Compiler Provenance

Homework 1 - Machine Learning
Engineering in Computer Science
"La Sapienza" University of Rome

Costa Marco 1691388

10 November 2019

Contents

1 Introduction

The purpose of this homework is to apply machine learning algorithms to solve the compiler provenance problem. It consists of identifying the **compiler** and the **optimization** who produced a given binary code of a function. So we have two types of classification problems:

- **Binary classification**: optimization can be *HIGH (H)* or *LOW (L)*
- **Multi-class classification**: compiler can be *gcc*, *icc* or *clang*

2 Dataset

The dataset contains 30000 functions compiled with 3 different compilers (*gcc*, *icc*, *clang*). The compiler distribution is very balanced (10000 functions per compiler), while the optimizations distribution is not balanced. For each compiler different versions were used. The Dataset is provided as a jsonl file. Each row of the file is a json object with the following keys:

- **instructions**: the assembly instruction for the function.
- **opt**: the ground truth label for optimization (*H*, *L*)
- **compiler**: the ground truth label for compiler (*icc*, *clang*, *gcc*)

Moreover, there is a blind test set, which does not contain the label for the function. The goal is to classify the functions of this dataset after training the algorithm on the previous set.

3 Find best classifier

To find the algorithm that best solves the problem I decided to analyze several cases, using different types of features, classifiers and vectorizers.

3.1 Feature

I decided to use the Bag-of-Words approach. It consists in:

- Splitting the documents into tokens by following some sort of pattern. In this case, for each function there is a string composed of instructions separated by a comma, so it's possible to split it on the comma.

- Assigning a weight to each token proportional to the frequency with which it shows up in the document and/or corpora.
- Creating a document-term matrix with each row representing a document and each column addressing a token.

I used two types of different approaches. The first consists of using only the mnemonics instructions (as suggested in the paper), while the second using all the complete instructions. I also tried to manipulate the instructions, but that lowered the accuracy a lot.

3.2 Vectorizer

I used two vectorizer objects provided by Scikit-Learn. They allow us to perform all the above steps at once efficiently, and even apply preprocessing and rules regarding the number and frequency of tokens.

- **Count Vectorizer**: It counts the number of times a token shows up in the document and uses this value as its weight.
- **TF-IDF Vectorizer**: TF-IDF stands for "term frequency-inverse document frequency", meaning the weight assigned to each token not only depends on its frequency in a document but also how recurrent that term is in the entire corpora.

Since the order of the instructions is important, I also used the vectorizers with the *ngram_ range* parameter in order to look for ngram relationships at multiple scales, setting it once to (3,3) and once to (4,4)

3.3 How to choose the best classifier

A good model is not the one that gives accurate predictions on the known data or training data but the one which gives good predictions on the new data and avoids overfitting and underfitting.
In order to choose the classifier that performs better on the dataset, it's possible to analyze and compare some parameters:

- **Accuracy** - is the most intuitive performance measure and it is simply a ratio of correctly predicted observation to the total observations
- **Precision** - is the ratio of correctly predicted positive observations to the total predicted positive observations

- **Recall** - is the ratio of correctly predicted positive observations to the all observations in actual class - yes
- **F1 score** - is the weighted average of Precision and Recall ($F1Score = 2 * (Recall * Precision)/(Recall + Precision)$)
- **K Fold Cross Validation** - to tackle problem of overfitting it's possible to perform the *Cross Validation*. It consists of partitioning the data into a number of subsets, hold out a set at a time and train the model on remaining set and test model on hold out set, repeating the process for each subset of the dataset.

3.4 Classifier

Different classifiers were used:

- **Logistic Regression**
- **Multinomial Naive Bayes**
- **Decision Tree**
- **SVC** with *rbf* kernel
- **K-Neighbors**
- **Random Forest**

3.5 Result

Using the ***mnemonic instructions*** as feature, the *SVC* classifier performed better for both optimization and compiler classification, using the *Tfidf vectorizer with ngram range (3,3)*

- **Optimization classification**:

$$\boldsymbol{Confusion\ matrix} = \begin{bmatrix} 2019 & 421 \\ 337 & 3223 \end{bmatrix}$$
$$\boldsymbol{Accuracy} = 0.874$$
$$\boldsymbol{Precision} = 0.871$$
$$\boldsymbol{Recall} = 0.866$$
$$\boldsymbol{Cross\ validation} =$$
$$\begin{bmatrix} 0.822 & 0.85 & 0.825 & 0.827 & 0.825 & 0.857 & 0.858 & 0.845 & 0.84 & 0.822 \end{bmatrix}$$
$$\boldsymbol{F1\ score} = 0.868$$

- **Compiler classification**:

$$\boldsymbol{Confusion\ matrix} = \begin{bmatrix} 1773 & 112 & 76 \\ 114 & 1800 & 95 \\ 107 & 69 & 1854 \end{bmatrix}$$
$$\boldsymbol{Accuracy} = 0.905$$
$$\boldsymbol{Precision} = 0.904$$
$$\boldsymbol{Recall} = 0.904$$
$$\boldsymbol{Cross\ validation} =$$
$$\begin{bmatrix} 0.862 & 0.853 & 0.863 & 0.868 & 0.843 & 0.837 & 0.84 & 0.832 & 0.868 & 0.863 \end{bmatrix}$$
$$\boldsymbol{F1\ score} = 0.904$$

Using the ***complete instructions***, instead, the *SVC* classifier performed better with the *Tfidf* vectorizer without the *ngram_ range* for both optimization and compiler classification.

- **Optimization classification**:

$$\boldsymbol{Confusion\ matrix} = \begin{bmatrix} 1982 & 427 \\ 445 & 3146 \end{bmatrix}$$
$$\boldsymbol{Accuracy} = 0.855$$
$$\boldsymbol{Precision} = 0.849$$
$$\boldsymbol{Recall} = 0.849$$
$$\boldsymbol{Cross\ validation} =$$
$$\begin{bmatrix} 0.802 & 0.805 & 0.81 & 0.822 & 0.823 & 0.807 & 0.805 & 0.815 & 0.817 & 0.838 \end{bmatrix}$$
$$\boldsymbol{F1\ score} = 0.849$$

- **Compiler classification**:

$$\boldsymbol{Confusion\ matrix} = \begin{bmatrix} 1832 & 77 & 71 \\ 125 & 1822 & 65 \\ 82 & 66 & 1860 \end{bmatrix}$$
$$\boldsymbol{Accuracy} = 0.919$$
$$\boldsymbol{Precision} = 0.919$$
$$\boldsymbol{Recall} = 0.919$$
$$\boldsymbol{Cross\ validation} =$$
$$\begin{bmatrix} 0.872 & 0.857 & 0.863 & 0.842 & 0.857 & 0.87 & 0.873 & 0.86 & 0.86 & 0.906 \end{bmatrix}$$
$$\boldsymbol{F1\ score} = 0.919$$

Previous results are the best results obtained. For completeness, the results of all the cases tested are attached in the "*scores*" folder.

4 Predictions

From the tests carried out it can be seen that the best classifier is *SVC* with the *mnemonic instructions* and the *Tfidf* vectorizer with *ngram_range = (3,3)* for the optimization classification, while *SVC* with the *complete instructions* without *ngram_range* for the compiler classification. Since the difference between the use of mnemonic and complete instructions is not so great in the compiler classification, for simplicity the prediction on the *blind test set* has been made using the mnemonic instructions.
Results:

- **Optimization classification**: 1793 functions with ***L*** optimization, 1207 functions with ***H*** optimization
- **Compiler classification**: 975 functions compiled with ***gcc***, 1013 functions compiled with ***icc***, 1012 functions compiled with ***clang***

We can say that the compiler distribution is balanced, while the second is not (as in the train dataset).
A csv file is also attached, in which for each instruction the prediction of the compiler and the optimization is given.

5 Code

Two scripts have been implemented in Python 3. The first allows you to test different classifiers, vectorizers and features in order to obtain useful parameters to compare the various cases and choose the best algorithm. The second one allows to classify the blind test set instructions according to the compiler (Multi-class Classification) and the optimization (Binary Classification), using the desired classifier, vectorizer and feature.

5.1 Usage

- ***find_best_classifier.py***: python3 find_best_classifier.py classification_type instruction_type
- ***predict.py***: *python3 predict.py opt_instruction_type opt_classifier opt_vectorizer comp_instruction_type comp_classifier comp_vectorizer*

You need to specify some parameters:

- *classification_type*: "Optimization"/"Compiler", to choose whether to classify by optimization or by compiler

- *instruction_type*: "mnemonics"/"instructions", to choose whether to use only mnemonic or complete instructions
- *opt_instructions*: "mnemonics"/"instructions", to choose whether to use only mnemonic or complete instructions for the optimization classification
- *opt_classifier*: an integer between 0 and 5, to choose the classifier for the optimization classification
- *opt_vectorizer*: an integer between 0 and 5, to choose the vectorizer for the optimization classification
- *comp_instructions*: "mnemonics"/"instructions", to choose whether to use only mnemonic or complete instructions for the compiler classification
- *comp_classifier*: an integer between 0 and 5, to choose the classifier for the compiler classification
- *comp_vectorizer*: an integer between 0 and 5, to choose the vectorizer for the compiler classification

The possible classifiers are:

- *0* - Logistic Regression
- *1* - Multinomial NB
- *2* - Decision Tree
- *3* - SVC
- *4* - K-Neighbors
- *5* - Random Forest

The possible vectorizers are:

- *0* - Count vectorizer
- *1* - Tfidf vectorizer
- *2* - Count vectorizer with ngram_range=(3,3)
- *3* - Tfidf vectorizer with ngram_range=(3,3)
- *4* - Count vectorizer with ngram_range=(4,4)
- *5* - Tfidf vectorizer with ngram_range=(4,4)

5.2 Code explanation

Let's analyze some important functions.

Read the dataset:

```python
def read_data():
  print("Reading data ...")
  dataset = []
  with open(train_dataset_path, 'r') as json_file:
    json_list = list(json_file)

  for json_str in json_list:
    result = json.loads(json_str)
    instructions = result["instructions"]
    instr = []
    mnemonics = []
    for instruction in instructions:
      instr.append(instruction)
      mnemonics.append(instruction.split(" ")[0])
    result["instructions"] = ",".join(instr)
    result["mnemonics"] = ",".join(mnemonics)
  dataset.append(result)

  data = pd.DataFrame(dataset)
  print("Data read successfully!")
  print("-"*50)
  print(data.shape)
  print("-"*50)
  return data
```

This function allows you to read the dataset in jsonl format. Once opened, the instructions of each function of the dataset are manipulated in order to obtain two strings (one for the mnemonic instructions and one for the complete instructions. The whole is returned as a pandas dataframe.

Split the dataset:

```python
def split_dataset(classifier_type, instruction_type):
  target = ""
  if classifier_type == "Optimization":
    target = "opt"
  else:
    target = "compiler"
  x_train, x_test, y_train, y_test = train_test_split(
     data[instruction_type], data[target], test_size =
     0.20)
  encoder = preprocessing.LabelEncoder()
```

```
    y_train = encoder.fit_transform(y_train)
    y_test = encoder.fit_transform(y_test)
    return x_train, x_test, y_train, y_test
```

This function allows you to divide the dataset according to the type of classification required (Optimization or Compiler). 80% of the dataset is used for model training and 20% for testing. In addition, the target is encoded.

Classify:

```
def classify(x_train_count, x_test_count, y_train, y_test, vectorizer, instruction_type, classifier_type):
    for classifier in classifiers:
        print("="*50, file = score_file)
        print("Classifier:",type(classifier).__name__,"[Vectorizer: " + vectorizer +"], [Instruction type: " + instruction_type +"], [Classifier type: " + classifier_type + "]", file = score_file)
        print("="*50, file = score_file)
        print("Fitting",type(classifier).__name__,"[Vectorizer: " + vectorizer +"], [Instruction type: " + instruction_type +"], [Classifier type: " + classifier_type + "]")
        classifier.fit(x_train_count, y_train)
        print("-"*50)
        print("Predicting",type(classifier).__name__,"[Vectorizer: " + vectorizer +"], [Instruction type: " + instruction_type +"], [Classifier type: " + classifier_type + "]")
        y_pred = classifier.predict(x_test_count)
        print("-"*50)
        print("Result:", file = score_file)
        print("-"*50, file = score_file)
        print("Confusion Matrix:", file = score_file)
        print(confusion_matrix(y_test, y_pred), file = score_file)
        print("-"*50, file = score_file)
        print("Classification Report:", file = score_file)
        print(classification_report(y_test, y_pred), file = score_file)
        print("-"*50, file = score_file)
        print("Accuracy:", file = score_file)
        print(accuracy_score(y_test, y_pred), file = score_file)
        print("-"*50, file = score_file)
        print("Precision:", file = score_file)
        print(precision_score(y_test, y_pred, average="macro"), file = score_file)
```

```python
    print("-"*50, file = score_file)
    print("Recall:", file = score_file)
    print(recall_score(y_test, y_pred, average="macro"), file = score_file)
    print("-"*50, file = score_file)
    print("Performing Cross Validation Score",type(classifier).__name__,"[Vectorizer: " + vectorizer +"], [Instruction type: " + instruction_type +"], [Classifier type: " + classifier_type + "]")
    print("Cross Validation Score:", file = score_file)
    print(cross_val_score(classifier, x_test_count, y_test, cv = 10, n_jobs = -1), file = score_file)
    print("-"*50)
    print("-"*50, file = score_file)
    print("F1 Score:", file = score_file)
    print(f1_score(y_test, y_pred, average="macro"), file = score_file)
    print("-"*50, file = score_file)
```

This last function allows you to train and test the six models used for this experiment. After obtaining the various parameters for each model, these are written into a *file.txt*, so that they can be analyzed. The cross validation is performed using 10 subsets.

6 Conclusions

From the results obtained it is possible to conclude that *SVC* is the classifier that performs better to solve the compiler provenance problem, both in the case of the mnemonic instructions and in the case of the complete instructions. In addition, it can be noted that considering the complete instructions does not significantly improve the results, so it is advisable to always use the mnemonic instructions. Another consideration is that the order in which the instructions appear is important, and in general consider blocks of 3 instructions to be more successful than considering blocks of 4 instructions.
However, *SVC* is much slower. You can see that *Logistic Regression* has slightly lower results($\sim$0.86 of accuracy for optimization classification and $\sim$0.88 for compiler classification, the same for precision, recall and F1 score), but it is performed in a much shorter time (*SVC*) cannot be run in multi-threaded mode. Running the *predict.py* script using the *Logistic Regression*, we obtain the following results which are very similar to those obtained with *SVC*:

- **Optimization classification**: 1783 functions with ***L*** optimization, 1217 functions with ***H*** optimization

- **Compiler classification**: 972 functions compiled with ***gcc***, 1031 functions compiled with ***icc***, 997 functions compiled with ***clang***