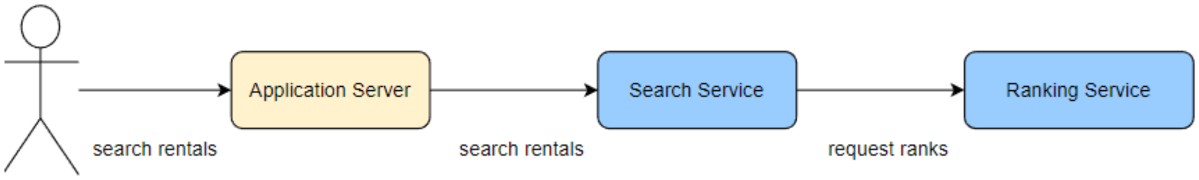
Rental Search Ranking
high level design

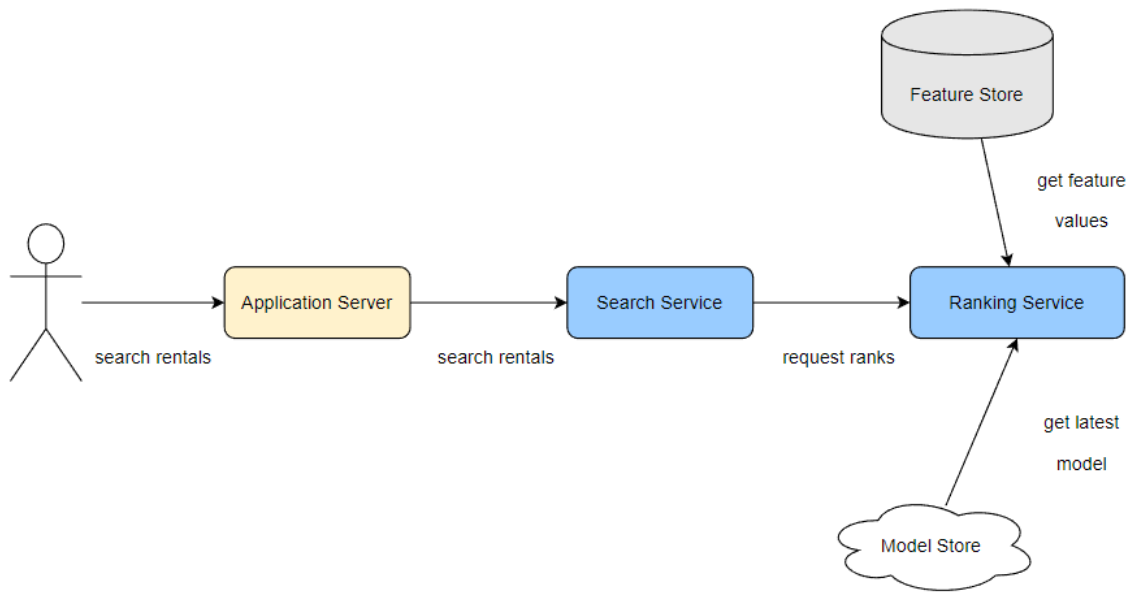
- Feature pipeline: Processes online features and stores features in key-value storage for low latency, down-stream processing.
- Feature store is a features values storage. During inference, we need low latency (<10ms) to access features before scoring. Examples of feature stores include MySQL Cluster, Redis, and DynamoDB.
- Model Store is a distributed storage, like S3, to store models.

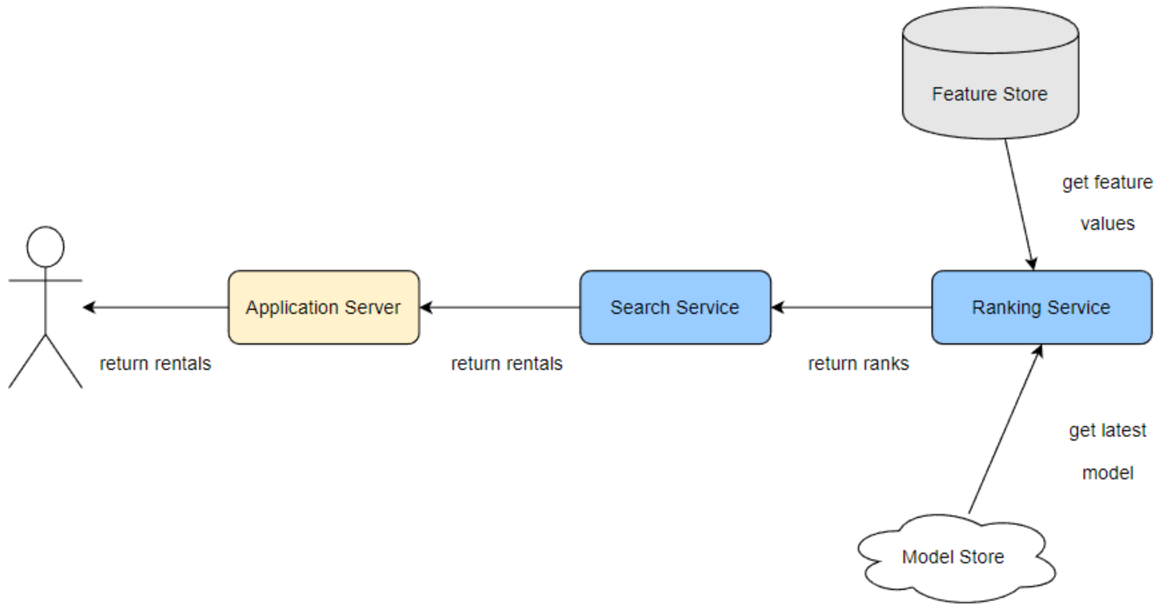
Let's examine the flow of the system:









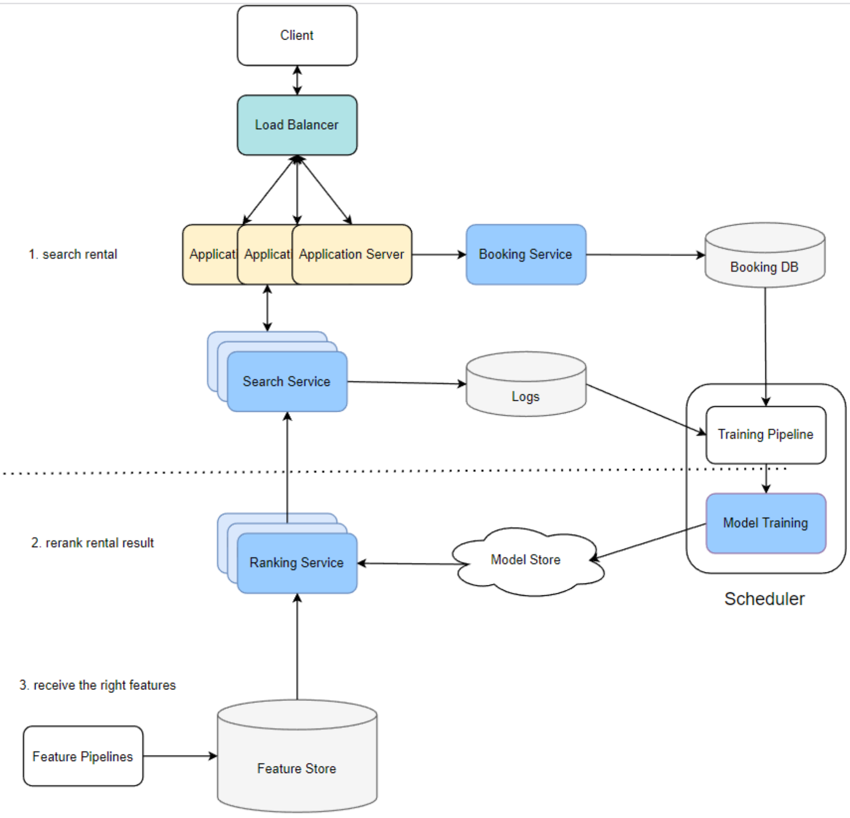


- User searches rentals and request Application Server for result
- Application Server sends search request to Search Service
- Search Service gets rentals from database and sends rental rank request to Ranking Service
- Ranking Service scores each rental results and returns the score to Search Service
- Search Service returns rentals to Application Server and Application Server returns rentals to user.

- The user searches for rentals with given queries, i.e., city and time. The Application Server receives the query and sends a request to the Search Service.
- Search Service looks up in the indexing database and retrieves a list of rental candidates. It then sends those candidates list to the Ranking Service.
- Ranking service uses the Machine Learning model to score each candidate. The score represents how likely it is a user will book a specific rental. The Ranking service returns the list of candidates with their score.
- Search Service receives the candidates list with booking probability and uses the probability to sort the candidates. It then returns a list of candidates to the Application server, which, in turn, returns it to the users.

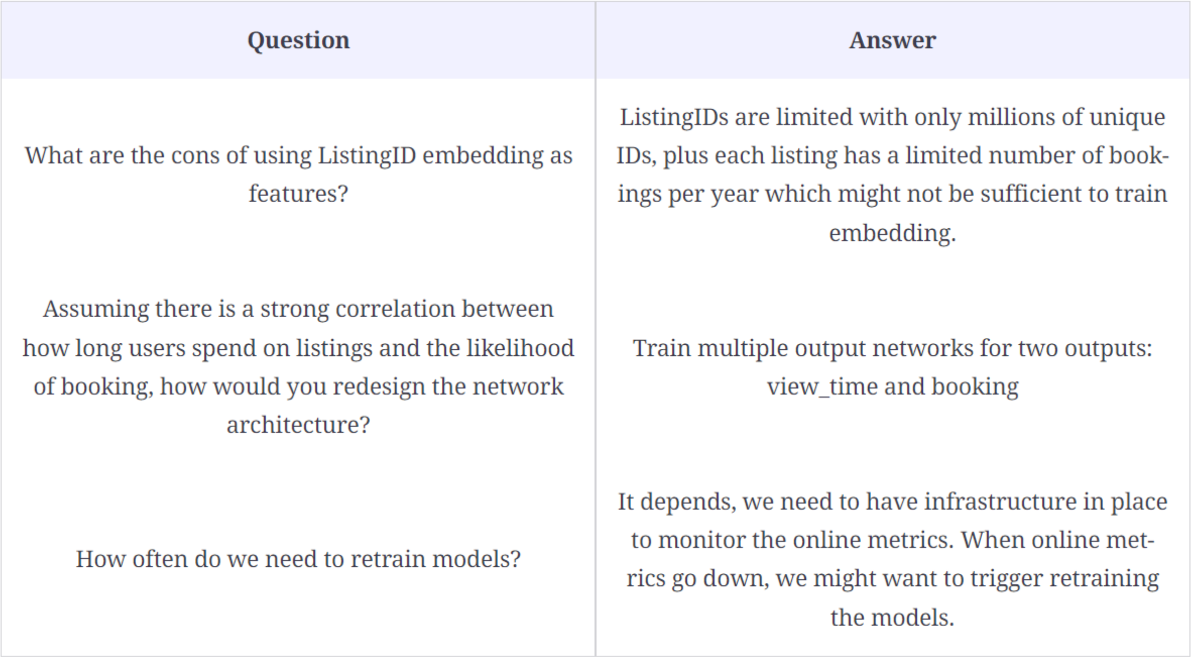
As you can see, we started with a simple design, but this would not scale well for our demands, i.e., 150 million users and 1.25 billion searches per month. We will see how to scale the design in the next section.

6. Scale the design#



- To scale and serve millions of requests per second, we can start by scaling out Application Servers and use Load Balancers to distribute the load accordingly.
- We can also scale out Search Services and Ranking Services to handle millions of requests from the Application Server.
- Finally, we need to log all candidates that we recommended as training data, so the Search Service needs to send logs to a cloud storage or send a message to a Kafka cluster.

7. Follow up questions#



Question	Answer
What are the cons of using ListingID embedding as features?	ListingIDs are limited with only millions of unique IDs, plus each listing has a limited number of bookings per year which might not be sufficient to train embedding.
Assuming there is a strong correlation between how long users spend on listings and the likelihood of booking, how would you redesign the network architecture?	Train multiple output networks for two outputs: view_time and booking
How often do we need to retrain models?	It depends, we need to have infrastructure in place to monitor the online metrics. When online metrics go down, we might want to trigger retraining the models.

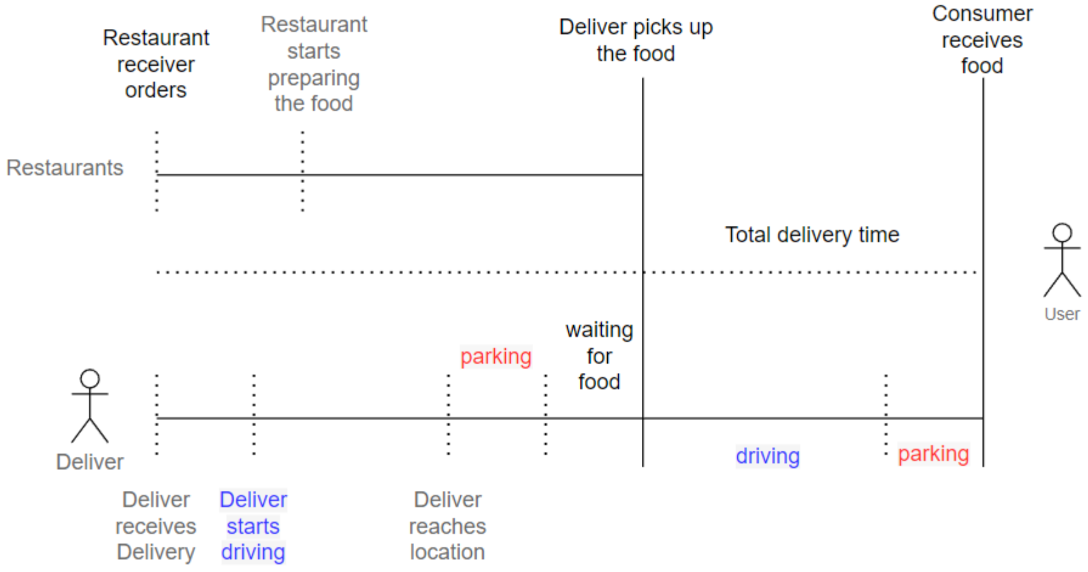
8. Summary#

- We learned to formulate search ranking as a Machine Learning problem by using booking likelihood as a target.
- We learned to use `Discounted Cumulative Gain` as one metric to train a model.
- We also learned how to scale our system to handle millions of requests per second.
- We can read more about how companies scale there design here.

Estimate Delivery Time#

1. Problem statement#

Build a model to estimate the total delivery time given order details, market conditions, and traffic status.



Food Delivery flow

To keep it simple, we do not consider batching (group multiple orders at restaurants) in this exercise.

$$DeliveryTime = PickupTime + Point\_to\_PointTime + Drop\_off\_Time$$

2. Metrics design and requirements#

Metrics#

- Offline metrics: Use Root Mean Squared Error (RMSE)

$$\sqrt{\sum_{k=1}^{n} \frac{(predict - y)^2}{n}}$$

where,

n is the total number of samples,

$predict$ is Estimated wait time,

y is the actual wait time.

- Online metrics: Use A/B testing and monitor RMSE, customer engagement, customer retention, etc.

Requirements#

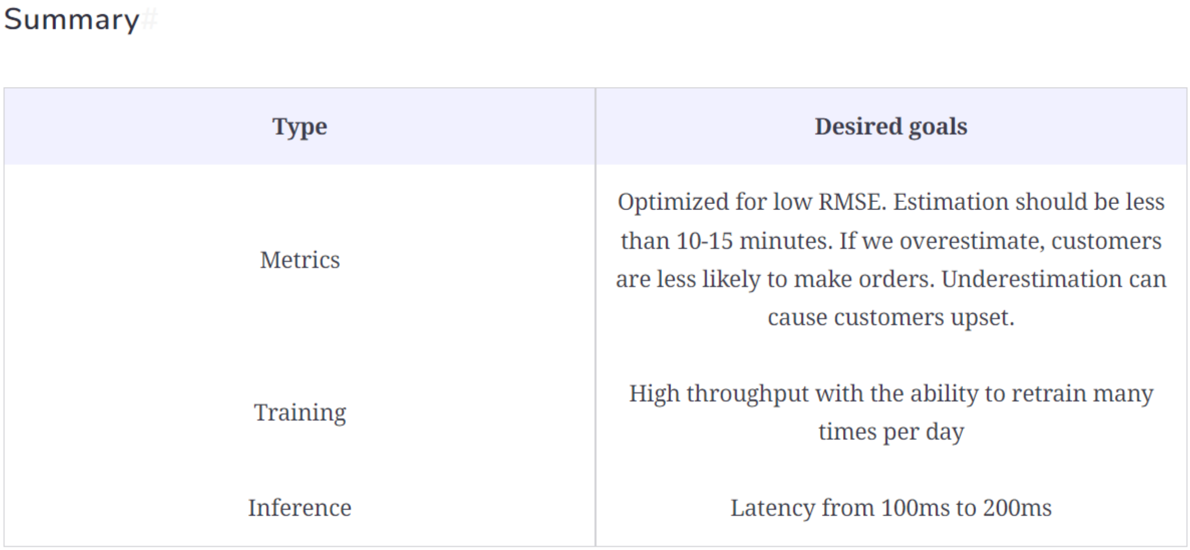
Training#

- During training, we need to handle a large amount of data. For this, the training pipeline should have a high throughput. To achieve this purpose, data can be organized in Parquet files
- The model should undergo retraining every few hours. Delivery operations are under a dynamic environment with a lot of external factors: traffic, weather conditions, etc. So, it is important for the model to learn and adapt to the new environment. For example, on game day, traffic conditions can get worse in certain areas. Without a retraining model, the current model will consistently underestimate delivery time. Schedulers are responsible for retraining models many times throughout the day.
- Balance between overestimation and under-estimation. To help with this, retrain multiple times per day to adapt to market dynamic and traffic conditions.

Inference#

- For every delivery, the system needs to make real-time estimations as frequently as possible. For simplicity, we can assume we need to make 30 predictions per delivery.
- Near real-time update, any changes on status need to go through model scoring as fast as possible, i.e., the restaurant starts preparing meals, the driver starts driving to customers.

- Whenever there are changes in delivery, the model runs a new estimate and sends an update to the customer.
- Capture near real-time aggregated statistics, i.e., feature pipeline aggregates data from multiple sources (Kafka, database) to reduce latency.
- Latency from 100ms to 200ms



Summary

Type	Desired goals
Metrics	Optimized for low RMSE. Estimation should be less than 10-15 minutes. If we overestimate, customers are less likely to make orders. Underestimation can cause customers upset.
Training	High throughput with the ability to retrain many times per day
Inference	Latency from 100ms to 200ms

3. Model#

Features engineering#

Features	Feature engineering	Description
Order features: subtotal, cuisine		
Item features: price and type		
Order type: group, catering		
Merchant details		
Store ID	Store Embedding	
Realtime feature	Number of orders, number of dashers, traffic, travel estimates	
Time feature	Time of day (lunch/dinner), day of week, weekend, holiday	
Historical Aggregates	Past X weeks average delivery time for: Store/City/market/TimeOfDay	
Similarity	Average parking times, variance in historical times	
Latitude/longitude	Measure estimated driving time between delivery of order(to consumer) & restaurants	

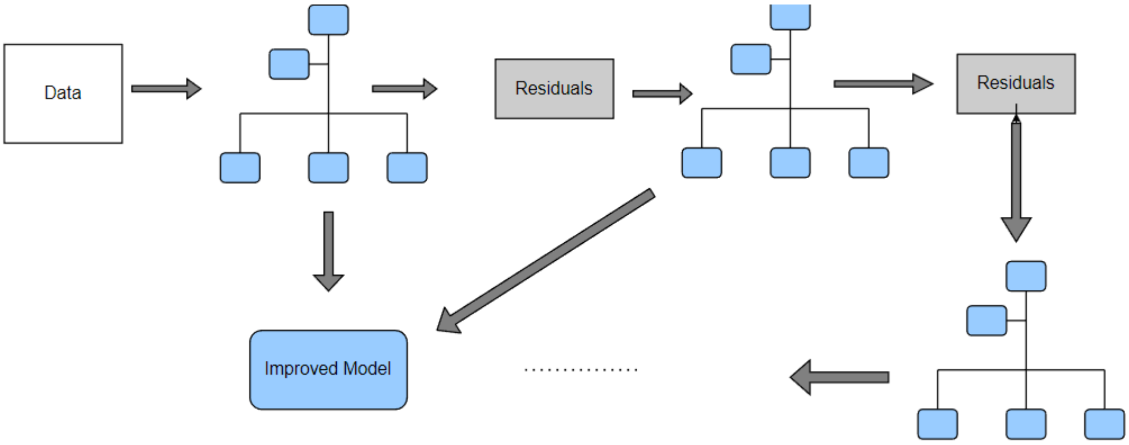
Training data#

- We can use historical deliveries for the last 6 months as training data. Historical deliveries include delivery data and actual total delivery time, store data, order data, customers data, location, and parking data.

Model#

Gradient Boosted Decision Tree#

- Gradient Boosted Decision Tree sample



How do Gradient Boosted Decision Trees work?

- Step 1: Given historical delivery, the model first calculates the average delivery time. This value will be used as a baseline.
- Step 2: The model measures the residual (error) between prediction and actual delivery time.

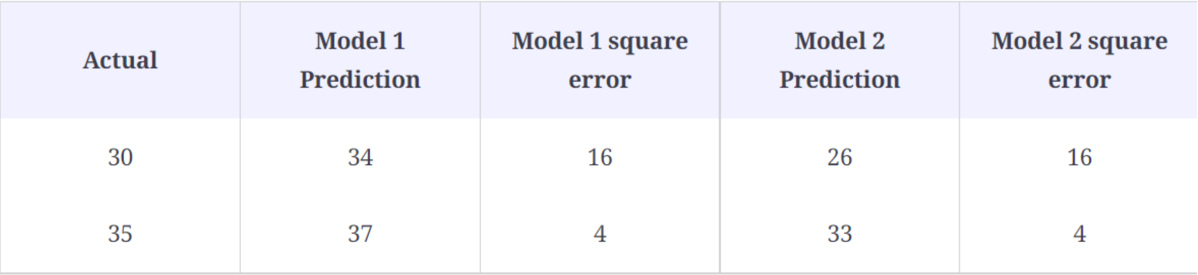
$$Error = ActualDeliveryTime - EstimatedDeliveryTime$$

- Step 3: Next, we build the decision tree to predict the residuals. In other words, every leaf will contain a prediction for residual values.
- Step 4: Next we predict using all the trees. The new predictions will be used to construct predictions for delivery time using this formula:

$$EstimatedDeliveryTime = Average\_delivery\_time + learning\_rate * residuals$$

- Step 5: Given the new estimated delivery time, the model then computes the new residuals. The new values will then be used to build new decision trees in step 3.
- Step 6: Repeat steps 3-5 until we reach the number of iterations that we defined in our hyperparameter.

One problem with optimizing RMSE is that it penalizes similarly between under-estimate prediction and over-estimate prediction. Have a look at the table below. Note that both models use boosted decision trees.



Actual	Model 1 Prediction	Model 1 square error	Model 2 Prediction	Model 2 square error
30	34	16	26	16
35	37	4	33	4

- Although Model 1 and Model 2 have the same RMSE error, model1 overestimates delivery time which prevents customers from making orders. Model2 underestimates the delivery time and might cause customers to be unhappy.

4. Calculation & estimation#

Assumptions#

For the sake of simplicity, we can make these assumptions:

- There are 2 million monthly active users, a total of 20 million users, 300k restaurants, and 200k drivers deliver food.
- On average, there are 20 million deliveries per year.

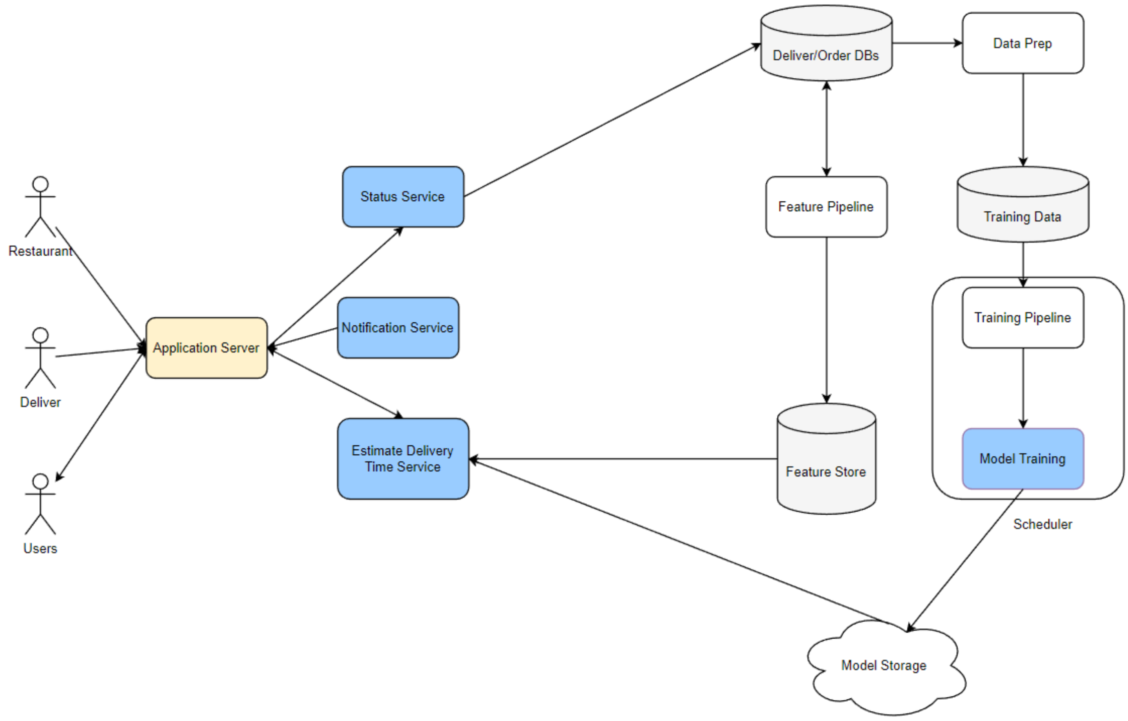
Data size#

- For 1 month, we collected data on 2 millions deliveries. Each delivery has around 500 bytes related features.
- Total size: 500 * 2 * 10^{6}1 = 10^9 bytes = 1 Gigabytes.

Scale#

- Support 20 million users

5. System Design#



- Feature Store: Provides fast lookup for low latency. A feature store with any key-value storage with high availability like Amazon DynamoDB is a good choice.

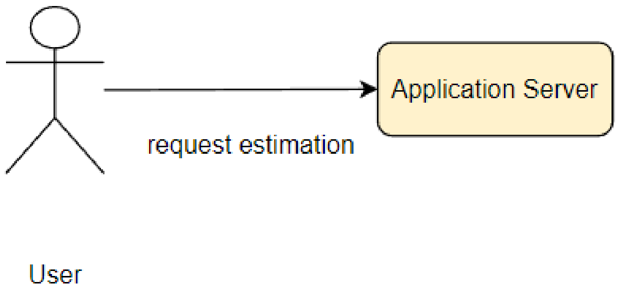
- Feature pipeline: Reads from Kafka, transforms, and aggregates near real-time statistics. Then, it stores them in feature storage.

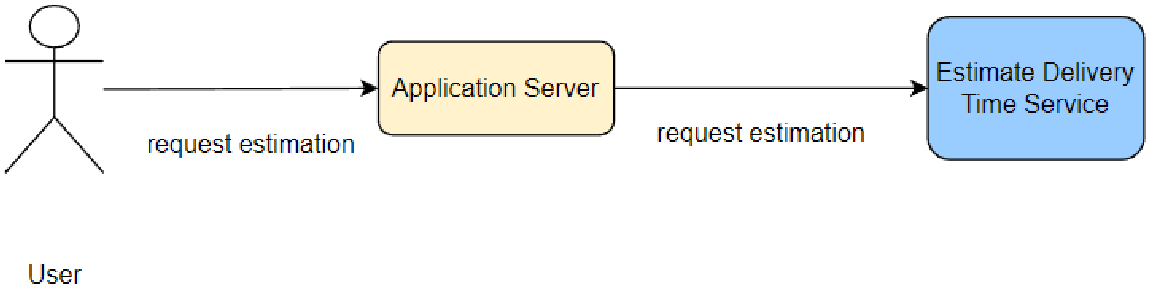
- Database: Delivery Order database stores historical Orders and Delivery. Data prep is a process to create training data from a database. We can store training data in cloud storage, for example, S3.

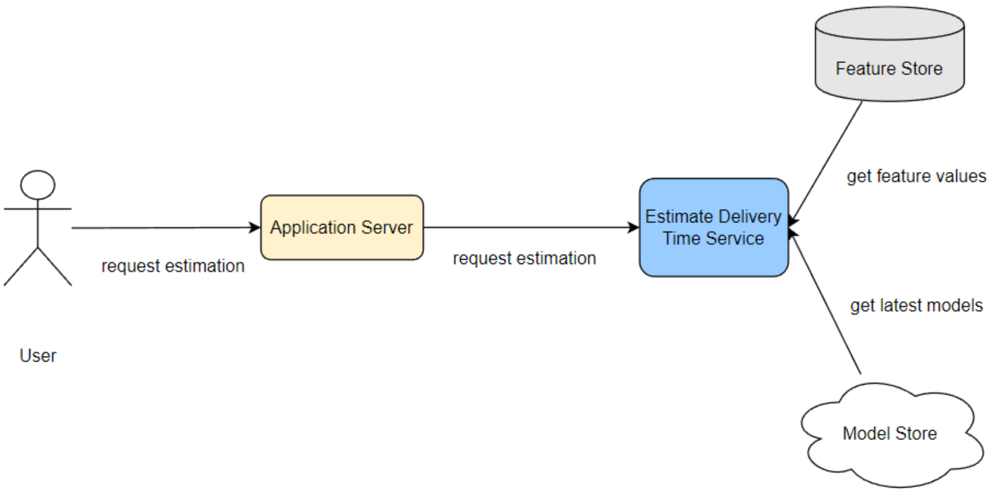
- We have three services: Status Service, Notification Service, and Estimate Delivery Time service. The first two services handle real-time updates and the Estimate Delivery Time service uses our Machine Learning Model to estimate delivery time.

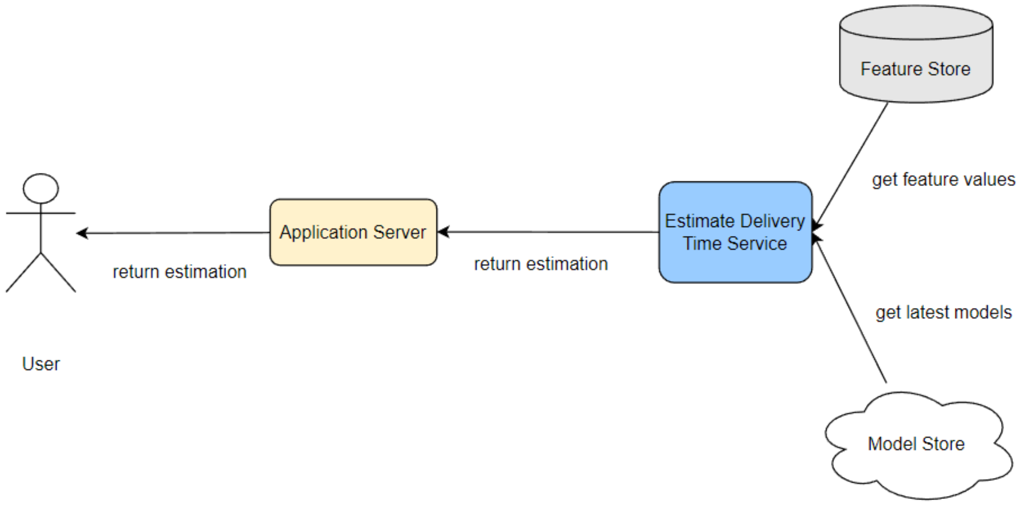
- We have a scheduler that handles and coordinates retraining models multiple times per day. After training, we store the Model in Model Storage.

Let’s examine the flow of the system:


Application Server
request estimation
User


Application Server
Estimate Delivery Time Service
request estimation
request estimation
User


Feature Store
get feature values
Application Server
Estimate Delivery Time Service
request estimation
request estimation
get latest models
User
Model Store

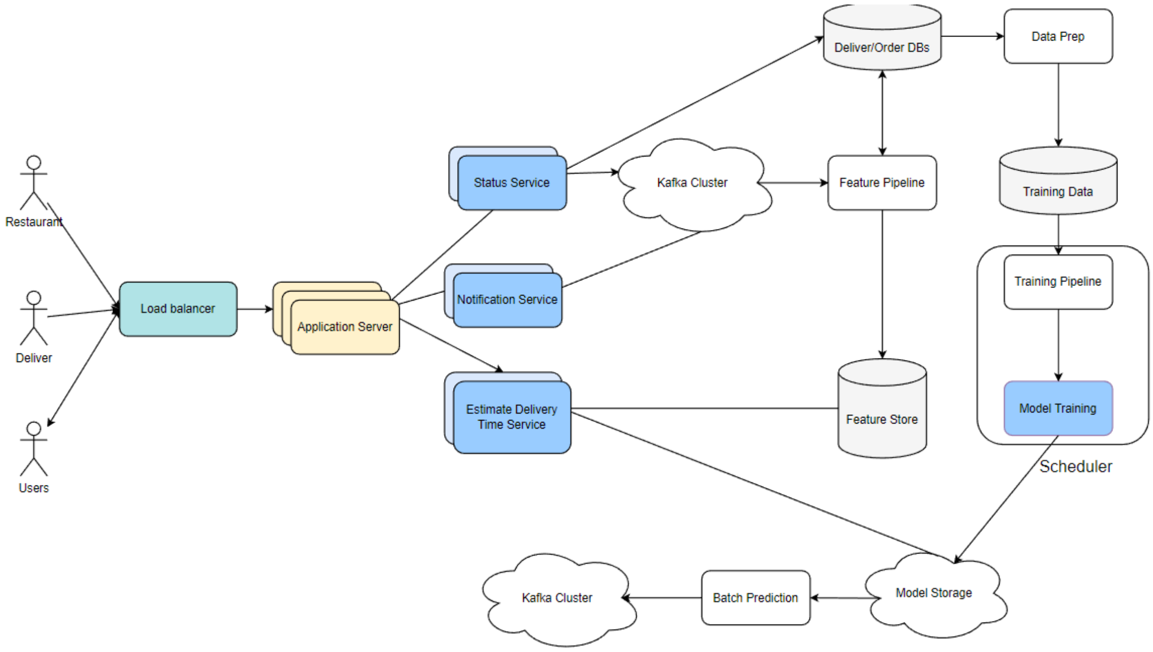

Feature Store
get feature values
Estimate Delivery Time Service
Application Server
return estimation
return estimation
get latest models
User
Model Store

- User requests for Estimated Delivery Time
- Estimate Delivery Time service returns time estimation to Application Server. Application Server returns time estimation to user.
- Estimate Delivery Time service returns time estimation to Application Server. Application Server returns time estimation to user.
- Estimate Delivery Time service returns time estimation to Application Server. Application Server returns time estimation to user.

- There are three main types of users: Consumer/User, Deliver, and Restaurant.
- User flow
  - User visits a homepage, checks their food orders, and requests Application Server for an estimated delivery time.
  - The Application Server sends the requests to the Estimate Delivery Time Service.
  - The Estimate Delivery Time service loads the latest ML model from Model Storage and gets all the feature values from the Feature Store. It then uses the ML model to predict delivery time and return results to the Application Server.
- Restaurant/Deliver flow:
  - When restaurants make progress, i.e., start making the dish or packaging the food, they send the status to Status Service.
  - Status Service updates the order status. This event is usually updated in a queue service, i.e, Kafka, so other services can subscribe and get updates accordingly.
  - Notification Service subscribed to the message queue, i.e., Kafka, and received the latest order status in near real-time.

6. Scale the design#

- We scale out our services to handle large requests per second. We also use a Load Balancer to balance loads across Application Servers.
- We leverage streaming process systems like Kafka to handle notifications as well as model predictions. Once our Machine Learning model completes its predictions, it sends them to Kafka so other services can get notifications right away.



7. Follow up questions#

Question	Answer
What are the cons of using StoreID embedding as features?	We need to evaluate if using StoreID embedding is efficient in handling new stores.
How often do we need to retrain models?	It depends, we need to have infrastructure in place to monitor the online metrics. When online metrics go down, we might want to trigger our models to retrain.

8. Summary#

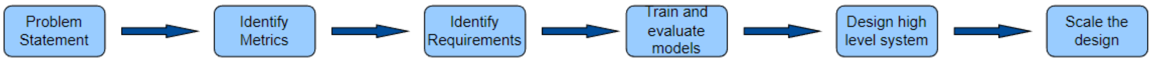
- We learned to formulate estimated delivery times as a Machine learning problem using Gradient Boosted Decision Trees.
- We learned how to collect and use data to train models.
- We learned how to use Kafka to handle logs and model predictions for near real-time predictions.
- We can read more about how companies scale there design here.

Conclusion

Summary

Congratulations! You have finished the Machine Learning System Design course.

- We have covered Machine Learning System Design from end to end and have learned to apply the same framework for various different problems.



The 6 basic steps to approach Machine Learning System Design

What’s next?#

- From Video Recommendation to Ad Click Prediction, we showed how Recommendation Systems can be used in various applications. In recent years, there are more advanced techniques that take advantage of the big data volume and combine different types of features to improve the recommendation system models. One example is Facebook’s recommendation model.

- Another common component across different use cases is the **Feature Store**. It became more critical in Machine Learning technical stacks.

- You can find more resources to learn about the modern techniques from this collection