# A BASELINE INTERPRETABLE APPROACH IN CODE UNDERSTANDING

## Contents

## Introduction

Code is an inevitable constituent of software education and software products. From a student in an algorithms and programming class to a developer writing a function to extract data, to a scientist designing a prototype with the help of previously written code to save time on non-novel sub-operations - code is a substantial part that demands care and precision. In light of this, a tool that aids the development and understanding of what a piece of code is performing, and more importantly, where it may be erring and may be improved, would be a value-add to the code development process - at the very best revealing errors directly in conflict with functional requirements, and at the very least enabling tracking and information provision for code audits and future collaborators.

However, there are a few significant hurdles in the way of this goal.

One is that software testing is computationally hard, and as an extension, so is software understanding (which relies on verifications, as in, tests, to explain results). Hence, this cannot be approached as a solely deterministic problem. This is where data comes into the picture. With the augmentation of a variant-enough codebase that solves a given problem - it is possible to improve confidence in explaining code with respect to the information derived from a code sample's peers (better or worse).

With the code data comes the responsibility of identifying causal features that directly and as closely as possible alter the results of a code, while still being general enough descriptors to be used in an algorithmic manner for pattern recognition. This is the second hurdle to accomplish and will require careful and researched feature engineering.

The last hurdle is in evaluating the effectiveness of such an endeavor - checking long codes of huge amounts manually is inhumanly tedious, while having a system check the algorithm is equivalent to having a testing system for a software tester - again a computationally hard problem. Hence, you will find that there are no massive codebases pertaining to a single solution of a problem where errors and summaries for each code sample are documented in a standard parsable format. Which means we have a benchmarking issue. In such a case, the best evaluation possible can be achieved through intelligent sampling on test datasets.

## Problem Statement

The goal of this endeavor is to solve the problem of code understanding to a minimum extent that renders the solution useful and worth augmenting current coding practices with.

**This project attempts to explain unseen yet valuable similarities in a codebase of functions all solving a single problem; this is done with the help of patterns that reveal logical flaws in a more detailed and specific manner than available today, and that have a causal relationship with the efficiency or quality of the code in question.**

There are four information sources available to comprehend an unknown piece of code, given a reference correct code (or the abstract makings of a reference). These four sources are -

1. Unit Tests - For correctness
2. Efficiency Tests - For space and time complexity
3. Syntactic Pattern Verification - For complexity, correctness and quality
4. Intermediate Data Tracing - For complexity, correctness and quality

This project will focus on syntactic patterns and descriptors for understanding code. We do not focus on correctness in our project, rather **we focus on building this as an auto-code review tool**. The project is an example of a **human-in-the-loop machine learning project** where initial input is a negligible time (around five-ten minutes) spent on providing seed inputs for engineering features, and then **the model reveals code quality patterns in a large number of files that a code reviewer has never seen before**. **We are going to assess the validity of this method and whether it can be trusted, and how falses must be handled in future.**

This method turns out to also be scalable, because these features may be reused across different problem solving codebases and while the file numbers may grow, the overall distribution for such code problems doesn't change very much.

## Data Source

The dataset being used to demonstrate this proof of concept is from an open source repository of Java functions where all pass unit tests but some have significant efficiency and quality issues.

*Link - https://github.com/IBM/Project_CodeNet#contributors*

*Credits -*

*CONTRIBUTORS FROM IBM - Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, Frederick Reiss*

The repository is open sourced under the **Apache 2.0 License**.

From this dataset, we choose the dataset for the TypeWriter puzzle that solves the following problem using Java. This is what the problem statement asks -

**—--**

***A string is given as a parameter. It contains the following characters - '0', '1', 'B'. The 'B' stands for backspace, which means that if that is typed in, a backspace operation must occur on that string as per its current state. The result is a printed output on the console that shows the final string.***

**—--**

## Methodology

The methodology covers five stages -

1. Research
2. Preprocessing
3. Feature Engineering
4. Clustering Algorithm Set-Up and Implementation

Let us look at each in more detail -

**RESEARCH -**

**Input** - Codebase characteristics, and set of all possible features..

**Method** - Study of all different relations possible pertaining to the problem and introspection on how to handle importance of features and combined meaning in an automated manner.

**Output** - A custom designed algorithm in pseudocode form that mathematically elaborates step by step how to create code file clusters of similar approaches, with reason.

**PREPROCESSING -**

**Input** - Codebase.

**Method** - Convert all files to XML format, and create an XPath query machine.

**Output** - XML database and syntactic query machine.

**FEATURE ENGINEERING -**

**Input** - XML database, query machine, domain research results, XPath expert knowledge.

**Method** - This is a human in the loop endeavor where we create a limited set (not more than 20) positive queries to capture the right necessary, sufficient and exclusive chain of control across variables and manipulations. Perform this for the few best files (of different types) that coded it right. This is the base set - a deterministic algorithm generates the recursive sub-clause negative query. This query is the actual feature to be used since it directly talks about a missing relation on code. The weight of this feature is determined by how many files did it identify with that also failed unit tests and how far along the query did the first negation occur (as in, where did the first missing relation to get to good code occur). This logic is subject to change though.

**Output** - A set of engineered features in the form of queries that find patterns on the XML database, with a weight derived from the frequency of failures observed with its existence. If a base positive XPath query had a nested structure of level $\ell$ with an average of $n$ sub-clauses at each level bound by conjunction, then total number of negative queries $m$ generated would be -

$$m = \ell(n + 1)$$

This can be imagined to be achieved from negating one of the subqueries within the conjunction at each level and one extra negation on the whole conjunction at that level (hence, $n + 1$), and then that done starting from the top level to the bottom. Hence, the factor of $\ell$.

The other part of this output is to create a data structure that maps the key XML file to the XPath negation queries (features) based on which files they hit.

As described above, these are the base queries -

***'0':***

```
"//unit[count(.//*[./st[text()='loop'] and not(ancestor::*[./st[text()='loop']])and
not(.//*[./st[text()='loop']])])=1]/(.//*[./st[text()='loop']])[1]",
```

**'1':**

```
"//unit[count(.//*[./st[text()='loop'] and not(ancestor::*[./st[text()='loop']]) and
not(.//*[./st[text()='loop']])])>1]/(.//*[./st[text()='loop']])[1]",
```

**'2':**

```
"//*[./st[text()='loop'] and .//*/st[text()='loop']]",
```

**'3':**

```
'//if_stmt//if/condition//literal[text()=(\'"B"\',"\'B\'")]',
```

**'4':**

```
'//if_stmt//if/condition//literal[text()=(\'"1"\',"\'1\'",\'"0"\',"\'0\'")]',
```

**'5':**

```
"//if_stmt//condition[.//call//name[text()=('isEmpty','size')] or
.//expr[.//operator//text()=('>','<','>=','<=','!=') and .//literal//text()=('0','1')]]",
```

**'6':**

```
"//call//name[text()=('replace','replaceAll','replaceFirst')]",
```

**'7':**

```
"//call//name[text()=('find','findFirst','findAll','indexOf','contains')]",
```

**'8':**

```
"//call//name[text()=('pop','push','delete','remove','deleteCharAt','removeLast','removeFirst')]",
```

**'9':**

```
"//call//name[text()=('substring')]",
```

**'10':**

```
'//unit[count(.//function)=1]//function/name',
```

***'11':***

*'//unit[count(.//function)>1]/(//function/name)[1]',*

***'12':***

*'//unit[count(.//if)>2]/(//if)[last()]',*

***'13':***

*'//unit[count(.//if)<2 and count(.//if)>0]/(//if)[last()]',*

***'14':***
*'//unit[count(.//if)=0]/(//import)[1]'*

## CLUSTERING ALGORITHM SETUP AND IMPLEMENTATION

**Input** - { XML file : XPath query hit list } dictionary, { XPath query: weight } dictionary.

**Method** - The journey to correct code and passing all unit tests is progressive one and we have no target class here. Hence, we use an unsupervised method here - K-Means clustering with cross validation (to determine number of clusters) over a training set using the engineered features.

**Why KMeans - an unsupervised clustering method?**

Code review has multiple aspects to it - many which we do know about (such as avoiding *while True* loops and simplifying *if var == True* to *if var*, to some bigger quality issues like repeating expensive operations redundantly. But there are plenty (give a dataset) that we might not have an idea of beforehand, which is why we need the algorithm to be able to reveal this for us.

Now you might ask - we have these queries engineered and used as features - doesn't that give a sound idea? Well, no existence (or the lack thereof), does not help assess quality - instead what helps is the coexistence of certain features together.

Think of it this way - the more information on the page - the less each information has contributed to the overall quality (we'll be wrong about this in a certain case - which we'll see later), and the less information on the page, the more telling each piece is - in fact, the absence of information is quite telling as well, and so is the combination of absences and presences that don't seem to make sense. The last thing is the most revealing of all - for it doesn't just reveal the quality of a known coding algorithm - but often can reveal another unseen algorithm altogether used to solve the same problem.

Hence, part of the problem is to normalize the weights of the features as per the presence of the other features on the same code submission.

These submission-wise normalized weights will finally be used in clustering. We will do this over 80% of the files which in this case is around 160 code files.

Now the way this 80% has been selected isn't random. It's actually selected as those leftover after 20% is selected. Now these 20% (40 files) have been hand-picked to represent a variance of quality and efficiency in code, and will finally place them in the cluster closest in terms of Euclidean distance measured using their feature weights and see if they match their cluster tendencies.

**K-Means algorithm as adapted to our problem -**

Step 1 - Select ‘k’ - the number of clusters.

Step 2 - Select ‘k’ random points as initial centroids.

Step 3 - Assign each data point to the nearest centroid.

a. **If** the data points were already assigned before **and** there is no change in assignment, terminate the algorithm and return clusters.
b. **Else** move to step 4.

Step 4 - Calculate the new centroids based on this assignment.

Step 5 - Go back to step 3.

There are two ways I have used here to determine ‘k’, one is by taking the number of combinations of features present on data, and the second is the elbow method that determines which ‘k’ gives the most similar clusters. The elbow method computes the “Within Cluster Sum of Squares” or WCSS score -

$$\text{WCSS} = \sum_{\text{Cj = Centroidj}} \sum_{\text{Pi = Pointi in Clusterj}} \text{distance}(P_i\ C_j)^2$$

Finally, the test files are mapped to nearest clusters and evaluated by hand for its trustworthiness.

**Output** - ‘k’ clusters of XML files, which will be mapped back to the original code file.

## Evaluation

**Primary Results on Training Set -**

As per both the evaluation methods of ‘k’, the best results for the cluster number was between 15 to 18 groups - we picked 16.

Now this is a PoC and note that the original groups of each of these files in terms of code quality did not exist on their own - we are going to look at these groups for the first time and decide whether it is acceptable before we move to the test set.

A few examples (that actually occurred on this dataset) of falses in a cluster are this -

Clusters have files like this -

```java
import java.util.Arrays;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        char[] inputs = scanner.nextLine().toCharArray();
        char[] outputs = new char[inputs.length];
        int oIndex = 0;
        for (int iIndex = 0; iIndex < inputs.length; iIndex++) {
            while (iIndex < inputs.length && inputs[iIndex] == 'B') {
                oIndex--;
                iIndex++;
            }
            if (iIndex >= inputs.length) break;
            if (oIndex < 0) oIndex = 0;
            outputs[oIndex++] = inputs[iIndex];
        }
        System.out.println(String.valueOf(Arrays.copyOf(outputs, oIndex)));
    }
}
```

And this...

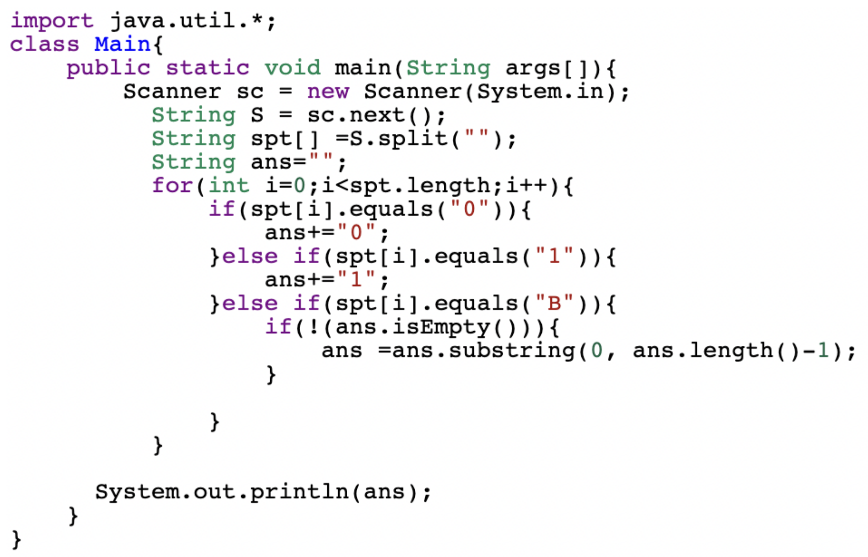
```java
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s = scan.next();
        scan.close();
        int check[]=new int[s.length()];
        for(int i=0;i<s.length();i++) {
            if(s.charAt(i)=='B') {
                check[i]=-1;
                for(int j=i-1;j>=0;j--) {
                    if(check[j]==0) {
                        check[j]=-1;
                        break;
                    }
                }
            }
            else check[i]=0;
        }
        for(int i=0;i<s.length();i++) {
            if(check[i]==0)System.out.print(s.charAt(i));
        }
        System.out.println();
    }
}
```

Also has a file like this -

```
import java.util.Arrays;
import java.util.Scanner;

public class Main {
public static void main(String[] args) {

        Scanner keyboard = new Scanner(System.in);

        String s = keyboard.next();

        String[] command = new String[s.length()];
        String[] result = new String[s.length()];

        for(int i = 0; i < s.length(); i ++) {
            command[i] = s.substring(i, i+1);
        }

        int jj = 0;
        int count = 1;

            for(int j = s.length() - 1; j >= 0; j--) {
                if(command[j].equals("B")){
                    while(j != 0 && command[j-1].equals("B")){
                        //一個前がBじゃないか調べる
                        count ++;
                        j--;
                    }
                    j -= count;
                }else {
                    result[jj] = command[j];
                    jj++;
                }
            }

        for(int k = jj - 1; k>=0; k --) {
            System.out.print(result[k]);
        }

        keyboard.close();
    }
}
```

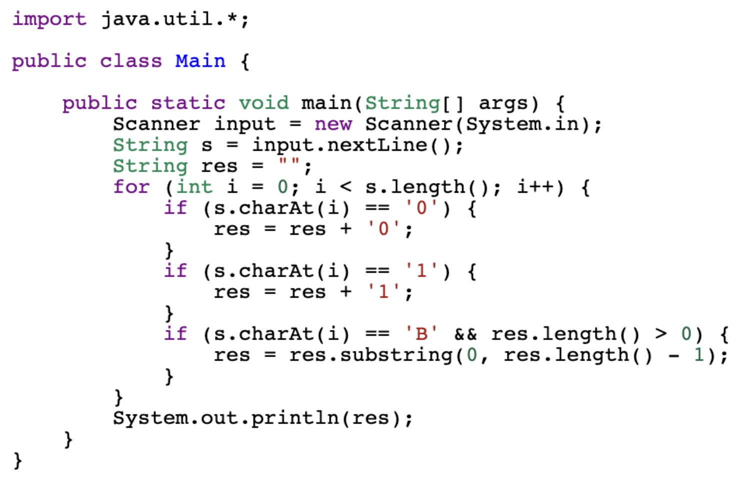
**Observe this** - while all three files have a nested loop and keep popping letters based on a clause, the first two use a break to while the third doesn’t, which could be made tighter using a better while loop. But more importantly, the third has three loops - one within the other, while the first two only have a single nested loop. This is ot at all necessary for the problem asked. I would classify this as a **false** hit.

Now we go to another cluster and we see files of this type -



```
import java.util.*;
class Main{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
          String S = sc.next();
          String spt[] =S.split("");
          String ans="";
          for(int i=0;i<spt.length;i++){
              if(spt[i].equals("0")){
                  ans+="0";
              }else if(spt[i].equals("1")){
                  ans+="1";
              }else if(spt[i].equals("B")){
                  if(!(ans.isEmpty())){
                      ans =ans.substring(0, ans.length()-1);
                  }

              }
          }

      System.out.println(ans);
    }
}
```

And then there's this file in the same cluster -



```
import java.util.*;

public class Main {

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        String s = input.nextLine();
        String res = "";
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '0') {
                res = res + '0';
            }
            if (s.charAt(i) == '1') {
                res = res + '1';
            }
            if (s.charAt(i) == 'B' && res.length() > 0) {
                res = res.substring(0, res.length() - 1);
            }
        }
        System.out.println(res);
    }
}
```

**Observe this** - The structure and logic between both files is similar (thanks to out feature engineering) - but there is something we missed in the features - and that is the *if - else if* differentiation. An *else if* clause is only reached if the preceding *if/else if* wasn't satisfied - this is a significant marker of code quality.

**These are definitive ways of finding false hits.** Now that we have understood the general idea of how these clusters are evaluated, look at the **accuracy of each of these clusters** in terms of how many files did belong there out of total files in the cluster -

| **C1** | **C2** | **C3** | **C4** | **C5** | **C6** | **C7** | **C8** | **C9** | **C10** | **C11** | **C12** | **C13** | **C14** | **C15** | **C16** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31/35 | 29/30 | 18/23 | 18/22 | 18/18 | 6/7 | 5/6 | 6/6 | 2/4 | 1/3 | 1/3 | 3/3 | 3/3 | 2/3 | 2/2 | 1/2 |

**Total Accuracy on Training Set: 146/160 = 91.25%**

**Updated Modeling Methodology -**

We understand the pitfalls well enough to improve our model accuracy for the future without seeing the future sets. And in this project, we have improved in two ways -

1) Looping structures are NOT normalized along with the rest of the features but are done in a rule-driven rigid manner to classify files first. Then the rest of the features are normalized to divide these rigid clusters further into sub-clusters.
2) The query for if statements now is broken into two queries of if statements alone as if/else if combined statements.

**With this we are able to reach a total accuracy of 158/160 which is 98.75%.**

It is in fact possible to bring it to a full 100% but we have put a limit on how much querying time and human-in-the-loop is permitted (no more than 5 to 10 minutes of feature selection within a maximum of 20 queries). Hence, we'll accept this accuracy.

**Validation Set Evaluation -**

For the validation set, these are the results after being placed in their nearest clusters -

| C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 | C11 | C12 | C13 | C14 | C15 | C16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8/8 | 5/7 | 7/7 | 5/5 | 3/4 | Nul | Nul | Nul | 3/3 | Nul | 2/2 | Nul | Nul | Nul | 2/2 | 2/2 |

**Total Accuracy on Validation Set: 37/40 = 92.5%**

This is considerably fair since we purposely picked a highly variant set of very few files (40 files) to match these clusters.

## Final Results

**Modeling Methodology (with algorithm) -**

KMeans clustering with some custom changes was used to create original code quality groups. The number of clusters was decided using a cross between present combinations of features on sets as well as the WCSS score. The idea of these clusters is that each would receive the same general feedback from a code reviewer. The way this feedback is generated is covered under *Interpretability as Future Work*.

**Performance of Model -**

1) **The training set first performed with an accuracy of 91.25% for the first round of engineered features and modeling methodology.**

2) **The training set performed with an accuracy of 98.75% for the second round after diving deeper into the features and coding pragmatics and after changing the modeling methodology slightly.**
3) **The training set was not pushed to a 100% accuracy due to limits of fitting and human effort placed on this endeavor.**
4) **The validation set assessed 92.5% of the incoming and highly variant files correctly.**

**Based on these results - we can use this kind of model for simple problems' code review to learners and developers in their prime of learning to code.**

Giving wrong feedback is highly unfair and damaging, hence *Detecting Unknown Falses as Further Work* covers some of the ways to catch false hits automatically.

**Please note - we did the false hits evaluation manually on this project, because this was a proof of concept to show that the described methodology was capable of being used and interpreted (as in, backtracking in to find the cause of false hits) on a real-world codebase.**

**This has been successfully demonstrated.**

## Interpretability as Further Work -

Using the SHAP library and a switchboard of phrases and clauses of explanation along with feature names, it is possible to automatically explain each cluster created in terms of the corresponding centroid coordinate of each feature.

## Detecting Unknown Falses as Further Work -

As we mentioned in the problem statement, data tracing and efficiency evaluations are other methods used to define code. A good way to detect false hits is to triangulate results between multiple aspects.

In other words, if I have a cluster in future (consisting of incoming data) that claims to represent files of the same type of coding, I can run the data trace and efficiency check as a secondary test -

1) If there are outliers in the results of these tests on these clusters, then these outliers might be probable false hits.
2) If there is a split in the secondary test results of a cluster in a significant manner (of the order 20-80, 30-70, and so on), then this is an indication that the cluster might be still split further into smaller better-defined clusters.

With this we come to the end of the report.

**Hope you enjoyed it!**

## Credits

1. A lot of my experience in this field has come from my contributing role at the Israeli company Sense Education, where I collaborate as an AI Solutions Engineer on the R&D and product team on endeavors of the same class.

2. I learnt a lot during the semester at Georgia Tech (ISYE 6740) from the TAs, assignments, notes, and Discussions and used some part of that in the methodology.

3. Project CodeNet by IBM - https://github.com/IBM/Project_CodeNet#directory-structure-and-naming-convention - is where I extracted my codebase from for this preliminary PoC.