BOĞAZİÇİ UNIVERSITY

CMPE 561

Natural Language Processing

Application Project Documentation

19.12.2017

Language Identification System

Submitted To:
Prof. Dr. Tunga Güngör
Computer Engineering

Submitted By :
Taha Küçükkatırcı
Yaşar Fatih Enes Yalçın

Contents

1 Introduction

The increasing size of the textual data in last decades caused some new problems to be handled and this made Natural Language Processing researchers focus on these problems. One of the problems is the language identification problem. There are lots of textual data all around the world, especially on the internet and they are in all languages of the world. Guessing and determining the language of the given content is usually the first task to do before processing the given language.

Language identification is a classification problem. There are several statistical approaches to classify the data. With these approaches, computer scientists both classify the given data and identify similar languages like Serbian and Croatian with help of their structural and lexical overlap. There are increasing number of datasets available on the internet now.

In this project, we are asked to implement a language identification system with two approaches. These models are Naive Bayes model which is a generative model and another model using Support Vector Machine (SVM) which is a discriminative model.

There is a corpus given to us to use in this project. The corpus is used both for training and testing. So, it is splitted into training set and test set during the execution. In the corpus, there are 13 languages and 2,000 sentences for each language. The languages involved are as follows:

- Bulgarian (bg)
- Bosnian (bs)
- Czech (cz)
- Argentine Spanish (es_AR)
- Peninsular Spanish (es_ES)
- Croatian (hr)
- Indonesian (id)
- Macedonian (mk)
- Malay (my)
- Brazilian Portuguese (pt-BR)
- European Portuguese (pt-PT)
- Slovak (sk)
- Serbian (sr)

2 Program Interface

The program works on command-line interface like Terminal app. So that there is not much to say about the interface. The input and output format on the Terminal is mentioned in the *Input and Output* section in this document.

The command to run the program is like the following:

```
$ python3 main.py [arguments]
```

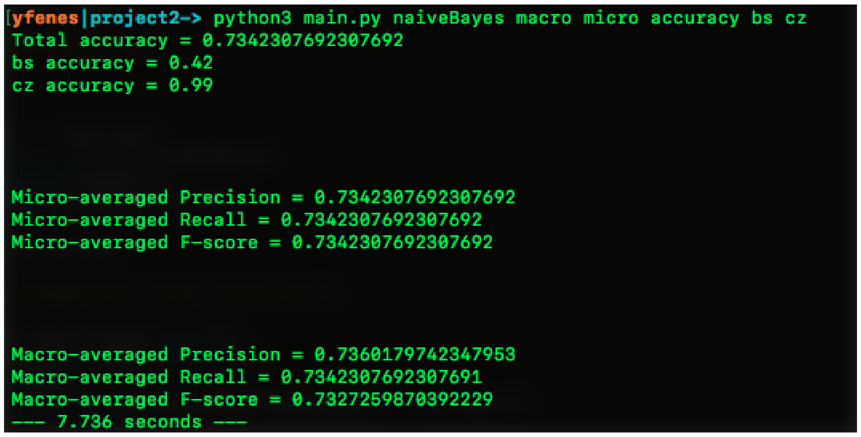
Of course, one should be in the correct directory to run the code with this command. The parameters are explained in the *Input and Output* section in detail.

3 Program Execution

To run the program, the computer should have Python3 installed in it. The program assumes that the corpus is named *corpus_new.txt* and it is 13 languages with 2000 lines for each language. The corpus must be in UTF-8 format. The program terminates itself after each run. To run the code again, one must run the *main.py* file again.

The output of the program is standard outputs on the command-line and an output file named *output.txt* which contains language estimations of each line in the test dataset with their probabilities if the SVM method is used. When the Naive Bayes method is used, the output is simply printed in the console.

Here is a screen output for an example run:



```
[yfenes|project2-> python3 main.py naiveBayes macro micro accuracy bs cz
Total accuracy = 0.7342307692307692
bs accuracy = 0.42
cz accuracy = 0.99



Micro-averaged Precision = 0.7342307692307692
Micro-averaged Recall = 0.7342307692307692
Micro-averaged F-score = 0.7342307692307692



Macro-averaged Precision = 0.7360179742347953
Macro-averaged Recall = 0.7342307692307691
Macro-averaged F-score = 0.7327259870392229
--- 7.736 seconds ---
```

4 Input and Output

Input format is basically running the main Python code with arguments. Usage of the arguments is explained below. These arguments are:

1. naiveBayes
   (a) accuracy
   (b) language abbreviation (e.g bg)
   (c) micro
   (d) macro
2. svm
3. svm_bonus

The *naiveBayes* argument basically solves the problem with Naive Bayes method. The arguments *accuracy*,*language abbreviation*, *micro* and *macro* can be used with the *naiveBayes* method.

The *accuracy* argument stands to calculate the general accuracy of the system, i.e the ratio of correct predicted sentences to all sentences in the test set.

The *language abbreviation* argument is given to the program to calculate the accuracy for a specific language.

The *micro* argument is for calculating the micro-averaged precision, recall and F-measure which gives equal weight to each sentence and is therefore considered as an average over all the sentence/language(class) pairs.

The *macro* argument is for calculating the macro-averaged precision, recall, F-measure which gives equal weight to each language (class), regardless of number of sentences in that class.

When there is *naiveBayes* argument, the arguments *accuracy*, *micro* and *macro* can be used together in any order.

Example:

```
$ python3 main.py naiveBayes accuracy
```

```
Total accuracy = 0.8543341538461538
```

Example:

```
$ python3 main.py naiveBayes accuracy micro

Total accuracy = 0.7338461538461538

Micro-averaged Precision = 0.7338461538461538
Micro-averaged Recall = 0.7338461538461538
Micro-averaged F-score = 0.7338461538461538
```

The *svm* argument basically solves the problem with discriminative method using Support Vector Machine.

Example:

```
$ python3 main.py svm

Reading training examples... (23400 examples) done
Training set properties: 268 features, 13 classes
Iter 1: .........*(NumConst=1, SV=1, CEps=100.0000, QPEps=0.0000)
Iter 2: *(NumConst=2, SV=1, CEps=82.4723, QPEps=0.0000)
Iter 3: .........*(NumConst=3, SV=2, CEps=17.6683, QPEps=8.1459)
Iter 4: *(NumConst=4, SV=3, CEps=8.1725, QPEps=0.2503)
Iter 5: .........*(NumConst=5, SV=3, CEps=1.8815, QPEps=0.0000)
Iter 6: *(NumConst=6, SV=3, CEps=0.4218, QPEps=0.0973)
Iter 7: *(NumConst=7, SV=4, CEps=0.2780, QPEps=0.0000)
Iter 8: .........*(NumConst=8, SV=5, CEps=0.9103, QPEps=0.0767)
Iter 9: *(NumConst=9, SV=6, CEps=0.1901, QPEps=0.0610)
Iter 10: *(NumConst=10, SV=7, CEps=0.1033, QPEps=0.0386)
Iter 11: .........*(NumConst=11, SV=8, CEps=0.2955, QPEps=0.0415)
Iter 12: .........*(NumConst=12, SV=9, CEps=0.1662, QPEps=0.0346)
Iter 13: .........*(NumConst=13, SV=10, CEps=0.1461, QPEps=0.0000)
Iter 14: .........(NumConst=13, SV=10, CEps=0.0973, QPEps=0.0000)
Final epsilon on KKT-Conditions: 0.09727
Upper bound on duality gap: 0.09727
Dual objective value: dval=99.94707
Primal objective value: pval=100.04434
Total number of constraints in final working set: 13 (of 13)
Number of iterations: 14
Number of calls to 'find_most_violated_constraint': 187200
Number of SV: 10
Norm of weight vector: |w|=0.32536
Value of slack variable (on working set): xi=99.89414
Value of slack variable (global): xi=99.99141
Norm of longest difference vector:
||Psi(x,y)-Psi(x,ybar)||=4.18661
Runtime in cpu-seconds: 2.30
Final number of constraints in cache: 115338
```

```
Compacting linear model...done
Writing learned model...done
Reading model...done.
Reading test examples... (2600 examples) done.
Classifying test examples...done
Runtime (without IO) in cpu-seconds: 0.02
Average loss on test set: 60.9615
Zero/one-error on test set: 60.96% (1015 correct, 1585
incorrect, 2600 total)
```

The *svm_bonus* argument basically solves the problem with discriminative method using Support Vector Machine but with extra feature which is character bigrams.
(Only the first and the last two lines of output is shown to keep it simple.)

```
$ python3 main.py svm_bonus

Reading training examples... (23400 examples) done
Training set properties: 72092 features, 13 classes
...
...
...
Average loss on test set: 33.7308
Zero/one-error on test set: 33.73% (1723 correct, 877
incorrect, 2600 total)
```

5 Program Structure

In the program structure, there is a main method which is basically takes the arguments and calls corresponding methods. These arguments are *naiveBayes*, *accuracy*,*language abbreviation*, *micro*, *macro*, *svm* and *svm_bonus*. The program first takes the corpus named *corpus_new.txt* and splits it into test and train sets randomly, for each language, the %90 of the sentences is in the training set and 10% is in the test set.

After splitting the test and training set, program splits each line to 2 parts with help of *tab* character just before the language abbreviation part at the end of each line.

5.1 Training

To train the each model, we send training set named $train_set$ to the method named *train*. This method basically takes each line of the training dataset. For each line, it splits sentence and language. For each sentence in each line, the method counts the each character. Meanwhile, there is a dictionary data structure named *lang*$_c$*har*$_d$*ict*. At last, the counted numbers of characters for each language is pushed to the corresponding language.

For example, for the Peninsular Spanish, it would look like:

```
lang_char_dict['es-ES'] = {'e': 62288, 'a':
58698, 'o': 39712, 's': 34308, 'n': 34301, 'r':
31711, 'i': 30285, 'l': 27350, 'd': 25492, 't':
21434, 'c': 20717, 'u': 18870, 'p': 12410, 'm':
11876, ',': 6359, 'b': 5216, 'g': 4971, 'q':
4651, 'v': 4381, '.': 4063, ' ': 4049, 'y': 3936,
'h': 3784, 'f': 3268, ' ': 2302, ' ': 2191, 'j':
1873, 'z': 1824, 'E': 1703, '0': 1391, 'A': 1305
...
...
...}
```

The counted characters goes on. We are not showing all of them to keep the report simple. After counting the characters for each language, the training is done.

5.2 Prediction

5.2.1 Naive Bayes Method

To make a prediction with the Naive Bayes method, we use the training dictionary named *train_dict*. For each sentence in the test set we need to make a prediction. These predictions will be in the in the $pred_y$ list.

To fill the *pred_y* list, we make a probability prediction for each language. These predictions are done in the method named *P_char_given_lang*. It takes these arguments:

1. characters list of the sentence
2. given language
3. training dictionary

In the *P_char_given_lang* method, we initialize a probability variable to zero. We calculate the probability for each language with help of this formula:

$$prediction = \sum_{i=1}^{\#ofchars} \log \frac{train\_dict[given\_lang][i^{th} char] + 1}{V + N} \tag{1}$$

V is the number of unique characters in the given language.
N is the number of all characters in the training set for the given language.
The $+1$ in the nominator and V in the denominator come from the Laplace smoothing.

For a sentence, after calculating probabilities for each language, we take language with the maximum probability as the prediction for the sentence.

5.2.2 Support Vector Machine

To train and test the system with SVMs, we use a library provided by Cornell University. According to the format of this library, we assign an id to each language and each unique character in the corpus. In the bonus part, character bigrams are also assigned by an id as well. Then each sentence in the training set and test set will be prepared to be written into files *train.txt* and *test.txt* respectively.

Then the system is trained via following command:

```
$ svm_multiclass_learn -c 1.0 train.txt model.txt
```

and then tested via following:

```
$ svm_multiclass_classify test.txt model.txt output.txt
```

The model is saved into *model.txt* and the predictions are saved into *output.txt*.

5.3 Evaluation

We are not explaining the evaluation metrics to keep the report as simple as possible and because they are well-explained in the project description. To take a look at the description, one can click this sentence, the sentence is a link to the description in Google Drive.

6 Examples

Examples are created with the given corpus which is *Discriminating between Similar Languages (DSL) Shared Task 2015 corpus.* There are examples for each method below:

Example: Generative Modeling

```
$ python3 main.py naiveBayes accuracy
Total accuracy = 0.7338461538461538
```

Example: Generative Modeling

```
$ python3 main.py naiveBayes accuracy bs cz
Total accuracy = 0.7338461538461538
bs accuracy = 0.395
cz accuracy = 0.985
```

Example: Generative Modeling

```
$ python3 main.py naiveBayes macro accuracy
Total accuracy = 0.7338461538461538

Macro-averaged Precision = 0.7360016431935276
Macro-averaged Recall = 0.7338461538461539
Macro-averaged F-score = 0.7321622941026569
```

Example: Discriminative Modeling
(Only the last two lines of output is shown to keep it simple.)

```
$ python3 main.py svm
Average loss on test set: 43.8846
Zero/one-error on test set: 43.88% (1459 correct, 1141
incorrect, 2600 total)
```

Example: Discriminative Modeling
(Only the last two lines of output is shown to keep it simple.)

```
$ python3 main.py svm_bonus
Average loss on test set: 35.6538
Zero/one-error on test set: 35.65% (1673 correct, 927
incorrect, 2600 total)
```

7 Improvements and Extensions

There are lots of aspects of the project that can be improved or extended. First, the input format is prefix, total number of lines are equal to 26000 and these lines are from 13 languages with 2000 lines for each language. The code is highly dependent on the given corpus. It can be more generic. The methods can be somehow merged. One can use the better parts of the each method.

Actually, this methods can be applied to most of the sequential statistical classification problems. Hence, the problem can be extended into another sequential problem. For example, the lines which contains sentences in different languages can be replaced with the RGB values of images. If each line were an image tagged at the end with apple or orange, we could train it and extend the problem to a new classification problem which distinguishes apples and oranges.

8 Difficulties Encountered

Comparing to the first application project in the course, frankly, this was a sleeker assignment. It was clearer and the tools were well documented. We only had difficulties to come together to code together. So that we had to code separately which causes differences between us, partners, in comprehension of some parts of the code. Other than that almost anything went well about the project.

9 Conclusion

In a nutshell, this was a beneficial project to understand the working mechanism of different approaches to the same problem. And implementing different models was good for us because in these days, almost all the times, we use libraries for implementing the methods. Which is a good thing but to really learn a topic, one must implement the most basic methods for at least one time. This assignment pushed us to implement these widely used models so it will be helpful for us when we are going to use these models again.

Also, since we know have a better idea and intuition about the field, we now feel more comfortable to extend our knowledge and implementation skills to new problems. Besides, it was interesting for us to implement this kind of problem from scratch.