The compiler provenance problem

HW1 - Machine Learning - La Sapienza university of Rome

Manuel Ivagnes 1698903

25/10/19

1 Introduction

Given the statement *"A malware is a malicious software that fulfills the deliberately harmful intent of an attacker"* the idea behind the homework is to apply the machine learning techniques to a very first step of the malware analysis, the compiler provenance problem. Indeed, the aim of the malware analysis is to develop a deeper understanding about several aspects of malwares through the study of malicious samples. This can be done with the reverse engineering, the process by which the malware code is disassembled to reveal its binary code.

The relentless growth in complexity and volume of malware is getting more and more challenging deal with all these informations, here comes the reason behind the introduction of machine learning, which provides a natural choice to support the process of knowledge extraction. This technology in fact permits to build helpful tools to support the analysis process while working on huge amount of already known samples.

2 Problem definition and Input data

The problem given is a supervised learning task, which requires to build a *binary classifier* to predict the level of optimization (high / low) and a *multiclass classifier* to predict the compiler (gcc / icc / clang) used to generate the function.

The dataset contains 30000 functions and it is provided as jsonl file, with the following format

- *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)

The compiler distribution is totally balanced, the optimization distribution is not totally balanced, there are about 60% H and 40% L.

3 Data Preprocessing

The first step in the construction of the solution is the data preprocessing, a data mining technique that involves transforming raw data into an understandable format.

Through a brief analysis is possible to categorize the task as a *text classification problem*, but, given the fact that this is not a colloquial language text, but a *formal language* text (which consists of words whose letters are taken from an alphabet and are well-formed according to a specific set of rules), it did not require many initial changes.

The unique alteration that I made to the text is the remotion of the instructions operands. I took this decision after a couple of tests as a result of a reduction on timing and accuracy performances, using all the opersands. Moreover, too many features can overfit the training dataset and give bad performances on unseen data.

I also tried to reduce the huge amount of features generated, clustering similar operations, but I did not receive enough benefits to justify the computational power needed.

As first step I saved the samples in a python list, trying both to remove and maintain the duplicates . Removing the duplicates, the dataset size become of 29000 samples, and the theoretical performance metrics in general seem to decrease. This is probably a consequence of the splitting procedure on the given set, which is used both as training set and test set. In fact, some of the duplicates can be in both sets increasing the chance of a positive match.

Anyway, presuming that the dataset is composed by Structured Data, I decided to maintain them, because it will increase the precision on those repeated samples (in the training set) also in real scenario.

To conclude this step, I converted everything into a *pandas Dataframe*, which is a 2-dimensional labeled data structure with columns of potentially different types. At this point the rows represents the samples and the columns in the format seen before, in the next step the the instructions column will be substituted by the features columns.

4 Features Engineering Overview

As said before this problem can be seen as a text classification problem, which is a major application field for machine learning algorithms. However the raw data, a sequence of symbols cannot be fed directly to the algorithms themselves as most of them expect *numerical feature vectors* with a *fixed size* rather than the raw text documents with variable length.

In order to address this, *Scikit-learn* (the library used for the homework) provides utilities for the most common ways to extract numerical features from text content, which are:

- *tokenizing* strings and giving an integer id for each possible token, in this case by using white-spaces as token separators.
- *counting* the occurrences of tokens in each sample.
- *normalizing* and weighting with diminishing importance tokens that occur in the majority of samples.

A set of instructions sample can thus be represented by a matrix with one row per sample and one column per instruction occurring in the set. This process is called *vectorization* and the specific strategy (tokenization, counting and normalization) is called the *Bag of Words* representation.

Scikit-learn provides 3 types of vectorizers for this process:

- *CountVectorizer()* : Simply convert a collection of text documents to a matrix (with a sparse representation) of token counts.
- *TfidfVectorizer()* : Convert a collection of text documents to a matrix of TF-IDF features

- *HashingVectorizer()* : Similar to the first one but it implements the hashing trick to find the token string name to feature integer index mapping, useful for huge dataset. Unfortunately this reduce the performances so I decided to skip it, given the fact that the dataset has not so many entries.

The **Tf–idf term weighting** is used to to re-weight the count features into floating point values. This process is useful in the case of texts where some specific words will be very present hence carrying very little meaningful information about the actual contents of the document. Tf–idf means *term-frequency times inverse document-frequency*:

$$\text{tf-idf}(t,d) = tf(t,d) * idf(t) \tag{1}$$

where, with the default parameters, the idf component is computed as:

$$idf(t) = \log \frac{1+n}{1+df(t)} + 1 \tag{2}$$

where n is the total number of documents in the document set and *df(t)* is the number of documents in the document set that contain term *t*. The resulting tf-idf vectors are then normalized by the Euclidean norm:

$$v_{norm} = \frac{v}{||v||_2} = \frac{v}{\sqrt{{v_1}^2 + {v_2}^2 + \cdots + {v_n}^2}} \tag{3}$$

Unfortunately, the Bag of Words alone cannot capture the order of the instructions which is very important for these 2 tasks, as will be seen in the comparative analysis. To remedy, instead of building a simple collection of unigrams, it is possible to opt for a collection of **n-grams**, where occurrences of n consecutive words are counted.

Other interesting parameters used for the homework are **max_df** and **min_df**, which permit to ignore terms that have a document frequency strictly higher/lower than the given threshold.

The optimization of these features can change a lot the performance metrics as it will be exposed in the comparative analysis in the conclusion.

The fit_transform method is used to apply all these changes to the dataset.

5 The first Naïve Prediction

Before the introduction of the techniques used to find the best classifier, I want to expose how I started to approach the problem. As first attempt, I decided to simply split the dataset by the train_test_split function in 2 dataset, 2/3 for the training set and 1/3 for the test set.

For the unbalanced optimization problem I used the stratify parameter, which makes a split so that the proportion of values in the sample produced will be the same as the proportion of values provided to parameter stratify. For the balanced compiler problem, instead, I simply used the random_state parameter, often used with the value 42, which I found to be the "Answer to the Ultimate Question of Life, the Universe, and Everything".

As simple try, for the features engineering, I used the CountVectorizer() with no parameters.

As model, I used the **MultinomialNB()**, which implements the Naive Bayes algorithm for multinomially distributed data, and is a variant often used in text classification. The idea is based on the Bayes' theorem with the naive assumption of conditional independence between every pair of feature, giving:

$$y_{NB} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y) \tag{4}$$

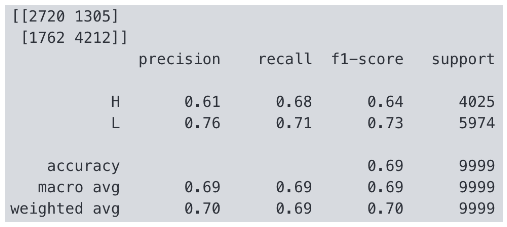
where each x_i is an attribute of the instance.

The distribution is parametrized by vectors $\theta_y = (\theta_{y1}, \ldots, \theta_{yn})$ for each class y, where n is the number of features (in text classification, the size of the vocabulary) and θ_{yi} is the probability $P(x_i \mid y)$ of feature appearing in a sample belonging to class y. The parameters θ_{yi} is estimated by:

$$\hat{\theta}_{yi} = \frac{N_{yi} + \alpha}{N_y + \alpha n} \tag{5}$$

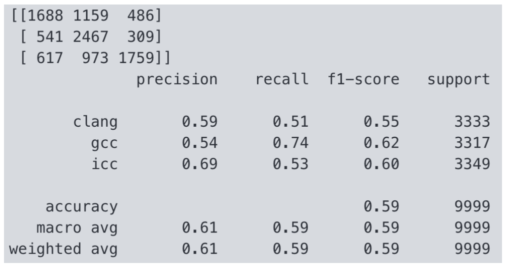
where $N_{yi} = \sum_{x \in T} x_i$ is the number of times feature i appears in a sample of class y in the training set T, and $N_y = \sum_{i=1}^{n} N_{yi}$ is the total count of all features for class y. The $\alpha > 0$ avoid that the value of y becomes equal to 0 in case of missing word in the text.

The code of this first experiment is in *naive-simple.py*, it gave me the following performance metrics:



```
[[2720 1305]
 [1762 4212]]
              precision    recall  f1-score   support

           H       0.61      0.68      0.64      4025
           L       0.76      0.71      0.73      5974

    accuracy                           0.69      9999
   macro avg       0.69      0.69      0.69      9999
weighted avg       0.70      0.69      0.70      9999
```

(a) MultinomialNB: optimization

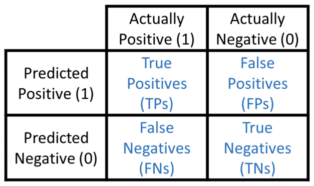


```
[[1688 1159  486]
 [ 541 2467  309]
 [ 617  973 1759]]
              precision    recall  f1-score   support

       clang       0.59      0.51      0.55      3333
         gcc       0.54      0.74      0.62      3317
         icc       0.69      0.53      0.60      3349

    accuracy                           0.59      9999
   macro avg       0.61      0.59      0.59      9999
weighted avg       0.61      0.59      0.59      9999
```

(b) MultinomialNB: compiler

6 Performance metrics Overview

The most intuitive and easiest metrics used for finding the correctness and accuracy of the model is the **Confusion matrix**.



	Actually Positive (1)	Actually Negative (0)
Predicted Positive (1)	True Positives (TPs)	False Positives (FPs)
Predicted Negative (0)	False Negatives (FNs)	True Negatives (TNs)

The matrix is divided in 4 ”blocks”

- *True Positive:* the actual class of the data point was Positive (1) and the predicted is also Positive (1). Ex. instructions with optimization H classified as optimization H
- *True Negative:* the actual class of the data point was Negative (0) and the predicted is also Negative (0). Ex. instructions with optimization L classified as optimization L
- *False Positive:* the actual class of the data point was Negative (0) and the predicted is Positive (1). Ex. instructions with optimization L classified as optimization H
- *False Negative:* the actual class of the data point was Positive (1) and the predicted is Negative (0). Ex. instructions with optimization H classified as optimization L

From these values is possible to derive the metrics to evaluate the model.

The **Accuracy** in classification problems is the number of correct predictions made by the model over all kinds predictions made.

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{6}$$

This is a good measure when the target variable classes in the dataset are nearly balanced, as in this case.

The **Precision** is the proportion of Positive predicted that are actually Positive

$$Precision = \frac{TP}{TP + FP} \tag{7}$$

The **Recall** is the proportion of actual Positive predicted as Positive

$$Recall = \frac{TP}{TP + FN} \tag{8}$$

These 2 parameters are used to minimize respectively the False Positive and False Negative.

The **F1-score** is the harmonic mean between Precision and Recall

$$\text{F1-score} = \frac{2 * Precision * Recall}{Precision + Recall} \tag{9}$$

7 K-cross validation

Now, with more informations about the performance metrics, it is possible to improve the evaluation, introducing a new technique, the K-cross validation.

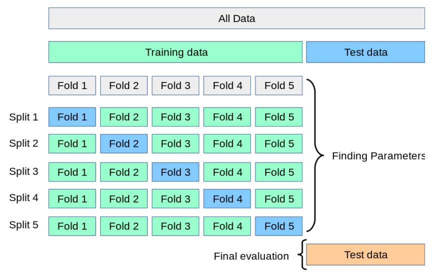
Indeed, partitioning the available data into 2 sets (train / test) gives some initial informations about the performance metrics but the results can depend on a particular random choice for the pair of (train, test) sets.

With the *K-fold cross validation*, the training set is split into k smaller sets, and for each subset is repeated the following procedure:

- A model is trained using $k - 1$ of the folds as training data
- The resulting model is validated on the remaining part of the data

The performance measure reported by this procedure is then the average of the values computed in the loop.

The fair result of this operation provides a better understanding of the model properties and can be used for a better comparison.



For the first comparison I used a model that I really appreciate, the **Decision Trees**, which aims to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. This process evaluation is simply given by a set of rules generated for each path to a leaf node, which correspond to a value of the target function.

For instance, the model can be generated by the *ID3 algorithm*.

```
ID3 (Examples, Target_Attribute, Attributes)
    Create a root node for the tree
    If all examples are positive:
        Return the single-node tree Root, with label = +
    If all examples are negative:
        Return the single-node tree Root, with label = -
    If number of predicting attributes is empty:
        then Return the single node tree Root
                with label = most common value of the target attribute in the examples
    Otherwise Begin:
        A ← The Attribute that best classifies examples
        Decision Tree attribute for Root = A
        For each possible value, $v_i$, of A:
            Add a new tree branch below Root, corresponding to the test A = $v_i$.
            Let Examples($v_i$) be the subset of examples that have the value $v_i$ for A
            If Examples($v_i$) is empty:
                Then below this new branch add a leaf node
                                with label = most common target value in the examples
            Else :
                below this new branch add the subtree
                                (Examples($v_i$), Target_Attribute, Attributes - {A})
    End
    Return Root
```

The selection of the best attributes is made by the concepts of entropy (measure of the amount of uncertainty in the dataset) and information gain (measure of the difference in entropy from before to after the set is split on an attribute A)

Given the fact that, for both tasks, the dataset is enough balanced, I used mostly the accuracy metrics for the comparison. The optimization task gave me the following results:

```
        MultinomialNB = [ 0.687   0.705   0.701   0.694   0.6955
0.702   0.713   0.7115   0.7055   0.7105 ]
        mean(MultinomialNB) =  70.257%
        DecisionTreeClassifier = [0.779   0.789   0.789   0.797
0.7815   0.7935   0.792   0.764   0.7845   0.797 ]
        mean(DecisionTreeClassifier) = mean:78.671%
```

The compiler task gave me the following results:

```
        MultinomialNB = [ 0.603   0.607   0.592   0.581  0.602
0.597   0.5935   0.6125   0.592   0.597 ]
        mean(MultinomialNB) =  59.777%
        DecisionTreeClassifier = [ 0.738   0.7305   0.75   0.726
0.7385   0.7455   0.7365   0.753   0.732   0.7455 ]
        mean(DecisionTreeClassifier) = 73.956%
```

Then, I concluded that, with this parameters and features, the DecisionTreeClassifier is a better option for both these tasks. The code containing these tests is in the *k-cross.py* file.

8 Grid Search

Until now, all the tests exposed were executed using the default models parameters, To increase the performance metrics is useful to set the *hyperparameters*, which are configuration variables internal to the model and whose values can be estimated from data. There exist more methods to find these values based on the idea of the k-cross validation. I used the Grid Search, which exhaustively generates candidates from a grid of parameter values specified. The optimization task gave me the following results:

```
MultinomialNB = {'alpha': 0.1,
        'class_prior': [0.1, 0.9], 'fit_prior': True}
mean(MultinomialNB) = 70.451%
DecisionTreeClassifier = {'max_depth': 11,
        'min_samples_split': 250}
mean(DecisionTreeClassifier) = mean:79.891%
```

The compiler task gave me the following results:

```
MultinomialNB = {'alpha': 0.1, 'fit_prior': True}
mean(MultinomialNB) =  59.972%
DecisionTreeClassifier = {'max_depth': 17,
        'min_samples_split': 10}
mean(DecisionTreeClassifier) = 75.016%
```

As expected most models gave in general better performances. Again, there is new code containing these tests in the *GridSearch.py* file.

9 Comparative Analysis and Conclusion

Following all these procedures, now it is possible to build a good classifier for these tasks. In reality I tried many different options and models but, to reduce the size of this document, I will add only one that I found to performe very well in reasonable time.

The **Linear Support-Vector Machine**, which constructs a hyperplane or set of hyperplanes in a high or infinite-dimensional space, used for the classification (or, in case, regression). This model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on the side of the gap on which they fall.

Up to this point I introduced changes in the features to improve the performance metrics. For the first test I used the CountVectorizer with parameters (ngram_range = (2,3), max_df = 5, min_df = 1.0):

- Optimization problem:
  - *Test1.1* : Multinomial Naive Bayes
    - parameters: {alpha: 0.1, class_prior: 0.1, 0.9, fit_prior: True}
    - average K-fold accuracy: 79.036%
  - *Test1.2* : Linear Support-Vector Machine
    - parameters: {C = 0.1}
    - average K-fold accuracy: 85.566%
- Compiler problem:
  - *Test2.1* : Multinomial Naive Bayes
    - parameters: {alpha: 0.1, fit_prior: True}
    - average K-fold accuracy: 84.126%
  - *Test2.2* : Linear Support-Vector Machine
    - parameters: {C = 0.1}
    - average K-fold accuracy: 88.416%

For the second test I used the TfidfVectorizer with the same parameters:

- Optimization problem:
  - *Test1.1* : Multinomial Naive Bayes
    - parameters: {alpha: 0.1, class_prior: 0.1, 0.9, fit_prior: True}
    - average K-fold accuracy: 75.936%

  - *Test1.2* : Linear Support-Vector Machine
    * parameters: {C = 0.1}
    * average K-fold accuracy: 84.816%

- Compiler problem:
  - *Test2.1* : Multinomial Naive Bayes
    * parameters: {alpha: 0.1, fit_prior: True}
    * average K-fold accuracy: 86.531%
  - *Test2.2* : Linear Support-Vector Machine
    * parameters: {C = 0.1}
    * average K-fold accuracy: 86.736%

Note: The figures are in the next page.

The introduction of the n-grams gave a considerable boost to the performance metrics, giving a meaning to the order of the instructions.

By these test is possible to notice that in general the SVC, as more complex model, performs better for all the task, except for the compiler problem using the TfidfVectorize. In this case the accuracy is very similar, even if looking at the confusion matrix, precision and recall, the SVC is more uniform. The Multinomial Naive Bayes has some problems the gcc compiler.

For both cases the best classifier is the SVC with the CountVectorizer.

The code used for these tests is in *final.py*.

References

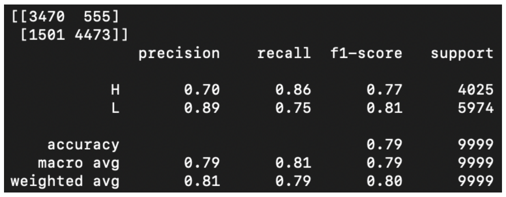
[1] Course slides

[2] Machine Learning, Tom Mitchell, McGraw Hill, 1997.

[3] Pattern Recognition and Machine Learning, Christopher M. Bishop, 2006

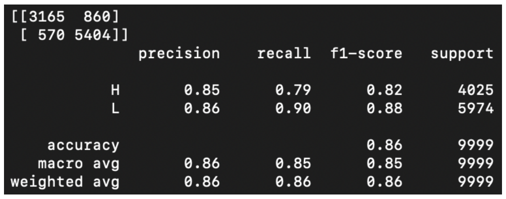
[4] `https://scikit-learn.org/stable/index.htmll`

[5] `https://www.ritchieng.com/machine-learning-resources/`



```
[[3470  555]
 [1501 4473]]
              precision    recall  f1-score   support

           H       0.70      0.86      0.77      4025
           L       0.89      0.75      0.81      5974

    accuracy                           0.79      9999
   macro avg       0.79      0.81      0.79      9999
weighted avg       0.81      0.79      0.80      9999
```

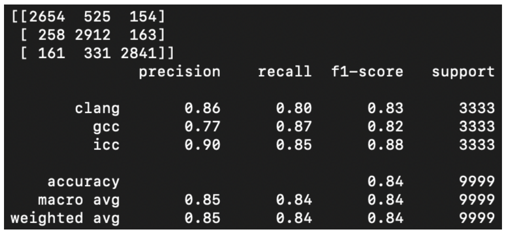
(a) Multinomial Naive Bayes



```
[[3165  860]
 [ 570 5404]]
              precision    recall  f1-score   support

           H       0.85      0.79      0.82      4025
           L       0.86      0.90      0.88      5974

    accuracy                           0.86      9999
   macro avg       0.86      0.85      0.85      9999
weighted avg       0.86      0.86      0.86      9999
```

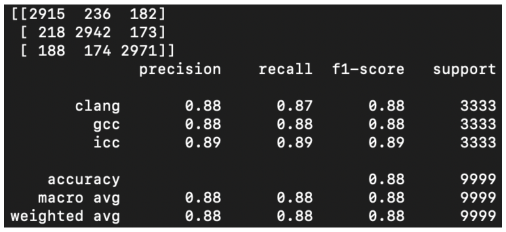
(b) Support-Vector Machine

Figure 2: Optimization tests (TfidfVectorizer)



```
[[2654  525  154]
 [ 258 2912  163]
 [ 161  331 2841]]
              precision    recall  f1-score   support

       clang       0.86      0.80      0.83      3333
         gcc       0.77      0.87      0.82      3333
         icc       0.90      0.85      0.88      3333

    accuracy                           0.84      9999
   macro avg       0.85      0.84      0.84      9999
weighted avg       0.85      0.84      0.84      9999
```

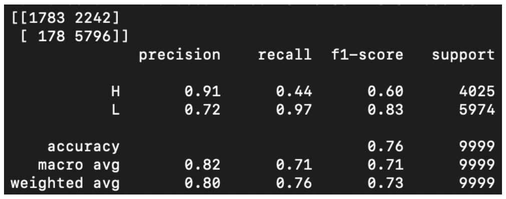
(a) Multinomial Naive Bayes



```
[[2915  236  182]
 [ 218 2942  173]
 [ 188  174 2971]]
              precision    recall  f1-score   support

       clang       0.88      0.87      0.88      3333
         gcc       0.88      0.88      0.88      3333
         icc       0.89      0.89      0.89      3333

    accuracy                           0.88      9999
   macro avg       0.88      0.88      0.88      9999
weighted avg       0.88      0.88      0.88      9999
```

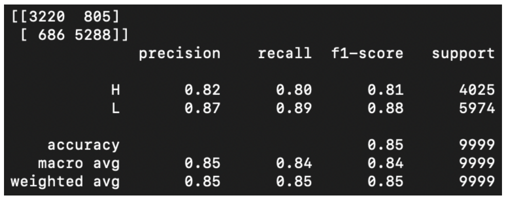
(b) Support-Vector Machine

Figure 3: Compiler tests (TfidfVectorizer)



```
[[1783 2242]
 [ 178 5796]]
              precision    recall  f1-score   support

           H       0.91      0.44      0.60      4025
           L       0.72      0.97      0.83      5974

    accuracy                           0.76      9999
   macro avg       0.82      0.71      0.71      9999
weighted avg       0.80      0.76      0.73      9999
```

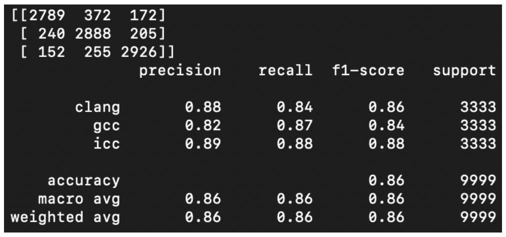
(a) Multinomial Naive Bayes



```
[[3220  805]
 [ 686 5288]]
              precision    recall  f1-score   support

           H       0.82      0.80      0.81      4025
           L       0.87      0.89      0.88      5974

    accuracy                           0.85      9999
   macro avg       0.85      0.84      0.84      9999
weighted avg       0.85      0.85      0.85      9999
```

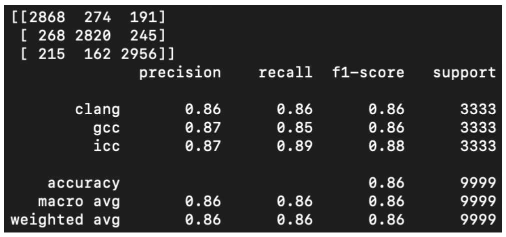
(b) Support-Vector Machine

Figure 4: Optimization tests (CountVectorizer)



```
[[2789  372  172]
 [ 240 2888  205]
 [ 152  255 2926]]
              precision    recall  f1-score   support

       clang       0.88      0.84      0.86      3333
         gcc       0.82      0.87      0.84      3333
         icc       0.89      0.88      0.88      3333

    accuracy                           0.86      9999
   macro avg       0.86      0.86      0.86      9999
weighted avg       0.86      0.86      0.86      9999
```

(a) Multinomial Naive Bayes



```
[[2868  274  191]
 [ 268 2820  245]
 [ 215  162 2956]]
              precision    recall  f1-score   support

       clang       0.86      0.86      0.86      3333
         gcc       0.87      0.85      0.86      3333
         icc       0.87      0.89      0.88      3333

    accuracy                           0.86      9999
   macro avg       0.86      0.86      0.86      9999
weighted avg       0.86      0.86      0.86      9999
```

(b) Support-Vector Machine

Figure 5: Compiler tests (CountVectorizer)