UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS num. 1768

Recency Ranking Models for Web Search

Paula Gombar

Zagreb, July 2018

I would like to thank Berkant Barla Cambazoğlu for his valuable help, guidance, and patience during my research visit in Barcelona. The same applies for the incredibly talented team members of the Web Ranking team. Back at home, I thank my supervisor Jan Šnajder for his mentorship throughout my entire studies, and Domagoj Alagić for reading my thesis. Last, but not least, I am immensely grateful to my family, friends, my better half, and my dog for their unconditional love and support, forever and always.

Contents

1. Introduction

1.1. Motivation

Given the vast amount of data on the Web today, search engines have become a key tool for finding information of interest. At their core, search engines are information retrieval (IR) systems. Information retrieval is an area of data science dealing with obtaining information from a document collection relevant to an information need. Nowadays, research in IR includes modeling, web search, text classification, systems architecture, user interfaces, data visualization, etc. (Baeza-Yates et al., 1999). In practice, IR consists of building efficient indexes, processing user queries with high performance, and developing ranking algorithms to improve the quality of the results.

The task of a search engine is to handle user queries and retrieve results relevant to that query in the form of a search engine result page (SERP). This can be a challenging problem, especially given the amount of data processed by search engines today. For example, Google, being the top search engine by market share, processes around 3.5 billion searches per day and has an index containing hundreds of billions of documents. Moreover, the rate of content change is higher than ever, emphasizing the need for results that are up to date.

The retrieved results are ranked according to some properties. The three main properties we identify are the importance of the document source, relevance to the query, and recency. This thesis focuses on introducing recency ranking to an existing commercial search engine.

1.2. Background

A traditional search engine supports three fundamental tasks: web crawling, indexing, and searching. The crawler is responsible for discovering pages and downloading them to the search engine's internal store. The content is parsed and indexed by the indexer. Finally, the search component is responsible for matching the documents to the user query and ranking them. A high-level overview of the search process is shown in

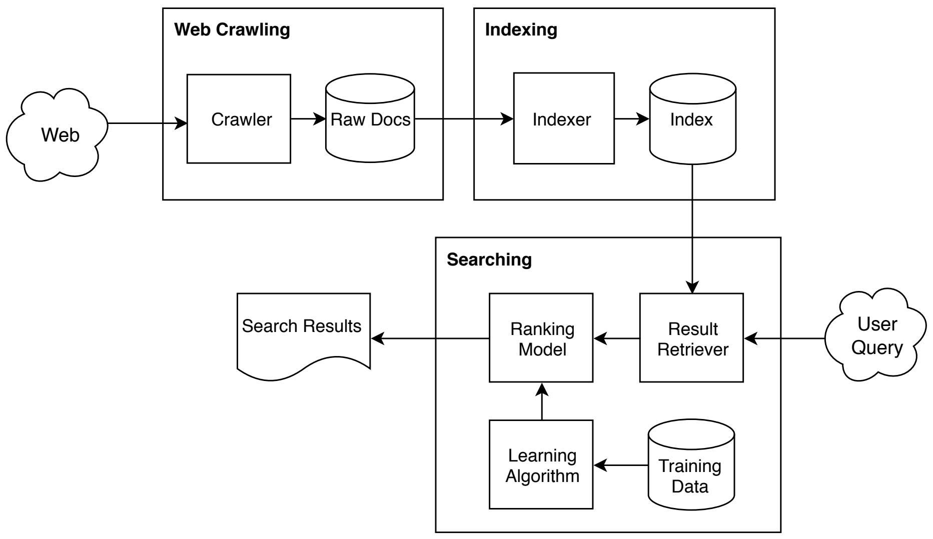


Figure 1.1: An example of a search engine architecture.

Figure 1.1.

Modern search engines use learning-to-rank algorithms for the ranking task. Learning to rank is the application of machine learning in information retrieval systems in order to construct ranking models. Such a model, typically supervised, is trained on training data consisting of query-document pairs, each with a relevance degree. The purpose of the model is to rank documents, i.e., produce an ordering that is optimal with respect to an evaluation metric.

Some of the evaluation metrics for ranking are mean average precision (MAP), discounted cumulative gain (DCG) and NDCG (normalized DCG), precision@n, NDCG@n (where @n denotes that the metrics are evaluated only on top n documents), mean reciprocal rank, Kendall's tau, and Spearman's rho. We use NDCG@n, so we will explain it in detail here. NDCG is a normalized version of discounted cumulative gain (Järvelin and Kekäläinen, 2002). Using a graded relevance scale of documents in a search-engine result set, DCG measures the usefulness, or gain, of a document based on its position in the result list. The gain is accumulated from the top of the result list to the bottom, with the gain of each result discounted at lower ranks. The graded relevance value is reduced logarithmically proportional to the position of the result, therefore penalizing highly relevant documents appearing lower in a search result list. The DCG at rank position p is defined as:

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

where rel_i is the graded relevance of the result at position i.

We use normalized DCG, because search result lists vary in length depending on the query. In this case, the cumulative gain at each position for a chosen value of p should be normalized across queries. We do this by sorting all relevant documents in the corpus by their relative relevance, producing the maximum possible DCG through position p, called Ideal DCG (IDCG) through that position. NDCG is defined as:

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

and IDCG is defined as:

$$\mathrm{IDCG_p} = \sum_{i=1}^{|REL|} \frac{rel_i}{\log_2(i+1)},$$

where $|REL|$ represents the list of ordered relevant documents up to position p.

In an online scenario, when a user types a query into a search engine, they expect to see results in a very short time, typically a few hundred milliseconds. This makes it impossible to evaluate a complex ranking model, so modern search engines use a two-phase retrieval and ranking strategy (Cambazoglu et al., 2010). In the first phase, the scoring is very simple, enabling a quick retrieval process. Consequently, the second phase uses a more complex model, since now only a small subset of the overall document index is being ranked.

The query-document pairs are scored, then ranked, based on their feature vectors. The features can be divided into three groups:

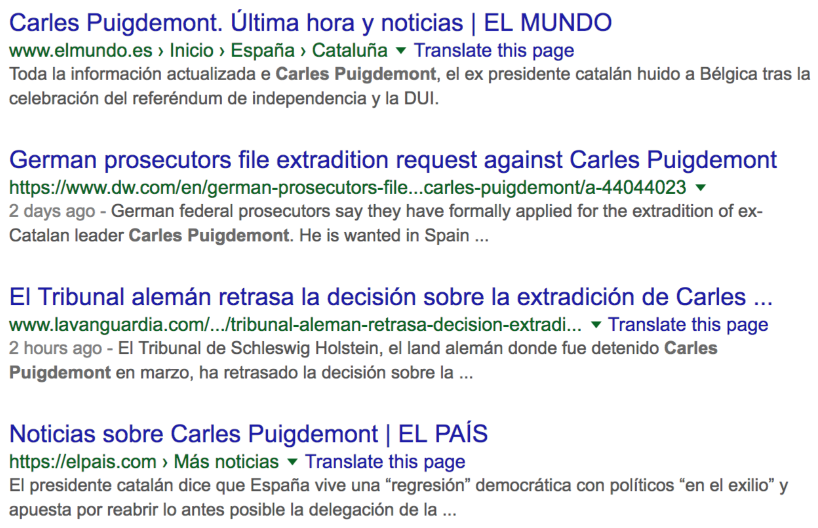
1. Query-independent or static features: features depending only on the document, but not on the query. For example, PageRank (Page et al., 1999) or the document's length. These features can be precomputed offline when indexing the document.

2. Query-dependent or dynamic features: features depending both on the contents of the document and the query, such as the similarity between the query terms and the document content.

3. Query features: features depending only on the query, such as the number of words in a query.

1.3. Problem Definition

The effectiveness of a search engine depends on the relevance of the result set it returns. Search engines employ ranking methods to provide the best results first. As more and more content is put on the Web, now more quickly than ever, the temporal aspect of results is very important for modern search engines. In web search, recency ranking refers to ranking the documents both by their relevance to the user query, and their freshness. The search engine can appear stale if it failed to recognize the



Carles Puigdemont. Última hora y noticias | EL MUNDO
www.elmundo.es › Inicio › España › Cataluña ▾ Translate this page
Toda la información actualizada e **Carles Puigdemont**, el ex presidente catalán huido a Bélgica tras la celebración del referéndum de independencia y la DUI.

German prosecutors file extradition request against Carles Puigdemont
https://www.dw.com/en/german-prosecutors-file...carles-puigdemont/a-44044023 ▾
2 days ago - German federal prosecutors say they have formally applied for the extradition of ex-Catalan leader **Carles Puigdemont**. He is wanted in Spain ...

El Tribunal alemán retrasa la decisión sobre la extradición de Carles ...
www.lavanguardia.com/.../tribunal-aleman-retrasa-decision-extradi... ▾ Translate this page
2 hours ago - El Tribunal de Schleswig Holstein, el land alemán donde fue detenido **Carles Puigdemont** en marzo, ha retrasado la decisión sobre la ...

Noticias sobre Carles Puigdemont | EL PAÍS
https://elpais.com › Más noticias ▾ Translate this page
El presidente catalán dice que España vive una "regresión" democrática con políticos "en el exilio" y apuesta por reabrir lo antes posible la delegación de la ...

Figure 1.2: An example of a search engine result page (SERP) from Google for the query *Carles Puigdemont* submitted on June 29, 2018.

temporal aspect of a query, which can also negatively affect the user experience. Effectively combining temporal features, as well as relevance-sensitive features in a ranking framework is still an open question for the research community.

The connection between relevance and recency is not always clear. In other words, if optimizing for only one of them, we might decrease the effectiveness of the search engine. Therefore, it is necessary to find an equilibrium. The trade-off can also depend on the user's query. For example, for a recency sensitive query, i.e., a query for which a user expects topically relevant documents to be also fresh, recency is very important. On the other hand, for timely queries, i.e., queries that do not show a spike in their popularity over time, relevance should be more important. Figure 1.2 shows an example of a blend of relevant and recent results.

Even when the query type is identified, in order to incorporate both recent and relevant results, we need to determine the recency component on the document side. When dealing with general documents from the Web, this is not a trivial task, as most of the documents do not have any temporal information.

In our experiments, we ask the following research questions:

1. How can we determine if a document's content is recent without having information on its publication date?

2. How can we determine when queries seek recent results?

3. How can we introduce recency ranking into an existing relevance ranking architecture?

1.4. Contributions

The contributions of this thesis are both theoretical and practical. In Chapter 3, we present a document age prediction model, used to predict how recent a document is by extracting temporal evidence from its content. In Chapter 4, we present a query recency sensitivity classifier, used to predict the need of recent results for a query. Finally, in Chapter 5, we present a novel method of incorporating recency with the help of these two classifiers into an existing search engine. We also present an exhaustive evaluation of all presented models.

2. Related Work

In the past decade, there have been a number of papers approaching the problem of combining relevance and recency in ranking. Some of them deal with more specific domains than web search, such as mail search (Carmel et al., 2017), microblog retrieval (Efron, 2012), and news search (Dakka et al., 2012). In the domain of mail search, a recency-only ranking is traditionally implemented. Even when time-ranked results are mixed with relevance-based results, as Carmel et al. (2017) report, showing duplicate results to the user is not discouraged. These two key points make mail search a different problem than web search. Moreover, when it comes to news search, it is assumed that the publication date of the document is available (Dakka et al., 2012). For the majority of documents on the Web, this is not the case, making web search a more difficult problem.

The works that directly improve ranking recency in web search are (Dong et al., 2010a,b; Dai et al., 2011; Styskin et al., 2011). Dong et al. (2010a) focus on breaking-news queries and build different rankers. If a query is classified as recency-sensitive, a recency-sensitive ranker is used to rank the documents matching that query. Similar to our work, they model different time slots by building language models from different sources (news content and queries) and compare them in order to classify if a query is recency-sensitive or not. Furthermore, they extract recency features from documents, and define the *page age* as the time between the query submission time and page publication time, which is either the page creation time or the time it was last updated. The recency features they extract from documents are content-based and link-based. In our approach, we only extract content-based evidence. Finally, they manually annotate query-url pairs and train two separate models: a recency ranker and a relevance ranker. The key difference between this work and ours is that we do not manually annotate data, but automatically produce labels, we use only one ranking model for all queries, and we focus on all types of queries.

Extending their existing approach, Dong et al. (2010b) introduce Twitter features to help with recency ranking. However, they do not focus on the fresh content produced by tweets to determine if the query terms are recency-sensitive, but instead they employ a URL mining approach to detect fresh URLs from tweets and learn to rank them.

Instead of having separate rankers, Dai et al. (2011) propose a machine learning framework for simultaneously optimizing recency and relevance, focusing on improving the ranking of results for queries based on their temporal profiles. They determine the temporal profile of a query by building a time series of the relevant documents' content changes. Next, they extract temporal features from the seasonal-trend decomposition of different time series. We could not use this approach, as we only keep the most updated snapshot of documents in our index.

Similarly, Cheng et al. (2013) model term distribution of results over time and conclude that this change is strongly correlated with the users' perception of time sensitivity. However, they only focus on improving ranking for timely queries, i.e., queries that have no major spikes in volume over time, but still favour more recently published documents. In a similar work, Efron and Golovchinsky (2011) assume that the document publication time is known. They compute a query-specific parameter that captures recency-sensitivity and is calculated based on the distribution of the publication times of the top retrieved documents by a relevance-based ranker.

However, when document publication time is not known, we have to search for other indicators of document recency and query recency sensitivity. For example, Campos et al. (2016) use temporal expressions from web snippets related to the query to improve the ranking. In this work, we do the same. Other possible sources of temporal features include click logs and query logs. Wang et al. (2012) learn the relevance and recency models separately from two different sets of features. Their divide-and-conquer learning approach is similar to (Dai et al., 2011), but in this work, they omit manually annotating data, and instead automatically infer labels using clickthrough data. We do not use click logs, but we do extract frequency features from our query log, similarly to (Metzler et al., 2009; Lefortier et al., 2014).

So far, we have explained how related work has extended existing learning-to-rank algorithms to take into account both relevance and recency. Another approach to recency ranking is an aggregated search strategy. In other words, recent results are extracted from a fresh vertical (such as news articles) and subsequently integrated into the result page. The blending of results from different verticals is called result set diversification. Examples of this approach are (Lefortier et al., 2014; Styskin et al., 2011). In these two papers, a classifier is used to score the recency sensitivity of a user query, and this score is used to determine to what extent the documents from the fresh vertical should be inserted. Even though we do not perform result set diversification, we share similarities with these papers. The query fresh intent detector from (Lefortier et al., 2014) is similar to our query recency classifier in terms of features used, and we have modeled our query ground truth labels according to the distribution reported in (Styskin et al., 2011).

3. Document Age Prediction Model

In order to model how recent a document is, we try to predict the time the current content was last updated. Given that information, we can define the document age as:

$$documentAge = currentTime - documentLastUpdateTime$$

Here, $currentTime$ is the time when we are observing the document, for example, at query issue time.

If we know the last update time of a document, it is trivial to calculate the document age on the fly. For example, in the case of news search, it is almost always the case that we are provided with a publication time and we can simply use it as a feature, as seen in (Dakka et al., 2012). Moreover, Spitz et al. (2018) have shown that, for news articles, we can also take advantage of the citation network, or the inbound and outbound links in the articles.

However, as we are focusing on documents on the Web in general, we cannot assume that we have this information readily available. To this end, we developed a machine-learned model to predict the last update time of a document.

We approach this as a supervised regression problem, so we first obtain a labeled set of documents to serve as ground truth. We extract features from the content of the document which indicate when the document was last updated, and learn and evaluate a machine learning model. We explain how we compile the ground truth in the next section.

3.1. Ground Truth Creation

We create a ground truth set for a random sample of 5K documents from our document index. Instead of manually annotating the time a document was last updated, we infer it automatically by querying the Memgator web service.[1] As described by SalahEldeen and Nelson (2013), this service can be used for navigating between the current and the past web. It provides a list of *mementos*, or timestamps, when the document was

[1]`http://memgator.cs.odu.edu/`.

changed. We note that a memento timestamp is the time of capture at the web archive, which might not entirely align with the actual update time of the document. However, we deemed this approximation good enough for our purposes.

For a given URI, the service returns a list of timestamps, ordered from oldest to newest, when there was an update to the Web page. Figure 3.1 shows an example of a request and response from the service. The list of mementos is shortened for readability.

Listing 3.1: Example of a request to and response from the Memgator web service.

```
curl "https://memgator.cs.odu.edu/timemap/json/www.ntent.com"
{
  original_uri: "http://www.ntent.com",
  self: "https://memgator.cs.odu.edu/timemap/json/www.ntent.com",
  mementos: {
    list: [
      {
        datetime: "2000-04-08T22:25:51Z",
        uri: "http://web.archive.org/web/20000408222551/www.ntent.com
            :80/"
      },
      (...)
      {
        datetime: "2018-06-14T23:55:54Z",
        uri: "http://web.archive.org/web/20180614235554/www.ntent.com/"
      }
  ]
    first: {
        datetime: "2000-04-08T22:25:51Z",
        uri: "http://web.archive.org/web/20000408222551/www.ntent.com
            :80/"
    },
    last: {
        datetime: "2018-06-14T23:55:54Z",
        uri: "http://web.archive.org/web/20180614235554/www.ntent.com/"
    }
  }
```

A naïve approach would be to take the `last` field from the JSON, signifying the time of the last recorded change. However, when dealing with documents on the Web, it is often the case that an update is not actually a content update, but a page maintenance update. In other words, the change in document content is negligible and we do not want to capture such *near-duplicates*.

To make sure we are capturing actual content updates, we traverse the list of last updates in reverse, from the latest to oldest timestamp, and compare the current document content to the previous one. If the similarity is above a certain threshold ($t = 90\%$), we discard the update and keep going back in time. We compute the similarity between two documents by *shingling* them (Broder, 1997).

Shingling is a technique used in information retrieval to detect near-duplicate documents. Given a positive integer k and a sequence of terms in a document d, we define the k-shingles of d to be the set of all consecutive sequences of k terms in d. Let $S(d_j)$ denote the set of shingles of document d_j. For example, for $d =$ *Barcelona is the capital of Catalonia*, and $k = 3$, the shingles are {(Barcelona, is, the), (is, the, capital), (the, capital, of), (capital, of, Catalonia)}. We then produce an MD5 hash of each shingle, sort them alphabetically, and take the first N shingles as a set. We denote this as $S(d_j)$. We compute the Jaccard coefficient to measure the degree of overlap between the sets $S(d_1)$ and $S(d_2)$ as:

$$J(S(d_1), S(d_2)) = \frac{|S(d_1) \cap S(d_2)|}{|S(d_1) \cup S(d_2)|}.$$

If $J(S(d_1), S(d_2)) < 0.90$, the documents are assumed to be distinct.

Finally, the ground truth dataset consists of 5248 documents, where each document object contains the following fields:

1. `URL`. The full URL of a document, also a unique identifier of a document.
2. `FullHTML`. The full HTML content retrieved by the crawler, without any preprocessing.
3. `ContentHTML`. The content of the document after boilerplate removal was performed. This does not contain JavaScript blocks, CSS, and such.
4. `DocumentLastUpdateTime`. The target variable, obtained using the aforementioned technique, expressed as Unix time.[2]

3.2. Feature Extraction

We extract features from the URL and the content of the document. For the content, we process both the full HTML of the document (before boilerplate removal), and

[2]Unix time is the number of seconds that have elapsed since January 1, 1970.

from the clean HTML (after boilerplate removal). Table 3.1 shows the statistics of the documents from our ground truth. The average length of the raw document HTML is ten times longer than the clean one. Although some of this is due to the HTML format (e.g., tags and attributes), there is still a lot of noisy content that could be used to extract features indicative of the document's last modification date. Therefore, we choose to process both versions of the document, and have both groups of extracted features in the resulting feature vector.

Table 3.1: Ground truth dataset statistics.

Number of documents	Average FullHTML size	Average ContentHTML size
5248	108054 characters	10961 characters

First, we develop a pattern-matching solution to extract any mention of dates. A comprehensive list of supported date formats is shown in Table 3.2.

Table 3.2: Date formats supported in regular expressions.

Date format	Example date	Date format	Example date
dd/mm/yy	27/05/93	yy-mmm-dd	93-Aug-27
dd/mm/yyyy	27/05/1993	yy-mmmm-dd	93-August-27
d/m/yy	5/5/93	mm/dd/yy	05/27/93
d/m/yyyy	5/5/1993	mm/dd/yyyy	05/27/1993
dd-mmm-yy	27-Aug-93	mmm-dd-yy	Aug-27-93
dd-mmm-yyyy	27-Aug-1993	mmm-dd-yyyy	Aug-27-1993
d-mmmm-yy	27-August-93	mmmm dd, yyyy	August 27, 1993
d-mmmm-yyyy	27-August-1993	dd mmmm yyyy	27 August 1993
yy/mm/dd	93/05/27	mmm yyyy	Aug 1993
yyyy/mm/dd	1993/05/27	mmmm, yyyy	August, 1993
yyyy-mm-dd	1993-05-27	yyyy	1993

We denote the features that capture dates as `timestamp features`. When extracting timestamps containing uncertainty of the exact date, we take the approach as Berberich et al. (2010). More specifically, we support both Americanized date formats (first month, then day) and standardized (first day, then month). We use several regular expressions to capture different date formats, transform all dates to a standardized format `YYYY-MM-DD`, then convert the extracted dates to Unix time. Moreover, since we aim to predict the last update time on a year-month scale, we set the day field to the 15th of the month.

Table 3.3: List of features.

Feature index	Feature name
1	TimestampFeatures_FullHtml_MinTimestamp
2	TimestampFeatures_FullHtml_MaxTimestamp
3	TimestampFeatures_FullHtml_AvgTimestamp
4	TimestampFeatures_FullHtml_FirstTimestamp
5	TimestampFeatures_FullHtml_LastTimestamp
6	TimestampFeatures_FullHtml_MostCommonYear
7	TimestampFeatures_FullHtml_MostCommonYearMonth
8	TimestampFeatures_Html_MinTimestamp
9	TimestampFeatures_Html_MaxTimestamp
10	TimestampFeatures_Html_AvgTimestamp
11	TimestampFeatures_Html_FirstTimestamp
12	TimestampFeatures_Html_LastTimestamp
13	TimestampFeatures_Html_MostCommonYear
14	TimestampFeatures_Html_MostCommonYearMonth
15	MetaTagFeatures_FullHtml_LastModified
16	MetaTagFeatures_FullHtml_Copyright
17	MetaTagFeatures_FullHtml_DCDateIssued
18	MetaTagFeatures_FullHtml_DCTermsModified
19	JQueryVersion
20	TimestampFeatures_Url

Apart from the `timestamp features`, we also extract features from specific `<meta>` tags, called `meta tag features`, and the `jQuery version feature`. The total number of features is 20, where each feature value is expressed as Unix time. The list of features is shown in Table 3.3.

3.2.1. Timestamp Features

We extract `timestamp features` from the full HTML of the document (pre-boilerplate removal), the clean HTML (post-boilerplate removal), and document URL. We want to be able to predict the last update time of a document on a monthly scale, so we only store the year and month of retrieved timestamps, whereas we always set the day value to the 15th.

The `timestamp features` are the following:

– `MinTimestamp`: the minimal timestamp mentioned;
– `MaxTimestamp`: the maximal timestamp mentioned;
– `AvgTimestamp`: the average timestamp mentioned;
– `FirstTimestamp`: the first timestamp mentioned;
– `LastTimestamp`: the last timestamp mentioned;
– `MostCommonYear`: the most commonly mentioned year, converted to June 15th of that year;
– `MostCommonYearMonth`: the most commonly mentioned year and month, converted to the 15th of that month.

We also extract any timestamps that are mentioned in the URL and model it as a separate feature.

3.2.2. Other Features

For the `meta tag features`, we capture four different tag types:

```
<meta name="Last-Modified" content="2017-06-05 11:21:43">
<meta name="copyright" content="Copyright (c) 2017 ABC News Internet
    Ventures">
<meta name="dcterms.modified" content="2007-12-20 11:17:23">
<meta property="DC.date.issued" content="2017-04-27T08:00:40-04:00">
```

Lastly, for the `jQuery feature`, we precomputed a lookup table of release dates of different jQuery versions.[3] Often times, documents include a snippet referencing the minimum jQuery version, such as:

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.
    min.js"></script>
```

If there is a mention of a jQuery version number in the full HTML, we set the `jQuery feature` value as the release date of that version.

3.3. Model Training

We use Gradient Boosted Regression Trees (GBRT) as the model. Gradient boosting is a machine learning technique where the prediction model is an ensemble of weak prediction models, in this case decision trees. The idea of gradient boosting was first

[3]`https://gist.github.com/0xdevalias/8e558f4ed452ec54880e8b8445d13e28`.

introduced by Breiman (1997). Later, Friedman (2001) introduced improvements to the boosting algorithms.

Decision trees were suitable for this problem, as they are easy to interpret, they require little data preparation, and have built-in feature selection. We use XGBoost (eXtreme Gradient Boosting), introduced by Chen and Guestrin (2016), an open-source parallel gradient boosting library to train and evaluate the model.

As mentioned earlier, we have a total of 20 features, where each feature value is Unix time. Each training instance is of the `LibSVM data format`:[4]

```
<label> <index_1>:<value_1> ...  <index_N>:<value_N>
```

Here, `<label>` is the target value, expressed in Unix time, and the rest of the fields model the feature index and value, respectively.

To investigate which features are more useful than others, we plot the feature importance scores, as seen in Figure 3.1. By looking at the feature overview in Table 3.3, we can see that the first five features, `timestamp features` retrieved from the full HTML content, are the most important, followed by `timestamp features` retrieved from the clean HTML. The worst-performing group of features are the `meta tag features`, due to the fact that these attributes are seldom included in documents in our collection. The same applies for the `jQuery version feature`, which, although very indicative of the document publication date, was rarely present in our document set.

3.4. Evaluation

For training the model, we split the input set into train and validation sets in the 70:30 ratio. We then proceeded to grid search the hyper-parameters of the model using a 5-fold cross-validation. Listing 3.2 shows which hyper-parameters are tuned.

Listing 3.2: Hyper-parameter tuning for the document age model.

```
'max_depth': [4,6,8,10],(best = 4)
'min_child_weight': [0,1,2],(best = 0)
'gamma': [0.0, 0.1, 0.2, 0.3], (best = 0.0)
'subsample': [0.6, 0.8, 1.0], (best = 1.0)
'colsample_bytree': [0.6, 0.8, 1.0], (best = 0.6)
'learning_rate': [0.01, 0.03, 0.1, 0.2, 0.3], (best = 0.01)
'n_estimators': [100,500,1000,5000],(best = 1000)
```

[4]https://www.csie.ntu.edu.tw/~cjlin/libsvm/.

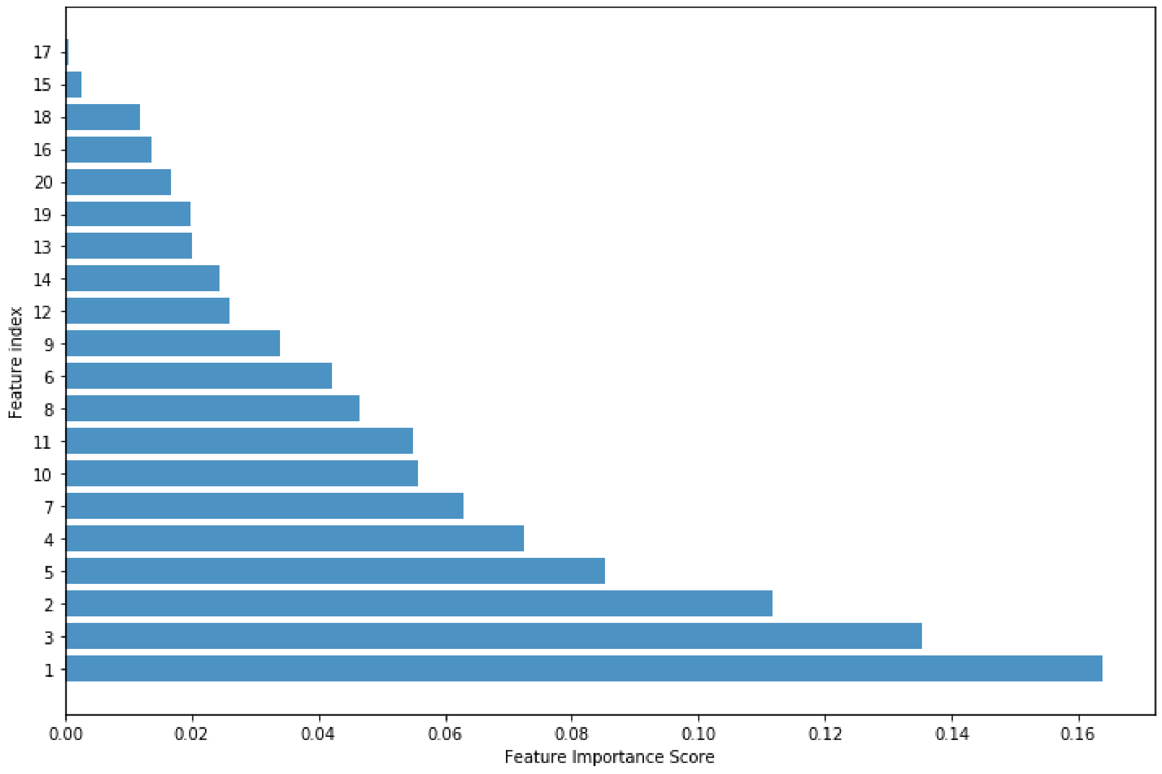


Figure 3.1: Feature importance scores of the document age prediction model.

Parameter `max_depth` is the maximum depth of a tree, and increasing this value will make the model more complex and likely to overfit. Parameter `min_child_weight` is the minimum sum of instance weight needed in a child to continue further partitioning. The larger, the more conservative the model. Parameter `gamma` is the minimum loss reduction required to make a further partition on a leaf node of the tree. The larger, the more conservative the model. Parameter `subsample` is the ratio of training instances to subsample. Setting it to 0.5 means that XGBoost randomly collects half of the data instances to grow trees and this will prevent overfitting. Parameter `colsample_bytree` is the subsample ratio of columns when constructing each tree. Parameter `learning_rate` is the step size shrinkage used in the update phase to prevent overfitting. Parameter `n_estimators` is the number of trees to construct.

The training of the model took 23 hours on a machine with CentOS 7.5.1804, 16 GB of RAM, and 8 Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz CPUs. The model was trained by minimizing the root mean squared error (RMSE) of the difference between the predicted Unix time value and the value from the ground truth.

The results on the train and test sets are shown in Table 3.4. To provide a better intuition for the results, the epoch timestamp (or Unix time) of `26 June 2018` is 1529971200. Moreover, an RMSE value of $3e+07$ is equivalent to `14 December 1970`. Since Unix time is defined as the number of seconds since `1 January 1970`, we can conclude that the model's predictions are accurate within a year.

Table 3.4: Document age prediction model evaluation.

RMSE train	RMSE test
3.202e+07	3.596e+07

To gain insight into how our model is performing, we plot two different learning curves. The first learning curve, shown in Figure 3.2, shows the model performance based on the number of training rounds. We run a 5-fold cross-validation on different subsets of training instances. We expect to see an increase in the error on the training set, as we increase the number of training instances. This is because the model is likely to overfit and predict well when given a smaller number of training instances. However, as the model is trained on more data, it manages to fit better the validation set. Thus, the validation error decreases. Based on this graph, it looks like we could benefit from adding more training instances, and still not be at risk of overfitting the model.

The second learning curve is shown in Figure 3.3. Here, we examine the model's performance based on the number of training rounds. We notice that both the training and validation error reduce rapidly. Nevertheless, the validation error does not reduce significantly after 400 rounds. If we wanted to reduce the time and computational power necessary to train the model, this graph shows we can stop training after 400 rounds, since no significant improvement is observed after that point. Note, however, that we trained the model with setting the configuration parameter `early_stopping_rounds` to 50, which means there was a reduction in validation error after all.

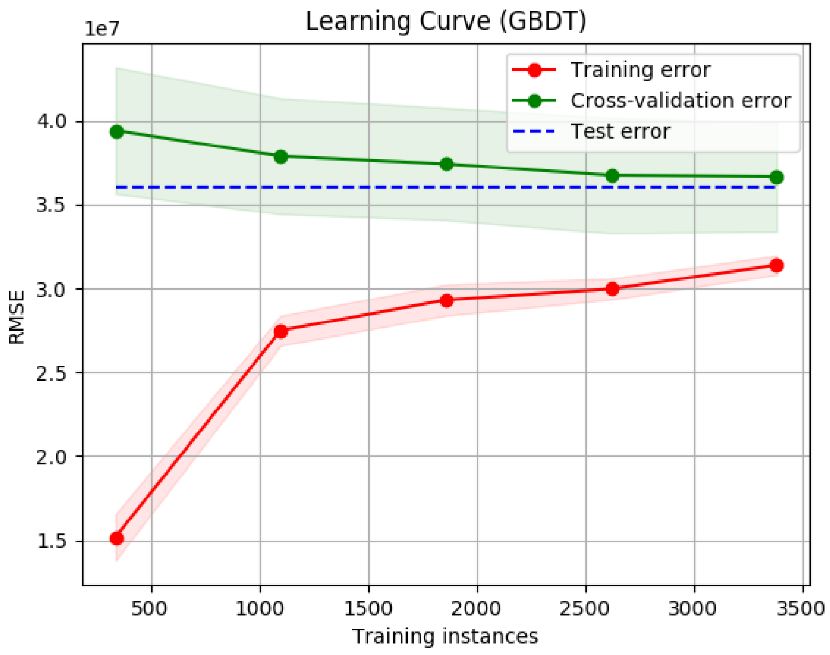


Figure 3.2: Learning curve of the document age prediction model based on the number of training instances.

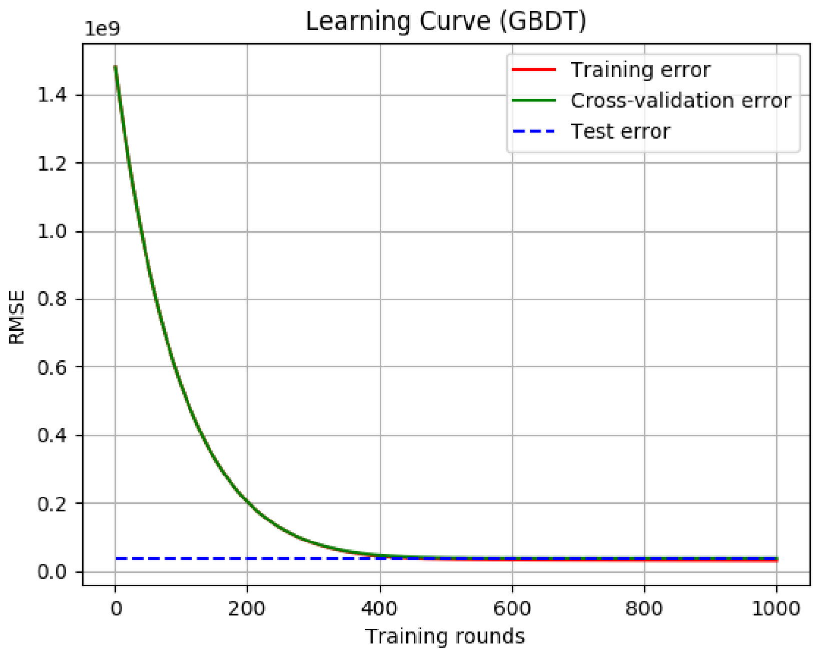


Figure 3.3: Learning curve of the document age prediction model based on the number of training rounds.

4. Query Recency Sensitivity Model

A web search query is a query that a user enters into a web search engine to satisfy their information needs. As described by Broder (2002), queries have empirically been divided into three different groups:

– Informational queries: queries that cover a broad topic for which there may be thousands of relevant results (e.g., *barcelona*).

– Navigational queries: queries that seek a single website or web page of a single entity (e.g., *youtube*, *facebook*).

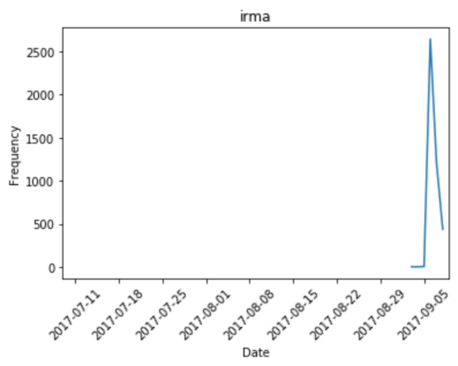
– Transactional queries: queries that reflect the intent of the user to perform a particular action, such as downloading a screen saver.

Another possible division of queries can be according to their temporal profiles, as defined by Jones and Diaz (2007). They propose three temporal classes of queries:

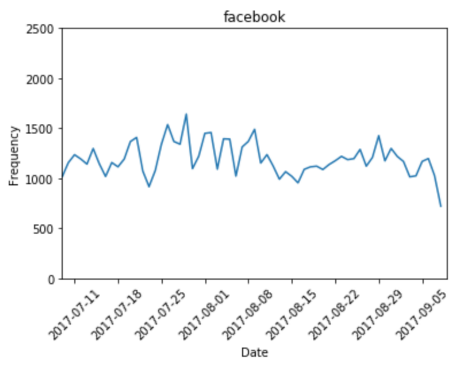
– Atemporal queries: queries that take place at any time.

– Temporally unambiguous queries: queries that take place at a specific period in time.

– Temporally ambiguous: queries that take place during one of several possible episodes.



(a) A temporally unambiguous query.



(b) An atemporal query.

Figure 4.1: Examples of queries with different temporal profiles.

Figure 4.1 shows an example of queries with different temporal profiles. The query *irma* is temporally unambiguous, as it refers to the hurricane Irma, whereas the query *facebook* is atemporal.

User experience is improved by integrating recency of retrieved results for most types of queries. Therefore, instead of focusing on a specific query type, we choose to introduce the recency vertical for queries in general.

As described by Styskin et al. (2011), not all queries require the same percentage of recent results, and not all queries seek the same granularity of result recency (e.g., hours, days, years). We call this property *query recency sensitivity.* We argue that the inference of query's recency sensitivity has an impact on the ranking of retrieved results.

In a conventional information retrieval setting, a query instance is defined only as the query string. However, as we are introducing a recency component, we have to capture the temporal aspect of the query as well. For example, for the query *stephen hawking*, the user might expect little to no change in results when submitting the same query in different points in time. Nonetheless, if searching for *stephen hawking* on March 14, 2018, we would expect to see more breaking-news results than usually, since that was the day he died. Thus, in a query recency sensitivity setting, we define a query instance as:

$$queryInstance = (queryText, submissionTime)$$

In other words, the same query can be submitted at different times, and yield a different ranking of results. Styskin et al. (2011) also defined different levels of recency sensitivity. They define four labels of recency sensitivity and ask human judges to annotate real user queries. The labels are probabilities from $[0, 1]$, and are defined as the following:

1. `0.95`: the query is strictly about a recent event, e.g., *stephen hawking death* on the day of the event;
2. `0.75`: the query's primary interest is related to a recent event, but the user also wants just topically relevant results, e.g., *oscar* on the day of the ceremony;
3. `0.25`: the query's primary interest is not likely to be focused on a particular event, but it makes sense to present some recent content, e.g., *britney spears*;
4. `0`: otherwise, the query is assigned zero probability to be recency sensitive.

We use these observations in constructing our ground truth dataset.

4.1. Ground Truth Creation

Instead of manually annotating queries, we choose to automatically infer recency sensitivity for queries. For this, we use a previously compiled ground truth dataset used for web ranking evaluation.[1] The dataset consists of 142807 queries submitted to a third party search engine in the period of July 2017 – September 2017.

Unfortunately, when this dataset was compiled, the query submission date was not recorded, as it was deemed unnecessary for relevance-based ranking evaluation. However, we were able to reconstruct it based on the order in which the queries were submitted, the rate they were submitted at, and the start and end date of the scraping. Since we are only able to reconstruct the submission date, not the exact time, we set the time as 23:59 of that day.

The queries were submitted to a third party search engine, and the first 50 results per query were scraped. An example of a partial result set for the query *kardashian* is shown in Listing 4.1.

Listing 4.1: Example of retrieved results for a query.

```
{
  query_text: "kardashian",
  query_submission_date: "2017-08-27 23:59:00",
  results: {
    list: [
      {
        rank: "0",
        uri: "https://www.instagram.com/kimkardashian/?hl=en",
        title: "Kim Kardashian West (@kimkardashian): Instagram photos
            and videos",
        snippet: "103m Followers, 111 Following, 3933 Posts - See
            Instagram photos and videos from Kim Kardashian West (
            @kimkardashian)"
      },
      {
        rank: "1",
        uri: "https://en.wikipedia.org/wiki/Kim_Kardashian",
        title: "Kim Kardashian - Wikipedia",
        snippet: "Kimberly Kardashian West is an American reality
            television personality, socialite, actress, businesswoman
```

[1]The details of this dataset cannot be disclosed, as it was constructed within a commercial project.

```
            and model. Kardashian first gained media attention..."
        },
        {
          rank: "2",
          uri: "http://people.com/babies/kanye-west-kim-kardashian-
            expecting-third-child-surrogate-pregnant/",
          title: "Kanye and Kim Kardashian West Expecting Third Child -
            People",
          snippet: "19 hours ago - Kim Kardashian and Kanye West Expecting
             Baby No. 3 via Surrogate! ..."
        },
        {
          rank: "3",
          uri: "http://people.com/babies/kim-kardashian-first-red-carpet-
            appearance-baby-number-3/",
          title: "Kim Kardashian Makes First Red Carpet Appearance
            Following News ...",
          snippet: "10 hours ago - Kim Kardashian West made her first red
            carpet appearance at New York Fashion Week on Wednesday
            following the news that she is expecting..."
        }
        (...)
      ]
    }
}
```

The snippet field shows the beginning of the document, and sometimes it includes information on when the document was published. This timestamp is relative to the query submission date, and can be in the form of `1-23 hours ago`, `2-7 days ago`, or a specific date, i.e., `May 27, 2018`. We take advantage of this temporal information when automatically determining which queries are more recency sensitive than others.

Our idea is to sort the queries in descending order based on a custom recency sensitivity function, then apply labels based on the distribution of the classes shown in Figure 4.2, and defined by Styskin et al. (2011). To score a query, we first take all the results with snippets that contain `1-23 hours ago`. We convert the relative timestamp to an absolute one by subtracting its value from the query submission time and express it as Unix time. We define the query score as the sum of the timestamps from the described snippets. Lastly, we sort the queries according to this score in

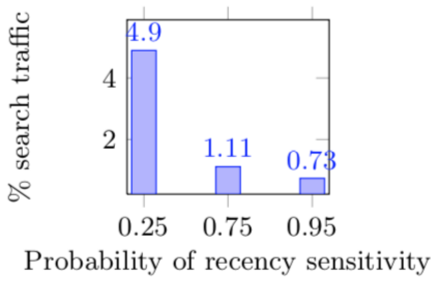


Figure 4.2: Distribution of query recency sensitivity classes defined by Styskin et al. (2011).

descending order, and assign the label `0.95` to the first 0.73% of queries, `0.75` to the next 1.11%, `0.25` to the next 4.9%, and `0.0` to the rest. Next, we downsample our query set from the initial 142807 queries to 4000 by maintaining the distribution of classes. Finally, we use this as ground truth for the query recency sensitivity model.

4.2. Feature Extraction

We use 34 features in our model. We extract features from three different sources:

– Our query log. Features such as the number of times the query was submitted over different time periods, and the ratio of the number of submissions of the query in the last day and in the last week. Additionally, the probabilities of the query being generated by language models based on the query log over different time periods.

– The query itself. These are features such as the number of tokens, and several boolean features.

– Twitter. We mine recent tweets with respect to the query submission time, and generate language models based on tweets in the last day, week, and two weeks.

A comprehensive list of features can be seen in Table 4.1. Here, the first six features are extracted from the query log, the next four features from the query itself, and the rest of the features are probabilities of the query being generated by different language models. Language models built on tweets and using bigrams are denoted as `LM_Tweet_2`, and the ones using trigrams are `LM_Tweet_3`. The same applies for language models built on the query log, denoted as `LM_QL`.

4.2.1. Language Model Features

Language models are models that assign probabilities to sequences of words, or in our case, queries. We use language models to determine the probability of a query being

Table 4.1: List of features.

Feature index	Feature name
1	QuerySubmissions_LastDay
2	QuerySubmissions_LastWeek
3	QuerySubmissions_LastMonth
4	QuerySubmissions_Day/Week
5	QuerySubmissions_Day/Month
6	QuerySubmissions_Week/Month
7	Contains_News
8	Contains_Now
9	Contains_Numeral
10	NumberTokens
11	LM_Tweet_2_Day
12	LM_Tweet_2_Week
13	LM_Tweet_2_TwoWeeks
14	LM_Tweet_2_Day/Week
15	LM_Tweet_2_Day/TwoWeeks
16	LM_Tweet_2_Week/TwoWeeks
17	LM_Tweet_3_Day
18	LM_Tweet_3_Week
19	LM_Tweet_3_TwoWeeks
20	LM_Tweet_3_Day/Week
21	LM_Tweet_3_Day/TwoWeeks
22	LM_Tweet_3_Week/TwoWeeks
23	LM_QL_2_Day
24	LM_QL_2_Week
25	LM_QL_2_Month
26	LM_QL_2_Day/Week
27	LM_QL_2_Day/Month
28	LM_QL_2_Week/Month
29	LM_QL_3_Day
30	LM_QL_3_Week
31	LM_QL_3_Month
32	LM_QL_3_Day/Week
33	LM_QL_3_Day/Month
34	LM_QL_3_Week/Month

generated by a set of tweets or queries in our query log over different time periods.

Formally, an n-gram language model makes the assumption that the probability of a word only depends on the previous n words. Therefore, the probability $P(w_1, \ldots, w_m)$ of observing the sentence $w_1, \ldots, w_m$ is approximated as:

$$P(w_1, \ldots, w_m) = \prod_{i=1}^{m} P(w_i \mid w_1, \ldots, w_{i-1}) \approx \prod_{i=1}^{m} P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1})$$

The conditional probability can be calculated from n-gram model frequency counts:

$$P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1}) = \frac{\text{count}(w_{i-(n-1)}, \ldots, w_{i-1}, w_i)}{\text{count}(w_{i-(n-1)}, \ldots, w_{i-1})}$$

A common problem when using frequency counts directly is the sparsity of the data. In other words, many words will yield zero and the overall probability will be zero. To prevent this, a technique called *smoothing* is used to assign some probability to unseen words. Various techniques exist, the simplest one being *add-one smoothing*, or assigning a count of 1 to unseen words. The most commonly used technique is the interpolated Kneser-Ney algorithm (Kneser and Ney, 1995).

We use the KenLM Language Model Toolkit[2] to construct our language models. The toolkit uses linear probing for storing the frequency counts and the Kneser-Ney algorithm for smoothing. The toolkit is decribed in detail in (Heafield, 2011) and (Heafield et al., 2013). The usage is straightforward and consists of two steps: estimating (building) the language models, and querying them (producing probabilities for queries).

An example for a bigram language model is shown in Listing 4.2. First, we estimate a language model by providing a corpus (in our case either tweets or queries in our query log), and the order of the model (in this case, two for bigrams). Next, we convert the produced ARPA file to a binary format to reduce loading time. Finally, we query the model by providing the queries as input, and get probabilities of the queries being generated by that language model as the output.

Listing 4.2: Estimating a bigram language model and producing predictions for queries.

```
kenlm/bin/lmplz -o 2 < INPUT_CORPUS > model.2.arpa
kenlm/bin/build_binary model.2.arpa model.2.binary
kenlm/query -v sentence model.2.arpa < INPUT_QUERIES >
    OUTPUT_PROBABILITIES
```

[2]https://kheafield.com/code/kenlm/.

Query log corpus

We estimate language models based on our query log by first extracting the corresponding corpus. For a date d, we create language models based on the queries in our query log from the previous day, week, and month. As explained earlier, our ground truth consists of queries submitted in the period of July 2017 – September 2017. For each date d from this time period, we model three different time slots t, and two different types of language models, bigrams and trigrams, which amounts to six language models per date d.

Inspired by Dong et al. (2010a), our idea is to compare the probabilities for the input query to be generated by language models $LM_{Q,t}$, where $t \in \{\text{prev\_day}, \text{prev\_week}, \text{prev\_month}\}$. If there is a spike in interest for a given query, resulting in more traffic and, consequently, higher need for recent results, we expect the probability of $LM_{Q,\text{prev\_day}}$ to be higher than $LM_{Q,\text{prev\_week}}$ and especially $LM_{Q,\text{prev\_month}}$. Features with indices 23–34 in Table 4.1 model the probabilities of different $LM_{Q,t}$, as well as their ratios.

Twitter corpus

Similar to the query log corpus, our idea is to extract a corpus of tweets from different time slots, construct language models, and model the probabilities for the input query to be generated by these language models. In this case, we build language models $LM_{T,t}$, where $t \in \{\text{prev\_day}, \text{prev\_week}, \text{prev\_two\_weeks}\}$. Features with indices 11–22 in Table 4.1 model the probabilities of different $LM_{T,t}$, as well as their ratios.

For the queries in our ground truth, collecting the tweets is simple since we are able to do it in an offline manner. More specifically, we download a precompiled archive of tweets produced by The Internet Archive,[3] a nonprofit digital library. They provide monthly archives of tweets, sampled from the general Twitter stream, as a simple collection of JSON objects.

In an online setting, where we cannot download a precompiled archive, we use a Python library called `twarc`[4] to grab fresh tweets. The library queries the Twitter API, handles Twitter API's rate limits, and stores the retrieved tweets as JSON objects. Before querying the API, we register our developer application with Twitter to use the Standard search API,[5] which means we are able to retrieve only the past seven days of tweets.

[3]`https://archive.org/about/`.

[4]`https://github.com/docnow/twarc`.

[5]`https://developer.twitter.com/en/docs/tweets/search/api-reference/get-search-tweets`.

We contributed to this library by adding the parameter `until`, which is a date specifying the cut-off date for tweet creation date. This option was needed because `twarc` downloads tweets for a specific date until exhaustion, which takes too long, so we decide to run the tweet retrieval separately for each day in the past week. Moreover, the library does not support stopping after retrieving a certain number of tweets, so we used the `timeout.sh` script written by Anthony Thyssen[6] to terminate the process after a certain time elapsed. An example of collecting tweets from a specific date is shown in Listing 4.3.

Listing 4.3: Downloading a collection of tweets for a specific date using `twarc`.

```
twarc configure
./timeout.sh 14400 twarc search 'a OR b OR c OR d OR e OR f OR g OR h
    OR i OR j OR k OR l OR m OR n OR o OR p OR q OR r OR s OR t OR u OR
    v OR w OR x OR y OR z' --lang en --until 2018-06-13 --log
    2018-06-12.log > tweets-2018-06-12.json
```

Since the library does not support searching all tweets without a search query, we search for any tweet consisting of any of the characters from the English alphabet. Next, the parameter for timing out is given in seconds, and 14400 seconds is equivalent to 4 hours, which is the time it takes to download $\sim 300K$ tweets created in a given day. The parameters for `twarc` are `log`, the path to the log file, and `lang` specifying the language of tweets to retrieve. In this work, we only focus on English.

4.2.2. Other Features

Other features are represented with indices 1–10 in Table 4.1. The first six are extracted from the query log, and represent the number of query submissions in the past day, week, and month, as well as their ratios. Intuitively, we can imagine that queries related to new events would have a spike in popularity. Moreover, boolean features 7–9 check if the query contains the word *news*, *now*, or a numeral. The last feature in this group is the number of tokens in the query.

4.3. Model Training

We use XGBoost (eXtreme Gradient Boosting), introduced by Chen and Guestrin (2016), an open-source parallel gradient boosting library to train and evaluate the model. Even though our ground truth labels are four classes with values ranging from

[6]`http://www.ict.griffith.edu.au/anthony/software/#timeout`.

0 to 0.95, we formulate this as a regression problem and train a model to predict a query recency sensitivity score between 0 and 1. To this end, we build Gradient Boosted Regression Trees (GBRT).

We have a total of 34 features, where each feature value is numerical. The training instances are of the LibSVM data format:[7]

```
<label> <index_1>:<value_1> ...  <index_N>:<value_N>
```

Here, <label> is the target value, a number between 0 and 1, and the rest of the fields model the feature index and value, respectively.

To determine which features are more useful than others, we plot the feature importance scores, as seen in Figure 4.3. Cross-referencing with Table 4.1, we can see that the feature with index 16, LM_Tweet_2_Week/Month is the most useful one. This confirms our intuition that the higher the ratio of probabilities of the more recent language model and the older the language model is, the more recency sensitive the query is. The other most important features are LM_Tweet_2_Day, LM_Tweet_2_Week, LM_QL_2_Day/Week, LM_Tweet_3_TwoWeeks, LM_Tweet_3_Week and LM_Tweet_3_Week/TwoWeeks. This indicates that language models are strong features, both those consisting of bigrams and trigrams.

On the other end, the worst performing features are boolean features Contains_Now and Contains_Numeral. It was expected that the boolean features would not have a great impact, as they are cheap to compute and very simple. Following these, the other worst performing features come from the query log, QuerySubmissions_Day/Week and QuerySubmissions_Day/Month. This may be because our query log is too sparse to provide meaningful features.

We plot the first tree created in the model to gain insight into the decision process. The tree is shown in Figure 4.4. We can see that the first decision is based on feature 10, but as the features are 0-indexed in the implementation, this is actually feature with index 11 from Table 4.1, LM_Tweet_2_Day. The next level is based on features with indices 1 and 2, QuerySubmissions_LastDay and QuerySubmissions_LastWeek. Arriving to the first leaf of the tree, the model concludes that if the probability of the language model based on bigrams from tweets from the past day is lower than a certain threshold, and the query was submitted less than 8 times in the past week, the resulting contribution to the overall score is 0.002, a very small value.

[7] https://www.csie.ntu.edu.tw/~cjlin/libsvm/.

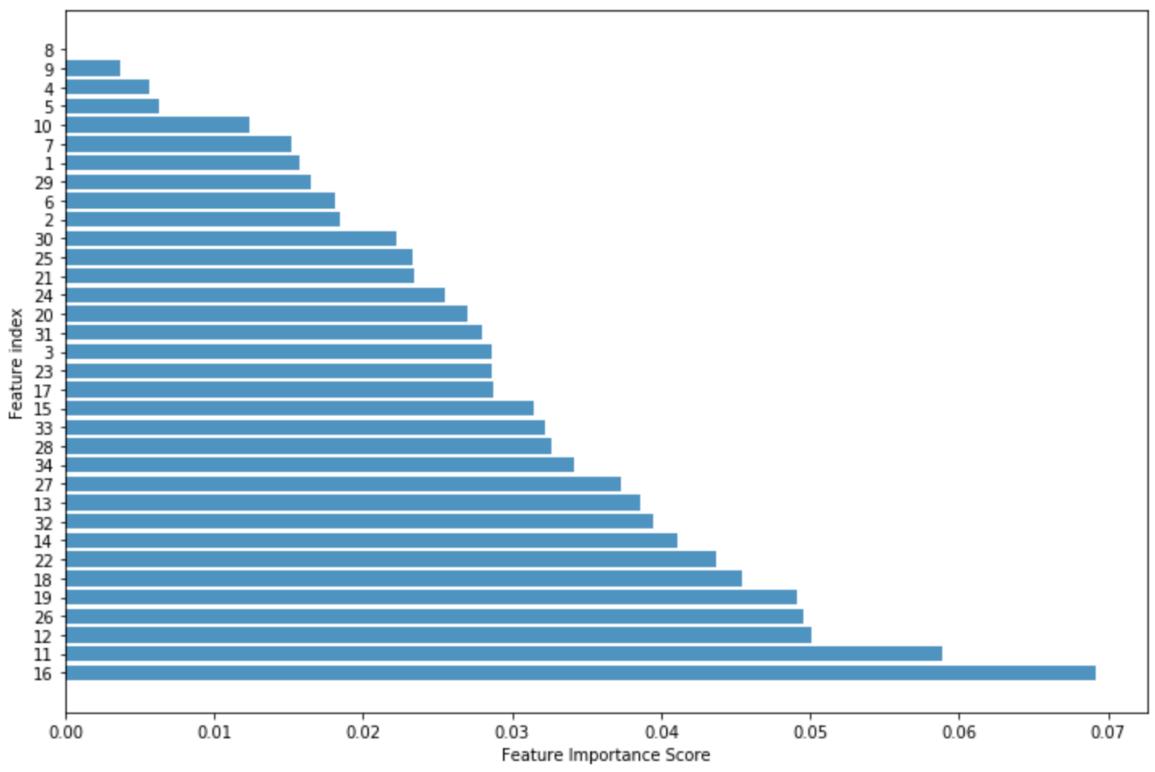


Figure 4.3: Feature importance scores for the query recency sensitivity model.

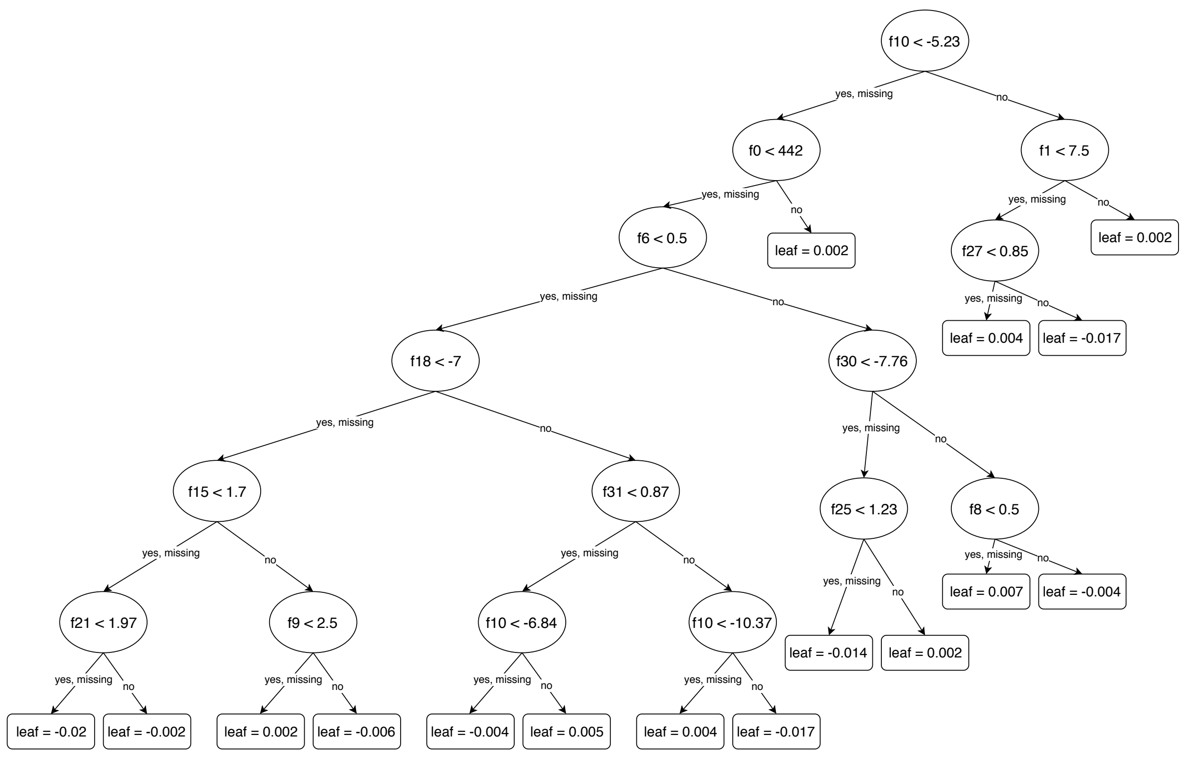


Figure 4.4: First tree of the query recency sensitivity model.

4.4. Evaluation

For training the model, we split the input set into train and validation sets in the 70:30 ratio. We then proceed to grid search the hyper-parameters of the model using a 5-fold cross-validation. Listing 4.4 shows which hyper-parameters are tuned.

Listing 4.4: Hyper-parameter tuning for the query recency sensitivity model.

```
'max_depth': [4,6,8,10], (best = 6)
'min_child_weight': [0,1,2], (best = 0)
'gamma': [0.0, 0.1, 0.2, 0.3], (best = 0.0)
'subsample': [0.6, 0.8, 1.0], (best = 0.6)
'colsample_bytree': [0.6, 0.8, 1.0], (best = 0.6)
'learning_rate': [0.01, 0.03, 0.1, 0.2, 0.3], (best = 0.01)
'n_estimators': [100,500,1000,5000] (best = 500)
```

The hyper-parameters are explained in detail earlier in the thesis, in Section 3.4. We also set the model to stop learning if there are not any improvements after 50 rounds.

The training of the model took 28 hours on a machine with CentOS 7.5.1804, 16 GB of RAM, and 8 Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz CPUs. The model was trained by minimizing the root mean squared error (RMSE) of the difference between the predicted query recency sensitivity score and the ground truth label (possible values: `0.0, 0.25, 0.75, 0.95`).

Table 4.2: Query recency sensitivity model evaluation.

RMSE train	RMSE test
0.0845	0.1077

The results on the train and test sets are shown in Table 4.2. Here, we report three queries with the highest predicted recency sensitivity score. The queries are taken from our web ranking ground truth collection, consisting of query-document pairs. The queries are:

1. *youtube*, submitted at `2017-08-20`. We explain this by looking at the snippets of the retrieved results. Most of the top ten results were less than a day old.

2. *daca news*, submitted at `2017-09-05`. This was a very hot topic in early September, and most retrieved results were news published on the same day.

3. *utah*, submitted at `2017-08-16`. There was a spike in this query's popularity due to wildfires in Utah.

To gain additional insight into how our model is performing, we plot two different learning curves. The motivation for these learning curves is explained in detail earlier, in Section 3.4. The first learning curve, shown in Figure 4.5, shows the model performance based on the number of training rounds. This graph can indicate if we can benefit from adding more training instances. In our case, we did not manually annotate data, so adding more instances should not be a problem. By looking at the validation error, we can see that the model's ability to generalize on unseen data plateaus at around 800 instances.

The second learning curve is shown in Figure 4.6. Here, we examine the model's performance based on the number of training rounds. We notice that both the training and validation error reduce rapidly. Nevertheless, the validation error does not reduce significantly after 300 rounds, making it a valid cut-off candidate.

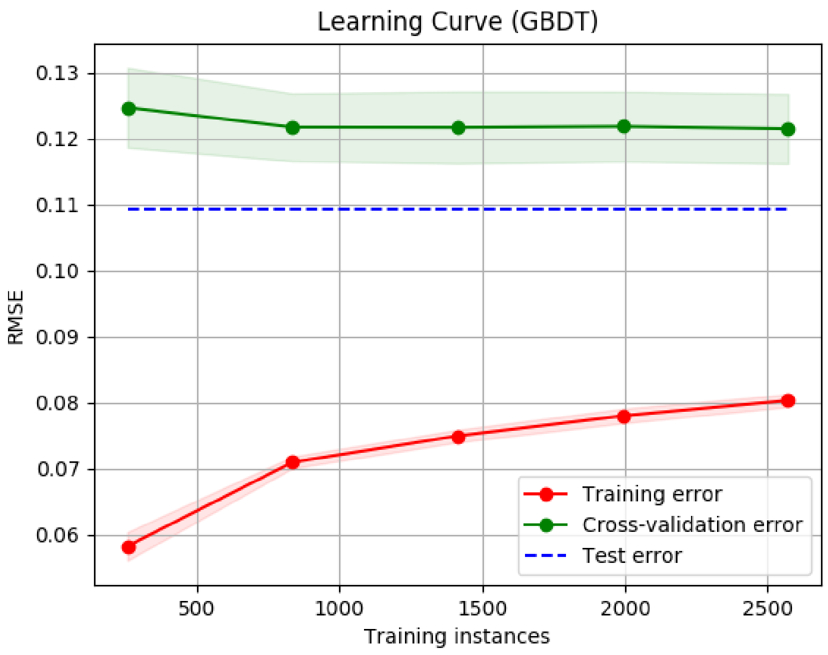


Figure 4.5: Learning curve of the query recency sensitivity model based on the number of training instances.

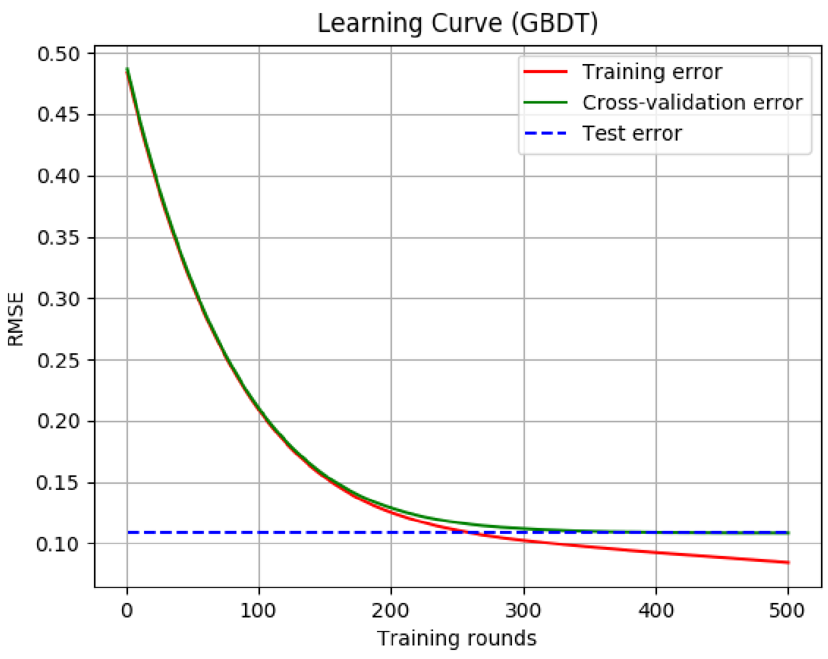


Figure 4.6: Learning curve of the query recency sensitivity model based on the number of training rounds.

5. Incorporating Recency in Web Search

Currently, the commercial search engine ranks documents according to their relevance to the query. As mentioned earlier, in this work we are upgrading an existing commercial search engine prototype to support recency as well. To further improve the quality of the search engine, we introduce the recency component on the query and the document side. The rest of this chapter first outlines the architecture of the commercial search engine, then describes in detail how the two machine-learning models explained in Chapters 3 and 4 are integrated.

5.1. Search Engine Architecture

The architecture of a search engine begins with crawling documents from the Web and ends with retrieving and ranking a subset of documents that best match a given search query. This can be done in multiple stages, and we identify the main stages as crawling, document selection, indexing, and ranking. Furthermore, the stages can be divided into two different groups: offline and online. An offline stage is query-independent and can be done in an offline manner, i.e., once in the beginning and later only when needed. On the other hand, online stages are query-dependent and are run any time a query is submitted. A high-level overview of ranking stages of the commercial search engine is shown in Figure 5.1.

The indexing and ranking stages are supported by Elasticsearch.[1] Described in (Gormley and Tong, 2015), Elasticsearch is a distributed, scalable, real-time search and analytics engine. It is built on top of Apache Lucene,[2] a search-engine library written in Java. Elasticsearch uses Lucene internally for indexing and searching, but extends the usability by exposing an easy-to-use RESTful API. The indexing and ranking stages are explained in more detail in their respective sections.

[1]https://www.elastic.co/products/elasticsearch.

[2]https://lucene.apache.org/core/.

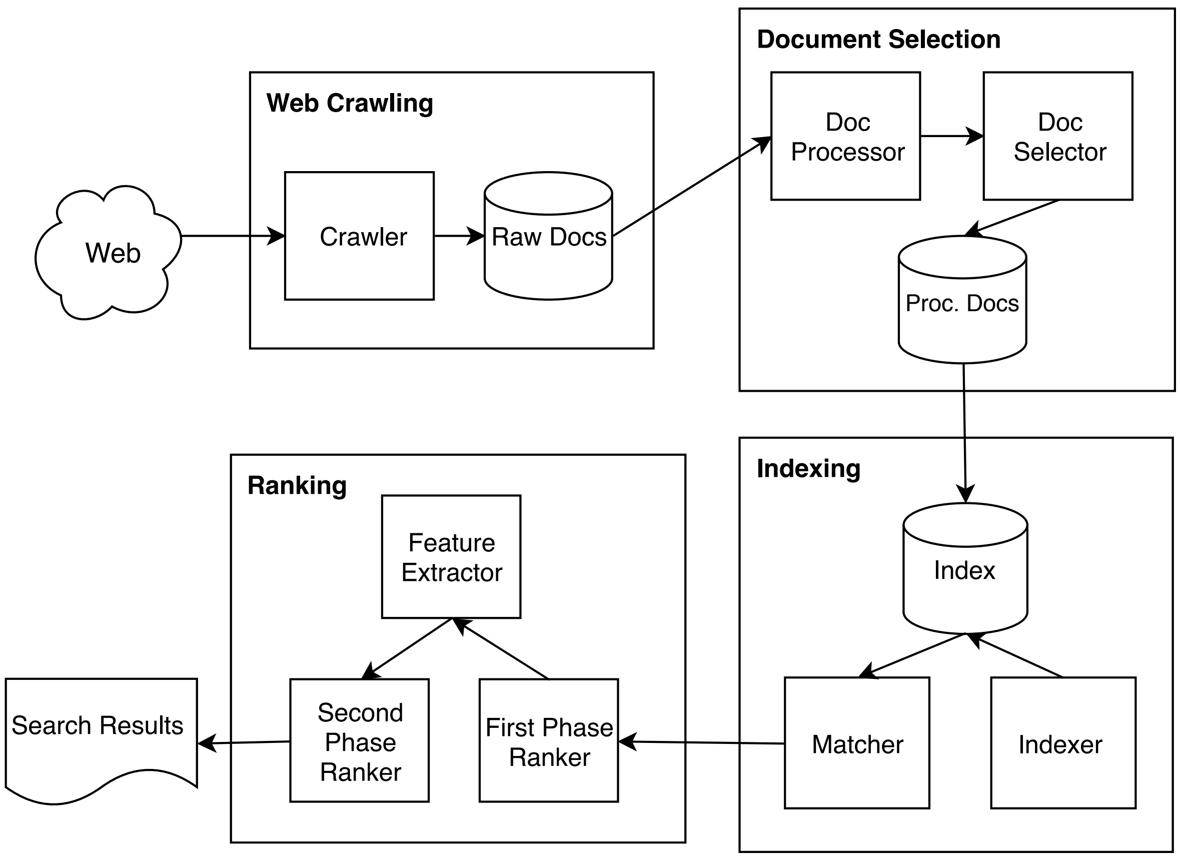


Figure 5.1: High-level architecture of the commercial search engine.

5.1.1. Web Crawling

A crawler (or spider, robot, etc.) is a program that traverses the Web and downloads raw documents to a database. The two main tasks of a crawler are discovery and download of new pages and refreshing of the existing pages. The simplest approach is to start with a seed set of URLs, download the pages the URLs point to, then scan these pages for new links, add them to the queue of pages to be crawled, and repeat the process.

The crawler is an important component of a search engine, especially with regard to recency. For example, if there has been a breaking event and users search for it, if the crawler still has not crawled the relevant documents, they cannot be shown to the user. Therefore, a good crawler must download content fast enough so it does not become the bottleneck of the entire process.

5.1.2. Document Selection

The crawler can download a very large and diverse amount of content from the Web. However, not all of that content is useful. For example, we do not want to waste resources processing spam content. Therefore, the document selection stage filters out unnecessary documents, and stores the rest to be indexed. This stage consists of two modules: `Document Processor` and `Document Selector`.

`Document Processor` is in charge of boilerplate removal and feature extraction. Boilerplate removal implies removing boilerplate parts from the Web page content (e.g., menus, footer, side bar) and unrelated parts (e.g., adverts, related articles). Consequently, we are left with a "clean" HTML, and we can extract features from the HTML and URL of the page, such as the number of tags and characters in the HTML, number of links, length of the link, etc.

`Document Selector` performs filtering of unwanted documents. It is a machine-learned regression model using Gradient Boosted Decision Trees (GBDT) to predict a document score. The features used are extracted from the HTML and the URL of the document. If the score of a document is below a certain threshold, it is discarded. Otherwise, the document is passed to the indexing stage. The scores are stored in a database, and the low-quality documents are removed periodically from the `Raw Docs` database.

5.1.3. Indexing

After a document is processed and stored in the `Processed Docs` database, it is sent to the indexing stage. Generally speaking, an index is a data structure that provides efficient retrieval for a given query. As mentioned earlier, this stage is handled by Elasticsearch and is run on an Elasticsearch cluster.

In Elasticsearch, documents are stored in an index as JSON objects which contain zero or more fields, or key-value pairs. In the `Document Selection` stage, `Document Processor` extracted features from the document content and its URL. These features are injected into the Elasticsearch index as key-value pairs in the document JSON.

An example of a simple keyword search query and a response from an index in Elasticsearch is shown in Listing 5.1. Additionally, Elasticsearch provides a Query DSL (Domain Specific Language) based on JSON to define queries, which is the way we query our indexes. An example of such query is shown in Listing 5.2.

Listing 5.1: Example of a keyword search query and response in Elasticsearch.

```
curl -XGET localhost:9200/books/_search?q=elasticsearch
{
  "took" :2,"timed_out" :false,
  "_shards" :{"total" :5,"successful" :5,"failed" :0},
  "hits" :{
    "total" :1,"max_score" :0.076713204,
    "hits" :[{
      "_index" :"books", "\_type" :"book", "\_id" :"1",
      "_score" :0.076713204, "_source" :{
        "title" :"Elasticsearch - The definitive guide",
        "authors" :["Clinton Gormley", "Zachary Tong" ],
        "started" :"2013-02-04", "pages" :230
      }
    }]
  }
}
```

Listing 5.2: Example of a search query specified in DSL in Elasticsearch.

```
curl -XGET 'localhost:9200/books/book/\_search' -d '{
  "query": {
    "filtered" :{
      "query" :{
        "match": {
          "text" :{
            "query" :"To Be Or Not To Be",
            "cutoff_frequency" :0.01
          }
        }
      },
      "filter" :{
        "range": {
          "price": {
            "gte": 20.0
            "lte": 50.0
   ...
  }
}'
```

The underlying data structure Elasticsearch uses for its index is the inverted index, designed to allow fast full-text searches. An inverted index consists of a list of all the unique words that appear in any document, and for each word, a list of the documents in which it appears. For example, if we have two documents D_1 = *Madrid is the capital of Spain* and D_2 = *Barcelona is the capital of Catalonia*. To create an inverted index, we first split the contents of each document into separate words, or tokens, normalize

Table 5.1: A simple example of an inverted index.

Token	D_1	D_2
barcelona	-	x
capital	x	x
catalonia	-	x
is	x	x
madrid	x	-
of	x	x
spain	x	-
the	x	x

them, create a sorted list of all the unique tokens, and then list in which document each term appears. An example of such inverted index is shown in Table 5.1. If we want to search for *capital spain*, we just need to find the documents in which each tokens appears, in this case the count is 2 for D_1, and 1 for D_2, so document D_1 seems more relevant for this query.

The indexing phase is finished when the document has been preprocessed, and the features have been extracted and stored as fields in the JSON object representing it.

5.1.4. Ranking

After the document has been indexed, once a user query is submitted, the `Matcher` matches the documents for the query. The matching is done by performing a logical conjunction (operator `AND`) between all of the terms from the query and the content of the document. Once we identify matching documents for a query, we can score them and rank them.

The ranking stage takes care of computing the relevance scores of the matching documents for the query and ranking them. Since there could be potentially millions of matching documents, computing all of the scores can be quite expensive. Therefore, we resort to a two-stage ranking architecture, where the first phase is essentially a cheap, but effective filter. The idea is to use query-independent features in the first phase ranking, because they can be precomputed in an offline manner and accessed by a simple lookup at query processing time. The first phase ranking function assigns scores to all of the matching documents N, and we only take the top M ($M << N$) for the next stage. Since we now have a substantially smaller subset of the initial matching documents, we can extract more sophisticated and computationally expensive features.

Finally, the second phase ranking function produces the ranking of the documents that are shown to the user.

We use and customize the Elasticsearch Learning to Rank plugin[3] for storing features, training datasets, models, and ranking the documents. We use the RankLib library[4] to train the ranking models. Each model is described in its respective section as follows.

First Phase Ranking

In the first phase, we extract a combination of query-dependent and query-independent features. As described by Cambazoglu et al. (2010), the first phase is used for selecting a small subset of potentially relevant documents from the entire collection. The typical features in this stage are:

– `DocScore:` The score of the document produced in the Document Selection stage by the `Document Selector`. This score is also used as an indicator if we should index a document or discard it;

– `BM25(query, content):` The score of the $BM25$ function between the query and the content of the document.

The BM25 function (Robertson et al., 1996) is a ranking function that can be computed automatically, without needing any relevance information. Given a query Q containing keywords $q_1, ..., q_n$, the BM25 score of a document D is:

$$\text{BM25}(Q, D) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)}$$

Here, $f(q_i, D)$ is q_i's term frequency in the document D, $|D|$ is the length of the document D in words, and $avgdl$ is the average document length in the text collection from which documents are drawn. Moreover, k_1 and b are free parameters, and their default value in Elasticsearch is $k_1 = 1.2$ and $b = 0.75$. $\text{IDF}(q_i)$ is the inverse document frequency weight of the query term q_i. It is usually computed as:

$$\text{IDF}(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5}$$

Here, N is the total number of documents in the collection, and $n(q_i)$ is the number of documents containing q_i.

Finally, we rank all the matching documents according to their first phase score, and get the top M results, where $M << N$.

[3]`https://github.com/o19s/elasticsearch-learning-to-rank`.

[4]`https://sourceforge.net/p/lemur/wiki/RankLib/`.

Second Phase Ranking

Contrary to the scoring function in the first phase, the second phase function is more complex. Since we only have a subset of documents to rank, we can afford extracting more computationally heavy features and training a more sophisticated model. The ranking produced by this model is the final ranking shown to the user. The model uses hundreds of features. Examples of different feature types are listed as follows:

– `Query string.` Number of words, boolean features, etc.

– `URL string.` URL length, slash count, etc.

– `Document content.` Number of pictures, different tags, etc.

– `Content classification.` Output of classifiers of document content for spam, malware, adult content, etc.

– `Content-query relevance.` BM25 scores for query-title, query-URL, query-body, etc.

– `Content-query similarity.` Jaccard similarity or edit distance between the query and title, URL, etc.

– `Web graph.` PageRank, TrustRank, indegree, outdegree, etc.

– `Wikipedia.` Wikipedia page view count, references from Wikipedia, etc.

– `Traffic data.` The Alexa Traffic rank.[5]

– `Document score.` Computed by the `Document Selector` model, trained using query-independent features.

– `First phase score.` Computed according to the ranking model in the first phase.

The offline features (query-independent) are computed and injected into the Elasticsearch index as document fields. The online features (query-dependent) are computed at query processing time using our customized Elasticsearch ranking plugin.

In this work, we introduce a type of features called `Recency features`, consisting of predicted document age and predicted query recency sensitivity. The document age feature is computed offline and injected into the ES index, whereas the query recency sensitivity feature should be computed at query processing time.

[5] `https://www.alexa.com/siteinfo`.

5.2. Offline Experiments and Results

In this section, we explain how we integrate the models explained in Chapters 3 and 4 with the existing multi-stage ranking architecture. Our goal is to investigate whether the addition of recency features improves the overall ranking.

Both recency models, document age prediction and query recency sensitivity, are developed in a Hadoop cluster, and the extracted features and predictions are stored in HDFS. Apache Hadoop[6] is a free, open source, Java-based programming framework for distributed processing of large data sets across clusters of computers. Hadoop provides its own file system, called Hadoop Distributed File System (HDFS). Moreover, Hadoop handles job scheduling and cluster resource management, as well as parallel processing of large data sets. Both models run in Spark,[7] a fast and general compute engine for Hadoop data. The query recency sensitivity model is written in Python, using PySpark,[8] the Spark Python API. The document age prediction model is written in Scala, which is supported natively in Spark.

5.2.1. Model Integration

Figure 5.2 shows the integration of the document age prediction model into the existing ranking architecture. As described in Chapter 3, the ground truth is obtained by querying Memgator, an external web service. The script to build the ground truth, and the script to extract features are both written in Scala, run in Spark on the Hadoop cluster, and save the output data to HDFS.

The document age model training module is written in Python to make use of XGBoost's Python API in scikit-learn.[9] We also run it on the Hadoop cluster, and save the predicted values to HDFS. Next, the Elasticsearch Feature Injector runs in Spark to read the predicted document last update time and inject it to Elasticsearch's index. Finally, we extend the Elasticsearch ranking plugin to calculate the document age as $currentTime - documentLastUpdateTime$ when queried (during the ranking models' training time).

Figure 5.3 shows the integration of the query recency sensitivity model into the existing ranking architecture. This model is described in detail in Chapter 4. It is more complicated than the previous model in terms of external calls, because the corpora needed to build the language models is obtained from external sources such as Twitter and the query log. The ground truth is created by a PySpark script run in Spark on

[6]`http://hadoop.apache.org/`.

[7]`http://spark.apache.org/`.

[8]`https://spark.apache.org/docs/0.9.0/python-programming-guide.html`

[9]`http://scikit-learn.org/stable/`.

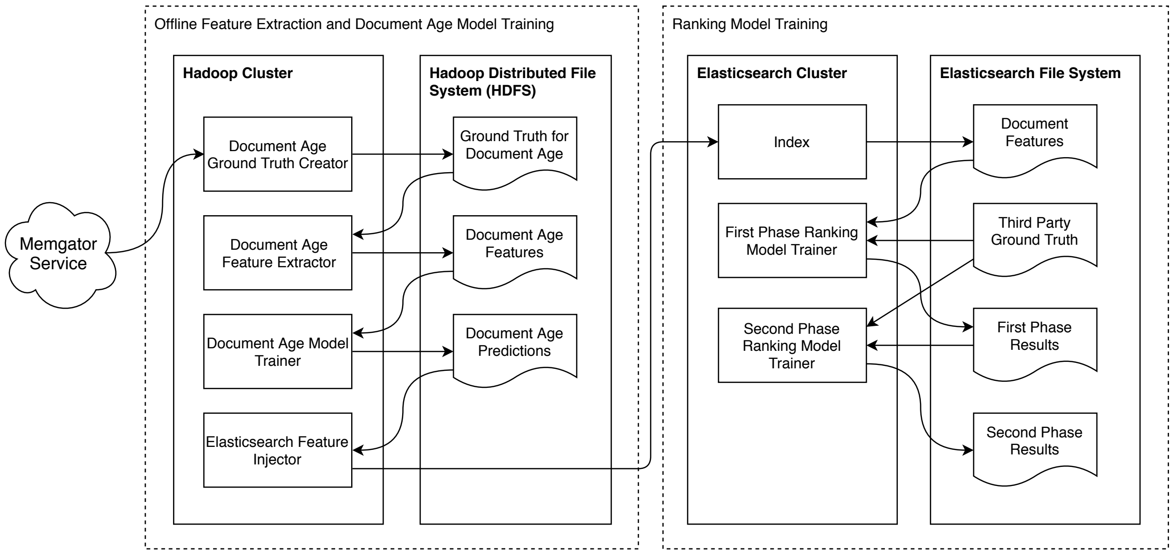

Offline Feature Extraction and Document Age Model Training
Ranking Model Training
Hadoop Cluster
Hadoop Distributed File System (HDFS)
Elasticsearch Cluster
Elasticsearch File System
Memgator Service
Document Age Ground Truth Creator
Document Age Feature Extractor
Document Age Model Trainer
Elasticsearch Feature Injector
Ground Truth for Document Age
Document Age Features
Document Age Predictions
Index
First Phase Ranking Model Trainer
Second Phase Ranking Model Trainer
Document Features
Third Party Ground Truth
First Phase Results
Second Phase Results

Figure 5.2: Integration of the document age prediction model into the existing ranking pipeline.

the Hadoop cluster that automatically labels queries according to the retrieved web snippets, and stores the output to HDFS.

The module that mines the Twitter corpus is a combination of Bash scripts (to retrieve the tweets using `twarc` and to extract the data) and PySpark (to filter and process the data, and save it to HDFS). The module that mines the query log corpus is a PySpark script that runs in Spark on the Hadoop cluster and saves the output to HDFS. The module that creates language models from both types of corpora is a Bash script which saves the resulting language models to HDFS. Next, the feature extractor is a combination of Bash scripts (to download the language models locally and to predict the language model feature type values) and PySpark (to extract the other feature type values).

Next, we need to inject the predictions of the query recency sensitivity model to the existing ranking pipeline. Since these predictions are related to the query itself, not the document, they cannot just be injected to the document index. Therefore, we inject them to our third party ground truth file used for training the ranking models. This is done by a Bash script that runs on the Elasticsearch cluster. The ground truth file consists of query-document pairs, so we simply append the query recency sensitivity score to each query-document pair. Finally, the recency sensitivity score is used as a feature in both the first- and second-phase ranking models.

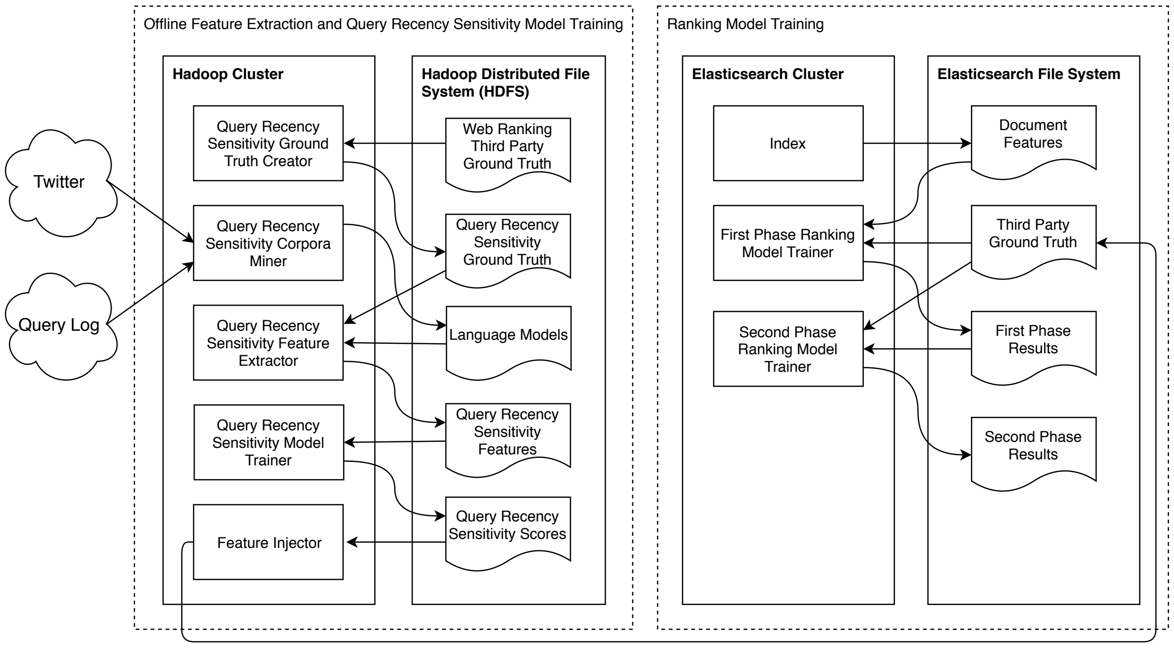

Offline Feature Extraction and Query Recency Sensitivity Model Training
Ranking Model Training
Twitter
Query Log
Hadoop Cluster
Query Recency Sensitivity Ground Truth Creator
Query Recency Sensitivity Corpora Miner
Query Recency Sensitivity Feature Extractor
Query Recency Sensitivity Model Trainer
Feature Injector
Hadoop Distributed File System (HDFS)
Web Ranking Third Party Ground Truth
Query Recency Sensitivity Ground Truth
Language Models
Query Recency Sensitivity Features
Query Recency Sensitivity Scores
Elasticsearch Cluster
Index
First Phase Ranking Model Trainer
Second Phase Ranking Model Trainer
Elasticsearch File System
Document Features
Third Party Ground Truth
First Phase Results
Second Phase Results

Figure 5.3: Integration of the query recency sensitivity model into the existing ranking pipeline.

5.2.2. Recency Ranking Results

The initial research question we asked ourselves is whether introducing recency features boosts the overall ranking performance. Since we are interested in the end result, or the quality of the search results shown to the user, we report the performance of the second-phase ranking model (which, in turn, also depends on the first-phase model, since its output is used as a feature).

Table 5.3 shows the evaluation of different ranking models. The evaluation metric used is NDCG@10 with respect to a third party search engine. The first model, denoted as F, is the model without any recency features. When adding just the document age feature (model F+1), we observe an increase in NDCG@10 of 0.35%. This means that the model is now able to capture recency on the document side, but still not on the query side. In other words, it might learn to favour more recent documents in general, irrespective of the query. Furthermore, we add the query recency sensitivity score (model F+2), an observe an total increase in NDCG@10 of 0.48%. Now, the model is able to learn to what extent to favour recent documents depending on the query. For example, if the query recency sensitivity score is low, there is no need to boost more recent documents. On the other hand, if the query recency sensitivity score is high, the ranking should be different than the one so far, without taking recency into account. In other words, more recent documents should be favoured.

Overall, we have introduced only two new features to the existing hundreds of

Table 5.2: Evaluation of the recency feature importances in the second-phase ranking model.

Model	Feature name	Normalized feature rank	First appeared tree
F+1	documentAge	0.72	541 (out of 2460)
F+2	documentAge	0.72	541 (out of 2937)
F+2	queryRecencySensitivity	0.73	660 (out of 2937)

Table 5.3: Evaluation of the second-phase ranking model. Model F does not contain recency features, model F+1 contains the document age feature, and model F+2 contains both the document age feature and the query recency sensitivity score.

Model	Number of trees	Improvement of NDCG@10
F	1657	-
F+1	2460	0.35%
F+2	2937	**0.48%**

features, and gained 0.48% improvement in NDCG@10. Moreover, we evaluated the feature importances, shown in Table 5.2. We can see that the model picks both features pretty early, as indicated by the rank of the tree they first appeared in. Therefore, the features are significant to the model's increase in performance.

5.3. Online Integration and Future Work

The previous section describes how we introduce the two recency features to the existing ranking models in an offline fashion. In other words, in that setup we are only training and evaluating the models. In an online, production setting, we want to be able to calculate these features on the fly. More specifically, when a user submits a query to the search engine, the document age model and the query recency sensitivity model must output a prediction.

Since the document age prediction model is not query-dependent, the online prediction involves calling the function in our Elasticsearch ranking plugin to calculate the document age with respect to the query submission time and the already injected document last update time, which is already supported.

However, for the online prediction of the query recency sensitivity model, we should already have the language models constructed and ready to use. This means we must already have the Twitter and query log corpora in place. As a reminder, we build language models based on the last day, week, and last two weeks (in case of tweets)

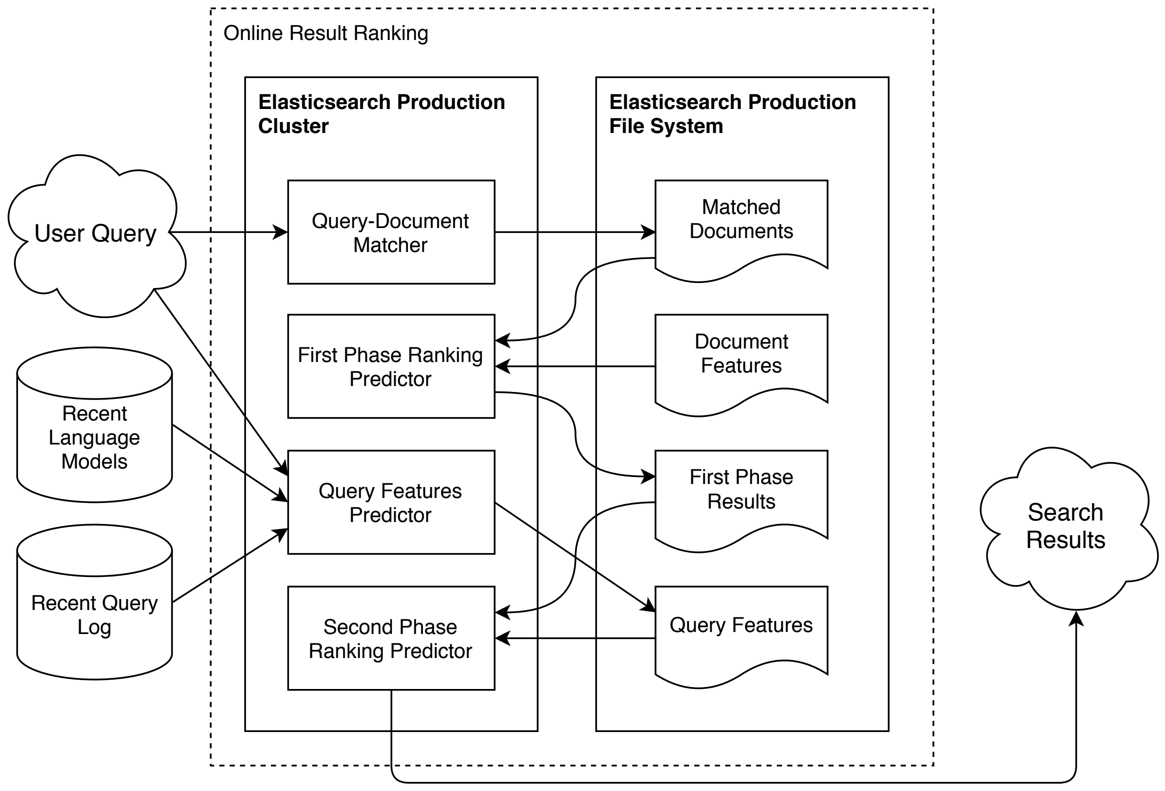


Figure 5.4: End-to-end pipeline for online result ranking.

or last month (in case of the query log). So, one challenge is to keep the language models fresh by updating the corpora they need daily. Another challenge, which has not been solved at to the time of writing, is to calculate query-dependent features in Elasticsearch which involve a call to an external store where the features would be located. This is a limitation of the ranking plugin we are using. Nevertheless, we are certain this feature will be supported soon, and the integration of it remains as future work. An example of an end-to-end online pipeline is shown in Figure 5.4.

6. Conclusion

Combining relevance and recency in Web search is an open problem. There are numerous approaches to this ranking problem, and existing solutions mostly focus on only one type of queries or documents. In this work, we introduce recency ranking for queries and documents on the Web in general, while preserving the existing relevance ranking.

We presented a novel combination of machine-learned models to predict recency, both on the query and document side. We propose a query recency sensitivity classifier to determine the need for recent documents, and a document age prediction model to determine how recent a document is.

We automatically labeled both ground truth datasets to avoid costly and time-consuming human annotation. We extracted features from several different sources. We recognized the challenges in keeping the feature values up to date and provided a solution on how to implement this in production. We trained two Gradient Boosted Regression Trees (GBRT) models and integrated them as features in an existing multi-stage ranking architecture.

Having added only two recency features, we managed to improve the NDCG@10 of our ranking model by 0.5%. We consider this a very good improvement, given that the model is already rich in strong features. Moreover, we did not introduce significant increase in scoring time. Finally, the introduced features are not exact, but predicted, which means they are open to further improvement.

Our future work consists of improving the query recency sensitivity classifier and the document age prediction model. For the query classifier, introducing language models built from other sources such as news articles might be beneficial. For the document classifier, we are currently extracting only content-based features, whereas we could also make use of, for example, link-based features.

Bibliography

Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

Klaus Berberich, Srikanta Bedathur, Omar Alonso, and Gerhard Weikum. A language modeling approach for temporal information needs. In *European Conference on Information Retrieval*, pages 13–25. Springer, 2010.

Leo Breiman. Arcing the edge. Technical report, Technical Report 486, Statistics Department, University of California at Berkeley, 1997.

Andrei Broder. A taxonomy of web search. In *ACM Sigir forum*, volume 36, pages 3–10. ACM, 2002.

Andrei Z Broder. On the resemblance and containment of documents. In *Compression and complexity of sequences 1997. proceedings*, pages 21–29. IEEE, 1997.

B Barla Cambazoglu, Hugo Zaragoza, Olivier Chapelle, Jiang Chen, Ciya Liao, Zhaohui Zheng, and Jon Degenhardt. Early exit optimizations for additive machine learned ranking systems. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 411–420. ACM, 2010.

Ricardo Campos, Gaël Dias, Alípio Jorge, and Célia Nunes. Gte-rank: A time-aware search engine to answer time-sensitive queries. *Information Processing & Management*, 52(2):273–298, 2016.

David Carmel, Liane Lewin-Eytan, Alex Libov, Yoelle Maarek, and Ariel Raviv. Promoting relevant results in time-ranked mail search. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1551–1559. International World Wide Web Conferences Steering Committee, 2017.

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

Shiwen Cheng, Anastasios Arvanitis, and Vagelis Hristidis. How fresh do you want your search results? In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 1271–1280. ACM, 2013.

Na Dai, Milad Shokouhi, and Brian D Davison. Learning to rank for freshness and relevance. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 95–104. ACM, 2011.

Wisam Dakka, Luis Gravano, and Panagiotis Ipeirotis. Answering general time-sensitive queries. *IEEE Transactions on Knowledge and Data Engineering*, 24(2): 220–235, 2012.

Anlei Dong, Yi Chang, Zhaohui Zheng, Gilad Mishne, Jing Bai, Ruiqiang Zhang, Karolina Buchner, Ciya Liao, and Fernando Diaz. Towards recency ranking in web search. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 11–20. ACM, 2010a.

Anlei Dong, Ruiqiang Zhang, Pranam Kolari, Jing Bai, Fernando Diaz, Yi Chang, Zhaohui Zheng, and Hongyuan Zha. Time is of the essence: improving recency ranking using twitter data. In *Proceedings of the 19th international conference on World wide web*, pages 331–340. ACM, 2010b.

Miles Efron. Query-specific recency ranking: Survival analysis for improved microblog retrieval. In *Proceedings of the 1st Workshop on Time-aware Information Access (# TAIA2012), TAIA*, volume 12. Citeseer, 2012.

Miles Efron and Gene Golovchinsky. Estimation methods for ranking recent information. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 495–504. ACM, 2011.

Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

Clinton Gormley and Zachary Tong. *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine.* " O'Reilly Media, Inc.", 2015.

Kenneth Heafield. KenLM: faster and smaller language model queries. In *Proceedings of the EMNLP 2011 Sixth Workshop on Statistical Machine Translation*, pages 187–197, Edinburgh, Scotland, United Kingdom, July 2011. URL `https://kheafield.com/papers/avenue/kenlm.pdf`.

Kenneth Heafield, Ivan Pouzyrevsky, Jonathan H. Clark, and Philipp Koehn. Scalable modified Kneser-Ney language model estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 690–696, Sofia, Bulgaria, August 2013. URL `https://kheafield.com/papers/edinburgh/estimate\_paper.pdf`.

Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

Rosie Jones and Fernando Diaz. Temporal profiles of queries. *ACM Transactions on Information Systems (TOIS)*, 25(3):14, 2007.

Reinhard Kneser and Hermann Ney. Improved backing-off for m-gram language modeling. In *icassp*, volume 1, page 181e4, 1995.

Damien Lefortier, Pavel Serdyukov, and Maarten De Rijke. Online exploration for detecting shifts in fresh intent. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 589–598. ACM, 2014.

Donald Metzler, Rosie Jones, Fuchun Peng, and Ruiqiang Zhang. Improving search relevance for implicitly temporal queries. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 700–701. ACM, 2009.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

Stephen E Robertson, Steve Walker, MM Beaulieu, Mike Gatford, and Alison Payne. Okapi at trec-4. *Nist Special Publication Sp*, pages 73–96, 1996.

Hany M SalahEldeen and Michael L Nelson. Carbon dating the web: estimating the age of web resources. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1075–1082. ACM, 2013.

Andreas Spitz, Jannik Strötgen, and Michael Gertz. Predicting document creation times in news citation networks. In *Companion of the The Web Conference 2018 on The Web Conference 2018*, pages 1731–1736. International World Wide Web Conferences Steering Committee, 2018.

Andrey Styskin, Fedor Romanenko, Fedor Vorobyev, and Pavel Serdyukov. Recency ranking by diversification of result set. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1949–1952. ACM, 2011.

Hongning Wang, Anlei Dong, Lihong Li, Yi Chang, and Evgeniy Gabrilovich. Joint relevance and freshness learning from clickthroughs for news search. In *Proceedings of the 21st international conference on World Wide Web*, pages 579–588. ACM, 2012.

Recency Ranking Models for Web Search

Abstract

Given the increase in the amount of accessible information on the Web, more attention has been drawn to information retrieval systems such as search engines. In web search, recency ranking refers to ranking documents by their relevance to the query, but also taking freshness into account. In this thesis, we propose two models for recency ranking. The first one is the query recency sensitivity model, and the second is a model to predict the publication time of documents. We extract temporal features from several sources and automatically construct ground truth datasets for both models. Furthermore, we integrate the models into an existing commercial search engine with a multi-stage ranking architecture. Our experiments demonstrate an improvement in the effectiveness of the commercial search engine.

Keywords: recency ranking, web search, search engine, machine learning, gradient boosted decision trees, information retrieval, big data

Modeli rangiranja po svježini za pretraživanje weba

Sažetak

Porastom raspoloživih količina dostupnih informacija na Webu, povećalo se zanimanje za sustavima koji dohvaćaju informacije, poput sustava za pretraživanje. Pri pretraživanju weba, rangiranje po svježini odnosi se na rangiranje dokumenata s obzirom na relevantnost na upit, a pritom uključujući i svježinu rezultata. U okviru diplomskog rada prezentiramo dva modela za rangiranje po svježini. Prvi model predviđa osjetljivost upita na svježinu rezultata, a drugi predviđa vrijeme objavljivanja dokumenata. Gradimo vremenske značajke na temelju više izvora i automatski gradimo skup podataka za vrednovanje oba modela. Nadalje, integriramo modele u postojeći komercijalni pretraživač Weba koji se sastoji od višefazne arhitekture za rangiranje. Naši eksperimenti ukazuju na poboljšanje korisnosti komercijalnog pretraživača Weba.

Ključne riječi: pretraživanje po svježini, pretraživanje Weba, tražilica, strojno učenje, stabla odluke potpomognuta gradijentom, dohvat informacija, veliki skupovi podataka