1

# Learning to Rank Extract Method Refactoring Suggestions for Long Methods

Roman Haas[1] and Benjamin Hummel[2]

[1] Technical University of Munich, Lichtenbergstr. 8, Garching, Germany
roman.haas@tum.de
[2] CQSE GmbH, Lichtenbergstr. 8, Garching, Germany
hummel@cqse.eu

**Summary.** Extract method refactoring is a common way to shorten long methods in software development. It improves code readability, reduces complexity, and is one of the most frequently used refactorings. Nevertheless, sometimes developers refrain from applying it because identifying an appropriate set of statements that can be extracted into a new method is error-prone and time-consuming.

In a previous work, we presented a method that could be used to automatically derive extract method refactoring suggestions for long Java methods, that generated useful suggestions for developers. The approach relies on a scoring function that ranks all valid refactoring possibilities (that is, all *candidates*) to identify suitable candidates for an extract method refactoring that could be suggested to developers. Even though the evaluation has shown that the suggestions are useful for developers, there is a lack of understanding of the scoring function. In this paper, we present research on the single scoring features, and their importance for the ranking capability. In addition, we evaluate the ranking capability of the suggested scoring function, and derive a better and less complex one using learning to rank techniques.

**Key words:** Learning to Rank, Refactoring Suggestion, Extract Method Refactoring, Long Method

## 1.1 Introduction

A long method is a bad smell in software systems [2], and makes code harder to read, understand and test. A straight-forward way of shortening long methods is to extract parts of them into a new method. This procedure is called 'extract method refactoring', and is the most often used refactoring in practice [20].

The process of extracting a method can be partially automated by using modern development environments, such as Eclipse IDE or IntelliJ IDEA, that can put a set of extractable statements into a new method. However, developers still need to find this set of statements by themselves, which takes a considerable amount of time, and is error-prone. This is because even experienced developers sometimes select statements that cannot be extracted (for example, when several output parameters are required, but are not supported by the programming language) [12].

The refactoring process can be improved by suggesting to developers which statements could be extracted into a new method. The literature presents several approaches that can be used to find extract method refactorings. In a previous work, we suggested a method that could be used to automatically find good extract method refactoring candidates for long Java methods [3]. Our first prototype, which was derived from manual experiments on several open source systems, implemented a scoring function to rank refactoring candidates. The result of our evaluation has shown that this first prototype finds suggestions that are followed by experienced developers. The results of our first prototype have been implemented in an industrial software quality analysis tool.

*Problem statement.* The scoring function is an essential part of our approach to derive extract method refactoring suggestions for long methods. It is decisive for the quality of our suggestions, and also important for the complexity of the implementation of the refactoring suggester. However, it is currently unclear how good the scoring function actually performs in ranking refactoring suggestions and how much complexity will be needed to obtain useful suggestions. Therefore, in order to enhance our work, we need a deeper understanding of the scoring function.

*Contribution.* We do further research on the scoring function of our approach to derive extract method refactoring suggestions for long Java methods. We use learning to rank techniques in order to learn which features of the scoring function are relevant, to get meaningful refactoring suggestions, and to keep the scoring function as simple as possible. In addition, we evaluate the ranking performance of our previous scoring function, and compare it with the new scoring function that we learned. For the machine learning setting, we use 177 training and testing data sets that we obtained from 13 well-known open source systems by manually ranking five to nine randomly selected valid refactoring candidates.

In this paper, we show how we derived better extract method refactoring suggestions than in our previous work using learning to rank tools.

## 1.2 Fundamentals

We use learning to rank techniques to obtain a scoring function that is able to rank extract method refactoring candidates, and use normalized discounted cumulative gain (NDCG) metrics to evaluate the ranking performance. In this section, we explain the techniques, tools and metrics that we use in this paper.

### 1.2.1 Learning to Rank

Learning to rank refers to machine learning techniques for training the model in a ranking task [4].

There are several learning to rank approaches, where the pairwise and the listwise approach usually perform better than common pointwise regression approaches [8]. The pairwise approach learns by comparing two training objects and their given ranks ('ground truth'), whereas in our case the listwise approach learns from the list of all given rankings of refactoring suggestions for a long method. Liu et al. [8] pointed out that the pairwise and the listwise approaches usually perform better than the pointwise approach. Therefore, we do not rely on a pointwise approach but use pairwise and listwise learning to rank tools.

Qin et al. [15] constructed a benchmark collection for research on several learning to rank tools on the Learning To Rank (LETOR) data set. Their results support the hypothesis that pointwise approaches perform badly compared with pairwise and listwise approaches. In addition, listwise approaches often perform better than pairwise. However, *SVM-rank*, a pairwise learning to rank tool by Tsochantardis et al. [18], performs quite well and the first experiments on our data set showed that *SVM-rank* may lead us to interesting results. We set the parameter `-c` to 0.5 and the parameter `-#` to 5,000 as a trade-off between time consumption and learning performance.

Beside SVM-rank, we used a listwise learning to rank tool, *ListMLE* by Xia et al. [21]. In their evaluation, they showed that ListMLE performs better than ListNet by Cao et al. [1], which was also considered to be good by Qin et al.. Lan et al. [7] improved the learning capability of ListMLE, but did not provide binaries or source code; so we were unable to use the improved version.

ListMLE needs to be assigned a tolerance rate and a learning rate. In a series of experiments we performed, we found that the optimal ranking performance on our data set was with a tolerance rate of 0.001 and a learning rate of 1E-15.

### 1.2.2 Training and Testing

The learning process consisted of two steps: training and testing. We applied cross-validation [16] with 10 sets, that is, we split our learning data into 10 sets of (nearly) equal size. We performed 10 iterations using these sets, where nine of the sets were considered to be training data and one set was used as test data.

Test data is used to evaluate the ranking performance of the learned scoring function by comparing the grade of a refactoring candidate determined by the learned scoring function with its grade given by the learning data. We use NDCG metric to compare different scoring functions and their performances.

NDCG is the normalized form of the discounted cumulative gain (DCG), which is described in more detail by Järvelin and Kekäläinen [5], and measures the goodness of the ranking list (obtained by the application of the scoring function). Mistakes in the top-most ranks have a bigger impact on the DCG measure value. This is useful and important to us because we will not suggest all possible refactoring candidates, but only the highest-ranked ones. Given a long method, $m_i$, with refactoring candidates, $C_i$, suppose that $\pi_i$ is the ranking list on $C_i$ and $y_i$, the set of manually determined grades, then, the DCG at position $k$ is defined as $DCG(k) = \sum_{j:\pi_i(j)\leq k} G(j)D(\pi_i(j))$, where $G(\cdot)$ is an exponential gain function, $D(\cdot)$ is a position discount function, and $\pi_i(j)$ is the position of refactoring candidate, $c_{i,j}$, in $\pi_i$. We set $G(j) = 2^{y_{i,j}} - 1$ and $D(\pi_i(j)) = \frac{1}{\log_2(1+\pi_i(j))}$. To normalize the DCG, and to make it comparable with measures of other long methods, we divide this DCG by the DCG that a perfect ranking would have obtained. Therefore, the NDCG for a candidate ranking will always be in $[0, 1]$, where the NDCG of 1 can only be obtained by perfect rankings. In our evaluation, we consider the NDCG value of the last position so that all ranks are taken into account. See Hang [4] for further details.

## 1.3 Approach

We discuss our approach to improve the scoring function in order to find the best suggestions for extract method refactoring.

### 1.3.1 Extract Method Refactoring Candidates

In our previous work [3], we presented an approach to derive extract method refactoring suggestions automatically for long methods. The main steps are: generating valid extract method refactoring candidates, ranking the candidates, and pruning the candidate list.

In the following, a *refactoring candidate* is a sequence of statements that can be extracted from a method into a new method. The *remainder* is the method that contains all the statements from the original method after applying the refactoring, plus the call of the extracted method. The suggested refactorings will help to improve the readability of the code and reduce its complexity, because these are main reasons for developers to initiate code refactoring [6].

We derived refactoring candidates from the control and data flow graph of a method using the Continuous Quality Assessment Toolkit (ConQAT[3]) open source software. We filtered out all invalid candidates, that is those that violate preconditions that need to be fulfilled for extract method refactoring (for details, see [12]). The second step of our approach was to rank the valid

[3] `www.conqat.org`

candidates using a scoring function. Finally, we pruned the list of suggestions by filtering out very similar candidates, in order to obtain essentially different suggestions.

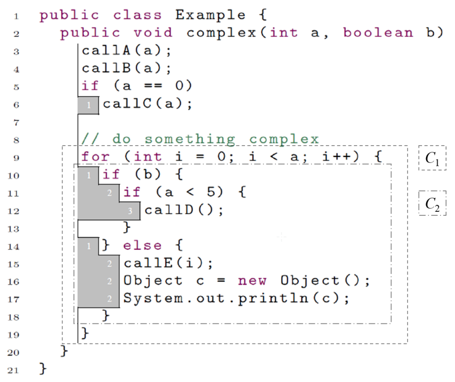
In the present paper, we focus on the ranking of candidates, and especially on the scoring function that defines that ranking.

### 1.3.2 Scoring Function

We aimed for an optimized scoring function that is capable of ranking extract method refactoring candidates, so that top-most ranked candidates are most likely to be chosen by developers for an extract method refactoring. The scoring function is a linear function that calculates the dot product of a coefficient vector, $c$, and a feature value vector, $f$, for each candidate. Candidates are arranged in decreasing order of their score.

In this paper, we use a basis of 20 features for the scoring function. In the following, we give a short overview about the features. There are three categories of feature: complexity-related features, parameters, and structural information.

We illustrate the feature values with reference to two example refactoring candidates ($C_1$ and $C_2$) that were chosen from the example method given in Figure 1.1. The gray area shows the nesting area, which is defined below. The white numbers specify the nesting depth of the corresponding statement.



```
 1  public class Example {
 2    public void complex(int a, boolean b) {
 3      callA(a);
 4      callB(a);
 5      if (a == 0)
 6        callC(a);
 7
 8      // do something complex
 9      for (int i = 0; i < a; i++) {
10        if (b) {
11          if (a < 5) {
12            callD();
13          }
14        } else {
15          callE(i);
16          Object c = new Object();
17          System.out.println(c);
18        }
19      }
20    }
21  }
```

Fig. 1.1: Example Method with Nesting Area of Statements And Example Candidates

| # | Feature | Type | $C_1$ ll. 9 − 19 | $C_2$ ll. 10 − 18 |
|---|---|---|---|---|
| 1 | LoC Red (abs) | int | 8 | 8 |
| 2 | Token Red (abs) | int | 33 | 43 |
| 3 | Stmt Red (abs) | int | 5 | 6 |
| 4 | LoC Red (rel) | double | 0.42 | 0.42 |
| 5 | Token Red (rel) | double | 0.36 | 0.47 |
| 6 | Stmt Red (rel) | double | 0.38 | 0.46 |
| 7 | Nest Depth Red | int | 0 | 1 |
| 8 | Nest Area Red | int | 1 | 6 |
| 9 | # Read Var | int | 4 | 4 |
| 10 | # Written Var | int | 1 | 1 |
| 11 | # Used Var | int | 4 | 4 |
| 12 | # Input Param | int | 2 | 3 |
| 13 | # Output Param | int | 0 | 0 |
| 14 | ∃ Introd Com | bool | 1 | 0 |
| 15 | # Introd Com | int | 2 | 0 |
| 16 | ∃ Concl Com | bool | 0 | 0 |
| 17 | # Concl Com | int | 0 | 0 |
| 18 | Same T Before | bool | 0 | 0 |
| 19 | Same T After | bool | 0 | 0 |
| 20 | # Branch Stmt | int | 3 | 2 |

Table 1.1: Features and Values in Example

*Complexity-related features*

We mainly focused on reducing complexity and increasing readability. For complexity indicators, we used length, nesting and data flow information. For length-related features, we implemented six different metrics to measure the reduction of the method length (with respect to the longest method after the refactoring). We considered length based on the number of lines of code (LoC), on the number of tokens, and on the number of statements – all of them as both absolute values and relative to the original method length.

We consider highly nested methods as more complex than moderately nested ones, and use two features to represent the reduction of nesting: reduction of nesting depth and reduction of nesting area. The nesting area of a method with statements $S_1$ to $S_n$, each having a nesting depth of $d_{S_i}$, is defined to be $\sum_{i=1}^{n} d_{S_i}$. The idea of nesting area comes from the area alongside the single statements of pretty printed code (see the gray areas in Figure 1.1).

Dataflow information can also indicate complexity. We have features representing the number variables that are read, written or read and written.

*Parameters*

We considered the number of input and output parameters as an indicator of data coupling between the original and the extracted methods, which we want to keep low using our suggestions. The more parameters that are needed for a set of statements to be extracted from a method, the more the statements will depend on the rest of the original method.

*Structural information*

Finally, we have some features that represent structural aspects of the code. A design principle for code is that methods should process only one thing [9]. Methods that follow this principle are easier to understand. As developers often put blank lines or comments between blocks of code that process something else, we use features representing the existence and the number of blank or commented lines at their beginning, or at their end. Additionally, for first statement of the candidate, we check to see whether the type of the preceding is the same; and for the last statement, we check to see whether the type of the following statement is the same. Our last feature considers a structural complexity indicator – the number of branching statements in the candidate.

### 1.3.3 Training and Test Data Generation

To be able to learn a scoring function, we need training and test data. We derived this data by manually ranking approximately 1,000 extract method refactoring suggestions. To obtain this learning data, we selected 13 Java open source systems from various domains, and of different sizes. We consider a method to be 'long' if it has more than 40 LoC. From each project we randomly selected 15 long methods. For each method, we randomly selected valid refactoring candidates, where the number of candidates depended on the method length.

Our approach seeks to find suggestions that do not introduce new smells into the code. Therefore, in the pruning step of our approach, we usually filter out candidates that need more than three input parameters, thus avoiding the 'long parameter list' mentioned by Fowler [2]. To avoid learning that too many input parameters are bad, we considered only candidates that had less than four input parameters.

We ranked the selected candidates manually with respect to complexity reduction and readability improvement. The higher the ranking we gave a candidate, the better the suggestion was for us.

Some of the randomly selected methods were not suitable for an extract method refactoring. That was most commonly the case when the code would not benefit from the extract method, but from other refactorings. In addition, for some methods, we could not derive a meaningful ranking because there were only very weak candidates. That is why we did not use 18 of the 195 randomly selected long methods to learn our scoring function.[4]

## 1.4 Evaluation

In this section, we present and evaluate the results from the learning procedure.

### 1.4.1 Research Questions

**RQ1: What are the results of the learning tools?** In order to get a scoring function that is capable of ranking the extract method refactoring candidates, we decided to use two learning to rank tools that implement different approaches, and that had performed well in previous studies.

**RQ2: How stable are the learned scoring functions?** To be able to derive implications for a real-world scoring function, the coefficients of the learned scoring function should not vary a lot during the 10-fold cross evaluation procedure.

**RQ3: Can the scoring function be simplified?** For practical reasons, it is useful to have a scoring function with a limited number of features. Additionally, reducing the search space may increase the performance of the learning to rank tools – resulting in better scoring functions.

**RQ4: How does the learned scoring function compare with our manually determined one?** In our previous work, we derived a scoring function by manual experiments. Now we can use our learning data set to evaluate the ranking performance of the previously defined scoring function, and to compare it with the learned one.

[4] On `http://in.tum.de/~haas/l2r_emrc_data.zip` we provide our rankings and the corresponding code bases from which we generated the refactoring candidates.

### 1.4.2 Study Setup

To answer RQ1 and RQ2, we used the learning to rank tools SVM-rank and ListMLE to perform a 10-fold cross validation on our training and test data set of 177 long methods, and a total of 1,185 refactoring candidates. We illustrate the stability of the single coefficients by using box plots that show how the coefficients are distributed over the ten iterations of the 10-fold cross validation.

To answer RQ3, we simplified the learned scoring function by omitting features, where the selection criterion for the omitted features is preservation of the ranking capability of the scoring function. Our initial feature set contained six different measures of length. For the sake of simplicity, we would like to have only one measure of length in our scoring function. To find out which measure best fits in with our training set, we re-ran the validation procedure (again using ListMLE and SVM-rank), but this time with only one length measurement, using each of the length measurements one at a time. We continued with the feature set reduction until only one feature was left.

### 1.4.3 Results

The following paragraphs answer the research questions.

*RQ1: What are the results of the learning tools?*

Figures 1.2 and 1.3 show the results of the 10-fold cross validation for ListMLE and for SVM-rank, respectively. For each single feature, $i$, there is a box plot of the corresponding coefficient, $c_i$.

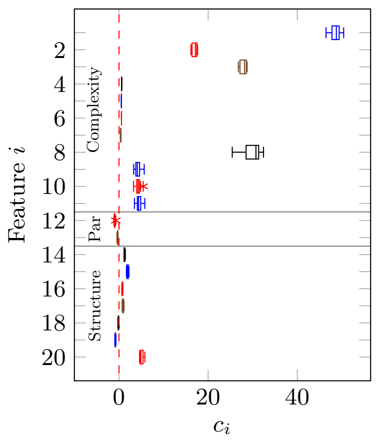


Fig. 1.2: Learning Result From ListMLE With All Features

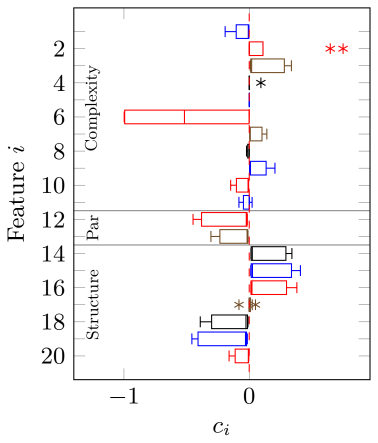


Fig. 1.3: Learning Result From SVM-rank With All Features

The average NDCG values of the learned scoring function for ListMLE is 0.873, whereas for SVM-rank it is 0.790. Therefore, the scoring function found by ListMLE performed better than the scoring function found by SVM-rank.

Table 1.2: Coefficients of Variation for Learned Coefficients

| | ListMLE | SVM-rank |
|---|---|---|
| \| AVG CV \| | 0.0087 | 22.522 |
| \| Min CV \| | 0.0053 | 0.8970 |
| \| Max CV \| | 0.5767 | 451.2 |

*RQ2: How stable are the learned scoring functions?*

Table 1.2 shows the average, minimum and maximum coefficients of variation (CV) for the learned coefficients for ListMLE and for SVM-rank. Small CVs indicate that in relative terms the results from the single runs in the 10-cross fold procedure did not vary a lot, whereas big CVs indicate big differences between the learned coefficients. As the CVs of the single features from ListMLE are much smaller than those of SVM-rank, the coefficients of ListMLE are much more stable compared with SVM-rank. SVM-rank shows coefficients with a big variance between the single iterations of the validation process; that is, despite the heavy overlapping of the training sets, the learned coefficients vary a lot and can hardly be generalized.

*RQ3: Can the scoring function be simplified?*

Figure 1.4 shows a plot of the averaged NDCG measure for all 12 runs. Remember that we actually had three length measures, and we considered the absolute and the relative values for all of them. As the reduction of the number of statements led to a higher NDCG for ListMLE (which outperformed SVM-rank with respect to NDCG), we chose to use it as our length measure. In practice, that seems sensible since, while LoC also count empty and commented lines, the number of statements only counts real code.

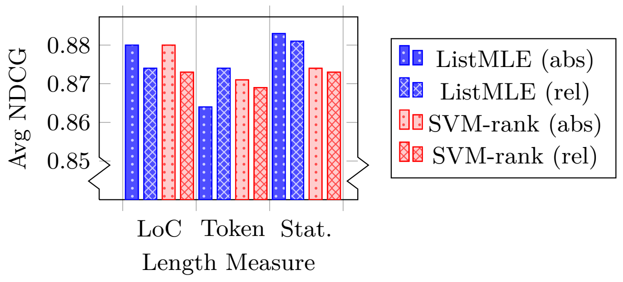


Fig. 1.4: Averaged NDCG When Considering Only One Length Measure

We iteratively identified a set of features that had no or only small influence on the ranking performance and removed it in the next iteration. A scoring function that only considered the number of input parameters and length and nesting area reduction still had an average NDCG of 0.885.

*RQ4: How does the learned scoring function compare with our manually determined one?*

The scoring function that we presented in [3] achieved a NDCG of 0.891, which is better than the best scoring function learned in this evaluation.

### 1.4.4 Discussion

Our results show that, in the initial run of the learning to rank tools, features indicating a reduction of complexity are much more relevant for the ranking, and therefore have a comparatively high impact. Furthermore, the stability of ListMLE is higher on our data set than the stability of SVM-rank. For SVM-rank there is a big variance in the learned coefficients, which might also be a reason for the comparatively lower performance measure values.

The results for RQ3 show that it is possible to achieve a great simplification without big reductions in the ranking performance. The biggest influences on the ranking performance were the reduction of the number of statements, the reduction of nesting area (both are complexity indicators), and the number of input parameters.

*Manual improvement* As already mentioned, the learned scoring functions did not outperform the manually determined scoring function from our previous work. Obviously, the learning tools were not able to find optimal coefficients for the features. To improve the scoring function from our previous work, we did manual experiments that were influenced by the results of ListMLE and SVM-rank, and evaluated the results using the whole learning data set.

We were able to find several scoring functions that had only a handful of features and a better ranking performance than our scoring function from previous work (column 'Previous' in Table 1.3). In addition to the three most important features that we obtained in the answer to RQ3 (features #3, #7, #10), we also took the comment features (#14-17) into consideration. The main differences between the previous scoring function and the manually improved one from this paper are the length reduction measure, the omission of nesting depth, and the number of output parameters.

By taking the results of ListMLE and SVM-rank into consideration, we were able to find a coefficient vector such that the scoring function achieved a NDCG of 0.894 (see Table 1.3). That means that we were able to find a better scoring function when we combined the findings of our previous work with the learned coefficients from this paper.

Table 1.3: Best Scoring Functions

| # | Feature \ Fct | Previous | Learned | Improved |
|---|---|---|---|---|
| 1 | LoC (abs) | 0.036 | - | - |
| 3 | Stmt (abs) | - | 0.681 | 0.066 |
| 7 | Nesting Depth | 0.362 | - | - |
| 8 | Nesting Area | 0.724 | 0.731 | 0.895 |
| 12 | # Input P. | -0.362 | -0.024 | -0.331 |
| 13 | # Output P. | -0.362 | - | - |
| 14 | ∃ Introd Com. | 0.181 | - | 0.166 |
| 15 | # Introd Com. | 0.181 | - | 0.166 |
| 16 | ∃ Concl Com. | 0.090 | - | 0.166 |
| 17 | # Concl Com. | 0.090 | - | 0.052 |
| | AVG NDCG | 0.891 | 0.885 | 0.894 |

## 1.5 Threats to Validity

Learning from data sources that are either too similar or too small means that there is a chance that no generalization of the results is possible. To have enough data to enable us to learn a scoring function that can rank extract method refactoring candidates, we chose 13 Java open source projects from various domains and from each project we randomly selected 15 long methods. We manually reviewed the long methods, and filtered out those that were not appropriate for the extract method. From the 177 remaining long methods, we randomly chose five to nine valid refactoring suggestions, depending on the method length. We ensured that our learning data did not contain any code clones to avoid learning from redundant data.

The manual ranking was performed by a single individual, which is a threat to validity since there is no commonly agreed way on how to shorten a long method, and therefore no single ranking criterion exists. The ranking was done very carefully, with the aim of reducing the complexity and increasing the readability and understandability of the code as much as possible; so, the scoring function should provide a ranking such that we can make further refactoring suggestions with the same aim.

We relied on two learning to rank tools, which represents another threat to validity. The learned scoring functions heavily depend on the tool. As the learned scoring functions vary, it is necessary to have an independent way of evaluating the ranking performance of the learned scoring functions. We used the widely used measure NDCG to evaluate the scoring functions, and applied a 10-fold cross validation procedure to obtain a meaningful evaluation of the ranking performance of the learned scoring function.

A threat to external validity is the fact that we derived our learning data from 13 open source Java systems. Therefore, results are not necessarily generalizable.

## 1.6 Related Work

In our previous work [3], we presented an automatic approach to derive extract method refactoring suggestions for long methods. We obtained valid refactoring candidates from the control and dataflow graph of a long method. All valid refactoring candidates were ranked by a manually-determined scoring function that aims to reduce code complexity and increase readability. In the present work, we have put the scoring function on more solid ground by learning a scoring function from many long methods, and manually ranked refactoring suggestions.

In the literature, there are several approaches that learn to suggest the most beneficial refactorings – usually for code clones. Wang and Godfrey [19] propose an automated approach to recommend clones for refactoring by training a decision-tree based classifier, C4.5. They use 15 features for decision-tree model training, where four consider the cloning relationship, four the context of the clone, and seven relate to the code of the clone. In the present paper, we have used a similar approach, but with a different aim: instead of clones, we have focused on long methods.

Mondal et al. [10] rank clones for refactoring through mining association rules. Their idea is that clones that are often changed together to maintain a similar functionality are worthy candidates for refactoring. Their prototype tool, *MARC*, identifies clones that are often changed together in a similar way, and mines association rules among these. A major result of their evaluation on thirteen software systems is that clones that are highly ranked by MARC are important refactoring possibilities. We used learning to rank techniques to find a scoring function that is capable of ranking extract method refactoring candidates from long methods.

## 1.7 Conclusion and Future Work

In this paper, we have presented an approach to derive a scoring function that is able to rank extract method refactoring suggestions by applying learning to rank tools. The scoring function can be used to automatically rank extract method refactoring candidates, and thus present a set of best refactoring suggestions to developers. The resulting scoring function needs less parameters than previous scoring functions but has a better ranking performance.

In the future, we would like to suggest sets of refactorings, especially those that remove clones from the code.

We would also like to find out whether the scoring function provides good suggestions for object-oriented programming languages other than Java and whether other features need to be considered in that case.

## Acknowledgments

Thanks to the anonymous reviewers for their helpful feedback. This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant ”Q-Effekt, 01IS15003A”. The responsibility for this article lies with the authors.

## References

1. Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li. Learning to rank: From pairwise approach to listwise approach. In *24th ICML*, 2007.
2. M. Fowler. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley object technology series. Addison-Wesley, Reading, PA, 1999.
3. R. Haas and B. Hummel. Deriving extract method refactoring suggestions for long methods. In *SWQD*, 2016.
4. L. Hang. A short introduction to learning to rank. *IEICE Transactions on Information and Systems*, 94(10):1854–1862, 2011.
5. K. Järvelin and J. Kekäläinen. Ir evaluation methods for retrieving highly relevant documents. In *23rd SIGIR*, 2000.
6. M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *20th International Symposium on the FSE*, 2012.
7. Y. Lan, Y. Zhu, J. Guo, S. Niu, and X. Cheng. Position-aware listmle: A sequential learning process for ranking. In *30th Conference on UAI*, 2014.
8. T.-Y. Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.
9. R. C. Martin. *Clean Code : A Handbook of Agile Software Craftsmanship*. Robert C. Martin series. Prentice Hall, Upper Saddle River, NJ, 2009.
10. M. Mondal, C. K. Roy, and K. Schneider. Automatic ranking of clones for refactoring through mining association rules. In *CSMR-WCRE*, 2014.
11. E. Murphy-Hill and A. P. Black. Why don't people use refactoring tools? In *1st WRT*, 2007.
12. E. Murphy-Hill and A. P. Black. Breaking the barriers to successful refactoring: Observations and tools for extract method. In *30th ICSE*, 2008.
13. W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
14. A. Ouni. *A Mono-and Multi-objective Approach for Recommending Software Refactoring*. PhD thesis, Université de Montréal, 2015.
15. T. Qin, T.-Y. Liu, J. Xu, and H. Li. Letor: A benchmark collection for research on learning to rank for information retrieval. *Information Retrieval*, 13(4):346–374, 2010.
16. C. Sammut, editor. *Encyclopedia of machine learning*. Springer, New York, 2011.
17. N. Tsantalis and A. Chatzigeorgiou. Ranking refactoring suggestions based on historical volatility. In *15th ECSMR*, 2011.
18. I. Tsochantaridis, T. Joachims, T. Hofmann, and Y. Altun. Large margin methods for structured and interdependent output variables. *Journal of Machine Learning Research*, 6:1453–1484, 2005.
19. W. Wang and M. W. Godfrey. Recommending clones for refactoring using design, context, and history. In *ICSME*, 2014.
20. D. Wilking, U. F. Kahn, and S. Kowalewski. An empirical evaluation of refactoring. *e-Informatica*, 1(1):27–42, 2007.
21. F. Xia, T.-Y. Liu, J. Wang, W. Zhang, and H. Li. Listwise approach to learning to rank: Theory and algorithm. In *25th ICML*, 2008.