Solutions

November 30, 2022

1 Preliminaries

Load the "SMS spam" dataset from ILIAS. This dataset contains several emails pre-labeled as spam (trash) or ham (legitimate e-mail).

[1]:
```
import pandas as pd

spam = pd.read_csv('sms-spam.csv', sep=',') # Spam dataset
spam.head(5)
```

[1]:

	type	text
0	ham	Hope you are having a good week. Just checking in
1	ham	K..give back my thanks.
2	ham	Am also doing in cbe only. But have to pay.
3	spam	complimentary 4 STAR Ibiza Holiday or £10,000 …
4	spam	okmail: Dear Dave this is your final notice to…

2 Question 1 and 2: Preprocess this dataset by applying a stemming procedure and remove the stopwords contained in the "stopwords.txt" file. While removing the stopwords, it is faster to store them into a list or a dictionary?

First, we define a list of all SMS messages.

[2]:
```
messages = [] # Init

for msg in spam['text']:
    messages.append(msg) # Create list of SMS texts

len(messages)
```

[2]:
```
5559
```

Then we tokenize our messages. We can apply some initial preprocessing, like removing the punctuations and turning every word to lowercase.

[3]:
```
import re # RegExp library
import nltk # Python library for NLP

punctuation_rule = re.compile(r'[^\w\s]+$') # RegExp that matches punctuations␣
 ↪that occur one or more times

def is_punctuation(string):
    """
    Check if STRING is a punctuation marker or a sequence of
    punctuation markers.
    """
    return punctuation_rule.match(string) is not None # Return punctuation if␣
 ↪present

def tokenize_text(text, language='english', lowercase=True):
    """
    Tokenize input text, remove punctuation and put everything to lowercase.
    """
    if lowercase:
        text = text.lower() # All words to lowercase
    tokens = nltk.tokenize.word_tokenize(text, language=language) # Tokenize␣
 ↪specifying the language
    tokens = [token for token in tokens if not is_punctuation(token)] # Exclude␣
 ↪punctuations
    return tokens
```

Then we apply tokenization.

[4]:
```
msg_token = [tokenize_text(msg, 'english') for msg in messages] # Tokenize␣
 ↪every SMS
```

Now we can define our stemming algorithm. A common one for the English language can be the s-stemmer.

[5]:
```
def s_stemmer(token):
    """
    Apply the s-stemmer. Three rules.
    If a word ends in «-ies», but not «-eies» or «-aies»
    then replace «-ies» by «-y».
    If a word ends in «-es», but not «-aes», «-ees» or «-oes»
    then replace «-es» by «-e».
    If a word ends in «-s», but not «-us» or «-ss»
    then remove the «-s».
    """
    token_length = len(token) - 1
    if (token is None) or (token_length < 3): # We ignore small tokens of size␣
 ↪< 3
```

```
        return token
    if token[token_length] != u's': # Token must end with -s
        return token
    if (token[token_length - 1] == u'e') and (token[token_length - 2] == u'i'):
        if (token[token_length - 3] != u'e') and (token[token_length - 3] !=␣
↪u'a'):
            return token[:token_length - 2] + u'y'  # First rule
        else:
            return token
    if token[token_length - 1] == u'e':
        if (token[token_length - 2] != u'a') and (token[token_length - 2] !=␣
↪u'e') and (token[token_length - 2] != u'o'):
            return token[:token_length] # Second rule
        else:
            return token
    if (token[token_length - 1] != u'u') and (token[token_length - 1] != u's'):
        return token[:token_length] # Third rule
    else:
        return token
```

Let's test our s-stemmer on a simple example.

[6]:
```
for token in ['stress', 'horses', 'policies', 'entries']: # Test our stemmer
    print(token, "\t:", s_stemmer(token))
```

```
stress  : stress
horses  : horse
policies        : policy
entries         : entry
```

It works. Now we can stem our SMS messages.

[7]:
```
for i in range(len(msg_token)): # For all msgs
    msg_token[i] = [s_stemmer(token) for token in msg_token[i]] # Stem every SMS
```

Now we can remove the stopwords. As a first attempt, we will store them in a list and check the computation time.

[8]:
```
import time

stopwords_list = [] # Init
start = time.time() # Start time

with open("stopwords.txt") as f:
    stopwords = f.readlines() # Read every word as an element in the list
    for word in stopwords:
        stopwords_list.append(word.strip()) # Append and delete whitespaces
    f.close()
```

```
stop = time.time() # End time
print("Duration of the run (list):", stop - start)
```

```
Duration of the run (list): 0.00045752525329589844
```

We will now try with a dictionary. The best approach is to create a hash table of stopwords.

[9]:
```
stopwords_dict = {} # Init
start = time.time() # Start time

with open("stopwords.txt") as f:
    stopwords = f.readlines() # Read every word as an element in the list
    for word in stopwords:
        key, val = stopwords.index(word), word.strip() # Append key and value
        stopwords_dict[int(key)] = val # Create hash table entry
        f.close()

stop = time.time() # End time
print("Duration of the run (dict):", stop - start)
```

```
Duration of the run (dict): 0.00025153160095214844
```

The dictionary was the fastest approach. We can then use the hash table to remove the stopwords from our SMS texts.

[10]:
```
stop_token = msg_token.copy()

for i in range(len(stop_token)):
    stop_token[i] = [token for token in stop_token[i] if token not in␣
 ↪stopwords_dict.values()] # Remove stopwords from SMS (dict)

print({'With stopwords': msg_token[0], 'Without stopwords': stop_token[0]})
```

```
{'With stopwords': ['hope', 'you', 'are', 'having', 'a', 'good', 'week', 'just',
'checking', 'in'], 'Without stopwords': ['hope', 'good', 'week', 'checking']}
```

3 Question 3: Apply the Naïve Bayes classifier to solve the spam detection problem. You can use the "sms-spam-train.csv" dataset to train your classifier and the "sms-spam-test.csv" dataset to evaluate your system. As features, you can consider the n most frequent words per category.

First, we load the datasets and apply the same preprocessing of the previous two questions.

[11]:
```
spam_train = pd.read_csv('sms-spam-train.csv', sep=',') # Spam train dataset
spam_test = pd.read_csv('sms-spam-test.csv', sep=',') # Spam test dataset
```

```
messages_train, messages_test = [], []
for msg in spam_train['text']:
    messages_train.append(msg) # Create list of SMS texts
for msg in spam_test['text']:
    messages_test.append(msg) # Create list of SMS texts

msg_token_train = [tokenize_text(msg, 'english') for msg in messages_train] #␣
 ↪Tokenize every SMS
msg_token_test = [tokenize_text(msg, 'english') for msg in messages_test] #␣
 ↪Tokenize every SMS

for i in range(len(msg_token_train)): # For all msgs
    msg_token_train[i] = [s_stemmer(token) for token in msg_token_train[i]] #␣
 ↪Stem every SMS
for i in range(len(msg_token_test)): # For all msgs
    msg_token_test[i] = [s_stemmer(token) for token in msg_token_test[i]] #␣
 ↪Stem every SMS

for i in range(len(msg_token_train)):
    msg_token_train[i] = [token for token in msg_token_train[i] if token not in␣
 ↪stopwords_dict.values()] # Remove stopwords from SMS (dict)
for i in range(len(msg_token_test)):
    msg_token_test[i] = [token for token in msg_token_test[i] if token not in␣
 ↪stopwords_dict.values()] # Remove stopwords from SMS (dict)

print(len(msg_token_train), len(msg_token_test))
```

```
5199 361
```

Now we create two DTM (Document-Term Matrices) that we will use as an input for our Naive Bayes classifier.

[12]:
```
import collections # Library to simplify tallies

def extract_vocabulary(tokenized_corpus, min_count=1, max_count=float('inf')):
    """
    Extract vocabulary from input tokenized text.
    Min frequency count of a vocabulary item is set to 1 and max to infinite.
    """
    vocab = collections.Counter() # Create a container object for rapid tallies
    for document in tokenized_corpus:
        vocab.update(document) # Update for each play
    vocab = {word for word, count in vocab.items()
             if min_count <= count <= max_count} # Append only if the word␣
 ↪frequency is between the boundaries
```

```
    return sorted(vocab) # Return a list alphabetically ordered of unique words␣
 ↪in the corpus

vocabulary_train = extract_vocabulary(msg_token_train) # Build the vocabulary
vocabulary_test = extract_vocabulary(msg_token_test) # Build the vocabulary
```

[13]:
```
import numpy as np

def corpus2dtm(tokenized_corpus, vocab):
    """
    Custom function to transform a tokenized corpus into a document-term matrix.
    """
    dtm = []
    for document in tokenized_corpus: # For each play
        document_counts = collections.Counter(document) # Get counters
        row = [document_counts[word] for word in vocab] # Count tf for each␣
 ↪word in the vocabulary
        dtm.append(row) # Append row
    return dtm


dtm_train = np.array(corpus2dtm(msg_token_train, vocabulary_train)) # Build the␣
 ↪DTM
dtm_test = np.array(corpus2dtm(msg_token_test, vocabulary_test)) # Build the DTM
```

We check the sizes of our DTMs.

[14]:
```
print(f'Document-term matrix (train) with {dtm_train.shape[0]} documents and␣
 ↪{dtm_train.shape[1]} words.')
print(f'Document-term matrix (test) with {dtm_test.shape[0]} documents and␣
 ↪{dtm_test.shape[1]} words.')
```

```
Document-term matrix (train) with 5199 documents and 8349 words.
Document-term matrix (test) with 361 documents and 1666 words.
```

Now we are set up to train our Naive Bayes classifier. As features, we can select the top n words that are present both in the train and test set.

[15]:
```
train = pd.DataFrame(dtm_train, columns=vocabulary_train) # Train set from DTM
test = pd.DataFrame(dtm_test, columns=vocabulary_test) # Test set from DTM
```

Let's start with $n = 100$.

[16]:
```
train_top_100 = train.sum(axis=0, numeric_only=True)
train_top_100 = train_top_100.sort_values(ascending=False).index[:100].tolist()␣
 ↪# Top-100 train

test_top_100 = test.sum(axis=0, numeric_only=True)
```

```
test_top_100 = test_top_100.sort_values(ascending=False).index[:100].tolist() #␣
 ↪Top-100 test
```

Now we select as features the n words that are in both lists.

[17]:
```
features = []

for i in range(len(train_top_100)):
    if train_top_100[i] in test_top_100:
        features.append(train_top_100[i])
    else:
        pass

print(len(features))
```

```
75
```

Then we have 75 feature words. We can finally create our Naive Bayes classifier.

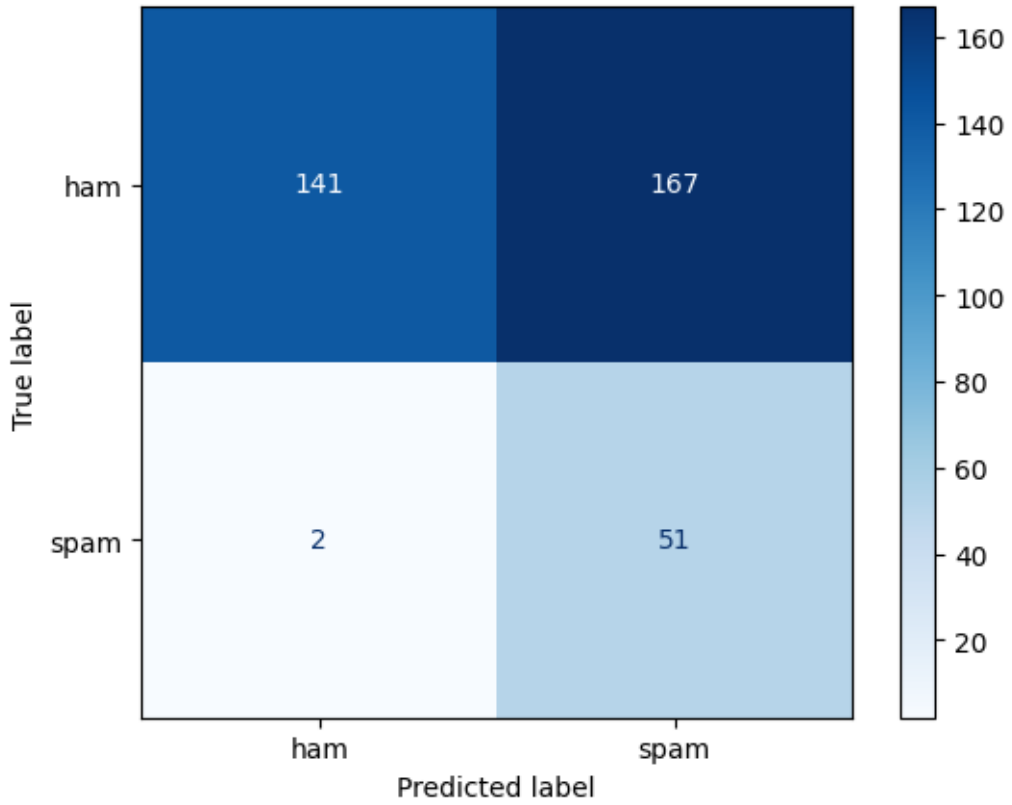
[18]:
```
from sklearn.naive_bayes import GaussianNB # Import NB

x_train = train[features] # Features for train
y_train = spam_train['type'] # Predicting variable for train
x_test = test[features] # Features for test
y_test = spam_test['type'] # Ground truth

model = GaussianNB() # Init Naive Bayes classifier
model.fit(x_train, y_train) # Fit NB on train
y_pred = model.predict(x_test) # Predict on test set
```

To visualize our predictions, we can plot a confusion matrix. In this matrix, we will have the prediction on the x-axis and the ground truth on the y-axis.

[19]:
```
import matplotlib.pyplot as plt # Lib to plot
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay #␣
 ↪Confusion matrix
cm = confusion_matrix(y_test, y_pred, labels=model.classes_) # Define a␣
 ↪confusion matrix using model classes (authors) as labels
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=model.
 ↪classes_) # Plot the matrix
disp.plot(cmap=plt.cm.Blues) # Plot in scale of blues for better visualization
plt.show()
```



As we can see, the model classified correctly 192 messages out of 361. Therefore, the accuracy of our model is 53.2%.