BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Learning to Rank Places

vorgelegt von

Shuaib Yunus

EDV.Nr.:s872436

dem Fachbereich II und VI
der Beuth Hochschule für Technik Berlin vorgelegte Masterarbeit
zur Erlangung des akademischen Grades
Master of Science (M.Sc.)
im Studiengang
Data Science

Tag der Abgabe November 18, 2019

Gutachter

Prof. Dr. Felix Bießmann	Beuth Hochschule für Technik Berlin
Prof. Dr. Alexander Löser	Beuth Hochschule für Technik Berlin

Abstract

Ranking problems are omnipresent in interactions with software that retrieve information. Advancements in machine learning (ML) have led to novel solutions for solving ranking problems using a set of approaches known as Learning to Rank (LTR). The goal of this thesis is to demonstrate the effectiveness of learning to rank in solving the problem of ranking geographic places intended for navigation by comparing it to an existing place search engine called Pelias. Clickthrough logs collected from Pelias usage were utilized to create a training dataset for the learning to rank models. Linear, tree-based, and neural learning to rank models were built using the standard ML workflow and evaluated offline using the Mean Reciprocal Rank (MRR) metric. The tree-based models show significant MRR improvements over Pelias, while a subset of the linear and neural models show marginal improvements. An analysis of the results revealed open questions and clear directions for future work on the LTR models.

Acknowledgments

Alhamdulillah.

I would like to express my gratitude to Felix Bießmann for his guidance and enthusiasm. I would also like to thank the Place Search team at Moovel for the brilliant work they put into the foundation that made this thesis possible.

Finally, I am profoundly grateful to my family for their unending support.

Contents

Chapter 1

Introduction

1.1 Problem Statement

The first step in most navigation systems or applications is to choose a destination and origin. The usual workflow is: a user provides a text query, and the system responds with a list of places that might satisfy the users intent. This act of searching for geographic places using a textual query is called Place Search, while the component in a navigation application that provides the place search functionality can be referred to as a place search engine, or simply a search engine.

Generally, a search engine is a system that retrieves documents that correspond to a query. In the context of place search, a document refers to a place; and a document's fields can contain data about different attributes of the place such as its name and location. A search engine always attempts to sort results based on the probability that a result would satisfy the user's need. This probability is also known as relevance. In other words, the objective of a search engine is to order results from most relevant to least relevant consistently. Search engines have to solve what is known as a ranking problem to achieve this objective.

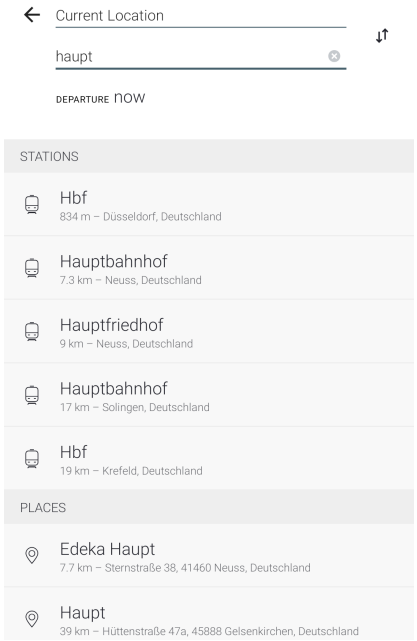

Figure 1.1: Example search screen on a Moovel application

The search query in Figure 1.1 is “haupt”; and the results are separated by section or type, where type could be public transport stop, point of interest (POI), or address. It should be noted that the results are sorted based on their relevance within each section, and not across all sections.

Ranking problems and search relevance are difficult, and as a result, search engines often get the sorting of results based on relevance wrong. In place search, this shortcoming is evident in situations where a user only finds their intended place at a lower rank position below irrelevant results or does not find it altogether.

Traditional information retrieval (IR) methods are widely used to solve ranking problems. These methods have shown impressive results through their use in full-text search engines such as Elasticsearch and Solr, which have consequently gained popularity. These search engines implement classical algorithms such as TF-IDF [40], BM25 [41], and PageRank [35], which have proven to be effective at solving ranking problems. The PageRank algorithm, for example, was the core algorithm behind the Google search engine at its inception. These traditional IR methods are usually based on vector space models, probabilistic models, or boolean models; and they may be query dependent or independent.

Place search engines based on traditional IR methods such as Pelias and Nominatim were built to tackle the problem of ranking geographic locations, and they have similarly seen a good amount of adoption. Despite the success of traditional IR in solving ranking problems, it has inherent limitations due to the hand-tuned nature of the solutions. One of the major limitations of systems built using these solutions is their inability to learn from implicit user feedback. In the place search use case, learning from implicit feedback means leveraging clickthrough data to improve relevance based on past user behaviour. The problem of learning from data is the essence of what machine learning tries to solve.

This thesis researches an alternative and measurably improved approach for solving place ranking using learning to rank (LTR) [27]. Learning to rank is a set of machine learning-based approaches designed to solve the problem of sorting collections of documents based on their relevance to a given query. Broadly, there are three learning to rank approaches: pointwise, pairwise, and listwise; and they differ primarily in terms of the input to their loss functions. Learning to rank is a well-studied and increasingly popular area in machine learning that is seeing success at companies such as Yahoo [54] and Microsoft [8]. Learning to rank and its different approaches are delved into in greater detail in Chapter 2.

The hypothesis that learning to rank can overcome the limitations inherent in hand-tuned methods is investigated by implementing multiple learning to rank models using clickthrough data. The types of learning to rank models implemented are: linear, tree-based, and neural. The dataset used in this thesis is from historical search data collected from navigation apps owned and operated by Moovel Group. The search engine used by the apps is Pelias. Therefore, Pelias is the foundation for this research, and also serves the benchmark for evaluating the performance of the learning to rank models.

1.2 Limitations of the Existing Solution

Pelias uses Elasticsearch under the hood, which is a full-text search engine that supports several functionalities for information retrieval, which includes indexing data using analysis chains; and querying documents using text-matching based on TF-IDF and edit distance, function scoring, hand-written boosts and weights on matched fields. A well-tuned Pelias search engine performs satisfactorily as measured by standard ranking evaluation metrics such as clickthrough rate (CTR)

and mean reciprocal rank (MRR). Pelias relies on continuously tuning the search engine with learned heuristics; this is error-prone and results in accrued and sprawling complexity over time. Additionally, Elasticsearch's cookie-cutter abstractions are insufficient for learning and leveraging the latent user and document features in clickthrough data. Learning to rank approaches, through their use of machine learning, are uniquely qualified to learn optimal search result orderings based on user feedback mined from clickthrough data.

The next sections highlight the limitations of using full-text search databases such as Elasticsearch for place search.

1.2.1 Learning from Historical Data

Massive amounts of data are generated as users interact with a search engine. Embedded within this data are latent features that can be leveraged to deliver a better search experience to users. However, full-text search engines are limited to the data stored in the static fields of their documents during retrieval. This limitation can be side-stepped using machine learning-based approaches such as learning to rank, which is capable of improving search performance as the amount and quality of training data increases.

1.2.2 Location Biasing

A common assumption in place search is that people tend to search for places closer to them than farther away. In Elasticsearch, document scoring can be influenced at query time by defining a decay function such as linear or exponential, and setting parameters; decay, offset, and scale for the function. This method of scoring based on location, also known as location biasing has some limitations. Firstly, finding optimal values for the decay functions parameters is not straightforward. Secondly, the parameters for one region are likely different from another's - this is expected since regions differ in terms of density and area. Lastly, the supported decay functions are able to approximate the general user tendency of searching based on proximity, but they fall short of capturing more sophisticated travel patterns. Figure 1.2 illustrates the distances in kilometers between users and the places they clicked on.

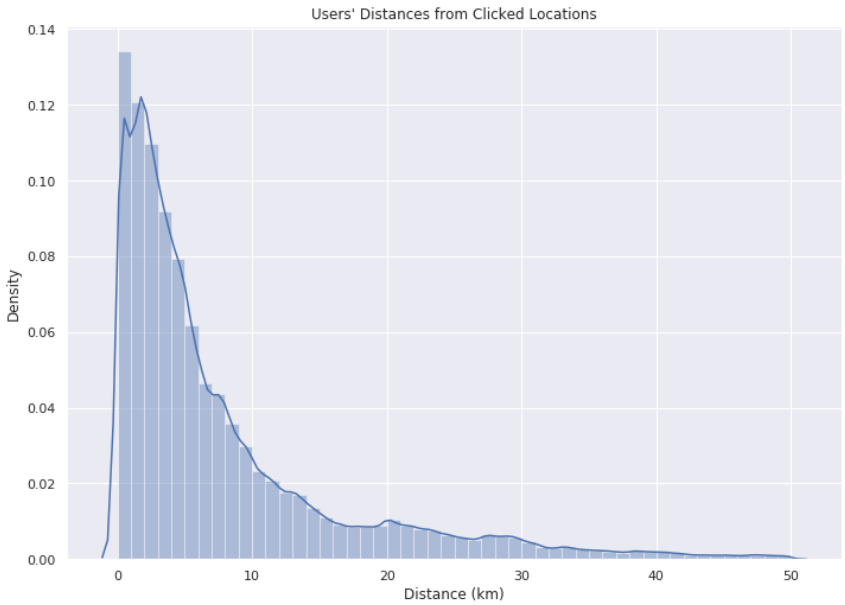

Figure 1.2: Users' distances from selected locations

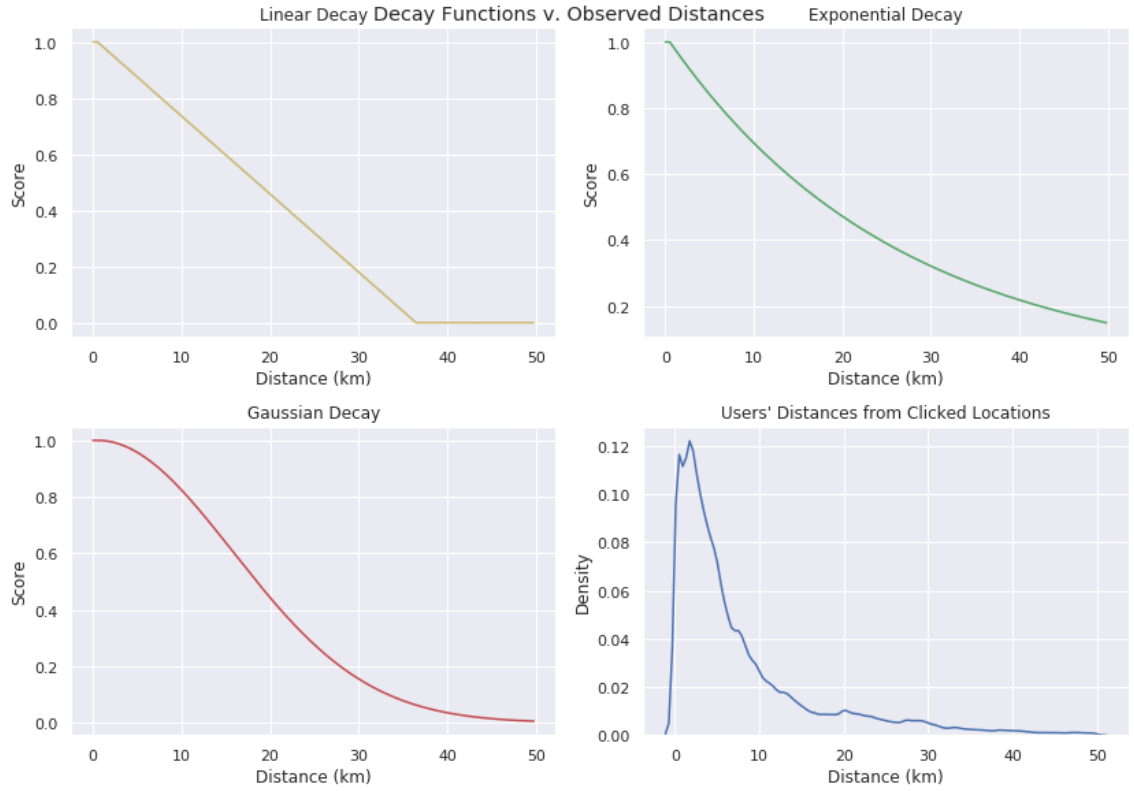

Figure 1.3: Distances between users and selected places vs. decay functions supported by Elasticsearch

Empirically, the Elasticsearch decay functions model the probability that a user will travel to a location, and places are scored based on that probability. The two top plots and bottom left plot in the figure above illustrate how documents would be scored depending on the chosen function and its parameters. The bottom right plot shows the actual distances between users and the places they clicked on. From this, we learn that none of the decay functions sufficiently models the observed distribution.

1.2.3 Temporal Relevance

The importance of certain places could vary based on the time of day, week, month, and year. Such importance could be represented in Elasticsearch, for example, by using the query operator called Field Value Factor. Field Value Factor accepts a field and a function that would be applied to the field value such as a square root or log function to influence a document's score. Such a field could be used by say, defining a field called importance that keeps track of the number of times a document is selected. However, this operator and conceivably other hand-tuned alternatives do not have the flexibility to capture variations in importance based on cyclical time windows like weekends, and factors like weather.

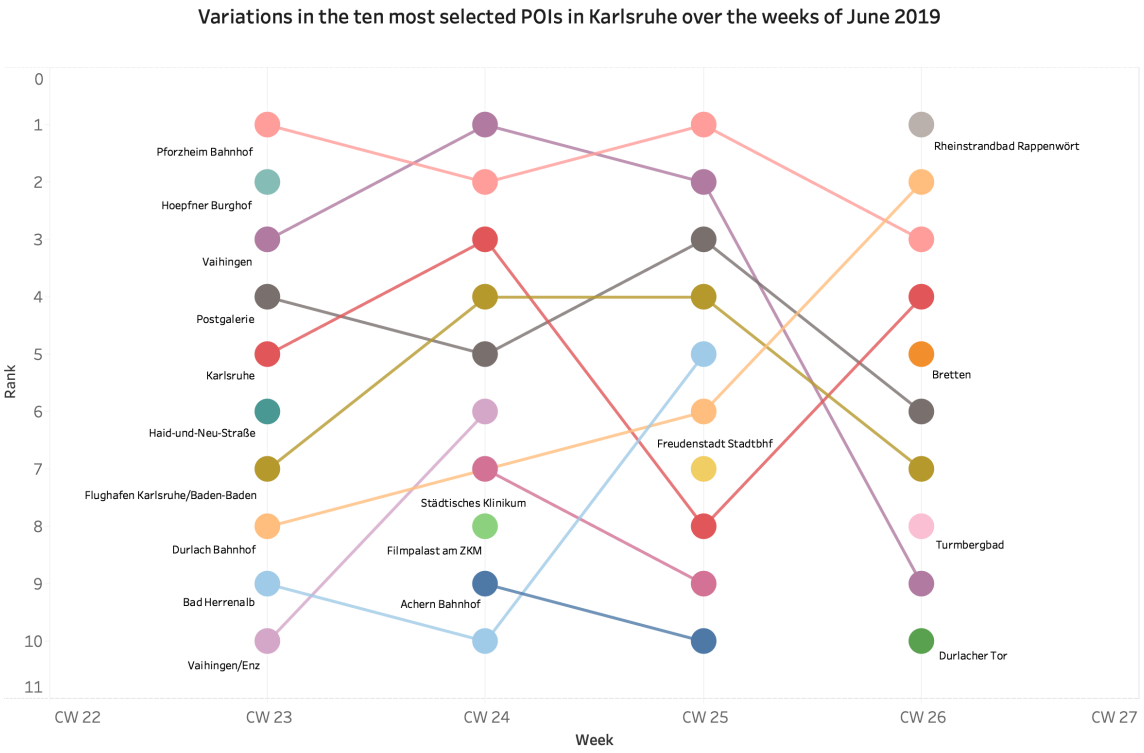

Figure 1.4: Variation in the ten most selected POIs in Karlsruhe over the weeks of June 2019

Figure 1.4 shows the weekly popularity patterns for points of interest within a month. For example, Rheinstrandbad Rappenwort happened to be particularly popular during calendar week 26. This might be correlated with the summer season, hot weather, or both.

1.2.4 Query Parsing

When Elasticsearch receives a query such as "McDonald's Friedrichstraße", it attempts to analyze it; which involves applying predefined tokenizers and character filters on the query. Given the query above, Elasticsearch might attempt to match all the fields of the documents in the index with the tokens "mcdonalds" and "friedrichstraße". A better approach would be to recognize the entities in the query, i.e., a restaurant or point of interest and a street, and attempt a match on the corresponding document fields. The problem of identifying entities in a query is known as Named Entity Recognition (NER), and it is well studied in the literature [31]. Elasticsearch has no native facility for this.

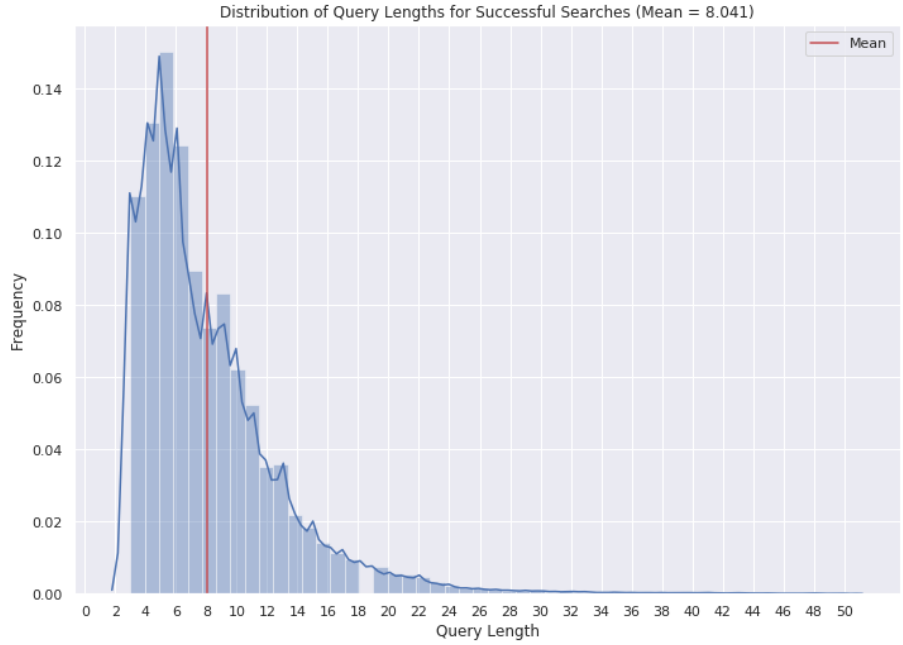

Figure 1.5: Distribution of Query Lengths for Successful Searches

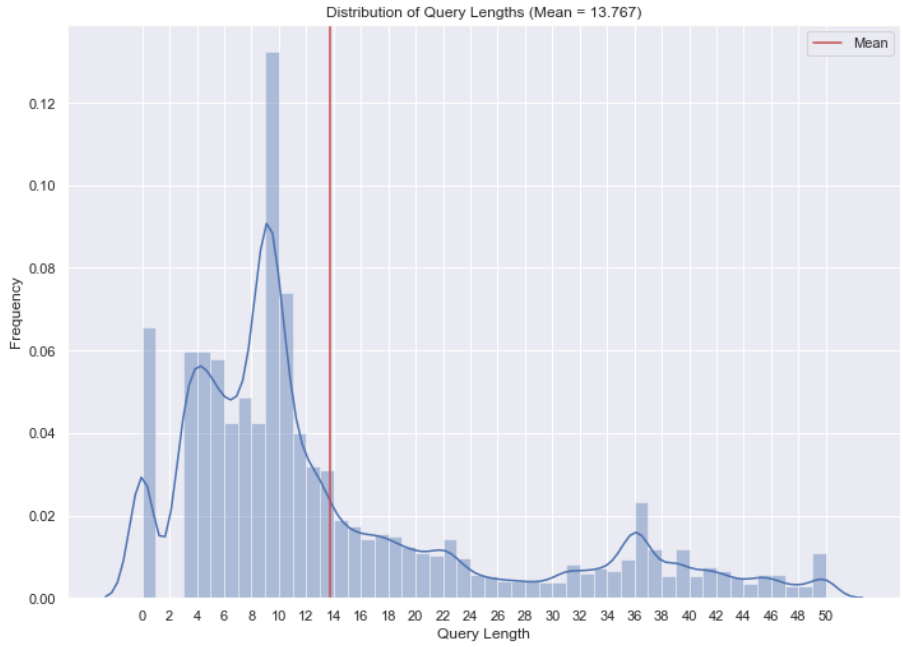

Figure 1.6: Distribution of Query Lengths for Failed Searches

An analysis of the clickthrough data shows that searches that result in a click tend to have shorter queries, while unsuccessful searches tend to have longer queries (Figure 1.5 and 1.6). Longer queries require a better query parsing mechanism, which the brute force search fails to satisfy.

1.2.5 Location Sharing

Section 1.2.2 described the use of location biasing to serve users with more relevant result lists. For location biasing to work, however, users have to permit location sharing with the navigation

application. The clickthrough logs show that a significant percentage of users do not allow location sharing. A cursory analysis of the impact of location sharing shows that users who do not share their locations experience worse rankings as measured by MRR (see section 2.2.1) (Figure 1.8) and type in slightly longer queries based on the mean character length (Figure 1.7).

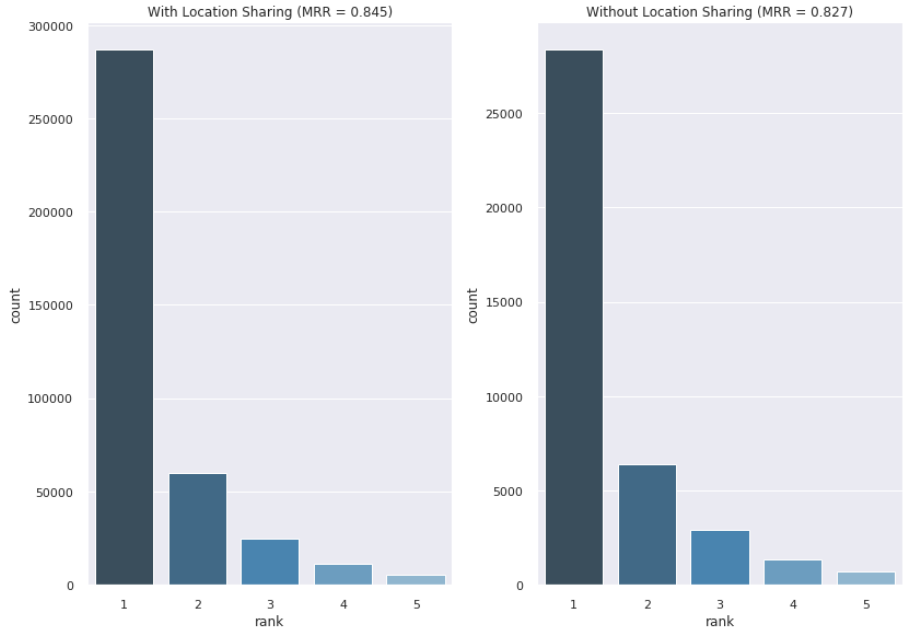

Figure 1.7: Rank distributions for searches with location sharing (left) and searches without (right).

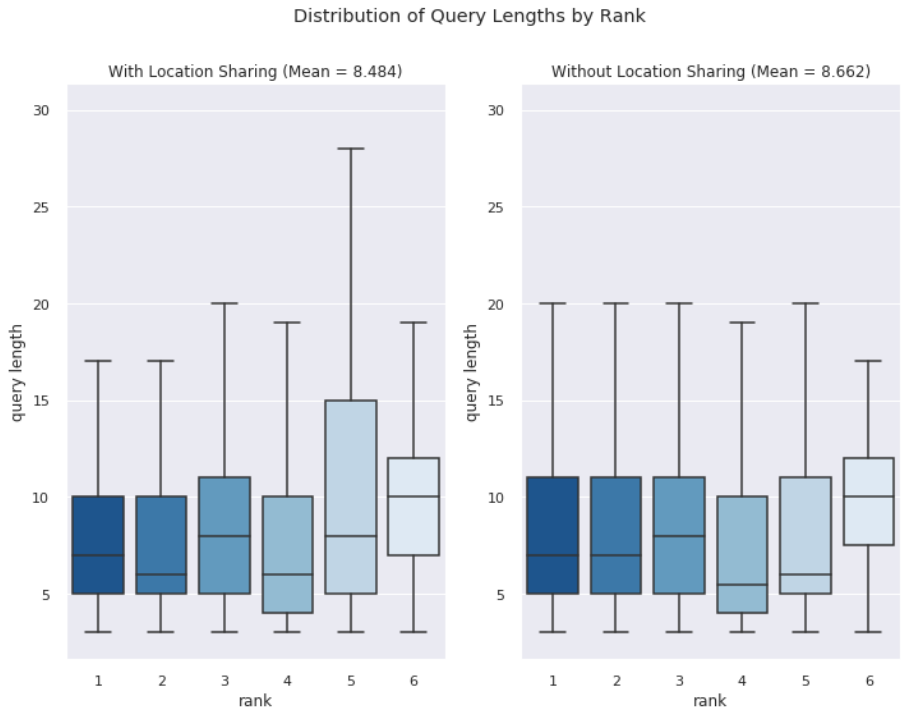

Figure 1.8: Distribution of query character lengths for searches with location sharing (left) and searches without (right).

1.3 Related Work

Learning to Rank (LTR) is an area of machine learning that is concerned with optimizing the utility of ordered lists [27]. A lot of the motivation behind LTR research is a direct result of the importance of web search engines. This is evident not only in the research produced but also in the popular benchmark datasets such as LETOR 3.0 [39] and MSLR [37]. LTR has broader applications beyond web search engines and document retrieval and has been applied to problems such as collaborative filtering [44], question answering [48], and computational biology [7].

A survey of the literature did not yield much work that is closely related to the application of LTR on place search. A slightly related area is Geographic Information Retrieval (GIR), which is concerned with retrieving documents containing geographic dimensions based on queries with geographical or spatial references [24]. LTR has been applied to GIR in [28]. Though similar in name, place search is different from GIR as it involves retrieving documents that represent geographic entities with named attributes using queries that are formulated based solely on the named attributes. Another related work from Foursquare studies the problem of mapping a user's location to a point of interest; or nearby search as they refer to it [43]. The paper mentions a system for place search using queries and doesn't discuss it further. We believe there is no widely available published work that demonstrates the application of learning to rank to place search using text queries.

1.4 Summary

This chapter introduced the ranking problem and its importance to place search. The existing solutions for solving the problem were also discussed, and their limitations were used to motivate the core argument behind this thesis. The chapter ends with a brief look at related work in the literature. The next chapter delves into the theoretical basis of learning to rank.

Chapter 2

Learning to Rank

Learning to Rank refers to a set of machine learning methods for optimizing the utility of ordered lists. The goal of an LTR model is to predict the relative ordering of a set of observations, in contrast with classification and regression models; which aim to accurately predict values of individual observations as separate from other observations. An LTR model determines a scoring function $f(\cdot)$ that defines a ranking over a set such that a loss L is minimized.

Before going further, it is important to define some terms within the context of LTR.

Query Group

A query group identifies a list of ranked documents $\{d_j\}_{j=1}^{m}$ associated with a single query input q, where m is the number of documents. For example, a search for the query "Hauptbahnhof" might return fifteen results that make up a query group. A definition of a query group is crucial because it affects how a training dataset is constructed and labelled. During our work, we identified two ways of defining a query group:

- A query group could be the set of documents returned for a unique query; with relevance judgements based on clicks defined as the normalized frequencies of the documents.
- A query group could be the set of documents returned for an instance of a query; with relevance judgements based on whether a document is clicked or not. This is the definition adopted by this thesis.

Scoring Function

In regression and classification models, a scoring function $f(.)$ determines the value of a single observation, while in LTR, the scoring function determines the rank of each document in a query group.

Loss Function

The loss function $L(.)$ determines the difference between the predicted ordering of documents and their optimal ordering based on their labels. The exact formulation of the loss function is determined by the learning to rank approach.

2.1 Learning to Rank as an Empirical Risk Minimization Problem

Empirical Risk Minimization offers a formulation of problems that can be used to find the best approximation of a function given that we don't know the true distribution of our data [47]. Given that queries and documents are represented as vectors in $\mathbb{R}^n$, learning to rank can be modelled as an empirical risk minimization problem as follows:
With a labelled dataset $\{(X_q, y_q)\}_{q=1}^{n}$, where

- n is the total number of training queries.
- $X_q = \{d_j\}_{j=1}^{m}$ is the set of documents for query $_q$, also known as a query group.
- $y_q = \{y_j\}_{j=1}^{m}$ is the set of relevance judgments for our documents.

The goal is to learn a function $h : \mathbb{R}^n \rightarrow \mathbb{R}$, which minimizes:

$$\hat{R}(h) = \frac{1}{n} \sum_{q=1}^{n} L\left(\pi\left(h, X_q\right), y_q\right)$$

where $\pi\left(h, X_q\right)$ is the ranking of documents for query q, and L measures the difference between the prediction $\pi\left(h, X_q\right)$ and the ground truth label y_q.

Based on the empirical risk minimization principle, an LTR algorithm chooses the scoring function f that minimizes the empirical risk $\hat{R}(h)$:

$$f = \arg\min_{h \epsilon \mathcal{H}} \hat{R}(h)$$

Based on the scoring function f, the rankings obtained from $\pi\left(f, X_q\right)$ should output the best ordering based on the relevance judgements in the form:

$$y_i^q > y_j^q \Leftrightarrow f\left(d_i^q\right) > f\left(d_j^q\right)$$

2.2 Evaluation Metrics

Several metrics have been devised to evaluate the performance of search engines at delivering the best rankings to users. Popular metrics include Click-Through Rate (CTR), Mean Average Precision (MAP) [27], Normalized Discounted Cumulative Gain (NDCG) [27], Mean Reciprocal Rank (MRR) [29], and Expected Reciprocal Rank (ERR) [12]. NDCG is designed to evaluate performance based on graded relevance judgments. MAP and MRR only consider whether a document is relevant or not i.e. binary relevance, and are better suited metrics compared to NDCG when only one document in a result list is considered relevant. CTR is an online evaluation metric, which can be approximated using Precision at k (P@k) .

MRR, MAP, and NDCG are discussed briefly. Then the distinction between online and offline evaluation metrics and how that impacts what can be measured in terms of information retrieval performance is discussed. Finally, the evaluation metric used in this thesis and the reasoning behind is explained.

2.2.1 Mean Reciprocal Rank (MRR)

The reciprocal of a query is the inverse of a clicked result's rank. Taking the mean of reciprocals across n queries gives the value of MRR.

$$MRR = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{rank_i}$$

where $rank_i$ refers to the rank of the selected document for the i-th query.

2.2.2 Mean Average Precision at k (MAP@k)

MAP is the mean of Average Precisions (AP). It is suited for cases where relevance is binary, i.e., 1 or 0. In order to understand MAP@k, P@k needs to be defined first. P@k corresponds to the number of relevant results in the result list. Since we assume that there's only a single relevant result, P@k can never be perfect except for $k = 1$. P@k does not account for the ranks at which of the selected document appears among k. P@k is computed as follows:

$$P_i@k = \frac{TP_i}{TP_i + FP_i} = \frac{\sum_{j=1}^{\min\{k,\rho_i\}} rel_{ij}}{k}$$

where P$_i$@k is the precision at k for a query i, k is a cutoff for the number of results to consider, TP is the number of true positives, FP is the number of false negatives, ρi is the number of results, and $rel_i j$ is the relevance of a result item (1 or 0).

P@k corresponds to the number of relevant results in the result list. $P@k$ does not account for the ranks at which the selected document appears among k. For example, if a result list contains items with relevances [0, 0, 0, 1, 0], then $P@3$, would be 0 or NaN because:

$$P@3 = (0 + 0 + 0)/3 = 0$$
$$P@4 = (0 + 0 + 0 + 1)/4 = 0.25$$
$$P@5 = (0 + 0 + 0 + 1 + 0)/5 = 0.2$$

Average Precision at k (AP@k) is calculated using:

$$AP_i@k = \frac{\sum_{j=1}^{\min\{k,\rho_i\}} rel_{ij} P_i@j}{\sum_{j=1}^{\min\{k,\rho_i\}} rel_{ij}}$$

where AP$_i$@k is the average precision at k and $P_i@j$ is the precision at j.

For example, if a result list contains items with relevances [0 1 0 1 1], then $AP@2$ would be calculated as:

$$AP@2 = 0 * 0 + 0.5 * 0.5 = 0.25$$
$$AP@4 = (0 * 0) + (0.5 * 0.5) + (0 * 0) + (0.5 * 0.5) + (0.5 * 0.5) = 0.75$$

Finally, MAP@k is formulated as:

$$MAP@k = \frac{\sum_{i=1}^{U} AP_i@k}{n}$$

where n is the number of queries.

2.2.3 Normalized Discounted Cumulative Gain at k (NDCG@k)

NDCG@k is similar to MAP@k, the key difference being that NDCG works for non-binary or graded relevance in addition to binary relevance. The idea behind NDCG is that each item in a result list has a relevance score that is referred to as the gain (G). Cumulative Gain (CG) is the result of summing up the relevance scores.

$$CG_i@k = \sum_{j=1}^{k} rel_j$$

where rel_j is the relevance of the item at rank j.

NDCG takes into account the order in which more relevant items compared to less relevant items appear. Before summing up the scores, each score is divided by a discounting factor that signals a reduction in gain as rank positions progress down the result list. Summing up the discounted gains produces the Discounted Cumulated Gain (DCG).

$$DCG_i@k = \sum_{j=1}^{k} \frac{2^{rel_j} - 1}{\ln(j+1)}$$

DCGs across different result lists are not necessarily comparable due to different result list sizes or the number of relevant results. In order to make DCGs comparable, they are normalized using the ideal DCG (IDCG).

$$IDCG_i@k = \sum_{j=1}^{|REL_k|} \frac{2^{rel_j} - 1}{\log_2(j+1)}$$

where $|REL_k|$ is the ideal ranking of items in the result list.

$$NDCG_i@k = \frac{DCG_i@k}{IDCG_i@k}$$

2.2.4 Online Evaluation vs. Offline Evaluation

Offline evaluation involves evaluating the performance of a model using clickthrough data, while online evaluation involves evaluating performance on a running system. Both methods have limitations.

Online Evaluation

- The model has to serve traffic from a deployed system (usually in an A/B testing setup).
- Users are experimented on, and they may experience performance degradation.

Offline Evaluation

- Offline can't evaluate how a model will affect user behavior in terms of queries. Say we want insight into whether the length of queries typed by users would change, or whether users would type certain queries differently.
- Unable to measure performance improvement on queries that had previously not resulted in a click.
- Unable to evaluate the interaction between the ranker and reranker. For example, it wouldn't be possible to re-rank a larger number of documents and test a model's performance, or update parameters in the search engine and test the relative change in the re-ranker's performance.

2.2.5 Evaluation Metrics Used

Online metrics are relatively difficult to obtain compared to offline metrics; hence, the metric of choice for evaluating the learning to rank models in this work is MRR. The choice of MRR over MAP and NDCG@k is based on the form of the training data. That is, each query group or result list has only one relevant item (the one clicked on by a user) that has the binary relevance label of 1; other items in the query group receive the label 0. Even though MRR@1 produces the same score as NDCG@1 and MAP@1, once the relevant item is found in MRR, the score remains the same regardless of the result list size. This is not the same with MAP and NDCG. This behavior makes MRR more desirable for the current use case.

2.3 Types of Features

The input space of machine learning models is made of vectors derived from features of the input data. Input vectors for LTR models can be constructed in several ways:

- Example / Document / Static Features: these are the features that are specific to each item in a query group. For example, the name of a place, its longitude and latitude, neighbourhood, etc.
- Dynamic Features: these features are derived from the relationship between queries and documents features. For example, the edit distance between a query and a place name.
- Query Level Features or Query Features: these are derived solely from the query e.g. query text.
- Context Features: these are features that apply to all items in a query group. For example timestamp, user's geographic coordinates, etc.

2.4 LTR Approaches

The two ways that LTR approaches differ is in their characterizations of the scoring and loss functions they are trained to minimize. Even though some LTR algorithms clearly fall under one approach, some LTR algorithms combine multiple approaches in their scoring and loss functions. A more thorough distinction can be drawn across the approaches based on their input space, output space, hypothesis space, and loss function [27].

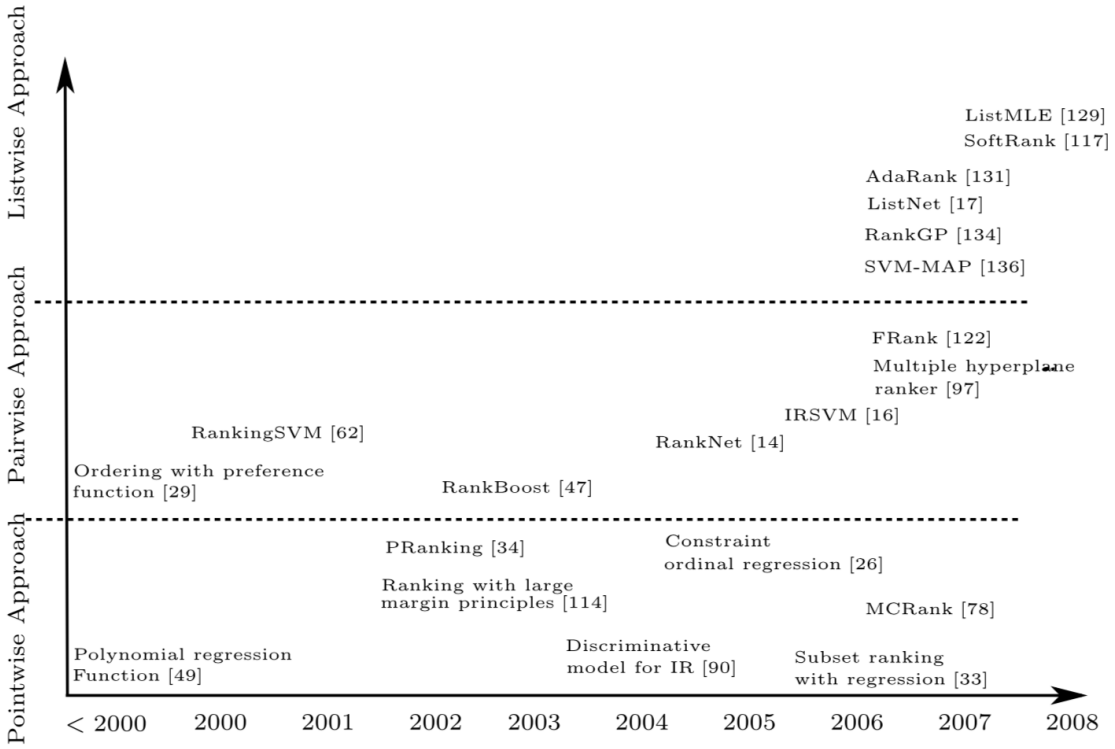

Figure 2.1: LTR Algorithms [27]

2.5 Pointwise Approaches

In the pointwise approach, given a query group, a score is computed for each document independent of other documents. Each document's loss is also defined in terms of the difference between its labelled score and its predicted score. The ranking of a result list is derived by simply sorting documents by their predicted scores. The pointwise approach amounts to training a classifier or regressor that predicts the relevance of a document given a query, where the loss function could be the following:

$$L\left(\pi\left(f, X_q\right), y_q\right) = \frac{1}{n}\sum_{i=1}^{n}\left(f\left(d_i^q\right) - y_i^q\right)^2$$

The pointwise approach bears the closest resemblance to standard ML models, but models the ranking problem poorly compared to other approaches. Additionally, the pointwise approach's loss can hardly be optimized based on the query-level and position-based characteristics of the evaluation metrics. The pairwise model tries to overcome this limitation by modeling the relative order of pairs of documents.

Examples of the pointwise approach include McRank [26] and Ordinal Regression [14].

2.6 Pairwise Approaches

In the pairwise approach, a collection of ordered document pairs is created by applying the cartesian product of the set of documents in a query group onto itself. In other words, each document in the query group is paired with every other document, which is then inputted into the scoring function to produce preferences between pairs of documents. After comparing all pairs, an overall

ordering of documents emerges. A preference function for each pair returns a value of 1 given that the preference is correct (concordance), or -1 otherwise (discordant). The loss function aims to minimize the number of discordant pairs. The pairwise approach bears a resemblance to the real world conception of ranking problems through its modeling of relative order. The following example is of a loss function for the pairwise approach:

$$L\left(\pi\left(f, X_q\right), y_q\right) = \sum\nolimits_{(i,j):\, y_i^q < y_j^q} \log\left(1 + \exp\left(f\left(d_i^q\right)\right) - \left(f\left(d_j^q\right)\right)\right)$$

Once we have a loss function that outputs the pairwise preferences between pairs of documents, an absolute ordering needs to be derived [27]. This derivation is known as a rank aggregation, and can be represented as:

$$\max_{\pi} \sum_{u<v} h\left(x_{\pi(u)}, x_{\pi(v)}\right)$$

Where $\pi\left(u\right)$ and $\pi\left(v\right)$ are the indices of the documents given by the ordering π.

Rank aggregation can be computationally intensive, and is a NP-hard problem. However, given that the LTR models in this thesis are top-k re-rankers, discussed in Section 3.1, this is a non-issue.

A potential issue with the pairwise approach is that its loss functions have a tendency to be dominated by queries with large numbers of document pairs [27]. A solution for this involves using a query-level normalization so the magnitude of losses between query-document pairs becomes comparable[38]. We find this to be a non-issue in this thesis as a result of the definition of query group adopted, and use of LTR models as top-k re-rankers where k is a bounded and much smaller subset of documents in the search index.

Examples of the pairwise approach include Ranking SVM, RankNet, and RankBoost.

2.6.1 Ranking SVM

The Ranking SVM algorithm reduces the ranking problem to a classification problem using the SVM algorithm. It was introduced in [21] as one of the pioneering applications of machine learning on clickthrough data to optimize search engine ranking. The author frames the idea of learning labels from clickthrough data as relative relevance judgements, which can be represented as:

$$link_i < r * link_j \text{ for all pairs } 1 \leq j \leq i, \text{ with } i \in C, j \notin C$$

where C is a set of clicked documents, and $link_i$ is preferred over link $link_j$ based on a click. After formulating the relative relevance judgments, Kendall's τ(pronounced Kendall's Tau) is introduced as an expected loss function that minimizes the lower bound of the Average Precision [6] metric or the average rank of relevant documents when maximized. Kendall's τ measures the number of inversions required in order to have the optimal ordering of results [2]. It is defined as the number of concordant over the number of discordant pairs. Given that r_1 and r_2 are two ranking functions, the Kendall's Tau between $r1$ and $r2$ is represented as:

$$\tau\left(r_1, r_2\right) = \frac{P - Q}{P + Q} = 1 - \frac{2Q}{P + Q}$$

where P is the number of concordant pairs and Q is the number of discordant pairs (inversions).

Empirical risk minimization can be used to optimize by defining the empirical loss as:

$$\tau_s(f) = \frac{1}{n}\sum_{i=1}^{n} \tau\left(r_{f(q_i)}, r_i^*\right)$$

where s is a training set containing queries q, and $r*$ is the expected relative relevance judgements.

Next, a training dataset is created by taking the cross-product of queries and documents to create a set of all possible query-documents pairs, and generating features based on the pairs. The training set can be denoted as $x = \{\Phi(q, d_i)\}_{i=1}^{n}$ where Φ projects query q and document d pairs onto a feature space. Features such as TF-IDF scores and pagerank are commonly used for document retrieval [37]. In place search, a feature might be the edit distance between a query and place name. A classifier is used to determine the relevance between pairs based on the hyperplane that minimizes the hinge loss i.e. the loss on a correct ordering is 0, while an incorrect ordering is 1. Preference between pairs is determined by their distance from the hyperplane, while the hyperplane is learned based on the weights of query-document pairs' features. This is formulated as follows:

$$\min \frac{1}{2}||w||^2 + C\sum_{i=1}^{n} \sum_{u,v:y_{u,v}^{(i)}=1} \xi u, v^{(i)}$$
$$\mathbf{s.t.}\ w^T\left(x_u^{(i)} - x_v^{(i)}\right) \geq 1 - \xi_{u,v}^{(i)}\ ,\ \mathbf{if}\ y_{u,v}^{(i)} = 1,$$
$$\xi_{u,v}^{(i)} \geq 0\ ,\ i = 1, ..., n.$$

Where C is a regularization parameter that controls the trade-off training error and generalizability of the classifier, ξ are the slack variables, and $\frac{1}{2}||w||^2$ is the margin term.

2.6.2 RankNet

The RankNet algorithm minimizes the cross-entropy loss for a pair of documents given by a scoring function that is the probability that one document should be ranked over another [27] [8]. The scoring function, which gives the probability u is scored higher than v is given as:

$$P_{u,v}(f) = \frac{\exp(f(x_u) - f(x_v))}{1 + \exp(f(x_u) - f(x_v))}$$

The cross-entropy loss function is then used:

$$L(f; x_u, x_v, y_{u,v}) = -\bar{P}_{u,v} \log P_{u,v}(f) - \left(1 - \bar{P}_{u,v}\right) \log\left(1 - P_{u,v}(f)\right)$$

Given that its loss function is differentiable, RankNet can be modelled using a neural network and optimized using the Stochastic Gradient Descent algorithm. By optimizing the loss function above, RankNet minimizes the number of incorrect orderings among pairs of documents. Fidelity loss is a loss function developed to overcome the problem with RankNet where the cross entropy loss has a non-zero minimum in certain cases [27] [46].

2.7 Listwise Approaches

The listwise approach directly reads in the list of all documents and tries to come up with the optimal ordering for it. The listwise approach has been found to perform consistently better than the other approaches [27][10]. Algorithms in the listwise approach fall into two categories: algorithms that minimize listwise ranking losses and algorithms that directly minimize the evaluation metric [27]. Algorithms in the first category include ListNet [10] and ListMLE [52]. The next two sections discuss two well-known algorithms in the first category: LambdaRank [9] and LambdaMART [51].

2.7.1 LambdaRank

Even though the performance of ranking models is evaluated using metrics such as NDCG and MAP, these metrics are not optimized directly because they are flat and discontinuous with respect to their parameters. The algorithms discussed so far use surrogate loss functions that indirectly optimize the evaluation metrics. The RankNet algorithm, for example, optimizes for the number of pairwise errors, but that doesn't work well for some evaluation measures [8]. In RankNet, the cost of misranking a pair of documents i_1 and i_{200} is the same as that of j_1 and j_{100}. This behavior of RankNet gradients is illustrated in Figure 2.2.

Figure 2.2: Gradient Magnitudes in RankNet vs. LambdaRank

In Figure 2.2, the list of results on the left has a lower pairwise loss compared to the list on the right when computed by RankNet. Informally, the red arrow indicates the magnitudes of gradients computed by RankNet, while the black arrow depicts LambdaRank's gradient magnitudes. [8].

LambdaRank, a listwise approach, bypasses this problem by introducing a new cost function that give weights to the pairwise errors, in other words, adjusts the gradients of the cost based on a discount factor such as the rank.

$$\lambda_{uv} = \frac{-1}{1 + e^{f(x_u) - f(x_v)}}$$

where λ is a penalty term that represents the cost of ranking u and v wrongly.

By using any one of several evaluation metrics like NDCG as λ, we can directly optimize an implicit cost function based on the metric in a listwise manner, even though the metric itself is a

non-smooth cost function. Similar to RankNet, LambdaRank was described in [9] using a neural network, and was shown to improve over RankNet in terms of training speed and performance.

2.7.2 LambdaMART

The LambdaMART algorithm is a combination of LambdaRank and MART, which stands for Multiple Additive Regression Trees, also known as Gradient Boosted Trees. The authors of [51] found that gradient boosted trees work well for ranking problems due to their suitability for handling discrete features and multi-class classification (if the ranking problem is framed as one). Since gradient boosted trees can be used to optimize any loss function that can compute a gradient at each training stage, and LambdaRank computes gradients based on its implicit cost function, the two can be combined [19] . LambdaMART integrates LambdaRank with gradient boosted trees by training a tree from scratch, computing lambda gradients based on the cost function of the scores outputted by the initial tree, and training the rest of the boosted trees using the algorithm outlined in [51].

2.8 Bias in Clickthrough Data

Clickthrough data indicates relative user preferences between documents and provides implicit user feedback; however, clickthrough data alone is not a reliable source of data for building robust learning systems, due to the bias and noise inherent in implicit user feedback. Bias is an important topic in learning to rank and is well studied in [23] and [49].

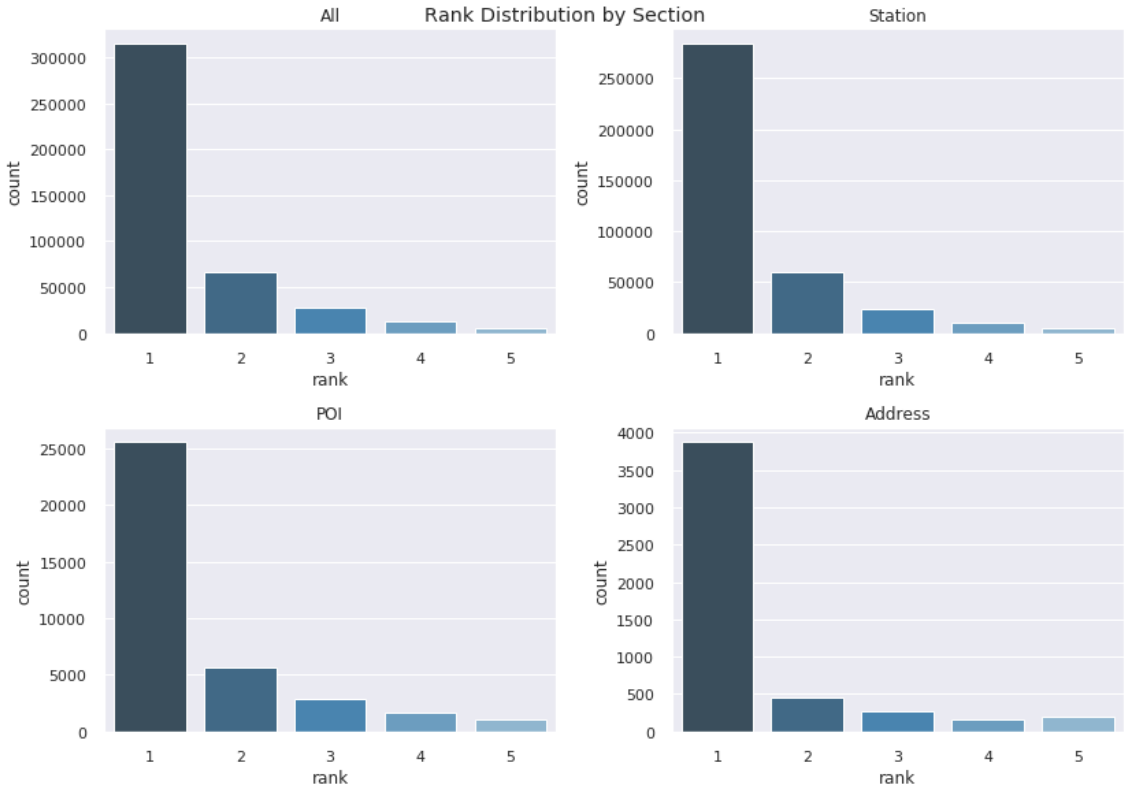

Figure 2.3: Rank distribution by types of places selected by users

There are multiple types of bias, which include: position bias [22], presentation bias [55], and trust bias [34]. These biases are all a type of selection bias, a phenomenon that's widely studied in other fields such as medical research. Position bias dictates that users are likely to click on results that occur at higher positions in the result list. Trust bias is similar to position bias; it occurs when users overestimate the relevance of results placed at a higher rank due to the trust they place in the search system. Presentation bias shows up in cases where users are unduly influenced by a result based on its display attributes, for example, label, icon, formatting, etc.

Several techniques have been proposed in the literature to address noise and bias in clickthrough data. Some techniques that address bias with an assumption of noise-freeness include result randomization, random pair harvesting where result pairs are randomly flipped, and position-based click models [3]. Most unbiasing techniques involve using inverse propensity weighting (IPW), a method popularized by Joachims T. et al. in [23], which involves estimating the click bias at each rank, and training an unbiased model with the estimated biases using a learning to rank algorithm. More recently, research in [3] presents a Bayesian approach to Inverse Propensity Scoring (Bayes-IPS) that address both trust bias and click noise.

The neural ranking framework used in this thesis supports implementing unbiased learning to rank models using inverse propensity weights [36]. It accomplishes this by computing inverse propensity weights for rank positions and including them in the computation of loss during training. We do not address the topic of implementing an unbiased learning to rank model within this thesis.

Signs of bias can be observed bias in our data by looking at rank distributions (Figure 2.3) and distribution of query lengths across rank position (Figure 2.4). The historical data shows that when queries are longer, the lower the position of the result selected by users. A longer query could indicate that a user included additional details, such as the street of a point of its interest and its administrative levels. It could also indicate that the user was searching for a location that has a weak popularity boosting, but is highly relevant for the given query. Research done in [49] found that longer queries resulted in more clicks at lower positions due to reduced positional bias.

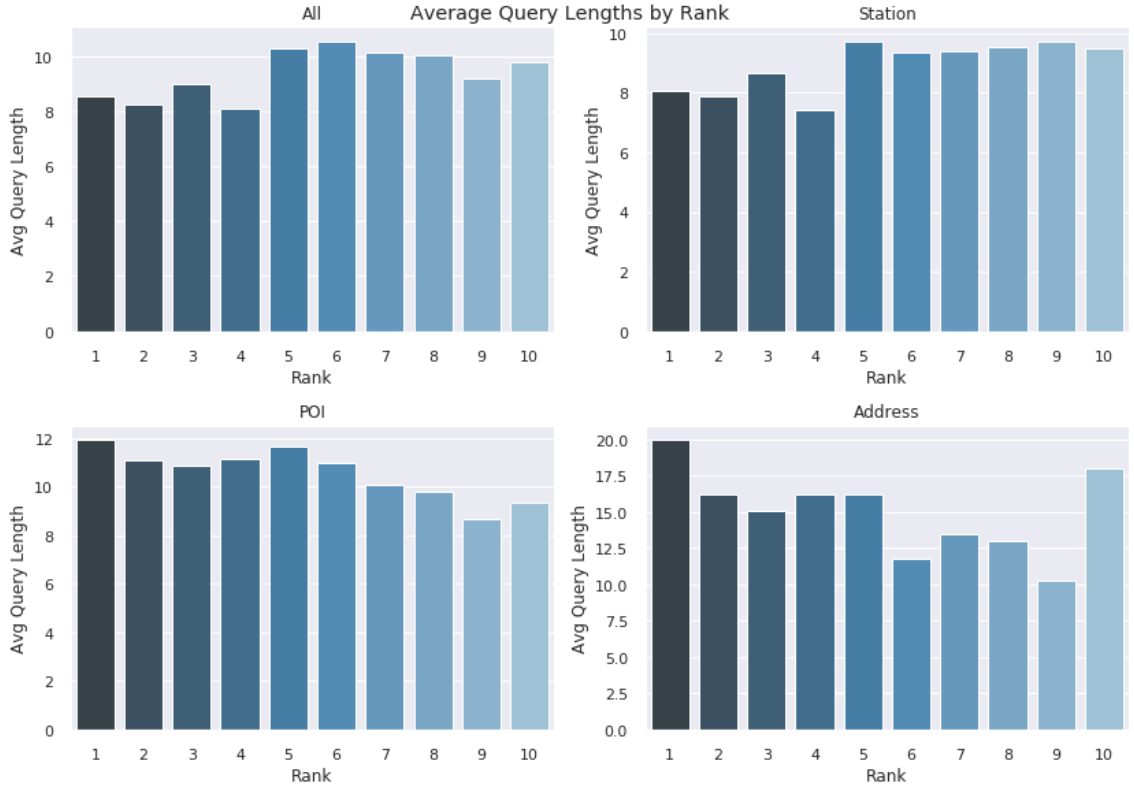

Figure 2.4: Average query lengths at selected ranks

2.9 Summary

This chapter provided an explanation of learning to rank, and its approaches and their differences. It also provided descriptions of the metrics used to evaluate learning to rank models and metrics used in this work. It concluded with a cursory glance at bias in clickthrough data and the use of unbiased learning to rank to counteract it. The next chapter lays the groundwork for the implementation of the learning to ranks models.

Chapter 3

Experiment Setup

3.1 LTR Framework

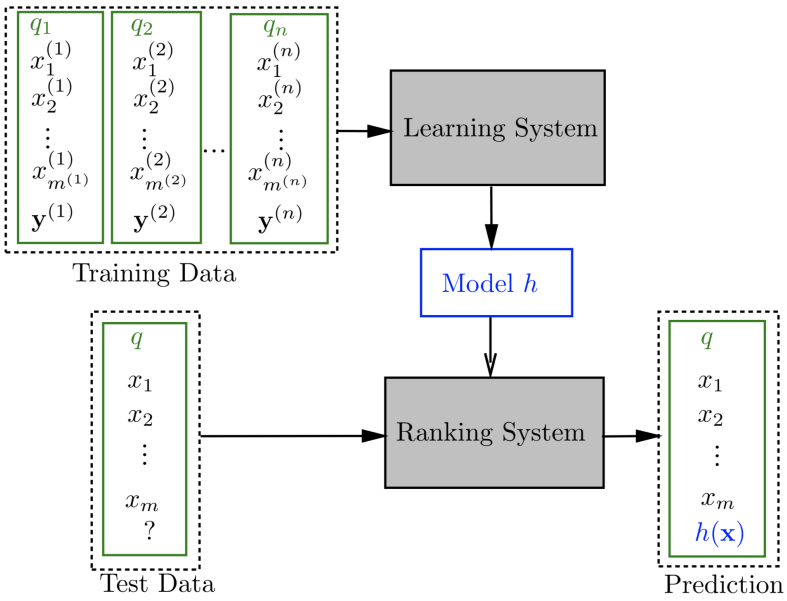

Figure 3.1: LTR Framework [27]

The learning to rank framework in [27] illustrates the interaction between a base ranker and re-ranker. The model h can be viewed as the re-ranker and the ranking system as the base ranker. Assuming test data is inbound search requests from a user, the ranking system would fetch the top-k results, and the model would re-rank them into the final order that is shown to a user. The user's interaction with the result list is fed back into the learning system as training data, completing a feedback loop.

3.2 Dataset

The data used to train the LTR models in this work is collected from search sessions of a navigation app's users. When a user types a query and selects a place from the list of results, the list is logged with an indication of the selected place. Also logged are the coordinates of the user, whether they were searching for an origin or destination, and the timestamp of when the search occurred. The collected data is further processed and enriched with the properties of each place in

the result list such as its name, city and coordinates.

Dimension	Description
text	Query text
focus point latitude	Latitude of the user
focus point longitude	Longitude of the user
timestamp	Timestamp of the search
target	Whether the place is an origin or destination
name	Name of the place
locality	Locality of the place
neighbourhood	Neighbourhood of the place
borough	Borough of the place
place latitude	Latitude of the place
place longitude	Longitude of the place
type	Type of the place i.e. poi, address, or station
freq	Number of times the place was selected (label)

Table 3.1: Dataset Features

3.3 Labelled Data

All searches performed using the navigation application, successful and unsuccessful, are recorded in what is known as a clickthrough log. A *successful* search is one where a user clicks on the result relevant to them, and the converse is an *unsuccessful* search. The LTR models we are interested in fall under the category of supervised machine learning. A key part of building supervised machine learning is acquiring labels or ground truth for the training data. In ranking contexts, a widely common method for acquiring labeled data or relevance judgments is to have humans judge how well queries match search results; where a judgment can be binary (a result is either relevant or not) or graded (a result is relevant based on some scale). This method of gathering relevance judgments can be tedious, expensive, and insufficient for capturing the range of intents and queries that users might express during the course of real-world usage.

The labels in the dataset used in this thesis are derived from clickthrough logs by labeling the clicked result for a query as relevant, and unclicked results as irrelevant, an approach that has been proven to work well [21]. The frequency ("freq") column, which is the label column, is set to 1 for selected, and 0 for unselected. Labels gathered from clickthrough logs are a type of implicit feedback as they do not explicitly signify relevance. Clickthrough logs are biased and noisy, but they bear a sufficient correlation with relevance as we can assume user preference given that users always scan results from the top to the bottom of result lists [21]. If a user selects the third result in a list, we can assume the third result is more relevant than the first, second, and all results below the third. Noisiness is introduced into clickthrough data as a result of factors that contribute to erroneous clicks during a search. Bias in clickthrough data was discussed in Section 2.8.

3.4 Data Engineering

In the previous section, it is explained that training data and labels are collected from clickthrough logs. The more detailed explanation is that the data for the clickthrough logs are collated from multiple data sources. When a user using the search function on a Moovel application clicks on a

place, a single data point is recorded about the metadata of the selected place. The training data for the LTR models, however, requires the rest of the result list that was shown along with the selected place. Data on the rest of the list can be found in a *data lake*, which stores outgoing search responses. By cross-referencing the *data lake* with metadata about the selected place, we can recreate the original search result list from which the user selects the result. The actual implementation of this data engineering effort involves intricate details that are specific to the technologies and architecture used within Moovel that are not covered here.

3.5 Feature Engineering

3.5.1 Temporal Features

Temporal features are time-based features. They are associated with when a search happens, and thus, are part of the user context. The temporal features are derived from the timestamp dimension of a search session by teasing out the day, month, year, part of the day, and whether a search occurred during the weekend. Cyclical features are features that are recurring in nature, for example, day, month, and year. Temporal features are further processed into cyclical features by applying sine and cosine transformations that capture the cyclical pattern of the time values using these functions:

$$x_{\sin} = \sin\left(\frac{2 * \pi * x}{\max(x)}\right)$$

$$x_{\cos} = \cos\left(\frac{2 * \pi * x}{\max(x)}\right)$$

3.5.2 Spatial Features

Spatial features identify points in geographic space and are derived from the coordinates of a user and each place shown on the result list. The raw form of the coordinates is a comma-delimited string. The coordinates are separated using a latitude and longitude features, that gives us two pairs of longitudes and latitudes for a user and each place on the result list. A great-circle distance between a user's location and places on the result list is also computed as an additional feature using the Haversine formula.

3.5.3 Textual Features

The textual features for each item in the query group are derived from the query text, the name of the place, neighbourhood, borough, and city. Additional place properties such street name, house number and zip codes; as well as other administrative levels such as county and region could also be considered. Textual features are processed into character n-grams with an upper bound of $n = 5$ and lower bound of $n = 2$. The n-gram matrices are created using a hashing vectorizer or a count vectorizer from Scikit-Learn. The choice of hashing vectorizer versus count vectorizer determines how the sparse matrix of the n-grams is created. The hashing vectorizer has the benefit of setting a fixed size for the sparse matrix as it does not need to learn a vocabulary on the training text, while the count vectorizer will expand to fit the vocabulary of the training text. We experimented with the hashing vectorizer and different numbers of features, as well as the count vectorizer.

3.5.4 Categorical Features

Categorical features have a fixed number of possible values. They include the search target of a user, i.e. origin or destination, type of a place in the result list; i.e. point of interest, public transportation station, or address, platform of a user's device, and other discrete variables. The categorical features are processed are encoded into one-hot numeric arrays using Scikit-Learn's OneHotEncoder.

3.6 Feature Transformation Pipeline

Scikit-Learn provides a convenient construct called pipelines that supports composing different feature transformers and ultimately handing them to a model function. We use Scikit-Learn's column transformer to combine our feature engineering into a series of steps that look like the following:

```
from sklearn.compose import ColumnTransformer
from sklearn.feature_extraction.text import HashingVectorizer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, FunctionTransformer

textual_features = ['text', 'name',    street    , 'locality',
                                        neighbourhood    ,    borough    ]
textual_transformer = Pipeline(steps=[
    ('char_ngram_analyzer', HashingVectorizer(n_features=2**20, analyzer='
                                            char_wb', ngram_range=[2, 5]))])

categorical_features = ['place_type', 'search_target']
categorical_transformer = Pipeline(steps=[
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

numerical_features = ['focus_lat', 'focus_lon', 'place_lat', 'place_lon', '
                                    hr_sin', 'hr_cos',
                      'day_sin', 'day_cos', 'month_sin', 'month_cos', '
                                                        is_weekend' 'rank
                                                        ']


preprocessor = ColumnTransformer(
    transformers=[
        ('cat', categorical_transformer, categorical_features),
        ('num', 'passthrough', numerical_features)
    ] + [('{}_trf'.format(i), textual_transformer, i) for i in textual_features
                                            ])
```

The output of the preprocessor is a sparse matrix with n x 5242894 dimensions. The data fed into the baseline model is preprocessed using the feature engineering outlined in Section 3.5.

3.7 Baseline Model

The baseline model used in this thesis is SVM^{rank}, written by Thorsten Joachims. SVM^{rank}, is an implementation of Ranking SVM [21] (discussed in Section 2.6.1), a pairwise ranking algorithm that has been shown to improve search engine performance using clickthrough data. SVM^{rank}, is written in C and uses the SVM^{struct} library underneath - also written by Joachims.

The training data for the baseline model is preprocessed using the feature engineering outlined in Section 3.5. Scikit-learn's 'dump_svmlight_file' function is then used to transform the sparse output of the feature transformation pipeline into the libsvm format expected by SVM^{rank}.

3.7.1 Train-Test Split

Standard machine learning practice dictates that data is split into subsets, each dedicated to a different part of the model building process. The sets are training data - which is used by the model for the actual learning of its parameters; validation data - which is used by the model to tune its hyperparameters; and test data - which is used to evaluate how well the model generalizes on unseen data.

For the baseline model, no hyperparameter tuning is performed. Thus, the data is split into training and test sets that make up 80% and 20% of the data respectively.

3.8 Summary

This chapter outlines the details of the initial steps of the model building process. The attributes of the dataset are enumerated, and the engineering involved in acquiring the dataset and preprocessing it into the right format for the learning to rank models is described. Finally, the chapter describes the baseline learning to rank model used during performance evaluation. The next chapter describes the tree-based ranker and its implementation details.

Chapter 4

Tree-Based Ranker

This section describes the implementation of LTR models for place search using tree based frameworks, namely XGBoost [13] and LightGBM [25]. XGBoost and LightGBM are two popular libraries that implement Gradient Boosted Trees (GBT). The libraries implement LambdaMART [51], which improves upon the LamdaRank algorithm [9] with Multiple Additive Regression Trees (MART).

Preprocessing is a crucial step in building ML models that involves transforming source data into the structure and representation that is best for consumption by a model. The preprocessing that was applied to the data before feeding it into the tree-based models is discussed in this chapter. Scikit-Learn toolkit, pandas, and numpy are used to implement the preprocessing on the training data. This chapter begins with an introduction to XGBoost and LightGBM, and then moves on to details of the steps it took to build LTR models using these frameworks.

4.1 Gradient Tree Boosting

Gradient tree boosting is based on the idea of using an ensemble of weak learners to build a strong learner by successively boosting regression trees [16]. Gradient tree boosting has proven to be effective at solving a wide range of machine learning problems [32]. It can be understood by explaining the concepts of additive modelling, boosting, and learning.

Additive Modelling involves building a model or function that predicts a dependent variable by adding linear terms to a composite model.

Boosting is an application of the additive modelling idea, where simple models or weak learners are combined to create a strong learner or strong predictor. The weak learners are built sequentially, improving the model's performance at each iteration. In the case of gradient tree boosting, these weak learners are regression trees [16].

The weak learners improve the performance of the model by learning from the prediction accuracy of preceding learners. In gradient boosting, gradient descent is used to create weak learners trained on earlier learners by deriving gradient vectors that minimize a cost function. The gradient vectors are usually adjusted based on a learning rate.

The regression trees are trained by parametrizing them based on the gradient vector. Specifically, the weights of the leaves are adjusted based on the gradient vector. The prediction from the new tree is then added to the preceding trees to improve the final model recursively by computing new gradients based on the loss and adjusting the weights.

The regression trees have certain hyperparameters such as the maximum depth of the tree, minimum child weight, and the number of trees. The optimal values for these hyperparameters are found using hyperparameter tuning. The algorithm for gradient boosting is outlined as follows [32]:

Algorithm 1 Friedman's Gradient Boost algorithm

Inputs:

- input data $(x, y)_{i=1}^{N}$
- number of iterations M
- choice of the loss-function $\Psi(y, f)$
- choice of the base-learner model $h(x, \theta)$

Algorithm:

1: initialize $\widehat{f_0}$ with a constant
2: **for** $t = 1$ to M **do**
3: compute the negative gradient $g_t(x)$
4: fit a new base-learner function $h(x, \theta_t)$
5: find the best gradient descent step-size ρ_t:
$$\rho_t = \arg\min_{\rho} \sum_{i=1}^{N} \Psi\left[y_i, \widehat{f}_{t-1}(x_i) + \rho h(x_i, \theta_t)\right]$$
6: update the function estimate:
$\widehat{f_t} \leftarrow \widehat{f}_{t-1} + \rho_t h(x, \theta_t)$
7: **end for**

In gradient tree boosting, regression trees are used as the learner model $h\left(x, \theta\right))$; and in learning to rank, $\Psi\left(y, f\right)$ could be LambdaMART's listwise loss with significations modifications to the learning algorithm.

4.2 XGBoost

When XGBoost, which stands for **E**xtreme **G**radient **Boost**ing, was introduced in 2016, it gained immense popularity because of the improvements in scalability and computational efficiency it had over its alternatives. It improves upon Gradient Boosted Trees using various performance and algorithmic enhancements such as parallelization, regularization, and handling of sparse data; all of which are significant to our objective [13]. At the time of its publication, XGBoost's LambdaMART implementation achieved state-of-the-art performance on the Yahoo! learning to rank challenge dataset [11].

4.3 LightGBM

LightGBM was built with a similar premise of providing a performant implementation of Gradient Boosted Trees compared to the alternatives, especially when the feature dimension and dataset are large [25]. LightGBM introduced two techniques to accomplish this called Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB). GOSS is a tree splitting algorithm that considers all data instances with large gradients and randomly sample data instances with small gradients, achieving a similar information gain but with a smaller data size. EFB works by reducing the number of features by grouping mutually exclusive features [25]. The authors of LightGBM show that its implementation of LambdaRank surpassed the performance of XGBoost on the Microsoft Learning to Rank dataset [25] [37].

4.4 Model Input

The input for the tree-based models are preprocessed using the feature engineering procedure outlined in Section 3.5. The preprocessor outputs a scipy sparse matrix, which is supported as a model input by both XGBoost and LightGBM.

Another input required by the XGBoost and LightGBM models is a group information file. The group information file specifies the indices at which each query group begins in the dataset sorted in ascending order from the least query group ID to the highest. Recall that a query group is defined as a query occurrence and its corresponding result list, and each query group has an identifier or ID assigned to it. The group information file is formatted as follows:

```
index of query group 1
index of query group 2
.
.
.
index of query group n
```

4.5 Train-Validation-Test Split

Dataset splitting is usually done by randomly assigning samples to the sets based on predefined ratios. However, ranking data is categorized by result lists or query groups; therefore, the train-validation-test split is done by query groups instead of samples to retain the ranking semantics. In order to perform a query group aware split, a list of the unique query group IDs is extracted and randomly sampled from to match the ratio of the different subsets. The ratio used to split the data into train-validation-test sets is 80% of the entire dataset for the training and validation set, which is further split into 80% for the training set and 20% for the validation. The remaining 20% of the entire dataset is used as the test set.

4.6 Model Fitting

LightGBM and XGBoost both support the Scikit-Learn API, which allows their model functions to be composed into Scikit-Learn pipelines.

```
params = {'scale_pos_weight': scale_pos_weight, 'objective': 'rank:map', '
                                        num_round': 500,
               'min_child_weight': 0.1, 'max_depth': 6, 'learning_rate': 0.1}

model = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('ranker', xgb.sklearn.XGBRanker(**params)) # or ('ranker', lgb.LGBMRanker
                                            (**params))
], verbose=True)

params_fit = {'ranker__group': group_train,
                      'ranker__eval_group': [group_valid]}

params_fit['ranker__eval_set'] = [
    (Pipeline(model.steps[:-1]).fit(X_train).transform(X_valid), y_valid)
]

ranker = model.fit(X_train, y_train, **params_fit)
```

The model is instantiated with predeclared hyperparameters. Then it is placed into a pipeline along with the pipeline of column transformers described in Section 3.6. In the model input section, we

note that the models require information on query group indices. The arrays of query group indices for the training and validation sets are specified as fitting parameters for the model. Finally, the model is trained by simply calling the 'fit' function.

4.6.1 Hyperparameter Tuning

XGBoost and LightGBM rankers accept hyperparameters that determine factors such as the number of successive trees trained on, the maximum depth for each tree, the learning rate, how much a negative sample should be weighted, etc. These hyperparameters have an impact on the performance of the trained model; therefore, it is important to tune them to achieve the best set of parameters given time and resource constraints. Scikit-Learn supports two strategies for hyperparameter tuning, namely: exhaustive grid search and randomized parameter optimization.

XGBoost and LightGBM's APIs don't support the use of Scikit-Learn's hyperparameter tuning classes, but a grid search function can easily be implemented using Scikit-Learns ParameterGrid class and an iterator. The training time for a single model with the dataset at hand is approximately fifteen (15) minutes, therefore, the duration of the hyperparameter search is equal to the number of permutations multiplied by the training duration. The hyperparameter search space grows exponentially with the number of hyperparameters to tune; therefore, a considerably limited grid of values was defined for the hyperparameters to constrain the search space. The following hyperparameters were tuned:

- Objective function
- Scaling of positive weights
- Number of rounds i.e. number of trees.
- Minimum child weight
- Maximum tree depth

4.7 Summary

This chapter delivered the background information and implementation details of the tree-based learning to rank model. The theoretical basis of XGBoost and LightGBM, frameworks that implement gradient tree boosting was discussed. Subsequently, the different aspects of the model's implementation were described to provide clarity on the modelling process. The next chapter discusses neural rankers using a similar approach to this chapter.

Chapter 5

Neural Ranker

A neural ranker is a ranking model built using a neural network. The application of LTR in building neural rankers is an emerging area of research [30] [56]. This chapter discusses the implementation of a neural ranker, starting with an exposition of the framework used, the preprocessing steps, and neural network architecture.

Our neural ranker is implemented using Tensorflow Ranking, a framework that is itself implemented on top of TensorFlow [1].

5.1 TensorFlow

TensorFlow is a popular framework for building large scale machine learning and deep learning models. The basic construct in TensorFlow are tensors, which can be described as multidimensional array containers that can be used to express complex logic and manipulated using tensor operations into what is known as a computational graph. The tensors in a computational graph or nodes as they are called, along with tensor operations, are used to represent the input space, hypothesis space, and output space of a TensorFlow model. The computational graph in TensorFlow is most popularly used to create deep neural networks, with nodes in the computational graph representing the input, neurons, activation functions, and output of the neural networks. Tensor operations enable the arithmetic required to compute values as they flow through the network. TensorFlow supports backpropagation through a mechanism known as automatic differentiation, where each tensor operation can compute the gradient for its input tensor.

5.2 TensorFlow Ranking (TF-Ranking)

TensorFlow Ranking was developed to be a scalable solution for building *learning to rank* models on large datasets with high dimensionality and sparsity. TensorFlow Ranking achieves its goal by leveraging the underlying support for building scalable and robust neural network models that comes with TensorFlow. TensorFlow is also able to take advantage of the feature engineering functionality of TensorFlow that enables the handling of textual features using text embeddings. TF-Ranking supports recent advances in Learning to Rank such as unbiased learning to Rank [23], LambdaLoss framework [50], and multi-item scoring [5].

5.2.1 Multi-Item Scoring

Multi-item scoring was introduced in [5] as a method for scoring documents that deviates from the method used by pointwise, pairwise, and listwise learning to rank approaches. Models built on standard LTR approaches score documents using univariate functions $f : \chi n \rightarrow \mathbb{R}^1$, which

doesn't factor in the relevance of other documents in a result list. Multi-item scoring involves using multivariate scoring functions $f : \chi n \rightarrow \mathbb{R}^n$ to produce an ordering of entire result lists. The framework for this method is known as Groupwise Scoring Functions (GSF) [5]. GSFs are parameterized using deep neural networks (DNN) due to their ability to scale to high-dimensional and highly sparse features [5]. A GSF can be represented as:

$$g\left(.;\theta : \chi^m \rightarrow \mathbb{R}^m\right)$$

where θ the parameters of a DNN, and χ are the documents of size m that are scored to produce a vector of size m.

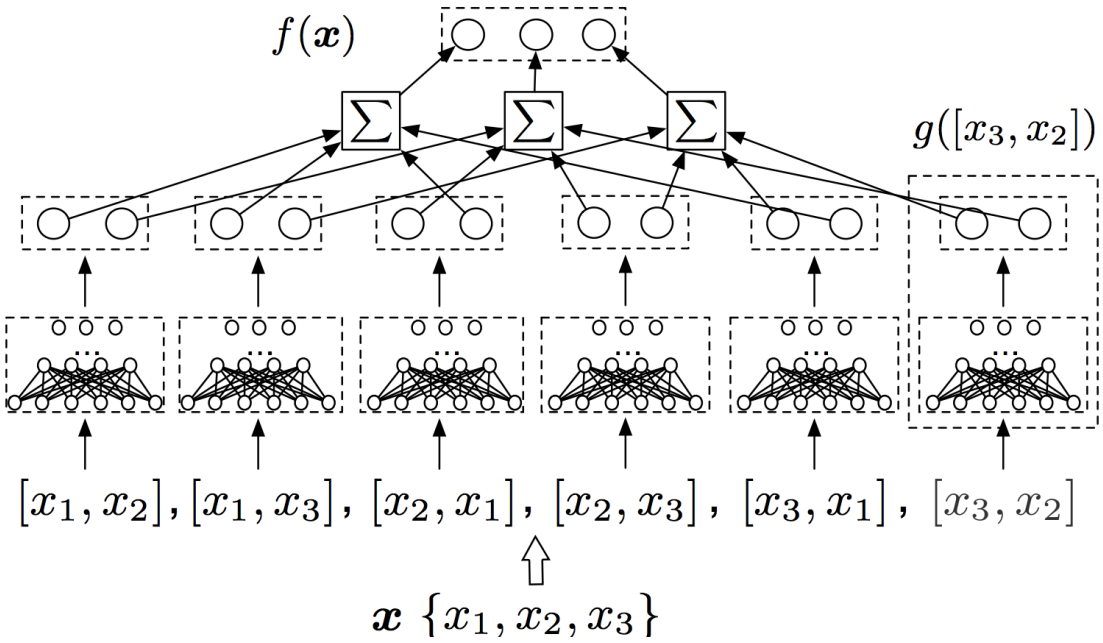

Figure 5.1: Illustration of how a GSF can be used, in this case as the bivariate function $g(.)$ [5]

The performance of GSFs has been shown to be comparable to state-of-the-art tree-based LTR models. A GSF-based ranker can be implemented constructed using TF-Ranking by calling the 'make_groupwise_ranking_fn' function like such:

```
import tensorflow_ranking as tfr

tfr.model.make_groupwise_ranking_fn(
  group_score_fn=make_score_fn(),
  group_size=_GROUP_SIZE,
  transform_fn=make_transform_fn(),
  ranking_head=ranking_head)
```

'make_score_fn' in the snippet above is a wrapper function over a function that fits a standard TensorFlow NN model and returns its scores or logits for the input. It is important to note that GSF is generalization of the pointwise and pairwise scoring functions. Pointwise scoring can be used by passing a value of one as the 'group_size' argument to the 'tfr.model.make_groupwise_ranking_fn' function. The value of group size significantly impacts training time, higher values resulting in longer training times.

5.2.2 LambdaLoss Framework

LambdaLoss is a framework for metric optimization using a principle probabilistic approach [50]. LambdaLoss provides a solution that allows for the direct optimization of query-level position-based ranking metrics, which are otherwise flat and discontinuous. Even though LTR approaches such as pairwise and listwise adopt loss functions that are smooth and continuous[37], the bounds of the functions are coarse because they don't directly optimize ranking metrics. The implementation of LambdaMART using the LambdaLoss framework achieved better performance compared to state-of-the-art LambdaMART implementations [50]. The standard loss functions, e.g. logistic and softmax loss, in TF-Ranking are implemented using the LambdaLoss framework.

5.2.3 TensorFlow Ranking Architecture

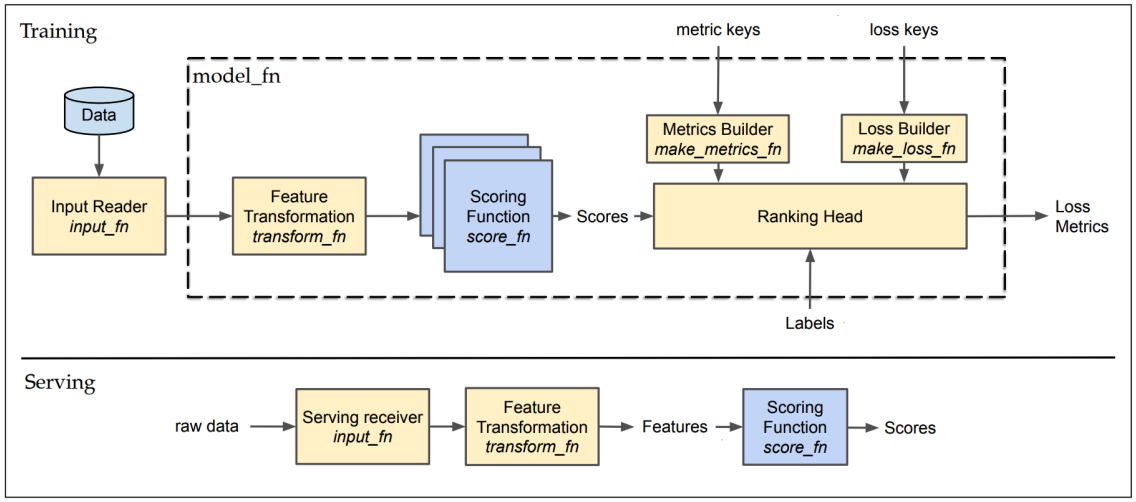

Figure 5.2: Tensorflow Ranking Architecture

TensorFlow Ranking's architecture consists of four main parts: an input reader, a feature transformer, a scoring function, and a ranking head. The input reader is responsible for fetching data from a data source and making it available as a properly structured TensorFlow dataset (see Section 5.3 on structuring data for TF-Ranking). The feature transformer is a function that creates feature columns out of the input functions output. TF-Ranking provides an estimator API, and estimators expect to receive data that has been formatted as feature columns such as numerical columns and categorical (one-hot encoded) columns. Next is the scoring function, which specifies the neural network that would generate scores or logit tensors during training. The scoring function specifies properties of the network such as the input layer, hidden layer, output layer, dropout, batch normalization, activation function, etc. Finally, the ranking head encapsulates functions that compute the training loss and evaluation metrics. Section 5.5 describes how the different components of the TF-Ranking framework are combined to build an LTR model. The decomposition of TF-Ranking into components provides modularity, which makes it easy to experiment with different LTR approaches and model configurations.

5.3 Model Input

TensorFlow Ranking provides an Estimator workflow, which is a convenient encapsulation of models such that once an estimator is instantiated, it can be easily trained, evaluated, and invoked to make predictions. The functions in estimators' standard API, i.e., train, evaluate, and predict

expect an input function that acts as a data source, when called. The following code snippet is an example of a function that can be passed as a thunk to the TF-Ranking estimator's train function:

```
import tensorflow as tf

def df_to_dataset(dataframe, batch_size=_BATCH_SIZE):
  dataset = tf.data.Dataset.from_generator(
    lambda: generator(dataframe),
    output_types = ({
        'example_features': tf.float32,
        'context_features': tf.float32
      },
      tf.float32
    ),
    output_shapes = ({
        'per_item_feature': tf.TensorShape([_LIST_SIZE, _NUM_PER_ITEM_FEAT_COLS
                                            ]),
        'context_feature': tf.TensorShape([_NUM_CONTEXT_FEAT_COLS])
      },
      tf.TensorShape([_LIST_SIZE])
    )
  )

  return dataset.repeat().batch(batch_size).make_one_shot_iterator().get_next()
```

The function 'df_to_dataset' accepts a data frame, creates a generator that allows for flow control by the estimator, and returns the next iteration of values from a TensorFlow dataset, which is a tuple of features and labels. The features variable in the tuple is a dictionary with feature names as keys and tensors as values. The dimensions of the feature tensors depend on the type of feature. TensorFlow ranking makes a distinction between two types of features: *per-item features* and *context features*. *Per-item* features are features that are specific to each item or place document in a query group, such as the name and city of a place, its coordinates. A *per-item* feature could also be derived from a relationship between the query and place attribute, for example, the distance between a user and a place. Context features, on the other hand, are generic to a query group, for example, user location and timestamp. A per-item feature is represented using a 3-D tensor where the dimensions in order are: batch size (for Stochastic Gradient Descent), number of items in the query group - fixed size of the result list, and number of columns for the feature - "1" in the case of scalar values or "1 x n" dimensions in the case of vectors. A context feature tensor is represented using 2-D tensor where the dimensions in order are: batch size and number of columns for the feature. Lastly, labels returned from the input function is a 2-D tensor where the dimensions are: batch size and list size.

List size is the number of items in a query group. The number of items in a query group, in turn, is the number of documents returned by the search engine, which can be different from one query to another. TF-Ranking expects a fixed list size, therefore, the query group needs to be padded or trimmed down depending on whether it is smaller or larger than the list size.

TensorFlow estimators train models in steps, where each step has a batch size. In the case of TF-Ranking, a batch size of 32 means 32 query groups or 32 result lists. The dimensions of per-item features and context features imply that 32 * 15 = 480 rows of per-item features are used in a single step, while only 32 rows of context features in used. The separation of features into the two types can be seen as an efficient way of representing the input data.

text
focus point latitude
focus point longitude
timestamp
target

Table 5.1: Context Features

name
locality
neighbourhood
borough
place latitude
place longitude
place type

Table 5.2: Example Features

5.3.1 Train-Test Split

The data is split into 80% and 20% for the training and test sets respectively. Unlike the tree-based models, no validation set is allocated. Validation sets are used to tune hyperparameters; which requires multiplying the training time by the number of hyperparameter combinations. This is only feasible with a large number of computing resources.

5.4 Model Fitting

Fitting a neural ranker using involves putting the different TF-Ranking components together. The following code snippet describes how to instantiate a ranker using the TF-Ranking estimator.

```
1  import tensorflow as tf
2  import tensorflow_ranking as tfr
3
4  def get_estimator():
5      """Create a ranking estimator."""
6      def _train_op_fn(loss):
7          """Defines train op used in ranking head."""
8          return tf.contrib.layers.optimize_loss(
9              loss=loss,
10             global_step=tf.train.get_global_step(),
11             learning_rate=_LEARNING_RATE,
12             optimizer='Adagrad')
13
14     ranking_head = tfr.head.create_ranking_head(
15         loss_fn=tfr.losses.make_loss_fn(_LOSS),
16         eval_metric_fns=eval_metric_fns(),
17         train_op_fn=_train_op_fn)
18
19     return tf.estimator.Estimator(
20         model_fn=tfr.model.make_groupwise_ranking_fn(
21             group_score_fn=make_score_fn(),
22             group_size=_GROUP_SIZE,
23             transform_fn=make_transform_fn(),
```

```
            ranking_head=ranking_head),
        model_dir='/tmp/tf-ranking_model',
        params=None)

ranker = get_estimator()
```

5.4.1 Training Op

First, a function to specify the training operation is defined. The training operation is executed after logits and losses are computed at the end of a forward pass. Thus, it accepts a loss tensor as an argument. The training operation involves optimizing a loss based on the global step, which informs TensorFlow on what epoch to resume training - helpful if training is interrupted for any reason; the learning rate; and lastly, the optimization algorithm. The Adagrad algorithm is used in all our experiments. Adagrad adapts the learning rate based on the frequencies of features' appearances, performing larger updates for infrequent features and smaller updates for infrequent features. Adagrad has been shown to perform well on sparse dataset [15], which is a particularly useful attribute for our use case.

5.4.2 Ranking Head

Next, the ranking head is constructed by supplying the training operation, evaluation functions, and a loss function. The loss function can be any one of several loss functions provided by TensorFlow ranking, which include pairwise logistic loss, pairwise hinge loss, approximated mean reciprocal rank (MRR) loss, etc. The loss functions are implemented using the LambdaLoss framework. The evaluation metrics are used when the estimator is run in 'evaluate' mode. The 'eval_metric_fns()' function returns a dictionary of key-value pairs, where the keys are user-defined names for the metrics, and the values are functions provided by the TensorFlow ranking framework. The available evaluation metrics include Average Relevance Position (ARP), NDCG, MRR, and precision@k.

5.4.3 Model Function

Lastly, the 'tf.estimator.Estimator' constructor accepts a model function. The aforementioned Groupwise Scoring Function (GSF) is used, and it accepts the group size that determines if scoring will be pointwise, pairwise, or listwise; a transform function that transforms context and per-item features to TensorFlow feature columns; and a score function that determines how logits will be computed. The score function does this by specifying the neural network that makes our ranker a neural ranker.

```
import tensorflow as tf
import tensorflow_ranking as tfr

def make_score_fn():
  """Returns a scoring function."""

  def _score_fn(context_features, group_features, mode, params, config):
    """Defines the network to score a group of documents."""
    with tf.compat.v1.name_scope("input_layer"):
      context_input = [
          tf.compat.v1.layers.flatten(context_features[name])
          for name in sorted(context_feature_columns())
      ]
      group_input = [
```

```
          tf.compat.v1.layers.flatten(group_features[name])
          for name in sorted(example_feature_columns())
      ]
      input_layer = tf.concat(context_input + group_input, 1)

    is_training = (mode == tf.estimator.ModeKeys.TRAIN)
    cur_layer = input_layer
    cur_layer = tf.compat.v1.layers.batch_normalization(
      cur_layer,
      training=is_training,
      momentum=0.99)

    for i, layer_width in enumerate(int(d) for d in _HIDDEN_LAYER_DIMS):
        cur_layer = tf.compat.v1.layers.dense(cur_layer, units=layer_width)
        cur_layer = tf.compat.v1.layers.batch_normalization(
            cur_layer,
            training=is_training,
            momentum=0.99)
        cur_layer = tf.nn.tanh(cur_layer)
        cur_layer = tf.compat.v1.layers.dropout(
            inputs=cur_layer, rate=_DROPOUT_RATE, training=is_training)
    logits = tf.compat.v1.layers.dense(cur_layer, units=_GROUP_SIZE)
    return logits

  return _score_fn
```

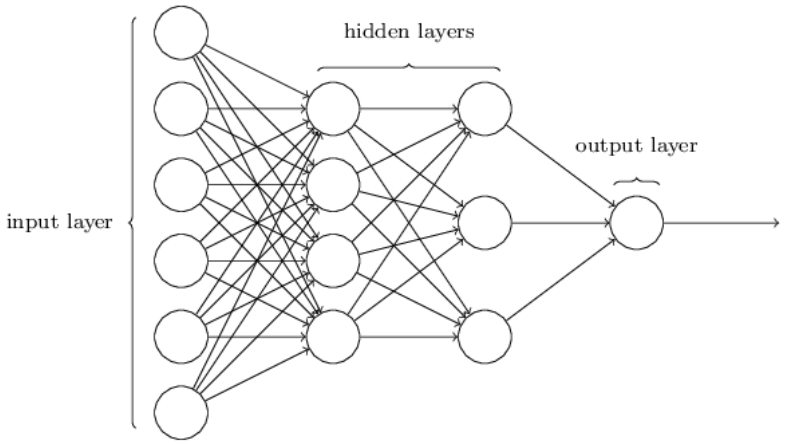

Figure 5.3: Architecture of Feed Forward Neural Network

The neural network architecture determines how the input of a neural network is transformed into its output. This includes the structure of the input fed into the network; the activation function at each layer applied to the weighted input from the preceding layer; and the dimensionality of the output layer. The only type of neural network considered is the feed forward neural network.

Input Layer

```
context_input = [
    tf.compat.v1.layers.flatten(context_features[name])
    for name in sorted(context_feature_columns())
]

group_input = [
    tf.compat.v1.layers.flatten(group_features[name])
    for name in sorted(example_feature_columns())
]

input_layer = tf.concat(context_input + group_input, 1)
```

The input layer of the neural ranker is a flattened and concatenated vector of the context and per-item features.

Hidden Layers

```
_HIDDEN_LAYER_DIMS = ['512', '256', '32']

 for i, layer_width in enumerate(int(d) for d in _HIDDEN_LAYER_DIMS):
        cur_layer = tf.compat.v1.layers.dense(cur_layer, units=layer_width)
```

The hidden layers of a neural network are the layers between the input layer and output layer. In the code snippet above, we are specifying three hidden layers, with the number of units in the layers set as 512, 256, and 32 in successive order.

Activation Function

```
cur_layer = tf.nn.relu(cur_layer)
```

The activation function used is ReLU or rectified linear unit, other loss functions such as Leaky ReLU, tanh or hyperbolic tangent are available via the TensorFlow API.

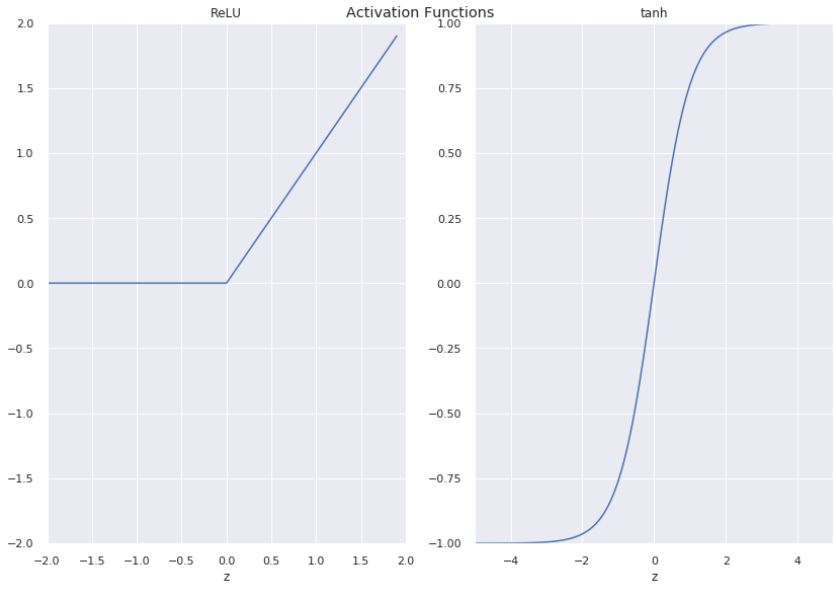

Figure 5.4: Activation functions using ReLU vs. tanh

Figure 5.4 shows how activation functions transform their weighted inputs differently; the primary difference being their effect on how partial derivative of the input changes with respect to

the weight and bias.

Output Layer

```
logits = tf.compat.v1.layers.dense(cur_layer, units=_GROUP_SIZE)
```

The output layer is the logits computed during a forward pass of the network, with dimensionality equal to the group size of the groupwise scoring function.

Regularization

```
_DROPOUT_RATE = 0.5

cur_layer = tf.compat.v1.layers.dropout(
            inputs=cur_layer, rate=_DROPOUT_RATE, training=is_training)
```

Regularization is a set of techniques used during the training of machine learning and deep learning models, so that the models generalize on unseen data [17]. A model that performs well on training data, but does not generalize well to unseen data, is exhibiting a phenomenon called overfitting. Neural networks are particularly susceptible to overfitting due to their typically large capacity, which means they could simply memorize the weights and biases that minimize the training loss. There are several regularization techniques for avoiding overfitting, such as L2 regularization [17] and dropout [45].

Dropout is a method for preventing overfitting that involves randomly dropping units and their connections during training. Dropout prevents units from excessively co-adapting [45]. In the simplest form of dropout, a probability p (a hyperparameter) is used to determine whether a unit is kept or not. Srivastava et al. found p =0.5 to work for a wide range of use cases. However, the optimal value of p is found through a hyperparameter search. For the neural ranker, the recommended value of 0.5 was used.

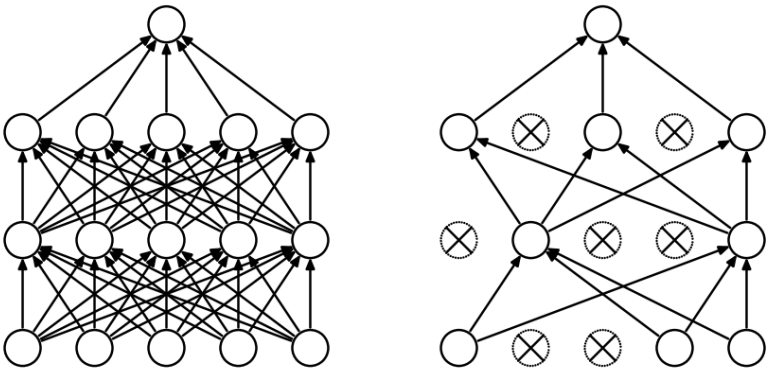

Figure 5.5: An illustration taken from [45] that shows on the left: a network before applying dropout; and on the right: a thinned network after applying dropout.

Batch Normalization

```
cur_layer = tf.compat.v1.layers.batch_normalization(
            cur_layer,
            training=is_training,
            momentum=0.99)
```

Batch normalization is used to correct for a phenomenon known as internal covariate shift, where the learning rate of a neural network model becomes unstable due to values used at parameter initialization and changes in the distributions of input parameters at different layers [20]. Batch

normalization acts as a form of preprocessing at each layer it is defined, where it either normalizes the output of a previous layer's activation function or input to a subsequent layer by subtracting the batch mean and dividing by the batch standard deviation.

The mean and standard deviation statistics used to normalize batches are usually moving averages of statistics across several batches. As a result, the 'momentum' argument to the 'batch_normalization' function is used to determine how much importance is given to the statistics from earlier mini-batches. The momentum is commonly set as a large value such as 0.99, which is also the default in TensorFlow.

Choosing a Neural Network Architecture

The constituents of this architecture are the key determinants of whether a neural network can approximate a target function. According to the Universal Approximation Theorem [18], a single layer network network can approximate any continuous function subject to some constraints on the activation function. Even though the universality theorem assures us of the applicability of a neural network to approximate our ranking function, searching for the neural network architecture that offers the best approximation is not an easy task. Neural architecture search [57] is a field of study that explores automated ways of generating state-of-the-art neural network architectures. Neural architecture search can be thought of similar to the hyperparameter search methods of our tree based models. For the neural ranker, the search space would be limited to the network type being considered for the problem, i.e. feedforward neural networks.

5.5 Summary

This chapter introduced neural rankers and provided brief explanations of some neural network concepts. TF-Ranking, the framework used to implement the neural rankers in this work, and its various components and their interactions are also described. The next chapter will share the findings that support the argument behind this thesis.

Chapter 6

Results and Discussion

In Section 2.2, we discussed several metrics that are used to evaluate the performances of learning to rank models. The metric chosen for evaluating the models in this thesis is the Mean Reciprocal Rank (MRR). The implementations of the evaluation metric applied is the same across all models, and the same test data is used based on an 80% and 20% split between training data and test data, respectively. Furthermore, query groups containing only a single result were excluded from the evaluation since the model would always perform perfectly for those cases.

This chapter begins with comparisons of model performances for the evaluation metric; then proceeds to discuss the performances of tree-based rankers and neural rankers from different aspects. The chapter wraps up with an in-depth comparison of the best performing model against the Pelias search engine.

6.1 Model Comparisons

As discussed in Chapter 1, LTR models are top-k re-rankers of results pre-fetched by the Pelias search engine. Since the objective is to examine the potential improvements of using LTR models as re-rankers, the performance of the search engine is included in the comparisons. Also, the results list a linear ranker and a deep neural net ranker separately; even though both are implemented using TF-Ranking.

Model	MRR
Pelias	0.8493
SVM^{rank}	0.7415
XGBoost	0.8754
LightGBM	**0.8922**
TF-Ranking (Linear)	0.8675
TF-Ranking (Deep)	0.8559

Table 6.1: MRR scores for learning to rank models and Pelias.

The MRR scores for the models show that the tree-based models perform better than the Pelias and the other learning to rank models. TF-Ranking's neural ranker and linear ranker show marginal improvements over Pelias, with SVM^{rank} performing worse than Pelias.

Figure 6.1 shows the position rank distributions used to calculate MRR. SVMrank is an exception with its heavier tail compared to the other models, while the other models exhibit a power law

distribution. Based on the results of the relevant metric, the LightGBM tree-based model offers the best improvement as a top-k re-ranker on top of Pelias.

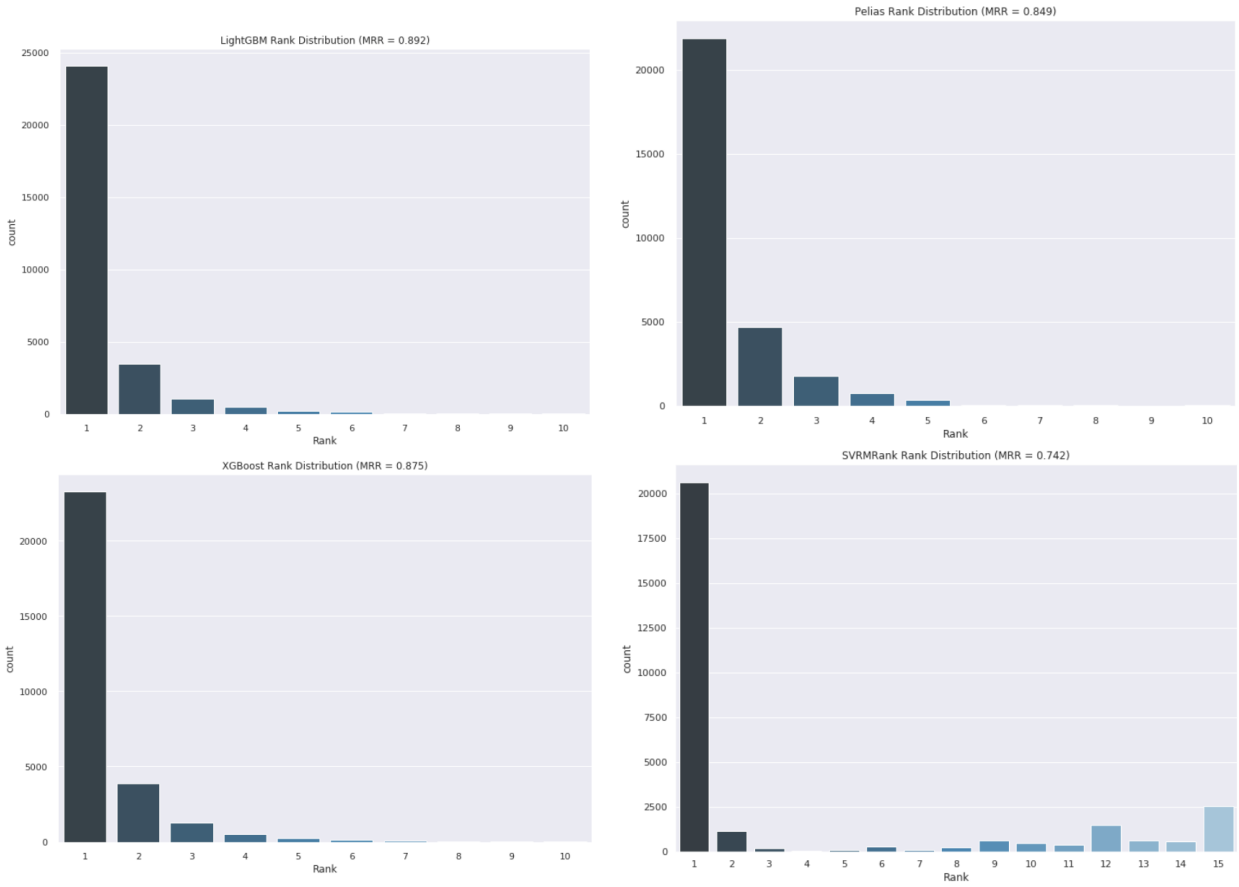

Figure 6.1: Distributions of ranks for pelias and learning to rank models.

6.2 Tree-Based Models

6.2.1 Hyperparameter Tuning

As mentioned in Section 5.4, hyperparameter tuning was applied to the tree-based models to find the set of parameters that yielded the best model. Table 6.1 shows that hyperparameter tuning provides a marginal improvement over the default configuration for both XGBoost and LightGBM models, with the XGBoost model showing the more significant improvement.

Model	MRR
XGBoost (Default)	0.8531
XGBoost (HP Tuned)	0.8754
LightGBM (Default)	0.8826
LightGBM (HP Tuned)	0.8922

Table 6.2: MRR scores for tree-based models with and without hyperparameter tuning.

6.2.2 Feature Importances

An advantage of tree-based models is the ease of extracting feature importances, and both XGBoost and LightGBM provide convenience functions for extracting feature importances. However, the default data preprocessing described in Section 3.5 involves the use of a hashing vectorizer on textual features such as the query text. A side-effect result of using the hashing vectorizer is that feature names in the data matrix become indices that cannot be transformed back to feature names. Since the count vectorizer is similar to the hashing vectorizer with a difference being that it supports retrieving feature names, a model was built using the count vectorizer for the sole purpose of inspecting feature importances.

Weight	Feature
0.1574	x0_stop
0.0173	he
0.0173	he
0.0173	he
0.0138	dur
0.0138	dur
0.0138	dur
0.0130	ruh
0.0130	ruh
0.0130	ruh
0.0111	(sp
0.0111	(sp
0.0079	markt
0.0079	markt
0.0053	baden
0.0053	baden
0.0053	baden
0.0050	mark
0.0050	mark
0.0048	ruchs

Figure 6.2: Feature importances of the top 20 features overall.

Weight	Feature
0.1574	x0_stop
0.0036	x0_address
0.0010	x0_poi
0.0005	place_lon
0.0005	place_lat
0.0003	focus_lat
0.0002	focus_lon
0.0000	month_sin
0.0000	hr_cos
0.0000	hr_sin
0.0000	day_sin
0.0000	is_weekend
0.0000	day_cos
0	month_cos

Figure 6.3: Feature importances of non-textual features.

Figure 6.2 shows the dominance of textual features compared to categorical and numerical features in terms of feature importance. The most important feature, which is the exception, is whether a location is a public transport stop. A cursory analysis of the clickthrough data depicted in figure 6.4 shows that transport stops are disproportionately the most selected type of place even though other types of places appear almost as often in search results. The importance of transport stops could be attributed to several reasons, two of which are; 1) the Moovel app as of the time of writing always places stops as the first section of results 2) the majority of the app's users use it for public transportation. Further analysis is required to ascertain the degree to which unintended bias affects the search results and click-through, as this bears relevance to the quality of models that can be learned from the data.

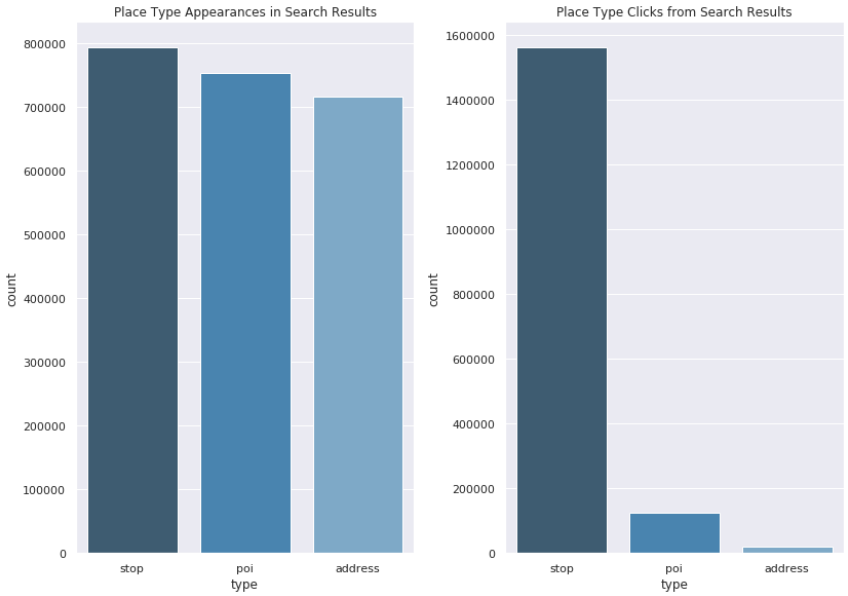

Figure 6.4: Place Type Counts.

6.3 Neural Ranker

6.3.1 Comparisons

Several neural network architectures for the neural ranker were evaluated. The neural networks varied in terms of the: 1) loss function, 2) number of units in the hidden layers, 3) number of hidden layers, 4) activation function, 5) scoring function, and 6) dropout rate.

Model	MRR
Linear (Sigmoid) (0 hidden layers)	**0.8675**
FC-64-32-16: 80% dropout	**0.8559**
FC-32 (shallow network, 80% dropout)	0.8523
FC-512-256-32 (groupwise with size=5)	0.8247
FC-512-256-32	0.8070
FC-512-256-32 (tanh)	0.8039
FC-64-32-16	0.8038
FC-512-256-32 (approximate MRR loss)	0.7790
Non-Linear (ReLU) (0 hidden layers)	0.7732

Table 6.3: MRR scores for neural rankers.

In Table 6.3, deep neural networks are prefixed with FC (full-connected) followed by a list of the number of units in each hidden layer. For example, FC-64-32-16 has 3 hidden layers with 64, 32, and 16 units in that order. Each neural network in the table has the following default values unless stated otherwise:

- Loss function: pairwise logistic loss
- Number of hidden layers: 3
- Activation function: ReLU

- Scoring function: pointwise
- Dropout rate: 50% dropout

The MRR scores in Table 6.3 show that linear model has the highest MRR score compared to the deep neural networks. The linear model is a one-layer neural network with a sigmoid for the activation function. In essence, the linear model or classifier is a logistic regression model with a pairwise loss. The reason behind the performance of this model is not investigated within this thesis. One hypothesis could be that gradient descent performs well due to the convexity of the logistic loss enabled by the use of LambdaLoss.

Among the deep neural networks, FC-64-32-16 with a dropout rate of 80% has the highest MRR score of 0.8559. A similar model; FC-64-32-16 with the default dropout rate of 50% has an MRR score of 0.8038. That is a significant difference based on the dropout rate, which indicates that FC-64-32-16 (50% dropout) is overfitting and is improved by using a more aggressive regularization. Other than the dropout rate, FC-64-32-16 (80% dropout) also has a much lower capacity compared to the other deep neural networks such as FC-512-256-32. Similarly, this indicates that the larger networks with the same configuration and default dropout are overfitting.

In terms of the scoring function, the model using a groupwise function showed an improvement in MRR over models using pointwise scoring functions with the same capacity. Also noteworthy is the worse performance of the approximate MRR loss compared to the pairwise logistic loss.

6.3.2 Training Loss

The following plots show the loss functions and global norms of the gradients at each training step for the TF-Ranking models

Linear (Sigmoid)

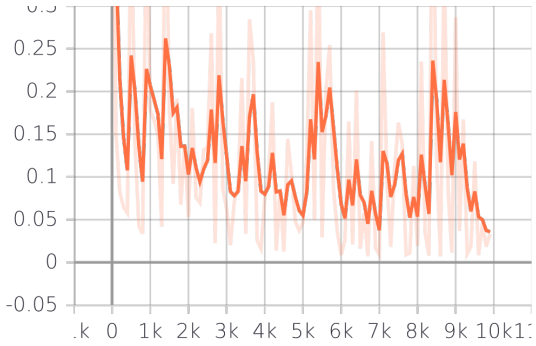

Figure 6.5: Loss at final step: 0.0202

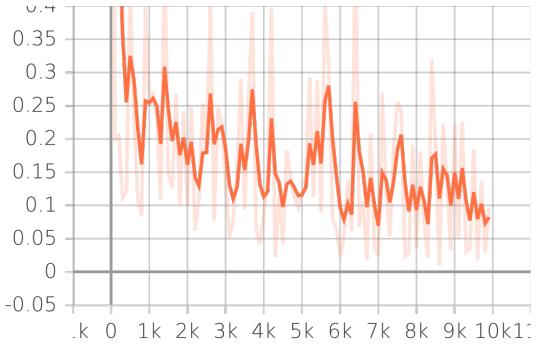

Figure 6.6: Gradient Norm at final step: 0.0961

FC-64-32-16 (80% Dropout)

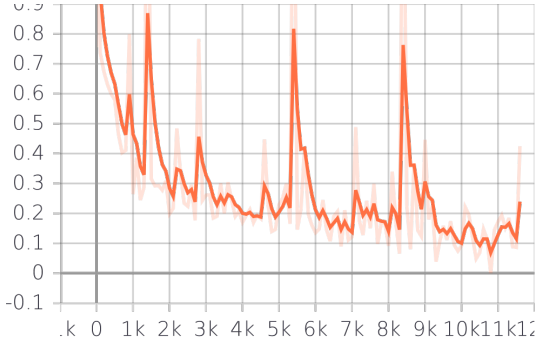

Figure 6.7: Loss at final step: 0.1386

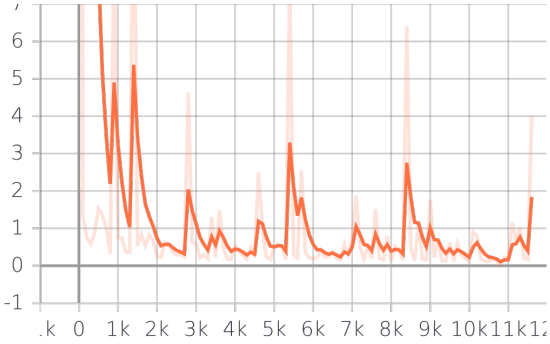

Figure 6.8: Gradient Norm at final step: 4.0069

FC-512-256-32

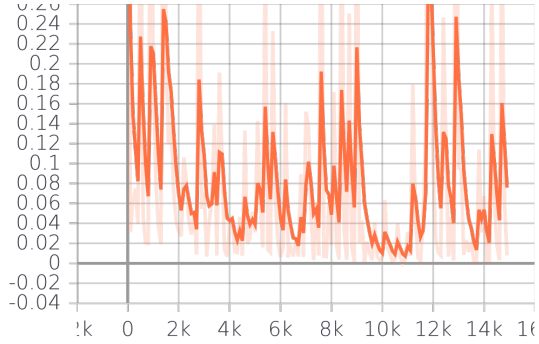

Figure 6.9: Loss at final step: 0.0747

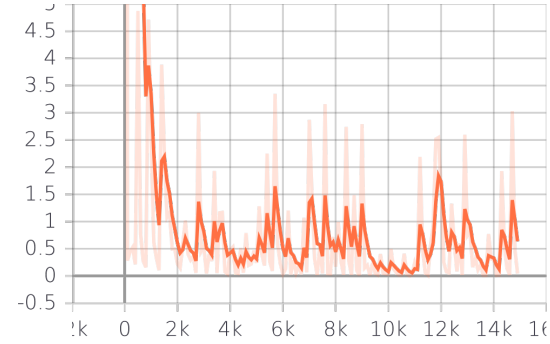

Figure 6.10: Gradient Norm at final step: 0.0435

FC-512-256-32 (tanh)

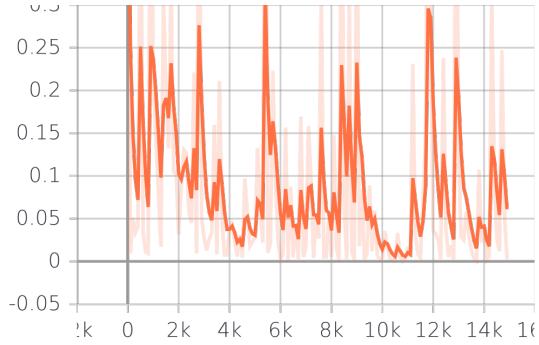

Figure 6.11: Loss at final step: 0.0748

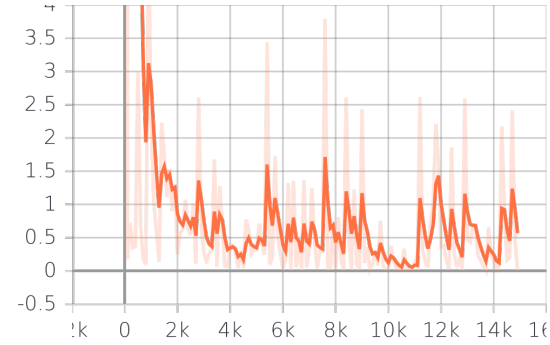

Figure 6.12: Gradient Norm at final step: 0.0180

FC-512-256-32 (MRR Loss)

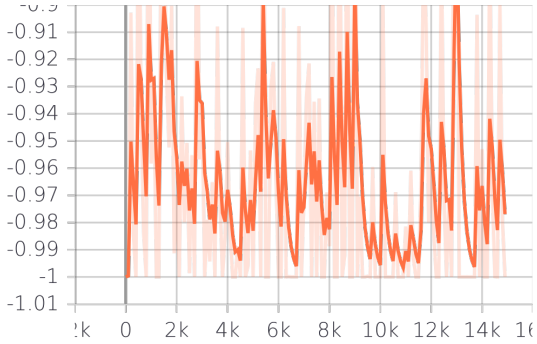

Figure 6.13: Loss at final step: -0.9067

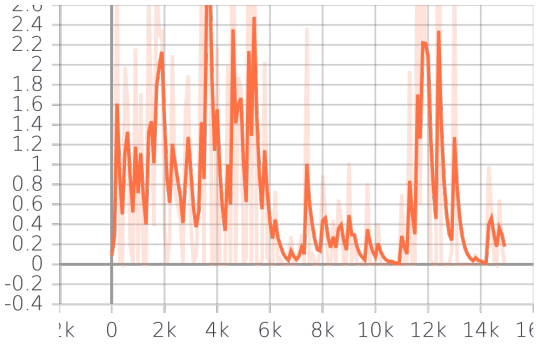

Figure 6.14: Gradient Norm at final step: 0.0

It is well known that the norm of the gradients with respect to the parameters of a neural network oscillates throughout the training. It is commonly assumed that as the network converges, the difference in the sizes of the oscillations becomes smoother [33].

It can be noted that the gradient norm for the linear model (Figure 6.6) is oscillating within a much smaller range compared to the deep neural networks. It also illustrates a more consistent decrease in gradient norm over the course of training. For the deep neural networks, FC-64-32-16 (80% dropout) (Figure 6.7) shows a smoother gradient norm curve compared to the others.

The pairs of loss and gradient norm charts show similar trends, demonstrating their correlation. This correlation between the two can be exploited by factoring the size of the gradient norm oscillations into the stopping criteria for training [33]. An area of improvement for the neural ranking models based on this idea and the charts above is to incorporate early stopping, a form of regularization.

6.4 Comparison against Pelias

6.4.1 Location Sharing

In Section 1.2.5, it was shown that users who do not share their location tended to click at lower rank positions compared to users who did. To assess the performance of the LightGBM model in terms of delivering a better ranking without knowing the location of users, the location feature was entirely removed from the test data set. The evaluation of the learning to rank model on this modified test data is shown in Figure 6.15.

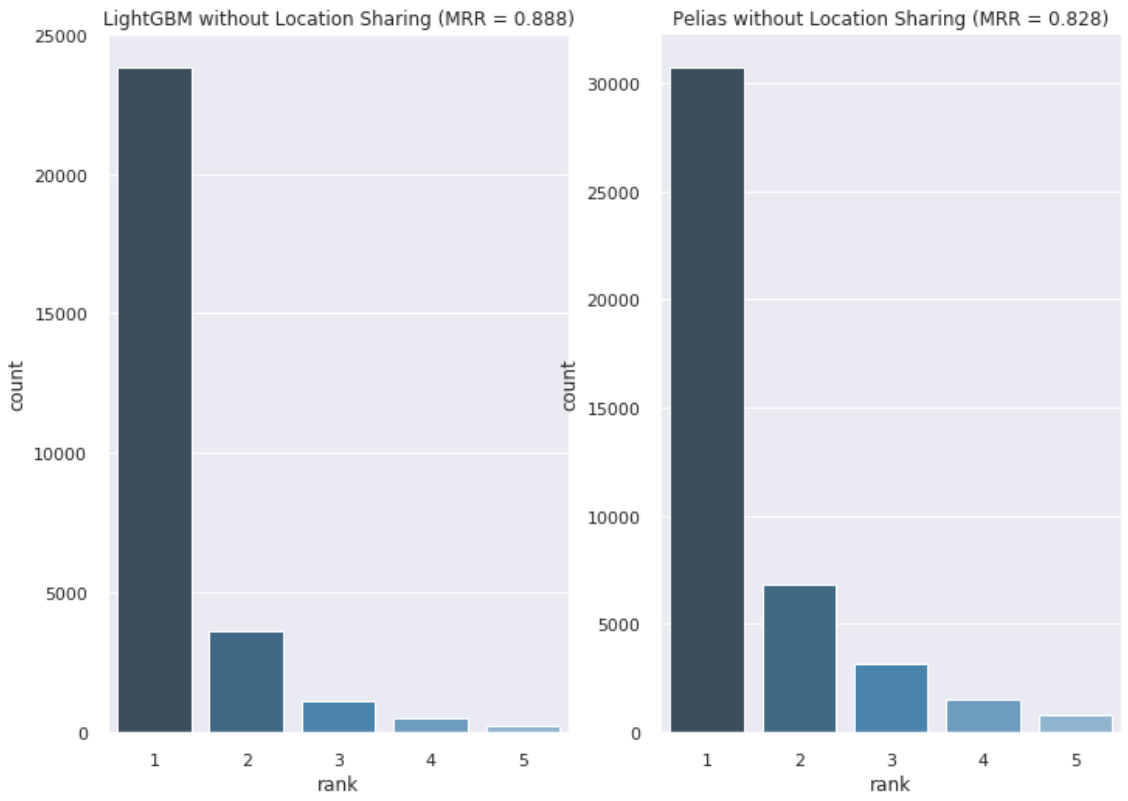

Figure 6.15: Rank distributions and MRR for Pelias vs. LightGBM when no location is shared.

LightGBM's MRR score of 0.8922 on the original test set is still higher than its score of 0.888 on the test data without location due to the importance of the location feature. However, this demonstrates a significant improvement over the original ranking delivered by Pelias.

6.4.2 Query Parsing

The challenge of query parsing is discussed in Section 1.2.5 as one of the limitations of Pelias. Query parsing in Pelias involves extracting entities from queries and using them to match on the corresponding fields. The rankers being evaluated all used character n-gram analyzers to preprocess the textual features of the dataset. In Figure 6.16, the LightGBM model shows a more evident worsening of rank positions with longer queries compared to Pelias. This is further illustrated in 6.4, where different MRR scores are calculated for specific ranges of query character lengths.

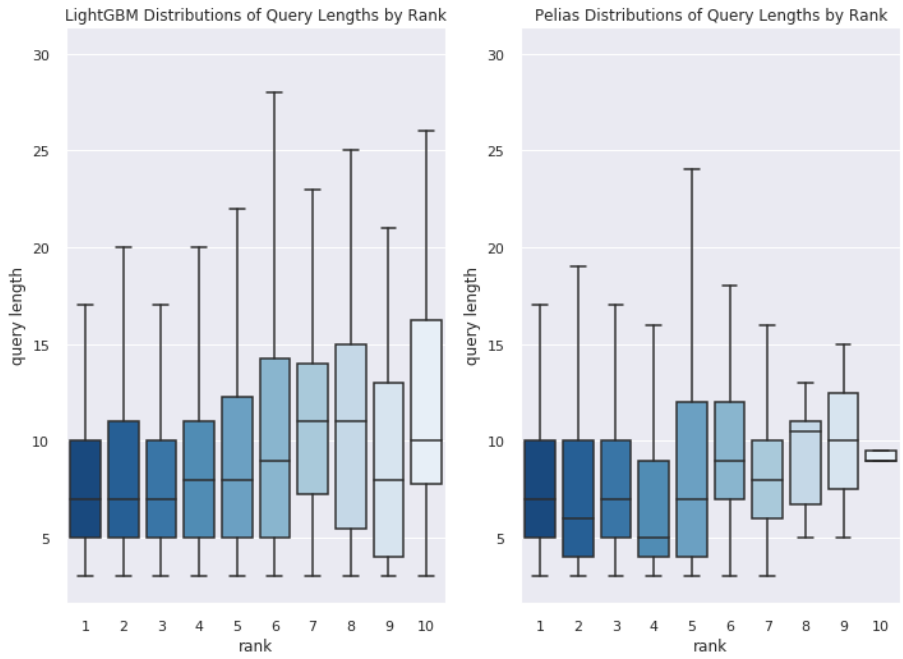

Figure 6.16: Distributions of query lengths (number of characters) by ranks of clicked results from LightGBM and Pelias.

Query Character Length Range	LightGBM MRR	Pelias MRR
[3, 10]	0.9003	0.8481
[11, 20]	0.8721	0.8531
[21, 30]	0.8044	0.8612
[31, 40]	0.5647	0.8167

Table 6.4: MRR scores for LightGBM and Pelias for different query character length ranges

Further investigation is required to understand the reason behind the low MRR scores for the learning to rank model's performance on longer queries. A potential reason is that a large number of shorter queries in the clickthrough data biases the training data. Figure 1.5 and 1.6 also show that users generally type in shorter queries, which lessens the impact of performing poorly on longer queries. However, the learning to rank model's poor performance, in this case, presents a clear area of improvement.

6.5 Summary

In this chapter, the results of evaluating the different learning to rank models were discussed. Various aspects of the models were then discussed, which included the feature importances and hyperparameter tuning of the tree base ranker; the different neural ranker architectures experimented with and their performances; and lastly, comparisons of Pelias against the learning to rank model with the best performance.

Chapter 7

Conclusion and Future Work

The growth of learning to rank has led to breakthroughs in solving ranking problems, which in turn has led to significant improvements in the performance of search systems. This motivated the goal of applying learning to rank to a place search engine based on traditional information retrieval called Pelias. The intended use of the learning to rank model is to serve as a re-ranker of top-k results already retrieved by Pelias. Several types of models; tree-based, neural, and linear were experimented on and evaluated against Pelias using the Mean Reciprocal Rank (MRR) metric. A number of the models showed varying levels of improvement over Pelias. Among them, a tree-based model implemented using the LightGBM framework had the best MRR score. It is shown, based on comparisons, that the learning to rank model improves over Pelias in cases such as when the user's location is not known. The findings from the experiments show strong evidence that the performance of a place search system such as Pelias can be augmented using learning to rank.

Due to the increasing prominence of learning to rank, there is an ever-growing body of work that could serve as the basis for potentially improving the models built in this work. Additionally, from the findings in this thesis, we identify the following promising directions for future work:

- Further research can explore how the interaction between Pelias and the learning to rank model can be refined so that both are optimized for their respective retrieval phases. For example, Pelias could have less granular retrieval rules and return a larger set of top documents.
- Another pertinent area of future work is the application of unbiased learning to rank methods to the clickthrough data used to build the models.
- Lastly, the data used in this work did not incorporate unsuccessful queries, i.e., queries that did not result in a click. Unsuccessful queries could prove to be particularly useful for improving the performance of the learning to rank models.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] H. Abdi. The kendall rank correlation coefficient. *Encyclopedia of Measurement and Statistics. Sage, Thousand Oaks, CA*, pages 508–510, 2007.

[3] A. Agarwal, X. Wang, C. Li, M. Bendersky, and M. Najork. Addressing trust bias for unbiased learning-to-rank. In *The World Wide Web Conference*, pages 4–14. ACM, 2019.

[4] Q. Ai, K. Bi, C. Luo, J. Guo, and W. B. Croft. Unbiased learning to rank with unbiased propensity estimation. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 385–394. ACM, 2018.

[5] Q. Ai, X. Wang, S. Bruch, N. Golbandi, M. Bendersky, and M. Najork. Learning groupwise multivariate scoring functions using deep neural networks. 2019.

[6] R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[7] M. Blondel, A. Onogi, H. Iwata, and N. Ueda. A ranking approach to genomic selection. *PloS one*, 10(6):e0128570, 2015.

[8] C. J. Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11 (23-581):81, 2010.

[9] C. J. Burges, R. Ragno, and Q. V. Le. Learning to rank with nonsmooth cost functions. In *Advances in neural information processing systems*, pages 193–200, 2007.

[10] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*, pages 129–136. ACM, 2007.

[11] O. Chapelle and Y. Chang. Yahoo! learning to rank challenge overview. In *Proceedings of the learning to rank challenge*, pages 1–24, 2011.

[12] O. Chapelle, D. Metlzer, Y. Zhang, and P. Grinspan. Expected reciprocal rank for graded relevance. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 621–630. ACM, 2009.

[13] T. Chen and C. 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.

[14] W. Chu and Z. Ghahramani. Gaussian processes for ordinal regression. *Journal of machine learning research*, 6(Jul):1019–1041, 2005.

[15] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

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

[17] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[18] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[19] M. Ibrahim and M. Carman. Comparing pointwise and listwise objective functions for random-forest-based learning-to-rank. *ACM Transactions on Information Systems (TOIS)*, 34(4):20, 2016.

[20] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[21] T. Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 133–142. ACM, 2002.

[22] T. Joachims, L. A. Granka, B. Pan, H. Hembrooke, and G. Gay. Accurately interpreting clickthrough data as implicit feedback. In *Sigir*, volume 5, pages 154–161, 2005.

[23] T. Joachims, A. Swaminathan, and T. Schnabel. Unbiased learning-to-rank with biased feedback. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 781–789. ACM, 2017.

[24] C. B. Jones and R. S. Purves. Geographical information retrieval. *Encyclopedia of Database Systems*, pages 1227–1231, 2009.

[25] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3146–3154, 2017.

[26] P. Li, Q. Wu, and C. J. Burges. Mcrank: Learning to rank using multiple classification and gradient boosting. In *Advances in neural information processing systems*, pages 897–904, 2008.

[27] T.-Y. Liu et al. Learning to rank for information retrieval. *Foundations and Trends® in Information Retrieval*, 3(3):225–331, 2009.

[28] B. Martins and P. Calado. Learning to rank for geographic information retrieval. In *proceedings of the 6th workshop on geographic information retrieval*, page 21. ACM, 2010.

[29] B. McFee and G. R. Lanckriet. Metric learning to rank. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 775–782, 2010.

[30] B. Mitra, N. Craswell, et al. An introduction to neural information retrieval. *Foundations and Trends® in Information Retrieval*, 13(1):1–126, 2018.

[31] D. Nadeau and S. Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes*, 30(1):3–26, 2007.

[32] A. Natekin and A. Knoll. Gradient boosting machines, a tutorial. *Frontiers in neurorobotics*, 7:21, 12 2013. doi: 10.3389/fnbot.2013.00021.

[33] J. Nocedal, A. Sartenaer, and C. Zhu. On the behavior of the gradient norm in the steepest descent method. *Computational Optimization and Applications*, 22(1):5–35, 2002.

[34] M. O'Brien and M. T. Keane. Modeling result–list searching in the world wide web: The role of relevance topologies and trust bias. In *Proceedings of the 28th annual conference of the cognitive science society*, volume 28, pages 1881–1886. Citeseer, 2006.

[35] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[36] R. K. Pasumarthi, S. Bruch, X. Wang, C. Li, M. Bendersky, M. Najork, J. Pfeifer, N. Golbandi, R. Anil, and S. Wolf. Tf-ranking: Scalable tensorflow library for learning-to-rank. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2970–2978. ACM, 2019.

[37] T. Qin and T.-Y. Liu. Introducing letor 4.0 datasets. *arXiv preprint arXiv:1306.2597*, 2013.

[38] T. Qin, X.-D. Zhang, M.-F. Tsai, D.-S. Wang, T.-Y. Liu, and H. Li. Query-level loss functions for information retrieval. *Information Processing & Management*, 44(2):838–855, 2008.

[39] T. Qin, T.-Y. Liu, J. Xu, and H. Li. Letor: A benchmark collection for research on learning to rank for information retrieval. *Information Retrieval*, 13(4):346–374, 2010.

[40] J. Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 133–142. Piscataway, NJ, 2003.

[41] S. Robertson, H. Zaragoza, et al. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389, 2009.

[42] D. Sculley. Large scale learning to rank. 2009.

[43] B. Shaw, J. Shea, S. Sinha, and A. Hogue. Learning to rank for spatiotemporal search. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 717–726. ACM, 2013.

[44] Y. Shi, M. Larson, and A. Hanjalic. List-wise learning to rank with matrix factorization for collaborative filtering. In *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys '10, pages 269–272, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-906-0. doi: 10.1145/1864708.1864764. URL http://doi.acm.org/10.1145/1864708.1864764.

[45] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[46] M.-F. Tsai, T.-Y. Liu, T. Qin, H.-H. Chen, and W.-Y. Ma. Frank: a ranking method with fidelity loss. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 383–390. ACM, 2007.

[47] V. Vapnik. Principles of risk minimization for learning theory. In *Advances in neural information processing systems*, pages 831–838, 1992.

[48] S. Verberne, H. van Halteren, D. Theijssen, S. Raaijmakers, and L. Boves. Learning to rank for why-question answering. *Information Retrieval*, 14(2):107–132, Apr 2011. ISSN 1573-7659. doi: 10.1007/s10791-010-9136-6. URL https://doi.org/10.1007/s10791-010-9136-6.

[49] X. Wang, M. Bendersky, D. Metzler, and M. Najork. Learning to rank with selection bias in personal search. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, pages 115–124. ACM, 2016.

[50] X. Wang, C. Li, N. Golbandi, M. Bendersky, and M. Najork. The lambdaloss framework for ranking metric optimization. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 1313–1322. ACM, 2018.

[51] Q. Wu, C. J. Burges, K. M. Svore, and J. Gao. Ranking, boosting, and model adaptation. Technical report, Technical report, Microsoft Research, 2008.

[52] F. Xia, T.-Y. Liu, J. Wang, W. Zhang, and H. Li. Listwise approach to learning to rank: theory and algorithm. In *Proceedings of the 25th international conference on Machine learning*, pages 1192–1199. ACM, 2008.

[53] J. Xu, T.-Y. Liu, M. Lu, H. Li, and W.-Y. Ma. Directly optimizing evaluation measures in learning to rank. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 107–114. ACM, 2008.

[54] D. Yin, Y. Hu, J. Tang, T. Daly, M. Zhou, H. Ouyang, J. Chen, C. Kang, H. Deng, C. Nobata, et al. Ranking relevance in yahoo search. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 323–332. ACM, 2016.

[55] Y. Yue, R. Patel, and H. Roehrig. Beyond position bias: Examining result attractiveness as a source of presentation bias in clickthrough data. In *Proceedings of the 19th international conference on World wide web*, pages 1011–1018. ACM, 2010.

[56] H. Zamani, M. Dehghani, W. B. Croft, E. Learned-Miller, and J. Kamps. From neural re-ranking to neural ranking: Learning a sparse representation for inverted indexing. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 497–506. ACM, 2018.

[57] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.