Homework 1

Machine Learning for Compiler Provenance

Lorenzi Flavio
1662963

2 Nov 2019

A Y 2019-2020

Contents

1. Description of the problems

This homework has the goal to understand how classifiers works in machine learning problems and to develop some classifiers to making predictions.

Two approaches are required:

-binary classifier whose detect if a function is optimized with High or Low

-multiclass classifier whose predict the current compiler (gcc,icc,clang)

This report will show a personal approach to a Naïve Bayes classifier for opt prediction based on a train dataset (file jsonl composed by 30000 instructions); another binary method (Support Vector Machine classifier) is implemented to show comparisons and performance. Then it will be shown how to solve the compiler prediction: it was used again the SVM classifier (multiclass implementation this time) and a simple variant, the K-Nearest Neighbours classifier.

After each problem we'll focus on comparing the algorithms used.

Note: while each algorithm is implemented through libraries (like scikit-learn) with a lot of functions very useful for our goal, the Naive Bayes is totally implemented by me from the scratch; I made this choice to put myself further to the test. The results found (in accuracy terms) are not perfect but acceptable, and certainly can be improved in the future.

2. Dataset and features extraction

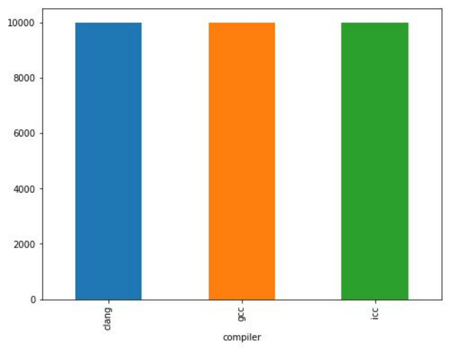
The provided dataset is a json list file where each row is a json object with the following keys:

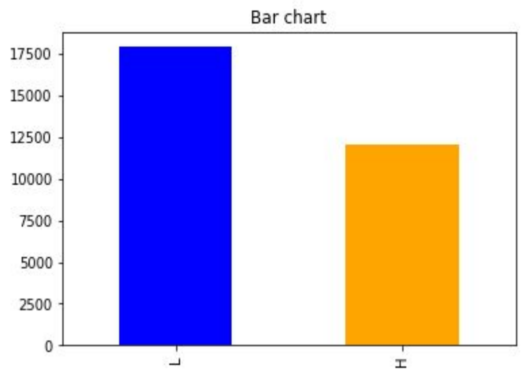
– **instructions**: the assembly instruction for the function.

– **opt**: the ground truth label for optimization (‘H’ , ‘L’)

– **compiler**: the ground truth label for compiler (icc, clang, gcc)

Note: following graphs show that optimizations are not balanced while number of compilers are.





example:

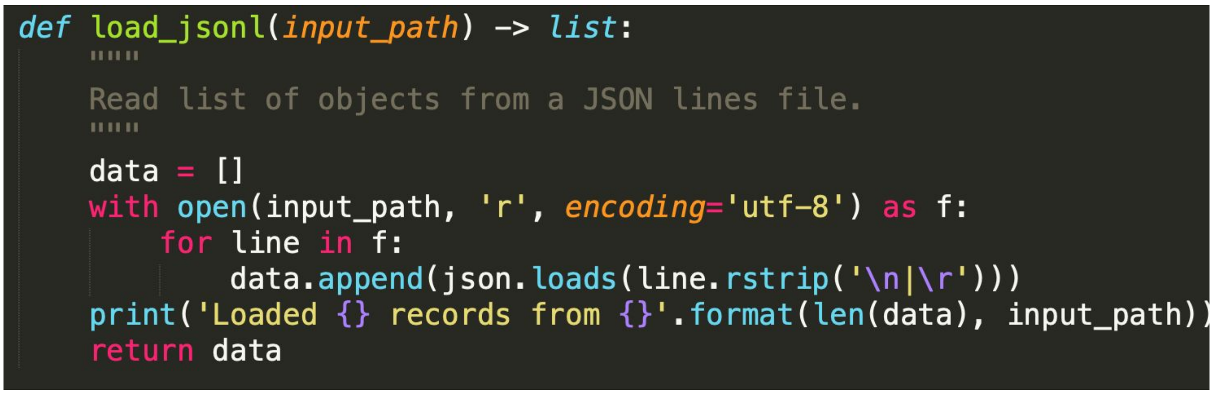
"instructions": ["xor edx edx", "cmp rdi rsi", "mov eax 0xffffffff", "seta dl", "cmovae eax edx", "ret"],

"opt": "H",

"compiler": "gcc "

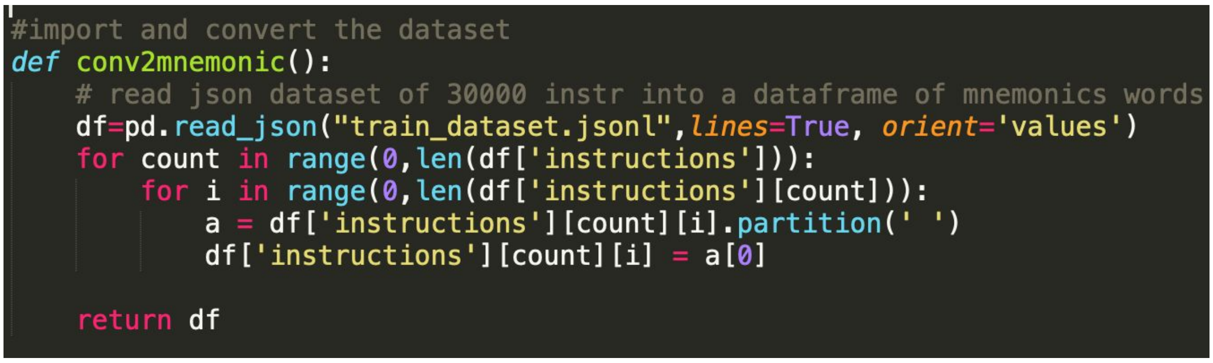
Through **load json()** method I was able to analyze the dataset, creating a python object and a list of lists easy to iterate.

With **data = utils.load_jsonl('train_dataset.jsonl')** I load the dataset, and with the method **get_instructions_optimizer(current)** it was easy iterate on each *current[‘opt’]* and *current[‘instruction’].*



```python
def load_jsonl(input_path) -> list:
    """
    Read list of objects from a JSON lines file.
    """
    data = []
    with open(input_path, 'r', encoding='utf-8') as f:
        for line in f:
            data.append(json.loads(line.rstrip('\n|\r')))
    print('Loaded {} records from {}'.format(len(data), input_path))
    return data
```

For the multiclass prediction problem the dataset was imported and iterate in a different way, to make easy the “mnemonic transformation” useful for the feature extraction. So we have the following method:



```python
#import and convert the dataset
def conv2mnemonic():
    # read json dataset of 30000 instr into a dataframe of mnemonics words
    df=pd.read_json("train_dataset.jsonl",lines=True, orient='values')
    for count in range(0,len(df['instructions'])):
        for i in range(0,len(df['instructions'][count])):
            a = df['instructions'][count][i].partition(' ')
            df['instructions'][count][i] = a[0]

    return df
```

To best implement our classifiers, we have to perform a good feature extraction by studying the form of the instructions and analyzing the most recurrent words. To simplify I worked first on a partial dataset (only 10 instructions) and then with a semi-partial dataset (3000 instructions); in the end the training is done with all the 30000 instructions.

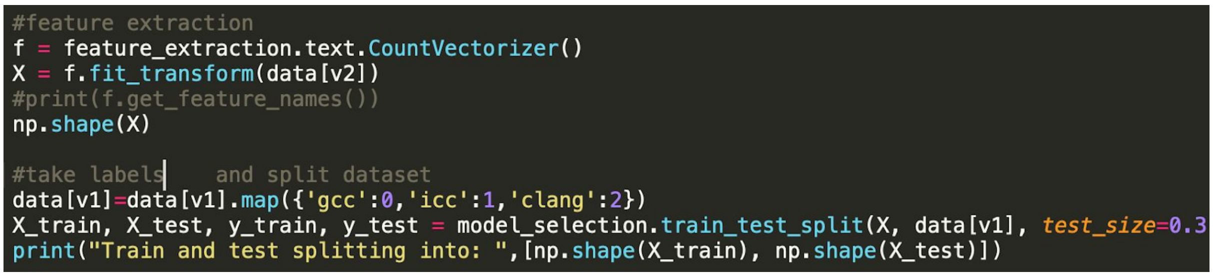
For the first problem (opt prediction) the Feature Extraction was made by observing the instructions:

- small instructions were Low-optimized and big ones were High;
- if ‘lea’,’call’ and ‘xor’ terms appeared more than a certain threshold (respectively 10,30,4) the current instruction was High-optimized.

The accuracy for each feature caption was around the 0.6% so I decided to go continue and implement the algorithms.

About the compiler prediction problem instead I was based on *sklearn* methods for the feature extraction. Computing the most recurrent words and features for each compiler I create a vector with **feature_extraction.text.CountVectorizer()** and fit the features to the variable X.

With the above methods I create a dictionary of common features between each extraction.



```
#feature extraction
f = feature_extraction.text.CountVectorizer()
X = f.fit_transform(data[v2])
#print(f.get_feature_names())
np.shape(X)

#take labels    and split dataset
data[v1]=data[v1].map({'gcc':0,'icc':1,'clang':2})
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, data[v1], test_size=0.3
print("Train and test splitting into: ",[np.shape(X_train), np.shape(X_test)])
```

As shown I also got the label vector, and start the training for the 70%, splitting it from the testing (remaining 30%).

3. Implemented classifiers (theory)

A) Optimization prediction problem

Naive Bayes

The NB Classifier has been applied in diverse fields, especially in language processing and prediction problems. The classifier is called “naive”, as it assumes that all features are independent, conditional on the class label.

In this predictive modelling we are interested in modelling a particular process that given the particular extracted feature it distinguishes the optimizations. The target function f(x) = y is the true function f that we want to model based on our chosen features to looking for.

It is probabilistic, which means that they calculate the probability of each tag for a given text, and then output the tag with the highest one.

The class conditional densities can be presented as follows:

$$P(v_j|x, D) = \sum_{h_i \in H} P(v_j|x, h_i)P(h_i|D)$$

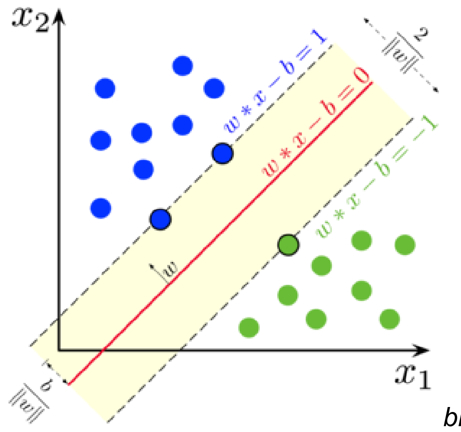
And so the Bayes Optimal Classifier for a new instance x:

$$v_{OB} = \arg\max_{v_j \in V} \sum_{h_i \in H} P(v_j|x, h_i)P(h_i|D)$$

So we have: high = P(H)*P(lea|H)*P(...) ; low = P(L)*P(lea|L)*P(...) and for each instruction compute the argmax(high,low).

Support Vector Machines

Support vector machines are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier.



binary svm

The *linear* SVM classifier works by drawing a straight line between two classes. All the data points that fall on one side of the line will be labeled as one class and all the points that fall on the other side will be labeled as the second.

How do we know which line will do the best job of classifying the data? This is the aim of the algorithm. It will select a line that not only separates the two classes but stays as far away from the closest samples as possible. In fact, the words “support vector” refers to two position vectors drawn from the origin to the points which dictate the decision boundary.

We have to compute the hyperplane equation (***h: w*x + w₀ = 0***) and the distance (margin) between classes d1 and d2 like this:

Given data set D and hyperplane $\bar{h}$, the margin is computed as

$$\min_{n=1,\dots,N} \frac{|y(\mathbf{x}_n)|}{||\mathbf{w}||} = \dots = \frac{1}{||\mathbf{w}||} \min_{n=1,\dots,N} [t_n(\bar{\mathbf{w}}^T \mathbf{x}_n + \bar{w}_0)]$$

using the property $|y(\mathbf{x}_n)| = t_n\, y(\mathbf{x}_n)$

So the hyperplane with maximum margin ***h**** is computed by:

$$\mathbf{w}^*, {w_0}^* = \underset{\mathbf{w}, w_0}{\operatorname{argmax}} \frac{1}{||\mathbf{w}||} \min_{n=1,\dots,N} [t_n(\mathbf{w}^T \mathbf{x}_n + w_0)]$$

So there will be at least 2 closest points (1 for each class).

In the most of machine learning algorithms, it's often used something like gradient descent to minimize the output function, but for support vector machines is implemented the Lagrangian, and in this way is computed ***a***.

$$\mathbf{w}^* = \sum_{n=1}^{N} a_n \, t_n \, \mathbf{x}_n$$

with a >= 0, t and x binary (-1,+1)

In the end the hyperplanes are expressed with support vectors by:

$$y(\mathbf{x}) = \sum_{\mathbf{x}_j \in SV} a_j \, t_j \, \mathbf{x}^T \mathbf{x}_j + w_0 = 0$$

B) Compiler prediction problem

K-nn

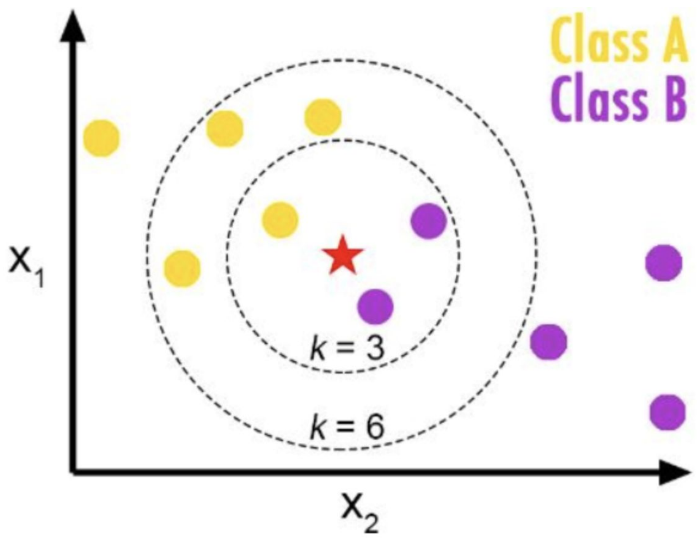
Because of the svm multiclass has the same theoretical structure of the binary version we move on knn:

K-nearest neighbours classifier is one of the simplest classification algorithm: it is a lazy learning model (store the training data and wait until a testing data appear) with local approximation. This classification algorithm does not depend on the structure of the data and it is a non-parametric method used for classification and regression too.

The basic logic behind KNN is to explore your neighborhood, assume the test datapoint to be similar to them and derive the output. So we look for k neighbors and come up with the prediction. Whenever a new example is encountered, its k nearest neighbours from the training data are examined. Distance between two examples can be the euclidean distance between their feature vectors.

The majority class among the k nearest neighbours is taken to be the class for the encountered example.

To understand this consider the following diagram with only two class:



In the diagram yellow and violet points corresponds to Class A and Class B in training data. When k = 3, we predict Class B as the output and when K=6, we predict Class A as the output.

4. Description of the algorithms (code)

Naive Bayes

Unlike what has been done with other classifier, NB was completely implemented by me from the scratch using Python.

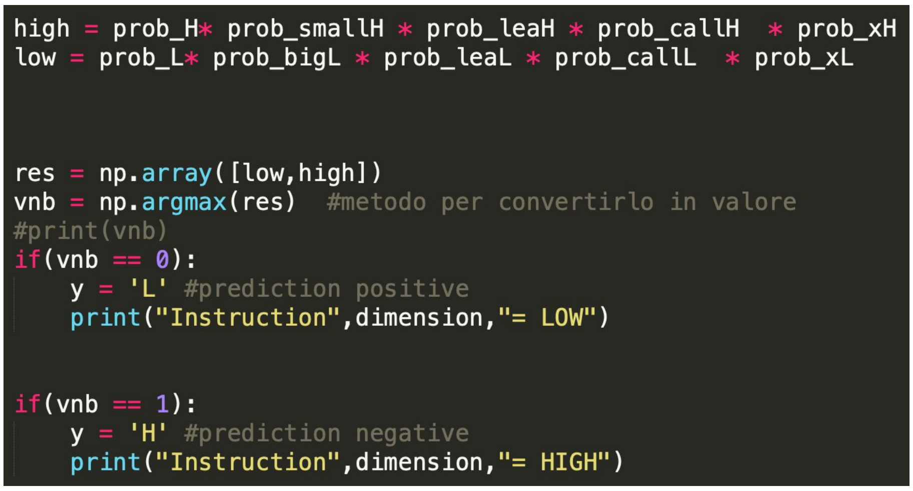
The first part of the algorithm is dedicated to the manipulation of the entire dataset, iterating over the current instructions.

The calculus of the priori probability of opt in the dataset is made up with the **get_opt_probability()** function which gives in output how many specific opt elements are in dataset according to dictionary table comparing hash names and a list of their indexes inside the dataset.

For each word is calculated the probability given a positive detection using **count_appearances()** function which simply compares the feature selected with the set of features of a file, then calculating occurrences and so the probability. The same is done for the probability to get the number of a word (call,lea …) inside a specific optimized instruction.

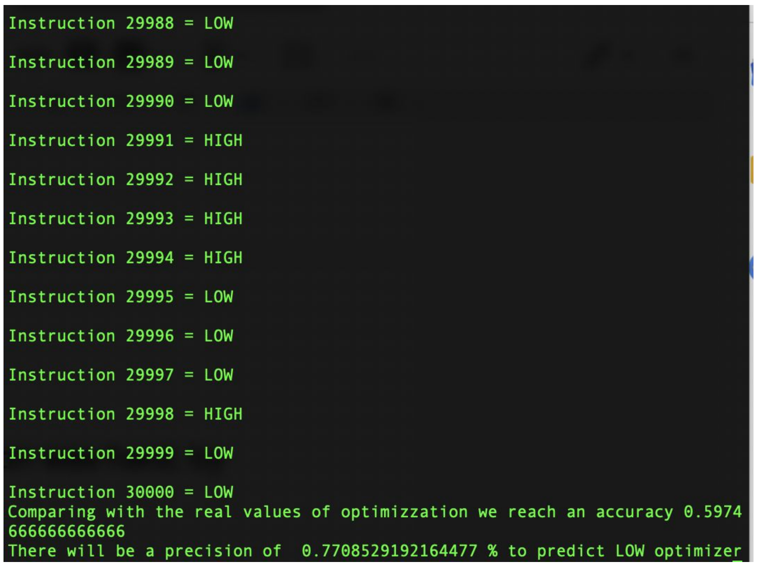
At the end of this process each word has its associated probability given positive or negative sample.

Then the posterior probabilities are computed by the methods **x_probability_counter()** where x is the specific feature; so in according to NB formula I computed the argmax between the found products.



```
high = prob_H* prob_smallH * prob_leaH * prob_callH  * prob_xH
low = prob_L* prob_bigL * prob_leaL * prob_callL  * prob_xL


res = np.array([low,high])
vnb = np.argmax(res)  #metodo per convertirlo in valore
#print(vnb)
if(vnb == 0):
    y = 'L' #prediction positive
    print("Instruction",dimension,"= LOW")


if(vnb == 1):
    y = 'H' #prediction negative
    print("Instruction",dimension,"= HIGH")
```

So I compute in this way the probability to have a High or Low optimization, for each instruction of the dataset. At the end the Accuracy and the Precision are calculated based on n° of opt (H or L) detected over all of them.



```
Instruction 29988 = LOW

Instruction 29989 = LOW

Instruction 29990 = LOW

Instruction 29991 = HIGH

Instruction 29992 = HIGH

Instruction 29993 = HIGH

Instruction 29994 = HIGH

Instruction 29995 = LOW

Instruction 29996 = LOW

Instruction 29997 = LOW

Instruction 29998 = HIGH

Instruction 29999 = LOW

Instruction 30000 = LOW
Comparing with the real values of optimizzation we reach an accuracy 0.5974
666666666666
There will be a precision of  0.7708529192164477 % to predict LOW optimizer
```

Result of the classifier over the train_dataset

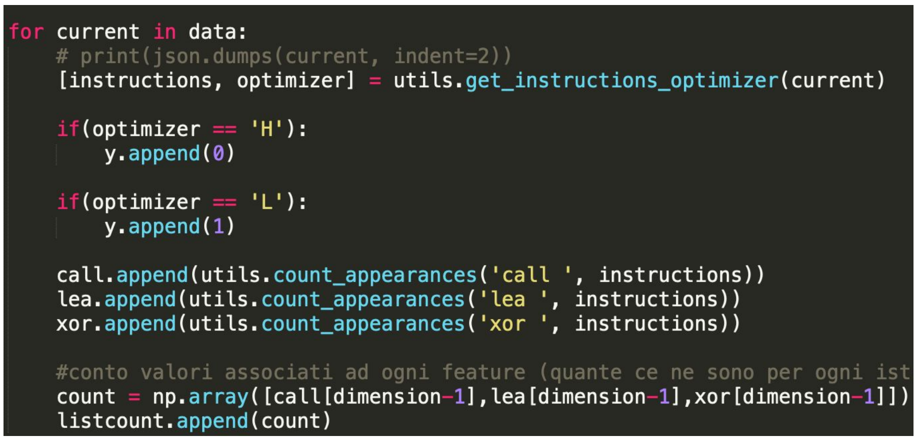
Note: the precision for High optimizer (0.41) and the Recall (0.58) are computed too but they are not shown. Once we have each true_positive, true_negative, false_positive and false_negative it's easy through the related formulas.

Binary SVC

Thanks to the python scikit-learn libraries and tools it was easy implement a support vector classification. The first part of the code is dedicated to the import dataset with the usual instruction **get_instructions_optimizer().**

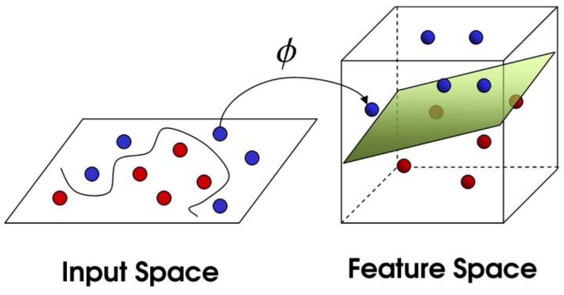
In the below figure I created the feature dictionary, containing each associated value, in terms of number of occurrences with the method **count_appearances().**

Note: the feature extraction is very similar to the naive bayes one, and it is not made with the sklearn libraries, with a relative small loss.

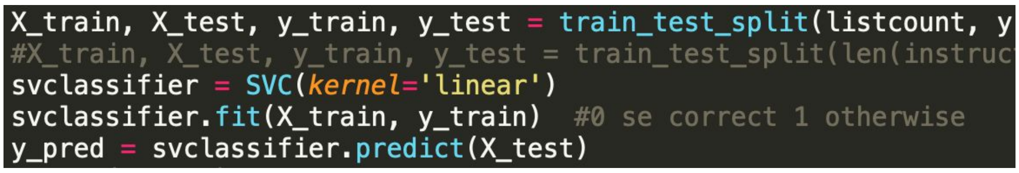


```
for current in data:
    # print(json.dumps(current, indent=2))
    [instructions, optimizer] = utils.get_instructions_optimizer(current)

    if(optimizer == 'H'):
        y.append(0)

    if(optimizer == 'L'):
        y.append(1)

    call.append(utils.count_appearances('call ', instructions))
    lea.append(utils.count_appearances('lea ', instructions))
    xor.append(utils.count_appearances('xor ', instructions))

    #conto valori associati ad ogni feature (quante ce ne sono per ogni ist
    count = np.array([call[dimension-1],lea[dimension-1],xor[dimension-1]])
    listcount.append(count)
```

Then the aim was to transform the input space in a sort of feature space, where the algorithm was able to create the hyperplane and predict the labels.

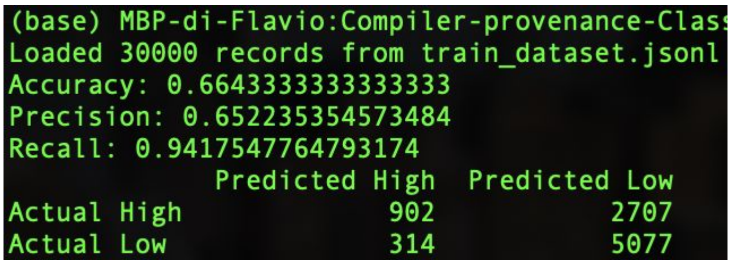


So the feature vector (“listcount”) and the label vector (“y”) are splitted into tests and training vectors, and the sv-classifier is called.



```
X_train, X_test, y_train, y_test = train_test_split(listcount, y
#X_train, X_test, y_train, y_test = train_test_split(len(instruc
svclassifier = SVC(kernel='linear')
svclassifier.fit(X_train, y_train)  #0 se correct 1 otherwise
y_pred = svclassifier.predict(X_test)
```

So I compute in this way the prediction High or Low optimization for each instruction of the dataset. In the end with the final test I compute Accuracy (0.6), the Precision and the Recall. A confusion matrix is shown too.



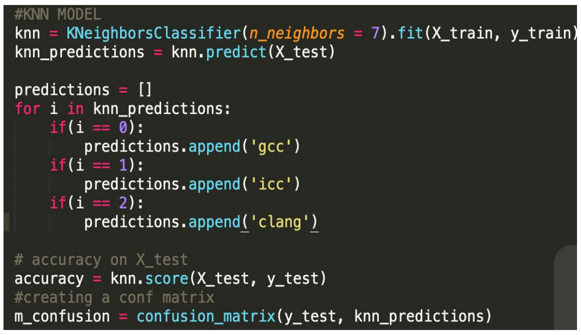
```
(base) MBP-di-Flavio:Compiler-provenance-Clas
Loaded 30000 records from train_dataset.jsonl
Accuracy: 0.6643333333333333
Precision: 0.652235354573484
Recall: 0.9417547764793174
             Predicted High  Predicted Low
Actual High             902           2707
Actual Low              314           5077
```

Result of the classifier over the train_dataset

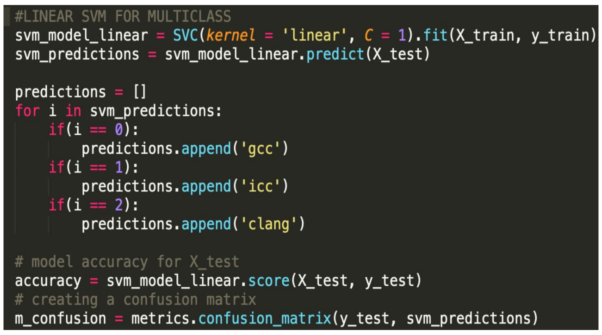
Multiclass methods: SVC and K-nn

For the multiclass problem I upset the previously work with binary predictions, by importing more libraries from sklearn. As we said the feature extraction is different, to guarantee better accuracy for each class prediction; the dataset is imported by the function **utils.conv2mnemonic()** useful to transform it in a sequence of operations (for example <call,mov,mov,lea…>) deleting the variable parts that are no important for our goal.

Both method implementations are very similar:



```
#KNN MODEL
knn = KNeighborsClassifier(n_neighbors = 7).fit(X_train, y_train)
knn_predictions = knn.predict(X_test)

predictions = []
for i in knn_predictions:
    if(i == 0):
        predictions.append('gcc')
    if(i == 1):
        predictions.append('icc')
    if(i == 2):
        predictions.append('clang')

# accuracy on X_test
accuracy = knn.score(X_test, y_test)
#creating a conf matrix
m_confusion = confusion_matrix(y_test, knn_predictions)
```

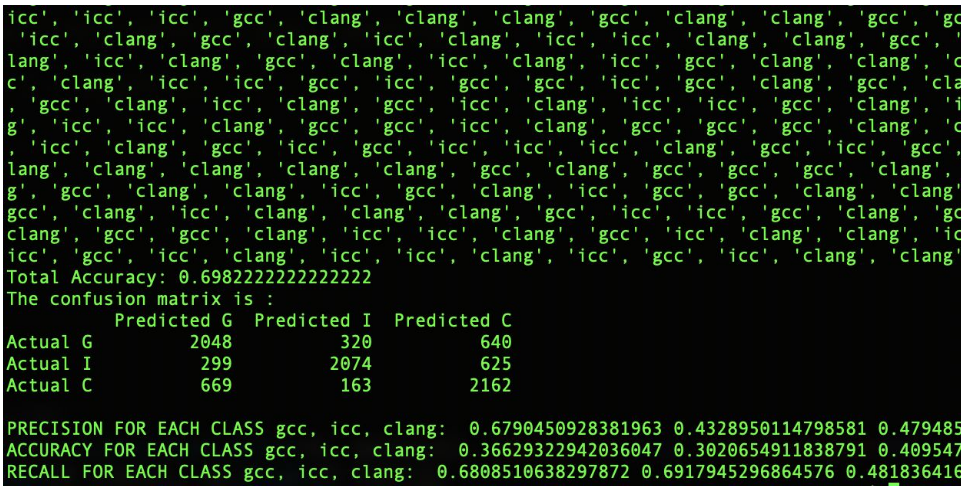


```
#LINEAR SVM FOR MULTICLASS
svm_model_linear = SVC(kernel = 'linear', C = 1).fit(X_train, y_train)
svm_predictions = svm_model_linear.predict(X_test)

predictions = []
for i in svm_predictions:
    if(i == 0):
        predictions.append('gcc')
    if(i == 1):
        predictions.append('icc')
    if(i == 2):
        predictions.append('clang')

# model accuracy for X_test
accuracy = svm_model_linear.score(X_test, y_test)
# creating a confusion matrix
m_confusion = metrics.confusion_matrix(y_test, svm_predictions)
```

They are implemented as above and the predictions are transformed back to string; then the total accuracy and the confusion matrix are computed by metrics library.

In the end we also compute for each class the single accuracy, precision and recall as required, finding formulas values from the confusion matrix.

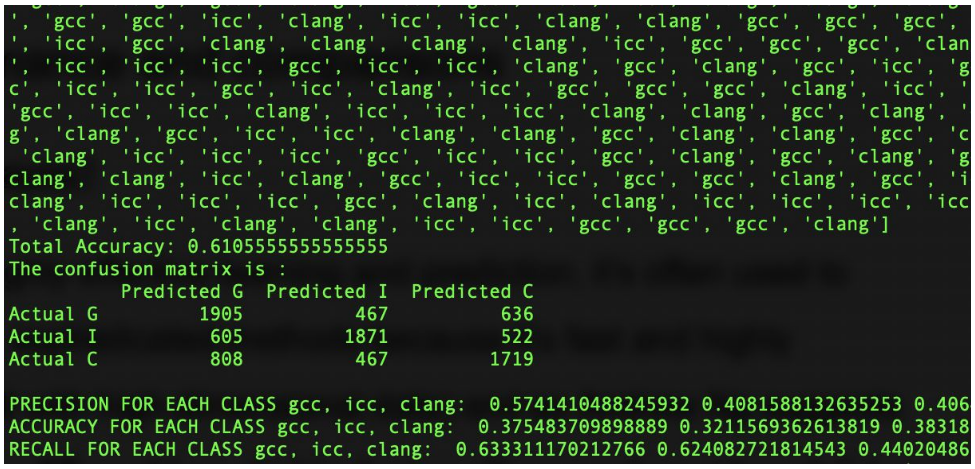
So for SVC classifier I got good values in terms of total accuracy (nearly 0.7) and acceptable results for others:



```
icc', 'icc', 'icc', 'gcc', 'clang', 'clang', 'clang', 'gcc', 'clang', 'clang', 'gcc', 'gc
 'icc', 'clang', 'gcc', 'clang', 'icc', 'clang', 'icc', 'icc', 'clang', 'clang', 'gcc', '
lang', 'icc', 'clang', 'gcc', 'clang', 'icc', 'clang', 'icc', 'gcc', 'clang', 'clang', 'c
c', 'clang', 'icc', 'icc', 'gcc', 'icc', 'gcc', 'gcc', 'icc', 'gcc', 'clang', 'gcc', 'cla
, 'gcc', 'clang', 'icc', 'clang', 'gcc', 'icc', 'clang', 'icc', 'icc', 'gcc', 'clang', 'i
g', 'icc', 'icc', 'clang', 'gcc', 'gcc', 'icc', 'clang', 'gcc', 'gcc', 'gcc', 'clang', 'c
, 'icc', 'clang', 'gcc', 'icc', 'gcc', 'icc', 'icc', 'icc', 'clang', 'gcc', 'icc', 'gcc',
lang', 'clang', 'clang', 'clang', 'clang', 'gcc', 'clang', 'gcc', 'gcc', 'gcc', 'clang',
g', 'gcc', 'clang', 'clang', 'icc', 'gcc', 'clang', 'icc', 'gcc', 'gcc', 'clang', 'clang'
gcc', 'clang', 'icc', 'clang', 'clang', 'clang', 'gcc', 'icc', 'icc', 'gcc', 'clang', 'gc
clang', 'gcc', 'gcc', 'clang', 'icc', 'icc', 'clang', 'gcc', 'icc', 'clang', 'clang', 'ic
icc', 'gcc', 'icc', 'clang', 'icc', 'icc', 'clang', 'icc', 'gcc', 'icc', 'clang', 'clang'
Total Accuracy: 0.6982222222222222
The confusion matrix is :
           Predicted G  Predicted I  Predicted C
Actual G          2048          320          640
Actual I           299         2074          625
Actual C           669          163         2162

PRECISION FOR EACH CLASS gcc, icc, clang:  0.6790450928381963 0.4328950114798581 0.479485
ACCURACY FOR EACH CLASS gcc, icc, clang:  0.36629322942036047 0.3020654911838791 0.409547
RECALL FOR EACH CLASS gcc, icc, clang:  0.6808510638297872 0.6917945296864576 0.481836416
```

SVC output

Good results (but minor) are found for K-nn too, with a total accuracy around 0.6:



```
', 'gcc', 'gcc', 'icc', 'clang', 'icc', 'icc', 'clang', 'clang', 'gcc', 'gcc', 'gcc',
', 'icc', 'gcc', 'clang', 'clang', 'clang', 'clang', 'icc', 'gcc', 'gcc', 'gcc', 'clan
', 'icc', 'icc', 'icc', 'gcc', 'icc', 'icc', 'clang', 'gcc', 'clang', 'gcc', 'icc', 'g
c', 'icc', 'icc', 'gcc', 'icc', 'clang', 'icc', 'gcc', 'gcc', 'gcc', 'clang', 'icc', '
'gcc', 'icc', 'icc', 'clang', 'icc', 'icc', 'icc', 'clang', 'clang', 'gcc', 'clang', '
g', 'clang', 'gcc', 'icc', 'icc', 'clang', 'clang', 'gcc', 'clang', 'clang', 'gcc', 'c
 'clang', 'icc', 'icc', 'icc', 'gcc', 'icc', 'icc', 'gcc', 'clang', 'gcc', 'clang', 'g
clang', 'clang', 'icc', 'clang', 'gcc', 'icc', 'icc', 'gcc', 'gcc', 'clang', 'gcc', 'i
clang', 'icc', 'icc', 'icc', 'gcc', 'clang', 'icc', 'clang', 'icc', 'icc', 'icc', 'icc
, 'clang', 'icc', 'clang', 'clang', 'icc', 'icc', 'gcc', 'gcc', 'gcc', 'clang']
Total Accuracy: 0.6105555555555555
The confusion matrix is :
           Predicted G  Predicted I  Predicted C
Actual G          1905          467          636
Actual I           605         1871          522
Actual C           808          467         1719

PRECISION FOR EACH CLASS gcc, icc, clang:  0.5741410488245932 0.4081588132635253 0.406
ACCURACY FOR EACH CLASS gcc, icc, clang:  0.375483709898889 0.3211569362613819 0.38318
RECALL FOR EACH CLASS gcc, icc, clang:  0.633311170212766 0.624082721814543 0.44020486
```

K-nn output

The confusion matrices show us which there are a lot of predictions that coincide with the real values (respectively over 2000 and 1500 in the major diagonal) and this is the key that indicates the success of the algorithms.

5. Performance and comparisons

Why Naive Bayes?

Naive Bayes has highly effective learning and prediction, it’s often used to compare with more sophisticated methods because it’s fast and highly scalable (works well with high-dimensional data) and as Andrew Ng suggests when dealing with an ML problem start by trying with a simple quick and dirty algorithm and then expand from that point.

As we said Naive Bayes can learn individual features importance but can’t determine the relationship among features. Besides, the training time with Naive Bayes is significantly smaller as opposed to alternative methods and it doesn’t require much training data. In my case a jsonl dataset of 30000 instructions was trained in less than 30 minutes.

Unfortunately the Accuracy found wasn't as good as I thought (0.6%) but with more time available this algorithm is certainly a solid base from which to start improving it, maybe going to improve the Feature Extraction (key of ML).

Support Vector Machine

SVM classifier is often preferred to many similar methods of Machine Learning because it delivers high accuracy results thanks to an optimization procedure. SVM builds a classifier by searching for a separating hyperplane which is optimal and maximises the margin that separates the categories (in our case high and low). Thus, SVM has the advantage of robustness in general and effectiveness when the number of dimensions is greater than the number of samples. It is also versatile: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

Unlike Naive Bayes, SVM is a non-probabilistic algorithm. Comparing the two version for the binary prediction the svm is better for every instance I found: precision, recall, accuracy. The only common feature is the good precision for low predictions and the less precision for the high: this depends obviously from my feature extraction, that is not too accurate.

It was used for multiclass problem too, where I reach the best results of the whole homework.

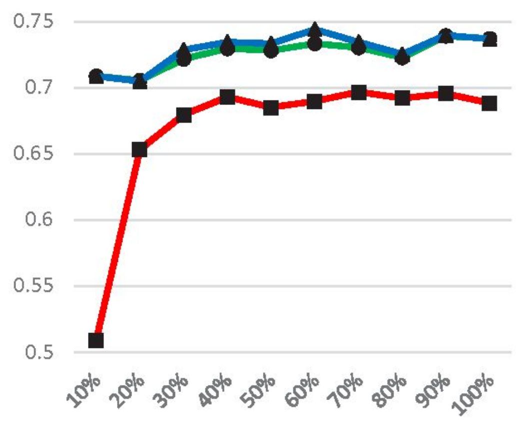
However, the svc algorithm may also present some weak points: (1) if the number of features is much greater than the number of samples, avoid over-fitting in choosing Kernel functions and regularization term is crucial; (2) SVMs do not directly provide probability estimates, these are calculated using an expensive k-fold cross-validation.

K-nn

It is a simple algorithm which make predictions based on k similar neighbors for the output. It is slower than NB and SVM classifiers, due to its real time execution and as we said there are few parameters to tune too (k value and distance function).

Comparing it in our context problem with svm results we can see that values are similar but still smaller. In fact we can say that svm take cares of outliers better than knn: SVM outperforms KNN when there are large features and lesser training data.

In general the trend of the algorithms in terms of accuracy (y axis) over the dataset (x axis) it’s the following:



Where we have svm, k-nn and naive bayes.

6. Test

Given the good accuracy found in the implemented methods, the work done was tested on a **test_dataset_blind.jsonl** file, where for each instruction (3000 in total) there are no labels for optimization and compiler. So in a file csv (attached) is put my prediction for each one label <opt> + <compiler>.

For testing I take binary and multiclass svm classifier: in 2 different file test.py I take into account the dictionary of features of training and with the function **trasform(data)** I compute the input X for the prediction operation of sklearn kit.

<u>Note</u>: no more **fit_trasnform(data)** is used.

7. Conclusion

Thanks to this homework, I was able to experience many learning methods studied in class related to linear classification for binary and multiclass approaches. In this way I managed to learn a little better about how the ML world works.

I saw mechanisms and the construction of a naive bayes classifier (completely implemented by me) and of a support vector classifier (by scikit-learn) that have been tested with acceptable results on a json list dataset of instructions (assembly) with the aim to predict the optimization.

The same it was been with the compiler prediction with a multiclass approach where I used the sklearn libraries for implementation and feature extraction too, obtaining better results with svm and k-nearest neighbors algorithms.

So we can say that prediction of two or more classes in Machine Learning field (or in general the *classification*) is a very useful task not only for AI, but it can be extended in many domains, for example: credit approval, medical diagnosis, target marketing, malware classification and filtering (…), which makes it a very important tool.