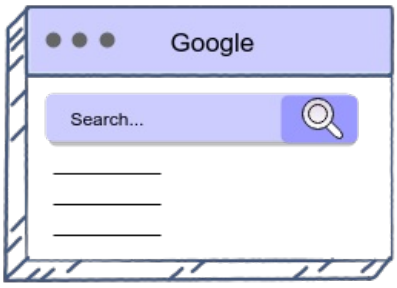
Problem Statement

Let's look at a problem statement that asks you to design a search relevance system for a search engine.

We'll cover the following

Problem statement

The interviewer has asked you to design a search relevance system for a search engine.



Design a search relevance system to display results on a search engine result page

Clarifying questions

Let’s clarify the problem statement by specifying three aspects: scope, scale, and personalization.

Problem scope

The interviewer’s question is really broad. Your best bet is to avoid ambiguities and ask the interviewer as many questions as you can. This will narrow down your problem space as you are thinking out loud for the best solution.

So, your first question for the interviewer would be something like the following:

> *Is it a general search engine like Google or Bing or a specialized search engine like Amazon products search?*

This scoping down of the problem is critical as you dive into finding the solutions. For the sake of this chapter, we will assume that you are working towards finding relevant results using a general search engine like Google search or Bing search, but the techniques and discussion apply to all search engines.

Finally, the problem statement can be precisely described as:

> *Build a generic search engine that returns relevant results for queries like "Richard Nixon", "Programming*

languages" etc.

This will require you to build a machine learning system that provides the most relevant results for a search query by ranking them in order of relevance. Therefore, you will be working on the *search ranking problem*.

> The approach to handle a specialized search engine will be fairly similar.

Scale

Once you know that you are building a general search engine, it's also critical to determine the scale of the system. A couple of important questions are:

- *How many websites exist that you want to enable through this search engine?*
- *How many requests per second do you anticipate to handle?*

We will assume that you have billions of documents to search from, and the search engine is getting around 10K *queries per second* (QPS).

Understanding this scale is important to architect our relevance system. For example, later in the chapter, we will go over the funnel-based approach where you will continue to increase model complexity and reduce document set, as you go down the funnel for this large scale search system.

Personalization

Another important question that you want to address is whether the *searcher is a logged-in user or not*. This will define the level of personalization that you can incorporate to improve the relevance of our results. You will assume that the user is logged in and you have access to their profile as well as their historical search data.

← Back
Model Debugging and Testing

Next →
Metrics

Mark as Completed

Report an Issue
Ask a Question
(https://discuss.educative.io/tag/problem-statement__search-ranking__grokking-the-machine-learning-interview)

Metrics

Let's explore some metrics that will help you define the “success” scenarios for a search session.

We'll cover the following

Choosing a metric for a machine learning model is of paramount importance. Machine learning models learn directly from the data, and no human intuition is encoded into the model. Hence, selecting the wrong metric results in the model becoming optimized for a completely wrong criterion.

There are two types of metrics to evaluate the success of a search query:

1. Online metrics
2. Offline metrics

We refer to metrics that are computed as part of user interaction in a live system as Online metrics. Meanwhile, offline metrics use offline data to measure the quality of your search engine and don’t rely on getting direct feedback from the users of the system.

Online metrics

In an online setting, you can base the success of a search session on *user actions*. On a *per-query level*, you can define success as the user action of *clicking on a result*.

A simple click-based metric is *click-through rate*.

Click-through rate

The click-through rate measures the ratio of clicks to impressions.

$$\text{Click through rate} = \frac{Number of clicks}{Number of impressions}$$

In the above definition, an *impression* means a view. For example, when a search engine result page loads and the user has seen the result, you will consider that as an impression. A click on that result is your success.

Successful session rate

One problem with the *click-through rate* could be that unsuccessful clicks will also be counted towards search success. For example, this might include short clicks where the searcher only looked at the resultant document and clicked back immediately. You could solve this issue by filtering your data to only successful clicks, i.e., to only consider clicks that have a long dwell time.

> *Dwell time* is the length of time a searcher spends viewing a webpage after they've clicked a link on a search engine result page (**SERP**).

Therefore, successful sessions can be defined as the ones that have a click with a ten-second or longer dwell time.

> Session success rate = $\frac{no.\ of\ successful\ sessions}{no.\ of\ total\ sessions}$

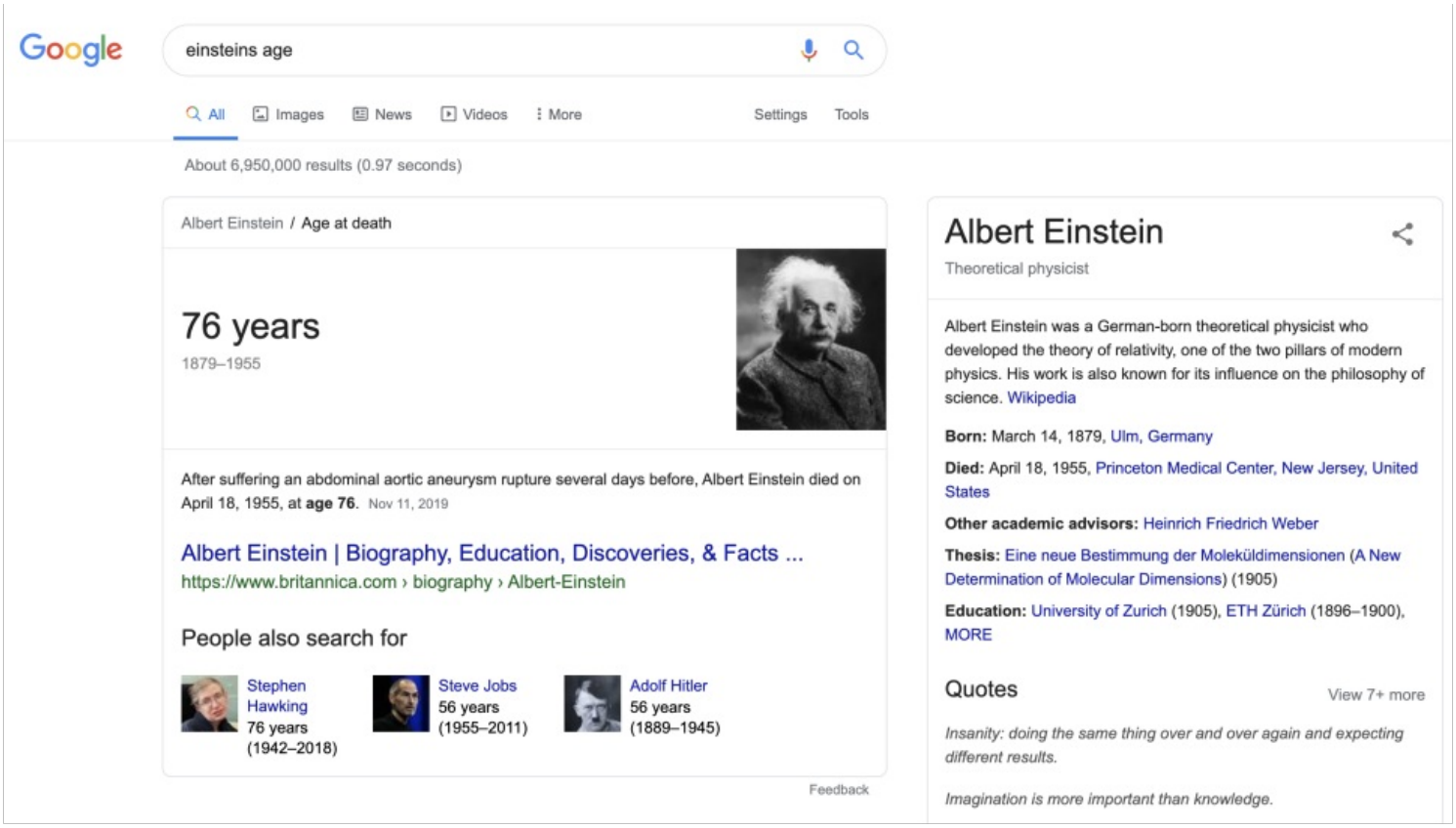
A session can also be successful without a click as explained next.

Caveat

Another aspect to consider is *zero-click searches*.

> **Zero-click searches**: A SERP may answer the searcher's query right at the top such that the searcher doesn't need any further clicks to complete the search.

For example, a searcher queries "einstein's age", and the SERP shows an excerpt from a website in response, as shown below:



The searcher has found what they were looking for without a single click!. The click-through rate would not work in this case (but your definition of a successful session should definitely include it). We can fix this using a simple technique shown below.

Time to success

Until now, we have been considering a single query-based search session. However, it may *span over several queries*. For example, the searcher initially queries: “italian food”. They find that the results are not what they are looking for and make a more specific query: “italian restaurants”. Also, at times, the searcher might have to go over multiple results to find the one that they are looking for.

Ideally, you want the searcher to go to the result that answers their question in the minimal number of queries and as high on the results page as possible. So, *time to success* is an important metric to track and measure search engine success.

> **Note:** For scenarios like this, a *low number of queries per session* means that your system was good at guessing what the searcher actually wanted despite their poorly worded query. So, in this case, we should consider a *low number of queries per session* in your definition of a successful search session.

Offline metrics

The offline methods to measure a successful search session makes use of trained human raters. They are asked to rate the relevance of the query results objectively, keeping in view well-defined guidelines. These ratings are then aggregated across a query sample to serve as the *ground truth*.

> Ground truth refers to the actual output that is desired of the system. In this case, it is the ranking or rating information provided by the human raters.

Let's see *normalized discounted cumulative gain* (NDCG) in detail as it's a critical evaluation metric for any ranking problem.

NDCG

You will be looking at NDCG as a common measure of the quality of search ranking results.

NDCG is an improvement on *cumulative gain* (CG).

> $CG_p = \sum_{i=1}^{p} rel_i$
>
> *where* rel = *relevance rating of a document,* i = *position of document, and* p = *the position up to which the documents are ranked*

Example

Let's look at this concept using an example.

A search engine answers a query with the documents D_1 to D_4. It ranks them in the following order of decreasing relevance, i.e., D_1 is the highest-ranked and D_4 is the lowest-ranked:

Ranking by search engine: D_1, D_2, D_3, D_4

A human rater is shown this result and asked to judge the relevance of each document for the query on a scale of $0 - 3$ (3 indicating highly relevant). They rank the documents as follows:

Documents	D_1	D_2	D_3	D_4
Ranking/Rating by human rater	3	2	3	0

The *cumulative gain* for the search engine's result ranking is computed by simply adding each document's relevance rating provided by the human rater.

$CG_4 = \sum_{i=1}^{4} rel_i = 3 + 2 + 3 + 0 = 8$

In contrast to cumulative gain, discounted cumulative gain (DCG) allows us to *penalize the search engine's ranking if highly relevant documents (as per ground truth) appear lower in the result list*.

DCG discounts are based on the position of the document in human-rated data. The intuition behind it is that the search engine will not be of much use if it doesn't show the most relevant documents at the top of search result pages. For example, showing starbucks.com (http://starbucks.com/) at a lower position for the query *"Starbucks"*, would not be very useful.

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

DCG for the search engine's ranking is calculated as follows:

$$DCG_4 = \sum_{i=1}^{4} \frac{rel_i}{\log_2(i+1)} = 3 + 1.262 + 1.5 + 0 = 5.762$$

In the above calculation, you can observe that the denominator penalizes the search engine's ranking of D_3 for appearing later in the list. It was a more relevant document relative to D_2 and should have appeared earlier in the search engine's ranking.

However, DCG can't be used to compare the search engine's performance across different queries on an absolute scale. The is because the length of the result list varies from query to query. So, the DCG for a query with a longer result list may be higher due to its length instead of its quality. To remedy this, you need to move towards NDCG.

NDCG normalizes the DCG in the 0 to 1 score range by dividing the DCG by the max DCG or the IDCG (ideal discounted cumulative gain) of the query. IDCG is the DCG of an ideal ordering of the search engine's result list (you'll see more on IDCG later).

NDCG is computed as follows:

$$NDCG_p = \frac{DCG_p}{IDCG_p} \text{ where } IDCG \text{ is ideal discounted cumulative gain}$$

In order to compute IDCG, you find an ideal ordering of the search engine's result list. This is done by rearranging the search engine's results based on the ranking provided by the human raters, as shown below.

Search engine's document ranking (D_1 is highest ranked)	D_1	D_2	D_3	D_4
Corresponding rating by human raters (Rank 3 is highest)	3	2	3	0
Ideal ordering by rearranging documents based on row 2	D_1	D_3	D_2	D_4

Finding the ideal ordering of search engine's result based on ground truth

Now let's calculate the DCG of the ideal ordering (also known as the IDCG) as shown below:

$$IDCG_4 = 5.898$$

Now, let's finally compute the NDCG:

$$\text{NDCG} = \frac{DCG}{IDCG} = \frac{5.762}{5.898} = 0.976$$

An NDCG value near one indicates good performance by the search engine. Whereas, a value near 0, indicates poor performance.

To compute NDCG for the overall query set with N queries, we take the mean of the respective NDCGs of all the N queries, and that's the overall relevance as per human ratings of the ranking system.

$$\text{NDCG} = \frac{\sum_{i=1}^{N} NDCG_i}{N}$$

Caveat

NDCG does not penalize irrelevant search results. In our case, it didn't penalize D_4, which had zero relevance according to the human rater.

Another result set may not include D_4, but it would still have the same NDCG score. As a remedy, the human rater could assign a negative relevance score to that document.

← Back

Problem Statement

Next →

Architectural Components

Mark as Completed

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/metrics__search-ranking__grokking-the-machine-learning-interview)

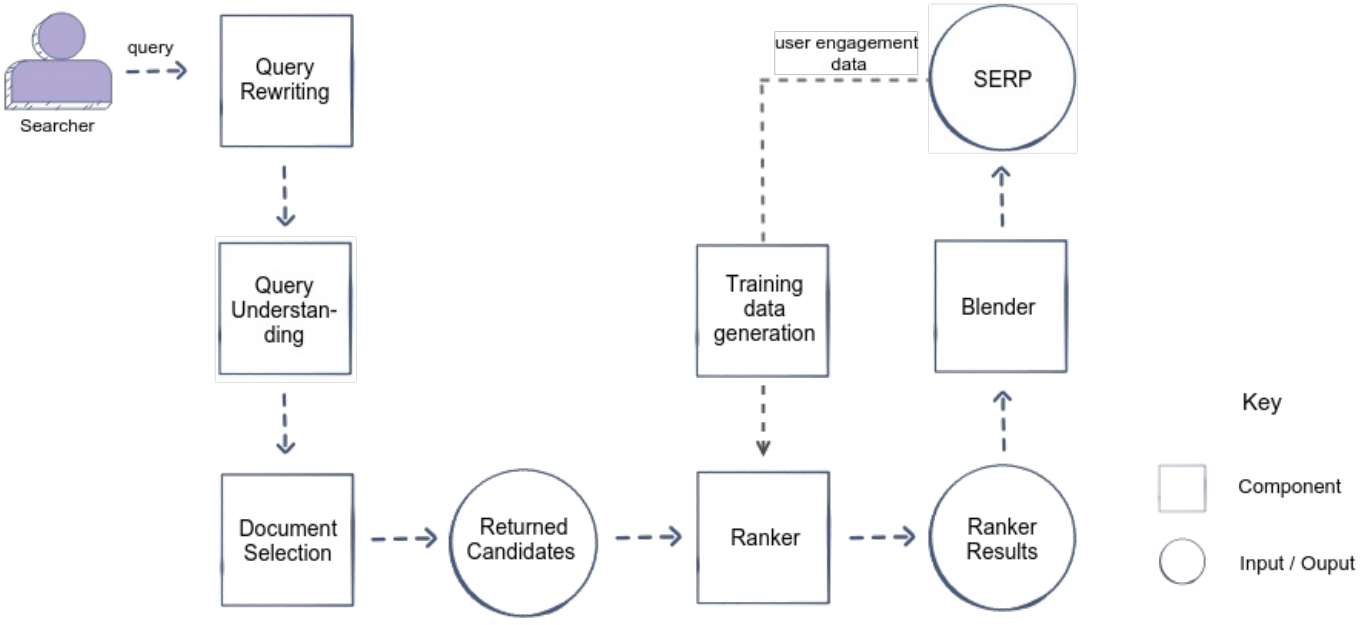
Architectural Components

Let's look at the architectural components of the search ranking system and their role in answering the searcher's query.

We'll cover the following

Architecture

Let’s have a look at the architectural components that are integral in creating the search engine.



Architectural components for the search engine

Let’s briefly look at each component’s role in answering the searcher’s query.

Query rewriting

Queries are often poorly worded and far from describing the searcher's actual information needs. Hence, we use *query rewriting* to increase recall, i.e., to retrieve a larger set of relevant results. Query rewriting has multiple components which are mentioned below.

Spell checker

Spell checking queries is an integral part of the search experience and is assumed to be a necessary feature of modern search. Spell checking allows you to fix basic spelling mistakes like "itlian restaurat" to "italian restaurant".

Query expansion

Query expansion improves search result retrieval by adding terms to the user's query. Essentially, these additional terms minimize the mismatch between the searcher's query and available documents.

Hence, after correcting the spelling mistakes, we would want to expand terms, e.g., for the query "italian restaurant", we should expand "restaurant" to food or recipe to look at all potential candidates (i.e., web pages) for this query.

> The reverse, i.e., *query relaxation*, serves the same purpose. For example, a search for "good italian restaurant" can be relaxed to "italian restaurant".

Query understanding

This stage includes figuring out the main intent behind the query, e.g., the query "gas stations" most likely has a local intent (an interest in nearby places) and the query "earthquake" may have a newsy intent. Later on, this intent will help in selecting and ranking the best documents for the query. You won't need more details about this for our current scope.

> Newsy intent means the searcher is looking for recent information

Document selection

The web has billions of documents. Therefore, our first step in document selection is to find a fairly large set of documents that seems relevant to the searcher's query. For example, some common queries like *"sports"*, can match millions of web pages. Document selection's role will be to reduce this set from those millions of documents to a smaller subset of the most relevant documents.

Document selection is more focused on recall. It uses a simpler technique to sift through billions of documents on the web and retrieve documents that have the potential of being relevant.

Ranking these selected documents in the right order isn't important at this point. We let the *ranking component* worry about finding out "exactly" how relevant (precision) each selected document is and in what order they should be displayed on the SERP.

> Since the *ranking component* receives only the documents that have gone through the "initial screening" its workload is greatly reduced. This allows us to use more complex ML modeling options (that have great precision) for the ranking component, without affecting the performance and capacity requirements of the system.

Ranker

The ranker will actively utilize machine learning to find the best order of documents (this is also called *learning to rank*).

If the number of documents from the document selection stage is significantly large (more than 10k) and the amount of incoming traffic is also huge (more than 10k QPS or queries per second), you would want to have multiple stages of ranking with varying degrees of complexity and model sizes for the ML models. Multiple stages in ranking can allow you to only utilize complex models at the very last stage where ranking order is most important. This keeps computation cost in check for a large scale search system.

For example, one configuration can be that your document selection returns $100k$ documents, and you pass them through two stages of ranking. In stage one, you can use fast (nanoseconds) linear ML models to rank them. In stage two, you can utilise computationally expensive models (like deep learning models) to find the most optimized order of top 500 documents given by stage one.

> When choosing an algorithm, remember to consider the model execution time. Cost vs benefit tradeoff is always an important consideration in large scale ML systems.

Blender

Blender gives relevant results from various search verticals, like, images, videos, news, local results, and blog posts.

For the "italian restaurant" search, we may get a blend of websites, local results, and images. The fundamental motivation behind blending is to satisfy the searcher and to engage them by making the results more relevant.

Another important aspect to consider is the diversity of results, e.g., you might not want to show all results from the same source(website).

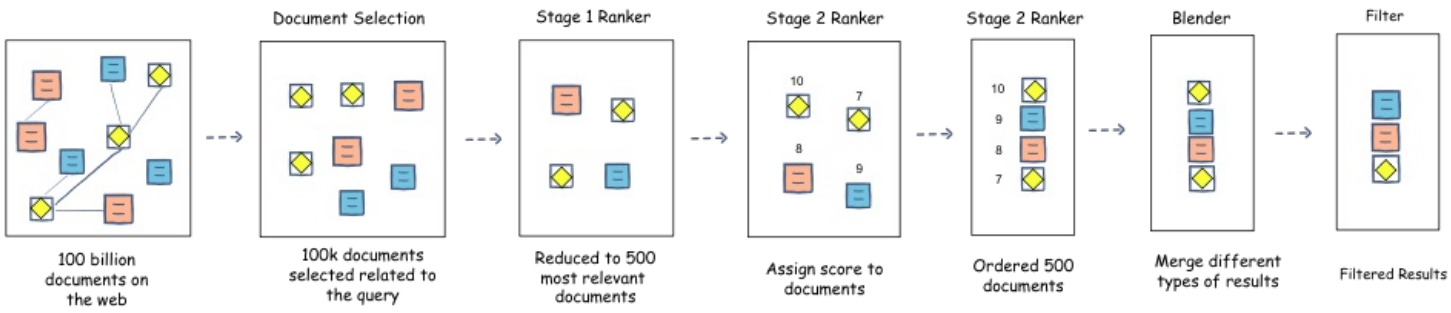
The blender finally outputs a *search engine result page* (SERP) in response to the searcher's query.

Training data generation

This component displays the cyclic manner of using machine learning to make a search engine ranking system. It takes online user engagement data from the SERP displayed in response to queries and generates positive and negative training examples. The training data generated is then fed to the machine learning models trained to rank search engine results.

Layered model approach

Above, we briefly discussed the multi-layered funnel of going from a very large set of documents selected for the query to the topmost relevant one. Let's go over how this configuration might look like for a large scale search system in a bit more detail below.



The layered model approach

These steps translate into a layered model approach that allows you to select the appropriate ML algorithm at each stage, which is thought through the perspective of scalability as well.

The above example configuration assumes that you are first selecting one-hundred thousand documents for the searcher's query from the index, then using two-stage ranking, with the first one reducing from one-hundred thousand to five-hundred documents and the second stage is then ranking these five-hundred documents. The blender can then blend results from different search verticals, and the filter will further screen irrelevant or offensive results to get good user engagement. This is just an example configuration, and it's important to point out that the number of stages and documents ranked at each stage should be selected based on capacity requirements as well as experimentation to see the impact on relevance based on documents scored at each layer.

We will zoom into the different segments of this diagram in the upcoming lessons.

← Back

Metrics

Next →

Document Selection

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/architectural-components__search-ranking__grokking-the-machine-learning-interview)

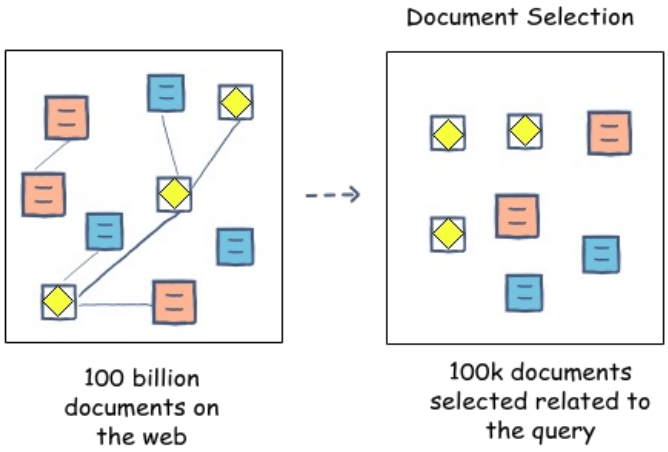
Document Selection

From the one-hundred billion documents on the internet, let's retrieve the top one-hundred thousand that are relevant to the searcher's query.

We'll cover the following

- Document selection process
  - Selection criteria
  - Relevance scoring scheme

Previously you saw the layered model approach. We will be adopting this approach to perform search ranking. Let's zoom in on the first step, i.e., *document selection*, as shown below:



The layered model approach

From the one-hundred billion documents on the internet, we want to retrieve the top one-hundred thousand that are relevant to the searcher's query by using *information retrieval* techniques.

Let's get some terminologies out of the way before we start.

> **Information retrieval** is the science of searching for information in a document. It focuses on comparing the query text with the document text and determining what is a good match.

Documents

Document types are as follows:

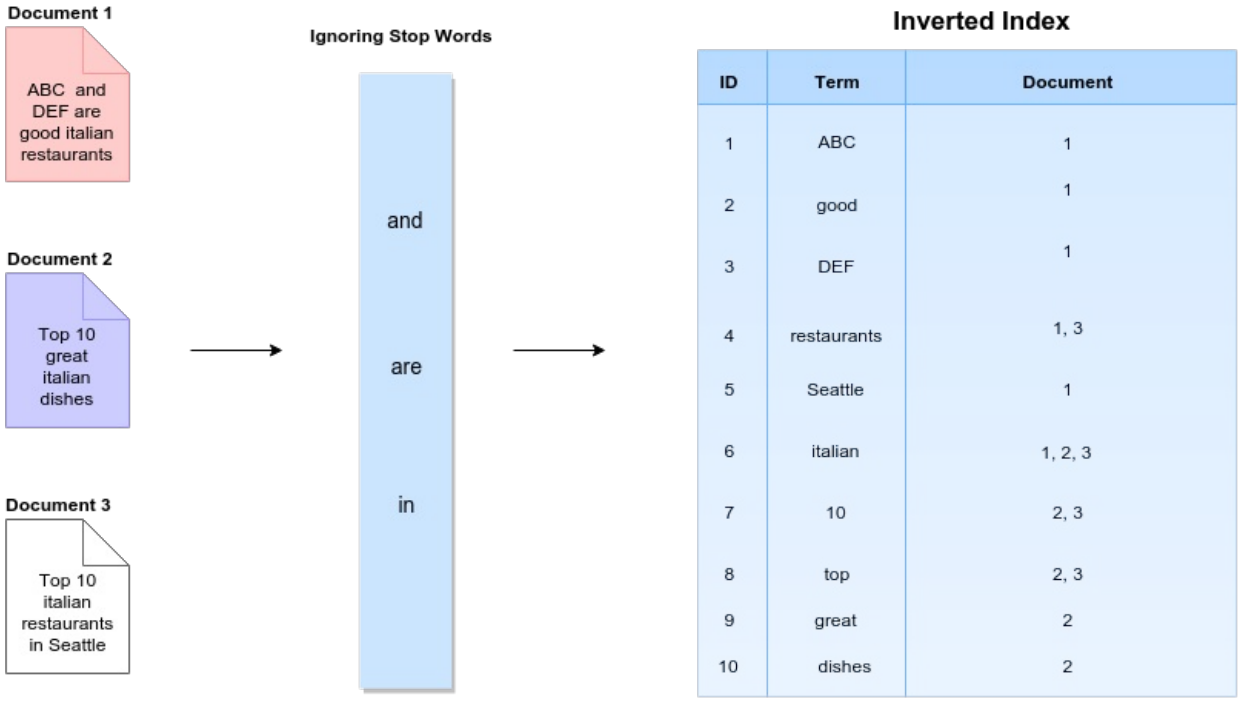
- Web-pages
- Emails
- Books
- News stories
- Scholarly papers

- Text messages
- Word™ documents
- Powerpoint™ presentations
- PDFs
- Patents, etc.

All of the above have a significant amount of textual content.

Inverted Index

> **Inverted index:** an index data structure that stores a mapping from content, such as words or numbers, to its location in a set of documents.

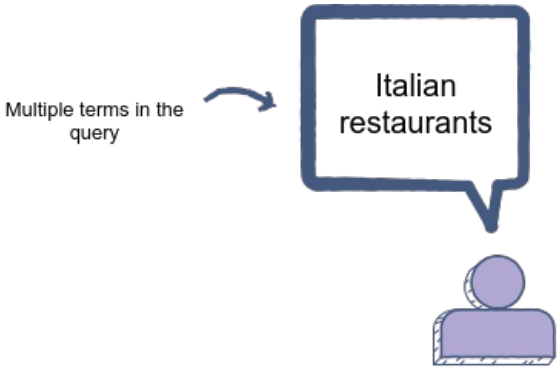


Inverted index: the term "restaurants" occurs in documents 1 and 3

Document selection process

The searcher's query does not match with only a single document. The selection criteria derived from the query may match a lot of documents with a *different degree of relevance*.

Let's begin by looking at the selection criteria and how it identifies matching documents. For instance, the searcher's query is:

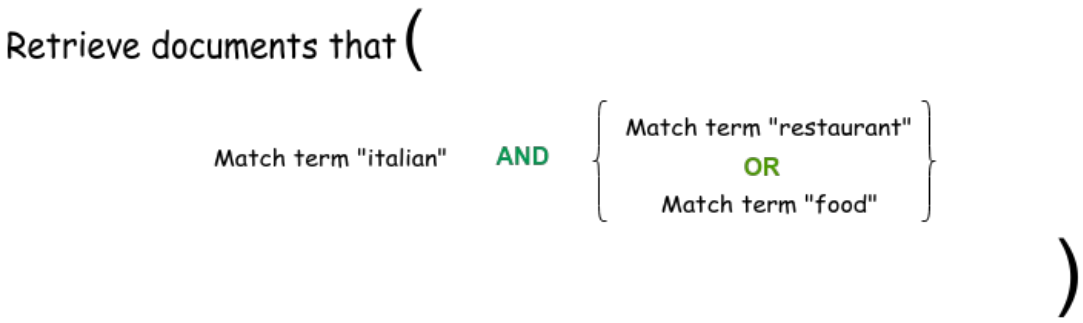


Searcher's query

The *query expansion* component tells us to look for *Italian food*, too.

Selection criteria

Our document selection criteria would then be as follows:



Selection criteria

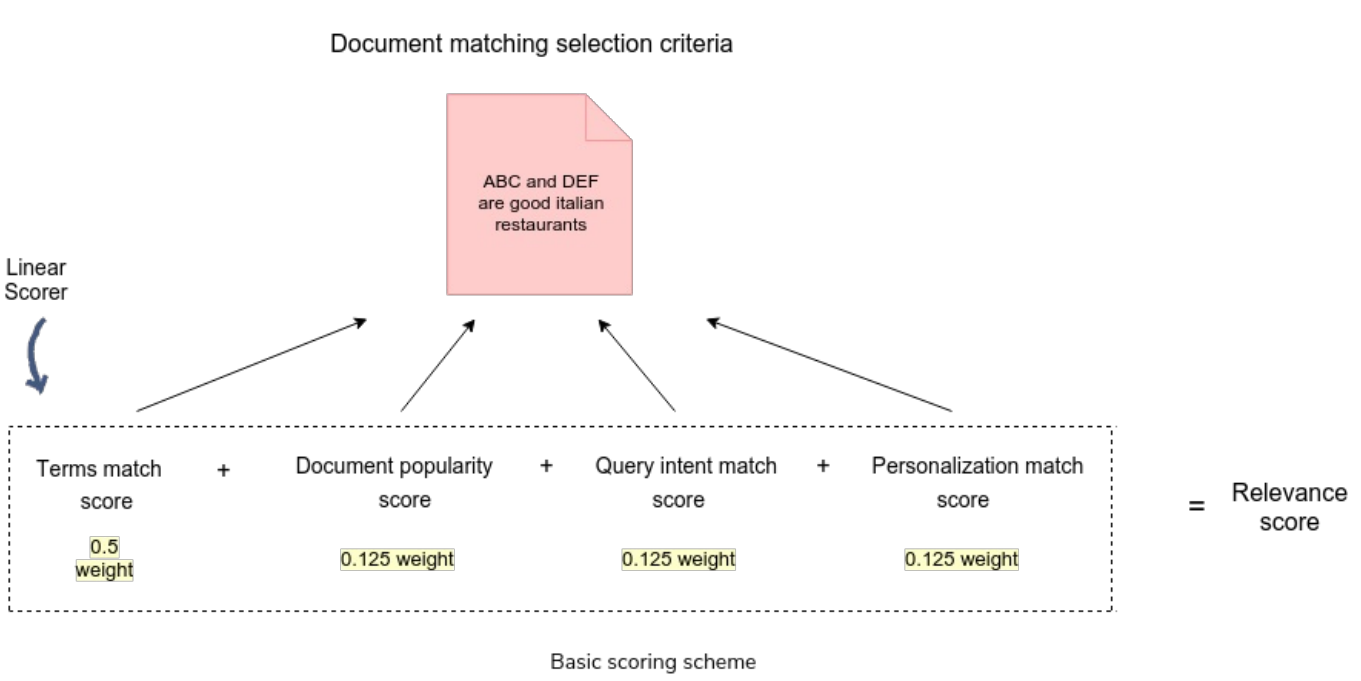
We will go into the *index* and retrieve all the documents based on the above selection criteria. While we would check whether each of the documents matches the selection criteria, we would also be assigning them a relevance score alongside. At the end of the retrieval process, we will have selected relevant documents sorted according to their relevance score. From these documents, we can then forward the top one-hundred thousand documents to the ranker.

Relevance scoring scheme

Let's see how the relevance score is calculated. One basic scoring scheme is to utilize a simple **weighted linear combination** of the factors involved. The weight of each factor depends on its importance in determining the relevance score. Some of these factors are:

1. Terms match
2. Document popularity
3. Query intent match
4. Personalization match

The diagram below shows how the linear scorer will assign a relevance score to a document.



> The weight of each factor in determining the score is selected manually, through the intuition, in the above scorer. Machine learning can also be used to decide these weights.

Let's look at each factor's contribution to the score.

Terms match

The term match score contributes with *0.5 weight* to the document's relevance score.

Our query contains multiple terms. We will use the inverse document frequency or IDF score (https://link.springer.com/referenceworkentry/10.1007%2F978-0-387-39940-9_933) of each term to weigh the match. The *match for important terms in the query weighs higher*. For instance, the term match for "italian" may have more weight in the total contribution of term match to the document's relevance score.

Document popularity

The document's popularity score is stored in the index. Its value will be given 0.125 weight during the document's relevance calculation.

Query intent match

The *query intent* component describes the intent of the query. The document's match with the query's intent will contribute with 0.125 weight to the document's relevance calculation.

For our query, the component may reveal that there is a very strong *local intent*. Hence, a 0.125 weight will be given for the documents retrieved to be local.

Personalization match

This factor contributes with 0.125 weight to the document's relevance score. It scores how well a document meets the searcher's individual requirements based on a lot of aspects. For instance, the searcher's age, gender, interests, and location.

Remember, we can also use ML to assign these scores following a fairly similar process, which we will use in our ranking stage.

Back
Architectural Components

Next
Feature Engineering

Mark as Completed

Report an Issue
Ask a Question
(https://discuss.educative.io/tag/document-selection__search-ranking__grokking-the-machine-learning-interview)

Feature Engineering

Let's engineer meaningful features to train the search ranking model.

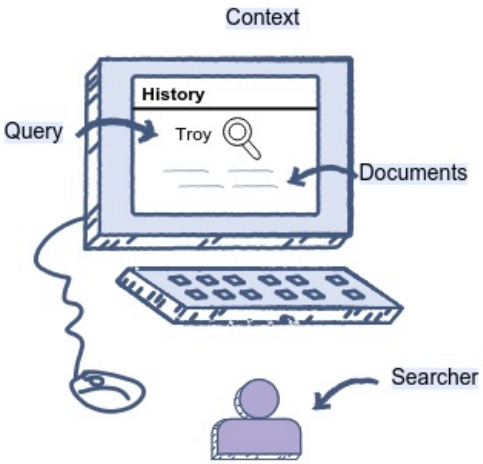
We'll cover the following

An important aspect of the feature generation process is to first think about the main actors that will play a key role in our feature engineering process.

> The terms “features” and “signals” are generally used interchangeably as we will do so in this course.

The **four such actors for search** are:

1. Searcher
2. Query
3. Document
4. Context



Actors

In the above figure, the context for a search query is browser history. However, it is a lot more than just search history. It can also include the searcher’s age, gender, location, and previous queries and the time of day.

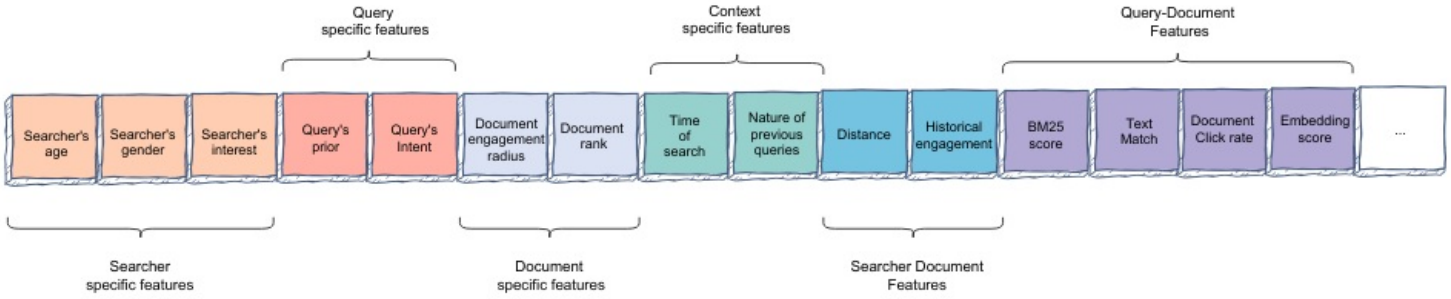
Let's go over the characteristics of these actors and their interactions to generate meaningful features/signals for your machine learning model.

This is essentially the process of feature engineering.

> The knowledge of feature engineering is highly significant from an interview perspective.

Features for ML model

We can generate a lot of features for the search ranking problem based on the actors identified above. A subset of these features is shown below.

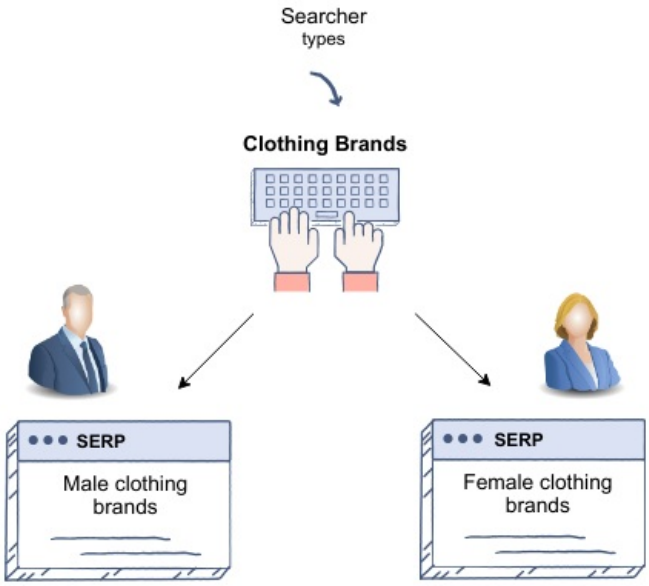


Features in the training data row

Let's discuss these features one by one.

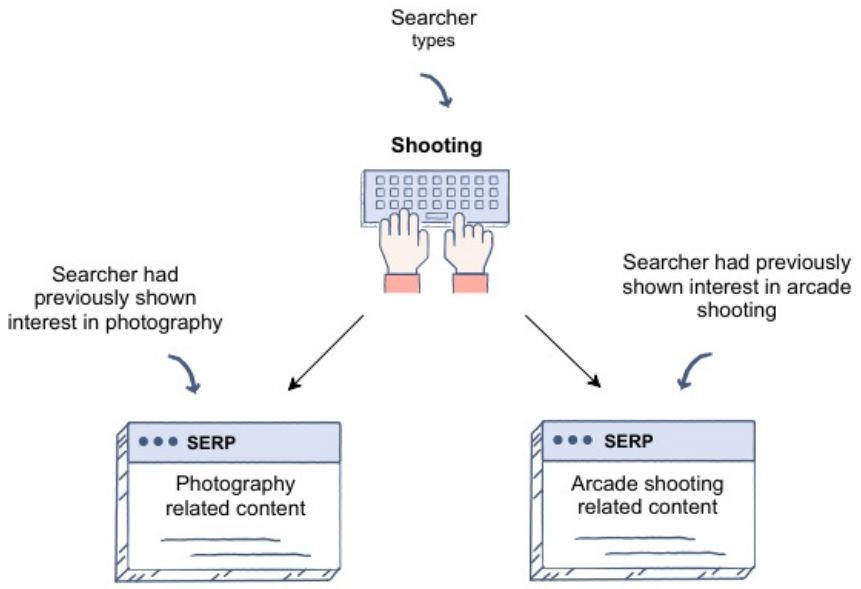
Searcher-specific features

Assuming that the searcher is logged in, you can tailor the results according to their age, gender and interests by using this information as features for your model.



Gender-based results

1 of 2



Interest-based results

2 of 2

Query-specific features

Let's explore some query-related aspects that can also be useful as features.

Query historical engagement

For *relatively popular queries*, historical engagement can be very important. You can use query's *prior* engagement as a feature. For example, let's say the searcher queries "earthquake". We know from historical data that this query results in engagement with "news component", i.e. most people who searched "earthquake", were looking for news regarding a recent earthquake. Therefore, you should consider this factor while ranking the documents for the query.

Query intent

The "query intent" feature enables the model to identify the kind of information a searcher is looking for when they type their query. The model uses this feature to assign a higher rank to the documents that match the query's intent. For instance, if a person queries "pizza places", the intent here is *local*. Therefore, the model will give high rank to the pizza places that are located near the searcher.

> A few examples of query intent are news, local, commerce, etc.

We can get query intent from the query understanding component.

Document-specific features

A lot of features can be engineered with respect to the document's characteristics.

Page rank

The rank of a document can serve as a feature. To estimate the relevance of the document under consideration, we can look at the number and quality of the documents that link to it.

Document engagement radius

The *document engagement radius* can be another important feature. A document on a coffee shop in Seattle would be more relevant to people living within a ten-mile radius of the shop. However, a document on the Eiffel Tower might interest people all around the world. Hence, in case our query has a local intent, we will choose the document with the *local scope of appeal* rather than that with a *global scope of appeal*.

Context-specific features

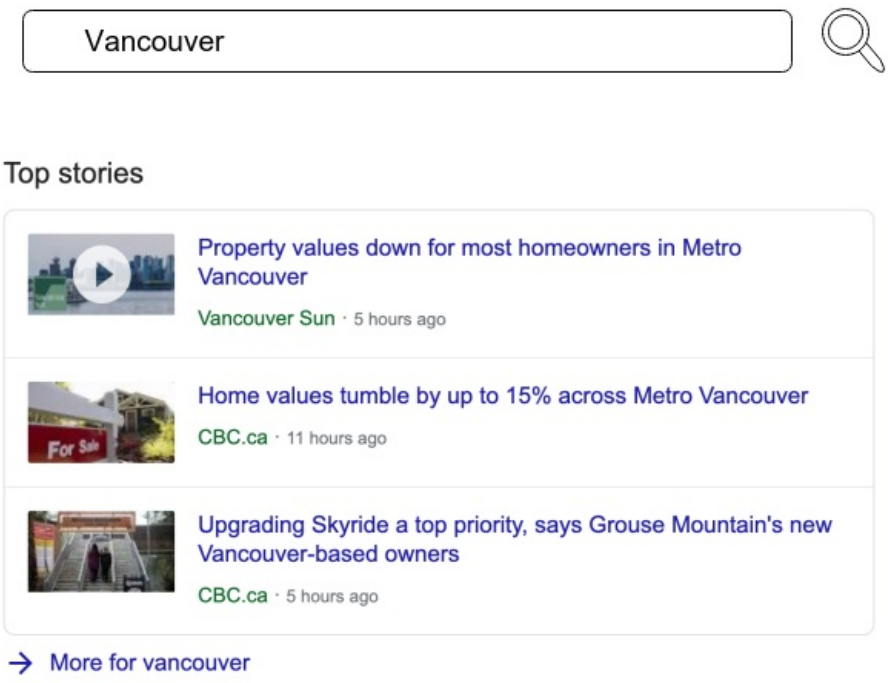
We can extract features from the context of the search.

Time of search

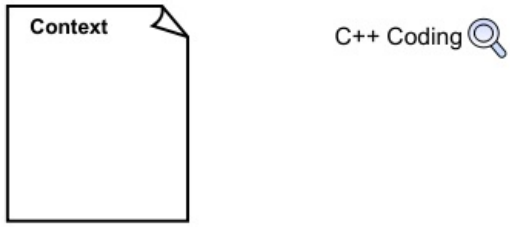
A searcher has queried for restaurants. In this case, a contextual feature can be the *time of the day*. This will allow the model to display restaurants that are open at that hour.

Recent events

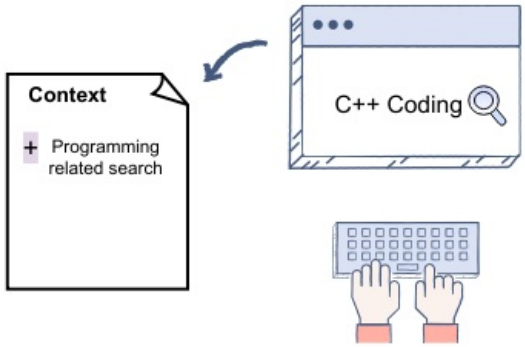
The searcher may appreciate any *recent events related to the query*. For example, upon querying "Vancouver", the results included:



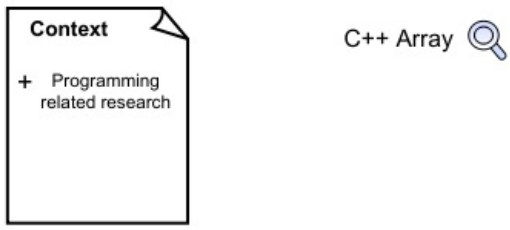
To provide relevant results for a query, looking at the *nature of the previous queries* can also help. Take a look at the example below.



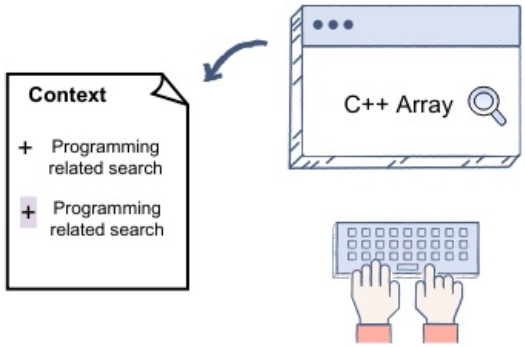
Searcher queries for "C++ Coding"



Search type is stored in context

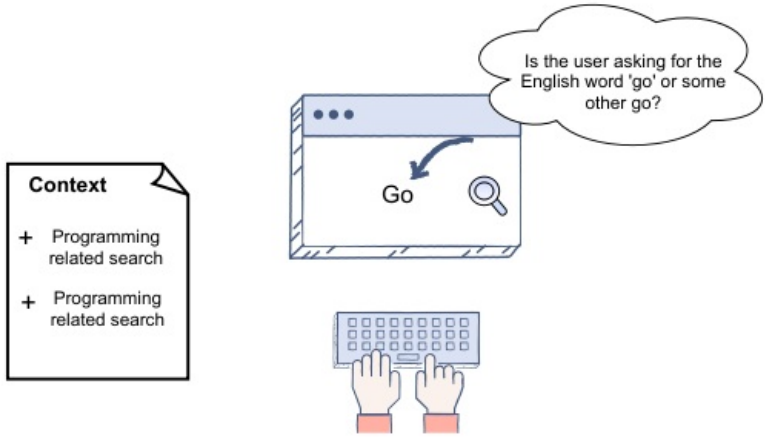


Searcher queries for "C++ Array"



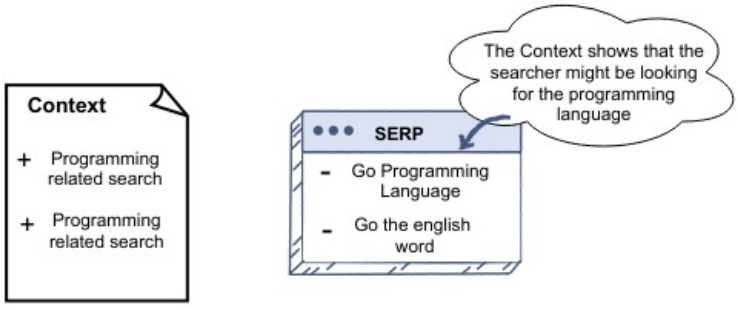
Search type is stored in context

4 of 6



Searcher queries for "GO"

5 of 6



The context suggests that the searcher is likely to be looking for something related to programming

6 of 6

Now, let's look at the actors jointly to create more features:

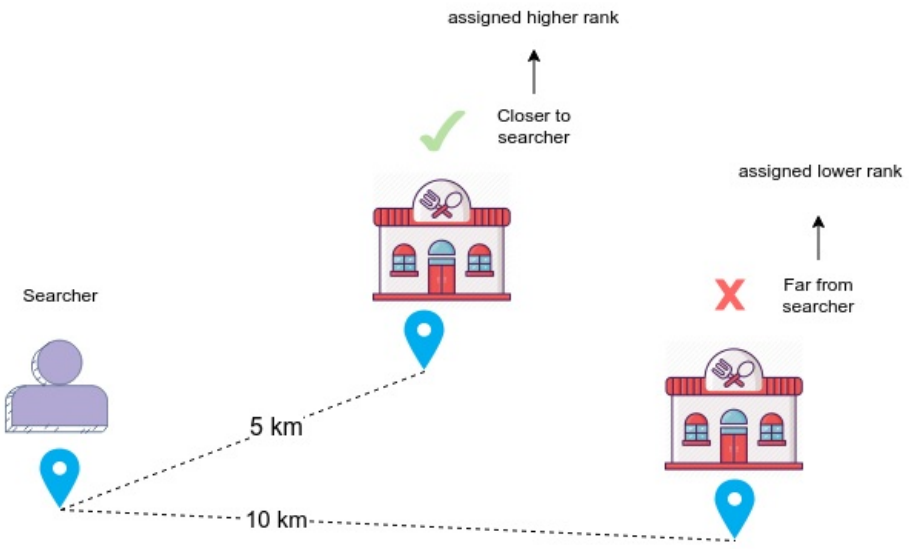
Searcher-document features

We can also extract features by considering both the searcher and the document.

Distance

For queries inquiring about nearby locations, we can use the distance between the searcher and the matching locations as a feature to measure the relevance of the documents.

Consider the case where a person has searched for restaurants in their vicinity. Documents regarding nearby restaurants will be selected for ranking. The ranking model can then rank the documents based on the distance between the coordinates of the searcher and the restaurants in the document.



Feature: the distance between searcher's location and the restaurant's location

Historical engagement

Another interesting feature could be the searcher's historical engagement with the *result type* of the document. For instance, if a person has engaged with video documents more in the past, it indicates that video documents are generally more relevant for that person.

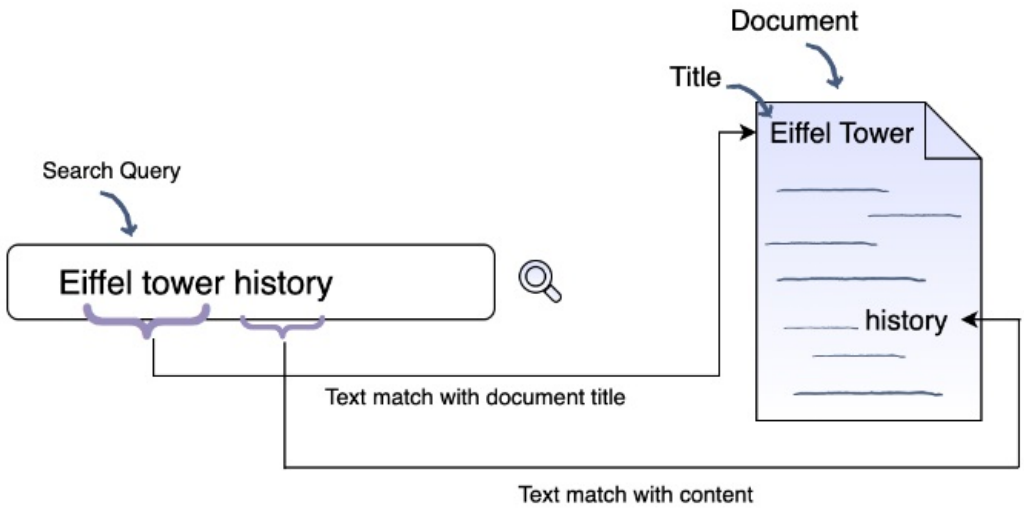
Historical engagement with a particular website or document can also be an important signal as the user might be trying to "re-find" the document.

Query-document features

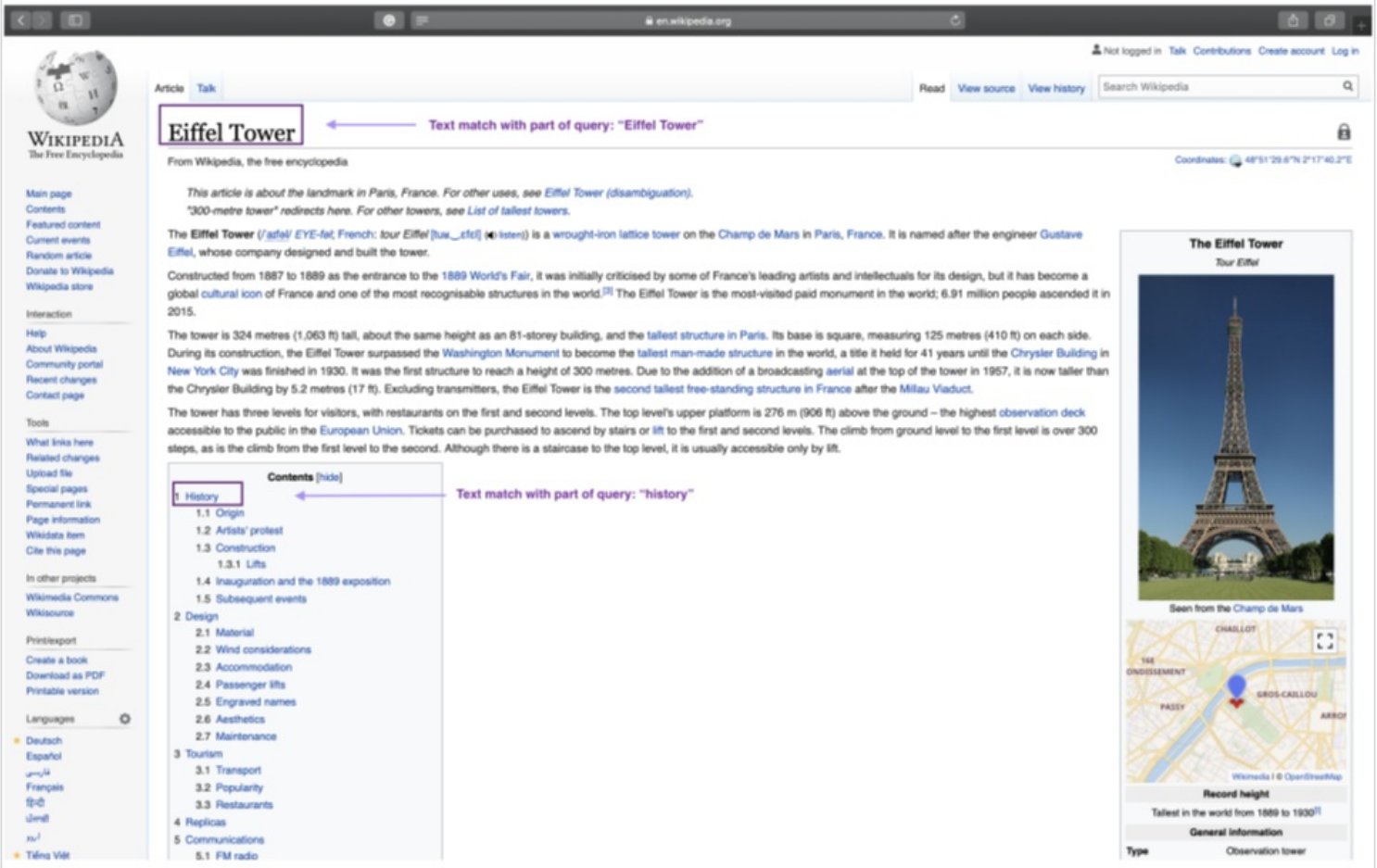
Given the query and the document, we can generate tons of signals.

Text Match

One feature can be the text match. Text match can not only be in the *title* of the document, but it can also be in the *metadata* or *content* of a document. Look at the following example. It contains a text match between the query and document title and another between the query and document content. These text matches can be used as a feature.



Text match



Query text match with a document: wikipedia

Unigram or bigram

We can also look at data for each unigram and bigram for text match between the query and document. For instance, the query: “Seattle tourism guide” will result in three unigrams:

1. Seattle
2. tourism
3. guide

These unigrams may match different parts of the document, e.g., *“Seattle”* may match the document title, while *“tourism”* may match the document’s content. Similarly, we can check the match for the bigram and the full trigram, as well. All of these text matches can result in multiple text-based features used by the model.

The **TF-IDF** match score can also be an important relevance signal for the model. It is a similarity score based on a text match between the query terms and the document. TF (term frequency) incorporates the importance of each term for the document and IDF (inverse document frequency) tells us about how much information a particular term provides.

Query-document historical engagement

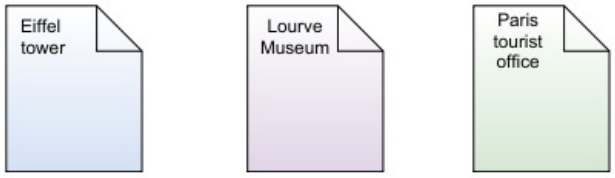
The prior engagement data can be a beneficial feature for determining the best ranking of the search results.

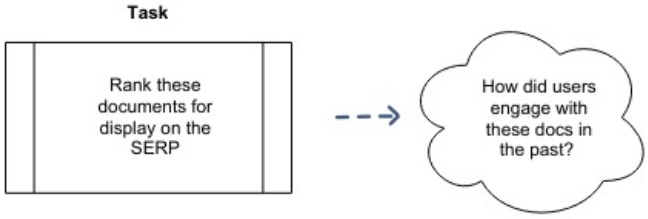
- **Click rate**

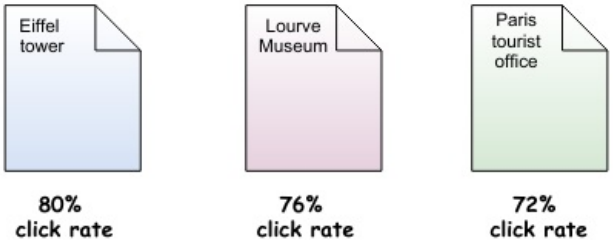
  We want to see users’ historical engagement with documents shown in response to a particular query. The click rates for the documents can help in the ranking process. For example, we might observe across people’s queries on “Paris tourism” that the click rate for the “Eiffel tower website” is the highest. So, the model will develop the understanding that whenever someone queries “Paris tourism”, the document/website on Eiffel tower is the most engaged with. It can then use this information in the ranking of documents.

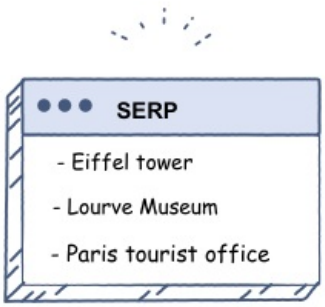
Searcher queries "Paris tourism"

Relevant documents retrieved









Documents ranked according to click rate

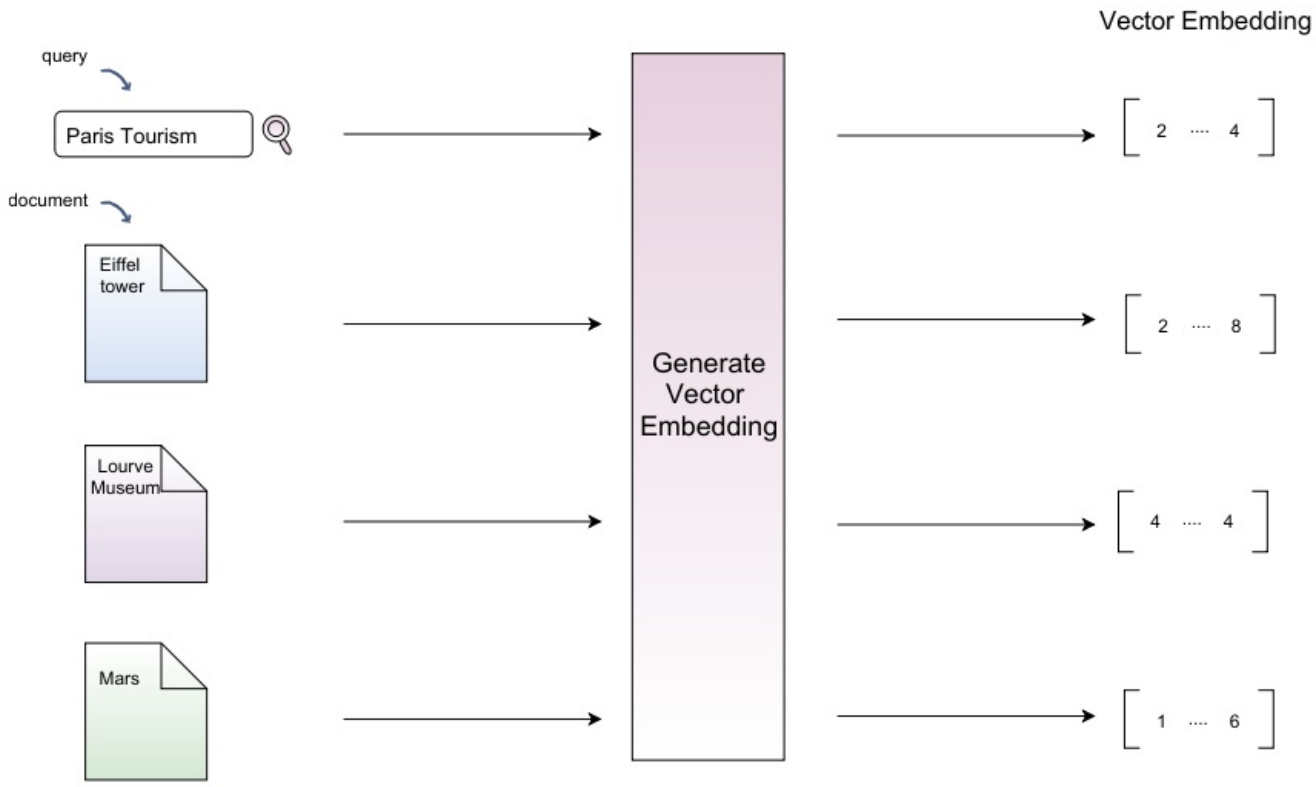
Embeddings

We can use embedding models to represent the query and documents in the form of vectors. These vectors can provide significant insight into the relationship between the query and the document. Let's see how.

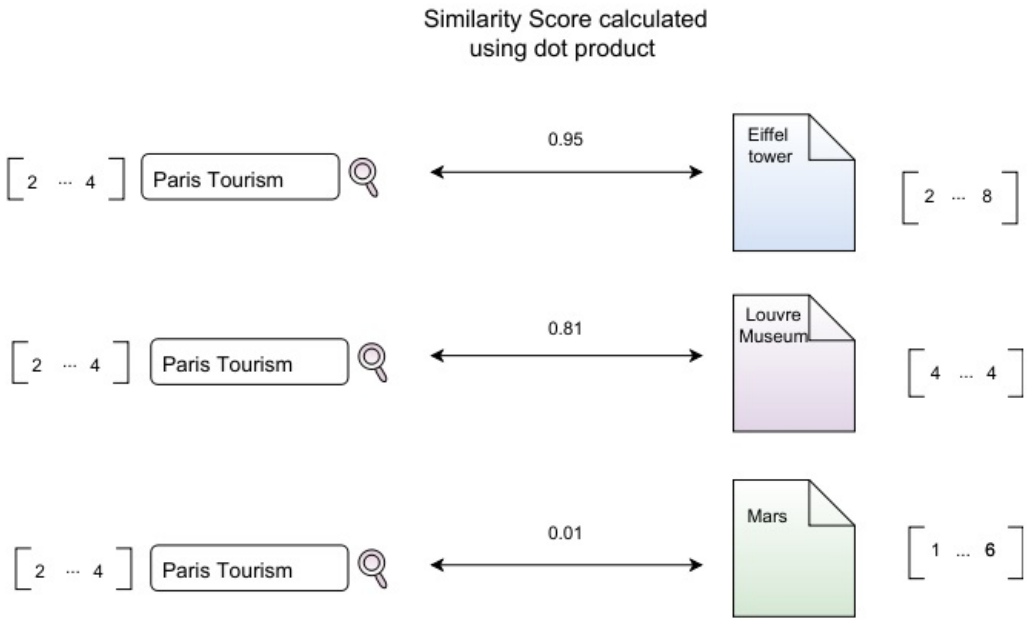
The embedding model generates vectors in such a manner that if a document is on the same topic/concept as the query, its vector is similar to the query's vector. We can use this characteristic to create a feature called "embedding similarity score". The similarity score is calculated between the query vector and each document vector to measure its relevance for the query. The higher the similarity score, the more relevant a document is for a query.

> Please refer to the embedding lesson to go over techniques that can be used to generate embeddings.

For a better understanding, let's refer to the query "Paris tourism", as shown below.

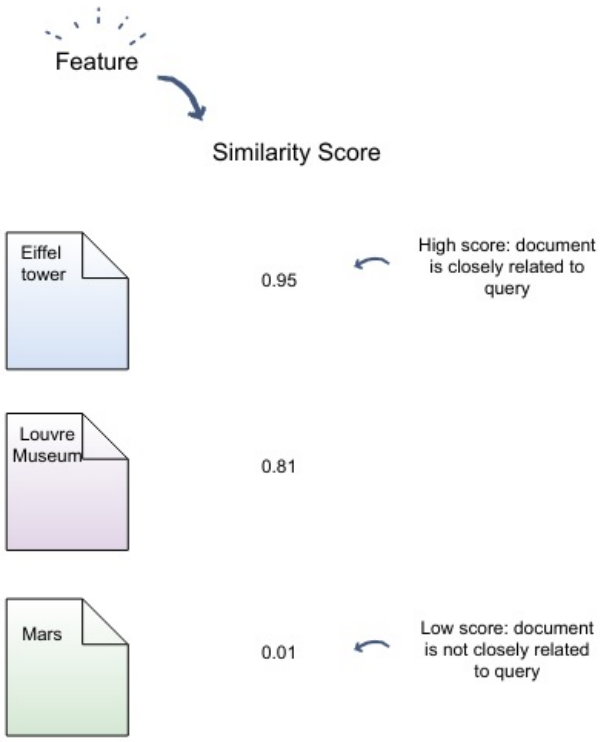


An embedding algorithm generates vectors for the query and documents, respectively



Similarity score calculated between query vector and document vectors

2 of 3



Similarity score used as a feature for the model

3 of 3

Document selection selects three documents in response to the query, namely the Eiffel Tower webpage, Louvre Museum webpage, and Mars (assume this to be a website where the word “Paris” is used) webpage. We use the

embeddings technique to generate vectors for the query and each of the retrieved documents. The similarity score is calculated for the query vector against each document vector. It can be seen that the Eiffel tower document has the highest similarity score. Therefore, it is the most relevant document based on semantic similarity.

← Back
Document Selection

Next →
Training Data Generation

Mark as Completed

Report an Issue Ask a Question

Training Data Generation

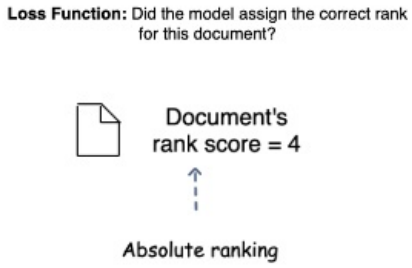
Let's look at methods for generating training data for the search ranking problem.

We'll cover the following

Let's generate training data for the search ranking ML model. Note that the term *training data row* and *training example* will be used interchangeably.

Training data generation for pointwise approach

Pointwise approach: In this approach of model training, the training data consists of relevance scores for each document. The loss function looks at the score of one document at a time as an absolute ranking.



Hence the model is trained to predict the relevance of each document for a query, *individually*. The final ranking is achieved by simply sorting the result list by these document scores.

While adopting the pointwise approach, our ranking model can make use of *classification algorithms* when the score of each document takes a *small, finite number of values*. For instance, if we aim to simply classify a document as relevant or irrelevant, the relevance score will be 0 or 1. This will allow us to *approximate* the ranking problem by a **binary classification problem**.

Now let's generate training data for the binary classification approximation.

Positive and negative training examples

We are essentially predicting user engagement towards a document in response to a query. A *relevant document* is one that successfully engages the searcher.

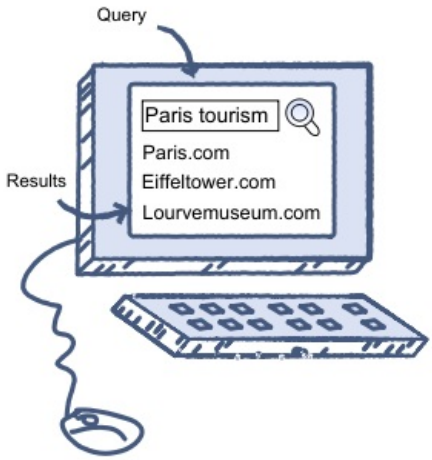
For instance, we have the searcher's query: "Paris tourism", and the following results are displayed on the SERP in response:

1. Paris.com
2. Eiffeltower.com
3. Lourvemusuem.com

We are going to label our data as positive/negative or relevant/irrelevant, keeping in mind the metric successful session rate, as shown in the following illustration.

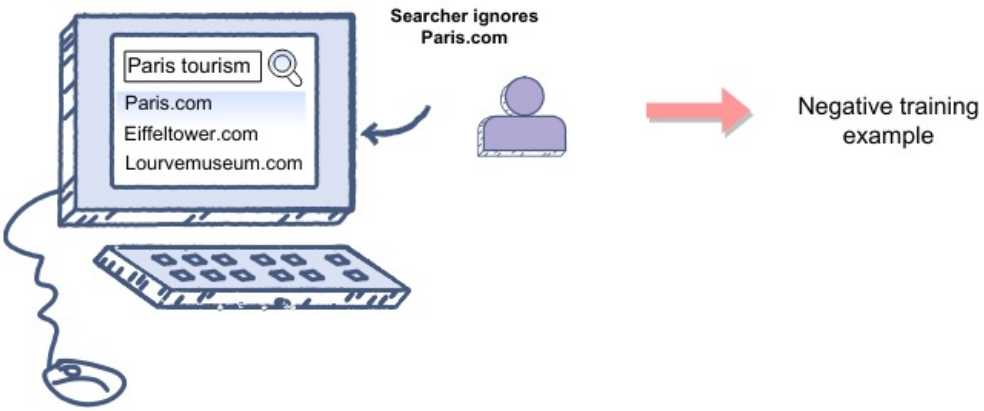
Assumption

Let's assume that the searcher did not engage with Paris.com but engaged with Eiffeltower.com. Upon clicking on Eiffeltower.com, they spent two minutes on the website and then signed up. After signing up, they went back to the SERP and clicked on Lourvemusuem.com and spent twenty seconds there.



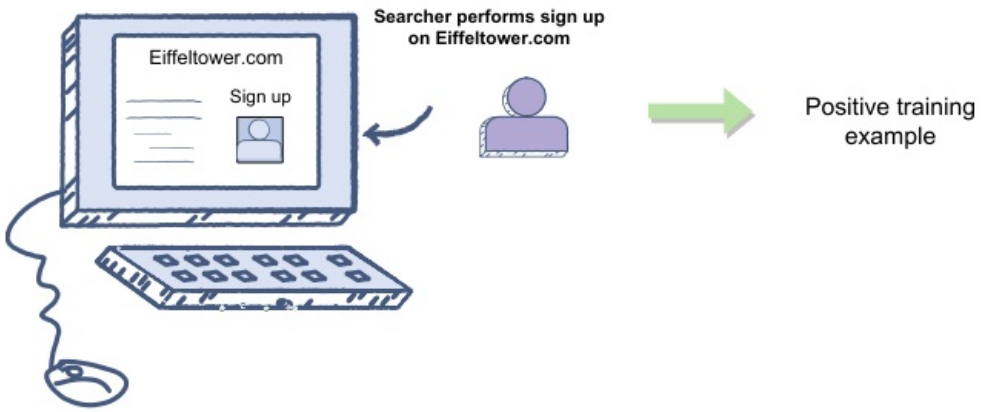
Training data generation through online user engagement

1 of 4



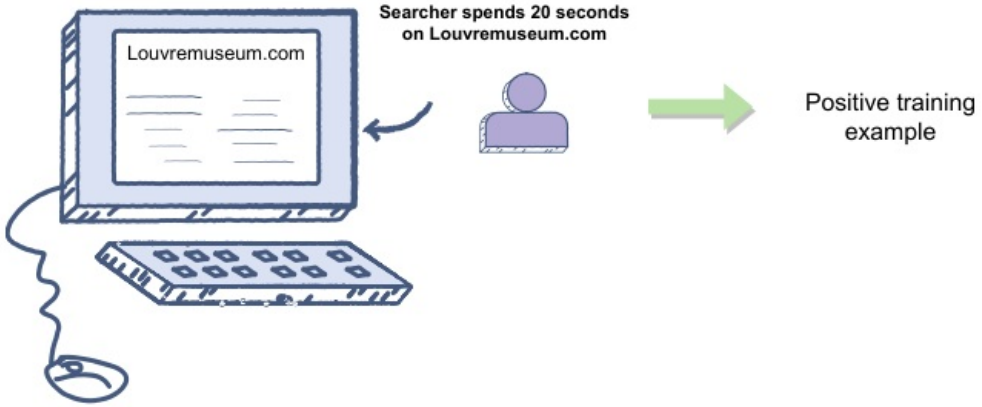
Training data generation through online user engagement

2 of 4



Training data generation through online user engagement

3 of 4



Training data generation through online user engagement

4 of 4

This sequence of events generates three rows of training data. “Paris.com” would be a negative instance because it only had a view with no engagement. The user skipped it and engaged with the other two links, which would become positive examples.

> If we had been predicting the click-through rate only, a positive example would have been one with just a click on the website.

Caveat: Less negative examples

A question may arise that if the user engages with only the first document on the SERP, we may never get enough negative examples to train our model. Such a scenario is pretty common. To remedy it, we use *random negative* examples. For example, all the documents displayed on the 50^{th} page of Google search results can be considered negative examples.

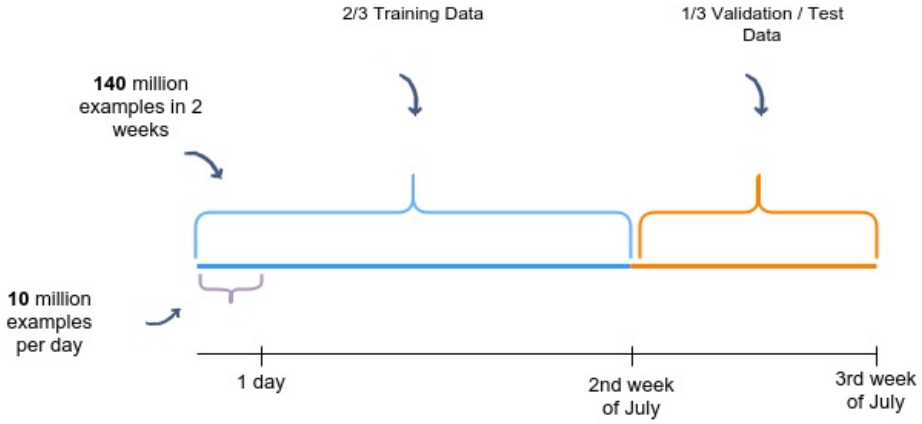
For the query discussed above, three rows of training data were generated. We may receive five million such queries per day. On average, we may generate two rows per query, one positive and one negative. This way, we would be generating ten million training examples per day.

User engagement patterns may differ throughout the week. For instance, the engagement on weekdays may differ from the weekend. Therefore, we will use a week's queries to capture all patterns during training data generation. At this rate, we would end up with around seventy million rows of training data.

Train test split

We may randomly select $\frac{2}{3}^{rd}$, or 66.6%, of the seventy million training data rows and utilize them for training purposes. The remaining of the $\frac{1}{3}^{rd}$, or 33.3%, can be used for the validation and testing of the model.

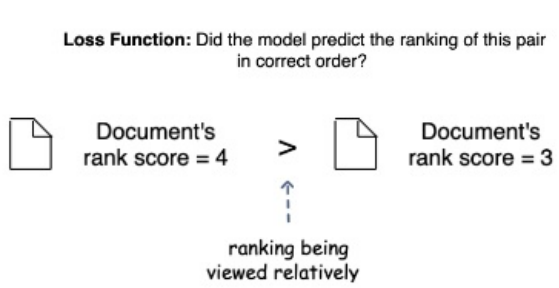
However, an approach this simple, may not be the best idea. Let's take a step back to see why. You are trying to build a model using historical data, that is trying to predict the future. You can better capture the essence of our goal by, say, training the model using data generated from the first and second week of July and data generated in the third week of July for validation and testing purposes.



Splitting data for training, validation, and testing

Training data generation for pairwise approach

> **Pairwise approach:** This approach differs from the pointwise approach during model training. Here, the loss function looks at the *scores of document pairs* as an inequality instead of the score of a single document. This enables the model to learn to rank documents according to their relative order which is closer to the nature of ranking.



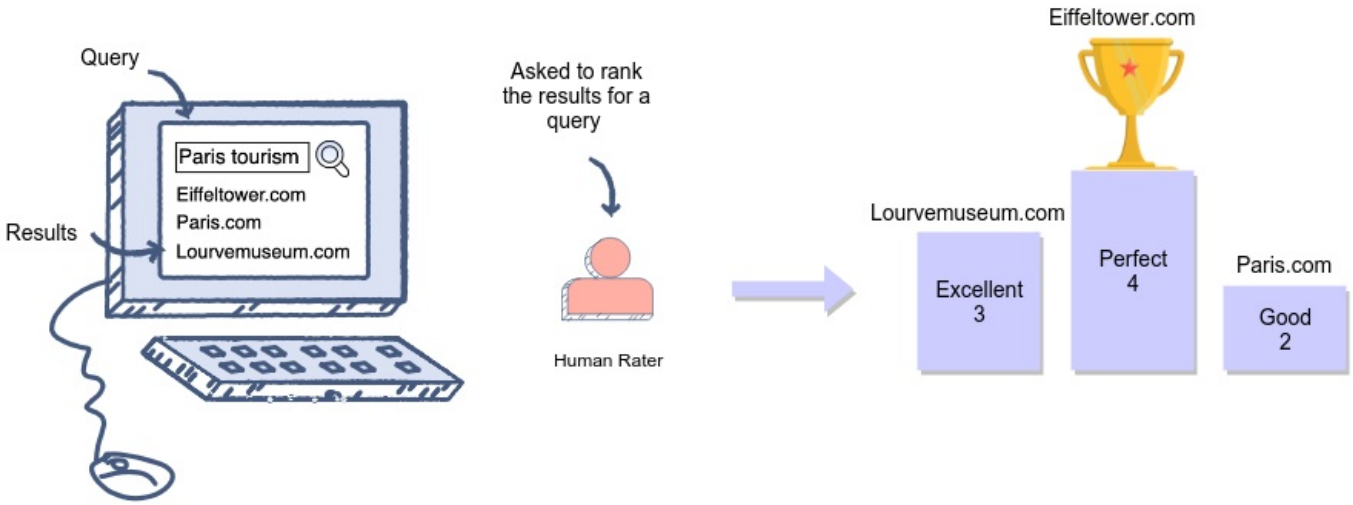
Hence, the model tries to predict the document scores such that the number of inversions in the final ranked results is minimum. Inversions are cases where the pair of results are in the wrong order relative to the ground truth.

We are going to explore two methods of training data generation for the pairwise approach to the ranking problem:

1. The use of human raters
2. The use of online user engagement

Human raters (offline method)

In order to generate training data, human raters can help us in the following manner:



A human rater will be shown the SERP for a query, and they will rank the results

Let's assume that the human rater will be presented with 100, 000 queries, each having ten results. They will then be asked to rate the results for each query. The rater might rate a document as:

Rating	Ranking
perfect	4
excellent	3
good	2

fair	1
bad	-1

As a result, we would have ten million training examples. The model will then learn to rank the results for a query based on these training examples.

User-engagement (online method)

Generating a large amount of training data with the help of human raters can turn out to be very expensive. As an alternative, you can use online user engagement data to generate rankings. The user's interaction with the results on the SERP will translate to ranks based on the type of interaction.

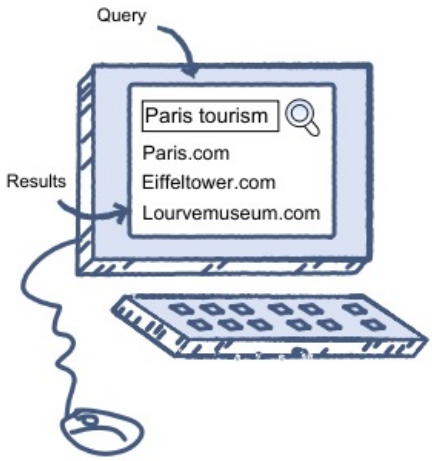
One intuitive idea can be to use online engagement to generate pairwise session data, e.g., one session can have three types of engagements varying from higher degrees of relevant actions to lower.

For instance, for the query: "Paris tourism", we get the following three results on the SERP:

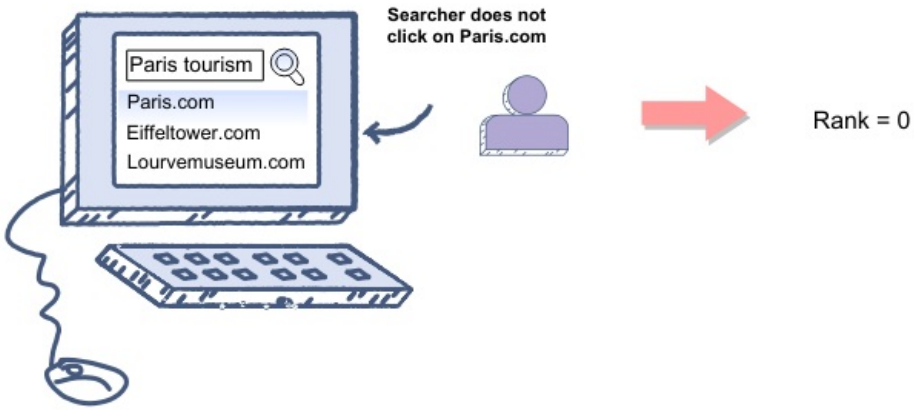
1. Paris.com
2. Eiffeltower.com
3. Lourvemusuem.com

Paris.com only has an impression with no click, which would translate to label 0. The searcher signs up on Eiffeltower.com. It will be rated as perfect and assigned the label 4. Lourvemusuem.com will be rated as excellent and assigned the label 3, as the searcher spends twenty seconds exploring the website.

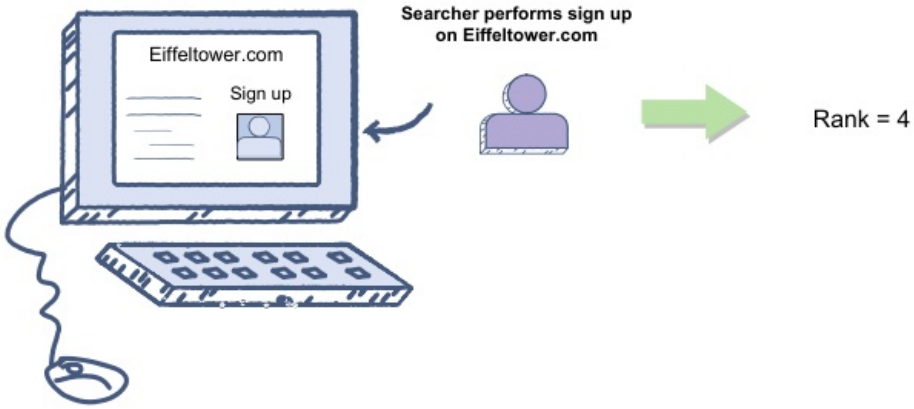
So, this can be another way to generate pairwise data for our learning algorithm using engagement on the search results page.



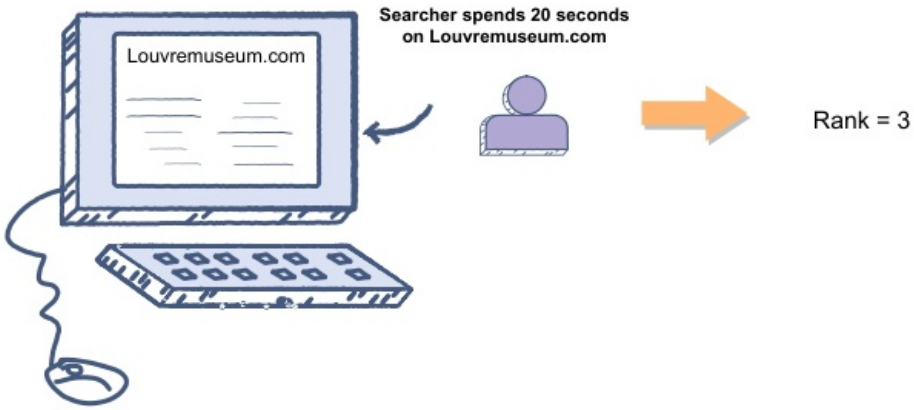
Ranking generation through online user engagement



Ranking generation through online user engagement



Ranking generation through online user engagement



Ranking generation through online user engagement

Back

Feature Engineering

Next

Ranking

Mark as Completed

Report an Issue

Ask a Question

Ranking

Let's see how to design the search ranking model.

We'll cover the following

Objective

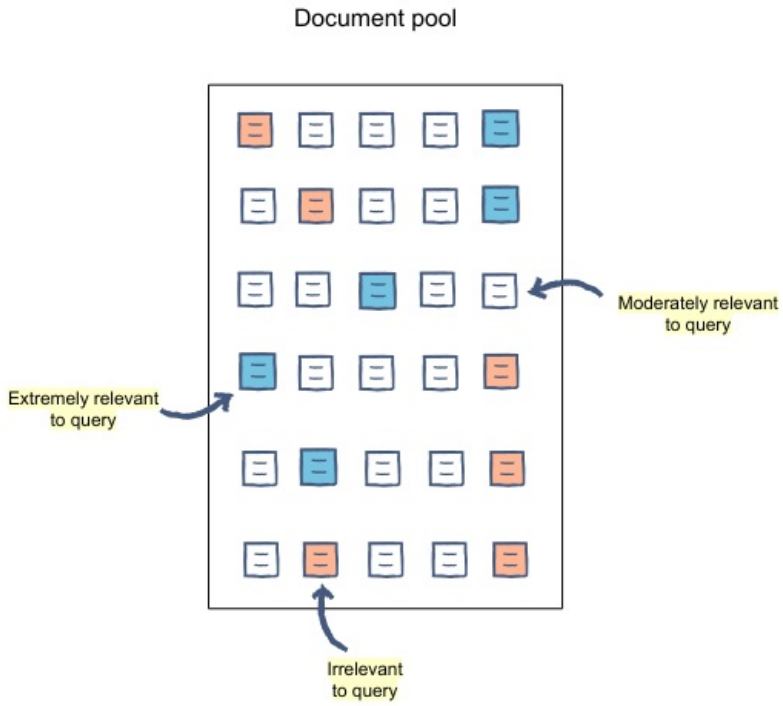
You want to build a model that will display the most relevant results for a searcher's query. Extra emphasis should be applied to the relevance of the top few results since a user generally focuses on them more on the SERP.

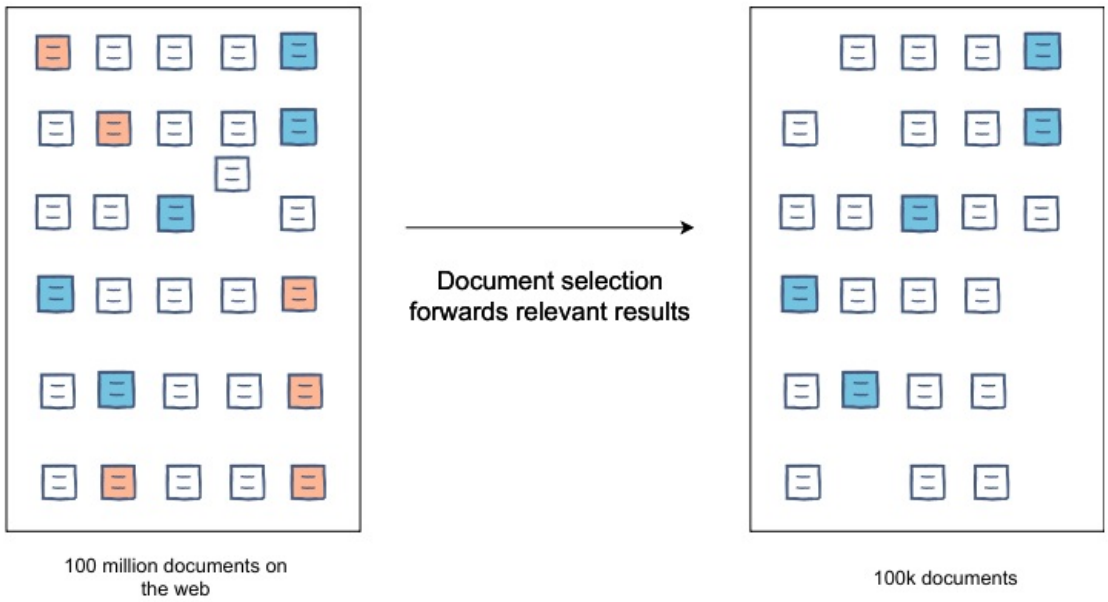
> **Learning to Rank (LTR)**: A class of techniques that applies supervised machine learning (ML) to solve ranking problems. The pointwise and pairwise techniques that we will apply fall under this class.

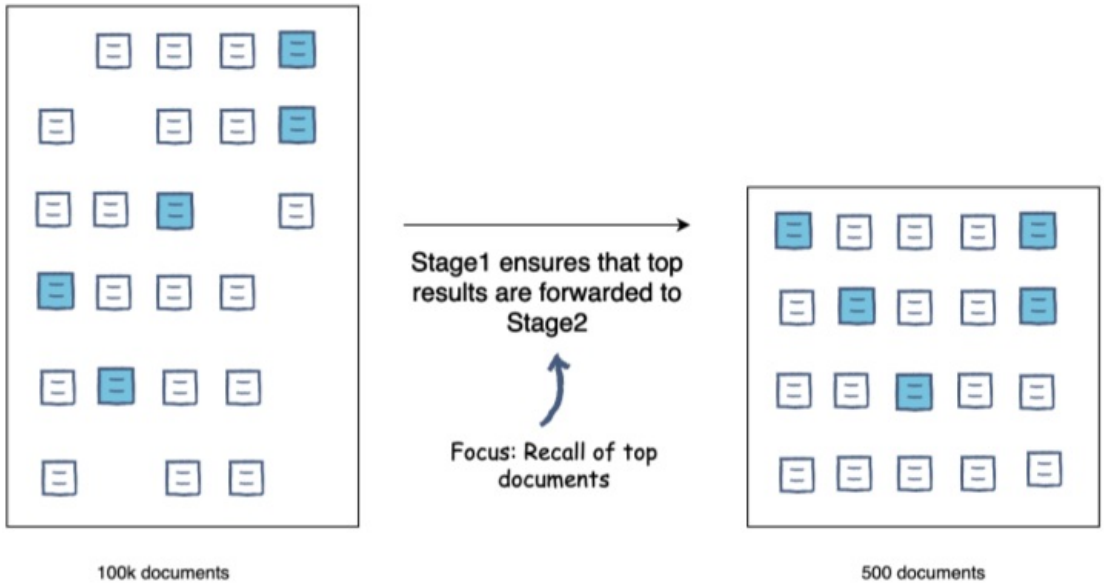
Stage-wise approach

As discussed in the architectural components section, the number of documents matching a single query can be very large. So, for a large scale search engine, it makes sense to adopt a multi-layer funnel approach. The top layer of the funnel looks at a large number of documents and uses simpler and faster algorithms for ranking. The bottom layer ranks a small number of documents with complex machine-learned models.

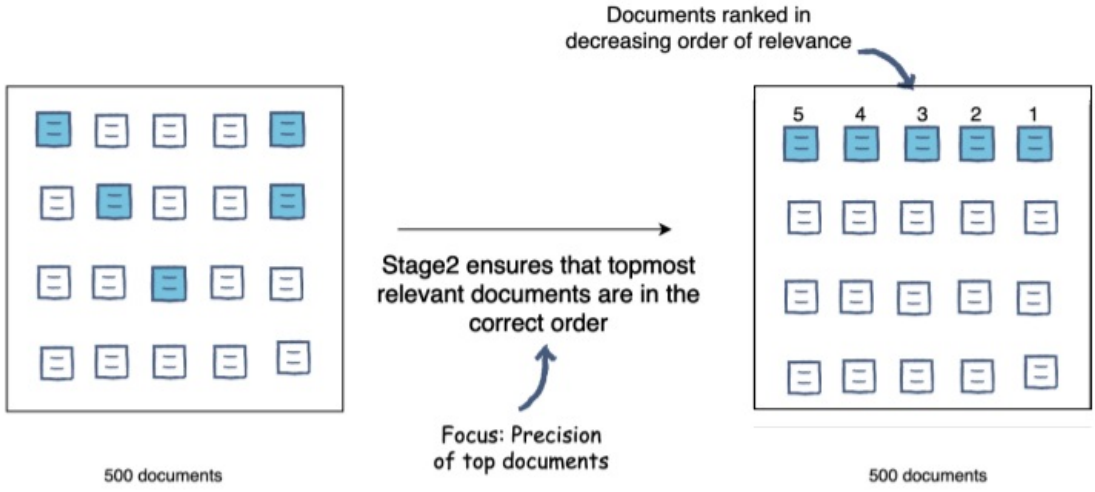
Let's assume here that you will use two layers for ranking with one-hundred thousand and five-hundred documents ranked at each layer, as shown below. Though, the choice of layers and documents ranked on each layer depends highly on the available capacity.


Document pool
Moderately relevant to query
Extremely relevant to query
Irrelevant to query


Document selection forwards relevant results
100 million documents on the web
100k documents



3 of 4



4 of 4

The configuration shown above assumes that the first stage will receive one-hundred thousand relevant documents from the document selection component. You then reduce this number to five-hundred after ranking in this layer, ensuring that the topmost relevant results are forwarded to the second stage (also referred to as the recall of the documents).

It will then be the responsibility of the second stage to rank the documents such that topmost relevant results are placed in the correct order (also referred to as the precision of the documents).

> First stage model will focus on the **recall** of the top five to ten relevant documents in the first five-hundred results while the second stage will ensure **precision** of the top five to ten relevant documents.

Stage 1

As we try to limit documents in this stage from a large set to a relatively smaller set, it's important that we don't miss out on highly relevant documents for the query from the smaller set. So, this layer needs to ensure that the top relevant documents are forwarded to stage 2. This can be achieved with the **pointwise approach**, which has been discussed previously. The problem can be approximated with the binary classification of results as relevant or irrelevant.

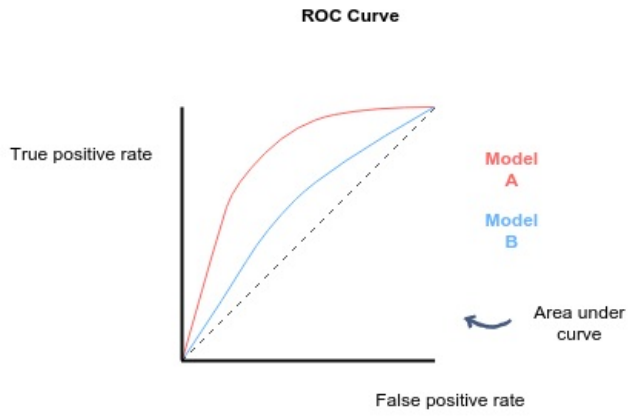
Logistic regression

A relatively less complex linear algorithm, like logistic regression or small MART(Multiple additive regression trees) model, is well suited for scoring a large set of documents. The ability to score each document extremely quickly (microseconds or less) for the fairly large document pool at this stage is super critical.

Analyzing performance

To analyze the performance of our model, we will look at the *area under curve* (AUC) of receiver operating characteristics curves or ROC curves.

For instance, if we train two models A and B on different feature sets, AUC will help us determine which model performs better.



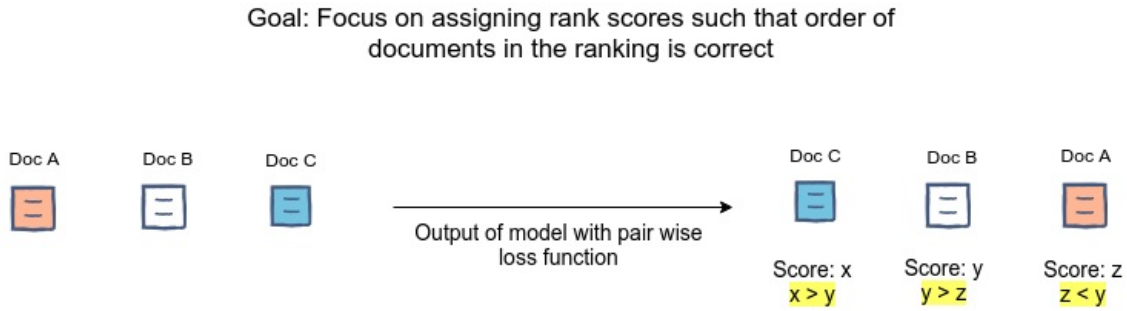
Model A is superior than Model B

We can observe that Model A is better than Model B because the area under its curve is greater.

Stage 2

As discussed previously, the main objective of our stage 2 model is to find the optimized rank order.

This is achieved by changing the objective function from a single pointwise objective (click, session success) to a pairwise objective. Pairwise optimization for learning to rank means that the model is not trying to minimize the classification error but rather trying to get as many pairs of documents in the right order as possible.



Let's look at two **pairwise learning algorithms**.

LambdaMART

LambdaMART is a variation of MART where we change the objective to improve pairwise ranking, as explained above. Tree-based algorithms are generally able to generalize effectively using a moderate set of training data. Therefore, if your training data is limited to a few million examples, this definitely will be the best choice to use in pairwise ranking in the second stage.

If we are optimizing for offline NDCG (based on human-rated data), then this is definitely one of the best options.

LambdaRank

LambdaRank is a neural network-based approach utilizing pairwise loss to rank the documents. Neural network-based models are relatively slower (given the large number of computations based on width and depth) and need more training data. So training data size and capacity are key questions before selecting this modeling approach. The online training data generation method for the pairwise approach can generate ranking examples for a popular search engine in abundance. So, that can be one option to generate ample pairwise data.

Your training data contains pairs of documents (i, j), where i ranks higher than j. Let's look at the Lambda rank model's learning process. Suppose we have to rank two documents i and j for a given query. We feed their corresponding feature vectors x_i and x_j to the model, and it gives us their relevance scores, i.e., s_i and s_j. The model should compute these scores (s_i and s_j) such that the probability of document i being ranked higher than document j is close to that of the ground truth. The optimization function tries to minimize the inversions in the ranking.

Both LambdaMART and LambdaRank are very well explained in this paper.

> We can calculate the NDCG score of the ranked results to compare the performance of different models.

← Back
Training Data Generation

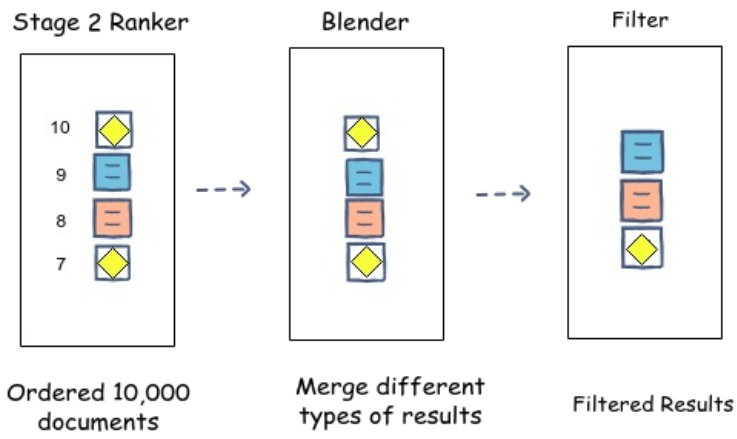
Next →
Filtering Results

Mark as

Filtering Results

Let's further filter the results based on how they might be perceived by the searcher despite good user engagement.

We'll cover the following



The layered model approach

Until now, you have selected relevant results for the searcher's query and placed them in order of relevance. The job seems to be pretty much completed. However, you may still have to filter out results that might seem relevant for the query but are inappropriate to show.

Result set after ranking

The result set might contain results that:

- are offensive
- cause misinformation
- are trying to spread hatred
- are not appropriate for children
- are inconsiderate towards a particular group

These results are **inappropriate despite having good user engagement**.

How do we solve this problem? How do we make sure that a search engine can be safely used by users of all age groups and doesn't spread misinformation and hatred?

ML problem

From a machine learning point of view, we would want to have a specialized model that removes inappropriate results from our ranked result set.

As we discussed for our main ranking problem, we would need training data, features, and a trained classifier for filtering these results.

Training data

Let's go over a couple of methods that we can use to generate training data for filtering undesired results.

Human raters

Human raters can identify content that needs to be filtered. We can collect data from raters about the above-mentioned cases of misinformation, hatred, etc. and from their feedback, we can train a classifier that predicts the probability that a particular document is inappropriate to show on SERP.

Online user feedback

Nowadays, good websites provide users with the option to report a result in case it is inappropriate. Therefore, another way to generate data is through this kind of online user feedback. This data can then be used to train another model to filter such results.

Features

We can use the same features for this model that we have used for training our ranker, e.g., document word embeddings or raw terms can help us identify the type of content on the document.

There are maybe a few particular features that we might want to add specifically for our filtering model. For example, *website historical report rate*, *sexually explicit terms used*, domain name, website description, images used on the website, etc.

Building a classifier

Once you have built the training data with the right features, you can utilize classification algorithms like logistic regression, MART(Boosted trees or Random Forest), or a Deep neural network to classify a result as inappropriate.

Similar to the discussion in the ranking section, your choice of the modelling algorithm will depend on:

- how much data you have
- capacity requirements
- experiments to see how much gain in reducing bad content do we see with that modelling technique.

← Back
Ranking

Next →
Problem Statement

Mark as Completed

Ask a Question