Programming Language Identification in Stack Overflow Post Snippets with Regex Based Tf-Idf Vectorization over ANN

Aman Swaraj and Sandeep Kumar
Indian Institute of Technology- Roorkee, India

Keywords: Tf-Idf, BERT, Word2Vec, Regex, Stack Overflow, Programming Language Classification.

Abstract: Software Question-Answer (SQA) sites such as Stack Overflow (SO) comprise a significant portion of a developer's resource for knowledge sharing. Owing to their mass popularity, these SQA sites require an appropriate tagging mechanism to better facilitate discussion among users. An intrinsic part of predicting these tags is predicting programming languages of the code segments associated with the questions. Usually, state of art models such as BERT and embedding-based algorithms such as word2vec are preferred for the text classification task, however, in the case of code snippets that are different from natural language in both syntactic as well as semantic composition, embedding techniques might not yield as precise results as traditional methods. To this predicament, we propose a regex-based tf-idf vectorization approach followed by chi-square feature reduction over an ANN classifier. Our method achieves an accuracy of 85% over a corpus of 232,727 stack overflow code snippets which surpasses several baselines.

1 INTRODUCTION

In the software engineering domain, dedicated forums like Stack Overflow (SO) play a significant role in knowledge sharing.

The potent ability of these Software Question-Answer (SQA) sites to facilitate discussion between so many developers on such a colossal scale has led to their rapid growth and popularity. As a result, they witness huge traffic on a daily basis and SO alone comprises of around 23 million questions till date. Further, since the queries are vastly diverse, SQA sites encourage users to tag their questions appropriately and adequately. These tags play a crucial role as they make the context of the question more concise, thus making it easier for potential experts to find relevant queries of their domain.

However, the quality of tags is often associated with the questioner's expertise, proficiency in English, writing styles, bias, and so on. Since these factors vary among developers, keeping the tags consistent becomes challenging and gives rise to issues such as tag synonyms and tag explosion (Barua et al., 2014). Such complications point to the need for a mechanized approach to tag recommendation that can predict tags for unseen posts based on historical data or generate tags based on the post's content.

Recently, NLP and ML based techniques have shown promising results in the domain of programming language identification and similar tasks such as code completion, code comment generation and source code summary generation etc.

Applying ML and NLP based techniques on large samples allow the models to extract many features, however, the same task becomes more challenging in the context of a code snippet as snippets are relatively very short in size with respect to a complete code file.

For our work, we have gathered code snippets from a corpus of stack overflow posts shared by (Alreshedy et al., 2020). Since, SO code snippets are quite unstructured, it further makes the classification task a non-trivial one.

In general, text classification tasks involves working on raw text which is often unstructured, inconsistent and of variable length, and thus it becomes important to adequately pre-process the text and apply suitable vectorization method. Again, the method for text vectorization has to be chosen carefully as the quality of text vectorization directly impacts the task at hand.

Traditional methods for text vectorization include bag of words and Tf-Idf, whereas, more recent ones comprise of embedding based algorithms such as Word2Vec (Mikolov et al., 2013), GloVe (Pennington et al., 2014) etc. However, these pre-

Swaraj, A. and Kumar, S.
Programming Language Identification in Stack Overflow Post Snippets with Regex Based Tf-Idf Vectorization over ANN.
DOI: 10.5220/0011989600003464
In *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2023)*, pages 648-655
ISBN: 978-989-758-647-7; ISSN: 2184-4895

trained models are trained on general purpose corpus such as Wikipedia or newspaper text, and therefore fine tuning them for source code classification task might not yield accurate results.

To this reasoning, in this work, we employ a regex based tf-idf approach where regex patterns are used for creating the tokens followed by tf-idf vectorization. Since the tf-idf vectorization yields huge sparse matrices containing lots of zeros, we devise chi square test to reduce the dimensionality of the feature matrices. This significantly reduces our computational time as well. Finally, the reduced subset of features are fed to a suitable classifier. Simultaneously, we also run the experiment on distil-Bert (Sanh et al., 2020) and Word2Vec methods to investigate the role of semantics in source code classification.

Rest of the paper is structured as follows: we start with literature survey in section 2. In section 3, we describe all the tools and methodologies adopted in the experiments. We present our results in section 4 along with the comparative analysis with existing baselines. Finally, section 5 concludes the work along with future scope.

2 RELATED WORK

SQA sites are pretty diverse and involve a vast span of topics. Since it is difficult to accurately capture the semantics of a post based on a single model alone, different researchers approach the task in different ways.

Many authors predict the tags based on the post's title and description while neglecting the post's snippet.

One of the earliest works in this connection is attributed to (Kuo, 2011) who classified SO posts with the help of a KNN classifier. Similarly, (Stanley and Byrne, 2013) made use of a Bayesian probabilistic model to classify the tag of the SO posts. (Kavuk and Tosum, 2020) also classified the body and title of the SO posts with the help of a multi tag classifier based on latent dirichlet allocation. Few more works which considered only title as the input include (Saha et al., 2013; Jain and Lodhavia, 2020; Swaraj and Kumar, 2022).

On the other hand, some works consider snippets vital in discerning the tag correctly since the queries are closely related to the attached code snippets (Cao et al., 2021).

Apart from predicting tags in SQA sites, source code classification in itself has been a topic of rising interest for developers.

In this connection, (Klein et al., 2011) trained their model on a corpus of 41,000 files gathered from github. However, their classifier based on selection of intelligent statistical features could achieve only 48% accuracy.

Similarly, (Khasnabish et al., 2014), gathered around 20,000 files from multiple github repositories. They employed Bayesian classifier to predict ten sets of languages yielding around 93.48% accuracy.

(Van Dam et al., 2016) also proposed a method based on statistical language model to classify source code files taken from github varying across 19 different languages giving an accuracy of around 97%. Another work by (Gilda, 2017) made use of convolutional neural networks to classify 60 programming languages taken from github repositories with a decent accuracy of 97%.

However, all these works have considered github repositories as their training dataset. Since a large source code file contains many distinguishing features, it is relatively easy for the classifier to predict the programming language as evident in the case of (Van Dam et al., 2016) and (Gilda, 2017). On the other hand, detection of programming language in a code segment on SQA sites is relatively much difficult.

In this connection, (Rekha et al., 2014) presented a hybrid model based on multinomial naïve bayes algorithm which automatically classifies the code snippets of SO posts with an accuracy of 72%.

On similar lines, (Baquero et al., 2017) detected programming language in 18000 code segments gathered from SO. However, their classifier based on support vector machine achieved very low accuracy of 44.6%. (Saini and Tripathi, 2018) in their work included SO snippets along with the post title and description and achieved an accuracy of 65%.

More recently, (Alrashedy et al., 2020) presented a method SCC, where they generated tags for their corpus of SO posts with a Random Forrest and XG boost classifier. They achieved an accuracy of 78% when they tried to predict the tags on the basis of snippets alone. However, combining the features of snippets with title and body significantly increased the overall performance which goes to show the crucial role of utilizing code snippets while classifying the posts.

3 METHODOLOGY

The overall flow of our approach is presented in figure 1. It comprises of 4 steps, namely – data collection, pre-processing (tokenization),

vectorization, dimensionality reduction, and classification.

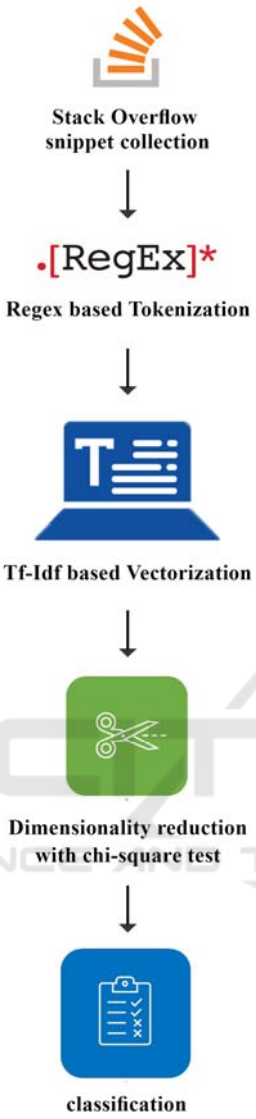

Figure 1: Overview of our proposed approach.

3.1 Data Pre-Processing

Stack overflow is typically a discussion forum where users often ask queries in a specific context and therefore the code snippets associated with the question can appear incoherent and unstructured if analysed independent of the title and description of the respective post. Some snippets are as small as having less than two lines of code, while some contain noise in terms of large comments and empty lines. To this predicament, we start by removing those anomalies followed by deletion of duplicate rows, NaN values and unnecessary stop words which don't add any relevant information to the model.

3.2 Regex Based Tokenization

Machine learning classifiers operate on numerical feature vectors and therefore it is necessary to transform the code snippets into feature vectors before they can be classified. However, prior to vectorization, we need to tokenize our data that would comprise the vocabulary of our model.

Earlier works have treated code snippets as regular text and simply removed white spaces and punctuation. However, since source code is different from regular language, we chose to customize Sklearn's tokenizer to better complement our requirements.

To describe the pattern of a token, we make use of regular expressions on three sub levels, i.e., for identifying the keywords, the operators and braces. Table 1 demonstrate one set of all the three corresponding regex.

Table 1: Regex patterns and their corresponding target keywords with example.

Pattern	Corresponding target	Sample	Expected tokens
[A-Za-z_]\w*\b	Keywords, identifiers, variables etc.	import matplotlib.pyplot as plt	['import', 'matplotlib', '.', 'pyplot', 'as', 'plt']
[@\#\&*\+^_\-\.$%\/<=|\~!\:>\?]+	Operators	'x-= 7-5'	['-=', '-']
[\)\,;\{\}\[\]`t\(""]	Spaces, Braces and Tabs	plt.show();	['(', ')', ';']

3.3 Vectorization

Post tokenization, we are left with a vocabulary of size '*N*', where *N* denotes the total number of unique tokens comprising our code snippet corpus. Next, we align a particular index to all the tokens by transforming each snippet into an array of size *N*. Figure 2 depicts the process of vectorization on a sample snippet.

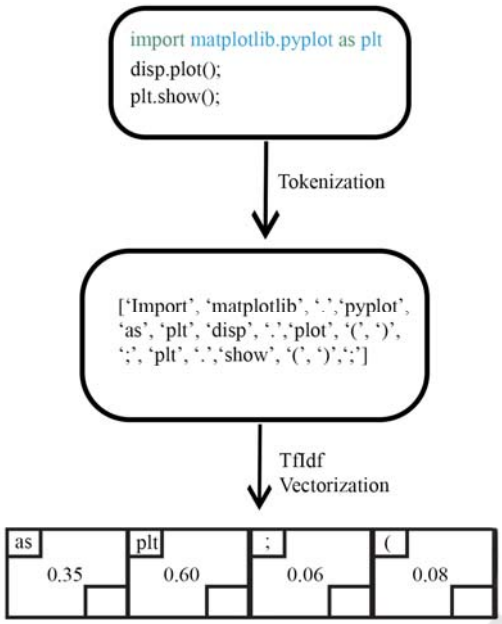

Figure 2: Tokenization followed by vectorization on a sample code snippet.

For our work, we choose Term Frequency – Inverse Document Frequency (TF-IDF) vectorizer to assign indices to all the respective tokens present in the snippet. The main idea of TF-IDF is to measure the relevance of specific word in proportion to the frequency of its appearance in the entire corpus. This potent ability of tf-idf to identify distinguishable words in a corpus makes it suitable for our code classification task as various keywords and identifiers can be easily recognized which can aid the classifier in segregating SO posts.

Equation (1 – 3) respectively depict the formula used for calculating tf and idf score of a word in a document followed by the combined tf-idf factor:

$$Tf(k,i) = \frac{Count\ (W_{k,i})}{\sum_j Count\ (W_{k,i})} \quad (1)$$

$$Idf(w_i) = log\ \frac{|D|}{|\{j|W_i \in D_j|} \quad (2)$$

$$Tf\text{-}dif(k,i) = tf(k,i) * idf(w_{k,i}) \quad (3)$$

Here, 'D' represents the entire corpus while D_j, denotes the j^{th} snippet in the corpus. W_i donates the i^{th} word of the vocabulary based on tokenization, and $|\{j|W_i \in D_j|$ represents the total count of sample snippets containing the word W_i. $W_{k,i}$ denotes the i^{th} frequency of the vocabulary in the K^{th} text. The count of the i^{th} word in the K^{th} text is denoted by $Count\ (W_{k,i})$ and $\sum_j Count\ (W_{k,i})$ represents the summation of the word frequencies of all words in the vocabulary in the k^{th} sample snippet. Finally, equation (3) is used for calculating the tf-idf factor of each token.

3.4 Dimensionality Reduction

Since our experiment subject is vast and diverse, the generated features are expected to be similarly huge as well. However, many a times, the feature matrix is occupied by irrelevant tokens that don't add up in the classification process in true sense. To this predicament, we make use of 'chi-square' or 'chi2' feature selection method (McHugh and Mary, 2013) to filter our impertinent data.

The chi2 technique aims to measure the degree of independence between a specific feature (tokens of the snippet in our case) and its respective class (programming language in our case).

After performing the chi-squared test, we keep only selective features having a certain p-value.

3.5 Model Selection

After performing the chi-squared test, we are left with the subset of distinguishable features which would be fed to a suitable classifier.

Based on previous studies, we selected two prominent ML based classifiers for our work, namely - Random Forrest (Breiman, 2001) and XG Boost (Chen et al., 2016) to better compare with the existing baseline.

We also employ neural networks as an alternative to our ML classifiers. Although, in terms of structure, logistic regression can be regarded as a basic neural network with no hidden layer, still we wish to investigate if neural networks could make a significant difference or not.

Finally, for our comparative analysis, we train our corpus on word2vec and distil Bert as separate tasks.

4 RESULTS AND COMPARITIVE ANALYSIS

4.1 Dataset

We gather the snippets for our experiment from the SO post corpus gathered by (Alreshedy et al., 2020). The corpus consisted of a total 232,727 questions varying across 21 different programming languages. These languages which include - Python, Objective-C, Bash, Ruby, Perl, C++, Lua, CSS, Markdown, Java, HTML, , C#, Scala, JavaScript, PHP, C, R,

SQL, , VB.Net, Swift and Haskell comprise 80% of questions on SO (Developer, 2017).

4.2 System Settings

We employ Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz 3.50 GHz with 32 GB Ram and 2GB NVIDIA Quadro K620 GPU with running OS of windows 10 for all the experiments.

We implement our model on keras framework with Tensorflow backend on a Jupyter notebook. For implementing distil-Bert, we make use of python library transformers made available through Hugging Face Corporation (Wolf et al., 2019).

4.3 Experimentation

In this subsection, we would elaborate on the various steps carried out during the experiment.

As fig 1 depicted the methodology, the first step is to pre-process the data. We start with our customized tokenization based on regular expressions.

Once the corpus is transformed into individual tokens, we apply tf-idf vectorization on them. We limit the maximum number of features up to 5000 tokens based on the frequency of their appearance in the corpus.

Further, to select the subset of relevant features, we perform the chi-squared test with p value of 0.95 that reduced the number of features to 3150. Some of the most statistically pertinent keywords are listed in table 2. The final filtered features are then passed out to the ML and Neural Network classifiers.

Earlier works have shown the efficacy of Random Forrest and XG Boost algorithms for this task. To have a better comparative analysis with the existing baselines, we decided to compete both these models.

For this purpose, we employ Sklearn's GridSearchCV and pipeline method to select the best estimators and hyper parameters (Pedregosa et al., 2011). To keep the computational cost low, we perform grid search over 25% of our dataset.

Finally, we deploy our winner classifier by unpacking the best parameters tuned through grid search, which in our case was Random Forrest which is in line with earlier work of (Alreshedy et al., 2020) as well.

Additionally, we also implement artificial neural network with 10,000 nodes in the input layer, 64 in the hidden layer and then 21 dense layers followed by a sigmoid function for classification. We make use of ADAM optimizer which is an optimized version of 'RMSProp' and 'momentum' combined followed by categorical cross entropy for fitting the model.

Table 2: Top keywords filtered after chi-square test.

Languages	Top features
bash	bin bash, then, grep, sh, fi, awk, done, sed, bash, echo
C	fopen, gcc, malloc, define, int, sizeof, struct, char, void, printf,
C#	get set, using, ilist, assembly, private void, typeof, ienumerable, foreach, public, new
C++	operator, endl, int, push_back, const, cpp, boost, void, std, cout
Css	width, moz, hover, border, style, webkit, margin, div, background, css
haskell	io, haskell, ghc, xs, cabal, hs, do, otherwise, where, let,
java	inputstream, synchronized, javax, jsp, system out, public, public void, extends, new, java
Java_script	settimeout, new date, document, javascript, js, prototype, onclick, alert, function, var
lua	torch, require, nil, function, lua_state, print, then, end, local, lua,

We keep the train-test split ratio to 75-25 percent for an optimum analysis.

Since neural networks cannot handle sparse data straight away, we convert the tf-idf vectors into dense array. However, owing to the huge size of the dataset, we pass the data in batches by creating an iterable generator object.

Further, we manually fine tune the hyper parameters such as changing the learning rate, increasing the number of hidden nodes, adding a drop out layer and so on for an optimized result.

4.4 Results of Proposed Methodology

After the posts are classified through respective ML and NN classifiers, we plot the confusion matrix and evaluate the performance. While Random Forrest outperformed other ML classifiers in grid search, NN even performed better than RF. However, the computational time of NN was slightly more than the RF classifier, so in a sense the trade-off was comparable. Still, for performance considerations, we chose to proceed with our ANN classifier.

We evaluate the performance for all the languages individually as well as collectively based on four metrics, namely accuracy, precision, recall and F1 score. The confusion matrix of the ANN classifier along with the individual prediction performance can be found in supplementary information on the following link - https://tinyurl.com/2p9b4bhb.

Table 3: Overall performance of the classifier.

Classifier/Metric	ANN	RF
Precision	85.60%	83.40%
Recall	81.90%	80.20%
F1 Score	82.50%	81.30%
Accuracy	85.04%	83.80%

4.5 Comparative Analysis

In this subsection, we compare our model's performance with earlier existing baselines and also investigate whether embedding based techniques and state of art models like Bert can perform better on source code or not.

Most of the earlier works have either considered snippets insignificant to be included or have merged their features with the features extracted from body and description of the post. However, we can still make an overall comparison keeping in view the aim of achieving the end goal to classify the programming language of code snippets.

However, identifying the language in large source code files is very much different from classifying snippets and therefore an equivalent comparison can't be made between the two.

Table 4 shows the overall comparison of similar works in the domain. Although (Alreshedy et al., 2018; 2020) in their work achieve better performance overall, but it is to be noted that when applied on the snippets alone, their accuracy dropped and our model surpasses their work. To combine the effect of textual part of the body and description is our future work. Table 5 further draws comparison between the work of (Alreshedy et al., 2020) and our work on individual programming language level.

Besides, our investigation concerning the performance of embedding based approaches and pre-trained models are also presented in table 6. The training-validation curve diagram regarding the same can be found at https://tinyurl.com/2p9b4bhb.

Table 4: Comparative analysis of our approach against existing baselines for identifying tags in SO posts.

Study	Dataset based on	Acc, Pr, Rc, F1
(Kuo, 2011)	Title and Description	47%
(Saha et al., 2013)	Only Title	68%
(Stanley et al., 2013)	Title and Description	65%
(Baquero, 2017)	Title and Description	60.8%, 68%, 60%, and 60%
(Alreshedy et al., 2018)	Title and Description Title, Description and code Snippet Only code Snippets	81%, 83% 81%, and 0.81%; 91.1%, 91%, 91% and 91% 73%, 72%, 72%, 72%
(Saini and Tripathi, 2018)	Title, Description and code Snippet	65%, 59%, 36%, and 42%
(Jain and Lodhavia, 2020)	Only Title	75%, F1 Score-81%
(Kavuk and Tosum, 2020)	Title and Description	75%, 62%, 55% and 39%
(Alreshedy et al., 2020)	Title and Description Title, Description and code Snippet Only code Snippets	78.9%, 81%, 79% and 79% 88.9%, 88%, 88% and 88% 79.9%, 80%, 80%, 80%
Our work	**Only Code Snippets**	**85%, 85%, 81% and 82%**

4.6 Discussion

The significant performance improvement of our tf-idf approach over existing works can be attributed to the decisive pre-processing of the corpus based on regular expressions and feature reduction technique. Neural network classifier further outperforming ML based classifiers also adds to the boost of performance.

Further, from table 4 and 6, we can see that out of all the 21 languages, HTML and Markdown performed worst. While Markdown had only 1300 snippets as compared to 12000 snippets of other 20 languages which very much explains it low performance due to lack of training data, the reason for bad performance of HTML could be attributed to the ambiguity of HTML snippets which simultaneously contain CSS and JavaScript code

Table 5: Comparative analysis of F1 score of our approach against existing baselines for individual programming languages (all results in percentage).

Classifier/ Model	PLI (PLI tool, 2018)	SCC (Alreshedy et al., 2018)	SCC+ (Alreshedy et al., 2020)	Our Work
lua	50	84	70	**92**
C	56	76	81	**84**
ruby	43	70	72	**89**
C#	51	79	78	**80**
C++	65	51	73	**84**
python	69	**88**	79	87
R	72	77	78	**88**
Css	30	**86**	77	80
Vb.net	60	83	77	**91**
swift	54	84	89	**96**
haskell	67	89	78	**93**
Html	35	54	**55**	53
Sql	50	65	79	**83**
Java	46	70	76	**81**
markdown	28	76	**91**	32
bash	67	76	**85**	82
objectivec	77	57	88	**93**
Perl	69	74	41	**90**
PhP	62	74	**88**	85
Scala	72	76	81	**94**
Javascript	48	**78**	74	75

Table 6: Comparative analysis of our approach against Word2Vec and Distil-Bert.

Classifier/ Metric	Tf-Idf +ANN	Word2Vec over Bi-LSTM	Distil-Bert
Accuracy	85.04%	68.30%	61.2%
Time taken	139.4s	3936.6s	36996s per step

segments which leads to miss-classification. One possible solution to fixing this anomaly could be to increase the training data for Markdown and to generate multiple tags for HTML snippets and not consider the miss-classification count of JavaScript and CSS tags.

Elsewhere, the performance of tf-idf approach excelling the embedding based approaches can be attributed to the fact that code classification is very much independent of semantic association of words as compared to regular text.

One may argue that recent advances that have transpired codebert (Feng et al., 2020), which is specifically trained on code corpus comprising six languages can outperform traditional methods. In this regard, we wish to point out that the computational time needed for distil bert only was around 1000 times more than that of tf-idf approach with a RF classifier, and distil bert is the most preliminary version of the Bert series. Therefore, keeping the basis of performance and computational time trade-off, we can safely argue that traditional approaches are more suited for code classification purposes.

However, a proper in-depth investigation in this regard can hold a promising future work.

5 CONCLUSION

Stack Overflow is one of the foremost resources for developers to seek technical assistance. The site receives massive traffic on a daily basis and thus needs a proper mechanism for post segregation. Earlier works have focused mainly on classification of large source code files from github repositories. Since prediction of programming languages in stack overflow posts is relatively much challenging task, we have proposed our model trained on regex-based tf-idf vectorizer over an ANN classifier. We achieve decent accuracy of 85% which excels several baselines in the task of code snippet classification.

Further, we also investigate the utility of embedding based algorithms such as word2vec and pre-trained models such as distil-bert. Our investigation shows that traditional approaches are much suited for source code classification tasks owing to the lack of semantic dependence in code scripts.

REFERENCES

Alreshedy, K., Dharmaretnam, D., German, D. M., Srinivasan, V., & Gulliver, T. A. (2018). Predicting the Programming Language of Questions and Snippets of StackOverflow Using Natural Language Processing. arXiv preprint arXiv:1809.07954.

Alrashedy, K., Dharmaretnam, D., German, D. M., Srinivasan, V., & Gulliver, T. A. (2020). Scc++: Predicting the programming language of questions and snippets of stack overflow. Journal of Systems and Software, 162, 110505.

Barua, A., Thomas, S. W., & Hassan, A. E. (2014). What are developers talking about? an analysis of topics and trends in stack overflow. Empirical Software Engineering, 19, 619-654.

Baquero, J. F., Camargo, J. E., Restrepo-Calle, F., Aponte, J. H., & González, F. A. (2017, September). Predicting the programming language: Extracting knowledge from stack overflow posts. In Colombian Conference on Computing (pp. 199-210). Springer, Cham.

Breiman, L. (2001). Random forests. Machine learning, 45, 5-32.

Cao, K., Chen, C., Baltes, S., Treude, C., & Chen, X. (2021, May). Automated query reformulation for efficient search based on query logs from stack overflow. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (pp. 1273-1285). IEEE.

Chen, T., & Guestrin, C. (2016, August). Xgboost: A scalable tree boosting system. In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining (pp. 785-794).

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.

Gilda, S. (2017, July). Source code classification using Neural Networks. In 2017 14th international joint conference on computer science and software engineering (JCSSE) (pp. 1-6). IEEE.

Jain, V., & Lodhavia, J. (2020, June). Automatic Question Tagging using k-Nearest Neighbors and Random Forest. In 2020 International Conference on Intelligent Systems and Computer Vision (ISCV) (pp. 1-4). IEEE.

Kavuk, E. M., & Tosun, A. (2020, June). Predicting Stack Overflow question tags: a multi-class, multi-label classification. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (pp. 489-493).

Klein, D., Murray, K., & Weber, S. (2011). Algorithmic programming language identification. arXiv preprint arXiv:1106.4064.

Khasnabish, J. N., Sodhi, M., Deshmukh, J., & Srinivasaraghavan, G. (2014, July). Detecting programming language from source code using bayesian learning techniques. In International Workshop on Machine Learning and Data Mining in Pattern Recognition (pp. 513-522). Springer, Cham.

Kuo, D. (2011). On word prediction methods. Technical report, Technical report, EECS Department.

McHugh, M. L. (2013). The chi-square test of independence. Biochemia medica, 23(2), 143-149.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. the Journal of machine Learning research, 12, 2825-2830.

Pennington, J., Socher, R., & Manning, C. D. (2014, October). Glove: Global vectors for word representation. In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP) (pp. 1532-1543).

Programming language identification tool, 2018. Available: https://www.algorithmia.com [Online].

Rekha, V. S., Divya, N., & Bagavathi, P. S. (2014, October). A hybrid auto-tagging system for stackoverflow forum questions. In Proceedings of the 2014 International Conference on Interdisciplinary Advances in Applied Computing (pp. 1-5).

Saha, A. K., Saha, R. K., & Schneider, K. A. (2013, May). A discriminative model approach for suggesting tags automatically for stack overflow questions. In 2013 10th Working Conference on Mining Software Repositories (MSR) (pp. 73-76). IEEE.

Saini, T., & Tripathi, S. (2018, March). Predicting tags for stack overflow questions using different classifiers. In 2018 4th International Conference on Recent Advances in Information Technology (RAIT) (pp. 1-5). IEEE.

Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. arXiv preprint arXiv:1910.01108.

Stanley, C., & Byrne, M. D. (2013, July). Predicting tags for stackoverflow posts. In Proceedings of ICCM (Vol. 2013).

Swaraj, A. and Kumar, S. A Methodology for Detecting Programming Languages in Stack Overflow Questions. DOI: 10.5220/0011310400003266. In Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)

Van Dam, J. K., & Zaytsev, V. (2016, March). Software language identification with natural language classifiers. In 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER) (Vol. 1, pp. 624-628). IEEE.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., ... & Rush, A. M. (2019). Huggingface's transformers: State-of-the-art natural language processing. arXiv preprint arXiv:1910.03771.