Machine Learning: Malware analysis

Alessio Devoto

November 5, 2019

Contents

1 Introduction

The purpose of this paper is to evaluate the performances of a set of classification techniques over a dataset and compare the results obtained. The dataset consists in a series of disassembled softwares, each one with its **level of optimization**, which can either be H or L, and its **compiler**, which can be gcc, icc or Clang. Provided this dataset, we need to devise two learning methods. Firstly, we need a binary classifier that is able to determine, given a new software, whether it is optimized with High or Low optimization option. Second, we need a multiclass classifier which is able to determine, given a new software, whether it was compiled with gcc, icc or Clang . These classifiers have to be as accurate as possible, thus several parameters have been taken into account during the assessment of each learning process: method of preprocessing data, method of vectorizing data, type of classifier used and hyperparameters. Much of the work was therefore tuning all these parameters to get the highest possible accuracy. Each task has been divided into four steps:

- Data preprocessing
- Data vectorization
- Learning
- Evaluation

In the first step, we transform the dataset in such a way that it can be used as input for a Vectorizer. In the following step, we transform our categorical data into numerical. This is necessary because a classifier has to be trained on numerical data. This phase constitutes the so called 'Feature extraction'. During the Learning step, we train different classifiers, each one with different parametric settings. Finally, in the evaluation step, we run some tests and decide which algorithm is more effective.

Because we have two tasks to complete, we split this paper into two sections, one per Task. Each section is structured according to the pattern presented in the previous list.

2 Dataset analysis

The dataset is provided in a *train_dataset.jsonl* file. As mentioned above, the file contains a series of softwares in assembly language along with their optimization level and their compiler. In particular, each software is composed by:

- a list of strings, that represent instructions
- an optimization level **opt** : either H or L
- a compiler **compiler** : gcc, Clang or icc

Listing 1: group by compiler

```
                    instructions  opt  compiler
count               30000         30000     30000
unique              29692             2         3
freq                 9            17924     10000
```

Listing 2: group by opt

```
             count  unique
opt
H            12076   11919
L            17924   17779
```

Listing 3: group by compiler

```
                  count  unique
compiler
clang             10000    9933
gcc               10000    9864
icc               10000    9897
```

Just so we have some notion of the structure of the dataset, we can analyze it with some Pandas basic commands, such as `data.describe()` (output shown in Listing 1) We gather that we have 30.000 different elements, 29692 of which are unique. Even if this kind of information may turn out to be unhelpful during our work, it is still useful in order to have a better understanding of the matter. We can provide deeper insights into the dataset structure, by adding a column that counts the length of each instruction and grouping our data based either on the 'opt' value or on the 'compiler' value. As we can see in Listing 2, for instance, softwares optimized with L option are more than those optimized with H option.

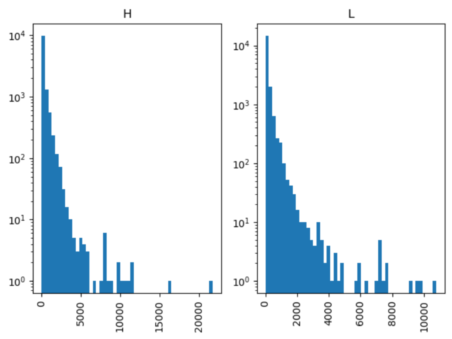
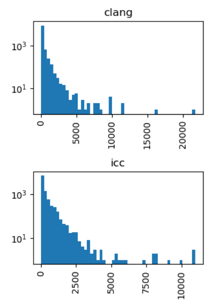
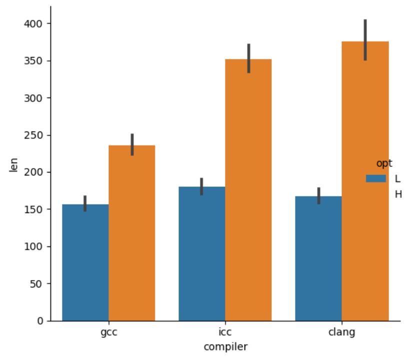


Figure 1: Grouped by opt.



(a) *grouped by compiler 1*



(b) *grouped by compiler 2*

Figure 2: Grouped by compiler

Running the same test considering the 'compiler' option, we see the average length of software compiled by each compiler (Figure 2).

3 Task 1: Binary classification

3.1 Preprocessing data

After having imported the dataset, we have to prepare it to be vectorized. This means we have to transform each list of instructions into a string, that can be used as input for the Vectorizer. To this end, we devised three possible options:

1. Method 1: transforming each list of instructions into a comma separated string of instructions, thus copying in the new string the registers used for each instruction.

   ```
   0 push r12, push rbp ,push rbx, test byte [rdi ...
   ```

2. Method 2: transforming each list of instructions into a comma separated string of mnemonics, thus ignoring the registers involved in the instruction.

   ```
   0 push, push, push,test, je, mov, mov, mov, call, mov ...
   ```

3. Method 3: transforming each list of instructions into a comma separated string of 'short mnemonics', considering just the first 3 letters of each mnemonic, in the attempt to eliminate not only the register, but also the suffix of the mnemonics. The idea behind this transformation is that we wish to abstract as much as possible from the particular instruction and consequently reduce the dimension of the vocabulary created by the Vectorizer.

   ```
   0 pus, pus, pus, tes, je, mov, mov, mov, cal, mov ...
   ```

3.2 Vectorizing data

Once data have been preprocessed, we can use a Vectorizer to convert them from categorical into numerical. Hence we need a way to split the lines of instructions into tokens by following some sort of pattern, and then to assign a weight to each token. To do so, we have two options, provided by SKlearn library: *CountVectorizer* and *TfidfVectorizer*. The former is the most straightforward: it counts the number of times a token shows up in the document and uses this value as its weight. The latter is more sophisticated: the weight assigned to each token not only depends on its frequency in a document but also on how recurrent that term is in the entire dataset. Furthermore, vectorizers come equipped with different features that allow us to be even more precise when splitting our strings into tokens. These features can all be set through arguments we pass to the vectorizer. In particular, we used two parameters: `tokenizer` and `ngram_range`. With `tokenizer` we provide the vectorizer with a function that splits a list into tokens. Provided we arranged our dataset so as to have values separated by commas, we use a tokenizer that returns the values in between two commas. This way, all values between two commas are considered by the Vectorizer to be different tokens. On the other hand with `ngram_range` we customize the fashion in which the vectorizer collects token. More precisely, the `ngram_range` parameter instructs the vectorizer on how many tokens it should consider at once. In other words, the wider the `ngram_range` is, the better we take into account the order in which words show up. For this first task, we tested four possible combinations:

1. CountVectorizer with `ngram_range = (1,1)`
2. CountVectorizer with `ngram_range = (1,2)`
3. TfidfVectorizer with `ngram_range = (1,1)`
4. TfidfVectorizer with `ngram_range = (1,2)`

Amongst the previous options, the vectorizers with a wider `ngram_range` performed on average better. This is likely due to the fact that the order in which instructions are arranged is affected by the level of optimization, as we see in the *evaluation* section.

3.3 Learning algorithms

For this first task we decided to implement and compare the following learning algorithms, tuning the parameters as shown below:

Logistic Regression
In this algorithm, the probabilities describing the possible outcomes of a single trial are modelled using a logistic function. It assumes all predictors are independent of each other, and assumes data is free of missing values. We modified the parameter **max_iter** with values 50, 100.

Multinomial Naive Bayes
Naive Bayes algorithm based is on Bayes' theorem, with the assumption of independence between every pair of features. We edited the parameter **fit_prior**, with values True, False. It states whether to learn class prior probabilities or not. If false, a uniform prior is used.

Linear SVC
Support vector machine exploits a representation of the training data as points in space separated into categories 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 which side of the gap they fall. We used 100,200,400 as values for **max_iter** parameter.

Perceptron

It is a type of linear classifier that makes its predictions based on a linear predictor function combining a set of weights with the feature vector.We edited the **max_iter** parameter with values 100, 200.

K-Neighbors

Neighbours based classification is a type of lazy learning as it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a majority vote of the k nearest neighbors of each point. In this case, the parameter we changed is **n_neighbors**, which states the number of neighbors to use. We assigned the values 2,3,5,8,10,12. We underline that the computational cost of this algorithms is higher if the number of neighbors is lower.

3.4 Evaluation

In order to attain the best possible evaluation when assessing our algorithms, we split the train set into two sets, thanks to the function:

```
x_train, x_test, y_train, y_test =
train_test_split(x_all, y_all, test_size=0.2, random_state=15)
```

Then evaluation part was divided into two phases. In the **first stage** we used the *GridsearchCV*, relying on the cross validation algorithm to test our functions, as shown below:

```
grid = GridSearchCV(
            classifier
            parameters,
            refit=True,
            n_jobs=-1,
            scoring='accuracy',
            cv=8,
        )
```

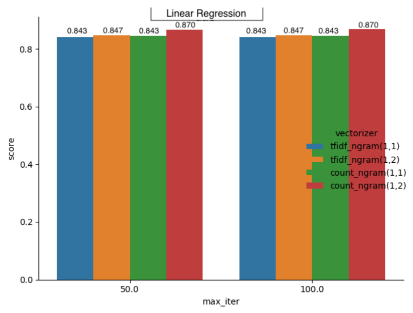
We set 'accuracy' as our scoring parameter to maximize. As a **second additional stage**, we tested algorithms which showed high performances in the first step, by the means of other metrics and tools, such as *f1-score*, *recall* and *confusion matrix.*

3.5 Results and comparative analysis

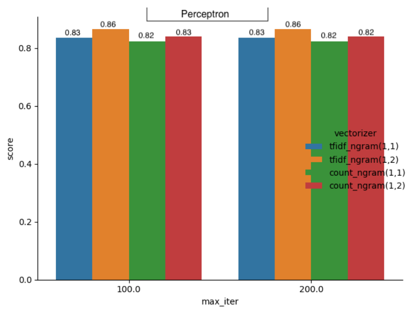
In this section we show the results and highlight some significative points.

3.5.1 First method to transform dataset: consider whole instruction

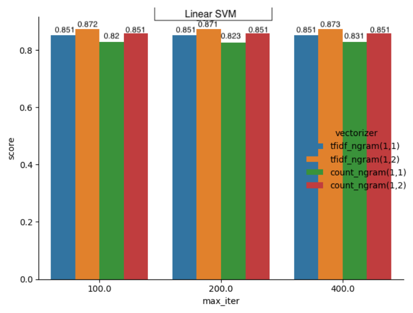
In Figure 3 we see a comparative plot that illustrates different performances for our learning algorithms when we use the first method to transform our dataset. Unsurprisingly for this first case there is very little difference between



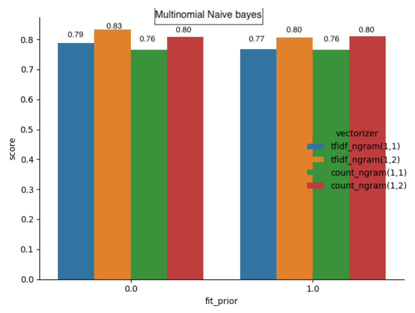
(a) *Logistic Regression*



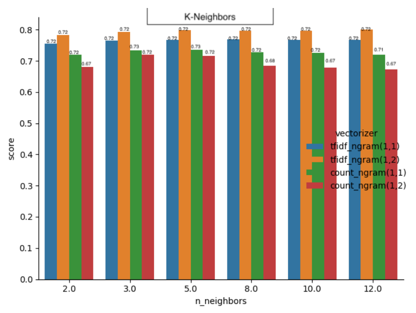
(b) *Perceptron*



(c) *Linear SVM*



(d) *Multinomial Naive Bayes*

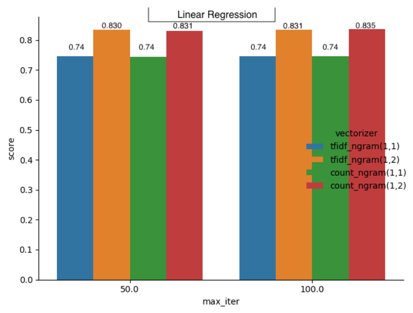


(e) *K-Neighbors*

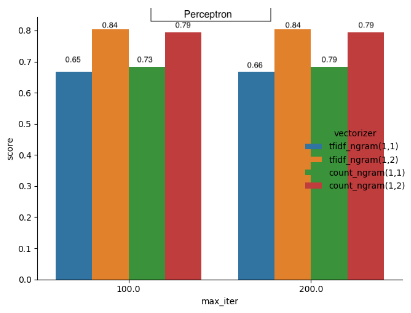
Figure 3: Method 1

the score we get when we use CountVectorizer with `ngram_range = (1,1)` and TfidfVectorizer with `ngram_range = (1,1)`. The gap becomes larger when we switch to `ngram_range = (1,2)`. In general, most of the learning algorithms seem to perform better with the TfidfVectorizer and `ngram_range = (1,2)`. This is a consequence of what has been said previously with regard to the the relation between order of instructions and optimization level.

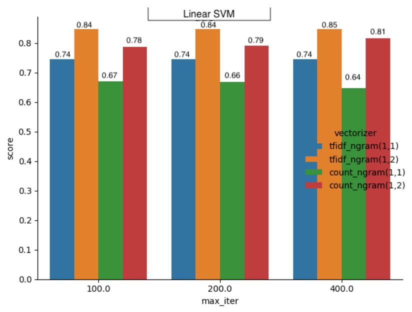
We observe that Multinomial Naive Bayes has the worst overall performances. Other points can be made concerning the performance of K-Neighbors



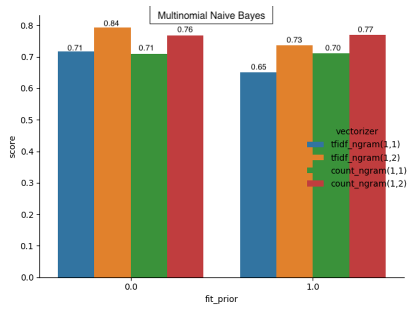
(a) *Logistic Regression*



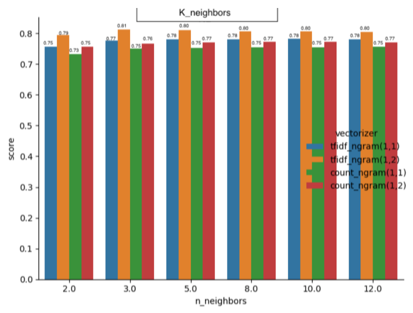
(b) *Perceptron*



(c) *Linear SVM*



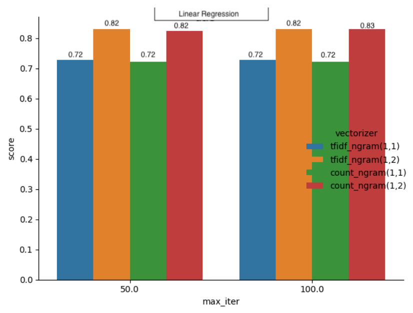
(d) *Multinomial Naive Bayes*



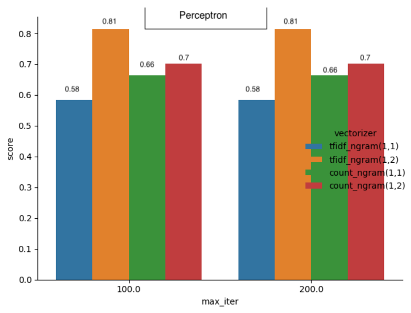
(e) *K-Neighbors*

Figure 4: Method 2

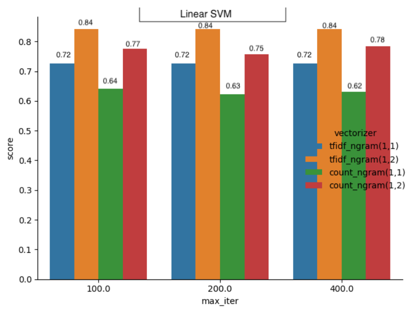
classifier. This algorithm indeed shows increasingly worse scores when we increment the number of neighbors. The two best performing algorithm are LSVC and Logistic Regression, which were able to get a 87% accuracy. It is worth mentioning though that when it comes to computational costs, the LSVC is by far better than Logistic Regression, especially in the case of CountVectorizer, the output of which is more complex to deal with. To have a better notion of this fact, it should be sufficient to look at the average fit time, as measured by our grid search algorithm: 351.97 seconds for Logistic Regression with `max_iter`



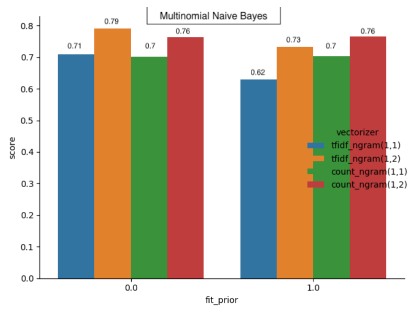
(a) *Logistic Regression*



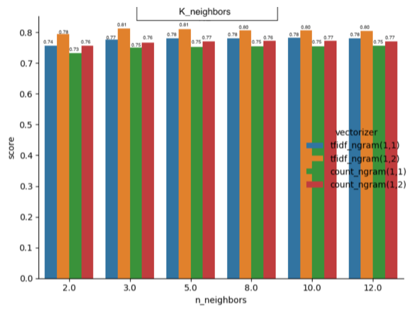
(b) *Perceptron*



(c) *Linear SVM*



(d) *Multinomial Naive Bayes*



(e) *K-Neighbors*

Figure 5: Method 3

= 50 versus 17.967 seconds for LinearSVC.

3.5.2 Second and third method to transform dataset

We remind that in the second method we split each instruction and consider only mnemonic,whereas in the third we split the instruction and remove not only the register but also the suffix. The performances presented for this test are fairly disappointing. As a general rule, all algorithms have performed better when classifying with the registers. When considering only the mnemonics, the gap between `ngram_range = (1,1)` and `ngram_range = (1,2)` grows definitely larger. In Figures 4 and 5 we can see that the former option achieves better scores, on average 0.5 points higher. In addition we witness the increasing of the gap between TfidVectorizer and CountVectorizer as well, at the expense of the latter. This is likely due to the fact that when considering only the suffix, the weighing and comparison to all the other elements in the corpus becomes even more meaningful.

It is interesting to notice that Multinomial Naive Bayes gets higher scores when we let it estimate the prior probabilities, with parameter `fit_prior`. This is a consequence of the structure of our dataset: in this case, namely the case in which our target is the optimization level, we have an unbalanced distribution of the target classes, unlike the second case, where the number of instance per each compiler is exactly the same, as we have seen in Figure 1.

3.5.3 Conclusion

The previous test shed light on several aspects of the problem. However, the best performances were attained by LinearSVC classifier, and this is the classifier we are going to test again and use for our blind test set. Below are the results we get when we test our LinearSVM on the whole dataset, without cross validation and only splitting data into train and test set.

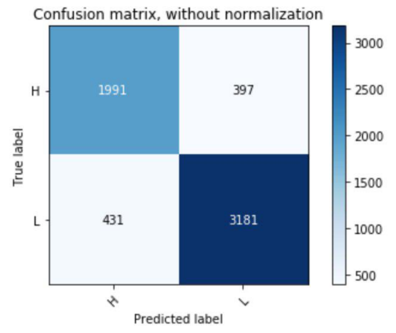


Figure 6: Confusion matrix for SVM.

```
              precision    recall  f1-score   support
           H       0.83      0.84      0.83      2388
           L       0.89      0.89      0.89      3612
    accuracy                           0.87      6000
   macro avg       0.86      0.86      0.86      6000
weighted avg       0.87      0.87      0.87      6000
```

As we can see, the results are consistent with the ones yielded by the cross validation algorithm.

4 Task 2: Multiclass classification

4.1 Preprocessing data

As in the previous case, we had to preprocess our data prior to feeding it to a Vectorizer. In this circumstance we have been able to exploit some knowledge about the dataset and the target classes. As we have discussed above, our task is that of identifying the compiler provenance of each malware. We well know that each of the three compilers usually introduces a number of instructions at the beginning and at the end of the code. Therefore we attempted to train our learning algorithms on a reduced dataset, that comprises only the ten first and ten last instructions of each list. In addition to this, we also used the same transformation as before in which we transform each instruction, namely mnemonic, suffix and register, into a token.

4.2 Vectorizing data

Like we did for the binary classification, we used both *CountVectorizer* and *TfidfVectorizer*. Given the particular feature we are trying to focus on, i.e. the very first and very last instructions of each line, we deemed that in this case the order of the words is even more meaningful to our classifiers. This is why we used two additional settings for our vectorizers, increasing the value of `ngram_range` parameter. Below is the list of all the combinations we used:

1. CountVectorizer with `ngram_range = (1,1)`
2. CountVectorizer with `ngram_range = (1,2)`
3. CountVectorizer with `ngram_range = (2,3)`
4. TfidfVectorizer with `ngram_range = (1,1)`
5. TfidfVectorizer with `ngram_range = (1,2)`
6. TfidfVectorizer with `ngram_range = (2,3)`

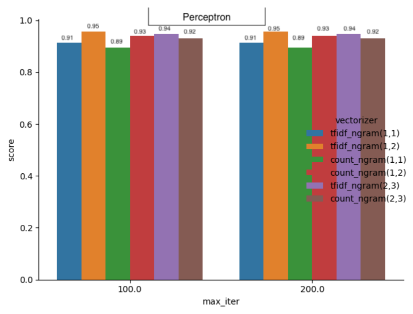
4.3 Learning algorithms

Following are the learning algorithms we implemented for the binary classification. Their main features have been described above. The parameters we edited are exactly the same as the first case

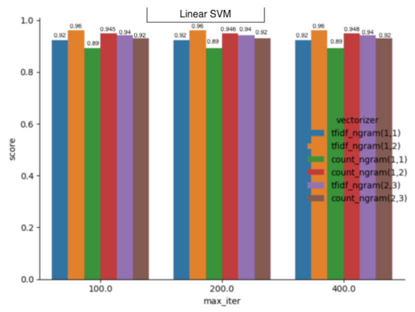
Multinomial Naive Bayes
fit_prior, with values True, False

Linear SVC
max_iter, with values 100,200,400

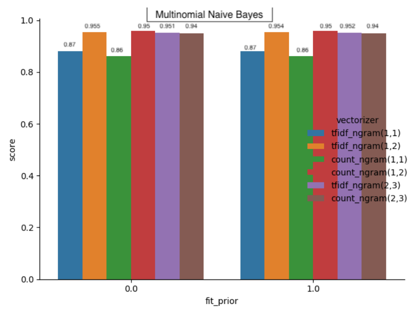
Perceptron
maxn_iter, with values 100, 200



(a) *Perceptron*



(b) *Linear SVM*



(c) *Multinomial Naive Bayes*

Figure 7: Method 1

4.4 Evaluation

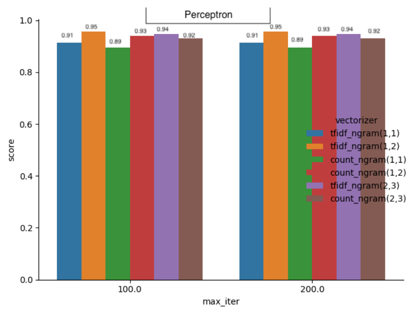
The evaluation stage was conducted with the same methods implemented for binary classification.

4.5 Results and comparative analysis

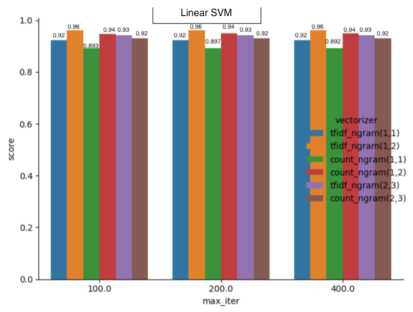
Like we have done before, in this section we present the results and focus on some points.

4.5.1 First and second method to transform dataset

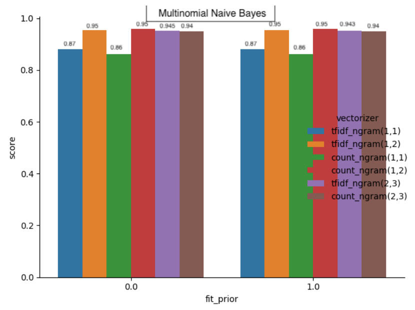
It seems, looking at Figures 7 and 8, that the different way of transoforming our dataset did not substantially affect our results. Our scores are quite similar, except for some very slight differences in the Naive Bayes. Nevertheless this means that our intuition of only considering the first and last instructions of a software was not wrong, since it yielded the same results as the other method. On the other hand, we unexpectedly see that the best option for Vectorizer is TfidfVectorizer with `ngram_range = (1,2)`. Although we were expecting the `ngram_range = (2,3)` to be more precise, we infer that the peculiar instructions



(a) *Perceptron*



(b) *Linear SVM*



(c) *Multinomial Naive Bayes*

Figure 8: Method 2

introduced by each compiler at the beginning and end of line- which is the feature our classifiers are counting on - is a smaller number than we thought.

4.5.2 Conclusion

In this second task all algorithms are able to get high scores, around 94% accuracy. Though the best is again Linear SVM, which is able to get a 96% accuracy with `max_iter = 100`. Evaluating the Linear SVM without cross validation, we get the following results:

```
                  precision    recall  f1-score   support
       clang         0.96        0.96     0.96      2009
         gcc         0.96        0.96     0.96      1994
         icc         0.97        0.97     0.97      1997
    accuracy                              0.97      6000
   macro avg         0.97        0.97     0.97      6000
weighted avg         0.97        0.97     0.97      6000
```

So we see that also in this case the results of the second test are consistent with those of the cross validation. Moreover, we observe that both precision (proportion of positive identifications actually correct) and recall (proportion of actual positives identified correctly) show high scores for all of the three classes.

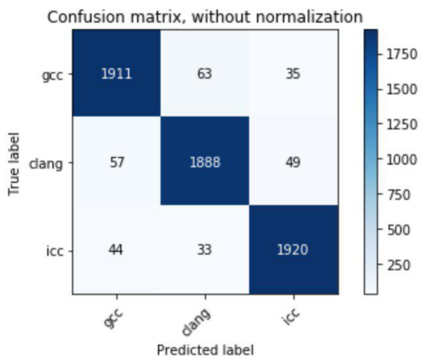


Figure 9: Confusion matrix for SVM.

5 Implementation and code

We decided to exploit Python's powerful libraries. We made extensive use of SKlearn, Pandas and Matplotlib libraries. For the sake of brevity, we just present a list of Python files we wrote and their role in our test. Of course the code itself is commented.

- data_analysis.py : contains some functions to 'explore' the dataset
- task1.py : contains the grid search cross validation for task 1
- task2.py : contains the grid search cross validation for task 2
- aux_functions.py : contains the functions to transform dataset (transform list of instructions into strings) and to tokenize values (tokenizer for vectorizer)
- final_test.py: contains the final evaluation test on the two chosen learning algorithms for task 1 and 2
- blind_test.py: contains the code responsible for reading the blind test set and outputting a new csv file

References

[1] https://kavita-ganesan.com/tfidftransformer-tfidfvectorizer-usage-differences/#.XcG37S2h3yI

[2] https://seaborn.pydata.org/tutorial/categorical.html

[3] https://towardsdatascience.com/hacking-scikit-learns-vectorizers-9ef26a7170af

[4] https://seaborn.pydata.org/tutorial/categorical.html

[5] https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html