Homework 1 - Bug finding classification

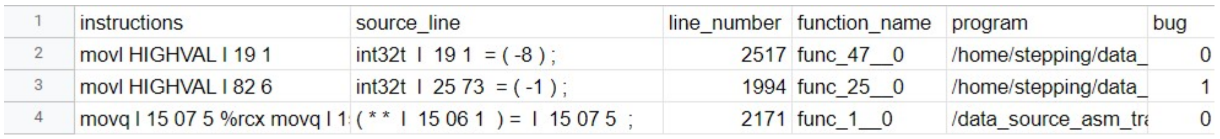
Alessandro Garbetta, 1785139

March 26, 2024

1 Introduction

This homework deals with the development and implentation of a classifier for detecting bugs within preprocessed lines of code. It is a binary classification problem; at the end of the work, we will tell if a line of code contains a bug or not.
When you run a source file written in a high-level language (e.g. C or similar), the files are processed to produce code in machine language. The translation of the file into this language is done by programs called compilers, which automatically translate the source files. Compilers are software that are written by programmers, who despite being experts, are not exempt from making mistakes (e.g. in the gcc compiler have been found more than 3000 bugs).

In our Homework we will work on pre-processed code instructions from assembly to source code (one step of compilation).
We were provided with a dataset ”mapping_traces_00.csv”, a pandas dataframe, i.e. a tab-separated csv file that contains 100 thousand samples. The dataset contains various information: the assembly language instructions, the related source lines, the number of the code line, the name of the function where the instruction is located, the name of the program and finally a binary value 1 0 that indicates whether there is a bug or not (0 if the mapping is correct, 1 if the mapping is not correct).



1	instructions	source_line	line_number	function_name	program	bug
2	movl HIGHVAL l 19 1	int32t l 19 1 = (-8) ;	2517	func_47__0	/home/stepping/data_	0
3	movl HIGHVAL l 82 6	int32t l 25 73 = (-1) ;	1994	func_25__0	/home/stepping/data_	1
4	movq l 15 07 5 %rcx movq l 1	(* * l 15 06 1) = l 15 07 5 ;	2171	func_1__0	/data_source_asm_tr	0

Figure 1: Example of dataset

To reach the goal of our classifier, we will use as information: the instructions, the source lines and of course the presence of bugs, which will be the label of our classifier. The provided dataset is ready to use, it has already been processed, there are no duplicates and it is perfectly balanced. The various memory accesses have been made homogeneous, and the immediate operands have been converted to base 10.
After the dataset loading and manipulation phase, a model was created to train the classifier to find bugs in the various lines of code. Various models were tested, observing the results with the help of evaluation techniques such as confusion matrix.
Finally, once an acceptable model was obtained, we worked on a data set of 1000 samples, the ”blind_test.csv”, also provided to predict the presence or absence of bugs. It

contains the same information as the first, with the difference that this time there is no information about the presence or absence of the bug. The results of the prediction are written to a .txt file respecting the order of the data set.

2 Load of dataset

First we loaded the dataset, using the pandas library, we read the csv file, and we split the various information using the tab separator. Only the first two columns and the last were extracted: "instruction", "source_line" and "bug".
The first operation that was done on the information just obtained was to sum the first two columns, and use it as a single character sequence.

3 Vectorization

Before being able to split the data set, an operation of vectorization was done on the data. The vectorization is an operation made essential whenever numeric characters are not used, but as in our case, strings. Machine Learning algorithms (explain better) are not able to process strings. It is necessary to use techniques that allow the training of the model. The vectorization operation creates a sparse matrix which associates to each element of the string the number of times the element repeats.
We have mainly considered 3 types of vectorization present in the sklearn.feature_extraction.text library, and they are: the HashingVectorizer, the CountVectorizer and the TfidfVectorizer.

- HashingVectorizer – it transforms a collection of text documents into a scipy.sparse matrix containing counts of token occurrences (or binary occurrence information), possibly normalized as token frequencies if norm = 'l1' or projected onto the Euclidean unit sphere if norm = 'l2'. This implementation of the vectorizer uses the hashing trick to find the string name of the token for integer index mapping. It is convenient because it uses little memory and is scalable for large data sets but there is no way to compute its inverse and there can be the problem of collisions.

- CountVectorizer – it provides a simple way to both tokenize a collection of text documents (our case list of strings) and build a vocabulary of known words, but also to encode new documents using that vocabulary.

- TfidfVectorizer – it's similar to CountVectorizer but instead of count words it calculate word frequencies. TF-IDF are word frequency scores that try to highlight words that are more interesting, e.g. frequent in a document but not across documents. The TfidfVectorizer will tokenize documents, learn the vocabulary and inverse document frequency weightings, and allow you to encode new documents.

Not one of the vectorization techniques was favored, but they were tried with the various models. It must be pointed out that while the size of the output of the CountVectorizer and the TfidfVectorizer are similar, that of the HashingVectorizer is much larger, which with some models greatly increases the fitting execution time. Comparisons between the various models and vectorizations will be made later.

4 Split of Dataset

One mistake that can be made is to not divide the dataset with the input samples into two subsets, the training dataset and the testing dataset. In fact, the testing of the model cannot be done on the same elements that have been used for the training. There are standard ratios for dividing the dataset, usually it is recommended to use 2/3 of the dataset for the training set, and the remaining 1/3 for the test set. In order to divide the dataset it has been used the function train_test_split of the library sklearn.model_selection. Among the required parameters there is the test_size, that is in percentage the size of the future test set, and then, for complementarity the train set. It was not decided for a static division of 1/3 and 2/3, but by doing various tests, it was seen that the prediction of the model was better by giving the test set a size of 25% of the total. As a random state parameter was given value 16. Several values were tried, and 16 seemed to be the optimal value.

5 Performance Metrics in Classification

Before treating the fitting and the evaluation of the model, we expose the metrics that have been used in order to study which was the best predictive model.

- Accuracy - is one of the values that in a more immediate way gives a complete vision of the model and its operations. Unfortunately this parameter does not give detailed information about the specific problems, this lose information about the prediction of the various classes.

$$error\_rate = \frac{\#errors}{\#instances} = \frac{FN + FP}{TP + TN + FP + FN}$$

$$accuracy = 1 - error\_rate = \frac{TP + TN}{TP + TN + FP + FN}$$

- Recall - it helps when the value of false negatives is high

$$recall = \frac{truepositive}{realpositive} = \frac{TP}{TP + FN}$$

- Precision - it helps when the value of false positives is high

$$precision = \frac{truepositive}{predictedpositive} = \frac{TP}{TP + FP}$$

- f1-score - this parameter improves the accuracy by combining precision and recall. In this way it is possible to have a low number of false positives and false negatives, calculating the true values of the model, not being disturbed by false alarms. f1-score is a decimal value between 1 and 0, indicating maximum and minimum operation respectively.

$$f1 - score = \frac{truepositive}{predictedpositive} = 2 * \frac{precision * recall}{precision + recall}$$

Finally, the confusion matrix was used: each column represents the predicted values, while each row represents the true values, the element on row i and column j identifies the number of cases in which the classifier classified the "true" class i as j. The 4*4 matrix that will be constructed in our case, represents in position [1,1] the true negatives, in position [1,2] the false positives, in position [2,1] the false negatives and in position [2,2] the true positives.
A good functioning of the model is identified in the matrix when the values in the diagonal are high and the values outside the diagonal are very low, tending to zero.

6 Modelling and Fitting

Several models were tried, and the ones that performed best were the the DecisionTreeClassifier, and the RandomForestClassifier.

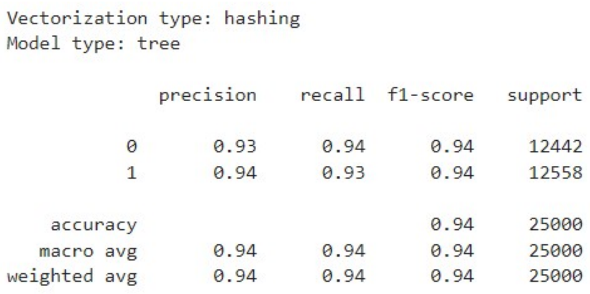
- DecisionTreeClassifier - As the name suggests, a decision tree model partitions the data by making decisions based on the answers to a set of questions. Based on the characteristics of the training data set, the decision tree model learns a set of questions to determine (via inference) the labels of the sample classes.
  Using the decision algorithm, we start at the root of the tree and partition the data according to the feature that yields the maximum information gain, i.e., we need to define an objective function that we want to optimize with the tree learning algorithm. The information gain is simply the difference between the impurity of the parent node and the sum of the impurities of the child nodes: the lower the impurity of the child nodes, the higher the information gain. However, for simplicity and to reduce the combinatorial search space, most libraries (including scikit-learn) implement binary decision trees. This means that two child nodes Dleft and Dright depend on each parent node.
  We must be careful: the deeper the decision tree, the more complex the decision boundary becomes, resulting in an overfitting problem. Using scikit-learn, one can restrict the decision tree to a maximum depth, say 3, by specifying it in the method parameters. In our case, we did not impose any depth limit. One can also specify a parameter defining the impurity criterion (by default Gini impurity). Intuitively, the Gini impurity can be considered as a criterion to minimize the probability of misclassification.
  Advantage: decision trees are particularly interesting if we are interested in interpretability.

- RandomForestClassifier - It generates multiple trees but always starting from the same training set, choosing however the optimal split in a random subset of the possible splits. In this way each tree is different.
  Random forests have gained great popularity in machine learning applications during the last decade due to their good classification performance, scalability and ease of use.
  The idea behind using a set of learning systems is to combine multiple weak learning systems to build a more robust model, a strong learning system that offers better generalization error and is less susceptible to overfitting problems.
  Advantage: we do not have to worry too much about choosing a good value for the hyperparameters. In general, we do not have to prune the random forest, since as a whole the model is quite noise resistant compared to the individual decision

trees. The only parameter we really need to worry about, in practice, is the number of k trees we chose for the random forest. Typically, the larger the number of trees, the better the performance of the random forest classifier will be, with the disadvantage of increased computational cost. Fortunately, we don't have to build the random forest classifier manually using individual decision trees; scikit-learn already provides a ready-to-use implementation.

7 Evaluation

Once the model fitting phase is finished, the prediction of the test set is done. For every case it has been visioned the confusion matrix.
We have tried to fit DecisionTreeClassifier with some hyperparameters. When we use



```
Vectorization type: hashing
Model type: tree

              precision    recall  f1-score   support

           0       0.93      0.94      0.94     12442
           1       0.94      0.93      0.94     12558

    accuracy                           0.94     25000
   macro avg       0.94      0.94      0.94     25000
weighted avg       0.94      0.94      0.94     25000
```

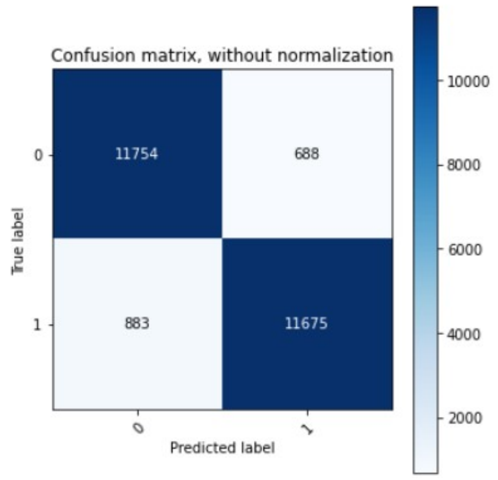


Figure 2: Confusion Matrix and metrics of DecisonTreeClassifier with HashingVectorizer

HashingVectorizer we have results better than CounterVetorizer(that are very similar with TfidVectorizer), the accuracy improve of 3 percentage points, also the precision and recall are higher of 4, 5 percentage points.
Default values of DecisionTreeClassifier for criterion and splitter are ”gini” and ”best”. The criterion is the function to misure the quality of split, gini use Gini_Impurity_Measure while entropy the information gain. Splitter instead is the strategy used to choose the split at each node. “best” strategy is that to choose the best split and “random” to choose the best random split. And in this case the change of this hyperparameters don't change the value of accuracy and other metrics.
With both vectorizer the choose of ”gini” o ”entropy” parameters not influence the result of the predict, but our effect are more visble with HashingVectorizer.

```
Vectorization type: count
Model type: tree

              precision    recall  f1-score   support

           0       0.90      0.91      0.91     12442
           1       0.91      0.90      0.90     12558

    accuracy                           0.91     25000
   macro avg       0.91      0.91      0.91     25000
weighted avg       0.91      0.91      0.91     25000
```

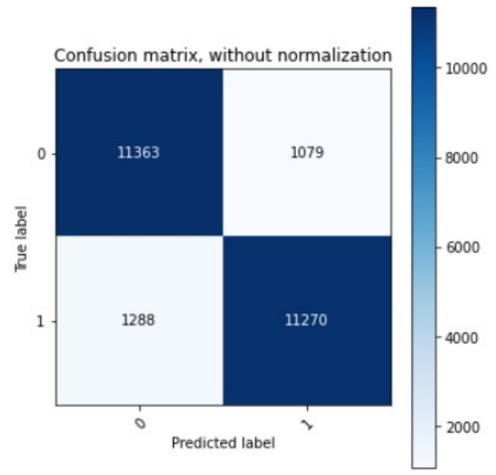


Figure 3: Confusion Matrix and metrics of DecisonTreeClassifier with CountVectorizer

```
Vectorization type: count
Model type: ran_for

              precision    recall  f1-score   support

           0       0.92      0.98      0.95     12442
           1       0.98      0.92      0.95     12558

    accuracy                           0.95     25000
   macro avg       0.95      0.95      0.95     25000
weighted avg       0.95      0.95      0.95     25000
```

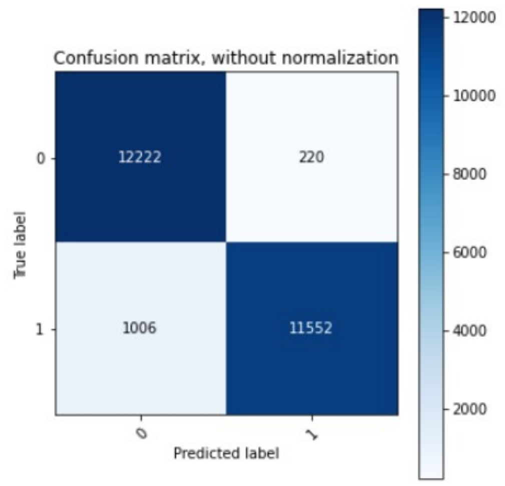
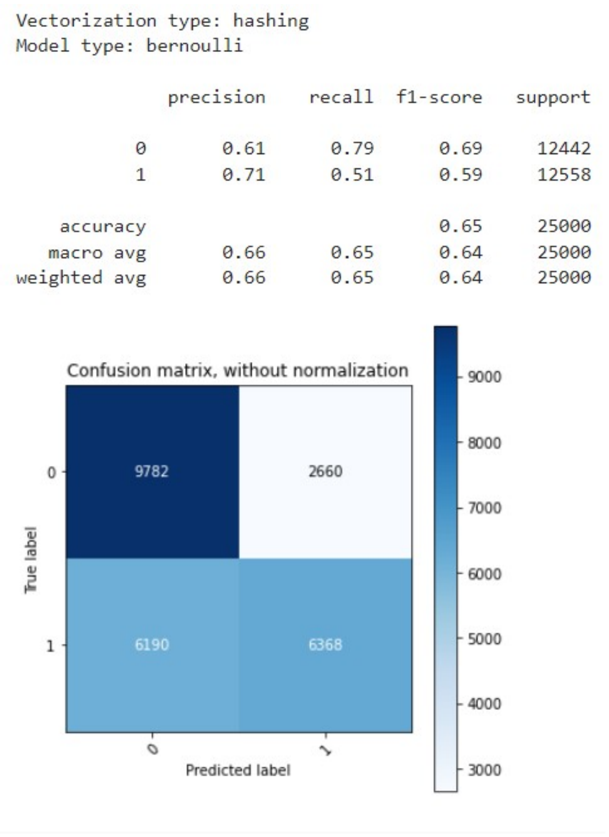


Figure 4: Confusion Matrix and metrics of RandomForestClassification with CountVectorizer

When we use RandomForestClassification we use n_estimators (that indicates the number of trees in the forest) egual to 100 and we obtain a better result of DecisionTree, we tried also to change the number of trees to 200 and use a criterion ”entropy” but the result remained the same.
Increasing the number of trees worsens the execution time of the model, so we kept the estimator value equal to 100, and decided to predict the new dataset with this classifier. The use of the vector is not the best solution because it increases the execution time a lot. With a random forest of 10 or 50 trees the metrics remain similar to the decision tree, and with a larger forest the execution time gets exponentially worse, so it was decided to use the compromise between a medium sized forest and a vectorizer that did not create a very large vector.

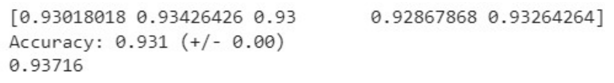
For completeness we point out that tests were also done with other models, including BernoulliNP, and the results were not very good. For this reason we have concentrated on the above models.



```
Vectorization type: hashing
Model type: bernoulli

              precision    recall  f1-score   support

           0       0.61      0.79      0.69     12442
           1       0.71      0.51      0.59     12558

    accuracy                           0.65     25000
   macro avg       0.66      0.65      0.64     25000
weighted avg       0.66      0.65      0.64     25000
```

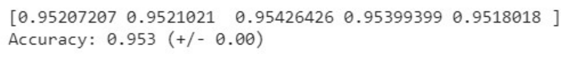
Figure 5: Confusion Matrix and metrics of BernoulliNP with HashingVectorizer

All model tests were also evaluated using K-fold CV, which is a cross-validation method. The set is shuffled randomly (or by setting a size, and this time 0.33 was entered), the data set is divided into k groups (5).
Each group is used as a validation set and all others as train sets. The model is established using the train set and evaluated using the validation set, the model evaluation score is stored in a list.
Each result was equivalent to those obtained with the confusion matrices



```
[0.93018018 0.93426426 0.93       0.92867868 0.93264264]
Accuracy: 0.931 (+/- 0.00)
0.93716
```

Figure 6: K-Fold Cross DecisionTree with HashingVectorizer



```
[0.95207207 0.9521021  0.95426426 0.95399399 0.9518018 ]
Accuracy: 0.953 (+/- 0.00)
```

Figure 7: K-Fold Cross RandomForest with CountVectorizer

8 Prediction

At this point it is possible to use our classifier on the blind test, and make the predictions of the dataset without label. The dataset is naturally loaded, the first two columns are seeded and vectorized as done for the training and testing dataset.
At that point the predict operation is performed and the result is saved to a text file.
At the end our classification with DecisionTree predicts 5107 lines without bugs and 4893 lines with bugs, and RandomForest predicts 5316 lines without bugs and 4684 with bugs.
We decided of save on file .txt the results obtained with RandomClassifier because although they worsen the evaluations of the 1's, they improve in greater ratio the evaluations of the 0's predictions.