1 Introduction

Text mining. Document classification (text categorization) in Python using the scikit-learn package.

The aim of text categorization is to assign documents to predefined categories as accurately as possible. We are within the supervised learning framework, with a categorical target attribute, often binary. The originality lies in the nature of the input attribute, which is a textual document. It is not possible to implement predictive methods directly, it is necessary to go through a data preparation phase.

Bag of words representation is often used to describe the corpus of texts in a document-term matrix format. It is joined to the target variable to form the dataset. The problem seems to be resolved at this stage as we find the usual structure of the data for predictive analysis. It is only just the beginning in reality because the matrix has the singularity of having a high dimensionality (several thousand of descriptors) and being sparse (many values are zero). Some machine learning techniques are more suitable than others. The reduction of dimensionality in particular is of considerable importance, on the one hand to improve performance and on the other hand to make the results interpretable, because in the end, beyond pure prediction, it is also a question of understanding the nature of the relationship between the documents and the predefined groups.

In this tutorial, we will describe a text categorization process in Python using mainly the text mining capabilities of the scikit-learn package, which will also provide data mining methods (logistics regression). We want to classify SMS as "spam" (spam, malicious) or "ham" (legitimate). We use the "SMS Spam Collection v.1" dataset[1] [CORPUS].

2 Document classification process

It is important not to use the same data for the learning and testing of classifiers in a predictive analysis approach. The holdout scheme is often used: a first part of the observations is extracted randomly, this is the learning sample, it is used for the construction of the model; the remaining part, called test sample, is devoted to performance measurement. We often use respectively 2/3 vs.1/3 of the instances for these samples, but there are no fixed rules in this domain.

[1] http://www.dt.fee.unicamp.br/~tiago/smsspamcollection/

In the text categorization context, it means that we must subdivide the corpus BEFORE constructing the document-term matrix. The instances of the test corpus must not be allowed to be used in the construction of the dictionary (the list of terms) and in the calculation of the weights (e.g. TF-IDF weighting) of the matrix used for the learning phase. The approach can be summarized as follows:

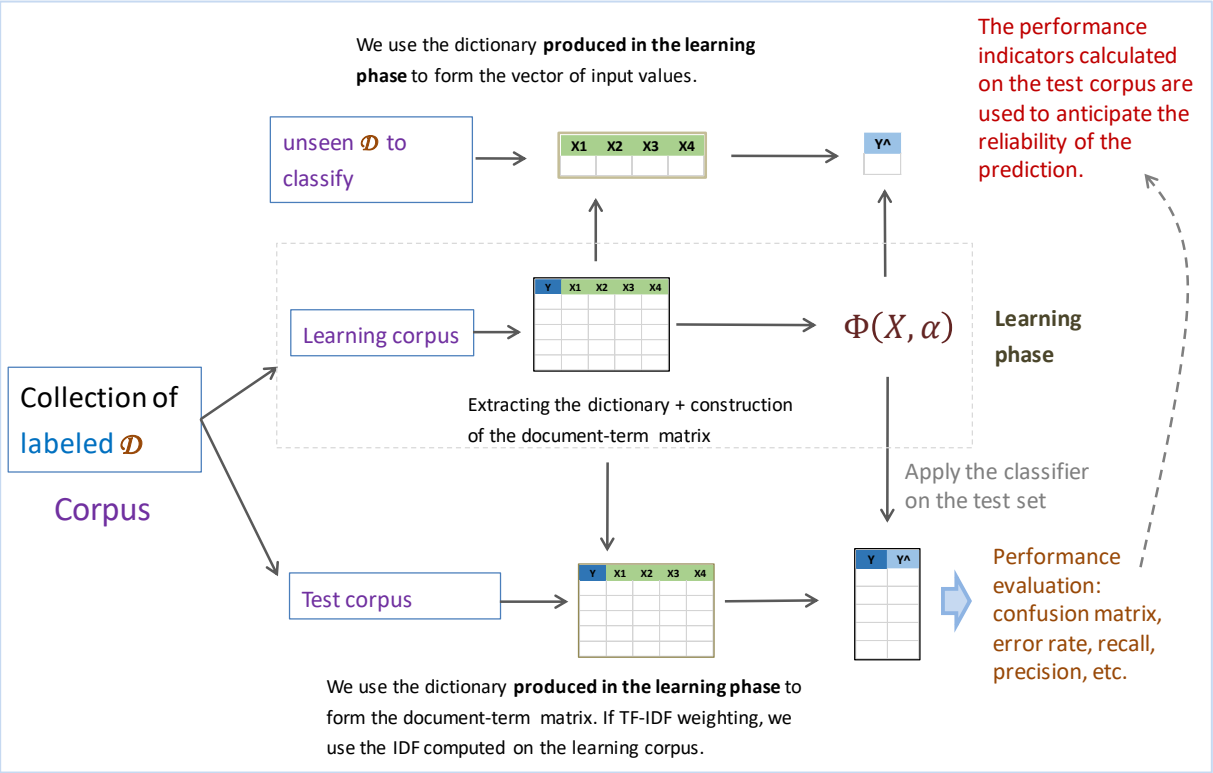

Figure 1 – Document categorization process

This constraint - partitioning of corpus before constructing the document-term matrices - takes on its full meaning when we use the final classifier in deployment i.e. when classifying an unseen document that is not available during the modeling phase. It is obvious that it must not interfere in any way with the construction of the model: if it introduces unknown terms during the modelling phase, they must be ignored; likewise, we do not know the number of documents to be classified during the model's life cycle, the calculation of the IDF of terms (inverse document frequency) must be based only on the information from the learning sample.

We must place ourselves in the same conditions in the learning and testing phases. The document term matrix used for modeling must come only from the learning corpus; the dictionary, and the resulting indicators (e. g. IDF), will then be used to construct the document term matrix for the test phase.

We comply with this roadmap (Figure 1) for the processing of the **SMS Spam Collection** dataset in this tutorial.

3 Spam detection in Python

3.1 Importation of the corpus

The "**SMSSPamCollection.txt**" corpus contains **n = 5572** messages, classified into 2 classes "spam" and "ham". Here are the first rows of the data file:

```
classe message
ham    Go until jurong point, crazy.. Available only in bugis n great world
ham    Ok lar... Joking wif u oni...
spam   Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005
ham    U dun say so early hor... U c already then say...
```

The first row corresponds to the names of the variables. The documents are then listed with in first column the class membership, in the second column the SMS message. The tabulation character "\t" is the column separator.

We use the Pandas package for importing the data file into a Data Frame structure `<class 'pandas.core.frame.DataFrame'>`, we have `2 columns` and `5572 rows`.

```
#numpy
import numpy as np

#change the current directory
import os
os.chdir("... your directory ...")

#importation of the corpus
import pandas
spams = pandas.read_table("SMSSpamCollection.txt",sep="\t",header=0)

#type of the object
print(type(spams))

#size of the dataset
print(spams.shape)
```

3.2 Description of the dataset

A quick inspection is always useful when we deal with a data set. We list the columns, their types and we calculate some descriptive statistics' indicators.

```
#list of columns
```

```
print(spams.columns)

#type of columns
print(spams.dtypes)

#description
print(spams.describe())
```

The columns have the generic type "object".

```
#list of columns
Index(['classe', 'message'], dtype='object')
#type of columns
classe     object
message    object
dtype: object
#description
       classe                 message
count    5572                    5572
unique      2                    5169
top       ham  Sorry, I'll call later
freq     4825                      30
```

The most frequent class is `''ham''` with 4825 instances (among 5572 instances); we observe that the document `"Sorry, I'll call later"` is observed 30 times.

We calculate explicitly the classes distribution:

```
#frequency distribution of the class attribute
print(pandas.crosstab(index=spams["classe"],columns="count"))
```

We observe 4825 "ham" messages, and therefore 747 "spam".

```
#frequency distribution of the class attribute
col_0   count
classe
ham      4825
spam      747
```

3.3 Partitioning the corpus into training and testing sets

We create the training and testing corpora with, respectively, $n_{train} = 3572$ and $n_{test} = (5572 - 3572) = 2000$ documents. We prefer a stratified draw to obtain the same classes proportions in both samples. We use the train_test_split function of the sklearn.model_selection module.

```
#subdivision into train and test sets
from sklearn.model_selection import train_test_split
```

```
spamsTrain, spamsTest = train_test_split(spams,train_size=3572,random_state=1,stratify=spams['classe'])

#frequency distribution of the class attribute
#train set
freqTrain = pandas.crosstab(index=spamsTrain["classe"],columns="count")
print(freqTrain/freqTrain.sum())
#test set
freqTest = pandas.crosstab(index=spamsTest["classe"],columns="count")
print(freqTest/freqTest.sum())
```

The proportions are well respected, the accuracy of the evaluation will be better.

```
#train set
col_0      count classe
ham     0.865901
spam    0.134099
#test set
col_0   count classe
ham     0.866
spam    0.134
```

3.4 Building the document-term matrix for the learning phase

We can now construct the document-term matrix for the learning corpus. We choose the binary weighting. The operation is carried out in two steps. (1) We instantiate the CountVectorizer tool. (2) We call the fit_transform() function by passing the learning corpus spamsTrain as parameter.

```
#import the CountVectorizer tool
from sklearn.feature_extraction.text import CountVectorizer

#instantiation of the objet – binary weighting
parseur = CountVectorizer(binary=True)

#create the document term matrix
XTrain = parseur.fit_transform(spamsTrain['message'])
```

The method creates the dictionary and document-term matrix that we assign to the variable XTrain. We can display the number and list of terms that make up the dictionary.

```
#number of tokens
print(len(parseur.get_feature_names()))

#list of tokens
print(parseur.get_feature_names())
```

We observe 6789 terms. Listing them would be too tedious. We just notice that the all characters are transformed in lowercase. To calculate the frequency of terms, we use XTrain. It is in the "sparse matrix" format, we transform it into a "numpy" matrix that we store in the variable mdtTrain.

```
#transform the sparse matrix into a numpy matrix
mdtTrain = XTrain.toarray()

#type of the matrix
print(type(mdtTrain))

#size of the matrix
print(mdtTrain.shape)
```

The dimension of the document term matrix is (3572, 6789). We calculate the number of documents in which each term appears, we can use the sum since we have chosen the binary weighting. Then we sort the frequency vector to highlight the most frequent terms.

```
#frequency of the terms
freq_mots = np.sum(mdtTrain,axis=0)
print(freq_mots)

#arg sort
index = np.argsort(freq_mots)
print(index)

#print the terms and their frequency
imp = {'terme':np.asarray(parseur.get_feature_names())[index],'freq':freq_mots[index]}
print(pandas.DataFrame(imp))
```

The 5 most frequent terms are:

```
#5 most frequent terms
522              and
528               in
647              the
1042              you
1091               to
```

The term ''to'' appears into 1091 documents, ''you'' into 1042, etc.

3.5 Modelling with logistic regression

We can launch the modelling process. We use the logistic regression from the scikit-learn package. We import and instantiate LogisticRegression tool, we call the fit() function.

```
#import the class LogistiRegression
```

```
from sklearn.linear_model import LogisticRegression

#instatiate the object
modelFirst = LogisticRegression()

#perform the training process
modelFirst.fit(mdtTrain,spamsTrain['classe'])
```

We have a vector with 6789 values, we have also the intercept.

```
#size of coefficients matrix
print(modelFirst.coef_.shape) #(1, 6789)

#intercept of the model
print(modelFirst.intercept_) #-4.4777
```

For the binary logistic regression, we have only one classification function:

$$D = a_0 + a_1 \times T_1 + a_2 \times T_2 + \cdots + a_p \times T_p$$

p is the number of terms, a_j is the coefficient for the term T_j, a_0 is the intercept. The decision rule is:

$$\textbf{IF } \mathrm{D(document)} > 0 \textbf{ THEN } \mathrm{Prediction} = \text{“spam”} \textbf{ ELSE } \mathrm{Prediction} = \text{“ham”}$$

3.6 Evaluation on the test set

To apply the classifier to the test corpus, we must build the corresponding document-term matrix, **by using the dictionary resulting from the learning process**. We apply the transform() function of the **parseur** objet instantiated during the learning phase (section **Erreur ! Source du renvoi introuvable.**) on the test corpus spamsTest (section 3.3).

```
#create the document term matrix
mdtTest = parseur.transform(spamsTest['message'])
#size of the matrix
print(mdtTest.shape)
```

We have a matrix of dimension (2000, 6789): 2000 rows because we have 2000 documents into the test corpus, 6789 columns because we have 6789 terms into the dictionary created during the learning process (page **Erreur ! Signet non défini.**).

We calculate the prediction of the classifier on the test set…

```
#prediction for the test set
predTest = modelFirst.predict(mdtTest)
```

… and we calculate the various performance indicators. We use the metrics tool form the scikit-learn library.

```
#import the metrics class for the performance measurement
from sklearn import metrics

#confusion matrix
mcTest = metrics.confusion_matrix(spamsTest['classe'],predTest)
print(mcTest)

#recall
print(metrics.recall_score(spamsTest['classe'],predTest,pos_label='spam'))

#precision
print(metrics.precision_score(spamsTest['classe'],predTest,pos_label='spam'))

#F1-Score
print(metrics.f1_score(spamsTest['classe'],predTest,pos_label='spam'))

#accuracy rate
print(metrics.accuracy_score(spamsTest['classe'],predTest))
```

We obtain respectively:

<table>
<tr><th>Indicator</th><th>Value</th></tr>
<tr><td>Confusion matrix</td><td>
<table>
<tr><td></td><td colspan="2">Prediction</td></tr>
<tr><td></td><td>Ham</td><td>Spam</td></tr>
<tr><td>Ham</td><td>1732</td><td>0</td></tr>
<tr><td>Spam</td><td>38</td><td>230</td></tr>
</table>
</td></tr>
<tr><td>Recall</td><td>0.858</td></tr>
<tr><td>Precision</td><td>1.0</td></tr>
<tr><td>F1-Score</td><td>0.924</td></tr>
<tr><td>Accuracy rate</td><td>0.981</td></tr>
</table>

The classifier seems not too bad. There are no false positive instances i.e. when we predict a spam, this is always correct. On the other hand, we observe that the recall is less good: 14.2% of the spams are not detected.

3.7 Dimensionality reduction 1 – Stop words and terms' frequencies

A brief study of the dictionary shows that some terms are very frequent (page 6), these common terms have no really meaning: « to », « you », « the », … These are the *stop words*. They do not enable to discriminate the documents. It seems better to remove them from the dictionary.

On the other hand, we can also consider that too rare terms are not relevant because they are anecdotal. We also remove them from the dictionary.

In this section, we repeat the previous analysis by introducing these two options when instantiating the CountVectorizer tool: stop_words = 'english' for the removing of stop words, min_df = 10 for the removing of the terms which occur in less than 10 documents.

```
#rebuild the parser with new options : stop_words='english' and min_df = 10
parseurBis = CountVectorizer(stop_words='english',binary=True, min_df = 10)
XTrainBis = parseurBis.fit_transform(spamsTrain['message'])
#number of tokens
print(len(parseurBis.get_feature_names()))
#document term matrix
mdtTrainBis = XTrainBis.toarray()
#instatiate the object
modelBis = LogisticRegression()
#perform the training process
modelBis.fit(mdtTrainBis,spamsTrain['classe'])
#create the document term matrix for the test set
mdtTestBis = parseurBis.transform(spamsTest['message'])
#prediction for the test set
predTestBis = modelBis.predict(mdtTestBis)
#confusion matrix
mcTestBis = metrics.confusion_matrix(spamsTest['classe'],predTestBis)
print(mcTestBis)
#recall
print(metrics.recall_score(spamsTest['classe'],predTestBis,pos_label='spam'))
#precision
print(metrics.precision_score(spamsTest['classe'],predTestBis,pos_label='spam'))
#F1-Score
print(metrics.f1_score(spamsTest['classe'],predTestBis,pos_label='spam'))
#accuracy rate
print(metrics.accuracy_score(spamsTest['classe'],predTestBis))
```

With more than 12 times less terms (**541** vs. 6789), we preserve the quality of prediction:

<table>
<tr><td>Indicator</td><td>Value</td></tr>
<tr><td>Confusion matrix</td><td>
<table>
<tr><td></td><td colspan="2">Prediction</td></tr>
<tr><td></td><td>Ham</td><td>Spam</td></tr>
<tr><td>Ham</td><td>1731</td><td>1</td></tr>
<tr><td>Spam</td><td>37</td><td>231</td></tr>
</table>
</td></tr>
<tr><td>Recall</td><td>0.862</td></tr>
</table>

Precision	0.996
F1-Score	0.924
Accuracy rate	0.981

We have a simpler classifier with the same performance.

3.8 Dimensionality reduction 2 – Post processing of the classifier

3.8.1 Variable selection strategy

Is it possible to further reduce dimensionality? We can consider the properties of the predictive model. Some coefficients of the classification function are almost zero, they influence negligibly the decision rule. A simple strategy (very rough I would say) consists in (1) removing the corresponding terms from the dictionary, (2) re-estimating the parameters of the model composed of the remaining terms.

I know this is really an unsophisticated approach. Conventionally, we use a sequential approach by adding or removing one attribute at each step, especially to handle appropriately the collinearity problem (many of the descriptors are redundant). Based on computation consideration, we simplify the variable selection approach to handle the high number of candidate descriptors. The test error rate becomes our main reference.

3.8.2 Implementation and predictive performance

We try to implement this idea in this section. We characterize the coefficients of the learning classifier. We transform them in absolute value, then we calculate the quantiles.

```
#absolute  value of the coefficients
coef_abs = np.abs(modelBis.coef_[0,:])

#percentiles of the coefficients (absolute value)
thresholds = np.percentile(coef_abs,[0,25,50,75,90,100])
print(thresholds)
```

We obtain…

```
#percentiles of the coefficients (absolute value)
[ 0.01367356  0.17817203  0.30258512  0.60639769  1.03953052  2.70949586]
```

The lowest value of the coefficients in absolute value is 0.01367356, the highest value 2.70949586. We choose the 1st quartile **0.17817203** as threshold value. We identify the terms corresponding to the coefficients higher than this threshold in absolute value.

```
#identify the coefficients "significantly" higher than zero
```

```
#use 1st quartile as threshold
indices = np.where(coef_abs > thresholds[1])
print(len(indices[0]))
```

405 descriptors are selected (against 541 in the previous step, section 3.7). We create the corresponding learning and testing document-term matrices.

```
#train and test sets
mdtTrainTer = mdtTrainBis[:,indices[0]]
mdtTestTer = mdtTestBis[:,indices[0]]

#checking
print(mdtTrainTer.shape)
print(mdtTestTer.shape)
```

The dimensions of the datasets are respectively $(3572, 405)$ et $(2000, 405)$.

Note: We can proceed directly from the document-term matrix of the preceding analysis because we use a simple weighting (presence/absence of terms). If then weighting takes into account the length of documents (e. g. relative frequency of terms), rather than performing complicated calculations, it would have been better to filter the dictionary and then repeat the construction of document-term matrix.

We launch again the learning (**modelTer**) and test processes.

```
#instatiate the object
modelTer = LogisticRegression()
#train a new classifier with selected terms
modelTer.fit(mdtTrainTer,spamsTrain['classe'])
#prediction on the test set
predTestTer = modelTer.predict(mdtTestTer)
#confusion matrix
mcTestTer = metrics.confusion_matrix(spamsTest['classe'],predTestTer)
print(mcTestTer)
#recall
print(metrics.recall_score(spamsTest['classe'],predTestTer,pos_label='spam'))
#precision
print(metrics.precision_score(spamsTest['classe'],predTestTer,pos_label='spam'))
#F1-Score
print(metrics.f1_score(spamsTest['classe'],predTestTer,pos_label='spam'))
#accuracy rate
print(metrics.accuracy_score(spamsTest['classe'],predTestTer))
```

The **F1-Score** is **0.926**. We note that the quality of modelling is not deteriorated by the dimensionality reduction. Here are the details of the results:

Indicator	Value		
Confusion matrix		Prediction	
		Ham	Spam
	Ham	1731	1
	Spam	36	232
Recall	0.866		
Precision	0.996		
F1-Score	0.926		
Accuracy rate	0.981		

The number of terms has been reduced from 6789 to **405 terms**, while preserving the predictive performance of the classifier. The result is rather positive.

3.8.3 Interpretation – Influence of the terms in the classifier

Let us try to identify the most discriminating terms. To do this, we sort the dictionary according to the absolute value of the model coefficients:

```
#selected terms
sel_terms = np.array(parseurBis.get_feature_names())[indices[0]]

#sorted indices of the absolute value coefficients
sorted_indices = np.argsort(np.abs(modelTer.coef_[0,:]))

#print the terms and theirs coefficients
imp = {'term':np.asarray(sel_terms)[sorted_indices],'coef':modelTer.coef_[0,:][sorted_indices]}
print(pandas.DataFrame(imp))
```

The 10 most important terms in the model are (with the value of the coefficients):

```
1.760636      text
1.798298      http
1.823208      free
1.884867        50
1.948201       txt
1.999089       new
2.058226      150p
2.201104   service
2.249400     claim
2.715046        uk
```

Since the coefficients of these terms are positive, they all contribute to the designation of "spam" i.e. when they are present in documents, the chances of dealing with "spam"

increase. The detailed analysis of the results begins at this stage. It is likely to be expected that the dictionary will need to be refined to improve its relevance…

3.8.4 Reservations concerning the variable selection

Again, the approach described in the previous section to eliminate irrelevant terms is particularly questionable. Simultaneous removal of descriptors of which estimated coefficients are "close to" zero - apart from any other consideration - is only valid if they (the descriptors) are statistically independent. In practice, we should take their covariances into account when performing the tests for significance of the coefficients. We can also use a likelihood-ratio tests. In both cases, the amount of calculations makes the approach impracticable on datasets containing several hundred or even thousands of descriptors.

The choice of the threshold is also questionable. But I am not as uncomfortable, actually. Machine learning algorithms are by nature parameterized. Choosing the first quartile as threshold value is not more uncertain than choosing the significance level for a variable selection process based on a succession of tests for significance. We can consider them as control parameters than enable to guide learning algorithms.

3.9 Deployment

One of the purposes of text categorization is to produce a function that automatically classify a new document as "spam" or "ham". It can be implemented in the SMS message reception software of your smartphone for example. In this section, we detail the different steps of operations.

We want to classify the phrase "this is a new free service for you only" from our third model **modelTer** (page 11).

Description compatible with the document-term matrix. We transform the document into a vector of presence absence of terms observed in the dictionary:

```
#document to classify
doc = ['this is a new free service for you only']

#get its description
desc = parseurBis.transform(doc)
print(desc)
```

Python says that it identifies the terms n° 166, 315 et 405. We have a "sparse" description of the data i.e. only the non-zero values are detected.

```
  (0, 166)      1
  (0, 315)      1
  (0, 405)      1
```

Which are these terms?

```
#which terms
print(np.asarray(parseurBis.get_feature_names())[desc.indices])
```

We have: 'free', 'new' et 'service'.

```
#print(np.asarray(parseurBis.get_feature_names())[desc.indices])
['free' 'new' 'service']
```

Therefore, the other terms ('this', 'is', 'a', 'for', 'you', 'only') are ignored because they are not listed in the dictionary. They have no influence on the classification of the message.

Note: Maybe wrongly, by the way. The sequence of the 3 terms "for you only" is an n-gram of words that can be very relevant…. But, to take account this kind of information would lead us to redo the analysis from the beginning. **A work is never definitive in text mining process**.

Application of the variable selection. A term may be present into the dictionary, but absent from the model because we performed an additional variable selection in our analysis. We must apply this processing - remove the terms which are not present into the model - before applying the classifier on the data vector.

```
#dense representation
dense_desc = desc.toarray()
#apply var. selection
dense_sel = dense_desc[:,indices[0]]
```

Prediction of the class membership. We can now call the predict() procedure from the **modelTer** object.

```
#prediction of the class membership
pred_doc = modelTer.predict(dense_sel)
print(pred_doc)
```

The predicted class is "spam".

Reliability of prediction. Obtaining a prediction is good. But, having an indication of the reliability of the prediction is better. We can obtain the class membership probabilities using the function predict_proba().

```
#prediction of the class membership probabilities
pred_proba = modelTer.predict_proba(dense_sel)
print(pred_proba)
```

The belonging to the class "spam" seems obvious with the probability 0.9215.

```
#print(pred_proba)
[[ 0.07846502  0.92153498]]
```

Verification of the calculations. Since we have the coefficients of the model (page 12), we can reproduce the calculations:

```
#checking - logit
logit = 1.823208 + 1.999089 + 2.201104 + modelTer.intercept_

#probability – logistic function
import math
p_spam = 1/(1+math.exp(-logit))
print(p_spam)
```

Of course, the intercept (**modelTer.intercept_**) must be used in the calculation.

We obtain the same value as with the function predict_proba() (we are slightly less accurate because we handle manually the coefficients with a restricted number of digits).

```
#print(p_spam)
0.9215349...
```

4 Conclusion

Statistical analysis of textual data is an exciting application of data mining. It requires both our statistical and computer skills. This tutorial outlines the document categorization process. We deal with the SPAMS detection problem in SMS messages. The problem is realistic enough, it allows to identify the main difficulties of the task.

Yet, we made it simple. There are many opportunities of improvement: reducing dimensionality through techniques based on the characteristics of terms (e. g. stemming, lemmatization,...), through more aggressive or more sophisticated statistical selection approaches (e. g. ranking methods based on correlation), through methods for transformation of the representation space (e. g. topic modeling); exploiting another weighting system; using other machine learning algorithms (e. g. SVM, random forest, gradient boosting, etc.)... There are many challenges.

5 References

[CORPUS] Almeida, T.A., Gómez Hidalgo, J.M., Yamakami, "A. Contributions to the Study of SMS Spam Filtering: New Collection and Results", in Proceedings of the 2011 ACM Symposium on Document Engineering (DOCENG'11), Mountain View, CA, USA, 2011.