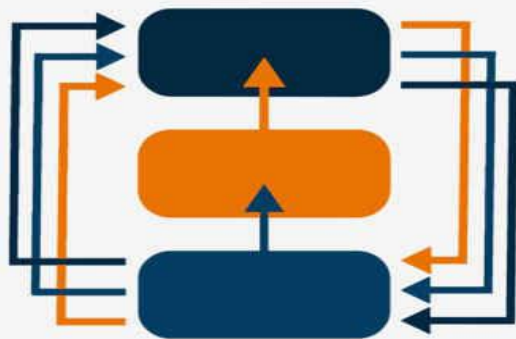


MACHINE LEARNING DESIGN INTERVIEW

HELPING HUNDREDS OF ENGINEERS SINCE 2019



MACHINE LEARNING DESIGN INTERVIEW

KHANG PHAM

CALIFORNIA, 2022

Machine Learning Design Interview

Copyright 2022 Khang Pham

All rights reserved under International and Pan-American Copyright
Conventions

No part of this book may be reproduced, stored in a retrieval system or
transmitted in any way by any means, electronic, mechanical, photocopy,
recording or otherwise without the prior permission of the author except as
provided by USA copyright law.

ISBN-13: 979-8-8130-3157-1 [paperback]

Success Stories

Name	Company Offer	Positions
Victor	Facebook	Facebook Senior Manager (E7) and others
Stanford Conor	Facebook, Amazon	Facebook ML Manager (E6/M0) and 5 other offers
Ted Vice	Amazon, Facebook	Amazon DS (L5) and Facebook DS (E5)
Mike Bloomberg	Facebook, Spotify	Facebook MLE (E5), Spotify MLE (Senior) and others
Jerry	Google, Apple, Facebook, Cruise	Google MLE (L5), Cruise MLE (L5), Apple MLE (Senior), FB MLE (E5) and others
Steven Chris	Google	Google MLE (L4) and others
Adam	Google	Google MLE (L5)
Patrick	Amazon, Wish	Amazon DS (L5), Wish DS
Bolton Chandra	Intuit	Intuit MLE (Senior)
David Nguyen	NVIDIA, NBCUniversal	Intern
Daniel	Series B startup	Senior MLE.
Steven	AccuWeather	SWE, ML
Sanchez	Pinterest, Citadel	Senior SWE, ML
Ben	Amazon	Applied Scientist
Mary	Twitter, Apple	Senior MLE
Mark	Facebook, Tiktok, other	Senior MLE (E5)
Ariana	Intuit, Bloomberg	Data Scientist
Michael		

Wong	Docusign	Senior SWE/ML
Teo	LinkedIn, Google, Tiktok	Senior SWE/AI, Google (L4)
Nick	Facebook, Google	MLE (E5), Google (L4)
Quinton	Microsoft, Vmware	Staff SWE/ML
Lex	Facebook, Google	Google Brain (L5) and Facebook (E5)
Strange	Facebook	Senior SWE/ML (E5)
Jim	Amazon	Amazon Applied Scientist (L4)
Shawn	Amazon, Google, Uber	Amazon Applied Scientist, Google Senior SWE/ML(L5) and Uber L4

I wanted to update you that I just signed my offer with Amazon. Thank you so much for your help!

Ted, Amazon Senior Data Scientist

I'm done with interviews. FB called with an initial offer, now I need to start negotiating

Michael, Facebook Senior SWE/ML

I got the offer from Intuit. Thank you so much, it would not be possible without your help.

Bolton, Intuit Senior SWE/ML

Hi Khang, thank you for your help in the past few months. I have received an offer from Docusign.

Michael Wong, DocuSign Senior SWE/ML

Thanks to your Github repo, I got a research internship at Samsung. Your notes are so helpful. Many thanks.

Dave Newton, Samsung MLE Intern

I received an offer from Facebook (exciting!!!)

Victor, Facebook Senior ML Manager

I'm very happy that I could crack the Google interview. So finally I could pass Uber, Google and Amazon. Amazon was an applied scientist while the other two were ML engineers. Lots of hard work and effort to get to this state Thank you for staying with me all this time.

Shawn, Google Senior SWE/ML

Thanks to ML system design on , I just wanted to say I thought the course was super helpful! I got offers from google, fb, apple and tesla.

Jerry, Google Senior SWE/ML

Hi Khang, I got the offer from the company. THANK YOU for your coaching!!!

Daniel, Unicorn startup Senior SWE/ML

Hi Khang, I got an offer from Apple and Twitter. Thanks for your support.

Mary, Apple Senior SWE/ML

Hi Khang, I got an offer from Intuit today. Thank you so much for all your help.

Ariana, Intuit Data Scientist

Hi Khang, I want to let you know that I got offers from FB and Google. I decided to take the FB offer in the end. Thank you so much for guiding me through the whole interviewing process. Best!

Mark, Facebook Senior SWE/ML

Hey Khang, Received an E5 offer :). Thanks for helping me throughout this process.

Strange, Facebook Senior SWE/ML

*This book is for my wife,
my son, my mom and my dad.*

Preface

Machine learning design is a difficult topic. It requires knowledge from multiple disciplines, such as big data processing, linear algebra, machine learning and deep learning. This is the first book that focuses on applied machine learning and deep learning in production. It covers all the design patterns and state-of-the-art techniques from top companies, such as Google, Facebook, Amazon, etc.

Who should read this book?

Data scientist, software engineer or data engineer who have a background in Machine Learning but never work on Machine Learning at scale will find this book helpful.

How to read this book?

- Section 1.1.1 to 1.1.4 helps you review Machine Learning fundamentals. If you have a lot of experience in Machine Learning you can skip these sections.
- Section 1.1.5 is very important at big tech companies, especially Facebook.
- Chapter 2 helps you review important topics in the Recommendation System.
- Chapter 3 to chapter 8 explains end to end design of the most popular Machine Learning system at big tech companies.
- Chapter 9 helps you test your understanding.

I'd like to acknowledge my friends Steven Tartakovsky, Tommy Nguyen and Jocelyn Huang for helping me in proofreading this book.

Keep up to date with Machine Learning Engineer: mlengineer.io. All the quiz solutions can be found at: <https://rebrand.ly/bookerrata>.

Khang Pham
California
April 2022

Machine Learning Primer

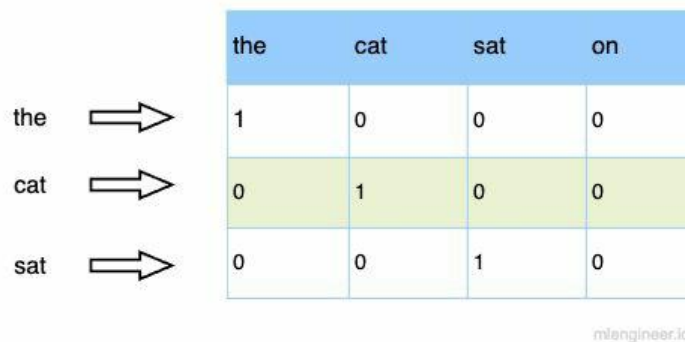
In this chapter, we will review commonly used machine learning techniques from industry. We focus on the application of Machine Learning techniques. The readers should already know most of these concepts in theory.

Feature Selection and Feature Engineering

One Hot Encoding

One hot encoding is very popular when you have to deal with categorical features having medium size cardinality.

In this example, when we have one column with four unique values, we create three more extra columns after one hot encoding. If one column has thousands of unique values, one hot encoding will create thousands of new columns.



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

mlengineer.io

One Hot Encoding. Source: mlengineer.io

Common Problems

Tree-based models, such as decision trees, random forests, and boosted trees, don't perform well with one hot encodings, especially when the tree has many levels (i.e., when there are values of categorical attributes). This is because they pick the feature to split, based on how well splitting the data on that feature will "purify" it. If we have several levels, only a small fraction of the data will usually belong to any given level, so the one hot encoded columns will be mostly zeros. Since splitting on this column will only produce a small gain, tree-based algorithms typically ignore the information

in favor of other columns. This problem persists, regardless of the volume of data you actually have. Linear models or deep learning models do not have this problem.

Expansive computation and high memory consumption: many unique values will create high-dimensional feature vectors. For example, if a column has a million unique values, it produces feature vectors, each with a dimensionality of one million.

Best Practices

When levels (categories) are not important, we can group them together in "Other" class. Make sure that the pipeline can handle unseen data in the test set. In python, you can use `pandas.get_dummies` or `sklearn OneHotEncoder`. However, `pandas.get_dummies` does not "remember" the encoding during training, and if testing data has new values, it can lead to inconsistent mapping.

OneHotEncoder in scikit-learn has the advantage as you can use `fit/transform/fit_transform`, therefore, you can persist and use it together with Pipeline.

One Hot Encoding in Tech Companies

One Hot Encoding is used a lot in tech companies. For example, at Uber, one hot encoding is used on features before training some of their production XGboost models. However, sometimes, when there are a large number of categorical values, such as in the tens of thousands, it becomes impractical to reasonably use One Hot Encoding. There is another technique, that's actually used at Instacart on their models, that's called mean encoding mean.

Mean Encoding

Take the Adult income data set example. We have data about the income of 50,000 people with different demographics: age, gender, education

background, etc. Let's assume we want to handle Age data as categorical. There can be 80-90 unique values for this column. If we apply one hot encoding, it will create a lot of new columns for this small data set.

Adult income dataset	
Age	Income
18	60,000
18	50,000
18	40,000
19	66,000
19	51,000
19	42,000

We treat the *Age* feature as continuous variables by taking the average of income for that *Age* value. For example, we can create a new column *Age_mean_enc*. It represents the mean value of income for a specific *Age*. The benefit is that we can use this new column as a continuous variable.

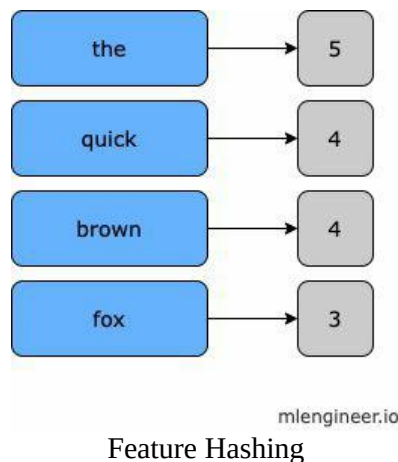
Mean Encoding for Income Data		
Age	Income	Age_mean_enc
18	60,000	50,000
18	50,000	50,000
18	40,000	50,000
19	66,000	53,000
19	51,000	53,000
19	42,000	53,000

If we use this method for the whole data we use for training, it will lead to label leakage. So it's important that we use separate data for computing mean encoding. To make mean encoding even more robust, we can also apply Additive Smoothing¹ or *Cross Validation* methods.

```
encode_type = train.groupby('age')['income'].mean()  
train.loc[:, 'Age_mean_enc'] = train['age'].map(  
    encode_type)
```

Feature Hashing

Feature hashing, or *hashing trick*, converts text data, or categorical attributes with high cardinalities, into a feature vector of arbitrary dimensionality. In some AdTech companies (Twitter, Pinterest, etc.), it's not uncommon for a model to have thousands of raw features.



Benefits

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

Feature Hashing Example

First, you decide on the desired dimensionality of your feature vectors. Then, using a hash function, you first convert all values of your categorical attribute (or all tokens in your collection of documents) into a number, and then

convert this number into an index of your feature vector. The process is illustrated in figure [1.1](#).

Let's illustrate how it would work for converting the text “the quick brown fox” into a feature vector. Let us have a hash function h that takes a string as input and outputs a non-negative integer, and 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
```

Then we build the feature vector as, $[1, 0, 0, 1, 2]$. Indeed, $h(\text{the}) \bmod 5 = 0$ means that we have one word in dimension 0 of the feature vector; $h(\text{quick}) \bmod 5 = 4$ and $h(\text{brown}) \bmod 5 = 4$ means that we have two words in dimension 4 of the feature vector, and so on.

As you can see, there is a collision between the words “quick” and “brown”: they both are represented by dimension 3. The lower the desired dimensionality, the higher are the chances of collision. This is the trade-off between speed and quality of learning.

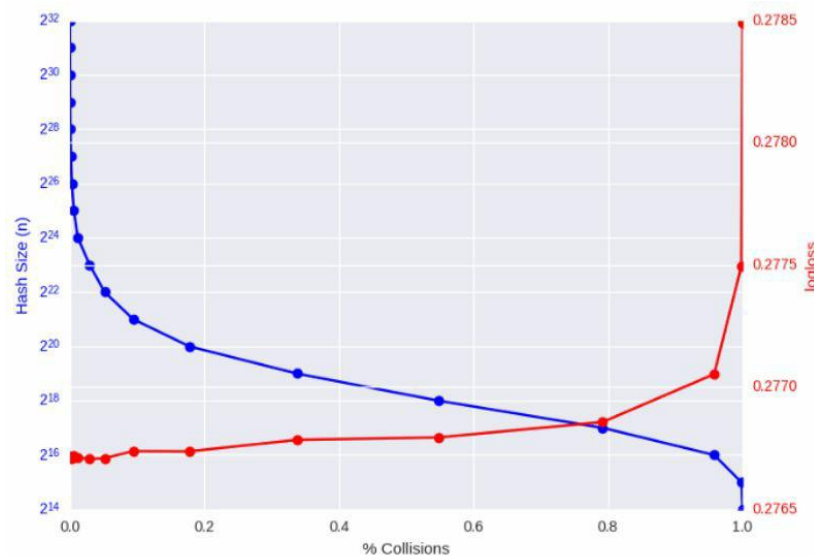
```
layer = tf.keras.layers.Hashing(num_bins=5)  
inp = [['the'], ['quick'], ['brown'], ['fox']]  
layer(inp)
```

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

Feature Hashing in Tech Companies

Feature hashing is widely popular in a lot of tech companies such as Booking, Facebook (Semantic Hashing using Tags and Topic Modeling, 2013), Yahoo, Yandex, Avazu, and Criteo.

One problem with hashing is collisions. 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, the model won't be able to differentiate coefficients between feature values. For example, the coefficient for "User login/User logout" might end up being the same, which makes no sense.



Feature Hashing: Hash Size vs Logloss. Source: booking.com

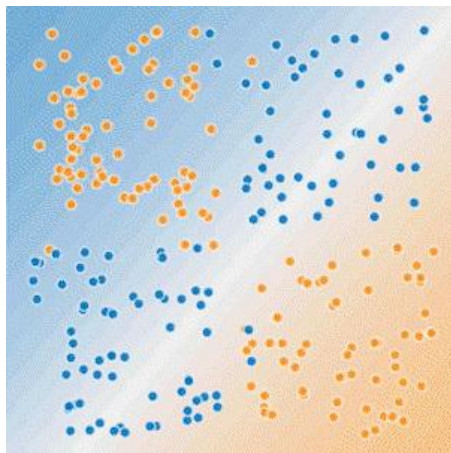
Depending on the application, you can choose the number of bits for feature hashing that provide the right balance between model accuracy and computing cost.

Cross Feature

Cross feature, or *conjunction*, between two categorical variables of cardinality c_1 and c_2 is just another categorical variable of cardinality $c_1 \times c_2$. If c_1 and c_2 are large, the conjunction feature has high cardinality, and the use of the hashing trick is even more crucial in this case. Cross feature is usually used with a hashing trick to reduce the high dimensions. As an example, suppose we have Uber pick up data with latitude and longitude stored in a database, and we want to predict demand at a

certain location. If we only use the feature latitude for learning, the model might learn that city blocks at particular latitudes are more likely to have higher demand than others. Likewise, for the feature longitude. However, if we cross longitude by latitude, the cross feature represents a well-defined city block and allows the model to learn more accurately.

What would happen if we don't create a cross feature? In this example, we have two classes: orange and blue. Each point has two features: x_1 and x_2 . Can we draw a line to separate them? Can we use a linear model to learn to separate these classes? To solve this problem, we can introduce a new feature: $x_3 = x_1 * x_2$. Now we can learn a linear model with three features: x_1 , x_2 and $x_1 * x_2$.



Cross feature. Source: developers.google.com

Cross features are also very common in recommendation systems. In practice, we can also use wide and deep architecture to combine many dense features and sparse features. You can see one concrete example in section Wide and Deep [\[sec-wide-and-deep\]](#).

```
crossed_feature = feature_column.crossed_column([
    age_buckets, animal_type], hash_bucket_size=10)
```

Embedding

Both one hot encoding and feature hashing can represent features in

multidimensions. However, these representations do not usually preserve the semantic meaning of each feature. For example, using OneHotEncoding can't guarantee the word 'cat' and 'animal' are close to each other in multidimensions; or user 'Kanye West' is close to 'rap music' in YouTube data. The proximity here can be interpreted from the semantic perspective or engagement perspective. This is an important distinction and has implications for how we train embedding.

How to Train Embedding

In practice, there are two ways to train embedding: *pre-trained* embedding i.e: word2vec² style or *co-trained*, (i.e., YouTube video embedding).

In Word2Vector representation, we want our vector representation for each word such that if $\text{vector}(\text{word1})$ is close to $\text{vector}(\text{word2})$, then they are somewhat semantically similar. We can achieve this by using the surrounding words to predict the middle word in the sentence or using one word to predict surrounding words. We can see an example in the section below.

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

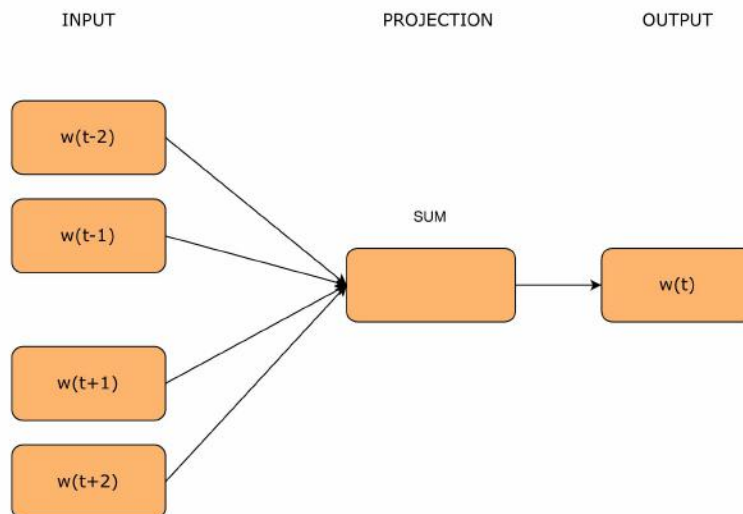
mlengineer.io

Embedding

As an example, each word can be represented as a 4d dimension vector, and we can train our supervised model. We then use the outputs of one of the fully connected layers near the output layer of the neural network model as embeddings of the input object. In this example, embedding for 'cat' is represented as a $[1.2, -0.1, 4.3, 3.2]$ vector.

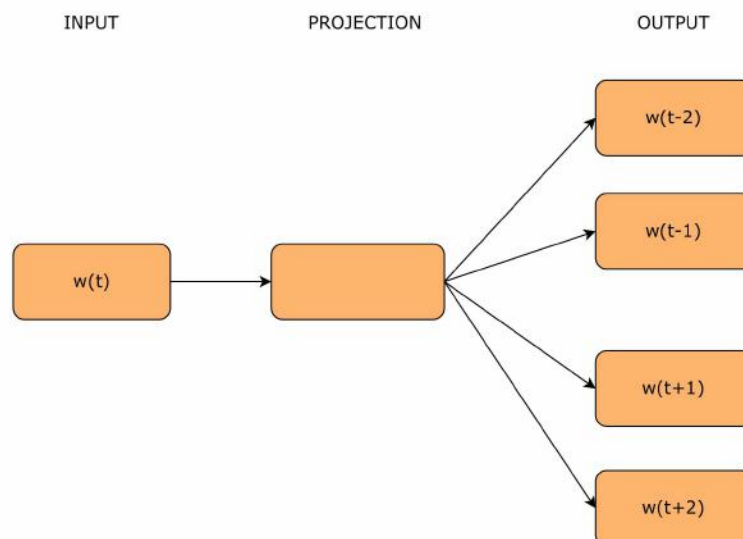
There are two ways to formulate the problems: Continuous Bag of Words

(CBOW) and Skip-gram. For CBOW, we want to predict one word based on the surrounding words. For example, if we are given: word1 word2 word3 word4 word5, we want to use (word1, word2, word4, word5) to predict word3.



CBOW. Source: Exploiting Similarities Among Languages for Machine Translation

In the skip-gram model, we use 'word3' to predict all surrounding words 'word1, word2, word4, word5'.



Skipgram. Source: Exploiting Similarities Among Languages for Machine Translation

Word2Vec example

Work2vec CBOW example

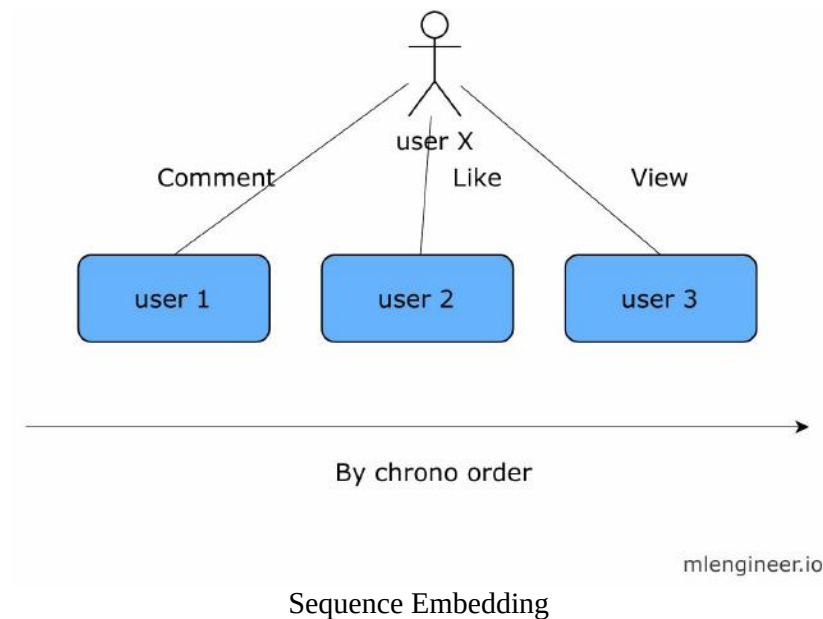
Model Input	Label
the, cat, on, the	sat
cat, sat, the, orange	on
sat, on, orange, tree	the

Instagram uses this type of embedding to provide personalized recommendations for their users, while Pinterest uses this as part of their Ads Ranking model. In practice, for some apps like Pinterest and Instagram where the user's intention is strong, we can use word2vec style embedding training.

How Does Instagram Train User Embedding?

Within one session, Instagram user A sees the photos for user B then user C's and so on. If we assume user A is currently interested in certain topics, we can also assume user B and user C's photos might be relevant to those topics of interest. For each user session, we have a collection of actions like the below diagram:

User A → see user
B photos → see
user C photos

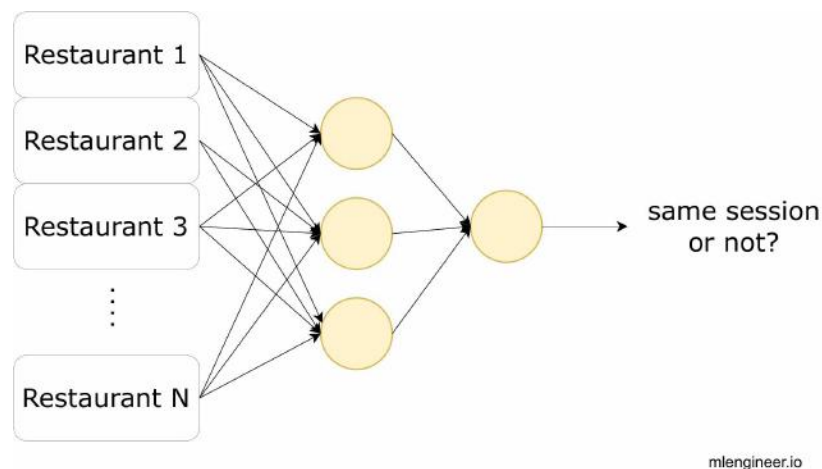


We can formulate each session as a sentence and each user's photos as words. This is suitable for users who are exploring photos or accounts in a similar context during a specific session.

How Does DoorDash Train Store Embedding?

DoorDash uses this approach to do store embedding. For each session, we assume users may have a certain type of food in mind, and they view store A, store B, etc. We can assume these stores are somewhat similar to the user's interests.

Store 1 → Store 2
→ Store 3



Store Embedding. Source: Doordash

We can train a model to classify a given pair of stores if they show up in a user session. Next, we will see another way to train embedding. It usually looks at embedding to optimize for some engagement metrics.

How Does YouTube Train Embedding in Retrieval?

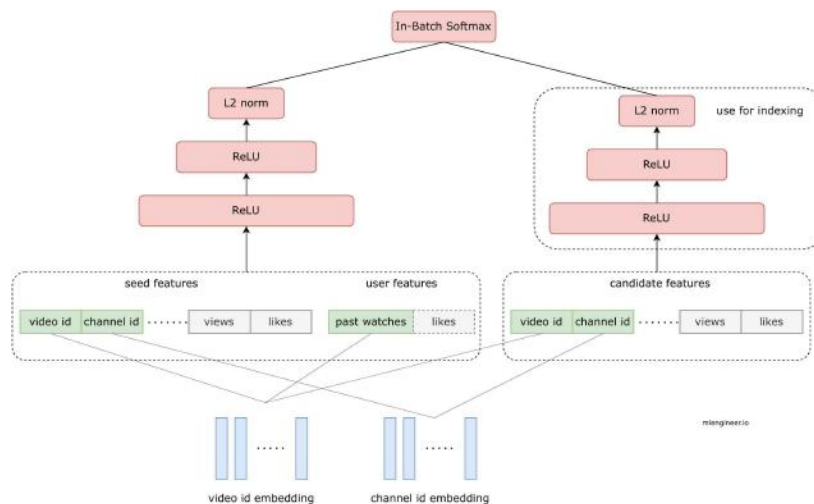
Recommendation System usually consists of three stages: Retrieval, Ranking and Re-ranking (read Chapter [\[rec-sys\]](#)). In this example, we will cover how YouTube builds Retrieval (Candidate Generation) component using Two-tower architecture.

Figure [1.2](#) provides an illustration of the two-tower model (read Common Deep Learning [1.5](#) section) architecture where left and right towers encode user, context and item respectively. Intuitively we can treat this problem as a multi-class classification problem. We have two towers³: left tower takes (users, context) as input and right tower takes movies as input.

- Two-tower Deep Neural Network⁴ is generalized from the multi-class classification neural network, a multi-layer perceptron (MLP) model, where the right tower of Figure [1.2](#) is simplified to a single layer with item embeddings.
- Given input x (user, context), we want to pick candidate y (videos) from

all available videos.

- A common choice is to use Softmax function $P(y|x;\theta) = \frac{e^{s(x,y)}}{\sum_{i=1}^m e^{s(x,y_i)}}$
- Loss function: use log-likelihood $L = -1/T \sum_{i=1}^T \log(P(y_i|x_i;\theta))$
- As a result, the two-tower model architecture is capable of modeling the situation where the label has structures or content features.
- StringLookup api maps string features to integer indices.
- Embedding layer API turns positive integers (indexes) into dense vectors of fixed size.



Two-tower architecture. Source: Sampling-Bias-Corrected Neural Modeling for Large Corpus Item Recommendations

```

from tf.keras import Model, Sequential
embedding_dimension = 32

user_model = Sequential([
    layers.StringLookup(
        vocabulary=unique_user_ids, mask_token=None),
    # We add an additional embedding to account for
    # unknown tokens.
    layers.Embedding(len(unique_user_ids) + 1,
        embedding_dimension)
])

movie_model = Sequential([
    layers.StringLookup(
        vocabulary=unique_movie_titles, mask_token=None),
    layers.Embedding(len(unique_movie_titles) + 1,
        embedding_dimension)
])

class MovielensModel(tfrs.Model):

    def __init__(self, user_model, movie_model):
        super().__init__()
        self.movie_model: Model = movie_model
        self.user_model: Model = user_model
        self.task: layers.Layer = task

    def compute_loss(self, features: Dict[Text, tf.
        Tensor], training=False) -> tf.Tensor:
        # We pick out the user features and pass them into
        # the user model.
        user_embeddings = self.user_model(features["user_id
            "])
        # And pick out the movie features and pass them
        # into the movie model,
        # getting embeddings back.
        positive_movie_embeddings = self.movie_model(
            features["movie_title"])

        # The task computes the loss and the metrics.
        return self.task(user_embeddings,
            positive_movie_embeddings)

```

The following are key questions we need to consider:

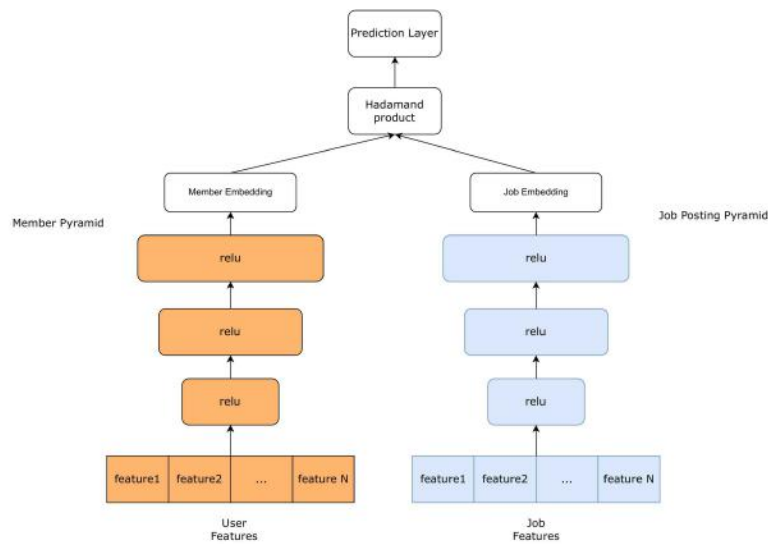
- For multi-classification where the video repository is huge, how feasible

is it approach? Solution: for each mini-batch, we sampled data from our videos corpus as negative samples. One example is to use power-law distribution for sampling.

- When sampling, it's possible that popular videos are overly penalized as negative samples in a batch. Does it introduce bias in our training data? One solution is to "correct" the logit output $sc(x_i, y_j) = s(x_i, y_j) - \log(p_j)$. Here p_j means the probability of selecting video j .
- What if an average user only watches 2% of the videos completely, the other 98% of videos they just watch a few seconds? Is it a good idea to consider all engaged videos equally important? We can introduce continuous reward rr to reflect the degree of engagement. For example: watch time.
- Why do we need to use dot product? Can we use other operators?
- How many dimensions for the embeddings?
- Does movie embedding dimension need to be the same as the user embedding dimension?
- Why do we use relu? Can we use other activation functions?

Facebook open sources their Deep Learning Recommendation Model⁵ with similar architecture.

How Does LinkedIn Train Embedding?



Pyramid two-tower network architecture/ Source: LinkedIn engineering blog

- LinkedIn used reverse pyramid architecture, which is the hidden layers growing in number of activations as we go deeper.
- LinkedIn used Hadamard product for Member Embedding and Job Embedding.
- The final prediction is a logistic regression on the Hadamard product between each seeker and job posting pair.

Example of Hadamard product:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \odot \begin{bmatrix} 5 & 3 \\ 2 & 6 \end{bmatrix} = \begin{bmatrix} 5 & 6 \\ 6 & 24 \end{bmatrix}$$

We chose the Hadamard product over more common functions, like cosine similarity, to give the model flexibility to learn its own distance function, while avoiding a fully connected layer to reduce scoring latency in our online recommendation systems.

How Does Pinterest Learn Visual Embedding

Take the Pinterest Visual Search⁶ example. When users search for a specific image, Pinterest uses input pins visual embedding and search for similar pins. How do we generate visual embedding? Pinterest used image recognition deep learning architecture, e.g., VGG16, ResNet152, Google Net, etc., to fine tune on the Pinterest dataset. The learned features will then be used as embedding for Pins. You can see an example in Chapter 10 with the Airbnb room classification use case.

We can also use collaborative filtering. Read Collaborative Filtering [\[collaborative-filtering\]](#) section.

Quiz About Two Tower Embedding

Recall that in two-tower user/movie embedding, we have the last layer of each tower as embedding. When I build a network, I decide to set the user embedding dimension to 32 and the movie embedding dimension to 64. Will this architecture work? [answer](#)

[A] Yes, as long as the model learns we can set any dimensions we want.

[B] No, a movie has too many embedding dimensions, and we will run out of memory during serving millions of movies.

[C] No, there is a shape mismatch between user embedding and movie embedding.

Application of Embedding in Tech Companies

- Twitter uses embedding for UserID, and it's widely used in different

use cases at Twitter, such as recommendation, nearest neighbor search, and transfer learning.

- Pinterest Ads ranking uses word2vec style where each user session can be viewed as: pin A \rightarrow pin B \rightarrow pin C, then co-trained with multitask modeling.
- Instagram's personalized recommendation model uses word2vec style where each user session can be viewed as: account 1 \rightarrow account 2 \rightarrow account 3 to predict accounts with which a person is likely to interact within a given session.
- YouTube recommendations uses two-tower model embedding then co-trained with multihead model architecture. (Read about multitask learning in section Common Deep Learning [1.5](#)).
- DoorDash personalized store feed uses word2vec style where each user session can be viewed as: restaurant 1 \rightarrow restaurant 2 \rightarrow restaurant 3. This Store2Vec model can be trained to predict if restaurants were visited in the same session using CBOW algorithm.

In the Tensorflow documentation, they recommend the “rule of thumb”: $d = \sqrt{4D}$ where D is the “number of categories”. Another way is to treat 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.

How Do We Evaluate the Quality of the Embedding?

There is no easy answer to this question. We have two approaches:

- Apply embedding to downstream tasks and measure their model

performance. For certain applications, like natural language processing (NLP), we can also visualize embeddings using t-SNE (t-distributed stochastic neighbor embedding), EMAP. We can look for clusters in 2-3 dimensions and validate if they match with your intuition. In practice, most embedding built with engagement optimization does not show any clear structure, UMAP (Uniform Manifold Approximation and Projection for Dimension Reduction).

- Apply clustering (kmeans, k-Nearest Neighbor) on embedding data and see if it forms meaningful clusters.

How Do We Measure Similarity?

To determine the degree of similarity, most recommendation systems rely on one or more of the following.

- Cosine: it's the cosine of the angle between the two vectors $s(q,x) = \cos(q, x)$
- Dot Product $s(q,x) = \sum_{i=1}^d (q_i * x_i)$ You will also see how LinkedIn uses Hadamard product in their embedding model (read section Embedding [\[subsec-embedding\]](#)).
- Euclidean distance $s(q,x) = \left[\sum_{i=1}^d (q_i - x_i)^2 \right]^{\frac{1}{2}}$ The smaller the value the higher the similarity.

Important Considerations

- Dot product tends to favor embeddings with high norm. It's more sensitive to the embeddings norm compared to other methods. Because of that it can create some consequences
 - Popular content tends to have higher norms, hence ends up dominating the recommendations. How do you fix this? Can you think of parameterized dot production metrics?
 - If we use bad initialization in our network and the rare content is

initialized with large values, we might end up recommending rare content over popular content more frequently.

Numeric Features

Normalization

For numeric features, the normalization must have mean 00 and range $[-1,1]$ $[-1, 1]$. There are some cases where you want to normalize data to the range $[0,1]$ $[0, 1]$.

$$v = \frac{v - \text{min_of_vv}}{\text{max_of_vv} - \text{min_of_vv}}$$
 where, vv is feature value, min_of_vv is min of feature value, max_of_vv is max of feature value.

Standardization

If the feature distribution resembles a normal distribution, we can apply a standardized transformation.

$$v = \frac{v - \text{mean_of_vv}}{\text{std_of_vv}}$$
 where, vv is feature value, mean_of_vv is min of feature value, std_of_vv is the standard deviation of feature value

If the feature distribution resembles power laws we can transform it by using the formula: $\log(1 + \frac{v - \text{median_of_v}}{\text{median_of_v}})$ In practice, normalization can cause an issue because the values of min and max are usually outliers. One possible solution is “clipping”, where we pick a “reasonable” value for min and max.

Netflix uses raw, continuous timestamp indicating the time when the user played a video in the past, a long with current time when making a prediction. They observed 30% increase in offline metrics. It leads to another challenge since the production model will always use the current timestamp, which was never observed in the training data. To handle this

situation, production models are regularly retrained.

Feature Selection and Feature Engineering Quiz

We have a table with columns UserID, CountryID, CityID, Zipcode, Age. Which of the following feature engineering is suitable to present data in machine learning algorithm?

[answer](#)

[A] Apply one hot encoding for all columns

[B] Apply embedding for CountryId, CityID; One Hot Encoding for UserID, Zipcode; apply normalization for Age

[C] Apply embedding for CountryId, CityID, UserID, Zipcode and apply normalization for Age

Summary

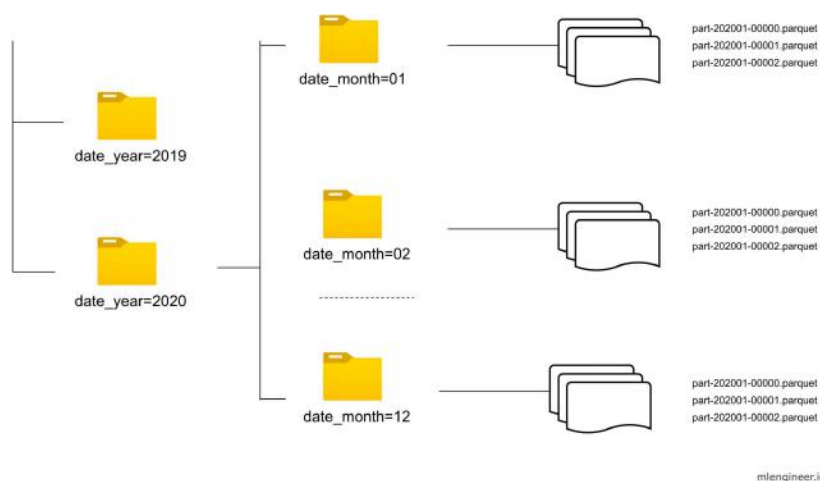
- We learn how to handle numerical features: normalization and standardization. In practice, we can often apply *log transformation* when features values are big with very high variance.
- For sparse features, there are multiple ways to handle them: one hot encoding, feature hashing, and entity embedding.
- With entity embedding, there are two popular ways to train embedding: pre-trained and co-trained. The common technique is to use engagement data and train the two-tower network model. One interesting challenge is how to select labels data when training entity embedding. We will explore some solutions in the later chapters.

Training Pipeline

Training pipeline needs to handle large volumes of data at a low cost. One common solution is to store data in a column-oriented format like Parquet, Avro, or ORC. These data formats enable high throughput for ML and analytics use cases because they are column-based. In other use cases, tfrecord (TensorFlow format for storing a sequence of binary records) data format is widely used in TensorFlow ecosystem.

Data Partitioning

Parquet and ORC files usually get partitioned by time for efficiency so that we can avoid scanning through the whole dataset. It's also beneficial for parallel training and distributed training. In this example, we partition data by year, then by month. In practice, the most common services on AWS, RedShift (Amazon fully managed, petabyte-scale data warehouse service in the cloud), and Athena (interactive query service that makes it easy to analyze data in Amazon S3 using standard SQL) support Parquet and ORC. Compared to other formats like CSV, Parquet can speed up queries by a factor of 30×30\times faster, saving 99% cost and reducing 99% data scanned.



Partition Data. Source: mlengineer.io

- Data is partitioned by year and month
- Within each partition (year = 2020, month = 02), we have all data stored in parquet format.

Handle Imbalance Class Distribution

In machine learning use cases like fraud detection, click prediction or spam detection, it's common to have imbalance labels. For example, in ad click prediction, it's very common to have 0.2% conversion rate. If there are 1,000 clicks, only two clicks lead to some desired actions, such as installing the app or buying the product. Why is this a problem? With too few positive examples compared to negative examples, your model spends most of the time learning about negative examples.

There are few strategies to handle them.

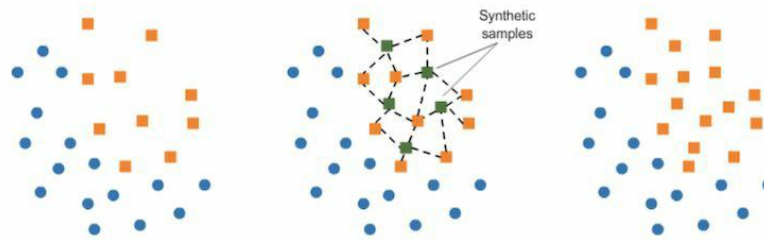
Use class weights in the loss function. For example, in spam detection problems, where non-spam data might account for 95% of data compared to other spam data that is only 5%, we want to penalize more on the major class. In this case, we can modify the entropy loss function using weight.

```
//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 major 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*: synthetic minority over-sampling 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 this 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. In practice, this method is not commonly used, especially for large-scale applications.



Resample Data. Source: imbalanced-learn.org

Common Resampling Use Cases

Due to the huge data size, it's more common for big companies like Facebook and Google to use downsampling for the dominant class. For training pipeline, if your feature store has a SQL interface, you can use the built-in `rand()` function for downsampling your dataset.

```
//sampling 10% of the data, source: nqbao.medium.com
SELECT
d.*
FROM dataset d
WHERE RAND() < 0.1
```

For deep learning models, we can sometimes use downsample as the majority class examples and then upweight them. It helps the model train faster and calibrate the model well with the true distribution.

$\text{example_weight} = \text{original_weight} * \text{downsampling_factor}$
 $\text{example_weight} = \text{original_weight} * \text{downsampling_factor}$

Quiz about Weight for Positive Class

If a dataset contains 100 positive and 300 negative examples of a single class, what should be the weight for the positive class? [Answer](#)

[A] Positive weight is $300/100=3$.

[B] Positive weight is $100/300=0.333$.

[C] It depends; can't tell.

Data Generation Strategy

When we first start a new problem that requires machine learning, especially when supervised learning is more suitable, we have to answer the question of, "How do we get labels data?"

- LinkedIn feed ranking: We can generate label data by order feeds chronologically first to collect data.
- Facebook place recommendation: We can use places people like first and then use them as positive labels. For negative labels, we can either sample all other places as negative samples or pick all places that users saw but didn't like as negative samples.

How LinkedIn Generates Data for Course Recommendation

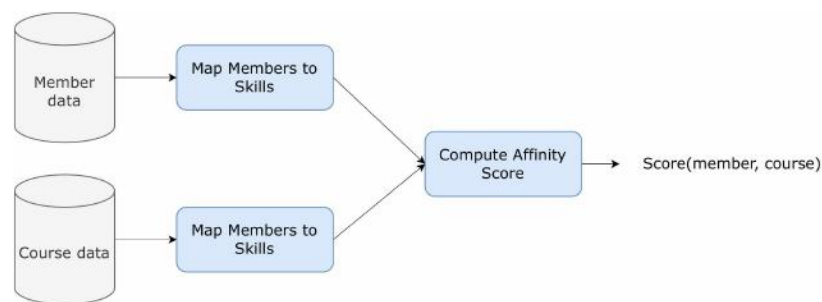
Design Machine Learning solution for Course Recommendations on LinkedIn Learning.

Problem

At the beginning, the main goal of Course Recommendations is to acquire new learners by showing highly relevant courses to learners. There are few challenges:

- Lack of label data: if we have user activities (browse, click) available, we can use these signals as implicit labels to train supervised model. As we're building this LinkedIn Learning system, we don't have any engagement signals yet. This is also called Cold start problem.
- One way to deal with it is to rely on user survey during their onboarding process, i.e: ask learners which skills they want to learn/improve. In practice, it's usually insufficient.

Let's take a look at one example: given learner Khang Pham with skills: BigData, Database, Data Analysis in his LinkedIn profile. Assume we have two courses: Data Engineering and Accounting 101, should we recommend Data Engineering or Accounting course? It's self-explained that Data Engineering would be a better recommendation because it's more relevant to this user's skillset. This lead us to one idea: we can use skills as a way to measure relevance. If we can map learners to Skills and map Course to Skills, we can measure and rank relevance accordingly.



mlengineer.io

Skill-based Model. Source: LinkedIn

Course to Skill: Cold Start Model

There are various techniques to build the mapping from scratch.

- Manual tagging using taxonomy (A). All LinkedIn Learning courses are tagged with categories. We asked taxonomist to perform mapping from categories to skills. This approach helps us acquired high precision human-generated courses to skill mapping. On the other hand, it doesn't

scale i.e: low coverage.

- Leverage LinkedIn skill taggers (B): leverage LinkedIn Skill Taggers features to extract skill tags from course data.
- Use supervised model: train a classification model such that for a given pair (course, skill): return 1 if the pair is relevant and 0 otherwise.
 - Label data: collect samples from A and B as positive training data. We then random samples from our data to create negative labels. We want our training dataset to be balance.
 - Features: course data (title, description, categories, section names, video names). We also leverage skill-to-skill similarity mapping features.
 - Disadvantage: a) relies heavily on the quality of the skill-taggers b) one single logistic regression model might not be able to capture the per-skill level effects.
- Use Semi supervised learning.
 - We learn a different model for each skill, as opposed to one common model for all (course, skill) pairs.
 - Data Augmentation: leverage skill-correlation graph to add more positive labels data. For example: if SQL is highly relevant to Data Analysis skill then we can add Data Analysis to training data as positive labels.
- Evaluation: offline metrics
 - Skill-coverage: measure how many LinkedIn standardized skills are present in the mapping.
 - Precision and Recall: we treat course to skill mapping from human as ground truth. We can evaluate our classification models using precision and recall.

Member to Skill

- Member to skill via profile: LinkedIn users can add skills to their profile by entering free-form text or choosing existing standardized skills. This mapping is usually noisy and needs to be standardized. In practice, the coverage is not high since not many users provide this data. We also train supervised model $p(\text{user_free_from_skill}, \text{standardized_skill})$ to provide a score for the mapping.
- Member to skill using title and industry: in order to increase the coverage we can use cohort-level mapping. For example: user Khang Pham work in Ad Tech industry and title Machine Learning Engineer and he didn't provide any skill set in his profile. We can rely on cohort of Machine Learning Engineer in Ad Tech to infer this user's skills. We then combine the profile-based mapping using weight combination with cohort-based mapping.

Member to skill					
Skill	Profile-based mapping	Cohort-based mapping	Weight	Weight	Final mapping
SQL	0.01	0.5	w1	w2	$0.01*w1 + 0.5*w2$
Database	0.3	0.2	w1	w2	$0.3*w1 + 0.2*w2$

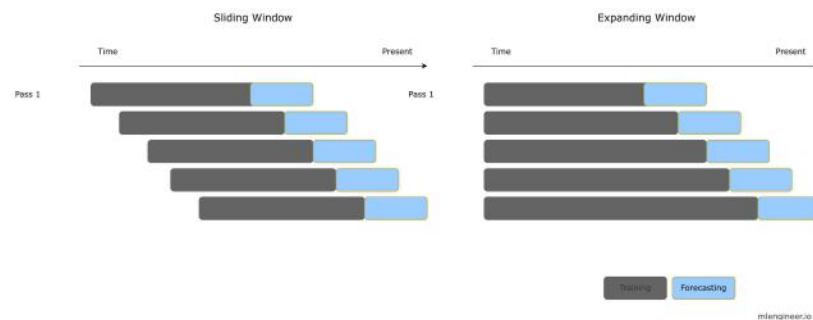
Further reading [Learning to be Relevant⁸](#)

How to Split Train/Test Data

This consideration is often overlooked but very important in the production environment.

- In forecast or any time-dependent use cases, it's important to respect the chronological order when you split train and test data.

- For example, it doesn't make sense to use data in the future to "forecast" data in the past.
- For sales forecast use case, we want to forecast sales for each store. If we randomly split data by storeID, that train data might not have data for some stores. Hence, the model can't forecast for such stores. In practice, we need to consider split data so that we can have storeId in train data as well as test data.



Uber Forecast Model Evaluation. Source: Uber

Sliding Window

- First, we select data from day 0 to day 60 as the train set and day 61 to day 90 as the test set.
- Then, we select data from day 10 to day 70 as the train set and day 71 to day 100 as the test set.

Expanding Window

- First, we select data from day 0 to day 60 as train set and day 61 to day 90 as test set.
- Then we select data from day 0 to day 70 as train set and day 71 to day 100 as test set.

Retraining Requirements

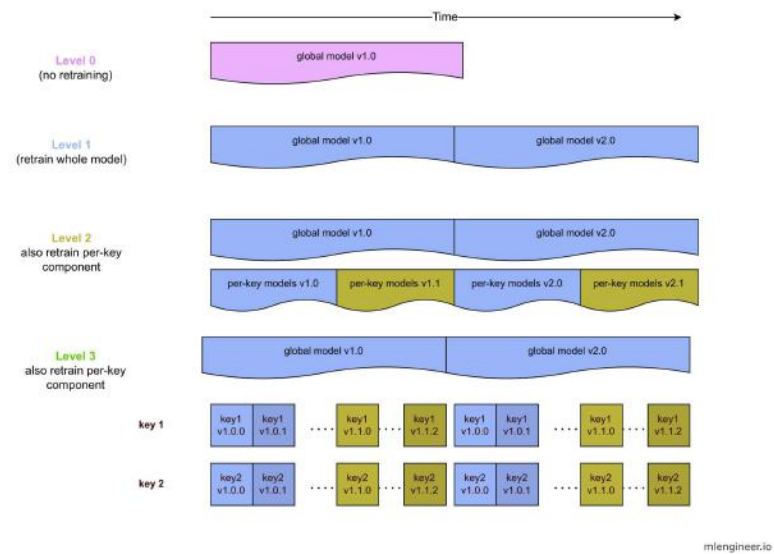
Retraining is a requirement in many tech companies. In practice, the data distribution is a nonstationary process, so the model does not perform well without retraining.

In AdTech and recommendation/personalization use cases, it's important to be able to retrain models to capture changes in users' behavior and trending topics. So the 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.

The common design pattern is to have a scheduler retrain the model on a regular basis, usually many times per day.

Four Levels of Retraining

- Level 0: *Train and forget*. Train the model once and never retrain it again. This is appropriate for the 'stationary' problem.
- Level 1: *cold-start retraining*: Periodically retrain the whole model on a batch dataset.
- Level 2: *Near-line retraining*: Similar to level 2, we retrain model per-key components individually and asynchronously nearline on streaming data.
- Level 3: *warm-start retraining*: If the model has personalized per-key components, retrain only these in bulk on data specific to each key (e.g., all impressions of an advertiser's ads) once enough data has accumulated.



Four Levels of Model Retraining - High Level. Source: LinkedIn

Loss Function and Metrics Evaluation

In this section, we will focus on Regression and Classification and use cases. Choosing loss functions and determining which metrics to track is one of the most important parts of Machine Learning products/services.

Regression Loss

Mean Square Error and Mean Absolute Error

Mean Square Error is one of the most common loss metrics in regression problems. $MSE = \frac{1}{N} \sum_{i=1}^n (\text{target}_i - \text{prediction}_i)^2$

Mean Absolute Error

$MAE = \frac{1}{N} \sum_{i=1}^n |\text{target}_i - \text{prediction}_i|$

MSE table

Actual Prediction		Absolute Square Error	
30	0	0	0
32	29	3	9
31	33	2	4
35	36.8	1.8	3.24

In this example, MAE is 1.7(6.8/4)1.7 (6.8/4) and MSE is 4.06(16.24/4)4.06 (16.24/4).

MAE table

Actual Prediction		Absolute Square Error	
30	0	0	0

32	32	0	0
31	30	1	1
50	35	15	225

In this example, MAE is $4(16/4)$ (16/4) and MSE is $56.5(226/4)$ (226/4). With one outlier value (50), it causes MSE error to increase significantly.

In practice, we always need to look for the outlier. If we have an outlier in our data, it will make the MSE loss model give more weight to the outlier than a MAE loss model. In that case, using MAE loss is more intuitive since it's more robust to an outlier.

Huber Loss

Huber Loss *fixed* the outlier-sensitive problem of MSE, and it's also differentiable at 00 (since MAE's gradient is not continuous). The idea is pretty simple: if the error is not too big, Huber loss uses MSE; otherwise, it's just MAE with some penalty.

- $\frac{1}{2}(\text{target} - \text{prediction})^2$, if $|\text{target} - \text{prediction}| \leq \delta$
- $\delta|\text{target} - \text{prediction}| - \frac{1}{2}\delta^2$, otherwise

The problem with Huber loss is that we need to tune the hyperparameter δ .

Quantile Loss

In certain applications, we value underestimation vs. overestimation differently. If you build a model to estimate arrival time, you don't want to overestimate; otherwise, customers might not make orders/requests, etc. Quantile loss can give more value to positive error or negative error.

$$\sum_{y < p} \lambda |y - p| + \sum_{y \geq p} (\lambda - 1) |y - p|$$

$$-p| + \sum_{y \geq p} (\lambda - 1) * |y - p|$$

If you set λ to 0.5, it becomes MAE.

Uber uses pseudo-Huber loss and log-cosh loss to approximate Huber loss and Mean Absolute Error in their distributed XGBoost training. Doordash Estimated Time Arrival models uses MSE then they move to Quantile loss and Custom Asymmetric MSE

It depends on the use case to decide when to use which loss function. For binary classification, the most popular one is cross_entropy. In the Ad Click prediction problem, Facebook uses Normalized Cross Entropy loss (a.k.a. log loss) to make the loss less sensitive to background conversion rate.

How Facebook Uses Normalized Cross Entropy for AdClick Prediction?

Problem: Suppose we build a machine learning model to predict click/not-click for an Ads System. We build two models: fixed prediction model and fancy model.

The fixed prediction model always predicts $\text{probability}(\text{click}) = 0.2$. The fancy model has slightly ‘better’ intuition; for positive labels, it *predicts* 0.3 and for negative labels, it *predicts* 0.1, which is intuitive and better than a random guess.

Intuitively, the fancy model should perform better because it doesn’t predict(click) with a constant value.

Fixed Prediction Model	Cross Entropy Loss
Model	Fixed Prediction

Actual	Predicted	Model Cross Entropy Loss
1	0.2	1.6094373
-1	0.2	0.22314353
-1	0.2	0.22314353
-1	0.2	0.22314353
-1	0.2	0.22314353
-1	0.2	0.22314353
-1	0.2	0.22314353
-1	0.2	0.22314353
-1	0.2	0.22314353
-1	0.2	0.22314353

The click-through rate is $\frac{1}{10}$. The overall cross entropy loss is 0.361772950.36177295

Fancy Model Cross Entropy Loss		
Actual	Model Predicted	Fancy Model Cross Entropy loss
1	0.3	1.2039728
1	0.3	1.2039728
1	0.3	1.2039728
1	0.3	1.2039728
1	0.3	1.2039728
-1	0.1	0.105360545
-1	0.1	0.105360545
-1	0.1	0.105360545
-1	0.1	0.105360545

-1	0.1	0.105360545
----	-----	-------------

The click-through rate is $\frac{1}{2}$ and cross entropy is 0.654666660.65466666.

Given smaller cross entropy loss, does the fixed prediction model perform better than the fancy model? In the two training data sets, the difference is that we have different underlying CTR. This is why Facebook and other big tech companies favor Normalized Cross Entropy⁹ (NCE).

$$\text{NCE} = \frac{\text{logloss}(\text{model})}{\text{logloss}(\text{rate})}$$

Properties of NCE:

- Always non-negative.
- Only 0 if your predictions match the labels perfectly.
- Unbounded; can grow arbitrarily large.
- Intuitive scale: $\text{NCE} < 1$: the model has learned something. $\text{NCE} > 1$: the model is less accurate than always predicting the average.

Assume a given training data set has N examples with labels $y_i \in \{-1, +1\}$ and estimated probability of click p_i where $i=1, 2, \dots, N$. The average empirical CTR as p

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

- The lower the value, the better the model's prediction.
- The reason for this normalization is that the closer the background CTR is to either 0 or 1, the easier it is to achieve a better log loss.

- Dividing by the entropy of the background CTR makes the NE insensitive to the background CTR.

In the above example, model 1 has $NCE = 0.361772950.325083 \frac{0.36177295}{0.325083} = 1.11$ and model 2 has $NCE = 0.654666660.6931472 \frac{0.65466666}{0.6931472} = 0.945$.

Forecast Metrics

In forecast problems, the most common metrics are Mean Absolute Percentage Error (MAPE) and Symmetric Mean Absolute Percentage Error (SMAPE). For MAPE, one needs to pay attention if your target value is skewed (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

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

where,

- M = mean absolute percentage error
- n = number of samples
- A_t = actual value
- F_t = forecast value

Mean Absolute Percentage Error		
Actual	Model Predicted	Absolute Percentage Error
0.5	0.3	0.4
0.1	0.9	8.0
0.4	0.2	0.5

0.15	0.2	0.334
------	-----	-------

In the second row, since the prediction is too high, we have a percentage error of 8.0. When we calculate the mean of all the errors, the MAPE metric value becomes too high and hard to interpret.

- Advantages
 - Expressed as a percentage, which is scale-independent and can be used for comparing forecasts on different scales. We should remember, though, that the values of MAPE may exceed 100%.
 - Easy to explain to stakeholders.
- Disadvantage
 - MAPE takes undefined values when there are zero values for the actual, which can occur, for example, demand forecasting. Additionally, it takes extreme values when the actual is very close to zero.
 - MAPE is asymmetric, and it puts a heavier penalty on negative errors (when forecasts are higher than actual) than positive errors. This is caused by the fact that the percentage error cannot exceed 100% for forecasts that are too low. There are no upper limits for the forecasts that are too high. As a result, MAPE will favor models that under-forecast rather than over-forecast.

Symmetric Absolute Percentage Error

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

- Advantages
 - Fixes the shortcoming of the original MAPE — it has both the lower (0%) and the upper (200%) bounds.

- Disadvantage
 - Unstable when both the true value and the forecast are very close to zero. When it happens, we will deal with division by a number very close to zero.
 - SMAPE can take negative values, so the interpretation of an “absolute percentage error” can be misleading.
 - The range of 0% to 200% is not that intuitive to interpret. Therefore, the division by the 2 in the denominator of the SMAPE formula is often omitted.

Other companies also use machine learning and deep learning for forecast problems. For example, Uber uses different algorithms like recurrent neural networks (RNNs), gradient boosting trees, and support vector regressor for various problems. Some problems include marketplace forecasting, hardware capacity planning, and marketing.

Classification Loss

In this section, we will focus more on the less popular metrics: focal loss and hinge loss.

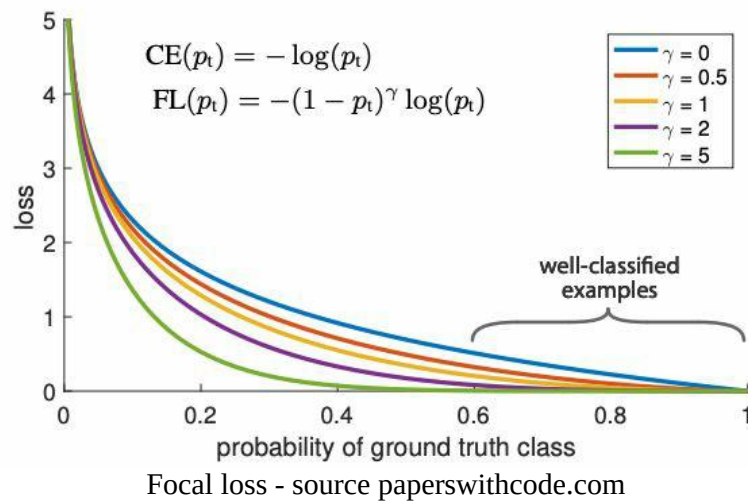
Focal Loss

When handling an imbalance class during training, a situation arises in which there are easy samples and hard samples. How can we make the model focus more on the hard examples? Focal loss¹⁰ addresses this by adding weight in such a way that if the samples are easy, the loss value is small and vice versa.

If we set γ as 0, it becomes traditional cross entropy.

$$FL(p_t) = -(1-p_t)^\gamma \log(p_t) \quad FL(p_t) = -(1-p_t)^\gamma \log(p_t)$$

When do we use it? Focal loss makes it easy for model to learn. It's popular in Object Detection.

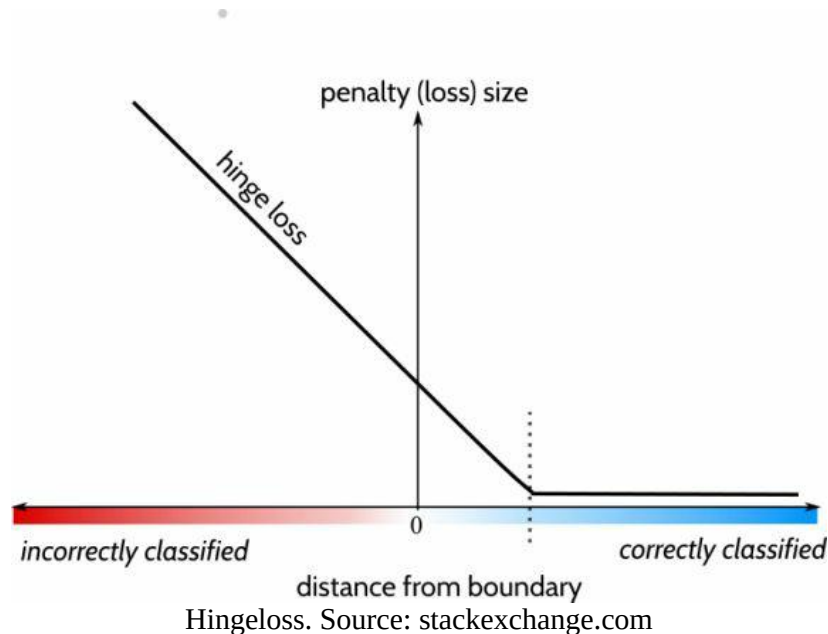


In practice, Amazon experiments with focal loss to identify optimal package size best suited to ship each product items.

Hinge Loss

The hinge loss is a special type of cost function that not only penalizes misclassified samples but also correctly classifies ones that are within a defined margin from the decision boundary.

User Rating vs. Prediction		
Actual	Predicted	Hinge loss
1	0.95	0.05
1	1.2	0.00
1	-0.3	1.3
-1	-0.9	0.1
-1	-1.05	0.0
-1	0	1.0
-1	0.5	1.5



Formally, hinge loss can be defined as:

- 0 if $y(\hat{w}x) \geq 1$
- $1 - y(\hat{w}x)$ otherwise

```
def Hinge(yHat, y):
    return np.max(0, y - (1-2*y)*yHat)
```

Or you can use TensorFlow

```
loss = tf.keras.losses.hinge(y_true, y_pred)
```

Airbnb uses Hinge loss in their machine learning library Aerosolve^{[11](#)} to support different use cases. Uber uses hinge loss with GraphSAGE^{[12](#)} for dish and restaurant recommendation^{[13](#)} for Uber Eats.

Model Evaluation

In many online applications, it's important that we improve both offline and online metrics. In this section, we will cover popular offline and online metrics.

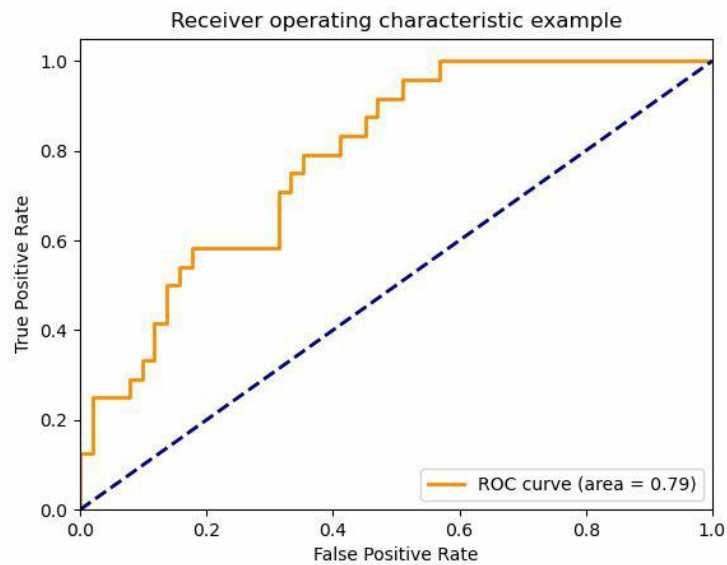
Offline Metrics

During offline training and evaluating models, we use metrics like log loss, MAE (mean absolute error), and R2 (coefficient of determination) to measure goodness of fit. Once the model shows improvement, the next step would be to move to a staging/sandbox environment to test for a small percentage of real traffic.

Area Under the Curve

Area under the curve (AUC) is a very popular metric and indicative of how a model would perform online. Receiver operating characteristics (ROC) curves typically feature true positive rate on the Y axis and false positive rate on the X axis. This means that the top left corner of the plot is the “ideal” point—a false positive rate of zero and a true positive rate of one. This is not very realistic, but it does mean that a larger AUC is usually better.

If we have a random guess binary classifier (predict 0.5 for all observations), then the AUC is 0.5.



Area Under the Curve. Source: scikitlearn

Mean Average Recall at K

User Rating vs. Prediction

Movie	UserA Predict	UserA Actual
movie1	2.1	4
movie2	2.5	3
movie3	3.5	3
movie4	3.5	N/A
movie5	4.5	5
movie6	2.2	4
movie7	2.2	4
movie8	2.2	4
movie9	2.2	4
movie10	4.8	2

Mean Average Precision (MAP)

To understand MAP, we need to review precision and recall first. In information retrieval, precision is the fraction of relevant instances over the retrieved instances.

$$\text{precision} = \frac{|\text{overlap of retrieved_documents and relevant_documents}|}{|\text{retrieved_documents}|} = \frac{|\text{overlap of retrieved_documents and relevant_documents}|}{|\text{retrieved_documents}|}$$

By default, precision takes all the retrieved documents, but it can also be evaluated where the model is only assessed by considering its top-most queries. The measure is called precision at k or $P@K$.

$$AP@n = \frac{1}{GTP} \sum_{k=1}^n P@k * \text{Relevance}@k \quad = \quad \frac{1}{GTP} \sum_{k=1}^n P@k * \text{Relevance}@k$$

where GTP refers to the total number of ground truth positives, n refers to the total number of documents you are interested in, $P@k$ refers to the precision@ k , and $\text{rel}@k$ is a relevance function. The relevance function is an indicator function, which equals 1 if the document at rank k is relevant and equals to 0 otherwise.

Mean Reciprocal Rank (MRR)

Mean Reciprocal Rank is commonly used on a listwise approach. It's more popular in the research community. It measures how far down the ranking the FIRST relevant document is. If MRR is close to 1, it means relevant results are close to the top of search results. Lower MRR indicates poorer search quality, with the right answer farther down in the search results.

Mean Reciprocal Rank				
Query	Proposed result	Correct response	Rank	Reciprocal rank
cat	catten, cati, cats	cats	3	1/3

torus	torii, tori, toruses	tori	2	1/2
virus	cviruses, virii, viri	virus	1	1

Given those three samples, we could calculate the mean reciprocal rank as $\frac{1}{3} + \frac{1}{2} + 1 = \frac{11}{6}$

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

If you care about more than one correct result, you should NOT use this metric.

In some cases, you will want to evaluate all relevant results using Mean Average Precision.

Normalized Discounted Cumulative Gain

Discounted cumulative gain (DCG) is a very popular concept in information retrieval and search engine application. Given a search result set, DCG measures the gain of a document based on its position in the result list.

There are two main assumptions about DCG:

- Highly relevant documents are more useful when appearing earlier in a search engine result list (have higher ranks).
- Highly relevant documents are more useful than marginally relevant documents, which are in turn more useful than non-relevant documents.

Cumulative Gain

Cumulative gain (CG) is the sum of the graded relevance values of all results in a search result list. $CG_p = \sum_{i=1}^p rel_i$

Properties of Cumulative Gain:

- rel_i means how relevant of the result at position i
- The value computed with the CG function is not affected by changes in the search results order. That is, moving a highly relevant document d_i above a higher ranked, less relevant document d_j does not change the computed value for CG (assuming $i \leq j$). Based on the two assumptions made above about the usefulness of search results, (N)DCG is usually preferred over CG.
- Discounted Cumulative Gain: $DCG_p = \sum_{i=1}^p rel_i \log_2(i+1)$ 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: $IDCG_p = \sum_{i=1}^p \frac{2^{rel_i} - 1}{\log_2(i+1)}$

Let's take a look at an example.

- 0: Restaurant was listed
- 1: Restaurant was clicked
- 2: Restaurant was ordered

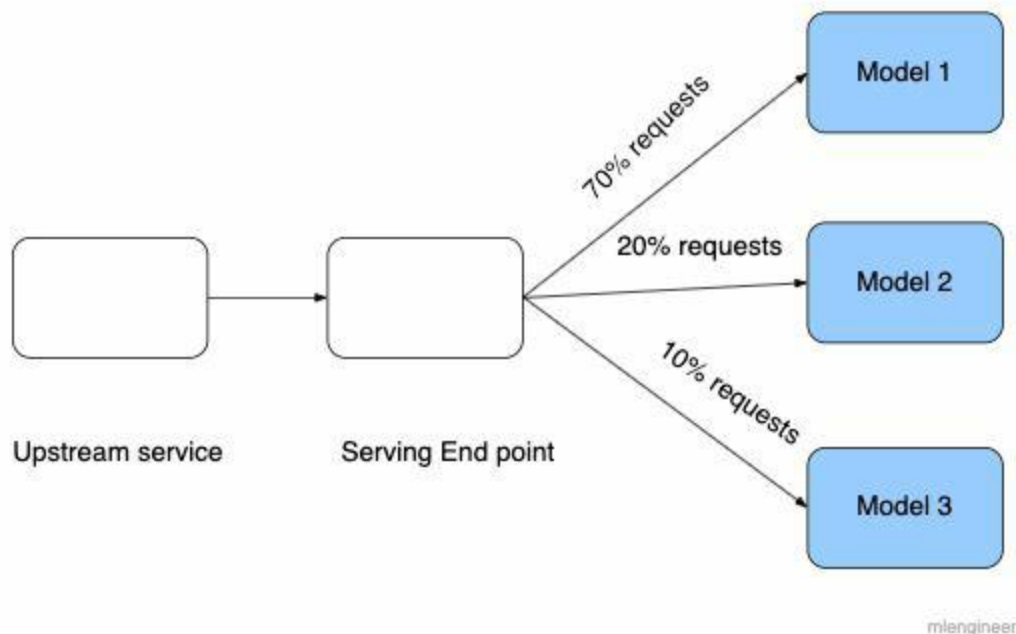
Position	Item	Relevance
1	Restaurant A	0
2	Restaurant B	1
3	Restaurant C	1
4	Restaurant D	2
5	Restaurant E	0

Using above formula, we have DCG@5 is 2.423 and IDCG@5 is 4.131. And the nDCG@5 is 0.587.

Online Metrics

During the staging phase, we measure metrics such as a 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, 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 current production models with new models. See how SageMaker 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 a few dozen models, each getting real traffic to serve online requests. This is one way to verify if a model actually generates lift in the production environment.



captionModel Service: Traffic Allocation

A/B testing is an extensive subject and use case specific. Read this highly regarded book called *Trustworthy Online Controlled Experiments: A Practical Guide to A/B Testing* to get the principals. One common technique for A/B testing is called budget-splitting (read 1.6 Budget-Splitting A/B Testing).

Other Metrics: Click-Through Rate, Time Spent

- Click-through rate (CTR) refers to the percentage of people who click on an element that they have been exposed to. The click-through rate is calculated by simply dividing the number of people who clicked on a given element by the total number of visitors to that page. CTR is a metric to analyze emails, webpages, and online advertising (Google, Bing, Yahoo, etc). CTR is normally used to measure the success of marketing efforts.
- Time spent per active user is a widely used metric that helps you understand how engaged your audience is. To measure it, we add up the time spent in the app by all users over a certain period and divide the sum by the number of active users in the same period (total time spent by all users/number of users). A common practice is to measure time spent metrics via A/B testing and cohort analysis.
- Satisfaction metrics such as rate of dismissal, user survey response, etc.

Common Sampling Techniques

Why do we need to know sampling techniques? In many classification use cases, we have positive labels without any negative labels. How do we handle that? How can we collect negative labels to train supervised classification models? While there are many novel deep learning architectures and advanced loss functions, they usually come with innovative usage of sampling techniques. For example, [14](#) uses negative sampling with noise contrastive estimation, and Twitter uses importance sampling^{[15](#)} along with wide and deep architecture to tackle delayed feedback in the recommendation. We must learn these basic sampling techniques as we will see how widely used they are in many practical use cases.

Random Sampling

This is a straightforward technique in which we can randomly select samples from a large set of data. While it sounds naive, it's still very viable for many machine learning applications. For example, to collect training data for user embedding for recommending videos, we can randomly select any not-yet-seen videos from the videos repository as negative samples. It's still a viable option, especially when we first start building the user embedding pipeline.

Rejection Sampling

Assume that we have a function `rand7()` that generates a uniform random integer in the range `[1,7][1, 7]`. Write a function `rand10()` that generates a uniform random integer in the range `[1,10][1, 10]`.

	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	8	9	10	1	2	3	4
3	5	6	7	8	9	10	1
4	2	3	4	5	6	7	8
5	9	10	1	2	3	4	5
6	6	7	8	9	10	*	*
7	*	*	*	*	*	*	*

Rejection Sampling: rand7 to rand10. Source: leetcode.com

```
def rand10(self):
    """
    :rtype: int
    """
    idx = 41
    while idx > 40:
        a = rand7()
        b = rand7()
        idx = a + (b-1) * 7
    return 1 + idx % 10
```

There is nothing special about rand7 or rand10, we can apply the same principal for other problems, such as generate rand9 from rand5, etc. In rejection sampling^{[16](#)}, all we need to have are:

- A proposal density $q(z)$ such that it's easier to sample than the original distribution $p(z)$
- $p(z)$ is difficult to sample from but we can evaluate it up to a certain proportionality constant.
- A constant k such that $kq(z) \geq p(z)$ for all values of z .

It's popular technique in practice. For Uber, they evaluate rejection sampling techniques in their text generation model^{[17](#)}.

Weight Sampling

The problem of random sampling calls for the selection of m random items out of a population of size n . If all items have the same probability, the problem is known as uniform random sampling. If each item has an associated weight and the probability of each item to be selected is determined by these item weights, then the problem is called weighted random sampling.

```
import random

def weighted_sample_without_replacement(population,
    weights, k, rng=random):
    v = [rng.random() ** (1 / w) for w in weights]
    plength = len(population)
    order = sorted(range(plength), key=lambda i: v[i])
    return [population[i] for i in order[-k:]]

rng = random.Random(42)
weighted_sample_without_replacement(population, weights,
    k, rng)
```

Python implementation[18](#)

Read more detail Sampling with weights over data stream[19](#).

Importance Sampling

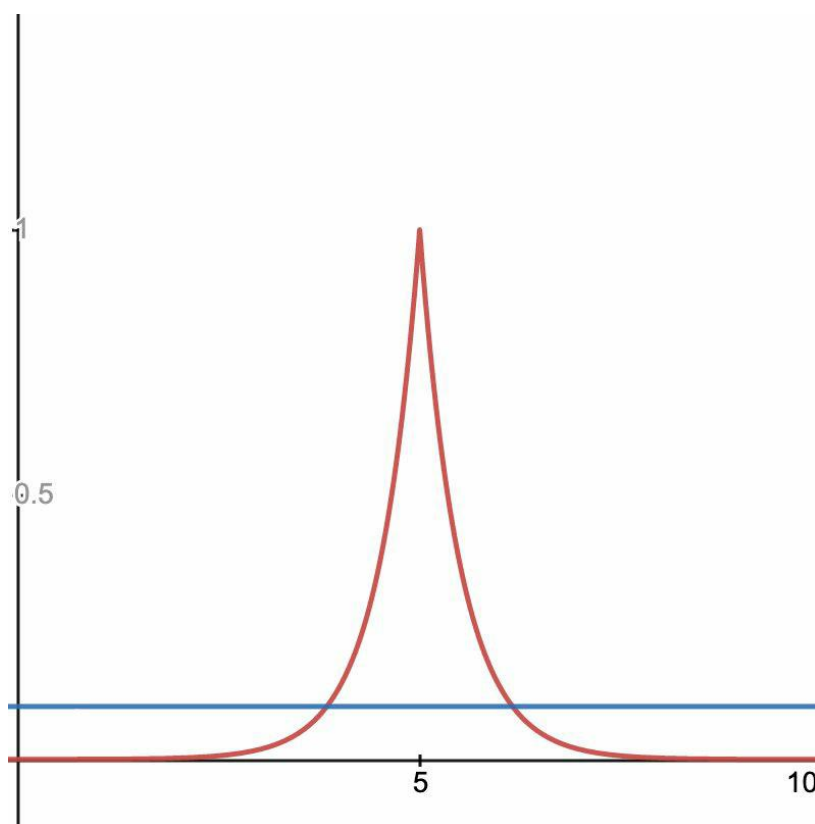
Importance sampling is a big topic with vast applications. In this section we will focus on the basic ideas and its application in modeling. Suppose we have a function $h(x)$ and we want to compute the estimation $E(h(x))$: $E_p[h(X)] = \int_R h(x)p(x)dx$

In some research, however, we didn't find any evidence of importance sampling[20](#). By applying the law of large number, we can take a large number of trials and use this formula: $\frac{1}{N} \sum_{i=1}^N h(x_i) = E_p[h(X)]$

In practice, sometimes it's very difficult or inefficient to sample from p . So

instead of sampling from p , we can sample through a proxy distribution q . The formula becomes: $\int_{\mathcal{R}} h(x) \frac{p(x)}{q(x)} dx$

and we can approximate by: $\frac{1}{N} \sum_{i=1}^N h(x_i) \frac{p(x_i)}{q(x_i)}$ Let's start looking at one concrete example.



Estimate $\int h(x) p(x) dx$ using p : High Level. Source: mlengineer.io

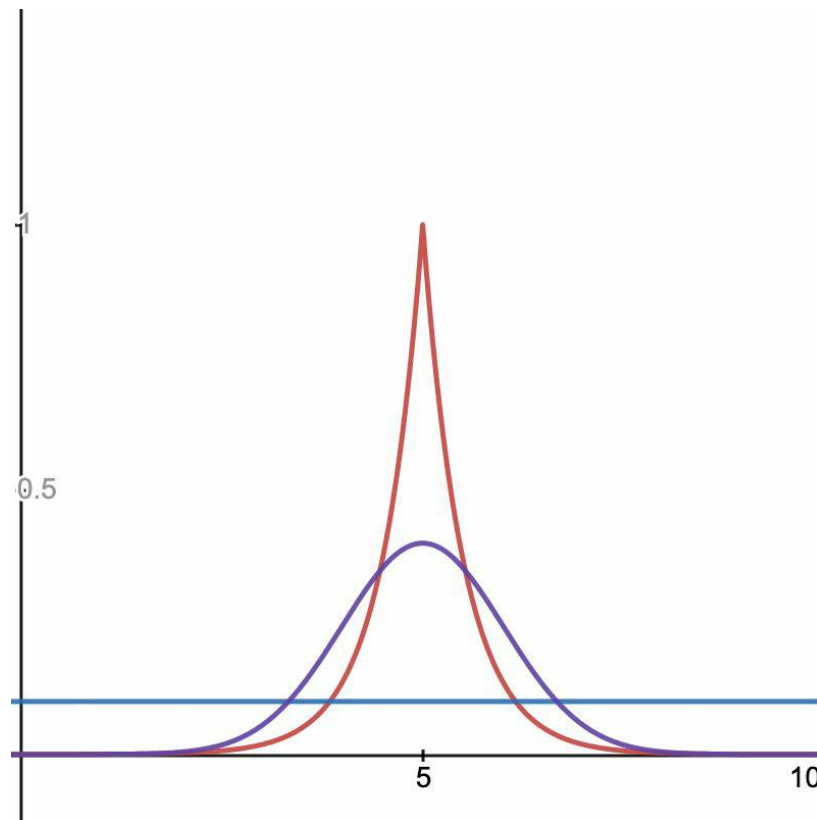
Assume we want to compute the area under the curve of h : $h(x) = \exp(-2|x-5|)$

One naive way to estimate is to use uniform distribution p where $p(x) = \frac{1}{10}$ with $X \sim \mathcal{U}(0, 10)$, then: $\int_0^{10} \exp(-2|x-5|) \frac{1}{10} dx \approx \frac{1}{N} \sum_{i=1}^N \exp(-2|x_i-5|)$

In Figure 1.17, this naive estimation is too far off. We can improve it by

using a normal distribution $q(x)$: $X \sim \mathcal{N}(5, 1)$
 $q(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x-5)^2}{2}\right)$
 then our new estimation becomes:

$$\approx \frac{1}{N} \sum_{i=1}^N h(x_i) \frac{p(x_i)}{q(x_i)}$$



Estimate h using pp . Source: mlengineer.io

Code Example

```

def f_x(x):
    return np.exp(-2*abs(x-5)))
#sampling
n = 1000
mu_target = 5
sigma_target = 0.9
value_list = []
for i in range(n):
    # sample from different distribution
    x_i = np.random.normal(mu_appro, sigma_appro)
    value = f_x(x_i)*(p_x.pdf(x_i) / q_x.pdf(x_i))
    value_list.append(value)
mean_value = np.mean(value_list)
var_value = np.var(value_list)
print("average {} variance {}".format(mean_value,
    var_value))

mu_appro = 5
sigma_appro = 1
p_x = 1/10
q_x = distribution(mu_appro, sigma_appro)
s=0
for i in range(n):
    # draw a sample
    x_i = np.random.normal(mu_target, sigma_target)
    s += f_x(x_i)
print("simulate value", s/n)
#calculate value sampling from a different distribution

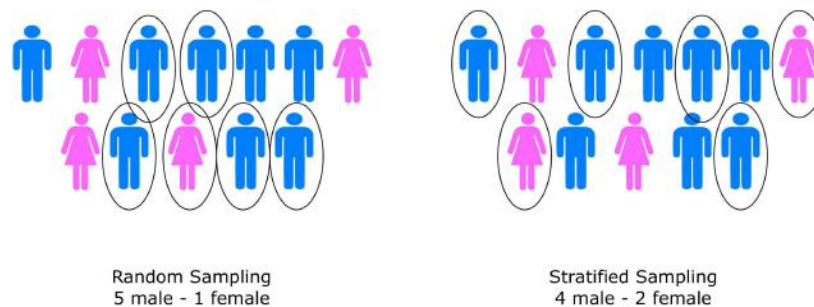
value_list = []
# need larger steps
for i in range(n):
    # sample from different distribution
    x_i = np.random.normal(mu_appro, sigma_appro)
    value = f_x(x_i)*(p_x.pdf(x_i) / q_x.pdf(x_i))
    value_list.append(value)
mean_value = np.mean(value_list)
var_value = np.var(value_list)
print("average {} variance {}".format(mean_value, np.var
    (value_list)))

```

Read more about importance sampling^{[21](#)} and importance sampling on Coursera^{[22](#)}

Stratified Sampling

Stratified sampling can be useful when you want to preserve original data distribution. In this example, the male/female ratio is 2/1. In random sampling, we might have a lot of males compared to females (5 vs. 1). If we use stratified sampling we can keep the ratio similar to the original data (4 males vs. 2 females).

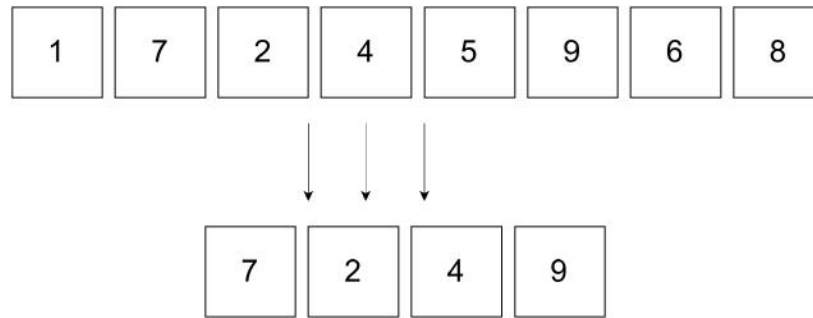


Stratified sampling. Source: mlengineer.io

```
#sample stratified sampling in scikitlearn
import numpy as np
from sklearn.model_selection import
    StratifiedShuffleSplit
stratified_sampling = StratifiedShuffleSplit(n_splits=5,
    test_size=0.5, random_state=0)
stratified_sampling.get_n_splits(X, y)
```

Reservoir Sampling

Reservoir sampling is a randomized algorithm that is used to select kk out of nn samples; nn is usually very large or unknown. For example, reservoir sampling can be used to obtain a sample of size kk from a population of people with brown hair. This algorithm takes $O(n)O(n)$ to select kk elements with uniform probability.



Reservoir Sampling. Source: mlengineer.io

#source stackoverflow: <https://stackoverflow.com/questions/2612648/reservoir-sampling>

```
import random
def random_subset( iterator, K ):
    result = []
    N = 0

    for item in iterator:
        N += 1
        if len( result ) < K:
            result.append( item )
        else:
            s = int(random.random() * N)
            if s < K:
                result[ s ] = item

    return result
```

Common Deep Learning Model Architecture

In this section, we will learn about common deep learning architectures in the recommendation system.

Wide and Deep Architecture

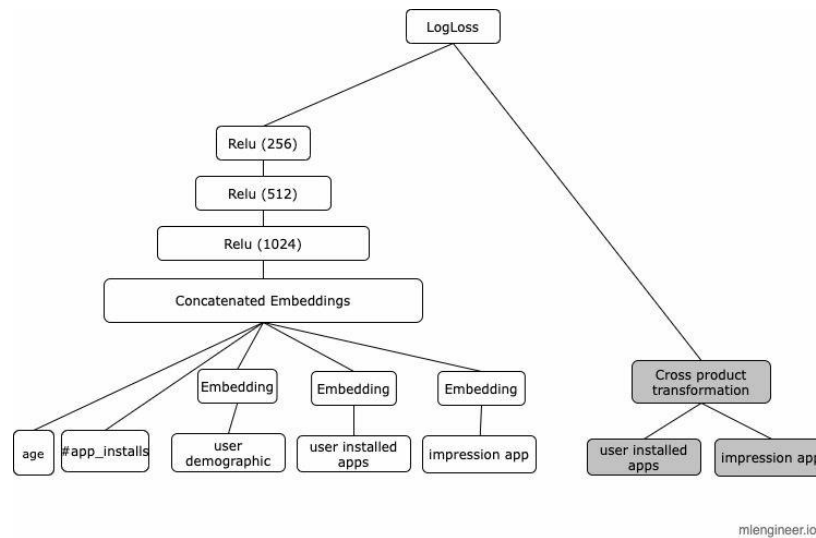
Benefits

Wide and Deep architecture can help achieve both memorization and generalization.

- Memorization helps model to learn the frequent co-occurrence of items or features and exploiting the correlation available in the historical data.
- Generalization helps model to explore new feature combinations that rarely occurred in the past.

In the Wide and Deep paper^{[23](#)}, Google showed good model performance in apps recommendation problem for the Google Play store.

Architecture



Wide and Deep Architecture

- Deep model: use embedding layers for categorical columns like user demographic, user installed apps, etc.
- Wide model: apply for feature: user installed apps, impression apps (apps that users saw but didn't install) etc.

In this example, we use Keras API in TensorFlow to demonstrate how to build wide and deep architecture.

```
from tf.keras.optimizers import Ftrl
from tensorflow.keras.layers import Dense
from tf.compat.v1.keras.experimental import LinearModel,
    WideDeepModel
epochs = 10
linear_model = LinearModel()
dnn_model = keras.Sequential([Dense(units=64),
    Dense(units=1)])
combined_model = WideDeepModel(linear_model, dnn_model)
combined_model.compile(optimizer=['sgd', 'adam'], 'mse',
    ['mse'])
# define dnn_inputs and linear_inputs as separate numpy
# arrays or
# a single numpy array if dnn_inputs is same as
# linear_inputs.
combined_model.fit([linear_inputs, dnn_inputs], y,
    epochs)
# separate tensors for dnn_inputs and linear_inputs.
dataset = tf.data.Dataset.from_tensors([linear_inputs,
    dnn_inputs], y))
combined_model.fit(dataset, epochs)
```

Two-Tower Architecture

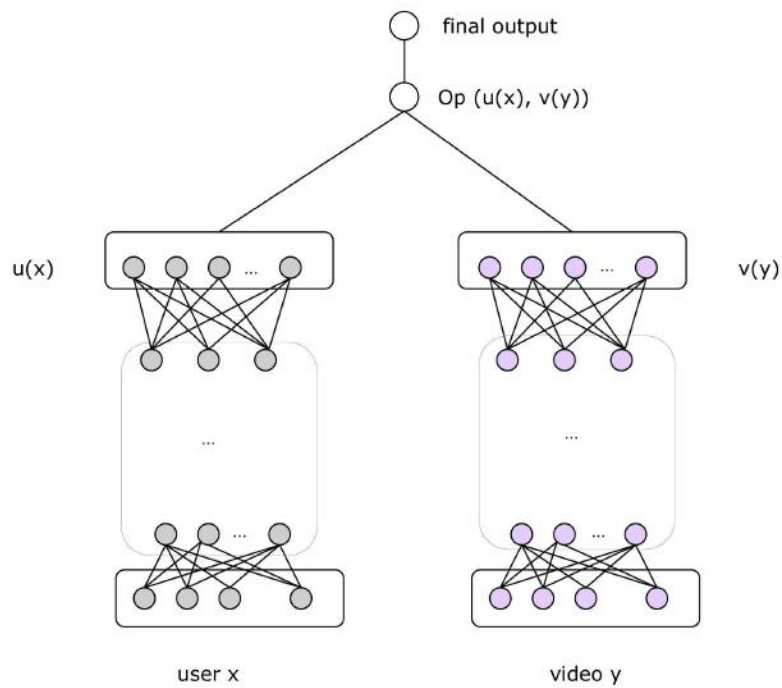
Given input data of user, context (time of day, user's device) and items, a common solution to build a scalable retrieval model is:

- Learn query and item representations for user, context and item respectively.
- Use a simple scoring function (e.g., dot product) between query and item representations to get recommendations tailored for the query.

The representation learning problem is typically challenging in two ways:

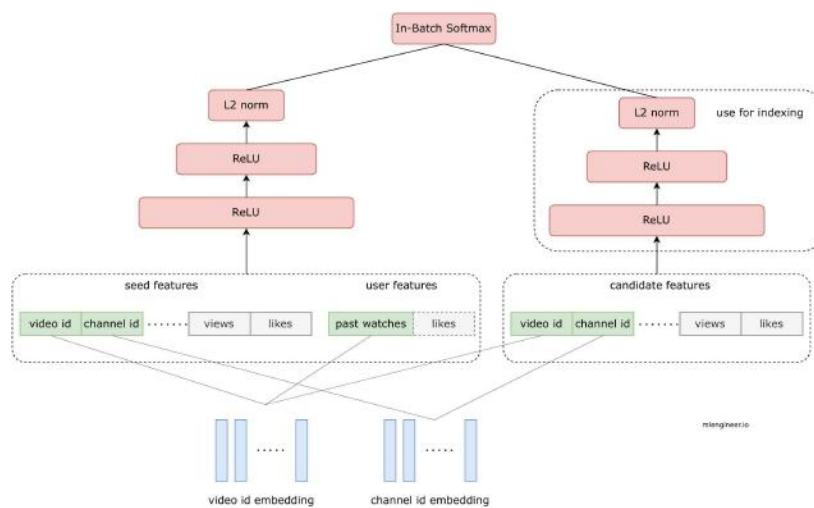
- There are many items in the corpus: hundreds of millions of YouTube videos, millions of Places on Facebook etc.
- Long-tail problem: training data collected from users' feedback is very sparse for most items.

You can find more details in YouTube Watch Next recommendation paper²⁴



mlengineer.io

Two-tower Architecture. Source: YouTube



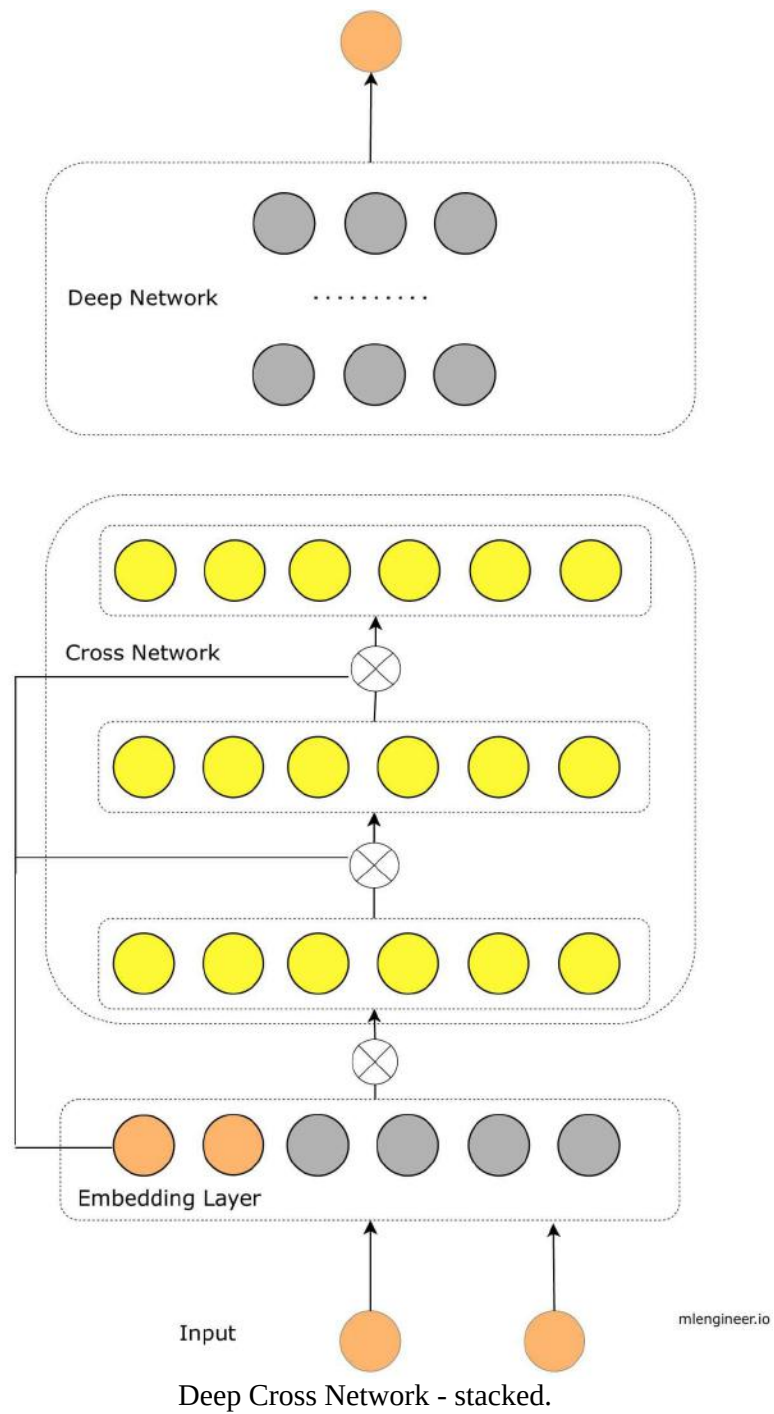
mlengineer.io

YouTube Video Retrieval Model. Source: Google

Deep Cross Network

As we discussed in the Cross Features [\[crossed-features\]](#) section, we often have a lot of sparse features in web applications. For example, in an ecommerce website, we might have `purchased_bananas``purchased_bananas` and `purchased_cooking_books``purchased_cooking_books` features in our dataset. If a customer purchased both bananas and a cookbook, then this customer will likely be interested in purchasing a blender. We also see how Wide and Deep architecture can take advantage of cross features in [\[sec-wide-and-deep\]](#) section. In practice, there can be hundreds or thousands of features which make is very difficult to decide which features to use in the "wide" input of the network.

Deep Cross Network V2²⁵ (DCN) was designed to learn explicit cross features more effectively.



- There are two networks: Cross Network and Deep Network
- Cross Network: It applies feature crossing at each layer, and the highest polynomial degree increases with layer depth.

- Deep Network: It is a traditional feedforward multilayer perceptron (MLP).

In this example, we have built a DCN v2 model with stacked architecture is built (Source: TensorFlow).

```

from tensorflow.keras import Sequential
from tf.keras.layers import Dense
import tensorflow_recommenders as tfrs
from tensorflow.keras.metrics import RootMeanSquaredError
from tensorflow.keras.losses import MeanSquaredError
from tf.keras.layers import StringLookup, IntegerLookup,
    Embedding
class DCN(tfrs.Model):

    def __init__(self, use_cross_layer, deep_layer_sizes,
        projection_dim=None):
        super().__init__()

        self.embedding_dimension = 32

        str_features = ["movie_id", "user_id", "user_zip_code",
            "user_occupation_text"]
        int_features = ["user_gender", "bucketized_user_age"]

        self._all_features = str_features + int_features
        self._embeddings = {}

        # Compute embeddings for string features.
        for feature_name in str_features:
            vocabulary = vocabularies[feature_name]
            self._embeddings[feature_name] =
                Sequential(
                    [StringLookup(
                        vocabulary=vocabulary,
                        mask_token=None),
                     Embedding(len(vocabulary) + 1,
                        self.embedding_dimension)
                    ])

        # Compute embeddings for int features.
        for feature_name in int_features:
            vocabulary = vocabularies[feature_name]
            self._embeddings[feature_name] =
                Sequential(
                    [IntegerLookup(
                        vocabulary=vocabulary,
                        mask_value=None),
                     Embedding(len(vocabulary) + 1,
                        self.embedding_dimension)
                    ])

        if use_cross_layer:
            self._cross_layer = tfrs.layers.dcn.Cross(
                projection_dim=projection_dim,
                kernel_initializer="glorot_uniform")
        else:
            self._cross_layer = None

        self._deep_layers = [Dense(layer_size, activation="relu"
            )
            for layer_size in deep_layer_sizes]

        self._logit_layer = Dense(1)

        self.task = tfrs.tasks.Ranking(
            loss=MeanSquaredError(),
            metrics=[RootMeanSquaredError("RMSE")]
        )

    def call(self, features):
        # Concatenate embeddings
        embeddings = []
        for feature_name in self._all_features:
            embedding_fn = self._embeddings[feature_name]
            embedding = embedding_fn(features[feature_name])
            embeddings.append(embedding)

        x = tf.concat(embeddings, axis=1)

        # Build Cross Network
        if self._cross_layer is not None:
            x = self._cross_layer(x)

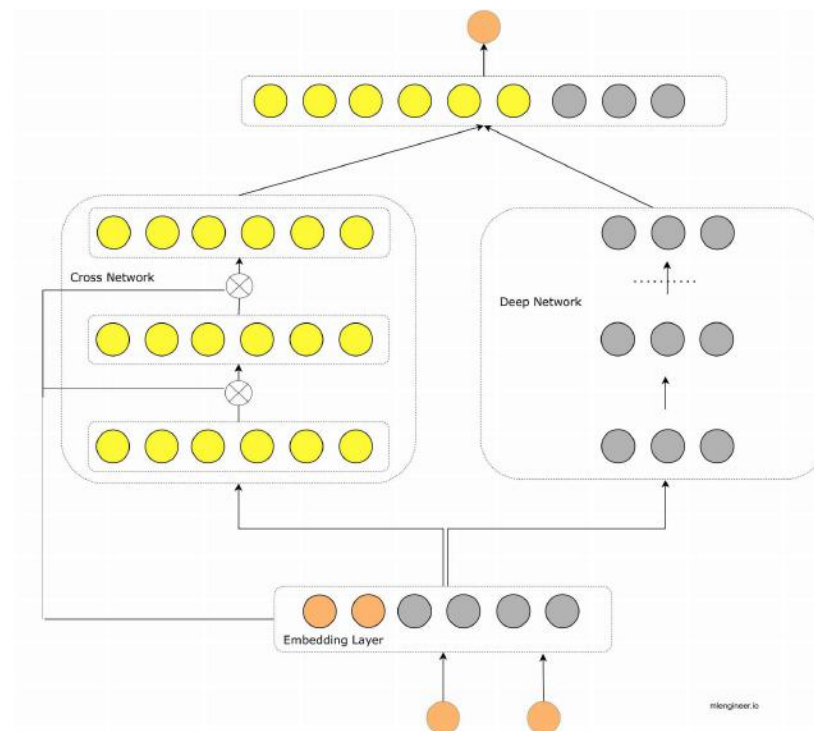
        # Build Deep Network
        for deep_layer in self._deep_layers:
            x = deep_layer(x)

        return self._logit_layer(x)

    def compute_loss(self, features, training=False):
        labels = features.pop("user_rating")
        scores = self(features)
        return self.task(
            labels=labels,
            predictions=scores,
        )

```

You can also combine deep layers and cross layers with parallel architecture.



Deep Cross Network with parallel architecture

Benefits

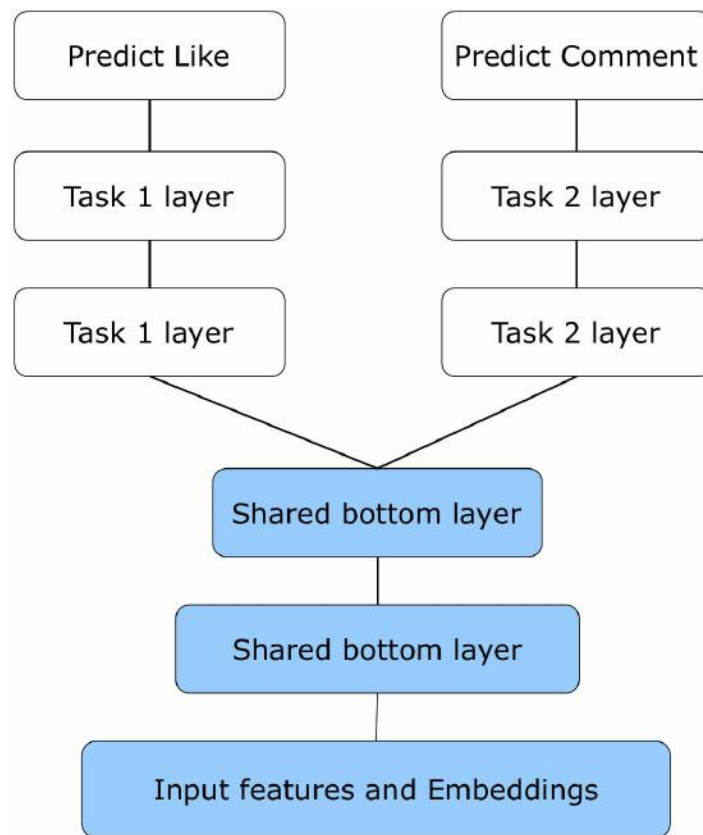
- DCN v2 automatically learns cross features and improves model performance compared to traditional DL architecture.
- Google uses DCN v2 in their production use cases. When compared with A production model, DCN-V2 yielded 0.6% AUCLoss (1 - AUC) improvement. For this particular model, a gain of 0.1% on AUCLoss is considered a significant improvement.
- It's better to insert the cross layers in between the input and the hidden layers of DNN (stacked-based).
- In practice, using two cross-layers yields the best model performance.

- Using low-rank DCN with rank $(\text{input size})/4$ can preserve model accuracy.

Multitask Learning

In this section, we will review how YouTube build "Watch Next" recommendation system, specifically the ranking stage.

Architecture



mlengineer.io

Multitask Learning. Source: mlengineer.io

Benefits

- Share bottom layers help us leverage learned representation from other tasks.
- Train model and ability for multiple tasks.
- We can also combine $p(\text{like})$ and $p(\text{comment})$ in the ranking phase, e.g, rank items based on weighted combination of $p(\text{like})$ and $p(\text{comment})$.
- How can you “blend” results together? For example, videos and photos have very different distribution for possible actions (like/comment). Read more Lessons learned at Instagram²⁶.

How Instagram “blend” video and photos in their recommendations? Instagram maps the probability (such as $p(\text{like})$) to a reasonably well-behaved distribution such as a Gaussian. Now it’s straightforward to compare and blend the results.

Read more detail about Recommending What Video to Watch Next: A Multitask Ranking System²⁷.

Facebook Deep Learning Recommendation Model (DLRM)

In this section, we will discuss how Facebook builds personalized and recommendation systems.

Requirements and Data

Recommend products to people, for example, which qualified ads to show to Facebook users. In this problem, we want to maximize the engagements, such as increasing the Click-through rate (CTR).

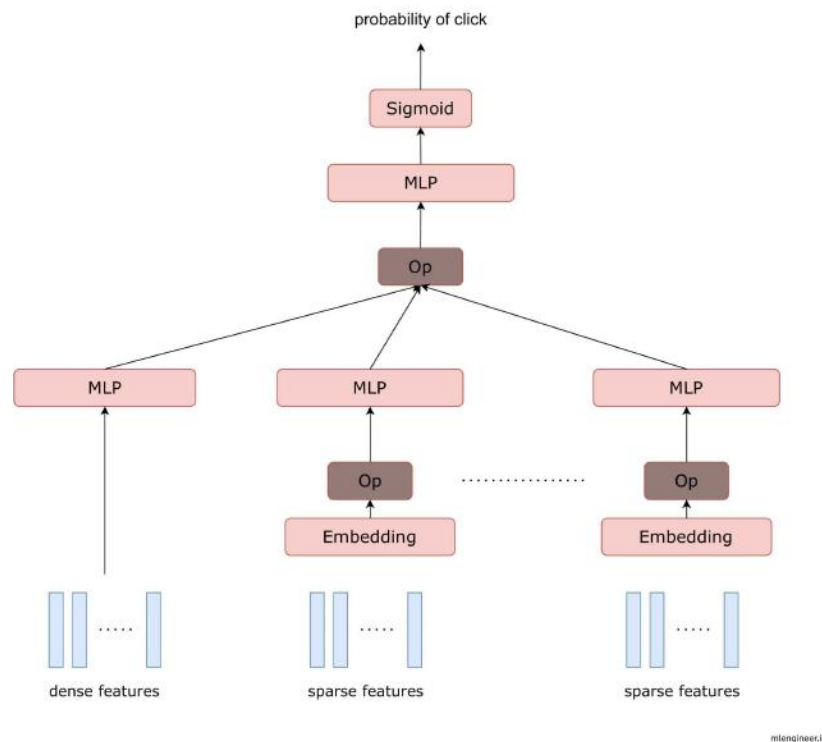
Metrics

We utilize accuracy, AUC since our use case is binary classification: if the user clicks or does not click on a particular product.

Features

- Categorical features: went through embedding layers with the same dimensions as we described in [\[subsec-embedding\]](#) section.
- Continuous features: went through a multilayer perceptron (MLP), which will yield a dense representation of the same length as the embedding vectors.

Model



- Op here can be concat, dot product or sum.
- After all features are transformed through MLP, we apply Op and the

output will be transformed through another MLP.

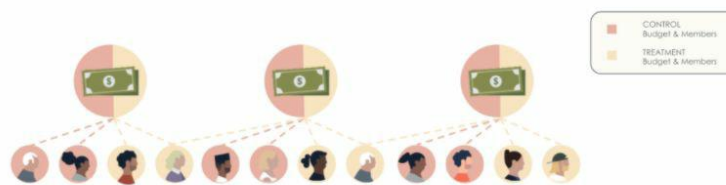
- Finally, we fed the output through the Sigmoid layer.

Further reading: Deep Learning Recommendation Model for Personalization and Recommendation Systems^{[28](#)}

A/B Testing Fundamental

There are two popular ways to perform A/B testing: general A/B testing and budget-splitting A/B testing²⁹. In normal A/B testing, users are split into two groups: control and treatment. The control group contained users with the current production model and the treatment group contained users with the new model. One problem with this setting is in Ads Marketplace industry, it's common to see control groups and treatment groups "fight" for the same budget. So during the test, when we observed increase "revenues" it can simply mean budget "shift" between groups. One solution is budget-splitting.

Budget-Splitting



Budget split A/B testing. Source: LinkedIn

- First, we split members into control group and treatment group. Each group has the same number of members.
- Then, we split the budget of each ad campaign into two “sub-campaigns”. Each “sub-campaign” has 50% of the original campaign’s budget.
- Finally, we assign one of these sub-campaigns to the treatment member group and the other to the control member group.

Benefits

- The two sub-campaigns act independently, so they can’t compete for budget between treatment and control members.

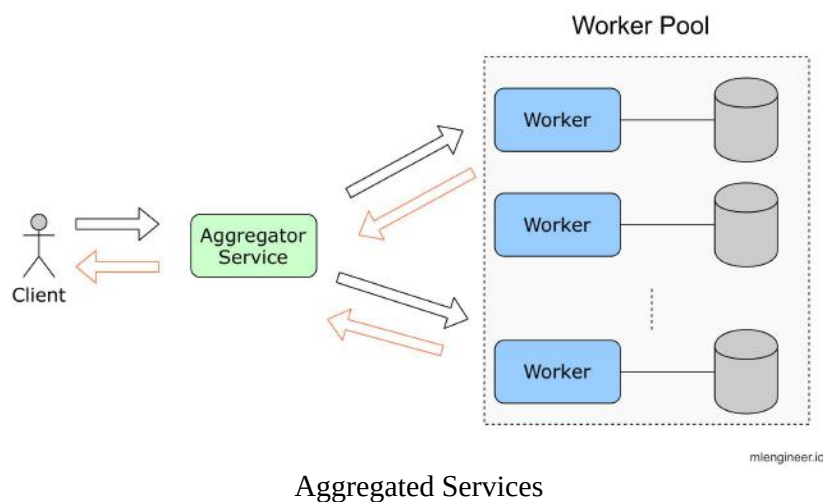
- These tests run with a large member population, which improves the statistical significant of our experiment.

You can read more about Budget-splitting A/B test [30](#).

Common Deployment Patterns

Imbalance Workload

During inference, one common pattern is to split workloads to multiple inference servers. We usually use similar architecture in Load Balancer, which is also sometimes called Aggregator Service or Broker.

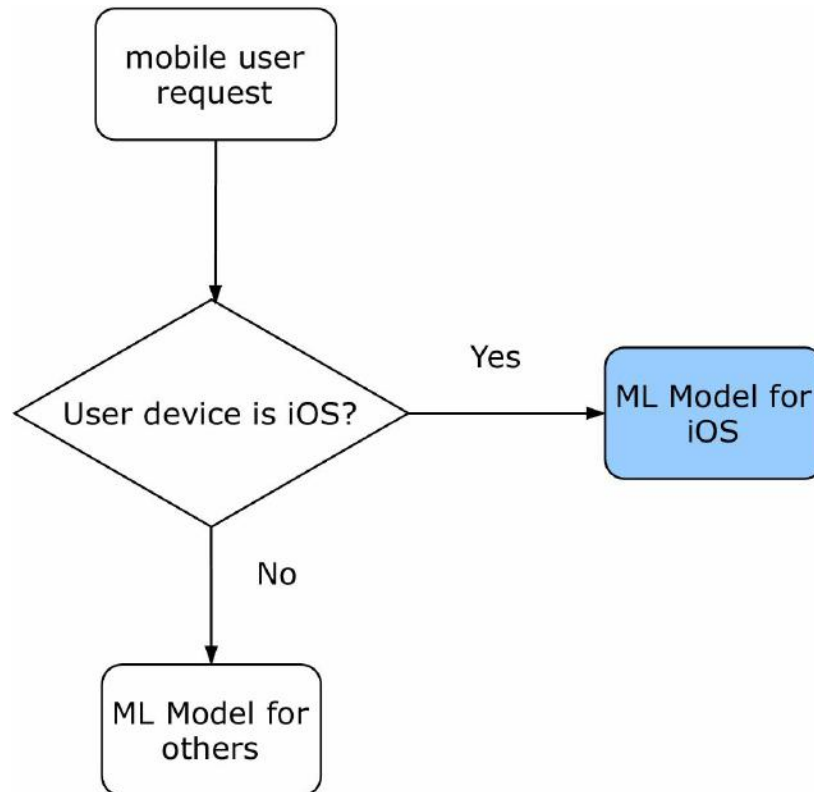


- Client (upstream process) sends request to Aggregator Service. If workload is too high, Aggregator Service splits the workload and assigns to some workers in the Worker pool. Aggregator Service can pick workers by workload, round-robin, or request parameter.
- Wait for response from workers.
- Forward 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, depending on the type of Ads candidates, we will route to a different model to get a score. In practice, we can sometime

see multiple models, each model serve some specific cohorts of users. The disadvantage is that it increases complexity, and we might not have enough data to train multiple models.



mlengineer.io

Serving Logic. Source:mlengineer.io

Serving Embedding

In this section, we look at how LinkedIn uses embedding during inference (batch mode and near real-time)

At a high level, there are three main components:

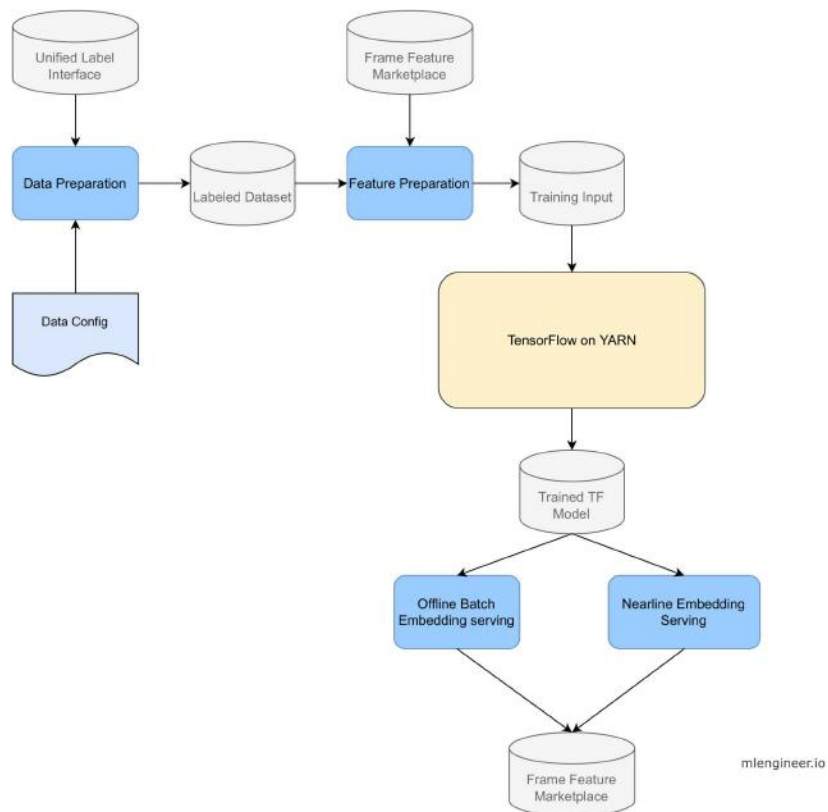
- Offline training pipeline: scale to hundreds of millions of data instances for training; ability to join millions of observations, and support distributed training on YARN (Yet Another Resource Negotiator).
- Modeling: we use pyramid two-tower architecture. Read Pyramid

architecture [\[linkedin-pyramid\]](#) section.

- Serving: optimize for high write throughput and fast experimentation velocity. Supports both offline and nearline serving.

Component	Input	Output>	Objective
Offline training pipeline	Join millions of observations for any sparse features from feature marketplace	Training pipeline, saved model	Fast experimentation when scaling training to hundreds of millions of instances
Modeling	Entity sparse features	Entity embeddings are stored in key-value storage and subgraphs are packaged/versioned and stored in cloud storage	Ability to handle extremely high cardinality features using two-tower architecture
Serving	Observations: new job posting, new members, etc., computed embedding, and high write throughput and fast experimentation velocity		

High-level Architecture



LinkedIn Pensive Architecture. Source: LinkedIn Engineering Blog

Offline Serving

- Once the model is trained, we have two subgraphs: one subgraph for the user and one subgraph for the job posting.
- The subgraphs are versioned, packaged, and distributed for offline serving.
- Then, we stored precomputed embeddings in Frame Feature Marketplace.

Nearline Serving

- When a job is posted, a nearline feature standardization process runs and produces input for the embedding model.

- Once standardization processes are completed, all registered models are loaded and executed in parallel.
- Once we have the embeddings, we batch and store them in key-value storage and Kafka topic to be published in Frame Feature Marketplace.

Component	Description	Trade-offs
Training pipeline	Offline training or warm-start retraining	See more detail in section Retraining [retraining]
Label generation	Positive labels: user engage with content or user save jobs. Negative: any items that users do not engage with	See more detail in section Data Generation [data-generation]
Network architecture	Two-tower with pyramid structure.	See more detail in section Common Deep Learning 1.5
Offline training pipeline	Join millions of observations for any sparse features	Training pipeline, saved model

from feature
marketplace

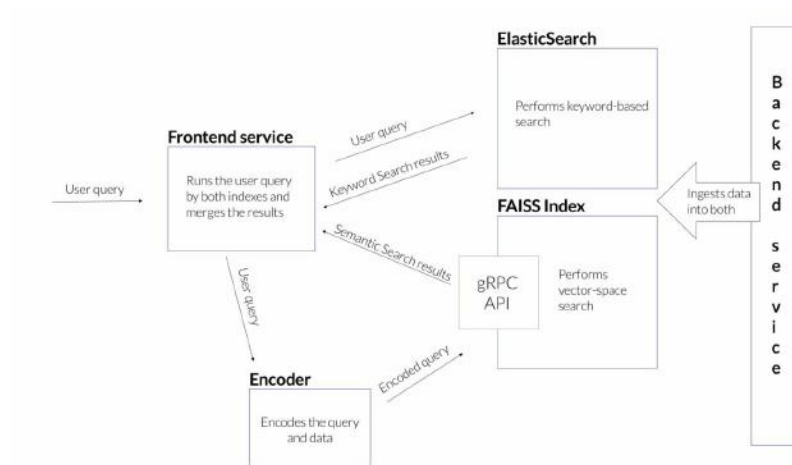
Approximate Nearest Neighbor Search

One popular application of embedding is approximate nearest neighbor search (ANN). For a large dataset with hundreds of millions or billions of items (e.g., users, movies, etc.), searching for similar objects based on their embeddings is challenging. There are two common libraries: Facebook AI Similarity Search (FAISS^{[31](#)}) and Google ScaNN^{[32](#)}.

How Onebar Uses ANN for Their Search Service

We want to build a search service Onebar^{[33](#)} that can support both keyword-based search and semantic search.

- Keyword-based search: users type a query and we use Elastic Search to return results.
- Encoder: The Universal Sentence Encoder^{[34](#)} from tensorflow-hub is used, and run as a REST service.
- Building a word embedding index using FAISS APIs.
- Semantic Search: return search results using word embedding similarity. In this example, we can dockerize FAISS and run it as a standalone service using gRPC as protocol.



Semantic Search with FAISS. Source: onebar.io

Further thoughts:

- How do we scale user embedding in near real-time?
- How do we generate embedding for new users?

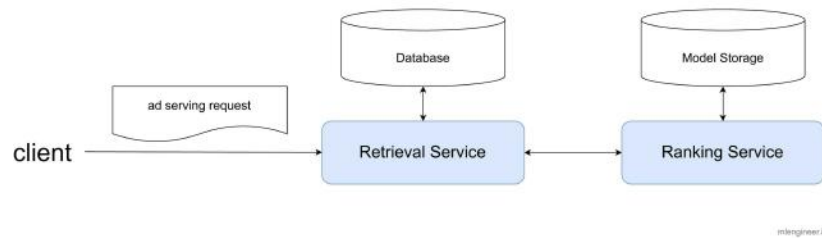
Component Description

Training method	Offline training or warm-start retraining (See 1.2 Training Pipeline for more details)
Label generation	Option 1: Engagement labels. pPositive labels are items that users engage with. Negative labels are items that users didn't engage with Option 2: Session-based (i.e., positive

	if items co-occurred in the same session)
Features	User related features: demographic, etc. Item related features: text, image, etc.
Network architecture	Two-tower architecture and pyramid two-tower architecture
Activations	Relu, LeakyRelu, Mish
Loss functions	Cross entropy loss or similar loss
Embedding dimensions	Decide based on model performance
Image embedding	Use pretrained Resnet 50 and use last layer values as embedding
Handle unseen data: new user	Use low-level feature (demographic, historical engagement) Or hash userid into buckets

Deployment Example

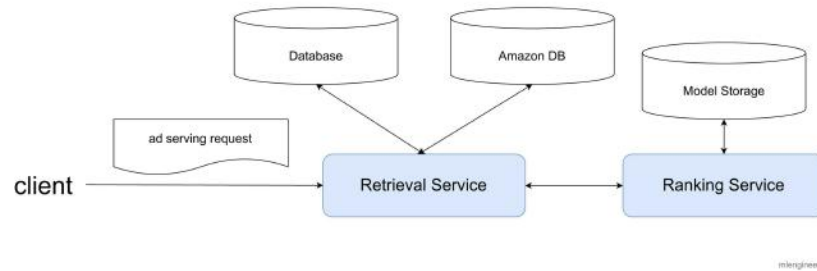
In this example, an end-to-end building block for the Ad ranking system is shown.



High level deployment diagram (logical) for ad ranking system

- Database: stores ad campaign data, ad creative data, and user information, etc.
- Model Storage: stores persisted machine learning model. One example is to store a trained model in AWS S3 with timestamps so the ranking service can pull the latest model.
- It's desirable to have a separated retrieval service and ranking service because they usually have different scalability requirements. In practice, they are usually built by different teams with different technical stacks.

In Ad ranking use cases, we typically have less frequently changed features (user home country) and frequently changed features (number of clicks in the last 3 hours, last 24 hours, etc). One way to update near real time features is to use Amazon DynamoDB.

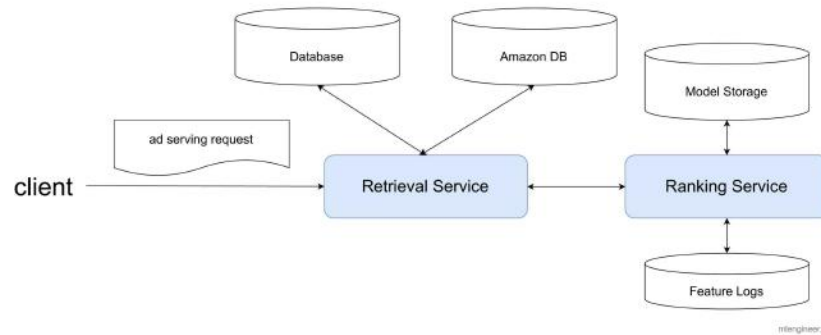


Ad ranking system design with near real-time feature storage. Source: mlengineer.io

Why don't we prepare input data in the ranking service?

- Pros:
 - Make it easier to release versioned and ship models. With a framework like TensorFlow Transform, we can encapsulate all feature processing steps in TensorFlow Graph.
 - Make it easier to change the backend database without affecting any machine learning models. Decouple the retrieval service with ranking service, therefore, improving overall system throughput and latency.
- Cons: Prone to training-serving skew, for example: data is processed differently between training and serving.

How do we handle training-serving skew? One simple solution is to log all processed data into log files.



Ad ranking with feature logging. Source: mlengineer.io

What if the logged files are too big, and we run out of disks? One simple solution is to use log rotating. In Kubernetes, you can redirect any output generated to a containerized application's stdout and stderr streams. For example, the Docker container engine redirects those two streams to a logging driver, which is configured in Kubernetes to write to a file in JSON format. There are three options to implement this:

- Option 1: Use a node-level logging agent that runs on every node.
- Option 2: Include a dedicated sidecar container for logging in an application pod.
- Option 3: Push logs directly to a backend from within an application.

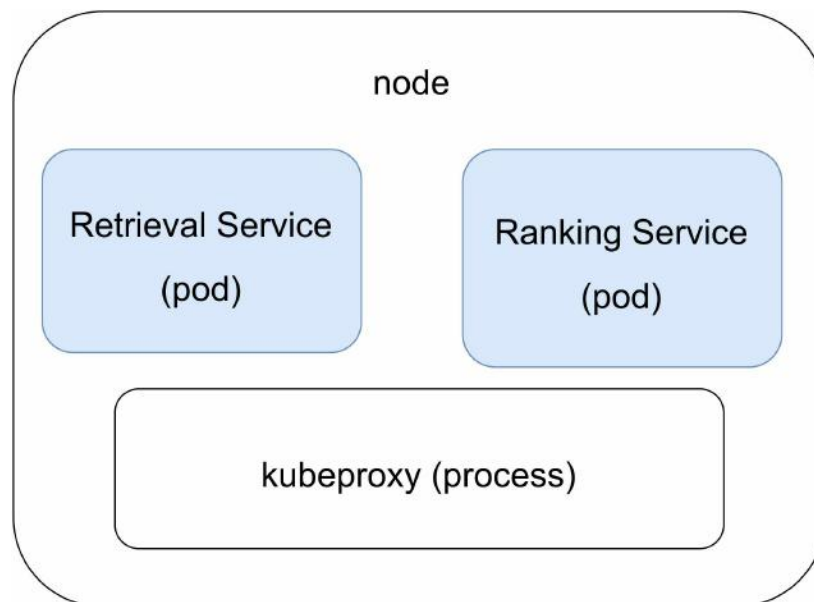
How do we scale the retrieval service? What if we have too many requests?

- Option 1: Manually tune a fixed number of pods/instances on the cluster
- Option 2: Use Kubernetes autoscaling. We have two options: 1. Demand base if requests > threshold then scale up; scale down using cool down events. 2. Other auto scaling modes based on CPU or memory metrics are less effective.

To make autoscaling work, we need to have:

- Retrieval service and other down-stream tasks are stateless.
- All services must support graceful shutdown otherwise, we won't know how to store states.
- Service should have similar performance instances.
- Load balancer is NOT a single point of failure. It's usually downstream services that are the bottlenecks.

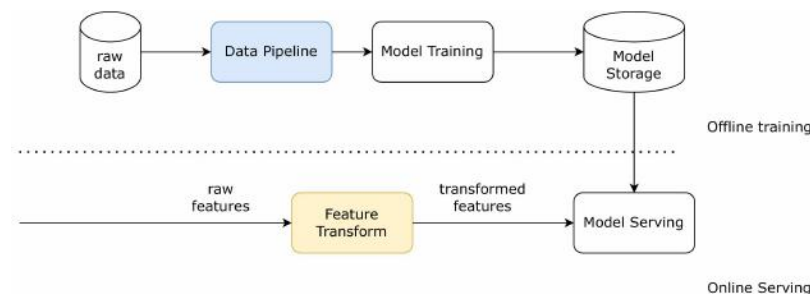
Concretely, one Kubernetes pod will look like this:



Ad Ranking: Kubernetes Pods/ Source: mlengineer.io

Spotify: one simple mistake took four months to detect

In this section, we learn how Spotify built The Podcast Model: predicts podcasts a listener is likely to listen to in the Shows you might like shelf.



Spotify PodCast Model: high level/ Source: mlengineer.io

How to make prediction with the wrong data?

- Training pipeline reads raw data, performs feature transformation, train model and save model to Model Storage.
- Online serving: a recommendation service (run on different infrastructure), transform feature then give predictions.
- Problem arises when Feature Transform in Online Serving is implemented differently.
- As a result, input data is different during training vs. serving. This training-serving skew³⁵ problem is very common in practice, especially when serving is implemented by different teams.
- The discrepancy implementation is just a few lines of code but it impacts model recommendations severely.
- This issue happened for four months before they can detect it.

Solutions

There are two main solutions: 1/ one transformation for both training and serving and 2/ use feature logging. Spotify decides to use feature logging (see [1.5](#) diagram).

- Implement feature transform in Java.
- Log already transformed features at serving stage. This is important because the up-stream service used for feature transform is owned by different teams. If we log features at up-stream services, it'll be difficult for Machine Learning team to own and make any changes.
- Feature logs will be used later for training.
- Use Tensorflow Data Validation (TFDV) to compare training and serving data schemas and feature distribution regularly.
- Set up alert to detect feature drift (using Chebyshev distance metric).

Chapter Exercises

Quiz 1: Quiz on Cross Entropy

Compute the Categorical cross-entropy metric for this classifier? (*answer at foot of page.*) Using this formula

$$-y\log p - (1-y)\log(1-p) - y\log p - (1-y)\log(1-p)$$

Actual label: 1 mean true label

ID	Apple	Pear	Orange
sample 1	1	0	0
sample 2	1	0	0
sample 3	1	0	0

Prediction table

ID	pApple	pPear	pOrange
sample 1	0.7	0.15	0.15
sample 2	0.7	0.15	0.15
sample 3	0.33	0.33	0.34

(A) 1.82220

(B) 2.1250

(C) -1.82220

[Answer](#)

Quiz 2: Quiz on Cross Entropy

Given two classifiers (classifier 1 and classifier 2), which one has better log loss? (*answer at foot of page.*)

Actual label: 1 mean true label			
ID	Apple	Pear	Orange
sample 1	1	0	0
sample 2	1	0	0
sample 3	1	0	0

Classifier 1 prediction			
ID	pApple	pPear	pOrange
sample 1	0.7	0.15	0.15
sample 2	0.7	0.15	0.15
sample 3	0.33	0.33	0.34

Classifier 2 prediction			
ID	pApple	pPear	pOrange
sample 1	0.5	0.25	0.25
sample 2	0.5	0.25	0.25
sample 3	0.5	0.25	0.25

- (A) Classifier 2 has better categorical cross entropy loss.
- (B) Classifier 1 has better categorical cross entropy loss.
- (C) Hard to tell. It depends on the threshold of classification.

[Answer](#)

Quiz 3: Quiz on Accuracy

Between Classifier1 and Classifier2, which one has higher accuracy?

- (A) Classifier 2 has better categorical cross entropy loss.
- (B) Classifier 1 has better categorical cross entropy loss.
- (C) Hard to tell. It depends on the threshold of classification.

[Answer](#)

Quiz 4: Quiz on accuracy

Given lower log loss value, the model may still perform worse (lower accuracy). Why don't we use accuracy as a loss function (in deep learning)?

- (A) It's expensive to compute accuracy across multiple mini-batch.
- (B) Accuracy isn't differentiable so we can't use back-propagation.

[Answer](#)

1. https://en.wikipedia.org/wiki/Additive_smoothing↵
2. <https://arxiv.org/abs/1301.3781>↵
3. <https://storage.googleapis.com/pub-tools-public-publication-data/pdf/6c8a86c981a62b0126a11896b7f6ae0dae4c3566.pdf>↵
4. <https://storage.googleapis.com/pub-tools-public-publication-data/pdf/6c8a86c981a62b0126a11896b7f6ae0dae4c3566.pdf>↵
5. <https://github.com/facebookresearch/dlrm>↵
6. <https://arxiv.org/abs/1505.07647>↵

7. Answer: A.
<https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.h>
8. <http://www.shivanirao.info/uploads/3/1/2/8/31287481/cikm-cameryready.v1.pdf>
9. <http://quinonero.net/Publications/predicting-clicks-facebook.pdf>
10. <https://arxiv.org/pdf/1708.02002v2.pdf>
11. <https://github.com/airbnb/aerosolve>
12. <https://arxiv.org/pdf/1706.02216.pdf>
13. <https://eng.uber.com/uber-eats-graph-learning/>
14. <https://arxiv.org/pdf/1301.3781.pdf>
15. <https://arxiv.org/pdf/1907.06558.pdf>
16. <https://www.coursera.org/learn/compstatsintro>
17. <https://eng.uber.com/pplm/>
18. <https://maxhalford.github.io/blog/weighted-sampling-without-replacement/>
19. <https://arxiv.org/pdf/1012.0256.pdf>
20. <http://proceedings.mlr.press/v97/byrd19a/byrd19a.pdf>
21. https://astrostatistics.psu.edu/su14/lectures/cisewski_is.pdf
22. <https://www.coursera.org/learn/compstatsintro>
23. <https://arxiv.org/abs/1606.07792>
24. <https://daiwk.github.io/assets/youtube-multitask.pdf>

25. <https://arxiv.org/pdf/2008.13535.pdf>
26. <https://instagram-engineering.com/lessons-learned-at-instagram-stories-and-feed-machine-learning-54f3aaa09e56>
27. <https://daiwk.github.io/assets/youtube-multitask.pdf>
28. <https://arxiv.org/abs/1906.00091>
29. <https://arxiv.org/abs/2012.08724>
30. <https://engineering.linkedin.com/blog/2021/budget-split-testing>
31. <https://arxiv.org/abs/1702.08734>
32. <https://arxiv.org/abs/1908.10396>
33. <https://blog.onebar.io/building-a-semantic-search-engine-using-open-source-components-e15af5ed7885>
34. <https://arxiv.org/abs/1803.11175>
35. <https://developers.google.com/machine-learning/guides/rules-of-ml>
36. Answer: C
37. Answer: C
38. Answer: A
39. Answer: B
40. Answer: B
41. Answer: B

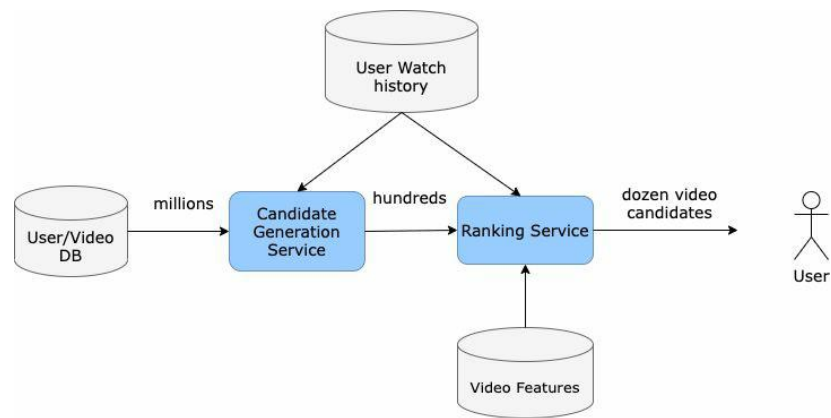
Common Recommendation System Components

In this chapter, we will cover common components of modern recommendation systems. Recommendation systems typically consist of the following components: candidate generation, ranking (scoring), and re-ranking.

- **Candidate generation:** we start from a huge list of candidates and generate a much smaller subset of candidates. For example, the candidate generator on YouTube reduces billions of videos down to hundreds or thousands.
- **Ranking (Scoring):** another ML model scores and ranks the candidates in order to select the set of items (on the order of 10) to display to the user. Given the candidate list is relatively small, we can use a more precise model.
- **Re-ranking:** we remove items that the user explicitly disliked or boost the score of fresher content or remove political content, etc. Re-ranking can also help ensure diversity, freshness, and fairness.

Candidate Generation

Candidate generation is an important step before the ranking step. There are two common approaches: content-based filtering and collaborative filtering.



mlengineer.io

Video Recommendation. Source: mlengineer.io

Content-Based Filtering

The main idea is to rely on what content (or videos) users like in the past and then recommend similar content to users. Let's take a look at the Google App Play recommendation.

User-App Table

	App	Education	Casual	Health	Entertainment	Science	Healthcare
App	1	1	0	0	0	0	0
App	2	1	0	1	0	1	0
App	3	0	0	0	0	1	0
App	4	1	1	0	0	0	0

- In this feature matrix, each row represents an app and each column represents a feature. To simplify, assume this feature matrix is binary: a non-zero value means the app has that feature.
- To recommend apps to users, we want to select the most similar apps that users already have.
- First, we pick a similarity metric (for example: dot product), then we score each candidate using a similarity metric (dot product). Candidates with higher dot product values will get higher ranking in recommendations.

Trade-Offs

- Pros
 - It's easier to scale since we don't use another user's features.
 - The model can recommend specific interests of a user and niche content to the other very few users who are interested.
- Cons
 - The model can only recommend content based on the user's existing interests and does not expand the user's interests.
 - The model doesn't leverage other user's features.

Collaborative Filtering

To address some of the limitations of content-based filtering, collaborative filtering simultaneously uses similarities between users and items to make recommendations. The basic idea is to recommend an item to user A based on the interests of a similar user B.

Let's use an example regarding movie recommendations.

User-Movies Table

User	Encanto	Frozen	Fast and Furious	Top Gun	Spiderman
An	1	1	0	0	0
Khang	1	0	1	1	0
Anh	0	0	0	1	1
Emma	1	1	0	0	0

- Each row is one user.
- Each column is one movie.
- For simplicity, each cell has values of 0 (user dislikes this movie) or 1 (user likes this movie).

Assume we care about two categories: children and action. And for each movie, we assume they have some specific score based on these categories.

Movies - Category Table		
Movie	Children	Action
Encanto	5	0
Frozen	5	0
Fast and Furious	1	5
Top Gun	0	5
Spiderman	4	1

- Each row is one movie.
- Each column is one category: children or action.
- For simplicity, each cell has values from 0-5; 5 means this movie highly belongs to this category.

We have the same table for user-category.

User - Category Table

User - Category Table		
User	Children	Action
An	1	0
Khang	0	1
Anh	1	1
Emma	1	0

The dot product of these two matrices becomes

User-Movies Feedback Table					
User	Encanto	Frozen	Fast and Furious	Top Gun	Spiderman
An	5	5	0	0	4
Khang	0	0	5	5	1
Anh	5	5	5	5	5
Emma	5	5	0	0	4

It's clear that An and Emma have very similar tastes in movies. Therefore, we can recommend Emma's favorite movies to An. But how do we define these categories? How do we "assign" values for each category for each user?

Collaborative Filtering uses Matrix Factorization technique to derive these matrices.

Given the feedback on matrix AA , how can we calculate the two matrices UU and VV so that: $UV^T \sim AU\{V\}^T \sim A$

- Matrix UU have dimension $m \times d$ where m is the number of users and d is the embedding dimension. Row i -th is the embedding of user i .
- Matrix VV have dimension $n \times d$ where n is the number of movies and d is the embedding dimension. Row i -th is the embedding of movie i .

There are two common algorithms: Stochastic gradient descent (SGD) and Weighted Alternating Least Squares (WALS). You can read more details [here](#)¹

Trade-Offs

- SGD
 - Very flexible, can use many different loss functions.
 - Can be parallelized.
 - Harder to handle the unobserved entries
- WALS
 - Converges faster than SGD.
 - Easier to handle unobserved entries.
 - Can be parallelized.

Finally you can read about more advanced techniques for example Softmax model².

How Pinterest Does Candidate Generation

Co-occurrences of Items to Generate Candidates

Take Pinterest Pins recommendation³ for example, users created boards and these boards usually contained similar items. If a pair of Pins appear together many times, we can argue they are semantically similar. So we use a traditional MapReduce job to count the occurrences of any given pair on Pins. This is similar to the content-based filtering example in the previous section.

Online Random Walk

The previous approach doesn't maximize the co-occurrence of related pins. Rare pins, pins occurring on only a few boards, did not have many candidates. With online random walk⁴, given a pins-board bipartite graph we can load all of these connections into memory and do random walk. Given an input pin, we can walk in the graph and increase visit counts for each visited pin. After a number of steps, we have aggregated counts of relevant pins. This is similar to Google PageRank approach.

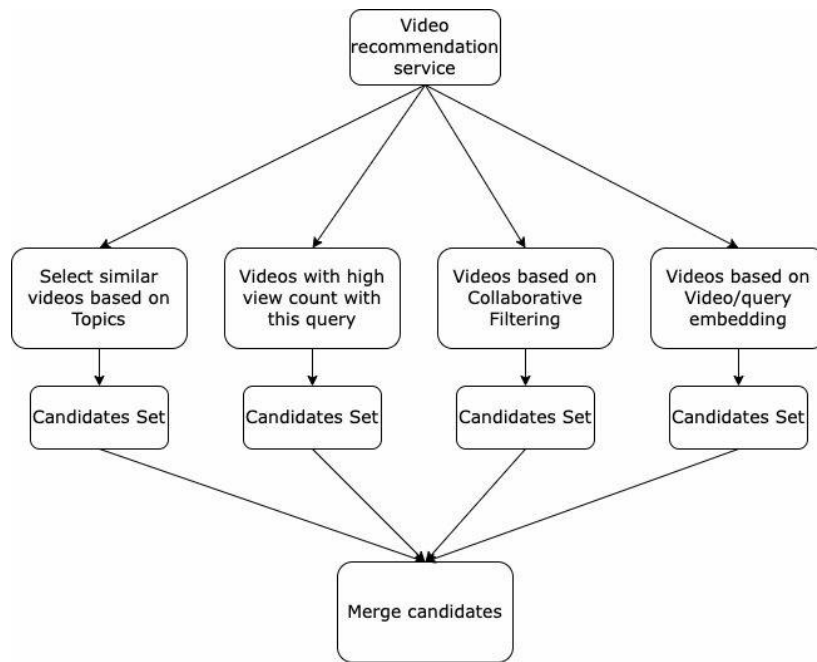
Session Co-occurrence

The previous approach considers Pins on the same board and semantically similar. In reality, a lot of long-lived boards are too broad and are usually a collection of relevant Pins. To address this problem, we can use Pins2Vec, similar to how we train embedding with word2vec style (read section Embedding [\[subsec-embedding\]](#)).

How YouTube Build Video Recommendation Retrieval Stack

YouTube video recommendation retrieval stack (candidate generation⁵) is quite comprehensive. It uses multiple candidate generation algorithms, each capturing one aspect of similarity between query video and candidate video.

- For example, one algorithm generates candidates by matching topics of query video.
- Another algorithm retrieves candidate videos based on how often the video has been watched together with the query video.
- They applied collaborative filtering and two-tower style embedding.



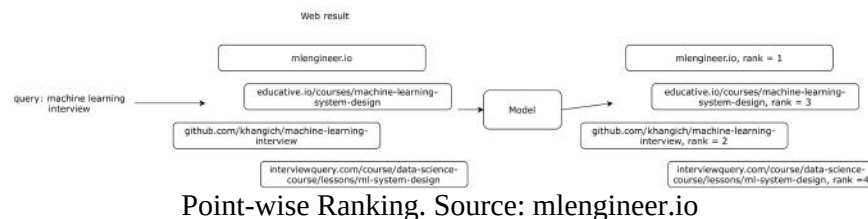
YouTube Candidate Generation. Source: mlengineer.io

Ranking

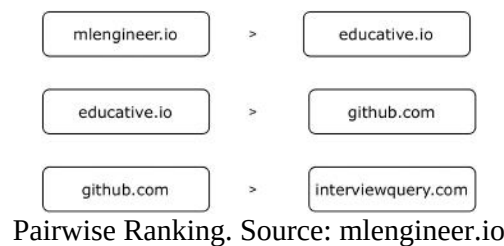
At the high level, a recommendation problem can be formulated as: point-wise ranking, pairwise ranking and list-wise ranking. While the point-wise ranking is the most popular formulation (read YouTube Recommendation WatchNext paper⁶). In practice, in some applications, pairwise formulation can offer a few interesting advantages. In this section, we will visit RankNet, a specific technique for recommendation systems.

How to Build a ML-Based Search Engine

Point-wise approach: given a search query, we train a Machine Learning model to predict their ranking. Given the list of results from a specific query, assume we have a Machine Learning model that predict the ranking for each result. Based on the prediction, we can then sort the results based on their rankings.



Pairwise approach: with this formulation, we don't really need to predict every result's ranking. All we really need to know is which result has higher rank than the other result.



In certain situations, point-wise ranking is very challenging. Not only do we need an abundance label (actual ranking of every possible result), we also need to build a model to do ordinal regression, which is unnecessarily hard. On the other hand, a naive pairwise approach would require a lot of computation. For example, if we have 100 web results to rank, we would need to do $100 \times 99 / 2 = 4950$ comparisons to produce final consistent ranking results. Can we do it better?

RankNet

At the core RankNet^Z proposes the following considerations

- The pairs of ranks do not need to be complete: they do NOT need to specify a complete ranking of the training data.
- The ranking results do NOT need to be consistent.

Assuming we have a machine learning model with function $f(x)$ such that given a query result x , function can map it to a real value such that $f(\text{'mlengineer.io'}) > f(\text{'educative.io'})$ if and only if the model thinks “mlengineer.io” ranks is higher than “educative.io”.

For simplicity, we use o_{ij} as $f(x_i) - f(x_j)$ and o_{ij} as $f(x_i) - f(x_j)$

- Probability $P(x_i > x_j) = P_{ij} = \frac{e^{o_{ij}}}{1 + e^{o_{ij}}}$
- Cost function: use cross entropy with $target_{ij}$ as desired probability for pair (i, j) $C_{ij} = -target_{ij} * \log(P_{ij}) - (1 - target_{ij}) * \log(1 - P_{ij})$

Model architecture: since the formulation and cost function is pretty generic, we can use any deep learning architecture for this problem. We can start with a fully connected neural network with two layers.

Example

Relevant score: query vs document

QueryId DocumentID:RelevantScore DocumentId:RelevantScore		
qid:1830	1:0.002736	1:0.002736
qid:1830	1:0.002736	1:0.002736

Assume we have this dataset containing query and matched result pages. This is a sample code in PyTorch.

```

class RankNet(torch.nn.Module):
    def __init__(self, inputs, hidden_size, outputs):
        super(RankNet, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(inputs, hidden_size),
            nn.ReLU(inplace=True),
            nn.Linear(hidden_size, outputs),
        )
        self.sigmoid = nn.Sigmoid()
    def forward(self, input_1, input_2):
        result_1 = self.model(input_1)
        result_2 = self.model(input_2)
        pred = self.sigmoid(result_1 - result_2)
    def predict(self, input):
        result = self.model(input)
        return result
    def train():
        inputs = 38
        hidden_size = 10
        outputs = 1
        learning_rate = 0.2
        num_epochs = 100
        batch_size = 100
        model = RankNet(inputs, hidden_size, outputs).to(
            device)
        #Loss function and optimizer
        criterion = nn.BCELoss()
        optimizer = optim.Adadelta(model.parameters(), lr
            = learning_rate)
        data_loader = get_loader(data_path, batch_size, ?
            False, 4)
        total_step = len(data_loader)
        data_shape = (label_size, 1)
        # The batch size method is used here, not every
        # time a pair of docs is passed in for forward
        # and backward propagation
        # (tips: There is also a way to input all docs
        # pairs under each query as batches into the
        # network for forward and backward for epoch in
        # range(num_epochs):
        for i, (data1, data2) in enumerate(data_loader):
            print('Epoch [{}/{}], Step [{}/{}]'.format(
                epoch, num_epochs, i, total_step))
            data1 = data1.to(device)
            data2 = data2.to(device)
            label_size = data1.size()[0]
            pred = model(data1, data2)
            loss = criterion(pred, torch.from_numpy(
                np.ones(shape=data_shape))
                .float().to(device))
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

```

Other notes: In practice, you can also use the NDCG metric (read section Evaluation [1.3](#)) to evaluate the model.

On the LinkedIn social network, we have rich data about members and their job titles. Although this information is highly valuable, they are neither consistent nor very clean. We want to be able to say that level 3 at company A is lower than level 2 at company B. Can you formulate this machine learning problem as a recommendation problem? Can you use RankNet? What metrics should you use?

Observations

In practice, pairwise learning scores relatively high in terms of AUC but does poorly on nDCG, even with calibration. Therefore, pairwise learning gives a good ranking of ads but fails at estimating the click probability. This is due to its objective of incurring less ranking loss by attempting to give a correct order of ads, however, without taking into account the accurate estimation of click probability.

Re-ranking

In the final stage of a recommendation system, the system can re-rank the candidates to consider additional criteria or constraints.

- One re-ranking approach is to use filters that remove some candidates.
- Another re-ranking approach is to manually transform the score returned by the ranker.
- Detect and remove click-bait content
- Re-rank content based on content age (promote freshness), content length, etc.

This section briefly discusses freshness, diversity, and fairness.

Freshness

Most recommendation systems aim to incorporate the latest usage information, such as current user history and the newest items. Keeping the model fresh helps the model make good recommendations.

- Retrain the model as often as possible to learn on the latest training data (read section Retraining [\[retraining\]](#)). For the deep learning model, you can use the warm-start training method. It helps reduce training time significantly.
- Create an “average” user to represent new users in matrix factorization models. You don’t need the same embedding for each user—you can create clusters of users based on user features.
- Use an advanced deep learning model, such as softmax model or two-tower model. Since the model takes feature vectors as input, it can be run on unseen sample data.
- Add document age as a feature. For example, YouTube can add a

video's age or the time of its last viewing as a feature.

Diversity

If the system always recommend items that are "closest" to the query embedding, the candidates tend to be very similar to each other. This lack of diversity can cause a bad or boring user experience. For example, if YouTube just recommends videos very similar to the video the user is currently watching, such as nothing but owl videos (as shown in the illustration), the user will likely lose interest quickly.

- Train many candidate generators using different sources.
- Train many rankers using different objective functions.
- Re-rank items based on genre or other metadata to ensure diversity.

Fairness

We want the model to treat all users fairly. Hence, we want to make sure the model isn't learning unconscious biases from the training data.

- Make different models for underserved groups.
- Increase training data: add more auxiliary data for underrepresented groups.
- Track online/offline metrics on each demographic to watch for biases.

Position Bias

In the ads recommendation, the position of the ads influences how likely users will click on it. In practice, many ads companies (Twitter, YouTube, etc.) observe a significant drop in engagement rate as a function of position. One of the reasons is that users tend to click on the result at the top, regardless of quality or relevance.

Why Would This be an Issue in Machine Learning Model Training?

We collect feedback (clicks) and train the model to optimize for relevancy and quality, but our training data has positive labels simply because the results show up on top.

Use Position as feature

A commonly used practice is to inject position as an input feature in model training and then remove the bias at serving. In probabilistic click models, the position is used to learn $P(\text{relevance}|\text{pos})$. Another method to remove position bias⁸, we can train a model using the position as an input feature and serve by setting position feature to 1 (or other fixed value, such as missing value).

Use Position as Feature

1. In a typical deep learning network, we can use position as an input feature to the network during training. At serving time, we set all position values as constant values.
2. In wide and deep (read Wide and Deep architecture [\[sec-wide-and-deep\]](#) section), we can use position as a feature into the shallow tower.

Instagram uses sparse position feature as input to the last

fully connected layer to handle position bias⁹.

Inverse Propensity Score

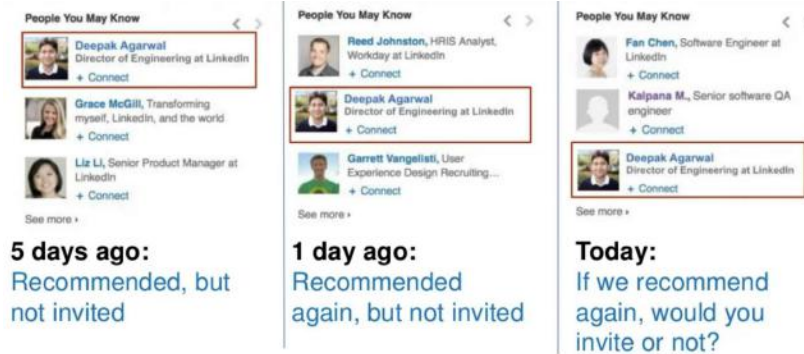
There are other approaches that learn a bias term from position like Inverse Propensity Score¹⁰. One way to estimate the Propensity Score is simply use a Random prediction

However, in real-world recommendation systems, especially social media platforms like Twitter and YouTube, user behaviors and item popularities can change significantly every day. Therefore, instead of IPS-based approaches, we need to have an efficient way.

1. Absolute positioning: $w = 1 + p$
2. Log: $w = \log(\beta + p)$
3. $w = 1 - \frac{1}{\alpha + p}$

Further reading Position Bias Estimation for Unbiased Learning to Rank in Personal Search¹¹.

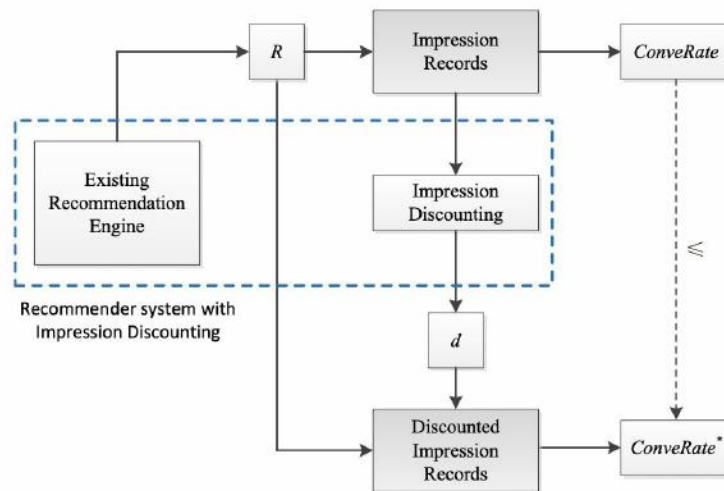
How LinkedIn Uses Impression Discount in People You May Know (PYMK) Features



How LinkedIn Handles Impression Discount. Source: LinkedIn

LinkedIn People You May Know (PYMK) is a feature that recommends relevant connections that users may know. The goal is to help users enrich their professional network and increase users' engagements.

- We showed users the top 3-5 connections they might know. What if users didn't follow any of these recommendations?
- Even with model re-scoring, we might keep recommending the same connections to users because their relevant score might still be on top of the connections list.
- It creates a negative user experience and ultimately diminishes the purpose of PYMK.



Impression Discount: high level workflow. Source: LinkedIn

- We keep the existing Recommendation model and use impression discount as a plugin.
- We only need to use historical impressions data to determine a discounting factor d to maximize Conversion rate improvement.
- Maximizes

$$\text{Conversion rate improvement} = \frac{\text{new_conversion_rate} - \text{conversion_rate}}{\text{conversion_rate}}$$

Further reading: Modeling Impression Discounting in Large-scale Recommender Systems¹²

Calibration

Definition

As we will study the Predict Ad Click problem in [\[deployment\]](#) section and chapter [\[ad-click-prediction\]](#), calibration is a very important concept in evaluating model performance.

Follow the definition in Predicting Clicks on Ads¹³ “Calibration is the ratio of the average estimated CTR and empirical CTR. In other words, it is the ratio of the number of expected clicks to the number of actually observed clicks.”

We define calibration as:

$$\text{Calibration} = \frac{\sum(\text{predictions})}{\sum(\text{labels})} = \frac{\sum(\text{predictions})}{\sum(\text{labels})}$$

When calibration is important?

- If we’re building a critical system where the actual prediction value is important, for example: the probability of patient being sick.
- We want to debug if model is over-confident with wrong prediction or under-confident with correct prediction.

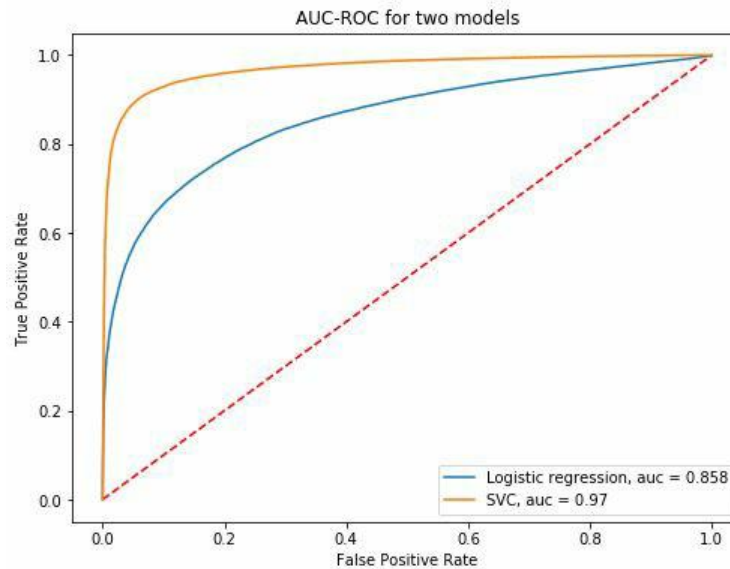
Causes of miscalibration

- Model generalization: model makes poor predictions on new data.
- Nonstationarity: changes in user behavior or marketplace that are not modeled.
- Selection bias: during generating training data, we resample data (up-sampling minority samples and/or downsampling majority samples).
- Training/serving misalignment: model was trained to predict a different

label than the calibration label.

Example and solution

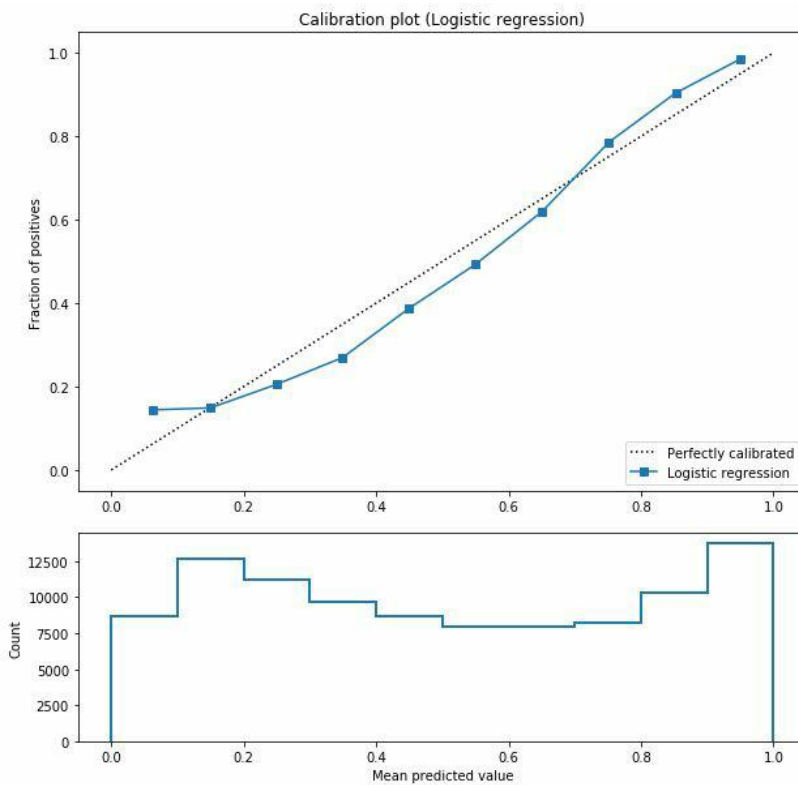
Let's take a look at simple binary classification problem. We have two models logistic regression and Support Vector Machines (SVM).



Calibration: logistic regression vs. SVM. Source: dimpo.me

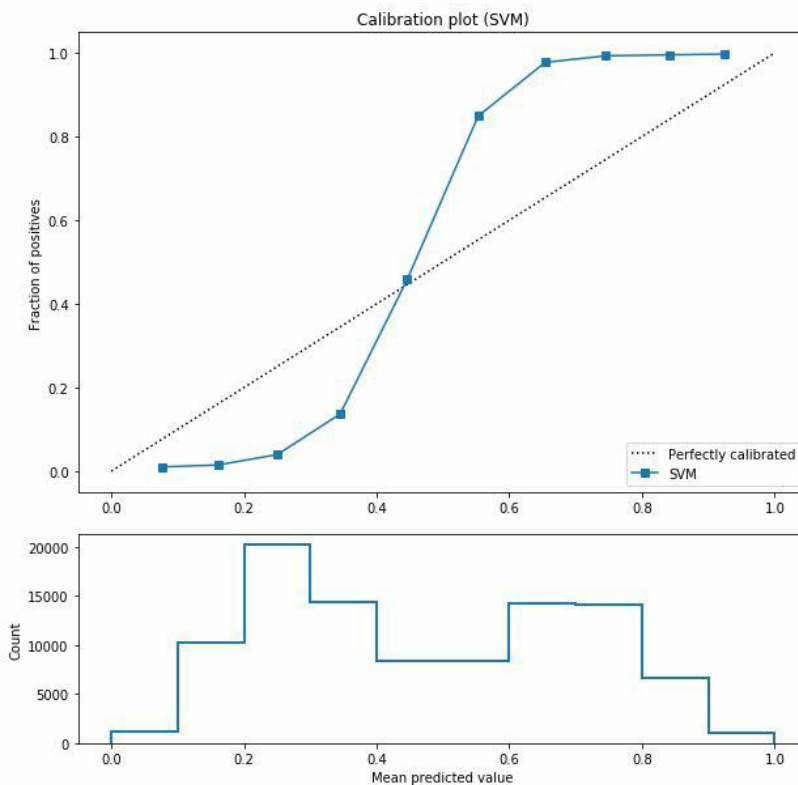
SVM model has better AUC (0.97) vs. logistic regression AUC (0.858). If we don't apply calibration, we might conclude SVM is a better model. In practice, it's always useful to look at the prediction distribution. One way to do that is to use calibration plot.

Calibration plot



Calibration plot: logistic regression model. Source: dimpo.me

- In the top chart, we plot the mean prediction values vs. the fraction of positive labels.
- The x-axis represents the mean prediction values within each bin, y-axis represents the true ratio of positive data over the whole population.
- Perfectly calibrated means: if the prediction is 0.2 then the likelihood of the prediction label being positive label is 20%.
- The bottom chart shows the histogram of prediction values (bins = 10). For example, we have 12500 predictions have probability within range [0.9, 1.0].



Calibration plot: SVM model. Source: dimpo.me

- In the top chart, the blue line is quite faraway from perfectly calibrated line especially on both the low prediction values (0.2) and high prediction values (0.8). We can interpret it as the model is under-confident on the low-end and over-confident on the high end.
- The x-axis represents the mean prediction values within each bin, y-axis represents the true ratio of positive data over the whole population.
- Perfectly calibrated means: if the prediction is 0.2 then the likelihood of the prediction label being positive label is 20%.
- The bottom chart shows the histogram of prediction values (bins = 10). For example, we have 12500 predictions have probability within range [0.9, 1.0].

In conclusion, using AUC metric alone in this case is not sufficient. There are many ways to calibrate model: apply *isotonic*¹⁴ method or *Platt's*¹⁵ method.

Isotonic method is a non-parametric method which minimizes $\sum_{i=1}^N (\text{actual}_i - \text{prediction}_i)^2$

Platt's method is based on Platt's logistic model
$$p(\text{actual}_i=1|\text{predict}_i) = \frac{1}{1 + \exp(A * \text{predict}_i + B)}$$

where A, B are real numbers to be determined when fitting the regressor via maximum likelihood.

Facebook uses this formula $q = \frac{p}{p + \frac{1-p}{w}}$ during post-processing to calibrate prediction where p is prediction value and w is negative downsampling rate.

Further reading: temperature scaling^{[16](#)}, calibrate decision tree^{[17](#)} calibration curve^{[18](#)} and Predicting Good Probabilities With Supervised Learning^{[19](#)}.

Nonstationary Problem

- In an online setting, data keeps changing. Hence, data distribution shift (joint distribution of inputs and outputs differs between the training and test stages) is very common. So, keeping the models fresh is critical to achieving sustained performance. Based on how frequently the model performance degrades, we can then decide how often models need to be updated/retrained. One common algorithm that can be used is Bayesian Logistic Regression. In practice, it's very cumbersome to maintain and update, so big tech companies rely on other solutions to handle this issue.
- LinkedIn applied Bayesian Logistic Regression²⁰ to combine training historical data with near-time data (Lambda Learner).

Exploration vs. Exploitation

In Ad Click prediction use cases, it's beneficial to allow some exploration in recommending new ads. However, if there has been too little conversion of advertising, it can reduce the company's revenue. This is a well-known exploration-exploitation trade-off. One common technique is Thompson Sampling²¹, where at time t , we need to decide which action to take at time t based on the reward.

Airbnb: Deep Learning is NOT a drop-in replacement

In this section, we review few learned lessons from Airbnb when they first [22](#) around 2018.

1. Deep Learning is NOT a drop-in replacement for Gradient Boosted Decision Tree (GBDT) models. At the beginning, Airbnb switched from GBDT to Deep Learning, but they couldn't see any offline performance gain.
2. Airbnb didn't find any success in using Dropout in training Deep Learning models. For many reasons, they actually found 1% improvement in the offline NDCG metric. However, it didn't result in any improvement in online metrics.
3. Don't initialize all weights and embedding to zeros: Airbnb later used Xavier initialization [23](#) for the network weights and random uniform in the -1, 1 range for embeddings.
4. Learning rate: Airbnb found that the variant LazyAdamOptimizer [24](#) helps training faster with large embeddings.

Interview Exercises

1. Recommend interesting places near me on Facebook. At the beginning, how do we sample positive and negative labels?
2. Design Machine Learning system to predict the number of people who will attend a Facebook event.
3. Design Machine Learning model to detect whether a human object detection system was actually detecting real life humans or humans on a tv/poster. Hint: leverage depth information.
4. Design feed ranking for Facebook.
5. Design Machine Learning model to detect illegal (or contraband) goods on Facebook Marketplace.
6. Design Job Suggestions Machine Learning model for LinkedIn members.
7. Design feed recommendation from non-friends on Facebook.
8. Design ML system to classify LinkedIn job seniority for all jobs in all countries and companies.
9. Design Smart Notification for Twitter users.
10. Design a ML system to filter out videos with irrelevant hashtags on TikTok.
11. Predict food category given a phrase (not user specific) in Instacart app.
12. Design a ML system for DoorDash users' demand prediction for one week in advance.

2. <https://research.google.com/pubs/pub45530.html>
3. <https://arxiv.org/pdf/1702.04680.pdf>
4. <https://arxiv.org/pdf/1702.07969.pdf>
5. <https://static.googleusercontent.com/media/research.google.com/en//pubs>
6. <https://daiwk.github.io/assets/youtube-multitask.pdf>
7. https://icml.cc/Conferences/2015/wp-content/uploads/2015/06/icml_ranking.pdf
8. <https://daiwk.github.io/assets/youtube-multitask.pdf>
9. <https://instagram-engineering.com/lessons-learned-at-instagram-stories-and-feed-machine-learning-54f3aaa09e56>
10. <https://static.googleusercontent.com/media/research.google.com/en//pubs>
11. <https://dl.acm.org/doi/pdf/10.1145/3159652.3159732>
12. <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.712.4260&rep=rep1&type=pdf>
13. <http://quinonero.net/Publications/predicting-clicks-facebook.pdf>
14. <https://scikit-learn.org/stable/modules/calibration.html#isotonic>
15. <https://scikit-learn.org/stable/modules/calibration.html#sigmoid>
16. <https://arxiv.org/pdf/1706.04599.pdf>
17. <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.29.3039&rep=rep1&type=pdf>
18. https://scikit-learn.org/stable/modules/generated/sklearn.calibration.calibration_curve

19. <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.60.7135&rep=rep1&type=pdf>[↵]
20. <https://github.com/linkedin/lambda-learner/>[↵]
21. <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/thompson.pdf>[↵]
22. Applying Deep Learning To Airbnb Search <https://arxiv.org/pdf/1810.09591.pdf>[↵]
23. <http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>[↵]
24. https://www.tensorflow.org/addons/tutorials/optimizers_lazyadam[↵]

Machine Learning Usecases from Top Companies

Airbnb - Room Classification

Requirement and Data

Design ML model to classify room data (image, text) is Bedroom/LivingRoom/Gym etc.

Challenges

- How do we leverage rich input features: room photos, room captions, etc.?
- How do we collect training labels?
- How do we leverage trained CV models like ResNet50?

Metrics

We formulate this problem as supervised learning, specifically Classification problem.

- Use common metrics in classification like *precision*, *recall*, *accuracy* and *f1score*.
- We assume the room type is not too imbalanced. We aim to have high *precision* ($\geq 95\%$).

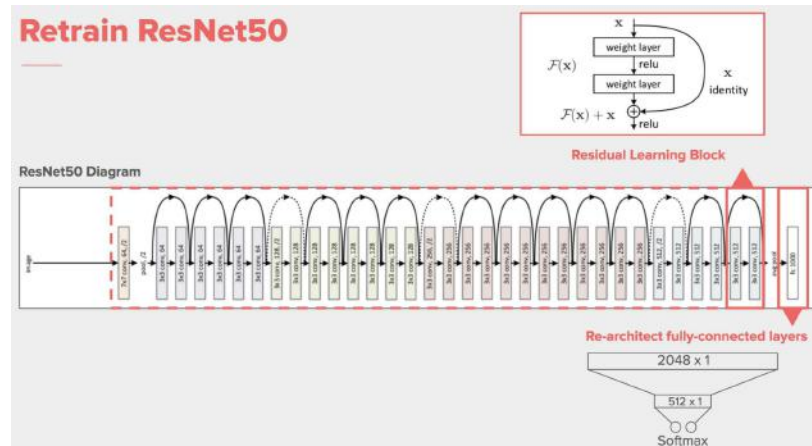
TensorFlow expects labels to be provided in a one_hot representation. If you want to provide labels as integers, please use SparseCategoricalCrossentropy loss.

Features

Image: Listing images, amenities images, image description, amenities and host description.

Model

Model Architecture



Room Classification Model Architecture. Source: AirBnB

Model Training and Serving

- Architecture: use ResNet50, add two more fully connected layers and a softmax activation in the end.
- Data: combination of 1) third-party vendors to obtain high-quality label data 2) images captured by hosts.
- Training method 1: Fine-tuning: Keep the base ResNet50 model fixed and only retrain the added two layers using minimal data. It's easy and leads to decent results.
- Training method 2: Retrain the whole modified ResNet50 from scratch. It's more difficult, but we can get better performance from the model.
- Serving: We perform offline batch inference since it's sufficient for this usecase.

Improvements

- Leverage object detection model: use pretrained Faster R-CNN model to classify amenities within each room image.
- We can then use these insights to improve label quality.

You can read more detail about Airbnb Room Classification^{[1](#)} and Categorizing listing photos^{[2](#)}

Instagram: Feed Recommendation from Non-friends

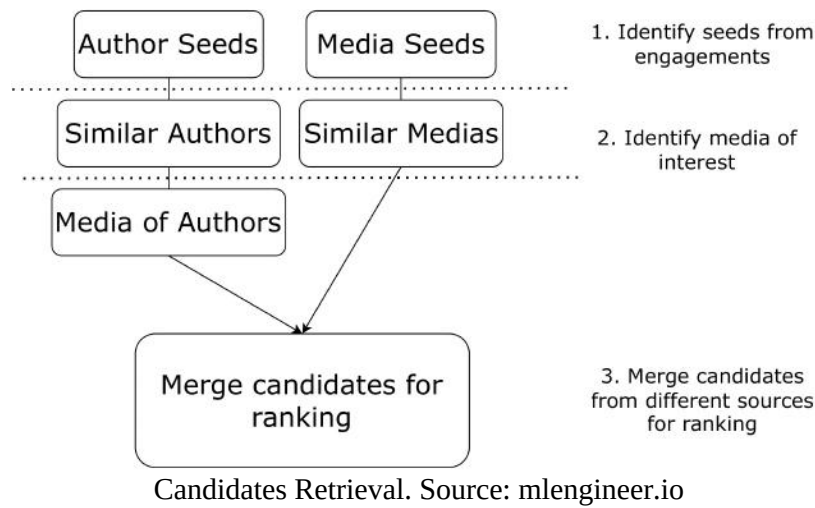
Scope/Requirements

- We want to design a recommendation for Instagram and accounts that are NOT in the user's network.
- We want to optimize for the relevant interest of users.
- Take one example, we have Instagram user Sheryl who follows a magazine about van life. She regularly likes their content and comments on it. This gives us an implicit signal that she might be interested in a similar genre of adventure and lifestyle magazines.
- Can we recommend new content that this person might like? Do you see the difference with the cold start problem?

Metrics

- High Recall for candidate generation.
- Use CTR metrics similar in ranking.
- Other metrics included rates of users following new users.

The main idea is to identify some seeds and expand them to further explore more content to show to users. A seed is an author or media in which one has shown a clear interest. Each seed can now be used as an input in our K-nearest neighbor pipelines that produce similar media or similar authors.



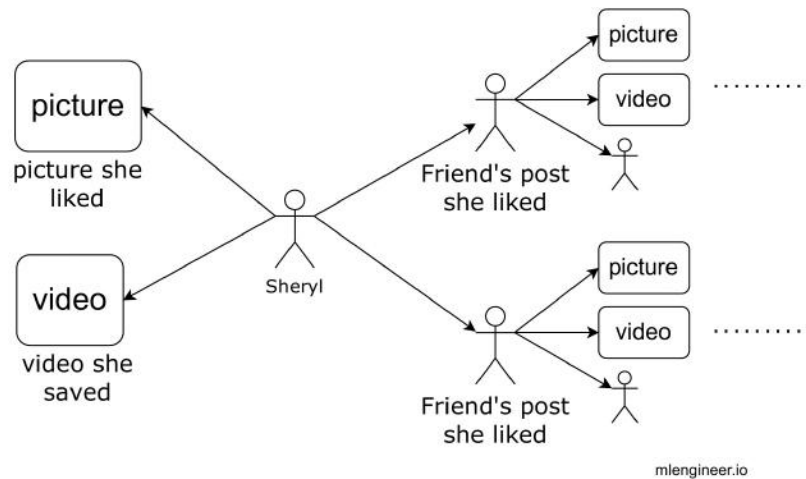
Data

We use all user historical sessions, in terms of what they like and who they interact with. These include positive engagement factors, such as likes, comments, and saves; and negative factors, such as not being interested and seeing fewer posts like these.

- Engagement features
- Author-Viewer interaction-based features
- Counters or trend-based features for author and media
- Content quality-based features
- Image or video understanding-based features
- Knowledge-based features
- Derived functional features
- Content understanding-based features
- User embeddings

- Content aggregation embeddings
- Content taxonomy-based features

Model Training and Serving



Instagram Explore Recommendation. Source: mlengineer.io

- Similarity method: apply both 1/ embedding similarity and 2/ co-occurrence similarity
- Train user embedding: engagement data with word2vec style architecture (read section Embedding [\[subsec-embedding\]](#)).

Co-occurrence Based Similarity

- Firstly, we generate co-occurred media lists by using user-media interaction data (e.g., our touring alien likes posts about van life and adventure lifestyle).
- Calculate co-occurrence frequencies of media pairs (e.g., van life posts and adventure lifestyle posts co-occur).
- Aggregate, sort, and get the top N of our co-occurred media as our recommendations for a given seed.

We use the Two tower architecture model and Gradient Boosted Decision Tree.

Cold Start Problem

Many new (and some seasoned) users may not have enough recent engagement on Instagram to generate a large enough inventory of candidates for them. This brings us to the familiar situation of dealing with a cold start problem in recommendation systems.

- Fallback graph exploration: For users whose immediate engagement graph is relatively sparse, we generate candidates for them by evaluating their one-hop and two-hop connections. For example: If a user A hasn't liked a lot of other accounts, we can probably evaluate the accounts followed by the accounts A has liked and consider using them as seeds. $A \rightarrow \text{Account liked by A} \rightarrow \text{Accounts followed by the accounts A likes (Seed Accounts)}$. The diagram below visualizes this line of thinking.
- Popular media: For extremely new users, we typically get them started with popular media items and then adapt our parameters based on their initial response.

During ranking, we can define value of a post (feed) as the following

$$\begin{aligned} \text{Value(Post)} &= (\text{prob_like})^{w_like} * \\ & (1 - \text{prob_not_interested})^{w_see_less} \text{Value(Post)} \\ & (\text{prob_like})^{w_like} * (1 - \\ & \text{prob_not_interested})^{w_see_less} \end{aligned}$$

Further reading, [Designing a Constrained Exploration System](#)³

LinkedIn: Talent Search and Recommendation

Scope/Requirements

Support Recruiter to fill in based on specific criteria. - We showed a list of matched candidates to recruiters. Recruiters then send Inmail requests and the candidate will hopefully accept this request. - We want to optimize the number of accepted Inmail requests.

Metrics

- Offline metrics: Precision At 5, Precision at 25 and NDCG (Normalized Discounted Cumulative Gain).
- Online metrics: overall Inmail acceptance rate.

Data and Features

There are three main feature sets. Recruiter context, query context and candidate profile (we focus more on job-seeking candidates). Since our query is text-based and candidate profile is also text-based we can use text-embedding. Before learning the text embedding we need to generate tri-gram like “machine learning engineer”, “deep learning” etc. to have better representations.

Other Considerations

- Context-aware ranking: one possible issue that can arise is when recruiters search for dentists, but we gave them software engineer candidates. One solution is to capture context as features for ML models. We can also use pairwise ranking instead of pointwise ranking since it worked better in practice. That means we look at pairs of candidates at a time, come up with the optimal ordering for that pair of candidates, and then use it to come up with the final ranking for all the results.

- Fuzzy matching: for a query like "Machine Learning Engineer," it's desirable to include some candidates with "Data Scientist" as the title. There are two solutions
 - Fuzzy semantic match on title id
 - Co-occurrence graph embedding based on profile data.
- Approach 2 scales better with our data size and less maintenance. The embedding can learn from the data. Additionally, we can experiment with pairwise two-tower embedding. One part of the tower is candidate embedding and the other part is the recruiter query.

Model

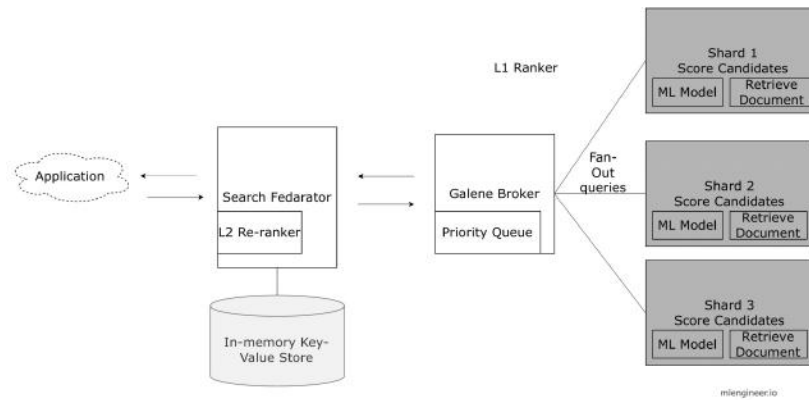
- XGBoost can be a good model, (i.e., ability to capture feature interactions and has rich model complexity).
- Alternative solution: two-tower network architecture with pairwise ranking.

Overall System

Like other search ranking systems, we need a retrieval stack and ranking stack.

Retrieval Stack

- Lucene-based search system is well suited. There are three main components: Search index of searcher;
- Broker: fan-out queries similar to aggregator service and live-updater which keep updates of the index near real-time.



Talent Search Layered. Source: mlengineer.io

Further reading LinkedIn talent search⁴

LinkedIn - People You May Know

Design Connection Recommendations: recommend users new connections that they may know.

Scope/Requirements

Metrics

For a member u , a connect member v are ranked by
$$\text{score}(u,v) = P(u,v) * \sum_{o=1}^O w(u,o) * V(u,v,o)$$

- $P(u,v) = P(u, v)$ = probability of member u invites v and v accepts
- where O is set of objects.
- $V(u,v,o) = V(u, v, o)$ = value for member u on objective o given a connection between u and v
- $w(u,o) = w(u, o)$ = weight objective o for member u

Link Prediction Features and Data

- Features
 - Number of common friends
 - Number of common close friends
 - Same school and major
 - Similar skillsets

Link Prediction Model

- Regression model: $P(u,v) = \lambda(\alpha Tfu + \beta Tfv + fuTAfv + \gamma T * fuv)P(u, v) = \lambda(\alpha Tfu + \beta Tfv + fuTAfv + \gamma T * fuv)$

where,

- f_u, f_v are member features
- $f_u T A f_v$ models interaction between features
- f_{uv} are edge features
- Number of common friends
- Number of common close friends
- Same school and major
- Similar skillsets

Value from Connection

$V(u,v,o)$ $V(u, v, o)$ = Expected marginal number of interactions per week if a connection between u and v is formed.

Members have many activities on LinkedIn, (e.g., post an article, connect to another member, join a professional group, etc.) These actions lead to interactions with member's connections.

- Use GLMs: $V(u,v,o) = F(f_u, f_v, f_{uv}, Net_u, Act_v)$ $V(u, v, o) = F(f_u, f_v, f_{uv}, Net_u, Act_v)$

where,

- f_u, f_v are member features
- f_{uv} are edge features

Linkedin - Learning Course Recommendation

Scope/Requirements

Design a course recommendation for LinkedIn course. We assumed we already have engagement data: which courses users engage with, which courses users saved etc.

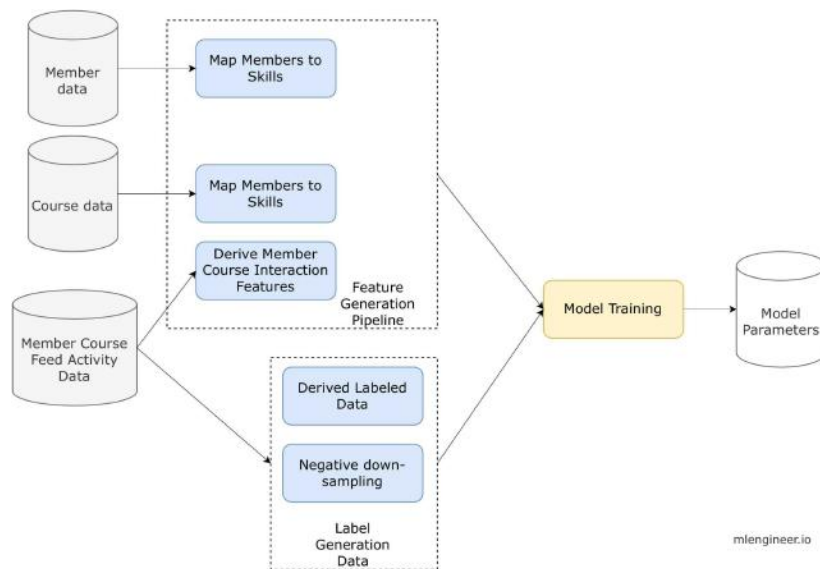
Metrics

- Area Under the Curve: AUC (read section Evaluation [1.3](#))
- Click-through Rate (CTR read section Evaluation [1.3](#))

Data

- User data: user skills set, user self-tagged skills etc.
- Course data: course description, tagged skills in courses, course video transcript, etc.
- User-course engagement data: which course users engage with, which courses users save, etc.

Model Training

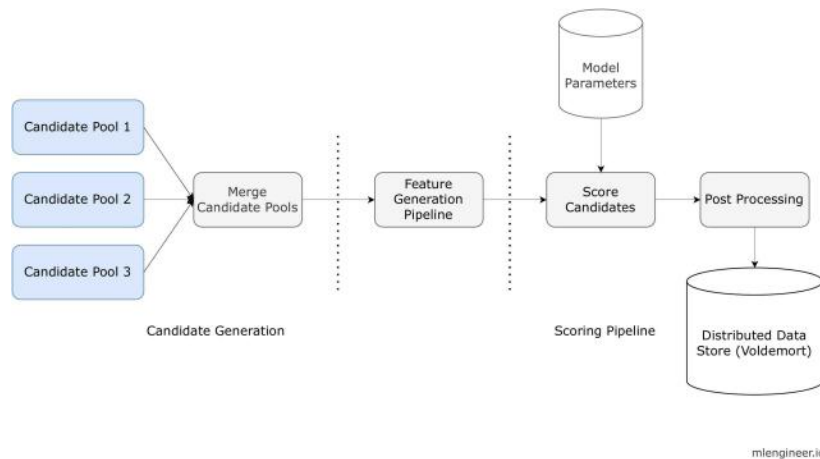


Supervised Learning System - High Level. Source: mlengineer.io

- Label generation
 - Training starts with label collection: clicks, impressions, and watched courses are extracted from members' activity logs for different channels and platforms (mobile vs. desktop) for a selected range of dates. The training set choice has a very strong impact on the performance of the model, so this step is very flexible.
 - After label collection, we preprocess the data: this includes splitting into train/test/validation sets, optional downsampling of negative labels, and injection of random negative examples and featurization.
- Model Training: support distributed model training. One option is to build on top of Apache Spark.
- Feature engineering
 - Extract data from activity logs, database snapshot. We can run the data pipeline daily.
 - Course transcript: extract LinkedIn skills and apply embedding.

- All features are stored and versioned. In the feature pipeline, we use the date of data point (user click) to select the right date-version of features.

Scoring



Scoring High-level System. Source: mlengineer.io

There are three major components:

- Offline scoring to reduce latency and reliability. It works for this use case since the user's course preference doesn't change very often.

Candidate Generation

- Select a reasonable number of recommendations (500-1,000).
- We can merged results from multiple algorithms (reference YouTube candidate generation)
 - high-precision skill affinity model based on taxonomy course skills
 - downsampled high recall model based on skills learned by a supervised model
 - courses similar to the courses that the member previously engaged

with

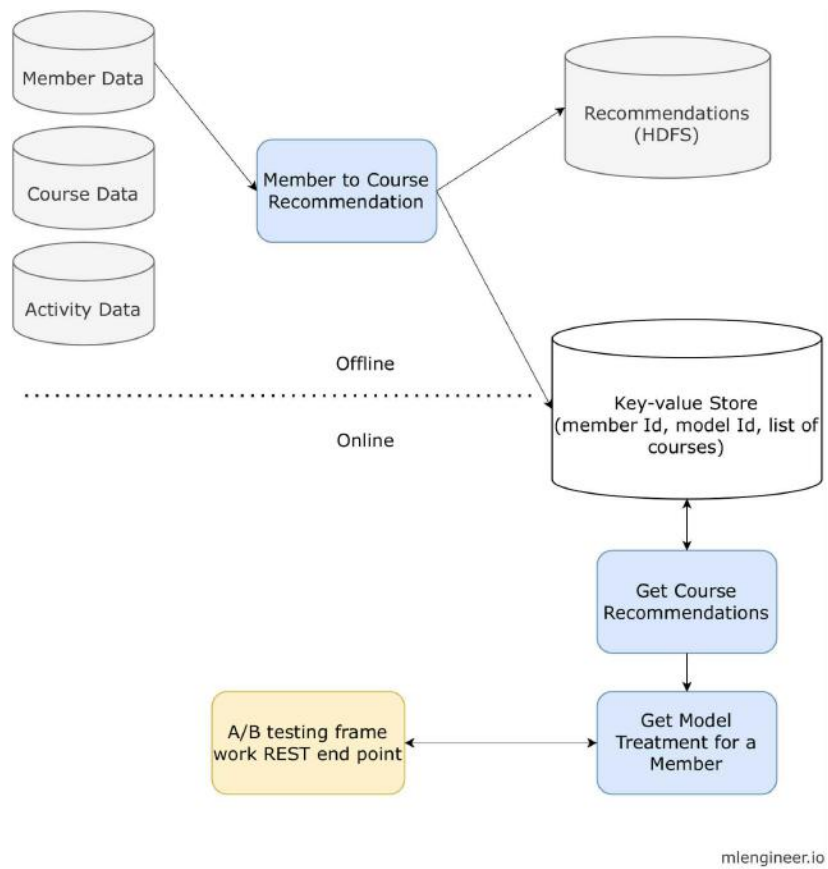
Data Streaming pipeline

- Map/Reduce flow that merges scoring candidates, member data and course data, and executes scoring function on it
- Optionally, we shard the course data to fit it in memory; currently, a single shard is sufficient.
- We replicate shards across multiple reducers, so that each reducer keeps just one shard, but each shard is replicated across many reducers.
- Finally, we group all the data that is keyed by members: features and list of candidates to score and send it to a random reducer that holds the course data shard.

To further improve the scoring pipeline, we can implement by storing course data in memory and do stream scoring.

Overall System

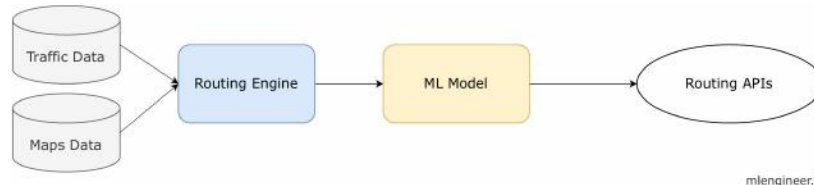
High-level Architecture



High-level Architecture. Source: mlengineer.io

Read more about course recommendation⁵

Uber - Estimate Time Arrival



Uber hybrid approach of ETA post-processing using ML models. Source: Uber Engineering Blog

Scope/Requirements

To keep it simple and flexible, we use the ML model as a post processing step from the routing ETA algorithm.

- **Latency:** The model must return an ETA within a few milliseconds at most.
- **Accuracy:** The mean absolute error (MAE) must improve significantly over the incumbent XGBoost model.
- **Generality:** The model must provide ETA predictions globally across all of Uber's lines of business such as mobility and delivery.

Metrics

- **Use loss function:** Asymmetric Huber Loss (read section Loss Functions [1.3](#)). This metric provides a way to tune
 - the level of robustness to outlier
 - tradeoff between underprediction vs. overprediction.

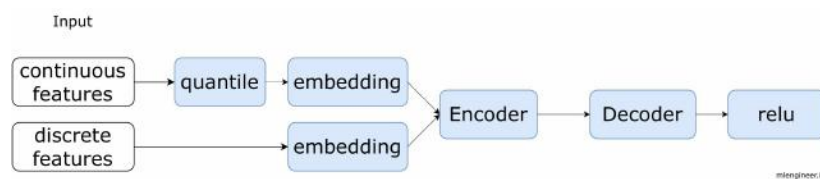
Data

Features

- For continuous feature, we apply quantile buckets: Gradient boosted decision tree neural network⁶ then apply embedding.
- For categorical features, we apply embedding.
- Location data: we use one of the geospatial embeddings techniques 1) Exact indexing 2) Feature hashing (read section Feature Hashing[\[feature-hashing\]](#)) or 3) Multiple feature hashing.

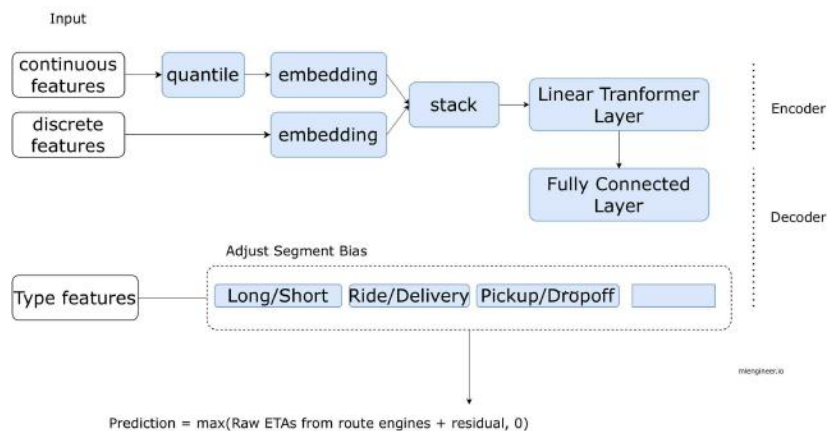
Model

At high level, we apply encoder-decoder architecture.



Uber DeepETA. Source: Uber Engineering Blog

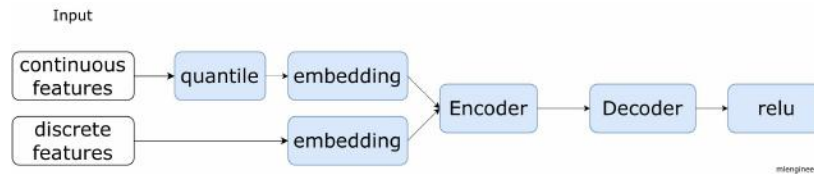
Details architecture



Prediction = max(Raw ETAs from route engines + residual, 0)
Uber DeepETA - source Uber engineering blog

Overall System

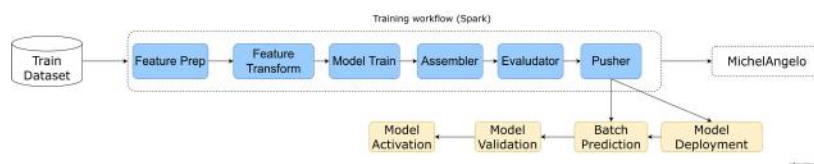
Real-time Serving High Level



Uber Real-time Serving - High Level. Source: Uber

- Uber consumers' requests are routed through various services to the uRoute service.
- The uRoute service serves as a front-end for all routing lookups.
- It makes requests to the routing engine to produce route-lines and ETAs. It uses this ETA and other model features to make requests to the Michelangelo online prediction service to get predictions from the DeepETA model.

Training Pipeline



Uber Training Pipeline - High Level. Source: Uber

- We periodically retrain and validate the model (read section Retraining [\[retraining\]](#))
- You can read more details about Uber DeepETA⁷

1. <https://medium.com/airbnb-engineering/widetext-a-multimodal-deep->

[learning-framework-31ce2565880c↵](#)

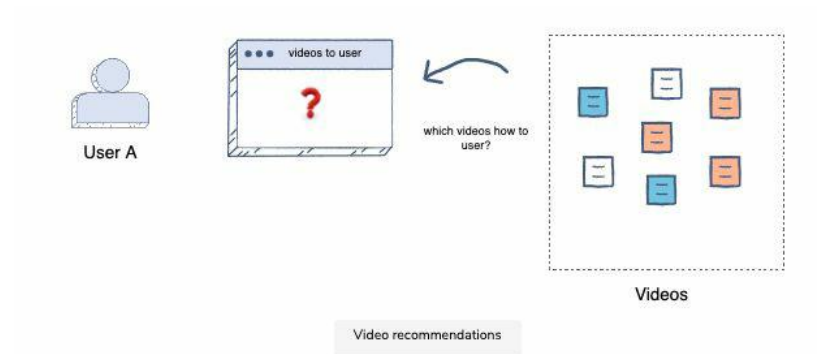
2. [https://medium.com/airbnb-engineering/categorizing-listing-photos-at-airbnb-f9483f3ab7e3↵](#)
3. [https://about.instagram.com/blog/engineering/designing-a-constrained-exploration-system?ref=shareable↵](#)
4. [https://arxiv.org/pdf/1809.06481.pdf↵](#)
5. [http://www.shivanirao.info/uploads/3/1/2/8/31287481/cikm-cameryready.v1.pdf↵](#)
6. [https://arxiv.org/pdf/1910.09340.pdf↵](#)
7. [https://eng.uber.com/deepeta-how-uber-predicts-arrival-times/↵](#)

YouTube Video Recommendations

In this chapter and the next chapters, we will put all concepts together and design machine learning use cases end to end.

Problem Statement

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



Video recommendation. Source: mlengineer.io

Metrics Design and Requirements

Metrics

- For offline metrics: use precision (the fraction of relevant instances among the retrieved instances), recall (the fraction of the total amount of relevant instances that were actually retrieved), ranking loss and logloss.
- For online metrics: use A/B testing and compare CTR, watch time, and conversion rate.

Requirements

Training

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

Inference

For each user to visit the homepage, we will have to recommend 100 videos for them. The latency must be under 200ms or ideally sub 100ms.

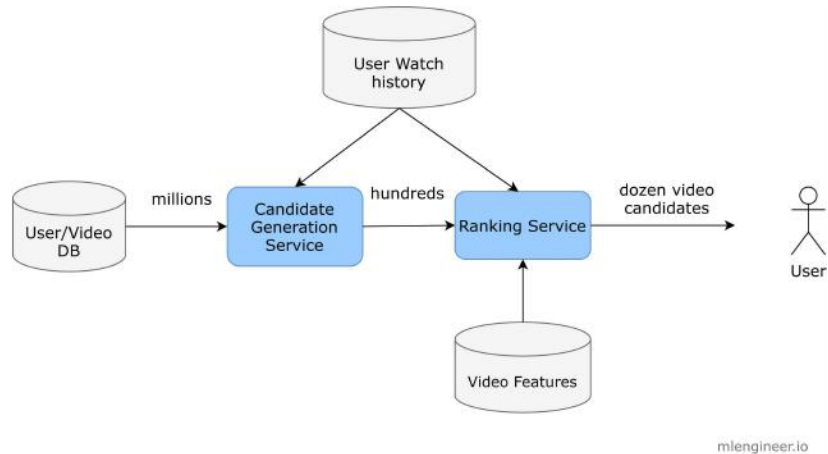
For online recommendations, it's important to find the balance between exploration vs. exploitation. If models overexploited historical data, new videos might not get exposed to users. We want to balance relevance and new content.

Summary

- Metrics: Reasonable precision, high recall
- Training with high throughput with the ability to retrain several times per day

- Inference need low latency from 100ms to 200ms
- We need away to be able to tune control exploration vs. exploitation during inference

Multistage Models



Video Recommendation Example. Source: mlengineer.io

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

- The candidate model would find 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., for videos having a high possibility of watching should be ranked high). It's a natural fit for the logistic regression algorithm.

Model Training

Candidate Generation Model

Training Data

For generating training data, we can make a user-video watch space.

- We can start by selecting a period of data: last month, last six months, etc. This would find a balance between training time and model accuracy.

Feature Engineering

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

Model

- The candidate generation can be done by matrix factorization. The purpose of candidate generation is to generate a ‘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.
- Read other solutions in section Candidate Generation [\[candidate-generation\]](#).

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.

Training Data

- We can use User Watched History data. Normally, the ratio between

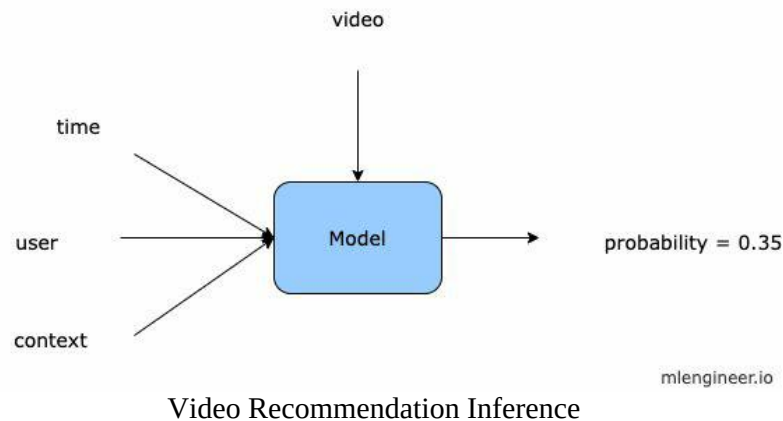
watch vs. not-watched is 2/98. This means that the vast majority of the time, the user does not watch a video.

Features Engineering

- Watched video Ids: apply video embedding
- Historical search query: apply text embedding. For example, word2vec or use pretrained word embedding, such as Global Vectors for Word Representation (GloVe) or Bidirectional Encoder Representations from Transformer (BERT)
- Training: optimize for high throughput with the ability to retrain many times per day
- Location: use geolocation embedding
- User associated features: age, gender with & Normalization or Standardization
- Aggregated impression with Normalization or Standardization
- Time-related features: Month, week_of_year, holiday, day_of_week, hour_of_day

Model

We can use a DCNv2 with a relurelu activation function at the hidden layer and sigmoid activation at the last layer. The loss function can be cross-entropy loss.



Quiz on Ranking Model

Question 1

Why do we use sigmoid activation at the last layer of the ranking model instead of relu?

- (A) Because it's faster to compute the sigmoid function.*
- (B) Because sigmoid function output probability from $[0, 1]$ presents the watch probability for recommended videos.*

[Answer](#)

Question 2

Given the imbalance distribution between watch vs. not-watch, we decide to perform negative downsampling before training the model. For example, the ratio between watch vs non-watch is 1:1 after downsampling. During online evaluation, the model has to predict non-sampling data. If we do not do anything, what would happen to the precision

metric (how many recommended videos are relevant)?

(A) online precision metric will be similar with offline precision metric

(B) online precision metric is lower than offline precision metric

(C) online precision metric is higher than offline precision metric

[Answer](#)

Calculation and Estimation

Assumptions

For simplicity, we can make these assumptions

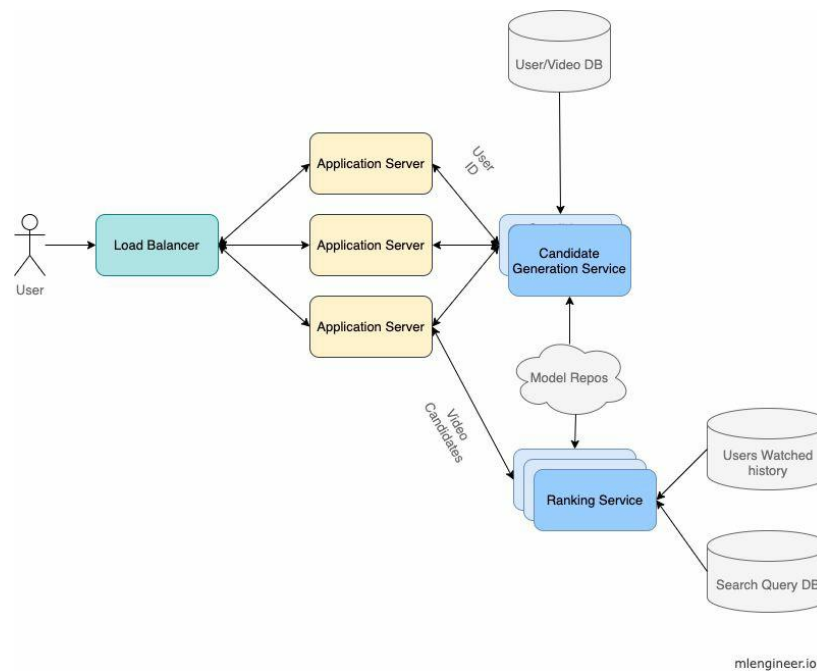
- Video views per month are 150 billion.
- 10% video watched are from recommendations, a total of 15 billion videos.
- On the homepage, users see 100 video recommendations.
- On average, a user watches 2 videos out of 100 video recommendations.
- If users do not click, or watch some video within a time window, (e.g., 10 minutes), then it is a missed recommendation.
- Total number of users are 1.3 billion.
- During one month, we collected 15 billion positive labels and 750 billion negative labels.
- Generally, we can assume for every data point we will collect hundreds of features. For simplicity, each row takes 500 bytes to store. In one month, we need 800 billion rows.
- Total data size: $500 \times 800 \times 10^9 = 4 \times 10^{15}$ bytes = 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 and Scale

- Assume every second we have to generate a recommendation request for 10 million users. Each request will generate ranks for 1k-10k videos.
- Support 1.3 billion users.

System Design

Training



Video Recommendation High Level Design. Source: mlengineer.io

- **Database:** User Watched history stores which videos are watched by particular users over time. Search Query DB stores historical queries that users search in the past. User/Video DB stores a list of Users and their profiles along with Videos metadata. User historical recommendations stores past recommendations for a particular user.
- **Resampling data:** scaling the training process by downsampling negative samples.
- **Feature pipeline:** a pipeline program to generate all required features for the training model. It's important for the feature pipeline to provide very high throughput as we require retraining models multiple times within days. We can use Spark (in-memory computation engine) or Elastic MapReduce or Google Dataproc (cloud-native Spark and Hadoop).

- Model Repos: storage to store all models, using AWS S3 (AWS Simple Cloud Storage) 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 inference component is to frequently pull the latest models from Model Repos based on timestamp.

Challenges

- Huge data size. Solution: Pick one month or six months of recent data. Train and fine-tuned existing DL model (section [Retraining](#) [\[retraining\]](#)).
- Imbalance data. Solution: Perform random negative downsampling.
- High availability. Solution: Model-as-a-service, each model will run in Docker containers. Solution: We can use Kubernetes to auto-scale the number of pods.

Inference

When user request video recommendations:

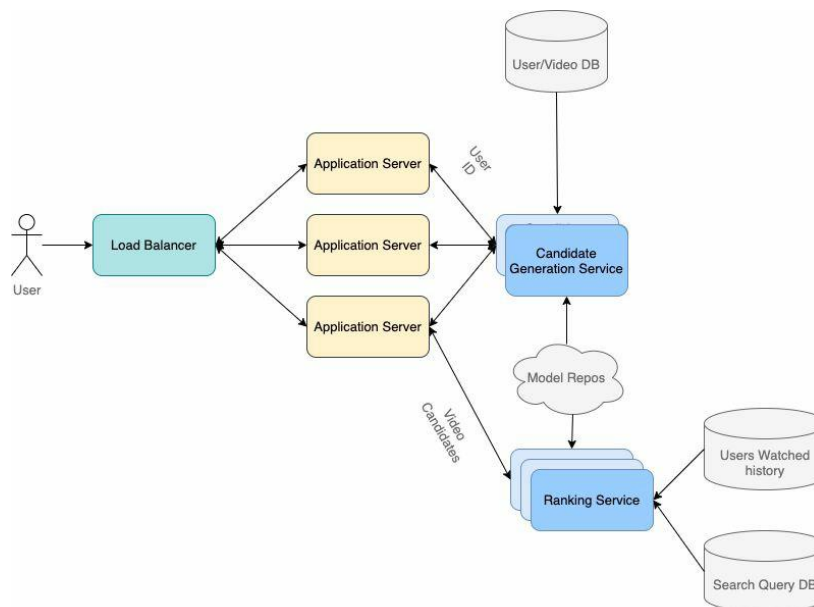
1. Application Server then requests video candidates from the candidate generation model.
2. Once it receives the candidates, it then passes the candidate list to the ranking model to get a sorting order.
3. The ranking model estimates the watch probability and returns the sorted list to the Application Server.
4. Application Server then can return the top videos that a user should

watch.

Scale the Design

- Scale out multiple Application Server and use Load Balancer to balance loads.
- Scale out multiple candidate generation services and ranking services.

We can also use 'Kube-proxy' so that the candidate generation service can call the ranking service directly and reduce latency even further. Read section [Deployment \[deployment\]](#).



mlengineer.io

Video Recommendation - Scale the Design. Source: mlengineer.io

Interview Exercise

- How do you adapt to the change in user behavior over time? 1. Read more about Multi-arm bandit. 2. Use Bayesian Logistic Regression Model, so we can update as needed. 3. Use different loss functions to be less sensitive with click-through rate, etc. Refer to section Loss function [1.3](#).
- How to handle when the ranking model is underexplored? We can introduce randomization in the ranking service.

Summary

- We first learned to separate recommendations into two services: candidate generation service and ranking service.
- We also learned using a multilayer perceptron as a baseline model and how to handle feature engineering.
- To scale the system and reduce latency, we learned to use ‘kube-flow’ so candidate generation services can communicate with ranking services directly.

Question 3

What are the following benefits of splitting the candidate generation service and ranking service? (select all correct answers, answer at foot of page.)

To build, deploy, release each service independently. Reduce memory requirement in each service, so each service can be deployed in individual pods. Scale candidate generation service and ranking service independently. Ranking service can load ML models faster.

Question 4

How often should you retrain models?(answer at foot of page.)

Once every few months At one-minute intervals It depends. Ideally, we want to train a new model when an old model is already finished training. It's also good to balance operation cost vs. online metrics improvement. improvement.

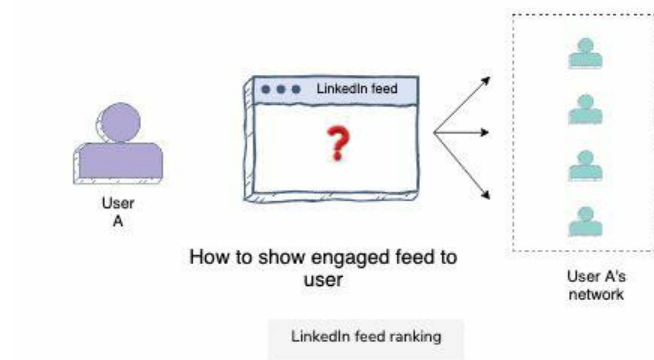
1. Answer: B↵

2. Answer: B↵

LinkedIn Feed Ranking

Problem Statement

Design personalized LinkedIn feeds to maximize long-term user engagement.



Feed Ranking. Source: mlengineer.io

We can measure the click probability or known as ClickThroughRateClick Through Rate (CTR). On the LinkedIn feed, there are five major activity types:

- Connections type (A connects with B)
- Informational
- Profile
- Opinion and site-specific.
- Intuitively different activities have very different CTR. This is important when we decide to build models and generate training data.
- Connection: Member connects/follows member/company; member joins group
- Informational: Member/company shares article/picture/message, etc.
- Profile: Member updates profile (e.g., picture, job-change, etc).

- Opinion: Member likes/comments on article, picture, job-change, etc.
- Site-Specific: Member endorses member, etc.

Challenges

- Scalability: The volume of users activities are extremely large and the LinkedIn system needs to handle 300 million users.
- Personalization: Support for a high level of personalization since different users have varying taste and style of consuming feed.
- Data freshness: Avoid showing repetitive feed on user home feed.

Metrics Design and Requirements

Metrics

Offline Metrics

Maximizing CTR can be formalized as training a supervised binary classification model. For offline metrics, we normalize cross entropy and AUC. Read section Loss function [1.3](#).

$$NCE = -\frac{1}{N} \sum_{i=1}^N (y_i \log(p_i) + (1-y_i) \log(1-p_i)) - (p \log(p) + (1-p) \log(1-p))$$
$$NCE = \frac{1}{N} \sum_{i=1}^N \left(\frac{1+y_i}{2} \log(p_i) + \frac{1-y_i}{2} \log(1-p_i) \right) - (p \log(p) + (1-p) \log(1-p))$$
Normalizing cross entropy helps the model be less sensitive to background CTR.

Online Metrics

For nonstationary data, offline metrics are usually not a good indicator of good performance. Online metrics need to reflect the level of engagement of users once models have been deployed, (i.e., *Conversion rate*), which is the ratio of clicks with the 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 be able to retrain the models (incrementally) multiple times per day.

Inference

- Scalability: To serve user feeds for 300 million users.
- Latency: When a user goes to LinkedIn, multiple pipelines and services will pull data from numerous sources before feeding activities into the ranking models. 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 if a user has already seen any particular activity. Otherwise, seeing repetitive activity will compromise the user's experiences. Hence, data pipelines need to run really fast.

Model

Training Data

Problems

Before building any ML models, we need to collect training data. The goal is to collect a lot of data across different types of posts and simultaneously improve user experiences. You can read more details in section Data Generation [\[data-generation\]](#).

Possible Solutions

- Rank by chronicle order: Use this approach to collect click/not-click data. The trade-off is the serving bias because of the user's attention on the first few feeds. Also, there is a data sparsity problem because different activities like job changes rarely happen compared to other activities on LinkedIn.
- Random serving: Leads to bad user experience and also does not help with the sparsity. There is a lack of training data about rare activities.
- Use an algorithm to rank feed. Within the top feeds, permute randomly. Then use clicks for data collection. This approach provides some randomness, and it's 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 machine learning models.

Feature Engineering

- User profile: job title, industry, demographic, etc. For low cardinality, use one hot encoding. For 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. Another approach is to build mixture models.
- Affinity between activity and user; similarity between activity and users.
- Opinion: Member likes/comments on article, picture, job-change, etc.
- Cross features: Combine multiple features. See example in Chapter 1 Machine Learning System Design Primer.

Model

- Use logistic regression with click as target. With the large volume of data, we need to use distributed training: Either Logistic Regression in Spark or Alternating Direction Method of Multipliers.
- We can also use deep learning in distributed settings. For deep learning, 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 reduce the imbalance. It's important to leave the validation set and test set intact to have an accurate estimation of model performance.

Model architecture: Multilayer perceptron (deep neural network).

Evaluation

One approach is to split 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 at the right position, then we record a match. The total match will be considered as total clicks.
- During the evaluation, we will also evaluate how big our training data set should be, and the frequency of retraining the model, among many other hyperparameters.

You can read more details in section Evaluation [1.3](#).

Model Requirements

Training

For a huge data size problem, one solution is to pick one month or six months of recent data. Train and fine-tune the existing DL model.

For imbalance data, we can perform random negative downsampling.

Calculation and Estimation

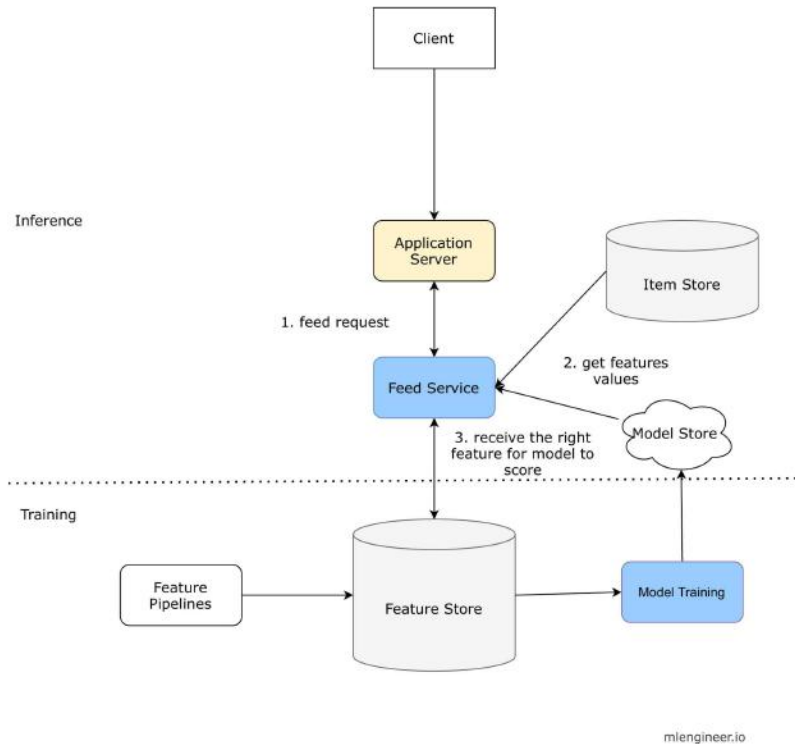
Assumptions

- 300 million monthly active users.
- On average users saw 40 activities per visit. Each user visits 10 times per month.
- We have $12 \times 10^9 \times 10^{10}$ or 120 billion observations/samples.

Data Size and Scale

- Assume click-through rate is about 1%, during one month. We collected 1 billion positive labels and about 110 billion negative labels. It's a huge dataset.
- Generally, we can assume for every data point we will collect hundreds of features. For simplicity, each row takes 500 bytes to store.
- In one month, we need 120 billions rows. Total size: $100 \times 120 \times 10^9 = 12 \times 10^{12}$ bytes = 12 Terabytes. 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.
- Scale: Support 300 million users.

High-level Design



Feed Ranking High-Level Design. Source: mlengineer.io

- User visits LinkedIn homepage and requests Application Server for feeds. Application Server sends feed requests to Feed Service.
- Feed Service gets the latest model from Model Repos, retrieves the right features from Feature Store, and receives all feeds from ItemStore. Feed Service will provide features for the model to get a prediction.
- Model will return recommended feeds sorted by CTR likelihood.

Feed Ranking Flow

Feature Store

Feature store is a key-value storage to store features values. During inference,

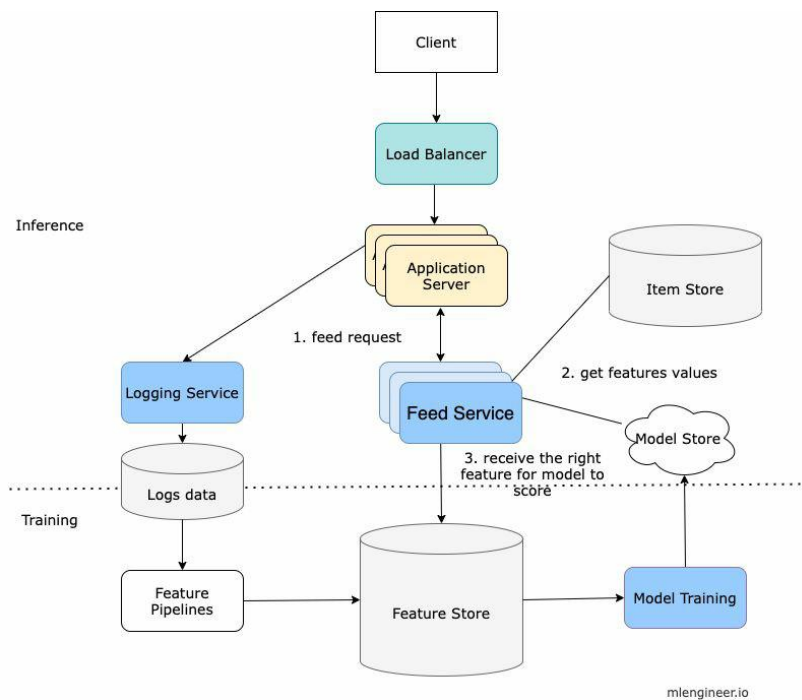
we need low latency to access features before scoring. During inference, we need low latency to access features before scoring.

Items Store

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

Scale the Design

- Feed Service represents both the retrieval service and ranking service for better visualization.
- Scale out Application Server and put Load Balancer in front of Application Server to balance load.



Feed Ranking High - Scale the Design. Source: mlengineer.io

Summary

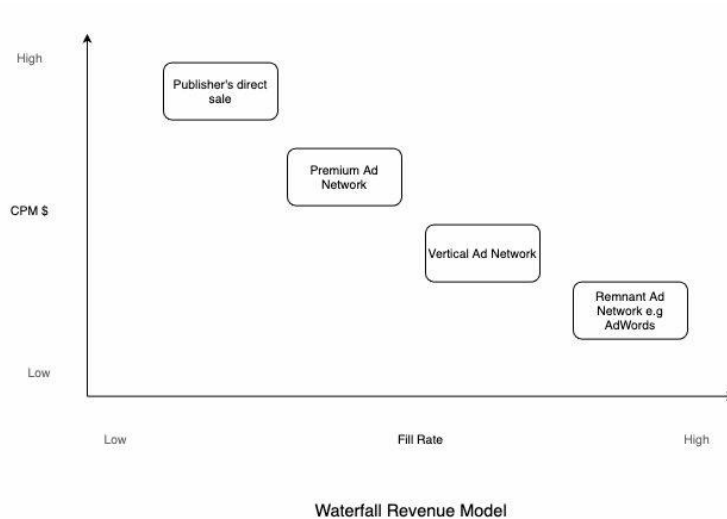
- We learn how to build a machine learning model to rank feeds. Specifically using binary classification with a custom loss function helps the model be less sensitive to background CTR.
- We learn how to design processes to generate training data for a machine learning model.
- We learn how to scale training and inference by scaling out Application Server, Feed Service.
- To retrain model, read section [\[retraining\]](#) Retraining.

Ad Click Prediction

Problem Statement

Build a machine learning model to predict if an ad will be clicked. For simplicity reasons, we will not focus on the cascade of classifiers that are commonly used in AdTech.

Ads request goes through a waterfall model where publishers try to sell their 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.



Adranking Waterfall Model. Source: mlengineer.io



Adclick High-level. Source: mlengineer.io

Challenges

- Latency: Ads 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 overspending the campaign budget and publishers lose money.

Metrics Design and Requirements

Offline metrics

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

- Normalized Cross Entropy (NCE): NCE¹ is the predictive log loss divided by the cross entropy of the background CTR. This way NCE is insensitive to background CTR.
- Calibration metrics are measured by the expected clicks vs. the actual observed clicks.

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

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: CTR is very small in practice (1% to 2%), which makes supervised training difficult. We need a way to train models that can handle highly imbalanced data.
- Retraining frequency: The ability to retrain models many times within

one day to capture data distribution shifts in a production environment.

- Train/validation data split: To simulate a production system, training data and validation is partitioned by time.

Inference

Serving: low latency (50ms - 100ms) for ad prediction.

Model

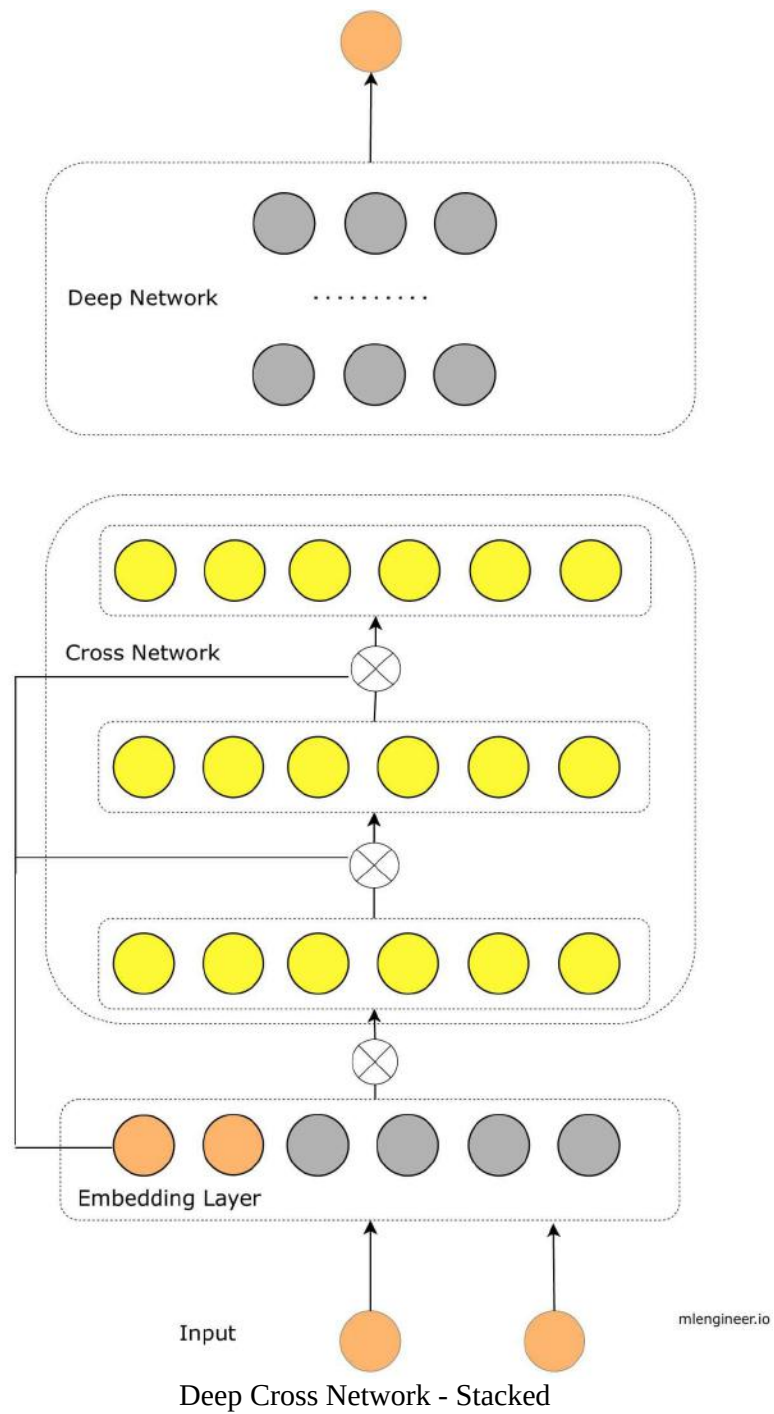
- We can use a probabilistic sparse linear classifier (logistic regression). It's popular because of its computation efficiency. Therefore, it is able to deal well with sparse features.
- One way is subsampling the majority negative class at different subsampling ratios. The key here is to ensure that the validation dataset has the same distribution as the test data set. We also need to pay attention to how this sampling affects predictions.
- Post-processing: Apply calibration for model prediction.

Feature Engineering

- AdvertiserID: Use Embedding or feature hashing since there can be several thousands of advertisers.
- Temporal: time_of_day, day_of_week, etc., using OneHotEncoding.
- User's historical behavior, such as the number of clicks on ads over a period of time with feature scaling (i.e., normalization).
- Connection strength between users are represented by the similarity between users. We can also use embedding for users and measure the distance vector.
- Cross features: combine multiple features. See example in Machine Learning System Design Primer.

Model Architecture

We can use DCNv2 model architecture.



Calculation and Estimation

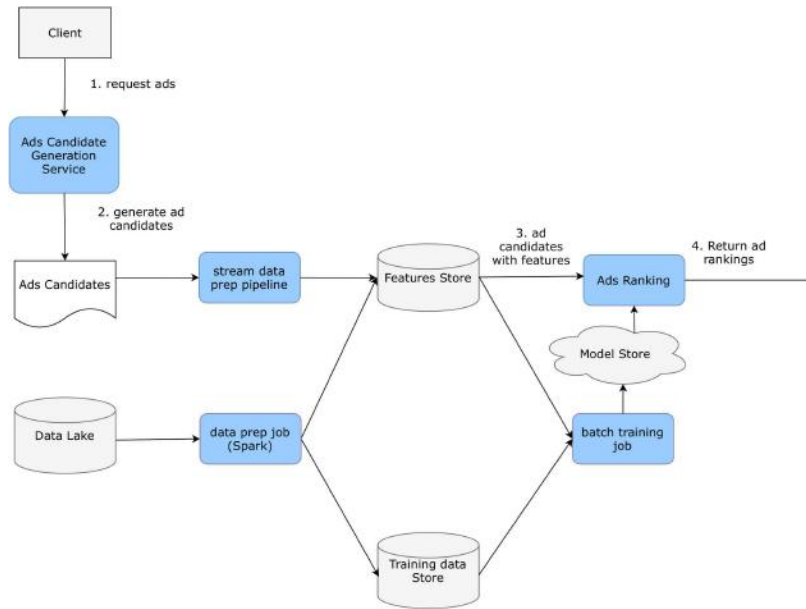
Assumptions

- 40k ads requests per second or 100 billion ads requests per month.
- Each observation (record) has hundreds of features, and it takes 50 bytes to store.

Data Size

- Data: historical ad clicks data includes (user, ads, click_or_not). With an estimated 1% CTR, it has 1 billion clicked ads. We can start with one month of data for training and validation. Within a month, we have $100 * 10^9 * 500 * 100 * 10^9 * 500 = 5 * 10^{13}$ bytes or 50 TB. One way to make it more manageable is to downsample the data (i.e., keep only 1% to 10% or use one week of data for training data and use the next day for validation data).
- Scale: Support 100 million users.

High-level Design



mlengineer.io

Adranking High-level Design. Source: mlengineer.io

Training

- Data is collected from multiple sources. For example, logs data, event-driven data (Kafka) and stored in Data Lake.
- Batch data prep jobs take care of ETL (Extract, Transform and Load) and then put data into the 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 is a distributed storage like S3 to store models.

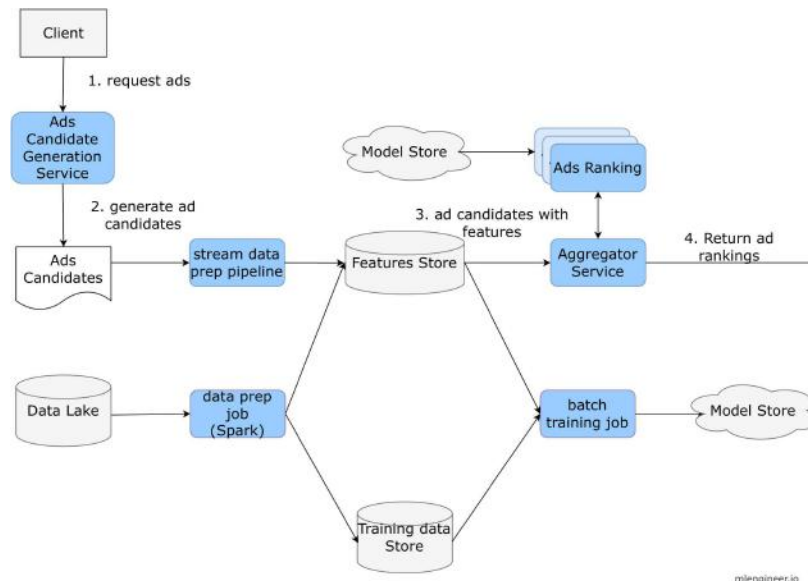
Serving

- **Ads Candidates:** Set of ads candidates provided by upstream services

(refer back to waterfall model).

- Stream data prep pipeline: Process online features and store features in key-value storage for low latency downstream processing.
- Model Serving: Standalone service that loads different models and provides ads click probability.

Scale the Design



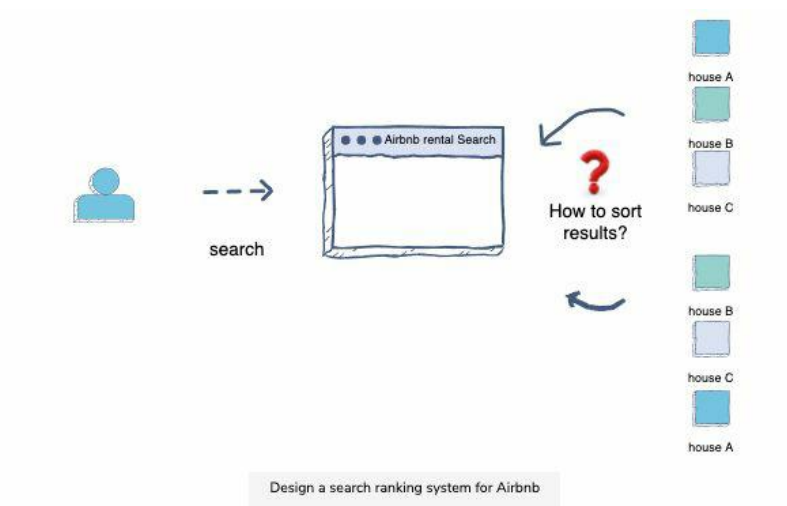
Adranking - Scale the Design. Source: mlengineer.io

- Given a latency requirement of 50ms-100ms for a large volume of ads candidate (50k-100k), if we partition one serving instance per request we might not achieve Service Level Agreement (SLA).
- 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 in Deployment [\[deployment\]](#) section.

Airbnb Rental Search Ranking

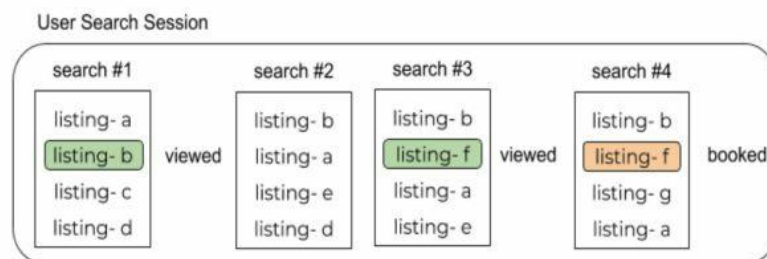
Problem Statement

Airbnb's users search for available homes for a particular location. The system should sort homes from multiple properties in the search result so that the most likely booked homes appear on top.



Search Ranking. Source: mlengineer.io

A typical user session looks like this:



Search Ranking: User Session. Source: AirBnB

- The naive approach would be to craft a custom score ranking function, such as a score based on text similarity given a query. It 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 may build a supervised ML model to predict booking likelihood. This is a binary classification model (i.e., classify booking and not-booking).

Challenges

- Latency: return top relevant rental results to users from thousands of possible matches within 200 ms.

Metrics Design and Requirements

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: $nDCG_p = \frac{DCG_p}{IDCG_p}$
- NDCG is ideal discounted cumulative gain: $NDCG_p = \frac{DCG_p}{IDCG_p}$

Example table

Online Metrics

Conversion rate and revenue lift: It measures number of bookings per number of search results per user session.

$conversion_rate = \frac{\text{number_of_bookings}}{\text{number_of_search_results}}$

Requirements

Training

- Imbalance data and clear-cut session: An average user might do extensive research before making a decision on booking. As a result, the number of non-booking labels has a higher magnitude than booking labels.
- Train/validation data split: Split 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 for validation data.

Inference

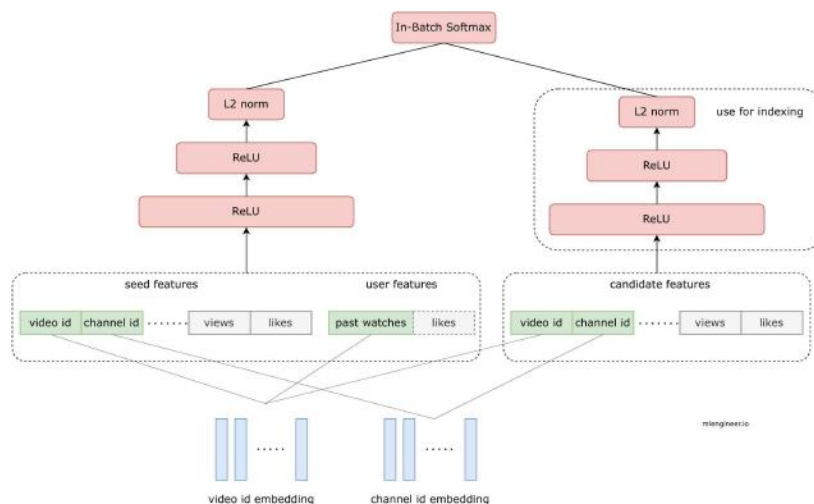
- Serving: Low latency (50ms - 100ms) for search ranking
- Under-predicting for new listings: Brand-new listings might not have enough data for the model to estimate likelihood. As a result, the model might end up underpredicting for a new listing. How can we handle unseen listingIds? Can we apply feature hashing for listingIds?

Model Training

Training Data

- User search history, view history, and bookings: We can start by selecting a period of data: last month, last six 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



Search Ranking: Two-Tower Architecture. Source: mlengineer.io

- Input: user data, search query, and listing data.
- Output: whether user books a rental or not (binary classification).

Feature Engineering

- Geolocation of listing (latitude/longitude): Taking raw latitude and raw longitude features is very noisy to model as feature distribution is not smooth. One way is to take a log of distance from the center of the map for latitude and longitude separately.
- Guest favorite at city neighborhood: Use map grid as two dimensions grid and encode in such a way to reserve neighborhood locality with a query. For example, a user search for San Francisco should translate into a specific cell and be transformed before training/serving.
- ListingID: apply listingID Embedding
- Listing feature: Number of bedrooms, list of amenities, listing city.
- Location: Measure latitude and longitude from the center of the user map, then normalize.
- Historical search query using text embedding.
- User associated features: age, gender with feature scaling (i.e., normalization or standardization).
- Number of previous bookings with feature scaling (i.e., normalization or standardization).
- Number of previous length of stays with feature scaling (i.e., normalization or standardization).
- Time-related features: month, weekofyear, holiday, dayofweek, hourofday.

Calculation and Estimation

Assumptions

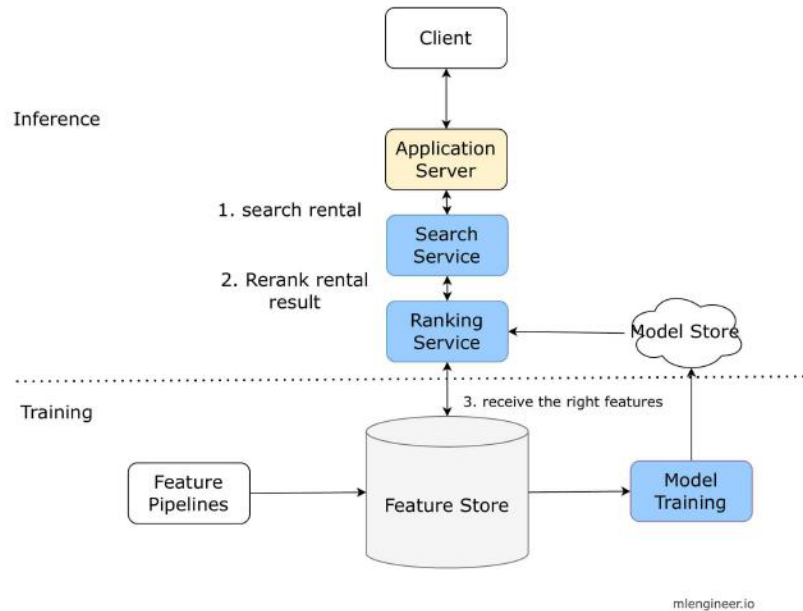
- 100 million monthly active users.
- On average, users book rental homes five times per year. Users see about 30 rentals from the search result before booking.
- There are $5 \times 30 \times 10^8 \times 30 \times \{10^8\}$ or 15 billion observations/samples per year or 1.25 billion samples per month.

Data Size

We need to estimate data size, bandwidth etc to inform our choice of technical components.

- Assume there are hundreds of features per sample. For simplicity, each row takes 500 bytes to store.
- Total data size: $500 \times 1.25 \times 10^9 = 625 \times 10^9$ bytes = 625 GB. 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.
- Scale: Support 150 million users.

High-level Design

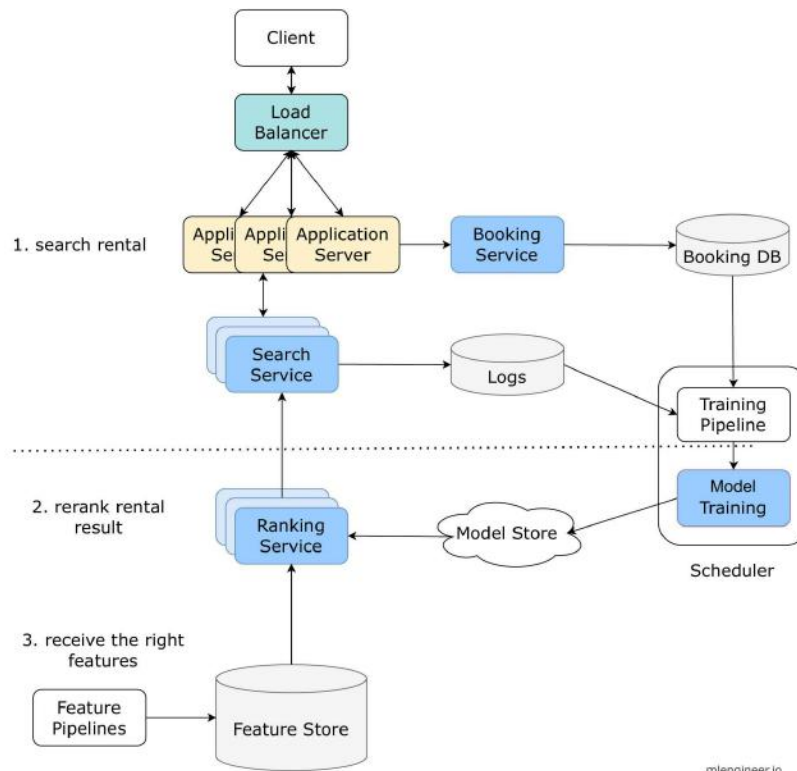


Airbnb Search Ranking High-level Design. Source: mlengineer.io

- When users search for rentals with given queries, for example, city or time. Application Server receives the query and sends requests to the search service.
- Search service looks up indexing database and retrieves a list of rental candidates and sends candidates list to the ranking service.
- Ranking service uses a machine learning model to score each candidate. The score represents how likely the user will book a specific rental. Ranking service returns the list of candidates with a score.
- Search service receives candidates list with booking probability and uses the probability to sort candidates. It then returns a list of candidates to Application Server, which in turn, returns to users. You can refer to the previous sections of Feature Pipeline, Feature Store, and Model Store for more details.

As you can see, we start with a simple design, but it will 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.

Scale the Design



Airbnb Search Ranking: Scale Design. Source: mlengineer.io

- 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 also scale out the search service and ranking service to handle millions of requests from Application Server.
- Lastly, we also 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.

Open Questions

- 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.
- As we have thousands of new listings every day, how can we handle unseen listingIDs?
- Assuming there is a strong correlation between how long users spend on listing with booking likelihood, how would you redesign network architecture? Train multiple output networks for two outputs: view_time and booking.
- How often do you retrain models? Balance between online metrics and training time.

Summary

- We learn to formulate the search ranking as a machine learning problem by using booking likelihood as a target.
- We learn to use ‘Discounted Cumulative Gain’ as one metric to train the model.
- We also learn how to scale our system to handle millions of requests per second.

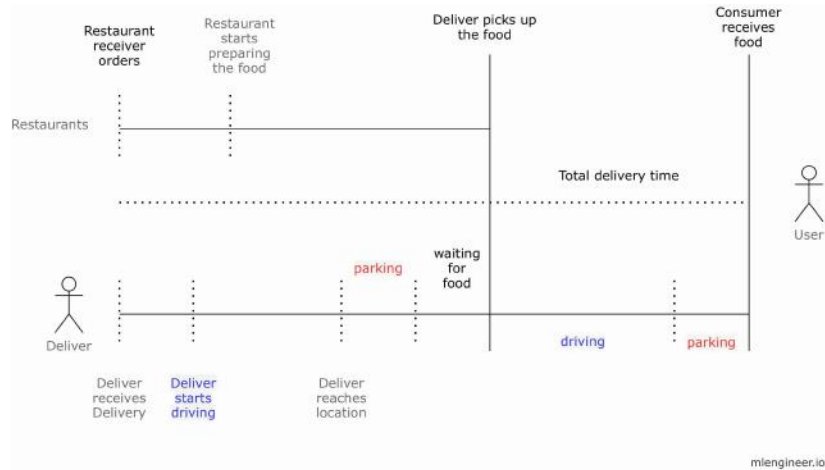
Further reading: Applying Deep learning to AirBnb Search¹

1. <https://arxiv.org/pdf/1810.09591.pdf>[↵]

Estimate Food Delivery Time

Problem Statement

Build a model to estimate total delivery time.



Food Delivery Workflow. Source: mlengineer.io

For simplicity, we do not consider batching (group multiple orders at restaurants). Inputs: Order details, market conditions, and traffic status.

$$\text{Delivery Time} = \text{Pickup Time} + \text{Point_to_Point Time} + \text{Drop_off_Time}$$

Metrics Design and Requirements

Metrics

Offline Metrics

Use Root Mean Squared Error (RMSE) $\sqrt{\sum_{k=1}^n (\text{predict} - y)^2} / \sqrt{n}$

where, n is a total number of samples, predict is estimated wait time, y is the actual wait time.

Online Metrics

A/B testing and monitoring RMSE, customer engagement, customer retention etc.

Requirements

- Training: Balance between overestimation and underestimation. For this, retrain multiple times per day to adapt to market dynamic and traffic conditions.
- Inference: For every user to visit the homepage, the system recommends 100 videos to them with a latency under 200ms or ideally sub 100ms.

Summary

- Metrics: Estimation should be less than 10-15 minutes. If we overestimate, customers are less likely to make orders. Underestimation can cause unsatisfied customers.
- Training: High throughput with the ability to retrain many times per day.
- Inference: Low latency from 100ms to 200ms.

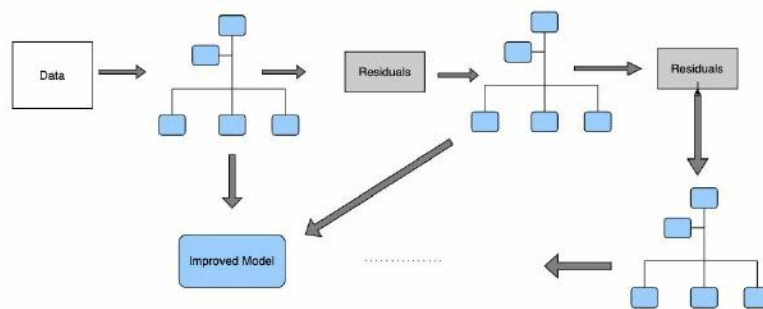
Model

Training Data

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

Model

Use gradient boosted decision trees to predict total delivery time.



Gradient Boosted Decision Tree Sample. Source: mlengineer.io

Actual Delivery Time=Estimated Delivery Time+Error
 $\text{Actual Delivery Time} = \text{Estimated Delivery Time} + \text{Error}$

The disadvantage is that RMSE penalties are similar between underestimate prediction and overestimate prediction.

Summary table

Actual	Model 1	Square error1	Model 2	Square error2
	Prediction		Prediction	
30	34	16	26	16

35	37	4	33	4
----	----	---	----	---

- Even though Model 1 and Model 2 have the same RMSE error, Model 1 overestimates delivery time, which prevents customers from placing orders. Model 2 underestimates delivery time and might cause customers to be unhappy.
- The next model addresses this issue by integrating confidence intervals.

Probabilistic Model with Confident Interval: Quantile Regression

$\text{prob}(\text{Actual Delivery Time} > \text{Estimated Delivery Time}) < X \text{ } \text{prob}(\text{Actual Delivery Time} < \text{Estimated Delivery Time}) < X\%$

$L(y, F(x)) = h_q(y - F(x))$ $L(y, F(x)) = h_q(y - F(x))$ where,

- $h_q = z * (q - I_{z < 0})$ $h_q = z * (q - I_{\{z < 0\}})$
- q is the percentile value to use
- I is the indicator function

Instead of using RMSE as error, we use quantile loss.

- Choosing two q values (0.05; 0.95) gives you a prediction interval where 90% predictions would be inaccurate.
- Model evaluated using % predictions within the interval and average estimated value.
- Ideal value of q identified through experiment to optimize for retention.

Features Engineering

- Order features: subtotal, cuisine.
- Item features: price and type.

- Order type: group, catering.
- Merchant detail.
- Store ID & Store Embedding.
- Real-time 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.

System Design

Requirements

Inference

Near –real-time update, any changes on order status need to go through model scoring as fast as possible (e.g., the restaurant starts preparing meals, a driver starts driving to customers). It's a natural fit for event-driven design with Kafka.

- Update by pushing instead of pooling. Whenever there are changes in delivery, model scoring is triggered instantly and updates the customers.
- Capture near real-time aggregated statistics, such as feature pipeline aggregates data from multiple sources (Kafka, database) to reduce latency.

Training

- Data preparation reads data from the database and stores it in a training data storage. To optimize for high throughput, store in Parquet files (see more in Chapter 1. Training Pipeline).
- The model should undergo retraining every few hours. Delivery operations are under a dynamic environment with many 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 days, 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 during the day.

Calculation and Estimation

Assumptions

For simplicity, we can make these assumptions:

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

Data Size

- During one month, we collected 2 million deliveries. Each delivery has around 500 bytes related features.
- Total size: $500 \times 2 \times 10^6 = 10^9$ bytes (1TB)
- Scale: Support 20 million users.

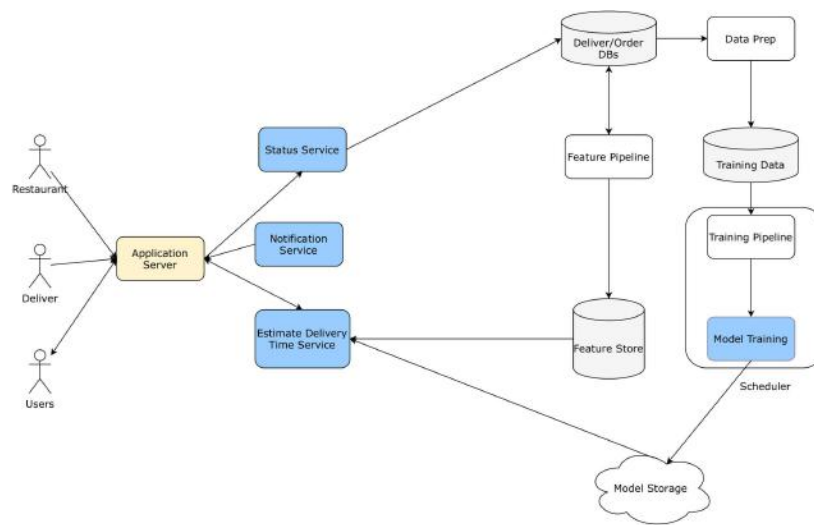
High-level Design

Inference

- Feature Store: provides fast look up 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 and transforms, aggregates near real-time statistics, then stores in feature storage.

Training

Needs to support batch read/write with high throughput for millions of rows per second. Feature pipeline stores data in S3 with Parquet format and uses Spark to generate training data.



Food Delivery: High-level Design. Source: mlengineer.io

There are three main types of users: consumer/user, delivery, and restaurant.

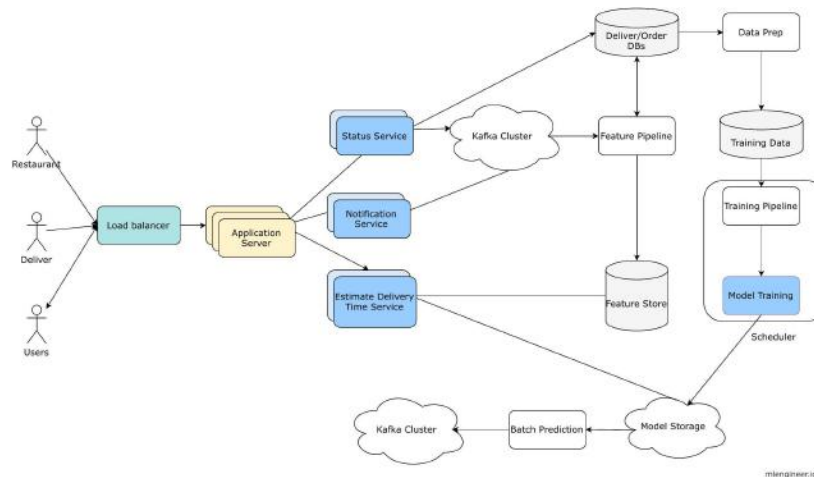
- Database: Delivery order database stores historical orders and delivery;

data prep is a process to create training data from 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 a model storage.

Scale the Design

- We scale out our services to handle large requests per seconds. We also use Load Balancer to balance loads across Application Servers.
- We leverage streaming process systems like Kafka to handle notification as well as model predictions. Once the machine learning model completes predictions, it sends out predictions to Kafka, so other services can get notifications right away.



Food Delivery: Detail. Source: mlengineer.io

Summary

- We learn to formulate estimate delivery time as a machine learning problem using the Quantile Regression model.
- We learn how to collect and use data to train the model.
- We learn to use Kafka to handle logs and model predictions for near real-time systems.

Machine Learning Assessment

Practice 1: Machine Learning Knowledge

This exam has been designed to test the machine learning model concepts (diagnosis and data preparation) that you have learned.

The exam consists of 8 multiple choice questions. You are given three possible answers and are asked to select one (and only one) correct answer from the four answers given.

The following concepts/skills of machine learning will be assessed in this exam: regression, confident interval, forecast model, correlation, clustering, data structures, SQL, and table optimization

Good Luck!



Regression

Question 1: Elon works as machine learning engineer at a contract company at Kennedy Space Center. Weather is the key factor for a successful launch. Elon needs to build a weather forecast model. The model needs to predict the temperature in the next 15 minutes. His boss said if the prediction is off by 1.5 degrees, then it's not good enough. Given the average temperature is about 20 Celsius, which metrics should he optimize for?

- (A) Make sure Mean Squared Error (MSE) metric is less than 1.5
- (B) Make sure Mean Absolute Error (MAE) metric is less than 1.5
- (C) Make sure Mean Absolute Percentage Error (MAPE) is less than 7.5 %

[answer](#)

Confidence Interval

Question 2: After finishing his best model, suppose the prediction for the next

minute is 20 Celsius degrees and the MSE is 2.25 (i.e., MAE is 1.5). Assume the confidence interval is 95%95\%, what is the lower bound and upper bound of the prediction? Assume the forecast errors are normally distributed with a mean of zero.

(A) [18.275, 21.725]

(B) [15, 25]

(C) [18.5, 21.5]

[answer](#)

Forecast Model

Question 3: Elon wants to deploy the latest forecast model to the ISS machine, so ISS astronauts can also run the forecast by themselves without waiting for the forecast from Earth. To do that he has to serialize the forecast model and send it to ISS. However, the machine on ISS only runs Java, while the forecast model is in Python. What do you recommend him to do? Please choose the BEST possible answer.

(A) Serialize to pickle file then use Java to read pickle file when loading the model.

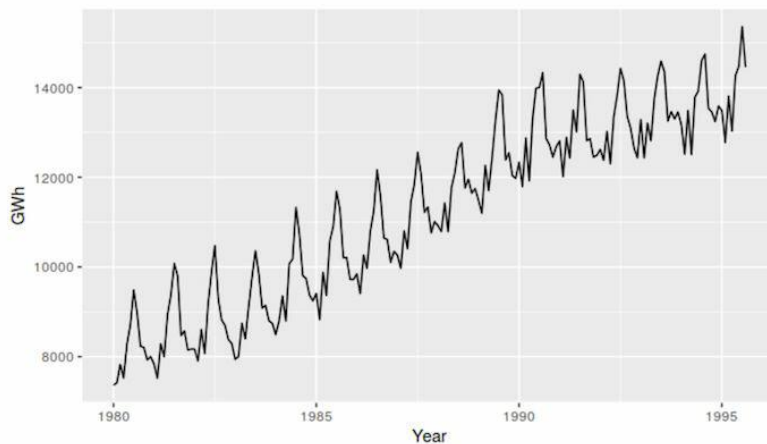
(B) Serialize to CSV file because Java can also read CSV files.

(C) Use ProtocolBuffer.

[answer](#)

Correlation

Question 4: This graph shows the electric consumption overtime at SpaceX headquarters, Hawthorne, CA. If we compute the correlation between this year and the previous year, what do you think its value would be?



Correlation

- (A) Negative
- (B) 0
- (C) High value, close to 1

[answer](#)

Coding

Question 5: The below code (Python) implements a depth-first search for a binary tree. We want to get results as a list of nodes with in order traversal. What problem would you see with this code?

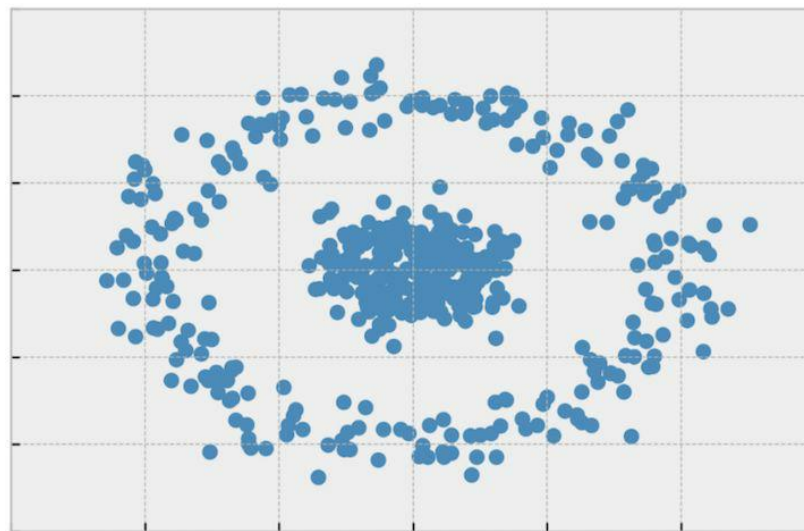
```
def dfs_iterative(root):
    stack, res = [], []
    n = root
    while n or len(stack) > 0:
        if n:
            stack.append(n)
            n = n.left
        n = stack.pop()
        res.append(n.val)
        n = n.right
    return res
```

- (A) It's not preorder traversal, not in order traversal.
- (B) It's post order traversal, not in order traversal
- (C) The result does not have all the tree nodes
- (D) The result has duplicated nodes

[answer](#)

Clustering

Question 6: We want to separate 2D points into two groups. Which algorithm is suitable? (Multiple choices)



Coding

- (A) Run PCA to reduce dimensions then apply k-means with number of clusters = 2
- (B) Use Spectral clustering algorithms

[answer](#)

SQL

Question 7: In the SpaceX employee database, we have two tables: Worker and Title. To celebrate the Crew Dragon launch, the board decides to give them a bonus. Write a query to return (unique) names and titles of all the workers who are also a manager. (*answer at foot of page.*)

Worker table & Title table						
Worker					Title	
workerid	first	last	salary	department	workerid	title
1	Monika	Arora	100000	HR	1	Manager
2	Niharika	Verma	80000	Admin	2	Executive
3	Vishal	Singhal	300000	HR	8	Executive
4	Amitabh	Singh	100000	Admin	1	Manager
5	Vivek	Bhati	100000	Admin	1	Asst. Manager
6	Vipul	Diwan	100000	Account	1	Executive
7	Satish	Kumar	100000	Account	1	Lead
8	Geetika	Chauhan	100000	Admin	1	Lead

(A)

```
select distinct W.first, T.title
  from Worker W
 inner join Title T on W.workerid = T.workerid
 and T.title in ('Manager');
```

(B)

```
select W.first, T.title
  from Worker W
 left join Title T on W.workerid = T.workerid
 and T.title in ('Manager');
```

(C)

```
select W.first, T.title
```

```
from Worker where T.title in ('Manager');
```

(D) All of the above.

[answer](#)

Database

Question 8: As SpaceX grows, they now have a lot of employees and contractors. Because of this, the above query can take quite some time to finish. The current table design has `workerid` as a String column and use `Lastname` as index. What do you recommend improving the design, so the query can run faster? Please choose the BEST possible answer.

(A) Change `workerid` as numeric column, create an index on `workerid`.

(B) Create index on `workerid` column and `title` column.

(C) Create index on all columns.

[answer](#)

Machine Learning Model Diagnosis

This exam has been designed to test the machine learning model concepts (diagnosis and data preparation) that you have learned.

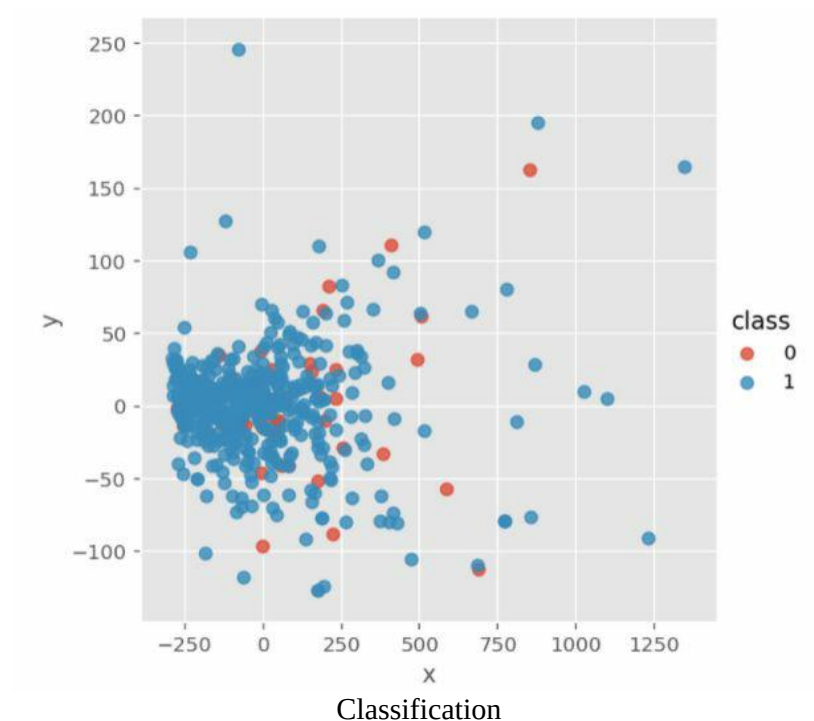
You are given four possible answers and are asked to select one (and only one) correct answer from the four answers given.

The following concepts/skills of machine learning will be assessed in this exam: classification, feature important, confusion matrix, random forest, statistics, random forest tuning, decision tree tuning, model deploy, and deep learning.

Good Luck!

Classification

Question 1: In his social network company, data scientist John Doe needs to build a classifier to detect hated speech. He collected data from users' posts and labeled it from the community flag. The data has 3,000 columns, and he ran PCA: Principal Component Analysis to reduce it to two dimensions and visualize two classes. To improve model performance, what metrics should he optimize for? (*answer at foot of page.*)



(A) Matthews Coefficient Score

(B) Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC)

[answer](#)

Feature Important

Question 2: John wants to choose some features for his model. One common idea is to use important features from random forest. John then runs this function: `get_feature_importance(full_data_set)` Do you agree with this method?

(A) Yes

(B) No, we should run `get_feature_importance` on the training set only.

(C) No, should use deep learning to `get_feature_importance` on

full_data_set.

(D) No, we should run `get_feature_importance` on the whole data set.

[answer](#)

Confusion Matrix

Question 3: Once they finish building the classifier, the business team is happy with the reasonable performance on the train set. However, they see this report for the test set and are concerned because several non-hatred posts are detected as hatred. What would you recommend John verify?

Confusion Matrix		
Predicted		
	normal	hatred
normal	600	400
hatred	10	100

(A) Reduce test size

(B) Check if the resampling cause 50% of hatred observations in training set

(C) Check if current threshold 0.5 in his logistic regression classifier

[answer](#)

Classification metrics

Question 4: Suppose we have a binary spam classifier. We know the following:

- 1% of all email is spam
- When an email is spam, the model classifies it as "spam" 99% of the

time

- When an email is not spam, the model classifies it as "not spam" 99% of the time

Question: What is the precision of this model? (*answer at foot of page.*)

(A) It depends.

(B) 99%

(C) 50%

[answer](#)

Random Forest

Question 5: Random forest has the ability to reduce variance because ...

(A) It trains on different samples of data

(B) Uses random subset of features

(C) All of the above

(D) None of the above

[answer](#)

Statistics

Question 6: If you flip a coin 50 times and try to land on heads, what is the variance of the binomial distribution?

(A) 12.5

(B) 14

(C) 12

[answer](#)

Random Forest Tuning

Question 7: For which of the following hyperparameters, higher value is better model performance on the training set for the decision tree algorithm? (1) Number of samples used for split; (2) Depth of tree; (3) Samples for leaf.

(A) Only (1)

(B) Only (2)

(C) Only (3)

(D) (1) and (2)

(E) (2) and (3)

(F) (1), (2) and (3)

[answer](#)

Decision Tree Tuning

Question 7: For which of the following hyperparameters that a higher value is a better model performance on the training set for the decision tree algorithm? (1) Number of samples used for split; (2) Depth of tree; (3) Samples for leaf.

(A) 1 and 2

(B) 2 and 3

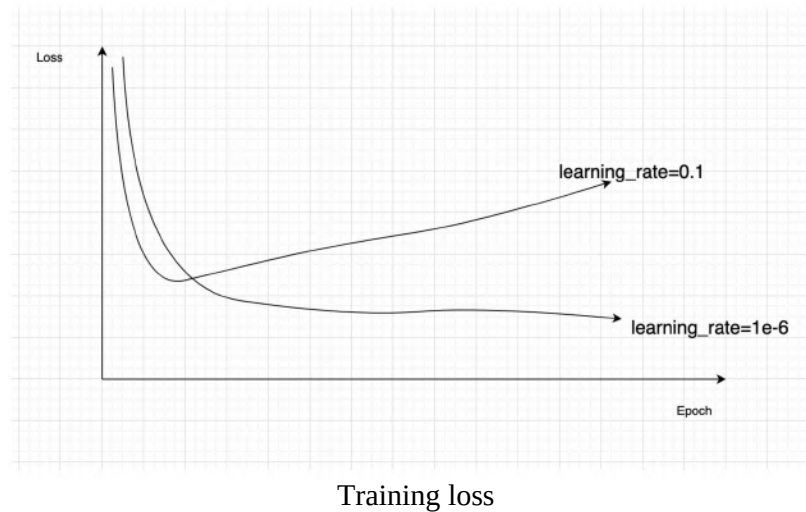
(C) 1 and 3

(D) Can't say

[answer](#)

Deep Learning Diagnosis

Question 9: John wants to improve his model, and he uses deep learning. When he plots training loss for different learning rates, he visualizes this chart. What learning_rate should he try to get better performance?



- (A) Use learning_rate = 0.1
- (B) Use learning_rate = 1e-8
- (C) Use learning_rate = 1e-4
- (D) Increase dropout to 0.5

[answer](#)

Deep Learning

Question 10: John thinks deep learning can get better performance, so he built three layers of “neural networks.” What else would you recommend him to start?

(A) The relu activation should be used for all layers with mean square error as a loss function. If the output is greater than 0.5, then assign it as class 1.

(B) The last layer should use sigmoid, the output will then be as the probability of class 1

.

(C) The last layer should use tan, the output will then be the probability of class 1.

(D) None of the above.

[answer](#)

Deep Learning Questions

Question 1: In deep learning architecture, how can we handle overfitting?

- (A) Apply weight normalization
- (B) Increase model capacity
- (C) Apply dropout
- (D) Use Regularization

[answer](#)

Question 2: Adding L2 regularization to an RNN can help with the vanishing gradient problem?

- (A) False! Adding L2 regularization will shrink the weights toward zero, which can actually make the vanishing gradients worse in some cases.
- (B) Yes.
- (C) Clipping the gradient (cutting off at a threshold) will solve the exploding gradients problem.

[answer](#)

Question 3: Why do we need momentum?

- (A) The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.
- (B) Help converge faster.

[answer](#)

-
1. Answer: B↵
 2. Answer: A↵
 3. Answer: C↵
 4. Answer: C↵
 5. Answer: C↵
 6. Answer: B↵
 7. Answer: A↵
 8. Answer: A↵
 9. Answer: A↵
 10. Answer: B↵
 11. Answer: B↵
 12. Answer: C↵
 13. Answer: C↵
 14. Answer: A↵
 15. Answer: B↵
 16. Answer: D↵
 17. Answer: B↵
 18. Answer: B↵
 19. Answer: A, C, D↵
 20. Answer: C source: Stanford cs224d course↵

21. Answer: C [↩](#)