# Artificial Intelligence: Dialect Identification

Stoleru Vlad-Stefan, 244

## Summary

From Native Bayes to Support Vector Machines, this project carried me through the entire scikit-learn API, from Basic SVM implementations (*SVC*), to linear-kernel models (*LinearSVC*, *LogisticRegression*, *Perceptron*, *RidgeClassifier*, *SGDClassifier*), to merging regression algorithms (*LinearRegression*, *Ridge*, *KernelRidge*) with value rounding, and finally to the Naïve Bayes class, in which the winner was, as stated, the *ComplementNB* model.

The features are extracted using the Bag of Words (BoW) principle, implemented via the *CountVectorizer* class, then transformed using *TfidfTransformer* (term-frequency, inverse-dictionary-frequency transformation) to get the train_features for the Naïve Bayes machine.

In the end, the F1 Macro Score is computed on the predicted classes the pre-existent scikitLearn method (*f1_score*); the scoring function is also the one used in the Kaggle Competition whose task this algorithm is aiming to solve.

## The algorithm

The training input is made of 7577 article-samples containing 1 sentence each, and their corresponding labels: 0 for Moldavian dialect, 1 for Romanian dialect. We are also provided with a validation s composed of 2656 samples representing tweets, their structure being the same as the training samples', and their respective labels. Finally, the test dataset is composed of 2623 tweets that we are tasked to determine in which dialect class they would fit.

After extracting the samples from the datasets and separating them from their identification codes (which will only be used in the end), we are parsing the data through a *CountVectorizer*[1], which reproduces the *Bag of Words* (BOW) principle:

- Using an internal *analyser*, identify the wanted features within the training samples and construct a *vocabulary*.
- Iterate over the train, validation and test datasets and memorise the frequencies of each feature within each sample (sentence) of the sets.

As a result, we will get 3 new matrixes of shape *[n_samples, n_features]* representing the sample-wise frequencies of each feature of the vocabulary we've just built.

The parameters set for the *CountVectorizer* are the *analyser*, which is set to *'char_wb'* indicating we are intending to use the characters near the word boundaries in our vocabulary – this is especially useful in our context giving the state of the data samples: encrypted – and the *ngram_range*; n-grams are contiguous sequences of items from a given sample of text – in our context, sequences of characters around the word boundaries – used to increase a classifier's understanding of a sample's context (and to better predict it's class) [2]; the algorithm uses a *ngram_range* of *(4, 7)*, because lower (sized) sequences were not viable to the classifier and upper (sized) sequences were reducing the over-all accuracy score.

The next step after extracting the features is performing a *Tf-Idf Transformation* over them – we will be using the built-in *TfidfTransformer* from scikitLearn. From the scikitLearn' documentation on the subject:

*"The goal of using tf-idf instead of the raw frequencies of occurrence of a token in a given document is to scale down the impact of tokens that occur very frequently in a given corpus and that are hence empirically less informative than features that occur in a small fraction of the training corpus."*[3]

Tf (term frequency) and idf (inverse document frequency) are two measures that relate to the features we extracted at the last step. The first one is exactly our computed value for document (in our context sentence-sample) $\boldsymbol{d}$ and term (feature: in our context ngram) $\boldsymbol{t}$: $\boldsymbol{df(d,t)}$, and the latter is computed as $idf(t) = \log\frac{n}{df(t)} + 1$, where $\boldsymbol{t}$ is the term, $\boldsymbol{n}$ is the total number of documents and $\boldsymbol{df(t)}$ is the overall frequency of term $\boldsymbol{t}$ in the $\boldsymbol{n}$ documents. The final formula is $tfidf(t,d) = tf(t,d) \cdot idf(t)$.

After transforming our features into tfidf-weighted matrixes of the same shape, we are ready to feed the data to a *Classifier* model that will learn our data and will attempt to guess (predict) the classes (labels) for the validation and the test features.

The Classifier Model used is a *Naïve Bayes* model within the scikitLearn library, the *ComplementNB*. Naïve Bayes is referring to Bayes' theorem from the Probability Theory:

$$P(y|x_1, \dots, x_n) = \frac{P(y)P(x_1,\dots,x_n|y)}{P(x_1,\dots,x_n)},$$

using the naïve conditional independence assumption that

$$P(x_i|y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i|y)$$

for all $\boldsymbol{i}$, simplifying the initial Bayes relation to

$$P(y|x_1, \dots, x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i|y)}{P(x_1,\dots,x_n)},$$

and because we know the nominator is constant given the input data, we can then state that

$$\hat{y} = \arg\max_y P(y)\prod_{i=1}^{n} P(x_i|y).$$

The justification behind using Naïve Bayes models is given by the fast-determination of the sample class compared to other methods I've previously tried (a few seconds compared to minutes, sometimes hours), and how easy is to implement and use one of them in text classification (I've used a similar model before to classify spam from non-spam emails). *ComplementNB* is used in place of more common known Bayesian models (such as *MultinomialNB*) because it tends to give more accurate estimates of the probabilities listed above, in comparation to the rough estimates given by MNB, and there are several studies (starting from the inventors of CNB themselves) that show how it outperforms other Naïve Bayes models. From the scikitLearn page:

*"ComplementNB implements the complement naive Bayes (CNB) algorithm. CNB is an adaptation of the standard multinomial naive Bayes (MNB) algorithm that is particularly suited for imbalanced data sets. Specifically, CNB uses statistics from the complement of each class to compute the model's weights. The inventors of CNB show empirically that the parameter estimates for CNB are more stable than those for MNB. Further, CNB regularly outperforms MNB (often by a considerable margin) on text classification tasks. The procedure for calculating the weights is as follows:*

$$\hat{\theta}_{ci} = \frac{\alpha_i + \sum_{j:y_j \neq c} d_{ij}}{\alpha + \sum_{j:y_j \neq c}\sum_k d_{kj}}$$
$$w_{ci} = \log \hat{\theta}_{ci}$$
$$w_{ci} = \frac{w_{ci}}{\sum_j |w_{cj}|}$$

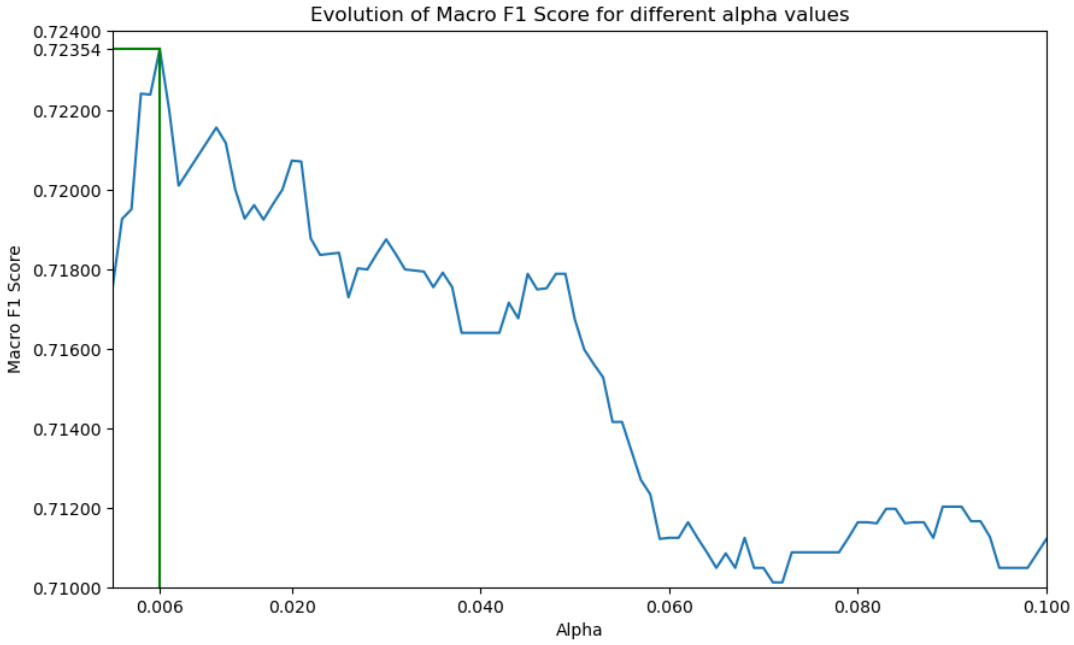
*where the summations are over all documents* $\boldsymbol{j}$ *not in class* $\boldsymbol{c}$, $\boldsymbol{d_{ij}}$ *is either the count or tf-idf value of term* $\boldsymbol{i}$ *in document* $\boldsymbol{j}$, $\boldsymbol{\alpha_i}$ *is a smoothing hyperparameter like that found in MNB, and* $\boldsymbol{\alpha} = \sum_i \boldsymbol{\alpha_i}$.

*The second normalization addresses the tendency for longer documents to dominate parameter estimates in MNB. The classification rule is:*

$$\hat{c} = \arg\min_{c} \sum_{i} t_i w_{ci}$$

*i.e., a document is assigned to the class that is the poorest complement match."* [4]

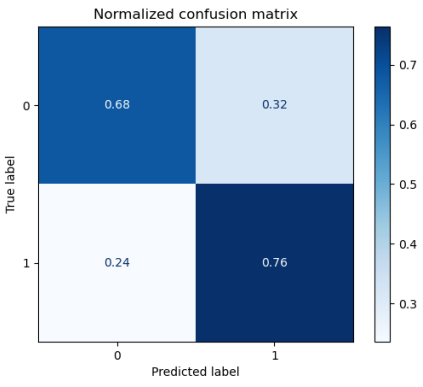
The *alpha* parameter used for our task was found through a *GridSearchCV* parameter-based search, which yielded the value **0.006** (or *6e-3* in scientific notation). The evolution of the final f1 score of the model as a function of alpha can be seen in the following graph:

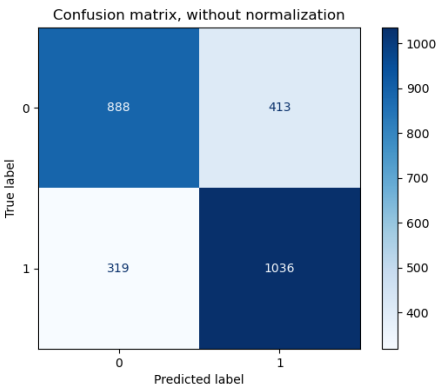



Using the *alpha* value of **0.006** and the features extracted from our input datasets, the model is able to obtain the following accuracy and f1 score on the validation dataset, as well as the following confusion matrix:

Accuracy: 0.72439

F1 Macro Score: 0.72354



### *The Python code of the algorithm*

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import f1_score, accuracy_score, plot_confusion_matrix
from sklearn.naive_bayes import ComplementNB
#Data reading
train_data_raw = np.genfromtxt('data/train_samples.txt', encoding='utf-8',
delimiter='\t', dtype=None, names=('id', 'text'), comments=None)
train_labels_raw = np.genfromtxt('data/train_labels.txt', dtype=None, names=('id',
'label'))
validation_data_raw = np.genfromtxt('data/validation_samples.txt', encoding='utf-
8', delimiter='\t', dtype=None, names=('id', 'text'), comments=None)
validation_labels_raw = np.genfromtxt('data/validation_labels.txt', dtype=None,
names=('id', 'label'))
test_data_raw = np.genfromtxt('data/test_samples.txt', encoding='utf-8',
delimiter='\t', dtype=None, names=('id', 'text'), comments=None)

#Data splitting
train_data = train_data_raw['text']
train_labels = train_labels_raw['label']
validation_data = validation_data_raw['text']
validation_labels = validation_labels_raw['label']
test_data = test_data_raw['text']
test_ids = test_data_raw['id']

#BoW creation an feature extraction, followed by Tfidf transformation
Vectorizer = TfidfVectorizer(lowercase=False, analyzer='char_wb', ngram_range=(4,
7))
train_features = Vectorizer.fit_transform(train_data, train_labels)
validation_features = Vectorizer.transform(validation_data)
test_features = Vectorizer.transform(test_data)

#Classification
Classifier = ComplementNB(6e-3)
np.savetxt('params.txt', fmt='%s', X=[Classifier.get_params()])
Classifier.fit(train_features, train_labels)
validation_predictions = Classifier.predict(validation_features)
test_predictions = Classifier.predict(test_features)

#Calculating validation data metrics
validation_accuracy = accuracy_score(validation_labels, validation_predictions)
np.savetxt('results/accuracy_nb.txt', [validation_accuracy], fmt='%s',
header='accuracy', comments='')
print('Accuracy:')
print('Source:' + str(validation_accuracy))
validation_score = f1_score(validation_labels, validation_predictions,
average='macro')
np.savetxt('results/score_nb.txt', [validation_score], fmt='%s', header='macro f1
score', comments='')
print('Macro F1 score:')
print('Source:' + str(validation_score))

#Generating test submission
submission = np.vstack((test_ids, test_predictions)).T
np.savetxt('submission_nb.csv', submission, fmt='%s', delimiter=',',
header='id,label', comments='')
```

## Bibliography


1. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html
2. https://en.wikipedia.org/wiki/N-gram#n-gram_models, last edited on 18 March 2020, at 17:43 (UTC).
3. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html;
4. https://scikit-learn.org/stable/modules/naive_bayes.html; Rennie, J. D., Shih, L., Teevan, J., & Karger, D. R. (2003). Tackling the poor assumptions of Naïve Bayes text classifiers. In ICML (Vol. 3, pp. 616-623).
   H. Zhang (2004). The optimality of Naïve Bayes. Proc. FLAIRS