```
# Text Classification
Now we're at the point where we should be able to:
* Read in a collection of documents - a *bbc corpus*
* Transform text into numerical vector data using a pipeline
* Create a classifier
* Fit/train the classifier
* Test the classifier on new data
* Evaluate performance
```

Perform imports and load the dataset

The dataset contains the text of 2000 movie reviews. 1000 are positive, 1000 are negative, and the text has been preprocessed as a tab-delimited file.

In [1]:

```python
import numpy as np
import pandas as pd

df = pd.read_csv('bbc-text.csv')
df.head()
```

Out[1]:

	category	text
0	tech	tv future in the hands of viewers with home th...
1	business	worldcom boss left books alone former worldc...
2	sport	tigers wary of farrell gamble leicester say ...
3	sport	yeading face newcastle in fa cup premiership s...
4	entertainment	ocean s twelve raids box office ocean s twelve...

In [2]:

```python
len(df)
```

Out[2]:

```
2000
```

In [3]:

```python
from IPython.display import Markdown, display
display(Markdown('> '+df['review'][0]))
```

```
<IPython.core.display.Markdown object>
```

In [4]:

```
# Check for the existence of NaN values in a cell:
df.isnull().sum()
```

Out[4]:

```
label      0
review    35
dtype: int64
```

35 records show **NaN** (this stands for "not a number" and is equivalent to *None*). These are easily removed using the `.dropna()` pandas function.

> CAUTION: By setting inplace=True, we permanently affect the DataFrame currently in memory, and this can't be undone. However, it does *not* affect the original source data. If we needed to, we could always load the original DataFrame from scratch.

In [5]:

```
df.dropna(inplace=True)

len(df)
```

Out[5]:

```
1965
```

Detect & remove empty strings

Technically, we're dealing with "whitespace only" strings. If the original .tsv file had contained empty strings, pandas **.read_csv()** would have assigned NaN values to those cells by default.

In order to detect these strings we need to iterate over each row in the DataFrame. The **.itertuples()** pandas method is a good tool for this as it provides access to every field. For brevity we'll assign the names `i`, `lb` and `rv` to the `index`, `label` and `review` columns.

In [6]:

```
blanks = []  # start with an empty list

for i,lb,rv in df.itertuples():  # iterate over the DataFrame
    if type(rv)==str:            # avoid NaN values
        if rv.isspace():         # test 'review' for whitespace
            blanks.append(i)     # add matching index numbers to the list

print(len(blanks), 'blanks: ', blanks)
```

```
27 blanks:  [57, 71, 147, 151, 283, 307, 313, 323, 343, 351, 427, 501, 63
3, 675, 815, 851, 977, 1079, 1299, 1455, 1493, 1525, 1531, 1763, 1851, 19
05, 1993]
```

Next we'll pass our list of index numbers to the **.drop()** method, and set `inplace=True` to make the change permanent.

In [7]:

```
df.drop(blanks, inplace=True)

len(df)
```

Out[7]:

```
1938
```

Great! We dropped 62 records from the original 2000. Let's continue with the analysis.

Take a quick look at the `category` column:

In [2]:

```
df.columns
```

Out[2]:

```
Index(['category', 'text'], dtype='object')
```

In [3]:

```
df['category'].value_counts()
```

Out[3]:

```
sport            511
business         510
politics         417
tech             401
entertainment    386
Name: category, dtype: int64
```

Split the data into train & test sets:

In [9]:

```
from sklearn.model_selection import train_test_split

X = df['review']
y = df['label']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=4
```

Build pipelines to vectorize the data, then train and fit a

model

Now that we have sets to train and test, we'll develop a selection of pipelines, each with a different model.

In [10]:

```
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import LinearSVC

# Naïve Bayes:
text_clf_nb = Pipeline([('tfidf', TfidfVectorizer()),
                     ('clf', MultinomialNB()),
])

# Linear SVC:
text_clf_lsvc = Pipeline([('tfidf', TfidfVectorizer()),
                     ('clf', LinearSVC()),
])
```

Feed the training data through the first pipeline

We'll run naïve Bayes first

In [11]:

```
text_clf_nb.fit(X_train, y_train)
```

Out[11]:

```
Pipeline(memory=None,
     steps=[('tfidf', TfidfVectorizer(analyzer='word', binary=False, deco
de_error='strict',
        dtype=<class 'numpy.float64'>, encoding='utf-8', input='content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), norm='l2', preprocessor=None, smooth_idf=Tru
e,...rue,
        vocabulary=None)), ('clf', MultinomialNB(alpha=1.0, class_prior=N
one, fit_prior=True))])
```

Run predictions and analyze the results (naïve Bayes)

In [12]:

```
# Form a prediction set
predictions = text_clf_nb.predict(X_test)
```

In [13]:

```
# Report the confusion matrix
from sklearn import metrics
print(metrics.confusion_matrix(y_test,predictions))
```

```
[[287  21]
 [130 202]]
```

In [14]:

```
# Print a classification report
print(metrics.classification_report(y_test,predictions))
```

```
              precision    recall  f1-score   support

         neg       0.69      0.93      0.79       308
         pos       0.91      0.61      0.73       332

   micro avg       0.76      0.76      0.76       640
   macro avg       0.80      0.77      0.76       640
weighted avg       0.80      0.76      0.76       640
```

In [15]:

```
# Print the overall accuracy
print(metrics.accuracy_score(y_test,predictions))
```

```
0.7640625
```

Naïve Bayes gave us better-than-average results at 76.4% for classifying reviews as positive or negative based on text alone. Let's see if we can do better.

Feed the training data through the second pipeline

Next we'll run Linear SVC

In [16]:

```
text_clf_lsvc.fit(X_train, y_train)
```

Out[16]:

```
Pipeline(memory=None,
     steps=[('tfidf', TfidfVectorizer(analyzer='word', binary=False, deco
de_error='strict',
        dtype=<class 'numpy.float64'>, encoding='utf-8', input='content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), norm='l2', preprocessor=None, smooth_idf=Tru
e,...ax_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0))])
```

Run predictions and analyze the results (Linear SVC)

In [17]:

```
# Form a prediction set
predictions = text_clf_lsvc.predict(X_test)
```

In [18]:

```
# Report the confusion matrix
from sklearn import metrics
print(metrics.confusion_matrix(y_test,predictions))
```

```
[[259  49]
 [ 49 283]]
```

In [19]:

```
# Print a classification report
print(metrics.classification_report(y_test,predictions))
```

```
              precision    recall  f1-score   support

         neg       0.84      0.84      0.84       308
         pos       0.85      0.85      0.85       332

   micro avg       0.85      0.85      0.85       640
   macro avg       0.85      0.85      0.85       640
weighted avg       0.85      0.85      0.85       640
```

In [20]:

```
# Print the overall accuracy
print(metrics.accuracy_score(y_test,predictions))
```

```
0.846875
```

Now let's repeat the process above and see if the removal of stopwords improves or impairs our score.

In [23]:

```
predictions = text_clf_lsvc2.predict(X_test)
print(metrics.confusion_matrix(y_test,predictions))
```

```
[[256  52]
 [ 48 284]]
```

In [24]:

```
print(metrics.classification_report(y_test,predictions))
```

```
              precision    recall  f1-score   support

         neg       0.84      0.83      0.84       308
         pos       0.85      0.86      0.85       332

   micro avg       0.84      0.84      0.84       640
   macro avg       0.84      0.84      0.84       640
weighted avg       0.84      0.84      0.84       640
```

In [25]:

```
print(metrics.accuracy_score(y_test,predictions))
```

```
0.84375
```

In [28]:

```
print(text_clf_lsvc.predict([myreview]))
```

```
['neg']
```