**Implementing your own spam filter**

While doing this hands-on exercise, you’ll work with *natural language data*, learn how to detect [the words spammers use](https://sm.asisonline.org/migration/Pages/learn-words-spammers-use-lure-you-006874.aspx) automatically, and learn how to use a *Naive Bayes classifier* for binary classification.

Classification

Classification is ubiquitous: many things around us can be divided into two or more classes based on their characteristics. For example, we call a vehicle with one wheel a “unicycle”, a vehicle with two wheels and no motor a “bicycle”, and a vehicle with two wheels and a motor a “motorcycle”. Here, we are using the *number of wheels* and the *presence of a motor* to classify the vehicle. Machines can do the same. Even better, being exposed to a sufficient amount of data describing the instances of different classes, they can learn about their characteristic properties and detect the classes of the new instances.

A special case of classification is *binary classification* which assumes that the choice is between just two classes.

Spam filtering

Spam filtering is a binary classification task familiar to any user of email services. Now you’ll learn how to implement your own spam filter.

The task is to distinguish between two types of emails, “spam” and “non-spam” often called “ham”. The machine learning classifier will detect that an email is spam if it is characterised by certain features. The textual content of the email – words like “Viagra” or “lottery” or phrases like “You've won a 100,000,000 dollars! Click here!”, “Join now!” – is crucial in spam detection and offers some of the strongest cues:

Since you’ll be working with text data, you’ll use the Python-based library [Natural Language Toolkit](http://www.nltk.org/)(NLTK). It is well supported and well documented. You can even read the [online book](http://www.nltk.org/book/).

Start by importing the toolkit:

import nltk

To train the classifier, we need a representative dataset with both spam and ham emails. We’ll be using the [Enron email dataset](http://labs-repos.iit.demokritos.gr/skel/i-config/downloads/enron-spam/preprocessed/) which contains emails of both types stored in plain text format.

You can download the archive and extract any of the directories – for example, the enron1/ directory contains 3672 legitimate (ham) emails and 1500 spam emails.

The spam detection algorithm will involve five steps:

1. Lodaing the data,
2. Preprocessing,
3. Extracting the features,
4. Training the classifier, and
5. Evaluating the classifier.

Let's get started!

Step 1: Loading the data

You need to read in the files from the spam and ham subfolders and keep them in two separate lists. To be able to iteratively read the files in a folder, add the following import statement:

import os

Then define the function initialise\_lists as follows:

**def** init\_lists(folder):

a\_list = []

file\_list = os.listdir(folder)

**for** a\_file in file\_list:

f = open(folder + a\_file, 'r')

a\_list.append(f.read())

f.close()

**return** a\_list

Now you can use this function to create spam and ham lists. For that, add the following two lines of code to the main part of your program:

spam = init\_lists('enron1/spam/')

ham = init\_lists('enron1/ham/')

Next, let’s combine the two lists keeping the labels. You can do this by creating a list of *tuples* – Python values that contain each a pair of values, – where the first member of the pair stores the text of the email and the second one its label.

spam\_emails = [(email, 'spam') **for** email in spam]

ham\_emails = [(email, 'ham') **for** email in ham]

all\_emails = spam\_emails + ham\_emails

Now, the first tuple in all\_emails list contains the first spam email read in from the spam subfolder, the following 1499 ones are spam emails as well, while the 1501th tuple contains the first ham email read in from the ham subfolder. You can always check if your data is loaded correctly by checking the size of the data structure:

print (len(all\_emails))

This should print out 5172. Before starting to build the classifier, let’s randomly shuffle the spam and ham examples. This way it will be easier to organise the training data because any portion of all\_emails will contain examples of both categories. Add the following import statement to the import statements block:

import random

random.shuffle(all\_emails)

Step 2: Preprocessing the data

Currently, the data is stored as lines of text, for example:

The purpose of the email is to recap the kickoff meeting held on yesterday.  
(ham email #013 in enron1/)

People are getting rich using this system! Now it's your turn! We've cracked the code and will show you...  
(spam email #046 in enron1/)

Subject: popular meds at lowest prices  
(spam email #838 in enron1/)

To be able to use the words in these texts as features for your classifier, you need to preprocess the data and normalise it (so that different forms of the same word are treated as the same word). You can acheive that by

* Splitting the text by white spaces and punctuation marks using a [*tokenizer*](http://www.nltk.org/book/ch03.html).
* Linking the different forms of the same word (for example, “price” and “prices”, “is” and “are”) to each other – using a [*lemmatizer*](http://www.nltk.org/book/ch03.html)
* Converting all words to lowercase so that the classifier does not treat “People”, “people” and “PEOPLE” as three separate features.

Add the following import statement to the import statements block to include the tokenizer and lemmatizer:

from nltk import word\_tokenize, WordNetLemmatizer

Then define the preprocess function that will take a sentence as an input and will return the result of all the preprocessing operations:

**def** preprocess(sentence):

tokens = word\_tokenize(sentence)

**return** [lemmatizer.lemmatize(word.lower()) **for** word in tokens]

Step 3: Extracting the features

Once text is pre-processed, you can extract the features characterising spam and ham emails. The first thing to notice is that some words such as “the”, “is” or “of” appear in all emails and don’t have much content to them. These words are not going to help you distinguish spam from ham. Such words are called *stopwords* and they can be disregarded during classification. NLTK has a corpus of stopwords for several languages including English, which you can import and use:

from nltk.corpus import stopwords

stoplist = stopwords.words(‘english’)

To extract the features – words that can tell the program whether the email is spam or ham – you’ll need to do the following:

1. Read in the text of the email.
2. Preprocess it using the function preprocess defined above.
3. For each word that is not in the stopword list, either
   * calculate how frequently it occurs in the text, or simply
   * register the fact that the word occurs in the email.

The former approach is called the *bag-of-words* (bow), and it allows the classifier to notice that certain keywords may occur in both types of emails but with different frequencies. Python's (https://docs.python.org/2/library/collections.html)(Counter subclass) allows to apply the bow model. To use the Counter subclass, add the import statement:

from collections import Counter

You can control which model you want to use with the parameter setting in the following code (the simple word occurrence model is the default):

**def** get\_features(text, setting):

**if** setting=='bow':

**return** {word: count **for** word, count in Counter(preprocess(text)).items() **if** not word in stoplist}

**else**:

**return** {word: True **for** word in preprocess(text) **if** not word in stoplist}

The code above uses (https://docs.python.org/2/tutorial/datastructures.html)(dictionary comprehensions).

Now you can extract the features from the emails and pair them with the email class label (“spam” or “ham”). Add the following line of code to the main part of the program if you want to use the bow model:

all\_features = [(get\_features(email, 'bow'), label) **for** (email, label) in all\_emails]

and the following one for the default model:

all\_features = [(get\_features(email, ''), label) **for** (email, label) in all\_emails]

Step 4: Training a classifier

Now that the data is in the correct format, you can split it into a training set that will be used to train the classifier, and a test set that will be used to evaluate it. Typically, the data is split using 80% for training and the other 20% for testing.

Define a function train that will take the set of features and the proportion of the examples assigned to the training set as arguments:

**def** train(features, samples\_proportion):

train\_size = int(len(features) \* samples\_proportion)

train\_set, test\_set = features[:train\_size], features[train\_size:]

print ('Training set size = ' + str(len(train\_set)) + ' emails')

print ('Test set size = ' + str(len(test\_set)) + ' emails')

The print statements will help you make sure that the data is split correctly: if you use 80% of the data in enron1/ to train the classifier, the training set should contain 4137 emails, and the test set 1035.

You can apply any classifier of your choice; NLTK provides [a number of them](http://www.nltk.org/book/ch06.html). This post will show you how to use *Naive Bayes classifier*, which is a simple yet powerful classification algorithm that has been widely applied to spam filtering.

The classifier tries to choose the most probable class, or label, among the two classes, spam and ham, i.e., c∈{ham,spam}c∈{ham,spam} based on what it has learned about the features (presence or frequency of words in the emails of each type). More precisely, it’s trying to choose the most probable class given the words in the e-mail:

c^=argmaxc∈{ham,spam}P(c|words)c^=argmaxc∈{ham,spam}P(c|words)

Don’t worry if the formula looks a bit complicated at first. It simply says that the classifier will assign the class (denoted as c^c^) it will choose among the two classes by looking which of the two probabilities – “spam” given the words in the email P(spam|words)P(spam|words), or “ham” given the words in the email P(ham|words)P(ham|words) – is higher (thus the argmaxargmax). These probabilities cannot be directly estimated, but Bayes rule allows you to swap the conditions and get:

P(c|words)=P(words|c)P(c)/P(words)P(c|words)=P(words|c)P(c)/P(words)

Now the classifier will need to compare these fractions for the two classes. When comparing these two fractions, you can disregard the denominator because it stays the same for both classes, and directly compare the product P(words|spam)P(spam)P(words|spam)P(spam) with P(words|ham)P(ham)P(words|ham)P(ham).

c^=argmaxc∈{ham,spam}P(words|c)P(c)c^=argmaxc∈{ham,spam}P(words|c)P(c)

Probabilities P(spam)P(spam) and P(ham)P(ham) are called the *prior probabilities*, and they show the distribution of “spam” and “ham” classes in the training set. The probabilities P(words|spam)P(words|spam) and P(words|ham)P(words|ham) are called *conditional probabilities* of having a particular set of features if the email is “spam” or if it is “ham”. Naive Bayes classifier assumes that each feature (word) occurs in a text independently of all other words, so we can multiply the conditional probabilities for each of the words directly. In short, the algorithm will say that an email is spam if:

P(spam)∏w∈wordsP(w|spam)>P(ham)∏w∈wordsP(w|ham)P(spam)∏w∈wordsP(w|spam)>P(ham)∏w∈wordsP(w|ham)

and ham otherwise.

The probabilities are calculated on the training data: for example, P(spam)P(spam) in enron1/ dataset is 0.29 (since there are 1500 spam emails out of the 5172), but you don’t have to estimate the probabilities yourself: the NLTK implementation of the algorithm will do it for you.

To use this classifier, make sure that it has been imported:

from nltk import NaiveBayesClassifier, classify

Then add the following code to the train function to train a model based on the training dataset:

classifier = NaiveBayesClassifier.train(train\_set)

**return** train\_set, test\_set, classifier

Now you have a working spam filter, which you can use by calling to the train function from the main part of the program using the appropriate data split, for example:

train\_set, test\_set, classifier = train(all\_features, 0.8)

Now you have train and test set and also the classifier ready to detect your spam emails.

Step 5: Evaluating your classifier performance

So far so good! But how do we know if your classifier is doing a good job at detecting spam messages?

One way to check this is to evaluate its performance on the test dataset – the part of the data you’ve set apart (20% if you’ve been following the settings from above). Let’s add a new function called evaluate to your program. It will take the train\_set, test\_set and classifier as arguments:

**def** evaluate(train\_set, test\_set, classifier):

print ('Accuracy on the training set = ' + str(classify.accuracy(classifier, train\_set)))

print ('Accuracy of the test set = ' + str(classify.accuracy(classifier, test\_set)))

Accuracy on the training set will tell you how good the classifier is at learning the relevant information about the features and detecting the classes on the very same data it has been trained on. The accuracy on the test set shows how well it generalises this knowledge when applying it to the emails it hasn’t seen before.

To see how well the classifier performs on both sets, call this function from the main part of the program:

evaluate(train\_set, test\_set, classifier)

Remember, that you have shuffled the data early on to avoid any bias, so every time you run your program you’ll be getting slightly different results. In any case, you should be getting accuracies above 90%.

From a natural language perspective, another thing to check is which words the classifier finds most discriminative when detecting a spam message. You can do this by adding just one line of code to the evaluate function:

classifier.show\_most\_informative\_features(20)

This will show you the top 20 most informative words (you can change this number).

The full script:

from \_\_future\_\_ import print\_function, division

import nltk

import os

import random

from collections import Counter

from nltk import word\_tokenize, WordNetLemmatizer

from nltk.corpus import stopwords

from nltk import NaiveBayesClassifier, classify

stoplist = stopwords.words('english')

**def** init\_lists(folder):

a\_list = []

file\_list = os.listdir(folder)

**for** a\_file in file\_list:

f = open(folder + a\_file, 'r')

a\_list.append(f.read())

f.close()

**return** a\_list

**def** preprocess(sentence):

lemmatizer = WordNetLemmatizer()

**return** [lemmatizer.lemmatize(word.lower()) **for** word in word\_tokenize(unicode(sentence, errors='ignore'))]

**def** get\_features(text, setting):

**if** setting=='bow':

**return** {word: count **for** word, count in Counter(preprocess(text)).items() **if** not word in stoplist}

**else**:

**return** {word: True **for** word in preprocess(text) **if** not word in stoplist}

**def** train(features, samples\_proportion):

train\_size = int(len(features) \* samples\_proportion)

*# initialise the training and test sets*

train\_set, test\_set = features[:train\_size], features[train\_size:]

print ('Training set size = ' + str(len(train\_set)) + ' emails')

print ('Test set size = ' + str(len(test\_set)) + ' emails')

*# train the classifier*

classifier = NaiveBayesClassifier.train(train\_set)

**return** train\_set, test\_set, classifier

**def** evaluate(train\_set, test\_set, classifier):

*# check how the classifier performs on the training and test sets*

print ('Accuracy on the training set = ' + str(classify.accuracy(classifier, train\_set)))

print ('Accuracy of the test set = ' + str(classify.accuracy(classifier, test\_set)))

*# check which words are most informative for the classifier*

classifier.show\_most\_informative\_features(20)

**if** \_\_name\_\_ == &amp;amp;quot;\_\_main\_\_&amp;amp;quot;:

*# initialise the data*

spam = init\_lists('enron1/spam/')

ham = init\_lists('enron1/ham/')

all\_emails = [(email, 'spam') **for** email in spam]

all\_emails += [(email, 'ham') **for** email in ham]

random.shuffle(all\_emails)

print ('Corpus size = ' + str(len(all\_emails)) + ' emails')

*# extract the features*

all\_features = [(get\_features(email, ''), label) **for** (email, label) in all\_emails]

print ('Collected ' + str(len(all\_features)) + ' feature sets')

*# train the classifier*

train\_set, test\_set, classifier = train(all\_features, 0.8)

*# evaluate its performance*

evaluate(train\_set, test\_set, classifier)