Radboud University

Master's Thesis,
Extended Research Project

Applying Learning-to-Rank to Human Resourcing's Job-Candidate Matching Problem: A Case Study.

Author:
Hans-Christiaan Braun
s4132416

External Daily Supervisor:
Koen Rodenburg,
NCIM-Groep
(Leidschendam)

External Supervisor:
Harold Kasperink,
NCIM-Groep
(Leidschendam)

Internal Supervisor:
Jason Farquhar,
Donders Institute for Brain,
Cognition and Behaviour
(Nijmegen)

June 23, 2017

Abstract

A challenge that every company or organization will continue to face regularly is the task of recruiting people that will perfectly fit their vacant jobs. This is especially vital for companies in the Human Resourcing industry, like employment agencies and secondment companies, whose livelihood depend on selecting the right candidates.

This selection process is currently performed by Human Resourcing professionals. The first step commonly consists of manually searching through the available applicants, eventually producing a list of suitable candidates that get invited to the next phase of the application process.

This already labor intensive process has been made even more challenging with the advent of online job recruitment. This made finding and posting vacancies simpler, but also increased the number of applicants.

However, the digitization of recruitment can alleviate this information overload by, for example, providing the Human Resourcer with an ordering of the applicants, based on their estimated suitability for a given position.

For the NCIM-Group, a secondment company near Leidschendam, Learning-to-Rank, a type of Machine Learning, was applied to automatically induce a way to do just this: to order a list of candidates based on a given job offer. The ranking model was learned from the company's historical placement data.

There are many ways of solving the learning-to-rank problem. Three state-of-the-art models, each one exemplifying one of the three common approaches, the point- pair- and list-wise approach, have been implemented to identify which one is best suited for this problem. Specifically, a Gradient Boosted Regression Trees (GBRT) model (a point-wise method), a LambdaMART model (a pair-wise method) and a SmoothRank model (a list-wise method) were applied. Their performance, plus a baseline Best-Single-Feature model, were compared with the existing Evolutionary Algorithm model on two common rank-based evaluation measures: Mean Average Precision (MAP) and mean Normalized Discounted Cumulative Gain (NDCG).

All three methods improved the performance significantly compared to the existing algorithm, with an increase in MAP score up to 59.7% (GBRT-model vs. Evo.-model, $p = 0.0001$).

Additional results indicate that adding Manifold Regularization, a semi-supervised technique, to SmoothRank may improve its performance slightly by 6% (although not statistically significant).

Acknowledgments

First of all, I like to thank Koen Rodenburg, my supervisor at NCIM, for his regular feedback, support and code reviews. Furthermore, without his work on the original CVMatcher system I would not have the ability to focus my research on modifying and extending it.

I also like to thank Harold Kasperink, the COO/CTO of the NCIM-Group, for giving me the time, support and freedom to work and do research on the CVMatcher system.

I am grateful for the support I got from Jason Farquhar, my supervisor at the Radboud University. His recommendations on using randomization tests for statistical testing, comparing the performance of the models with the performance of a baseline model and looking into semi-supervised techniques were very helpful in shaping my research.

A big shout-out to my fellow interns and colleagues at NCIM: Rutger, Rick, Nick, Tom, Janneke, Branda, Thien, Jeroen and Bram, just to name a few. You made my time at NCIM very enjoyable.

Last, but not least, I want to thank my family and friends for their general support and advice in times of need.

Contents

Chapter 1

Introduction

Until the year that the entire workforce has been replaced by robots or software, which is not foreseeable in the near future[1], a challenge that every company or organization will continue to face regularly is the task of recruiting people that will perfectly fit their vacant jobs. This is a task with many nuances, since a good fit depends on many different factors. These include 'hard' factors, like a match between the asked-for and supplied work experience, education and skills, but also many 'soft' factors like a match between the personality of the company and candidate employee.

The advent of online job recruitment in the '90's made finding and posting vacancies simpler[2]. This increased the amount of digital applicants, making it difficult to separate the possibly matching candidates from the unsuitable ones. Especially for large firms, staffing agencies and secondment companies. However, the digitalization of recruitment can provide many chances for automating this process as well. Since the vacancies and resumes of candidates are already stored in a digital format, either on websites or in documents, information can be readily extracted and matched using the right system.

The CVMatcher system [3] tries to provide this functionality. It has been made as a graduation project for NCIM, a secondment company based in Leidschendam, in 2015. Its main function is, when given a job description, to generate a list of recommended candidates, ordered on their suitability for the given job. The information about the candidates are gathered by parsing their resumes using Natural Language Processing methods, while the job descriptions can be manually entered and are also crawled from major job portals on the Internet.

This graduation project is about improving the list generation algorithm of the CVMatcher system by applying machine learning to automatically induce a way to match jobs with possible candidates from historic placement data of NCIM. Specifically, learning-to-rank will be applied, an application of machine learning. Learning to rank is naturally suited for the problem at hand, since it tries to learn a way to order or rank items or documents (in this case candidates) based on a given query or condition (job descriptions, in this case), by means of existing ranked lists of items.

This thesis will describe how three different supervised learning-to-rank techniques, each exemplifying a different approach, were integrated, implemented and tested.

1.1 Problem Description

As a medium sized secondment company, NCIM receives multiple requests each month from a variety of different sources, including existing clients, new clients, job websites or other secondment agencies. It is quite a challenge to match these with the pool of personnel from NCIM herself, from other secondment companies and persons who are self-employed.

The main challenge lies in the fact that it is labor intensive to manually sift through the resumes of potential candidates, one by one, in search for a list to invite to the next stage of the application process. Over the last few years, the resourcing department collected over 6500 resumes, making it impossible to search through them all in a reasonable amount of time. This, in turn, can lead to job requests that are 'lost': when no selected candidate gets placed, or when no suitable candidate could be found at all.

The CVMatcher system can give relief by A) parsing the resumes and providing a means to search through the parsed data and B) providing an automatic way to generate lists of recommended candidates for a given vacancy.

However, the current CVMatcher system can be improved. It is not used by the resourcing department yet, because of its still limited maturity. Of all the components, the matching part lies at the heart and is, in my opinion, also the most suited for improvement.

The matching part of the system currently uses an Evolutionary Algorithm, a rather unorthodox approach to the ranking and matching problem. Also, the algorithm was tested on providing a list of suitable jobs for a given candidate, instead of the other way around. This gives us the additional opportunity to test the algorithm on the job-candidate matching problem, a viewpoint that corresponds more to the way of working of the resourcing department.

1.2 Research Questions & Main Approach

A better method to learn to rank a list of items from existing ranked lists may be found in the appropriately named domain of learning-to-rank, a domain closely related to machine learning.

For this thesis, four learning-to-rank algorithms were implemented and compared with each other and the existing Evolutionary Algorithm. These included a Gradient Boosted Regression Trees (GBRT)[4], a LambdaMART[5] and SmoothRank[6] model. The fourth algorithm was the Best Single Feature (BSF) model, a simple baseline model, to assess the quality of the data.

Each model was trained and tested on the historical placement data of NCIM: which persons were selected by Human Resourcing for a job in the past, and who of these selected candidates got actually placed? The performance of these algorithms were measured on two metrics for ranked lists, widely used in the domain of Information Retrieval: Mean Average Precision (MAP) and mean Normalized Discounted Cumulative Gain (NDCG).

During the course of conducting the research, it became apparent that the amount of labeled training data was lacking. However, unlabeled data points could easily be generated by combining jobs with other, not selected candidates. This begged the question if semi-supervised techniques, learning from labeled as well as unlabeled data, could be used to improve the algorithms. To this end, a method called manifold regularization was added to the SmoothRank algorithm. On top of this another, new kind of manifold regularization, specifically adapted to the ranking problem and the SmoothRank algorithm, was developed and tested.

In the end, the goal was to provide NCIM with a substantiated claim on which of the implemented algorithms, and additions, will most likely perform the best when the system is up-and-running and subsequently used by the department of Human Resourcing.

To provide this substantiation, these research questions were devised, to be answered by the following chapters of this thesis:

1. Of the four implemented (learning-to-rank) techniques: Single Best Feature, Gradient Boosted Regression Trees, LambdaMART and SmoothRank, which one perform better than the current Evolutionary Algorithm, when trained and tested on historical placement data of candidates on job offers..

   (a) .. when measured on Mean Average Precision (MAP)?

   (b) .. when measured on mean Normalized Discounted Cumulative Gain (NDCG)?

2. Does adding a semi-supervised technique, in the form of manifold regularization, help the SmoothRank algorithm perform better, in a sense that its MAP and mean NDCG scores improve when trained and tested on the historical placement data?

   (a) Which of the two added types of manifold regularization techniques adds the most performance, if any?

Why these specifics were chosen, the theory behind learning-to-rank, each algorithm, evaluation measure and statistical test and their results will be explained in the remaining chapters of this thesis.

1.3 Structure of the Thesis

To put the system into context, Chapter 2 will discuss the relevant research done on automatic matching of jobs and candidates in the labor market.

In Chapter 3, knowledge about learning-to-rank is put forth, and the algorithms used are described, which will help in understanding the workings of the implemented algorithms and machine learning pipeline.

The subsequent chapter, Chapter 4, will discuss the current system, since the learning-to-rank component will directly build upon the information extracted by the Parser and Crawler parts of the system.

Chapter 5 will describe the research, which includes a description of the data and how it was gathered and how from this data features were generated. It also describes the results in detail, after which Chapter 6 and 7 discusses and concludes the research respectively.

Chapter 2

Literature Review

In the last decade, an increasing amount of research has been done on ways to counteract the information overload provided by e-recruitment by coming up with appropriate means of searching and filtering the vacancies on the side of the job seekers, and filtering and searching job applications on the side of the companies.

This chapter will give an overview on this research. The focus of this review will lie on studies that try to learn to filter candidates and jobs through data, since this is the approach that this thesis is primarily concerned with.

The research can roughly be divided into two approaches. Many articles treat the job-candidate matching problem as a recommendation and information *filtering* problem that can be solved using a Recommender System (Section 2.1). Others treat it, like my approach, as a ranking and information *retrieval* problem that can be (partly) solved using learning-to-rank techniques (Section 2.2).

2.1 The Matching Problem as a Recommendation Problem

Recommender Systems (RS) are systems that are primarily used for recommending items to users in the setting of online stores. There is a considerate amount of research that transplants this approach to the job market, in which the goal is to either recommend vacancies to job seekers, or to recommend possible candidates to companies.

2.1.1 Recommender Systems

The main goal of a Recommender System is to predict the value of an item to a user. In our case, the user would in fact be a job offer, by proxy of a human resourcer, and an item would be a candidate employee.

There are two main ways that Recommender Systems try to reach this goal: Collaborative Filtering (CF) and Content-Based Filtering (CBF).

The main intuition behind Collaborative Filtering is that users who rate items in a similar way as you also have a similar taste in items. The system then proceeds to recommend the items that these similar users have rated highly, but you did not rate or have bought yet. A big advantage of CF is that it does not need to have an underlying model of the users or items, since it only compares the ratings among users. This is also its main disadvantage, since the algorithm does not exploit the structure and characteristics of the items and users.

Content-Based Filtering does in fact work with a model. It keeps track of the characteristics of the items a user liked in the past and recommends items that have similar characteristics. ”Liked” can be defined in many ways, like the items a user has bought or rated highly in the past. CBF does have a few disadvantages. One pitfall is that it has the tendency to capture the user in a ”filter bubble” by recommending similar items, whereas the user actually needs complementary ones. A user who bought forks from a cutlery web shop, for example, would likely be recommended additional forks, whereas he actually needs an accompanying knife.

To get the best of both worlds, efforts have been made to combine both CF and CBF techniques into hybrid models. There are a variety of different ways that this can be done, from simply combining the predictions of a CF and CBF model by taking a weighted sum of their outputs, to adding the output of one model as an input to the other. Although a few years old, Burke has made an excellent review of the standard hybrid techniques and their individual parts[7].

One problem that Recommender Systems have in general is called the ”cold start” problem: when a user uses the system for the first time, he or she does not have an interaction history from which his or her values of the unseen items can be induced. This is especially problematic for model-free methods like CF. This makes them unsuitable for recommending candidates to vacancies, because vacancies are naturally short-lived: when they are filled, the need for more candidates disappears, making their interaction histories small. Therefore, many Recommender Systems that have been made for the domain of HR focus on recommending job offers to job seekers. Primarily in the setting of job portals.

2.1.2 Literature

Hong et al. implemented a Recommender System that clusters candidates in groups, depending on their eagerness in finding a job [8]. ”Proactive” users have clear image of the work they want to do and pro-actively search for a job, ”Passive” users have only a vague idea and therefore retain a more passive attitude and ”Moderate” users are neither particularly active nor passive.

For proactive users, the system uses a Content-Based Recommender System approach. For the passive users, the system uses a Collaborative Filtering approach. For moderate users a hybrid approach was used.

Hong et al. were not the only ones that used a Recommender System to in e-recruitment. Al-Otaibi and Ykhlef wrote a survey [9] of ten other systems.

Most of them used a hybrid approach, combining two or more methods like Collaborative Filtering, Content-based filtering or a knowledge-based approach.

2.2 The Matching Problem as a Ranking Problem

Another way to look at the job-candidate matching challenge is our approach: by defining it as a learning-to-rank problem.

Faliagka et al. made a system [10] that broadly works in the same manner as the CVMatcher system.

On the one side, it provides the candidates with two ways to fill in their details: by means of a form or by uploading their LinkedIn profile. On top of this information, the system extracts data about the personality of the applicant from their blog, if provided. This personality is captured in one 'extraversion' score by applying the LIWC[11] model to the blog text.

On the other side, the recruiter enters the details of the job using another form.

To match the entered job details with the data about the candidate, the system implemented and compared five different methods: linear regression, two kinds of regression trees and two types of Support Vector Regression Machines. All of them can be filed under the point-wise approach, where the learning-to-rank problem is treated as a traditional machine learning problem: trying to predict the label of a data point defined on a job-candidate pair, indicating the suitability of a candidate for the given job.[1]

TextKernel[2] is an Amsterdam-based company that sells a system analogous to the CVMatcher system. Like the CVMatcher, it provides an interconnected system of software modules for extracting information from resumes (named 'Extract!'), for crawling vacancies from job portals ('Jobfeed') and for matching vacancies with candidates ('Match!'). It also provides modules to monitor and search through candidates and vacancies and to publish jobs on websites, functionality that the current resourcing system of NCIM partly provides.

Textkernel only recently started using learning-to-rank to try to improve their matching algorithm. An ensemble method, consisting of 'bagging' multiple boosted regression trees algorithms [3] provided them, according to their first results, with a 22% increase in ranking effectiveness when compared to their original model, according to the NDCG metric [12].

[1]More information on the point-wise and other approaches can be found in Chapter 3
[2]`www.textkernel.com`
[3]These concepts will be explained in detail in Chapter 3

2.3 Other Approaches to the Matching Problem

On top of the Recommender Systems and Learning-to-Rank approaches, there are plenty of other ways that job-candidate matching can be tackled.

In the EXPERT system [13], for example, jobs and candidates are matched using ontology mapping.

Ontologies are formal ways to capture the entities and their (possible) relationships in a given domain. Ontology mapping tries to find semantic similarities between the entities and relationships of two ontologies.

The EXPERT system uses two ontologies: one for jobs and one for candidates. The job ontology can capture, for example, that a job requires knowledge about the Python programming language. The candidate ontology can capture the fact that a candidate knows Python. By trying to map the "knows" on to the "requires knowledge of" relationship, these can be linked and used to find matches.

Another system is PROSPECT [14]. As the CVMatcher and other systems, it also provides information extraction from resumes. The matching algorithm, however, was not learned but was hand-crafted by the researchers themselves.

Chapter 3

Learning-to-Rank Theory

Before we are going to dive into the methods and results of the current research, it is best to lay a theoretical foundation on learning-to-rank first.[1].

3.1 Learning-to-Rank

Learning-to-Rank can be seen as a generalization of machine learning, in which the goal is not to give a single label or score to a single object, but to rank a list of objects, based on a given query or condition.

The ranking problem is common in Information Retrieval, for example in search engines like Yahoo, Google and Bing, where the goal is to sort the found documents or web pages, based on their relevance for the query the user has given. When searching for "grumpy cat" on Google Search, for example, the user would most likely want web pages of the quintessential animal at the top of the results, whilst excavators of the "CAT" brand should have a lower place in the list.

The terminology used in the literature on Learning-to-Rank is primarily focused on Information Retrieval. The condition on which a list is ranked is therefore commonly called the query, whereas the items that are ranked are called documents. However, Learning-to-rank can be applied to many other fields like Recommender Systems[15], in which the goal is to rank the list of recommendations the user might be interested in.

In our case the jobs can be seen as the 'query' and the candidate as the 'document'. In a sense, we want to query the database of candidates on a job description, and want to have a list of suitable candidates to be returned.

[1]It is assumed that you, the reader, have a good foundation on classic machine learning theory. If this is not the case, I advise you to read Appendix B

Although the basis of learning-to-rank lies firmly rooted in classical machine learning, it is different in a few regards. In essence, learning-to-rank makes two key assumptions that are different than the assumptions made in classical machine learning:

- The relevance of a document or object depends for a large part on the given query or condition. For example, a resume of a Java developer is less relevant for a C# position than for a Java software engineer vacancy.
- A document's score or label depends on its rank in the list. Users generally give more attention to documents high up in the list. This attention rapidly diminishes with the document's rank[2]. The idea is that emphasis should be laid on giving relevant documents a place at the top of the list, and irrelevant documents a place in the lower parts.

The first assumption is generally approached by adapting the feature vector representation. This adaptation is described in the next section.

The second assumption is generally approached by choosing the right objective function, based on common learning-to-rank evaluation measures, or by choosing the right optimization method, or both. Common evaluation functions are explained in Section 3.3

Since many of these evaluation measures have some kinks that make them ill-suited for direct optimization, they are generally adapted, or maximized using robust methods that can deal with their irregularities. Why the evaluation measure have these kinks is explained in Section 3.4.

3.2 Feature Vector Representations in Learning-to-Rank

Feature vector representations in learning-to-rank are not that different than in classical machine learning. The main difference is the fact that a majority of the methods use vectors defined on query-document pairs, instead of individual document objects. The labels tied to these pairs indicate the relevance of the document to the respective query. This labels can be binary, e.g. "relevant" or "irrelevant", or can be a score giving a more fine-grained relevance judgment, e.g. a score between 1 (not relevant) and 4 (highly relevant).

This way of representing a data point has the advantage that features can be defined on the relationship between the query and the document. A simple example of such a feature could be the amount of overlapping terms between the query and the document.

[2]There is a joke that the best place to hide a dead body is on page 2 of Google's Search results. There is definitely a kernel of truth in that statement.

3.3 Evaluation Measures for Learning-to-Rank

When we are faced with a ranked list of items, even when the model that made and ordered the list has been hand-crafted, we would like to know if the list is any good. Just like we would like to capture the error of a prediction of a single data point in one number, we would like to capture some sort of error or loss of a predicted ranking of a list of items, in comparison with the "true" ranking, in a single score.

What encompasses a "good" ranked list? There are a few notions that we could like a ranked list to have:

A ranked list should..

- .. have few non-relevant documents.
- .. contain all the documents that are relevant to the query.
- .. emphasize having relevant documents at the top of the list.
- .. provide the correct (pair-wise) order between the documents.

Many measures have been developed to capture one or more of these notions. Examples include precision and recall, Average Precision (AP), Normalized Discounted Cumulative Gain (NDCG), Kendall's RCC and Spearman's RCC.

The next section will describe NDCG in more detail, since it is one of the more popular evaluation measures[16] and it is used in two of the, for this thesis, implemented algorithms.

3.3.1 Normalized Discounted Cumulative Gain (NDCG)

This mouthful can best be explained by looking at the Cumulative Gain measure first, on which this metric is based, and iteratively build upon it to get to its final form.

Cumulative Gain (CG) at rank r is calculated by taking all the items of a list until rank r and adding their relevance scores y together. In mathematical notation:

$$\mathrm{CG}_R(q) = \sum_{\hat{r}=1}^{\hat{r}<=R} y_{q\hat{r}}$$

Where $y_{q\hat{r}}$ is the relevance score of the document at the predicted rank $\hat{r}$.

The intuitive notion is that we want to have a ranked list of items in which many highly relevant documents occur. However, it has its drawbacks.

One drawback is the fact that CG does not look at the place of the items in the list. We would like to have a list where the relevant documents are at the top. A list where all the relevant documents are at the end, and all the irrelevant documents are at the top, has exactly the same CG score as the same list in reverse order.

This drawback can be counteracted by discounting the documents by their rank. This means that the scores of documents at the top of the list have more weight than the ones at the bottom. This version of CG is called the Discounted Cumulative Gain or DCG. The discounting is implemented by dividing the relevance score by the logarithm of the document's rank.

$$\mathrm{DCG}_R(q) = y_{q\hat{1}} + \sum_{\hat{r}=2}^{R} \frac{y_{q\hat{r}}}{\log_2(\hat{r})}$$

Another drawback comes from the fact that the total amount of relevant documents in the entire collection is different for each query.

A ranked list of twenty items that contains all two relevant documents from the entire collection should be better than a ranked list containing five of all twenty relevant documents. DCG, however, only looks at the relevant documents in the list. In a sense, it is a measure of precision. This means that the list with five relevant documents gets a higher score.

One way to take this into account is to normalize the DCG based on the ideal DCG score. The ideal DCG score is computed by sorting all the documents in the collection based on their true relevance score and computing its DCG.

$$\mathrm{nDCG}_R(q) = \frac{\mathrm{DCG}_R(q)}{\mathrm{iDCG}_R(q)}$$

3.4 The Challenge in Learning Ranking Models

The obvious approach to learn a ranking model would be to take the derivative of one of the evaluation measures and use gradient descent, or another optimization method that uses derivatives, to find the model that maximizes the function. There is one large challenge however in the way that ranked lists are represented and the fact that these objective functions (like NDCG) rely primarily on the rank of the documents.

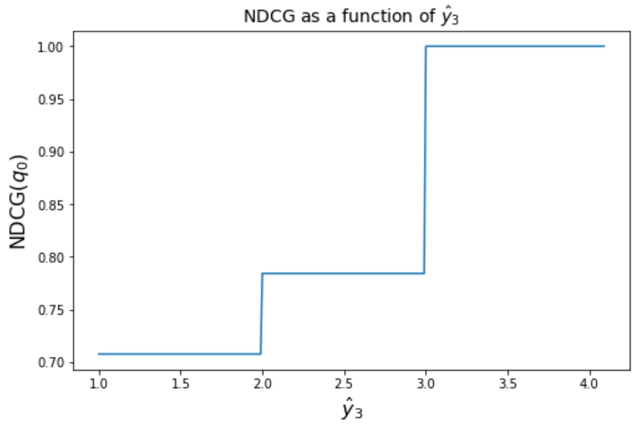
Algorithm 3.1 gives, in pseudo code, the common learning-to-rank approach. It is similar in the way classical machine learning models are learned, except for the fact that this algorithm works on lists, instead of individual (x_i, y_i) pairs.

Algorithm 3.1 General approach of learning a ranking model.

```
Require: A list of ranked lists $Q$ of $(x_{qi}, y_{qi})$ pairs.
1: Initialize the ranking model $f(x_{qi})$.
2: for a (fixed) number of iterations do
3:     for each ranked list $q$ in $Q$ do
4:         Rank all $x_{qi}$'s based on the relevance scores predicted by $f(x_{qi})$.
5:         Evaluate the objective function, based on the predicted ranking.
6:         Update $f(x_{qi})$, based on the score on the objective function.
7:     end for
8: end for
```

Ranking at the start (NDCG = 0.7077):

rank	doc (d_i)	true relevance (y_i)	predicted score ($\hat{y}_i$)
1.	d_1	1.0	3.0
2.	d_2	1.0	2.0
3.	d_3	4.0	1.0



Ranking at the end (NDCG = 1.0):

rank	doc (d_i)	true relevance (y_i)	predicted score ($\hat{y}_i$)
1.	d_3	4.0	4.0
2.	d_1	1.0	3.0
3.	d_2	1.0	2.0

Figure 3.1: This figure exemplifies the behavior of NDCG based on changes in the predicted scores of one document (d_3, in this example) in a list of three. Note the plateaus and steep edges. Also note the higher increase in NDCG score after d_3 overtakes d_1, courtesy of the emphasis NDCG puts on relevant documents at the top of the ranked list.

The challenge lies in the fact that changes in the predicted relevance scores do not necessarily change the ranking of the documents. This means that changing one score slightly does not provide the algorithm with any information on the direction of the maximum or minimum in many cases. Only when a document's score overtakes, or gets behind, another document's score, the ordering changes and subsequently the score on the objective measure.

Figure 3.1 illustrates this type of behavior for the NDCG measure, although it is a challenge for all evaluation measures based on rank.

The main question of learning-to-rank is as follows: how can we solve this issue, so we can use NDCG, MAP or any other rank-based measure either way? Roughly three different approaches have developed over time, which are explained in the next section.

3.5 Three Approaches to Learning-to-Rank

Learning-to-rank is a very hot topic. Its natural application to Information Retrieval has sparked the interest of big players in the field like Microsoft, Google and Yahoo!, who in turn invest heavily in learning-to-rank research. This is exemplified by the fact that many learning-to-rank algorithms have been developed in recent years. Tax et al., for example, have identified a total of 84 different algorithms and variations in 2015 [16]. All of these methods, however, can roughly be divided into three common approaches: the point-wise, pair-wise and list-wise approach respectively.

3.5.1 The Point-wise Approach

One approach is to completely discard the objective measures defined on lists, and use a measure defined on individual documents and their relevance scores instead, like the common Sum of Squared Errors[3]. The intuition is that predicting the relevance scores correctly would lead to a sufficient ranking, since the query-document pairs are ordered on their predicted score either way. This is called the **point-wise approach**. Almost all classical (supervised) machine learning methods can be shared under this approach, although there are methods that adapt the used algorithms to be better suited to solve the ranking problem, like Mcrank [17].

For the point-wise approach in this research, a Gradient Boosted Regression Tree algorithm [4] was chosen. This method was chosen because it is the basis of the LambdaMART algorithm, the implemented pair-wise method, and it would be interesting to see how their performance compares. GBRT's are also one of the better performing classical machine learning algorithms [18] in general (disregarding Artificial Neural Networks).

3.5.2 Gradient Boosted Regression Trees

Gradient boosting is a so-called ensemble method. Instead of learning and relying on one model, ensemble methods learn a whole group of sub-models. Predicting the label of a data point happens by feeding each sub-model the feature vector, and by combining their individual outputs in some way.

The intuition behind this is that "more heads are better than one": by combining the output of models with different "viewpoints" on the data, the quality of the total output is improved[4].

Ensemble methods differ in three main ways: on the type of sub-model that it uses, on how the output of the sub-models are combined to provide one

[3]See also Appendix B

[4]This is perfectly summarized as a quote by C.S. Lewis: *"Two heads are better than one, not because either is infallible, but because they are unlikely to go wrong in the same direction."*

overall prediction of a data point, and how the ensemble tries to ensure that each sub-model captures the training data in a slightly different way.

The Type of Model Used

Most ensemble methods do not strictly enforce one type of model, but can combine any type of classification or regression model. However, models that can be trained fast are generally preferred. In some ensemble methods, models are also kept deliberately small and 'weak'. This promotes diversity between models and reduces the time needed to learn a single model.

Many methods therefore work with decision trees. Decision trees can be learned quickly by a variety of different algorithms, for example CART[19] and C4.5[20], and their complexity can be easily kept in tome by, for example, setting a maximum to the amount of generated nodes.

Gradient Boosting typically works with regression trees: a type of decision trees for solving regression problems.

How the Outputs are Combined

There are a couple of approaches to combine the output of the sub-models into one prediction.

One of the most straightforward combination methods for regression models is to take the average of the individual predictions (Equation 3.1), while classification methods can take the majority vote.

$$f(\vec{x}_i) = \frac{1}{K}\sum_k m_k(\vec{x}_i) \tag{3.1}$$

Many ensemble methods, however, take a weighted average or vote. In this combination method, each sub-model gets a weight, depending on their performance during training. In AdaBoost[21], for example, a vote of a sub-model with a high error rate during training counts less as a vote of a sub-model with a higher performance.

$$f(\vec{x}_i) = \sum_k \alpha_k m_k(\vec{x}_i) \tag{3.2}$$

How the Different "Viewpoints" of the Models are Enforced

The strength of an ensemble lies in the differences in which the sub-models model the data. Giving them all the same data, however, would lead to exactly the same models, when using a purely deterministic base model type. To ensure that every model differs, many ensemble methods vary the training data that each model is given in some way or form.[5]

[5]In humans, differing views on a subject is ensured by the fact that everyone's viewpoint is most likely based on their previous experiences. Since a machine learning model is generally learned from scratch, they do not have this luxury (or hindrance, depending on your viewpoint).

There are three main methods of training an ensemble: Bagging, Boosting and Gradient Boosting.

Bagging In Bagging, each model learns from a random subset of the training data. Every data point has exactly the same chance to be selected and can even be selected multiple times. The outputs of the sub-models are generally combined by taking the majority vote (Equation 3.1).

Boosting In Boosting, each model is trained on the entire training set. However, the ensemble is built in an iterative manner, where the problem a sub-model faces depends on the performance of the previous iteration of the ensemble.

In AdaBoost[21], for example, data points that are miss-classified get more weight, while points that are correctly classified get less weight. This leads to each sub-model focusing on the data points that the current version of the ensemble gets wrong.

Each added model effectively tries to correct the output of the ensemble's previous iteration. In a sense, Boosting performs gradient descent in function or model space: instead of updating the weights of a single model in a direction that minimizes the loss function, Boosting adds a new model that tries to do the same.

Gradient Boosting Gradient Boosting takes this notion of functional gradient descent a step further.

An ensemble f can be assumed, at each iteration t, to model the training data incompletely, and thus predict, for some data points x_i, the wrong label. In other words, there is an error e between the predicted and actual label y_i.

$$y_i = f_t(x_i) + e_t(x_i)$$
$$e_t(x_i) = y_i - f_t(x_i)$$

These errors or residuals can be seen as the gradients that still need to be modeled by the next sub-models in the ensemble. Gradient Boosting exploits this viewpoint by letting the next sub-model m predict exactly these residuals e, in the form of the score on a specified loss-function between the predicted score $\hat{y}_i$, as predicted by the previous version of the ensemble $f_{t-1}(x_i)$, and the actual score y_i:

$$y_i^t = L(f_{t-1}(x_i), y_i) \tag{3.3}$$

Combining the judgments of all sub-models on a data point is done by summing their predictions.

$$f_T(x_i) = \sum_{t \leq T} \alpha_t \cdot m_t(x_i)$$

Like in AdaBoost, α is a weight added to each model to affect their influence on the final prediction. In Gradient Boosting, this can be seen as analogous to the learning rate in Gradient Descent. It can be fixed, or a more advanced approach can be taken to determine it over the course of learning the model.

The first sub-model can be held relatively simple, for example by always predicting the mean or majority label of the training set.

3.5.3 The Pair-wise Approach

Another way to tackle the learning-to-rank problem is to define a loss function on pairs of documents instead. The goal when using this approach is to minimize the number of discordant pairs, as in Kendall's RCC. This is commonly called the **pair-wise approach**.

For the pair-wise approach the LambdaMART[5] algorithm was integrated. The LambdaMART algorithm can be seen as the successor to the RankNet[22] and subsequent LambdaRank[23] algorithms, as described in the paper "From RankNet to LambdaRank to LambdaMART: An Overview" [5]. All three algorithms have been highly influential in the learning-to-rank research with a combined total of 2208 citations[6]. On top of this, an ensemble of LambdaMART classifiers won the 2011 Yahoo! Learning-to-Rank Challenge [24]. Besides, a "bagged version of a boosted regression trees algorithm"[12] provided TextKernel (see Chapter 2) with the best results. It is likely that the "boosted regression trees algorithm" used corresponds with the LambdaMART algorithm.

3.5.4 LambdaMART

The LambdaMART algorithm uses a pair-wise approach based on the GBRT algorithm. In fact, MART stands for Multiple Added Regression Trees, a synonym for GBRT. It uses a loss function based on the NDCG measure, but this loss function can be easily changed to incorporate any other list-based metric like MAP. As can be seen in Equation 3.3, the GBRT algorithm takes a loss-function that is defined on a single data point.

Whereas the loss-function used in the GBRT-algorithm compares the actual relevance score with the predicted score, LambdaMART's loss-function on a document is defined as the sum of all pair-wise gradients, or 'lambdas' between it and all other documents.

The intuitive notion behind LambdaMART is that this pair-wise gradient can be modeled as simply the absolute change in NDCG, MAP or other list-based evaluation measure when the two are swapped (and the order of the rest of the documents remains unchanged).

To make this gradient differentiable it is multiplied with the sigmoid function, eventually leading to this formula[7]:

[6] According to Google Scholar. It may very well be more.

[7] The sigma (σ) term in this formula is a hyper-parameter. It controls the 'smoothness' of the derivative. Its behavior is explained in the next section about SmoothRank 5.5.2. e is the

$$\lambda_{qij} = \frac{-\sigma}{1 + e^{-\sigma \cdot (\hat{y}_{qi} - \hat{y}_{qj})}} \cdot |\Delta\, \text{NDCG}(q)| \tag{3.4}$$

These lambdas or gradients are summed to tie them to the individual query-document pairs in the ranked lists. The sign of each lambda is chosen such that a document receives a push upwards from docs with a lower true relevance score and the other way around. This finally leads us to the following equation:

$$\lambda_{qi} = \sum_j \begin{cases} \lambda_{qij} & \text{if } y_{qj} < y_{qi} \\ -\lambda_{qij} & \text{if } y_{qi} > y_{qj} \\ 0 & \text{if } y_{qi} = y_{qj} \end{cases} \tag{3.5}$$

These summed lambdas (and respective single lambdas) are computed after each added sub-model, and are taken as the new scores that have to be predicted by the next one. As in standard gradient boosting, this is repeated until the pre-specified number of sub-models has been trained.

3.5.5 The List-wise Approach

The last, but not least, set of learning-to-rank methods is called the **list-wise approach**. This approach tries to optimize an objective function defined on a list of documents, like the evaluation measures described in Section 3.3.

One way is to optimize measures like NDCG and MAP by either approximating the measure, for example by smoothing it, like in the SoftRank[25] and SmoothRank[6] algorithms, or by optimizing an objective function that provides a bound to one these measures instead, like in SVM$^{\Delta}_{map}$[26].

To complete the trinity of implemented learning-to-rank methods, SmoothRank[6] was eventually chosen to exemplify the list-wise method. One of the main reasons for this choice was the fact that it has been shown to be one of the better learning-to-rank algorithms in the meta-study by Tax et al.[16]. In this study 87 learning-to-rank methods were compared, based on their results on common benchmark datasets like LETOR[27].

3.5.6 SmoothRank

SmoothRank tries to maximize a modified version of the NDCG measure by means of the gradient descent method. To make the derivative easier to work with, the authors of SmoothRank use a slightly different formulation of (N)DCG, consisting of the sum of a Gain function $G(y_{qi}) = 2^{y_{qi}} - 1$ and a Discount function $D(r_{qi}) = 1/\log_2(1 + r_{qi})$.

$$\text{NDCG}_q = \sum_{i=1} G(y_{qi}) \cdot \text{ND}(r_{qi}) \tag{3.6}$$

Euler constant.

Where $\mathrm{ND}(r_{qi})$ is the Normalized Discount: $D(r_{qi})/\mathrm{iDCG}_q$.

Now the idea behind SmoothRank is that this formulation of NDCG can be rewritten by adding an indicator term to it that outputs 1 whenever the rank of a document corresponds with another number j:

$$\mathrm{NDCG}_q = \sum_{i,j} G(y_{qi}) \cdot \mathrm{ND}(r_{qi}) \cdot 1_{r_{qi}=j} \tag{3.7}$$

This does not seem to add anything to the formula. Indeed, it does not change the behavior of it at all. However, it does mean that we can alter this addition to make this version of NDCG behave a little bit better when trying to optimize it using gradient descent.

This alteration comes in the form of a probabilistic interpretation of the indicator function. Instead of a document having a 100% chance of being at a specific rank, it gets a non-zero probability to have any rank, based on its predicted relevance score (Equation 3.8).

$$h_{qij} = \exp\left(-\frac{(f(\vec{x}_{qi}) - f(\vec{x}_{qj}))^2}{\sigma}\right) \Bigg/ \sum_k \exp\left(-\frac{(f(\vec{x}_{qk}) - f(\vec{x}_{qj}))^2}{\sigma}\right) \tag{3.8}$$

In other words: a document has a higher probability to be at a certain rank when its predicted score is similar to the predicted score of the document currently ranked at that position. Since the predicted score changes whenever the model does, and therefore the probability mass function (PMF) of this new indicator, this effectively 'smooths' out the original NDCG measure. This makes it possible to take its derivative and optimize it using, for example, gradient descent.

The derivative of the smooth indicator variable is this monstrosity:

$$\begin{aligned}\frac{d\,h_{qij}}{d\,f(\vec{x}_{qp})} = \frac{2}{\sigma(\sum_k e_{qkj})^2} \cdot \Bigg[& e_{qpj} \cdot (f(\vec{x}_{qp}) - f(\vec{x}_{qj})) \cdot (e_{qij} - 1_{i=p} \cdot \sum_k e_{qkj}) \\ & + 1_{j=p} \cdot e_{qij} \cdot \left(f(\vec{x}_{qi}) \cdot \sum_k e_{qkj} - \sum_k e_{qkj} \cdot f(\vec{x}_{qk}) \right) \Bigg]\end{aligned} \tag{3.9}$$

Where:

$$e_{qij} = \exp\left(-\frac{(f(\vec{x}_{qi}) - f(\vec{x}_{qj}))^2}{\sigma}\right) \tag{3.10}$$

This leads us to the final derivative in Equation 3.11 when we transplant Formula 3.9 into the new formulation of NDCG:

$$\frac{d\,A_q(\vec{w},\sigma)}{d\,\vec{w}} = \sum_q \sum_{i,j,p} G(y_{qi}) \cdot \mathrm{ND}(r_{qj}) \cdot \frac{d\,h_{qij}}{d\,f(x_{qp})} \cdot \vec{x}_{qp} \tag{3.11}$$

This derivative is the function that SmoothRank tries to maximize, by using gradient descent to find the optimal weights $\vec{w}$ of a weighted linear model.

To nudge gradient descent into finding the optimal model, SmoothRank employs a few additional techniques.

The first technique is to use a simulated annealing procedure to iteratively reduce the value of the sigma parameter during training. The sigma parameter controls the smoothness of the smoothed function. A large value produces a gently sloped function, that, unfortunately, does not look similar to the original function. A small value, on the other hand, contains more peaks and valleys, that in turn make it difficult to find the global maximum, but approaches the form of the original function. By starting with a large sigma, and reducing it every few iterations, the chance of finding the global maximum is increased.

The second technique is to use a simpler machine learning method, linear regression, to try to predict the gain values $G(y_{qi})$ first. Since both linear regression and SmoothRank use the same type of model, the learned weights can be directly used as a starting point for the SmoothRank model.

3.5.7 Best Single Feature: A Baseline Technique

In addition to GBRT, LambdaMART and SmoothRank, another algorithm has been implemented: Best Single Feature, or BSF.

The Best Single Feature algorithm is one of the more simple learning-to-rank methods, if not the simplest. It is implemented as a baseline to compare the performance of the other algorithms with.

BSF learns a model by walking through each feature, ordering all ranked lists in the training set on this feature and measuring the score on a list-based evaluation measure. This is done twice for each feature, once sorting the values in ascending order, and once in descending order. The feature and order that produces the largest mean score on the evaluation measure on the training set is remembered. After training, the model is applied to ranked lists by sorting them on the remembered feature and order.

Chapter 4

Description of the Current System

The last two chapters gave us a good amount of background information about the general theoretical notions about learning-to-rank and about the existing literature on applying machine learning on matching jobs and candidates.

Before we can dive into the methods and results of my research project, however, there is still one more piece of context to address: the current system. Since this project will reuse the subsystems that extract candidate information from their resumes and vacancy information from job portal websites, it is helpful to understand how they work. This will help in designing the addition, since the system can be adapted to the strengths and weaknesses of the current system. Furthermore, it will help in the analysis of the performance of the learning-to-rank addition later on in this thesis.

The system can be divided into five different components, of which an overview can be seen in Figure 4.1. In the first section, the internal representations of a candidate, job and match are described. In the second section, the different components of the system are put forward. For a more in-depth description of the current system it is advised that the reader reads the master's thesis of Rodenburg, who has built the main part of the original system for his own research project [3].

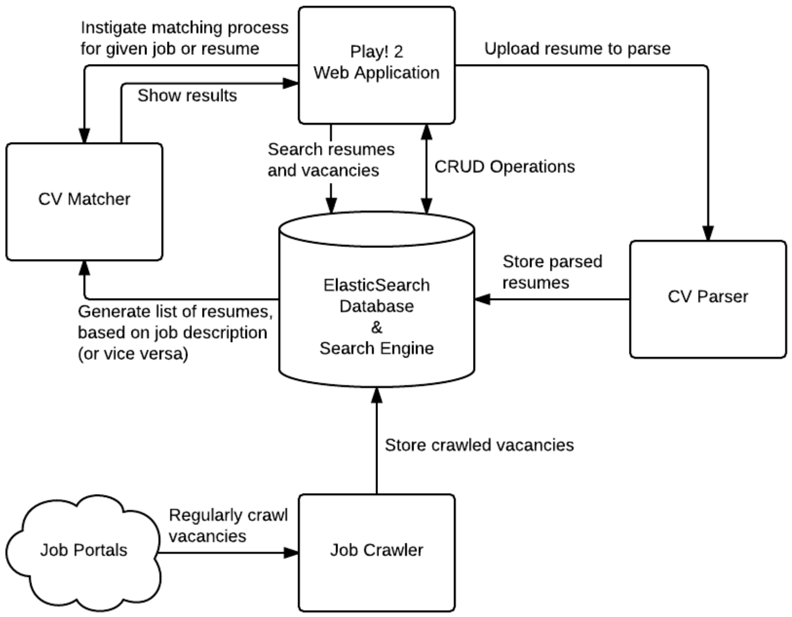

Instigate matching process for given job or resume
Play! 2 Web Application
Upload resume to parse
Show results
Search resumes and vacancies
CRUD Operations
CV Matcher
ElasticSearch Database & Search Engine
Store parsed resumes
CV Parser
Generate list of resumes, based on job description (or vice versa)
Store crawled vacancies
Regularly crawl vacancies
Job Portals
Job Crawler

Figure 4.1: An overview of the architecture of the current system.

4.1 The Candidate, Job and Match Representations

Each component in the system uses the same representation for a candidate, job and match. They do not only consist of data, but also of meta-data like the date at which the representation or object was made and last updated. It is important to know what information is stored and in what form, since our feature vector representation will be based upon it.

The system identifies three different objects: the candidate, representing a human resource; the job, representing a job vacancy; and the match, representing a match between a candidate and a job.

The next subsections will explicate these representations in a tabular format that provides a short description of each property of the candidate, job and match object. The extraction types are further explained in Section 4.2.2.

4.1.1 The Candidate Representation

Data

Name	Extraction Type	Description
`name`	Rule-based	First name and surname of the candidate.
`birthdate`	Rule-based	Birth date of the candidate.
`nationality`	-	Nationality of the candidate. Is not automatically parsed.
`emailaddress`	Rule-based	Candidate's email address.
`location`	Rule-based	The candidate's place of residence.
`role`	Rule-based	A comma-separated list of all the job roles this candidate has fulfilled.
`employmentStatus`	-	Whether the candidate is employed by NCIM ("Intern"), from another company ("Extern") or is self-employed ("ZZP"). Is not automatically parsed.
`education`	Gazetteer	Education history of the candidate. In practice a comma-separated list of automatically recognized educational institutes in the resume.
`programmingLanguages`	Gazetteer	A comma-separated list of all programming languages that have been recognized in the resume.
`software`	Gazetteer	A comma-separated list of all software that has been recognized.
`cvContent`	-	The complete contents of the resume, as a html-formatted string.
`availableFrom`	-	The date after which this candidate becomes available for a (new) job. Is not parsed.

Meta-Data

Name	Extraction Type	Description
`id`	-	Unique identifier.
`created`	-	The date on which this candidate object was created.
`lastUpdated`	-	The date on which this candidate object was last updated.

4.1.2 The Job Representation

Data

Name	**Extraction Type**	**Description**
`title`	Rule-based	The job title.
`description`	Rule-based	The job description.
`keywords`	Rule-based	A list of applicable keywords. This field is only set when the job portal the job was crawled from provides it, which is not always the case.
`location`	Rule-based	The location of the job, where the candidate is going to end up working.
`source`	Rule-based	The job portal from which the vacancy was crawled.
`link`	Rule-based	A link to the original vacancy on the job portal.

Meta-Data

Name	**Extraction Type**	**Description**
`id`	-	Unique identifier.
`created`	-	The date on which this job object was created.
`lastUpdated`	-	The date on which this job object was last updated.

4.1.3 The Match Representation

Data

Name	**Extraction Type**	**Description**
`job`	-	The job-object tied to this match.
`candidate`	-	The candidate-object tied to this match.
`scoreManual`	-	The manually allocated score to this match, as given by a human annotator. Can get an integer value between 1 and 4. 1 being a bad match, and 4 being an especially good match.
`scoreAuto`	-	The automatic score as given by a ranking model. This score is used to rank the matches when viewed, by ordering them from high to low.

Meta-Data

Name	**Extraction Type**	**Description**
`id`	-	Unique identifier.
`jobID`	-	The unique identifier of the job-object.
`candidateID`	-	The unique identifier of the candidate-object.
`created`	-	The date on which this match object was created.
`lastUpdated`	-	The date on which this match object was last updated.

4.2 The Components of the System

4.2.1 The Web Application

The primary way to interact with the system is by means of the web application. This application provides the user with three main pages: one with the list of candidates, one with the list of jobs and one with the list of matches that are stored in the database.

The candidate- and job page provide so called CRUD-operations on the objects. This means that users can create new candidates or jobs and read (view), update or delete existing ones from the database[1].

By clicking on its accompanying button marked "View" in the list, the user is shown a more detailed description of the candidate or job. On this page, the user is also given the option to generate a list of suitable jobs or candidates, for the viewed candidate or job, by means of the Matcher (4.2.5). This gives a list of the five most suitable ones according to the current ranking model.

On top of the function to add a new candidate by means of a form, the user is also able to parse a candidate's information from his or her resume by uploading it to the application. This is done by the CV Parser (4.2.2).

Last but not least, all the matches that have been made by the system are stored and can be seen on the match page of the application. On this page the matches can be viewed and more importantly, given a score from 1 to 4 by the user.

4.2.2 The CV Parser

The task of this component is to extract structured information, in the form of a candidate object, from a person's resume. This makes it easier to search through the resume based on specific criteria like software known. It also helps the matching process, since more specific and powerful features can be devised from the object than from the original, unstructured text.

[1]Deleting, however, does not actually delete the object, but hides it so it can still be used for training and testing the matching algorithm.

Information Extraction is an entire field on its own. Many techniques have been developed, from rule- and grammar-based methods like CPSL[28] and more recently IBM's SystemT[29], to probabilistic methods like Conditional Random Fields (CRF) and other machine learning approaches.

The current CV Parser uses three different methods: a rule-based approach, based on regular expressions; a Named Entity Recognizer based on a CRF, pre-trained on recognizing names of persons; and Gazeteers, lists of known named entities. All of them have their advantages and disadvantages.

IE by Means of Regular Expressions

Like many commercial Information Extraction systems[30], the CV Parser uses a primarily rule- and knowledge based approach.

The CV Parser makes use of regular expressions as the main way of extracting information. Regular expressions are a way to express repeating patterns of characters using a special language, which the computer can use to extract matching instances from a series of characters. In our case, the series of characters consists of the full text of the resume.

Fields that are extracted in this way include the name of the candidate, the candidate's birth date, location, his or her job roles and his or her email address.

The parser assumes that each field is preceded by an accompanying label. For example, a candidate's name can be signified by the label *"Name:"*, as in *"Name: Hans-Christiaan Braun"*. For each field, the parser defines a single regular expression.

Regular expressions are a powerful tool to extract information in limited domains, where we know or can predict the structure of the text. While this is somewhat the case for resumes, that for a large part share common sections like *Education History*, *Work Experience* and *Personalia*, their structure is, arguably, still to variable to reliably extract information using a collection of regular expressions. Most of all because of he fact that natural language is very rich: concepts and their relations can be expressed in a myriad of (slightly) different ways.

IE by Means of Conditional Random Fields

As a backup system for recognizing the candidate's name, the parser makes use of a Named Entity Recognition (NER) model. The NER-model used is a Conditional Random Field from Stanford's OpenNLP library[2] that has been pre-trained to recognize names of persons[3].

To make the NER-model work, the resume is first tokenized. This means that the text is split into words and punctuation marks. In the next step, the model is applied to the tokenized text and the recognized name with the highest certainty is assumed to be the name of the candidate.

[2]`http://opennlp.apache.org`

[3]Namely `nl-ner-person.bin`, trained to recognize names of Dutch persons. `http://opennlp.sourceforge.net/models-1.5/`

While this approach is more advanced than a regular expression, it is still unreliable. Since the scope in which the name is searched consists of the entire resume, it leads to many false positives. Found names do not necessarily belong to the candidate, but can be names of employers or other people.

There is another disadvantage of this approach. The high density of other types of named entities, like software packages and programming languages means that, in many cases, other types of named entities are mistaken for person's names, leading to names like "Windows Server".

The unreliable nature of the system's name recognition, however, can be assumed to have no influence on the performance of the matching algorithm, since the name of the candidate is not going to be used for matching. In fact, it can be seen as discriminatory to do so.

IE by means of Gazeteers

The third approach that the component uses to extract information is based on Gazetteers. Gazetteers are directories of known entities like names of cities, days of the week, countries and others and are used to find occurrences of these entities in a text.

The CV Parser uses a web service of DBSpotlight, to recognize occurrences of programming languages, locations, software and educational institutes in the resume's text.

DBSpotlight is a collection of services provided by the DBPedia project[31]. The project's goal is to extract structured information from Wikipedia. It uses a framework called Resource Description Framework, or RDF, to describe entities and their relations, as represented on Wikipedia, in a computer readable way.

DBSpotlight's services annotate a given text with recognized entities. It provides the user with several ways of annotation: from simply identifying possible entities ('Spotting'), to disambiguating a list of already found entities.

The service that the parser uses is called 'Candidates'. It gives back a list of all spotted and disambiguated entities of a given type, ordered on a measure of confidence.

The parser takes all of the found software, programming languages and educational institutes and concatenates them in separate comma-separated strings. For the candidate's location, the most confident recognized location is chosen.

4.2.3 The Job Crawler

At the other side of the CV Parser lies the Job Crawler. Its job is to crawl a list of job portals at regular intervals for new vacancies, parse their contents into job objects and store them in the database.

It works in a straightforward manner. A crawler has been manually written for every job portal that needs to be crawled. It makes use of the structure of the underlying html-code of the web page containing the vacancy to apply hand written rules for extracting the required information.

This has advantages and disadvantages. The advantage is that it gives, in general, reliable results. Since web pages of vacancies are generally generated using predefined templates, the content differs, but the structure remains largely the same across pages. A disadvantage is that the technique is prone to break whenever the underlying template changes, for example after a visual overhaul. However, this can be assumed to happen not too regularly, so in practice a hand written crawler works quite reasonable.

4.2.4 The ElasticSearch Database

At the heart of the application lies an ElasticSearch search engine annex database. This means that ElasticSearch provides two core functionalities: it stores the data and provides a way to search through this data, returning a ranked list of the stored documents that matches the search criteria.

The search capabilities are used in two components. The Web Application and the CV Matcher.

It is used by the web application to provide the users with a way to search through the database of crawled vacancies and candidates using a simple text search field. This will search for occurrences of the entered text through the entire text of the candidate or vacancy objects, depending on the current section of the application.

4.2.5 The CV Matcher

The CV Matcher is the component that ties the functionality of the other components together to provide the user with a way to rank a list of candidates, based on a given job. It does this by leveraging the search functionality of ElasticSearch.

There are two types of queries in Elastic Search, 'leaf' and 'compound' queries.

Leaf queries map a single value to a single field in a document. A candidate, for example, can be queried on matching the term "Amsterdam" in his or her location field. The CV Matcher uses the 'match' query in particular. This leaf query gives a score to each document, representing the respective field's relevance of this document on this particular query. This is particularly useful for fields that contain free text, like the job description.

Compound queries combine leaf- and other compound queries to produce a super-query. The most straightforward compound queries are boolean ones. A 'should' query that combines two 'match' queries, for example, says that either one of the match queries should match, and that the overall score of the document depends on both. A 'must' query tells Elastic Search that all sub-queries must match and documents that do not are discarded.

Each query can be given a 'boost' parameter, a weight that determines the influence of this query's score to the overall score. A higher boost means a bigger influence.

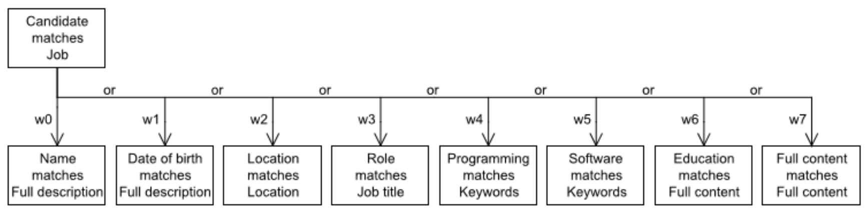
The query model that the CV Matcher uses can be seen in Figure 4.2. It uses a single 'should' compound query, combining eight 'match' leaf queries, where each leaf query has a boosting parameter (represented by the weights w_0 to w_7 in the figure).

The CV Matcher uses a Machine Learning method called an Evolutionary Algorithm to find the optimal values for the booster parameters. Evolutionary Algorithms are inspired by the way nature optimizes organisms by means of evolution. In effect, the evolution of a population of 'individuals' is simulated. In our case an individual consists of a set of weights $\{w_0, w_1, ..., w_n\}$. At each iteration, these steps are taken, in order:

1. Compute the 'fitness' or utility of each individual: the score on a problem-specific objective measure. The fitness function that the CV Matcher uses is the Mean Absolute Error between the ranks of the returned documents and their actual ranks, when applying the weights of this individual to the query (see Equation 4.1).
2. Randomly choose a number of individuals based on their fitness. Individuals with a high fitness have a higher chance to be selected.
3. Let the chosen individual 'reproduce': combine half of the weights from one individual with half of the weights of the other.
4. Randomly mutate some individuals, creating some needed variation to keep the maximal fitness of the population from plateauing.
5. Rinse and repeat steps 1 to 4 for a number of generations.
6. After producing the last generation, pick the individual with the highest fitness.

$$MAE_q = \frac{1}{N} \sum_{i=1}^{n} |r_{qi} - \hat{r}_{qi}| \tag{4.1}$$

In conclusion, the current system gives me a good basis to build the new matcher upon. It provides a way to extract information from resumes and job vacancies, although the quality of the resume parser may provide some challenges. This data can, in turn, be used to make feature vectors for training and testing the learning-to-rank algorithms. How these were implemented, tested and compared the algorithms is described in the next chapter: Methods & Results.



Candidate matches Job
or
w0
w1
w2
w3
w4
w5
w6
w7
Name matches Full description
Date of birth matches Full description
Location matches Location
Role matches Job title
Programming matches Keywords
Software matches Keywords
Education matches Full content
Full content matches Full content

Figure 4.2: The mapping of a job object search query with the candidate documents. The ”or’s” are Elastic Search ’should’ queries. (Reproduced from [3])

Chapter 5

Methods & Results

The first three sections of this chapter consists of a description on how the data was gathered and how it was transformed into feature vectors that could be used by the learning-to-rank algorithms for training and testing. The statistical testing of the results of the algorithms will be described in Section 5.4.

The rest of the chapter consists of the methods and results of the two main experiments:

In the first experiment, the GBRT, LambdaMART, SmoothRank and BSF algorithms are implemented and their performance are compared with the Evolutionary Algorithm on mean NDCG and mean AP.

In the second experiment, two types of manifold regularization are added to the SmoothRank algorithm and compared with the original SmoothRank algorithm, on mean NDCG an mean AP as well.

5.1 Gathering Data for Training- and Testing

Before we can start learning any ranking model, we need to have data to learn this model from.

The initial plan was to use relevance feedback on the generated matches, made by the users in the resourcing department of NCIM. Since there was already an option to provide this feedback within the current web application, this seemed, at first, to be a straightforward way to gather data. The feedback, as designed in the system, comes in the form of a score on a scale of 1 to 4, that the user could give to a generated match. This could be directly used as a label for the machine learning algorithms. By deploying the current system, the users in the resourcing department could already use the matches made by the current implemented algorithm, and at the same time, without much effort, score the generated matches.

In practice, this proved to be more challenging than expected.

First of all, since the system was not already up and running, it would have already been quite challenging to gather enough data in the six months the project would be running.

Second of all, the adoption rate of the system was poor, for which a couple of reasons can be identified. For one, there was barely any intrinsic motivation in the resourcing department to use the system, since the matches it generated were deemed not up to par. It did not help that it was separate from the resourcing system they used in practice, leading to the consistent need of switching between the two applications. Second of all, I, the author, am not a natural communicator, and I found it challenging to persuade the resourcing department to use the CVMatcher system either way.

There is also a more theoretical problem with using relevance feedback as labels. The fact that the documents are already ranked by the system means that it brings bias into the documents that are clicked, viewed and scored. The top-most documents have a higher chance to be viewed, and thus scored, than the ones at the bottom.

In the end, it was decided to use the company's historic placement data instead, in other words: which professional got placed on which project.

It has a few advantages over relevance feedback:

First of all, it can be seen as more objective and less biased, since it represents the actual acceptance and rejections of candidates by the clients and account managers.

Second of all, it provides us with an amount of data right from the start. We do not need to wait for labels to come in during the course of the project.

Fortunately, the data as stored in the database of the resourcing system follows the same general lines as the representation used by the current system. It consists of a table of matches, a table of requests (e.g. jobs) and a table of resources (e.g. candidates). The job information, the descriptions, titles and locations, could be taken directly from the request table. The information about the candidates was obtained by automatically parsing their resumes with the CV Parser: the migration code automatically looked through a folder containing all of the resumes from 2000 to 2013 and parsed the first document which title contained the candidate's name. Since the resumes followed a naming convention where the candidate's full name was used in the title, this approach worked quite well [1] .

5.2 Characteristics of the Gathered Data

Before we look into the transformation of the data into feature vectors, we are going to take a closer look at the data. When we know the characteristics of the data, we can adapt our approach to them and, hopefully, counteract weaknesses.

This section describes some of the most influential characteristics of the data and their challenges when we are using them for training and testing.

5.2.1 Matches

The most important characteristic of data, when it is used for machine learning, is its amount. The number of matches is, sadly, low: only 335. They are, in turn, divided over 49 jobs. The original amount of matches were higher, but the ranked lists were removed in which no relevant (read: placed) candidates could be found and the lists that contained only one candidate. The first were removed because the evaluation measures used are not defined on them, the second were removed because they have no ranking information that can be used to learn a ranking function.

Another important characteristic is the data's distribution with respect to the labels. It is more difficult to learn the difference between a good candidate and a weak one for a given job, when there are only a few candidates labeled as good or the other way around.

Unfortunately, the historical placement data contains few accepted candidates. This lies in the nature of the job-candidate matching problem: every vacancy is normally filled by one candidate. This means that each list of candidates consists of one, or at most three, candidates that have been placed. The rest are rejected: either by the account manager, responsible for the company-client contact, or the client themselves.

Another challenge lies in the distribution of the matches with respect to the jobs, in other words the lengths of the ranked lists. Many jobs have only a few candidates, while few jobs have many: almost 50% of the jobs have three or fewer candidates. The length-count of these lists seems to follow a power distribution, as can be seen in Figure 5.1.

The skewed distribution in favor of small ranked lists means that it is more challenging to learn a good ranking function. There is no ranking information contained in a list of only one item. In lists of two items the slightest change

[1] Only candidates with names consisting of at least two words were considered, since one-word names (consisting of either the first- or last name) were nearly impossible to point to a unique resume.

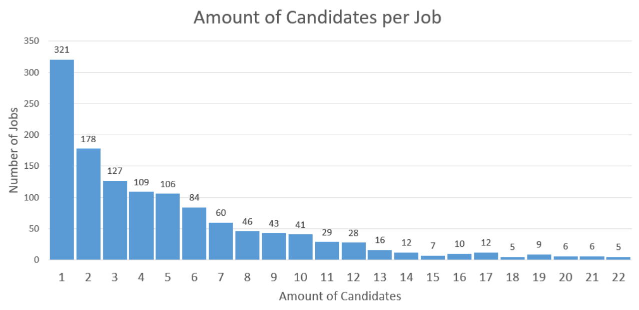


Figure 5.1: The number of candidates per job. As an example: for 321 jobs, only one candidate has been selected, while for only 109 jobs, four candidates were selected.

in the model, even when it is based on noise in the data, can lead to the either the absolute worst, or absolute best, possible ordering.

One may argue that a list of one or a few item does provide evidence on its rank when compared with other, unscored candidates. However, it is difficult to tell why a candidate has not been initially selected. Considering the large number of resumes, over 600, there is a non-neglectable probability that a candidate has not been selected based solely on the fact that his or her resume was never seen at all. It may very well be that other candidates, had they been considered, would have been more suitable than the accepted candidate.

The skewed length-count distribution will most likely also influence the testing of the learned models. On small lists a model has a more difficult time to 'prove itself', given the thin line between the perfect and worst ranking. Ranking small lists is also not representative of the setting in which the system is used, where in theory all of the candidates in the database must be ranked and the top-N is shown to the user.

5.2.2 Jobs

In general, the quality of the jobs stored in the resourcing database is okay. Sadly, 28% of the jobs do not have a description. This makes them for a large part unsuitable for matching, considering that the description makes up the bulk of a vacancy and generally stores useful information like the work experience and knowledge required for the job. Without a description, the most worthwhile information about a vacancy would consist of its title and location. The title generally contains the job title, e.g. 'Java Developer', so this may still provide some evidence towards a match. For example when it overlaps with one of the roles a candidate had in the past. Additionally, a job's location may overlap with a candidate's location, which may also provide some evidence.

Despite the loss of information on a job that a lack of description provides, the vacancies that lack a description were added to the training- and test data, because the amount of data is already meager as is.

Since the jobs have been manually filled in by the resourcing department over the last years, the quality of the fields that have been filled in is generally good.

5.2.3 Candidates

The quality of the candidates, as been given by the CVParser, is better than anticipated. However, since there was no time to test the parser thoroughly, by means of running it on a test set, its quality can only be judged in part by looking at the data from a broader perspective.

The role field is generally populated with the right values. Unfortunately, for 33% of the candidates no roles were extracted. There were doubts about the effectiveness of using one regular expression to extract the roles a candidate fulfilled in the past, given the myriad of ways that it can be expressed in a resume. The parser, however, does seem to provide good results in practice. Figure 5.2 gives a histogram of the 20 most frequent roles. The large amount of unique roles: more than twice as the amount of candidates (note that candidates can have multiple roles from multiple past jobs), does seem to indicate that there are many false positives.

The software and programming languages fields are filled with values that seem to be correct.

The education field is also filled in correctly with the names of the educational institutes the candidate attended, like the University of Amsterdam or the Haagse Hogeschool. However, it is questionable if this will be useful information for matching: it would be, in my opinion, better to have information about the studies a person followed than the institutes where the person followed them.

One problem with the extracted information is that a large part of the context is missing. How many years a person worked at a company, for example, and how many years ago and the amount of years a candidate has experience with a specific software package. Educational info, like (un)finished degrees and certificates would also be useful information to have when matching, since many vacancies state at least a preference for certain degrees.

5.3 Feature Generation

Feature generation is one of the more important parts of Machine Learning: a model is only as good as the information it can learn from. Now that we know how the raw data is structured and what their characteristics are, we can try to come up with a informative feature vector representation.

We start out with a collection of match objects, as stored in the Elastic-Search database. Each match consists of a job object, a candidate object and a respective manual score, indicating the goodness of this match. A job and candidate object, in turn, consists of a number of fields. Their structure has been described in Section 4.1.

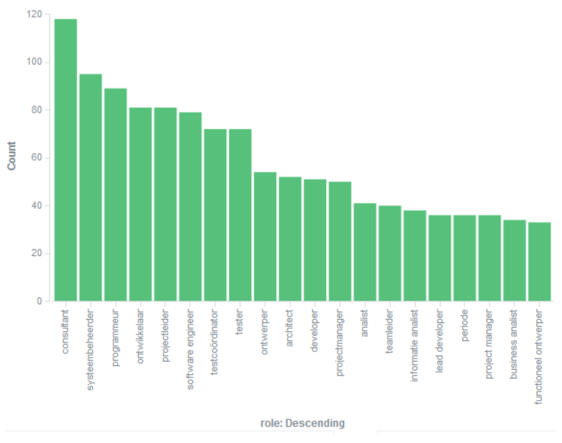


Figure 5.2:

The feature vector representation is based on the bag-of-words model, common in machine learning for Natural Language Processing. Table 5.1 gives an overview of the fields that are used in the final feature representation. For each field a separate bag-of-words model is produced by following these steps:

1. **Tokenization:** splitting the field into separate tokens or words.
2. **Stop word removal:** removing common, uninformative words, like ”the” and ”it” from large, free-form text fields.
3. **Dictionary generation:** generating a list of unique terms, where each term gets connected to a unique position in the feature vector.
4. **Term scoring:** generating a score for each job-candidate object, for each term in the dictionary.

Finally, the bag-of-words models (one for each field) are concatenated to form one feature vector representation.

5.3.1 Tokenization

The first step is to split each field into tokens. The comma-separated fields were divided on the commas. For example the roles ”software engineer, functional tester” would become: [”software engineer”, ”functional tester”]. This approach was chosen because each value can be considered named entities in their own rights; splitting them further into separate words would lose information: a ”test engineer”, for example, is different than a ”software engineer”.

The fields that contained free-form text, like the title and description of a job, are split on spaces and punctuation marks, using a regular expression, catered to the Dutch language, instead.

5.3.2 Stop Word Removal

From the fields that contain free-form text (the job title and description) Dutch stop words were removed. Stop words are common, mostly uninformative terms like "the" and "who". This de-emphasizes common terms (by removing them completely).

5.3.3 Dictionary Generation

The third step is to generate dictionaries, one for each field. A dictionary in this regard is a set of unique tokens. A simple, but effective approach was followed: for each field, all of the tokens were put together into one 'bag-of-words' and the top-N most frequent ones were taken as the dictionary terms. The amount of terms differed, based on the field. For the software, programming languages, education, roles and job title, the top-50 terms were taken, while for the job description the top-100 terms were taken. The dictionary size of the job description was increased because it contains the most information about the job. More importantly, the diversity of the information needs to be captured: job requirements can range from a specific degree to knowledge of a specific software package.

5.3.4 Term Scoring

In the last step, the score for each term in the dictionary, for each match, is calculated. Two scoring methods were used: Term Frequency scoring (TF) and Term Frequency - Inverse Document Frequency scoring (TF-IDF).

In Term Frequency scoring, the frequency of each dictionary term is counted and the result is placed at the dictionary term's position. For example, if the term "Microsoft" occurs five times in the job description and the dictionary term "Microsoft" is tied to the second position in the feature vector, this position will get the value 5 for this job-candidate pair.

This way of computing the scores, however, has its drawbacks. For one, we want the score to be a measure of how important or salient the given term is in the respective field. TF may provide an indication: important terms in a text generally occur frequently. However, this is not always the case the other way around: frequently occurring terms like "the" and "if" are generally not deemed important.

A more sophisticated approach is Term Frequency - Inverse Document Frequency scoring. It uses combines two scores, the TF-score and the Document Frequency, or DF, score of a term.

The Document Frequency of a term is the amount of documents in which this term occurs. In our case, a document corresponds with a field. If we take the inverse of the DF score (Equation 5.1) we get a measure of the informativeness of the given term. Combining the TF and IDF scores is commonly done by multiplying them together.

Object	Field	Regular Expression used for Tokenization	Preprocessing	Dictionary Size
Can.	`software`	”, ”		50
Can.	`programming Languages`	”, ”		50
Can.	`education`	”, ”		50
Can.	`roles`	”, ”		50
Can.	`location`			50
Job	`description`	”([, .&?!”′]\s+)|(\s+)”	Stop word removal	100
Job	`title`	”([, .&?!”′]\s+)|(\s+)”	Stop word removal	50
Job	`location`			50

Table 5.1: The job and candidate fields that are used in the feature vector representation and the characteristics of their transformation in a bag-of-words model.

$$\mathrm{IDF}(t) = \log \frac{N}{1 + \mathrm{DF}(t)} \tag{5.1}$$

Where N is the total amount of documents.

5.3.5 Apache Spark

I, the author, did not implement the feature generation pipeline (tokenization, stop word removal, tf-idf-score generation) myself. Rather, I used the methods from the Apache Spark library.

Spark is a machine learning, streaming and graph analysis library from the Apache software foundation. MLib [32], Spark's machine learning sub-library, implements many common feature generation methods. So, instead of reinventing the wheel, pre-existing ones were used.

An advantage of Spark is the fact that it is designed to be used with Apache Hadoop, a popular distributed computing platform. In the future, when more data would be available, Hadoop can be used to spread the computation across multiple computers and decrease the running time by a significant amount. During the project, however, a company provided laptop was used, which sufficed for now, considering the relatively low amount of data.

run	portion					test score
	1	2	3	4	5	
1	test	train				s_1
2	tr.	test	train			s_2
3	train		test	train		s_3
4	train			test	tr.	s_4
5	train				test	s_5
					average	$\mu([s_1, \cdots, s_5])$

Table 5.2: 5-fold cross validation. In each run a different permutation of 1 part test set, 4 parts training set is used. The final test score (for example mean NDCG) is computed by taking the average over all the runs.

5.4 Testing

The initial plan for gathering the testing data was to let an employee of the HR department and the CTO of the company, who offered his time and effort as well, make a test set manually. 30 candidates were randomly sampled and 10 jobs were hand-pick that represented NCIM's information need. The CTO and the employee, in turn, would have been asked to rank and label each job-candidate pair.

However, this plan was abandoned during the course of the project, because of the same reason the plan for gathering training data was changed. It would have been too much effort to make a sufficiently large test set this way.

As a solution, cross validation was used instead. Cross validation means splitting the (historical placement) data into a training and test set multiple times, each time using one, not yet used, portion as a test set and the rest for training. The average of all runs on, for example, mean NDCG on the test set, is reported. Table 5.2 illustrates this for the 5-fold case.

The main advantage of cross validation is that it makes the most effective use of the gathered data. When randomly splitting a data set once in a train and test set, there is a high chance that the test set is not representative of the whole data set. When using large test sets the individual differences cancel each other out, but in small data sets, as we have, this is generally not the case. By averaging over multiple runs, however, we can generally get a good idea of the tested model's overall performance.

Cross-validation only works when the training and test set come from the same pool of data, so this was another advantage of getting the test data from the same source.

5.4.1 Statistical Testing

Measuring the performance of two systems and comparing them using cross validation is not enough, however. Like in every experiment, there is a fair amount of spurious factors involved that could influence the outcome, either positively or negatively, but that do not necessarily represent one system outperforming the other.

This is where statistical testing comes in to play. With a statistical test we can quantify the probability of the results being different only because of chance or other factors. If this probability (also called the $p-$value) is significantly small, smaller than a predefined threshold (alpha), we conclude that it is likely that the two systems perform differently because they *are* different.

There are a couple of statistical significance tests that can be used to check the performance of Information Retrieval systems.

T-test

One test that can be used is the t-test. It assumes that the probability of the results are normally distributed. In our context this means that, simply said, it assumes that the results of different runs of one system on the same test set tend to cluster together around an average. The further results lie away from this average, the more unlikely they are.

Wilcoxon Signed-rank Test

There are also tests that do not make this assumption. One of them is the Wilcoxon signed-rank test.

This test has the advantage that it does not assume that the results are normally distributed. However, the disadvantage of the test is that it has to transform the results first, before the test can be applied.

The transformation takes the paired differences of the results (for example the differences in NDCG score of two systems on the same ranked lists) and orders them based on their absolute value (smallest difference first). In the next step, the rank of each difference is multiplied by its original sign (e.g -1 or $+1$) and the sum of these scores is computed. Whether the resulting score is statistically significant or not, based on the amount of results and selected alpha-value, can be looked up in a table.

During this process, information about the differences is lost, which may lead to wrong $p-$values, and thus wrong conclusions. Smucker at al.[33] compared five different statistical tests, and found that the Wilcoxon signed-rank test was ill-suited for evaluating IR-systems.

Permutation and Randomization Tests

Another test that can be used is the Permutation test. This can be seen as the brute force approach to statistical testing.

Under the hypothesis that both the results of system A and system B are only produced by mere chance, we can model this behavior by having a system N that produces a random labeling of the result-pairs. When producing a large amount of results, the difference in the mean NDCG score, for example, would be roughly the same for both system A and B, and system N.

	Actual NDCG scores	
List	System A	System B
l_1	0.75	0.5
l_2	0.9	0.4
l_3	0.5	0.8
l_4	0.43	0.6
l_5	0.6	0.7
mean	0.636	0.6
diff.	0.036	

$\rightarrow$

	Sampled NDCG scores	
List	System A'	System B'
l_1	0.75	0.5
l_2	0.4	0.9
l_3	0.8	0.5
l_4	0.6	0.43
l_5	0.6	0.7
mean	0.63	0.606
diff.	0.024	

Table 5.3: An example permutation in a permutation or randomization test. The results of System A and System B are shuffled to produce two new result sets A' and B'. The same testing regime as used on the original results (e.g. cross-validation) is used on the shuffled ones, producing two new scores: one for system A' and one for B'. The difference between the two results is remembered, giving a distribution of score-differences at the end of the permutation test. Finally, the actual difference is compared with the distribution of sampled differences.

The permutation test operates on this idea by implementing such a system N. It takes the results of both systems under analysis and labels each result as either coming from System A or from System B. Afterwards, it takes the mean of both the scores from 'System A' and 'System B' and remembers their difference. It does this for each possible labeling of the scores.

In the end, we get a probability distribution that we can compare with our found difference of system A and B. The p-value can be gotten by looking at the fraction of differences produced by system N that are higher or lower than our found difference. If this is lower than our predefined alpha, we can conclude that the results are statistically significant.

The Permutation test has a few advantages over other statistical tests. First of all, the found $p-$ value is exact, and not approximated like in the t- or Wilcoxon test. Second of all, we can test any statistic with the Permutation test, like the median, while other tests like the t-test are only defined on means or on a special measure like in the Wilcoxon's signed-rank test. And last but not least, like the Wilcoxon's signed-rank test, it does not make assumptions on the distribution of the data.

A big disadvantage, however, is the fact that computing all the possible labelings is an expensive operation, even on today's computers. A list of 30 result-pairs, for example, already has $2^{30} = 1,073,741,824$ different possible labelings.

One solution is to randomly sample from the possible labelings, which is commonly called a randomization test. If we take enough samples, the results should serve as a good approximation of the actual distribution. This comes at the cost of accuracy: the computed p-value is not guaranteed to be exact anymore. However, given enough samples, we can generally receive a good trade-off between running time and low error margin.

Ultimately, the randomization approach was chosen for the aforementioned reasons. The α-value is set to 0.0042, instead of the common value of 0.05, because we will be doing many statistical tests by comparing the four different learning-to-rank algorithms on two measures (MAP and NDCG) with the Evolutionary Algorithm. This leads to a total of 8 tests, plus an additional 4 tests for the semi-supervised techniques in the second experiment. By doing this substantial amount of tests, the probability increases, statistically speaking, that we get one or more significant results because of pure chance. We can, fortunately, control for this challenge by lowering the α-value by dividing the total alpha through the number of tests (which is called Bonferroni correction). This leads us to the new alpha of $0.05/12 = 0.0042$. Note that this is a conservative bound, since it assumes that the statistical tests are completely independent.

The amount of randomization test samples was set to $100,000$. Unfortunately, the amount of samples needed to limit the error rate on the p-value is difficult to compute, since a randomization test makes no assumptions on the data distribution.

5.5 Experiment 1: Supervised Learning-to-Rank

Now that we know how we are going to gather the data, how we turn them into feature vectors and how we are going to test the learned ranking models, it is time to implement and compare the GBRT, LambdaMART, SmoothRank and BSF algorithms.

5.5.1 Hypothesis

As in every well crafted experiment, we need to write down our hypotheses, which we will test and discuss later on in this thesis.

I, the author, expect that the SmoothRank algorithm performs the best, compared to the Evolutionary Algorithm, since it performed well on other datasets in the past[16] and, since it is a list-wise approach, optimizes the NDCG measure the most directly.

LambdaMART will be in second place. According to Tax et al.[16], it has been tested on only 3 datasets on the NDCG@3 measure and was better than the other models that have been tested on them in 40% of the cases. However, since an ensemble of this type of model won the Yahoo! challenge[27], it could very well be a good contender nonetheless.

The GBRT classifier will, according to my prediction, rank third. Since it does not take any one common ranking measure into account during training, I expect it to provide worse results than the ones that do (SmoothRank and LambdaMART).

The Evolutionary Algorithm will rank fourth. I think that the learning potential of the algorithm is limited, since it learns only a couple of boosting parameters of a fixed Elastic Search query model. I have doubts if this handful of parameters can capture the complex relations between a job and a candidate. It does have the advantage that its fitness function is based on absolute ranking errors, and thus catered to the ranking problem, in contrast with the GBRT classifier.

I predict that the trained Best Single Feature model will provide the worst performance. This model is the simplest of all of the implemented algorithms and is specifically provided as a baseline for comparison with the others.

Making the Hypotheses Testable

It is not enough, however, to put the hypotheses in words. We need to define them in a way to make them testable with a statistical test.

Since the main question is whether one or more of the implemented algorithms perform better than the original Evolutionary Algorithm, the tests and accompanying hypotheses will compare the mean NDCG and AP scores of the Evolutionary Algorithm with the performance of each of the new algorithms.

Keeping all of this in mind, the tests and accompanying hypotheses were defined as below. Since the assessed performance of the models was expected not to change based on the measure used, these hypotheses are replicated for each individual measure (mean NDCG and MAP), indicated by the M superscript in each test.

$$
\begin{array}{llll}
ST_1^M: & H_0: \mu_{\text{SmoothRank}} \leq \mu_{\text{Evo. Algorithm}} & H_1: & \mu_{\text{SmoothRank}} > \mu_{\text{Evo. Algorithm}} \\
ST_2^M: & H_0: \mu_{\text{LambdaMART}} \leq \mu_{\text{Evo. Algorithm}} & H_1: & \mu_{\text{LambdaMART}} > \mu_{\text{Evo. Algorithm}} \\
ST_3^M: & H_0: \mu_{\text{GBRT}} \leq \mu_{\text{Evo. Algorithm}} & H_1: & \mu_{\text{GBRT}} > \mu_{\text{Evo. Algorithm}} \\
ST_4^M: & H_0: \mu_{\text{Evo. Algorithm}} \leq \mu_{\text{BSF}} & H_1: & \mu_{\text{Evo. Algorithm}} > \mu_{\text{BSF}}
\end{array}
$$

5.5.2 Methods

The algorithms that are compared cannot be used directly, as is. Like many machine learning algorithms, they have so-called hyper parameters that should be chosen before they can be run.

A common hyper parameter is the amount of iterations that an algorithm should take. Too few and the learned model has the risk of not modeling the training data correctly, too many and there is a risk of "overfitting": modeling the training data too well, at a cost of generalizing on unseen data.

Not only the amount of iterations has a large influence on an algorithm's performance, hyper parameters have a large influence in general. In this regard, choosing the right values for them is an important factor. It is also a challenge, since the value of a parameter that gives the "best" performance more often than not depends on the data used. And, like in the example of the amount of iterations, there is generally not a rule of "bigger (or smaller) is better".

Many times, the best values for the hyper parameters [2] can only be found by testing the model with different parameters using cross-validation or on a separate validation set of data points, not used for training or testing. This can be done by trial-and-error, or by a more rigorous approach, of which the simplest is an exhaustive search of all combinations of parameter values.

In this section, the hyper parameters and other different setting of the used algorithms are described, so the results of the experiments can hopefully be reproduced in the future, may the need arise.

GBRT

The GBRT-algorithm was not implemented during the research, the one from the Apache Spark library was used instead. It was trained for 10 iterations, meaning that the final model consisted of 10 trees. The Squared Error (Equation B.1) was used as the loss function and the learning rate was set to the default of 0.1.

These parameter values were chosen by means of manual trial and error.

LambdaMART

The LambdaMART model was trained for 1000 iterations with a learning rate of 0.1.

To keep the individual models 'weak', RankLib's implementation of LambdaMART deliberately fixes the amount of nodes in each tree to 10.

These were the default parameters, which seemed to work well in practice.

SmoothRank

Like the authors in the paper, simulated annealing was used. The sigma started at a value of 64 (2^6) and it was divided by two after 100 iterations until it

[2]the ones that lead to a model that generalizes well to the test set and other unseen data)

reached a value of 1 (2^0), leading to a combined total of 600 iterations. These correspond with the same values as used in the paper, since these produced good results. Changing them did not lead to a gain in performance.

Best Single Feature

The only parameter BSF needs is the evaluation metric used to measure its performance. Since both SmoothRank and LambdaMART use NDCG, NDCG was used for the BSF model as well, to make comparison easier.

5.5.3 Results

Table 5.4 shows the results of training and testing the five models using 5-fold cross-validation. Table 5.5 shows the p-values of the differences in mean NDCG score after running permutation tests on the results. It also indicates the results deemed statistically significant.

Now that we know the p-values we can take a look at our hypotheses we wrote down in Section 5.5.1. Based on these results, we can safely reject these four null-hypotheses and accept their alternative counterparts:

$$\begin{array}{llll}
ST_8^{\text{MAP}}: & H_0: \ \mu_{\text{GBRT}} \leq \mu_{\text{Evo. Algorithm}} & H_1: & \mu_{\text{GBRT}} > \mu_{\text{Evo. Algorithm}} \\
ST_7^{\text{MAP}}: & H_0: \ \mu_{\text{LambdaMART}} \leq \mu_{\text{BSF}} & H_1: & \mu_{\text{LambdaMART}} > \mu_{\text{BSF}} \\
ST_6^{\text{MAP}}: & H_0: \ \mu_{\text{LambdaMART}} \leq \mu_{\text{Evo. Algorithm}} & H_1: & \mu_{\text{LambdaMART}} > \mu_{\text{Evo. Algorithm}} \\
ST_9^{\text{MAP}}: & H_0: \ \mu_{\text{GBRT}} \leq \mu_{\text{BSF}} & H_1: & \mu_{\text{GBRT}} > \mu_{\text{BSF}}
\end{array}$$

The rest of the differences were not statistically significant, having p-values higher than the threshold of 0.0025.

Model	Mean Cross-Validation Score		
	NDCG	AP	Kendall's RCC
GBRT	0.816	0.741	0.236
LambdaMART	0.819	0.727	0.188
SmoothRank	0.766	0.652	0.163
Best Single Feature	0.659	0.474	-0.233
Evolutionary Algorithm	0.652	0.464	-0.013

Table 5.4: Mean score of the five integrated models on three common evaluation measures after training and testing using 5-fold cross-validation (The total amount of ranked list used for training and testing amounts to 49).

	$p(\mu_M < \mu_{Evo.Algo.})$	
Model (M)	NDCG	AP
GBRT	0.049	0.0001*
LambdaMART	0.054	0.0006*
SmoothRank	0.172	0.0176
Best Single Feature	0.731	0.6986

Table 5.5: The p-values of the difference in mean NDCG and AP-score between each newly implemented algorithm and the current Evolutionary Algorithm. The rows are taken as the baseline. As an example: the GBRT model performed significantly better than the EA model on mean AP, with a p-value of 0.0001. Significant results are noted with a star.

5.6 Experiment 2: SmoothRank with Manifold Regularization

One of the main challenges of the used data is its small amount. Unfortunately, more labeled training data would be difficult to come by, since it is labor intensive to manually annotate the data. However, we can easily generate new, unlabeled data points by combining jobs with candidates to produce new pairs not already contained in the dataset. Can we somehow use this unlabeled data to guide the training process of models?

Good news! We can. This is commonly called semi-supervised learning, since we still use supervised learning methods, but add techniques to it to make them able to learn from unlabeled points as well.

There are many semi-supervised techniques, but this experiment will focus on a concept called Manifold Regularization (MR).

5.6.1 Manifold Regularization

Manifold Regularization is build on the idea that data points that are similar in some shape or form should have similar scores as well. For example, a web-developer that shares knowledge of software packages with a developer that has been accepted for a job, is likely to be a good candidate for the same job offer as well. MR actually turns this notion around: a good model should give a scores to the labeled data points that are similar to its most similar, unlabeled neighbors.

To reach this goal, MR adds a regularization term to the used loss-function that penalizes models where the sum of the score differences between a labeled point and its neighbors, weighted by the distance between each pair, is too high:

$$L(f, S_{train})^{+} = L(f, S_{train}) + \gamma \cdot \sum_{x_i \in S_{train}} \sum_{x_k \in \mathrm{NN}(x_i)} W_{ki} \cdot (f(x_k) - f(x_i))^2 \quad (5.2)$$

Where $L(f, S_{train})$ is a loss-function, defined on a model f and a training set S_{train}, $\mathrm{NN}(x_i)$ is a list of unlabeled data points that are similar to the labeled data point x_i, and W_{ki} is the weight or distance between the two points, for example $(1 - \mathrm{Sim}(x_k, x_i))$ and γ (gamma) is a hyper parameter controlling the influence of the MR term on the original loss function.

However, there is a characteristic of high dimensional (feature) spaces that can lead to issues when following the aforementioned approach: the ominous sounding *curse of dimensionality.* One effect of this curse is that in high dimensional spaces, similarity between data points become meaningless, since every data point becomes as dissimilar to each other as the rest.

This, however, is only the case when the data points are roughly independent and identically distributed throughout the feature space, which may be a too stringent assumption to make. Not every feature is as important or completely independent from the others. In fact, it would be difficult to learn a model

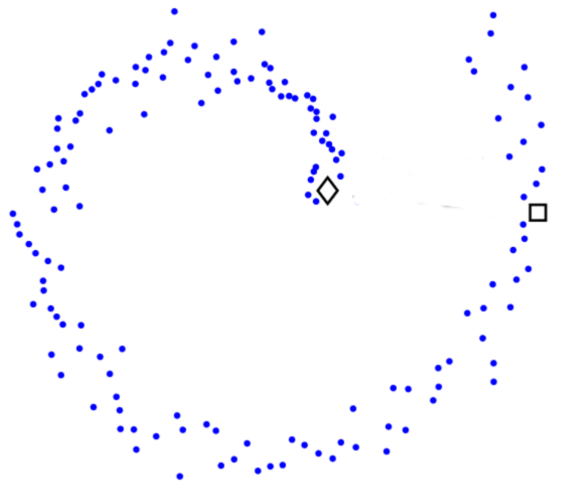


Figure 5.3: The spiral, a common example of a manifold.

when this would not be the case. Instead, we can assume that the data lie on a so-called manifold.

The Manifold Assumption

The manifold assumption states that the data points tend to cluster together. More specifically, it is assumed that the points lie on a so-called lower dimensional manifold, embedded into the often high dimensional feature space. An n-dimensional manifold can be seen as a shape that can be transformed into another, n-dimensional shape by 'twisting and turning' it, but not 'cutting' it. In other words, this higher dimensional shape is *homomorphic* to a lower-dimensional one.

A famous example of a manifold is the spiral, which can be seen in Figure 5.3. Despite all of the points being embedded in a two-dimensional space, a clear one-dimensional relationship can be seen in the data, although dis-formed into a spiral. This spiral, however, can be unrolled into a line (plus/minus some noise).

There is one issue that still needs to be addressed before we can implement manifold regularization. That is: how are the lists of similar, unlabeled data points, NN(x_i) in Equation 5.2, generated?

Nearest Neighbor List Generation

The most common approach of generating the list of similar data points is by calculating the similarity between a labeled data point with each unlabeled one and taking the k most similar ones. This is commonly called the k-nearest neighbor method.

Another approach is to take all of the unlabeled data points that have a similarity value higher than a given threshold.

Similarity in these cases is defined as a score on a similarity measure. There are many similarity measures, but the one that is the most suited for our problem is the cosine similarity. It is defined as the cosine of the angle between two (feature) vectors. Considering that it can be applied to any two real-valued vectors, it is perfect for our bag-of-words feature representation.

SmoothRank with Manifold Regularization

Since the SmoothRank model provided the best access to the source code, considering it had been implemented from scratch, it was the prime candidate for adding Manifold Regularization. In fact, two types of MR have been tried. The first is the standard method, based on differences in predicted scores (see Equation 5.2). The second method, however, is catered to the ranking problem and may provide better results.

SmoothRank with Standard Manifold Regularization The first version of the new SmoothRank loss function can be seen in the following equation.

$$\sum_{i,j} G(y_{qi}) \cdot \mathrm{ND}(r_{qi}) \cdot h_{qij} + \gamma \cdot \sum_{x_k \in \mathrm{NN}(x_i)} 0.5 \cdot (1 - \mathrm{Sim}(x_k, x_i)) \cdot (f(x_k) - f(x_i))^2 \tag{5.3}$$

Since the derivative of a sum is the sum of its derivatives, we can easily rewrite the original derivative to include the MR addition, by adding its derivative to the end:

$$\sum_{q} \sum_{i,j,p} G(y_{qi}) \cdot \mathrm{ND}(r_{qj}) \cdot \frac{d\, h_{qij}}{d\, f(x_{qp})} \cdot \vec{x}_{qp} + \gamma \cdot \sum_{x_k \in \mathrm{NN}(x_i)} (1 - \mathrm{Sim}(x_k, x_i)) \cdot (f(x_k) - f(x_i)) \tag{5.4}$$

SmoothRank with Rank-based Manifold Regularization Constraining the model in such a way as to make similar items have similar score may be too strict for the ranking problem. For a ranked list, the scores do not directly matter, only the rank, based on these scores, have influence on the final value of the evaluation measure.

The smooth indicator function h_{qij}, as SmoothRank uses, could be used to this end. By replacing the predicted model scores $f(x_k)$ and $f(x_i)$ with h_{qij}

and h_{qkj}, the Manifold Regularization term gets a different explanation: similar data points should have a similar probability of ranking at the same places. In other words: they should have similar probability mass functions, defined on the rank variable.

This leads to a new formulation of the loss function..

$$\sum_{i,j} G(y_{qi}) \cdot \mathrm{ND}(r_{qi}) \cdot h_{qij} + \gamma \cdot \sum_{x_k \,\in\, \mathrm{NN}(x_i)} 0.5 \cdot (1 - \mathrm{Sim}(x_k, x_i)) \cdot (h_{qij} - h_{qkj})^2 \quad (5.5)$$

..and its derivative:

$$\sum_{q} \sum_{i,j,p} G(y_{qi}) \cdot \mathrm{ND}(r_{qj}) \cdot \frac{d\, h_{qij}}{d\, f(x_{qp})} \cdot \vec{x}_{qp} + \gamma \cdot \sum_{x_k \,\in\, \mathrm{NN}(x_i)} (1 - \mathrm{Sim}(x_k, x_i)) \cdot \left(\frac{d\, h_{qij}}{d\, f(x_{qp})} - \frac{d\, h_{qkj}}{d\, f(x_{qp})} \right) \quad (5.6)$$

5.6.2 Methods

Like the algorithms in Experiment 1, some parameters have to be chosen before the algorithms can be run. For Manifold Regularization, there are a couple.

The first question that needs to be addressed is how the unlabeled data points, from which the lists of nearest neighbors are computed, are generated. This was done by combining unseen job-candidate pairs, 100 for each job.

The second question: how are the lists of nearest neighbors computed? In the end, a hybrid approach between the k-Nearest Neighbors and the threshold method was used. Data points in the list should have at least a similarity of 0.5, and the list should have at most 5 items. This approach was chosen to counteract the weaknesses of both methods: the k-NN technique can include items that are dissimilar to each other, whilst the threshold technique can lead to inclusion of too many items, which can influence the running time of the algorithm negatively.

The third question: what similarity function is used to compute the list of nearest neighbors? The cosine similarity function was chosen, because of the same reasons as described in Paragraph 3 of Section 5.6.1: it is directly applicable to the bag-of-words model used.

The fourth and final question is what to set the gamma value, that controls the influence of the regularizer on the final result, too. Gamma was eventually set to 0.1, after trial and error, since this proved to provide a good trade-off between true supervised learning ($\gamma = 0.0$) and a too big of an influence of the MR-term on the resulting model.

The parameters of the original SmoothRank algorithm remained unchanged (see Section 5.5.2)

5.6.3 Results

Table 5.6 shows the results of both types of Manifold Regularization, next to the results of the original SmoothRank algorithm. Tables 5.7 and **??** contain the results of the statistical tests for NDCG and AP respectively.

Model	Mean Cross-Validation Score		
	NDCG	AP	Kendall's RCC
SmoothRank + MR Type 2	0.797	0.694	0.130
SmoothRank + MR Type 1	0.777	0.679	0.132
SmoothRank (original)	0.766	0.652	0.163

Table 5.6: Mean score of the three SmoothRank models, the original and including both types of Manifold Regularization, on three common rank-based evaluation measures after training and testing using 5-fold cross-validation.

	$p(\mu_M < \mu_{SmoothRank})$	
Model (M)	NDCG	AP
SmoothRank + MR2	0.861	0.688
SmoothRank + MR1	0.936	0.780

Table 5.7: The p-values of the difference in mean NDCG and AP-score between both semi-supervised versions and the supervised version of the SmoothRank algorithm. The rows are taken as the baseline. There are no statistically significant results.

Chapter 6

Discussion

Having results, as we have gotten in the previous chapter, is fine and dandy. The question is: what do these results mean? This meaning depends for a large part on your viewpoint. Two main viewpoint can be identified for this research: the viewpoint of a scientist and researcher, and the view of NCIM, the company for which this new matching implementation has been made.

From the company's point of view, there seems to be one major result. Two of the four algorithms, namely GBRT and LambdaMART, perform significantly better than the previously used Evolutionary Algorithm.

SmoothRank performs better as well, although not significantly. It should be noted, however, that the used alpha value of 0.0042 is relatively low. The used method to control for multiple comparisons (Bonferoni) is a conservative one that assumes that every test is independent. This might be too strict of an assumption, since an algorithm performing better than another on one metric makes it more likely to perform better on another as well.

Which one of the two highest-performing algorithms is in fact the best is difficult to say, given the high p-values. This is presumably for a large part due to the limited amount of training and testing data, namely 49 ranked lists in total. The need for training data is especially great for learning-to-rank in contrast to classical machine learning, since we do not only need one score for each document but a score for each document-query pair.

On top of more training data in general, having more evenly distributed data may help as well. Each job should, in the best case, have a similar amount of scored candidates. Right now, this distribution is skewed, with few jobs having many candidates and the other way around. Especially the fact that many lists have only two or three candidates is influential: there is only one switch in the order of two documents necessary to get from the worst to the perfect ranking of a list of two items. This makes the results on the evaluation measures highly volatile during training and testing.

Having more qualitative data might help as well. The candidate data is produced by a relatively simple algorithm that mainly uses regular expressions and ontology lookups to extract information from the candidate's resumes. More sophisticated models, like Conditional Random Fields or rule-based models based on GATE/JAPE/ANNIE[34], may provide better and more informative data to extract features from.

On the topic of features: adding more advanced features in general may help as well. This research used a common, but relatively simple, feature vector representation, based on a bag-of-words model. More informative features, like the amount of overlapping terms between the job title and role of a candidate, or the distance between a candidate's and a job's location may provide better results.

Another step is to apply normalization to some of the fields. A single job role can have multiple aliases. A Software Engineer can be, for example, also described as a Software Developer. By applying a mapping from aliases to a common concept, say by means of a hand-made ontology, the precision of these kinds of features can be increased.

From a researcher point of view, one result especially caught my eye. The GBRT algorithm, who has not been tailor-made for the ranking problem, in contrast to SmoothRank and LambdaMART, actually performed the best. Although, as said, the differences are not significant and relatively small. One would have thought that LambdaMART and SmoothRank would have a particular edge compared to the non-learning-to-rank GBRT model. Again, with more training and testing data, where the lists are larger and thus the ordering of the lists has more influence, this may very well change the results in favor of the learning-to-rank approaches.

This leads us to the Manifold Regularization techniques, added to SmoothRank. They perform only slightly better than the original, purely supervised algorithm, and these differences are far from statistically significant. Nonetheless, they should not be completely written off. The fact that Manifold Regularization is rooted in existing, validated research does warrant more research into this subject.

In fact, more sophisticated Manifold Regularization methods can be devised on the basis of the smooth indicator function. Since it can be regarded as a probability mass function, regarding the rank of a document, similarity techniques on the basis of divergences or other statistical distances can be used, instead of directly comparing the individual rank probabilities.

Chapter 7

Conclusion

Matching job vacancies with candidate employees is in no way a simple problem that is easily solved. As said, all the way back in the introduction, a good match not only depends on 'hard' factors, like relevant job experience and diplomas, but also on many 'soft' ones. These include traits like a person's motivation and passion for the job, and her or his eagerness and ability to learn. Traits that are just as important, perhaps even more.

The CVMatcher system is not, and may very well never be, able to select suitable candidates based on these soft factors, since this requires a human touch. However, it is definitely able to support NCIM's Human Resourcing department in the first steps of the selection process, by providing a recommendation in the form of an ordered list of potentially suitable candidates.

This research, however, has indicated that the current automatic way of ordering the candidates can be improved. Two of the three methods that have been integrated, a Gradient Boosted Regression Trees and a LambdaMART model, perform better than the current ranking model, that has been learned by an Evolutionary Algorithm.

The two added Manifold Regularization techniques seem to improve the performance of SmoothRank slightly, but not enough to draw any definitive conclusions. More research is needed, for example on other, simulated datasets.

All in all, when taking the results and discussion into account, these recommendations can be made to NCIM:

- Replace the current ranking method with the learned LambdaMART model. Although this model is, statistically speaking, not significantly better than the two other new models, it has, in my opinion, a good enough providence to warrant using this method instead of the GBRT model.

- Let the Human Resourcing department use the system. More importantly: let them give the needed feedback through the already implemented relevance feedback option. This will give the learning-to-rank algorithm the data it needs to improve the model. It will also lead to extra test data:

with enough data points we can say with more confidence which of the methods is the best for this given challenge.

- Regularly train the learning-to-rank model whenever a good amount of new data has been gathered.
- When the amount of training data becomes too large to train on a single machine, consider using the GBRT implementation in Apache Spark instead. Since it has been designed specifically for computer clusters in mind, and the newly made learning-to-rank addition uses it for feature generation as well, it can easily be adapted to run on a cluster.

Appendix A

Mathematical Notation

A.1 General

Notation	Meaning
$1_{condition}$	Evaluates to 1 when the boolean condition is true and 0 in all other cases. $1_{i=5}$, for example, evaluates to 1 whenever i is equal to 5.

A.2 Machine Learning

Notation	Meaning
$C = \{c_1, c_2, \cdots, c_k\}$	The set of all possible classes in a classification problem.
$\vec{x_i}$	The i-th feature vector in a set of feature vectors.
y_i	The actual label of the i-th feature vector.
$S = \{(\vec{x_1}, y_1), (\vec{x_2}, y_2), \cdots, (\vec{x_n}, y_n)\}$	A set of feature vectors and accompanying labels.
S_{train}	A training set.
S_{test}	A test set.
f	A model.
$f(\vec{x_i})$	The output of the model, after being applied to the feature vector $\vec{x_i}$.
$\hat{y_i}$	The label of $\vec{x}_i$, as predicted by a model. Synonym for $f(\vec{x})$.

Notation	Meaning
L	A loss or objective function.
$L(f, S_{train})$	The value of the loss function L on the model f and the training set S_{train}.

A.3 Learning to Rank

Notation	Meaning
$\vec{x}_{qi}$	A feature vector, defined on the query q and the document i.
r_{qi}	The actual rank of a document i on a query or in a ranked list q.
$\hat{r}_{qi}$	The predicted rank of a document i on a query or in a ranked list q.
y_{qi}	The relevance score of a document i on a query or in a ranked list q.
$\hat{y}_{qi}$	The predicted score of a document i on a query or in a ranked list q.
y_{qr}	The relevance score of a document at the true rank r on a query or in a ranked list q.
$y_{q\hat{r}}$	The relevance score of a document at the predicted rank r.

A.4 Ensemble Methods

Notation	Meaning
f	An ensemble.
$f(\vec{x}_i)$	The output of the ensemble, after being applied to the feature vector $\vec{x_i}$.
m_k	The k-th sub-model of an ensemble.

Appendix B

Machine Learning Theory

Machine Learning is a field in the domain of Artificial Intelligence. It concerns giving a computer the ability to learn to make predictions based on a collection of data points. An example would be to predict the cost of a car, based on a database of existing cars (and accompanying price tags).

More formally, a Machine Learning problem can be defined as the problem of learning a function $f(\vec{x})$ (commonly called a *model* of the data) to map a data point x to a value y. This data point can be a single number, but generally takes the form of a list (a *vector* ($\vec{x}$)) of values.

B.1 Learning a Model

The main question is: how do we learn this model? In general, a so-called supervised machine learning algorithm learns to predict y by applying a model $f(\vec{x})$ on a data point, measuring how well it performs (how well it 'fits' the data, by comparing the predicted score $\hat{y}$ with the actual score y) using an *objective function*, and adapting the model in a way that improves the score on this measure in the future. This is repeated over a set of data points, commonly called the training set, either until a fixed amount of repetitions has been reached, or if another condition has been met. For example if a certain objective measure score on the training set has been reached.

Now the goal is to find a model that either maximizes the objective function, or minimizes it, depending on the type of measurement used. When the goal is to minimize the function, the objective function is commonly called a *loss function*, since it represents some 'loss' or 'cost' that needs to be minimized. When the objective is to maximize the function, it is commonly called a *utility* function.

B.1.1 Objective Functions

Objective functions can be difficult to wrap ones head around, so an example can help.

A common loss function used in (supervised) regression is the *Sum of Squared Errors* or SSE. Its formula is defined as follows:

$$SSE(f, S_{\text{train}}) = \sum_{(\vec{x}_i, y_i) \in S_{\text{train}}} (y_i - f(\vec{x}))^2 \tag{B.1}$$

S_{train} are the training data, consisting of $(\vec{x}, y)$ pairs in which $\vec{x}$ corresponds with the item and y corresponds with its score. $\vec{x}$ represents, for example, a car, while y is its price in Euros. In the above formula, $f(\vec{x})$ corresponds with the output of the model[1], after given an appropriate $\vec{x}$.

Intuitively speaking, Formula B.1 computes a modified version of the sum of *errors* of each data point in the training set for the current model f. If the model, for example, predicts 4.5 for a point $\vec{x}_i$, but the actual class is 2.1, the error becomes:

$$E(f, (\vec{x}_i, y_i)) = \quad y_i - f(x_i) = \quad 2.1 - 4.5 = \quad -2.4 \tag{B.2}$$

As you may have noticed, this can result in negative errors, which would cancel out positive errors when we sum them as in Formula B.1. In practice, we want to have both negative and positive errors count towards the total loss. This is solved by squaring the error, as in Formula B.3.

$$SE(f, (\vec{x}_i, y_i)) = \quad (y_i - f(x_i))^2 = \quad (2.1 - 4.5)^2 = \quad 5.67 \tag{B.3}$$

In the last step, the squared errors are summed to get one score indicating the total loss, as can be seen in Equation B.1.

The ultimate goal, in this simplified case, is, given a training set S_{train} of $(\vec{x}_i, y)$ pairs, to find a model $f(x)$ that minimizes the SSE loss function (Equation B.4).

$$\arg\min_f \sum_{(\vec{x}_i, y) \in S_{\text{train}}} (y_i - f(\vec{x}_i))^2 \tag{B.4}$$

Before we can try to reach this goal, we need to know how to transform an arbitrary concept, like a person's credit card history, a car or any other tangible or intangible thing, into a list of numbers $\vec{x}$, a feature vector. We also need to know how we can transform this feature vector into a label y, by applying the model $f(\vec{x})$.

[1] I will also use $\hat{y}$ as an alias for $f(\vec{x})$ in this thesis. For a full list of used mathematical notation I refer to Appendix A.

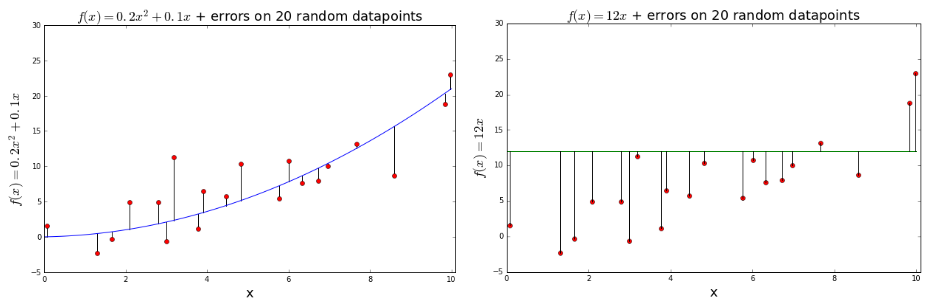


Figure B.1: An illustration of the SSE of two functions on 20 data points. The left function intuitively provides a better fit on the data, since the sum of the (squared) errors (indicated by the black lines) is smaller. Its SSE reflects this, with a score of 237.21. The right function, in contrast, does not fit the data well, it has an SSE of 1183.11 on the same 20 points.

B.1.2 Transforming Raw Data into Feature Vectors

In general, a feature vector consists of a list of facts (or features) about the concept that needs to be modeled, where each fact is in turn modeled as one or more numbers. These need to be numbers, because it makes it a lot easier to transform it into a label later on by the model, since we can use any number of mathematical operations and transformations.

There are a few straightforward ways to transform features into numbers.

When the feature is already a number, for example for prices and counts of sorts, we can take these numbers as is.

When the feature is categorical, that is: it can take on one of a number of predefined labels, it can be transformed by mapping each category to a unique number. For example, if the possible categories are {sedan, station wagon, SUV, cabrio}, the new categories could become {1, 2, 3, 4}.

Ordinal features, categorical features that have an inherent ordering, for example {low, medium, high}, are transformed in the same way as normal categorical features. However, the number mapping is chosen in such a way as to preserve the ordering.

Type of problem	Desired output	Type of feedback	Example
Classification	Class label ($y \in \{c_1, c_2, \cdots, c_n\}$)	Correct class label	Labeling an animal as a mammal or not.
Clustering	Class label ($y \in \{c_1, c_2, \cdots, c_n\}$)	None	Identifying new target groups in a database of customer data.
Regression	Number ($y \in \mathbb{R}$)	Correct real number	Predicting the price of a house in euros.

Table B.1: Different types of Machine Learning problems, with examples. The main focus of this thesis will be on supervised Machine Learning (classification and regression), although unlabeled data points will make an appearance.

Transforming text to a feature vector, however, is not as straightforward. One of the challenges is the inherent ordering of text. Changing the order of words in a sentence, or letters in a word, completely changes its meaning.

One easy way is to just ignore the ordering, which is done in the *bag-of-words* transformation. In this transformation, a collection of unique words is taken, called the dictionary. Each word is tied to a unique position in the feature vector representation. A raw text is transformed by setting each position to 1 whenever the dictionary word tied to that position is contained within the text, and leaving it at 0 otherwise.

We can make this transformation more advanced by letting more salient terms in the text have a higher weight, and less salient terms have a lower one, instead of using plain ones and zeros. For example by replacing them with tf-idf scores, as will be explained in Section 5.3.

Just as we need to transform the original concept into a vector of numbers, we need to transform its corresponding label y into a number as well.

This is generally done using the same transformation techniques as in the feature vector, depending on the type of label: categorical, number or ordinal.

However, the type of label is of vital importance of the models that can and should be used on the problem at hand and the possible ways to learn these models. When the goal is to label a data point with a class or category, the problem is commonly called *classification*. When the goal is to label it with a (continuous) number, it is called *regression* (see Table B.1). The SSE we use as an example is commonly used in regression.

B.1.3 Transforming the Feature Vector to a Prediction

Predictions are made by the model $f(\vec{x})$. This takes the feature vector x and transforms it into one number $\hat{y}$, the prediction.

A model generally consists of a vector of weights, embedded in a mathematical formula or algorithm. These weights are the numbers that are eventually learned. In a way, the mathematical formula itself controls the **form** of the model, whereas the weights control its **shape**.

One of the simplest models is the weighted linear model. This model introduces a weight for each position in the feature vector representation. To generate the label, each number in a feature vector is multiplied with its corresponding weight, and the results are summed, giving one predicted score $\hat{y}$ (see Equation B.5).

$$f(\vec{x}) = w_0 + \sum_{i=1} w_i \cdot f_i \tag{B.5}$$

The form of this model is always a straight line in two dimensions, a plane in three and a so called hyperplane in four dimensions and upwards. The weights control the position and slope of this line or plane.

B.1.4 Finding the Minimum or Maximum of an Objective Function

Now that we know the goal and the form of the model, the question becomes: how do we find the model that minimizes or maximizes the objective function?

There are a couple of ways, but many methods rely on the so called gradient of the function. The gradient tells us about the slope of the tangent line to each of the points in the function. This information is very useful, since the gradient naturally points towards a maximum in the function.

The gradient of a function can be computed by deriving another function from it, appropriately called the **derivative**. This is traditionally done by rewriting the original function by applying common rewriting rules.

Computing the derivative of the SSE, for example, is relatively straightforward.

We want to know how the gradient of the SSE changes on the basis of changes in the output of f, that is we want to know the derivative

$$\frac{d\,SSE(f, S_{\text{train}})}{d\,f}$$

The first derivation rule that we can apply is the **summation rule**. It reads as follows:

The derivative of a sum of functions is the sum of their derivatives.

$$\frac{d \sum g(x)}{d\,x} = \sum \frac{d\,g(x)}{d\,x}$$

In our case, the function $g(x)$ corresponds with the function $(y_i - f(\vec{x}_i))^2$

$$\frac{d \sum_{(\vec{x}_i, y_i) \in S_{\text{train}}} (y_i - f(\vec{x}_i))^2}{d\, f} = \sum_{(\vec{x}_i, y_i) \in S_{\text{train}}} \frac{d\, (y_i - f(\vec{x}_i))^2}{d\, f}$$

To find the derivative of $g(x)$ we can apply the **power rule**. This states that:

Multiply the term by its power and reduce the power by one.

$$\frac{d\ \cdot g(x)^k}{d\, x} = k \cdot g(x)^{(k-1)}$$

In practice we use a slightly different version of the SSE, to make the derivative a little bit simpler. By multiplying each squared error by 0.5, the derivative becomes:

$$\frac{d\, 0.5(y_i - f(\vec{x}_i))^2}{d\, f} = 0.5 \cdot 2 \cdot (y_i - f(\vec{x}_i))^1 = (y_i - f(\vec{x}_i))$$

Combining the summation and the power rule gives us this final derivative of the SSE:

$$\frac{d\, SSE(f, S_{\text{train}})}{d\, f} = \sum_{(\vec{x}_i, y_i) \in S_{\text{train}}} (y_i - f(\vec{x}_i)) \tag{B.6}$$

How do we use this derivative to find the minimum?

B.1.5 Gradient Descent

One intuitive manner to find the minimum or maximum of a function using its derivative is gradient descent.

Standard gradient descent simply starts at a value of the objective function for which to find the minimum, and starts to take small steps in the **opposite** direction of the output of the objective function's derivative[2]. Since the gradient always points towards a maximum, this is guaranteed to eventually reach a minimum.

This can be visualized as the function to be minimized as being a hill-scape of peaks and valleys, and gradient descent as dropping a ball somewhere above this landscape. The ball would move towards a valley due to gravity and eventually rest at a minimum of the function (see Figure B.2).

One challenge with gradient descent is choosing the right step size. A small step size means a low rate of convergence, but choosing a step size that is too large means that the algorithm can 'overshoot' the minimum when the value is close to it.

[2]For simplicity's sake I am assuming that the objective function is a loss function, and thus needs to be minimized, from now on. To find a maximum, instead of a minimum, the gradient descent can be easily adapted to take steps in the direction of the gradient instead.

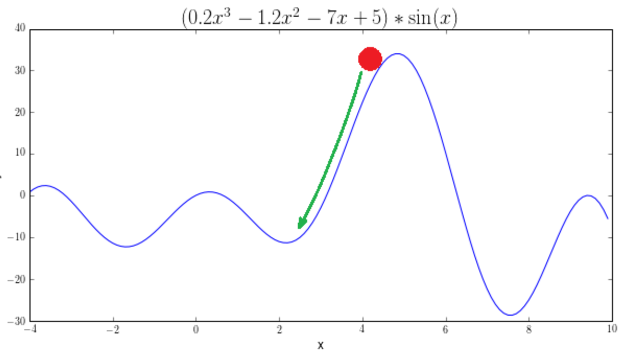


Figure B.2: A one-dimensional example of gradient descent. Note that while the red ball has found a minimum, this is not the global one, since the minimum around 7.5 has a lower value for y.

One solution is to choose a relatively large step size initially, but reduce it gradually as the algorithm progresses.

Another challenge is the fact that the found minimum is not necessarily the optimal one, depending on the initial model where the algorithm started, as can be seen in the example in Figure B.2.

This can be counteracted by choosing a particular objective function that has only one minimum, like the SSE. It can also be mitigated, by, for example, running gradient descent multiple times, each time starting at different initial models, and choosing the model with the lowest minimum after training.

Bibliography

[1] "A Future that Works: Automation, Employment and Productivity," tech. rep., McKinsey & Company, 2017.

[2] C. T. Haas, R. W. Glover, R. L. Tucker, and R. K. Terrien, "Impact of the internet on the recruitment of skilled labor," *Center for Construction Industry Studies, Report no. 17, University of Texas at Austin*, 2001.

[3] K. Rodenburg, "Using information extraction and evolutionary algorithms to improve matchmaking on the labor market," 2014.

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

[5] C. J. Burges, "From ranknet to lambdarank to lambdamart: An overview," *Learning*, vol. 11, pp. 23–581, 2010.

[6] O. Chapelle and M. Wu, "Gradient descent optimization of smoothed information retrieval metrics," *Information retrieval*, vol. 13, no. 3, pp. 216–235, 2010.

[7] R. Burke, "Hybrid recommender systems: Survey and experiments," *User modeling and user-adapted interaction*, vol. 12, no. 4, pp. 331–370, 2002.

[8] W. Hong, S. Zheng, H. Wang, and J. Shi, "A job recommender system based on user clustering.," *JCP*, vol. 8, no. 8, pp. 1960–1967, 2013.

[9] S. T. Al-Otaibi and M. Ykhlef, "A survey of job recommender systems," *International Journal of Physical Sciences*, vol. 7, no. 29, pp. 5127–5142, 2012.

[10] E. Faliagka, K. Ramantas, A. Tsakalidis, and G. Tzimas, "Application of machine learning algorithms to an online recruitment system," in *Proc. International Conference on Internet and Web Applications and Services*, 2012.

[11] J. W. Pennebaker, M. E. Francis, and R. J. Booth, "Linguistic inquiry and word count: Liwc 2001," *Mahway: Lawrence Erlbaum Associates*, vol. 71, no. 2001, p. 2001, 2001.

[12] A. van Belle, “Improving the match of jobs and profiles with learning-to-rank,” Jul 2016.

[13] V. Senthil Kumaran and A. Sankar, “Towards an automated system for intelligent screening of candidates for recruitment using ontology mapping (expert),” *International Journal of Metadata, Semantics and Ontologies*, vol. 8, no. 1, pp. 56–64, 2013.

[14] A. Singh, C. Rose, K. Visweswariah, V. Chenthamarakshan, and N. Kambhatla, “Prospect: a system for screening candidates for recruitment,” in *Proceedings of the 19th ACM international conference on Information and knowledge management*, pp. 659–668, ACM, 2010.

[15] A. Karatzoglou, L. Baltrunas, and Y. Shi, “Learning to rank for recommender systems,” in *Proceedings of the 7th ACM conference on Recommender systems*, pp. 493–494, ACM, 2013.

[16] N. Tax, S. Bockting, and D. Hiemstra, “A cross-benchmark comparison of 87 learning to rank methods,” *Information processing & management*, vol. 51, no. 6, pp. 757–772, 2015.

[17] P. Li, Q. Wu, and C. J. Burges, “Mcrank: Learning to rank using multiple classification and gradient boosting,” in *Advances in neural information processing systems*, pp. 897–904, 2007.

[18] R. Caruana and A. Niculescu-Mizil, “An empirical comparison of supervised learning algorithms,” in *Proceedings of the 23rd international conference on Machine learning*, pp. 161–168, ACM, 2006.

[19] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees.* CRC press, 1984.

[20] J. R. Quinlan, *C4. 5: programs for machine learning.* Elsevier, 2014.

[21] Y. Freund and R. E. Schapire, “A decision-theoretic generalization of on-line learning and an application to boosting,” in *European conference on computational learning theory*, pp. 23–37, Springer, 1995.

[22] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, “Learning to rank using gradient descent,” in *Proceedings of the 22nd international conference on Machine learning*, pp. 89–96, ACM, 2005.

[23] C. J. Burges, R. Ragno, and Q. V. Le, “Learning to rank with nonsmooth cost functions,” in *NIPS*, vol. 6, pp. 193–200, 2006.

[24] O. Chapelle and Y. Chang, “Yahoo! learning to rank challenge overview.,” in *Yahoo! Learning to Rank Challenge*, pp. 1–24, 2011.

[25] M. Taylor, J. Guiver, S. Robertson, and T. Minka, "Softrank: optimizing non-smooth rank metrics," in *Proceedings of the 2008 International Conference on Web Search and Data Mining*, pp. 77–86, ACM, 2008.

[26] Y. Yue, T. Finley, F. Radlinski, and T. Joachims, "A support vector method for optimizing average precision," in *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 271–278, ACM, 2007.

[27] T.-Y. Liu, J. Xu, T. Qin, W. Xiong, and H. Li, "Letor: Benchmark dataset for research on learning to rank for information retrieval," in *Proceedings of SIGIR 2007 workshop on learning to rank for information retrieval*, pp. 3–10, 2007.

[28] D. E. Appelt and B. Onyshkevych, "The common pattern specification language," in *Proceedings of a workshop on held at Baltimore, Maryland: October 13-15, 1998*, pp. 23–30, Association for Computational Linguistics, 1998.

[29] L. Chiticariu, R. Krishnamurthy, Y. Li, S. Raghavan, F. R. Reiss, and S. Vaithyanathan, "Systemt: an algebraic approach to declarative information extraction," in *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pp. 128–137, Association for Computational Linguistics, 2010.

[30] L. Chiticariu, Y. Li, and F. R. Reiss, "Rule-based information extraction is dead! long live rule-based information extraction systems!," in *EMNLP*, no. October, pp. 827–832, 2013.

[31] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer, "DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia," *Semantic Web Journal*, 2014.

[32] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, *et al.*, "Mllib: Machine learning in apache spark," *Journal of Machine Learning Research*, vol. 17, no. 34, pp. 1–7, 2016.

[33] M. D. Smucker, J. Allan, and B. Carterette, "A comparison of statistical significance tests for information retrieval evaluation," in *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pp. 623–632, ACM, 2007.

[34] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan, "Gate: an architecture for development of robust hlt applications," in *Proceedings of the 40th annual meeting on association for computational linguistics*, pp. 168–175, Association for Computational Linguistics, 2002.