Data Mining Portfolio

Nathan Cooper Jones - nathancooperjones@gmail.com

In [1]:
```
import numpy as np
import pandas as pd
from scipy import stats

from sklearn import datasets
from sklearn import decomposition
from sklearn.preprocessing import Imputer
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import export_graphviz
from sklearn import model_selection
from sklearn import metrics

import matplotlib.pyplot as plt
import matplotlib.image as mpimg
%matplotlib inline
```

Using a Collaborative Recommendation System for Netflix Users

I will begin by creating the utility matrix and centering it by subtracting it by the user means.

In [2]:
```
rating_cols = ['user_id', 'movie_id', 'rating', 'timestamp']

ratings = pd.read_csv('ml-100k/u.data', sep='\t', names=rating_cols)
```

In [3]:
```
utility = ratings.pivot(index='user_id', columns='movie_id', values='rating')

utility.head()
```

Out[3]:

movie_id	1	2	3	4	5	6	7	8	9	10	...	1673	1674	1675	1676	1677	167
user_id																	
1	5.0	3.0	4.0	3.0	3.0	5.0	4.0	1.0	5.0	3.0	...	NaN	NaN	NaN	NaN	NaN	Na
2	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2.0	...	NaN	NaN	NaN	NaN	NaN	Na
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	Na
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	Na
5	4.0	3.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	Na

5 rows × 1682 columns

In [4]:
```
user_means = utility.mean(axis=1)

user_means.head() # sanity check
```

Out[4]:
```
user_id
1    3.610294
2    3.709677
3    2.796296
4    4.333333
5    2.874286
dtype: float64
```

In [5]:
```
utility_centered = utility - user_means
utility_centered = utility_centered.where((pd.notnull(utility_centered)),0)

utility_centered.head()
```

Out[5]:

	1	2	3	4	5	6	7	8	9	1
user_id										
1	1.389706	-0.709677	1.203704	-1.333333	0.125714	1.364929	0.034739	-2.79661	0.727273	-
2	0.389706	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	-
3	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0
4	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0
5	0.389706	-0.709677	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0

5 rows × 1682 columns

Next, I will go through and calculate the cosine similarity for each user compared to user 1 and add it to a list which will then be sorted to get the top 10 users who are closest to user 1 in rankings via cosine similarity calculation.

In [6]:
```
lst = []

feat1 = utility_centered.loc[1:1]
for x in range(2, utility_centered.shape[0] + 1):
    feat2 = utility_centered.loc[x:x]
    lst.append([metrics.pairwise.cosine_similarity(feat1, feat2)[0][0], x])
```

In [7]:
```
user_means[1]
```

Out[7]:
```
3.6102941176470589
```

In [8]:
```
lst = sorted(lst, reverse=True)
lst = lst[0:10]

lst
```

Out[8]:
```
[[0.29148679307800707, 738],
 [0.2784017205961094, 592],
 [0.26815054175880981, 276],
 [0.26476146556668312, 267],
 [0.26400260297782174, 643],
 [0.26236784527028273, 757],
 [0.26233704478060194, 457],
 [0.26084701039863195, 606],
 [0.25562438236025764, 916],
 [0.2529544008014209, 44]]
```

In [9]:
```
top_10 = []
for x in lst:
    top_10.append(x[1])

top_10 = sorted(top_10)
top_10
```

Out[9]:
```
[44, 267, 276, 457, 592, 606, 643, 738, 757, 916]
```

Now with the top 10 users most similar to user 1 via cosine similarity calculation being users 44, 267, 276, 457, 592, 606, 643, 738, 757, and 916, we can now see what each of these ten users rated movie 508 [*The People vs. Larry Flynt* (1996)] and use the mean of all of these values added to the average rating that user 1 gives as our predicted value for user 1 (the exact process which Leskovec describes in *Mining of Massive Datasets* accessible at http://infolab.stanford.edu/~ullman/mmds/book.pdf (http://infolab.stanford.edu/~ullman/mmds/book.pdf)). Note that some of these ten users did not rate movie 508 and thus are not counted in the calculation of the mean (example, user 44 did not rate the movie at all but user 276 did, so 276 is counted towards the mean. In total, four users out of the ten actually rated movie 508).

In [10]:
```
# now let's do the second half of the question
sum = 0
count = 0

for x in top_10:
    val = utility_centered.loc[x, 508]
    if val:
        sum += val
        count += 1

mean = sum / count
mean
```

Out[10]:
```
0.6724137931034484
```

In [11]:
```
expected_rating = user_means[1] + mean
expected_rating
```

Out[11]:
```
4.2827079107505073
```

With all that work finally done, we can conclude that user 1 would likely rate *The People vs. Larry Flynt* a **4.28 out of 5**. One thing is for sure: an algorithm using this approach would most definitely recommend the movie to user 1 to watch, since multiple users similar to this user watched and liked this selection.

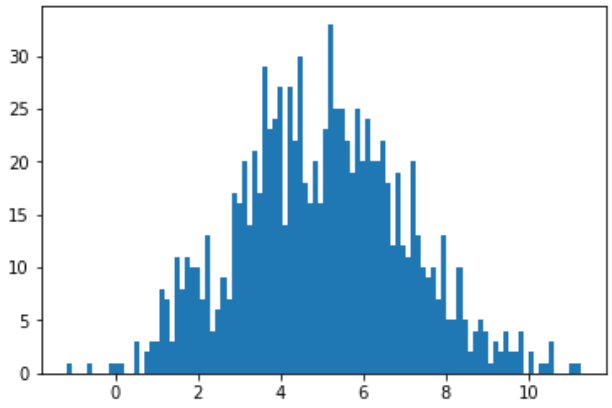
Using Decision Trees to Draw Conclusions from Two Distributions

We will begin by randomly generating two different sets of Normal distributions, each with N = 1000. However, x1 will be centered with mean 5 while x2 will be centered with mean -5.

In [12]:
```
N = 1000
x1 = np.random.normal(5, 2, N)
c1 = np.repeat('0', N)
x2 = np.random.normal(-5, 2, N)
c2 = np.repeat('1', N)
```

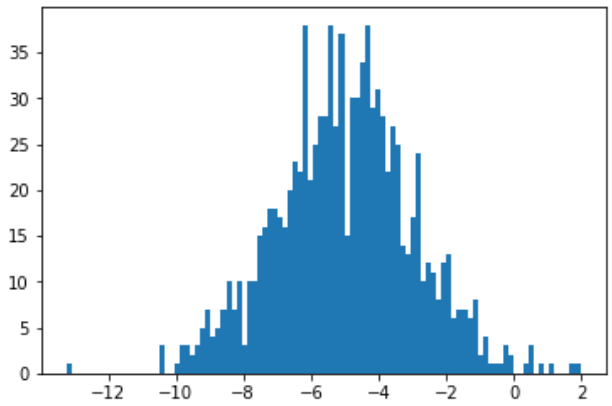
In [13]:
```
plt.hist(x1, bins=100)
```

Out[13]:
```
(array([  1.,   0.,   0.,   0.,   1.,   0.,   0.,   0.,   1.,   1.,   1.,
          0.,   0.,   3.,   0.,   2.,   3.,   3.,   8.,   7.,   3.,  11.,
          8.,  11.,  10.,  10.,   7.,  13.,   4.,   6.,   9.,   7.,  17.,
         16.,  20.,  14.,  21.,  17.,  29.,  23.,  24.,  27.,  14.,  27.,
         22.,  30.,  18.,  16.,  20.,  16.,  23.,  33.,  25.,  25.,  22.,
         19.,  25.,  20.,  24.,  20.,  20.,  22.,  18.,  12.,  19.,  12.,
         11.,  20.,  13.,  10.,   9.,  10.,   7.,  13.,   5.,   5.,  10.,
          5.,   2.,   4.,   5.,   4.,   1.,   3.,   2.,   4.,   2.,   2.,
          4.,   0.,   2.,   0.,   1.,   1.,   3.,   0.,   0.,   0.,   1.,
          1.]), array([ -1.16186638,  -1.03779776,  -0.91372914,  -0.78966051,
         -0.66559189,  -0.54152327,  -0.41745465,  -0.29338603,
         -0.16931741,  -0.04524879,   0.07881983,   0.20288845,
          0.32695707,   0.45102569,   0.57509431,   0.69916293,
          0.82323156,   0.94730018,   1.0713688 ,   1.19543742,
          1.31950604,   1.44357466,   1.56764328,   1.6917119 ,
          1.81578052,   1.93984914,   2.06391776,   2.18798638,
          2.312055  ,   2.43612363,   2.56019225,   2.68426087,
          2.80832949,   2.93239811,   3.05646673,   3.18053535,
          3.30460397,   3.42867259,   3.55274121,   3.67680983,
          3.80087845,   3.92494708,   4.0490157 ,   4.17308432,
          4.29715294,   4.42122156,   4.54529018,   4.6693588 ,
          4.79342742,   4.91749604,   5.04156466,   5.16563328,
          5.2897019 ,   5.41377052,   5.53783915,   5.66190777,
          5.78597639,   5.91004501,   6.03411363,   6.15818225,
          6.28225087,   6.40631949,   6.53038811,   6.65445673,
          6.77852535,   6.90259397,   7.0266626 ,   7.15073122,
          7.27479984,   7.39886846,   7.52293708,   7.6470057 ,
          7.77107432,   7.89514294,   8.01921156,   8.14328018,
          8.2673488 ,   8.39141742,   8.51548604,   8.63955467,
          8.76362329,   8.88769191,   9.01176053,   9.13582915,
          9.25989777,   9.38396639,   9.50803501,   9.63210363,
          9.75617225,   9.88024087,  10.00430949,  10.12837812,
         10.25244674,  10.37651536,  10.50058398,  10.6246526 ,
         10.74872122,  10.87278984,  10.99685846,  11.12092708,  11.2449957 ]),
 <a list of 100 Patch objects>)
```



```
In [14]: plt.hist(x2, bins=100)
```

```
Out[14]: (array([  1.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
                   0.,   0.,   0.,   0.,   0.,   0.,   0.,   3.,   0.,   0.,   1.,
                   3.,   3.,   2.,   3.,   5.,   7.,   4.,   5.,   7.,  10.,   7.,
                  10.,   3.,  10.,  10.,  15.,  16.,  18.,  18.,  17.,  16.,  20.,
                  23.,  22.,  38.,  21.,  25.,  28.,  28.,  38.,  27.,  37.,  15.,
                  30.,  30.,  34.,  38.,  29.,  31.,  28.,  22.,  27.,  25.,  14.,
                  13.,  17.,  24.,  10.,  12.,  11.,   8.,  12.,  13.,   6.,   7.,
                   7.,   6.,   8.,   2.,   4.,   1.,   1.,   1.,   3.,   2.,   0.,
                   0.,   1.,   3.,   0.,   1.,   0.,   1.,   0.,   0.,   0.,   1.,
                   1.]), array([ -1.32144464e+01,  -1.30625927e+01,  -1.29107389e+01,
                  -1.27588852e+01,  -1.26070314e+01,  -1.24551777e+01,
                  -1.23033239e+01,  -1.21514702e+01,  -1.19996164e+01,
                  -1.18477627e+01,  -1.16959089e+01,  -1.15440552e+01,
                  -1.13922014e+01,  -1.12403477e+01,  -1.10884939e+01,
                  -1.09366402e+01,  -1.07847864e+01,  -1.06329327e+01,
                  -1.04810789e+01,  -1.03292251e+01,  -1.01773714e+01,
                  -1.00255176e+01,  -9.87366390e+00,  -9.72181015e+00,
                  -9.56995639e+00,  -9.41810264e+00,  -9.26624889e+00,
                  -9.11439514e+00,  -8.96254139e+00,  -8.81068764e+00,
                  -8.65883389e+00,  -8.50698014e+00,  -8.35512639e+00,
                  -8.20327264e+00,  -8.05141889e+00,  -7.89956513e+00,
                  -7.74771138e+00,  -7.59585763e+00,  -7.44400388e+00,
                  -7.29215013e+00,  -7.14029638e+00,  -6.98844263e+00,
                  -6.83658888e+00,  -6.68473513e+00,  -6.53288138e+00,
                  -6.38102763e+00,  -6.22917387e+00,  -6.07732012e+00,
                  -5.92546637e+00,  -5.77361262e+00,  -5.62175887e+00,
                  -5.46990512e+00,  -5.31805137e+00,  -5.16619762e+00,
                  -5.01434387e+00,  -4.86249012e+00,  -4.71063637e+00,
                  -4.55878261e+00,  -4.40692886e+00,  -4.25507511e+00,
                  -4.10322136e+00,  -3.95136761e+00,  -3.79951386e+00,
                  -3.64766011e+00,  -3.49580636e+00,  -3.34395261e+00,
                  -3.19209886e+00,  -3.04024511e+00,  -2.88839135e+00,
                  -2.73653760e+00,  -2.58468385e+00,  -2.43283010e+00,
                  -2.28097635e+00,  -2.12912260e+00,  -1.97726885e+00,
                  -1.82541510e+00,  -1.67356135e+00,  -1.52170760e+00,
                  -1.36985385e+00,  -1.21800009e+00,  -1.06614634e+00,
                  -9.14292593e-01,  -7.62438842e-01,  -6.10585091e-01,
                  -4.58731340e-01,  -3.06877589e-01,  -1.55023839e-01,
                  -3.17008766e-03,   1.48683663e-01,   3.00537414e-01,
                   4.52391165e-01,   6.04244916e-01,   7.56098667e-01,
                   9.07952418e-01,   1.05980617e+00,   1.21165992e+00,
                   1.36351367e+00,   1.51536742e+00,   1.66722117e+00,
                   1.81907492e+00,   1.97092867e+00]), <a list of 100 Patch objects>)
```



In [15]:
```
print("x1 mean: {}, x2 mean: {}".format(x1.mean(), x2.mean()))
```

```
x1 mean: 5.003800218971892, x2 mean: -4.992537301817326
```

In [16]:
```
df1 = pd.DataFrame(dict(zip(['value','class'],[x1,c1])))
df2 = pd.DataFrame(dict(zip(['value','class'],[x2,c2])))
df = df1.append(df2)
df.describe()
```

Out[16]:

	value
count	2000.000000
mean	0.005631
std	5.391157
min	-13.214446
25%	-5.019632
50%	0.490347
75%	5.038763
max	11.244996

We will now select a sample of 80% from each of the two Normal distributions and use it to train our model to differentiate between the two (we will expect positive numbers to be grouped with x1 while negative numbers should be grouped with x2). We will then use the remaining 20% of the data to test the model; the results of this test are below.

In [17]:
```
X_train, X_test, y_train, y_test = model_selection.train_test_split(df[['value']],
df['class'], test_size=0.2)
```

In [18]:
```
clf = DecisionTreeClassifier(max_depth=2)
clf = clf.fit(X_train, y_train)
expected = y_test
predicted = clf.predict(X_test)

print(metrics.classification_report(expected, predicted))
```

```
             precision    recall  f1-score   support

          0       0.99      1.00      0.99       209
          1       1.00      0.98      0.99       191

avg / total       0.99      0.99      0.99       400
```

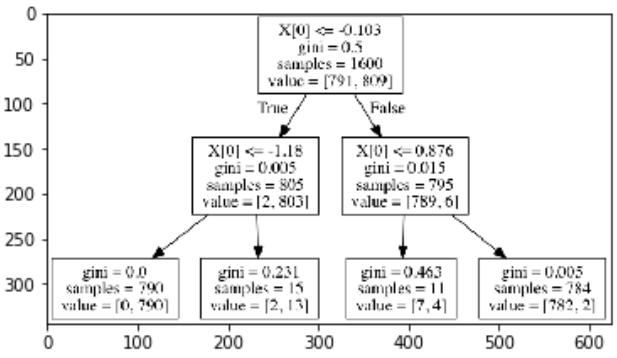
In [19]:
```
from sklearn.tree import _tree
threshold = clf.tree_.threshold
threshold[0]
```

Out[19]:
```
-0.10302744805812836
```

In [20]:
```
from sklearn import tree
tree.export_graphviz(clf, out_file='tree.dot')

import pydot

(graph,) = pydot.graph_from_dot_file('tree.dot')
graph.write_png('tree.png')
```

Out[20]: True

In [21]:
```
img = mpimg.imread('tree.png')
plt.imshow(img)

plt.show()
```



Naturally by combining two normal distributions on exact opposite sides of the y-axis, we would expect our threshold value for choosing which distribution a given value falls into would be a threshold of 0 on average, since a negative number should fall into the negative distribution (-5, 2) and a positive value should fall into the positive distribution (5,2), naturally. However, during this randomized run, our threshold value for the split actually tests if the number is <= -0.103 or > -0.103, a threshold value which is very close to 0, but not quite 0. In other runs, I've even had the threshold number be almost as big as 0.5! Hence, because we are taking a randomized distribution and then randomly sampling from that to form a decision tree, the threshold value might be some positive or some negative value centered around 0 that decides where to split to put the value in the correct positive or negative normal distribution.

Working with PCA on the Iris Dataset

In [22]:
```
df = pd.read_csv('iris.data', sep=',')
df.columns=['sepal_len', 'sepal_wid', 'petal_len', 'petal_wid', 'class']
pca = decomposition.PCA(n_components=2)
pca.fit(df.iloc[:,0:4])
X = pca.transform(df.iloc[:,0:4])
```

In [23]:
```
dataset = datasets.load_iris()
dataset.keys()
df = pd.DataFrame(dataset.data, columns=dataset.feature_names)
df['class'] = dataset.target_names[dataset.target]
df.head()
```

Out[23]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	class
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

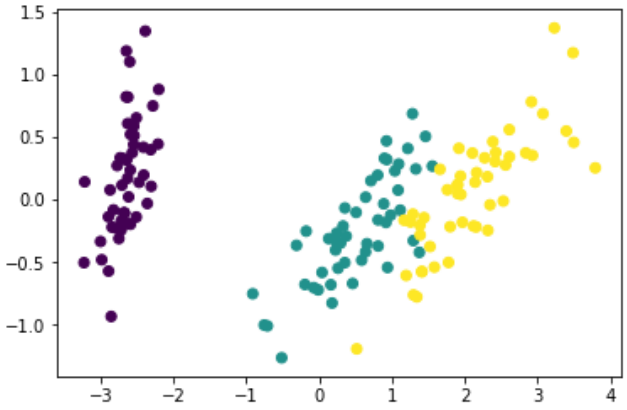
In [24]:
```
pca = decomposition.PCA(n_components=2)
pca.fit(df.iloc[:,0:4])
X = pca.transform(df.iloc[:,0:4])

X[:,0].var()
```

Out[24]:
```
4.1966751631979804
```

In [25]:
```
dfX = pd.DataFrame(data=X[:,0])
```

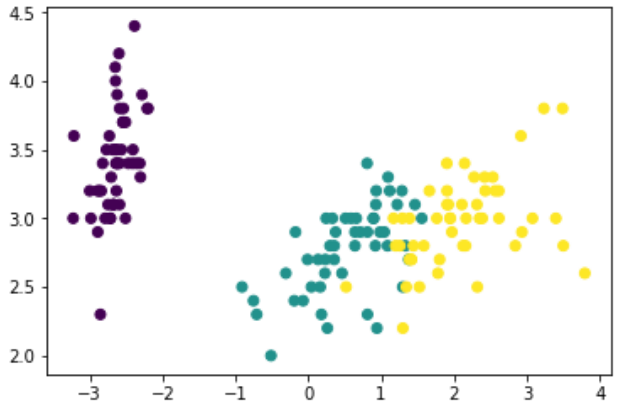
In [26]:
```
plt.scatter(X[:,0], X[:,1], c=dataset.target)
# original PCA
```

Out[26]:
```
<matplotlib.collections.PathCollection at 0x125699588>
```



In [27]:
```
plt.scatter(X[:,0], df['sepal width (cm)'], c=dataset.target)
```

Out[27]: <matplotlib.collections.PathCollection at 0x11f32cac8>

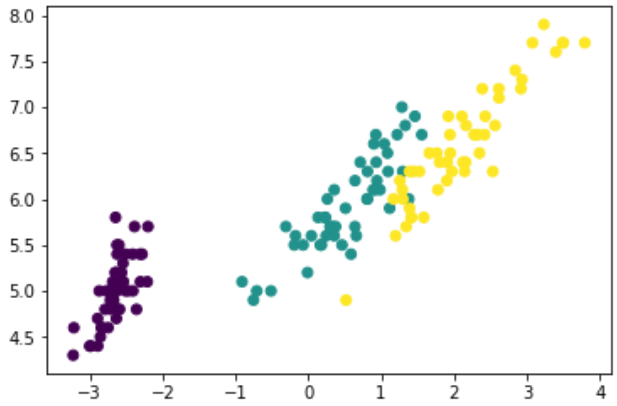


In [28]:
```
dfX[0].corr(df['sepal width (cm)'])
```

Out[28]: -0.38999337904750564

In [29]:
```
plt.scatter(X[:,0], df['sepal length (cm)'], c=dataset.target)
```

Out[29]: <matplotlib.collections.PathCollection at 0x125ef0668>

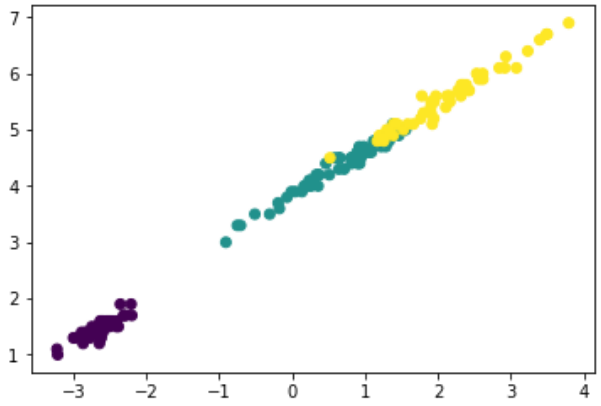


In [30]:
```
dfX[0].corr(df['sepal length (cm)'])
```

Out[30]: 0.89754488494076157

```
In [31]: plt.scatter(X[:,0], df['petal length (cm)'], c=dataset.target)
```

```
Out[31]: <matplotlib.collections.PathCollection at 0x125f8bb38>
```

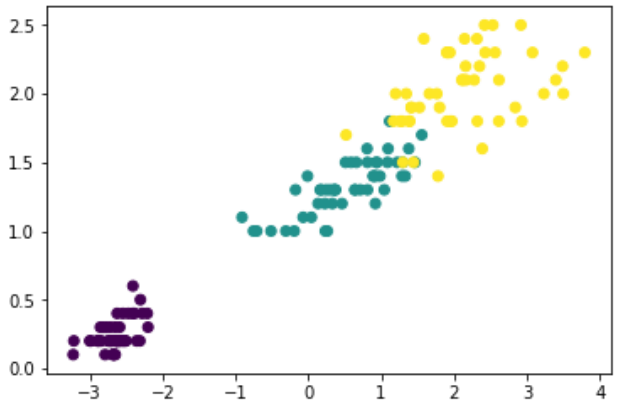



```
In [32]: dfX[0].corr(df['petal length (cm)'])
```

```
Out[32]: 0.99785405063545973
```

```
In [33]: plt.scatter(X[:,0], df['petal width (cm)'], c=dataset.target)
```

```
Out[33]: <matplotlib.collections.PathCollection at 0x1260297b8>
```




```
In [34]: dfX[0].corr(df['petal width (cm)'])
```

```
Out[34]: 0.96648418315379248
```

From the above four graphs, it is clear that the graph comparing the first component of our PCA (PC1) and petal length produce the tightest, most correlated scatter plot. When looking at the cosine distance which measures the angle between the two different axes in this case, we see that the distance is minimized almost entirely with this graph compared to one such as PC1 vs sepal width. A simple inspection from the above problem 3 shows that PC1 contains the largest amount of variation of all the components as petal length does the same out of the other three features. When we further solidify our inspection by actually calculating the correlation, we find that PC1 vs petal length gives a correlation of about 0.998, which is nearly 1, the most linear reading a correlation can give. With the assurance that PC1 vs sepal width gives the correlation closest to 0, we can assure ourselves that we have selected the correct pair of features.

In [35]:
```
pca = decomposition.PCA(n_components=4)
pca.fit(df.iloc[:,0:4])
X = pca.transform(df.iloc[:,0:4])
pca.explained_variance_
```

Out[35]:
```
array([ 4.22484077,  0.24224357,  0.07852391,  0.02368303])
```

In [36]:
```
pca.explained_variance_.sum()
```

Out[36]:
```
4.569291275167787
```

In [37]:
```
df.var().sum()
```

Out[37]:
```
4.5692912751677826
```

In [38]:
```
pca.explained_variance_ratio_.cumsum()
```

Out[38]:
```
array([ 0.92461621,  0.97763178,  0.99481691,  1.        ])
```

As alluded to in Problem 3, we see fully that when we compute the PCA with the same number of components as the original features, we find that we still capture the full variance of the original features. The advantage of PCA over using the original features, however, is that we can still capture a large majority of the variability by using less features. We see this in action when we look at the sum of the explained variance ratios. By using one component, we capture about 92.4% of the total variance, by using two, we capture about 97.8%, three we capture nearly 99.5%, and of course by using all four, we capture the full, total variability. Unlike the original attributes where the best one feature will get us is around 66%, one feature after PCA will get drastically more variability. To get 95% of our total variability of our original features would require all of three of our features, but after PCA, it only takes two features to get us far above the 95% mark. Hence, our PCA worked in decreasing our dimensions while not decreasing the majority of our variability to work with.

Clustering Data with an Optimal Number of Clusters

In [39]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn import preprocessing
from sklearn import metrics

from sklearn.datasets import load_boston

%matplotlib inline
```

In [40]:
```
dataset = load_boston()
print(dataset.DESCR)
```

```
Boston House Prices dataset
===========================

Notes
------
Data Set Characteristics:

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive

    :Median Value (attribute 14) is usually the target

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,000 sq.
ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 ot
herwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by tow
n
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
http://archive.ics.uci.edu/ml/datasets/Housing


This dataset was taken from the StatLib library which is maintained at Carnegie
Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnostics
...', Wiley, 1980.   N.B. Various transformations are used in the table on
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that a
ddress regression
problems.

**References**

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Dat
a and Sources of Collinearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Pr
oceedings on the Tenth International Conference of Machine Learning, 236-243, Un
iversity of Massachusetts, Amherst. Morgan Kaufmann.
   - many more! (see http://archive.ics.uci.edu/ml/datasets/Housing)
```

We will begin by scaling the dataset and then find the best number of clusters that reduces the overall silhouette score of the model.

In [41]:
```
df_scaled = pd.DataFrame(data=preprocessing.scale(dataset.data), columns=dataset.f
eature_names)
df_scaled.head()
```

Out[41]:

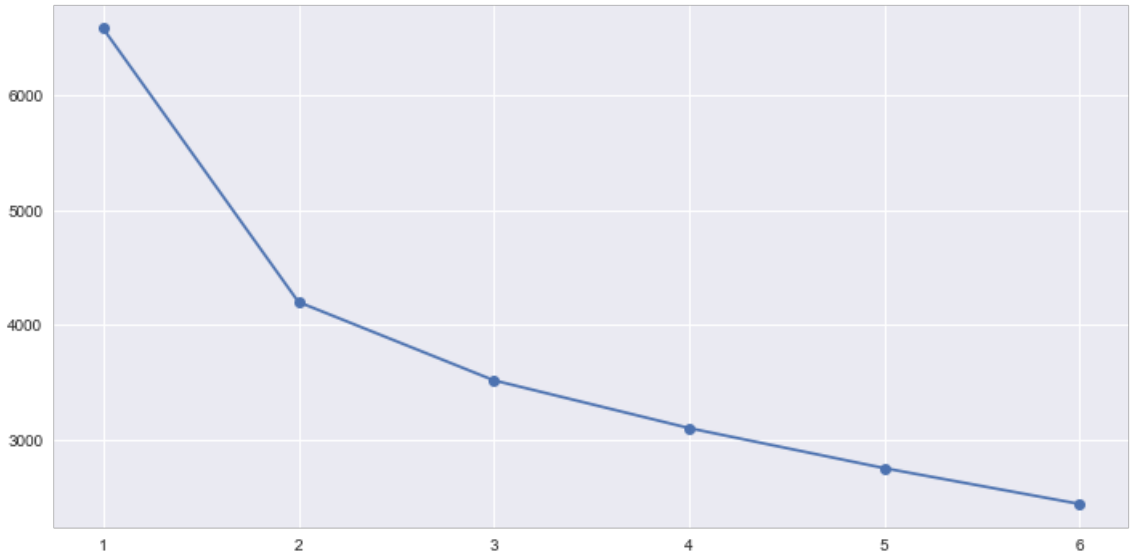
	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	
0	-0.417713	0.284830	-1.287909	-0.272599	-0.144217	0.413672	-0.120013	0.140214	-0.982843	-0.
1	-0.415269	-0.487722	-0.593381	-0.272599	-0.740262	0.194274	0.367166	0.557160	-0.867883	-0.
2	-0.415272	-0.487722	-0.593381	-0.272599	-0.740262	1.282714	-0.265812	0.557160	-0.867883	-0.
3	-0.414680	-0.487722	-1.306878	-0.272599	-0.835284	1.016303	-0.809889	1.077737	-0.752922	-1.
4	-0.410409	-0.487722	-1.306878	-0.272599	-0.835284	1.228577	-0.511180	1.077737	-0.752922	-1.

In [42]:
```
for n in range(2, 7):
    clust_model = KMeans(n_clusters=n, init='k-means++')
    clust_labels = clust_model.fit_predict(df_scaled)
    silhouette_avg = metrics.silhouette_score(df_scaled, clust_labels)
    print("Silhouette score for k = {} = {}".format(n, silhouette_avg))
```

```
Silhouette score for k = 2 = 0.35997734237402634
Silhouette score for k = 3 = 0.2573437376110001
Silhouette score for k = 4 = 0.2896770051612737
Silhouette score for k = 5 = 0.287487114888814
Silhouette score for k = 6 = 0.26139281284115573
```

In [43]:
```
# just to double check, here's an elbow analysis mostly
# taken from http://www.awesomestats.in/python-cluster-validation/

X = df_scaled[list(df_scaled.columns)]

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

cluster_range = range(1, 7)
cluster_errors = []

for num_clusters in cluster_range:
    clusters = KMeans(num_clusters)
    clusters.fit(X_scaled)
    cluster_errors.append(clusters.inertia_)

clusters_df = pd.DataFrame({ "num_clusters":cluster_range, "cluster_errors": clust
er_errors})

plt.figure(figsize=(12,6))
plt.plot(clusters_df.num_clusters, clusters_df.cluster_errors, marker = "o")
```

Out[43]:
```
[<matplotlib.lines.Line2D at 0x11e408470>]
```



In [44]:
```
# clearly from the silhouette scores above, the optimal value of k is 2 for the sc
aled data
clust_model = KMeans(n_clusters = 2, init='k-means++')
clust_labels = clust_model.fit_predict(df_scaled)
print(clust_labels)
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 0 0 1 0 0 0 0 0 1 0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

In [45]:
```
silhouette_avg = metrics.silhouette_score(df_scaled, clust_labels)
print(silhouette_avg)
```

```
0.359977342374
```

With optimal number of clusters being k=2, we will continue on by comparing the mean values for all features in each cluster as well as the centroid coordinates.

In [46]:
```
df_scaled['CLUST'] = clust_labels
C0 = df_scaled.loc[df_scaled['CLUST'] == 0]
C0.mean()
```

Out[46]:
```
CRIM      -0.388039
ZN         0.262392
INDUS     -0.620368
CHAS       0.002912
NOX       -0.584675
RM         0.243315
AGE       -0.435108
DIS        0.457222
RAD       -0.583801
TAX       -0.631460
PTRATIO   -0.285808
B          0.326451
LSTAT     -0.446421
CLUST      0.000000
dtype: float64
```

In [47]:
```
df_scaled['CLUST'] = clust_labels
C1 = df_scaled.loc[df_scaled['CLUST'] == 1]
C1.mean()
```

Out[47]:
```
CRIM       0.721270
ZN        -0.487722
INDUS      1.153113
CHAS      -0.005412
NOX        1.086769
RM        -0.452263
AGE        0.808760
DIS       -0.849865
RAD        1.085145
TAX        1.173731
PTRATIO    0.531248
B         -0.606793
LSTAT      0.829787
CLUST      1.000000
dtype: float64
```

In [48]:
```
print("Cluster coordinates: \n{}".format(clust_model.cluster_centers_))
```

```
Cluster coordinates:
[[-0.38803894  0.26239167 -0.62036759  0.00291182 -0.58467512  0.24331476
  -0.43510819  0.45722226 -0.58380115 -0.63145993 -0.28580826  0.32645106
  -0.44642061]
 [ 0.72127012 -0.48772236  1.15311264 -0.00541237  1.086769   -0.45226302
   0.80876041 -0.8498651   1.0851445   1.1737306   0.53124811 -0.60679321
   0.82978746]]
```

In [49]:
```
sns.clustermap(df_scaled, method='average', metric='euclidean', z_score=None, stan
dard_scale=None)
```

```
//anaconda/lib/python3.5/site-packages/matplotlib/cbook/deprecation.py:106: Matp
lotlibDeprecationWarning: The axisbg attribute was deprecated in version 2.0. Us
e facecolor instead.
  warnings.warn(message, mplDeprecation, stacklevel=1)
```

Out[49]: <seaborn.matrix.ClusterGrid at 0x12566a588>