# **Sorting documents based on language**

## Evan Kvidera & Uzma Ghazanfar

# Table of Contents

[**Sorting documents based on language**](#_9obf6x7cy0hr) **1**

[Evan Kvidera & Uzma Ghazanfar](#_1hw5ipkudaba) 1

[**Table of Contents**](#_ggzyfqxukz46) **2**

[Introduction](#_haw3blg4y22q) 3

[Algorithm description](#_wlj1j8riwvny) 3

[Feature Extraction using Count Vectorizer](#_n7nh7ncjy0zb) 3

[The Multinomial Naïve Bayes Classifier](#_5hch8plj4maq) 4

[User’s manual](#_6ep1j8vmnttc) 5

[Example problems](#_b5gju7srqxj) 5

[Computer code description](#_8kr2ixd4v2mf) 6

[Dataset](#_hd8ir53t9wuu) 6

[The Count-Vectorizer and the Multinomial Naive Bayes classifier](#_xrfia9hn3mj8) 7

[The Classification](#_ks3wa8502su0) 7

[Confusion and scores](#_a5hqm3pyy5br) 8

[Comparison](#_k6b1293gcknu) 9

[References](#_ogt98hjb1fj) 9

# 

## **Introduction**

With 56% of the people in the world today being bilingual, there is always going to be a large number of data in different languages that you might need to be able to categorize for later use. This can be the sorting of technical documents written by different scientists in their language, homework assignments written by students in bilingual schools in their preferred language, or the sorting of documents and texts in a library setting for classification. The reasons for creating a classification software are abundant and one of the main reasons is because to manually classify documents would take hours upon hours, depending on how many documents you had to classify.

To achieve this, we created a 2-step application. The first step (which we do in the back-end) is to train the system using the Multinomial Naïve Bayes algorithm which then assigns a target class to the different text files used for training. In the next step, the module uses the training to classify and sort documents into different folders based on the attributes and word counts.

The software we designed currently sorts documents by categorizing them into either English or Spanish, however, this can very easily be extended to take up another languages or even sort based on different attributes (other than language). The algorithm is versatile enough to be easily modified and enhanced into then sorting based on context.

## **Algorithm description**

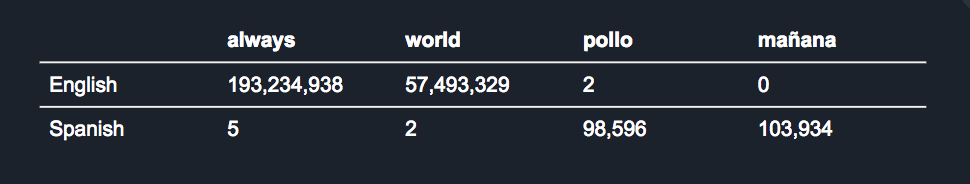
The algorithm has a two components that it needs to sort the documents:

1. Feature Extraction
2. Multinomial Naïve Bayes Classifier

We will be discussing these in detail.

### Feature Extraction using Count Vectorizer

Before the system can be trained to classify documents, we need to be able to extract features from it, i.e. we need to create a uniform set of attributes that the algorithm can learn from (they expect numerical features to learn from). For our problem, this means word-counts. Here is what our feature extraction would look like:

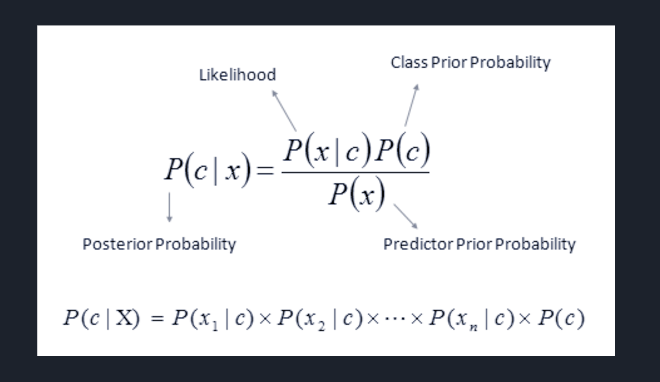


Essentially, we are produce a table of each word mentioned in the training documents and its corresponding frequency of appearance for each class of documents.

We do this using sklearn’s feature\_extraction module which contains the count\_vectorizer. The count\_vectorizer tokenizes each word in the string it is passed and then the count\_vectorizer calculates the individual token occurrence frequency. This technique is also called the “Bag of Words” technique. These become the features vector that the algorithm will then learn from. Now that we have the features and counts, we can move on and train the system.

### The Multinomial Naïve Bayes Classifier

For the classifier, we used the sklearn’s package called MultinomialNB,. The multinomial Naïve Bayes classifier is suitable for classifications that have discrete features. It decides class membership of a document by using the Bayes theorem with Naïve independence assumptions and assigns documents to the class with the most likeliest probability of it belonging to. Hence, each feature (in this case word counts) is independent from every other one and each one contributes to the probability that an example belongs to a particular class.

The classifier then predicted what sort of document one is, and based on that prediction, moves the file to a different location. Bayes theorem provides a way of calculating the posterior probability, P(c|x), from P(c), P(x), and P(x|c). Naive Bayes classifier assume that the effect of the value of a predictor (x) on a given class (c) is independent of the values of other predictors. This assumption is called class conditional independence.   
1. P(c|x) is the posterior probability of class (target) given predictor (attribute). 

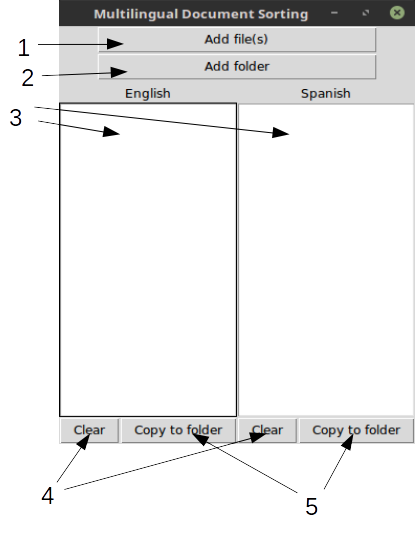
2. P(c) is the prior probability of class.

3. P(x|c) is the likelihood which is the probability of predictor given class.

4. P(x) is the prior probability of predictor.

The techniques we used are great for something that just requires us to tell the difference between the languages used in different documents. If we wanted to scale our project further, we could use the n-gram counts in the vectorizer instead of just the counts and this could help us read the context of our documents as well. If we wanted to, say, classify documents depending on a particular subject, we could use n-gram vectorizing instead of just counting the words. The n-gram vectorizing doesn’t just count the words in the documents, but gleans the context that is gained from the ordering of the words in a phrase.

## **User’s manual**

1. Add file(s) – Click to open a dialog that lets you select one or more text files for sorting. After selection, the files will be automatically sorted into the correct language panes.

2. Add folder – Add and sort all text files contained within the selected folder.

3. Language panes – Displays lists of files whose contents are in English or Spanish, respective

4. Clear – Click to clear all selected files in the above language pane

5. Copy to folder – Click to open a dialog that lets you select a folder to copy all files in the above language pane to.

## **Example problems**

We imagined a scenario where there is a teacher who teaches in a bilingual school and who has submissions from students in either Spanish or English. The professor has two teacher’s assistants, one who can grade in English and one who can grade in Spanish. He or she wants to be able to split the submissions into either Spanish or English without having to open each document and looking at its contents.

Another scenario that will be addressed by our software is if a scientist has a collection of documents written by other scientists in their language, that he/she might need to sort. It could also be used by a large corporation in routing their customer support emails to the technician who is best capable of answering their questions. Lastly, it can be used by library staffers to sort through multilingual articles in a quick and efficient manner, saving the documents in different folders.

## **Computer code description**

### Dataset

To begin, we first needed datasets to train the system against. We used the raw data in the form of a collection wikipedia documents gathered by [Ferrero Jeremy](https://github.com/FerreroJeremy/Cross-Language-Dataset), who already had the dataset in folders, split by languages: English, Spanish, and French. He also had the documents divided up by phrases, sentences, noun chunks etc. However, we only used the full documents in either English or Spanish, so that our classifier could have a more wholesome training.

We created one directory for our dataset within which were two more subdirectories, one for each training document set in English and Spanish. We then read the files into python and created an array of strings comprised of each file, which we then return along with the file path.

*# READ ALL LINES IN ALL FILES IN ALL DIRECTORIES IN ROOT*

**def** read\_files(path):

**for** root, dir\_names, file\_names **in** os.walk(path):

**for** file\_name **in** file\_names:

**if** file\_name **and not** file\_name.startswith(**"."**):

file\_path = os.path.join(root, file\_name)

**if** os.path.isfile(file\_path):

lines = []

f = open(file\_path, encoding=**"latin-1"**)

**for** line **in** f:

lines.append(line)

f.close()

content = NEWLINE.join(lines)

**yield** file\_path, content

Our next job is to create a dataframe from the files that we read in. This will be in the form that each string will be given a row (which is a two dimensional array), the first of which will contain the text, and the next is going to contain its classification.

*# NOW CREATE A DATA FRAME FROM ALL THE INFO YOU COLLECTED*

**def** build\_data\_frame(path, classification):

rows = []

index = []

**for** file\_name, text **in** read\_files(path):

rows.append({**'text'**: text, **'class'**: classification})

index.append(file\_name)

data\_frame = DataFrame(rows, index=index)

**return** data\_frame

### The Count-Vectorizer and the Multinomial Naive Bayes classifier

We first initialize the count\_vectorizer. We use the fit\_transform method to do two things at the same time: first, learn the words of all the documents, and next, extracts the features of these. We could have used the methods fit and transform separately, however, it’s easier to just do them together.

*# ======== COUNT VECTORIZER AND CLASSIFIER ====================*

*# Creates the object that will count the words in a document*

count\_vectorizer = CountVectorizer()

*# returns the count value for the text values in data's text column*

counts = count\_vectorizer.fit\_transform(data[**'text'**].values)

We then initialize the multinomial naive Bayes classifier and assign the target classes to the values from our data frame object. We then use the classifier to fit our target classes to the counts of the tokens.

*# creates the classifier*

classifier = MultinomialNB()

targets = data[**'class'**].values

*# classifies fits the counts to class targets*

classifier.fit(counts, targets)

### The Classification

We will now take in prompt from the user for the files locations. We will need the location for retrieving the testing files. We will also need the destination folders for the two classes of documents. At the end of this step, the classifier will have learnt to classify the documents. We can now use its predictions to classify the documents.

*# Path for the test files*

os.chdir(**"../../../../Desktop/test/"**)

path = os.getcwd()

dst\_span = **"../SPAN"**

dst\_eng = **"../ENG"**

predictions = []

examples = []

*# THIS IS WHERE IT WILL CLASSIFY THE DOCUMENTS*

**for** root, dir\_names, file\_names **in** os.walk(path):

**for** file\_name **in** file\_names:

content = []

**if** file\_name **and not** file\_name.startswith(**"."**):

file\_path = os.path.join(root, file\_name)

**if** os.path.isfile(file\_path):

print(file\_path)

content = read\_test\_files(file\_path)

examples = [content]

example\_counts = count\_vectorizer.transform(examples)

predictions = classifier.predict(example\_counts)

print(predictions) *# [1, 0]*

**if** predictions == [**'SPANISH'**]:

shutil.move(file\_path, dst\_span)

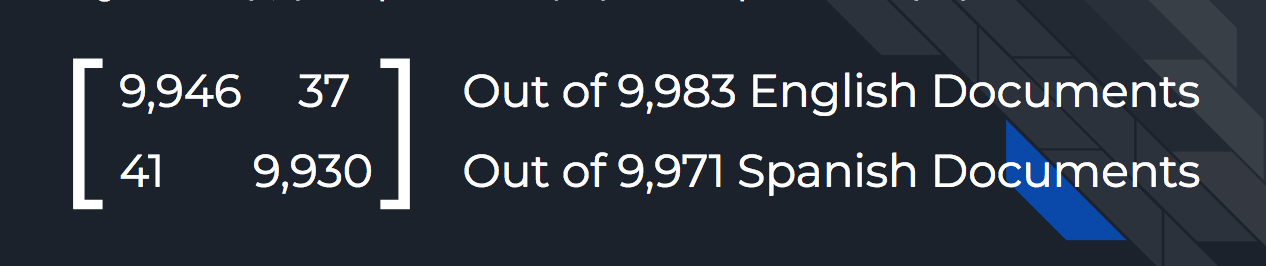
**else**:

shutil.move(file\_path, dst\_eng)

This piece of the code will move the files to their destination folder depending on which class the classifier has predicted they belong to.

## **Confusion and scores**

After the system is trained successfully, it prints out its confusion matrix that evaluated the accuracy of a classification.   
In binary classification, the count of true negatives is C{0,0}, false negatives is C{1,0}, true positives is C{1,1} and false positives is C{0,1}.



This was the confusion matrix of the current classifier, using the regular countvectorizer. We also had the system compute the F1 score, also known as balanced F-score or F-measure. The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The relative contribution of precision and recall to the F1 score are equal. The formula for the F1 score is:



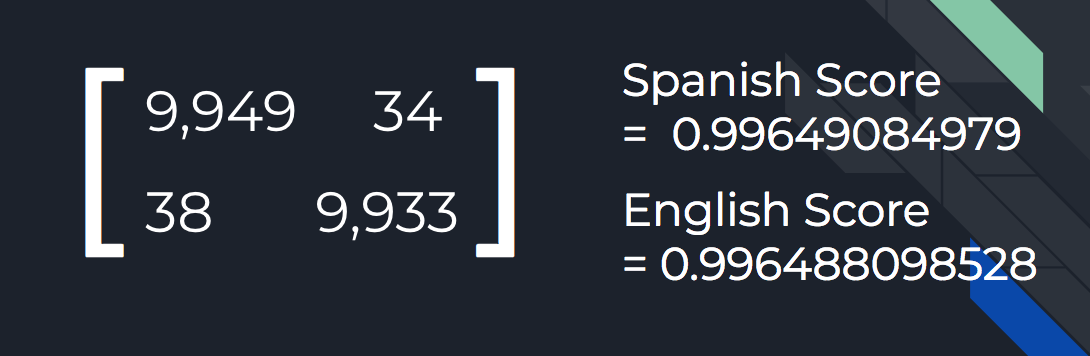
The scores for the classifier we got were as follows:

## 

## **Comparison**

To find out how well our classifier worked and to see how we could improve it, we decided to use the n-gram count vectorizer to evaluate and compare our results to. As mentioned earlier, the n-gram count vectorizer would allow us to look at context in terms of phrases and not just a bag of words which our current algorithm does.

It would look at not only the frequency with which words occur in a document, but also the order that the words appear in. However, the n-gram count vectorizer definitely allowed the results to improve, however, the improvement was very slight. The misclassification reduced by 0.0004~.



## **References**

* <http://www.tkdocs.com/tutorial/intro.html#audience>
* <http://stevenloria.com/how-to-build-a-text-classification-system-with-python-and-textblob/>
* <http://universityofbigdata.net/competition/tutorial/5681717746597888?lang=en>
* <https://www.python-course.eu/text_classification_python.php>
* <http://universityofbigdata.net/competition/tutorial/5681717746597888?lang=en>
* <https://github.com/justmarkham/pycon-2016-tutorial>
* <https://github.com/javedsha/text-classification/blob/master/Text%2BClassification%2Busing%2Bpython%2C%2Bscikit%2Band%2Bnltk.ipynb>
* <https://github.com/FerreroJeremy/Cross-Language-Dataset>
* <https://archive.ics.uci.edu/ml/datasets/Reuters+RCV1+RCV2+Multilingual,+Multiview+Text+Categorization+Test+collection>
* <http://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html>