A Project Report
On

CODE DETECTION USING MACHINE LEARNING

Submitted in partial fulfillment of the requirements for the award of the Degree of

BACHELOR OF COMPUTER APPLICATION
IN
INFORMATION TECHNOLOGY
By

VASIREDDY . LIKHITHA (191FJ01036)
BORRA . SRI HARSHA (191FJ01057)

Under the Esteemed Guidance of
Mrs. P. R. S. M. Lakshmi, M. Tech
Assistant Professor

VIGNAN'S
Foundation for Science, Technology & Research
UNIVERSITY
(Estd u/s 3 of UGC Act of 1956)

DEPARTMENT OF INFORMATION TECHNOLOGY (BCA)
VFSTR, VADLAMUDI
GUNTUR-522213, ANDHRA PRADESH, INDIA
JUNE, 2022.

VIGNAN'S
Foundation for Science, Technology & Research
(Deemed to be University)
-Estd. u/s 3 of UGC Act 1956

CERTIFICATE

This is to certify that this project report entitled "**CODE DETECTION USING MACHINE LEARNING**" is a bonafide record of work carried out by V. Likhitha (191FJ01036), B. Sri Harsha(191FJ01057) under the guidance and supervision of Mrs. P. R. S. M. Lakshmi in partial fulfillment of the requirements for the award of the degree of Bachelor of technology in Information Technology of VIGNAN'S FOUNDATION FOR TECHNOLOGY & SCIENCE FOR during the year 2021-2022.

Project Guide	**Head of the Department**
Mrs. P. R. S. M. Lakshmi	**Dr. K. V. Krishna kishore**
Asst. Professor	

ACKNOWLEDGEMENT

After completion of this project, it gives us great pleasure to express our gratitude and heart full thanks to all those people whose help went long way in completing this project. There are several people who deserve more than a written acknowledgement for practical help.

We would like to place on record the help received from all our respected lecturers from school of IT under the inspiring guidance of **MRS.P.R.S.M.LAKSHMI** Assistant professor which helped during the course of completing this project.

We would also like to place on record the help received from **Dr. K. V. Krishna Kishore** Head of Department for their valuable support and resources guided us towards the completion of course and project and we would also like to thank all our class mates for their valuable suggestions and support in completion of this work.

PROJECT ASSOCIATES

VASIREDDY.LIKHITHA(191FJ01036)

BORRA SRI HARSHA(191FJ01057)

DECLARATION

I hereby declare that the project entitled **“ Code detection using Machine Learning ”** submitted to the **DEPARTMENT OF INFORMATION TECHNOLOGY** . This dissertation is our original work and the project has not formed the basis for the award of any degree, associate-ship and fellowship or any other similar titles and no part of it has been published or sent for publication at the time of Submission

By

DATE :

Vasireddy Likhitha(191FJ01036)
Borra sai sri Harsha(191FJ01057)

ABSTRACT

Code smells indicate poor implementation choices that may hinder the system maintenance. Their detection is important for the software quality improvement, but studies suggest that it should be tailored to the perception of each developer. Therefore, detection techniques must adapt their strategies to the developer’s perception. Machine Learning (ML) algorithms is a promising way to customize the smell detection, but there is a lack of studies on their accuracy in detecting smells for different developers. This paper evaluates the use of ML algorithms on detecting code smells for different developers, considering their individual perception about code smells. The results show that ML-algorithms achieved low accuracies for the developers that participated of our study, showing that are very sensitive to the smell type and the developer. These algorithms are not able to learn with limited training set, an important limitation when dealing with diverse perceptions about code smells

LIST OF TABLES

CODE DETECTION USING MACHINE LEARNING

CHAPTER - 1

INTRODUCTION

INTRODUCTION

Nowadays, the complexity of software systems is growing fast and software companies are required to continuously update their source code. Those continuous changes frequently occur under time pressure and lead developers to set aside good programming practices and principles in order to deliver the most appropriate but still immature product in the shortest time possible. This process can often result in the introduction of so-called technical deb design problems likely to have negative consequences during the system maintenance and evolution.

To overcome these limitations, machine-learning (ML) techniques are being adopted to detect code smells. Usually a supervised method is exploited, i.e., a set of independent variables (a.k.a. predictors) are used to determine the value of a dependent variable (i.e., presence of a smell or degree of the smelliness of a code element) using a machine-learning classifier (e.g., Logistic Regression).

In order to empirically assess the actual capabilities of ML techniques for code smell detection, Arcelli Fontana et al. conducted a large-scale study where 32 different ML algorithms were applied to detect four code smell types, i.e., Data Class, Large Class, Feature Envy and Long Method. The authors reported that most of the classifiers exceeded 95% both in terms of accuracy and of F-Measure, with J48 and RANDOM FOREST obtaining the best performance. The authors see in these results an indication that "using machine learning algorithms for code smell detection is an appropriate approach" and that "performances are already so good that we think it does not really matter in practice what machine learning algorithm one chooses for code smell detection".

In our research, we have observed important limitations of the work by Arcelli Fontana et al. that might affect the generalizability of their findings. Specifically, the high performance reported might be due to the way the dataset was constructed: for each type of code smell analyzed, the dataset contains only instances affected by this type of smell or non-smelly instances, with a non-realistic balance of smelly and non-smelly instances and a strongly different distribution of the metrics between the two groups of instances, which is far from reality. During software maintenance and evolution, software systems need to be continuously changed by developers in order to (i) implement new requirements, (ii) enhance existing features, or (iii) fix important bugs. Due to time pressure or community-related factors, developers do not always have the time or the willingness to keep the complexity of the system under control and find good design solutions before applying their modifications. As a consequence, the development activities are often performed in an undisciplined manner, and have the effect to erode the original design of the system by introducing the so-called technical debt. Code smells, i.e., symptoms of the presence of poor design or implementation choices in the source code, represent one of the most serious forms of technical debt. Indeed, previous research found that not only they strongly reduce the ability of developers to comprehend the source 2code, but also make the affected classes more change- and fault-prone. Thus, they represent an important threat for maintainability effort and costs.

In past and recent years, the research community was highly active on the topic. On the one hand, many empirical studies have been conducted with the aim of understanding (i) when and why code smells are introduced, (ii) what is their evolution and longevity in software projects, and (iii) to what extent they are relevant for developers. On the other hand, several code smell detectors have been proposed as well. Most of them can be considered as heuristics-based: they apply a two-step process where a set of metrics are firstly computed, and then some thresholds are applied upon such metrics to discriminate between smelly and non-smelly classes.

They differ from each other for (i) the specific algorithms used to identify code smells (e.g., a combination of metrics or through the use of more advanced methodologies like Relational Topic Modeling) and (ii) the metrics exploited (e.g., based on code metrics or historical data). Although it has been showed that such detectors have reasonable performance in terms of accuracy of the recommendations, previous works highlighted a number of important limitations that might preclude the use of such detectors in practice.

In particular, code smells identified by existing detectors can be subjectively interpreted by developers. At the same time, the agreement between them is low. More importantly, most of them require the specification of thresholds to distinguish smelly code components from non-smelly one: naturally, the selection of thresholds strongly influence their accuracy. For all these reasons, a recent trend is the adoption of Machine Learning (ML) techniques for approaching the problem.

In this scenario, a supervised method is exploited: a set of independent variables (a.k.a., predictors) are used to predict the value of a dependent variable (i.e., the smelliness of a class) using a machine learning classifier (e.g., Logistic Regression). The model can be trained using a sufficiently large amount of data available from the project under analysis, i.e., within-project strategy, or using data coming from other (similar) software projects, i.e., cross-project strategy. These approaches clearly differ from the heuristics-based ones, as they rely on classifiers to discriminate the smelliness of classes rather than on predefined thresholds upon computed metrics

CHAPTER 2
LITERATURE SURVEY

2.1. Detecting Code Smells using Machine Learning Techniques

THE REFERENCE WORK

In the authors analyze three main aspects related to the use of machine-learning algorithms for code smell detection: (i) performance of a set of classifiers over a sample of the total instances contained in the dataset, (ii) analysis of the minimum training set size needed to accurately detect code smells, and (iii) analysis of the number of code smells detected by different classifiers over the entire dataset.

In this paper, we focus on the first research question of the reference work. In the following subsections we detail the methodological process adopted in

A. Context Selection

The context of the study was composed of software systems and code smells. The authors have analyzed systems from the Qualitas Corpus, release 20120401r, one of the largest curated benchmark datasets to date, specially designed for empirical software engineering research. Among 111 Java systems of the corpus, 37 were discarded because they could not be compiled and therefore code smell detection could not be applied. Hence, the authors focused on the remaining 74 systems. For each system 61 source code metrics were computed at class level and 82—at method level. The former were used as independent variables for predicting class-level smells Data Class and God Class, the latter for predicting method-level smells Feature Envy and Long Method:

God Class.

It arises when a source code class implements more than one responsibility; it is usually characterized by a large number of attributes and methods, and has several dependencies with other classes of the system.

Data Class.

This smell refers to classes that store data without providing complex functionality.

Feature Envy.

This is a method-level code smell that appears when a method uses much more data than another class with respect to the one it is actually in Long Method. It represents a large method that implements more than one function. The choice of these smells is due to the fact that they capture different design issues, e.g., large classes or misplaced methods.

B. Machine Learning Techniques Experimented

In six basic ML techniques have been evaluated: J48, JRIP, RANDOM FOREST, NAIVE BAYES, SMO, and LIBSVM. As for J48, the three types of pruning techniques available in WEKA were used, SMO was based on two kernels (e.g., POLYNOMIAL and RBF), while for LIBSVM eight different configurations, using C-SVC and V-SVC, were used. Thus, in total 16 different ML techniques have been evaluated. Moreover, the ML techniques were also combined with the DABOOSTM1 boosting technique, i.e., a method that iteratively uses a set of models built in previous iterations to manipulate the training set and make it more suitable for the classification problem, leading to 32 different variants. An important step for an effective construction of machinelearning models consists of the identification of the best configuration of parameters: the authors applied to each classifier the Grid-search algorithm, capable of exploring the parameter space to find an optimal configuration.

C. Dataset Building

To establish the dependent variable for code smell prediction models, the authors applied for each code smell the set of automatic detectors . However, code smell detectors cannot usually achieve 100% recall, meaning that an automatic detection process might not identify actual code smell instances (i.e., false negatives) even in the case that multiple detectors are combined. To cope with false positives and to increase their confidence in validity of the dependent variable, the authors applied a stratified random sampling of the classes/methods of the considered systems: this sampling produced 1,986 instances (826 smelly elements and 1,160 non smelly ones), which were manually validated by the authors in order to verify the results of the detectors. As a final step, the sampled dataset was normalized for size: the authors randomly removed smelly and non-smelly elements building four disjoint datasets, i.e., one for each code smell type, composed of 140 smelly instances and 280 non smelly ones (for a total of 420 elements). These four datasets represented the training set for the ML techniques above.

D. Validation Methodology

To test the performance of the different code smell prediction models built, the authors applied 10-fold cross validation: each of the four datasets was randomly partitioned in ten folds of equal size, such that each fold has the same proportion of smelly elements. A single fold was retained as test set, while the remaining ones were used to train the ML models. The process was then repeated ten times, using each time a different fold as the test set. Finally, the performance of the models was assessed using mean accuracy, F-Measure, and AUC-ROC over the ten runs.

E. Limitations and Replication Problem Statement

The results achieved in reported that most of the classifiers have accuracy and F-Measure higher than 95%, with J48 and RANDOM FOREST being the most powerful ML techniques. These results seem to suggest that the problem of code smell detection can be solved almost perfectly through ML approaches, while other unsupervised techniques (e.g., the ones based on detection rules) only provide suboptimal recommendations. However, we identified possible reasons for these good results:

Selection of the instances in the dataset.

The first factor possibly affecting the results might be represented by the characteristics of smelly and non-smelly instances present in the four datasets exploited (one for each smell type): in particular, if the metric distribution of smelly elements is strongly different than the metric distribution of non smelly instances, then any ML technique might easily distinguish the two classes. Clearly, this does not properly represent a real-case scenario, where the boundary between the structural characteristics of smelly and non-smelly code components is not always clear. In addition, the authors built four datasets, one for each smell. Each dataset contained code components affected by that type of smell or non-smelly components. This also makes the datasets unrealistic (a software system usually contains different types of smells) and might have made easier for the classifiers to discriminate smelly components.

Balanced dataset.

In the reference work, one third of the instances in the dataset was composed of smelly elements. According to recent findings on the diffuseness of code smells, a small fraction of a software system is usually affected by code smells: e.g., Palomba et al. found that God Classes represent less than 1% of the total classes in a software system. Because of the points above, we argue that the capabilities of ML techniques for code smell detection should be reevaluated. In this paper, we replicate the reference study with different datasets containing instances of more than one type of smells and with a different balance between smelly and non smelly instances.

CONCLUSION AND FUTURE DIRECTIONS

In this work, we presented a replicated study of the work in. The main difference between our study and the reference study is in the construction of the datasets used to train and validate the code smell classifiers. While in the reference study each dataset contained instances of one type of smells (besides non smelly components), in our study we built datasets containing instances of more than one type of smell. Another difference is in the distribution of smelly and non smelly instances in the datasets, which in our case are less balanced and contain more non smelly instances. As a result, we believe that the problem of detecting code smells using machine learning techniques is extremely relevant and with several facets remaining open. For example, more research is needed towards structuring datasets appropriately and jointly with the predictors to be used. As future work, we firstly plan to assess the impact of (i) dataset size, (ii) feature selection, and (iii) validation methodology on the results of our study. At the same, we aim at addressing these issues, thus defining new prediction models for code smell detection.

CHAPTER 3

METHODOLOGY

3.1 EXISTING SYSTEM

The goal of the empirical study reported in this paper was to analyze the sensitivity of the results achieved by our reference work with respect to the metric distribution of smelly and non-smelly instances, with the purpose of understanding the real capabilities of existing prediction models in the detection of code smells.

The perspective is of both researchers and practitioners: the former are interested in understanding possible limitations of current approaches in order to devise better ones; the latter are interested in evaluating the actual applicability of code smell prediction in practice.

DISADVANTAGE:

- Subjective of developers which respect to code smell detection by such tools
- Scarce agreement between different detectors

Difficulties in finding good thresholds to be used for detection

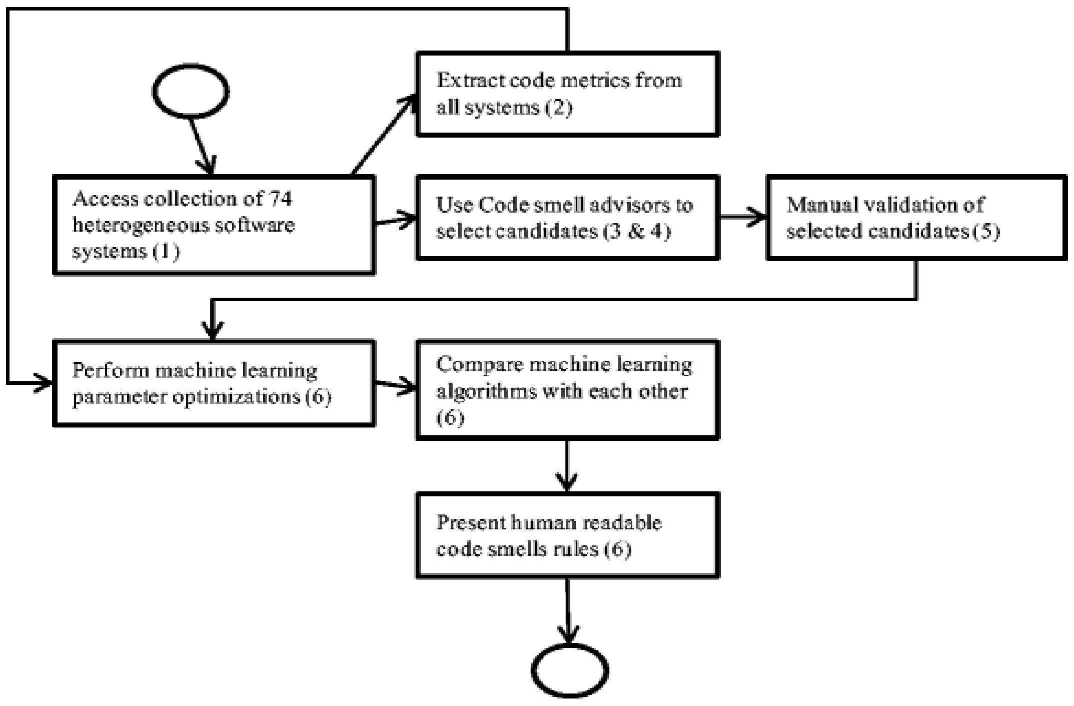
3.2 PROPOSED MODEL

- Stage 1: Code smell classification
- Stage-2: Evaluation of machine learning model
- Navie bayes
- Random forest
- Decision tree classifier
- Stage-3: Model variance test

ADVANTAGE:

- performance of a set of classifiers over a sample of the total instances contained in the dataset,
- analysis of the minimum training set size needed to accurately detect code smells, and
- analysis of the number of code smells detected by different classifiers over the entire dataset.

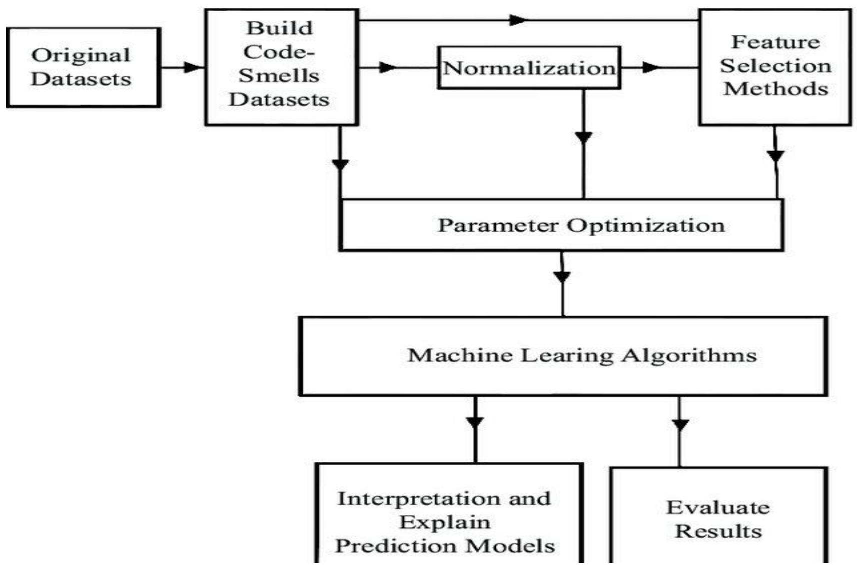
3.3 SYSTEM ARCHITECTURE



Extract code metrics from all systems (2)
Access collection of 74 heterogeneous software systems (1)
Use Code smell advisors to select candidates (3 & 4)
Manual validation of selected candidates (5)
Perform machine learning parameter optimizations (6)
Compare machine learning algorithms with each other (6)
Present human readable code smells rules (6)

Fig 3.1 System Architecture

FLOW CHART



Original Datasets
Build Code-Smells Datasets
Normalization
Feature Selection Methods
Parameter Optimization
Machine Learing Algorithms
Interpretation and Explain Prediction Models
Evaluate Results

Working of Random Forest Algorithm :

Before understanding the working of the random forest we must look into the ensemble technique. Ensemble simply means combining multiple models. Thus a collection of models is used to make predictions rather than an individual model.
Ensemble uses two types of methods:
Bagging– It creates a different training subset from sample training data with replacement & the final output is based on majority voting. For example, Random Forest.
Boosting– It combines weak learners into strong learners by creating sequential models such that the final model has the highest accuracy. For example, ADA BOOST, XG BOOST
As mentioned earlier, Random forest works on the Bagging principle. Now let's dive in and understand bagging in detail.

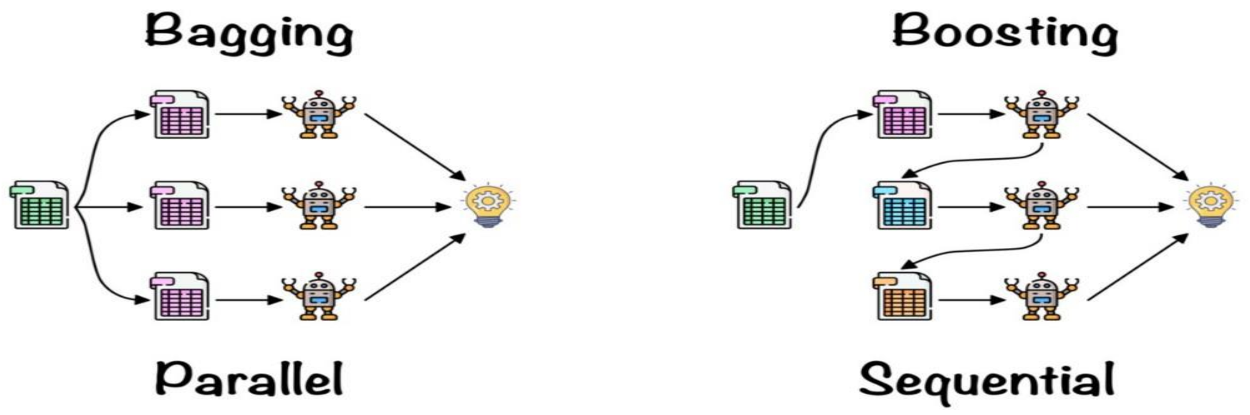


Fig 3. 2 Bagging Parallel & Boosting Sequential

Bagging

Bagging, also known as Bootstrap Aggregation is the ensemble technique used by random forest. Bagging chooses a random sample from the data set. Hence each model is generated from the samples (Bootstrap Samples) provided by the Original Data with replacement known as row sampling. This step of row
sampling with replacement is called bootstrap. Now each model is trained independently which generates results. The final output is based on majority voting after combining the results of all

models. This step which involves combining all the results and generating output based on majority voting is known as aggregation.

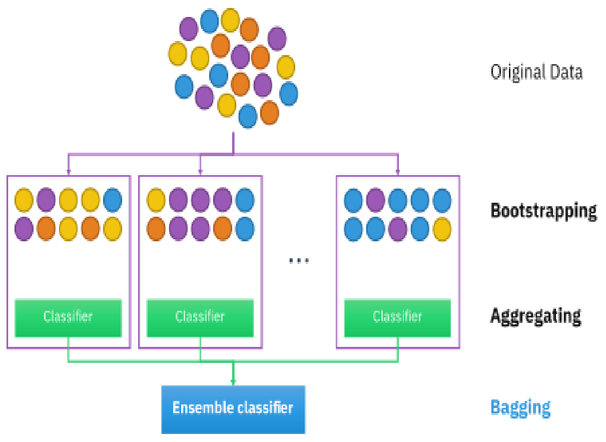


Fig 3.3 Bagging

Now let’s look at an example by breaking it down with the help of the following figure. Here the bootstrap sample is taken from actual data (Bootstrap sample 01, Bootstrap sample 02, and Bootstrap sample 03) with a replacement which means there is a high possibility that each sample won’t contain unique data.

Now the model (Model 01, Model 02, and Model 03) obtained from this bootstrap sample is trained independently. Each model generates results as shown. Now Happy emoji is having a majority when compared to sad emoji. Thus based on majority voting final output is obtained as Happy emoji.

Steps involved in random forest algorithm:

Step 1: In Random forest n number of random records are taken from the data set having k number of records.

Step 2: Individual decision trees are constructed for each sample.

Step 3: Each decision tree will generate an output.

Step 4: Final output is considered based on Majority Voting or Averaging for Classification and regression respectively.

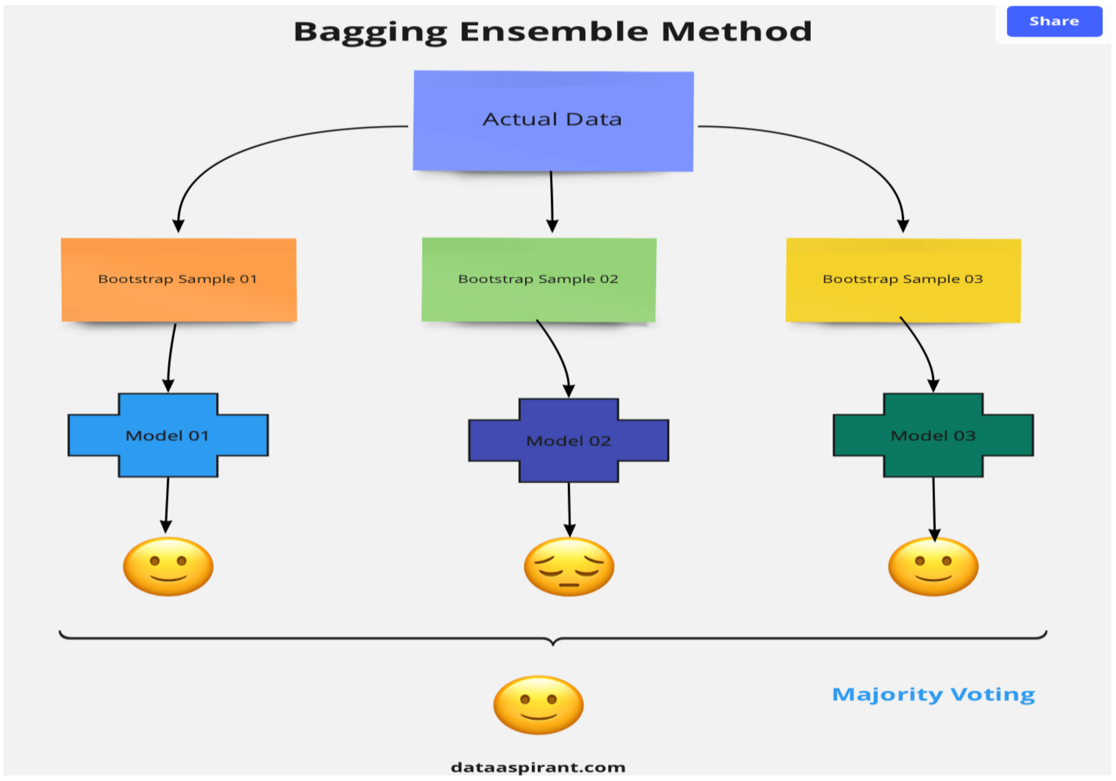


Fig 3. 4 Bagging Ensemble Method

For example: consider the fruit basket as the data as shown in the figure below. Now n number of samples are taken from the fruit basket and an individual decision tree is constructed for each sample. Each decision tree will generate an output as shown in the figure. The final output is considered based on majority voting. In the below figure you can see that the majority decision tree gives output as an apple when compared to a banana, so the final output is taken as an apple.

Important Features of Random Forest

Diversity- Not all attributes/variables/features are considered while making an individual tree, each tree is different.

Immune to the curse of dimensionality- Since each tree does not consider all the features, the feature space is reduced.

Parallelization-Each tree is created independently out of different data and attributes. This means that we can make full use of the CPU to build random forests.

Train-Test split- In a random forest we don’t have to segregate the data for train and test as there will always be 30% of the data which is not seen by the decision tree.

Stability- Stability arises because the result is based on majority voting/ averaging.

Difference Between Decision Tree & Random Forest

Random forest is a collection of decision trees; still, there are a lot of differences in their behavior.

Decision trees	Random Forest
1. Decision trees normally suffer from the problem of overfitting if it's allowed to grow without any control.	1. Random forests are created from subsets of data and the final output is based on average or majority ranking and hence the problem of overfitting is taken care of.
2. A single decision tree is faster in computation.	2. It is comparatively slower.
3. When a data set with features is taken as input by a decision tree it will formulate some set of rules to do prediction.	3. Random forest randomly selects observations, builds a decision tree and the average result is taken. It doesn't use any set of formulas.

Table 3.1 Decision trees and Random forest

Thus random forests are much more successful than decision trees only if the trees are diverse and acceptable.

Important Hyperparameters

Hyperparameters are used in random forests to either enhance the performance and predictive power of models or to make the model faster.

Following hyperparameters increases the predictive power:

n_estimators– number of trees the algorithm builds before averaging the predictions.

max_features– maximum number of features random forest considers splitting a node.

mini_sample_leaf– determines the minimum number of leaves required to split an internal node.

Following hyperparameters increases the speed:

n_jobs– it tells the engine how many processors it is allowed to use. If the value is 1, it can use only one processor but if the value is -1 there is no limit.

random_state– controls randomness of the sample. The model will always produce the same results if it has a definite value of random state and if it has been given the same hyperparameters and the same training data.

oob_score – OOB means out of the bag. It is a random forest cross-validation method. In this one-third of the sample is not used to train the data instead used to evaluate its performance. These samples are called out of bag samples.

Let's import the libraries.

# Importing the required libraries import pandas as pd, numpy as np import matplotlib. pyplot as plt, seaborn as sns %matplotlib inline

import the dataset.

# Reading the csv file and putting it into 'df' object df = pd.read_csv('heart_v2.csv') df.head()

Putting Feature Variable to X and Target variable to y.

# Putting feature variable to X = df.drop('heart disease',axis=1) # Putting response variable to y = df['heart disease']

Train-Test-Split is performed

# now lets split the data into train and test from sklearn.model_selection import train_test_split

# Splitting the data into train and test X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, random_state=42) X_train.shape, X_test.shape

Now let's visualize

from sklearn. tree import plot_tree plt.figure(figsize=(80,40)) plot_tree(rf_best.estimators_[5], feature_names = X.columns,class_names=['Disease', "No Disease"],filled=True);from sklearn.tree import plot_treeplt.figure(figsize=(80,40))plot_tree(rf_best.estimators_[7], feature_names = .columns,class_names=['Disease', "No Disease"],filled=True);

The trees created by estimators_[5] and estimators_[7] are different. Thus we can say that each tree is independent of the other.

Now let's sort the data with the help of feature importance rf_best.feature_importances_

imp_df = pd.DataFrame({ "Varname": X_train.columns, "Imp": rf_best.feature_importances_ })

imp_df.sort_values(by="Imp", ascending=False)

Use Cases

This algorithm is widely used in E-commerce, banking, medicine, the stock market, etc.

For example: In the Banking industry it can be used to find which customer will default on the loan.

Advantages and Disadvantages of Random Forest Algorithm

Advantages:

- It can be used in classification and regression problems.

- It solves the problem of overfitting as output is based on majority voting or averaging.
- It performs well even if the data contains null/missing values.
- Each decision tree created is independent of the other thus it shows the property of parallelization.
- It is highly stable as the average answers given by a large number of trees are taken.
- It maintains diversity as all the attributes are not considered while making each decision tree though it is not true in all cases.
- It is immune to the curse of dimensionality. Since each tree does not consider all the attributes, feature space is reduced.
- We don’t have to segregate data into train and test as there will always be 30% of the data which is not seen by the decision tree made out of bootstrap.
- Disadvantages:
- Random forest is highly complex when compared to decision trees where decisions can be made by following the path of the tree.
- Training time is more compared to other models due to its complexity. Whenever it has to make a prediction each decision tree has to generate output for the given input data.

Summary

Now, we can conclude that Random Forest is one of the best techniques with high performance which is widely used in various industries for its efficiency. It can handle binary, continuous, and categorical data. Random forest is a great choice if anyone wants to build the model fast and efficiently as one of the best things about the random forest is it can handle missing values.Overall, random forest is a fast, simple, flexible, and robust model with some limitations.

Decision Tree

Introduction

Till now we have learned about linear regression, logistic regression, and they were pretty hard to understand. Let’s now start with Decision tree’s and I assure you this is probably the easiest algorithm in Machine Learning. There’s not much mathematics involved here. Since it is very easy to use and interpret it is one of the most widely used and practical methods used in Machine Learning.

Contents

What is a Decision Tree?

Example of a Decision Tree

Entropy

Information Gain

When to stop Splitting?

How to stop overfitting?

- max_depth
- min_samples_splitmin_samples_leafmax_features
- Pruning
- Post-pruning Pre-pruning
- Endnotes

What is a Decision Tree?

It is a tool that has applications spanning several different areas. Decision trees can be used forclassification as well as regression problems. The name itself suggests that it uses a flowchart like a tree structure to show the predictions that result from a series of feature-based splits. It starts with a root node and ends with a decision made by leaves.

Before learning more about decision trees let's get familiar with some of the terminologies.

Root Nodes – It is the node present at the beginning of a decision tree from this node the population starts dividing according to various features.

Decision Nodes – the nodes we get after splitting the root nodes are called Decision Node

Leaf Nodes – the nodes where further splitting is not possible are called leaf nodes or terminal nodes

Sub-tree – just like a small portion of a graph is called sub-graph similarly a sub-section of this decision tree is called sub-tree.

Pruning – is nothing but cutting down some nodes to stop overfitting.

Example of a decision tree.

Let's understand decision trees with the help of an example.

Decision trees are upside down which means the root is at the top and then this root is split into various several nodes. Decision trees are nothing but a bunch of if-else statements in layman terms. It checks if the condition is true and if it is then it goes to the next node attached to that decision.

Did you notice anything in the above flowchart? We see that if the weather is cloudy then we must go to play. Why didn't it split more? Why did it stop there?

To answer this question, we need to know about few more concepts like entropy, information gain, and Gini index. But in simple terms, I can say here that the output for the training dataset is always yes for cloudy weather, since there is no disorderliness here we don't need to split the node further. The goal of machine learning is to decrease uncertainty or disorders from the dataset and for this, we use decision trees.

Now you must be thinking how do I know what should be the root node? what should be the decision node? when should I stop splitting? To decide this, there is a metric called "Entropy" which is the amount of uncertainty in the dataset.

Entropy

Entropy is nothing but the uncertainty in our dataset or measure of disorder. Let me try to explain this with the help of an example.

Suppose you have a group of friends who decides which movie they can watch together on Sunday. There are 2 choices for movies, one is "Lucy" and the second is "Titanic" and now everyone has to tell their choice. After everyone gives their answer we see that "Lucy" gets 4 votes and "Titanic" gets 5 votes. Which movie do we watch now? Isn't it hard to choose 1 movie now because the votes for both the movies are somewhat equal.

This is exactly what we call disorderness, there is an equal number of votes for both the movies, and we can't really decide which movie we should watch. It would have been much easier if the votes for "Lucy" were 8 and for "Titanic" it was 2. Here we could easily say that the majority of votes are for "Lucy" hence everyone will be watching this movie.

How do Decision Trees use Entropy?

Now we know what entropy is and what is its formula, Next, we need to know that how exactly does it work in this algorithm.

Entropy basically measures the impurity of a node. Impurity is the degree of randomness; it tells how random our data is. A pure sub-split means that either you should be getting "yes", or you should be getting "no".

Suppose feature 1 had 8 yes and 4 no, after the split feature 2 get 5 yes and 2 no whereas feature 3 gets 3 yes and 2 no.

We see here the split is not pure, why? Because we can still see some negative classes in both the feature. In order to make a decision tree, we need to calculate the impurity of each split, and when the purity is 100% we make it as a leaf node.

To check the impurity of feature 2 and feature 3 we will take the help for Entropy formula. We can clearly see from the tree itself that feature 2 has low entropy or more purity than feature 3 since feature 2 has more "yes" and it is easy to make a decision here.
Always remember that the higher the Entropy, the lower will be the purity and the higher will be the impurity.
As mentioned earlier the goal of machine learning is to decrease the uncertainty or impurity in the dataset, here by using the entropy we are getting the impurity of a feature or a particular node, we don't know if the parent entropy or the entropy of a particular node has decreased or not.
For this, we bring a new metric called "Information gain" which tells us how much the parent entropy has decreased after splitting it with some feature.
Information Gain
Information gain measures the reduction of uncertainty given some feature and it is also a deciding factor for which attribute should be selected as a decision node or root node.
It is just entropy of the full dataset – entropy of the dataset given some feature. To understand this better let's consider an example:
Suppose our entire population has a total of 30 instances. The dataset is to predict whether the person will go to the gym or not. Let's say 16 people go to the gym and 14 people don't
Now we have two features to predict whether he/she will go to the gym or not. Feature 1 is "Energy" which takes two values "high" and "low"
Feature 2 is "Motivation" which takes 3 values "No motivation", "Neutral" and "Highly motivated".
Let's see how our decision tree will be made using these 2 features. We'll use information gain to decide which feature should be the root node and which feature should be placed after the split.
Let's calculate the entropy:
To see the weighted average of entropy of each node we will do as follows:
Now we have the value of E(Parent) and E(Parent|Energy), information gain will be:
Our parent entropy was near 0.99 and after looking at this value of information gain, we can say that the entropy of the dataset will decrease by 0.37 if we make "Energy" as our root node.Similarly, we will do this with the other feature "Motivation" and calculate its information gain.
Let's calculate the entropy here:
To see the weighted average of entropy of each node we will do as follows:
Now we have the value of E(Parent) and E(Parent|Motivation), information gain will be:

We now see that the “Energy” feature gives more reduction which is 0.37 than the “Motivation” feature. Hence we will select the feature which has the highest information gain and then split the node based on that feature.

In this example “Energy” will be our root node and we’ll do the same for sub-nodes. Here we can see that when the energy is “high” the entropy is low and hence we can say a person will definitely go to the gym if he has high energy, but what if the energy is low? We will again split the node based on the new feature which is “Motivation”.

When to stop splitting?

You must be asking this question to yourself that when do we stop growing our tree? Usually, real-world datasets have a large number of features, which will result in a large number of splits, which in turn gives a huge tree. Such trees take time to build and can lead to overfitting. That means the tree will give very good accuracy on the training dataset but will give bad accuracy in test data.

There are many ways to tackle this problem through hyperparameter tuning. We can set the maximum depth of our decision tree using themax_depth parameter. The more the value of max_depth, the more complex your tree will be. The training error will off-course decrease if we increase the max_depth value but when our test data comes into the picture, we will get a very bad accuracy. Hence you need a value that will not overfit as well as underfit our data and for this, you can use GridSearchCV.

Another way is to set the minimum number of samples for each spilt. It is denoted by min_samples_split. Here we specify the minimum number of samples required to do a spilt. For example, we can use a minimum of 10 samples to reach a decision. That means if a node has less than 10 samples then using this parameter, we can stop the further splitting of this node and make it a leaf node.

There are more hyperparameters such as :

min_samples_leaf – represents the minimum number of samples required to be in the leaf node. The more you increase the number, the more is the possibility of overfitting.

max_features – it helps us decide what number of features to consider when looking for the best split. To read more about these hyperparameters you can read it here.

Pruning

It is another method that can help us avoid overfitting. It helps in improving the performance of the tree by cutting the nodes or sub-nodes which are not significant. It removes the branches which have very low importance.

There are mainly 2 ways for pruning:

Pre-pruning – we can stop growing the tree earlier, which means we can prune/remove/cut a node if it has low importance while growing the tree.

Post-pruning – once our tree is built to its depth, we can start pruning the nodes based on their significance.

Endnotes

To summarize, in this article we learned about decision trees. On what basis the tree splits the nodes and how to can stop overfitting. why linear regression doesn't work in the case of classification problems.

In the next article, I will explain Random forests, which is again a new technique to avoid overfitting. To check out the full implementation of decision trees please refer to my Githubrepository.

What is Naive Bayes algorithm?

It is a classification technique based on Bayes' Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature.

For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as 'Naive'.

Naive Bayes model is easy to build and particularly useful for very large data sets. Along with simplicity, Naive Bayes is known to outperform even highly sophisticated classification methods.

How Naive Bayes algorithm works?

Let's understand it using an example. Below I have a training data set of weather and corresponding target variable 'Play' (suggesting possibilities of playing). Now, we need to classify whether players will play or not based on weather condition. Let's follow the below steps to perform it.

- Step 1: Convert the data set into a frequency table
- Step 2: Create Likelihood table by finding the probabilities like Overcast probability = 0.29 and probability of playing is 0.64.
- Step 3: Now, use Naive Bayesian equation to calculate the posterior probability for each class. The class with the highest posterior probability is the outcome of prediction.

- Naive Bayes uses a similar method to predict the probability of different class based on various attributes. This algorithm is mostly used in text classification and with problems having multiple classes.

  What are the Pros and Cons of Naive Bayes?

  Pros:

  It is easy and fast to predict class of test data set. It also perform well in multi class prediction When assumption of independence holds, a Naive Bayes classifier performs better compare to other models like logistic regression and you need less training data.

  It perform well in case of categorical input variables compared to numerical variable(s). For numerical variable, normal distribution is assumed (bell curve, which is a strong assumption).

  Cons:

  If categorical variable has a category (in test data set), which was not observed in training data set, then model will assign a 0 (zero) probability and will be unable to make a prediction. This is often

  known as "Zero Frequency". To solve this, we can use the smoothing technique. One of the simplest smoothing techniques is called Laplace estimation.

  On the other side naive Bayes is also known as a bad estimator, so the probability outputs from

  predict_proba are not to be taken too seriously.

  Another limitation of Naive Bayes is the assumption of independent predictors. In real life, it is almost impossible that we get a set of predictors which are completely independent.

  4 Applications of Naive Bayes Algorithms

  Real time Prediction: Naive Bayes is an eager learning classifier and it is sure fast. Thus, it could be used for making predictions in real time.

  Multi class Prediction: This algorithm is also well known for multi class prediction feature. Here we can predict the probability of multiple classes of target variable.

  Text classification/ Spam Filtering/ Sentiment Analysis: Naive Bayes classifiers mostly used in text classification (due to better result in multi class problems and independence rule) have higher success rate as compared to other algorithms. As a result, it is widely used in Spam filtering (identify spam e- mail) and Sentiment Analysis (in social media analysis, to identify positive and negative customer sentiments)-Recommendation System: Naive Bayes Classifier and Collaborative Filtering together builds a Recommendation System that uses machine learning and data mining techniques to filter unseen information and predict whether a user would like a given resource or not

How to build a basic model using Naive Bayes in Python and R?

Again, scikit learn (python library) will help here to build a Naive Bayes model in Python. There are three types of Naive Bayes model under the scikit-learn library:

Gaussian: It is used in classification and it assumes that features follow a normal distribution.

Multinomial: It is used for discrete counts. For example, let's say,   we   have   a   text   class ification problem. Here we can consider Bernoulli trials which is one step further and instead of "word occurring in the document", we have "count how often word occurs in the document", you can think of it as "number of times outcome number x_i is observed over the n trials".

Bernoulli: The binomial model is useful if your feature vectors are binary (i.e. zeros and ones). One application would be text classification   with   'bag   of words' model where the 1s   & 0s are "word   occurs in the document" and "word does not occur in the document" respectively.

R Code:

require(e1071) #Holds the Naive Bayes Classifier Train <- read.csv(file.choose()) Test <- read.csv(file.choose()) #Make sure the target variable is of a two-class classification problem only levels(Train$Item_Fat_Content) model <- naiveBayes(Item_69Fat_Content~., data = Train) class(model) pred <- predict(model,Test) table(pred)

Above, we looked at the basic Naive Bayes model, you can improve the power of this basic model by tuning parameters and handle assumption intelligently.   Let's look at the methods to improve the performance of Naive Bayes Model. I'd recommend you to go through this document for more details on Text classification using Naive Bayes.

Tips to improve the power of Naive Bayes Model

Here are some tips for improving power of Naive Bayes Model:

If continuous features do not have normal distribution, we should use transformation or different methods to convert it in normal distribution.

If test data set   has   zero frequency issue, apply smoothing techniques "Laplace Correction"   to predict the class of test data set.

Remove correlated features, as the highly correlated features are voted   twice in   the model and it   can lead to over inflating importance.

Naive Bayes classifiers has limited options for parameter tuning like alpha=1 for smoothing, fit_prior= [True|False] to learn class prior probabilities or not and   some   other   options (look at detail here). I would recommend to focus on your pre-processing of data and the feature selection.

You might think to apply some classifier combination technique like ensembling, bagging and boosting but these methods would not help. Actually, “ensembling, boosting, bagging” won’t help since their purpose is to reduce variance. Naive Bayes has no variance to minimize.

End Notes

In this article, we looked at one of the supervised machine learning algorithm “Naive Bayes” mainly used for classification. Congrats, if you’ve thoroughly & understood this article, you’ve already taken you first step to master this algorithm. From here, all you need is practice.

Chapter 4
RESULTS AND DISCUSSION

4.1 Machine learning :

What are the 7 steps of machine learning?

7 Steps of Machine Learning

Step 1: Gathering Data. ...

Step 2: Preparing that Data. ...

Step 3: Choosing a Model. ...

Step 4: Training. ...

Step 5: Evaluation. ...

Step 6: Hyper parameter Tuning. ...

Step 7: Prediction.

Introduction:

In this blog, we will discuss the workflow of a Machine learning project this includes all the steps required to build the proper machine learning project from scratch.

We will also go over data pre-processing, data cleaning, feature exploration and feature engineering and show the impact that it has on Machine Learning Model Performance. We will also cover a couple of the pre-modelling steps that can help to improve the model performance.

Python Libraries that would be need to achieve the task:

1. Numpy
2. Pandas
3. Sci-kit Learn
4. Matplotlib

Understanding the machine learning workflow

We can define the machine learning workflow in 3 stages.

Gathering data

Data pre-processing

Researching the model that will be best for the type of data

Training and testing the model

Evaluation

Okay but first let’s start from the basics

What is the machine learning Model?

The machine learning model is nothing but a piece of code; an engineer or data scientist makes it smart through training with data. So, if you give garbage to the model, you will get garbage in return, i.e. the trained model will provide false or wrong prediction

1. Gathering Data

The process of gathering data depends on the type of project we desire to make, if we want to make an ML project that uses real-time data, then we can build an IoT system that using different sensors data. The data set can be collected from various sources such as a file, database, sensor and many other such sources but the collected data cannot be used directly for performing the analysis process as there might be a lot of missing data, extremely large values, unorganized text data or noisy data. Therefore, to solve this problem Data Preparation is done.

We can also use some free data sets which are present on the internet. Kaggle and UCI Machine learning Repository are the repositories that are used the most for making Machine learning models. Kaggle is one of the most visited websites that is used for practicing machine learning algorithms, they also host competitions in which people can participate and get to test their knowledge of machine learning.

2. Data pre-processing

Data pre-processing is one of the most important steps in machine learning. It is the most important step that helps in building machine learning models more accurately. In machine learning, there is an 80/20 rule. Every data scientist should spend 80% time for data per-processing and 20% time to actually perform the analysis.

What is data pre-processing?

Data pre-processing is a process of cleaning the raw data i.e. the data is collected in the real world and is converted to a clean data set. In other words, whenever the data is gathered from different sources it is collected in a raw format and this data isn’t feasible for the analysis.

Therefore, certain steps are executed to convert the data into a small clean data set, this part of the process is called as data pre-processing.

Why do we need it?

As we know that data pre-processing is a process of cleaning the raw data into clean data, so that can be used to train the model. So, we definitely need data pre-processing to achieve good results from the applied model in machine learning and deep learning projects.Most of the real-world data is messy, some of these types of data are:

1. Missing data: Missing data can be found when it is not continuously created or due to technical issues in the application (IOT system).
2. Noisy data: This type of data is also called outliners, this can occur due to human errors (human manually gathering the data) or some technical problem of the device at the time of collection of data.
3. Inconsistent data: This type of data might be collected due to human errors (mistakes with the name or values) or duplication of data.

Three Types of Data

1. Numeric e.g. income, age
2. Categorical e.g. gender, nationality
3. Ordinal e.g. low/medium/high

How can data pre-processing be performed?

These are some of the basic pre — processing techniques that can be used to convert raw data.

1. Conversion of data: As we know that Machine Learning models can only handle numeric features, hence categorical and ordinal data must be somehow converted into numeric features.
2. Ignoring the missing values: Whenever we encounter missing data in the data set then we can remove the row or column of data depending on our need. This method is known to be efficient but it shouldn't be performed if there are a lot of missing values in the dataset.
3. Filling the missing values: Whenever we encounter missing data in the data set then we can fill the missing data manually, most commonly the mean, median or highest frequency value is used.
4. Machine learning: If we have some missing data then we can predict what data shall be present at the empty position by using the existing data.
5. Outliers detection: There are some error data that might be present in our data set that deviates drastically from other observations in a data set. [Example: human weight = 800 Kg; due to mistyping of extra 0]

Researching the model that will be best for the type of data

Our main goal is to train the best performing model possible, using the pre-processed data.

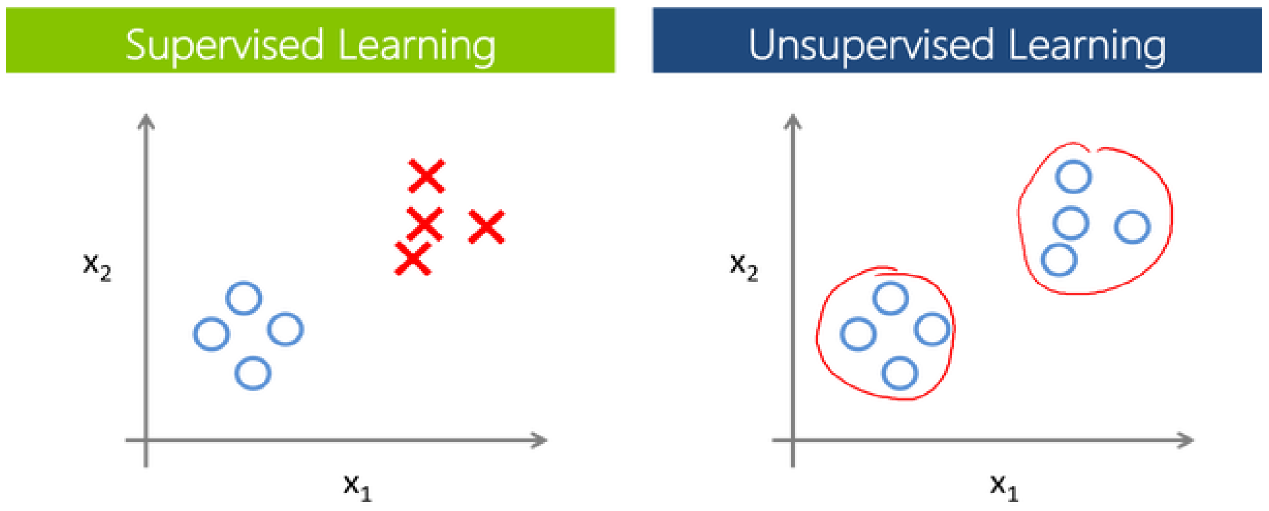


Fig 4.1 Supervised Learning and Unsupervised Learning

Supervised Learning:

In Supervised learning, an AI system is presented with data which is labelled, which means that each data tagged with the correct label.

The supervised learning is categorized into 2 other categories which are "Classification" and "Regression".

Classification:

Classification problem is when the target variable is categorical (i.e. the output could be classified into classes — it belongs to either Class A or B or something else).

A classification problem is when the output variable is a category, such as "red" or "blue" , "disease" or "no disease" or "spam" or "not spam".

Fig 4.2 Decision Boundary

As shown in the above representation, we have 2 classes which are plotted on the graph i.e. red and blue which can be represented as 'setosa flower' and 'versicolor flower', we can image the X-axis as there 'Sepal Width' and the Y-axis as the 'Sepal Length', so we try to create the best fit line that separates both classes of flowers.

These some most used classification algorithms.

K-Nearest Neighbor

Naive Bayes

Decision Trees/Random Forest

Support Vector Machine

Logistic Regression

Regression:

While a Regression problem is when the target variable is continuous (i.e. the output is numeric).

Fig 4.3 Gradient Search and Points and Line

As shown in the above representation, we can imagine that the graph's X-axis is the 'Test scores' and the Y-axis represents 'IQ'. So we try to create the best fit line in the given graph so that we can use that line to predict any approximate IQ that isn't present in the given data.

These some most used regression algorithms.

Linear Regression

Support Vector Regression

Decision Tress/Random Forest

Gaussian Progresses Regression

Ensemble Methods

Unsupervised Learning:

The unsupervised learning is categorized into 2 other categories which are "Clustering" and "Association".

Clustering:

A set of inputs is to be divided into groups. Unlike in classification, the groups are not known beforehand, making this typically an unsupervised task.

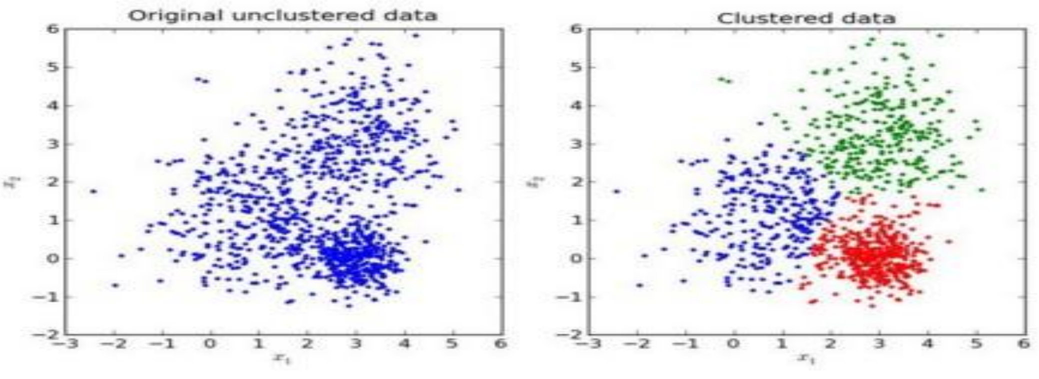


Fig 4.4 Original Clustered Data and Clustered Data

Methods used for clustering are:

Gaussian mixtures

K-Means Clustering

Boosting

Hierarchical Clustering

K-Means Clustering

Spectral Clustering

Overview of models under categories:

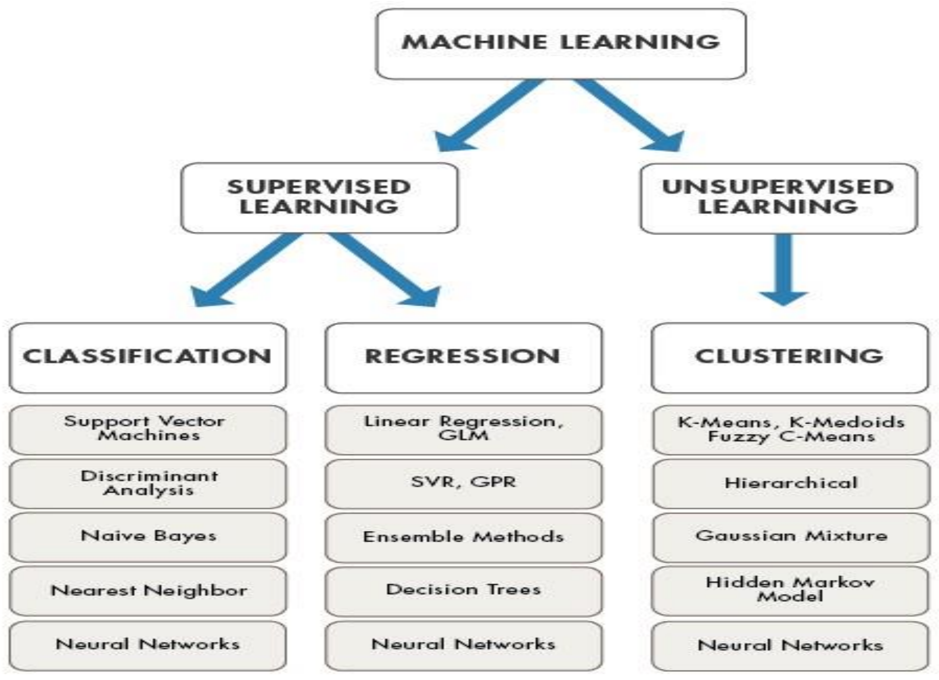


Fig 4.5 Machine Learning

4. Training and testing the model on data

For training a model we initially split the model into 3 three sections which are 'Training data' ,'Validation data' and 'Testing data'.You train the classifier using 'training data set', tune the parameters using 'validation set' and then test the performance of your classifier on unseen 'test data set'. An important point to note is that during training the classifier only the training and/or validation set is available. The test data set must not be used during training the classifier. The test set will only be available during testing the classifier.

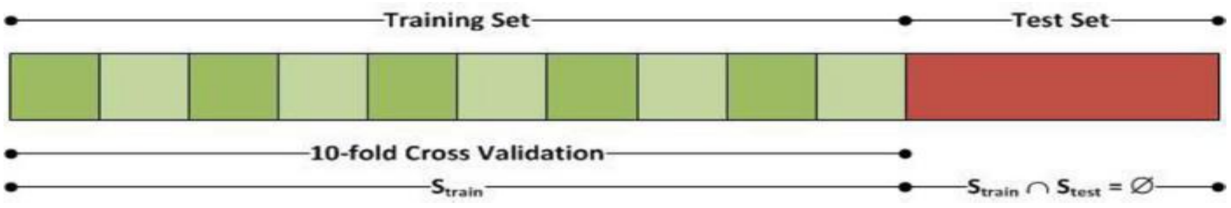


Fig 4.6 Training Set

Training set:

The training set is the material through which the computer learns how to process information. Machine learning uses algorithms to perform the training part. A set of data used for learning, that is to fit the parameters of the classifier.

Validation set:

Cross-validation is primarily used in applied machine learning to estimate the skill of a machine learning model on unseen data. A set of unseen data is used from the training data to tune the parameters of a classifier.

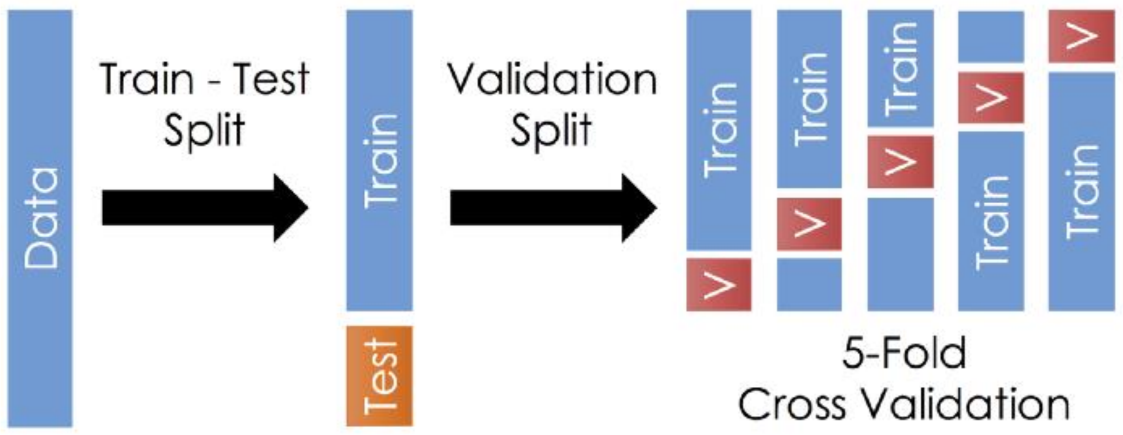


Fig 4.7 Validation Set

Once the data is divided into the 3 given segments we can start the training process.

In a data set, a training set is implemented to build up a model, while a test (or validation) set is to validate the model built. Data points in the training set are excluded from the test (validation) set. Usually, a data set is divided into a training set, a validation set (some people use ‘test set’ instead) in each iteration, or divided into a training set, a validation set and a test set in each iteration.

The model uses any one of the models that we had chosen in step 3/ point 3. Once the model is trained we can use the same trained model to predict using the testing data i.e. the unseen data.

Once this is done we can develop a confusion matrix, this tells us how well our model is trained. A confusion matrix has 4 parameters, which are 'True positives', 'True Negatives', 'False Positives' and 'False Negative'. We prefer that we get more values in the True negatives and true positives to get a more accurate model. The size of the Confusion matrix completely depends upon the number of classes.

n=165	Predicted: NO	Predicted: YES
Actual: NO	50	10
Actual: YES	5	100

Table 4.1 Predicted Output

True positives : These are cases in which we predicted TRUE and our predicted output is correct.
True negatives : We predicted FALSE and our predicted output is correct.
False positives : We predicted TRUE, but the actual predicted output is FALSE.
False negatives : We predicted FALSE, but the actual predicted output is TRUE.
We can also find out the accuracy of the model using the confusion matrix.
Accuracy = (True Positives +True Negatives) / (Total number of classes)
i.e. for the above example:
Accuracy = (100 + 50) / 165 = 0.9090 (90.9% accuracy)

5. Evaluation

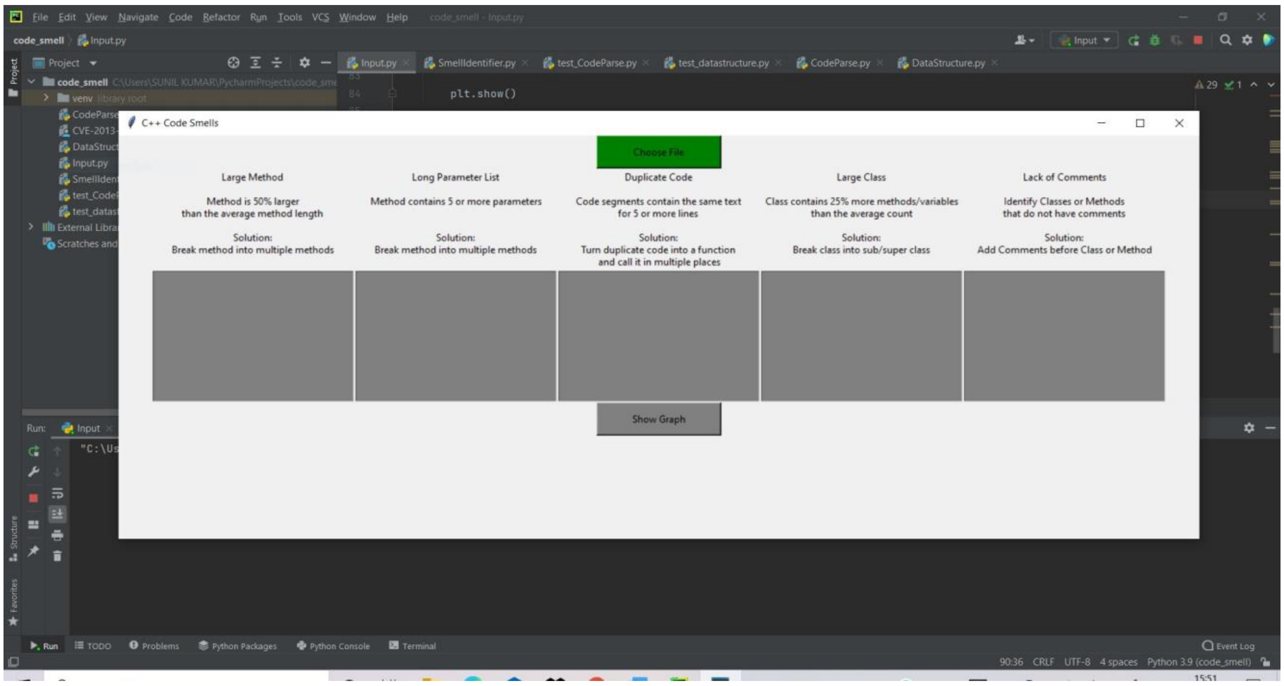
Model Evaluation is an integral part of the model development process. It helps to find the best model that represents our data and how well the chosen model will work in the future.
To improve the model we might tune the hyper-parameters of the model and try to improve the accuracy and also looking at the confusion matrix to try to increase the number of true positives and true negatives.

6.Conclusion

In this blog, we have discussed the workflow a Machine learning project and gives us a basic idea of how a should the problem be tackled.

OUTPUTS

OUTPUTS:



Large Method

Method is 50% larger
than the average method length

Solution:
Break method into multiple methods

Long Parameter List

Method contains 5 or more parameters

Solution:
Break method into multiple methods

Duplicate Code

Code segments contain the same text
for 5 or more lines

Solution:
Turn duplicate code into a function
and call it in multiple places

Large Class

Class contains 25% more methods/variables
than the average count

Solution:
Break class into sub/super class

Lack of Comments

Identify Classes or Methods
that do not have comments

Solution:
Add Comments before Class or Method

Fig 4.8 Sample code Window

GRAPH:

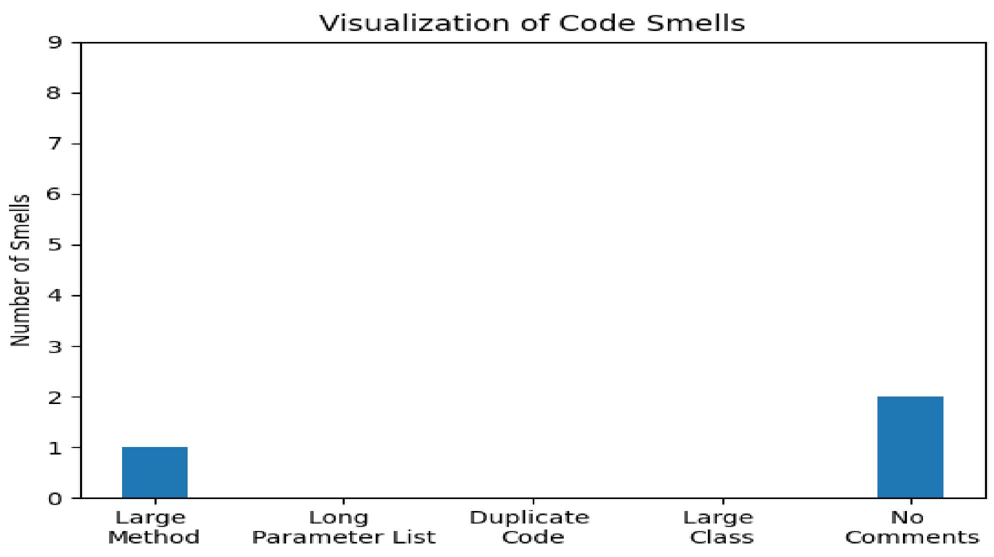


Fig 4.9 Code Smell Graph

Chapter 5

CONCLUSION

Conclusion :

We observing from various codes smell regarding research paper that detects bad codes smell by using various tools and techniques. For Code smell detection various machine learning techniques are used. Common instances are used to combined for dataset and remove the dissimilar instances. Multilabel classification is one of the most used techniques of machine learning to detect the code smells from object-oriented programming code. Multilabel classification is used to detect two types of code smells.

Existing previous researches were done to detect only one type of smell in the source code. Multilabel classification choose the dataset then finds the affected and non-affected bad smells from the source code and common instances are avoid dissimilar data. The chain classifier provides the best result as compared to long-chain that improves the performance and accuracy.

REFERENCES

1. Stol, K.-J. and Fitzgerald, B. (2014). Researching crowdsourcing software development: Perspectives and concerns. In Proceedings of the 1st InternationalWorkshop on CrowdSourcing in Software Engineering, CSI-SE 2014, page 7–10,
2. Stone, M. (1974). Cross-validatory choice and assessment of statistical predictions.Journal of the Royal Statistical Society. Series B (Methodological), 36(2):111–147.
3. Tahir, A., Yamashita, A., Licorish, S., Dietrich, J., and Counsell, S. (2018). Canyou tell me if it smells? a study on how developers discuss code smells and antipatterns in stack overflow. In Proceedings of the 22nd International Conferenceon Evaluation and Assessment in Software Engineering 2018, EASE'18, page68–78, New York, NY, USA. Association for Computing Machinery.
4. Travassos, G., Shull, F., Fredericks, M., and Basili, V. R. (1999). Detecting defectsin object-oriented designs: Using reading techniques to increase software quality.InProceedings of the 14th conference on object oriented programming, systems,languages, and applications, pages 47–56, New York, NY, USA. ACM Press.
5. Wang, C., Hirasawa, S., Takizawa, H., and Kobayashi, H. (2015). Identification andElimination of Platform-Specific Code Smells in High Performance ComputingApplications. International Journal of Networking and Computing, 5(1):180–199.
6. Yamashita, A. and Moonen, L. (2013). To what extent can maintenance problemsbe predicted by code smell detection? - an empirical study. Inf. Softw. Technol.,55(12):2223–2242.
7. Sjøberg D I K, Yamashita A, Anda B C D, Mockus A, Dyb˚a T. Quantifying the effect of code smells on maintenance effort. IEEE Trans. Softw. Eng., 2013, 39(8): 1144-1156.
8. Sahin D, Kessentini M, Bechikh S, Ded K. Code-smells detection as a bi-level problem. ACM Trans. Softw. Eng. Methodol., 2014, 24(1): Article No. 6.
9. Olbrich S, Cruzes D S, Basili V, Zazworka N. The evolution and impact of code smells: A case study of two open source systems. In Proc. the 3rd International Symposium on Empirical Software Engineering and Measurement, October 2009, pp.390-400.
10. Olbrich S M, Cruzes D S, Sjoøberg D I K. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In Proc. the 26th IEEE Int. Conf. Softw. Maintenance, September 2010.1444 J. Comput. Sci. & Technol., Nov. 2020, Vol.35, No.6
11. Khomh F, Penta D M, Gu´eh´eneuc Y G. An exploratory study of the impact of code smells on software changeproneness. In Proc. the 16th Working Conference on Reverse Engineering, October 2009, pp.75-84.

12. Deligiannis I, Stamelos I, Angelis L, Roumeliotis M, Shepperd M. A controlled experiment investigation of an objectoriented design heuristic for maintainability. J. Syst. Softw., 2004, 72(2): 129-143.
13. P´erez-Castillo R, Piattini M. Analyzing the harmful effect of god class refactoring on power consumption. IEEE Softw., 2014, 31(3): 48-54.
14. Li W, Shatnawi R. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J. Syst. Softw., 2007, 80(7): 1120-1128.
15. Ciupke O. Automatic detection of design problems in object-oriented reengineering. In Proc. the 30th International Conference on Technology of Object-Oriented Languages and Systems, Delivering Quality Software, August 1999, pp.18-32.
16. Travassos G, Shull F, Fredericks M, Basili V R. Detecting defects in object-oriented designs: Using reading techniques to increase software quality. ACM SIGPLAN Notices, 1999, 34(10): 47-56.
17. Dashofy E M, van der Hoek A, Taylor R N. A comprehensive approach for the development of modular software architecture description languages. ACM Trans. Softw. Eng. Methodol., 2005, 14(2): 199-245.
18. Vidal S, V´azquez H, D´ıaz-Pace J A, Marcos C, Garcia A, Oizumi W. JSpIRIT: A flflexible tool for the analysis of code smells. In Proc. the 34th Int. Conf. Chil. Comput. Sci. Soc., November 2016.
19. Marinescu R. Measurement and quality in object-oriented design. In Proc. the 21st IEEE Int. Conf. Softw. Maintenance, September 2005, pp.701-704.
20. Moha N, Gu´eh´eneuc Y, Duchien L, le Meur A. DECOR: A method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng., 2010, 36(1): 20-36.

APPENDICES

A. SOURCE CODE:

```
import SmellIdentifier
import tkinter.filedialog as filedialog
import tkinter as tk
import numpy as np
import matplotlib
matplotlib.use("TkAgg")
from matplotlib import pyplot as plt

class SmellType:
    name = str()
    fix = str()
    cause = str()

    def __init__(self, name, fix, cause):
        self.name = name
self.fix = fix
self.cause = cause

smell_types = [SmellType("Large Method", "Break method into multiple methods\n", "Method is 50% larger\nthan the average method length"),
SmellType("Long Parameter List", "Break method into multiple methods\n", "Method contains 5 or more parameters\n"),
SmellType("Duplicate Code", "Turn duplicate code into a function\n and call it in multiple places", "Code segments contain the same text\nfor 5 or more lines"),
SmellType("Large Class", "Break class into sub/super class\n", "Class contains 25% more methods/variables\nthan the average count"),
SmellType("Lack of Comments", "Add Comments before Class or Method\n", "Identify Classes or Methods\nthat do not have comments")]

labels = list()
boxes = list()
```

```
cur_file_name = str()
smell_counts = list()

def find_smell(name):
    for i in range(len(smell_types)):
        if (name == smell_types[i].name):
            return i
    return -1

def open_file():
    global cur_file_name
    global smell_counts
    # try to open file
    try:
cur_file_name = filedialog.askopenfilename(filetypes =(("C++ File", "*.cpp"),("All Files","*.*")),
                          title = "Choose a file.")
        smells = SmellIdentifier.get_smells(cur_file_name)
smell_counts = [0] * len(smell_types)


        # clearlistboxes
        for box in boxes:
box.delete(0, tk.END)
box.config(bg="LIGHT GREY")
        button2.config(bg="LIGHT GREEN")
        # add each smell
        for smell in smells:
            index = find_smell(smell.smell_type)
            if (index >= 0):
smell_counts[index] += 1
                boxes[index].insert(tk.END, f"line #{smell.line_num}")
                boxes[index].insert(tk.END, "    " + smell.description)
            else:
print("Invalid Smell", smell.smell_type, smell.description)
    except Exception as e:
        button2.config(bg="GREY")
cur_file_name = str()
        print(str(e))

def display_file():
    if (cur_file_name != ""):
        highest = max(smell_counts)

ind = np.arange(5)    # the x locations for the groups
        width = 0.35       # the width of the bars: can also be len(x) sequence

plt.bar(ind, smell_counts, width, yerr=0)

plt.ylabel('Number of Smells')
plt.title('Visualization of Code Smells')
plt.xticks(ind, ('Large \nMethod', 'Long \nParameter List', 'Duplicate \nCode',
            'Large \nClass', 'No \nComments'))
        if highest > 11:
plt.yticks(np.arange(0, highest + 2, 10))
        else:
plt.yticks(np.arange(0, 10, 1))

plt.show()

if (__name__ == "__main__"):

    window = tk.Tk()
window.config(width=1300, height=500)
window.title("C++ Code Smells")
    #window.state("zoomed")

    canvas = tk.Canvas(window)
canvas.place(relx=0.5, rely=0.0, anchor=tk.N)
```

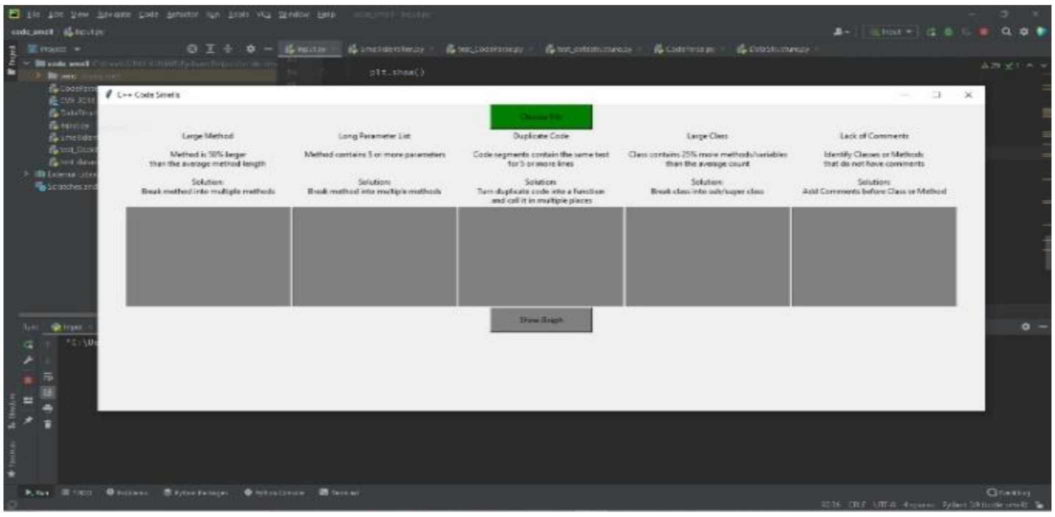
```
    button = tk.Button(canvas, text="Choose File")
button.config(width=20, height=2, command=open_file, bg="GREEN")
button.grid(row=0, column=0, columnspan=5)

    for i, smell_type in enumerate(smell_types):
        header = f"{smell_type.name}\n\n{smell_type.cause}\n\nSolution:\n{smell_type.fix}"
labels.append(tk.Label(canvas, text=header))
        labels[i].grid(row=1, column=i)
boxes.append(tk.Listbox(canvas, width=40, height=10, bg="GREY"))
        boxes[i].grid(row=2, column=i)

    button2 = tk.Button(canvas, text="Show Graph")
    button2.config(width=20, height=2, command=display_file, bg="GREY")
    button2.grid(row=3, column=0, columnspan=5)

window.mainloop()
```

SCREENSHOTS



Large Method

Method is 50% larger
than the average method length

Solution:
Break method into multiple methods

Long Parameter List

Method contains 5 or more parameters

Solution:
Break method into multiple methods

Duplicate Code

Code segments contain the same text
for 5 or more lines

Solution:
Turn duplicate code into a function
and call it in multiple places

Large Class

Class contains 25% more methods/variables
than the average count

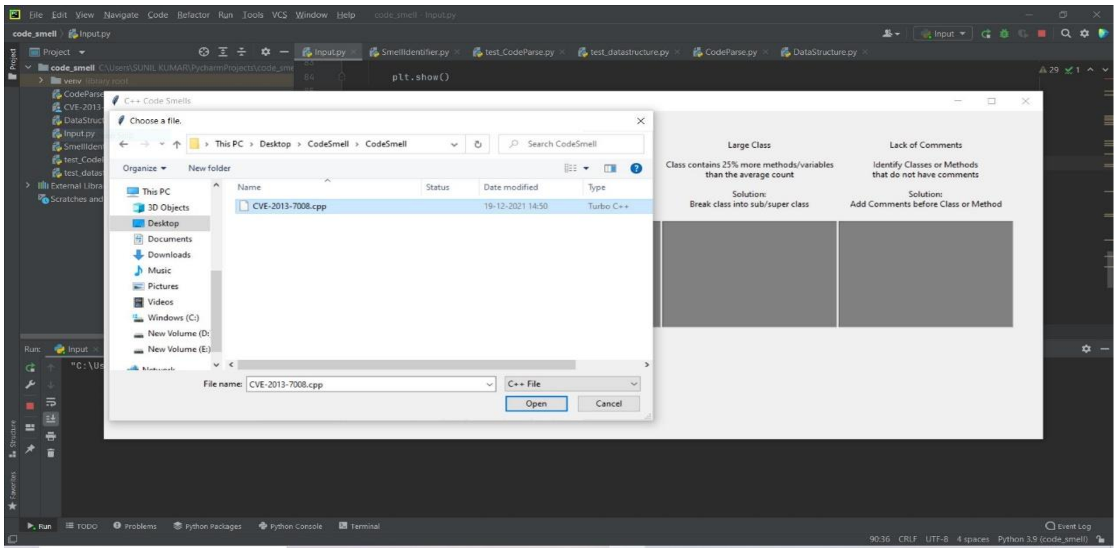
Solution:
Break class into sub/super class

Lack of Comments

Identify Classes or Methods
that do not have comments

Solution:
Add Comments before Class or Method

Fig 1 Input Window



Name	Status	Date modified	Type
CVE-2013-7008.cpp		19-12-2021 14:50	Turbo C++

File name: CVE-2013-7008.cpp

C++ File

Large Class

Class contains 25% more methods/variables
than the average count

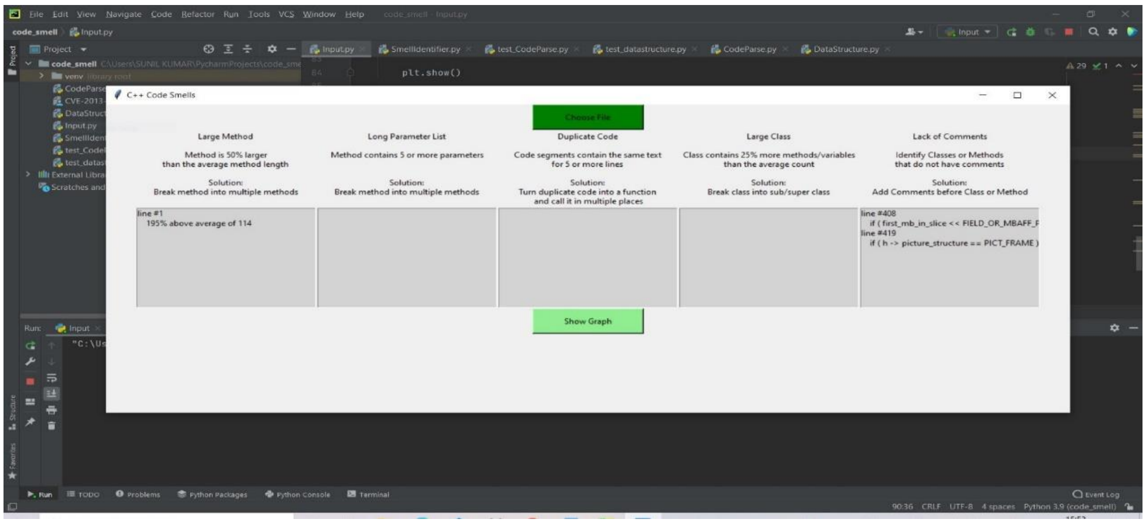
Solution:
Break class into sub/super class

Lack of Comments

Identify Classes or Methods
that do not have comments

Solution:
Add Comments before Class or Method

Fig 2 Selecting the cpp file



```
plt.show()
```

Large Method	Long Parameter List	Duplicate Code	Large Class	Lack of Comments
Method is 50% larger than the average method length	Method contains 5 or more parameters	Code segments contain the same text for 5 or more lines	Class contains 25% more methods/variables than the average count	Identify Classes or Methods that do not have comments
Solution: Break method into multiple methods	Solution: Break method into multiple methods	Solution: Turn duplicate code into a function and call it in multiple places	Solution: Break class into sub/super class	Solution: Add Comments before Class or Method

```
line #1
  195% above average of 114
```

```
line #408
  if ( first_mb_in_slice << FIELD_OR_MBAFF_F
line #419
  if ( h -> picture_structure == PICT_FRAME )
```

Fig 3 Sample Code Window

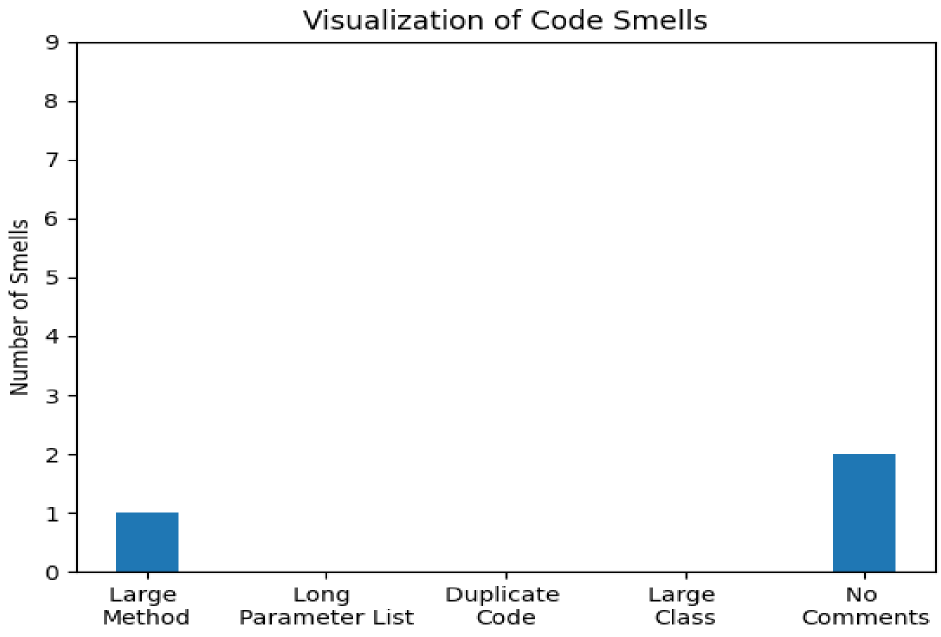
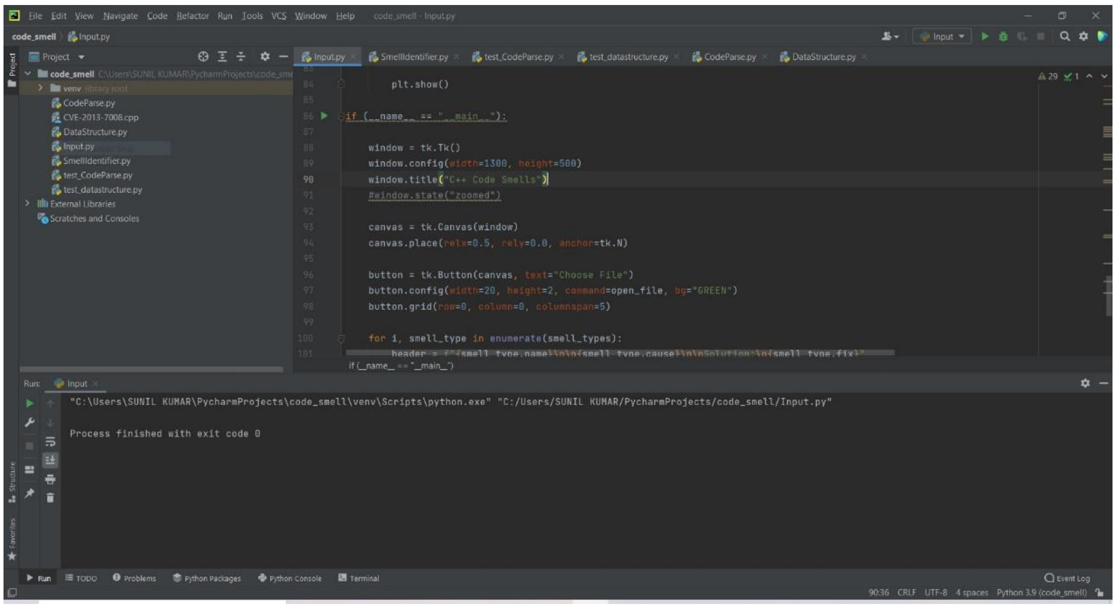


Fig 4 Code Smell Graph


```
        plt.show()

if (__name__ == "__main__"):

    window = tk.Tk()
    window.config(width=1300, height=500)
    window.title("C++ Code Smells")
    #window.state("zoomed")

    canvas = tk.Canvas(window)
    canvas.place(relx=0.5, rely=0.0, anchor=tk.N)

    button = tk.Button(canvas, text="Choose File")
    button.config(width=20, height=2, command=open_file, bg="GREEN")
    button.grid(row=0, column=0, columnspan=5)

    for i, smell_type in enumerate(smell_types):
        header = f"{smell_type.name}\n\n{smell_type.cause}\n\nSolution:\n{smell_type.fix}"
```

```
"C:\Users\SUNIL KUMAR\PycharmProjects\code_smell\venv\Scripts\python.exe" "C:/Users/SUNIL KUMAR/PycharmProjects/code_smell/Input.py"

Process finished with exit code 0
```

File Edit View Navigate Code Refactor Run Tools VCS Window Help
code_smell - Input.py
code_smell
Input.py
SmellIdentifier.py
test_CodeParse.py
test_datastructure.py
CodeParse.py
DataStructure.py
code_smell C:\Users\SUNIL KUMAR\PycharmProjects\code_sme
venv library root
CVE-2013-7008.cpp
External Libraries
Scratches and Consoles
if (__name__ == "__main__")
Run: Input
Run TODO Problems Python Packages Python Console Terminal
Event Log
90:36 CRLF UTF-8 4 spaces Python 3.9 (code_smell)

Fig 5 Output Window