M2 MAP670
Lab: 19 October 2015
PART II: Opinion Mining on Movie Reviews

Fragkiskos Malliaros, Christos Giatsidis, Michalis Vazirgiannis, Apostolos N. Papadopoulos

1 Description of the Lab

The goal of this part of the lab is to work with textual data, applying data mining techniques. The basic objective of *text mining* concerns the discovery and extraction of relevant and valuable information from large volumes of text data. More precisely, we will focus on the *opinion mining* problem (also known as sentiment analysis), which refers to the use of natural language processing, text analysis and machine learning tools in order to identify and extract subjective information from text corpora.

Our goal is to identify *positive* and *negative* opinions in reviews expressed by natural language (i.e., text) about a specific product. Opinion mining can be useful in several ways. It can help a company to evaluate the success of an ad campaign or of new product, determine which versions of a product or service are popular and identify which demographics like or dislike particular product features.

There are several challenges in the problem of opinion mining. The first one concerns the fact that a specific word used to describe a product can be considered either as positive or as negative depending the context or the product. Let's take as example a review about a laptop that contains the word "long". If a customer said that the battery life of the laptop was long, that would be a positive opinion. However, if the customer said that the laptop's start-up time was long, that would be a negative opinion. This example indicates that an opinion system, trained to gather opinions on one type of product or product's feature, may not perform very well on another. A second challenge is that people do not always express their opinions the same way. Most traditional text processing relies on the fact that small differences between two pieces of text do not change the meaning very much. However, in opinion mining, "the laptop was great" is very different from "the laptop was not great". Lastly, people can combine contradictory statements in their reviews, where both positive and negative comments are written in the same sentence. Although this is easy for a human to understand, it is more difficult for a computer to parse.

In the context of this lab, we will analyze reviews about movies (e.g., obtained from *IMDb*[1], an online database of information related to films and television programs). We will follow a text categorization (or classification) approach to infer if the review is positive or negative (i.e., classify a review text as positive or negative). Next, we briefly describe the reviews dataset that will be used and then we present the steps of the pipeline that need to be performed for the sentiment analysis task.

2 Dataset Description

The dataset that will be used in the lab concerns movie reviews. As you may have observed, in movie review websites like IMDb, any user can write its own review for a particular movie or television show. For example, Fig. 1 shows the review of the movie *Midnight in Paris* given by a user. As we see, the opinion of this user for the movie is positive. Here, we will consider such a dataset, where each review has been characterized as *positive* or *negative*. The data is given in the `movie_reviews.csv` file. The file is composed by $1,959$ reviews (one per line) and each one has been characterized as positive or negative. For example, Fig. 2 shows the first entry of the file. As you can observe, this review has been categorized as positive.

[1] `http://www.imdb.com/`

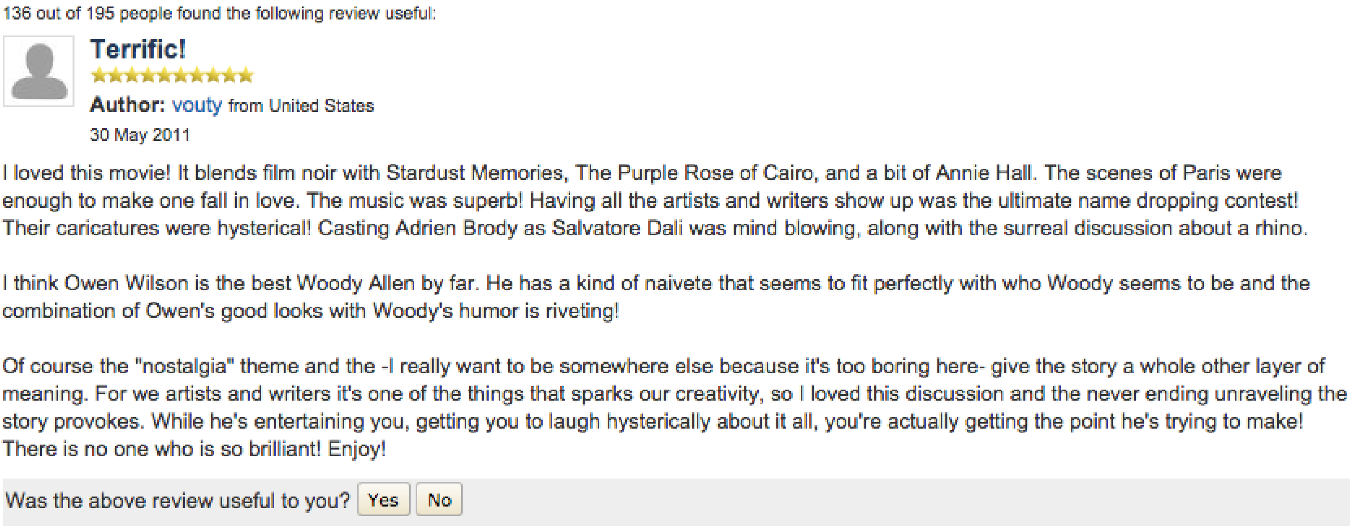

Figure 1: Example review of the movie *Midnight in Paris* in IMDb.

```
"This movie is really not all that bad. But then again, this movie genre is right down my alley.
Sure, the sets are cheap, but they really did decent with what they had. If you like cheap,
futuristic, post-apocalyptic B movies, then you'll love this one!! I sure did!",positive
```

Figure 2: Example review from the dataset.

3 Description of the Task and the Pipeline

Our goal is to build a system that, given the review text of a movie, it will be able to predict if the opinion of the user is positive or negative. We will treat the problem as a classification one, where the goal is predict the class label, i.e., positive or negative, for a given review. In the description that follows, we use the word "document" to refer to text reviews. The outline of the task is given in Fig. 3.

The pipeline that typically is followed to deal with the problem is similar to the one applied in any classification problem; the goal is to learn the parameters of a classifier from a collection of training documents (i.e., those reviews that we know if they are positive or negative) and then to predict the class of unlabeled documents. The first step in text categorization is to transform documents, which typically are strings of characters, into a representation suitable for the learning algorithm and the classification task. Here, we will employ the *Vector Space Model*, a spatial representation of text documents. In this model, each document is represented by a vector in a n-dimensional space, where each dimension corresponds to a term (i.e., word) from the overall vocabulary of the given document collection. More formally, let $\mathcal{D} = \{d_1, d_2, \ldots, d_m\}$ denote a collection of m documents, and $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ be the dictionary, i.e., the set of words in the corpus $\mathcal{D}$. As we will describe later, $\mathcal{T}$ is obtained by applying some standard natural language processing techniques, such as stop words removal and stemming. Each document $d_i \in \mathcal{D}$ is represented as a vector of term weights $d_i = \{w_{i,1}, w_{i,2}, \ldots, w_{i,n}\}$, where $w_{i,k}$ is the weight of term k in document d_i. That way, data can be represented by the *Document-Term matrix* **DT** of size $m \times n$, where the rows correspond to documents and the columns to the different terms (i.e., features) of set $\mathcal{T}$. Additionally, each document d_i is associated with a class label $y_i = \{+, -\}$ (i.e., positive/negative opinion) forming the class vector $\mathbf{Y}$, and the goal is to predict the class labels for a set of test documents. Based on this formulation, traditional classification algorithms (e.g., SVMs, Logistic Regression, Random Forests, etc.) can be applied to predict the category label of test documents.

Next, we describe the tasks that need to be performed for each each part, providing also parts of Python code.

3.1 Load the data

Initially, we should load the data contained in the `movie_reviews.csv` file. Later on, we will describe how part of this dataset will be used to train (i.e., build) our model and then, to test it. To import the data, we have to parse the .csv file line by line; recall that each line contains the text review and the opinion (positive/negative). To do that, we will use Python's built-in function `csv.reader(csv_file, delimiter=',',quotechar='"')`. The main

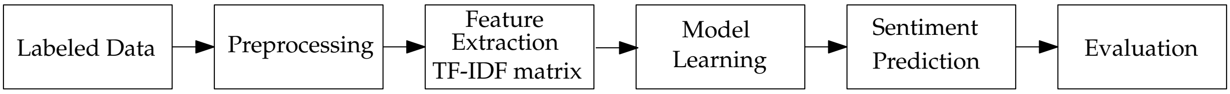

Figure 3: Pipeline of the sentiment analysis task.

point here is to split each line of the file into two parts, the one that corresponds to the text review (list `data`) and the other to the opinion (list `labels`). Next, we give the code for this part.

```
with open('movie_reviews.csv') as csv_file:
        reader = csv.reader(csv_file, delimiter=',',quotechar='"')
        # Initialize lists for data and class labels
        data =[]
        labels = []
        # For each row of the csv file
        for row in reader:
        # skip missing data
                if row[0] and row[1]:
                        #data.append(row[0])
                        data.append(row[0].decode('utf-8'))
                        y_label = -1 if row[1]=='negative' else 1
                        labels.append(y_label)
```

3.2 Data preprocessing

Before applying any learning algorithm to the data, it is necessary to apply some preprocessing tasks as shown below:

- Remove punctuation marks (e.g, . , ? : ( ) [ ])[2] and transform all characters to lowercase. This can be done using Python's *NLTK* library (`http://www.nltk.org/`) as follows:

```
# Remove punctuation and lowercase characters
punctuation = set(string.punctuation)
doc = ''.join([w for w in text.lower() if w not in punctuation])
```

- Remove stop words. These are words that are filtered out before processing any natural language data. This set of words does not offer information about the content of the document and typically corresponds to a set of commonly used words in any language. For example, in the context of a search engine, suppose that your search query is "how to categorize documents". If the search engine tries to find web pages that contain the terms "how", "to", "categorize", "documents", it will find many more pages that contain the terms "how" and "to" than pages that contain information about document categorization. This is happening because the terms "how" and "to" are commonly used in the English language.

  Task to be done. In the code, we provide a list of stop words that should be removed (list `stopwords`). Fill in your Python code to remove these stop words from all the documents.

- The third preprocessing step is the one of stemming[3], i.e., the process of reducing the words to their word stem or root. For example, a stemming algorithm reduces the words "fishing", "fished", and "fisher" to the root word, "fish".

  Task to be done. In the lab, we will use *Porter's* stemmer, contained in the NLTK library. Use the following code to perform stemming:

```
# Stemming
stemmer = PorterStemmer()
doc = [stemmer.stem(w) for w in doc]
```

[2]Wikipedia's lemma for *Punctuation*: `http://en.wikipedia.org/wiki/Punctuation`.
[3]Wikipedia's lemma for *Stemming*: `http://en.wikipedia.org/wiki/Stemming`.

3.3 Feature extraction and the TF-IDF matrix

After applying the preprocessing step, the text data (i.e., all the possible documents-reviews) should be transformed to a format that will be used in the learning (i.e., classification) task. As we describe above, the data will be represented by the Document-Term matrix, where the rows correspond to the different documents of the collection (i.e., reviews) and the columns to the features, which in our case are the different terms (i.e., words). Here, we are interested to find relevant weighting criteria for the Document-Term matrix, i.e., assign a relevance score w_{ij} to each term t_j for each document d_i as shown in Fig. 4.

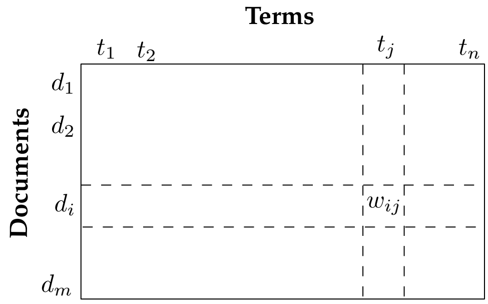

Figure 4: Schematic representation of the Document-Term matrix.

We will consider the *Bag-of-Words* representation of the documents. In this case, the importance of a term $t \in \mathcal{T}$ within a document $d \in \mathcal{D}$ is based on the frequency $tf(t,d)$ of the term in the document (TF). Furthermore, terms that occur frequently in one document but rarely in the rest of the documents, are more likely to be relevant to the topic of the document. This is known as the inverse document frequency (IDF) factor, and is computed at the collection level. It is obtained by dividing the total number of documents by the number of documents containing the term, and then taking the logarithm of that quotient, as follows:

$$idf(t,\mathcal{D}) = \log \frac{m}{|\{d \in \mathcal{D} : t \in d|\}},$$

where m is the total number of documents in collection $\mathcal{D}$, and the denominator captures the number of documents that term t appears. Having computed the TF and IDF metrics, we can combine them to get the TF-IDF score:

$$tf\text{-}idf(t,d,\mathcal{D}) = tf(t,d) \times idf(t,\mathcal{D}).$$

This function captures the intuitions that (i) the more often a term occurs in a document, the more it is representative of its content, and (ii) the more documents a term occurs in, the less discriminating it is[4]. Using the TF-IDF score of each term for each document, we can fill in the weights w_{ij} of the Document-Term matrix (see Fig. 4).

Task to be done. In the lab, we will use the *scikit-learn* Python library to automatically construct the TF-IDF matrix, and more specifically the `TfidfVectorizer` class. Let `data` be the list with the preprocessed documents created in the previous step. Then, the following code will return the TF-IDF matrix (sparse matrix stored in `tfidf_matrix`). Also examine the size and the sparsity of this matrix (i.e., fraction of non-zero elements). What do you observe?

```
# Create the Document-Term TF-IDF matrix
m = TfidfVectorizer()
tfidf_matrix = m.fit_transform(data)
tfidf_matrix = tfidf_matrix.toarray() # convert to numpy array
print "Size of TF-IDF matrix: ", tfidf_matrix.shape
```

3.4 Model learning and prediction

After having form the TF-IDF matrix, the problem has been transformed to a typical classification task: the rows of the matrix correspond to the instances (i.e., reviews) and the columns to the features (i.e., terms). Recall that, for each document we also have class information, i.e., positive or negative review, contained in the `labels` list (this list was

[4] Several variants of the *tf-idf* score have been proposed. See also the description given in Ref. [1], pages 12-14.

created during the data load step). In order to train a classification model, we will split the dataset (`tfidf_matrix`) into two sets: training and test. The training set will be used to learn the parameters of the classification model, that later will be applied to the test part in order to examine the accuracy of the model. We will use the following code to split the TF-IDF matrix:

```
# Split the data into random train and test subsets. Here we use 40% # of the data for testing
data_train, data_test, labels_train, labels_test = cross_validation.train_test_split(
                                        tfidf_matrix, labels, test_size=0.4, random_state=42)
```

Now, the `data_train` is the matrix for the training step, while the `data_test` will be used in the evaluation phase.

Task to be done. The next step of the pipeline involves the selection of the appropriate learning (i.e., classification) algorithm for the problem (e.g., Logistic Regression, SVMs, Random Forests). Here, you can test the performance of different algorithms and choose the best one. We will use built-in implementations of the classification algorithms provided by *scikit-learn*. Typically, in *scikit-learn*, we follow three steps in order to apply a classification algorithm: (i) create an object of the classifier, (ii) fit the parameters of the classifier to the training data, and (iii) do predictions on the test data. Run the following code, to predict the class labels using the Logistic Regression classification model:

```
# Initialize the model
clf = LogisticRegression()
# Fit the model to the training data
y_score = clf.fit(data_train, labels_train).predict_proba(data_test)
# Perform classification of test data
labels_predicted = clf.predict(data_test)
```

Now, the list `labels_predicted` will contain the predicted labels (positive/negative review) of the test data. In the following section, we present how to evaluate the results.

3.5 Evaluation of the classification resutls

For the evaluation of the classification results, we will use the precision, recall[5] and F_1-score (combination of precision and recall)[6]:

$$\text{precision} = \frac{TP}{TP + FP}, \qquad \text{recall} = \frac{TP}{TP + FN}, \qquad F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}.$$

The precision for a class is the number of true positives TP (i.e., the number of items correctly labeled as belonging to the positive class) divided by the total number of elements labeled as belonging to the positive class (i.e., the sum of true positives TP and false positives FP, which are items incorrectly labeled as belonging to the class). Recall in this context is defined as the number of true positives TP divided by the total number of elements that actually belong to the positive class (i.e., the sum of true positives TP and false negatives FN, which are items which were not labeled as belonging to the positive class but should have been).

We will also use the ROC curve[7] which is a graphical plot that illustrates the performance of a binary classification system as its discrimination threshold is varied. The curve is created by plotting the true positive rate TPR against the false positive rate FPR at various threshold settings. The TPR defines how many correct positive results occur among all positive samples available during the test. FPR on the other hand, defines how many incorrect positive results occur among all negative samples available during the test. The curve is defined in the 2-dimensional space (x-axis and y-axis min and max values: 0 to 1). The best possible prediction method would yield a point in the upper left corner or coordinate $(0, 1)$ of the ROC space. Thus, the area under the ROC curve (auc) is also an important characteristic.

Task to be done. We will use build-in tools of *scikit-learn* to do the evaluation. Run the following code that computes the precision, recall, F_1 score and also plots the ROC curve.

[5]Wikipedia's lemma for *Precision and Recall*: `http://en.wikipedia.org/wiki/Precision_and_recall`.
[6]Wikipedia's lemma for *F1-score*: `http://en.wikipedia.org/wiki/F1_score`.
[7]Wikipedia's lemma for *Receiver operating characteristic*: `http://en.wikipedia.org/wiki/Receiver_operating_characteristic`.

```
# Evaluation of the prediction
print classification_report(labels_test, labels_predicted)
print "The accuracy score is {:.2%}".format(accuracy_score(labels_test, labels_predicted))

# Compute ROC curve and area under the curve
fpr, tpr, thresholds = roc_curve(labels_test, y_score[:, 1])
roc_auc = auc(fpr, tpr)
print "Area under the ROC curve : %f" % roc_auc

# Plot ROC curve
plt.clf()
plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")
plt.show()
```

What do you observe about the performance of the algorithm? Repeat the same experiment replacing the Logistic Regression classifier in the previous step with the Random Forest and SVM classifiers.

References

[1] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47, March 2002.