Compiler Provenance

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

Costa Marco 1691388

8 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.

2.1 Read the dataset

```
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
```

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.

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, setting it once to (3,3) and once to (4,4)

3.3 Classifier

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

3.4 Result

4 Predictions

5 Conclusions