In [1]:

```
import numpy as np
import pandas as pd
import os
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import RFE, SelectPercentile, f_classif
from sklearn import tree
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
```

Reading Data

In [2]:

```
def segmentWords(s):
    return s.split()

def readFile(fileName):
    # Function for reading file
    # input: filename as string
    # output: contents of file as list containing single words
    contents = []
    f = open(fileName)
    for line in f:
        contents.append(line)
    f.close()
    result = segmentWords('\n'.join(contents))
    return result
```

Create a Dataframe containing the counts of each word in a file

In [3]:

```
d = []

for c in os.listdir("data_training/train"):
    directory = "data_training/train/" + c
    for f in os.listdir(directory):
        words = readFile(directory + "/" + f)
        e = {x:words.count(x) for x in words}
        e['__FileID__'] = f
        e['__CLASS__'] = c
        d.append(e)
```

Create a dataframe from d - make sure to fill all the nan values with zeros.

Hint: Consider the `fillna()` function for Dataframes

In [4]:

```
df = pd.DataFrame(d).fillna(0)
```

In [5]:

```
df = df.rename(columns = {'fit': 'fit_feature'})
```

Split data into training and validation set

- Sample 80% of your dataframe to be the training data
- Let the remaining 20% be the validation data (you can filter out the indicies of the original dataframe that weren't selected for the training data)

- Split the dataframe for both training and validation data into x and y dataframes - where y contains the labels and x contains the words

Hint: Try looking at the Dataframe `drop()` *function*

In [6]:

```
dfX = df.drop(['__FileID__'], axis=1)
dfX = dfX.drop(['__CLASS__'], axis=1)
X_train, X_test, y_train, y_test = train_test_split(dfX, dfX["sentiment"],
test_size = 0.2)
```

In [7]:

```
'__FileID__' in df.columns.values
```

Out[7]:

```
True
```

In [8]:

```
print("Training images: ", X_train.shape, "Training labels: ", y_train.shap
e)
print("Testing images: ", X_test.shape, "Testing labels: ", y_test.shape)
```

```
('Training images: ', (1120, 42774), 'Training labels: ', (1120,))
('Testing images: ', (280, 42774), 'Testing labels: ', (280,))
```

Logistic Regression

Basic Logistic Regression

- Use sklearn's linear_model.LogisticRegression() to create your model.
- Fit the data and labels with your model.
- Score your model with the same data and labels.

In [9]:

```
lr = LogisticRegression()
lr.fit(X_train, y_train)
predictions = lr.predict(X_test)
lr.score(X_test,y_test)
```

Out[9]:

```
0.98928571428571432
```

Changing Parameters

In [10]:

```
lr = LogisticRegression(penalty="l1")
lr.fit(X_train, y_train)
predictions = lr.predict(X_test)
lr.score(X_test,y_test)
```

Out[10]:

```
1.0
```

Feature Selection

- In the backward stepsize selection method, you can remove coefficients and the corresponding x columns, where the coefficient is more than a particular amount away from the mean - you can choose how far from the mean is reasonable.

Hint: Numpy's `argwhere()` might be useful here

Hint: Instead of defining a hard-coded constant to determine which features to keep or remove, consider using values relative to the distribution of the weight magnitudes

In [11]:

```
lr = LogisticRegression(penalty="l1")
sp = SelectPercentile(f_classif, percentile=10)
X_train = sp.fit_transform(X_train,y_train)
X_test = sp.transform(X_test)
lr.fit(X_train, y_train)
predictions = lr.predict(X_test)
lr.score(X_test,y_test)
```

```
/home/dhanush/anaconda2/lib/python2.7/site-
packages/sklearn/feature_selection/univariate_selection.py:113: UserWarning
: Features [    30     43     60 ..., 42746 42765 42766] are constant.
  UserWarning)
/home/dhanush/anaconda2/lib/python2.7/site-
packages/sklearn/feature_selection/univariate_selection.py:114: RuntimeWarn
ing: divide by zero encountered in divide
  f = msb / msw
/home/dhanush/anaconda2/lib/python2.7/site-
packages/sklearn/feature_selection/univariate_selection.py:114: RuntimeWarn
ing: invalid value encountered in divide
  f = msb / msw
```

Out[11]:

```
1.0
```

How did you select which features to remove? Why did that reduce overfitting?

In [13]:

```
'''
I selected SinglePercentile to remove features because instead of finding a
fixed number of features to select, it selects the best % of them. This is
good
because a relatively best number of features makes sure that only if featur
es
reach a certain threshold of impact are they used instead of using a certai
n
number regardless of quality/relevance.
'''
```

Out[13]:

```
'\nI selected SinglePercentile to remove features because instead of findin
g a\nfixed number of features to select, it selects the best % of them. Thi
s is good\nbecause a relatively best number of features makes sure that onl
y if features\nreach a certain threshold of impact are they used instead of
using a certain \nnumber regardless of quality/relevance.\n'
```

In [14]:

```
1 in X_train.values
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-14-234fb2e988f1> in <module>()
----> 1 1 in X_train.values

AttributeError: 'numpy.ndarray' object has no attribute 'values'
```

Single Decision Tree

Basic Decision Tree

- Initialize your model as a decision tree with sklearn.
- Fit the data and labels to the model.

In [15]:

```
dtc = tree.DecisionTreeClassifier().fit(X_train, y_train)
predictedVals = dtc.predict(X_test)
dtc.score(X_test,y_test)
```

Out[15]:

```
1.0
```

Changing Parameters

- To test out which value is optimal for a particular parameter, you can either loop through various values or look into `sklearn.model_selection.GridSearchCV`

In [16]:

```
dtc = tree.DecisionTreeClassifier(max_features=2).fit(X_train, y_train)
predictedVals = dtc.predict(X_test)
dtc.score(X_test,y_test)
```

Out[16]:

```
0.9857142857142857
```

How did you choose which parameters to change and what value to give to them? Feel free to show a plot.

In [17]:

```
'''
I chose max_features because as mentioned in the logistic regression sectio
n,
the dataset has a lot of features and using too many features for a
prediction
task can overfit the model. As such reducing the total number of possible
features made sense to do. I chose the value 2 because generally speaking
sentiment is either positive or negative (and arguably all other types of
emotions fall into one of those two categories).
'''
```

Out[17]:

```
'\nI chose max_features because as mentioned in the logistic regression sec
tion, \nthe dataset has a lot of features and using too many features for a
prediction\ntask can overfit the model. As such reducing the total number o
f possible\nfeatures made sense to do. I chose the value 2 because generall
y speaking\nsentiment is either positive or negative (and arguably all othe
r types of \nemotions fall into one of those two categories).\n'
```

Why is a single decision tree so prone to overfitting?

In [18]:

```
'''
This is the case becase a single decision tree samples over the entire
data set instead of only part of it as is standard when splitting the data
for training and testing (in say logistic regression).
'''
```

Out[18]:

```
'\nThis is the case becase a single decision tree samples over the entire\n
data set instead of only part of it as is standard when splitting the data\
nfor training and testing (in say logistic regression).\n'
```

Random Forest Classifier

Basic Random Forest

- Use sklearn's ensemble.RandomForestClassifier() to create your model.
- Fit the data and labels with your model.
- Score your model with the same data and labels.

In [19]:

```
clf = RandomForestClassifier(max_depth=3, random_state=0)
```

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

Out[19]:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=3, max_features='auto', max_leaf_nodes=None,
            min_impurity_split=1e-07, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            n_estimators=10, n_jobs=1, oob_score=False, random_state=0,
            verbose=0, warm_start=False)
```

In [20]:

```
clf.predict(X_test)
clf.score(X_test, y_test)
```

Out[20]:

```
0.98928571428571432
```

Changing Parameters

In [21]:

```
clf = RandomForestClassifier(max_depth=3, random_state=0,max_leaf_nodes=2)
clf.fit(X_train, y_train)
```

Out[21]:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=3, max_features='auto', max_leaf_nodes=2,
            min_impurity_split=1e-07, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            n_estimators=10, n_jobs=1, oob_score=False, random_state=0,
            verbose=0, warm_start=False)
```

In [22]:

```
clf.predict(X_test)
clf.score(X_test, y_test)
```

Out[22]:

```
0.98928571428571432
```

What parameters did you choose to change and why?

In [23]:

```
'''
I changed the number of max leaf nodes because it would decrease
the size of the relationships between different data points.
'''
```

Out[23]:

```
'\nI changed the number of max leaf nodes because it would decrease\nthe si
ze of the relationships between different data points.\n'
```

How does a random forest classifier prevent overfitting better than a single decision tree?

A random forest prevents overfitting by creating random subsets of the features and then builds smaller trees using those subsets which are then combined into subtrees. Thus the random forest is a better choice to prevent overfitting.

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```