Applying Machine Learning to a Job-Candidate Matching Problem

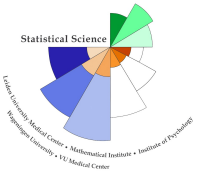
Natalia Bukarina (s2045842)

Thesis advisors: Dr. Suzan Verberne & Dr. Tim van Erven

External supervisor: Ferdi van de Kamp

MASTER THESIS

Specialization: Data Science



STATISTICAL SCIENCE
FOR THE LIFE AND BEHAVIOURAL SCIENCES

Abstract

The task of finding suitable candidates for a job has never been an easy one, and now that recruiters have access to various online job boards and are not necessarily constrained by national borders, it can be argued that shortlisting relevant candidates is more difficult than ever. This is especially true for online recruitment agencies that have huge databases of potential candidates and no effective ways to quickly identify which of those candidates have the required experience and skills for the vacancy at hand.

There are many ways that different companies go about solving the aforementioned problem. In case of YoungCapital, a Dutch recruitment agency, all candidates can state their preferred profession and location when creating a profile on the company's website, and recruiters can then create a search query based on those stated preferences. It is also possible to get keyword matches with candidates' resumes, which, however, is a manual task where recruiters have to decide on the specific keywords they want to find.

Given recent advances in machine learning and natural language processing, it was decided that a learning-to-rank (LTR) approach should be tried to see whether the candidate search process could be improved by presenting recruiters with a ranked list of candidates for each job, with the most suitable candidates at the top of the list. The LambdaMART model was chosen for this task as the state-of-the-art algorithm, and the baseline ranking model was a simple Linear Regression. Most of the features were designed using custom word embeddings. The results were evaluated with common rank-based measures: Normalised Discounted Cumulative Gain (NDCG) and Mean Average Precision (MAP). Precision, which ignores the order of results, was reported as well.

Overall, we found a significant improvement over the current method according to all three measurements. We also demonstrated the impact of different feature sets on the performance of ranking models.

Acknowledgements

First of all, I would like to thank the Data Science team at YoungCapital: Ferdi, Gergely, Jaap, Pieter and Zoltan, who helped me gather all the information necessary for my experiments, and provided continuous moral support. Furthermore, I am thankful to YoungCapital management as a whole for providing me with such an opportunity and giving me the freedom to concentrate on my research.

I am also grateful for all the constructive feedback that Suzan Verberne, my supervisor at Leiden University, provided me with. Her guidance and advice were invaluable in my discovery of Text Mining and Information Retrieval fields, especially when I needed to choose the appropriate metrics to evaluate results. Furthermore, I would like to thank another supervisor from Leiden University, Tim van Erven, for his timely support and positive comments regarding my project.

Finally, a big thank you to all my friends and family for cheering me up when I needed it most, and for their general support throughout my time at Leiden University.

Contents

Chapter 1

Introduction

Finding the "right" candidate for a job opening has never been an easy task. Not only should a prospective employee have the right qualifications and work experience, but they often need to fit into an existing team and share a company's vision as well. The era of online job boards and globalisation has brought about an additional challenge. Since it became relatively easy to create an online profile and apply to a vacancy with just a few clicks, recruitment personnel in this day and age often need to review hundreds of online profiles and resumes just to decide who to approach.

Automating the process of shortlisting candidates can lead to reduced costs and increased recruiter productivity [1], thus it is not surprising that numerous technological solutions have been proposed to assist recruiters along the way. In order to compare competencies mentioned in resumes and those outlined in job requirements, most of the early approaches to job-candidate matching problem either employed an ontology mapping system [2, 3, 4], or computed a similarity score between respective candidate and vacancy profiles [5, 6]. Neither of the aforementioned methods, however, can learn from data, and in case of the former it is necessary to manually construct elaborate ontologies that primarily rely on expert knowledge and various heuristics.

In this work we use a supervised learning approach called learning-to-rank (LTR), as it can automatically find patterns in historical data and combine together complex features, as well as make predictions for previously unseen instances. LTR is a branch of machine learning that focuses on ranking relevant items at the top of the list, and previous literature indicates that it could be a promising way to find suitable candidates [7]. To our knowledge, existing work in this field concentrated on analysing the effectiveness of various LTR algorithms [8], determining whether recruiters' queries can adequately capture vacancy requirements [9], and designing language models to improve ranking and retrieval of relevant candidates [7]. However, information about candidates was solely drawn from resumes, and only count-based methods for transforming resumes into features were used. A notable exception is [10], where structured LinkedIn information and free-text blog posts

were used instead.

We thus saw an opportunity to contribute to the existing body of knowledge by conducting several of our own LTR experiments within the recruitment domain, with a focus on constructing features from both profile and resume data, as well as on examining the effect of different feature sets on ranking. In order to do so we collaborated with YoungCapital, a Dutch recruitment agency that collects and stores thousands of candidate profiles and resumes. These serve as a pool of information on available workers that recruiters often need to tap into. The company's current candidate search system is based on the ElasticSearch [1] search engine, which has been configured in such a way that recruiters can indicate what information should candidate profiles contain, such as HBO level of education or Amsterdam as preferred work location, and what keywords should be found in resumes. After specifying all the criteria, recruiters receive a list with candidate profiles that matched their query, where the results are sorted based on the match score calculated by the search engine.

The goal of this research is to leverage YoungCapital's large collection of historical job-candidate evaluations and build a model that would be able to rank suitable candidates higher on the list, and thus improve upon the current method. While constructing the model, we aim to address several points of interest that have not yet been explored in the literature. Specifically, we concentrate on the following research questions. How can feature sets be constructed from structured and unstructured information about candidates, and what is the influence of such feature sets on a job-candidate ranking model? What effect do different approaches to turning resumes into numeric features have on the model? How well does the final ranking model perform compared to the current system? The main contributions of this thesis can therefore be summarised as follows:

- We demonstrate four different approaches to constructing numerical features from a raw text that can be used in a learning-to-rank model, and evaluate which method works best. The methods that we examine are Bag-of-Words, TF-IDF, and document embeddings created by taking either weighted or unweighted mean of individual word embeddings (see more details on these techniques in Section 4.2.3).
- We describe, similar to [7], different sets of features and their respective impact on the performance of our job-candidate ranking model. We, however, go beyond just resume features as we also include candidate preferences and location.
- We show that by using our model the company can expect a 28% relative improvement in ranking of the top-10 candidates in a list compared to the method currently in place.
- As there was no resume parser available for our experiments, we needed to construct our own so that we could exclude personal information and retrieve vocabulary from

[1] More information about ElasticSearch can be found on `https://www.elastic.co/guide/en/elasticsearch/reference/current/getting-started.html`

different resume sections. Therefore we present a technique to train a resume parsing model using shallow features and semi-structured training set.

- We also create a manually annotated dataset of 150 resumes that could be used for further research within the company.

The subsequent chapters will provide more details on how the research was conducted. Chapter 2 will provide the background on learning-to-rank and other job-candidate matching approaches, including a comprehensive description of the matching method currently used by YoungCapital. In Chapter 3 we will demonstrate how we trained and tested our resume parsing model, and explain how results of this model were used in subsequent LTR experiments. Chapter 4 is the main chapter of this project. It will explain in detail how various features were generated, what experiments were conducted and how our final model compares to ranking created at random, as well as to ranking provided by ElasticSearch. Note that since we have essentially trained two different models, Resume Parser and LTR, both of the respective chapters contain method descriptions and showcase results. Chapter 5 will discuss implications and limitations of this research, and Chapter 6 will provide concluding remarks.

Chapter 2

Background and Related Work

In this chapter we will provide some background on existing technologies and research aimed at matching candidates and jobs. Specifically, Section 2.1 will describe in detail the way YoungCapital recruiters search for candidates using the current system. Section 2.2, on the other hand, will outline how other companies and researchers sought to match candidates and jobs, focusing on experiments that have been done using LTR algorithms, and what features were used. Since our main framework for this challenging problem is learning-to-rank (LTR), the key points behind this approach will be summarised as well.

2.1 Current Approach to Candidate Search

Currently YoungCapital's recruiters are mainly using ElasticSearch search engine to look for candidates. For each job opening recruiters specify what information should be found in candidates' profiles or resumes, and the engine returns an ordered list of candidates, with those that match query specifications the best sorted at the top of the list.

In general, most recruiters go through the following steps when they embark on a head-hunting mission using the current system:

1. Fill in the desired candidate attributes, such as education and location, using a dedicated form that transforms these details into an ElasticSearch query. If a search is conducted for a specific job opening, most fields are already prefilled using the job's profile data.

2. If search results do not seem relevant, narrow down the search by specifying resume keywords or expand the search by including more locations.

3. Manually check the profiles of the first few candidates that appear in the search results. If the profile/resume seem suitable for the vacancy, try calling or emailing this candidate and asking whether he/she would be interested in applying for the position.

4. Repeat steps 2 and 3 until enough candidates are found to fill the vacancy.

The following sections will describe what candidate information is available for recruiters to search in and how ElasticSearch is set up to find matching candidates. This information will serve as a basis for our LTR features, which is why we provide a rather comprehensive overview.

2.1.1 Candidate Representation

YoungCapital collects and stores data about every candidate that has ever created a profile through any of their websites, so that recruiters could look for candidates using the company's database. As having a resume is a prerequisite for most jobs, candidates can either upload their resume as a document, in which case all the text from a resume is stored as a single string, or use the company's app called CVBuilder to create one. If candidates go with the latter option, then resume data is stored in a semi-structured format, namely in a json string together with all of the field names where information was entered (name, address, start date of education, etc.). It is, however, not compulsory to upload or create a CV.

The company also collects different types of structured information about candidates. When creating a profile, apart from providing some general background information like name and address, every candidate is also required to indicate a preferred job type, such as full-time or part-time, a job location and a job function. Furthermore, another required section is education, where at least one type of education (secondary school, MBO, HBO, university) should be selected. In addition to these compulsory fields candidates can specify what languages they know and whether they have a driver's licence or not. All of the choices to fill in the above information are predetermined, in other words, candidates can either select an option from a drop-down menu, or tick a box.

It is important to note here that candidates have to state their preferences and declare the attained education level, but they are not required to provide any information regarding their past work experience or skills. Although this might seem unusual at a first glance, it becomes less surprising when taking into consideration the fact that for a long time YoungCapital's primary target group were inexperienced students looking for a part-time job. At the point of writing a resume was the primary resource where recruiters could find data on candidate's experience level.

2.1.2 Vacancy Representation

In the same manner as candidates have to create a profile upon registering, recruiters have to create a vacancy profile when posting a new job on any of the company's websites. The information that recruiters have to fill in is directly comparable to the structured information in candidates' profiles. In particular, each job should have one or more functions, a list of acceptable educations, indication whether it is a full-time or a part-time position,

Table 2.1: Example of a candidate and a job profile information in respective fields.

Field	Candidate Profile	Job Opening Profile
job type	full-time, part-time	full-time
job function	administrator, receptionist, waitress	administrator, receptionist
region	amsterdam, hoofddorp, leiden	utrecht
education	university	university

and a zipcode for its primary location. If more than one location is advertised in the same vacancy, it is possible to select different regions in addition to the primary one. Table 2.1 illustrates what kind of information can be found within candidate and job profiles.

Again, all of this information is generated when recruiters tick respective boxes. Apart from that each vacancy also has a supplementary job description, which is stored in a semi-structured format. In other words, when writing a description recruiters have to adhere to predefined spaces (or boxes) for each separate piece of information such as title and requirements.

2.1.3 ElasticSearch

ElasticSearch is a search engine that is capable of a full text search. In order to retrieve candidates using this tool, recruiters have to create a so-called search query. Using the provided interface users need to specify education, location, job type, function etc., which ElasticSearch internally converts to a query and searches for matches within candidate profiles.

In particular, the search engine interprets users' input as Boolean queries[1]. All candidate profile information is matched with the "must" clause, which implies that a certain candidate will be included in the search results if and only if there has been at least one match with each respective field in the search query. If any keywords are specified, then only candidates that mentioned those keywords in their resume will be retrieved. One of the main drawbacks of matching keywords in this way is the inability of the search engine to interpret inflections like programmer-programming, thus often failing to retrieve relevant candidates [11]. Although it is also possible to use more advanced keyword matching techniques such as including n-grams and synonyms within ElasticSearch, at this stage YoungCapital is only using the default configurations for text search, and keywords can be matched anywhere in a resume.

If recruiters don't change the default query composed from vacancy profiles, then ElasticSearch will, in essence, look for candidate profiles that match the respective job profile, without including any keywords. Taking the example illustrated in Table 2.1, the query would be composed from all the information given in the job opening profile column,

[1]Information about ElasticSearch Boolean queries can be found on `https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-bool-query.html`

and since there is a mismatch between vacancy and candidate regions, the candidate will not be retrieved. ElasticSearch query that is generated could be referred to as a "standard" query, and is used for most candidate searches as a starting point.

Every match adds to the overall score computed for each candidate. Boolean query takes more-matches-is-better approach, such that the closer candidate's profile is to the search query and the more times specified keywords appear in the resume, the higher the final match score and hence the higher the rank of this candidate in the resulting list. Non-keyword fields contribute equally to the final match score, as every found match carries the same weight.

Text search, however, works slightly differently, as it is easier to match some words than others. Moreover, importance of a single word to the overall meaning of a document can vary depending on how often this word is used in the whole corpus. In order to partially capture this property of natural language, under the hood of the ElasticSearch scoring function is a TF-IDF similarity algorithm[2], and the algorithm in use was implemented by Apache Lucene [3]. As retrieved candidates are sorted according to their overall match score, we could use ElasticSearch ranking as a baseline ranking that we could compare our LTR models to.

In short, YoungCapital's current system has the basic functionality to search for candidates and is in principle similar to other recruitment tools. The system, however, is highly dependent on recruiters' input and using it can be extremely time-consuming. Choosing the right keywords to find candidates with relevant experience and skills is not straightforward and highly subjective, which is why recruiters often have to try several different queries to get the output that seems appropriate. We would like to simplify this process by applying machine learning to a job-candidate matching problem. The goal is to have recruiters spend less time composing queries and scanning through unrelated candidate profiles, and more time establishing personal contact with potential recruits.

2.2 Related Work on Job-Candidate Matching

Being able to automatically select suitable candidates for different vacancies has undoubtedly been the holy grail of recruitment industry for quite some time now. Until recently, however, most companies (including YoungCapital) took a keyword match approach to candidate search, with very little help from the machine learning field. In the sections below we review some of the ranking methods that were suggested in the literature.

[2]Aparth from TF-IDF there is also Okapi BM25 similarity algorithm available, which is the default in later versions `https://www.elastic.co/guide/en/elasticsearch/reference/2.4/similarity.html`

[3]A lot of ElasticSearch functionality, including both TF-IDF and BM25 similarity algorithms, is based on Apache Lucene which is an open-source search engine software library `http://lucene.apache.org/core/4_0_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html`

2.2.1 Query-Resume Similarity Approach

Several different similarity based methods aimed at comparing vacancy and candidate information, analogous to the current search system described in Section 2.1.3, have been proposed by the research community. Singh et al. [6] presented an e-recruitment tool called PROSPECT that was designed to shortlist qualified candidates for jobs by mining candidate resumes. This tool can parse relevant information such as skills, highest degree, work experience in years etc. from resumes, and then calculate similarity between these details and a query automatically constructed from job requirements. Candidates would then be ranked according to several different similarity metrics, where a large emphasis was put on matching candidate skills to those mentioned in job descriptions. The ranking algorithm that showed the best performance in their experiments was TF-IDF similarity model from Lucene.

The score that indicates a match between a candidate's resume and a job description could also be assigned according to a hand-crafted set of rules, as explained in the paper by Zimmermann et al. [12]. The authors suggested assigning a score between 0 and 100 to each candidate-job pair, where this score is the weighted average of education, work experience and skills sections. In their prototype application the information for each section was extracted from resumes, and in case of education parsed data was combined with external sources. Namely, if candidate obtained a university degree, the ranking of the respective educational institution, as determined by Times Higher Education [4] and QS[5], was included in the overall education score in an attempt to objectively improve judgement about the quality of different educations. The work experience section score depended on the duration of employment and the employer score, while the skills score was calculated using word embeddings and the distance between each of the required skills and those parsed from candidate resumes.

When designing an approach that includes calculating an overall score from different fields where matches occurred, a decision has to be made on how to aggregate all the sub-scores together. It is not always clear which of the fields are more important for determining a good match between candidates and jobs, however. Should, for instance, having an appropriate education contribute more to the final match score than possessing relevant skills, or less? Both [6] and [12] decided that skills match score should account for more than any other field matches, although it has been mentioned that user research is necessary to evaluate different weighing configurations [12]. This strategy is certainly only feasible when there is a small number of weights to optimise.

Instead of configuring the above manually, Rodenburg [13] went a step further with a proposed CVMatcher system. Similar to [6], a query is constructed from parsed job openings, and information extracted from resumes is stored in a structured manner such that matches in respective query-document fields could be checked (e.g. if roles mentioned

[4] urlhttps://www.timeshighereducation.com/world-university-rankings

[5] `https://www.topuniversities.com/university-rankings`

in resume match job title, or programming languages known by candidates match keywords from job description). However, the author demonstrated how an Evolutionary algorithm could be employed together with ElasticSearch to assign relative importance parameters to these field matches. Notably, the performance of such an approach was tested on providing a list of suitable jobs for a candidate rather than ranking candidates for jobs.

In all of the above approaches the underlying assumption is that the more similar a candidate's information is to job requirements or a user specified query, the higher the chances of this candidate to be regarded as suitable for the vacancy by recruiters. Although such a premise does make intuitive sense, it is possible to imagine scenarios where similarity approach might fail. For instance, if chosen keywords for a query are "junior data analyst", then an individual who is a senior data analyst is very likely to have these words in her or his resume as past work experience, but is not likely to be looking for a junior position now. There is thus a potential for improvement that has been recognised and addressed, among others, with learning-to-rank methods.

2.2.2 Learning-to-Rank Approach

Before we describe LTR experiments that have been done on ranking candidates for jobs, we will first provide an overview of the field. Learning-to-rank is a collection of machine learning techniques that, given a certain query or condition, aim to find an optimal ranking of items in a list such that the most relevant results appear at the top. Not surprisingly, the most common application of LTR models is optimizing search engine results using user click behaviors [14].

Given the widespread usage of LTR techniques in Information Retrieval (IR) domain, it is commonplace to use the terms query and document when referring to condition and list item, respectively. As we have seen in previous sections, when this idea is translated to recruitment domain, the query could, for instance, be comprised of keywords from a certain job opening description, and the documents could be candidate profiles and/or resumes. Thus an LTR problem becomes learning a ranking function to rank candidates (documents) for a certain job posting (query), using recruiters' evaluations such as hired or not hired. For the rest of this section we will continue using terms query and document as we will mainly be discussing IR concepts.

Since the Yahoo! learning-to-rank challenge held in 2010, numerous LTR algorithms have been developed and studied extensively [15]. Already by 2015 at least 87 different methods have been identified [16], and this number is undoubtedly higher by now with latest method described in early 2019 [17]. Nevertheless, most of those approaches can be split based on their input representation and loss function into three distinct groups: pointwise, pairwise and listwise [14].

Models that are associated with the **pointwise** approach discard the list format of the initial problem, and view each feature vector as an independent data point. The loss function of these models evaluates how accurately the ground truth label was predicted for

each individual query-document pair. Most traditional supervised statistical and machine learning models such as regression, ordinal regression and classification fit this description and could thus be used for ranking problems. Examples of the above could easily be found in literature [18, 19], although some pointwise methods were developed specifically with a ranking problem in mind [20].

The **pairwise** approach models, on the other hand, compare pairs of documents that belong to the same query and try to predict which of the two documents in a pair should be ranked higher relative to the other. A significant number of pairwise ranking algorithms model ranking as a pairwise classification, with corresponding loss function defined on a pair of documents [14].

Out of the three approaches being discussed in this section, the **listwise** approach is the only one that does try to minimize a loss functions defined on the whole list of documents that are associated with a certain query [21]. In order to train a model using one of the listwise approaches, however, the labels should indicate the ground truth ordering of the items in a list.

Overall, regardless of the approach, every LTR model tries to solve a ranking problem for a list of documents by predicting a relevance score for each query-document pair. Documents are then sorted for each query based on that score, from highest to lowest. One implication of the above is that, although individual predictions can be very different from model to model, two models can be equally successful as long as they produce an optimal ordering of items in a list.

An example of using all three approaches for a job-candidate matching problem can be found in a graduation project by Braun [8]. The author used the CVMatcher system [13] mentioned above as a baseline method, and three different LTR algorithms (Gradient Boosted Regression Trees, LambdaMART and SmoothRank) were implemented for ranking experiments, this time trying to order a list of candidates given a vacancy. Features were based on terms that occurred in candidate resumes and job descriptions, with respective TF-IDF scores. Furthermore, Manifold Regularization (constraining the model in such a way as to make similar data points have a similar probability of ranking at the same place) was used together with SmoothRank as well.

In another graduation project Fang [7] described a candidate search tool used by Textkernel [6] at the time, the broad architecture of which was similar to PROSPECT [6]. In particular, it also included resume parsing module to extract candidate information in a structured manner, as well as vacancy parsing software to automatically create a query. Furthermore, their software could also automatically assign extracted job titles to job classes and job codes, which facilitated the matching process. The search engine to retrieve and rank candidates was again ElasticSearch. Where the difference lies is the LTR based re-ranking algorithm that the author suggested, and specifically the features used for

[6] Textkernel is a company that develops software products for the HR and recruitment market `https://www.textkernel.com/`

training the model. Apart from the common word-based (or stem-based) features, Fang also introduced a divergence feature that examined whether the language model of a query is similar to the language model of a resume.

Textkernel also published a paper of their own describing LTR experiments that were slightly different from Fang's [9] as they analyzed matching/ranking problem with manual queries, rather than automatically created ones. One of their main conclusions was that users seem to have little intuition on how to design a "rich" query that can accurately represent the vacancy needs. As a result, there were a number of sparse queries that made a model susceptible to bias. Nevertheless, they demonstrated that their re-ranker improved upon the original results by returning, on average, one additional candidate in the top 10 of retrieved list.

Faliagka et al. [10] took a different approach to candidate ranking. They proposed a system that could infer personality of applicants by analyzing blog posts of respective individuals. The focus of this personality mining exercise was the extraversion trait, as it is assumed to be one of the crucial personality characteristics for client-facing jobs. The outcome was an extraversion score for each candidate, which was used as a feature in their ranking models. Other candidate features came from LinkedIn profiles, which were either Boolean (presence/absence of a certain skill), or numeric (number of work years). Pearson's correlation with actual scores assigned by professional recruiters was used to assess performance of the aforementioned models. Overall, to estimate applicants' relevance scores the proposed system relied on objective criteria extracted from the applicants' LinkedIn profiles and subjective criteria extracted from their social presence, which gave promising results.

Overall, systems described in literature appear to have improved on the baseline ElasticSearch output through including an LTR model [7, 8], hence we adopt this framework as well. In this thesis, however, we combine resume and profile information about candidates to create features for our model, while most of previous literature mainly focused on resume features alone. Furthermore, it has been demonstrated that resumes can be transformed into numeric features using count-based approaches such as Bag-of-Words [7] and a TF-IDF [8], but, to our knowledge, there is no published work that compares different methods and demonstrates the effect that those transformations have on a ranking model. Moreover, there is limited literature available on using word embeddings as features in recruitment domain [12, 22], which is why for our experiments we include comparison of this technique to count-based methods as well. We also use word embeddings to create some of the match features to capture how well a candidate fits job requirements, while previous work mainly included ElasticSearch match scores for this purpose.

Chapter 3

Resume Parsing

Most of the resumes in YoungCapital's database are stored as a single string of text. Previous research, however, indicated that extracting various candidate attributes such as education, skills and previous work experience could help in creating meaningful features for ranking models. Furthermore, not all of the information provided in resumes is relevant for deciding how candidates should be ordered in a list, and it is considered unethical to include information like gender and nationality in machine learning models.

It was thus decided that we should first create a resume parsing model that could exclude personal information from CVs, as well as help us construct features for learning-to-rank models. In particular, the main goal for the model described in this chapter is to learn how to identify four main sections in a resume: personal information (PI), education (EDU), work experience (WORK) and skills (SKILLS). Section 3.1 will specify what data was gathered for training of our resume parser, Section 3.2 will describe how the model was created and tested, and Section 3.3 will present the results.

3.1 Data

There were several different types of data that we collected specifically for resume parsing task. As mentioned in Section 2.1.1, apart from unstructured resumes YoungCapital also stores semi-structured information gathered through the CVBuilder application. We managed to obtain 32,180 CVBuilder resumes in total, mainly in Dutch. All the information contained in those resumes was distributed among the following ten sections: personal information, profile, work experience, education, skills, interests, languages, trainings, references and extra. These sections, in turn, were often split into subsections like summary, start year, end year, name of educational institution, level of a certain skill etc. This dataset could thus be a starting point for us to create a suitable training set.

We also manually annotated a sample of 150 unstructured resumes, which would serve as a test set for the parsing model. Furthermore, we collected a total of 224,464 unstructured

resumes in order to train a word embeddings model (see Section 3.2.2).

3.2 Methods

Ideally, the resume parsing problem could be viewed as a supervised learning task. To solve this task we would need a large number of resumes with word-by-word or line-by-line labels. Unfortunately, there was no such dataset available, and hence we needed to improvise. The following sections outline how the training set was constructed from CVBuilder resumes, what features were created and what algorithm was used to train a resume parser. Note that since the overall goal is to be able to split a resume into different sections, the general idea for the model is to correctly label different lines, rather than tokens as many other CV-parsing models do [6, 13, 23].

3.2.1 Constructing a Training Set

By leveraging the structure of json strings that CVBuilder resumes were stored in, we could get labels for resume sections that we were interested in (PI, EDU, WORK, SKILLS), but there were still a few crucial pieces missing. Firstly, since all the information was entered by users into predefined fields on the front-end side of the application, there was no need to include section names such as "Opleiding" or "Werkervaring" (education and work experience, respectively). One could argue that section names are the primary clue for a human reader as to which section different resume lines belong to, and, unfortunately, this critical information was missing from the dataset. Furthermore, as all data was arranged into different sections and subsections, there were no resume lines as such. Information about spaces between different lines or sections was absent for the same reason.

Considering that we were nevertheless hoping to use CVBuilder data as a training set for our resume parsing model, we had to design a set of rules to overcome these shortcomings and essentially mimic the format of unstructured resumes. All of the above data was therefore reorganised into resume lines, and suitable section names, keywords like "telefoon", "naam" and "rijbewijs", as well as spaces between different sections were added. Each line had a corresponding label (PI, EDU, WORK, SKILLS or O) attached to it, where O stands for other.

It should be noted here that such an approach has a considerable drawback, namely reduced variance between our artificially generated training documents. In reality candidates have the freedom to choose not only different section names than the ones we introduced to our dataset, but also combine sections together and include the same type of information in different parts of a resume. An example for the latter could be mentioning skills in profile and work experience sections. CVBuilder resumes, on the other hand, were all converted to a specific format using the same rules, and it was not possible to account for all of the scenarios mentioned above. This is the main reason why we wanted to test the performance

of our model on manually annotated resumes, rather than splitting the CVBuilder resumes into train and test sets.

3.2.2 Word Embeddings

In order to reduce dimensionality of textual data, as well as to account for syntactic and semantic similarity between words, most contemporary research projects make use of word embeddings. We decided to base some of the model features on word embeddings as well, therefore we provide a brief summary on this technique.

Word embeddings, in fact, is an umbrella term for a class of language modelling techniques that map words to a real-valued, low-dimensional vector space. If words are similar to each other, namely if they occur in similar contexts, then vectors representing these words should be in close proximity in a respective vector space.

In this thesis we employ the neural network approach to generating word embeddings, colloquially known as `word2vec` and introduced by Mikolov et al. [24]. Specifically, we trained word vectors using the continuous-bag-of-words (CBOW) method. In the CBOW architecture, the model predicts the current word w_t from a window of surrounding context words $w_{t-c}, ..., w_{t-1}, w_{t+1}, ..., w_{t+c}$, where c is the number of context words. The training objective is thus to learn word representations that are good predictors for the middle word w_t. The maximum likelihood estimation of the CBOW model is equivalent to minimizing the loss function:

$$-\sum_{t=1}^{T} \log P(w^{(t)}|w^{(t-m)}, ..., w^{(t-1)}, w^{(t+1)}, ..., w^{(t+1)})$$

This prediction based approach was shown to be a significant improvement over the count-based approaches previously common in Natural Language Processing (NLP) tasks [25], and since it is an unsupervised learning method, we could use all of our 224,464 unstructured resumes (around 46 million tokens in total) to train a word embeddings model. The text processing steps that we took beforehand were mainly aimed at removing numbers and occasional images, as well as substituting special characters with ASCII alternatives. We chose $c = 5$ context words and 100 dimensions for each vector as parameters for the embeddings model.

3.2.3 Features

For this model we needed to construct features that would span the whole resume line. After experimenting with different ideas, we arrived at the following three types of features:

- Spacing features
- Keyword match features

- Cosine similarity features

The first two types are binary features. Spacing features check whether a line in question has any empty lines above or below it, as having empty space above often signals the start of a new section, below - the end of the current section. Furthermore, we also included a variable that signals to the model that a line is the first one or the last one in the resume. Keyword match features indicate presence or absence of certain words, such as "opleiding" and "werkervaring". These keyword matches might seem trivial in our artificially created training set as by design each resume contained these words, but we wanted the model to learn the importance of specified terms as they are commonly used as section headers in free text resumes.

Although both of the aforementioned types of features could help the model determine the boundaries of different sections and occasionally hint what section a line belongs to, none of them contained enough information to reliably determine the label for each line of a resume. We thus thought that it would be useful to also have features that quantify the level of similarity between a line and different resume sections. In order to do so we firstly gathered 500 most frequent words using all CVBuilder resumes from each of the following CVBuilder sections: "name", "address", "function", "work summary", "education", "skills", "interests" and "languages". Several keywords that commonly occur in Dutch resumes, for example "geboorteplaats" and "mobiel", were also included as a separate "PI tokens" group.

Once all the words for each category were gathered, we needed a distance measure to assess the similarity between those word groups and resume lines. Cosine similarity between word embeddings is a popular choice when judging text similarities, and since, as pointed out in Section 3.2.2, YoungCapital had a large collection of raw resumes that were suitable for unsupervised training of word embeddings, cosine similarity seemed like a reasonable option. Formally, cosine similarity for vectors $\mathbf{x}$, $\mathbf{y} \in \mathbb{R}^d$ is the cosine of the angle between them, and is represented using a dot product and magnitude as:

$$\frac{\mathbf{x}^{\mathrm{T}}\mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} \in [-1, 1]$$

Cosine similarity of 1 means that the vectors are identical, 0 - that they form a right angle, and -1 indicates that vectors are diametrically opposed.

Using trained word vectors from our custom embeddings model, we calculated a mean embedding vector for each of the above-mentioned word categories. As a final step we created a mean vector for each resume line (excluding numbers), and calculated cosine similarities to each of the 9 mean word group vectors. Consequently, each line had 9 cosine similarity features in addition to binary features. The detailed overview of all features can be found in Appendix, Table A.1.

3.2.4 Conditional Random Fields

Although, as it has been stressed in sections above, non-CVBuilder resumes fall into the category of unstructured data, most people still follow a set of conventions when writing their CV. For instance, it is common to start a resume with personal information such as name and email address, whereas it is rare to see a reference section before work experience information. Furthermore, lines that belong to the same section follow each other and hardly ever mix with lines from other sections.

Given this sequential nature of our data, we needed to choose a model that could account for it. Traditional classification models assume independence between data points and thus might not be the best option. Conditional Random Fields (CRF) models, on the other hand, are commonly used for different sequence modelling tasks that involve natural language processing, such as Part-of-Speech Tagging and Named Entity Recognition. Several resume parsing systems also included a CRF model to identify different segments [6] and individual tokens [23].

In short, CRF are undirected graphical models first described by Lafferty et al. [26], and are considered to be a state-of-the-art sequence labeling method. CRF models are trained to maximize a conditional probability distribution given a set of features. Let $Y = (y_1, ..., y_T)$ denote a sequence of labels and $X = (x_1, ..., x_T)$ denote the corresponding observations sequence. The sequence of labels is the concept we wish to predict, such as named entity, whereas the observations are the words or sentences in the input string. In this thesis we train a linear chain CRF, for which the conditional probability $P(Y|X)$ is computed as follows:

$$P(Y|X) = \frac{1}{Z_X} \prod_{t=1}^{T} \exp\left\{ \sum_{k=1}^{K} \lambda_k f_k(y_t, y_{t-1}, x_t) \right\}$$

where Z_X is a normalizing constant, f_k is a feature function, and λ_k is a feature weight. CRF methods offer an advantage over generative approaches by relaxing the conditional independence assumption, and can be understood as an extension of the logistic regression classifier [27].

Since predictions of a CRF model are labels, its performance was evaluated with common information retrieval metrics: precision, recall and F1-score. Precision is the proportion of selected items which are relevant, recall refers to the proportion of relevant results that were retrieved, and F1-score is the average between the two.

3.3 Results

As most of our resume parser features are reliant on word embeddings, we first needed to examine the quality of trained word vectors. Using t-Distributed Stochastic Neighbor Embedding, or t-SNE for short, we projected some vectors onto a 2-D plane. Figure 3.1a demonstrates that word vectors from selected categories (name, function, education etc.)

Table 3.1: Resume parsing model results on the (manually annotated) test set.

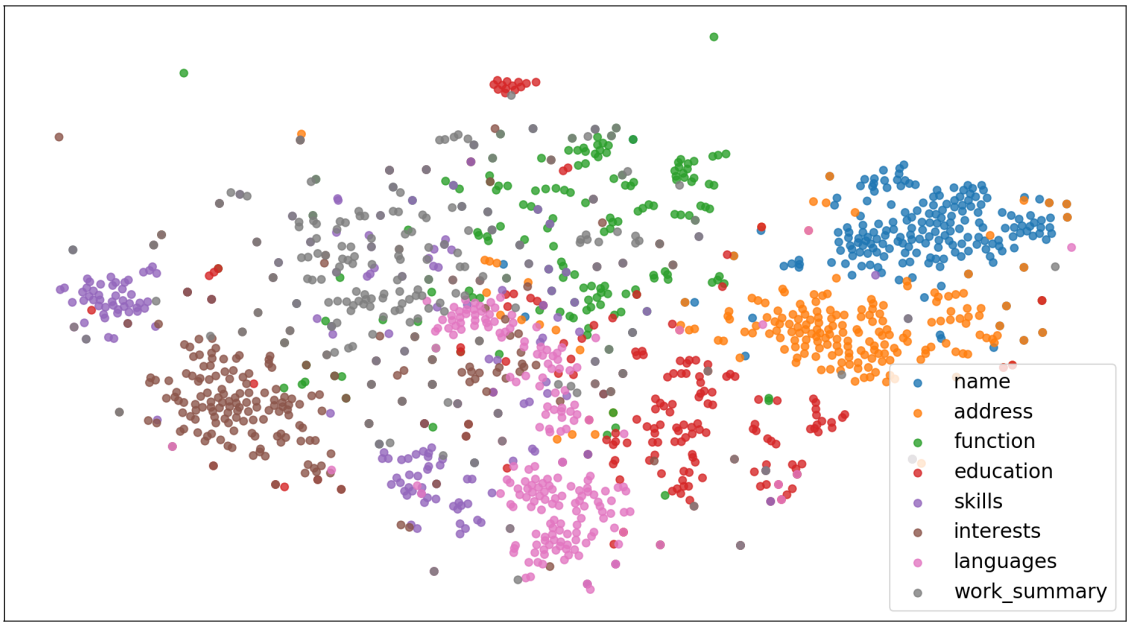
Section Label	Precision	Recall	F1	Nr. of Samples
PI	0.991	0.896	0.941	1496
WORK	0.924	0.915	0.919	3486
EDU	0.873	0.824	0.848	1525
SKILLS	0.455	0.778	0.574	418
O	0.733	0.717	0.725	1462
Avg / Total	0.870	0.854	0.859	8387

are mostly clustered together, as expected. It is clear that names, addresses and interests are largely separable from other words (blue, orange and brown clusters, respectively). Skills cluster (purple), on the other hand, is split into two small groups, and there are a lot of individual skills mixed with other words.

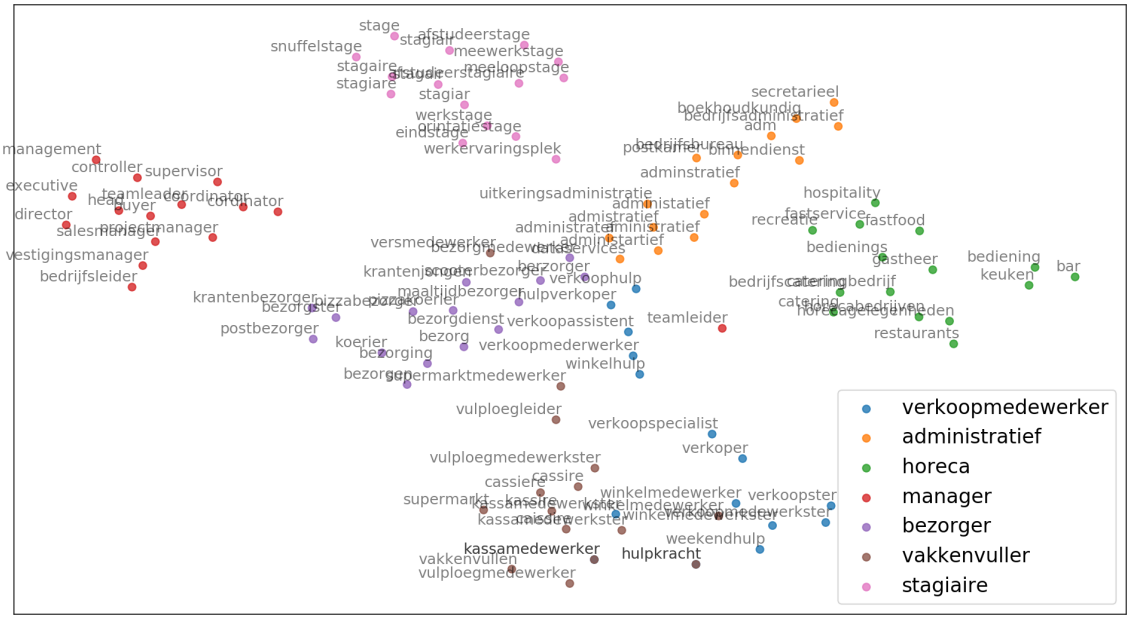
Figure 3.1b takes a closer look at what words are considered similar by the embeddings model. We selected 7 different functions and plotted top-15 most similar words for each one of them. Evidently, most of the words that a human would group together are recognised as related by the model as well.

Figure 3.2, on the other hand, showcases what cosine similarity scores one can expect between different resume lines (y-axis) and word categories (x-axis), when both are represented as averaged word embeddings. Using these example resume lines we can see the same broad pattern that we noticed among the actual resumes in both the train and the test set, namely that lines containing personal information tend to get positive scores when compared to names, addresses and PI tokens (dark blue rectangle in the top left of the plot), but generally negative scores when compared to all other word groups. In other words, it appears that when constructing features using cosine similarities between lines and different word categories, these features alone can distinguish relatively well between lines that contain personal information and those that do not.

The results of our CRF model on the manually annotated test set are presented in Table 3.1. The best performance as measured by precision and F1-score was shown for personal information lines (99% and 94% respectively), which is in line with our expectations based on Figure 3.2. Recall, however, was highest for work lines (92%). The section that the model struggled to identify was the skills section, which was likely caused by the fact that some individuals tend to combine languages and skills sections together, whereas other mention obtained skills in work experience and profile sections. Overall, considering that we only used 17 features for this rather complex NLP task, as well as an artificially constructed training set, we find these results quite satisfactory.



(a) Illustration of the word clusters, where each dot is a word vector from the respective group.



(b) Examples of similar words for 7 different functions.

Figure 3.1: Projections of word embeddings on a 2-D plane.

Figure 3.2: Cosine similarities between example resume lines (y-axis) and word categories (x-axis). Each line was transformed into a mean line embedding using our custom word embeddings model. Then, for each word category 500 most frequent words were identified and their word embeddings were again averaged to form the respective group embedding. Finally, cosine similarities were calculated between each line embedding and each group embedding.

Chapter 4

Ranking Candidates for Job Openings

This chapter explains what LTR experiments were conducted and how. The two primary goals behind the experiments were to understand how different feature sets influence the performance of an LTR model, as well as to compare the machine-learned ranking to that of the currently used system. Section 4.1 explains what data was collected, Section 4.2 looks at how various features and feature sets we constructed, Section 4.3 explains what evaluation metrics were used, Section 4.4 gives a background on the LTR models that were trained, Section 4.5 outlines the way our experiments were performed and Section 4.6 summarises the results.

4.1 Data

Before we could train an LTR model, we needed to collect the appropriate information. This section provides the details on various candidate and vacancy data sources that were available to us (all within YoungCapital's database), as well as on the quality of the data contained in them. A general overview on collected candidate and vacancy information can be found in Table 4.1, which also demonstrates that at the outset of our project we set aside 500 randomly selected job openings and respective candidate lists. The lists in the training and validation set were used to experiment with different feature sets and to arrive at the final model, whereas the lists in the hold-out dataset were used to evaluate the performance of our final LTR model versus ElasticSearch ranking.

4.1.1 Candidate-Job Evaluations

User feedback regarding the relevance of search results, frequently referred to as relevance feedback, is commonly used as an outcome variable (label) in LTR framework, and it can

Table 4.1: An overview of training and hold-out datasets

Data Source	Training/Validation Set	Hold-out Set
Job Profiles	4,625	500
Job Descriptions	4,625	500
Candidate Profiles	84,602	14,622
Candidate Resumes	65,268	11,731
Job-Candidate Evaluations	145,657	16,234

be either binary or graded. Binary relevance feedback is simply an indication whether document was deemed relevant or not for the particular query, whereas graded relevance feedback specifies the degree of relevance, such as very relevant, somewhat relevant and irrelevant. Each query-document should only have one associated relevance score.

In case of a job-candidate matching it is common to use hiring decisions, namely whether a candidate was accepted or rejected for the job [7, 8]. For our experiments, however, we decided to use a slightly different type of information, namely whether candidate was suitable for the next step in the recruitment process (such as interview at a YoungCapital branch or meeting with a client) following a job application. These data were readily available and could be easily traced to respective job opening details and candidate information.

We refer to this type of data as recruiter evaluations, and it should be stressed here that we only had these evaluations for candidates that applied for vacancies administered by YoungCapital. Each candidate could get a positive, neutral or negative evaluation with respect to the job. In order to transform this information into a numeric label we needed to decide how to treat neutral evaluations, and whether to use binary or graded relevance. After running some experiments it was decided that 0-1 encoding should be used, where 1 stands for suitable, whereas 0 for all other evaluations. Figure 4.1 summarises what evaluations were present in our dataset and what label each evaluation was transformed into, along with demonstrating how many job-candidate pairs received a certain evaluation. Overall, our training set contained 34,874 (24%) positive evaluations, and 110,783 (76%) negative/neutral ones.

4.1.2 Vacancy & Candidate Data

As discussed in Section 2.1.2, each job has a profile and a description. Apart from the information entered by recruiters, we also record what cluster different jobs belong to. The clustering model was built by the company's Data Science team to assign jobs that are likely to attract the same type of candidates to the same job cluster, with 14 clusters in total. For instance, there is a "Traineeships cluster" since it was discovered that candidates that are looking for traineeships rarely apply to other types of jobs. "Easy, outgoing jobs" cluster, on the other hand, includes job functions like being a festival staff member, distributing flyers

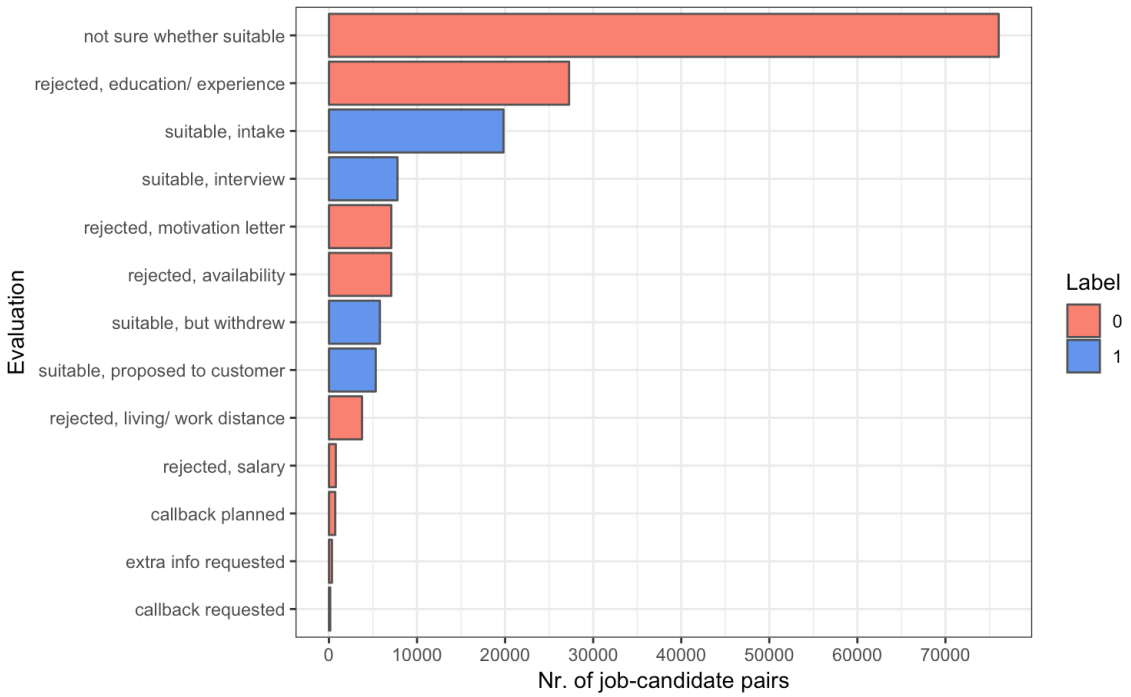


Figure 4.1: Counts of different recruiter evaluations.

or filling in as a part-time shop assistant, all of which could be appealing to candidates looking for a simple temporary job.

As a result, for every job opening we gathered three types of information: job description, job profile data and predicted cluster, where the model that predicts clusters uses the first two sources, thus a cluster could be assigned to any job in the database. Note that job description is stored in a semi-structured format, such that job title, job function and job requirements are easily identifiable. Overall, data quality with regards to job openings was at a reasonably high level, with minimal missing values.

The same cannot, unfortunately, be said for the quality of candidate data. For candidates there are two main sources of information: profile and resume. It is not required to submit a resume when signing up, therefore around 23% of candidates from our job-candidate pairs have resumes missing. All the resumes that were available went through our resume parser to get personal information removed.

Most of the profile information that we are interested in, on the other hand, resides in required fields at sign up, therefore there were few missing values. There is another difficulty, however. There are a lot of candidates that have created an account with YoungCapital a few years back, but haven't necessarily updated their information since. We thus expect a substantial amount of noise in the data for candidates that have old profiles.

4.1.3 ElasticSearch Queries & Lists

In order to compare the current ElasticSearch-based system to our LTR model, we needed to find out how ElasticSearch would rank lists in the hold-out dataset. We therefore constructed 500 "standard" queries for each job posting (namely queries that look for candidates fitting the job profile the best as described in Section 2.1.3), and retrieved ranked candidate lists. In particular, we identified the logic behind query generation, determined what fields are different between queries for different vacancies, and wrote a script that could automatically fill out those fields. Each list only contained candidates that we had evaluations for, in other words we did not consider job-candidate pairs for which we did not have a label.

Due to the setup of ElasticSearch, however, we did not always get a full list back. This implies that when searching for candidates with ElasticSearch, occasionally a few applicants were dropped from the resulting list as they did not match some aspect(s) of the query. In 8% of the cases the search engine returned none of the evaluated candidates.

Since the goal was to use ElasticSearch ranking as the baseline method that LTR models could be compared to, we had to make one of the following choices: either append missing candidates to ElasticSearch results at the bottom of the list in random order, or remove job-candidate pairs omitted by ElasticSearch from the hold-out dataset all together. Both of these options would make all lists have the same content and length, irrespective of ranking method used, otherwise it would not be possible to make a meaningful comparison between different methods. For example, consider two ranking methods that are equally good at ranking any given list, but then we let the first method to only rank a subset of

any provided list, whereas method 2 could always rank the full list. If the first method occasionally gets empty subsets and thus receives ranking score of 0 for those lists, then average score over all lists for method 1 would by definition be lower that that for the second method, as the latter would always have more non-zero scores (i.e. positive) scores which push its average score up.

Considering that in this thesis we are interested in the ranking quality of a method rather than its retrieval ability, we decided to proceed with the first choice, namely to add missing job-candidate pairs to ElasticSearch output, such that all candidates that have an evaluation for a certain job would always be included in resulting ranked lists. Since by doing so we introduced an element of randomness, and since we wanted to have an idea whether our baseline ElasticSearch results are any different from lists ranked at random, we also generated some data as described below.

4.1.4 Random Ranking

For each of the 500 hold-out lists, we performed a random permutation of items in the list and calculated different evaluation metrics for each list (see Section 4.3). In other words, we calculated what scores would different lists get if all candidates were ranked at random, and we could then average the results across all 500 lists to get the overall mean scores.

We then repeated this procedure a 1,000 times, recording evaluation results in each iteration. As a result, we had 1,000 samples of average scores, which provided us with an idea of evaluation results we could get by chance alone on our hold-out dataset.

4.2 Features & Feature Sets

Transforming data into features is a crucial step in creating a machine learning model, and the quality of the features often determine a model's performance. In the majority of cases the dataset for LTR problems consists of feature vectors defined on query-document pairs, which is in sharp contrast to traditional machine learning models where features are usually specified for each individual object. Moreover, the query-document feature vector can be split into three distinct parts:

- Match features, which indicate how well a given document matches the specific query
- Query-independent features
- Document-independent features

It is quite common to see all three types of features combined together into one feature vector.

For our learning-to-rank problem we created different sets of features to experiment with, all of which fall into one of the three categories discussed above. Summary and a short

description of all individual features can be found in Appendix 1, Table A.2, whereas main ideas of each feature set are provided below.

4.2.1 Basic Match Features

Given the availability of structured and unstructured data, we split match features into two different sets: Basic Match Features (BMF) and Extended Match Features (EMF). The former set measures similarities between candidate and job profiles, or more precisely a match between candidate and job opening education, location, employment type and function fields. Essentially, BMF match the same information as ElasticSearch, but the way the comparison is made is slightly different.

There are six features in total that belong to this set, and three out of those (regarding preferred locations, job types and functions) are calculated using Jaccard Similarity $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$, where A and B are sets of candidate and job tags (examples of profile tags can be found in Table 2.1). Here we wanted to capture the idea that, since candidates can tick as many boxes for their job preferences as they wish, candidates with more selective preferences should have a higher similarity score than those who mark every possible option.

Since education tags are ordinal data, they were converted to integers. There are four education types in total - Middelbare school, MBO, HBO and Universiteit - that were assigned numbers 1 to 4, respectively. For candidates we then retrieved their maximum level of education, whereas for job openings - the minimum education required, and calculated the match between the two using a custom formula because we wanted to capture a non-linear relationship between different levels and get a score bounded between -1 and 1. Furthermore, we also created a simple education match feature that shows whether candidate is over- or under-educated for a specific job.

The last feature determines how far a certain candidate lives from the respective job location by calculating the great-circle distance between candidate's home address and job's primary address. Overall, the goal behind creating basic features was to see how well could a ranking model perform using readily available profile information about candidates and job openings only.

4.2.2 Extended Match Features

Extended Match Features (EMF), on the other hand, use textual information to capture the match between candidates and jobs. For languages and skills we simply calculated the proportion of required attributes stated in a job description that matched attributes parsed from candidate resumes. We also calculated the overall proportion of words in a job description that matched resume words (without personal information).

For more complex features we again used word embeddings (see Section 3.2.2). As we were comparing job information to candidate resumes, we updated our word embeddings model to also include all the vocabulary from available job openings. Using this model we

calculated cosine similarities between job functions and required experience to information in the work experience sections in resumes, which is comparable to the way we computed similarities between lines and word groups in the resume parser.

4.2.3 Vacancy & Resume Features

In addition to match features, for each job-candidate pair we created separate high-dimensional vectors, one representing the vacancy, and another - the resume. These vectors were then concatenated together with match data to form a single feature vector.

The only information that was used as vacancy features were job clusters, as we felt that they reasonably capture the differences between jobs. For resumes, however, we used a completely different strategy as we wanted to capture as many unique properties of candidates and their resumes as possible.

Below are four different ways to transform text to features that we experimented with: Bag-of-Words (BoW), Term Frequency - Inverse Document Frequency (TF-IDF), Mean embeddings (Mean Word2Vec) and Smooth Inverse Frequency embeddings (SIF Word2Vec). For each approach we briefly discuss their advantages and disadvantages.

BoW. The simplest way to transform text to a feature vector for any document is undoubtedly the Bag-of-Words approach. In this case every word becomes a feature by itself, and the value is the raw count of that word in the document.

There are two main disadvantages associated with the BoW method. As the number of documents grows, so does the number of features, which leads to a large and mainly sparse feature matrix, as most documents will only contain a small fraction of all the words available in the whole corpus of documents. It is thus common to limit the features to only top n words found in the corpus (after removing so-called stop-words), where the number n has to be decided by researchers.

Another disadvantage is that some words might have a high count in a document, but contribute very little to the meaning of that document. For instance, in the field of recruitment words like “experience” and “responsibilities” could occur very frequently in a job description, but might not be very helpful in determining what field the described position is in.

TF-IDF. In order to overcome the aforementioned challenge, raw word counts are routinely transformed into respective TF-IDF scores which indicate statistical importance of a word for the specific document. TF-IDF scores are a product of Term Frequency and Inverse Document frequency, which are calculated as follows. Let $f_{t,d}$ be the number of times the term t occurs in document d, where $d \in D$ for D representing the set of all documents in the corpus. Also let N be the total number of documents in a corpus, and let $|d \in D : t \in d|$ be the number of documents where the term t appears. Then:

$$tf_{t,d} = f_{t,d}$$

$$idf_{t,D} = \log \frac{N}{|d \in D : t \in d|}$$

$$\text{TF-IDF} = tf_{t,d} \times idf_{t,D}$$

Athough TF-IDF scores do, arguably, provide a model with a better representation for words in a model by indicating relevance of each word present in a document, they do not change the number of dimensions needed to encode the whole corpus. In essence, the shape of the sparse feature matrix is the same as that of the BoW representation, with the only difference being the TF-IDF scores instead of raw word counts as values in that matrix.

Furthermore, both BoW and TF-IDF methods ignore the semantic similarity between words and fail to capture the relationship between different terms in the corpus. Each word acts as a standalone feature, and terms like "chauffeur" and "driver" in sentence classification task, for instance, would be treated by a model as separate variables, thus potentially failing to predict correct labels for the respective sentences.

Mean Word2Vec. We have already discussed word embeddings and their advantages in detail in the previous chapter, therefore here we would like to point out what difficulties can be encountered when using word embeddings to construct features for a whole document.

Let a document d have m words, where each word vector was trained to have n dimensions. Then the whole document could be represented as an $m \times n$ matrix, which, unfortunately, could not be used directly with traditional machine learning models as they expect a single feature vector for each document, not a matrix.

The simplest way to get such a vector is to take an average of all word vectors. The resulting n-dimensional document vector could then directly serve as a feature vector. The downside of this approach is that each word has equal weight in calculating the resulting mean vector, therefore if the document contains a lot of "filler" words, the meaning of the document could be lost.

SIF Word2Vec. Several suggestions have been made in literature to use a weighted sum of word vectors instead, and approach that we tried in this thesis was proposed by Arora et al. [28]. They introduced a new word weight function called smooth inverse frequency (SIF)

$$w_i = \frac{a}{a + \text{tf}_{ic}}$$

where w_i is the weight function for the word i, tf_{ic} is the corpus wide term frequency of the word i, and a is a configurable parameter which we set to 10^{-3}. The authors also proposed to apply a post processing step to the weighted sum that they referred to as a common component removal. This approach to constructing sentence and document embeddings achieved significantly better performance than baselines on various textual similarity tasks

in the original research [28], and showed promising results on comparing TripAdvisor reviews [29]. To our knowledge, this method has not yet been used in LTR problems.

Consequently, for our experiments we constructed four different feature sets from resumes, two count-based and two based on word embeddings. For all approaches stop words and personal information were excluded. Furthermore, for BoW and TF-IDF we selected 500 most frequent words that appeared in our training resumes, whereas for word2vec methods all non-excluded words were used to create individual document embeddings. It should be noted here that for count-based methods we also tried increasing the number of features to 1,000 words. This step, however, only increased the training time while not providing any considerable gain in performance, which is why we reverted back to 500 words. All missing values across the four feature sets were treated as zeros.

4.3 Evaluation Metrics for Ranking Problems

In order to evaluate performance of a ranking model, it is first of all important to establish what a "good" ranking means. Ideally, we would like our model to order documents in a decreasing order of their actual relevance scores (ground truth labels), such that all highly relevant documents are at the top of the list, and all irrelevant - at the bottom. It is very rare to have such a perfect ranking, however, so when there are two lists with equivalent numbers of relevant and irrelevant documents at the top, we define the list with higher ranks for relevant documents as being better.

This implies that an evaluation metric for a learning problem should take into account the position of documents in the list. There are several measures frequently used in Information Retrieval field that are able to do so, and in subsections below we will focus on two of those: Normalized Discounted Cumulative Gain (NDCG) and Mean Average Precision (MAP). Furthermore, we also include an easy to interpret Precision which only evaluates the proportion of relevant items in a list, regardless of rank. More information on all of these metrics can be found, for instance, in [14] or [30].

NDCG. The idea behind NDCG is quite simple - the higher the document on the list, the more its relevance score should contribute to the overall evaluation grade. This idea is based on the assumption that only the documents at the top of the list are valuable to the end user, because documents lower on the list are less likely to be viewed at all [31]. In order to express the above mathematically, we look at two main components of NDCG: Discounted Cumulative Gain (DCG) and its ideal version (iDCG). Let y be the relevance scores of ranked documents associated with a certain query q, such that y_i is the relevance score of the document at rank i. Then the sum of all n relevance scores is $\sum_{i=1}^{n} y_i$. When

using a logarithmic discount function, we get

$$DCG(q) = \sum_{i=1}^{n} \frac{y_i}{log_2(1+i)}$$

which is the DCG score over all n documents in the list. The ideal DCG is the score that the list would get if all documents were ranked in the best possible way, namely in decreasing order of their true relevance scores. Some refer to iDCG as maximum DCG for that list of documents, and it is used to normalize the DCG score

$$NDCG(q) = \frac{DCG}{iDCG}$$

such that NDCG score is bounded between 0 and 1, and could be used to assess performance of ranking functions for different lists and queries. Furthermore, since in theory y_i could be any positive number, NDCG is a great evaluation metric for documents with graded relevance feedback. The cutoff version of NDCG is NDCG@k, which only calculates NDCG score for the first k documents on the list instead of all N. Some of the common choices for the value of k are 5 and 10, and in general it is suggested to choose this value based on the size of the dataset [32].

Precision and MAP. Precision measures the proportion of the results that are correct, as mentioned in Section 3.2. In ranking context it is common to consider Precision at position k (Precision@k or P@k) [14], which, analogous to NDCG@k, only evaluates the number of relevant items retrieved among the first k items on the list:

$$P@k(q) = \frac{\#\{\text{relevant documents in the top } k \text{ positions}\}}{k}$$

Unlike NDCG@k, however, Precision@k does not take into account the position of those relevant items, therefore an Average Precision (AP) metric is commonly used alongside (or instead of) it:

$$AP(q) = \frac{\sum_{k=1}^{n} P@k(q) \cdot l_k}{\#\{\text{relevant documents}\}},$$

where n is the total number of documents in the list related to query q, while l_k is an indicator function that equals 1 if the item in the k-th position is a relevant document, zero otherwise. The mean value of AP taken over all queries is known as MAP [14]. Both AP and MAP take values from 0 to 1, and can only be calculated if relevance feedback for queries is binary.

To evaluate the results of our experiments, we decided to calculate NDCG@10, AP and P@10 for each list of candidates associated with a certain job, as these evaluation metrics would give us an idea how well could different methods rank candidates in the first 10 positions, overall in the whole list, and how many relevant (read: suitable for an

interview/intake) candidates there are among the first 10. All three of the chosen metrics can have values between 0 and 1, and the closer the results are to 1, the better the ranking of the respective lists.

4.4 Models

We applied two different LTR algorithms to our data:

- **Linear regression.** A simple least-squares linear regression algorithm was used as a baseline ranking approach. This pointwise approach (see Section 2.2.2) produces a score per document by which results are then sorted.
- **LambdaMART.** LambdaMART [33] is the state-of-the-art LTR algorithm that demonstrated impressive results both in the job-candidate matching domain [7, 9] and in other information retrieval tasks [16]. This is a gradient boosting, pairwise approach that optimizes document order.

4.4.1 Linear Regression

Linear regression is the well-known model:

$$\mathbf{y} = \mathbf{X}\beta + \mathbf{e}$$

where $\mathbf{y}$ is a vector of observations, $\mathbf{X}$ is a matrix of explanatory variables (features) and $\mathbf{e}$ is a vector of randomly distributed errors. The least squares estimate of the vector of unknown parameters, β, is calculated as:

$$\hat{\beta} = (\mathbf{X^T X})^{-1}\mathbf{X^T y}.$$

The only difference between using the linear model for traditional statistical learning tasks and learning-to-rank problems is the fact that feature vectors in the matrix $\mathbf{X}$ are defined on query-document pairs, and predictions are used to order these pairs.

We chose this model as we wanted a simple ranking algorithm that we could compare to our main model, LambdaMART. Since the linear regression technique in its basic form does not require any hyper-parameter tuning and is very fast to train, it was a good candidate.

4.4.2 LambdaMART

LambdaMART is a combination between two techniques: LambdaRank and Multiple Added Regression Trees (MART) [33]. LambdaRank is a method for learning arbitrary information retrieval measures, and can be used with any algorithm that learns through gradient descent [34]. MART is one of those compatible algorithms; it is a boosted tree model, where the output of the model is a linear combination of the output of several regression trees [35].

The key observation made in [33] is that in order to train a ranking model, only gradients (or λ's) are needed, not the costs themselves. These λ's can be interpreted as forces: if a document d_i is more relevant than a document d_j, then d_j will get a downward push of size $|\lambda|$, whereas d_i will get an equal push upwards. The rank positions of all other documents will remain unchanged. Formally, if the chosen objective function that we wish to optimize is NDCG, then:

$$\lambda_{ij} = \frac{-\sigma}{1 + e^{s_i - s_j}} |\Delta \text{NDCG}|$$

where σ is a hyper-parameter that determines the "smoothness" of the derivative, s_i and s_j are relevance scores of the respective documents, and ΔNDCG is the size of the change in NDCG when documents are swapped.

The task of MART is to learn these λ's. As with other gradient boosting techniques, training of the LambdaMART algorithm is stopped only when a certain number of boosting iterations have been completed. This number of iterations, along with other tree-based parameters, should be configured by a researcher.

4.5 Experiments

The sections above explained what data was gathered, how it was transformed into features and what algorithms we wanted to experiment with. This section will, in turn, outline what experiments were performed and how, whereas Section 4.6 will present the results.

4.5.1 Cross-Validation with Feature Sets

The first set of experiments that we conducted focused on different feature sets and their impact on ranking performance of LTR models. We chose to conduct such experiments because, to our knowledge, there is no research that compared different resume representations or included structured candidate profile information as features (see Section 2.2.2).

As for these experiments we did not want the results to be dependent on one particular test set, we used 5-fold cross-validation and measured ranking performance in each fold for different combinations of feature sets. Folds were formed by splitting lists into 5 equal parts, instead of splitting individual job-candidate pairs, and only the job-candidate pairs that were assigned to the training set were used in cross-validation (Table 4.1).

When the best-performing combination of feature sets was discovered, we performed an elimination procedure to better understand the impact of each separate feature set on the performance of ranking models. In other words, we first measured the ranking performance of the model with all of the selected feature sets, and then removed one of the sets and assessed the outcome. This process was repeated until all the combinations with one of the sets missing were tested, again using 5-fold cross validation.

Both Linear Regression and LambdaMART algorithms were used to train ranking models with different sets. Although LambdaMART does support hyper-parameter tuning,

for these experiments we left the default values unchanged.

4.5.2 Hyper-Parameter Optimization & Methods Comparison

Once we determined what combination of features works best, we performed hyper-parameter tuning for that specific feature vector, and selected the parameters that gave the best NDCG@10 score. As a result, we then trained the final LambdaMART model with 300 trees, maximum depth of 6, sampling fraction rate per tree 1 and learning rate of 0.1, using the whole training dataset. Similarly, we trained the Linear Regression on the full training set as well.

We then compared different ranking methods (Random Ranking, ElasticSearch, Linear Regression, LambdaMART) using the 500 lists from the hold-out dataset (Table 4.1). Here the main aspect we wanted to study was whether LTR models perform better (read: produce higher scores for the chosen evaluation metrics) than the current system, and to what extent. In other words, we needed to assess whether the mean difference between lists produced by ElasticSearch and those ranked by LTR models was significantly different from zero. The chosen statistical test was a paired sample t-test, which was appropriate because in our experiment we were re-ranking observations in the same hold-out dataset, and hence we were comparing how the same lists could be ranked with different methods.

4.6 Results

The results of the aforementioned experiments are summarised below. We will first look at the findings related to using various feature sets, and then we will review the outcomes of applying different ranking methods to the hold-out dataset.

4.6.1 Feature Sets

Table 4.2 and Table 4.3 demonstrate how adding or substituting different feature sets impacted the ranking results using Linear Regression and LambdaMART respectively as training algorithms. Here the baseline is simply the order in which the records were saved in the database, which is the order in which candidates applied for a job. All the presented evaluation metrics (NDCG@10, MAP and P@10) were averaged over 5 folds. It is clear that for both algorithms adding more feature sets had a positive impact on ranking, although in case of LambdaMART adding job clusters (JC) appears to result in a negligible improvement (Table 4.3). This is in sharp contrast with Linear Regression results, where adding JC resulted in a visible gain in all ranking scores (Table 4.2).

With respect to comparing different resume representations (BoW, TF-IDF, W2V, SIF), the results are not as straightforward. It does seem that going from count-based representations (BoW and TF-IDF) to word embeddings-based representation (W2V and SIF) improves ranking for both algorithms in terms of all ranking scores, but within

Table 4.2: 5-fold Cross-Validation results, Linear Regression

Feature sets	NDCG@10	MAP	P@10
Baseline (no re-ranking by the model)	0.370	0.340	0.249
Basic Match Features (BMF)	0.392	0.358	0.263
BMF + Extended Match Features (EMF)	0.420	0.380	0.274
BMF + EMF + Job Clusters (JC)	0.435	0.391	0.281
BMF + EMF + JC + Resume BoW (R-BoW)	0.448	0.401	0.287
BMF + EMF + JC + Resume TF-IDF (R-TFIDF)	0.448	0.403	0.286
BMF + EMF + JC + Resume Mean Word2Vec (R-W2V)	0.454	**0.408**	**0.288**
BMF + EMF + JC + Resume SIF Word2Vec (R-SIF)	**0.455**	0.407	**0.288**

Table 4.3: 5-fold Cross-Validation results, LambdaMART. All hyper-parameters were set to default.

Feature sets	NDCG@10	MAP	P@10
Baseline (no re-ranking by the model)	0.370	0.340	0.249
Basic Match Features (BMF)	0.399	0.364	0.267
BMF + Extended Match Features (EMF)	0.440	0.394	0.282
BMF + EMF + Job Clusters (JC)	0.440	0.395	0.283
BMF + EMF + JC + Resume BoW (R-BoW)	0.453	0.406	0.289
BMF + EMF + JC + Resume TF-IDF (R-TFIDF)	0.452	0.404	0.289
BMF + EMF + JC + Resume Mean Word2Vec (R-W2V)	0.459	0.410	**0.291**
BMF + EMF + JC + Resume SIF Word2Vec (R-SIF)	**0.461**	**0.412**	0.290

approaches of the same type there is hardly any difference. For instance, when studying the results for feature sets with BoW and TF-IDF resume representations in Table 4.2, we notice that NDCG@10 value is approximately the same, MAP is slightly higher whereas P@10 is slightly lower for TF-IDF.

Nevertheless, it is clear that the best results for both Linear Regression and LambdaMART could be achieved when combining BMF, EMF, JC, and resume information captured in word2vec-based embeddings. For our final models we decided to use SIF transformation of resumes as the last feature set. Then, as we wanted to better understand how these four different feature sets influenced the ranking results, we eliminated each set from the full feature vector one at a time. Results of this procedure can be found in Table 4.4. For each model the outcomes were obtained through a 5-fold cross validation, and the presented percentages illustrate the relative change in performance when compared to the model trained with all features.

There are several observations that can be made. Firstly, removing resume features

Table 4.4: Impact of removing feature sets from the full feature vector on ranking results.

	Linear Regression			LambdaMART		
Removed Set	NDCG@10	MAP	P@10	NDCG@10	MAP	P@10
BMF	0.450	0.402	0.286	0.453	0.406	0.287
	(-1.19%)	(-1.36%)	(-0.82%)	(-1.63%)	(-1.47%)	(-1.16%)
EMF	0.435	0.390	0.279	0.449	0.402	0.288
	(-4.65%)	**(-4.46%)**	**(-3.24%)**	(-2.49%)	(-2.35%)	(-0.83%)
JC	0.437	0.393	0.282	0.460	0.411	0.290
	(-4.16%)	(-3.59%)	(-2.22%)	(-0.04%)	(-0.17%)	(0.00%)
R-SIF	0.435	0.391	0.281	0.440	0.395	0.283
	(-4.66%)	(-4.18%)	(-2.59%)	**(-4.75%)**	**(-4.32%)**	**(-2.75%)**

(R-SIF) results in the largest decrease in NDCG@10 score for both algorithms. This, however, seems to be the only common effect, as eliminating other feature sets has quite a different impact on our ranking models. For Linear Regression, discarding BMF has the smallest relative decrease in performance, while removing any of the other three sets has a considerably large negative impact. This is in contrast to LambdaMART results, where removing JC has an almost negligible effect (close to 0% relative decrease in performance in term of both NDCG@10 and P@10), and discarding EMF has a much smaller impact than discarding R-SIF. Such a result was not unexpected, however, as we already noticed above (Table 4.2 and Table 4.3) that adding JC to other feature sets had a much smaller effect on LambdaMART model than on Linear Regression.

All of the aforementioned outcomes indicate that LambdaMART learns very different patters from data than Linear Regression, and overall we observed that even with default hyper-parameters LambdaMART appears to produce better rankings than Linear Regression, for all feature sets (Table 4.2 and Table 4.3).

4.6.2 Comparison to Random & ElasticSearch Rankings

The next step was then to train the LambdaMART algorithm with appropriate hyper-parameters, and compare its performance to the current system (ElasticSearch) on the hold-out dataset. To provide a complete picture, in Table 4.5 we also include results of applying random permutations to the hold-out dataset as described in Section 4.1.4 and illustrated in Figure 4.2. In short, Figure 4.2 demonstrates what ranking results can be expected when lists in the hold-out dataset are ranked completely at random. For instance, only in 1 out of 1,000 iterations MAP score of randomly permuted lists was greater or equal than that of our baseline ElasticSearch ranking (Table 4.5). This implies that it is rather unlikely that ElasticSearch orders results at random.

From the Table 4.5 we also see that LambdaMART performed noticeably better than

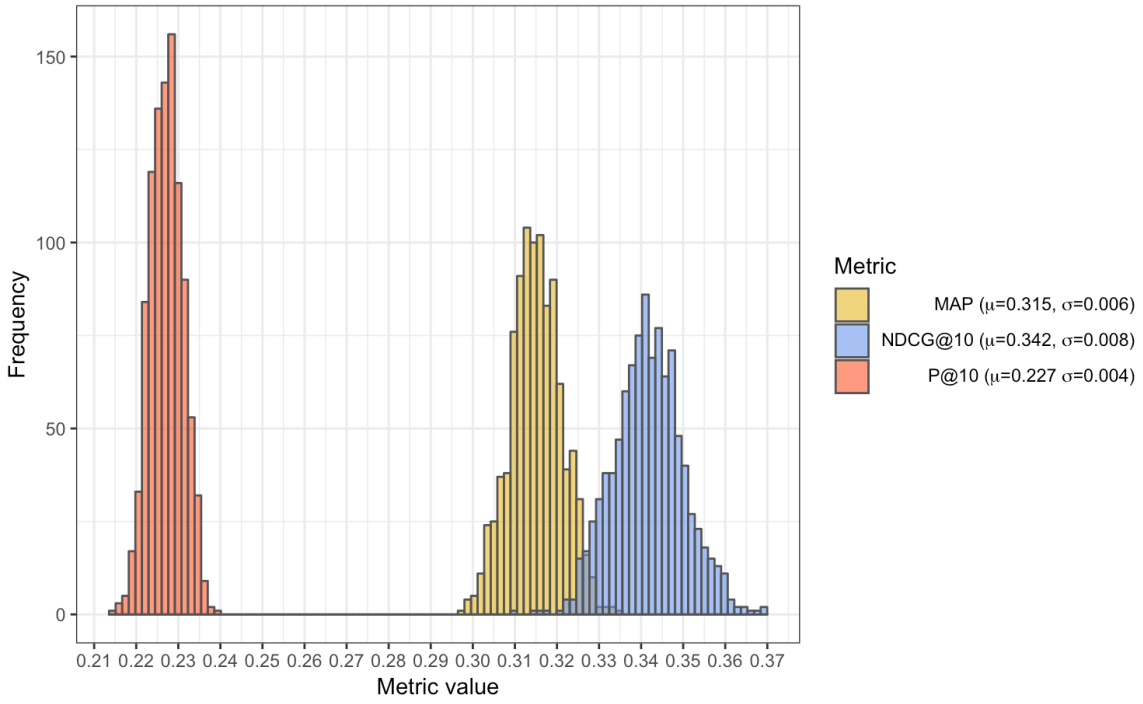


Figure 4.2: Histograms of evaluation metrics, generated from applying random permutations to each list in the hold-out dataset 1,000 times. In each iteration candidates in every list were ranked at random, and NDCG@10, AP and P@10 for individual lists were calculated. Results were then averaged across all lists to arrive at a mean value for each metric, and the whole process was repeated 1,000 times.

Table 4.5: Hold-out dataset results

Method	NDCG@10	MAP	P@10
Random Ranking (mean values)	0.342	0.315	0.227
ElasticSearch (**baseline**)	0.369	0.335	0.244
Linear Regression (best features)	0.449 (+22%)	0.397 (+19%)	0.280 (+15%)
LambdaMART (best features + hyperparameters)	0.473 (**+28%**)	0.417 (**+24%**)	0.289 (**+18%**)

ElasticSearch, with 28% improvement in NDCG@10 score and 24% improvement in MAP score. Both of these metrics indicate that our best model managed to put some of the suitable candidates higher on the list, more often than the current method. It is easier to interpret P@10 metric, however. We can say that, on average, after viewing 2 lists with top ten candidates, recruiters would find approximately 1 more candidate if those lists were ranked with LambdaMART rather than ElasticSearch. All of these results were statistically significant (p-value $< .001$) according to the paired-sample t-tests, even when accounting for multiple testing with Bonferroni correction.

Figure 4.3 provides a more detailed view on these results. For each of the 500 lists in the hold-out dataset we computed the difference between metric values of LambdaMART and ElasticSearch. The resulting histograms indicate that, although on average our model performed better than baseline, there were job openings for which LambdaMART ranking of candidates was worse than the current system's (lists for which difference was less than 0). For instance, there were 138 lists in total that received a worse NDCG@10 score when ranked by LambdaMART rather than ElasticSearch. Furthermore, for each metric we see that there was a considerable number of queries that were ranked in a similar manner (difference around 0).

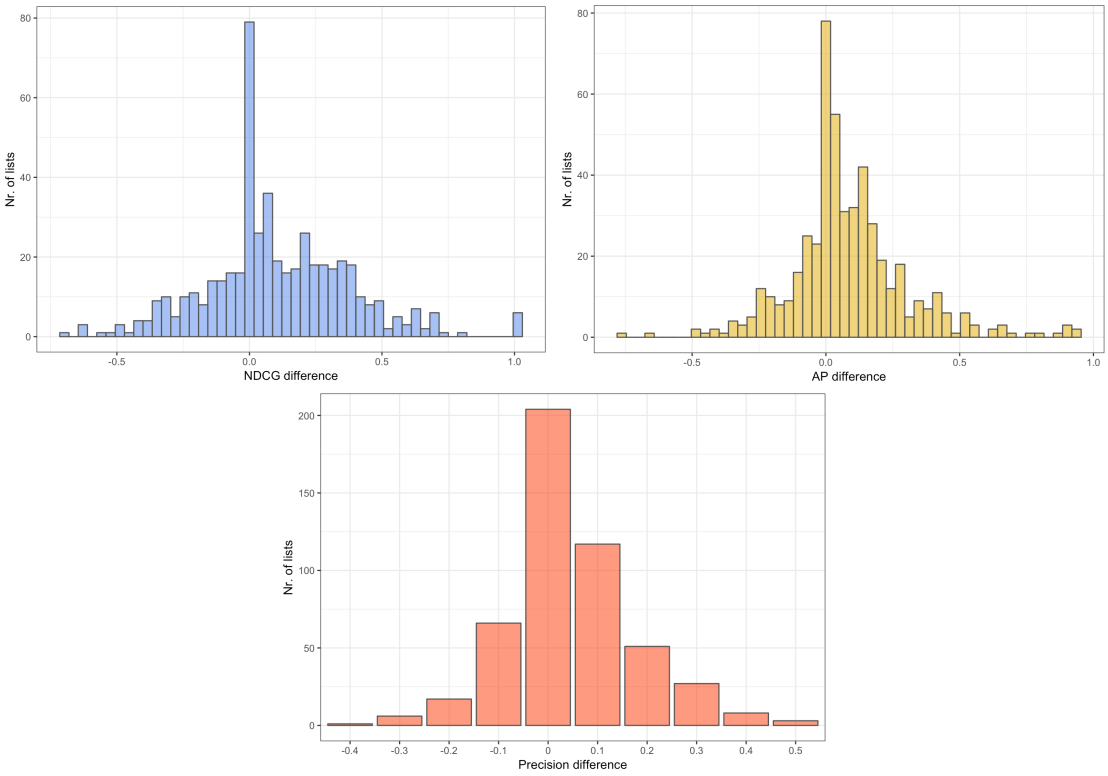


Figure 4.3: Per-vacancy ranking score differences: LambdaMART model minus ElasticSearch baseline. Positive difference implies that LambdaMART ranking score for a certain list of candidates was higher than that of ElasticSearch for the same list, whereas negative score means that the score was lower.

Chapter 5

Discussion

Seeing the results of resume parsing model and LTR experiments, we will discuss the limitations (Section 5.1) and implications (Section 5.2) of our research, as well as provide practical guidelines on where and how could our model be used (Section 5.3). Suggestions for future research will be listed in Section 5.4.

5.1 Limitations

One of the limitations of our study is that features for both the Resume Parser and LTR models were generated primarily with the Dutch language in mind. Some job postings and submitted resumes were, however, occasionally in other languages, especially English and German. Therefore, if there was a language mismatch, it is likely that all text-based match features for the respective job-candidate pairs were quite poor. Resume parsing model also rarely managed to correctly predict sections for non-Dutch CVs.

Furthermore, for a long period of time YoungCapital's primary target group was students interested in part-time jobs to support their studies. Even though in recent years the company has been expanding into fields other that temporary work, a significant proportion of resumes in our dataset still belongs to people with barely any work experience, and a number of job descriptions do not have any hard requirements apart from education. It is likely that for such jobs recruiters decide whether candidate is suitable or not based on the personality fit, which our model in it's current form cannot assess.

Another limitation might be the fact that, when comparing different resume representations, we did not use models that might be better suited to analyze sparse data. For instance, it is known that Naive Bayes and Support Vector Machines give good results on text classification tasks when document features are BoW or TF-IDF, and we did not use these models in our experiments. Hence, one should keep in mind that the finding concerning word2vec-based features performing better than count-based features for ranking candidates has only been verified for LambdaMART and Linear Regression models.

Our comparison to the current system has a drawback as well, since we only considered "standard" queries that didn't include any keywords. It is possible that the difference in ranking performance between our model and ElasticSearch would have been smaller if we did use relevant keywords when retrieving candidate lists, however we could not test this hypothesis in the current experimental setup that only relied on historical data and did not require any extra input from recruiters. Moreover, it has already been shown that LambdaMART still performs better than ElasticSearch even with manually compiled queries [9].

5.2 Implications

The results from the LTR experiments have several implications for research in the job-candidate matching field. Firstly, we demonstrated a way to create match features from structured profile information and free-text documents that have an impact on ranking candidates for jobs. Therefore, as the majority of previous analyses incorporated ElasticSearch match scores as match features, our examples can serve as an inspiration for researchers that do not work with the specified search engine, but do wish to create meaningful pair features.

Furthermore, the finding that resume features based on word embeddings have an advantage over count-based features imply that it might be easier for the future research to make a decision on how to incorporate resume information into, for instance, a LambdaMART model.

Moreover, other work in the LTR field outside of the recruitment domain could also potentially enhance performance of ranking models by incorporating the document embedding dimensions as features. Published research on including word embeddings in LTR models that we are aware of only used cosine similarities between a vector of words in document body/title and a vector of words that appear in the query [36], or incorporated word embeddings into a complex neural network architecture [37].

5.3 Implementation Ideas

In the context of YoungCapital's recruitment software our research has straightforward use cases. As we have shown that our final ranking model (LambdaMART) produces much better results than currently used ElasticSearch when "standard" queries are used for retrieving a list of candidates, we suggest to apply the model to re-rank search results outputted by the search engine. In other words, the list that is initially produced by the current system could be ranked by the LambdaMART model before showing it to the recruiters, as then more suitable candidates will appear higher on the list. This process would hopefully shorten the time spent on head-hunting and let the recruitment personnel concentrate on other tasks.

Furthermore, our LTR model could be used not only for candidate sourcing, but also for ranking candidates who applied for a job. Some vacancies that are located in large cities, for instance, receive more than a hundred applications per week, and reviewing every single one could be very time consuming. Therefore we suggest to rank lists of applicants as well, again to help recruiters find suitable candidates faster.

5.4 Future Work

There are several suggestions that we can make for future research:

- Investigate whether there is another way to transform word embeddings into document embeddings, such as using `doc2vec` [38], that could have a greater impact on LTR results.
- Dutch language is known for containing a lot of composite words. It would thus be interesting to investigate whether word embeddings that are based on n-grams instead of full words could improve performance of both the LTR and the resume parsing models.
- Generate different job-level features. Our analysis suggests that Boolean features indicating a job cluster do not provide the LambdaMART model with much information. Could features based on keywords from the job title/ job function work better in combination with existing features?
- Given the same set of features for the resume parser model, investigate whether including manually annotated training resumes with separate labels for section headers could help the CRF model better learn how to identify different sections.
- Explore whether it is possible to identify in advance what lists might be ranked worse than the current baseline.

Chapter 6

Conclusion

In this thesis we have extended previous research on applying learning-to-rank models to a job-candidate matching problem, investigated several aspects related to feature generation and made several comparisons that have not yet been documented in the literature. Overall, we proposed and evaluated two different models, one to handle resumes and another to rank candidates, as we needed a way to parse resumes before we could create features for an LTR model.

Specifically, the first model was developed to help us identify different sections in resumes, as we wanted to remove personal information and extract work and skills related information from the respective sections. Contrary to other research in the CV-parsing field, however, we only used a small number of features that primarily rely on cosine similarities between mean word embeddings, and we predicted labels line-by-line instead of token-by-token. Furthermore, we only engaged in manual annotation to create a test set, as the training set was constructed using semi-structured CV information. This information was gathered from an application that provides templates for individuals to create resumes.

The proposed model, although quite simple and with shallow features, produced surprisingly good results, especially for the personal information section. The section that our model struggled with was the skills section, which we suspect is due to the fact that individuals tend to provide skills information in other sections as well, such as the profile and the work experience sections, or to combine skills with languages or interests. In other words, from our experiments we concluded that the skills section is not as well defined as education and work experience, for example, and that for our use case it was sufficient to create a list of skills instead and then match this list to words used in resumes.

The second model that we built was inspired by the previous research in applying LTR ideas to ranking candidates for jobs. Our main task was to learn a ranking function that could, given a job-candidate feature vector, order candidates such that individuals that fit hard requirements well are higher on the list. In order to do so we utilised historical data that indicated whether an applicant was suitable or not for the next stage of the hiring

process, such as interview or intake, and experimented with various feature sets that we could construct from candidate profiles and resumes.

We demonstrated the impact that different feature sets have on the ranking performance of both a basic ranking model (Linear Regression) and a state-of-the-art ranking algorithm (LambdaMART). The conclusion was that, in both cases, resume information can be captured better by document embeddings rather than word count methods. Furthermore, we established that the LambdaMART model produces better ranking than the current search system used by YoungCapital, and we thus recommend to use this model to re-rank the initial candidate lists retrieved by the search engine, or even order candidates that applied for a vacancy.

The proposed model can only judge hard attributes of an individual, such as whether a candidate's education and experience match those required by a specific job opening. Therefore we suggest to use this model to help recruitment personnel find candidates that have the right background, so that recruiters could spend more time getting to know the prospective employees and assessing their personality fit.

Appendix A

Model Features

Table A.1: Overview of features used in CRF model for resume parsing

Feature Name	Description
BOCV	Is a line beginning of CV
EOCV	Is a line end of CV
containsOpleiding	Line contains "Opleiding"
containsWerkervaring	Line contains "Werkervaring"
containsVaardigh	Line contains "Vaardigh"
containsKennis	Line contains "Kennis"
moreSpaceAbove	There are more empty lines above
moreSpaceBelow	There are more empty lines below
similarityName	Cosine similarity of the line to name vector
similarityAddress	Cosine similarity of the line to address vector
similarityFunction	Cosine similarity of the line to function vector
similarityEducation	Cosine similarity of the line to education vector
similaritySkills	Cosine similarity of the line to skills vector
similarityInterests	Cosine similarity of the line to interests vector
similarityLanguages	Cosine similarity of the line to languages vector
similarityWork_Summary	Cosine similarity of the line to work summary vector
similarityPitokens	Cosine similarity of the line to personal information tokens vector

Table A.2: Overview of different features and feature sets used in LTR models.

Feature Name	Feature Set	Description
function_match	BMF	Match between candidate function tags and job function tags
job_type_match	BMF	Match between candidate job-type tags and job job-type tags
region_match	BMF	Match between candidate region tags and job region tags
edu_match	BMF	Match between candidate max education and job min education
over_under_educated	BMF	Is candidate's max edu in / above/ below required job educations
dist	BMF	Distance between candidate's zipcode and job's primary zipcode
text_overlap	EMF	Prop. of job description words that are in candidate's resume
text_language_match	EMF	Prop. of languages in job description that match those in resume
text_skills_match	EMF	Prop. of skills in job description that match skills in resume
text_function_sim	EMF	Cosine sim. between job's function and functions found in resume
text_experience_sim	EMF	Cosine sim. between job description keywords and words in resume
clust0 - clust14	JC	Dummy features that indicate top cluster for jobs, 14 in total
word0-word500	R-BoW	Top 500 resume words and their raw count for each resume
word0-word500	R-TFIDF	Top 500 resume words and their TF-IDF weights for each resume
dim0-dim100	R-W2V	100 features from mean of all resume word vectors
sif0-sif100	R-SIF	100 features from applying SIF to all resume word vectors

Appendix B

Software

The data for this project was retrieved from databases with `SQL` language. All the experiments and feature engineering tasks were done using `python` programming language. Details of the primary third-party `python` libraries that simplified our modelling and data handling tasks are provided below.

- `numpy` adds support for large, multidimensional arrays and matrices, as well as high-level mathematical functions to operate on these arrays [39]. This library was mainly used to create evaluation metric functions for LTR models and mean line/document embeddings.
- `sklearn` features various classification, regression and clustering algorithms and is one of the main libraries used for machine learning [40]. We used this library to construct BoW and TF-IDF features, as well as its implementation of Linear Regression in LTR modelling tasks.
- `sklearn_crfsuite` is a sklearn adaptation of `CRFsuite` library for modelling of sequential data [41] and was thus used to create our CRF model for resume parsing.
- `nltk` combines different methods for Natural Language Processing [42]. Removing stopwords and tokenizing sentences were some of the main tasks aided by this library.
- `gensim` is a popular library for unsupervised topic modeling and natural language processing [43], which provided us with implementation of word2vec and thus allowed for training all of our word embeddings.
- `fse` helped us create SIF embeddings from pretrained word embeddings in a simple and efficient manner [44].
- `xgboost` is a library that provides a gradient boosting framework for various programming languages [45]. We employed its implementation of LambdaMART to train our main LTR model.

References

[1] Sharon Pande. E-recruitment creates order out of chaos at sat telecom: system cuts costs and improves efficiency. *Human Resource Management International Digest*, 19(3):21–23, 2011.

[2] Simona Colucci, Tommaso Di Noia, Eugenio Di Sciascio, Francesco M Donini, Marina Mongiello, and Marco Mottola. A formal approach to ontology-based semantic match of skills descriptions. *J. UCS*, 9(12):1437–1454, 2003.

[3] Vladimir Radevski and Francky Trichet. Ontology-based systems dedicated to human resources management: an application in e-recruitment. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 1068–1077. Springer, 2006.

[4] Maryam Fazel-Zarandi and Mark S Fox. Semantic matchmaking for job recruitment: an ontology-based hybrid approach. In *Proceedings of the 8th International Semantic Web Conference*, volume 525, 2009.

[5] Rémy Kessler, Nicolas Béchet, Mathieu Roche, Marc El-Bèze, and Juan Manuel Torres-Moreno. Automatic profiling system for ranking candidates answers in human resources. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 625–634. Springer, 2008.

[6] Amit Singh, Catherine Rose, Karthik Visweswariah, Vijil Chenthamarakshan, and Nandakishore Kambhatla. Prospect: a system for screening candidates for recruitment. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 659–668. ACM, 2010.

[7] Ming Fang. Learning to rank candidates for job offers using field relevance models. Master's thesis, University of Groningen and Saarland University, 2015. Downloaded from `https://lct-master.org`.

[8] Hans-Christiaan Braun. Applying learning-to-lank to human resourcing's job-candidate matching problem: a case study. Master's thesis, Radboud University, 2017. Downloaded from `https://theses.ubn.ru.nl`.

[9] Daniel Foster Agnes van Belle, Eike Dehling. Improving candidate to job matching with machine learning, 2018. Textkernel, Amsterdam, the Netherlands,`http://www.textkernel.com`.

[10] Evanthia Faliagka, Kostas Ramantas, Athanasios Tsakalidis, and Giannis Tzimas. Application of machine learning algorithms to an online recruitment system. In *Proc. International Conference on Internet and Web Applications and Services*. Citeseer, 2012.

[11] Jochen Malinowski, Tobias Keim, Oliver Wendt, and Tim Weitzel. Matching people and jobs: A bilateral recommendation approach. In *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, volume 6, pages 137c–137c. IEEE, 2006.

[12] Tim Zimmermann, Leo Kotschenreuther, and Karsten Schmidt. Data-driven hr-r\'esum\'e analysis based on natural language processing and machine learning. *arXiv preprint arXiv:1606.05611*, 2016.

[13] Koen Rodenburg. Using information extraction and evolutionary algorithms to improve matchmaking on the labor market. Master's thesis, Utrecht University, 2014. Downloaded from `https://dspace.library.uu.nl/handle/1874/298123`.

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

[15] Olivier Chapelle and Yi Chang. Yahoo! learning to rank challenge overview. In *Proceedings of the Learning to Rank Challenge*, pages 1–24, 2011.

[16] Niek Tax, Sander Bockting, and Djoerd Hiemstra. A cross-benchmark comparison of 87 learning to rank methods. *Information processing & management*, 51(6):757–772, 2015.

[17] Fatih Cakir, Kun He, Xide Xia, Brian Kulis, and Stan Sclaroff. Deep metric learning to rank. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1861–1870, 2019.

[18] William S Cooper, Fredric C Gey, and Daniel P Dabney. Probabilistic retrieval based on staged logistic regression. In *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 198–210. ACM, 1992.

[19] David Cossock and Tong Zhang. Subset ranking using regression. In *International Conference on Computational Learning Theory*, pages 605–619. Springer, 2006.

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

[21] Suzan Verberne, Hans van Halteren, Daphne Theijssen, Stephan Raaijmakers, and Lou Boves. Learning to rank for why-question answering. *Information Retrieval*, 14(2):107–132, 2011.

[22] Luis Matos Pombo. Landing on the right job: a machine learning approach to match candidates with jobs applying semantic embeddings. Master's thesis, Universidade Nova de Lisboa, 2019. Downloaded from `https://run.unl.pt/handle/10362/60405`.

[23] Melanie Tosik, Carsten Lygteskov Hansen, Gerard Goossen, and Mihai Rotaru. Word embeddings vs word types for sequence labeling: the curious case of cv parsing. In *Proceedings of the 1st Workshop on Vector Space Modeling for Natural Language Processing*, pages 123–128, 2015.

[24] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[25] Marco Baroni, Georgiana Dinu, and Germán Kruszewski. Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 238–247, 2014.

[26] John Lafferty, Andrew McCallum, and Fernando CN Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.

[27] Charles Sutton, Andrew McCallum, et al. An introduction to conditional random fields. *Foundations and Trends® in Machine Learning*, 4(4):267–373, 2012.

[28] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. A simple but tough-to-beat baseline for sentence embeddings. 2016.

[29] Craig W Schmidt. Improving a tf-idf weighted document vector embedding. *arXiv preprint arXiv:1902.09875*, 2019.

[30] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. Introduction to information retrieval. *Natural Language Engineering*, 16(1):100–103, 2010.

[31] Kalervo Järvelin and Jaana Kekäläinen. Ir evaluation methods for retrieving highly relevant documents. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 41–48. ACM, 2000.

[32] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, Wei Chen, and Tie-Yan Liu. A theoretical analysis of ndcg ranking measures. In *Proceedings of the 26th annual conference on learning theory (COLT 2013)*, volume 8, page 6, 2013.

[33] Christopher JC Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11(23-581):81, 2010.

[34] Christopher J Burges, Robert Ragno, and Quoc V Le. Learning to rank with nonsmooth cost functions. In *Advances in neural information processing systems*, pages 193–200, 2007.

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

[36] Faezeh Ensan, Ebrahim Bagheri, Amal Zouaq, and Alexandre Kouznetsov. An empirical study of embedding features in learning to rank. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 2059–2062. ACM, 2017.

[37] Aliaksei Severyn and Alessandro Moschitti. Learning to rank short text pairs with convolutional deep neural networks. In *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*, pages 373–382. ACM, 2015.

[38] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[39] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.

[40] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[41] Jan Wijffels and Naoaki Okazaki. crfsuite: Conditional random fields for labelling sequential data in natural language processing based on crfsuite: a fast implementation of conditional random fields (crfs), 2007-2018. R package version 0.1.

[42] Edward Loper and Steven Bird. Nltk: the natural language toolkit. *arXiv preprint cs/0205028*, 2002.

[43] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[44] Sentence embeddings. fast, please! https://towardsdatascience.com/fse-2b1ffa791cf9. Accessed: 2019-06-26.

[45] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.