Learning to Rank Improves IR in SE

David Binkley    Dawn Lawrie
Loyola University Maryland, Baltimore, MD 21210-2699, USA.
{binkley, lawrie}@cs.loyola.edu

Abstract—Learning to Rank (LtR) encompasses a class of machine learning techniques developed to automatically learn how to better rank the documents returned for an information retrieval (IR) search. Such techniques offer great promise to software engineers because they better adapt to the wider range of differences in the documents and queries seen in software corpora. To encourage the greater use of LtR in software maintenance and evolution research, this paper explores the value that LtR brings to two common maintenance problems: feature location and traceability. When compared to the worst, median, and best models identified from among hundreds of alternative models for performing feature location, LtR ubiquitously provides a statistically significant improvement in MAP, MRR, and MnDCG scores. Looking forward a further motivation for the use of LtR is its ability to enable the development of software specific retrieval models.

Keywords: Information Retrieval; Software Specific Retrieval

I. INTRODUCTION

Haiduc et al. [11] noted in their recent ICSE paper that there are more than twenty distinct software engineering (SE) tasks addressed using information retrieval (IR) techniques. Dang and Croft observe that traditional and more modern IR techniques are all based on a surprisingly small number of features such as term frequency, inverse document frequency, and document length [5]. Unfortunately, incorporating new features (such as page rank or proximity information) is often difficult, especially when it requires a change to the underlying model [5].

Learning to Rank (LtR) [3], [16] provides an excellent automated framework for combining features including query dependent features, such as the scores assigned to documents by existing search engines, and query independent features, such as page rank information. Within the SE domain, LtR provides a convenient mechanism to build effective software-specific retrieval models that, for example, exploit the semi-structure nature of source code.

As a machine learning technique, LtR requires a training phase. Once the model is trained, it can be applied to new queries and even new corpora. Thus the expense of computing the features and learning the final model is done once, off-line. As seen in the experiments, the resulting model can then be successfully used in a variety of settings including its application to previously unseen programs.

This paper considers the value that LtR brings to two key software maintenance tasks – feature location and traceability – by considering the following research questions:

- **RQ1**: What is the impact of Learning to Rank over the baseline models?
- **RQ2**: What is the stability of the learned models?

The first research question is the key question: does LtR work (i.e., does it improve retrieval)? A successful result finds LtR providing universal improvement. The significance of this outcome is that a developer need not wrestle with the configuration problem but can use LtR, to automatically derive a model that is better than the best configuration.

The importance of getting the configuration wrong was recently noted by Thomas et al. who observe that differing parameter settings have a considerable impact on performance [21]. This means that a developer attempting to configure a model could make a particularly good choice, but alas, could also make a particularly bad choice. To explore LtR’s expected improvement over this range both possibilities and a hypothetical median case are considered.

The second research question asks how robust the learned models are: how similar do the training examples have to be to the test examples? Specifically, how well does a model trained using one program (say a past project) perform when tested on a different program (say a new project)? Here a successful result finds that the learned models are robust against changes in the system to which they are applied. This stability is important. Many previous evaluations build and evaluate using the same system. This is unrealistic in the field where the answers (required for training) for the program a developer is working with likely do not exist. Stability allows an existing model (perhaps from a previous project) to be successfully applied to a new project without a significant loss of performance. This robustness is also important because it means that the computational effort that goes into feature generation and learning can be done once, off-line, as part of model construction.

When addressing the two research questions, the evaluation errs on the side of being excessive by including three standard IR performance measures: Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), and Mean normalized Discounted Continuous Gain (MnDCG). Doing so also provides three views of performance and thus supports a more complete picture of LtR’s impact.

The details of LtR are presented in the next section, followed by a description of the experimental design used to address the two research questions. Section IV presents the empirical results. Finally, related work, future work, and a summary appear in Sections V and VI.
II. LEARNING TO RANK

LtR is a class of machine learning (ML) algorithms designed to exploit the nature of a retrieval task [3]. The algorithms can learn a retrieval model by training from sets of queries and their relevant documents. What make these algorithms different from other ML algorithms is that they can learn from a ranked list. One of these algorithms, Coordinate Ascent [17], learns the weights for a linear combination of features. Here the features can include the output of various search engines as well as classic measures such as tf-idf. Coordinate Ascent iteratively optimizes a multivariate objective function by solving a series of one dimensional searches for the optimal weight of a particular feature (variable). It repeatedly cycles through each feature, holding all other features fixed.

The tool used in the experiments, RankLib [4], implements Coordinate Ascent, which is used in experiments at the recommendation of the tool’s author. RankLib takes as input the set of features and produces a retrieval model, which includes a weight for each feature. The tool supports cross validation, testing and training sets, as well as the ability to save a trained model to be used to evaluate performance on a completely different set of queries. In the experiments cross validation and the ability to train on one dataset and test on another are used to evaluate learned models.

III. EXPERIMENTAL DESIGN

This section begins by introducing the two tasks and the software projects used in the experiments. It then introduces the features employed and then briefly discusses the evaluation measures used.

A. Tasks

Tasks from two different domains are used in the experiments: feature location and traceability. They were selected because of the existence of pre-existing test collections, which helps avoid experimenter bias. This bias arises if an experimenter’s knowledge influences either the project selected or the particular queries used. Each test collection contains problem instances as well as the correct answers. From the perspective of IR, this means that there exists a set of queries and for each query a set of relevant documents.

1) Feature Location: This task encompasses the problem of locating the implementation of a feature within source code. The experiments reported herein focus on features described by maintenance requests [8].

2) Traceability: This task is the ability to link software artifacts, such as the source code and the stakeholder’s requirements. Traceability between software artifacts is critical to a rigorous software development process [7].

The six projects used in the experiment are shown in Table I. The first four are for feature location where queries come from bug reports and the relevance judgements are supplied at the method level [8]; therefore, the documents are methods. For traceability (the bottom two entries) queries are use-cases and the relevance judgements are supplied at the class level [7]; therefore, the documents are classes rather than methods as in the feature location collection.

B. Features

LtR requires a set of features. In these initial experiments, the features are the numeric scores produced using different search engine configurations. The three aspects of these configurations are described in this section.

1) Corpus Preparation: Corpus preparation begins by dividing the software project into documents. In this case the definition of a document is stipulated by the granularity of the relevance judgements. The next step is tokenization (i.e., separating identifiers into words). In the final step, a stoplist is applied to remove programming language keywords (e.g., while and if) and English stop words (e.g., is and the).

Three variants of tokenization are used. The least invasive, original source, leaves identifiers unchanged. The middle option, conservative split [9], separates multiword identifiers at an underscore, at a transition from lower to uppercase, and between alphabetic and numeric characters. The most invasive technique, Normalization, aggressively splits mixed case strings, same case strings, and strings including numeric characters [12]. It also replaces abbreviations and acronyms with the words that express the concept [12].

2) Query Preparation: The formulation of queries is an important step in the design of an experiment involving IR. As Haiduc et al. [11] show, the queries issued can have a dramatic impact on the results. LtR affords the opportunity to integrate multiple representations of a query as features for the learner so that selection of a single representation is not required; thus, eliminating the risk of making a bad choice.

Depending on the task, different query source options are available. For feature location both the supplied summary and detailed description are used. In addition, three variants are included in the experiments: both, keyword, and nouns. The first variant, both, is simply the composition of the summary and the description. The remaining two are query reduction schemes that include keywords obtained by asking humans, in this case the authors, to manually select useful words from the summary, and nouns, constructed by automatically selecting nouns from both.

<table>
<thead>
<tr>
<th>Project</th>
<th>Number of Methods</th>
<th>Size (LoC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML 0.22</td>
<td>12,448</td>
<td>289K</td>
</tr>
<tr>
<td>JabRef 2.6</td>
<td>4,741</td>
<td>117K</td>
</tr>
<tr>
<td>jEdit 4.3</td>
<td>7,118</td>
<td>177K</td>
</tr>
<tr>
<td>muCommander 0.8.5</td>
<td>8,538</td>
<td>171K</td>
</tr>
<tr>
<td>eTour SMOS</td>
<td>116</td>
<td>25K</td>
</tr>
<tr>
<td></td>
<td>99</td>
<td>17K</td>
</tr>
</tbody>
</table>

1) Private Communication, August 2013.
2) This is the number of documents identified while indexing the corpus and thus, is not guaranteed to agree with counts from other methods of preprocessing.
For traceability, there is only a single description in the form of a use case. In the experiments, this description and the nouns found in it are used as the two query formulations.

3) Retrieval Models: Given a corpus and a query, the third part of a configuration, a retrieval model, produces a score for each corpus document. Having multiple retrieval models enables the learning algorithm to combine the most informative configurations. In total seven different retrieval models are included: Vector Space Model (VSM) [14] with cosine similarity and formulated as a weighted sum, Latent Semantic Indexing (LSI) [6], and the query likelihood models (QL) [19] with three different smoothing techniques where one has two different parameter settings. Background on these models can be found in a companion TR [2].

C. Evaluation

This section first briefly introduces the three standard IR evaluation measures used to compare retrieval performance. Further background can be found in the literature [1], [14]. Each measure is the average over a collection of queries and seeks to ascertain the goodness of the ranked lists produced by a given technique. The first, Average Precision (AP), is the average of the precision values calculated at several positions in a ranked list. The particular positions of interest are where relevant documents occur. The Mean Average Precision (MAP) is the mean of the AP scores over a set of queries. The second, Reciprocal Rank (RR), is also a measure of precision. RR is the reciprocal rank of the top ranked relevant document. As a precision measure, this includes the set all documents up to the first relevant document in the ranked list. Mean Reciprocal Rank (MRR) is the mean of the RR scores over a set of queries. The third, Normalized Discounted Cumulative Gain (nDCG) observes that each new relevant document brings less value than those that came before it. For example, a second method from a relevant class is of less interest because the first already leads the developer to the relevant class. The formula can be found in Binkley and Lawrie [1]. MnDCG is the mean of the nDCG scores over a set of queries.

All three evaluation measures are used to assess the effectiveness of LtR. While it might be deemed excessive to use all three, each provides a different view of the results and thus together they provide a more complete picture. For example, when the discovery of any one relevant document is sufficient then MRR is the preferred choice. This occurs in tasks such as feature location where finding any relevant function quickly leads to other relevant parts of the code (e.g., via the call graph).

The results are analyzed using statistical methods to minimize the possibility that differences are due to chance factors. The Friedman test was used for head-to-head comparisons of two samples, and the Wilcoxon signed-rank test (using the Bonferroni correction for multiple comparisons) is used to compare more than two samples.

Finally Glass’s effect size is used to measure the magnitude of the difference between LtR and the comparison group. This measure is similar to Cohen’s D statistic; however Cohen’s D statistic assumes that the variance of each population is the same, which is not the case. Glass’s effect size does not make this assumption and is thus preferred. The effect size is considered small if greater than 0.2, medium if greater than 0.5, and large if greater than 0.8.

The standard evaluation practice in machine learning is to use k-fold cross validation, which first randomly divides a dataset into k subsets. The subsets are then used to build k different models, where k – 1 subsets are used for training and the remaining set is used for evaluation. In the experiments k = 5 is used, which in a side experiment worked well overall; thus, training uses four of the parts (80% of the data) while the 5th part is used for evaluation.

IV. RESULTS

The two research questions are designed to establish that LtR works (RQ1) and is robust (RQ2). First, to investigate the impact of LtR, RQ1 compares three of the baseline configurations (the features) with LtR. Such comparisons are the standard practice in explorations with LtR [16]. These three reflect an unfortunate user who selects the worst possible configuration, a hypothetical median choice, and a fortunate user who selects the best possible configuration.

It is important to keep in mind that the best configuration is program and query dependent, so there is a risk of choosing a particularly poor configuration, especially with code new to the developer. The same is true of tools that attempt to automate the selection [18] based on training data; they too risk using a poor choice when applied to code new to the tool. LtR can overcome this by combining multiple outputs (features).

An additional value that LtR brings is stability. This is the topic of the the second research question, which considers the viability of training a model with data from one project (say an older project) and then using this model with a different project (say a newly begun project). This question is especially important when training data is not readily available and costly to construct. The section concludes by considering threats to validity.

A. RQ1: The Impact of LtR

As is standard practice in explorations with LtR [16], the individual configurations used as features are used for comparison. Table II summarizes the search engine configurations considered. Consider a developer with no prior knowledge tasked with choosing a configuration. Such a developer might
make a particularly fortuitous choice or a particularly poor choice. To investigate RQ₁, these two possibilities and a hypothetical median case are considered. The median data is produced by taking the median score for each feature. Given the distribution of the data this is very close to the average. The median is preferred because each value is grounded in that it was actually produced by one of the configurations, something that is not true of the average.

Table III shows the results for RQ₁. From the table, it is clear that the configuration chosen has a significant impact on the result. For example the MRR values range from 0.035 to 0.273 when working with feature location.

Statistical analysis is necessary to determine if LtR brings improvement that is more than just a change in favor. For feature location and then traceability, the worst, median, and best models are compared with the results obtained using LtR. Starting with the feature-location problem, the top of Table IV presents the results of the statistical analysis for all three metrics. These results include the p-value, which indicates a significant difference in all nine comparisons, and the Glass effect size, which ranges from (very) large to small, and is always positive. Summarizing all nine comparisons, LtR provides a significant improvement for all three metrics. Thus a developer need not wrestle with the configuration problem but can use an automatic tool, based on LtR, to derive a model that is better than even the most fortuitous choice.

Turning to traceability, the results are shown in the bottom of Table IV. Here the results are weaker and effect sizes smaller. However, LtR yields a statistically significant improvement in six of the nine comparisons.

In summary for RQ₁, LtR leads to a statistically significant improvement in 15 of 18 cases where the learned model outperforms the best of the baseline configurations for feature location and is never worse for traceability. This is an encouraging result given that the LtR model is automatically learned from the baseline features and thus requires no particular expertise to configure a good model.

2. RQ₂: The Stability of the Learned Model

Turning to RQ₂, the stability of the learned models is investigated. The question here is are the learned models robust against changes in the code base. If they are then LtR can eliminate the risk of and the need for the clever effort-intensive feature selection. A preliminary comparison of training measures found that nDCG was the best training measure. While a more thorough presentation of the training phase is warranted, it is omitted in the interest of space. The stability of LtR is considered in this section using nDCG as the training measure.

Rather than using the more typically leave-one-out method, one might expect, the experiments take a far more demanding approach, building an LtR model using only one of the systems. The resulting model is then evaluated using all programs. The expected pattern here is that the best output will come when training and evaluating using the same program. For the two traceability programs, this expectation plays out as seen on the right of Figure 1: when training on SMOS, SMOS leads to the highest MnDCG value (this is the tallest bar shown in the graph). Likewise, for the model trained using eTour, eTour performs better (this is the rightmost bar).

It should be noted that training and testing on the same data is considered an unfair practice since the learner is specifically designed to perform well on that data. Therefore, it is extremely interesting that the expected pattern is not ubiquitous in the feature location data where ArgoUML produces models different from training on eTour and muCommander. This suggests that muCommander is a particularly good choice for training. On the other hand, looking at the performance of the models trained using JabRef (other than when evaluated using JabRef) indicates that JabRef is a poor choice of a training program. However, statistical analysis is necessary to find if these observations are significant.

With feature location applying the Wilcoxon signed-rank test that training on JabRef produces models different from training on ArgoUML and muCommander. In this case the JabRef-trained models are indeed inferior, perhaps because JabRef contains fewer than half the number of queries...
relative to the other projects. Finally, for traceability there was no significant difference based on the training program. Thus, in conclusion for RQ2, the stability of LtR falls into the “very good” range. The training program does have an influence, but with the six programs considered only one leads to statistically inferior results.

C. Threats to Validity

As with any empirical experiment, there are several standard threats to the validity of this analysis. Examples include the programs not being representative and the misinterpretation of the statistical models. A threat specific to the feature location experiment concerns the set of relevant documents, which was automatically mined by identifying changed documents between the time a modification request was created and the time it was marked completed. Clearly, there are other reasons that a method might have been changed. On the other hand, of the set of methods that would help a developer locate a concept, not all will require a change. Thus mining changed methods both under and over estimates the true relevant set.

V. RELATED WORK

There is related work in both the domain of IR and software engineering. In IR features lie at the very heart of the development of retrieval models. Term frequency and inverse document frequency form the core of most modern retrieval models. The only difference between many models is how the elementary features are combined [16]. Query reformulation provides an excellent source of features for LtR and is a rich area of current IR research, including techniques for query expansion, dependency analysis, query segmentation, and term selection [15].

In SE a multitude of applications of IR exist [11]. For feature location, Dit et al. provide an excellent systematic survey [8]. The work most closely related is that of Saha et al. [20], which uses the structure of the source code (e.g., methods headers, body, etc.) as features. LtR provides a more informed (and automated) way of integrating structured information. Turning to traceability, the work most similar to this is that of Gethers et al. where two retrieval models were combined heuristically using a weighted sum [10]. LtR can be thought of as a generalization of this idea to multiple features combined automatically using learned weights.

VI. SUMMARY AND FUTURE WORK

This paper demonstrates LtR works in a software engineering context. In addition to providing better answers, the approach is computationally feasible. For example, many of the retrieval models use the same underlying counts; therefore, the time required to respond to a new query using LtR is not vastly different from time for a single retrieval model given that looking up the values is much slower than combining the values using a linear expression. The experiments also show that LtR is externally robust, which means that the technique is widely applicable, even when given limited training data.

Learning to Rank provides an exciting platform on which to develop software specific retrieval models. Such models are advantageous because of observed differences in documents and queries seen in software corpora. LtR provides a great opportunity to exploit the difference between software engineering tasks and more traditional natural language IR search tasks.

Looking forward, this paper merely scratches the surface of LtR’s potential. To assess the initial feasibility of using LtR in SE, the features considered in this initial experiment are all different search engine configurations. Many more interesting possibilities exist including both query dependent and query independent features, which can include software specific information such a software metrics.

VII. ACKNOWLEDGEMENTS

Many thanks to Mattie Dziejma for his work on traceability. Our thanks to Van Dang for making his LtR software available (http://sourceforge.net/p/lemur/wiki/RankLib/).

REFERENCES


