Contents

Section 1

Overview

The objective of the homework is to perform an analysis of different classification methods. In order to make this comparison we take into consideration a bug detection problem. We have a dataset with 100000 samples, each of which contains a list of assembly instructions, the related source code and a label indicating whether the mapping is correct or not. Discarding the meta information (line number, function name, program name), we have to learn a binary classification function $f(a, s) = \{0, 1\}$. The problem can be simplified by considering assembly instructions and source code lines as a unique sequence of tokens.

The methods exploited are the basic classification algorithms, such as Naive Bayes, Decision Trees, Support Vector Machines and Logistic Regression. To evaluate the obtained results we can leverage on cross validation technique (useful for re-sampling the training data several times) and on metrics like

Accuracy $A = (TP + TN) \,/\, (TP + FP + TN + FN)$

Precision $P = TP \,/\, (TP + FP)$

Recall $R = TP \,/\, (TP + FN)$

F1-score $F1 = 2\,PR \,/\, (P + R)$

where

	Predicted Class	
True Class	Yes	No
Yes	True Positive (TP)	False Negative (FN)
No	False Positive (TP)	True Negative (TN)

Best results are reached with some data manipulation and hyperparameter tuning, as explained in the following sections.

Section 2

Naive Bayes

The first method used is the Naive Bayes classifier, a quite robust and ubiquitous probabilistic method often associated with the problem of text classification and used as baseline.

First of all, we need to load and extract our data from the *mapping_traces* dataset.

```
db = pd.read_csv(filename, sep="\t", header=None, names=["assembly", "
    source", "line", "func", "program", "bug"])
text = list(map(lambda n1, n2: n1 + " " + n2, db.assembly, db.source))
label = list(db.bug)
```

Now we have to process the raw data. The feature extraction step is vital to give information that are useful and presented in the expected way to the subsequent algorithms. Each individual token is treated as a feature. We transform the list of strings given by the joint of assembly and code instructions (the input of our function) using the bag of words representation for the Bernoulli model and for the multinomial model. The bag of word vectorization produces a vector representing the token occurrence count (or binary occurrence) for each string. This step hides a quite strong assumption of independence between the tokens and between each token and its position within the string. The vectorization is performed by *HashingVectorizer* and *CountVectorizer* and the output is a sparse matrix of occurrences.

```
if selected_model == "bernoulli":
  vectorizer = HashingVectorizer()
else:
  vectorizer = CountVectorizer()

X_all = vectorizer.fit_transform(text)
y_all = label
```

Given the input $x = (t_1, t_2, ..., t_n)$ and the corresponding label c, Naive Bayes tries to learn the classification function estimating

$$\begin{aligned} c_{NB} &= \arg\max_{c_j \in \{0,1\}} P(c_j \,|\, t_1, t_2, ..., t_n) \\ &= \arg\max_{c_j \in \{0,1\}} \frac{P(t_1, t_2, ..., t_n \,|\, c_j) P(c_j)}{P(t_1, t_2, ..., t_n)} \\ &= \arg\max_{c_j \in \{0,1\}} P(t_1, t_2, ..., t_n \,|\, c_j) P(c_j) \\ &= \arg\max_{c_j \in \{0,1\}} P(c_j) \, \Pi_i \, P(t_i \,|\, c_j) \end{aligned}$$

It exploits Bayes theorem and the same independence assumptions stated before. We can have different implementation of this method depending on the chosen model. Multinomial Naive Bayes works with ordinal feature vectors; Bernoulli Naive Bayes works with boolean feature vectors.

Before applying a certain model we have to partition our initial data into two sets, the training set and the test set. The first one is useful to perform the actual learning steps, while the second one is needed to evaluate the performances: in fact, the classifier behaves well only if it achieves good results in classifying new instances of the input variable. A rule of thumb is to partition so that the training test is two thirds of the dataset and the test set is the remaining part.

```
X_train, X_test, y_train, y_test = train_test_split(X_all, y_all,
    test_size=0.33, random_state=16)

if selected_model == "bernoulli":
  model = BernoulliNB()
elif selected_model == "multinomial":
  model = MultinomialNB()

model.fit(X_train, y_train)
```

In order to verify how good the performances are, we can print the confusion matrix and compute the metrics presented before. Using the cross validation, we can find the mean of each metric with respect to multiple shuffles of training and test sets.

```
y_pred = model.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
cv = ShuffleSplit(n_splits=5, test_size=0.33, random_state=16)
scores = cross_val_score(model, X_all, y_all, cv=cv)
print("Accuracy: %0.2f" %scores.mean())
```

Here are the results

	Bernoulli NB	Multinomial NB
Accuracy	0.61	0.50
Precision	0.63	0.50
Recall	0.61	0.50
F1-score	0.60	0.50

They are not encouraging. To improve them, we can make some pre-processing manipulation on input data. Looking at the structure of the dataset, we notice that a lot of instruction are of this kind:

$$< \text{movb 0 l 53 78, uint8t l 53 78} = 0 \text{ ;, } 0 >$$

The name of the variable l_{5378} and the constant value 0 are not so significant with respect to our task if they are consistent in both assembly instructions and source code lines. In this case the vectorized representation for the multinomial model counts these elements more than once when the consistency criteria holds. Thus, we can ignore the related entries of the vector setting them to zero.

```
X_all = vectorizer.fit_transform(text)
y_all = label

X_all[X_all >=2] = 0
```

Thanks to this simple modification we already observe an improvement

	Bernoulli NB	Multinomial NB
Accuracy	0.61	0.83
Precision	0.63	0.83
Recall	0.61	0.83
F1-score	0.60	0.83

A further move is given by the model parameters tuning. For what concerns Naive Bayes, we can modify the alpha parameter, that is the value of the additive smoothing parameter used in estimating probabilities $P(t_i \mid c_j)$. When alpha is one, we have the so-called Laplace smoothing.

```
parameters = {"alpha": [i/10 for i in range(1,20)]}

grid_obj = GridSearchCV(model, parameters)
grid_fit = grid_obj.fit(X_train, y_train)
best_model = grid_fit.best_estimator_
```

In this case we cannot appreciate a consistent boost in the performances

	Bernoulli NB	Multinomial NB
Accuracy	0.62	0.83
Precision	0.63	0.83
Recall	0.62	0.83
F1-score	0.61	0.83

Section 3

Logistic Regression

Logistic Regression is a discriminative model based on maximum likelihood estimation. We compute the probability

$$P(c_i \,|\, x) = \sigma(w^T x)$$

. Given a cross-entropy error function, that is the negative log likelihood $E(w) = -\ln P(c \,|\, w)$, we can solve the optimization problem of minimizing this error with an iterative reweighing technique.

We execute the same code as in Naive Bayes with the only difference of the model. Adjusting the maximum number of iterations (*max_iter* parameter) is useful for allowing the solver to converge. On the contrary, tuning other parameters such as the penalty (*l2* norm, *l1* norm) or C (inverse of regularization coefficient) does not affect the performance with respect to default values.

```
elif selected_model == "logistic":
  model = LogisticRegression(max_iter=1000)
  parameters = [{"C": np.logspace(-3,3,10),"max_iter":[1000],
    "penalty":["l1","l2"]}]
```

The results of the classification using Logistic Regression in both cases of plain and pre-processed data are

	plain data	pre-processed data
Accuracy	0.50	0.90
Precision	0.50	0.90
Recall	0.50	0.90
F1-score	0.50	0.90

Section 4

Perceptron

Simple Perceptron is a linear binary classification model composed by two elemants: a linear combination between a set of weights w and the feature vector x, and a threshold function

$$f(x) = \begin{cases} 1 & \text{if } w^T x > 0 \\ 0 & \text{otherwise} \end{cases}$$

We have to learn w_i that minimize the squared loss error function $E(w)$, updating the value at each iteration

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

As happens in Logistic Regression, we can add a regularization term. The parameters on which to perform the tuning process can be again the penalty, its related coefficient (alpha parameter) and the coefficient of the weight updates (eta parameter).

```
elif selected_model == "perceptron":
  model = Perceptron()
  parameters = [{"eta0":[i/10 for i in range(1,20)],
    "alpha":[i/50000 for i in range(1,10)],"penalty":["l1","l2","none"]}]
```

The results of the classification using Perceptron are

	plain data	pre-processed data
Accuracy	0.51	0.84
Precision	0.51	0.84
Recall	0.51	0.84
F1-score	0.51	0.84

Section 5

Support Vector Machine

Support Vector Machine is a linear model that tries to separate instances of different classes with the "best" possible hyperplane $w^T x + w_0 = 0$. In particular, the maximum-margin hyperplane is the one that maximize the distance between itself and the nearest point from each group. The method is quite robust, and it works even in the case of data that are not linearly separable (introducing the slack variables).

After rescaling all the points in order to obtain the canonical representation, we have to solve the optimization problem

$$w^*, w_0^* = \arg\min \frac{1}{2} \left|\left|w\right|\right|^2 \quad \text{s.t. } c_n(w^T x_n + w_0) \geq 1 \; \forall n$$

Solving with Lagrangian method we can express both weights and the hyperplane using the Lagrangian multipliers a^*.

```
elif selected_model == "svc":
  model = LinearSVC(max_iter=2000)
  parameters = [{"C":np.logspace(-3,3,10),
    "max_iter":[2000], "penalty":["l1","l2"]}]
```

The results of the classification using Support Vector Machine are

	plain data	pre-processed data
Accuracy	0.52	0.90
Precision	0.51	0.90
Recall	0.51	0.90
F1-score	0.50	0.90

Section 6

Decision Trees

Decision Tree is the last classification model taken into consideration. A Decision Tree is tree in which an internal node analyze one chosen feature attribute A_i and the leaves assigns a classification value. The branching is done on the basis of the feature values, and the path from the root to a leaf represent a rule to classify an instance.

The algorithms that construct a decision tree are determined by the way in which they choose what attribute to consider at a given time. Actually, output depends on this attribute order. One criteria to make this choice is the information gain, the expected reduction in entropy

$$IG(D, A_i) = E(D) - \sum_j \frac{|D_{a_{i,j}}|}{|D|} E(D_{a_{i,j}})$$
$$E(D) = -p_+ \log_2 p_+ - p_- \log_2 p_- \in [0, 1]$$

where p_+ is the proportion of positive instances, p_- is equal to $1 - p_+$, and $D_{a_{i,j}}$ is the subset of D in which each element has value $a_{i,j}$. Information gain can be seen has an heuristic to speed up the search in the space of all possible decision trees.

A different criteria is the Gini impurity (the one used as default parameter in scikit-learn) based on the probability of choosing an item and the probability of a mistake in the classification of this item. The parameter tuning can include these criteria, besides many more elements that characterize trees (such as *max_depth*, *min_samples_split* and *min_samples_leaf*). In this case it has been choosen the *min_samples_split*, that gives us an actual improvement on the results.

```
elif selected_model == "tree":
  model = tree.DecisionTreeClassifier()
  parameters = [{"min_samples_split":[2,3,4,5],
    "criterion": ["gini","entropy"]}]
```

The results of the classification using Decision Trees are

	plain data (gini, 2)	preprocessed data (gini, 2)	plain data (entropy, 3)	preprocessed data (entropy, 4)
Accuracy	0.90	0.88	0.91	0.90
Precision	0.90	0.89	0.91	0.90
Recall	0.90	0.88	0.91	0.90
F1-score	0.90	0.88	0.91	0.90

In this case we can observe a different behavior with respect to all other methods: if we execute the pre-processing computation described in Naive Bayes section, we obtain a slightly worse performance. This is because setting to zero a lot of entries of the sparse matrix in input means to loose some information. However, those information can be useful to build a better decision tree, with the ”right” branching at each internal node and rules with greater generalization power.