# Deep Code Recommendation

Naimat Ullah and Hammad Rizwan [⋆]

Lahore University of Management Science, Pakistan
{18030045,18030048}@lums.edu.pk

**Abstract.** Software development has become a highly resource-consuming process with the developers spending most of their time rewriting code for software functionalities that have already been implemented. Over the years many code search databases have been implemented to aid developers to search relevant code for their required functionality. The majority of existing approaches treat code snippets as textual documents and find matching code with a given natural language query. These text similarity based approaches often lose semantic information of code.
In this project, we develop a method recommender (DeepCodeRecommendor) based on API usage and functionality of the code functions. We aim to identify software functionalities that are available in the code repository of various projects, that can be reused for future development. Instead of just relying on textual similarity of code we define **M**ethod **R**epresentation **E**mbeddings **V**ector **(MREV)** that contains information about method functionality, tokens extracted from comments, method name and API usage in the method. Using pre-trained word embeddings of software engineering data, we convert text to numeric vectors. Based on MREV we cluster methods and learn methods co-occurrence patterns across projects. To evaluate the effectiveness of DeepCodeRecommendor, we pick 5 random methods for each test project and pass get 5,10 recommendations of method clusters and match those with actual methods' clusters present in the project. We achieve precision of 66% on 33 test projects.

**Keywords:** Deep learning · data mining · Code recommendation.

## 1 Introduction

Every industry's main focus is improving productivity and quality. The productivity of a system can be enhanced by reusing existing tools and techniques which have already proven to be useful. Software industry is of no exception in this regard, productivity of software development can be improved through reusability of existing code for similar functionalities. Code search is very common technique and has been widely studied in software [8,14,11,6] domain to increase productivity of system. Most of the existing code search techniques are based on Information-Retrieval(IR) techniques. For example, [15] proposed solution for

[⋆] Both authors contributed equally

code searching from a repository of available code-set by computing similarity between search queries and code methods. Some of the researchers make use of Natural language processing techniques by incorporating/extracting information from textual data, like methods name and comments Code search techniques lack semantic information of code and cannot effectively handle irrelevant/ noisy keywords in query. Moreover, these code search techniques don't serve the purpose of code re-usability as they are time consuming and require writing similar queries on which the system was initially trained on. There is a dire need for systems which can automatically recommend method/code based on a users previous activity in a project. In this paper, we propose technique for methods recommendation in a users working directory. For a certain project, based on users existing code and type of project, our systems recommends methods for some functionality for that project. For method recommendation, we learn method co-occurrence patterns in a variety of projects. To further generalize the co-occurrence, we perform clustering on methods based on functionality they are performing, which are learned from generating embeddings vectors for methods. Our system recommends methods based on high co-occurrence, and we evaluate our technique by measuring precision of recommend methods with actually used method in the test project.Our proposed technique achieves precision of 64%. To our knowledge, we are the first to propose deep learning based method recommendation. The main contribution of our work are as follows

– We propose a deep learning based high dimensional methods representation vectors that we called as MREV
– We develop system for methods clustering and formulate clusters co-occurrence matrix
– We empirically evaluate our technique using real-world project implementations

## 2 Related Work

Code search is very common among software community [13]. To implement a certain functionality, developers search for already written code and reuse the available functions. Already existing techniques treat the code search as information retrieval(IR) task and search code based on text queries.[9] proposed sourcerer, an information retrieval based code search tool that combines the textual content of a program with structural information. A fundamental problem with these IR techniques is the mismatch between high-level intent reflected in the natural language query and low level implementations in the source code [10], as source code and natural language queries are heterogeneous they may not share common lexical tokens, synonyms, or language structures. Instead, they may only be semantically related.

Previously developed systems recommending API sequences [6] for different software functionalities and [7] recommends methods from user queries by finding similar methods to a given query through use of method embeddings. This helps to reduce the time that developer spends searching and reading online documentation which might be in many cases vague and difficult to grasp in a short amount of time. The authors [15] use techniques from natural language processing and information retrieval to find software features commonalities across different methods by using text from source code such as comments, method name and API call names that are called within that method. Very similar work to our approach, [4] proposed solution for method recommendation by using data mining technique on API usage patterns within methods.

## 3 Dataset

We have dataset of 1288 java-based projects that have implemented various functionalities using different methods. These projects are collected from F-droid [1] . F-droid is a Website with free and open source apps on the Android platform. For each one of the app projects, there are several versions. We only focus on the latest version in our experiments. Instead of just keeping complete source file, we have extracted features from data namely Method_Name, Method_Parameters, API_Call_Sequence and Comments which are stored in a tabular format. Projects are composed of different methods and each method within the projects use a set of APIs to implement a given functionality. To fulfil our objective of method recommendation we will clustering these methods based on their usage and textual semantic information. To do this we need to convert natural language text into a vector representation to be able to apply mathematical operations. In the following section we explain techniques used for the creation of method vector representations.

## 4 Preliminaries

### 4.1 Embeddings

Nature language text or words are represented as numeric vectors when used in computations, there are several ways of creating these numeric vector representations, three of most major ones are stated below:

**One hot Vectors:** A one-hot vector is a $1 \times$ Vocabulary Size *(Vocabulary is the number of unique words in a data-set)* vector used to distinguish each word in a vocabulary from every other word in the vocabulary. The vector consists of 0 in all cells with the exception of a single 1 in a cell used uniquely to identify the word.

**Word Co-Occurrence/ Word-Word Matrix:** This is a word co-occurrence count matrix of size Vocabulary Size $\times$ Vocabulary Size, where each row represents a word vector, it contains the words co-occurrence count's with respect to all other words in the vocabulary.

**Word Embeddings** Both the representations stated above are quite sparse and very large in size, as vocabulary size can be in millions. In order to alleviate this problem a technique called embedding has been proposed [12] which can be used to generate vector representations using deep learning. These embeddings are generally dense and are able to represent words in such a way that similar words are closer to each other in the vector space. Hence based on the distance between each word's embedding we can identify semantic relations (words that have similar meaning). We will be using these word embeddings[5] that have been trained by fine tuning Word2Vec representations on 15GB of textual data from Stack Overflow posts.

A diagrammatic image of the skip-gram model is shown in Fig. 1., which has been used to train Word2Vec representations previously stated.

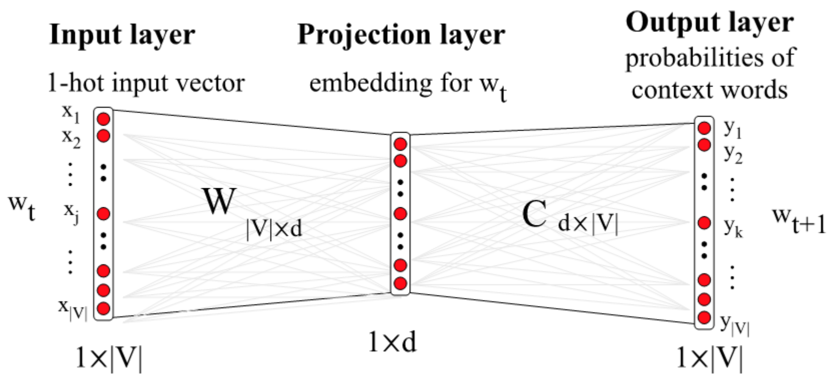
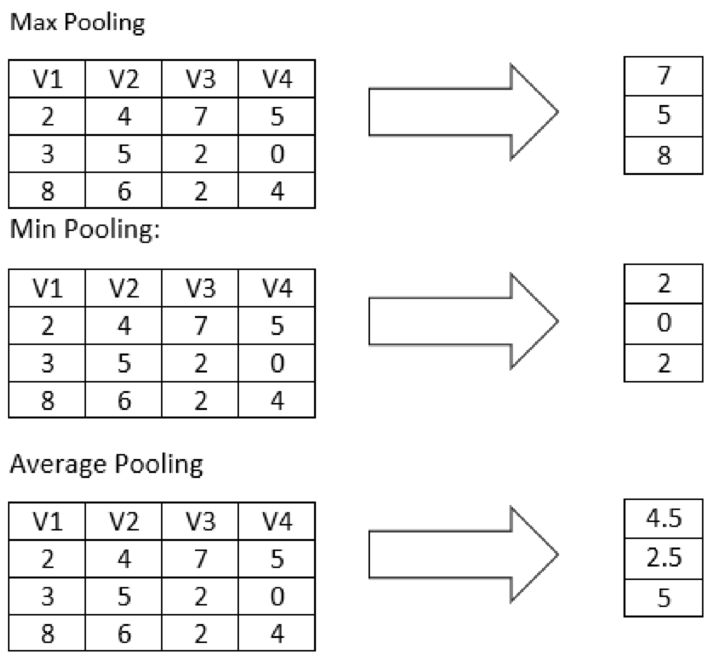


**Fig. 1.** Word Embeddings

### 4.2 Pooling

Pooling is used to capture information from multiple vectors into a fixed size vector. This is done generally by three techniques namely max pooling, min pooling and average pooling. Max pooling results in obtaining the most important information by taking maximum value at a specific index from all vectors involved in the pooling process, min pooling does the opposite by taking minimum value and average pooling gives equal weightage to all vectors.



Max Pooling

| V1 | V2 | V3 | V4 |
|---|---|---|---|
| 2 | 4 | 7 | 5 |
| 3 | 5 | 2 | 0 |
| 8 | 6 | 2 | 4 |

| |
|---|
| 7 |
| 5 |
| 8 |

Min Pooling:

| V1 | V2 | V3 | V4 |
|---|---|---|---|
| 2 | 4 | 7 | 5 |
| 3 | 5 | 2 | 0 |
| 8 | 6 | 2 | 4 |

| |
|---|
| 2 |
| 0 |
| 2 |

Average Pooling

| V1 | V2 | V3 | V4 |
|---|---|---|---|
| 2 | 4 | 7 | 5 |
| 3 | 5 | 2 | 0 |
| 8 | 6 | 2 | 4 |

| |
|---|
| 4.5 |
| 2.5 |
| 5 |

**Fig. 2.** Pooling

### 4.3 RNN

Recurrent Neural Networks is a class of neural networks where the hidden layers are recurrent over time. This gives the network the ability to model dynamic time-series/temporal data. The reason for using RNN for our task is that we break features such as **api_name** into a sequence of natural language words/tokens. Since Feed Forward or vanilla neural network do not preserve the time-series or temporal nature of the data, using them would be counterproductive.

There are many variations of RNN's, here we will mention the vanilla or simple RNN. This model consists of three layers, an input layer which maps each input to a vector, a recurrent layer that recurrently computes and updates the hidden state after receiving input and output layer that uses the output of the hidden state for specific tasks. To further elaborate the workings of an RNN consider a natural language sentence with a sequence of $T$ words $s = \{w_1, \ldots, w_T\}$. RNN read words in the sentence one by one and updates a hidden state at each time step. Each word $w$ is first mapped to a d-dimensional vector $w_t \in R_d$ by a one-hot representation or word embedding. Then, the hidden state (values in the hidden layer) $h_t$ is updated at time t by considering the input word $w_t$ and the preceding hidden state $h_{t-1}$:

$$h_t = tanh(W_h[h_{t-1}; w_t]) \;\; \forall\, t = 1, 2, \ldots, T \tag{1}$$

The output is generated by using the current state:

$$y_t = W_o[h_t] \;\; \forall\, t = 1, 2, \ldots, T \tag{2}$$

Here tanh is an activation function, generally Relu activation function is preferred now a days. A diagrammatic representation of this model is shown in Fig. 3.

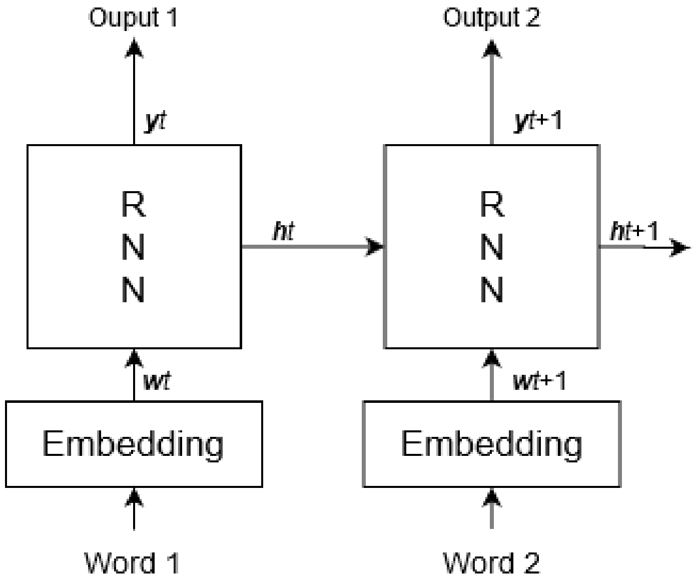


**Fig. 3.** Recurrent Neural Network

## 4.4 Method Representation Embedding Vector

To create Method level embeddings we propose two models:

- For the first technique we use natural language word sequences for each feature e.g. (Api_Call_Sequence) are fed as input to the embedding layer which generates an embedding against each word that is inputted. These embeddings are then pooled together to create a single representation for one feature. To create a method level representation, we fuse these representations from all features to create a method level representation. Figure 4 shows the architecture of this model.

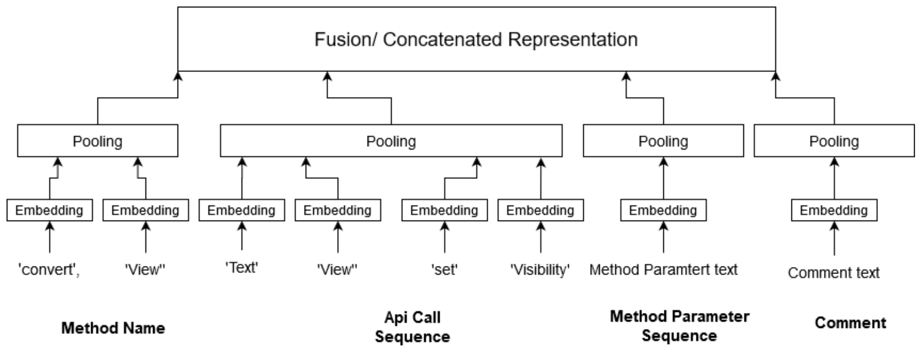


**Fig. 4.** Word Embeddings without RNN model

– For our second technique, instead of directly pooling the embeddings we pass the embeddings to a language model based on RNN's. We use the hidden outputs of the RNN's as new vectoral representation for each word, these are then pooled together. Architecture of this model is shown in figure 5.

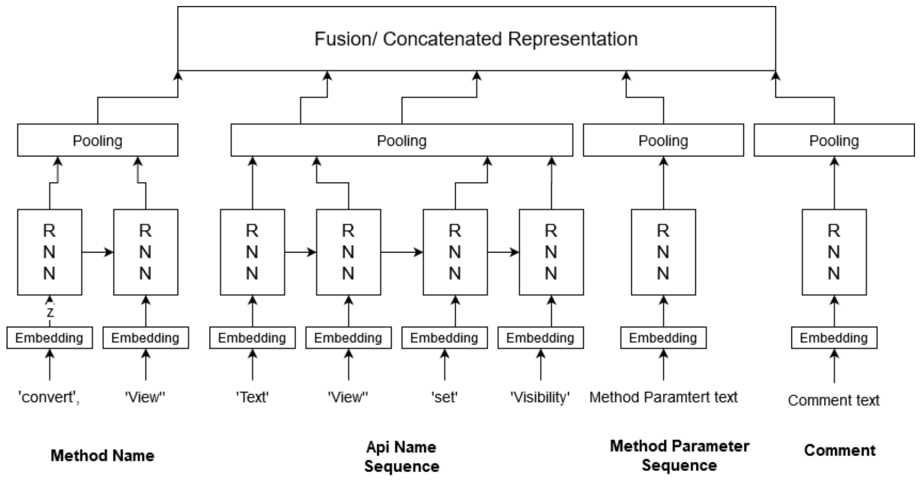


**Fig. 5.** Word Embeddings with RNN model

## 4.5 Clustering

After generating Method representation embedding vectors, we can now use these for clustering of methods. Clustering is essential component while recommending new code method for project based on already written code functions. We use Euclidean distance(Cosine Similarity) as distance metric to form clusters, number of clusters will be decided based on empirical results.

## 5 Proposed Algorithm

The overall workflow for our model is as follows:

– Features such as Method_Name, API_Call_Sequence, etc. which have been written using different coding conventions are preprocessed by splitting each of them based on camel casing, dot casing and underscore to generate a sequence of words that provides us with useful information about a method.
– To use these sequences of words in our model, we need to convert them into numeric vectors for which we use word embeddings. Using our method representation embeddings network in figure 4 and 5 we generate methods embeddings that we call MREV for all methods. After generating MREV vectors we have two different approaches that we evaluate.
– **Clustering-based Approach:** We cluster all the methods in repository using MREV for methods.Clustering is done based on assumption that similar methods have similar have similar word sequences for each of the features. We split project into train and test set, the split ratio is standard 80 percent for train and 20 for test. For methods in training projects, we create a co-occurrence matrix for each cluster with respect to all other clusters , the counts in this matrix are based on clusters co-occurring with other clusters within projects. For test project's method recommendation, we find top $k$ related clusters for each method that is available in the project that the user is working on. Then using these clusters along with our co-occurrence matrix, we identify relevant methods to be recommend for that project.
– For the seconds approach at training time we do not clusters methods together instead we simply store method co-occurrence status within projects. At test time we find top $k$ similar methods for each method that is available in the project that the user is working on. Then using the method co-occurring status that we have stored, we extract those co-occurring methods and based on some similarity threshold, reduce the number of methods that provide similar functionality and recommend these methods to the user for project completion.

Diagramatic version of or algorithm is show in figure 6.

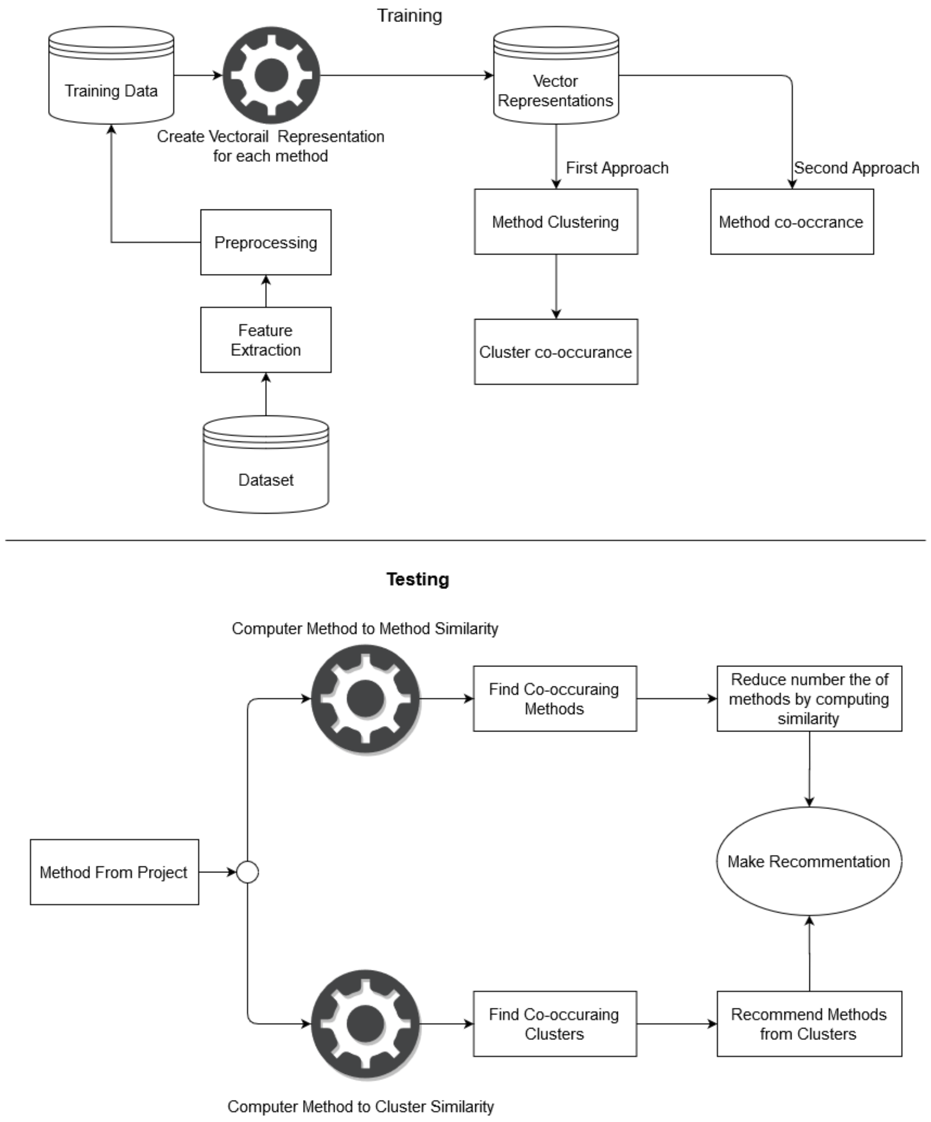

Training
Training Data
Create Vectorail Representation for each method
Vector Representations
First Approach
Second Approach
Method Clustering
Method co-occrance
Preprocessing
Cluster co-occurance
Feature Extraction
Dataset
Testing
Computer Method to Method Similarity
Find Co-occuraing Methods
Reduce number the of methods by computing similarity
Make Recommentation
Method From Project
Find Co-occuraing Clusters
Recommend Methods from Clusters
Computer Method to Cluster Similarity

**Fig. 6.** The Overall workflow of Deep Code Recommendor

## 6 Evaluation

In this we evaluate our deep code recommender through experiments and we present results with different metrics.

### 6.1 Experimental Setup

We have total 1288 project having 777176 methods. For each Java-method we extract a (Api_Call, Method_Name and Method_Parameter and Comments) to generate its representation embeddings vectors. We use multiple approach for the generation process of MREVs that are mentioned in above section. For methods grouping we use python scikitlearn package for agglomerative clustering algorithm for different values of similarity threshold, $\alpha(0.5, 0.7)$ and report results for each. We use k-fold cross validation for evaluations with $k = 10$. For projects in train data fold we generate methods' co-occurrence matrix, and for each method in projects of test fold we ask recommender to recommend top $q$(5,10) clusters of methods. We evaluate resulting methods' clusters with following measures

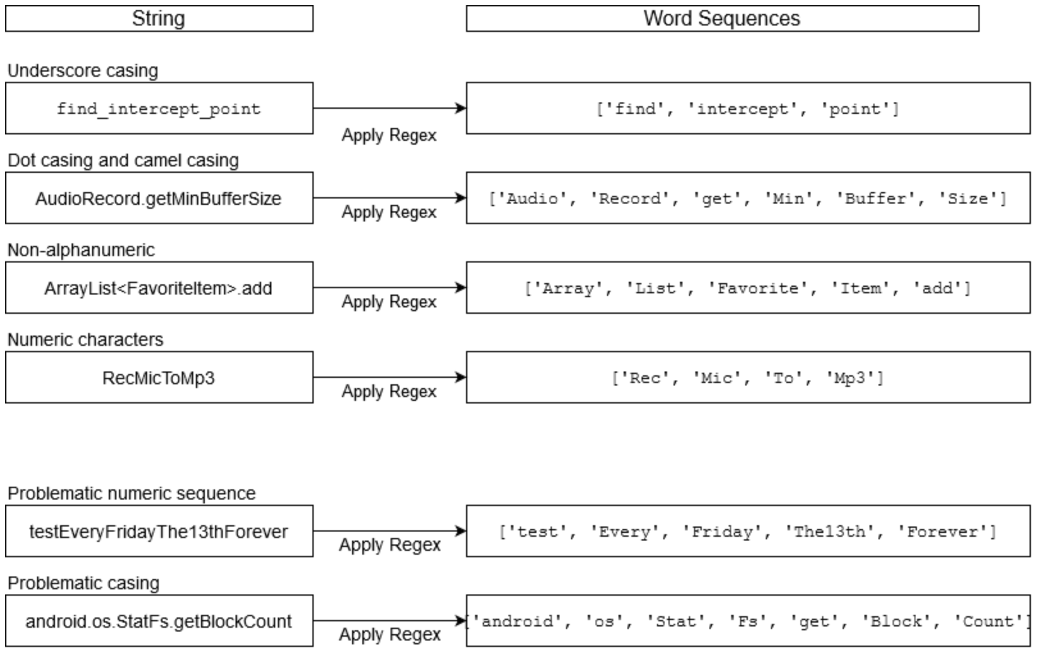
### 6.2 Preprocessing

After analyzing the entries in the Api_Call, Method_Name and Method_Parameter columns. We split each string entry into meaningful word sequences by applying the following operations using regular expressions(regex):

- Split string on camel casing, underscore casing and dot notation. There are certain cases where casing difference does not separate words see 'Problematic casing' in figure 7 'StatFs' has been split, it should have been one word).
- Strings with numeric characters are split on the basis of first capital character that is encountered after numeric sequence. There is the problem not knowing whether the characters before the numeric digits are part of the same word or not see 'Problematic numeric sequence' in figure 7 'the13th' should have been split into 'the' and '13th' in this case.
- Non-alphanumeric characters cannot be used in our model hence we remove characters such as '¿', '¡', '', '', '(' and '(' etc.

Since the number of problematic strings is quite low and its hard to create or find pattern's to split these into meaningful words properly, we must allow for these minor errors by assuming that other words extracted feature provide enough information.

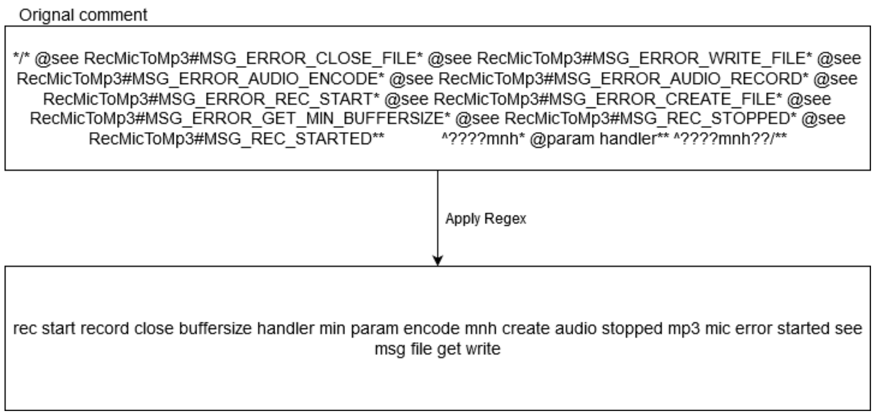
For comments, we find quite a major portion of these are not written in proper English, in these comments words are split by using the symbols such as hashtag/pound (#), dash(-) , dot(.), slash(/) and underscore(_). We first remove non-alphanumeric characters such as$*, \hat{}\ , @, ?, >, <, [, ], (, )$ etc. and then split these sentences based on space and the previous stated symbols. There quite a lot of repetition of words in a single comment, we only use unique words to



String
Word Sequences
Underscore casing
find_intercept_point
Apply Regex
['find', 'intercept', 'point']
Dot casing and camel casing
AudioRecord.getMinBufferSize
Apply Regex
['Audio', 'Record', 'get', 'Min', 'Buffer', 'Size']
Non-alphanumeric
ArrayList<FavoriteItem>.add
Apply Regex
['Array', 'List', 'Favorite', 'Item', 'add']
Numeric characters
RecMicToMp3
Apply Regex
['Rec', 'Mic', 'To', 'Mp3']
Problematic numeric sequence
testEveryFridayThe13thForever
Apply Regex
['test', 'Every', 'Friday', 'The13th', 'Forever']
Problematic casing
android.os.StatFs.getBlockCount
Apply Regex
['android', 'os', 'Stat', 'Fs', 'get', 'Block', 'Count']

**Fig. 7.** Preprocessing

maximize information in a comment although it can be argued that repetitive words might provide insights to what the method for that comment focuses on but since we are using pooling in our models, this will be counter productive as values in the pooled representation will be dominated by these repeating words. An example of comment preprocessing is shown in the Figure 8.



Orignal comment
*/* @see RecMicToMp3#MSG_ERROR_CLOSE_FILE* @see RecMicToMp3#MSG_ERROR_WRITE_FILE* @see RecMicToMp3#MSG_ERROR_AUDIO_ENCODE* @see RecMicToMp3#MSG_ERROR_AUDIO_RECORD* @see RecMicToMp3#MSG_ERROR_REC_START* @see RecMicToMp3#MSG_ERROR_CREATE_FILE* @see RecMicToMp3#MSG_ERROR_GET_MIN_BUFFERSIZE* @see RecMicToMp3#MSG_REC_STOPPED* @see RecMicToMp3#MSG_REC_STARTED** ^????mnh* @param handler** ^????mnh??/**
Apply Regex
rec start record close buffersize handler min param encode mnh create audio stopped mp3 mic error started see msg file get write

**Fig. 8.** Preprocessing

### 6.3 Creating Method Representation Embedding Vector(MREV)

To create MREV, we use the word sequences that we have created in the preprocessing phase and preprocessed comments. For our first approach we load the pretrained embeddings using the genism library [2] and create the method representation using the first approach as stated in section 4.4. Analyzing the process of MREV creation using our first approach, we have found three major issues.

- The First issue is regarding length of comments, there are comments that are significantly longer than others, for example the maximum length of a comment is greater than 400 while the mean comment size is 5. Pooling on large number of word vector will be futile. Authors of paper [6] claims that most of the comment's information is captured in the first sentence, hence we can be sure the most of the information will be captured in the first 20 words of a comment.
- The second issue is that some of the methods do not have either method parameters or any comments *(illustrated in Fig 9 and 10)*. This is problematic due to the fact we concatenate the vectors from Method_Name, API_Call_Sequence, Method_Parameters and Comments to create a MREV, we have to use a vector of zero's as a replacement to keep representation for all method's at a fixed size. This will result in such methods representation being closer together in the vector space, hence they will have a smaller distance to each other for any distance metric that would be used for clustering. Hence we also try pooling operation instead of concatenation as-well.
- The third issue is that there are many words that are not found in the pretrained vectors, hence we ignored these words, this results in lost information but hopefully our second approach that is based on RNN's will be able to generate representation for out of vocabulary*( words a model has never seen before)* based on its training.
  For our second approach we have created our RNN' model's using 'Keras'.[3]

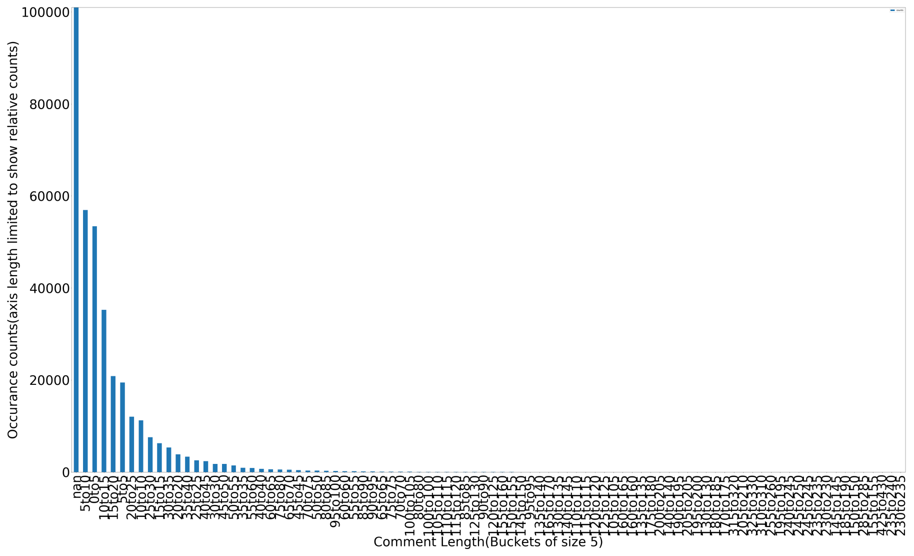


**Fig. 9.** Comment length vs Occurrence counts

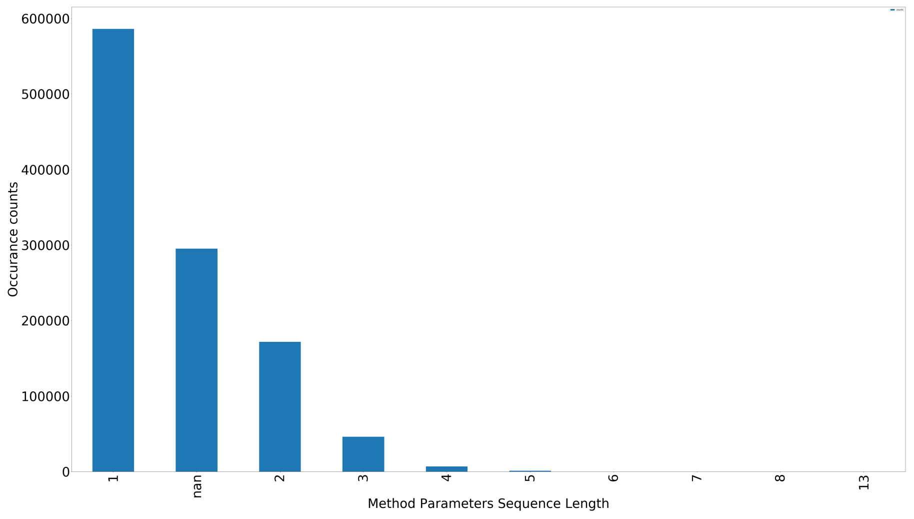


**Fig. 10.** Method parameter sequence length vs Occurrence counts

## 6.4 Performance Measure

We use five common metrics to measure the effectiveness of code recommendation, namely, Precision**@**q, Recall**@**q, , SuccessRate**@**q, Mean Reciprocal Rank (MRR) and Normalized Discounted Commulative Gain(NDCG).

**Precision:** We calculate precision for each method in a test project and take average of all methods for project precision. And precision for all test projects is the mean of projects' precision.

$$Precision = \frac{|\text{recommended clusters} \cap \text{test project actual method clusters}|}{|\text{ recommended clusters}|} \tag{3}$$

**Recall:**

$$Recall = \frac{|\text{recommended clusters} \cap \text{test project actual method clusters}|}{|\text{test project actual method clusters}|} \tag{4}$$

**SuccessRate @q:** Success rate is the percentage of times at least one recommendation was relevant in top $q$ ranked results

$$SuccessRate@q = \frac{1}{|Q|}\sum_{i=1}^{Q} I(q \in \text{Project cluster}) \text{ I is an indicator function} \tag{5}$$

**Mean Reciprocal Rank(MMR):**

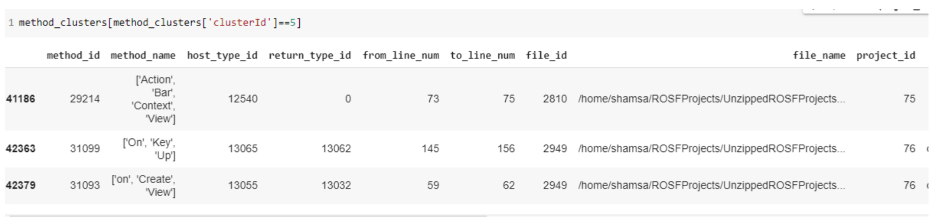
$$MMR = \frac{1}{|Q|}\sum_{i=1}^{Q}\frac{1}{rank_i} \tag{6}$$

Due to time constraint in this work we just evaluate our experimental results with Precision. Other measures can be reported in extended work.

## 6.5 Results and Discussion

The projects selected for evaluation are chosen in chronological order. Due to computational limitations, we were only able to process methods that are contained in first 165 projects. We reach this threshold by decreasing from total 1288 projects. Increasing projects for evaluation with enhanced computation power will improve results and it will make recommendation more generalised as the number of unique method do-not scale linearly with the number of project, which will result in reduction of the total number of small clusters. We show some sample methods in randomly selected clusters in Figures 11,12,13,14.

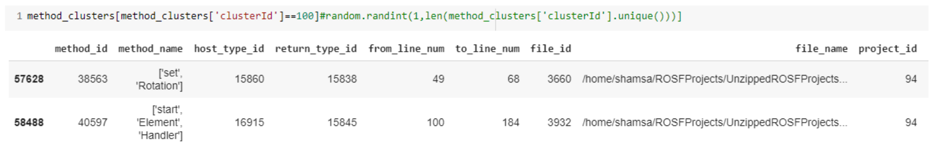
These 165 project we have selected contain a total of $93,103$ methods. We perform agglomertive clustering for these method with distance threshold $\alpha$ as



```
method_clusters[method_clusters['clusterId']==5]
```

| | method_id | method_name | host_type_id | return_type_id | from_line_num | to_line_num | file_id | file_name | project_id |
|---|---|---|---|---|---|---|---|---|---|
| **41186** | 29214 | ['Action', 'Bar', 'Context', 'View'] | 12540 | 0 | 73 | 75 | 2810 | /home/shamsa/ROSFProjects/UnzippedROSFProjects... | 75 |
| **42363** | 31099 | ['On', 'Key', 'Up'] | 13065 | 13062 | 145 | 156 | 2949 | /home/shamsa/ROSFProjects/UnzippedROSFProjects... | 76 |
| **42379** | 31093 | ['on', 'Create', 'View'] | 13055 | 13032 | 59 | 62 | 2949 | /home/shamsa/ROSFProjects/UnzippedROSFProjects... | 76 |

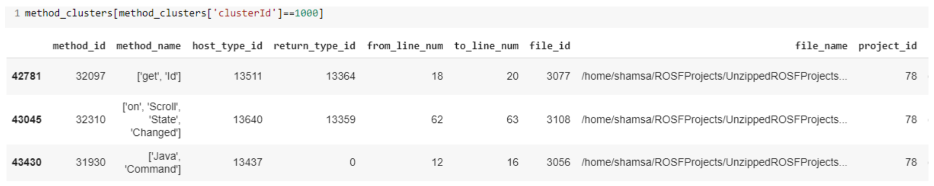
**Fig. 11.** Methods sample in cluster 5



```
method_clusters[method_clusters['clusterId']==100]#random.randint(1,len(method_clusters['clusterId'].unique()))]
```

| | method_id | method_name | host_type_id | return_type_id | from_line_num | to_line_num | file_id | file_name | project_id |
|---|---|---|---|---|---|---|---|---|---|
| **57628** | 38563 | ['set', 'Rotation'] | 15860 | 15838 | 49 | 68 | 3660 | /home/shamsa/ROSFProjects/UnzippedROSFProjects... | 94 |
| **58488** | 40597 | ['start', 'Element', 'Handler'] | 16915 | 15845 | 100 | 184 | 3932 | /home/shamsa/ROSFProjects/UnzippedROSFProjects... | 94 |

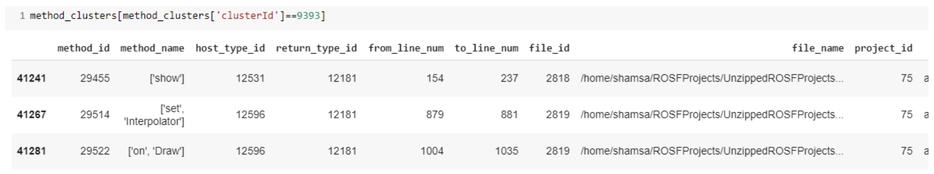
**Fig. 12.** Methods sample in cluster 100



```
method_clusters[method_clusters['clusterId']==1000]
```

| | method_id | method_name | host_type_id | return_type_id | from_line_num | to_line_num | file_id | file_name | project_id |
|---|---|---|---|---|---|---|---|---|---|
| **42781** | 32097 | ['get', 'Id'] | 13511 | 13364 | 18 | 20 | 3077 | /home/shamsa/ROSFProjects/UnzippedROSFProjects... | 78 |
| **43045** | 32310 | ['on', 'Scroll', 'State', 'Changed'] | 13640 | 13359 | 62 | 63 | 3108 | /home/shamsa/ROSFProjects/UnzippedROSFProjects... | 78 |
| **43430** | 31930 | ['Java', 'Command'] | 13437 | 0 | 12 | 16 | 3056 | /home/shamsa/ROSFProjects/UnzippedROSFProjects... | 78 |

**Fig. 13.** Methods sample in cluster 100



```
method_clusters[method_clusters['clusterId']==9393]
```

| | method_id | method_name | host_type_id | return_type_id | from_line_num | to_line_num | file_id | file_name | project_id |
|---|---|---|---|---|---|---|---|---|---|
| **41241** | 29455 | ['show'] | 12531 | 12181 | 154 | 237 | 2818 | /home/shamsa/ROSFProjects/UnzippedROSFProjects... | 75 |
| **41267** | 29514 | ['set', 'Interpolator'] | 12596 | 12181 | 879 | 881 | 2819 | /home/shamsa/ROSFProjects/UnzippedROSFProjects... | 75 |
| **41281** | 29522 | ['on', 'Draw'] | 12596 | 12181 | 1004 | 1035 | 2819 | /home/shamsa/ROSFProjects/UnzippedROSFProjects... | 75 |

**Fig. 14.** Methods sample in cluster 9393

defined earlier. We randomly split projects to train/test with 80% , 20% ratio. We do recommendation for randomly selected 5 methods of each test project, we run this experiment for 10 times and take the average score. Due to time limitations we only report precision. Results of the experiments are summarized in Table s1.

| **Embeddings** | $\alpha$ | **no of cluster** | $q$ | **Average Precision@q** |
|---|---|---|---|---|
| Concatenated with avg pooling | 0.5 | 35551 | 5 | **64.6 %** |
| Concatenated with avg pooling | 0.5 | 35462 | 10 | **61.4 %** |
| Concatenated with avg pooling | 0.7 | 35462 | 5 | **64.3 %** |
| Concatenated with avg pooling | 0.7 | 35462 | 10 | **61.7 %** |
| Concatenated with max pooling | 0.5 | 35462 | 5 | **64.5 %** |
| Concatenated with max pooling | 0.5 | 35462 | 10 | **62.4 %** |
| Concatenated with max pooling | 0.7 | 35462 | 5 | **66.2 %** |
| Concatenated with max pooling | 0.7 | 35462 | 10 | **61.5 %** |

**Table 1.** Method recommendations for randomly selected 5 methods of each project in test data.(Not RNN fine tuned)

We have also fined tuned the pretrained embeddings on our dataset via language modelling with RNNs, Table 2 shows the results

| **Embeddings** | $\alpha$ | **no of cluster** | $q$ | **Average Precision@q** |
|---|---|---|---|---|
| Concatenated with avg pooling | 0.5 | 35551 | 5 | **64.2 %** |
| Concatenated with avg pooling | 0.5 | 35462 | 10 | **61.4 %** |
| Concatenated with avg pooling | 0.7 | 35462 | 5 | **64.3 %** |
| Concatenated with avg pooling | 0.7 | 35462 | 10 | **61.7 %** |
| Concatenated with max pooling | 0.5 | 35462 | 5 | **64.2 %** |
| Concatenated with max pooling | 0.5 | 35462 | 10 | **61.4 %** |
| Concatenated with max pooling | 0.7 | 35462 | 5 | **64.5 %** |
| Concatenated with max pooling | 0.7 | 35462 | 10 | **62.4 %** |

**Table 2.** Method recommendations for randomly selected 5 methods of each project in test data.(RNN fine tuned)

From the results in Figure 1 and 2 we conclude that we can achieve a highest of 66% precision score. Due to random nature of choosing methods for evaluation the precision score can vary with each run, this is due to the fact that there are many clusters that contain either small number of methods or just one method. Removing or merging these small clusters further can increase our algorithms performance. Fine tuning the embedding for out of vocabulary words is not that useful as the results are similar to directly using pre-trained embeddings.

## 7 Conclusion and Future Directions

We introduce the coding methods representation in high dimensions by using pre-trained word embeddings. We provide two approaches creating methods representation vectors(MREV). Clustering is performed on these MREVs to learn about similar functionality provided by these methods.We recommend methods for test project based on selecting a random method of the project, our recommendation algorithm works on clusters co-occurrence matrix. We achieve precision of 66% on 33 test projects that our technique outperform the baseline [4]. Due to time limitation we were not able to complete evaluation for our second method of creating MREV which uses hidden output of RNN's, although we have completed the code for this part as-well, the only requirement left is to write the hidden representations using the model to a file and use these in the evaluation code. Upon analysis of our work up -till now we recommend that the projects used for such recommendation should be from a single specific domain so that there is significant overlap of functionality, also some of these projects are quite large as compared to other,hence it would be better to fix a maximum number of methods per project, so that higher quality methods can be picked for which all features such as Comments, Method_Parameters and API_Sequences are available.

## References

1. F-droid, accessed: 2020-01-05
2. Gensim libray, accessed: 2020-07-05
3. Keras library, accessed: 2020-07-05
4. Abid, S.: Recommending related functions from api usage-based function clone structures. Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (2019)
5. Efstathiou, V., Chatzilenas, C., Spinellis, D.: Word embeddings for the software engineering domain. In: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR) (2018)
6. Gu, X., Zhang, H., Kim, S.: Deep code search. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). pp. 933–944 (2018)
7. Gu, X., Zhang, H., Zhang, D., Kim, S.: Deep api learning. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. p. 631–642. Association for Computing Machinery (2016)
8. Haiduc, S., Bavota, G., Marcus, A., Oliveto, R., De Lucia, A., Menzies, T.: Automatic query reformulations for text retrieval in software engineering. In: Proceedings of the 2013 International Conference on Software Engineering (2013)
9. Linstead, E., Bajracharya, S., Ngo, T., Rigor, P., Lopes, C., Baldi, P.: Sourcerer: Mining and searching internet-scale software repositories. Data Min. Knowl. Discov. **18**, 300–336 (2009)
10. McMillan, C., Grechanik, M., Poshyvanyk, D., Fu, C., Xie, Q.: Exemplar: A source code search engine for finding highly relevant applications. IEEE Transactions on Software Engineering (2012)
11. McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., Fu, C.: Portfolio: Finding relevant functions and their usage. pp. 111–120 (2011)
12. Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J.: Distributed representations of words and phrases and their compositionality (2013)
13. Nie, L., Jiang, H., Ren, Z., Sun, Z., Li, X.: Query expansion based on crowd knowledge for code search. IEEE Transactions on Services Computing **9**, 771–783 (2016)
14. Raghothaman, M., Wei, Y., Hamadi, Y.: Swim: Synthesizing what i mean: Code search and idiomatic snippet synthesis. In: Proceedings of the 38th International Conference on Software Engineering. p. 357–367 (2016)
15. Sachdev, S., Li, H., Luan, S., Kim, S., Sen, K., Chandra, S.: Retrieval on source code: A neural code search. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. p. 31–41. Association for Computing Machinery (2018)