Ad Selection

Let's see what role the ad selection component plays in the overall prediction system.

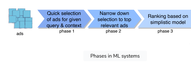
We'll cover the following

- Phase 1: Selection of ads
- Phase 2: Narrow down selection set to top relevant ads
- Phase 3: Ranking of selected ads using a simplistic model
- How does the system scale?

The main goal of the ads selection component is to narrow down the set of ads that are relevant for a given query. In a search-based system, the ads selection component is responsible for retrieving the top relevant ads from the ads database (built using all the active ads in the system) according to the user and query context. In a feed-based system, the ads selection component will select the top k relevant ads based more on user interests than search terms.

Based on our discussions about the funnel-based approach for modeling, it would make sense to structure the ad selection process in the following three phases:

- **Phase 1:** Quick selection of ads for the given query and user context according to selection criteria
- **Phase 2:** Rank these selected ads based on a simple and fast algorithm to trim ads.
- **Phase 3:** Apply the machine learning model on the trimmed ads to select the top ones.



Phases in ML systems

Phase 1: Selection of ads

Advertising platforms can have hundreds of millions of active ads. Our main motivation in this phase is to quickly reduce this space from millions of ads to one ads that can be shown to the current user in the given context (e.g. for a user searching "machine learning" on a mobile device).

The first key requirement to be able to achieve the quick selection objective is to have the ads stored in a system that is fast and enables complex selection criteria. This is where building an in-memory index to store the ads will be massively helpful. Index allows us to fetch ads quickly based on different targeting and relevance information from the user. This is similar to our Search system discussion, where we had to similarly select documents quickly at the selection time to narrow our focus to relevant documents. We will index ads on all possible indexing terms that can be used for selection e.g. targeted terms, city, state, country, region, age etc.

Let's take an example of a search ads system to explain this concept further. Let's say that a male user, aged twenty-five, and located in San Francisco, California is searching stuff related to machine learning. He types in a query "machine learning". In this case, we would want to select all potential ads that we can show to the user, i.e., the ads targeting criteria matches the user's profile.

The selection query in this case will look like:

```
(term = "machine learning")
                and
(age = "\*" or age contains "25")
                and
(gender="\*" or gender="male")
                and
(city = "\*" or city = "San Fracisco")
                and
(state="\*" or state="CA")
                and
(status="Active")
                and
(has_budget = true)
```

Selection query for search system

In a feed-based system, the selection of ads will base more on user interests rather than search terms. Let's assume that the same user is interested in Computer science and football, and we want to now fetch ads for his feed. The selection query will look like the following:

```
(interest = "Computer Science" or interest = "Football" or interest = "*")
                and
(age = "\*" or age contains "25")
                and
(gender="\*" or gender="male")
                and
(city = "\*" or city = "San Fracisco")
                and
(state="\*" or state="CA")
                and
(status="Active")
                and
(has_budget = true)
```

Selection query for feed based system

So, the above selection criteria in phase 1 will reduce our space set from all active ads to **ads that are targeted for the current user.**

Phase 2: Narrow down selection set to top relevant ads

The number of ads selected in phase 1 can be quite large depending on the query and the user, e.g., we can have millions of ads targeted for a sports fan. So, running a complex ML model to predict engagement on all these selected ads will be slow and expensive. At this point, it makes sense to narrow down the space using a simplistic approach before we start running complex models.

The eventual ranking of ads is going to be based on (bid * predicted score). We already know the bid at this point and can use the prior engagement score (CPE - cost per engagement) based on the ad, advertiser, ad type, etc. as our predicted score.

$$(bid) * (prior$$

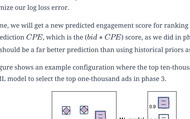
For a new ad or advertiser where the system doesn't have good prior scores, a slightly higher score can be given to ads also called *score boost*. We can use time decay to reduce it. The ranking might look like the following with a new ad boost of 10% to CPE score for the first forty-eight hours of ad with time decay.

```
boost = 0;

if ad_age < 48:
    boost = 0.1 * (1 - ad_age / 48)
    ad_score = (bid) * (1 + boost) * CPE
```

Based on these prior scores, we can narrow down ads from our large selected set to the top k ads. In phase 3, we will use a much stronger predictor than prior scores on the top selected ads from phase 2.

The following diagram shows an example configuration where the selection expression retrieves one million ads from the database. We then select the top ten-thousand ads based on $bid * CPE$ score.



Phase 1 on left with one million ads selected. Phase 2 on the right using bid * prior CPM to narrow it down to top ten-thousand ads

Phase 3: Ranking of selected ads using a simplistic model

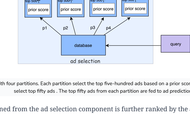
As the ranking in phase 2 is super simplistic, we should still select a sizable ads(e.g., ten thousand) by running a better model as we work towards reducing the ad set. We can run a simplistic and efficient model on ads selected in phase 2 to further narrow it down. The top ads from this phase will be passed to our ad prediction stage to run more complex and accurate models for better user engagement prediction.

We can use either **logistic regression** or **additive trees** based models (such as random forest or boosted decision trees) as they are quite efficient. **Neural network**-based models need more capacity and time so they might not be the best choice at this stage.

The target is to select the top k relevant ads from the set given by phase 2. We will use training data and dense features, which were discussed in previous sections to train this model to predict engagement scores better and minimize our log loss error.

At evaluation time, we will get a new predicted engagement score for ranking of ads. Ads will be sorted based on this prediction CPE, which is the ($bid * CPE$) score, as we did in phase 2. However, our predicted score should be a far better prediction than using historical priors as we did in phase 2.

The following figure shows an example configuration where the top ten-thousand ads from phase 2 are ranked by our ML model to select the top one-thousand ads in phase 3.



Ranking of ads in phase 3 from an efficient ML model of results coming from phase 2

> As discussed earlier in the architectural components lessons, the ad set in each phase is narrowed down for the phase below it, thus making it a funnel-based approach. As we progress down the funnel, the complexity of the models increases, and the number of results decrease. Within the ad selection component, we adopt the same funnel based approach. First, we select ads in phase 1, then rank them based on prior scores in phase 2. Finally, we rank them using an efficient model in phase 3 to come up with the top ads that we want to send to the ad prediction stage.

How does the system scale?

Note that our index is sharded, i.e., it runs on multiple machines and every machine selects the top k ads based on the prior score. Each machine then runs a simplistic **logistic regression** model built on dense features (there are no sparse features to keep the model size small) to rank ads.

> The number of partitions (shards) depends on the size of the index. A large index results in more partitions as compared to a smaller index. Also, the system load measured in queries per second (QPS) decides how many times the partition is replicated.

Consider a scenario where the size of the index is 1 tera-byte and memory of a single shard is 250 giga-bytes. The index data gets distributed in four partitions. Each partition selects ads that satisfy the selection criteria. Then, we use the prior score of the selected ads and select top five-hundred ads based on that score. To further narrow down the ad set, we run logistic regression which returns the top 50 ads. The top fifty ranked ads from each of the four partitions (200 ads in total) are fed to the ad prediction component.

> Note that for each level we are selecting top "k" ads. The selection of the number "k" for each level is very arbitrary. It is based on experimentation and system available load/capacity.

Sharded index with four partitions. Each partition select the top five-hundred ads based on a prior score and then run logistic regression to select top fifty ads . The top fifty ads from each partition are fed to ad prediction component

The ad set returned from the ad selection component is further ranked by the ad prediction component.

Ad Prediction

Let's have a look at how the most relevant ads will be predicted from a set of selected ads.

We'll cover the following

The ad prediction component has to make predictions for the final set of candidate selected ads. It needs to be robust and adaptive and should be able to learn from massive data volume.

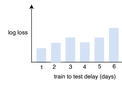
Let's go over the best setup and models for this problem.

Modeling approach

Ads are generally short-lived. So, our predictive model is going to be deployed in a dynamic environment where the ad set is continuously changing over time.

Given this change in an ad set, keeping the model up to date on the latest ads is important. In other words, model performance will degrade with each passing day if it isn't refreshed frequently.

Online learning



If we have to plot the log loss of the model, it might look like the graph on the right. Here we are assuming that the model is trained on day one and we are observing log loss on six consecutive days. We can observe the degradation of prediction accuracy (increase of log loss) because of the increased delay between the model training and test set.

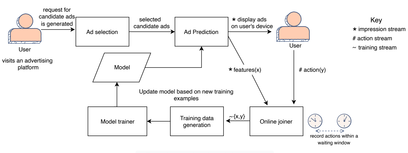
So, keeping models updated with the latest data is important for ad prediction.

One approach to minimize this loss could be refreshing the model more frequently e.g. training it every day on new data accumulated since the previous day.

However, a more practical and efficient approach would be to keep refreshing the model with the latest impressions and engagements after regular internals (e.g. 15 mins, 30 mins, etc.). This is generally referred to as **online learning** or **active learning**.

In this approach, we train a base model and keep adding new examples on top of it with every ad impression and engagement. From an infrastructure perspective, we now need a mechanism that collects the recent ad data and merges it with the current model.

The following figure shows some key components that are critical for enabling online learning. We need a mechanism that generates the latest training examples via an online joiner. Our training data generator will take these examples and generate the right feature set for them. The model trainer will then receive these new examples to refresh the model using stochastic gradient descent. This forms a tightly closed loop where changes in the feature distribution or model output can be detected, learned on, and improved in short successions. Note that the refresh of the model doesn't have to be instantaneous, and we can do it in same batches at a certain frequency, e.g., every 30 mins, 60 mins etc.



Flow of events for ad prediction system

Model for online learning

One model that easily supports online learning and has the ability to update it using stochastic gradient descent using mini-batches is **logistic regression**.

Auto non-linear feature generation

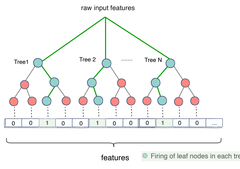
One potential drawback is that simple logistic regression (generalized linear model) relies on manual effort to create complex feature crosses and generating non-linear features. Manually creating such features is cumbersome and will mostly be restricted to modeling the relationship between two or three features. On the other hand, tree and neural network-based models are really good at generating complex relationships among features when optimizing the model.

To overcome this, we can use additive trees and neural networks to find such complex feature crosses and non-linear feature relationships in data, and then these features are input to our logistic regression model.

Additive trees

To capture complex feature crosses and non-linear relationships from the given raw features, additive trees can be extremely handy. We can train a model to predict the probability of engagement using trees and minimize our loss function.

We can represent an additive tree to generate features by taking into account triggered leaf nodes, e.g., let's say that we have one-hundred trees in our ensemble with four leaf nodes each. This will result in 100 * 4 = 400 leaf nodes in which one-hundred nodes will trigger for every example. This binary vector itself captures the prediction (or learning) of our tree ensemble. We can just feed this vector as one input feature to our logistic regression model.

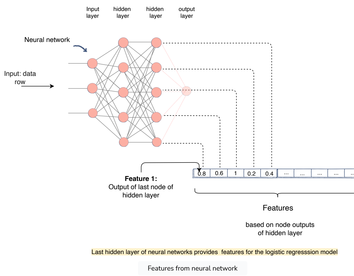


Features from additive trees

Neural Network

A deep neural network can be trained on the raw input features to predict our ads engagement to generate features for our logistic regression model. As discussed in the additives trees section, neural network-based models are also really good at capturing non-linear, complex relationships between features.

The following examples show a network trained with two hidden layers to predict our engagement rate. Once trained, we can use the activation from the last hidden layer to feed in as input feature vector to our logistic regression model.

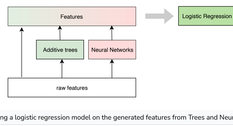


Features from neural network

So, let's combine the above two ideas together:

- We train additive trees and neural network to predict non-linear complex relationships among our features. We then use these models to generate features.
- We use raw features and features generated in the previous step to train a logic regression model.

The above two steps together solve both of our problems to capture complex non-linear relationships and also enable us to use online learning to refresh the model frequently for ads that are fast-changing and dynamic in nature.



Training a logistic regression model on the generated features from Trees and Neural Network

This concludes our course on machine learning system design! Hopefully, you are now confident in your ability to answer machine learning design interview questions in an effective and systematic way.

Back

Mark As Completed

Ad Selection