Lab 8 – Text Clustering and Classification

This lab is based on the Kaggle tutorial: Bag of Words Meets Bags of Popcorn which focuses on **sentiment analysis**. Sentiment analysis is a challenging subject in machine learning. People express their emotions in language that is often obscured by sarcasm, ambiguity, and plays on words, all of which could be very misleading for both humans and computers. A basic task in sentiment analysis is classifying the polarity of a given text at the document, sentence, or feature/aspect level – whether the expressed opinion in a document, a sentence or an entity feature/aspect is positive, negative, or neutral.

I. Dataset

We will use a dataset of 50,000 IMDB movie reviews, specially selected for sentiment analysis. The sentiment of reviews is binary, meaning the IMDB rating < 5 results in a sentiment score of 0, and rating >=7 have a sentiment score of 1. No individual movie has more than 30 reviews. The 25,000 review labeled training set does not include any of the same movies as the 25,000 review test set. In addition, there are another 50,000 IMDB reviews provided without any rating labels.

File descriptions

- **labeledTrainData** - The labeled training set. The file is tab-delimited and has a header row followed by 25,000 rows containing an id, sentiment, and text for each review.
- **testData** - The test set. The tab-delimited file has a header row followed by 25,000 rows containing an id and text for each review. Your task is to predict the sentiment for each one.
- **unlabeledTrainData** - An extra training set with no labels. The tab-delimited file has a header row followed by 50,000 rows containing an id and text for each review.
- **sampleSubmission** - A comma-delimited sample submission file in the correct format.

Data fields

- **id** - Unique ID of each review
- **sentiment** - Sentiment of the review; 1 for positive reviews and 0 for negative reviews
- **review** - Text of the review

II. Introduction

We will go through part 1 of the Kaggle tutorial which covers basic natural language processing (NLP) techniques. NLP is a set of techniques for approaching text problems. This lab will help you get started with loading and cleaning the IMDB movie reviews, then applying (a) a simple Bag of Words model and (b) a TF/IDF model to get surprisingly accurate predictions of whether a review is thumbs-up or thumbs-down.

This part of the tutorial is not platform dependent. Throughout this tutorial we'll be using various Python modules for text processing, clustering, and random forests.

There are many good tutorials, and indeed entire books written about NLP and text processing in Python. This tutorial is in no way meant to be exhaustive - just to help get you started with the movie reviews.

III. Reading the Data

The first file that you'll need is unlabeledTrainData.tsv, which contains 25,000 IMDB movie reviews, each with a positive or negative sentiment label.

Next, read the tab-delimited file into Python. To do this, we can use the pandas package, introduced in the Titanic tutorial, which provides the read_csv function for easily reading and writing data files. The pandas library is already pre-installed within anaconda.

```
# Import the pandas package, then use the "read_csv" function to
read the labeled training data

import pandas as pd
train = pd.read_csv("labeledTrainData.tsv", header=0, \
                    delimiter="\t", quoting=3)
```

Here, `"header=0"` indicates that the first line of the file contains column names, `"delimiter=\t"` indicates that the fields are separated by tabs, and `quoting=3` tells Python to ignore doubled quotes, otherwise you may encounter errors trying to read the file.

We can make sure that we read 25,000 rows and 3 columns as follows:

```
print(train.shape)
# (25000, 3)
print(train.columns.values)
# array(['id', 'sentiment', 'review'], dtype=object)
```

The three columns are called `"id"`, `"sentiment"`, and `"review"`. Now that you've read the training set, take a look at a few reviews:

```
print(train["review"][0])
```

As a reminder, this will show you the first movie review in the column named `"review"`. You should see a review that starts like this:

```
"With all this stuff going down at the moment with MJ i've started
listening to his music, watching the odd documentary here and there,
watched The Wiz and watched Moonwalker again. Maybe i just want to get
a certain insight into this guy who i thought was really cool in the
eighties just to maybe make up my mind whether he is guilty or
innocent. Moonwalker is part biography, part feature film which i
remember going to see at the cinema when it was originally released.
Some of it has subtle messages about MJ's feeling towards the press
and also the obvious message of drugs are bad m'kay. <br/><br/>..."
```

There are HTML tags such as "
", abbreviations, punctuation - all common issues when processing text from online. Take some time to look through other reviews in the training set while you're at it - the next section will deal with how to tidy up the text for machine learning.

IV. Data Cleaning and Text Preprocessing

Removing HTML Markup: The BeautifulSoup Package

First, we'll remove the HTML tags. For this purpose, we'll use the Beautiful Soup library.

```
# Import BeautifulSoup into your workspace
from bs4 import BeautifulSoup

# Initialize the BeautifulSoup object on a single movie review
example1 = BeautifulSoup(train["review"][0], "html.parser")

# Print the raw review and then the output of get_text(), for
# comparison
print(train["review"][0])
print(example1.get_text())
```

Calling `get_text()` gives you the text of the review, without tags or markup. If you browse the BeautifulSoup documentation, you'll see that it's a very powerful library - more powerful than we need for this dataset. However, it is not considered a reliable practice to remove markup using regular expressions, so even for an application as simple as this, it's usually best to use a package like BeautifulSoup.

V. Dealing with Punctuation, Numbers, Tokenization, Stop words and Stemming: NLTK and regular expressions

When considering how to clean the text, we should think about the data problem we are trying to solve. For many problems, it makes sense to remove punctuation. On the other hand, in this case, we are tackling a sentiment analysis problem, and it is possible that "!!!" or ":-(" could carry sentiment, and should be treated as words. In this tutorial, for simplicity, we remove the punctuation altogether, but it is something you can play with on your own.

Similarly, in this tutorial we will remove numbers, but there are other ways of dealing with them that make just as much sense. For example, we could treat them as words, or replace them all with a placeholder string such as "NUM".

To remove punctuation and numbers, we will use a package for dealing with regular expressions, called `re`. The package comes built-in with Python; no need to install anything. For a detailed description of how regular expressions work, see the package documentation. Now, try the following:

```
import re
# Use regular expressions to do a find-and-replace
```

```
letters_only = re.sub("[^a-zA-Z]",    # The pattern to search for
                      " ",      # The pattern to replace it with
                      example1.get_text() ) # The text to search
print(letters_only)
```

A full overview of regular expressions is beyond the scope of this tutorial, but for now it is sufficient to know that [] indicates group membership and ^ means "not". In other words, the re.sub() statement above says, "Find anything that is NOT a lowercase letter (a-z) or an upper case letter (A-Z), and replace it with a space."

We'll also convert our reviews to lower case and split them into individual words (called "tokenization" in NLP lingo) in order to create a list of words (vocabulary):

```
lower_case = letters_only.lower()        # Convert to lower case
words = lower_case.split()               # Split into words
print(words)
```

The process of tokenization can be performed using the Python Natural Language Toolkit (NLTK) instead of using `split()`. You'll need to install the library if you don't already have it on your computer; you'll also need to install the data packages that come with it, as follows:

```
import nltk
# Download punkt english tokenizer
nltk.download('punkt')
# first tokenize by sentence, then by word
nltk_words = [word for sent in nltk.sent_tokenize(lower_case)
for word in nltk.word_tokenize(sent)]
```

Finally, we need to decide how to deal with frequently occurring words that don't carry much meaning. Such words are called "stop words"; in English they include words such as "a", "and", "is", and "the". Conveniently, there are Python packages that come with stop word lists built in. Let's import a stop word list from NLTK:

```
# Download stop words
nltk.download('stopwords')
```

Now we can use nltk to get a list of stop words:

```
from nltk.corpus import stopwords # Import the stop word list
print(stopwords.words("english"))
```

This will allow you to view the list of English-language stop words. To remove stop words from our movie review, do:

```
# Remove stop words from "words" using list comprehension
filtered_words = [w for w in words if not w in
stopwords.words("english")]
print(filtered_words)
```

This looks at each word in our "filtered_words" list, and discards anything that is found in the list of stop words. After all of these steps, your review should now begin something like this:

```
['stuff', 'going', 'moment', 'mj', 'started', 'listening', 'music',
'watching', 'odd', 'documentary', 'watched', 'wiz', 'watched',
'moonwalker', 'maybe', 'want', 'get', 'certain', 'insight', 'guy',
'thought', 'really', 'cool', 'eighties', 'maybe', 'make', 'mind',
'whether', 'guilty', 'innocent', 'moonwalker', 'part', 'biography',
'part', 'feature', 'film', 'remember', 'going', 'see', 'cinema',
'originally', 'released', 'subtle', 'messages', 'mj', 'feeling',
'towards', 'press', 'also', 'obvious', 'message', 'drugs', 'bad',
'kay'...]
```

Don't worry if you see a "u" before each word; it just indicates that Python is internally representing each word as a unicode string.

The next thing to do to the data is to break each word of the filtered_words list to its root. For example, Snowball Stemming and Lemmatizing (both available in NLTK) would allow us to treat "messages", "message", and "messaging" as the same word, which is certainly useful. We can use the NLTK Porter Stemmer as follows:

```
# load nltk's SnowballStemmer
from nltk.stem.snowball import SnowballStemmer
stemmer = SnowballStemmer('english')
# Provide stemming on "filtered_words" list
stemmed_words = [stemmer.stem(w) for w in filtered_words]
print(stemmed_words)
```

There is another NLTK stemmer caller PorterStemmer.

Putting it all together

Now we have code to clean one review - but we need to clean 25,000 training reviews! To make our code reusable, let's create a function that can be called many times:

```
# Stemming is the process of breaking a word down into its root.
stemmer = SnowballStemmer('english')
# In Python, searching a set is much faster than searching
# a list, so convert the stop words to a set
stops = set(stopwords.words("english"))

def review_to_words( raw_review ):
    # Function to convert a raw review to a string of words
    # The input is a single string (a raw movie review), and
    # the output is a single string (a pre-processed movie
    # review)
    #
    # 1. Remove HTML
    review_text = BeautifulSoup(raw_review,
```

```
"html.parser").get_text()
    #
    # 2. Remove non-letters
    letters_only = re.sub("[^a-zA-Z]", " ", review_text)
    #
    # 3. Convert to lower case, split into individual words
    words = letters_only.lower().split()
    #
    # 4. Remove stop words
    meaningful_words = [w for w in words if not w in stops]
    #
    # 5. Stem words
    stemmed_meaningful_words = [stemmer.stem(w) for w in
meaningful_words]
    #
    # 6. Join the words back into one string separated by space,
    # and return the result.
    return( " ".join( stemmed_meaningful_words ))
```

Two elements here are new: First, we converted the stop word list to a different data type, a set. This is for speed; since we'll be calling this function tens of thousands of times, it needs to be fast, and searching sets in Python is much faster than searching lists.

Second, we joined the words back into one paragraph. This is to make the output easier to use in Bag of Words and TF/IDF models, below. After defining the above function, if you call the function for a single review:

```
clean_review = review_to_words( train["review"][0] )
print(clean_review)
```

it should give you exactly the same output as all of the individual steps we did in preceding tutorial sections. Now let's loop through and clean all of the training set at once (this might take a few minutes depending on your computer):

```
# Get the number of reviews based on the dataframe column size
num_reviews = train["review"].size
# Initialize an empty list to hold the clean reviews
clean_train_reviews = []
# Loop over each review; create an index i that goes from 0 to
# the length of the movie review list
for i in range( 0, num_reviews ):
    # Call our function for each one, and add the result to the
    # list of clean reviews
    clean_train_reviews.append( review_to_words(
train["review"][i] ) )
```

Sometimes it can be annoying to wait for a lengthy piece of code to run. It can be helpful to write code so that it gives status updates. To have Python print a status update after every 1000 reviews that it processes, try adding a line or two to the code above:

```
print("Cleaning and parsing the training set movie
reviews...\n")
clean_train_reviews = []
for i in range( 0, num_reviews ):
    # If the index is evenly divisible by 1000, print a message
    if( (i+1)%1000 == 0 ):
        print("Review %d of %d\n" % ( i+1, num_reviews ))
    # Call our function for each one, and add the result to the
    # list of clean reviews
    clean_train_reviews.append( review_to_words(
train["review"][i] ) )
```

VI. Creating Features from a Bag of Words (Using scikit-learn)

Now that we have our training reviews tidied up, how do we convert them to some kind of numeric representation for machine learning? One common approach is called a Bag of Words. The Bag of Words model learns a vocabulary from all of the documents, then models each document by counting the number of times each word appears. For example, consider the following two sentences:

Sentence 1: "The cat sat on the hat"

Sentence 2: "The dog ate the cat and the hat"

From these two sentences, our vocabulary is as follows:

{ the, cat, sat, on, hat, dog, ate, and }

To get our bags of words, we count the number of times each word occurs in each sentence. In Sentence 1, "the" appears twice, and "cat", "sat", "on", and "hat" each appear once, so the feature vector for Sentence 1 is:

{ the, cat, sat, on, hat, dog, ate, and }

Sentence 1: { 2, 1, 1, 1, 1, 0, 0, 0 }

Similarly, the features for Sentence 2 are: { 3, 1, 0, 0, 1, 1, 1, 1}

In the IMDB data, we have a very large number of reviews, which will give us a large vocabulary. To limit the size of the feature vectors, we should choose some maximum vocabulary size. Below, we use the 5000 most frequent words (remembering that stop words have already been removed).

We'll be using the feature_extraction module from scikit-learn to create bag-of-words features. If you did the Random Forest tutorial in the Titanic competition, you should already have scikit-learn installed; otherwise you will need to install it.

```
print("Creating the bag of words...\n")
from sklearn.feature_extraction.text import CountVectorizer

# Initialize the "CountVectorizer" object, which is scikit-
learn's bag of words tool.
vectorizer = CountVectorizer(analyzer = "word",   \
                             tokenizer = None,    \
                             preprocessor = None, \
                             stop_words = None,   \
                             max_features = 5000)

# fit_transform() does two functions: First, it fits the model
# and learns the vocabulary; second, it transforms our training
# data into feature vectors.
# Input to fit_transform(): a list of strings
# Output: a document-term sparse matrix [n_samples, n_features]
train_data_features =
vectorizer.fit_transform(clean_train_reviews)
# Numpy arrays are easy to work with, so convert the result to
an array
train_data_features = train_data_features.toarray()
```

To see what the training data array now looks like, do:

```
print(train_data_features.shape) # (25000, 5000)
```

It has 25,000 rows and 5,000 columns-features (one for each vocabulary word).

$$\begin{array}{c} \\ D_1 \\ D_2 \\ \vdots \\ D_n \end{array} \begin{array}{c} \begin{array}{ccc} T_1 & \dots & T_t \end{array} \\ \begin{vmatrix} d_{11} & \dots & d_{1t} \\ d_{21} & \dots & d_{2t} \\ \vdots & \ddots & \vdots \\ d_{n1} & \dots & d_{nt} \end{vmatrix} \end{array}, where\ d_{ij} = value\ of\ T_j\ in\ D_i$$

Note that CountVectorizer comes with its own options to automatically do preprocessing, tokenization, and stop word removal -- for each of these, instead of specifying "None", we could have used a built-in method or specified our own function to use. See the function documentation for more details. However, we wanted to write our own function for data cleaning in this tutorial to show you how it's done step by step.

Now that the Bag of Words model is trained, let's look at the vocabulary:

```
# Take a look at the words in the vocabulary
vocab = vectorizer.get_feature_names()
print(vocab)
```

If you're interested, you can also print the counts of each word in the vocabulary:

```
import numpy as np
# Sum up the counts of each vocabulary word
dist = np.sum(train_data_features, axis=0)
# For each, print the vocabulary word and the number of times it
# appears in the training set
for tag, count in zip(vocab, dist):
    print(count, tag)
```

VII. Creating Features from TF/IDF Vectorizer (Using scikit-learn)

Beyond the Bag of Words model, the term frequency/inverse document frequency (TF/IDF) model learns a vocabulary from all of the documents, then models each document by calculating a numerical statistic for each word of the document that reflects how important the word is to the document.

```
from sklearn.feature_extraction.text import TfidfVectorizer
print("Creating the tf/idf...\n")
# Initialize the "TfidfVectorizer" object, which is scikit-
learn's tf/idf tool.
tfidf_vectorizer = TfidfVectorizer(max_df=0.8,          \
                                   max_features=5000, \
                                   min_df=0.2,          \
                                   stop_words=None,     \
                                   use_idf=True,        \
                                   tokenizer=None,      \
                                   ngram_range=(1,3))
# Tf-idf-weighted term-document sparse matrix
tfidf_train_data_features =
tfidf_vectorizer.fit_transform(clean_train_reviews)
# Convert the result to nampy array
tfidf_train_data_features = tfidf_train_data_features.toarray()
print(tfidf_train_data_features.shape) # (25000, 48)
# Take a look at the words in the vocabulary
tfidf_vocab = tfidf_vectorizer.get_feature_names()
print(tfidf_vocab)
```

Note that TfidfVectorizer comes with options to limit the number of features by setting a "max_features" option, by ignoring terms that have a document frequency strictly lower than the "min_df" (cut-off) threshold and/or by ignoring terms that have a document frequency strictly higher than the "max_df" threshold (corpus-specific stop words). See the function documentation for more details.

VIII. Creating Features from TF/IDF Vectorizer (Using scikit-learn)

At this point, we have numeric training features from the Bag of Words model and the TF/IDF model so we can do clustering (if sentiment information was not available) or classification (if sentiment information is not available).

Clustering

Let's say that the sentiment information was not available, we can run the kmeans clustering algorithm using k=2.

```
from sklearn.cluster import KMeans
# Perform k-means clustering
num_clusters = 2
kmeans = KMeans(n_clusters=num_clusters)
# Use k-keans to make sentiment label clustering
kmeans.fit(tfidf_train_data_features)
# get cluster assignments; a label (0 or 1) for each review
labels = kmeans.labels_.tolist()
```

In order to get a good sense of the main topic of each cluster, we can identify which are the top n words that are nearest to the cluster center (centroid).

```
# Fancy indexing and sorting on each cluster to identify which
# are the top n words that are nearest to the cluster centroid.
# This gives a good sense of the main topic of the cluster.
print("Top terms per cluster:")
print()
# Find which features contribute more to each cluster center.
# Sort in descending order by feature contribution and return
indexes. For example:
# ascending: np.array([3, 4, 10, 1, 8]).argsort() returns
array([3, 0, 1, 4, 2])
# descending: np.array([3, 4, 10, 1, 8]).argsort[::-1] returns
array([2, 4, 1, 0, 3])
sorted_index_centroids = kmeans.cluster_centers_.argsort()[:,
::-1]

for i in range(num_clusters):
    print("Cluster %d words:" % i, end='')
    #replace 10 with n words per cluster
    for ind in sorted_index_centroids[i, :10]:
        print(' %s' % tfidf_vocab[ind].split(' ')[0], end=',')
    print() #add whitespace
    print() #add whitespace
```

Classification

Since the sentiment information is available, we can do some supervised learning. Here, we'll use the Random Forest classifier. The Random Forest algorithm is included in scikit-learn (Random Forest uses many tree-based classifiers to make predictions, hence the "forest"). Below, we set the

number of trees to 100 as a reasonable default value. More trees may (or may not) perform better, but will certainly take longer to run. Likewise, the more features you include for each review, the longer this will take.

Prior running the classifier, we split both the tfidf features (tfidf_train_data_features) and the real (known) sentiment labels (train['sentiment']) into random train and test subsets using a dedicated sklearn function train_test_split .

```
from sklearn.model_selection import train_test_split

# split features and labels (e.g. test = 20%, training = 80% )
# try both bag of words and tfidf features
x_train, x_test, y_train, y_test =
train_test_split(tfidf_train_data_features,
train['sentiment'].values, test_size=0.2)

print("Training the random forest...")
from sklearn.ensemble import RandomForestClassifier
# Initialize a Random Forest classifier with 100 trees
forest = RandomForestClassifier(n_estimators = 100)
# Fit the forest to the training set, using the tfidf as
# features and the sentiment labels as the response variable
# This may take a few minutes to run
forest_model = forest.fit( x_train, y_train )
```

After training the random forest model, we can try to predict the sentiment labels (y_pred) on the unseen portion of data features (x_test). The accuracy of the prediction model will be tested using the hamming loss metric that takes into account the predicted sentiment values (y_pred) and the true sentiment values (y_test). The Hamming loss is the fraction of labels that are incorrectly predicted. Therefore, the hamming loss value will range from 0 (low loss) to 1 (high loss), whereas 1-hamming_loss will give the accuracy of the prediction model. Other related metrics are shown here: http://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics.

```
# Use the random forest model to make sentiment label
predictions
y_pred = forest_model.predict( x_test )

# evaluate accuracy using hamming loss metric
from sklearn.metrics import hamming_loss
print(1-hamming_loss(y_pred, y_test))
```

Beyond the random forest classifier, there exist a number of classifying methods like k nearest neighbors classifier, support vector classifier (sklearn.svm.SVC), etc. A long list of supervised learning techniques supported by sklearn library is shown here: http://scikit-learn.org/stable/supervised_learning.html.