3027020 - Noah Kuntner - Individual Project:

Algorithms assigned: K-NN and the Support Vector Machine

In [25]:
```
import pandas as pd
import numpy as np
```

First of all, I import pandas and numpy, which we will use later in this Notebook.

In [26]:
```
pd.set_option('display.max_columns', None)
```

Now I let the code read the .csv file.

In [27]:
```
df = pd.read_csv('mldata_0302702001.csv')
```

I delete the first column, as it was redundant.

In [28]:
```
df = df.drop(['Unnamed: 0'], axis= 1)
```

The output of the list looks like this:

In [29]:
```
df
```

Out[29]:

	label	num.feature 1	num.feature 2	num.feature 3	num.feature 4	num.feature 5	num.feature 6	num
0	0	5.278714	0.324092	0.281235	0.986022	0.979793	0.672533	2
1	1	-9.386273	0.883694	0.999989	-4.158979	0.009589	0.481705	-4
2	0	1.536305	0.509786	1.457337	1.820560	0.344157	-0.384595	1
3	1	-1.270615	0.491006	1.746655	-2.204529	0.397099	2.341209	C
4	0	-4.161705	0.179791	-0.305659	3.713710	-0.884374	2.150797	-4
5	1	5.150655	-0.685245	0.424581	-3.536314	-0.035324	-5.341769	4
6	1	-10.252747	-0.096146	0.412634	-1.285352	-1.034310	-1.629590	-4
7	1	-2.611854	0.432359	0.560259	0.636612	0.590489	-1.069463	-2
8	1	7.068904	1.268848	-0.193895	0.639236	0.251287	-2.813840	3
9	1	0.063814	-0.532857	-0.261324	-0.268284	0.098563	2.768212	-4

To gather general data on the dataset's value I use .describe()

In [30]:
```
df.describe()
```

Out[30]:

	label	num.feature 1	num.feature 2	num.feature 3	num.feature 4	num.feature 5	num.feature 6
count	1400.000000	1400.000000	1400.000000	1400.000000	1400.000000	1400.000000	1400.000000
mean	0.507857	-0.724115	0.091569	0.102644	-0.361414	0.087938	0.104113
std	0.500117	5.601112	1.013677	0.975019	2.397074	0.968176	2.716397
min	0.000000	-17.314077	-2.965912	-3.038995	-7.519363	-3.178170	-10.013249
25%	0.000000	-4.264076	-0.610171	-0.569367	-2.059120	-0.560448	-1.630251
50%	1.000000	-0.524906	0.082815	0.098251	-0.378130	0.077125	0.111857
75%	1.000000	2.947003	0.789409	0.740277	1.290954	0.716536	1.796137
max	1.000000	23.669955	3.326676	3.789623	9.400184	3.704749	8.586753

Now to control if the dataset has any missing values, I use .info()

In [31]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1400 entries, 0 to 1399
Data columns (total 31 columns):
label              1400 non-null int64
num.feature 1      1400 non-null float64
num.feature 2      1400 non-null float64
num.feature 3      1400 non-null float64
num.feature 4      1400 non-null float64
num.feature 5      1400 non-null float64
num.feature 6      1400 non-null float64
num.feature 7      1400 non-null float64
num.feature 8      1400 non-null float64
num.feature 9      1400 non-null float64
num.feature 10     1400 non-null float64
num.feature 11     1400 non-null float64
num.feature 12     1400 non-null float64
num.feature 13     1400 non-null float64
num.feature 14     1400 non-null float64
num.feature 15     1400 non-null float64
num.feature 16     1400 non-null float64
num.feature 17     1400 non-null float64
num.feature 18     1400 non-null float64
num.feature 19     1400 non-null float64
num.feature 20     1400 non-null float64
num.feature 21     1400 non-null float64
num.feature 22     1400 non-null float64
num.feature 23     1400 non-null float64
num.feature 24     1400 non-null float64
num.feature 25     1400 non-null float64
num.feature 26     1400 non-null float64
num.feature 27     1400 non-null float64
num.feature 28     1400 non-null float64
num.feature 29     1400 non-null float64
num.feature 30     1400 non-null float64
dtypes: float64(30), int64(1)
memory usage: 339.1 KB
```

Preprocessing

Now to normalize the dataframe:

In [32]:
```
from sklearn import preprocessing
```

I copy the df dataframe, but drop the first (and only categorical) variable.

In [33]:
```
df1 = df.drop(['label'], axis=1)
```

Now I preprocess our dataframe.

In [34]:
```
X_scaled = preprocessing.scale(df1)
```

In [35]:
```
df_scaled = pd.DataFrame(X_scaled)
```

Now I check if the scale succeeded.

In [36]:
```
df_scaled.describe()
```

Out[36]:

	0	1	2	3	4	5
count	1.400000e+03	1.400000e+03	1.400000e+03	1.400000e+03	1.400000e+03	1.400000e+03
mean	-1.630144e-17	4.060244e-17	-2.648675e-17	3.370320e-18	8.405974e-18	-7.876636e-17
std	1.000357e+00	1.000357e+00	1.000357e+00	1.000357e+00	1.000357e+00	1.000357e+00
min	-2.962963e+00	-3.017306e+00	-3.223282e+00	-2.987187e+00	-3.374672e+00	-3.725882e+00
25%	-6.322362e-01	-6.925194e-01	-6.894750e-01	-7.084944e-01	-6.699378e-01	-6.387075e-01
50%	3.557866e-02	-8.638482e-03	-4.506810e-03	-6.976060e-03	-1.117240e-02	2.851848e-03
75%	6.556608e-01	6.886707e-01	6.542036e-01	6.895733e-01	6.494926e-01	6.231153e-01
max	4.356775e+00	3.192598e+00	3.782793e+00	4.073752e+00	3.737032e+00	3.123870e+00

The mean of the dataframe is now way closer to 0, which tells us that the normalization has succeeded.

Coding

To get an overall view, I divide the columns into numerical and categorical variables.

In [37]:
```
columns = ['num.feature 1', 'num.feature 2', 'num.feature 3', 'num.feature 4', 'n
           'num.feature 11', 'num.feature 12', 'num.feature 13', 'num.feature 14'
           'num.feature 21', 'num.feature 22', 'num.feature 23' , 'num.feature 24
```

In [38]:
```
label = df['label']
```

Principal Component Analysis

Now I perform PCA and check if it actually improves the accuracy in the result in this case.

In [39]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
```

The following class is used to divide the data in train and validation set and gives us the possibility to perform PCA.

In [40]:
```
#class Data:
#    def __init__(self, columns, label, df = df, seed=42):
#        X = data[columns]
#        y = data[label[0]]
#
#        x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
#
#        self.X, self.y = x_train, y_train
#        self.vX, self.vy = x_test, y_test
#
#    def train(self):
#        return self.X, self.y
#
#    def valid(self):
#        return self.vX, self.vy
```

In [41]:
```
#df = df.as_matrix()
#pca = PCA(n_components=2)
#new_data = pca.fit_transform(df)
```

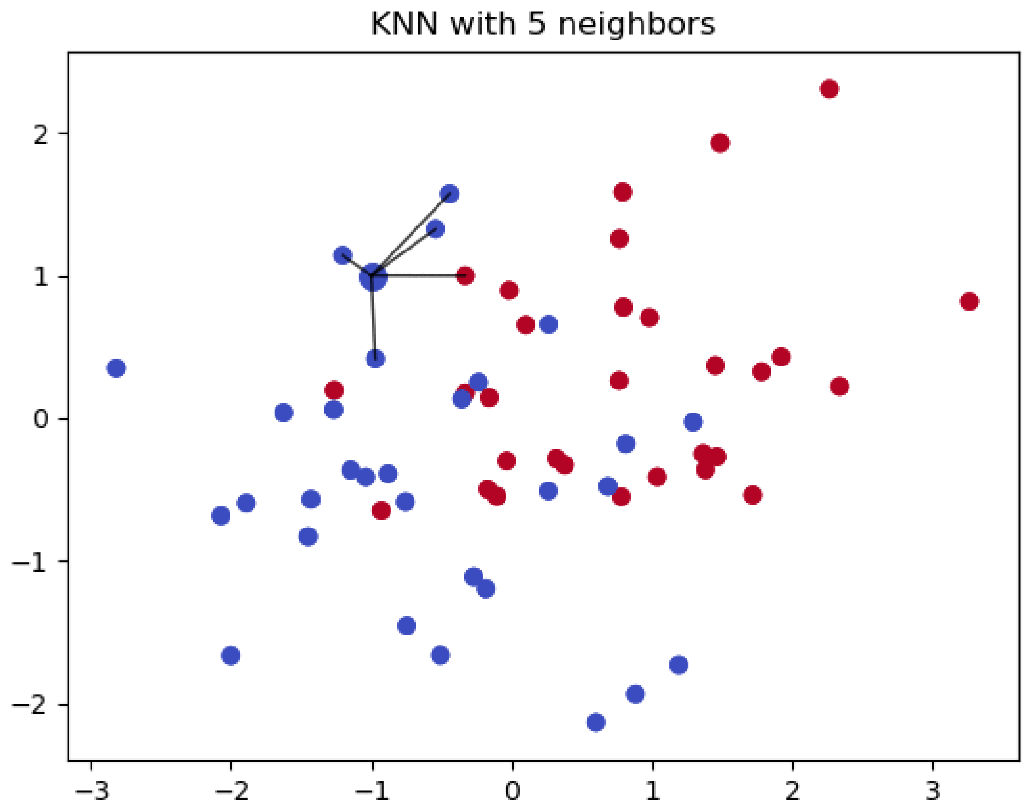
In [42]:
```
#print(pca.explained_variance_ratio_)
#print(pca.singular_values_)
#print(new_data)
```

K-Nearest Neighbour

Now I perform the K-NN algorithm.

In [43]:
```
from IPython.display import Image
from IPython.core.display import HTML
Image(url= "https://importq.files.wordpress.com/2017/11/knn_mov5.gif?w=640&zoom=2
```

Out[43]:



In [44]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
```

I define my X as the scaled version of my numerical dataframe, whereas the Y will be my categorical variable "label".

In [45]:
```
X = df_scaled
```

Now I perform the KNN-algorithm:

In [46]:
```
def knn(X, label, n_neighbors = 5, **knn_params):
    scores = np.zeros(n_neighbors)

    for i in range(1, n_neighbors):
        neigh = KNeighborsClassifier(i, **knn_params)
        scores[i] = np.mean(cross_val_score(neigh, X, label))



    print('n_neighbors:', np.argmax(scores))
    print('accuracy:', np.max(scores))
```

I have set the parameter 'n_neighbors' to 5, as it was giving me in this case an optimal result with minimal computation time (odd numbers).

In [47]:
```
knn(X, label)
```

```
n_neighbors: 3
accuracy: 0.8935662142
```

I did not use PCA in this case, as applying PCA worsened my accuracy be roughly 27%. Accuracy obtained with PCA: [0.615735847785]

In [48]:
```
from sklearn.neighbors import kneighbors_graph
A = kneighbors_graph(X, 5, mode='connectivity', include_self=True)
```

In [49]:
```
A.toarray()
```

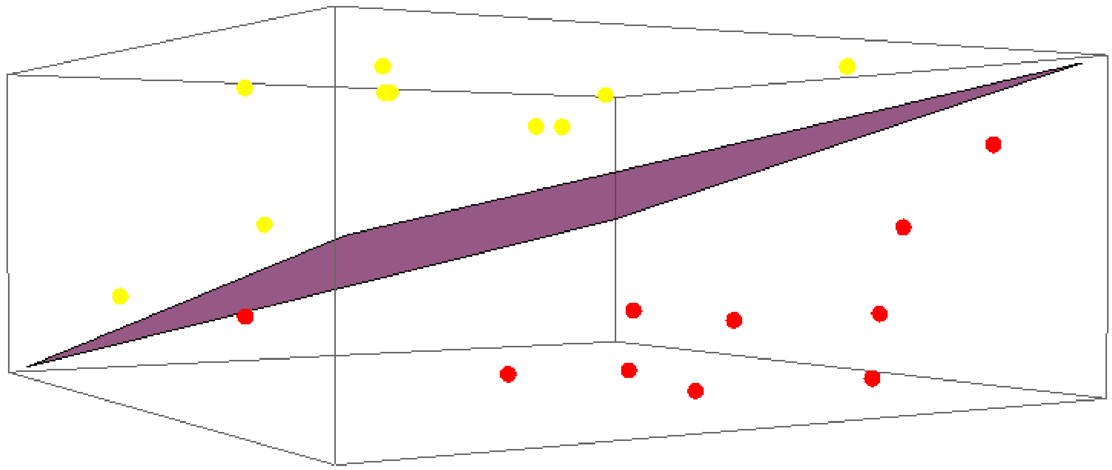
Out[49]:
```
array([[ 1.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  1.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  1., ...,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  0., ...,  1.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  1.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  1.]])
```

Support Vector Machine

Now I perform the Support Vector Machine. I import from sklearn SVM, SVC and train_test_split

In [50]:
```
Image(url= "http://prodata.swmed.edu/Lab/SVMl.gif")
```

Out[50]:



In [175]:
```
from sklearn.svm import LinearSVC
from sklearn import svm
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
```

We divide the dataframe into train and test sets:

In [161]:
```
X_train, X_test, y_train, y_test = train_test_split(X, label, test_size = 0.20, r
```

Now I take on the LinearSVC case: I have achieved the highest accuracy using the hinge loss, determining the max_iter to be 10000 and with the random state of 15.

In [172]:
```
svclassifier=LinearSVC(loss='hinge', max_iter=10000, random_state = 15)
svclassifier.fit(X_train, y_train)
```

Out[172]:
```
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='hinge', max_iter=10000, multi_class='ovr',
     penalty='l2', random_state=15, tol=0.0001, verbose=0)
```

Now I predict the y-value.

In [155]:
```
y_pred = svclassifier.predict(X_test)
```

Now I gather the accuracy results of the LinearSVC.

In [156]:
```
print(classification_report(y_test, y_pred))
```

```
             precision    recall  f1-score   support

          0       0.70      0.73      0.71       125
          1       0.77      0.75      0.76       155

avg / total       0.74      0.74      0.74       280
```

In [173]:
```
print(confusion_matrix(y_test, y_pred))
```

```
[[132   9]
 [ 11 128]]
```

From the total of 280 data points, 132 data points have been rightly determined to be 0 and 128 data points have been rightly determined to be 1. On the other hand, 9 data points have been wrongly classified as 0 and 11 data points have been wrongly classified as 1.

In this case I have not really received satisfying results, thus I will continue and try to outperform this LinearSVC in the following lines.

To gather optimal results I also followingly have used the Gaussian Kernel, as it was easily outperforming both the Polynomial and the Sigmoid Kernel (82% and 68%) and obviously the above LinearSVC (74%). Furthermore, I used as random state 15.

In [169]:
```
svclassifier = SVC(kernel='rbf', random_state = 15)
svclassifier.fit(X_train, y_train)
```

Out[169]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=15, shrinking=True,
  tol=0.001, verbose=False)
```

Now I predict the y-value.

In [170]:
```
y_pred = svclassifier.predict(X_test)
```

Followingly, I gather the accuracy results.

In [171]:
```
from sklearn.metrics import classification_report, confusion_matrix
print(classification_report(y_test, y_pred))
```

```
             precision    recall  f1-score   support

          0       0.92      0.94      0.93       141
          1       0.93      0.92      0.93       139

avg / total       0.93      0.93      0.93       280
```

In [152]:
```
print(confusion_matrix(y_test, y_pred))
```

```
[[116   9]
 [  9 146]]
```

In the confusion matrix we can see how many points the Support Vector Machine was able to classify right and how many he classified wrongly in the respective case.

From the total of 280 data points, 116 data points have been rightly determined to be 0 and 146 data points have been rightly determined to be 1. On the other hand, 9 data points have been wrongly classified as 0 and 9 data points have been wrongly classified as 1.

Summary

In the both cases I've achieved rather high results in the accuracy (89% and 93%). Only with the LinearSVC I could not obtain a good score (74%).

In the K-NN case I have used 5 neighbors as k, as it was the lowest odd number with which I have achieved the highest accuracy. I did not use PCA, as it was severely worsening my result. On the other hand, in the Support Vector Machine case I have used the Gausian Kernel to optimize my results, as I was not satisfied with the results that the LinearSVC was providing..

However, the Support Vector Machine (in the case with the Gaussian Kernel) beat the K-NN algorithm by approximately 4% and thus proved (in this case) to be the slightly better algorithm with this data set.

Furthermore, the K-NN algorithm took more time in gathering the results than the Support Vector Machine, which is due to the fact that K-NN is not very efficient in the classification of high dimensional data sets.