Addressing Scalability in API Method Call Analytics

Ervina Cergani, Sebastian Proksch, Sarah Nadi*, Mira Mezini
Software Technology Group
Technische Universität Darmstadt, Germany
{cergani, proksch, nadi, mezini}@st.informatik.tu-darmstadt.de trovato@corporation.com

ABSTRACT

Intelligent code completion recommends relevant code to developers by comparing the editor content to code patterns extracted by analyzing large repositories. However, with the vast amount of data available in such repositories, scalability of the recommender system becomes an issue. We propose using Boolean Matrix Factorization (BMF) as a clustering technique for analyzing code in order to improve scalability of the underlying models. We compare model size, inference speed, and prediction quality of an intelligent method call completion engine built on top of canopy clustering versus one built on top of BMF. Our results show that BMF reduces model size up to 80% and increases inference speed up to 78%, without significant change in prediction quality.

CCS Concepts

•Software and its engineering → Software libraries and repositories; •Information systems → Data analytics; Data mining; •Computing methodologies → Factorization methods;

Keywords

Intelligent Method Call Completion, Analytics of code repositories, Boolean Matrix Factorization, Scalability

1. INTRODUCTION

Code completion is incorporated in many modern Integrated Development Environments (IDEs). Specifically, method call completion is heavily used by developers to decide which method to call next given the current context [13]. While traditional code completion systems only exploit the type system and propose an alphabetically sorted list of all possible methods, *intelligent code completion systems* propose relevant method calls by comparing the editor content to code patterns extracted by analyzing large repositories.

*Current affiliation: Department of Computing Science, University of Alberta, Canada, nadi@ualberta.ca

SWAN'16, November 13, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4395-4/16/11...
http://dx.doi.org/10.1145/2989238.2989240

To be accurate, and thus useful, intelligent code completion systems need to mine large numbers of code repositories to increase the probability that the detected patterns are indeed relevant for developers. In our previous work [16], we found that this results in increased model sizes for the recommender systems. Large model sizes require more main memory to be loaded and slow down querying time, limiting the usefulness of recommender systems in practice.

In this paper, we investigate *Boolean Matrix Factorization* (BMF) as a means to create smaller models. BMF is a well-known machine learning approach used to represent big data sets through smaller dimensions, while at the same time removing noise [12]. Matrix Factorization (MF) has already been shown to perform well for recommender systems used by Amazon and Netflix [6]. We want to bring the same benefits to recommender systems for software engineering to deal with increasing amounts of data. Specifically, we investigate if BMF can be used to improve analytics of code repositories in the context of intelligent method call completion.

To evaluate the effect of using BMF, we use the *MDL4BMF* algorithm developed by Miettinen et al. [12], which automatically calculates the factorization rank (number of clusters in clustering terminology) by using the *Minimum Description Length* (MDL) principle. However, we adapt *MDL4BMF* to the code completion problem by implementing a heuristic on top of it (see Section 3.1).

We evaluate our approach on the SWT framework APIs in the code of 3,186 plug-ins obtained from the Eclipse Kepler update site. We compare prediction quality, model size, and inference speed of BMF to those of our previous intelligent method call completion recommender that uses canopy clustering. Our evaluations shows that BMF reduces the model size by up to 80% and makes inference speed up to 78% faster, with no significant effect on prediction quality. Based on these results, we conclude that BMF is promising in the context of intelligent method call completion and speculate that other software engineering applications that rely on large amounts of input data, may also benefit from such an approach. To summarize, our contributions are:

- Application of BMF in the context of intelligent method call completion to analyze code repositories.
- Implementation of a heuristic on top of BMF to further improve pattern detection for method call completion.
- Integration of BMF into an existing intelligent code completion engine.
- Investigation of the effect of BMF on scalability of recommendation models.
- A systematic empirical evaluation of the effect of using BMF for intelligent method call completion.

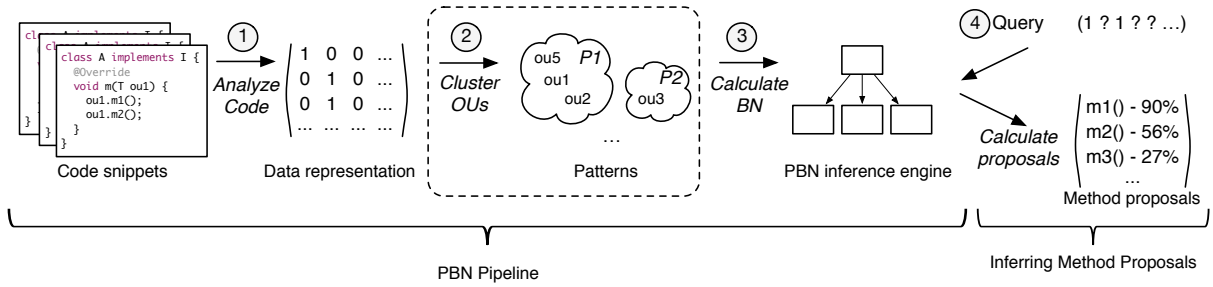

Figure 1: Pattern-based Bayesian Network Pipeline, where clustering (dashed frame) is exchanged with BMF

2. BACKGROUND & MOTIVATION

This paper focuses on method call completion. In other words, the developer knows the object type she needs but has to decide which method to call next. Previous work that focused on improving code completion often used only parts of the context information available such as the type of the receiver object, the set of already performed calls on the receiver, and the enclosing method definition [2, 5, 18]. In our previous work [16], we showed that using additional context information such as definition sites, parameter call sites, and class context does improve the prediction quality (i.e., the F_1-measure). While we introduced *Pattern-based Bayesian Networks* (PBN) and used canopy clustering to allow us to better handle the increased amount of input data, we found that using the additional contextual information nearly doubled the model size forcing us to consider the tradeoffs between adding more useful contextual information and the increase in model size. We advocated for more intelligent machine learning algorithms that can further reduce the model size to allow using more context information.

This paper proposes using Boolean Matrix Factorization (BMF) as a means of building smaller models. We build BMF on top of the PBN pipeline, shown in Figure 1, where we replace the dash framed part with BMF (see Section 3).

2.1 PBN Pipeline

The Pattern-based Bayesian Network (PBN) is an extensible inference engine for intelligent method call completion. It is structured as a four step pipeline shown in Figure 1.

Step 1: Analyze code repositories. The input data for PBN is generated by statically analyzing code repositories for object usages. An *object usage* is an abstract representation of an example usage of a specific instance of an API type. It contains different *features* that describe the specific usage, such as the method calls invoked on the given instance, the enclosing class and method context, and the definition site.

Figure 2 shows five code snippets that we use as a running example throughout the paper. They are analyzed to collect information for the object type `T`. The pipeline is executed separately for each object type in the API for which a recommender is built. The first object usage *ou1* has a method context `I.m` and two receiver call sites `m1` and `m2`. Note that the method context always points to the type in the hierarchy in which the method signature is defined first. Therefore, it is `I.m` and not `A.m`. This is a generalization that may lead to replications of the same object usage. For example, *ou5* of type T is observed in class *A2* that implements *I*. It contains the same combination of method invocations as *ou1*, and will thus have the same binary vector as *ou1*.

Based on such analysis, code snippets are transformed into the matrix format shown in Table 1, where columns represent the set of all features that appear in the code (the *feature space*). While all feature kinds (*method calls*, *method context*, *class context*, *definition site*, and *receiver call site*) are considered, the examples in this paper are reduced to method calls and method context for easier illustration.

Table 1: Object Usages represented in the feature space

	in: I.m	in: J.n	in: K.o	call: m1	call: m2	call: m3	call: m4
ou1	1	0	0	1	1	0	0
ou2	0	1	0	1	1	0	1
ou3	0	1	0	0	1	1	0
ou4	0	0	1	0	0	0	1
(ou5)	1	0	0	1	1	0	0

Step 2: Identify patterns. The matrix generated in Step 1 is passed to the clustering component that looks for similar vectors (object usages) that can be grouped into patterns. The clustering component is exchangeable and represents the extension point that is addressed in this work. The output of the clustering step is a list of patterns. The original pipeline uses canopy clustering, which identifies three patterns from the example in Table 1: `P1` contains `ou1`, `ou2`, and `ou5`; `P2` contains `ou3`; and `P3` contains `ou4`.

Step 3: Calculate Bayesian Network (BN). The patterns from Step 2 are used to create the network where we assign a probability of occurrence to each pattern and to each feature-space dimension within a pattern. The probability of a pattern `P` is calculated as follows:

$$p(P) = \frac{n_p}{n_{total}} \tag{1}$$

where n_p is the number of object usages in `P`, and n_{total} is the total number of object usages for the given API type.

The probability of a feature f in a given pattern `P`, where n_f is the number of object usages in P that contain f is:

$$p(f|P) = \frac{n_f}{n_p} \tag{2}$$

The root node in the Bayesian Network contains all identified patterns with corresponding probabilities. The other nodes contain the different features with the corresponding probabilities within each pattern.

Step 4: Query the BN. When a *query* is provided to the recommender (top right of Figure 1), the constructed BN is used to infer method proposals. A query is an object usage extracted from the source code under edit. All observed information is set as evidence in the network and the probability for all remaining methods (unobserved features shown as question marks) is calculated. The output is a list

```
class A implements I {
  @Override
  void m(T ou1) {
    ou1.m1();
    ou1.m2();
  }
}
```

```
class B implements J {
  @Override
  void n(T ou2) {
    ou2.m1();
    ou2.m2();
    ou2.m4();
  }
}
```

```
class C implements J {
  @Override
  void n(T ou3) {
    ou3.m2();
    ou3.m3();
  }
}
```

```
class D implements K {
  @Override
  void o(T ou4) {
    ou4.m4();
  }
}
```

```
class A2 implements I {
  @Override
  void m(T ou5) {
    ou5.m1();
    ou5.m2();
  }
}
```

Figure 2: Code snippet examples from code repositories

of method calls with an assigned probability. Only methods with a probability higher than 30% are proposed to the user. While this threshold is configurable, we follow previous work and select the same threshold for comparability.

2.2 Motivation and Problem Statement

We selected the top five frameworks (according to the number of object usages) in the Eclipse plug-in dataset. Figure 3 shows the model sizes obtained by PBN for all API types in the selected frameworks. As the number of object usages available for a type increases, the model size linearly increases. With the increasing number of available code repositories that can be mined, a single API type can have more than 100,000 usages. A quick search for `org.eclipse.swt.widgets.Composite` (a framework-specific type) and `java.util.ArrayList` (a core Java library type) on Github returns over 220,000 and 750,000 files, respectively. If we assume each of these files contains only one object usage of the respective types and extrapolate on the shown graph, the model size for a single type would reach 75MB. A recommender system should be able to support hundreds of types. If only 100 API types would be loaded simultaneously, the model sizes could sum up to 7.5 GB.

One problem with such large model sizes is that bloated models can greatly slow down the querying time. Additionally, models need to be loaded from the hard drive and deserialized, before they can be used in the IDE. This *startup delay* can be avoided by caching loaded models. Note that a recommender may need to load multiple models in-memory to instantly support different API types that developers may use in an IDE. Smaller models consume less main memory such that it is possible to load more models at the same time, preventing unnecessary delays.

Problem 1: As more contextual information is used to describe the code in the models and as the number of input object usages increases, model sizes become increasingly big.

As more code repositories are being mined and additional context information is being used, more noise (erroneous data) is likely to appear in the models. If the clustering technique being used does not effectively filter out noise, then accurate models cannot be produced.

Problem 2: More input data may result in more noise that should be filtered to provide accurate recommendations.

2.3 Intuition Behind Using BMF

Given the above two problems, we need to find a way to reduce the size of the model without loosing important information. The pipeline in Figure 1 shows that the clustering technique used in Step 2 (dashed frame) affects the number of patterns detected, which in turn affects the size of the calculated Bayesian Network. Therefore, by using advanced clustering techniques, we can reduce the resulting model size. Matrix factorization techniques provide an alternative clustering technique [3]. Previous work shows that BMF performs better with binary data compared to other

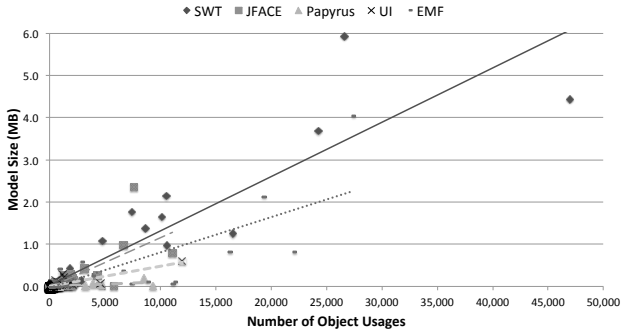

Figure 3: Scalability of model size in existing PBN

MF methods [19]. Since the matrix used to represent object usages is already Boolean, BMF is well-suited in this context. We expect BMF to produce smaller model sizes with accurate recommendations than the currently used canopy clustering because of its following characteristics:

(1) Identifies outliers and removes them from the data set. Canopy clustering assigns all the data points to patterns. This means that canopy clustering cannot handle erroneous data points (*outliers*). BMF automatically removes noisy data during the factorization process.

(2) Avoids user-defined parameters to cluster the data. In the background, canopy clustering uses two user-specified parameters t_1 and t_2 in order to determine the distance between the data points that will be clustered. In practice, the user has two choices. *The first* is to specify global values for t_1 and t_2 without taking into consideration that different API types would require different values (as currently implemented in PBN [16]). *The second* is to perform extensive analysis for each API type individually in order to achieve better results. This of course introduces a human-involvement bottleneck and scalability issues. By using *MDL4BMF* [12], we can automatically calculate the optimal number of patterns needed to represent every API type in the code repositories specific to the given data set.

3. INTEGRATING BMF INTO PBN

In this section, we first give a brief description of BMF and then describe how we integrate it into PBN.

BMF Problem Definition: *Given a Boolean matrix A of size $m \times n$ and an integer k representing the expected factorization rank, find a factorization of A into a Boolean matrix B of size $m \times k$ and a Boolean matrix C of size $k \times n$ such that the error introduced by the factorization is minimized:*

$$min(|A \oplus (B \circ C)|) \tag{3}$$

where matrices B and C are *factor matrices* of A. The *factorization rank* k represents the number of *clusters* in clustering terminology. The *xor* operation ($\oplus$) is used to calculate the factorization error between the data matrix A and the Boolean product of the two factor matrices B and C.

To provide the data matrix A for BMF, we use the same object usage representation shown in Table 1, which is shown again in Figure 4 (differences to Table 1 are explained in

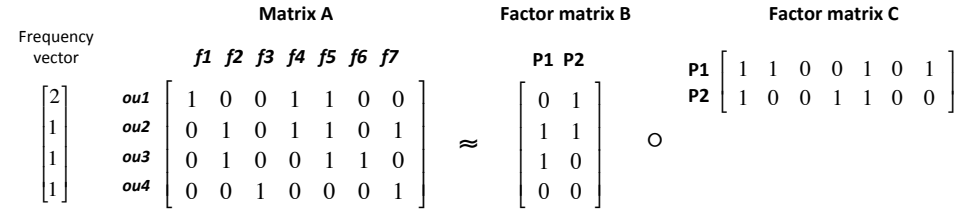

Figure 4: An approximation of the Boolean matrix A by the Boolean product of the two factor matrices B and C. The frequency vector shows the number of occurrence for the object usages.

Section 3.1). The object usages within a cluster are defined by the factor matrix B, where $B(i, j) = 1$ means that object usage i is in pattern j. For example, matrix B in Figure 4 indicates that *ou2* and *ou3* belong to pattern `P1`.

As in clustering approaches, similar (different) data points need to be assigned to the same (different) rank. This is ensured through matrix product algebra. Every element in row r of matrix A is equal to the sum of the products of row r in matrix B and its corresponding column in matrix C.

$$A(r,1) = \sum B(r,:) \circ C(:,1)$$
$$A(r,2) = \sum B(r,:) \circ C(:,2)$$
$$..$$
$$A(r,m) = \sum B(r,:) \circ C(:,m)$$

In this way, if there are two (or more) similar object usages in the data set, they will have similar rows in matrix A. Following the above explanation, they will also have similar rows in factor matrix B. Consequently, they will be assigned to the same rank in factor matrix B.

3.1 Using BMF to Generate Patterns

To apply BMF on our input data, we introduce some modifications to the data representation. The left side of Figure 4 shows the same matrix as in Table 1. However, instead of repeating duplicate rows (i.e., *ou1* and *ou5*), we introduce a *frequency vector* that stores the number of times a specific object usage is observed in the code repository. Thus, the input data matrix A is reduced to four rows, but a frequency vector is introduced that preserves count information.

Given the data matrix A, BMF produces the factor matrices B and C shown on the right of Figure 4. Note that for BMF, a given object usage may be assigned to one or more patterns or to none of them (outliers). Even though *canopy clustering* is considered a *soft clustering* algorithm, the current variant used in the PBN pipeline configures the distance threshold in a way that reduces it to hard clustering (each data point is assigned to exactly one cluster).

To ensure that patterns closely represent their contained object usages and to be comparable with the canopy clustering configuration, we introduce a heuristic to handle corner cases where the same usage is assigned to multiple patterns. The heuristic assigns each object usage to *one* pattern that it is most similar to based on the *Hamming distance* between the feature vector of the object usage in question and the patterns it belongs to. The object usage is then removed from the other pattern(s) by changing its corresponding value to 0 in factor matrix B. The object usage will only be assigned to multiple patterns if they share the smallest *Hamming distance*. For example, in Figure 4, *ou2* has been assigned to both patterns `P1` and `P2`. Since the *Hamming* distance to pattern `P1` is 2 and to `P2` is 3, we would remove it from `P2` and assign it only to `P1`.

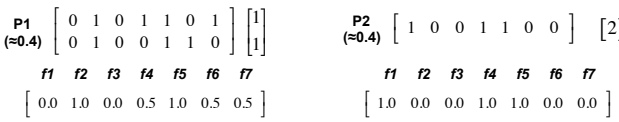

Figure 5: Patterns generated from BMF

3.2 Calculating PBN

After the patterns are detected, we need to calculate the probabilities of the patterns and of the features within a pattern. Such probabilities are calculated using Equation 1 and Equation 2, respectively. Note that the counts are taken from the introduced frequency vector.

In canopy clustering, the probabilities of all the patterns sum up to 1, but this is not the case for BMF since there are object usages assigned to multiple patterns or to none of them (outlier). This is internally handled by the BN implementation, which does a normalization of the pattern probabilities to sum up to 1.

BMF can be used in Step 2 of Figure 1. Its generated patterns from our example (Figure 5) have the same format as those generated using canopy clustering and can directly be used to calculate the BN in Step 3. After that, the inference engine can be used to infer method proposals (Step 4).

4. EVALUATION

We use the PBN pipeline from Figure 1 to compare the performance of the two clustering approaches that have been discussed in this work: canopy clustering and BMF. For comparability, we reuse the publicly available dataset that was previously used to evaluate PBN [16]. The dataset was obtained from the Eclipse Kepler update site, which is the main source of plug-ins for all Eclipse developers. We focus our evaluation on the `SWT` framework[1], the open source UI toolkit used in the Eclipse development environment. The static analysis identified 44 different API types used in our evaluation, with a total of 190,000 object usages. We use 10-fold cross-validation to evaluate each extracted API type. The object usages are disjointly assigned to 10 folds where the union of 9 folds (*training set*) is used to learn the models, and the remaining one (*validation set*) is used for querying the learned models. To avoid intra-project comparisons that may introduce a positive bias to prediction quality, we ensure that object usages generated from the same project are assigned to the same fold.

4.1 Recommender Evaluation

We focus our evaluation on three properties: *prediction quality*, *model size*, and *inference speed*.

Prediction quality. Any new clustering approach should not have a big negative effect on the prediction quality. A

[1]http://www.eclipse.org/swt/

Table 2: F_1-measure of different recommenders

App.	PBN_{BMF}	PBN_{BMF+}	PBN_{15}	PBN_{40}	PBN_{60}
F_1	0.455	0.470	0.517	0.488	0.367

big negative effect might outweigh any reduction in model size or gain in inference speed. We therefore analyze the prediction quality first. For each API type, the object usages in the validation set are used to query the model (learned from the training set). Multiple queries are constructed by randomly removing about half of the call sites from the original object usage (e.g., given an object usage with 3 calls, it is possible to create 3 queries with 1 call). We build queries that mimic both sequential (coding from up to bottom, or from bottom to up) and random (randomly adding snippets of code in the program) coding styles. The code completion engine is called on these incomplete object usages and the prediction quality is measured by calculating the F_1-measure between the ranked (list of) proposals and the removed method calls. Proposal ranking is used to filter out proposals with a probability lower than 30%. In a last step, the different results of all queries generated from a single object usage are averaged.

Model size. We report the total model size for both approaches (canopy clustering and BMF) in Bytes. This is calculated by multiplying the number of stored float values in the Bayesian Network, representing confidence levels, by the number of Bytes needed to store float values on disk. Since each approach produces a different number of patterns, the resulting model sizes differ. Specifically, more patterns result in more values to be stored in the network.

Inference speed. The inference speed measures the time needed for the code completion engine to predict the relevant method calls for a given query. Inference speed is directly related to model size. A smaller model size means that less time is needed to read the models and calculate the proposals, and vice versa. We measure this time in milliseconds and report an average inference speed for each API type (total computation time divided by the total number of queries). For each type are selected at most 3.000 queries.

4.2 Results

Prediction quality. We compare PBN_{BMF} with and without the heuristic mentioned in Section 3.1, to the three clustered configurations of canopy clustering originally used in PBN [16]: PBN_{15}, PBN_{40}, and PBN_{60}. The indices represent different distance threshold values used for canopy clustering, where smaller indices mean "stricter" clustering (more patterns).

Table 2 shows the F_1-measure averaged over all the analyzed API types. The table shows that our heuristic (PBN_{BMF+}) does have a positive impact on prediction quality compared to PBN_{BMF}. This impact is more noticeable for specific API types.

When compared to PBN_{15}, PBN_{BMF+} compromises the prediction quality (-0.047). This is expected since PBN_{15} is an almost unclustered model. PBN_{BMF+} is, however, comparable to PBN_{40}. The difference (-0.018) in prediction quality is not statistically significant (p-value $= 0.1257$) according to the Mann Whitney U-Test [14]. Note that PBN_{BMF+} reaches a higher prediction quality than PBN_{60} ($+0.103$). Thus, to have a fair comparison, we only compare PBN_{BMF+} and PBN_{40} in the remaining experiments.

Model size. After verifying that prediction quality is not compromised, we analyze the effect of BMF on the model size. To do so, we compare the model sizes of PBN_{BMF+} versus those of PBN_{40}. The model size depends on the number of available object usages for a type since more object usages might result in more patterns and vice versa. Therefore, we show the reduction of model size separately for each API type that has more than 2,000 object usages. We skip API types with less than 2,000 object usages since their model sizes are already small.

In addition to model size, we also show the difference in prediction quality for each of the analyzed types in order to have a fair comparison between the difference in model size and the corresponding impact on prediction quality. Figure 6 shows this comparison for each analyzed type, where model size is shown on the left-lower part of the y-axis and prediction quality is shown on the right-upper part of the y-axis. The plot shows that for almost all the analyzed types, PBN_{BMF+}'s prediction quality is comparable to PBN_{40}, but the model sizes obtained by BMF are much smaller. This is especially obvious for types with a bigger number of object usages (more to the left), showing that PBN_{BMF+} performs better for a larger number of object usages.

The reduction in model size ranges from 30% (`Button`) up to 80% (`Table`). The model size is proportional to the number of patterns created by each of the approaches. For `Table`, the model of PBN_{40} contains 176 patterns on average over all folds, while the model of PBN_{BMF+} contains only 36 patterns after an average of 270 outliers over all folds has been detected during factorization. In our dataset, `Table` is the type for which BMF detects the highest number of outliers. This suggests that the object usages in `Table` differ a lot. While canopy clustering creates separate patterns for these varying object usages, BMF is able to detect the ones that differ significantly from the other object usages and treat them as outliers. A closer inspection of the data shows that the object usages of `Table` are declared in very different contexts, with a rough estimation 75% of all the extracted features are related to context information. Thus, the difference between the object usages of type `Table` is related to the fact that they are declared in very different method contexts. Additionally, we see that the difference between the F_1-measure of PBN_{BMF+} and PBN_{40} for type `Table` is only 0.03. This shows that BMF is not removing important object usages that greatly influence prediction quality.

On the other hand for `Button`, BMF only detects 80 outliers averaged over all folds, even though it has almost five times more object usages compared to `Table`. However, the context information roughly accounts for only 40% of the extracted features while almost all the remaining features are definition sites. Definition sites indicate how an object becomes available in the source code but not how it is used.

For the call site features, on average, two method calls are invoked on object usages of `Button` and `Table`. While method calls in `Button` account for only 0.7% of the total number of features, they account for 6% in `Table`. This suggests that even though the object usages from both types call on average the same number of methods, the total number of methods available in `Table` is almost ten times higher compared to `Button`. This means that more object usages from `Button` will be similar (have a lot of common context information and call almost the same methods), which is why fewer patterns are identified. This is true for both BMF and canopy clustering and explains why BMF results in a

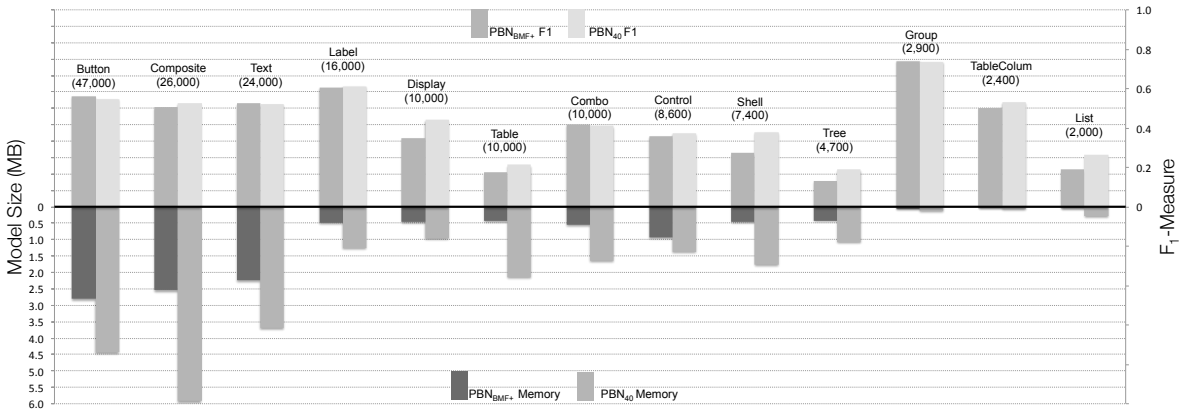

Figure 6: F_1-measure and model size for API types with more than 2,000 object usages. Number of object usages used for each type shown in parenthesis.

smaller reduction in model size here.

Inference speed. After showing that BMF reduces the model size with no significant loss in prediction quality, we expect a speedup in inference speed since a smaller Bayesian network should be faster to query. When compared to PBN_{40}, PBN_{BMF+} does indeed result up to 78% faster in inference speed (from 3 ms to 0.6 ms), and 46% faster (from 3.2 ms to 1.7 ms) when averaged across all API types.

Limitations. Even though the reported numbers in terms of model size (6 MB) and inference speed (3.2 ms) are not a scalability issue for current recommender systems, we use the same dataset as in our previous work [16] for comparability. Note that this work is the first step in using BMF as a means to create smaller models within the context of intelligent method call completion, and our results show that BMF is a promising approach in this direction.

5. THREATS TO VALIDITY

Internal Validity. MDL4BMF is a general machine learning algorithm, not bound to a specific application domain. We experimented with different values of the factorization rank for various API types to ensure that the factorization rank calculated by the algorithm is indeed optimal in our intelligent method call completion context. Our results (not shown due to space restrictions) showed that the factorization rank automatically calculated by *MDL4BMF* does indeed provide the best tradeoff between model size and prediction quality. This gives us confidence that the algorithm correctly calculates the optimal factorization rank and suggests that it is independent from the nature of input data.

External Validity. We test the use of BMF for one dataset within the context of one method call recommender. Different datasets and recommenders might exhibit different behaviors in terms of model size and prediction quality. Additionally, different heuristics might be required for different datasets or recommenders. For example, we analyzed the effects of the following heuristic to ensure that the detected outliers by *MDL4BMF* are indeed valid outliers: *If the frequency ratio of a noisy object usage is lower (higher) than a specific threshold t, we (don't) consider it to be a valid outlier.* Our empirical analyses showed that BMF does indeed filter out valid outliers that have a low frequency ratio without a significant impact in prediction quality. In other datasets, the effect of this heuristic might be different. We do not generalize our results to other datasets but only point out potential applications in the related work. Our work here is a first step to illustrate the use of BMF, and the PBN pipeline allowed us a fair comparison since we have all implementation details.

6. RELATED WORK

Since our main goal is to address scalability and not to propose a new recommender, we do not focus on other intelligent method call completion techniques. We discuss four categories of related work:

Matrix Factorization. Non-Negative Matrix Factorization (NMF) [1] represents a non-negative data matrix using two factor matrices, given a pre-defined factorization rank. In comparison to BMF, NMF requires that the input data matrix and the factor matrices have non-negative values. The algorithm is shown to scale well for large data ranges [8] and is widely used in text mining and data-clustering [21]. One drawback is that we cannot use the MDL principle with NMF to calculate the optimal factorization rank for a given data set. Instead, we need to input the factorization rank as a parameter. We tried NMF with our data by using the same factorization rank calculated by the BMF approach and the results (not shown) were not significantly better than BMF. For a broader overview on different Matrix Factorization methods and their computational complexity, we refer the reader to Miettinen's work [11].

Potential Applications in Code Recommenders. Precise [22] is an approach to recommend parameters for method calls, and the work by Zhang et al. [23] recommends combination of method calls. They both use binary representation of the data and clustering algorithms respectively for parameter and method recommendation. Since the data is already represented in Boolean format, their work might potentially benefit from BMF to construct clusters without requiring the user-specified threshold needed by their approaches.

Potential Applications in Pattern Mining. Frequent itemset mining is a common technique for detecting patterns in datasets. DynaMine [9], PR-Miner [7] and the work by Michail et al. [10] are few examples in this direction. Some of these approaches [9, 10] make use of the *Apriori* algorithm to detect frequent item sets in the data, whose runtime is exponential with respect to the number of items. It is worth investigating whether it is possible to mine patterns using BMF instead. This requires the data to be represented as Boolean matrices in the form *methods* (or other program-

ming elements) by *items* (object, class etc.).

Scalability in Code Recommenders. Weimer et al. [20] applied a different matrix factorization algorithm, Maximum Margin Matrix Factorization to code recommenders. Their technique builds a single model for a complete framework, rather than a model for each API type. The authors point out the scalability issues they face due to the complexity of the optimization problem of the underlying factorization algorithm. This forced them to limit the size of the input data they provide to the algorithm.

Recommender techniques that treat code as plain text [4] or as some form of structured sentences with underlying statistical language models [17], naturally scale to large repositories. On the other hand, GraLan [15] needs a large number of trees/graphs to capture the context information of the code under editing. Therefore, it uses two thresholds to limit the number and size of the generated trees/graphs. However, such techniques don't consider some of the structural information of the code, which is important to make more useful code predictions.

7. CONCLUSIONS

Intelligent code completion systems learn models by analyzing a large number of code repositories to increase the probability of detecting relevant patterns for developers. With the vast increase of available data in such repositories, scalability becomes an issue, especially with respect to the learned model sizes. Another factor that influences model sizes is the use of additional contextual information to improve prediction quality. In this paper, we investigate *Boolean Matrix Factorization* (BMF) as a means to create smaller models by adapting our previously developed Pattern-based Bayesian Network (PBN) framework [16] and replacing the originally used canopy clustering with BMF. We compare both approaches on the SWT framework and show that we obtain model sizes that are up to 80% smaller, which in return reduces inference speed by up to 78%, all while not compromising prediction quality (F_1-measure). Our results suggest that BMF is promising in the context of intelligent method call completion and speculate that other software analytics applications may also benefit from it.

8. ACKNOWLEDGMENTS

This work has been supported by the European Research Council with grant No. 321217, by the German Federal Ministry of Education and Research (BMBF) with grant no. 01IS12054, and by the German Science Foundation (DFG) in the context of the CROSSING Collaborative Research Center (SFB #1119, project E1). The authors take full responsibility for the content of the paper.

9. REFERENCES

[1] M. W. Berry, M. Browne, A. N. Langville, V. P. Pauca, and R. J. Plemmons. Algorithms and applications for approximate nonnegative matrix factorization. *Computational statistics & data analysis*, 52(1):155–173, 2007.

[2] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proc. of ESECFSE*, pages 213–222, 2009.

[3] E. Cergani and P. Miettinen. Discovering relations using matrix factorization methods. In *Proc. of ACM CIKM*, pages 1549–1552, 2013.

[4] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proc. of ICSE*, pages 837–847, 2012.

[5] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proc. of ICSE*, pages 117–125, 2005.

[6] Y. Koren, R. Bell, C. Volinsky, et al. Matrix factorization techniques for recommender systems. *Computer*, 42:30–37, 2009.

[7] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315, 2005.

[8] C. Liu, H.-c. Yang, J. Fan, L.-W. He, and Y.-M. Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *Proc. of WWW*, pages 681–690, 2010.

[9] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *Proc. of FSE*, volume 30, pages 296–305, 2005.

[10] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proc. of ICSE*, pages 167–176, 2000.

[11] P. Miettinen. *Matrix decomposition methods for data mining: Computational complexity and algorithms.* PhD thesis, Helsingin yliopisto, 2009.

[12] P. Miettinen and J. Vreeken. Model order selection for boolean matrix factorization. In *Proc. of SIGKDD*, pages 51–59, 2011.

[13] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Elipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[14] N. Nachar. The mann-whitney u: A test for assessing whether two independent samples come from the same distribution. *Tutorials in Quantitative Methods for Psychology*, 4(1):13–20, 2008.

[15] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code. In *Proc. of ICSE*, volume 1, pages 858–868, 2015.

[16] S. Proksch, J. Lerch, and M. Mezini. Intelligent code completion with bayesian networks. *ACM Transactions on Software Engineering and Methodology*, 25:3, 2015.

[17] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *ACM SIGPLAN Notices*, volume 49, pages 419–428, 2014.

[18] R. Robbes and M. Lanza. Improving code completion with program history. *Automated Software Engineering*, 17(2):181–212, 2010.

[19] V. Snael, P. Kromer, J. Platos, and D. Húsek. On the implementation of boolean matrix factorization. In *Proc. of DEXA*, pages 554–558, 2008.

[20] M. Weimer, A. Karatzoglou, and M. Bruch. Maximum margin matrix factorization for code recommendation. In *Proc. of ACM RecSys*, pages 309–312, 2009.

[21] W. Xu, X. Liu, and Y. Gong. Document clustering based on non-negative matrix factorization. In *Proc. of ACM SIGIR*, pages 267–273, 2003.

[22] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical api usage. In *Proc. of ICSE'12*, pages 826–836.

[23] D. Zhang, Y. Guo, and X. Chen. Automated aspect recommendation through clustering-based fan-in analysis. In *Proc. of ASE*, pages 278–287, 2008.