Project Report
CSN-352
Compiler Design

Group ID: 29

Member 1: Rajat Raj Singh E. No: 21114079
Member 2: Pranavdeep Singh E. No: 21119036
Member 3: Piyush Arya E. No: 21114074

Problem statement:

Detection of high-level language of a given source code using lexical analysis followed by NLP and Machine Learning techniques.

Project Implementation:

The development of our project is divided into two stages:

1. Development of the lexer and preparation of the dataset which contains source files of different languages.
2. Development of different machine learning models for training and prediction.

Dataset preparation:

For dataset preparation we have downloaded source codes of 4 different languages which are- C++, Java, Python and SQL from various repositories on GitHub. We have collected 250 source codes of each of these languages. Our dataset consists of a directory which has 4 folders in it and each folder corresponds to a language which contains the respective source code files. We will run our lexer on this directory.

Lexical Analysis:

Each source code file is first pre-processed before passing to the lexer for lexical analysis. In the pre-processing, all the types of comments (single line or multi line) and redundant empty lines are removed.

The lexer takes all these preprocessed files as input one by one and prepares a feature vector. This feature vector stores the frequency of the tokens in the source file i.e. the number of times a particular token identified by the lexer has occurred in the entire source code file. Its length is equal to the total number of tokens that can be identified by the lexer.

In our lexer implementation, we have identified only 64 of the key tokens across all the 4 different languages as we want to limit the size of our feature vector. Since we are not identifying all the tokens of a given source file, we modified the lexer so as not to give any lexical error on these unidentified tokens.

Each token in the lexer is defined by a Regular Expression. The Regular Expressions are implemented as a Deterministic Finite Automata. Every token is passed through each of these Regular Expressions and longest prefix-matching principle is used to match the input string.

If the string has potential to match more than one RE, we will continue to match until potential becomes one and it finally matches a RE. If the string has more than one potential and it matches more than one RE and matching the next character makes its potential zero for all RE's then the RE which is defined first in the implementation is taken as the match. When a string fails to match any RE but still holds the potential to do so, the lexer will attempt to match the subsequent character in the string until its potential for matching becomes zero. If no RE's are matched once the potential reaches zero, the lexer will backtrack to the starting character of the failed match and start the matching process from the next character onwards. In case an unidentified string is encountered, it will be disregarded, and the lexer will continue its search for the next valid token. After the lexical analysis phase is complete all the source code files are processed, the resultant feature vectors generated from all the files will be used as input to the machine learning models for training and analysis purposes.

Applying ML models:

We use classical machine learning models for our multi-class classification task. Given that our feature vector contains only numerical data, we can use a variety of machine-learning models.

We experimented using various ML models like Random Forests, Gradient Boosted Trees, and K Nearest Neighbors Classifiers. The reason for our choice of random forests and gradient-boosted trees was due to the ability of decision trees to partition the data into different segments based on a feature when creating a node in the tree for splitting. Such behaviour is desirable for our use case as the presence of a token like "SELECT" is a very strong indicator that the source language can be SQL. Similarly, for other tokens like "print" or "def" in Python and "cout" in C++, etc. The reason for using a K Nearest Neighbors model was based on the underlying assumption that source programs written in the same language have certain common patterns regarding the distribution of frequency of tokens, which can be explored using a model that predicts on the basis of the distance between these high-dimensional feature vectors.

Experimentation:

Before training the models, we separate our data into training and testing data in order to keep some data unseen to the model which can be used to assess the model's performance based on unseen data.

We applied a stratified test-train split on our dataset to ensure that our training data and test data contain the same ratio of examples of each of the 4 classes. This ensures that the results we observe will be fair and not skewed towards any particular class. In case we don't stratify the split based on the class, and the test data contains only examples from 3 of the classes due to random sampling, we

would not be able to get a realistic measure of how the model will perform toward unseen data at the time of deployment.

Once we have separated our data into train and test, we only train our models and tune it's hyperparameters based on the training data. Only once the models are trained, do we use the test data to avoid any data leakage.

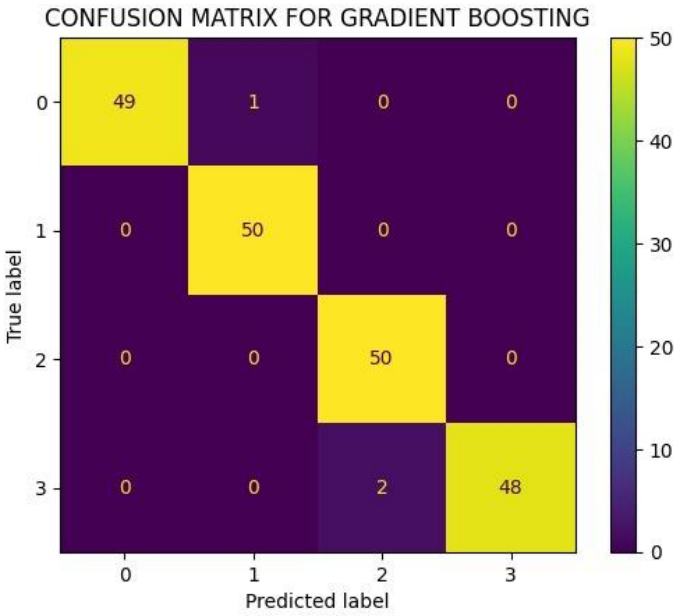
Training the models without any hyperparameter tuning may lead to poor results. As we know, Random Forests and Gradient Boosted Trees are ensembles of Decision Trees, so it is very important to prevent overfitting in these models. Decision trees are extremely prone to overfitting due to their high model flexibility.

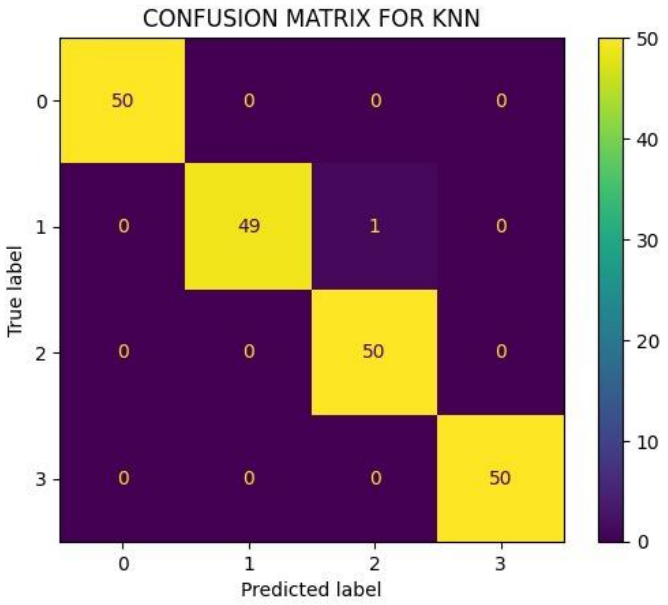
To regularise all of our models, we used k-fold cross-validation to tune our hyperparameters and chose the parameters that gave the best results on the validation dataset to avoid overfitting. K-fold cross-validation is a technique where we split our training data into k equal segments. Then we use (k-1) of these segments to train the model, and use the remaining segment for calculating the validation score for that model. This is run k times, taking all segments as the validation segment once, and the overall validation error is the average of the k validation errors we observed. This provides a simple framework to tune the hyperparameters of the model to avoid overfitting while ensuring optimal performance.

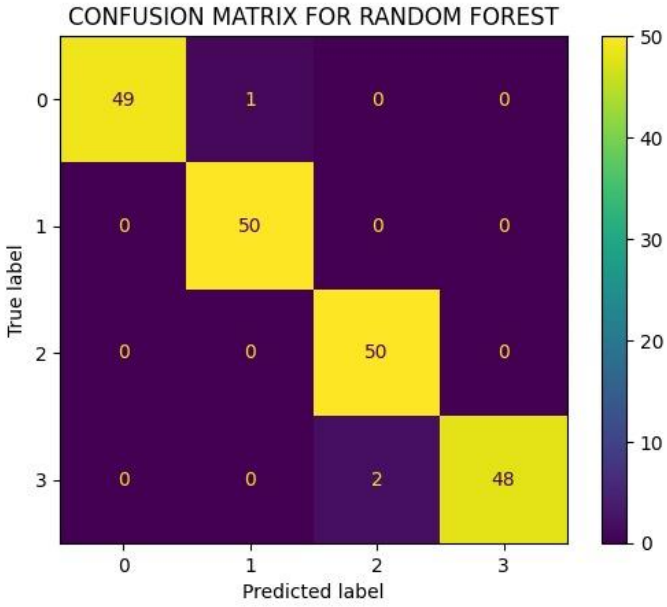
The hyperparameters we optimized for the various models:

1. Random Forest:
    a. Max depth: This applies a limit onto the maximum depth a single decision tree/ estimator can have in the ensemble.
    b. Number of estimators: This represents the total number of decision trees in the ensemble.
    c. Criteria for splitting: This is a function which is used to determine which feature should be used to split at a given node in a tree.
2. Gradient Boosted Trees:
    a. Learning Rate: This is a parameter which decides the rate at which misclassified examples' weights are increased as we train subsequent trees.
    b. Number of estimators: This represents the total number of decision trees in the ensemble.
    c. Max depth: This applies a limit onto the maximum depth a single decision tree/ estimator can have in the ensemble.
    d. Max features: This applies a limit to the number of features a single decision tree can use. This allows for the implementation of feature bagging.
3. K Nearest Neighbors:
    a. Number of neighbours: This parameter controls the number of examples that are used to classify a data point at the time of prediction.
    b. Weights: This parameter allows to weight the voting of the neighbours based on their distance from the data point being classified.
    c. P: This parameter signifies the distance function. P=1 represents L1 distance, and P=2 represents L2 distance.

Upon training all of these models, we constructed their confusion matrices for the model's prediction on the unseen test data to analyse the models.







The labels represent-

0 -> C++

1 -> Java

2 -> Python

3 -> SQL

We can observe that the results are satisfactory using all 3 models.

Since the KNN model does not have a very big improvement in performance, we do not use this model. The reason is that KNN is a lazy learner, i.e., it is a very time and compute-intensive algorithm when

we want to predict the label for some data point. Since our use case requires a fast detection of the source language, we prefer the other 2 models.

Inferencing the results:

The trained Random Forest provides us with valuable information other than the prediction capability itself. It provides us the relative importance of the features as used by the model. Upon inspecting these, we can get a deeper understanding about which features were useful in our task. Such analysis can help scale our model to other languages in an easier manner as we can try to extract similar features for other languages as well.

The top 10 features, as identified by the Random Forest Model, were:

{“bits/stdc++.h”, “namespace”, “main”, “;”, “input()”, “String”, “args”, “SELECT”, “FROM”, “}”}

Since our model provided us with satisfactory results, we do not need to apply any additional techniques. However, if this model is to be scaled to a large number of source languages, the task will become harder. To tackle that issue, we can use various techniques to make our feature vector more robust, which will allow our model to perform well in those cases as well. Some of these techniques are:

1. Use Term Frequency Inverse Document Frequency (TF-IDF) on the tokens.
2. Use a technique that captures the relative ordering of the tokens like n-grams or a time-series model.
3. Use of attention-based transformer models.

Relation to compiler design:

There are 2 phases in a compiler: the front-end phase and the back-end phase. The front-end phase is dependent on the source language, which is to be compiled by the compiler. The backend phase is independent of the source language as it takes the Intermediate Representation as input. The front end of the compiler consists of the lexical analysis phase, the syntax analysis phase, i.e., the parser, and finally, the semantic analysis phase.

As our problem statement suggests, we aim to create a pipeline that takes the source code as input and returns the predicted high-level programming language of that given source code. We use Machine Learning and NLP techniques to facilitate the above-mentioned task, where the feature extraction from the source code is enabled by the lexical analysis phase.

We specifically choose the lexical analysis phase due to a variety of reasons. As lexical analysis is the first stage of a compiler, it enables us to collect base-level features from the source code and provide a rich feature set to our model. Secondly, it is the only phase in the front end of the compiler that can be made language-independent with ease. In the case of a parser, it requires the production rules for the tokens as input, which are highly dependent on the source language and will not be known at the time of prediction. Lastly, tokenization is a well-known practice in NLP tasks as it is seen to capture information from the text in a robust way.

Taking all these points into consideration, our task is highly intermingled with the compiler design framework and our approach uses one of the most essential stages of the compiler with a few modifications to enable accurate predictions.

Test source code files examples:

Test code 1:

```
#include <iostream>
using namespace std;

int calculateSum(int arr[], int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }
    return sum;
}

int main() {
    int size;
    cout << "Enter the size of the array: ";
    cin >> size;

    int arr[size];

    cout << "Enter " << size << " integers:\n";
    for (int i = 0; i < size; i++) {
        cin >> arr[i];
    }

    int sum = calculateSum(arr, size);

    cout << "The sum of the array is: " << sum << endl;

    if (sum % 2 == 0) {
        cout << "The sum is even." << endl;
    } else {
        cout << "The sum is odd." << endl;
    }

    return 0;
}
```

Output: C++

Test code 2:

```
import java.util.Scanner;

public class Main {
    public static int calculateSum(int[] arr, int size) {
        int sum = 0;
        for (int i = 0; i < size; i++) {
            sum += arr[i];
        }
        return sum;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the size of the array: ");
        int size = scanner.nextInt();

        int[] arr = new int[size];

        System.out.println("Enter " + size + " integers:");
        for (int i = 0; i < size; i++) {
            arr[i] = scanner.nextInt();
        }

        int sum = calculateSum(arr, size);

        System.out.println("The sum of the array is: " + sum);

        if (sum % 2 == 0) {
            System.out.println("The sum is even.");
        } else {
            System.out.println("The sum is odd.");
        }

        scanner.close();
    }
}
```

Output: Java

Test Code 3:

```python
def calculate_sum(arr):
    return sum(arr)

def main():
    size = int(input("Enter the size of the array: "))
    arr = []

    print(f"Enter {size} integers:")
    for _ in range(size):
        arr.append(int(input()))

    sum_ = calculate_sum(arr)

    print(f"The sum of the array is: {sum_}")

    if sum_ % 2 == 0:
        print("The sum is even.")
    else:
        print("The sum is odd.")

if __name__ == "__main__":
    main()
```

Output: Python

Test code 4:

```sql
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    name VARCHAR(50),
    age INT,
    department VARCHAR(50),
    salary DECIMAL(10, 2)
);

INSERT INTO employees (employee_id, name, age, department, salary)
VALUES
    (1, 'John Doe', 30, 'Engineering', 60000.00),
    (2, 'Jane Smith', 25, 'Marketing', 45000.00),
    (3, 'Samuel Adams', 40, 'Finance', 75000.00),
    (4, 'Emily Johnson', 35, 'HR', 55000.00);

SELECT * FROM employees;

UPDATE employees
SET salary = 48000.00
WHERE employee_id = 2;

DELETE FROM employees
WHERE employee_id = 3;

SELECT * FROM employees;
```

Output: SQL