SAPIENZA
UNIVERSITÀ DI ROMA

Machine Learning Homework 1:

Function classification

Simone Giordano

1772347

Index

Introduction

The scope of the homework is to identify various functions that are in a dataset. This program can be very useful during the reversing of an unknown program to understand better how it works and what are the functions that it uses.

The functions in the dataset are divided in 4 different categories, each of them has different properties:

- math (use a lot of arithmetic operations)
- encryption (they are complex functions, strong use of xor and shift operations)
- sorting (the easier function, use of compare and for operations)
- String manipulation (a lot of comparison and swap operations)

To train the system, the data have been preprocessed in two different manners, depending on what type of fitting have been used, i.e., for the Multinomial classifier have been used the count and tfid vectorizers, and for the KNN classifier have been used a custom way to find all the features in the functions.

The types of fitting used are two, i.e., Multinomial classifier and KNN neighbor classifier.

The multinomial Naive Bayes classifier is suitable for classification with discrete features (in this case the number of every instructions in the functions). The multinomial distribution normally requires integer feature counts. The Neighbors classifier implementing the k-nearest neighbors vote.

1 Data preprocessing

As seen in the introduction, the data preprocessing is based on what type of fitting is used in the program's function.

In the first case, the data are vectorized in two ways that are very similar, using count and tfid vectorizer. This is due to the fact that the multinomial Naïve Bayes classifier is suitable for classification with discrete features, and in this case we can take the values that are in the dataset saved in a jsonl file (file where every line represents a specific function). Those specific values are saved in a list called "lista_asm" that is the linear list of assembly instruction of each function. After opening the jsonl file, and saved the values in a list, the multinomial function has been called. This function begins the preprocessing data initializing two lists, where will be saved the values in lista_asm and the corresponding label. Later will begin the parsing of the list initialized before.

```
#Iteration on every string that represent an istance of the dataset
for datasetStr in d_list:

    #json.load works just on a single line of a json value
    result = json.loads(datasetStr)


    label = str(result.get("semantic"))

    #Change the ' with " to work better with the re library
    stringChanged = str(result.get("lista_asm")).replace("'", '"')

    #Find the single values of lista_asm that are between " ", remember that this includes also the , that are between two values
    singleValues = re.findall('"([^"]*)"', stringChanged)

    TotalDB.append(str(singleValues))
    TotalLabel.append(label)
```

Figure 1 Dataset parsing

In this for cycle, for every line loaded as json value, will be saved in the list of labels, previously initialized, the label of each line and in the list of values, the operations of the specific instruction. Those operations in the "lista_asm" of the json line are all the values

between two quotes; this operation can be achieved thanks to the function "findall(constraint,list)" of re library.

After this operation, for instance, for just the first element of the dataset, we got two lists of this type, where the first one is the list of instructions and the second one is the corresponding label:

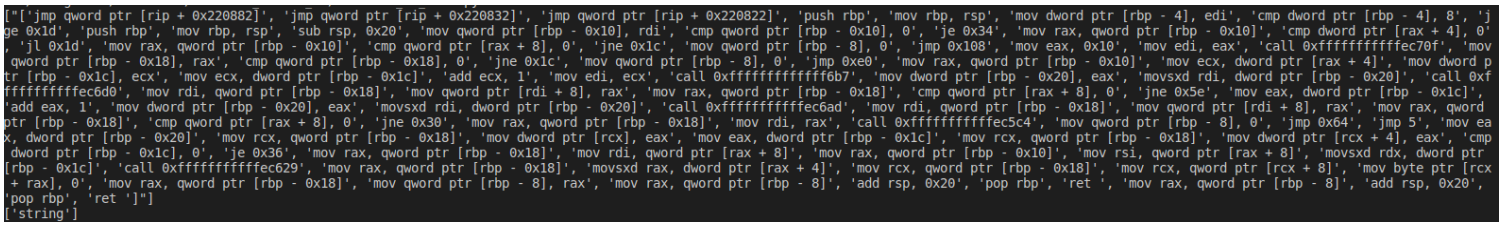


Figure 2 Example for lists of instructions and label for the first element

For better results, after creating the two lists of label and corresponding instructions, the lists are converted in panda series Objects, that are One-dimensional ndarray with axis labels, creating a 1:1 correspondence between the labels and the instructions.

Later, based on what type of vectorization we want to use (the program automatically uses both the types), the data in the lists of instructions are converted via countVectorizer() or TfidVectorizer().

```
if vectorizer == "count":
    vectorizerForBlind = 'count'
    vectorizer = CountVectorizer()
elif vectorizer == "tfid":
    vectorizerForBlind = 'tfid'
    vectorizer = TfidfVectorizer()

X_all = vectorizer.fit_transform(dfDB)
```

Figure 3 Types of vectorization

Those two types of vectorization are very similar, but they differ on how is presented the output, in fact the TfidfTransformer transforms a count matrix to a normalized tf or tf-idf representation. So, although both the CountVectorizer and TfidfTransformer produce term frequencies, TfidfTransformer is normalizing the count. More precisely, the

CountVectorizer counts the word frequencies. With the TFIDFVectorizer the value increases proportionally to count but is inversely proportional to frequency of the word in the corpus. In comparison, following the previous example of the first function, the two vectorization are presented in this way, where, in the brackets, there is the correspondence between the number of the function of the dataset and the instruction:

```
(0, 44)       0.019718075168127475
(0, 35)       0.019718075168127475
(0, 22)       0.009859037584063738
(0, 16)       0.009859037584063738
(0, 43)       0.009859037584063738
(0, 46)       0.009859037584063738
(0, 11)       0.009859037584063738
(0, 41)       0.0788723006725099
(0, 13)       0.009859037584063738
(0, 15)       0.009859037584063738
(0, 9)        0.009859037584063738
(0, 17)       0.009859037584063738
(0, 12)       0.009859037584063738
(0, 18)       0.009859037584063738
```

Figure 4 Tfid Vectorization

```
(0, 38)       36
(0, 36)       55
(0, 45)       3
(0, 8)        1
(0, 7)        1
(0, 6)        1
(0, 37)       2
(0, 40)       44
(0, 33)       46
(0, 47)       5
(0, 24)       18
(0, 27)       3
(0, 23)       8
(0, 29)       1
```

Figure 5 Count vectorization

This dictionary will be used in the training and testing of the system.

Regarding the KNN classifier, the vectorization of the instructions is done “manually”, retrieving the number of the main instructions for every function, and later those values are saved in a numpy matrix, initialized to 0, where the number of the rows are the total number of functions in the dataset and the number of columns are the features found.

```
arrayFeatures = np.zeros(shape = (14397,12))
arrayLabel = np.chararray((14397,1),itemsize=10)

counter = 0

#Iteration on every string that represent an istance of the dataset
for datasetStr in d_list:

    #json.load works just on a single line of a json value
    result = json.loads(datasetStr)

    label = str(result.get("semantic"))

    #Change the ' with " to work better with the re library
    stringListaASM = str(result.get("lista_asm")).replace("'", '"')

    #Find the single values of lista_asm that are between " ", remember that this includes also the , that are between two values
    singleValuesLista = re.findall('"([^"]*)"', stringListaASM)


    #Find the features of the function in the "lista_asm"

    numXMM = str(singleValuesLista).count("xmm")
    numXOR = str(singleValuesLista).count("xor")
    numCMP = str(singleValuesLista).count('cmp')
    numMOV = str(singleValuesLista).count('mov')
    numSHL = str(singleValuesLista).count('shl')
    numRO = str(singleValuesLista).count('ro')
    numAND = str(singleValuesLista).count('and')
    numRCR = str(singleValuesLista).count('rcr')
    numRCL = str(singleValuesLista).count('rcl')
    numOR = str(singleValuesLista).count("'or")
    numSHR = str(singleValuesLista).count('shr')
    numNOT = str(singleValuesLista).count('not')

    arrayLabel[counter] = [label]
    arrayFeatures[counter] = [numXMM,numXOR,numCMP,numMOV,numRO,numRCR,numRCL,numSHL,numSHR,numAND,numOR,numNOT]
    counter += 1
```

Figure 6 Vectorization for KNN

For every function, the name of the label is saved in a numpy chararray.

Following the previous examples, for the first function in the dataset, the two arrays are the following, that is consistent with the previous definition of string:

```
[[ 0.  0.  8. 50.  0.  0.  0.  0.  0.  0.  0.  0.]]
[[b'string']]
```

2 Fitting and processing of data

The model classifiers used in the program are the Multinomial and the KNN ones. For both, there is a division in the processed data between the set of the train values and the set of the test values with a percentage of respectively 80/20%, which, according to me, is the best split percentage to get good results.

```
X_train, X_test, y_train, y_test = train_test_split(X_all, y_all,
        test_size=0.2, random_state=10)
```

Later, the X_train taken from the split, that are the 80% of the instructions of the dataset, and the Y_train, that are the 80% of the corresponding labels, are fitted in the classifier of the running function creating the two models.

For knn classifier has been used a number of neighbors equal to 5, having the best results in the computation.

The model is then used to predict the labels of the X_test 's list (the 20% remaining of the dataset) with the function ".predict".

3 Results comparison

To check the results of the three systems, after creating the prediction of the 20% of the remaining dataset not used in the training of the system, is created the classification report that takes in input the predicted labels and the real ones, where, after a comparison between them, creates a clear report on the accuracy of the system and, later on, a plot of the confusion matrix that shows the accuracy of the classification based on the labels of the functions.

3.1 Classification reports and Confusion matrices

The classification reports are the following:

	precision	recall	f1-score	support
encryption	0.98	0.80	0.88	549
math	0.97	0.91	0.94	874
sort	0.86	0.87	0.86	827
string	0.78	0.96	0.86	630
accuracy			0.89	2880
macro avg	0.90	0.88	0.88	2880
weighted avg	0.90	0.89	0.89	2880

Multinomial classification w/ count vectorization

	precision	recall	f1-score	support
encryption	0.99	0.73	0.84	549
math	1.00	0.92	0.96	874
sort	0.73	1.00	0.84	827
string	0.97	0.83	0.89	630
accuracy			0.89	2880
macro avg	0.92	0.87	0.88	2880
weighted avg	0.91	0.89	0.89	2880

Multinomial classification w/ count vectorization

	precision	recall	f1-score	support
b'encryption'	0.98	1.00	0.99	526
b'math'	0.98	0.99	0.99	913
b'sort'	0.94	0.97	0.95	808
b'string'	0.95	0.90	0.92	633
accuracy			0.96	2880
macro avg	0.96	0.96	0.96	2880
weighted avg	0.96	0.96	0.96	2880

KNN neighbor classification w/ custom vectorization

The confusion matrices are the following.

For multinomial classification:

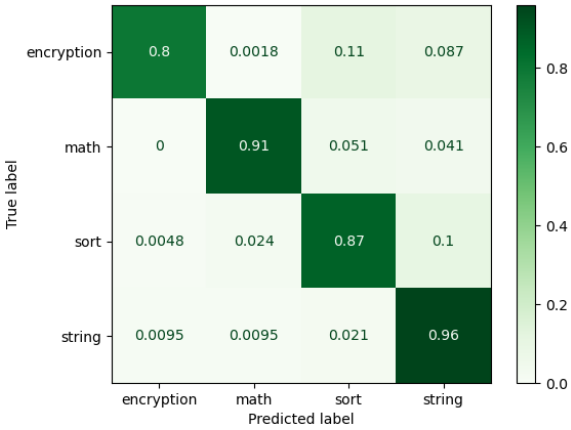


Figure 8 Multinomial classification w/ count vectorization

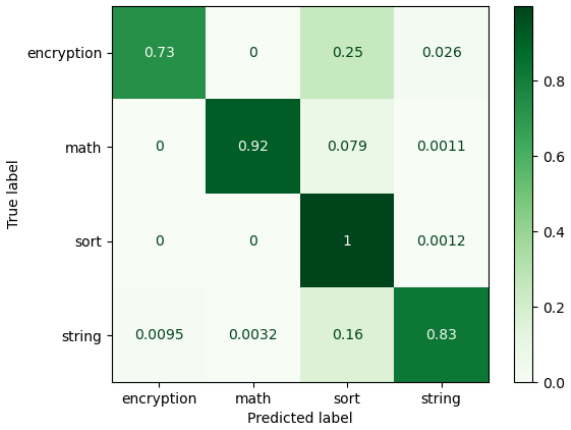
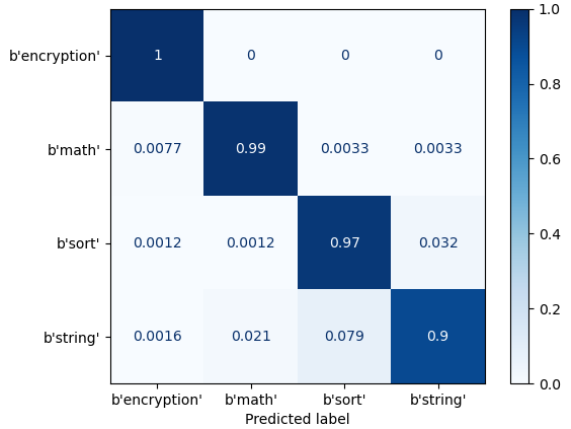


Figure 7 Multinomial classification w/ tfid vectorization

For KNN classification:



As can be seen in the plots and in the classification reports, the knn classification is the one that achieved the best results. This achievement can be due to the fact that the knn classifier works on values that are perfect for the evaluation (the instructions of the functions) thanks to the custom vectorization and to the fact that the knn classifier, with k

valor of 5, is the best representation of the main features of the functions (all the functions have in average 5 main instructions);

multinomial classification has worse results because it is perfect for dataset composed by texts, and also if the instructions of the functions have been considered as texts, they differ a priori from dataset composed, for instance, by phrases.

3.2 **BlindTest computation**

For the blindest have been used the custom vectorization used in the knn classifier and the relative trained model.

4 Sources

- https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html
- https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier.predict
- https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html
- https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html
- https://numpy.org/doc/stable/reference
- MLEx3. SMS SPAM Classification
- Various StackOverflow help