Dardish Code:
Extracting Information About Code from Relevant Comment Block

Abdallah Dandashli Mohammad Abou Harb Omar Al Jaroudi
Department of Computers and Communications Engineering
American University of Beirut
{ayd06, maa299, oaa20}@mail.aub.edu

ABSTRACT

Comments, which are contracts between the developer and the client, can convey insufficient or sometimes misleading information about their relevant code block. Insightful and clear comments can be critical for maintainability of code, given that software developers are often obliged to work with pre-written code, whether through working in a team or improving legacy code. In this paper, we address the problem of comments related to function blocks, that do not convey enough information about the function's signature, namely its parameters. Our model has achieved significant results in predicting a function's parameters from its relevant comment block, a feature which can then be used to evaluate the integrity of the written comment and provide recommendations as to if it can be improved.

1. INTRODUCTION

All programming languages support comments; they help simplify complicated chunks of code by pointing out the key ideas (e.g. recursive implementation) and highlighting any special impact the code might have on other parts of a program (e.g. mutability). If written properly, such comments can assist developers in catching errors or simply improving parts of the code in general, and in reusing a piece of code somewhere else without necessarily going into details of its implementation.

For function implementations, an important feature that should be highlighted in the comment, is the parameters the function takes. A developer that wants to use a pre-written function like a black box without understanding its small details would look at the comments to understand the function signature; what each parameter means. So, it would be problematic if the comment doesn't even make mention of these parameters, for instance. In our compiled dataset, which we will discuss later, around 7% of function implementations have comments that fail to mention any of the function's parameters.

Accordingly, our objective was to be able to predict a function's parameters from its corresponding comment. If we can do this, then we can deem this comment as insightful or not, as far as parameters are concerned. If the parameters cannot be automatically inferred from the comment, then either the developer fails to mention them, or the language they have used can be improved or made clearer.

By using a Decision Tree Model on a set of 6 features for each word, we were able to correctly predict the parameter names from the comment blocks with an accuracy of 98 % and an F1 score of 0.91.

2. RELATED WORKS

Similar works are those related to the field of Named Entity Recognition (NER), which is a subtask of information retrieval. Its objective is to locate and classify named entities in unstructured texts into predefined categories.

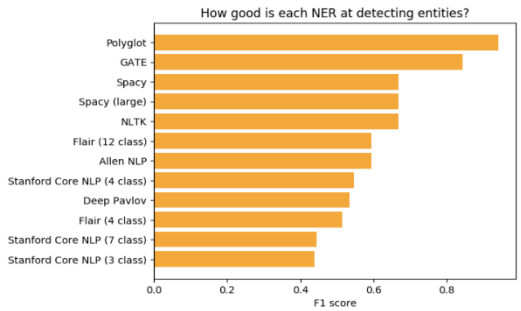


Fig.1 Different NER system F1 average scores

Among the NER systems, one system stands out from its F1 score computed for predicting three types of entities: People, Locations and Dates. Polyglot's NER system (Al-Rfou et al., 2014) takes a different approach than most NER systems by avoiding human-annotated training datasets and using unlabeled datasets with automatically inferred labels. Labeling is done via several features such as hyperlinks in Wikipedia articles, if the link points to an article identified as an entity article then the text is included as a positive training example. This "language independent extraction" method has "boosted the performance by at least 45% on human annotated gold standards".

SpaCy[1], on the other hand, is a free open-source library for Natural Language Processing in Python. NER is just one of its several features. It uses several statistical models to increase its prediction performance such as binary weights, lexical entries, data files, word vectors, and configuration options. What is special about it, nonetheless, is that it is implemented in a way that gives each word a unique representation for each distinct context it is in.

3. DATASET

We collected data from source code of libraries written in Python, which were available on GitHub. This was motivated by the fact that these libraries are often quite large and well-documented. Namely, the libraries we extracted functions from are: NLTK, SciPy, TensorFlow, and Sklearn. To capture each function block on its own, we used regular expression (1), knowing that functions in Python always begin with the keyword "def". In total, we extracted 1736 function blocks with their relevant comment block, having discarded all the functions that do not take any parameters.

```
Regex = \t*def\s*.*\(          (1)
Regex = """|'''                (2)
```

To separate each code block from its corresponding comment block, we exploited the fact that each function has its comment written inside its scope, and was surrounded by three quotations, which is used to write multi-line comments in Python. Regular expression (2) allowed us to capture this. Then for each code block, we used the Abstract Syntax Trees[2] (AST) module, which allowed us to extract the function name and its parameters.

4. PREPROCESSING COMMENTS

The first step was to tokenize the comments, that is to generate a list of word tokens for each comment, that would exclude separators and special characters. Then we omitted a list of English stop-words, after verifying of course that none of them occur as parameter names in all the dataset. We also removed all words starting with numerical digits, since it is illegal to name parameters that way in Python. Now we have each comment as a list of tokens, which are the words that are still relevant to us.

For each word-token, we generated a vector of features that we would later use in our chosen model. We realized each of these features through intuitive hints, such as exploiting human tendencies when writing code, and then verified that each feature was statistically relevant to our objective.

4.1 Word itself

For each word in a comment block, we assumed the word itself to be a feature. The logic behind this being that across the four libraries which we extracted data from, we noticed a tendency to use some common words as parameter names. In fact, out of 5000 parameter names mentioned in the comments of our dataset (disregarding repetition of parameter names in the same comment), the number of unique parameter names was just above 1000. This means that 4/5 of the parameter names being used across the comments of the dataset are not unique.

4.2 OUT-OF-LEXICON

Our first feature was whether the word belonged to the English lexicon or not. Usually, developers tend to use parameter names that are outside the lexicon (e.g. containing underscores, concatenating two words etc.) to portray what this

[1] https://pypi.org/project/spacy/

[2] https://docs.python.org/3/library/ast.html

parameter is used for, for readability and debugging purposes etc. To this extent, we used PySpellChecker[3], which is a library that uses classical NLP techniques to check for spelling of words and suggest corrections for misspellings. As such, for each word, we assigned a Boolean value for whether it was in the English lexicon, or a misspelled word. Statistically, around 60% of the parameters turned out to be out-of-lexicon words. Thus, we saw that this feature was relevant to our objective.

4.3 PYTHON-POS-TAG

Another intuition we had regarding parameters is that it is illegal to use Python's predefined types and keywords as variable names, so we knew for a fact that any function parameters would need to be words outside the list of reserved Python keywords and data-types. As such, we annotated each word in a comment as either a "PythonPOS", which meant that it is a Python keyword, or "NotPythonPOS", which meant that it wasn't.

4.4 NORMAL POS-TAG

Using NLTK's built in POS tagger, we tagged the part-of-speech for each word. Our collected dataset happened to include all possible POS tags, which are 35 in total. Statistically, 93 % of the parameters turned out to be either nouns, verbs, or adjectives, in their different forms (plural, singular etc.). As such, we appended to the feature vector of each word, the word's POS tag.

4.5 RELATIVE INDEX

Even though our dataset included comments from different libraries, we had to take context into account. This means that every word had to be attributed to a certain comment, or else the model would perceive the dataset as just one big comment. As such, we appended for each word the comment it occurs in, which is accomplished by encoding each comment to a unique number. The encoding scheme used to handle all these features will be discussed later.

4.6 TF-IDF

We knew, before attacking the problem, that parameters names are words of high importance in a comment; even though the author might use them more than once in a comment, they won't be used so much as to be considered jargon words. To make use of this importance, we computed the TF-IDF score for each word as:

$$w_{t,d} = (1 + \log tf_{t,d}) \times \log_{10}(N / df_t) \qquad (3)$$

Doing this would allow us to emphasize the significance of parameter names and rule out the words that occur frequently in every document, which would naturally have a lower score.

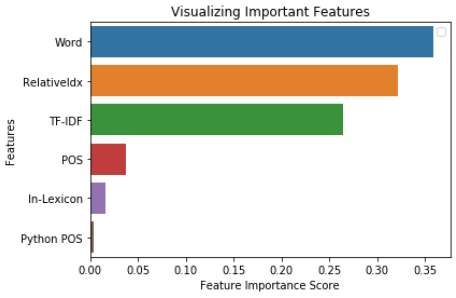
5. DECISION TREE MODEL

The model used was a decision tree classifier. A decision tree makes a series of splits on our data based on which feature maximizes our information gain. Equation (4) is the formula for calculating information entropy. Information gain is the entropy removed between two decisions in the tree.

$$E = -\sum_{i}^{C} p_i \log_2 p_i \qquad (4)$$

Using Sklearn, we designed an encoding scheme for each of the features, whereby each value would map to a unique integer. The decision tree was then fed the encoded values of the aforementioned features(4.1-4.6) and run on several standard splits of our data.
The feature importance can be seen below and give us insight as to which feature minimizes the entropy of our data and maximizes our information gain.

Fig. 2



[3] https://pypi.org/project/pyspellchecker/

6. RESULTS

The metrics we used to evaluate the performance of our decision tree where precision, recall, and a combined f1 score.
The results of the model can be seen below:

Fig. 3

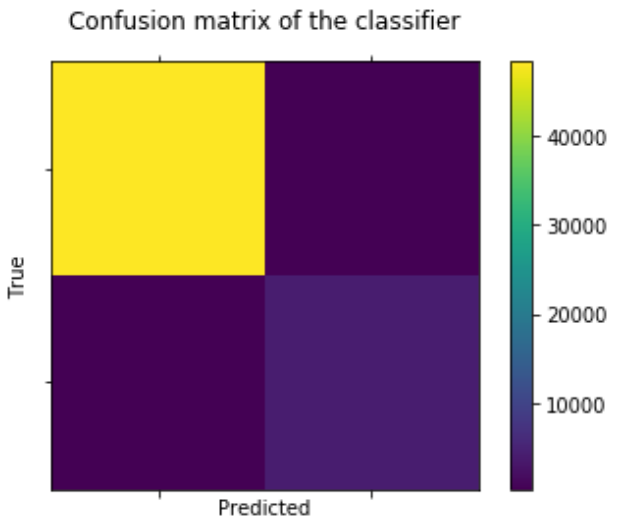


Fig. 4: Performance metric scores

	precision	recall	f1-score
0	0.99	0.99	0.99
1	0.90	0.91	0.91
accuracy			0.98
macro avg	0.95	0.95	0.95
weighted avg	0.98	0.98	0.98

Note in figure 4 that a word was tagged "1" if it's a parameter name, and 0 otherwise.
In addition, the optimal depth of the tree was found to be 30. This was found by iteratively increasing the depth starting from 1.

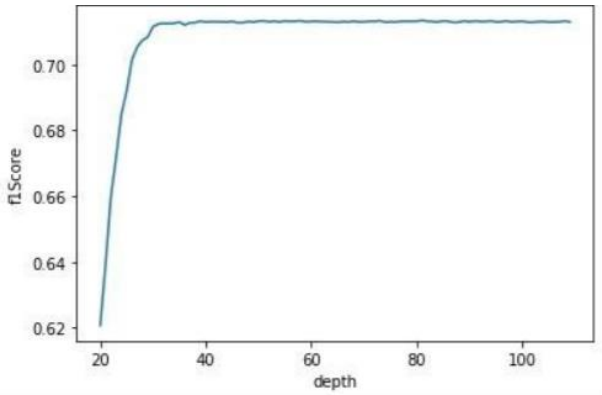


Fig.5: Variation of f1 score versus tree depth

7. FAILED ATTEMPTS

Here, we will list several other alternatives that we tried but did not produce any significant results.

7.1 BAYESIAN NETWORK

Before settling on a decision tree model, we tried to construct a Bayes Belief Network, which is a probabilistic graph model that establishes causality between events whose conditional probabilities are bigger than a certain threshold. As such, we had to find a probabilistic relation between our selected features and the fact that a word is a parameter. However, the conditional probability that a word is a parameter given the occurrence of any of the features or a combination of them for that matter, was insignificant, and so we could not construct such a network.

7.2 N-GRAMS

We also tried to make use of what is in the vicinity of a parameter name in a comment, driven by the intuition that a parameter name would likely be followed by its type for example. And so, we tried to implement an n-gram model. On its own, it did not yield any significant results that we could use, even if to rule out certain words as certainly not parameter names. We also tried integrating it in our decision tree model, but it ended up decreasing the performance of the model, which we attributed to the fact that an n-gram exponentially increases the number of possible feature combinations.

8. CONCLUSION

Based on the selected features, namely a word, its part-of-speech, whether it's a reserved keyword or not, whether its in the lexicon or not, its relative index, and its tf-idf score, we constructed a decision tree model that can decide whether this word is a parameter name or not, with 98% accuracy.

Such a prediction can be used to evaluate the clarity and usefulness of comments written by software developers, which is important for cases where a client might have access to the comment and the header of a function and not its implementation.

To further capitalize on this information, a possible course of action would be to implement the

same model to predict the return type of the function from its comment block. Further insight can also be provided by analyzing the objective that the function accomplishes from the high-level English description also contained in the comment block. Nevertheless, these steps would require manual annotation of each word of the comment dataset in the training phase, which can prove tedious.

That way, however, we would be able to systematically and consistently generate a skeleton of the function from its relevant comment, which can be viewed as an even higher-level way of writing code; a person would write a clear enough comment in natural language, and the computer can translate it into more machine-friendly instructions. At this point the machine can systematically convert these instructions into low-level machine code. This way a person would be effectively programming in natural language without necessarily sticking to classically rigid syntax.

9. REFERENCES

Al-Rfou, Rami & Perozzi, Bryan & Skiena, Steven. 2014. Polyglot: Distributed Word Representations for Multilingual NLP. *Proceedings of the Seventeenth Conference on Computational Natural Language Learning.*