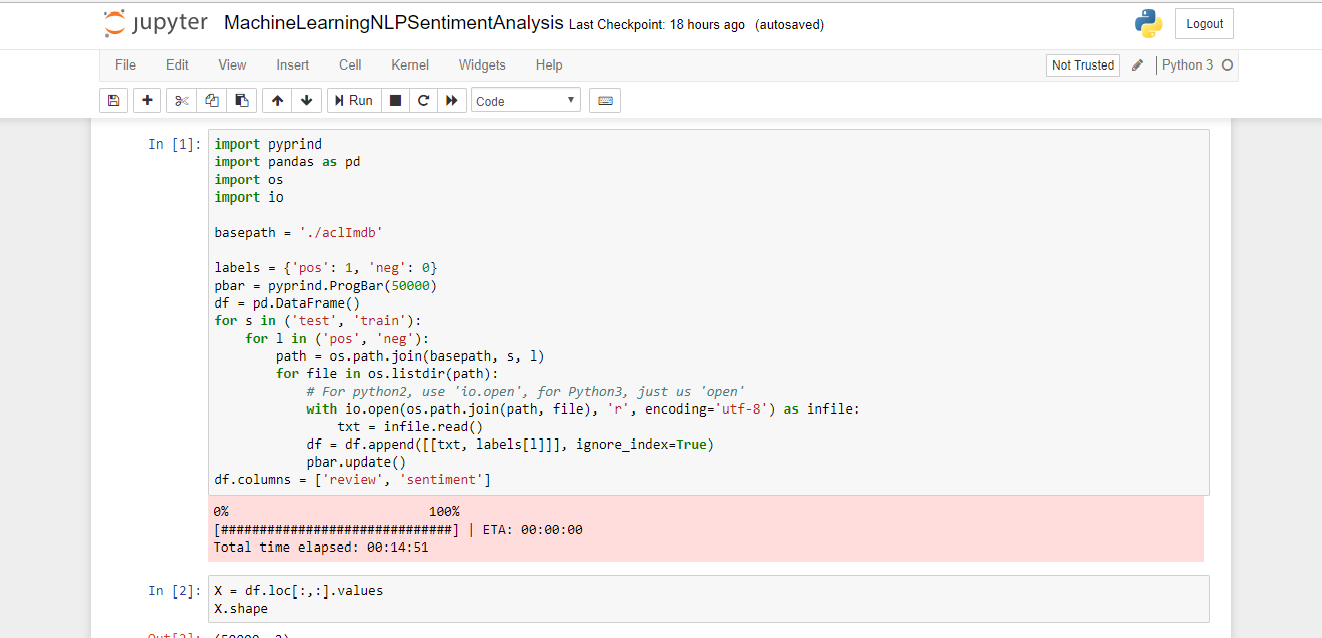
 In this project,I have implemented how to use machine learning algorithms for classifying the attitude of a writer with regard to a particular topic or the overall contextual polarity of a document.

In the following code section, we will be reading the movie reviews into a pandas DataFrame object,

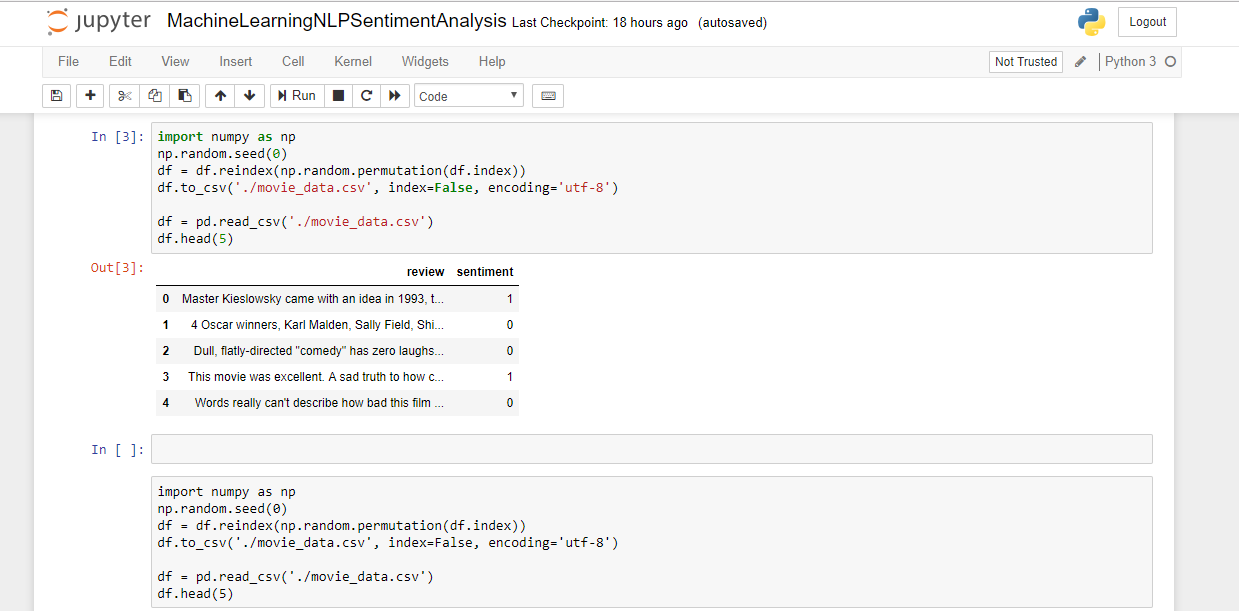


For our sentiment analysis, we will use a large dataset of movie reviews from the **Internet Movie Database (IMDb)**

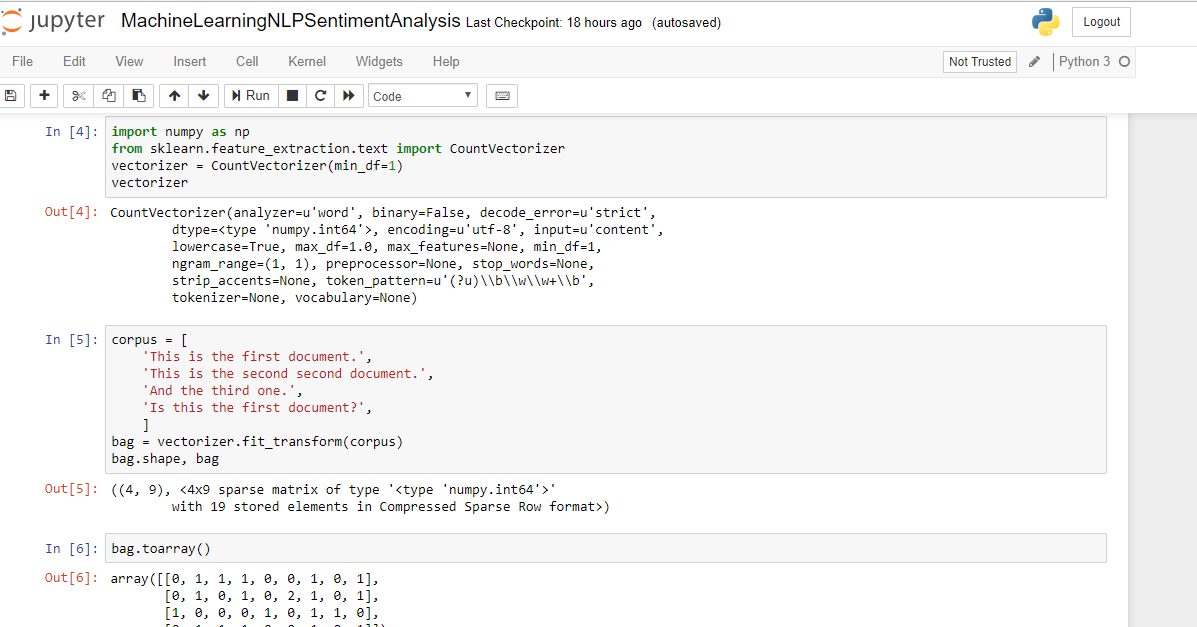
It has 50,000 polar movie reviews, and they are labeled as either positive or negative.

A movie was rated positive if it has more than six stars and a movie was rated negative if it has fewer than five stars on IMDb.

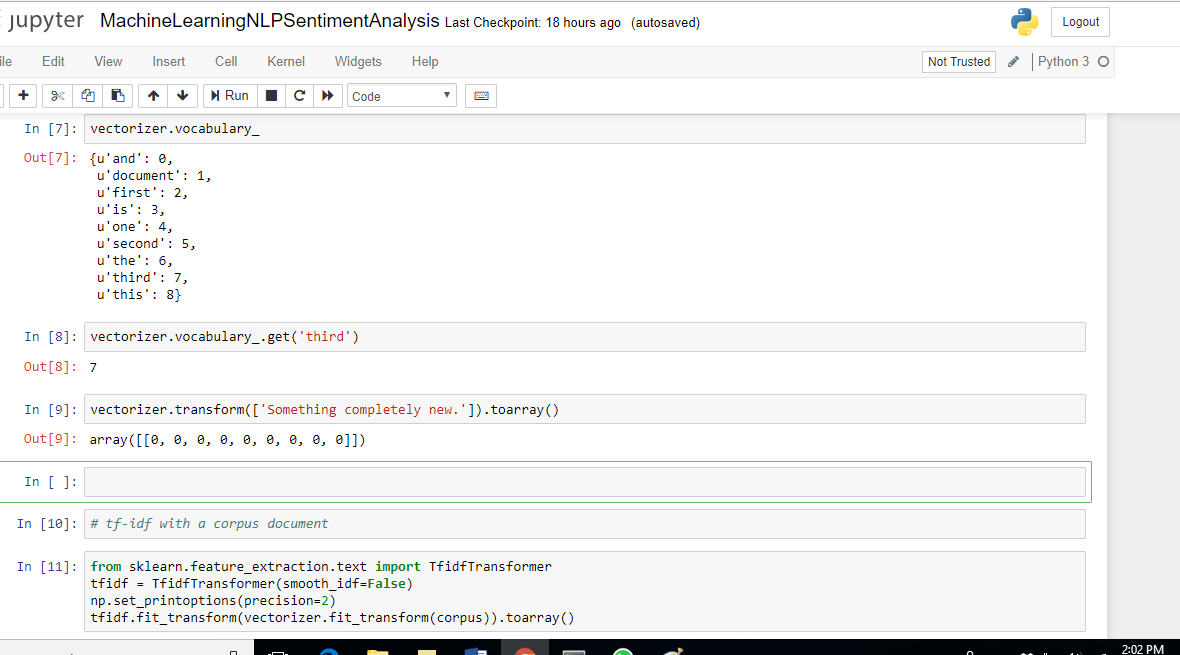
We may also want to store the assembled and shuffled movie review dataset as a CSV file:



We can use **sklearn.feature\_extraction.text.CountVectorizer** to construct a bag-of-words model based on the word counts in the respective documents. It implements both tokenization and occurrence counting in a single class:

The **fit\_transform** method on **CountVectorizer** constructs the vocabulary of the bag-of-words model. The default configuration tokenizes the string by extracting words of at least 2 letters. The two sentences are transformed into sparse feature vectors:

The words are stored in a Python dictionary, which maps the unique words that are mapped to integer indices



The converse mapping from feature name to column index is stored in the **vocabulary\_**attribute of the vectorizer:

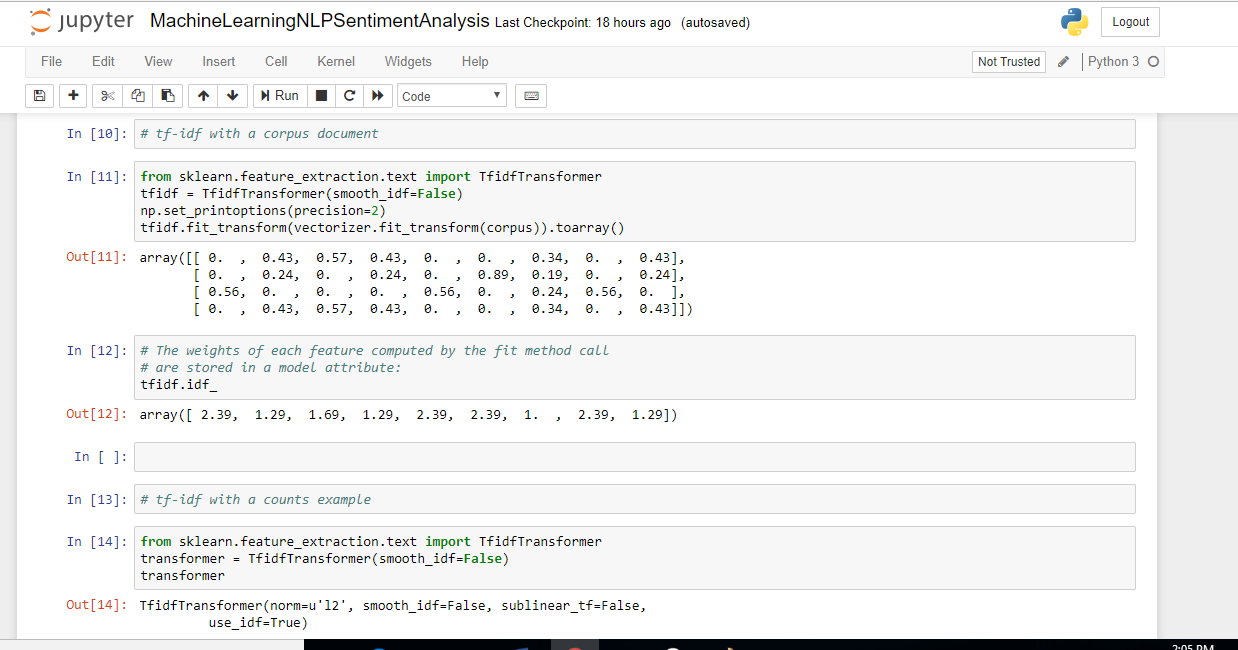
VectorizerVocabularyGet.png

So, any words that were not seen in the training corpus will be completely ignored in future calls to the transform method:

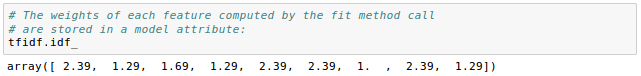
UnseenWords.png

 I have used a transformation technique called **term frequency-inverse document frequency (tf-idf)**. We'll use it to **re-weight** those frequently occurring words in the feature vectors. In order to re-weight the count features into floating point values suitable for usage by a classifier we need to use the **tf-idf** transform which can be defined as the product of the term frequency and the inverse document frequency:

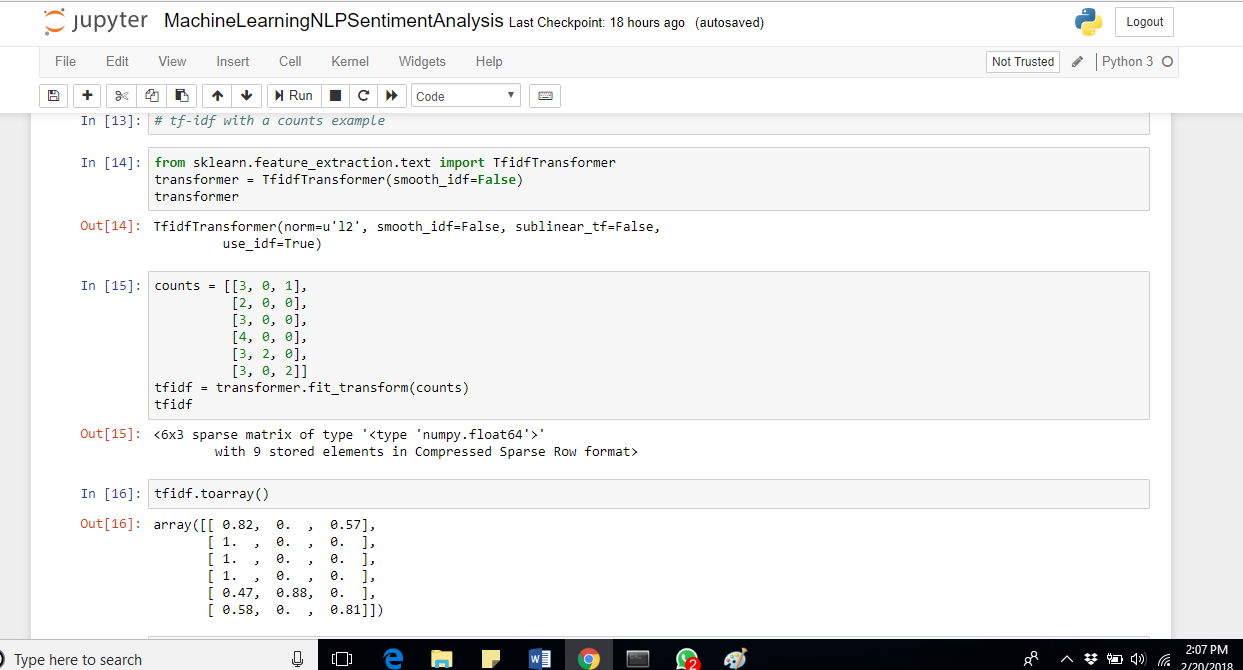
tf-idf(t,d)=tf(t,d)×idf(t,d)



The weights of each feature computed by the fit method call are stored in a model attribute:

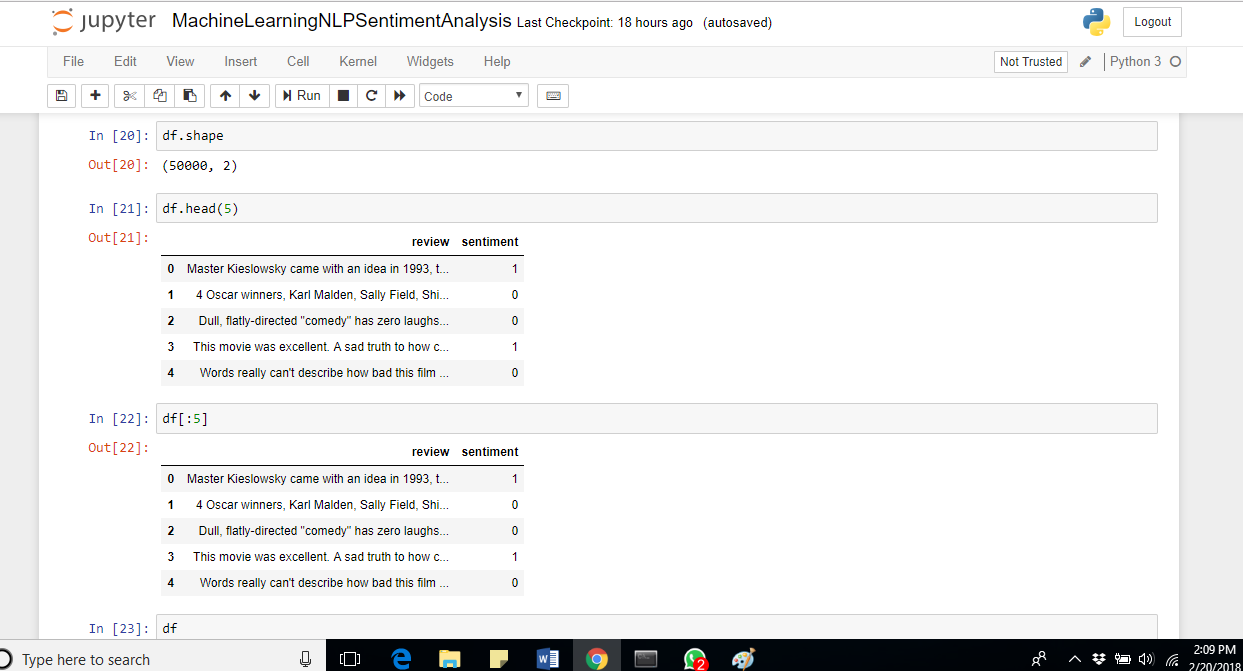


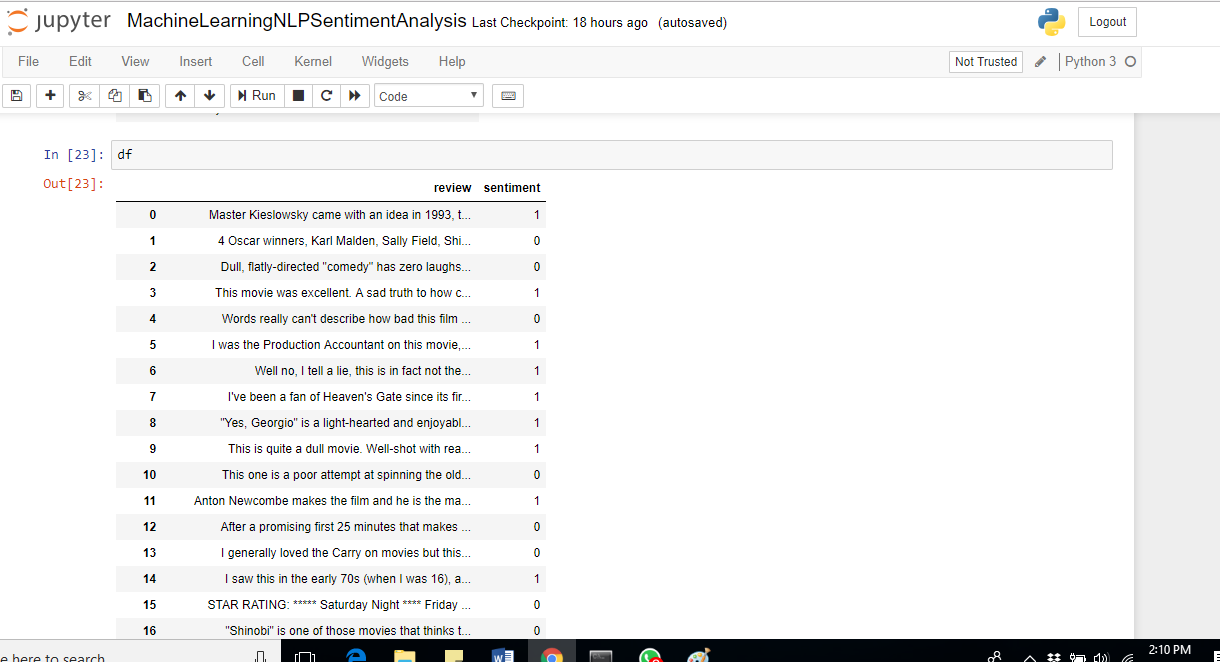
The following sample is from [Feature extraction](http://scikit-learn.org/stable/modules/feature_extraction.html).



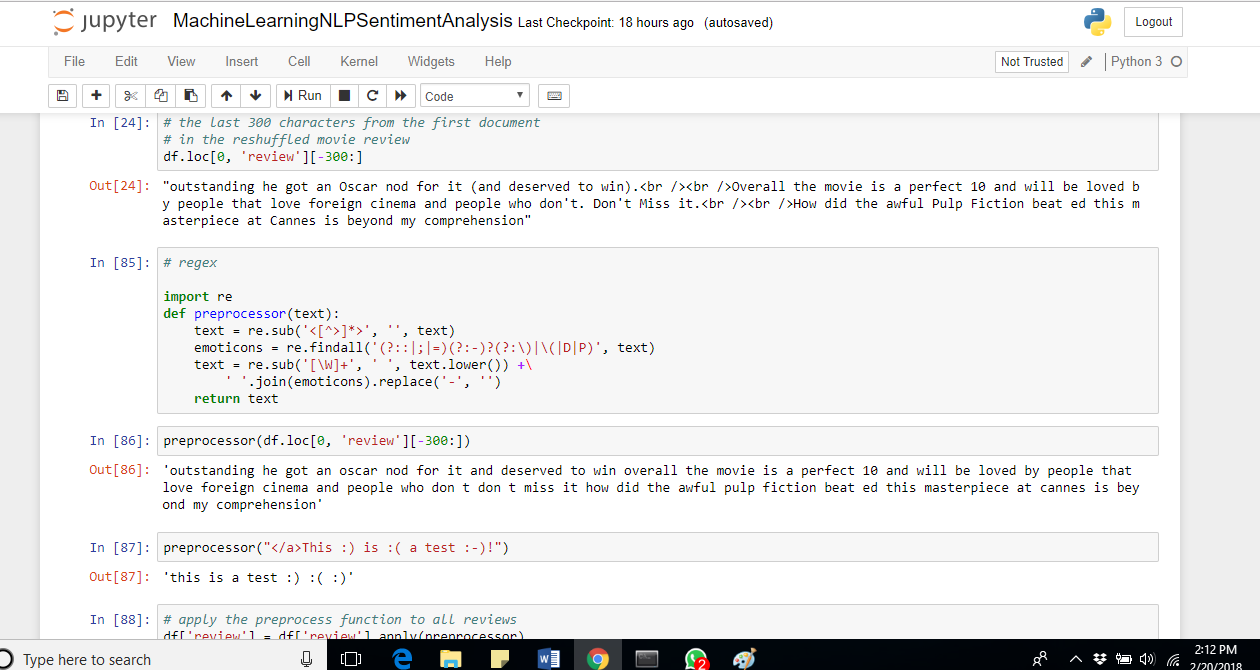
Clean up the text data

Here is our dataset





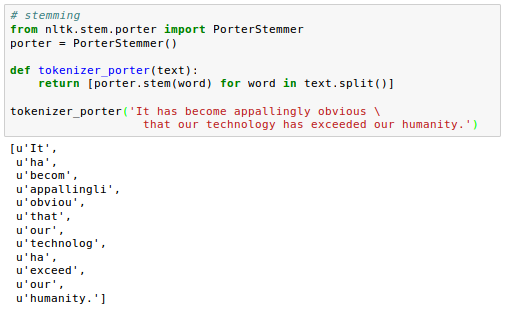
Before we build our bag-of-words model, we may want to clean up our movie review dataset by stripping it of all unwanted characters.



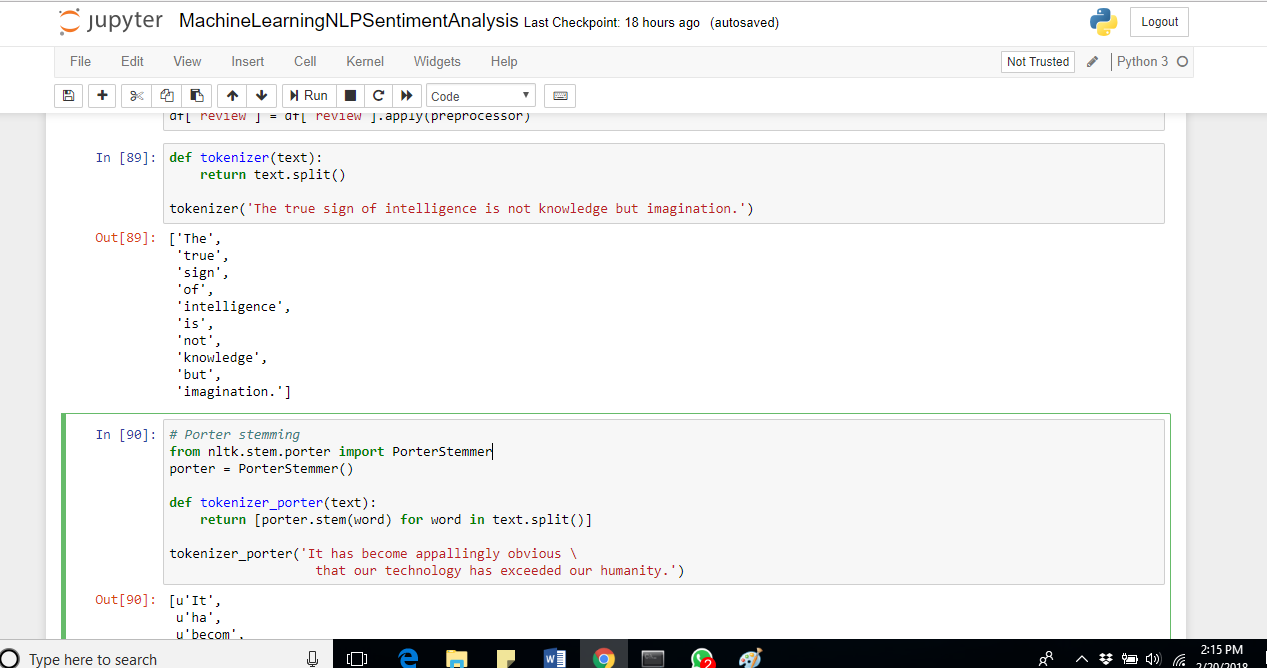
While HTML markup does not contain much useful semantics, punctuation marks can represent useful NLP contexts. However, just for now, we may want to remove all punctuation marks while keeping emoticon characters such as ":)" since those are certainly useful for sentiment analysis.

To accomplish the task, we're going to use regular expression (regex) library,

Now we need to apply the preprocessor to all of our movie reviews in our DataFrame:



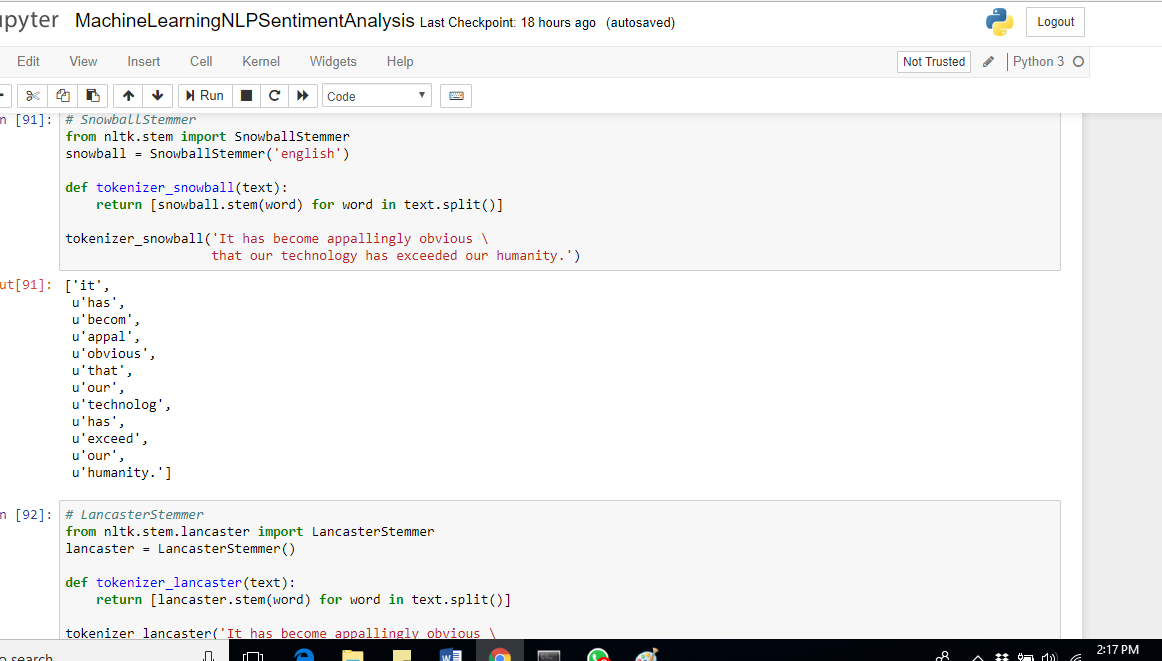
**Tokenization** is the process of breaking up a stream of text into words, phrases, or other meaningful elements called **tokens**.

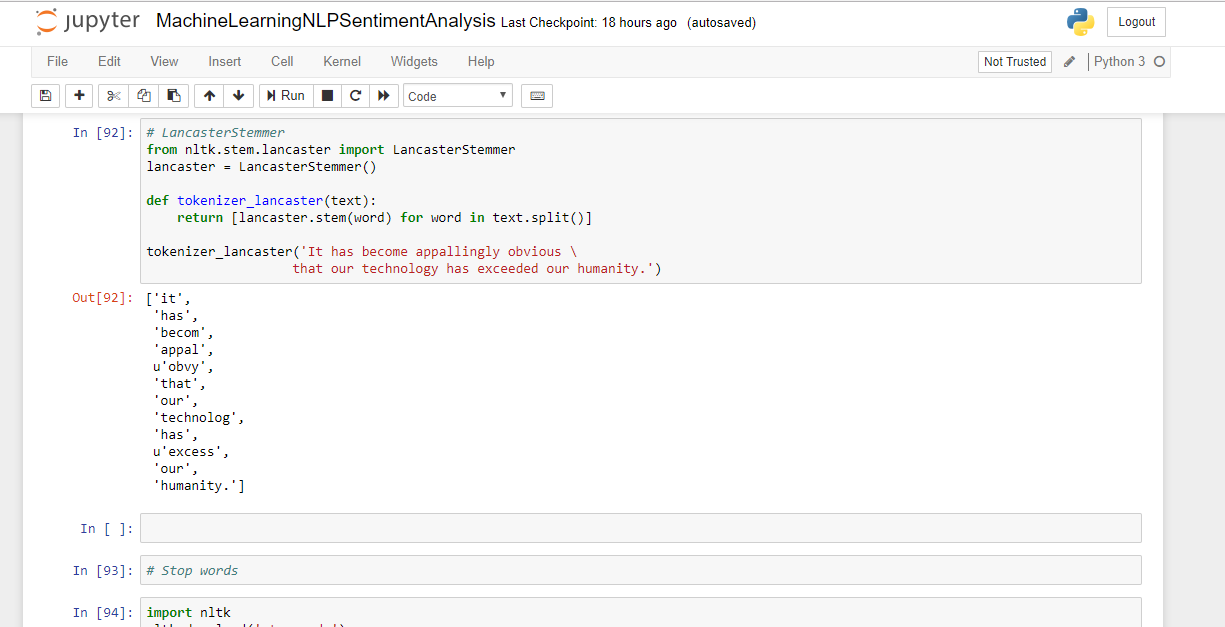


**Stemming** transforms a word into its root form that allows us to map related words to the same stem.

We'll use **Porter** stemming algorithm from Natural Language Toolkit for Python

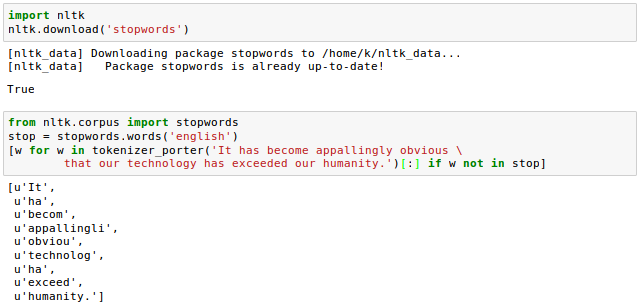
Other popular stemming algorithms include the newer **Snowball** stemmer (Porter2 or "English" stemmer) or the **Lancaster** stemmer (Paice-Husk stemmer), which is faster but also more aggressive than the Porter stemmer.





**Stop words** usually refer to the most common words in a language, there is no single universal list of stop words used by all natural language processing tools, and indeed not all tools even use such a list.

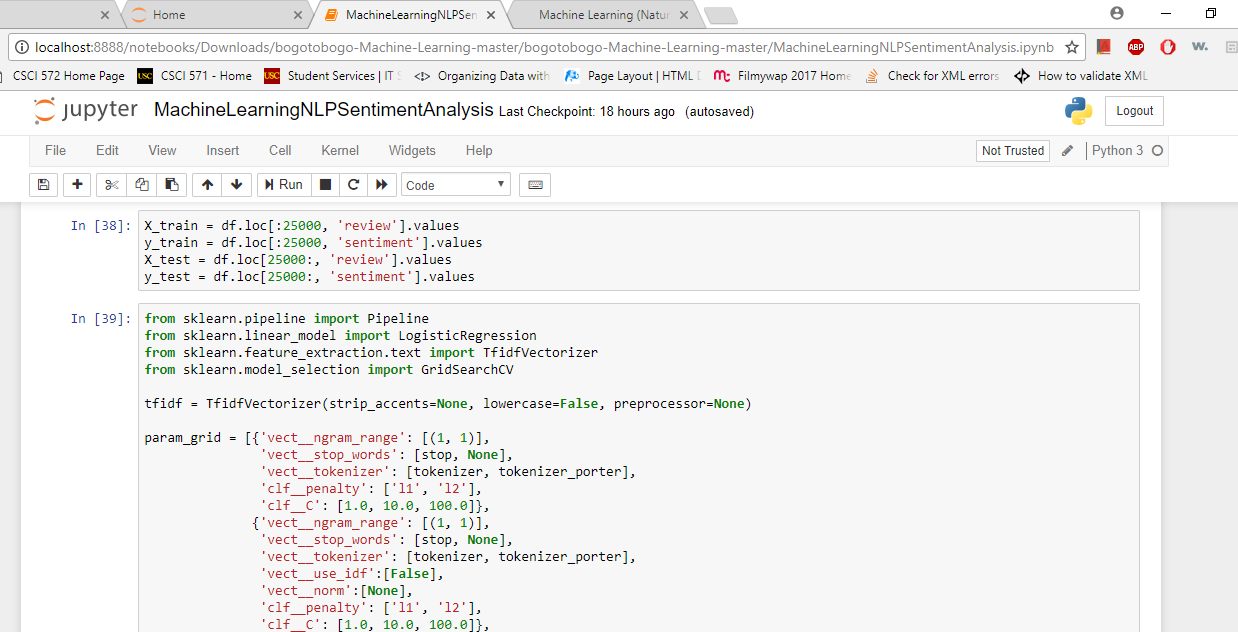
NLTK library provides the set of 127 English stop-words, and we're going to use it to remove stop-words from the movie reviews:

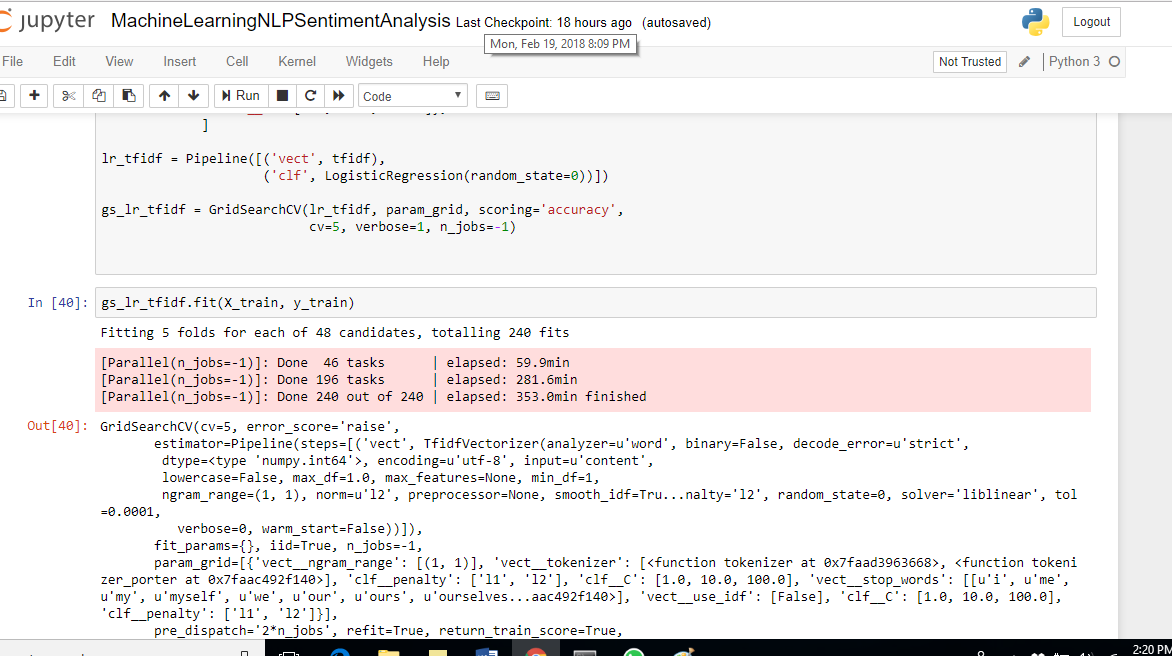


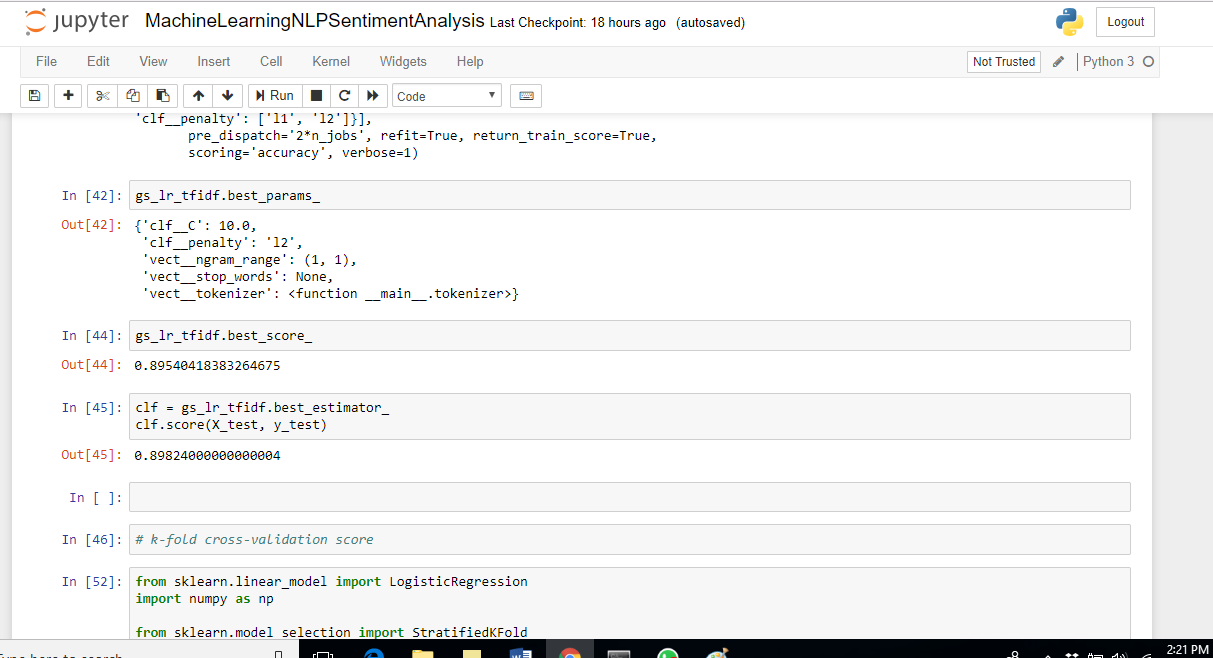
Training a model

We're now almost ready to classify the movie reviews into positive and negative reviews.

First of all, we want to divide the DataFrame data which we cleaned-up in previous articles into 25,000/25,000 documents for training/testing:







Next, using 5-fold stratified cross-validation, we will use a **GridSearchCV** object to find the optimal set of parameters for our logistic regression model:

LogisticRegression_pipeline.png

The [sklearn.model\_selection. GridSearchCV](http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html" \t "_blank) returns training score after exhaustive search over specified parameter values for an estimator. It implements a "fit" and a "score" method which we are going to use once the grid search finish.

Using the best model from the grid search, we can get the output for the 5-fold cross-validation accuracy scores on the training set and the classification accuracy on the test dataset:



Here the **best\_estimator\_** is the estimator that was chosen by the search, i.e. estimator which gave highest score on the left out data and the **best\_score\_** is the score of best\_estimator on the left out data.

**The output shows us that our machine learning model can predict whether a movie review is positive or negative with almost 90 percent accuracy.**