Question 1: Effect of Normalization, FeatureExtraction and Distance Metrics

1.1 Tasks

1.1.1 Train/Test Data Split

In [108]:
```
# loading the librarries libraries
import numpy as np
import pandas as pd
import random
import seaborn as sns
sns.set(style="ticks", color_codes=True)
from sklearn import neighbors
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
```

In [109]:
```
# reading and loading the data
#Columns/Features
D = ['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar', 'chlo
L = 'quality'
C = 'color'
DL = D + [L]
DC = D + [C]
DLC = DL + [C]

#Loading Data set
wine_r = pd.read_csv("winequality-red.csv", sep=';')
#Loading Data set
wine_w = pd.read_csv("winequality-white.csv", sep=';')
wine_w= wine_w.copy()
wine_w[C]= np.zeros(wine_w.shape[0])
wine_r[C]= np.ones(wine_r.shape[0])
wine = pd.concat([wine_w,wine_r])


# lets understand the dataframes we have with us right now
print("Lets understand the data shapes : ")
print(" ")
print("number of columns represented in D is :",len(D))
print("shape of wine_w is : ", wine_w.shape)
print("shape of wine_r is : ", wine_r.shape)
print("shape of dataframe wine is :", wine.shape)
print("shape of wine [D] is", wine[D].shape)
print(" ")
print(" ")

# Splitting the dataset into train and test data
X = wine[D]
y_c = wine[C]
y_q = wine[L]
ran = 42
X_train_c, X_test_c, y_train_c, y_test_c = train_test_split(X, y_c, test_size=0.
X_train_q, X_test_q, y_train_q, y_test_q = train_test_split(X, y_q, test_size=0.

#Lets see the shape of splitted data
print("Shape of dataframes used/created in the split :")
print(" ")
print("shape of X is : ", X.shape)
print("shape of y_c is : ", y_c.shape)
print("shape of X_train_c is :", X_train_c.shape)
print("shape of X_test_c is :", X_test_c.shape)
print("shape of y_train_c is :", y_train_c.shape)
print("shape of y_test_c is :", y_test_c.shape)
print("shape of y_q is : ", y_q.shape)
print("shape of X_train_q is :", X_train_q.shape)
print("shape of X_test_q is :", X_test_q.shape)
print("shape of y_train_q is :", y_train_q.shape)
print("shape of y_test_q is :", y_test_q.shape)
```

```
number of columns represented in D is : 11
shape of wine_w is :  (4898, 13)
```

```
shape of wine_r is :  (1599, 13)
shape of dataframe wine is : (6497, 13)
shape of wine [D] is (6497, 11)


Shape of dataframes used/created in the split :

shape of X is :  (6497, 11)
shape of y_c is :  (6497,)
shape of X_train_c is : (5197, 11)
shape of X_test_c is : (1300, 11)
shape of y_train_c is : (5197,)
shape of y_test_c is : (1300,)
shape of y_q is :  (6497,)
shape of X_train_q is : (5197, 11)
shape of X_test_q is : (1300, 11)
shape of y_train_q is : (5197,)
shape of y_test_q is : (1300,)
```

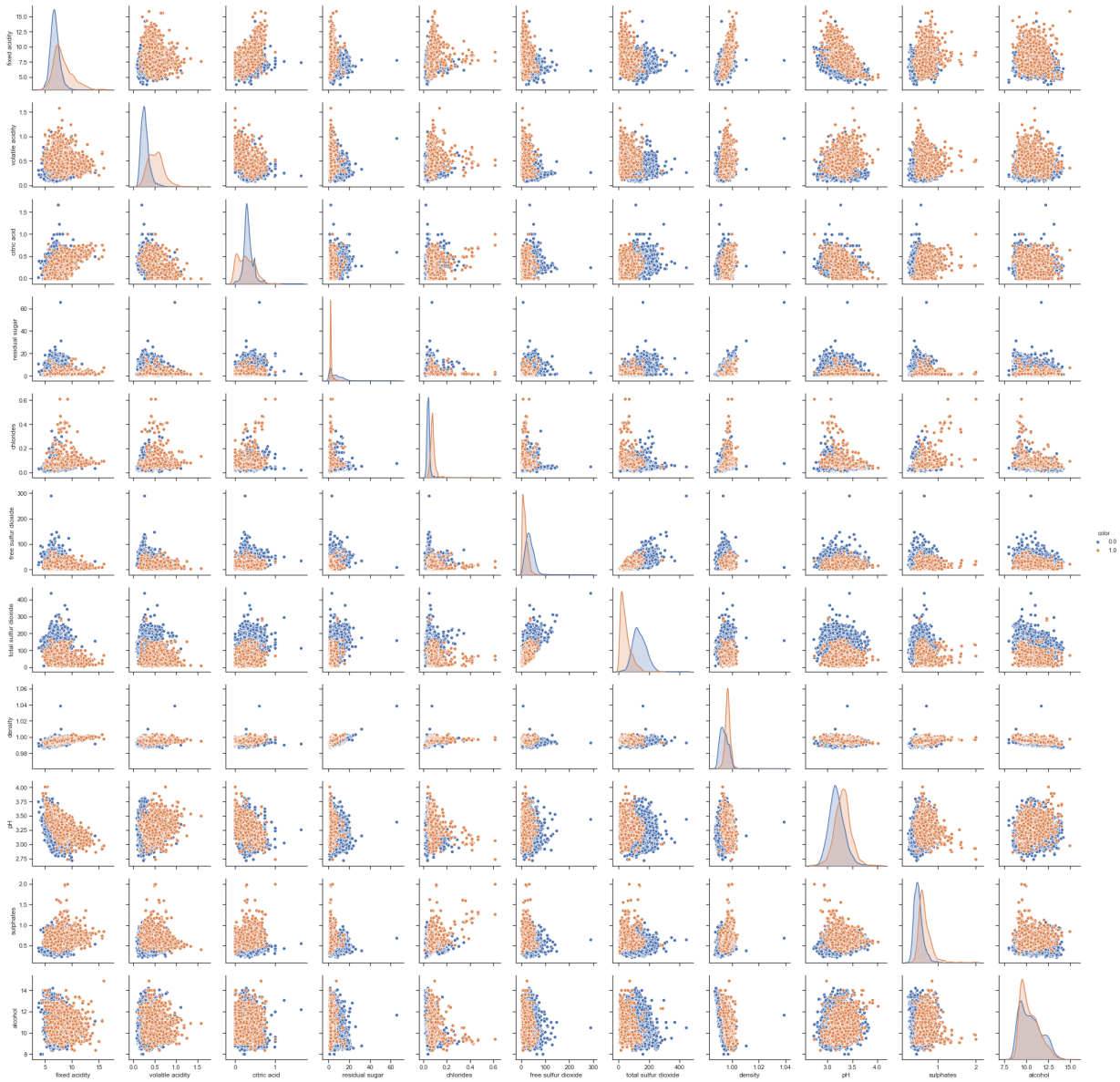
1.1.2 Normalization

In [110]:
```
#importing standard scalar, used for z-score normalization
from sklearn.preprocessing import StandardScaler
scalar = StandardScaler(copy=False, with_mean=True, with_std=True)
X_scaled = wine[D]
X_scaled =scalar.fit_transform(X_scaled)
X_scaled = pd.DataFrame(data = X_scaled, columns = D)
scalar = StandardScaler()
X_train_scaled_c = X_train_c.copy()
X_train_scaled_q = X_train_q.copy()
X_train_scaled_c = scalar.fit_transform(X_train_scaled_c)
X_test_scaled_c = scalar.transform(X_test_c)
X_train_scaled_q = scalar.fit_transform(X_train_scaled_q)
X_test_scaled_q = scalar.transform(X_test_q)
```

Pairplot for non-normalized Data

In [111]:
```
sns.pairplot(wine[DC], vars = wine[DC].columns[:-1], hue = 'color')
```

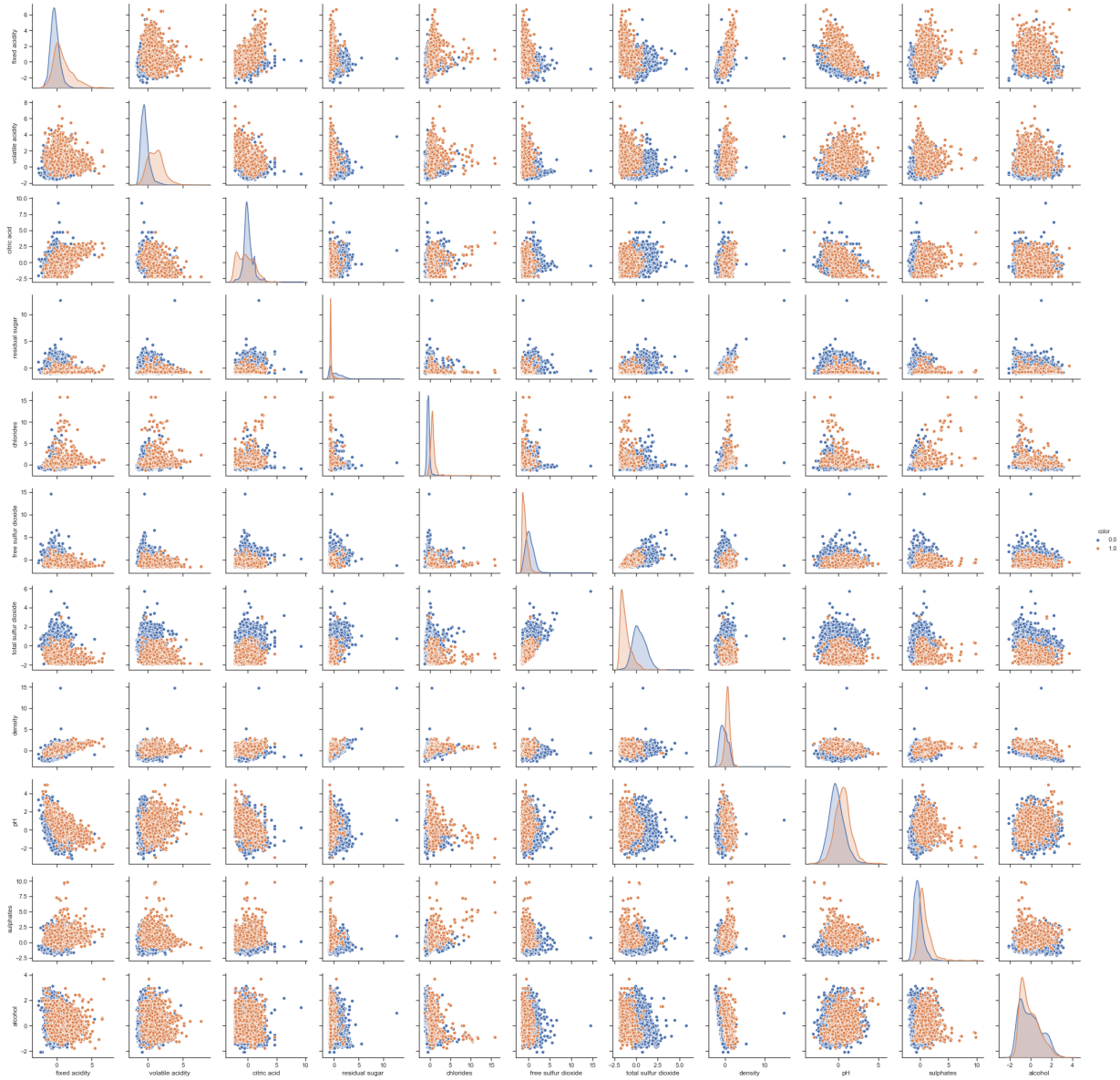
Out[111]:
```
<seaborn.axisgrid.PairGrid at 0x1e204a0f518>
```



Pairplot for normailized Dataset

In [112]:
```
yc_new= y_c.reset_index()
yc_new = yc_new.drop(['index'], axis =1)
X_scaled ['color'] = yc_new
sns.pairplot(X_scaled,vars = X_scaled.columns[:-1], hue = 'color')
```

Out[112]: <seaborn.axisgrid.PairGrid at 0x1e2122ca7f0>



Comparing Pairplot for normalized and non- nonrmalized features

It can be visualized from above plots :

the normalized plots are a bit more symmetric and not tend to look too elliptical compared to non-normalized plots. I believe as the normalized plot, are Z-score normalized and hence they represent the variation of normalized variance among the features, whereas the non-normalized

plot show variation of features among each other. Hence, the non-normalized plot tend to look affected by the actual values (cause of different scales of each feature), and hence do not provide a clear picture of variations of variances amongst different features.

Z-score converts all features to a common scale with an average of zero and standard deviation of one. The average of zero means that it avoids introducing aggregation distortions stemming from differences in feature means.

As different features have different scales, normalizing these features by z-score would make the plots normalized, the plots would still show the relationship between two variables maintaining the dispersion between features but would be less affected by the difference in scales of features.

1.1.3 Classification : Color

KNN Classification for wine color on normalized data with all features

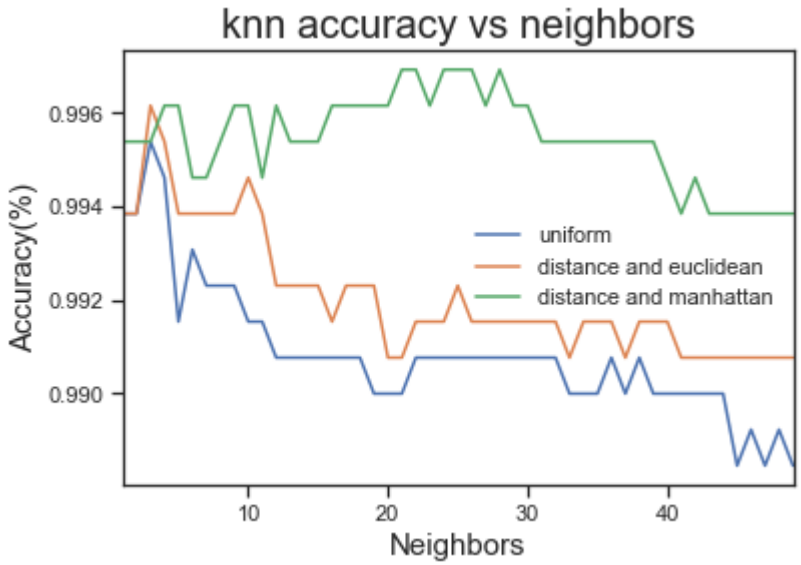
In [113]:
```
## lets work on normalized data for color prediction for the wine
#we must not forget to transform the test data too

scalar = StandardScaler(copy=False, with_mean=True, with_std=True)
X_train_scaled_c = X_train_c.copy()
X_train_scaled_q = X_train_q.copy()
X_train_scaled_c = scalar.fit_transform(X_train_scaled_c)
X_train_scaled_q  = scalar.fit_transform(X_train_scaled_q)



X_test_scaled_c = scalar.transform(X_test_c)
n_neighborslist = list(range(1,50))
col_names=['uniform','distance and euclidean','distance and manhattan']
accarray = np.zeros((len(n_neighborslist),3))


#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights=col_names[0])
    neigh.fit(X_train_scaled_c, y_train_c)
    y_pred_c = neigh.predict(X_test_scaled_c)
    accscore = accuracy_score(y_test_c, y_pred_c)
    acc.at[k,col_names[0]] = accscore


for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance', me
    neigh.fit(X_train_scaled_c, y_train_c)
    y_pred_c = neigh.predict(X_test_scaled_c)
    accscore = accuracy_score(y_test_c, y_pred_c)
    acc.at[k,col_names[1]] = accscore


for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance', me
    neigh.fit(X_train_scaled_c, y_train_c)
    y_pred_c = neigh.predict(X_test_scaled_c)
    accscore = accuracy_score(y_test_c, y_pred_c)
    acc.at[k,col_names[2]] = accscore


acc.describe()
acc.head()
```

Out[113]:

	uniform	distance and euclidean	distance and manhattan
0	0.000000	0.000000	0.000000
1	0.993846	0.993846	0.995385
2	0.993846	0.993846	0.995385
3	0.995385	0.996154	0.995385
4	0.994615	0.995385	0.996154

In [114]:
```
graph = acc[1:].plot.line()
graph.set_title('knn accuracy vs neighbors', fontsize = 20)
graph.set_xlabel("Neighbors", fontsize = 15)
graph.set_ylabel("Accuracy(%)", fontsize = 15)
```

Out[114]:
```
Text(0, 0.5, 'Accuracy(%)')
```



Trying diffrent metrics on normalized wine data for color to compare performances (All features are used)

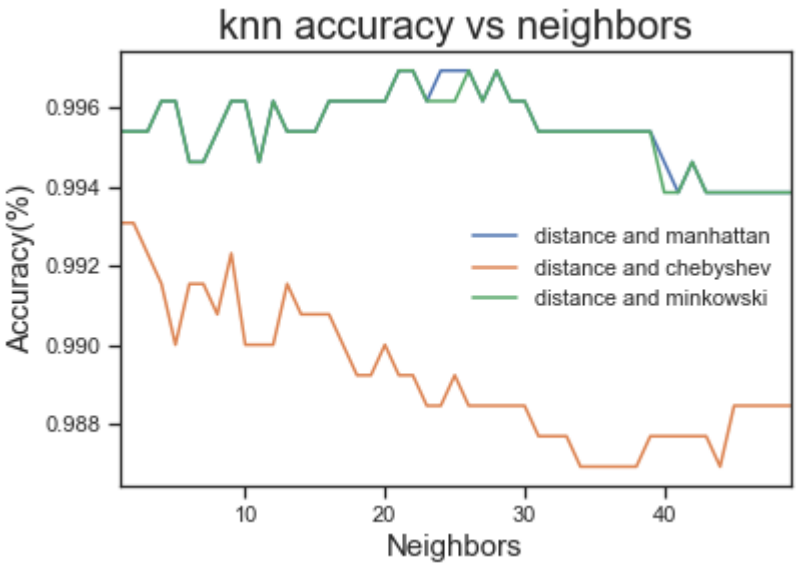
In [185]:
```
# Let us try multiple possilities for normalized datset
n_neighborslist = list(range(1,50))
col_names=['distance and manhattan','distance and chebyshev', 'distance and mink
accarray = np.zeros((len(n_neighborslist),3))


#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance', me
    neigh.fit(X_train_scaled_c, y_train_c)
    y_pred_c = neigh.predict(X_test_scaled_c)
    accscore = accuracy_score(y_test_c, y_pred_c)
    acc.at[k,col_names[0]] = accscore

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance', me
    neigh.fit(X_train_scaled_c, y_train_c)
    y_pred_c = neigh.predict(X_test_scaled_c)
    accscore = accuracy_score(y_test_c, y_pred_c)
    acc.at[k,col_names[1]] = accscore

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance', me
    neigh.fit(X_train_scaled_c, y_train_c)
    y_pred_c = neigh.predict(X_test_scaled_c)
    accscore = accuracy_score(y_test_c, y_pred_c)
    acc.at[k,col_names[2]] = accscore


acc.describe()
acc.head()
```

Out[185]:

	distance and manhattan	distance and chebyshev	distance and minkowski
0	0.000000	0.000000	0.000000
1	0.995385	0.993077	0.995385
2	0.995385	0.993077	0.995385
3	0.995385	0.992308	0.995385
4	0.996154	0.991538	0.996154

In [186]:
```
graph = acc[1:].plot.line()
graph.set_title('knn accuracy vs neighbors', fontsize = 20)
graph.set_xlabel("Neighbors", fontsize = 15)
graph.set_ylabel("Accuracy(%)", fontsize = 15)
```

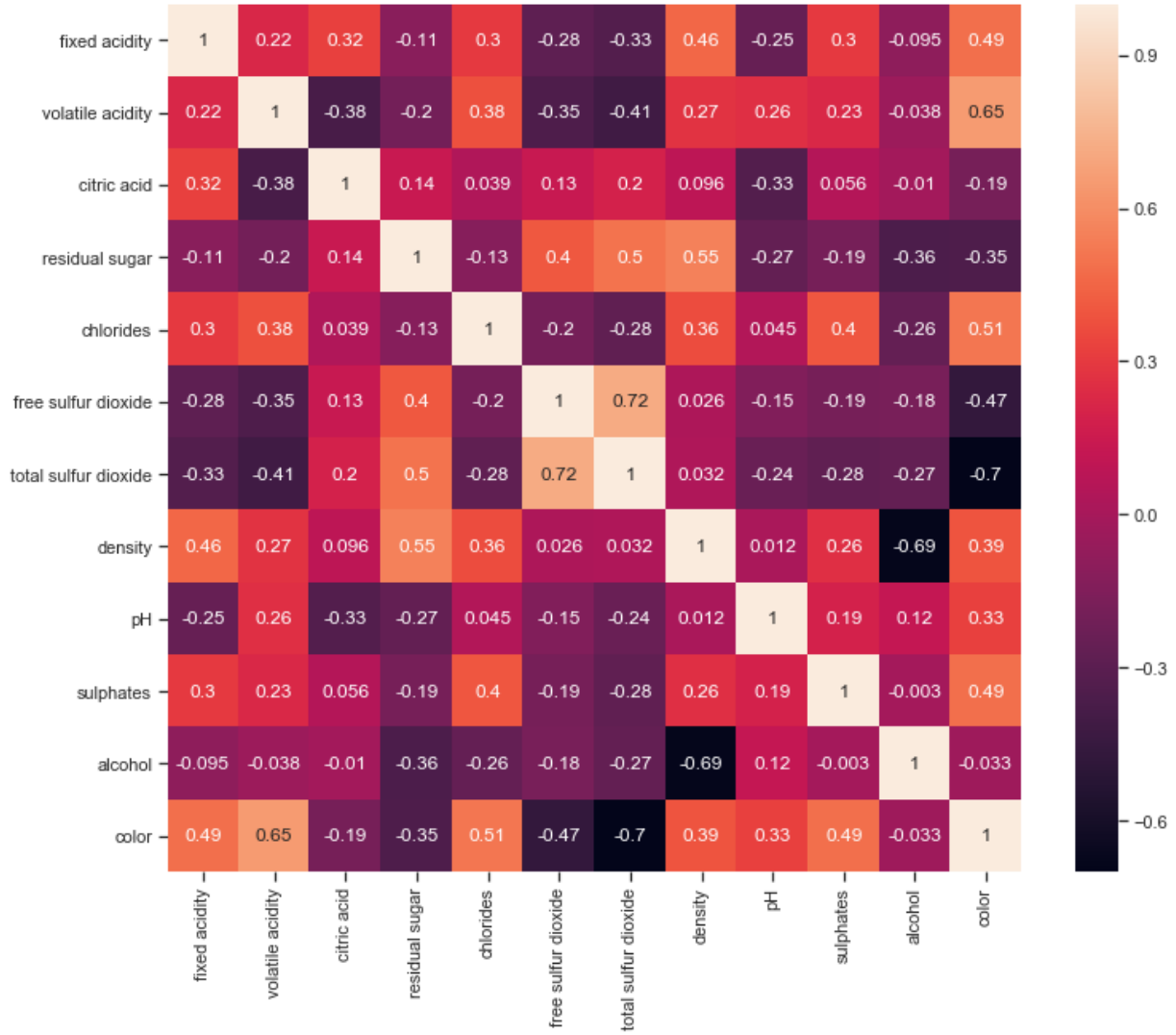
Out[186]:
```
Text(0, 0.5, 'Accuracy(%)')
```



1.1.4 Feature Selection : Color

Using Pearson Correlation coefficient for feature selection

In [117]:
```
# lets see the correlation among features
#Using Pearson Correlation
plt.figure(figsize=(12,10))
cor = X_scaled.corr()
sns.heatmap(cor, annot=True)
#cmap=plt.cm.Reds
plt.show()
```



running KNN on normalized data with a subset of features selected using pearson correlation coefficient with color

In [118]:

```
#Selecting the 4 most correlated features
#Total sulpher oxide(-0.7),Volatile acidity(0.65), chloride(0.51), fixed acidity
selected_features_c  = ['total sulfur dioxide','volatile acidity', 'chlorides','
X_sel_c = wine[selected_features_c]
X_sel_train_c, X_sel_test_c, y_sel_train_c, y_sel_test_c = train_test_split(X_se
X_sel_train_scaled_c = X_sel_train_c.copy()
scalar.fit_transform(X_sel_train_scaled_c)
#we must not forget to transform the test data too
X_sel_test_scaled_c = scalar.transform(X_sel_test_c)

n_neighborslist = list(range(1,50))
col_names=['uniform','distance and euclidean','distance and manhattan']
accarray = np.zeros((len(n_neighborslist),3))


#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights=col_names[0])
    neigh.fit(X_sel_train_scaled_c, y_sel_train_c)
    y_sel_pred_c = neigh.predict(X_sel_test_scaled_c)
    accscore = accuracy_score(y_sel_test_c, y_sel_pred_c)
    acc.at[k,col_names[0]] = accscore

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance', me
    neigh.fit(X_sel_train_scaled_c, y_sel_train_c)
    y_sel_pred_c = neigh.predict(X_sel_test_scaled_c)
    accscore = accuracy_score(y_sel_test_c, y_sel_pred_c)
    acc.at[k,col_names[1]] = accscore

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance', me
    neigh.fit(X_sel_train_scaled_c, y_sel_train_c)
    y_sel_pred_c = neigh.predict(X_sel_test_scaled_c)
    accscore = accuracy_score(y_sel_test_c, y_sel_pred_c)
    acc.at[k,col_names[2]] = accscore




acc.describe()
```

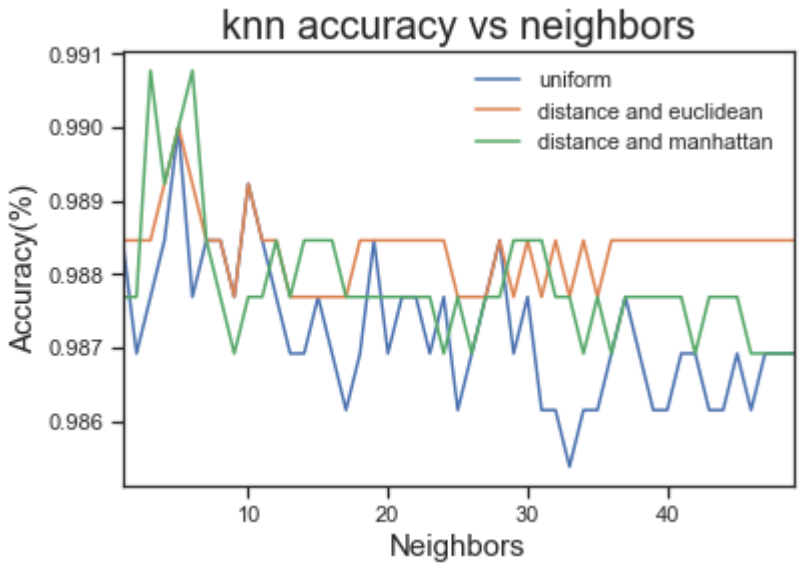
Out[118]:

	uniform	distance and euclidean	distance and manhattan
count	50.000000	50.000000	50.000000
mean	0.967477	0.968569	0.968108
std	0.139617	0.139773	0.139708
min	0.000000	0.000000	0.000000
25%	0.986346	0.987692	0.987692
50%	0.986923	0.988462	0.987692

	uniform	distance and euclidean	distance and manhattan
75%	0.987692	0.988462	0.987692
max	0.990000	0.990000	0.990769

In [119]:
```
graph = acc[1:].plot.line()
graph.set_title('knn accuracy vs neighbors', fontsize = 20)
graph.set_xlabel("Neighbors", fontsize = 15)
graph.set_ylabel("Accuracy(%)", fontsize = 15)
```

Out[119]:
```
Text(0, 0.5, 'Accuracy(%)')
```



1.1.5 Feature Extraction: Color

1.1.5.1 PCA : Color

Running KNN on normalized and PCA tranformed data with 5 principal components for wine color prediction

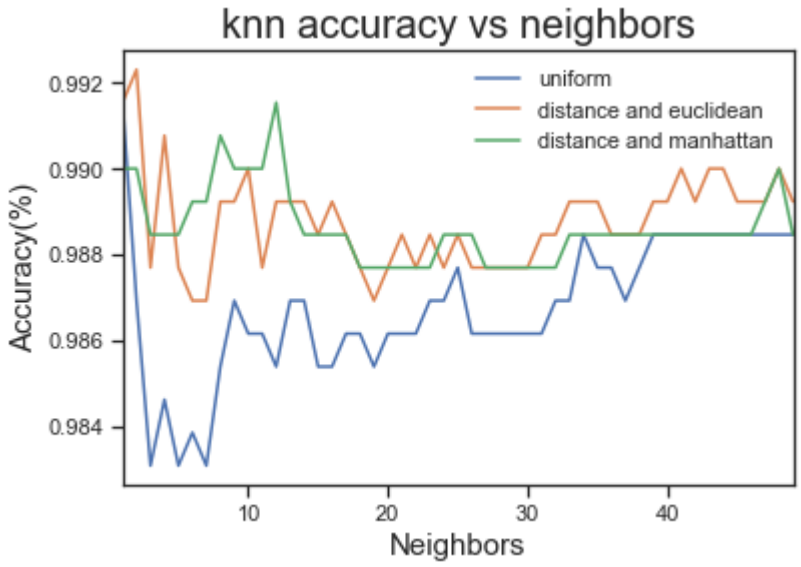
In [120]:
```
from sklearn.decomposition import PCA
pca = PCA(n_components = 5,random_state = 42)
principalComponents = pca.fit_transform(X_train_scaled_c)
col_name_pca = ["Principal Component " + str(i) for i in range(1,6)]
X_pca_train_c = pd.DataFrame(data = principalComponents, columns = col_name_pca)
y_pca_train_c = y_train_c
X_pca_test_c = pca.transform(X_test_scaled_c)
y_pca_test_c = y_test_c

n_neighborslist = list(range(1,50))
col_names=['uniform','distance and euclidean','distance and manhattan']
accarray = np.zeros((len(n_neighborslist),3))

#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights=col_names[0])
    neigh.fit(X_pca_train_c, y_pca_train_c)
    y_pca_pred_c = neigh.predict(X_pca_test_c)
    accscore = accuracy_score(y_pca_test_c, y_pca_pred_c)
    acc.at[k,col_names[0]] = accscore

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights= 'distance', me
    neigh.fit(X_pca_train_c, y_pca_train_c)
    y_pca_pred_c = neigh.predict(X_pca_test_c)
    accscore = accuracy_score(y_pca_test_c, y_pca_pred_c)
    acc.at[k,col_names[1]] = accscore

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights= 'distance', me
    neigh.fit(X_pca_train_c, y_pca_train_c)
    y_pca_pred_c = neigh.predict(X_pca_test_c)
    accscore = accuracy_score(y_pca_test_c, y_pca_pred_c)
    acc.at[k,col_names[2]] = accscore



acc.describe()
```

Out[120]:

	uniform	distance and euclidean	distance and manhattan
count	50.000000	50.000000	50.000000
mean	0.967015	0.969000	0.968862
std	0.139557	0.139838	0.139817
min	0.000000	0.000000	0.000000
25%	0.986154	0.987692	0.987885
50%	0.986923	0.988462	0.988462
75%	0.988269	0.989231	0.988462
max	0.991538	0.992308	0.991538

In [121]:
```
graph = acc[1:].plot.line()
graph.set_title('knn accuracy vs neighbors', fontsize = 20)
graph.set_xlabel("Neighbors", fontsize = 15)
graph.set_ylabel("Accuracy(%)", fontsize = 15)
```

Out[121]:
```
Text(0, 0.5, 'Accuracy(%)')
```



1.1.5.2 LDA : Color

Running KNN on normalized and LDA tranformed data with1 component for wine color prediction

In [122]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
lda = LinearDiscriminantAnalysis()
col_name_lda = ["LDA component 1"]
lda = LinearDiscriminantAnalysis(n_components = 1)
ldaComponents = lda.fit_transform(X_train_scaled_c, y_train_c)
X_lda_train_c = pd.DataFrame(data = ldaComponents, columns = col_name_lda)
y_lda_train_c = y_train_c
y_lda_test_c = y_test_c
X_lda_test_c = lda.transform(X_test_scaled_c)
n_neighborslist = list(range(1,50))
col_names=['uniform','distance and euclidean','distance and manhattan']
accarray = np.zeros((len(n_neighborslist),3))



#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights=col_names[0])
    neigh.fit(X_lda_train_c, y_lda_train_c)
    y_lda_pred_c = neigh.predict(X_lda_test_c)
    accscore = accuracy_score(y_lda_test_c, y_lda_pred_c)
    acc.at[k,col_names[0]] = accscore

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights= 'distance', me
    neigh.fit(X_lda_train_c, y_lda_train_c)
    y_lda_pred_c = neigh.predict(X_lda_test_c)
    accscore = accuracy_score(y_lda_test_c, y_lda_pred_c)
    acc.at[k,col_names[1]] = accscore

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights= 'distance', me
    neigh.fit(X_lda_train_c, y_lda_train_c)
    y_lda_pred_c = neigh.predict(X_lda_test_c)
    accscore = accuracy_score(y_lda_test_c, y_lda_pred_c)
    acc.at[k,col_names[2]] = accscore



acc.describe()
```

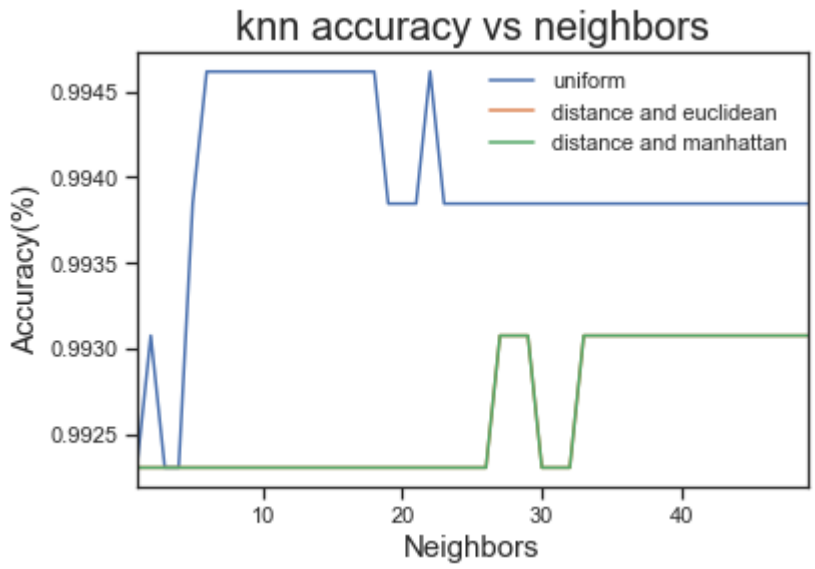
Out[122]:

	uniform	distance and euclidean	distance and manhattan
count	50.000000	50.000000	50.000000
mean	0.974077	0.972769	0.972769
std	0.140568	0.140378	0.140378
min	0.000000	0.000000	0.000000
25%	0.993846	0.992308	0.992308
50%	0.993846	0.992308	0.992308
75%	0.994615	0.993077	0.993077

	uniform	distance and euclidean	distance and manhattan
max	0.994615	0.993077	0.993077

In [123]:
```
graph = acc[1:].plot.line()
graph.set_title('knn accuracy vs neighbors', fontsize = 20)
graph.set_xlabel("Neighbors", fontsize = 15)
graph.set_ylabel("Accuracy(%)", fontsize = 15)
```

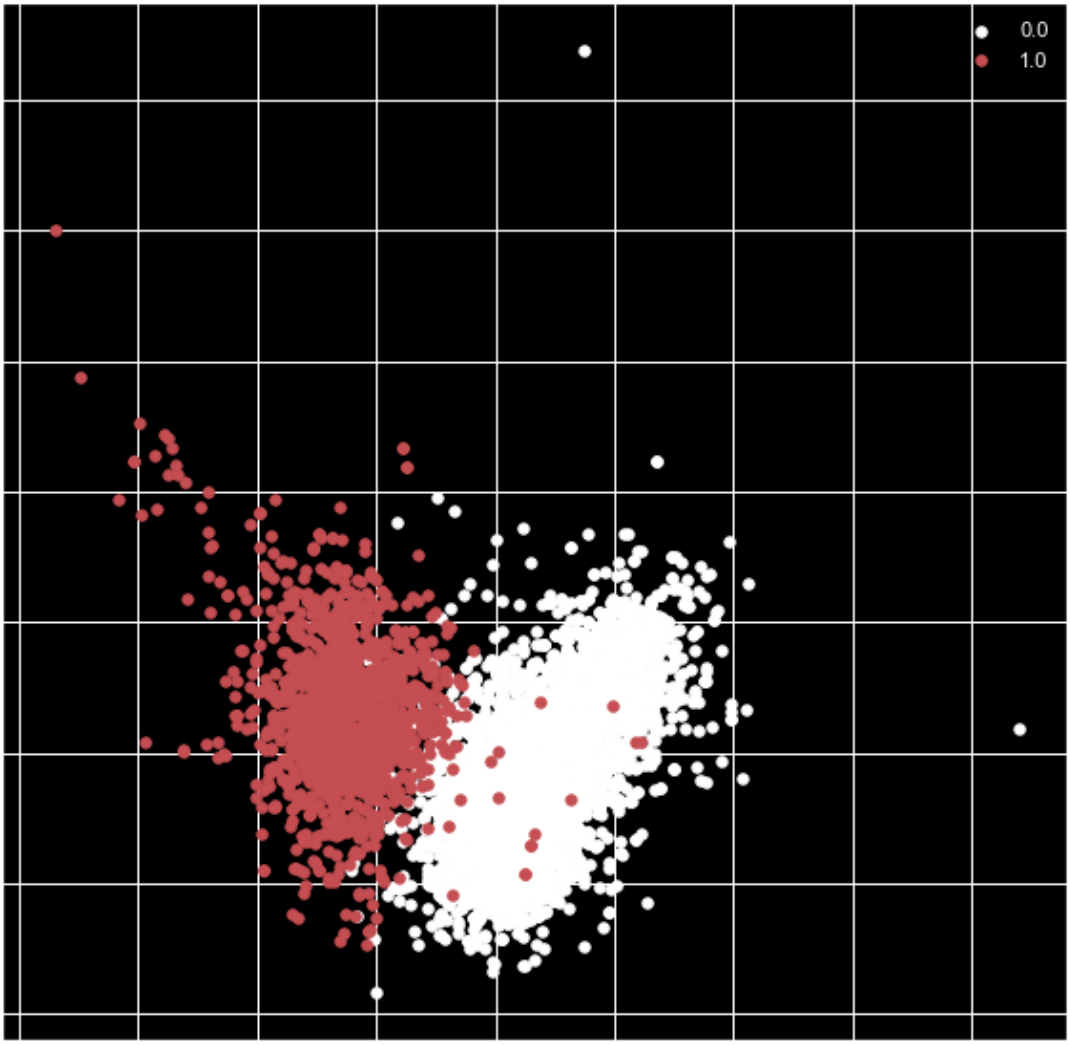
Out[123]:
```
Text(0, 0.5, 'Accuracy(%)')
```



Comparing the KNN classification accuracy on PCA and LDA features, maximum accuracy achieved using LDA features for color as a target variable is 99.46% which is greater than the accuracy obtained using PCA features of 99.23%. Thus can conclude that LDA works better compared to PCA in this case.

Visualization in 2D using PCA for Different wine colors : White and Red

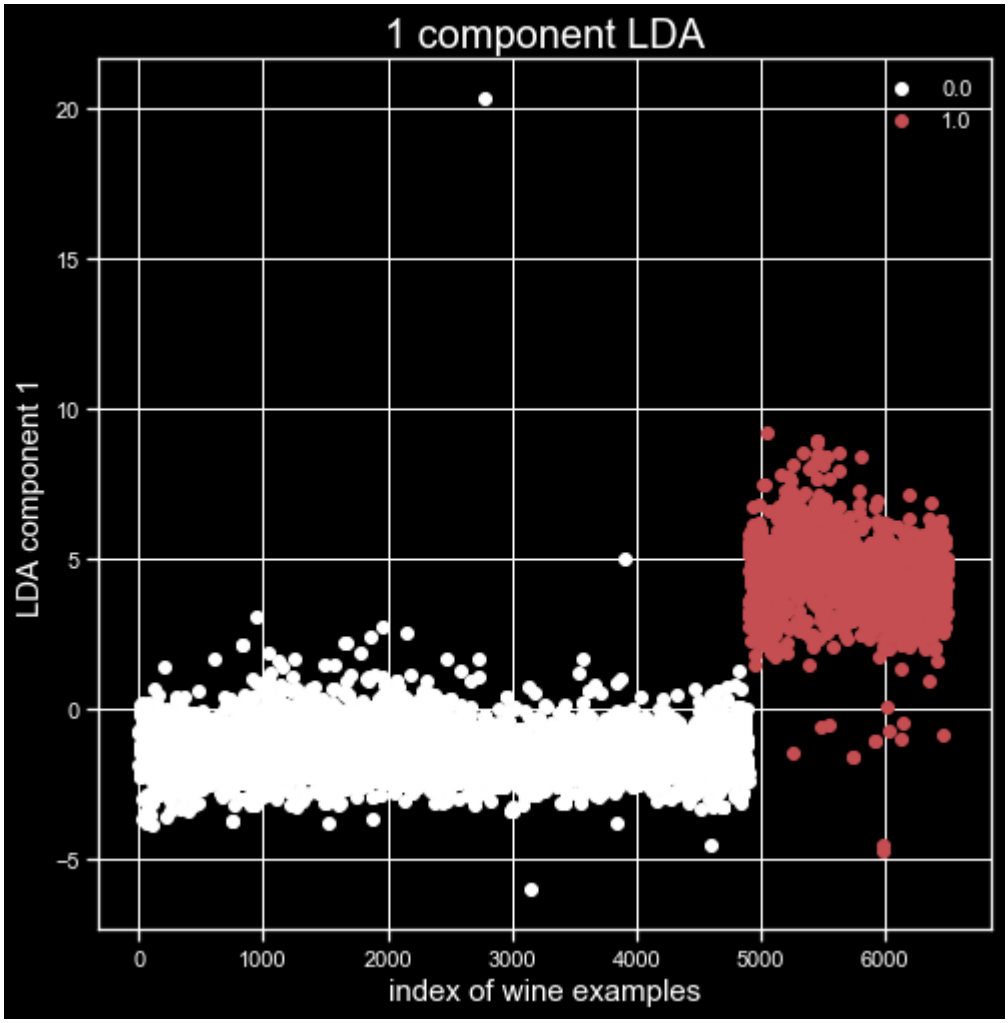
In [124]:
```
#for data visulaization I will be using full data
X_full_c = wine[D]
# further this data needs to standarized
from sklearn.preprocessing import StandardScaler
scalar = StandardScaler(copy = False)
X_full_scaled_c = X_full_c.copy()
scalar.fit_transform(X_full_scaled_c)
# lets fit and transform data to two principal components on PCA
pca = PCA(n_components = 2)
principalComponents = pca.fit_transform(X_full_scaled_c)
column_names = ["Principal component 1", "Principal component 2"]
PCA_2d = pd.DataFrame(data = principalComponents, columns = column_names)
yc = wine[C]
yc_new = yc.reset_index()
yc_new = yc_new.drop(['index'], axis =1)
final_PCA_2d_df = pd.DataFrame.join(self = PCA_2d, other = yc_new)
fig = plt.figure(figsize = (10,10))
plt.style.use('dark_background')
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_title('2 component PCA', fontsize = 20)
wine_colors = [0.0, 1.0]
colors = ['w','r']
for wine_color, color in zip(wine_colors,colors):
    indicesTokeep = final_PCA_2d_df['color'] == wine_color
    ax.scatter(final_PCA_2d_df.loc[indicesTokeep, 'Principal component 1']
               , final_PCA_2d_df.loc[indicesTokeep, 'Principal component 2']
               , c = color
               , s =30)
ax.legend(wine_colors)
ax.grid()
```



visuliazation in 1D using LDA

In [125]:
```
# I would be visualizing on the whole data
X_full_scaled_c
yc_new
plt.style.use('dark_background')
lda = LinearDiscriminantAnalysis(n_components = 1)
ldaComponents = lda.fit_transform(X_full_scaled_c, yc_new)
lda_1d_c = pd.DataFrame(data = ldaComponents, columns= ['LDA component'])
lda_1d_final_c = pd.DataFrame.join(lda_1d_c, yc_new)
lda_1d_final_c.rename_axis('index')
lda_1d_final_c =lda_1d_final_c.assign(new_index = lambda z: z.index)
lda_1d_final_c.shape

# lets visualize the thing
figure = plt.figure(figsize = (8,8))
ax = figure.add_subplot(1,1,1)
ax.set_ylabel('LDA component 1', fontsize = 15)
ax.set_xlabel('index of wine examples', fontsize = 15)
ax.set_title('1 component LDA', fontsize = 20)
wine_colors = [0.0, 1.0]
colors = ['w','r']
for wine_color, color in zip(wine_colors,colors):
    indicesTokeep = lda_1d_final_c['color'] == wine_color
    ax.scatter(x = lda_1d_final_c.loc[indicesTokeep,'new_index'],y = lda_1d_final
               , c = color)
ax.legend(wine_colors)
ax.grid()
```

```
C:\Users\aksha\Anaconda3\lib\site-packages\sklearn\utils\validation.py:724: Dat
aConversionWarning: A column-vector y was passed when a 1d array was expected.
Please change the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```



1.1.3 Classifiacation : Quality

1.1.3.1 KNN Classification for wine Quality on normalized data with all features

In [126]:
```
## lets work on normalized data for Quality prediction for the wine
#we must not forget to transform the test data too
from sklearn.preprocessing import StandardScaler
s = StandardScaler()
X_train_scaled_q= s.fit_transform(X_train_q)
X_test_scaled_q = s.transform(X_test_q)

sns.set(style="ticks", color_codes=True)

n_neighborslist = list(range(1,50))
col_names=['uniform','distance and euclidean','distance and manhattan']
accarray = np.zeros((len(n_neighborslist),3))


#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights=col_names[0])
    neigh.fit(X_train_scaled_q, y_train_q)
    y_pred_q = neigh.predict(X_test_scaled_q)
    accscore = accuracy_score(y_test_q, y_pred_q)
    acc.at[k,col_names[0]] = accscore


for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance', me
    neigh.fit(X_train_scaled_q, y_train_q)
    y_pred_q = neigh.predict(X_test_scaled_q)
    accscore = accuracy_score(y_test_q, y_pred_q)
    acc.at[k,col_names[1]] = accscore


for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance', me
    neigh.fit(X_train_scaled_q, y_train_q)
    y_pred_q = neigh.predict(X_test_scaled_q)
    accscore = accuracy_score(y_test_q, y_pred_q)
    acc.at[k,col_names[2]] = accscore


acc.describe()
```

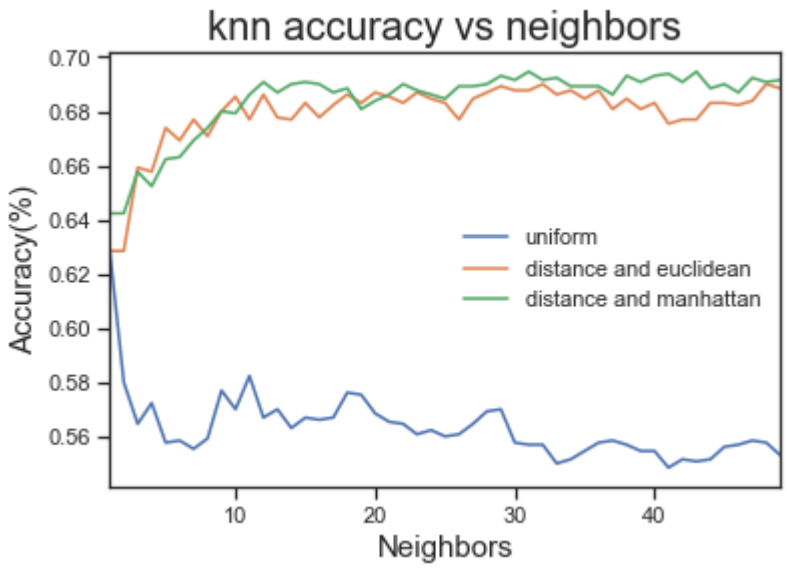
Out[126]:

	uniform	distance and euclidean	distance and manhattan
count	50.000000	50.000000	50.000000
mean	0.551954	0.665769	0.670262
std	0.080610	0.096883	0.097561
min	0.000000	0.000000	0.000000
25%	0.556346	0.676923	0.684038
50%	0.559615	0.683077	0.689231
75%	0.566923	0.686154	0.690769

	uniform	distance and euclidean	distance and manhattan
max	0.628462	0.690000	0.694615

In [127]:
```
graph = acc[1:].plot.line()
graph.set_title('knn accuracy vs neighbors', fontsize = 20)
graph.set_xlabel("Neighbors", fontsize = 15)
graph.set_ylabel("Accuracy(%)", fontsize = 15)
```

Out[127]:
```
Text(0, 0.5, 'Accuracy(%)')
```



Trying diffrent metrics on normalized wine data for quality to compare performances (All features are used)

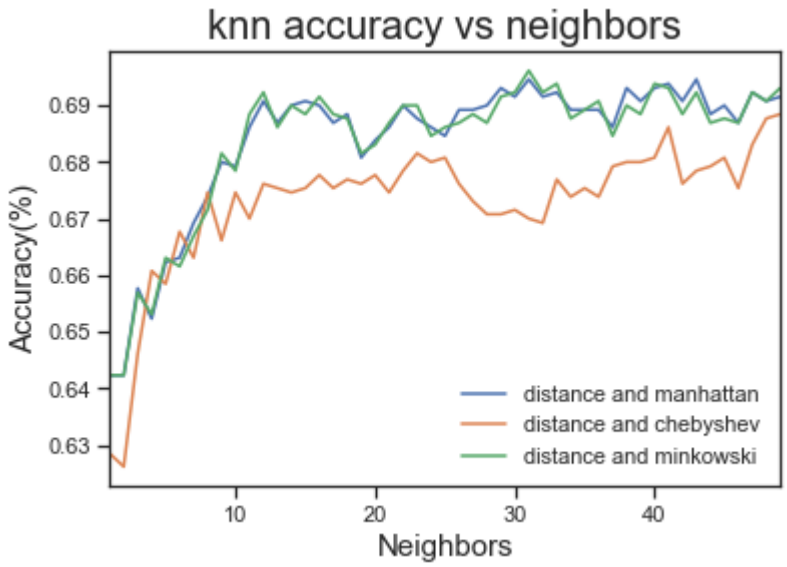
In [128]:
```
# Let us try multiple possilities for normalized datset
n_neighborslist = list(range(1,50))
col_names=['distance and manhattan','distance and chebyshev', 'distance and mink
#'distance and minkowski','distance and wminkowski', 'distance and seuclidean','
accarray = np.zeros((len(n_neighborslist),3))


#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance', me
    neigh.fit(X_train_scaled_q, y_train_q)
    y_pred_q = neigh.predict(X_test_scaled_q)
    accscore = accuracy_score(y_test_q, y_pred_q)
    acc.at[k,col_names[0]] = accscore

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance', me
    neigh.fit(X_train_scaled_q, y_train_q)
    y_pred_q = neigh.predict(X_test_scaled_q)
    accscore = accuracy_score(y_test_q, y_pred_q)
    acc.at[k,col_names[1]] = accscore

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance', me
    neigh.fit(X_train_scaled_q, y_train_q)
    y_pred_q = neigh.predict(X_test_scaled_q)
    accscore = accuracy_score(y_test_q, y_pred_q)
    acc.at[k,col_names[2]] = accscore

acc.describe()
```

Out[128]:

	distance and manhattan	distance and chebyshev	distance and minkowski
count	50.000000	50.000000	50.000000
mean	0.670262	0.659477	0.669954
std	0.097561	0.095904	0.097524
min	0.000000	0.000000	0.000000
25%	0.684038	0.670769	0.683462
50%	0.689231	0.675385	0.688077
75%	0.690769	0.679038	0.690769
max	0.694615	0.688462	0.696154

In [129]:
```
graph = acc[1:].plot.line()
graph.set_title('knn accuracy vs neighbors', fontsize = 20)
graph.set_xlabel("Neighbors", fontsize = 15)
graph.set_ylabel("Accuracy(%)", fontsize = 15)
```

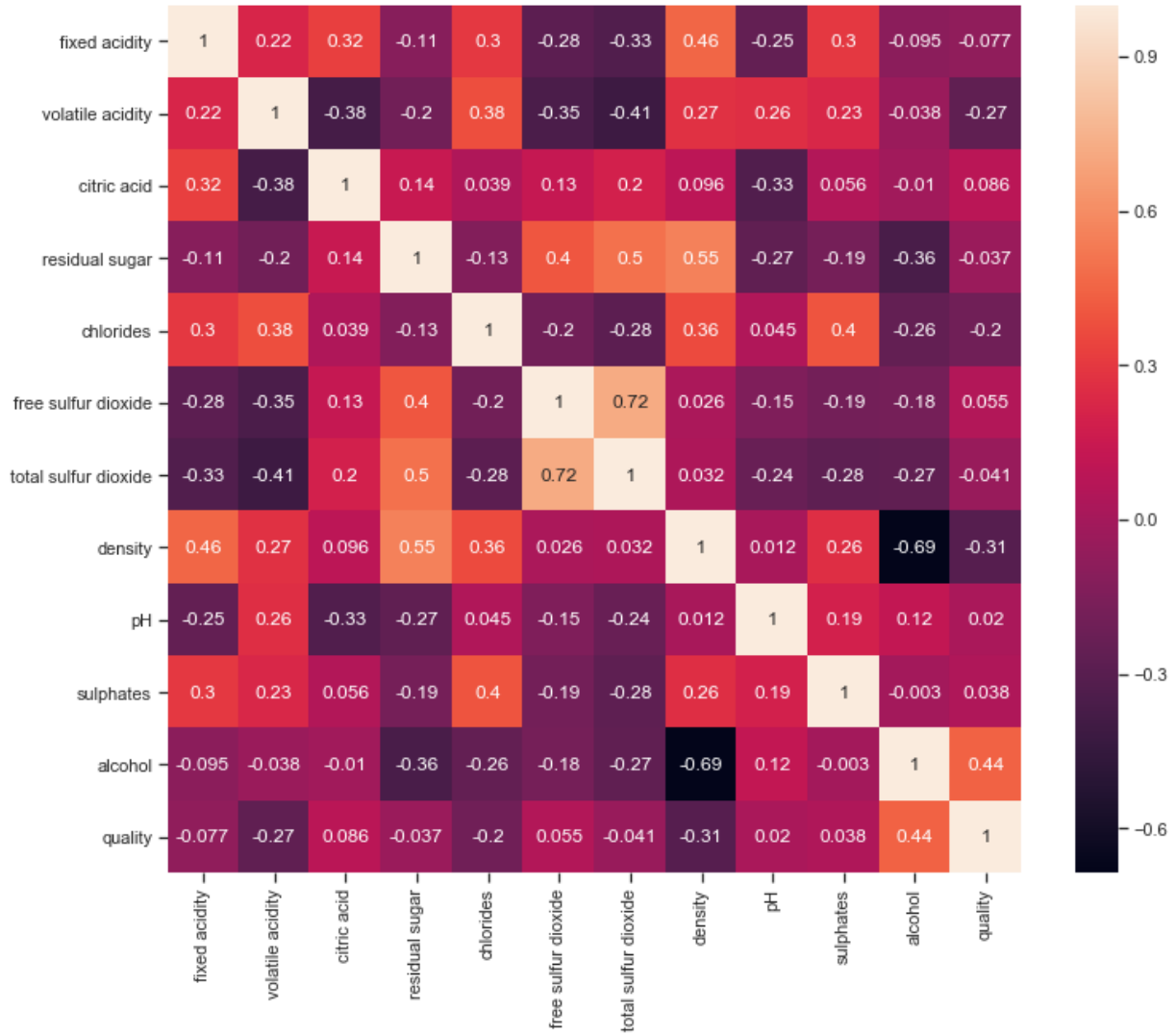
Out[129]:
```
Text(0, 0.5, 'Accuracy(%)')
```



1.1.4 Feature Selection : Quality

Using Pearson Correlation coefficient to select features

In [130]:
```
#Using Pearson Correlation
plt.figure(figsize=(12,10))
cor = wine[DL].corr()
sns.heatmap(cor, annot=True)
#cmap=plt.cm.Reds
plt.show()
```



1.1.4.1 Running KNN on normalized data for quality prediction with a subset of features ('alcohol','density', 'volatile acidity','chlorides')

In [131]:
```
# selecting most correlated features : alcohol (0.44),density(-0.31),volatile ac
selected_features_q  = ['alcohol','density', 'volatile acidity','chlorides']
X_sel_q = wine[selected_features_q]
X_sel_train_q, X_sel_test_q, y_sel_train_q, y_sel_test_q = train_test_split(X_se
X_sel_train_scaled_q = X_sel_train_q.copy()
scalar.fit_transform(X_sel_train_scaled_q)
## lets work on normalized data for quality prediction for the wine
#we must not forget to transform the test data too
X_sel_test_scaled_q = scalar.transform(X_sel_test_q)
n_neighborslist = list(range(1,50))
col_names=['uniform','distance and euclidean','distance and manhattan']
accarray = np.zeros((len(n_neighborslist),3))


#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights=col_names[0])
    neigh.fit(X_sel_train_scaled_q, y_sel_train_q)
    y_sel_pred_q = neigh.predict(X_sel_test_scaled_q)
    accscore = accuracy_score(y_sel_test_q, y_sel_pred_q)
    acc.at[k,col_names[0]] = accscore

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance', me
    neigh.fit(X_sel_train_scaled_q, y_sel_train_q)
    y_sel_pred_q = neigh.predict(X_sel_test_scaled_q)
    accscore = accuracy_score(y_sel_test_q, y_sel_pred_q)
    acc.at[k,col_names[1]] = accscore

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance', me
    neigh.fit(X_sel_train_scaled_q, y_sel_train_q)
    y_sel_pred_q = neigh.predict(X_sel_test_scaled_q)
    accscore = accuracy_score(y_sel_test_q, y_sel_pred_q)
    acc.at[k,col_names[2]] = accscore




acc.describe()
```

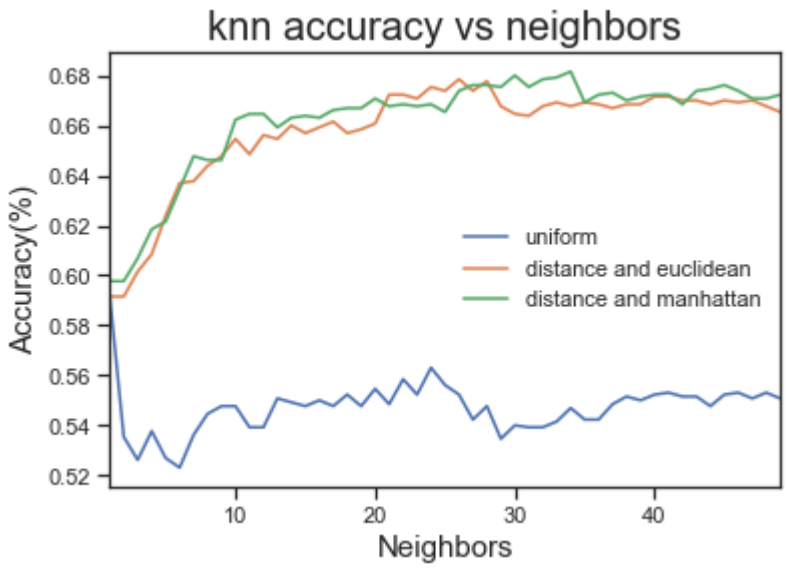
Out[131]:

	uniform	distance and euclidean	distance and manhattan
count	50.000000	50.000000	50.000000
mean	0.536169	0.644831	0.648877
std	0.078055	0.095380	0.095834
min	0.000000	0.000000	0.000000
25%	0.540385	0.655000	0.663077
50%	0.547692	0.667308	0.668462
75%	0.552115	0.670000	0.673654

	uniform	distance and euclidean	distance and manhattan
max	0.591538	0.678462	0.681538

In [132]:
```
graph = acc[1:].plot.line()
graph.set_title('knn accuracy vs neighbors', fontsize = 20)
graph.set_xlabel("Neighbors", fontsize = 15)
graph.set_ylabel("Accuracy(%)", fontsize = 15)
```

Out[132]:
```
Text(0, 0.5, 'Accuracy(%)')
```



1.1.5 Feature Extraction: Quality

1.1.5.1. PCA : Quality

Running KNN on normalized and PCA tranformed data with 5 principal components for wine Quality prediction

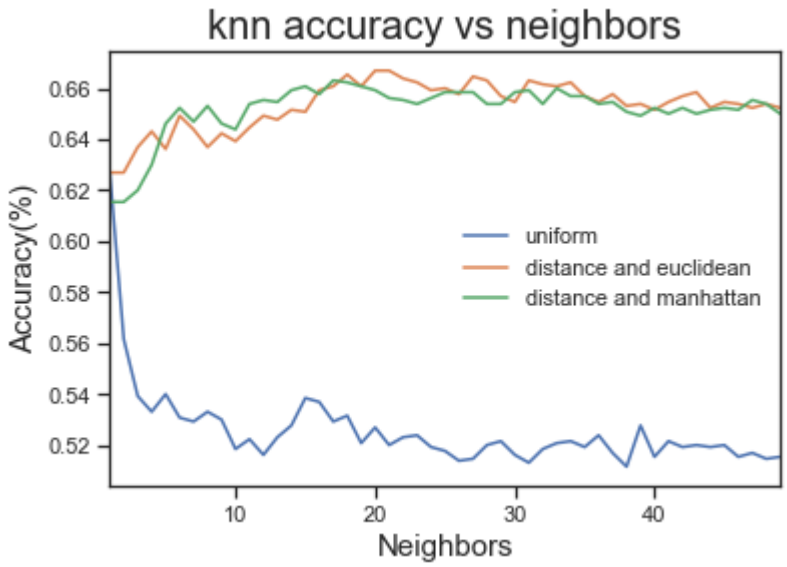
In [133]:
```
pca = PCA(n_components = 5,random_state =42)
principalComponents = pca.fit_transform(X_train_scaled_q)
col_name_pca = ["Principal Component " + str(i) for i in range(1,6)]
X_pca_train_q = pd.DataFrame(data = principalComponents, columns = col_name_pca)
y_pca_train_q = y_train_q
X_pca_test_q = pca.transform(X_test_scaled_q)
y_pca_test_q = y_test_q
n_neighborslist = list(range(1,50))
col_names=['uniform','distance and euclidean','distance and manhattan']
accarray = np.zeros((len(n_neighborslist),3))


#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights=col_names[0])
    neigh.fit(X_pca_train_q, y_pca_train_q)
    y_pca_pred_q = neigh.predict(X_pca_test_q)
    accscore = accuracy_score(y_pca_test_q, y_pca_pred_q)
    acc.at[k,col_names[0]] = accscore

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights= 'distance', m
    neigh.fit(X_pca_train_q, y_pca_train_q)
    y_pca_pred_q = neigh.predict(X_pca_test_q)
    accscore = accuracy_score(y_pca_test_q, y_pca_pred_q)
    acc.at[k,col_names[1]] = accscore

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights= 'distance', m
    neigh.fit(X_pca_train_q, y_pca_train_q)
    y_pca_pred_q = neigh.predict(X_pca_test_q)
    accscore = accuracy_score(y_pca_test_q, y_pca_pred_q)
    acc.at[k,col_names[2]] = accscore



acc.describe()
acc.head()
```

Out[133]:

	uniform	distance and euclidean	distance and manhattan
0	0.000000	0.000000	0.000000
1	0.626923	0.626923	0.615385
2	0.561538	0.626923	0.615385
3	0.539231	0.636923	0.620000
4	0.533077	0.643077	0.630000

In [134]:
```
graph = acc[1:].plot.line()
graph.set_title('knn accuracy vs neighbors', fontsize = 20)
graph.set_xlabel("Neighbors", fontsize = 15)
graph.set_ylabel("Accuracy(%)", fontsize = 15)
```

Out[134]:
```
Text(0, 0.5, 'Accuracy(%)')
```



1.1.5.2 LDA : Quality

1.1.5.2.1 Running KNN on normalized and LDA tranformed data with 5 principal components for wine Quality prediction

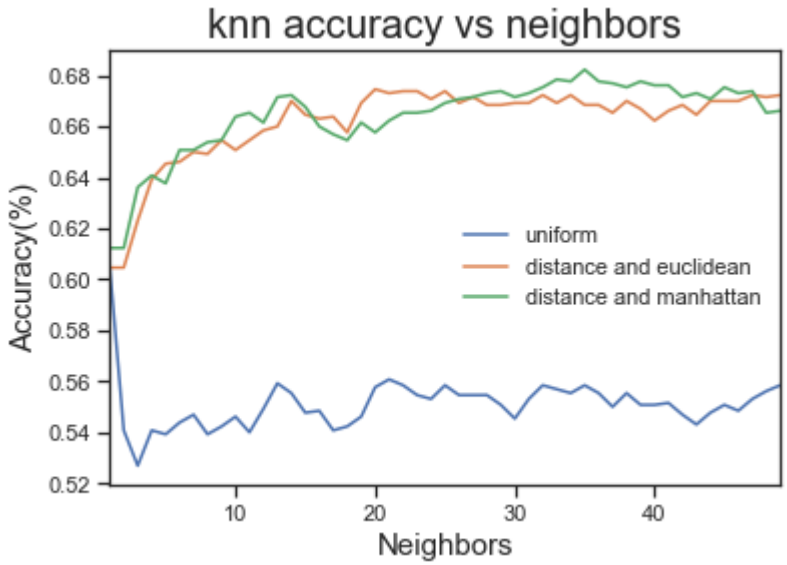
In [135]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
lda = LinearDiscriminantAnalysis()
col_name_lda = ["LDA component 1","LDA component 2","LDA component 3","LDA compor
lda = LinearDiscriminantAnalysis(n_components = 5)
ldaComponents = lda.fit_transform(X_train_scaled_q, y_train_q)
X_lda_train_q = pd.DataFrame(data = ldaComponents, columns = col_name_lda)
y_lda_train_q = y_train_q
y_lda_test_q = y_test_q
X_lda_test_q = lda.transform(X_test_scaled_q)
n_neighborslist = list(range(1,50))
col_names=['uniform','distance and euclidean','distance and manhattan']
accarray = np.zeros((len(n_neighborslist),3))


#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights=col_names[0])
    neigh.fit(X_lda_train_q, y_lda_train_q)
    y_lda_pred_q = neigh.predict(X_lda_test_q)
    accscore = accuracy_score(y_lda_test_q, y_lda_pred_q)
    acc.at[k,col_names[0]] = accscore

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights= 'distance', me
    neigh.fit(X_lda_train_q, y_lda_train_q)
    y_lda_pred_q = neigh.predict(X_lda_test_q)
    accscore = accuracy_score(y_lda_test_q, y_lda_pred_q)
    acc.at[k,col_names[1]] = accscore

for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights= 'distance', me
    neigh.fit(X_lda_train_q, y_lda_train_q)
    y_lda_pred_q = neigh.predict(X_lda_test_q)
    accscore = accuracy_score(y_lda_test_q, y_lda_pred_q)
    acc.at[k,col_names[2]] = accscore


acc.describe()
```

Out[135]:

	uniform	distance and euclidean	distance and manhattan
count	50.000000	50.000000	50.000000
mean	0.540077	0.648523	0.650954
std	0.078624	0.094883	0.095128
min	0.000000	0.000000	0.000000
25%	0.545577	0.657885	0.658269
50%	0.550769	0.668462	0.668462
75%	0.555385	0.670000	0.673654
max	0.604615	0.674615	0.682308

In [136]:
```
graph = acc[1:].plot.line()
graph.set_title('knn accuracy vs neighbors', fontsize = 20)
graph.set_xlabel("Neighbors", fontsize = 15)
graph.set_ylabel("Accuracy(%)", fontsize = 15)
```

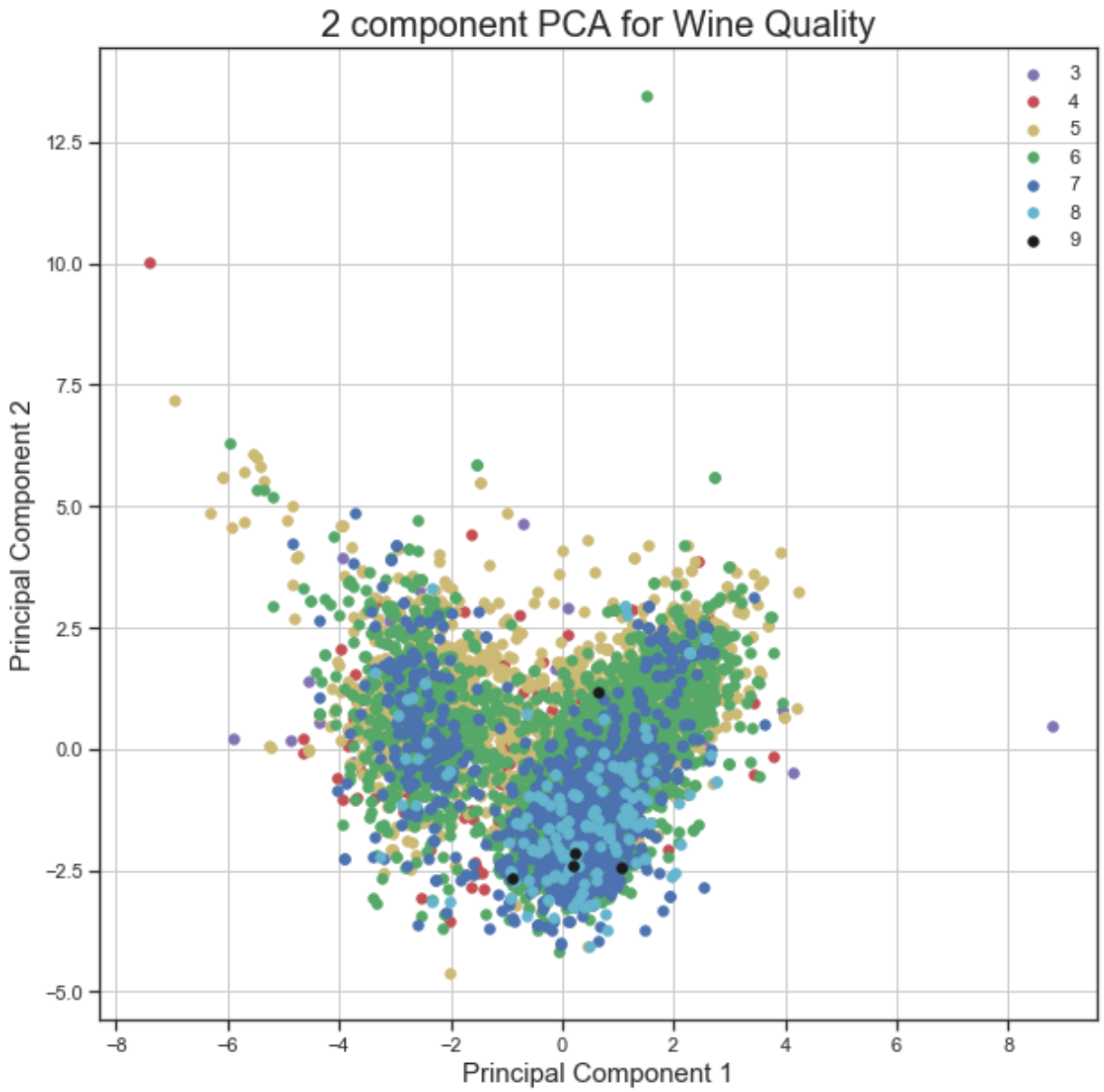
Out[136]:
```
Text(0, 0.5, 'Accuracy(%)')
```



Comparing the KNN classification accuracy on PCA and LDA features, maximum accuracy achieved using LDA features for quality target variable is 68.23% which is greater than the accuracy obtained using PCA features of 66.69% respectively. Thus can conclude that LDA works better compared to PCA in this case.

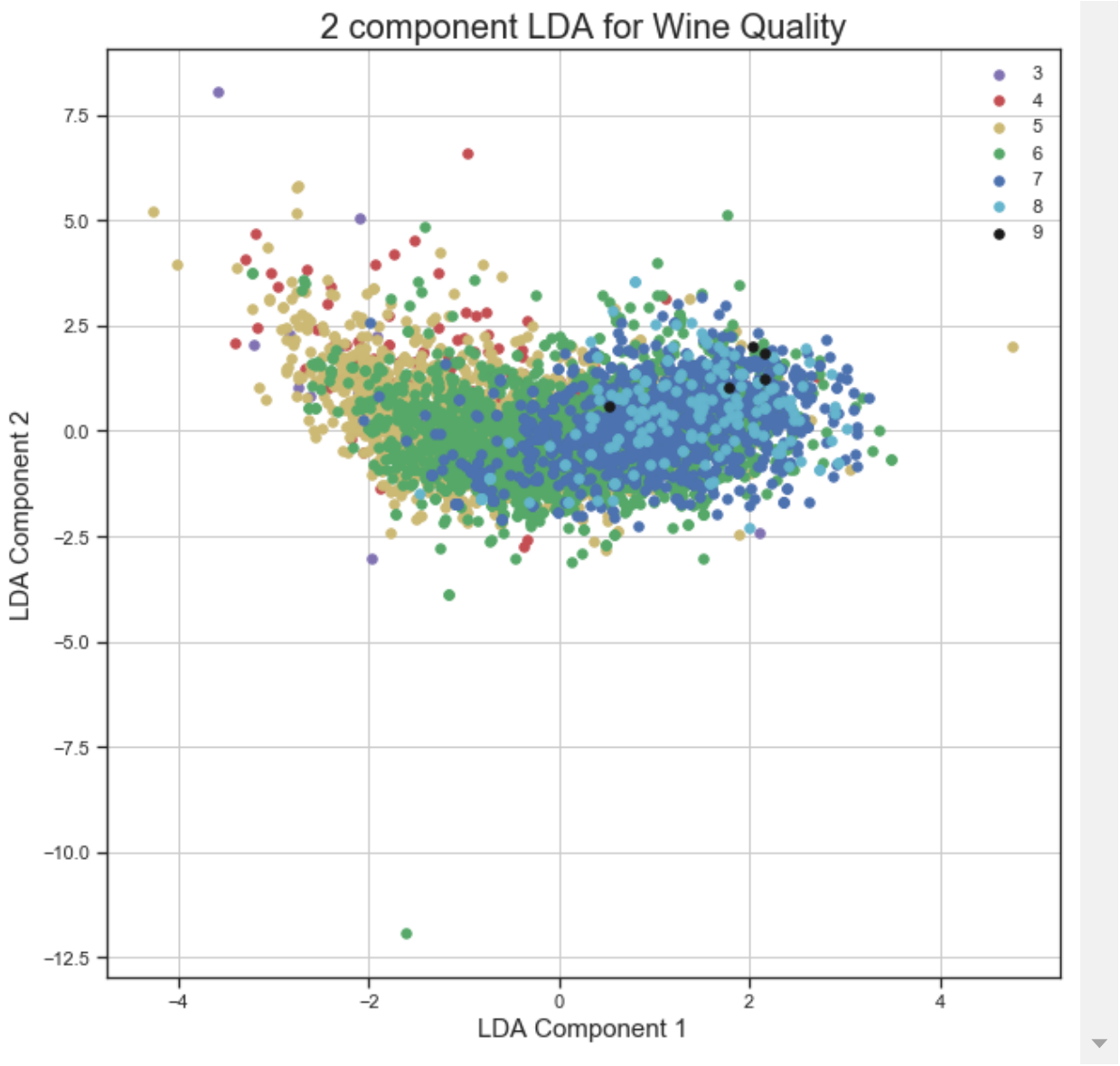
Visualization in 2D using PCA for different Wine Quality : 3,4,5,6,7,8,9

In [137]:
```
# further this data needs to standarized
#for data visulaization I will be using full data
X_full_q = wine[D]
from sklearn.preprocessing import StandardScaler
scalar = StandardScaler(copy = False)
X_full_scaled_q = X_full_q.copy()
scalar.fit_transform(X_full_scaled_q)
# lets fit and transform data to two principal components on PCA
pca = PCA(n_components = 2)
principalComponents = pca.fit_transform(X_full_scaled_q)
column_names = ["Principal component 1", "Principal component 2"]
PCA_2d_q = pd.DataFrame(data = principalComponents, columns = column_names)
yq = wine[L]
yq_new = yq.reset_index()
yq_new = yq_new.drop(['index'], axis =1)
final_PCA_2d_df = pd.DataFrame.join(self = PCA_2d_q, other = yq_new)

fig = plt.figure(figsize = (10,10))
plt.style.use('seaborn-ticks')
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_title('2 component PCA for Wine Quality', fontsize = 20)
wine_qualities = [3, 4,5,6,7,8,9]
colors = ['m','r','y','g','b','c','k']
for wine_quality, color in zip(wine_qualities,colors):
    indicesTokeep = final_PCA_2d_df['quality'] == wine_quality
    ax.scatter(final_PCA_2d_df.loc[indicesTokeep, 'Principal component 1']
               , final_PCA_2d_df.loc[indicesTokeep, 'Principal component 2']
               , c = color
               , s =30)
ax.legend(wine_qualities)
ax.grid()
```



visuliazation in 2D using LDA for Wine Quality

In [138]:
```
# I would be visualizing on the whole data
lda = LinearDiscriminantAnalysis(n_components = 2)
ldaComponents = lda.fit_transform(X_full_scaled_q, np.ravel(yq_new.to_numpy()))
lda_2d_q = pd.DataFrame(data = ldaComponents, columns= ['LDA component 1', 'LDA
lda_2d_final_q = pd.DataFrame.join(lda_2d_q, yq_new)
lda_2d_final_q.rename_axis('index')
lda_2d_final_q =lda_2d_final_q.assign(new_index = lambda z: z.index)
lda_2d_final_q.shape
fig = plt.figure(figsize = (10,10))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('LDA Component 1', fontsize = 15)
ax.set_ylabel('LDA Component 2', fontsize = 15)
ax.set_title('2 component LDA for Wine Quality', fontsize = 20)
wine_qualities = [3,4,5,6,7,8,9]
colors = ['m','r','y','g','b','c','k']
for wine_quality, color in zip(wine_qualities,colors):
    indicesTokeep = lda_2d_final_q['quality'] == wine_quality
    ax.scatter(lda_2d_final_q.loc[indicesTokeep, 'LDA component 1']
               , lda_2d_final_q.loc[indicesTokeep, 'LDA component 2']
               , c = color
               , s =30)
ax.legend(wine_qualities)
ax.grid()
```



1.1.6 Analysis and Discussion

1.1.6.1 K plots :

These plots are plotted above, please find these plots in the relevant parts of the notebook

1.1.6.2 Features :

From the correlation heatmap plotted for feature selection methods and by looking at the scatter plots, we can say that following features have higher effect on each other :

1. Free sulpher dioxide and total sulfur di oxide share have a correlation of .72 and share 51.84% of variance.
2. Density and alcohol have a correlation of .69 and share a variance of 47.61% of variance.
3. Residual sugar and density share a high correlation of .55 and hence share 30.25 % of variance.

4. Fixed acidity and density share a correlation of .46 and hence share a medium variance of 21.16 % of variance.

1.1.6.3 Selected Features:

1. The best accuracy achieved on wine color using KNN is 99.61% , where as best accuracy achieved on color using selected features as 'total sulpher dioxide', 'volatile acidity', 'chlorides' and 'sulphates' is 99.07%.
2. The best accuracy achieved on wine quality using KNN is 69.46% , whereas best accuracy achieved on quality using selected features as 'alcohol', 'density','volatile acidity' and 'chlorides' is 68.15%.
3. It can be seen that better accuracy is achievd using the full features instead of a selection of features.
4. The best performance of PCA(with 5 components) on wine color is achieved as 99.23%, whereas best performance of PCA(with 5 features) on wine quality as 66.69 %
5. Best performance of LDA (1 component) for wine color is achieved as 99.46% and best performance of LDA (5 component) for wine quality is achieved as 68.23%

To conclude, we can say that PCA(5 components) and LDA(1 component) performed better for wine color prediction compared to four features selected using the correlation among features and wine colors. Whereas, In the case of wine quality prediction, four selected features based model (using correlation among features and wine quality), performed better than PCA(5 components) based model, but could not perform better than LDA(5 components).

1.1.6.4 PCA vs. LDA:

1. From the above k-plots of KNN classification accuracy for different features from the dataset, PCA and LDA, it can be depicted that using PCA or LDA for n_components =5 does not result into better accuracy than the accuracy obtained using given dataset features.

2. Comparing the KNN classification accuracy on PCA and LDA features, maximum accuracy achieved using LDA features for color and quality target variables are 99.46% and 68.23% which is greater than the accuracy obtained using PCA features of 99.23% and 66.69% respectively. Thus can conclude that LDA works better compared to PCA in this case.

3. Normlization does affect the performance of PCA,In PCA we are interested in the components that maximize the variance. If one component (e.g. density) varies less than another (e.g. total sulfur dioxide) because of their respective scales (g/cm3 vs. ppm), PCA might determine that the direction of maximal variance more closely corresponds with the ‘density’ axis, if those features are not scaled, which would be incorrect. (Reference: https://scikit-learn.org/stable/auto_examples/preprocessing/plot_scaling_importance.html (https://scikit-learn.org/stable/auto_examples/preprocessing/plot_scaling_importance.html))

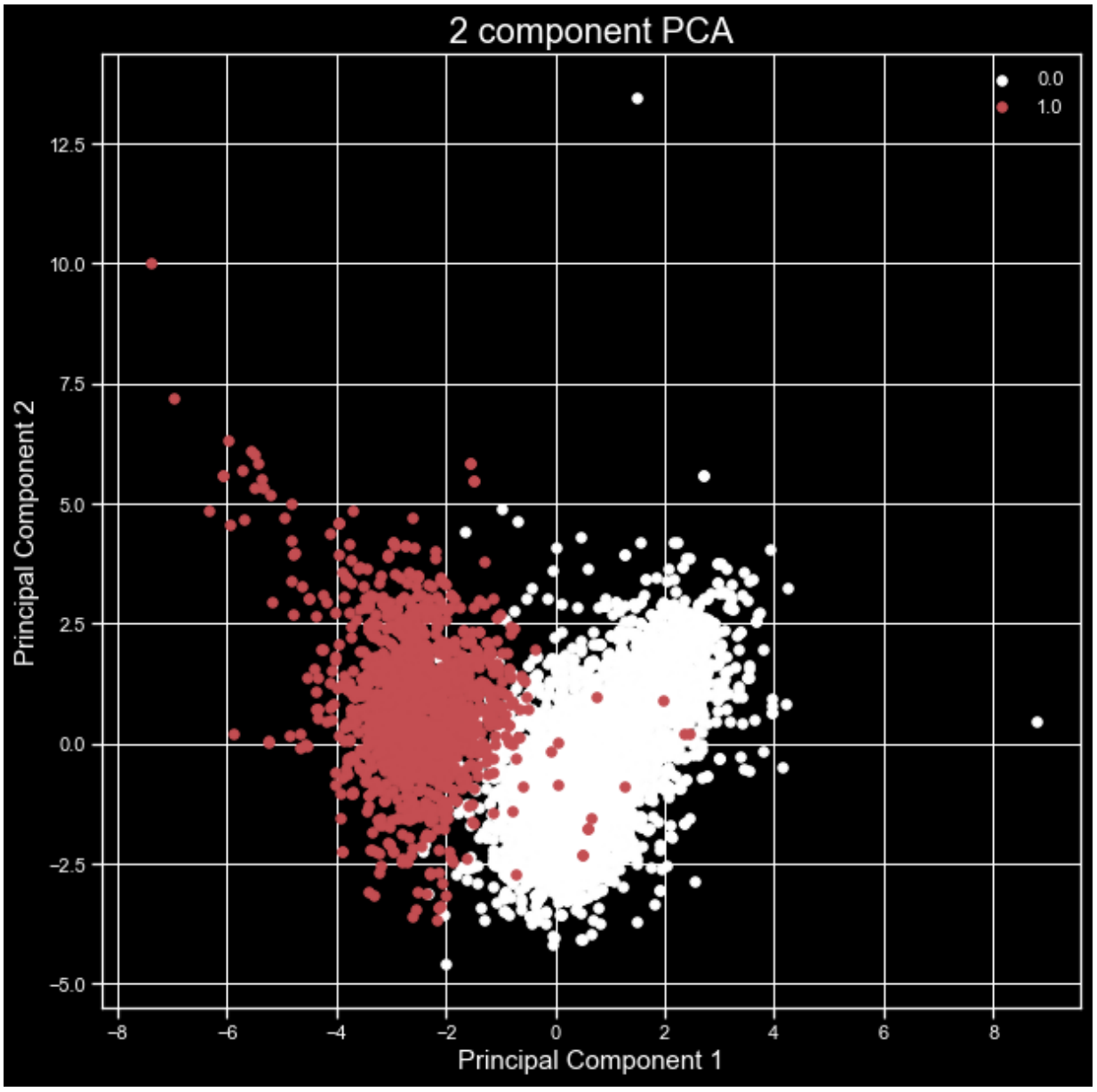
However, Normalization does not affect the LDA. LDA is the classification technique using target variable to separate variables in the lower dimension. It tries to minimise the variance within group and maximise the separation between the group which is not affected by the normalization.

(Reference: https://stats.stackexchange.com/questions/109071/standardizing-features-when-using-lda-as-a-pre-processing-step (https://stats.stackexchange.com/questions/109071/standardizing-features-when-using-lda-as-a-pre-processing-step))

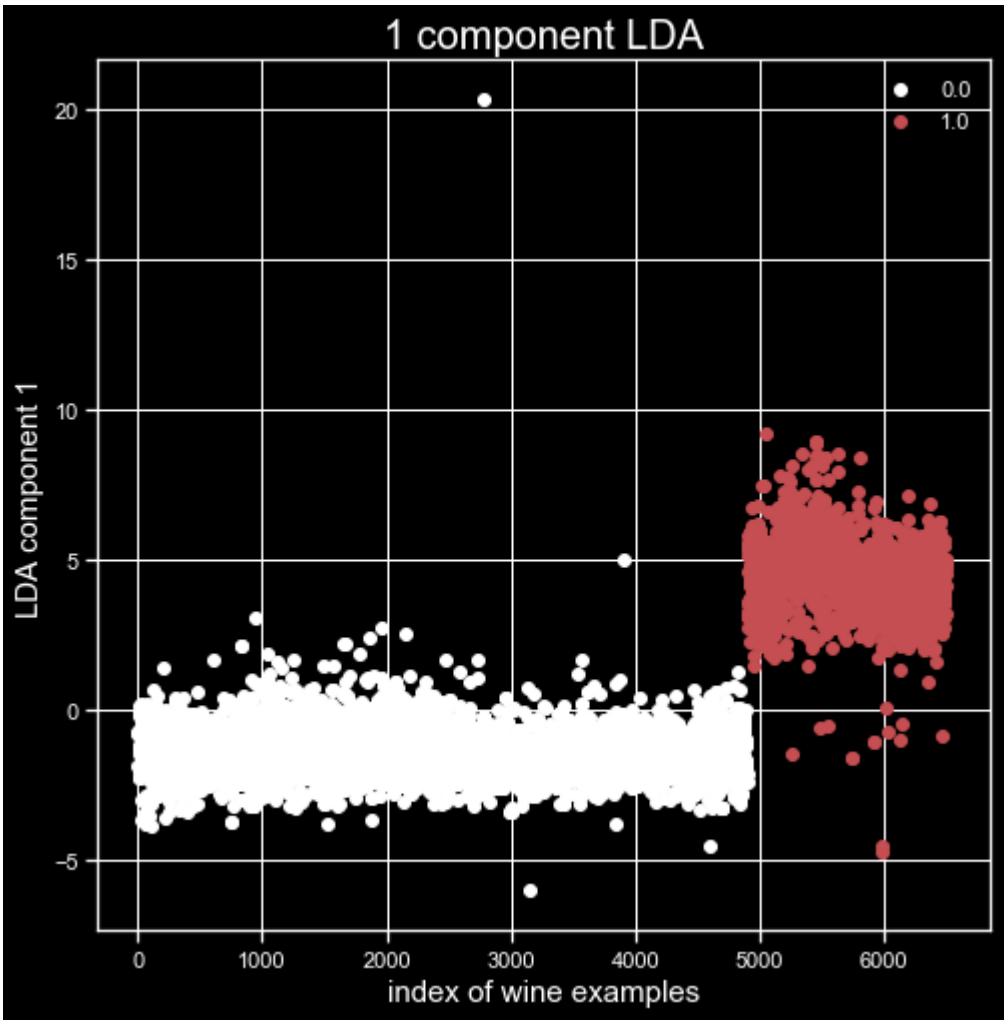
1.1.6.5 Plot

These plots are plotted above, please refer to relevant parts of code.

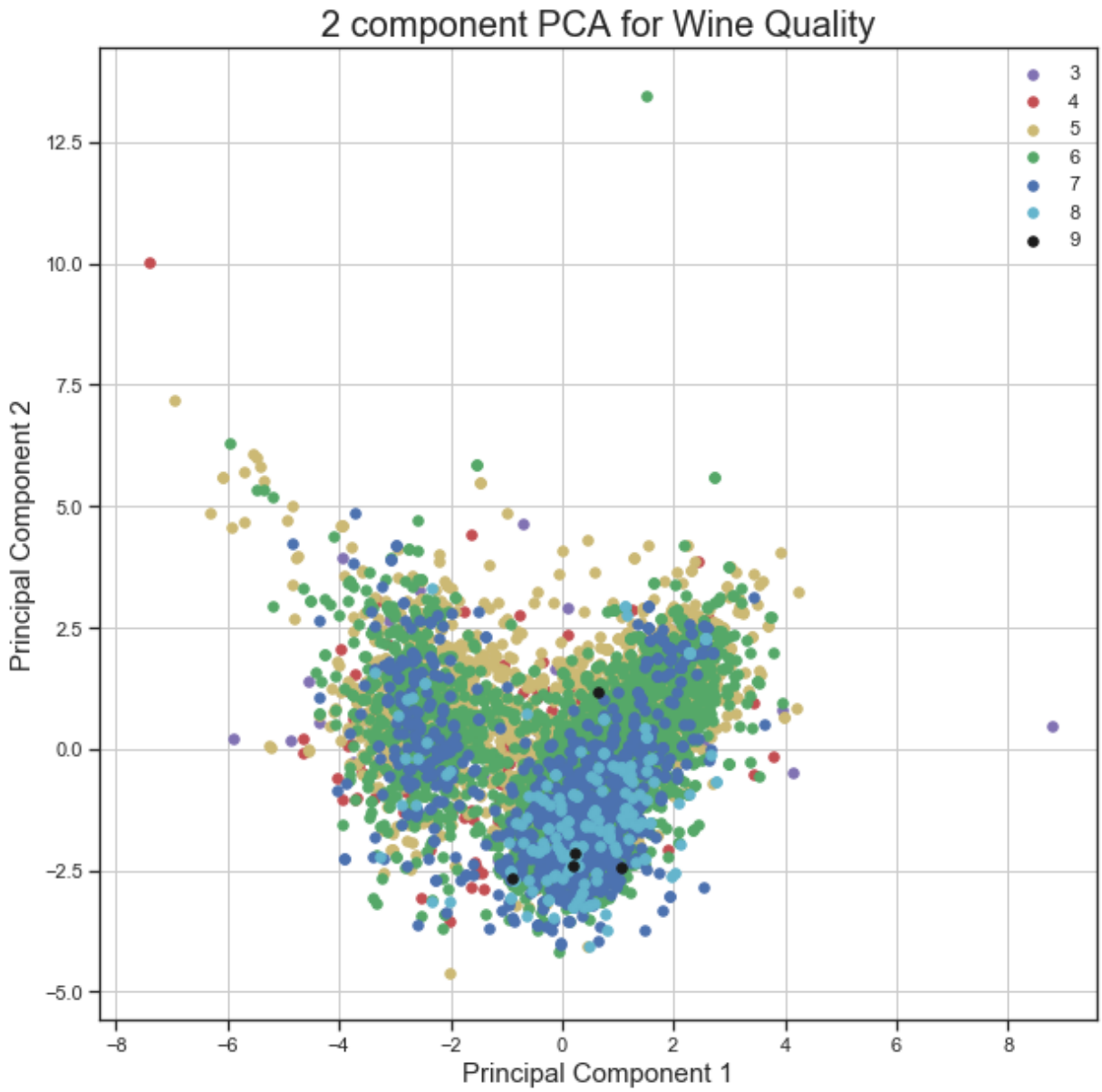
here 1 represents "red" and 0 represents "white" color for wine
Wine color projection using PCA in 2D



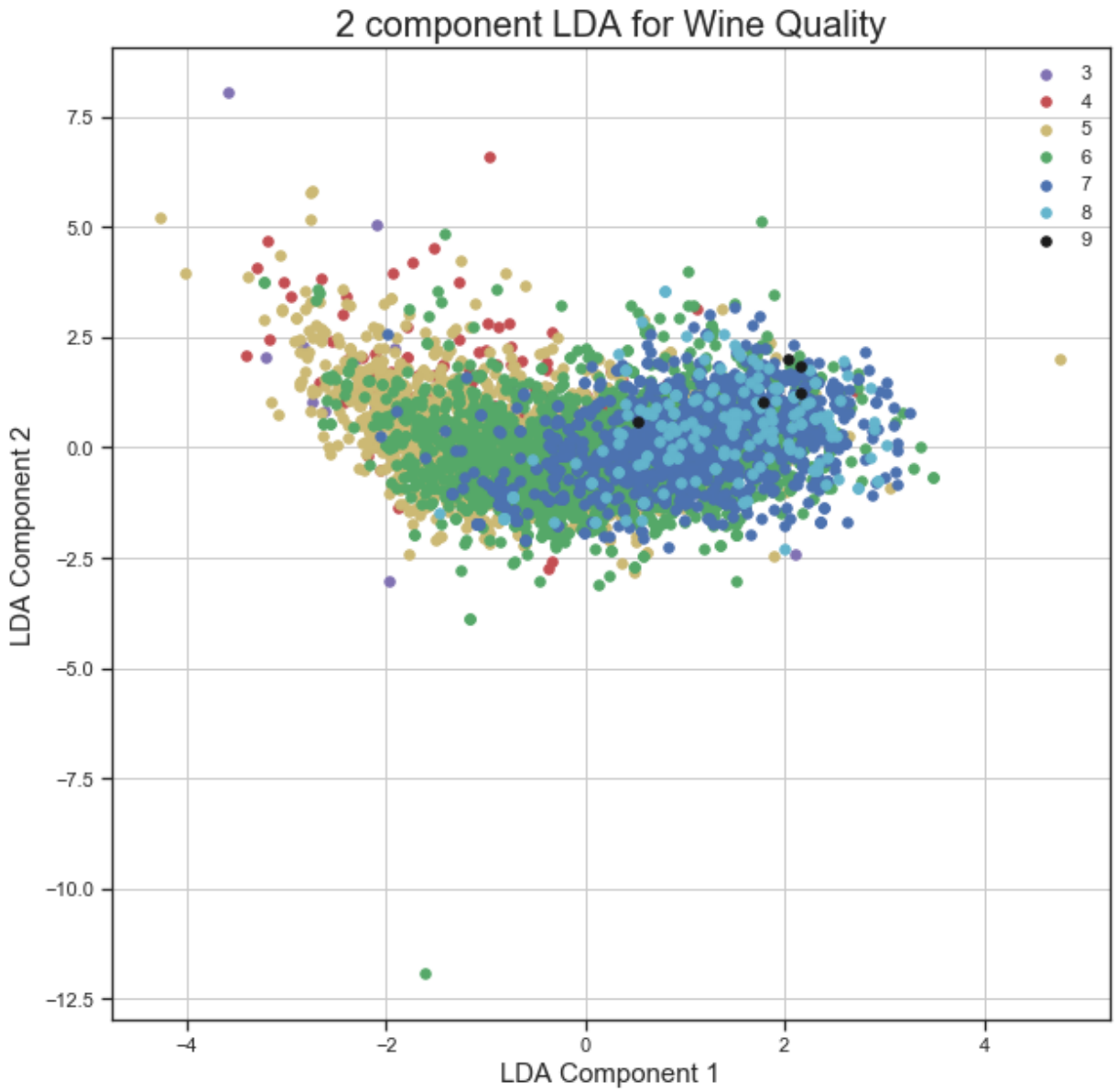
here 1 represents "red" and 0 represents "white" color for wine
Wine color projection using LDA in 1D



Wine quality projection using PCA in 2D



Wine quality rojection using LDA in 2D



By looking at the above plots and comparing these plots with pairplots plotted above, we have the following observation :

1. PCA and LDA plots are more informative compared to pairplots, and it is easier to comprehend data on the projected axis of PCA and LDAs.
2. In PCA, data is projected into principal axis which are orthogonal to each other (thus having a correlation = 0), whereas in pair plots the feature axis might share some correlation. This independence of principal components (among each other) helps in better visualization of data, as can be seen for wine color data projection on first two principal components.
3. The basis for LDA is to maximize separation among different classes while minimizing variance within all classes, thus it is expected from LDA to perform better for separating target variables. This can be seen from the plots above, The projection of wine quality data on LDA components gives good information about data separability compared to pair plots.

To conclude, it can be said, the projection of data on PCA and LDA components is better representative of separation of classes, as these components are formed using all the data features to maintain maximum data variance and to maximize separation between classes while

minimizing variance shared among all classes respectively. Thus a few components obtained using PCA and LDA(few plots using few components) might summarize the dataset well compared to same number of features used directly from dataset.

Q 2

2.1 Dataset

In [139]:

```
import numpy as np
import pandas as pd
import random
import seaborn as sns
sns.set(style="ticks", color_codes=True)
from sklearn import neighbors
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
DataB = pd.read_csv("DataB.csv", sep=',')
DataB_features = DataB.drop(['gnd'], axis =1)
DataB_features.columns
DataB_features.rename(columns={'Unnamed: 0': "indexed"}, inplace = True)
DataB_features.indexed = DataB_features.indexed -1
DataB_features.set_index('indexed', inplace = True)
DataB_features.shape
X = DataB_features
DataB_class = DataB.gnd
DataB_class = DataB_class.to_frame()
DataB_class.rename_axis('indexed', inplace = True)
DataB_class.rename(columns={'gnd': "class"}, inplace = True)
```

2.2 Principal Component Analysis

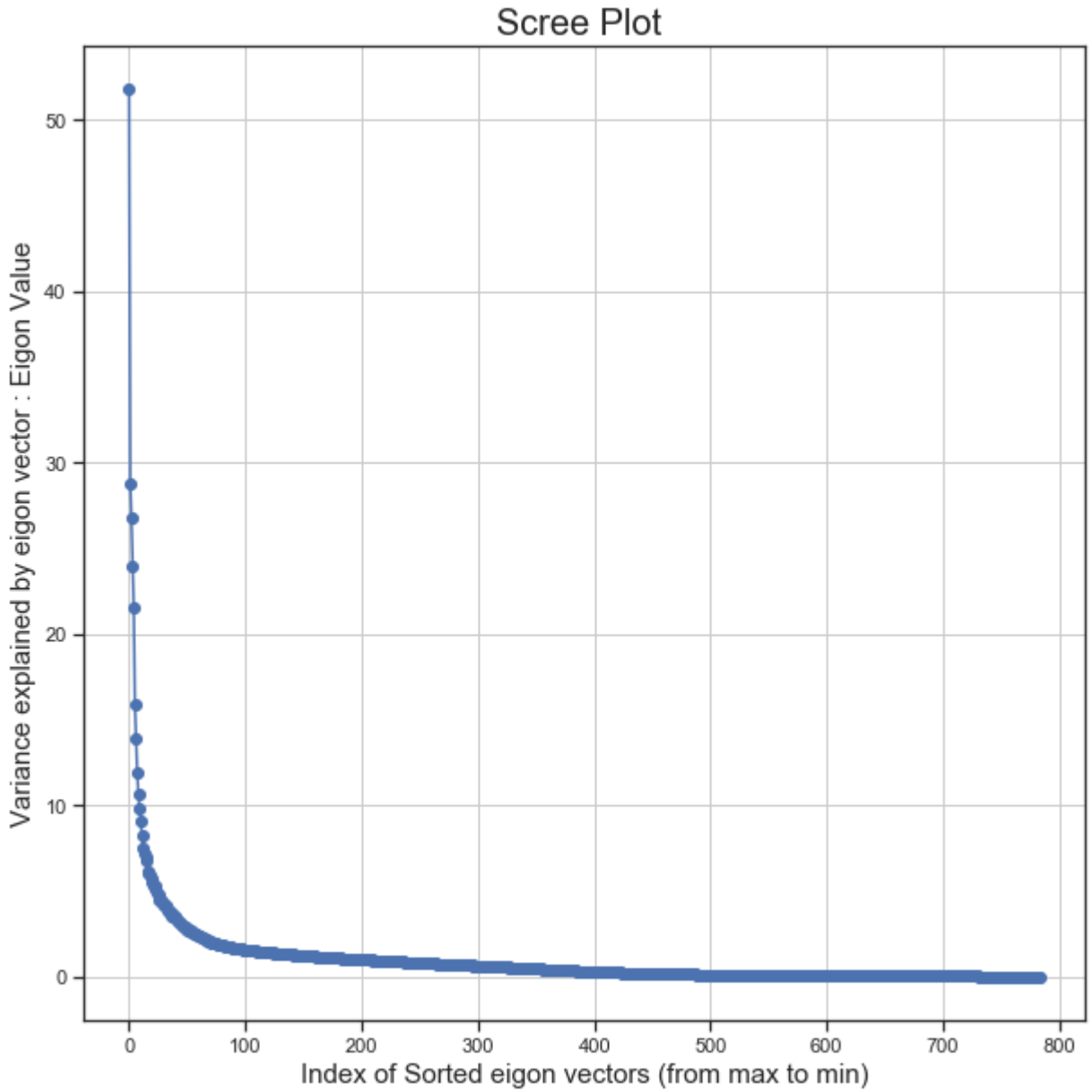
2.2.1 Practical Questions

1) In PCA, compute the eigenvectors and eigenvalues. Plot the scree plot and visually discuss which cut-off is good.

In [140]:
```
from sklearn.preprocessing import StandardScaler
Scaler = StandardScaler()
DataB_features_scaled = Scaler.fit_transform(DataB_features)
from sklearn.decomposition import PCA
pca = PCA(random_state =42)
X_pca = pca.fit_transform(DataB_features_scaled)
PCA_eigon_vector_data = pca.components_.T
col_names = ["Eigon Vector " + str(i) for i in range(1,785)]
PCA_eigon_vector_dataframe = pd.DataFrame(data = PCA_eigon_vector_data, columns=
eigon_values = pca.explained_variance_
print("eigon_values are : ", eigon_values)
print("eigon vectors are : ", PCA_eigon_vector_data)
```

```
1.80894894e+00 1.80037769e+00 1.79143455e+00 1.77821823e+00
 1.75344083e+00 1.74420633e+00 1.73235499e+00 1.71703221e+00
 1.69560958e+00 1.67660264e+00 1.67204995e+00 1.66413703e+00
 1.65298469e+00 1.64607810e+00 1.62658205e+00 1.61023500e+00
 1.59724824e+00 1.59323756e+00 1.58168294e+00 1.56824196e+00
 1.55793648e+00 1.55649266e+00 1.55126745e+00 1.53478544e+00
 1.52250847e+00 1.51844179e+00 1.50526693e+00 1.50403086e+00
 1.49832877e+00 1.49084155e+00 1.48107619e+00 1.47692460e+00
 1.47144049e+00 1.46645002e+00 1.44965194e+00 1.44504737e+00
 1.44001722e+00 1.42705114e+00 1.42253474e+00 1.41615253e+00
 1.40832241e+00 1.40224124e+00 1.39570127e+00 1.38879684e+00
 1.37904784e+00 1.37790604e+00 1.36580346e+00 1.36372072e+00
 1.35608476e+00 1.34791805e+00 1.33889606e+00 1.33011761e+00
 1.32803706e+00 1.32266875e+00 1.31925957e+00 1.30663444e+00
 1.30378173e+00 1.29900900e+00 1.29700677e+00 1.29230361e+00
 1.28317401e+00 1.27869566e+00 1.27308404e+00 1.26556014e+00
 1.25805205e+00 1.25428149e+00 1.25068531e+00 1.24549638e+00
 1.24082952e+00 1.23076321e+00 1.22902220e+00 1.22246312e+00
 1.21690294e+00 1.21022711e+00 1.20439342e+00 1.19694226e+00
 1.19615017e+00 1.18957391e+00 1.18452868e+00 1.17925785e+00
```

In [141]:
```
# lets plot scree plot for all components
eigon_value_df = pd.DataFrame(data = eigon_values, columns = ['Eigon Values'])
#plt.plot(eigon_value_df)
fig = plt.figure(figsize =(10,10))
ax = fig.add_subplot(1,1,1)
ax.plot(eigon_value_df, marker ='o')
ax.set_ylabel('Variance explained by eigon vector : Eigon Value', fontsize = 15)
ax.set_xlabel('Index of Sorted eigon vectors (from max to min)', fontsize = 15)
ax.set_title('Scree Plot ', fontsize = 20)
ax.grid()
```



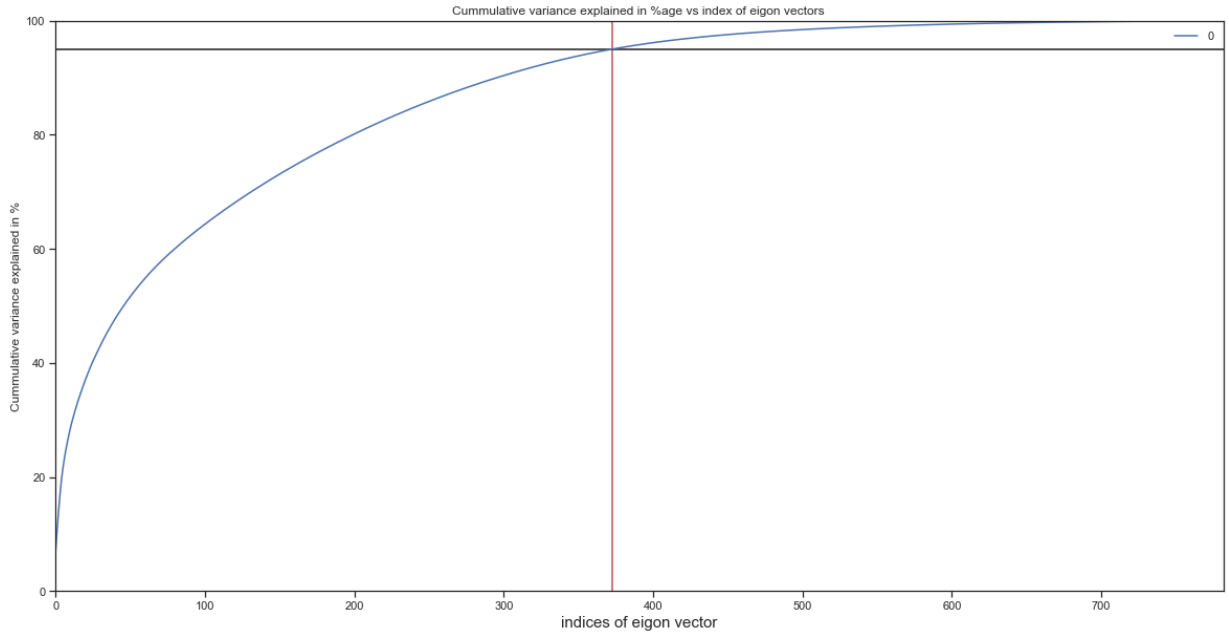
Deciding how to choose cut off point

Looking at the above scree plot of the principal components and their corresponding eigen values, it can be observed that with increase in the index of the principal components their corresponding eigen value decreases. Moreover, decrease in the eigen values are very steep for the initial principal components while after certain value they seem to have approximately equal value which can be observed at approximately 370th principal component.

Moreover, similar threshold can also be obtained by plotting the cumulative variance explained by the principal component. Looking at the below plotted graph it can be depicted that the 95% variance of the total data can be explained using the first 373 principal components.

In [142]:
```
#plotting cummulative % variance plot
L = pca.explained_variance_ratio_.tolist()
for i in range(0,len(L)-1):
    L[i] = L[i]*100
Cum_L= []
Cum_L.append(L[0])
sum1 = L[0]
for i in range(1, len(L)-1):
    sum1 = sum1 + L[i]
    Cum_L.append(sum1)

for k in range(0, len(Cum_L)):
    if Cum_L[k] >= 95:
        break


Cum_L = pd.DataFrame(data=Cum_L)
ax = Cum_L.plot(figsize = (20,10), ylim =(0,100), title = "Cummulative variance e
ax.set_xlabel('indices of eigon vector', fontsize = 15)
ax.set_ylabel('Cummulative variance explained in %')
ax.vlines(x = k,ymin = 0, ymax =100 , color = 'r')
ax.hlines( y= 95, xmin =0, xmax =785, color = 'k' )
print("number of eigon vectors that explain 95 % variance ",k)
```

```
number of eigon vectors that explain 95 % variance  373
```

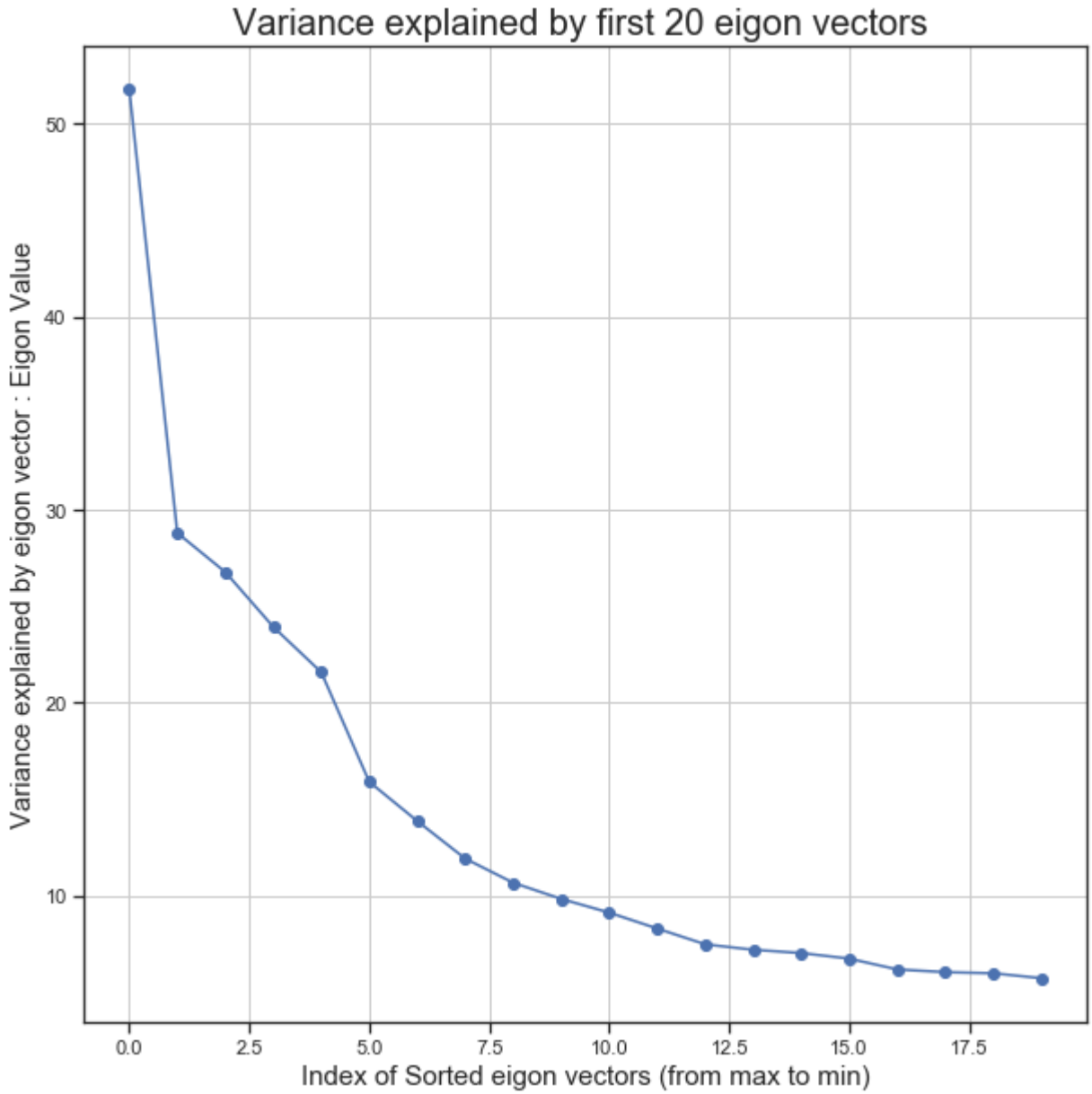


PCA results for first 20 eigon vectors :

$\Longrightarrow$ Since its difficult to look at elbows in the scree plot plotted for 784 principal components, lets plot scree plot for first 20 principal components which can be used for the analysis below

In [143]:
```
# lets scale the features
from sklearn.preprocessing import StandardScaler
Scaler = StandardScaler()
DataB_features_scaled = Scaler.fit_transform(DataB_features)
# so we have data of features now
X1 = X
X = DataB_features_scaled
# lets do PCA on X
from sklearn.decomposition import PCA
pca = PCA(n_components= 20,random_state =42)
X_pca = pca.fit_transform(X)
PCA_eigon_vector_data = pca.components_.T
col_names = ["Eigon Vector " + str(i) for i in range(1,21)]
PCA_eigon_vector_dataframe = pd.DataFrame(data = PCA_eigon_vector_data, columns=
eigon_values = pca.explained_variance_
```

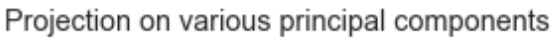
In [144]:
```
# lets plot scree plot for 20 components
eigon_value_df = pd.DataFrame(data = eigon_values, columns = ['Eigon Values'])
#plt.plot(eigon_value_df)
fig = plt.figure(figsize =(10,10))
ax = fig.add_subplot(1,1,1)
ax.plot(eigon_value_df, marker ='o')
ax.set_ylabel('Variance explained by eigon vector : Eigon Value', fontsize = 15)
ax.set_xlabel('Index of Sorted eigon vectors (from max to min)', fontsize = 15)
ax.set_title('Variance explained by first 20 eigon vectors ', fontsize = 20)
ax.grid()
```

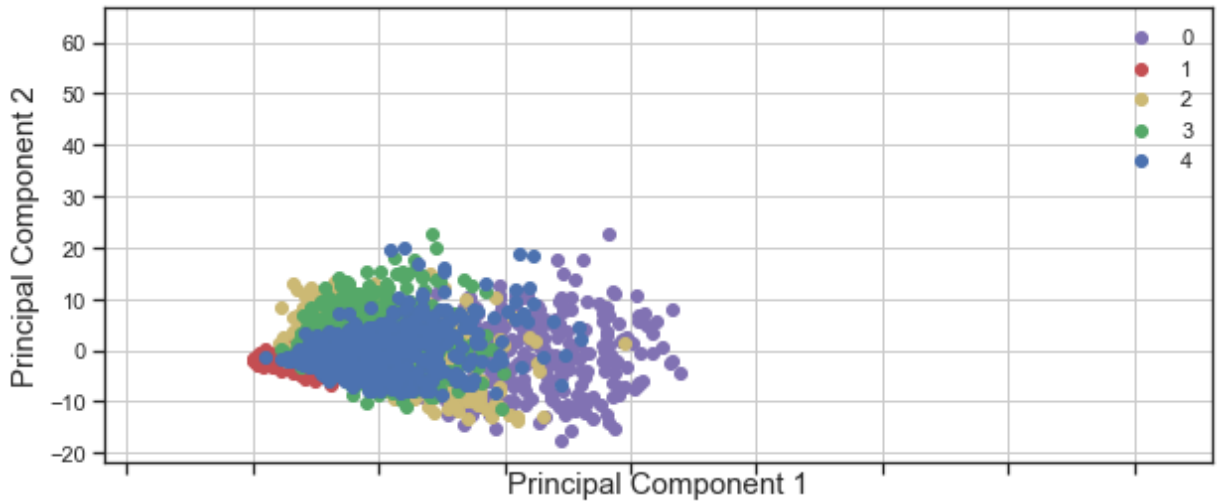


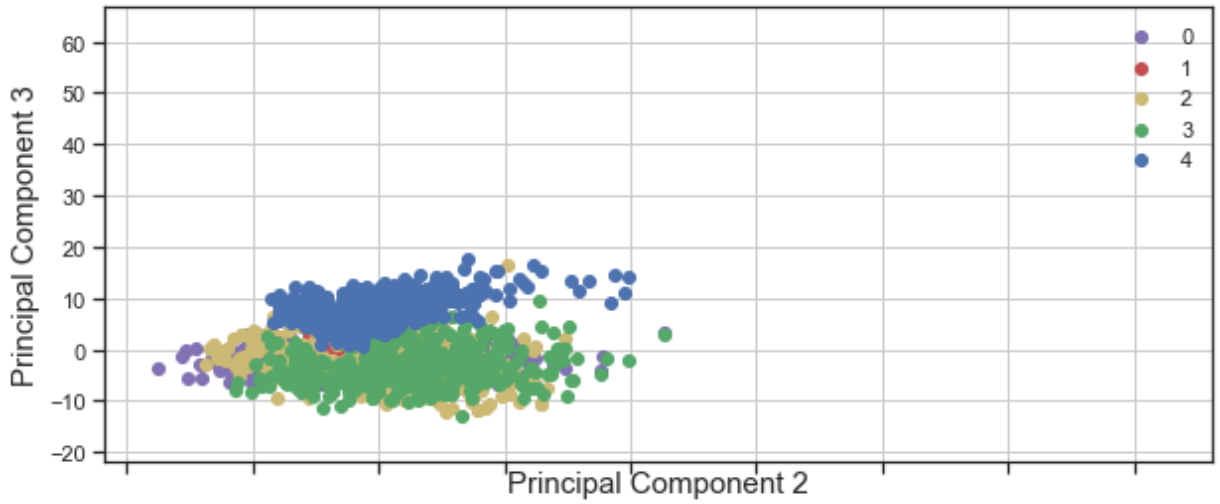
2) Using subplot in python matplotlib, plot the scatter plot of the projected data with the top 20 eigenvalues

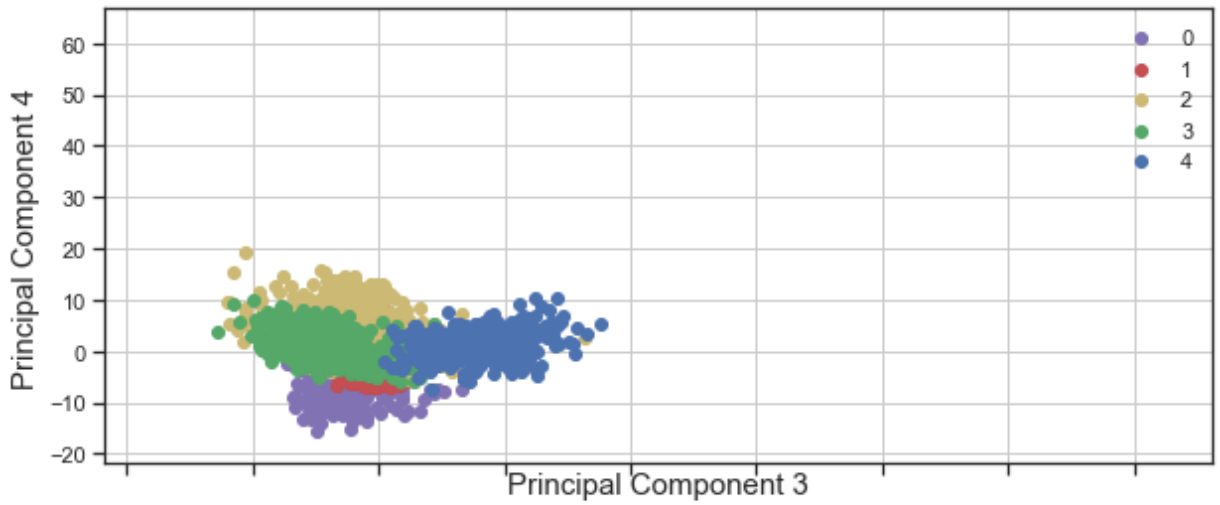
In [145]:
```
col_namess = ["Principal Component " + str(i) for i in range(1,21)]
X_pca_dataframe = pd.DataFrame(data = X_pca, columns= col_namess)
total_PCA_df_class = pd.DataFrame.join(X_pca_dataframe,DataB_class)
total_PCA_df_class.rename_axis('index')
total_PCA_df_class =total_PCA_df_class.assign(new_index = lambda z: z.index)
```

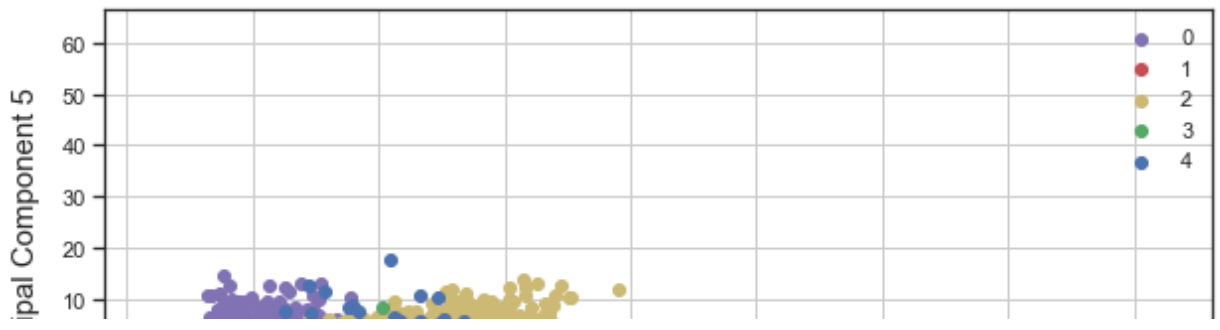
In [146]:
```
fig,axw = plt.subplots(20, sharex=True, sharey=True, figsize =(10,100))
fig.suptitle('Projection on various principal components')
for i in range(0,19):
    axw[i].set_ylabel('Principal Component '+ str(i+2), fontsize = 15)
    axw[i].set_xlabel('Principal Component '+ str(i+1), fontsize = 15)
    class_colors = [0,1,2,3,4]
    colors = ['m','r','y','g','b']
    for class_color, color in zip(class_colors,colors):
        indicesTokeep = total_PCA_df_class['class'] == class_color
        axw[i].scatter(x = total_PCA_df_class.loc[indicesTokeep,'Principal Compo
                       , c = color)
        axw[i].legend(class_colors)
        axw[i].grid()
```



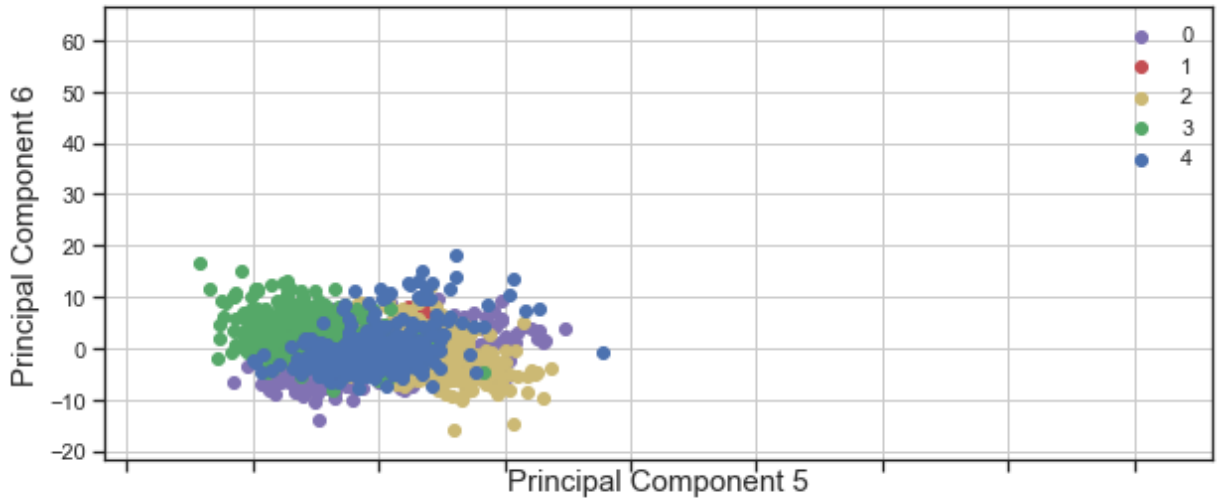

Principal Component 2
Principal Component 1
0
1
2
3
4

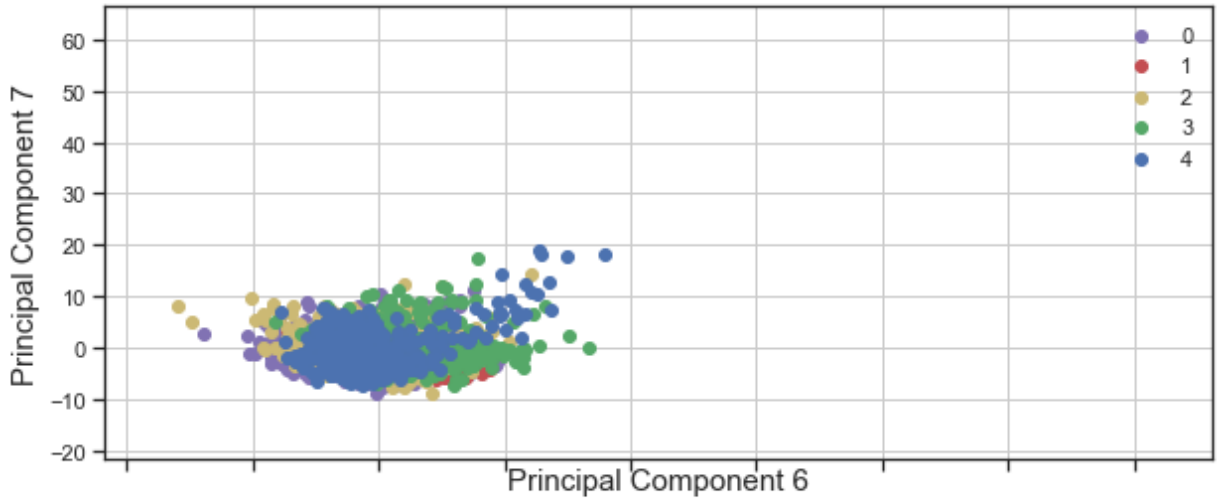

Principal Component 3
Principal Component 2
0
1
2
3
4

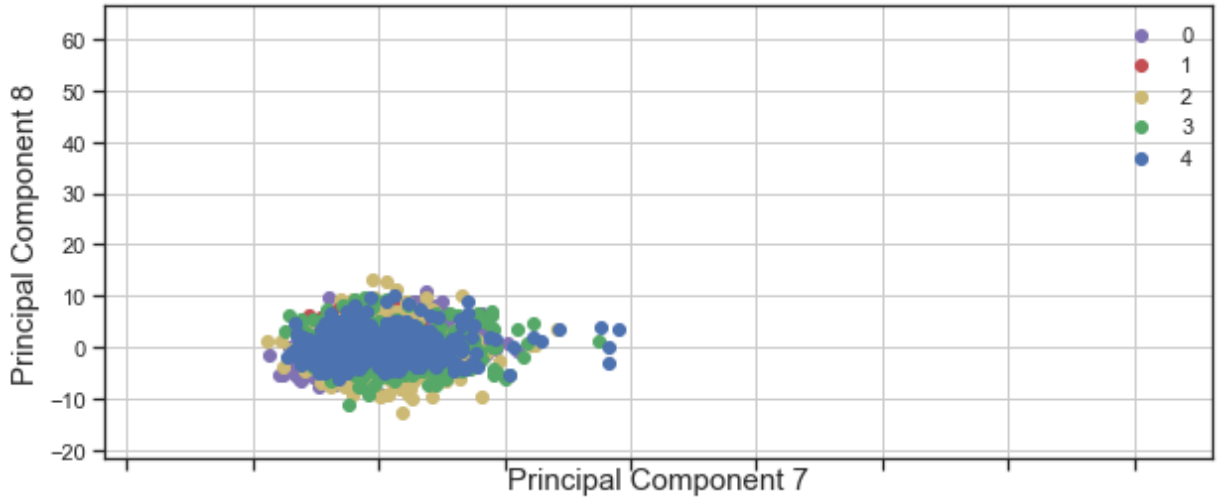

Principal Component 4
Principal Component 3
0
1
2
3
4

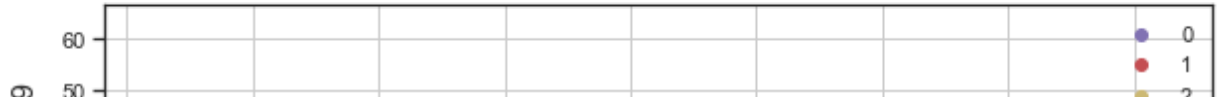

0
1
2
3
4

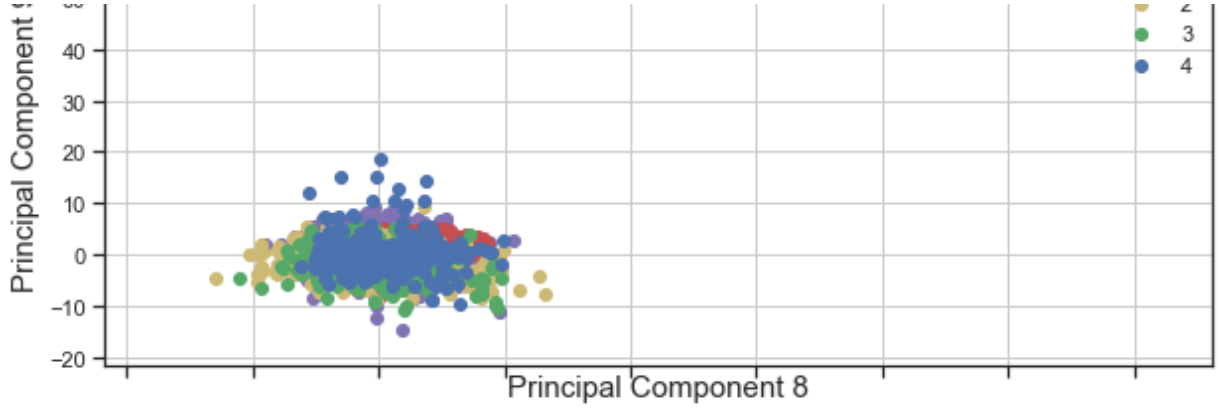

Princi
0
−10
−20
Principal Component 4

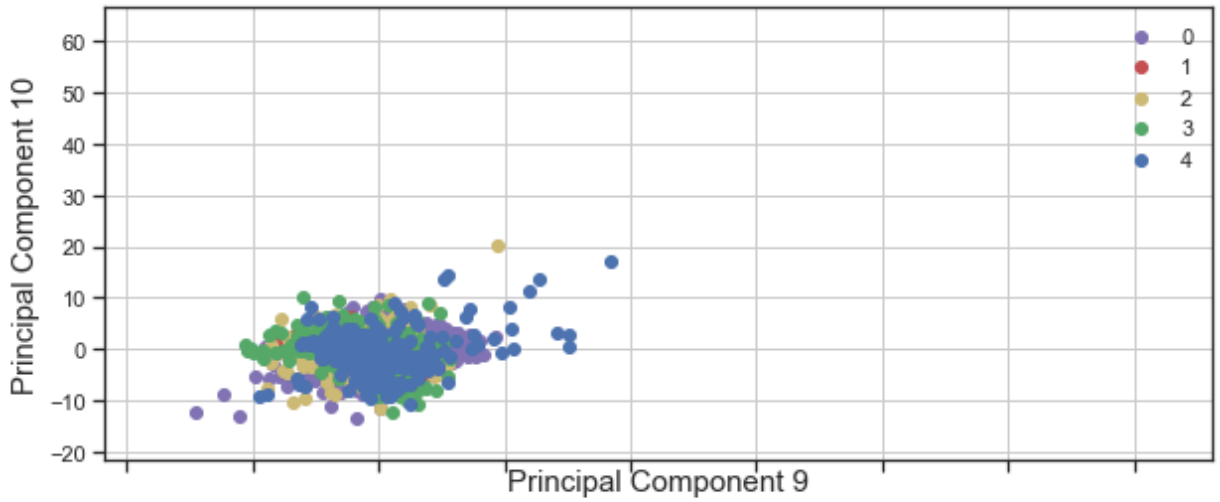

Principal Component 6
60
50
40
30
20
10
0
−10
−20
Principal Component 5
0
1
2
3
4

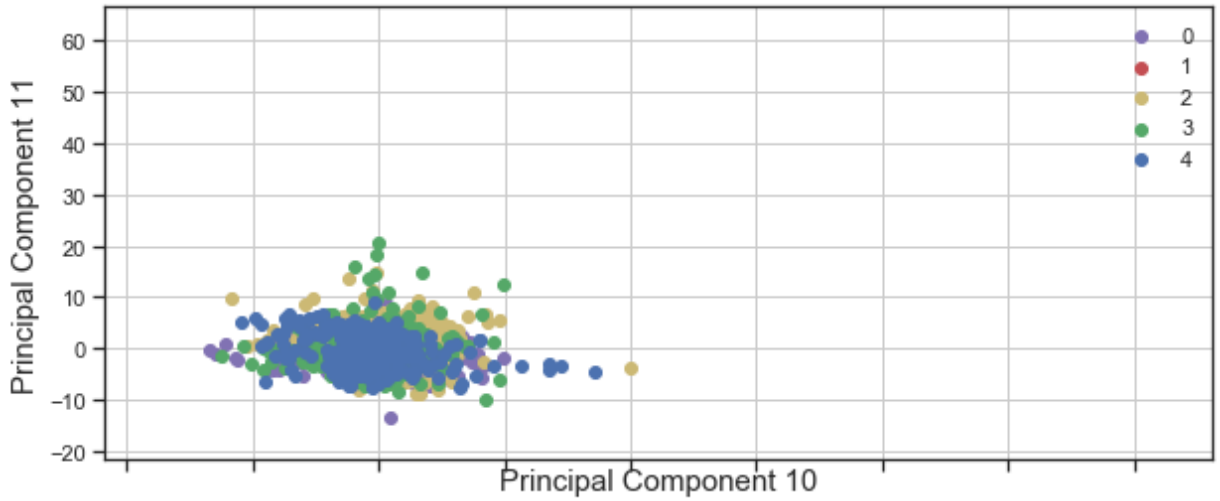

Principal Component 7
60
50
40
30
20
10
0
−10
−20
Principal Component 6
0
1
2
3
4

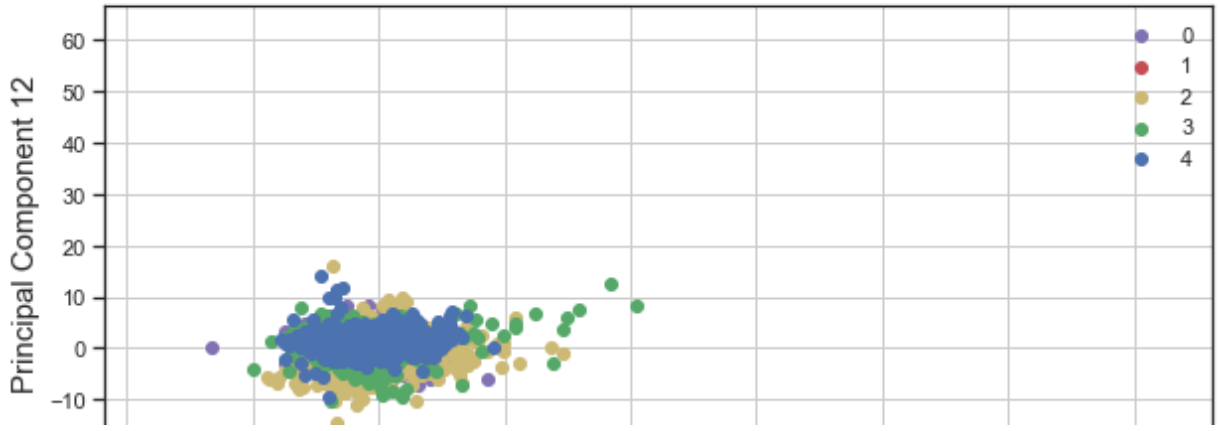

Principal Component 8
60
50
40
30
20
10
0
−10
−20
Principal Component 7
0
1
2
3
4


9
60
50
0
1

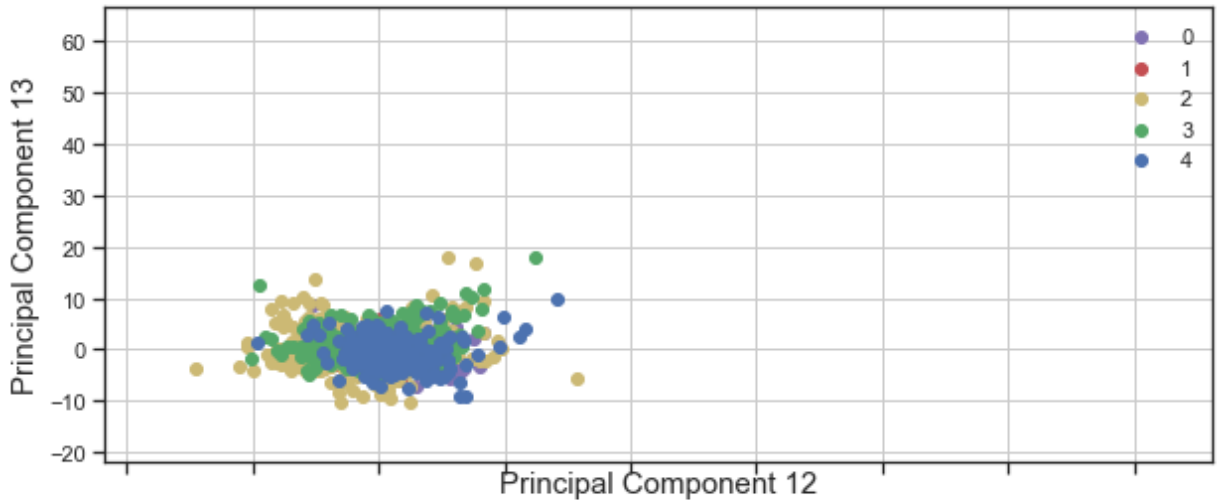

Principal Component
Principal Component 8
2
3
4

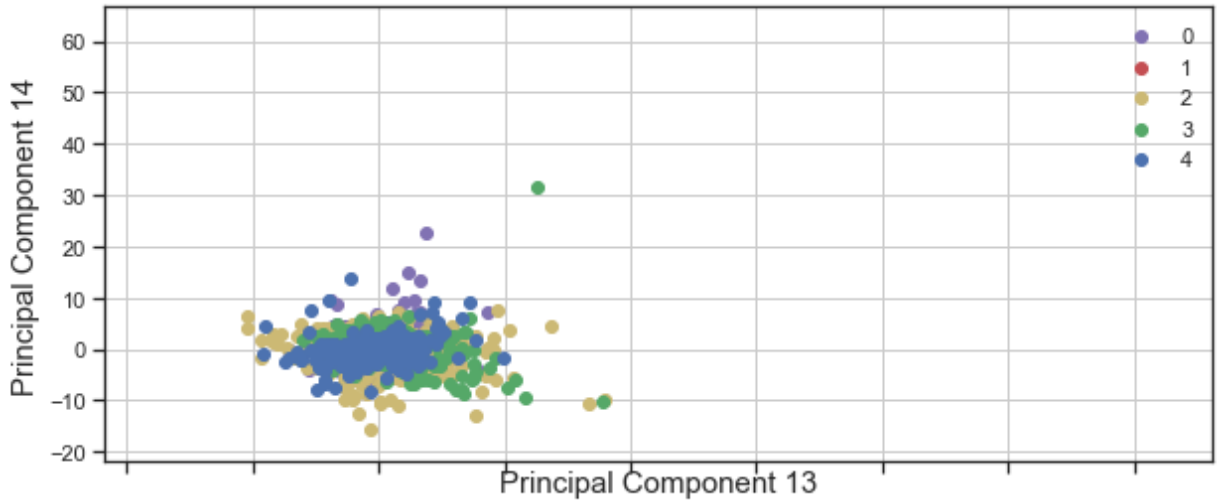

Principal Component 10
Principal Component 9
0
1
2
3
4

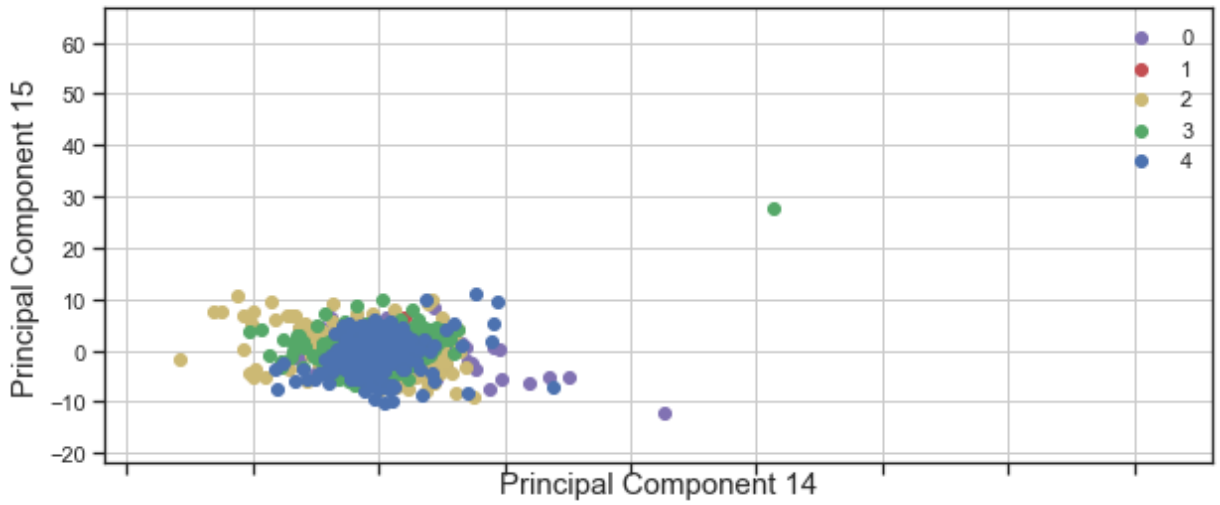

Principal Component 11
Principal Component 10
0
1
2
3
4

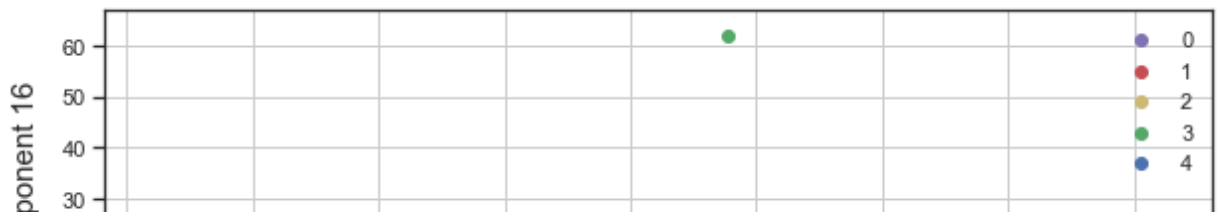

Principal Component 12
0
1
2
3
4

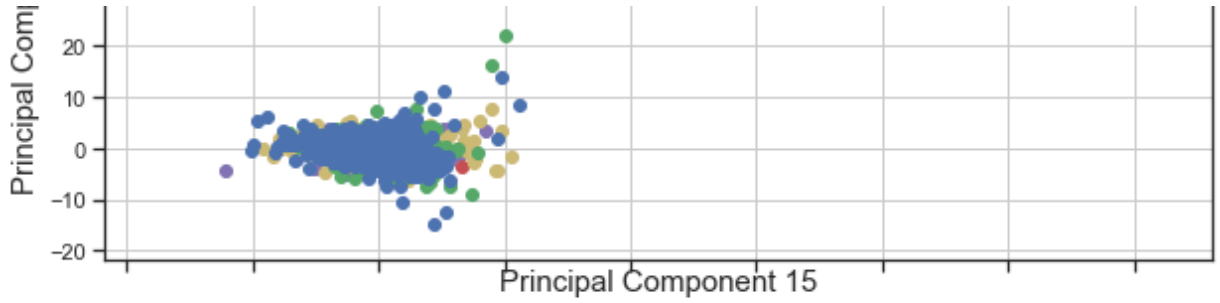

−20
Principal Component 11

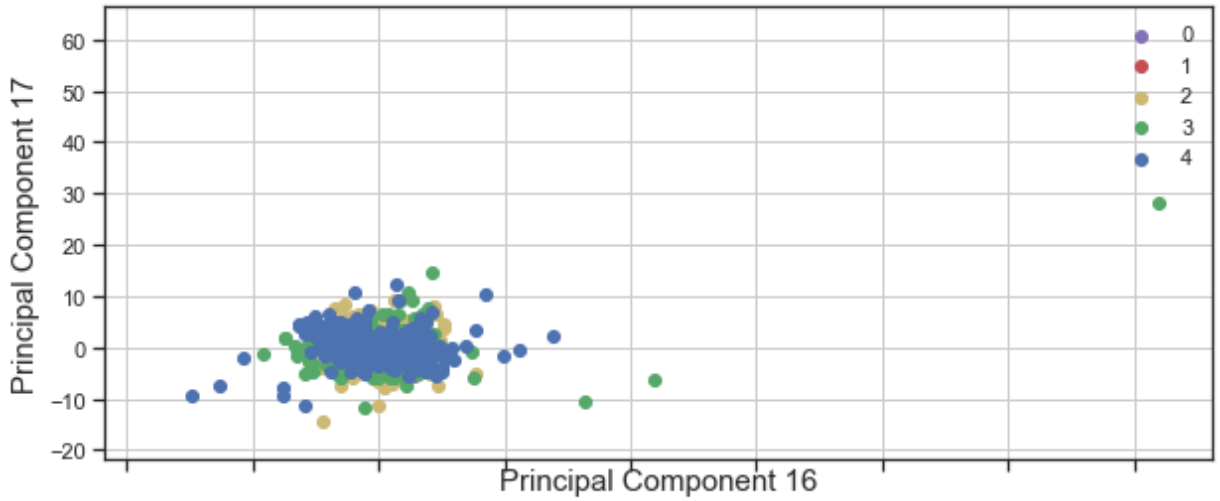

Principal Component 13
60
50
40
30
20
10
0
−10
−20
Principal Component 12
0
1
2
3
4

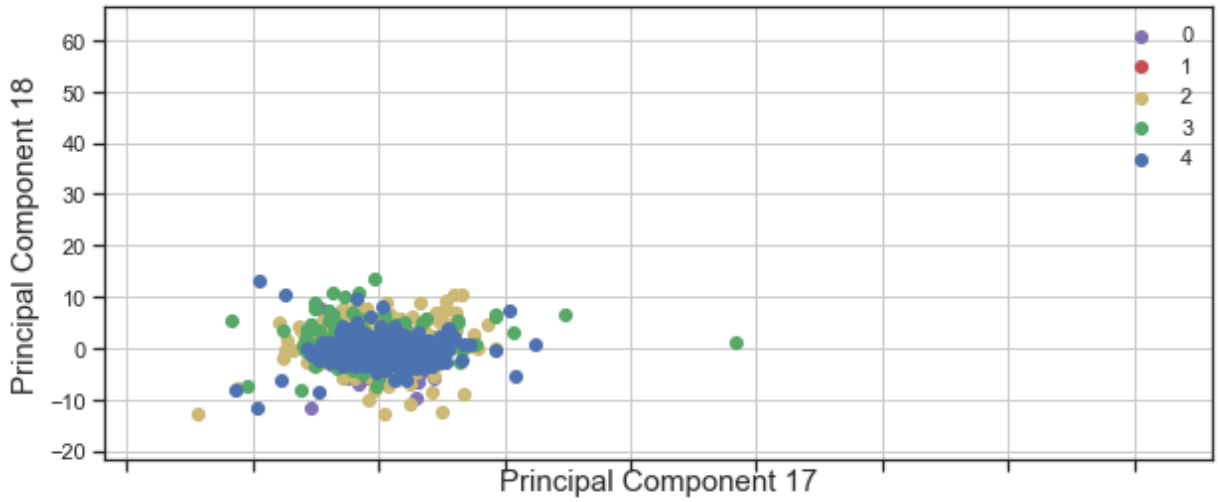

Principal Component 14
60
50
40
30
20
10
0
−10
−20
Principal Component 13
0
1
2
3
4

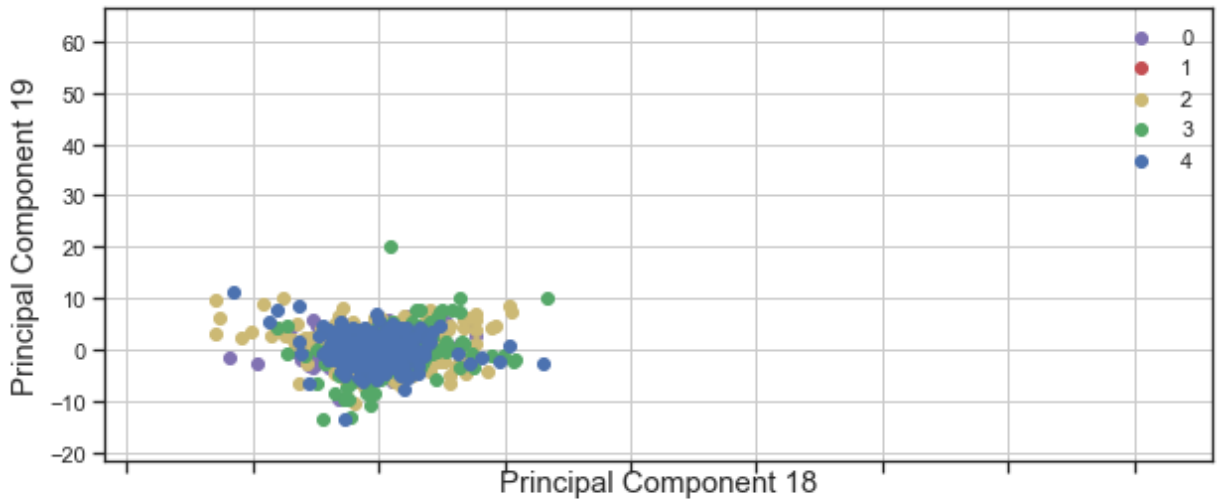

Principal Component 15
60
50
40
30
20
10
0
−10
−20
Principal Component 14
0
1
2
3
4

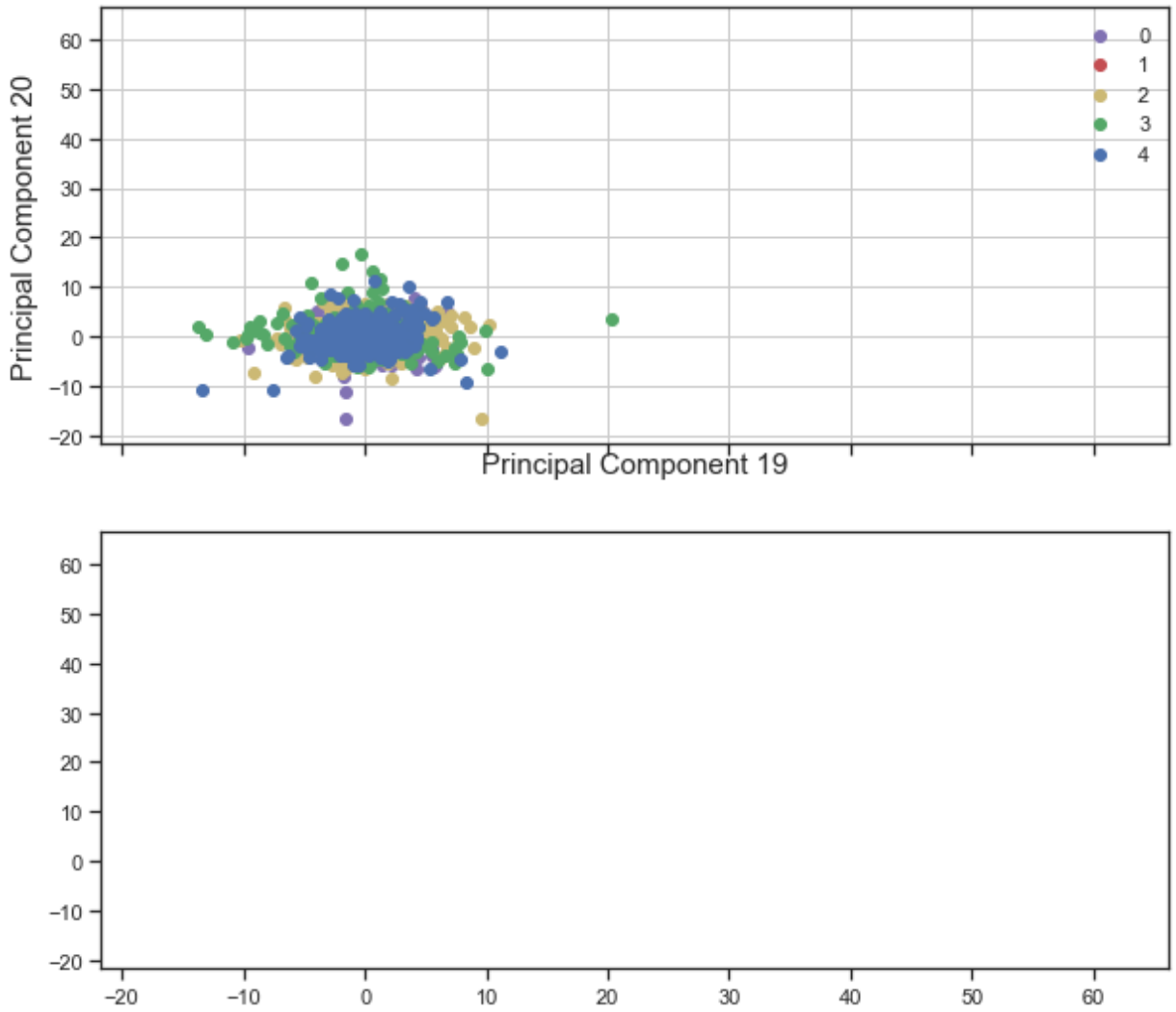

ponent 16
60
50
40
30
0
1
2
3
4


Principal Component 15


Principal Component 17
Principal Component 16
0
1
2
3
4


Principal Component 18
Principal Component 17
0
1
2
3
4


Principal Component 19
Principal Component 18
0
1
2
3
4

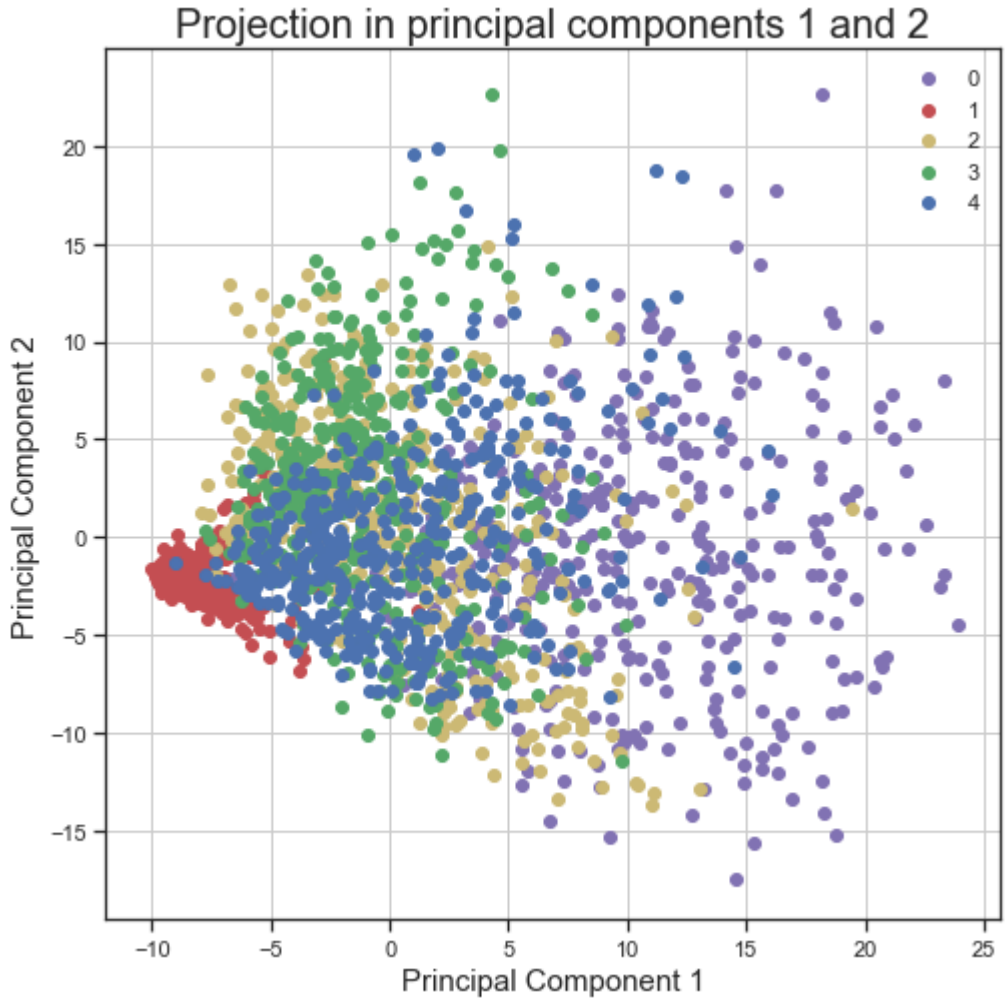


The above plots show the scatter plot of the transformed data on the top 20 principal components corresponding, it can be observed from the range of the variance(range of axis) described by each principal component that variance decreases with an increase in the number of principal components.

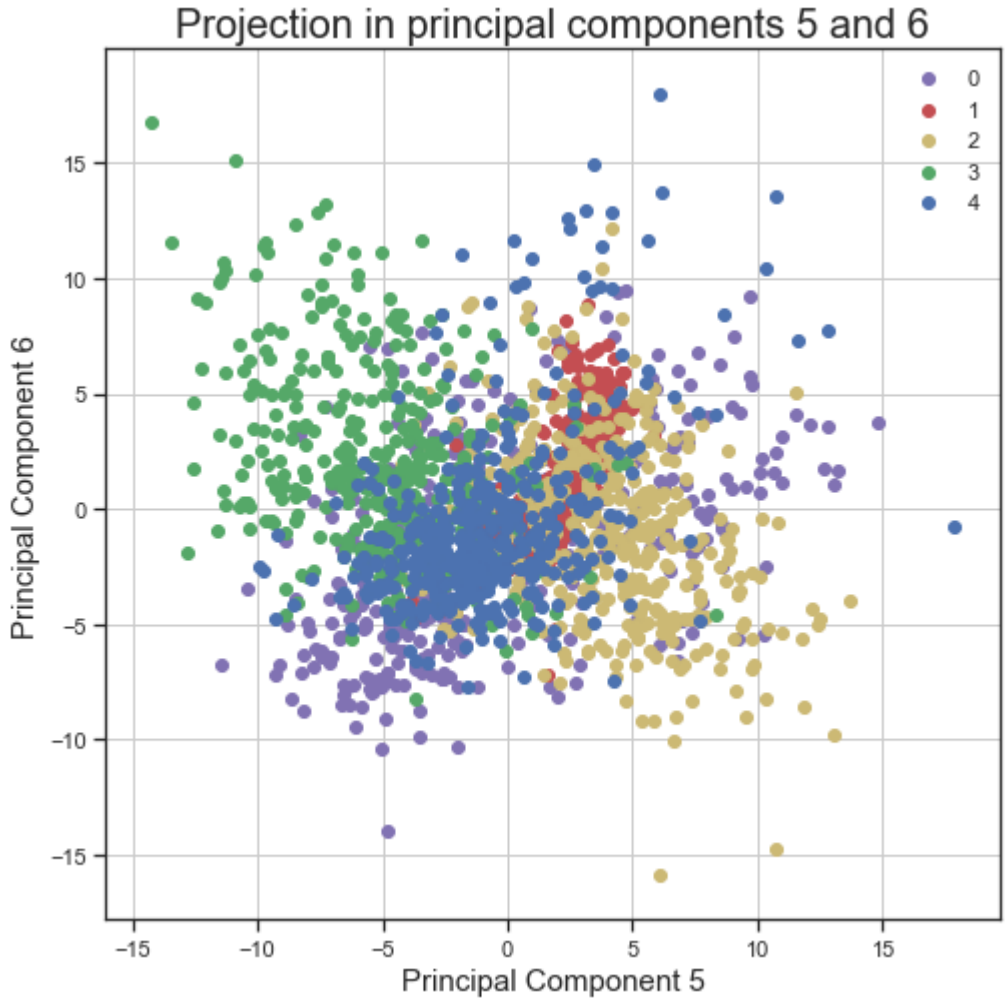
Looking at the range of variance explained by principal components it can be seen, the range of variance explained decreases as the eigenvalues corresponding to these principal components decrease. Also, it can be observed from the 5th plot that variance along principal components 13 and 14 drops drastically, this effect can also be seen from scree plotted for the first 20 eigenvectors.

3) Plot two 2-dimensional representations of the data points based on the first vs second principal components and 5th vs 6th displaying the data points of each class with a different color

In [147]:
```
figure = plt.figure(figsize = (8,8))
ax = figure.add_subplot(1,1,1)
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_title('Projection in principal components 1 and 2', fontsize = 20)
class_colors = [0,1,2,3,4]
colors = ['m','r','y','g','b']
for class_color, color in zip(class_colors,colors):
    indicesTokeep = total_PCA_df_class['class'] == class_color
    ax.scatter(x = total_PCA_df_class.loc[indicesTokeep,'Principal Component 1']
               , c = color)
ax.legend(class_colors)
ax.grid()
```



In [148]:
```
figure = plt.figure(figsize = (8,8))
ax = figure.add_subplot(1,1,1)
ax.set_ylabel('Principal Component 6', fontsize = 15)
ax.set_xlabel('Principal Component 5', fontsize = 15)
ax.set_title('Projection in principal components 5 and 6', fontsize = 20)
class_colors = [0,1,2,3,4]
colors = ['m','r','y','g','b']
for class_color, color in zip(class_colors,colors):
    indicesTokeep = total_PCA_df_class['class'] == class_color
    ax.scatter(x = total_PCA_df_class.loc[indicesTokeep,'Principal Component 5']
               , c = color)
ax.legend(class_colors)
ax.grid()
```



1. From the above plot of the data on first and second principal components and fifth and sixth principal components, it can be visualized from the range of the different principal components that the variance described by the first two principal components is greater than the fifth and sixth components.
2. Classes of the MNIST digit dataset have maximum variance in the first principal components describing 6.6% of the total variance. While the second principal component explains 3.67% of

the total variance.

3. The explained variance corresponding principal component(individual) having low eigon-values, decrease as can be understood from the plot.
4. Also, the classes look more separated when projected on principal components 1 and 2 compared to the projection on principal components 5 and 6.

4) Implement (A) PCA and (B) dual PCA with singular value decomposition.

Here PCA will be implemented manually, no PCA or SVD library would be used

implementing PCA using svd

In [149]:
```
DataB_features = DataB_features_scaled
DataB_features_np = DataB_features.T
mean_array = np.zeros((784,1))
for i in range(0,784):
    mean_array[i,0] = np.mean(DataB_features_np[i,:])
scatter_matrix = np.zeros((784,784))
DataB_features_np_normalized = DataB_features_np - mean_array
DataB_features_np_normalized_T = DataB_features_np_normalized.transpose()
scatter_matrix = np.matmul(DataB_features_np_normalized,DataB_features_np_normali
np.shape(scatter_matrix)
```

Out[149]: (784, 784)

In [150]:
```
eigon_value, eigon_vector = np.linalg.eig(scatter_matrix)
```

In [151]:
```
np.shape(eigon_value)
```

Out[151]: (784,)

In [152]:
```
np.shape(eigon_vector)
```

Out[152]: (784, 784)

In [153]:
```
# Make a list of (eigenvalue, eigenvector) tuples
eigon_pairs = [(np.abs(eigon_value[i]), eigon_vector[:,i]) for i in range(len(eig

# Sort the (eigenvalue, eigenvector) tuples from high to low
eigon_pairs.sort(key=lambda x: x[0], reverse=True)
```

In [154]:
```
u_matrix = np.zeros((784,20))
for i in range(0,20):
    u_matrix[:,i]= eigon_pairs[i][1]
np.shape(u_matrix)
```

Out[154]: (784, 20)

In [155]:
```
u_matrix_transpose = u_matrix.transpose()
np.shape(u_matrix_transpose)
```

Out[155]: (20, 784)

In [156]:
```
# lets project data
pca_transformed_data = np.matmul(u_matrix_transpose,DataB_features_np).transpose
np.shape(pca_transformed_data)
#pca_transformed_data
```

Out[156]: (2066, 20)

In [157]:
```
#Lets see how scree plot looks like here
eigon_vector_list = []
for i in range(0,20):
    eigon_vector_list.append(eigon_pairs[i][0])
eigon_value_manual_df = pd.DataFrame(data = eigon_vector_list, columns= ['Eigon
```

Implementing dual PCA using svd

In [158]:
```
DataB_features_np_normalized = DataB_features_np - mean_array
DataB_features_np_normalized_T = DataB_features_np_normalized.transpose()
A_transpose_A = np.matmul(DataB_features_np_normalized_T, DataB_features_np_norma
np.shape(A_transpose_A)
```

Out[158]: (2066, 2066)

In [159]:
```
A_T_A_eigon_value, A_T_A_eigon_vector = np.linalg.eigh(A_transpose_A)
```

In [160]:
```
# Make a list of (eigenvalue, eigenvector) tuples
A_T_A_eigon_pairs = [(np.abs(A_T_A_eigon_value[i]), A_T_A_eigon_vector[:,i]) for
# Sort the (eigenvalue, eigenvector) tuples from high to low
A_T_A_eigon_pairs.sort(key=lambda x: x[0], reverse=True)
```

In [161]:
```
S_matrix = np.zeros((20,2066))
for i in range(0,20):
    S_matrix[i][i] = np.sqrt(A_T_A_eigon_pairs[i][0])
V_matrix = np.zeros((2066,2066))
for i in range(0,2066):
    V_matrix[:,i]= A_T_A_eigon_pairs[i][1]
np.shape(V_matrix)
```

Out[161]: (2066, 2066)

In [162]:
```
#calculating SV as Utranspose X
dualpca_transformed_data = (np.matmul(S_matrix,V_matrix.transpose())).transpose(
np.shape(dualpca_transformed_data)
```

Out[162]: (2066, 20)

lets compare time across both PCA and Dual PCA

In [163]:
```
#PCA
import time
start = time.time()
mean_array = np.zeros((784,1))
for i in range(0,784):
    mean_array[i,0] = np.mean(DataB_features_np[i,:])
scatter_matrix = np.zeros((784,784))
DataB_features_np_normalized = DataB_features_np - mean_array
DataB_features_np_normalized_T = DataB_features_np_normalized.transpose()
scatter_matrix = np.matmul(DataB_features_np_normalized,DataB_features_np_normali
eigon_value, eigon_vector = np.linalg.eig(scatter_matrix)
# Make a list of (eigenvalue, eigenvector) tuples
eigon_pairs = [(np.abs(eigon_value[i]), eigon_vector[:,i]) for i in range(len(eig
# Sort the (eigenvalue, eigenvector) tuples from high to low
eigon_pairs.sort(key=lambda x: x[0], reverse=True)
u_matrix = np.zeros((784,20))
for i in range(0,20):
    u_matrix[:,i]= eigon_pairs[i][1]
u_matrix_transpose = u_matrix.transpose()
pca_transformed_data = np.matmul(u_matrix_transpose,DataB_features_np).transpose
end = time.time()

print("Time taken for implementing PCA is ", end - start)

import time
start_dual = time.time()
mean_array = np.zeros((784,1))
DataB_features_np_normalized = DataB_features_np - mean_array
DataB_features_np_normalized_T = DataB_features_np_normalized.transpose()
A_transpose_A = np.matmul(DataB_features_np_normalized_T, DataB_features_np_norma
A_T_A_eigon_value, A_T_A_eigon_vector = np.linalg.eigh(A_transpose_A)
# Make a list of (eigenvalue, eigenvector) tuples
A_T_A_eigon_pairs = [(np.abs(A_T_A_eigon_value[i]), A_T_A_eigon_vector[:,i]) for
# Sort the (eigenvalue, eigenvector) tuples from high to low
A_T_A_eigon_pairs.sort(key=lambda x: x[0], reverse=True)
S_matrix = np.zeros((20,2066))
for i in range(0,20):
    S_matrix[i][i] = np.sqrt(A_T_A_eigon_pairs[i][0])
V_matrix = np.zeros((2066,2066))
for i in range(0,2066):
    V_matrix[:,i]= A_T_A_eigon_pairs[i][1]
dualpca_transformed_data = (np.matmul(S_matrix,V_matrix.transpose())).transpose(

end_dual = time.time()
print("Time taken for implementing dual PCA is ", end_dual - start_dual)
```

```
Time taken for implementing PCA is  0.3450784683227539
Time taken for implementing dual PCA is  1.0068280696868896
```

In [164]:
```
# lets see results of projected data for PCA
print("Results of PCA transformed data is :",pca_transformed_data)
```

```
Results of PCA transformed data is : [[ -9.97069222   6.18172201  -4.99286326
...  -0.26257488   1.42584762
   -1.16252257]
 [-11.41599978   6.94158705  -5.06302886 ...   0.96317397   1.11655238
    0.06708945]
 [ -3.69011918   4.69309729  -2.9086564  ...   2.65907012  -0.66109634
   -5.12371489]
 ...
 [  0.34942153   0.93368106   8.10744188 ...  -1.28086781   1.19700404
    1.08146006]
 [  3.11526327   2.09047425   6.27251911 ...  -1.30774666  -0.11716451
    1.59384718]
 [  5.64409375  -0.24616663   4.14018317 ...   2.8474039   -1.14882287
   -3.39490069]]
```

In [165]:
```
# lets see results of projeted data for dual PCA
print("Results of dual PCA transformed data is :",dualpca_transformed_data)
```

```
Results of dual PCA transformed data is : [[ 9.97069222 -6.18172201  4.99286326
...  0.26257488 -1.42584762
   1.16252257]
 [11.41599978 -6.94158705  5.06302886 ... -0.96317397 -1.11655238
  -0.06708945]
 [ 3.69011918 -4.69309729  2.9086564  ... -2.65907012  0.66109634
   5.12371489]
 ...
 [-0.34942153 -0.93368106 -8.10744188 ...  1.28086781 -1.19700404
  -1.08146006]
 [-3.11526327 -2.09047425 -6.27251911 ...  1.30774666  0.11716451
  -1.59384718]
 [-5.64409375  0.24616663 -4.14018317 ... -2.8474039   1.14882287
   3.39490069]]
```

For PCA, we decompose the matrix $X^T * X$ which has $d * d$ dimension where $d = 784$ for MNIST dataset whereas for Dual PCA, we decompose the matrix $X * X^T$ which has $n * n$ dimensions where $n = 2066$ for MNIST dataset. Decomposition of matrix with dimension $2066 * 2066$ takes longer time than decomposition of matrix with $784 * 784$ dimensions. That is why Dual PCA takes more time than PCA.

This is further being supported by the fact that the execution time of PCA is 0.35 seconds while dual PCA takes around 1.01 seconds.

2.2.2 Theoritical Questions

Prove that PCA is the best linear method for reconstruction (with orthonormal bases).

$\hat{X}$ is data point in original space and $UU^T\hat{X}$ is reconstruction of projected data on the principal components.

In order to reduce the recunstruction erroe we need to form an optimization problem and minimize it. The optimization problem is described as below:

$$\begin{aligned} \underset{U}{\text{minimize}} \quad & ||\hat{X} - UU^T\hat{X}||_F^2 \\ \text{subject to} \quad & U^TU = I. \end{aligned}$$

$||\hat{X} - UU^T\hat{X}||_F^2$

$= tr((\hat{X} - UU^T\hat{X})^T(\hat{X} - UU^T\hat{X}))$

$= tr((\hat{X}^T - \hat{X}^TUU^T)(\hat{X} - UU^T\hat{X}))$

$= tr(\hat{X}^T\hat{X} - 2\hat{X}^TUU^T\hat{X} + \hat{X}^T\underbrace{UU^TU}_{\text{I}}U^T\hat{X}) = tr(\hat{X}^T\hat{X} - \hat{X}^TUU^T\hat{X})$

$= tr(\hat{X}^T\hat{X}) - tr(\hat{X}^TUU^T\hat{X})$

$= tr(\hat{X}^T\hat{X}) - tr(\hat{X}\hat{X}^TUU^T)$

Using Lagrange multiplier, we have:
$\mathcal{L} = tr(\hat{X}^T\hat{X}) - tr(\hat{X}\hat{X}^TUU^T) - tr(\Lambda^T(U^TU - I)),$

where $\Lambda \in \mathbb{R}^{p\times p}$ is a diagonal matrix $diag([\lambda_1, \ldots, \lambda_p]^T)$ containing the Lagrange multipliers. Equating the derivative of Lagrangian to zero gives:

$\mathbb{R}^{d\times p} \ni \frac{\partial\mathcal{L}}{\partial U} = 2\hat{X}\hat{X}^TU - 2U\Lambda = 0$

$\implies \hat{X}\hat{X}^TU = U\Lambda, \implies SU = U\Lambda$

This is the eigen value problem for the covariance matrix S. We had same eigen value problem in PCA.

PCA subspace is the best linear projection in terms of reconstruction error as reconstruction error is minimized when maximum variance are captured along the data points. In other words, PCA has the least squared error in reconstruction.

2.3 Fisher Discriminant Analysis (FDA)

2.3.1 Practical Question

1) Applying LDA to reduce Dimensionality

In [166]:
```
#X_full_scaled_c
#yc_new
#lda = LinearDiscriminantAnalysis(n_components = 2)
#ldaComponents = lda.fit_transform(X_full_scaled_c, yc_new)
col_name_features = ["Feature " + str(i) for i in range(1,785)]

DataB_features_scaled_df = pd.DataFrame(data = DataB_features_scaled, columns = 
X_lda_df = DataB_features_scaled_df
y_lda_df = DataB_class
y_lda_np = pd.DataFrame.to_numpy(y_lda_df)
```

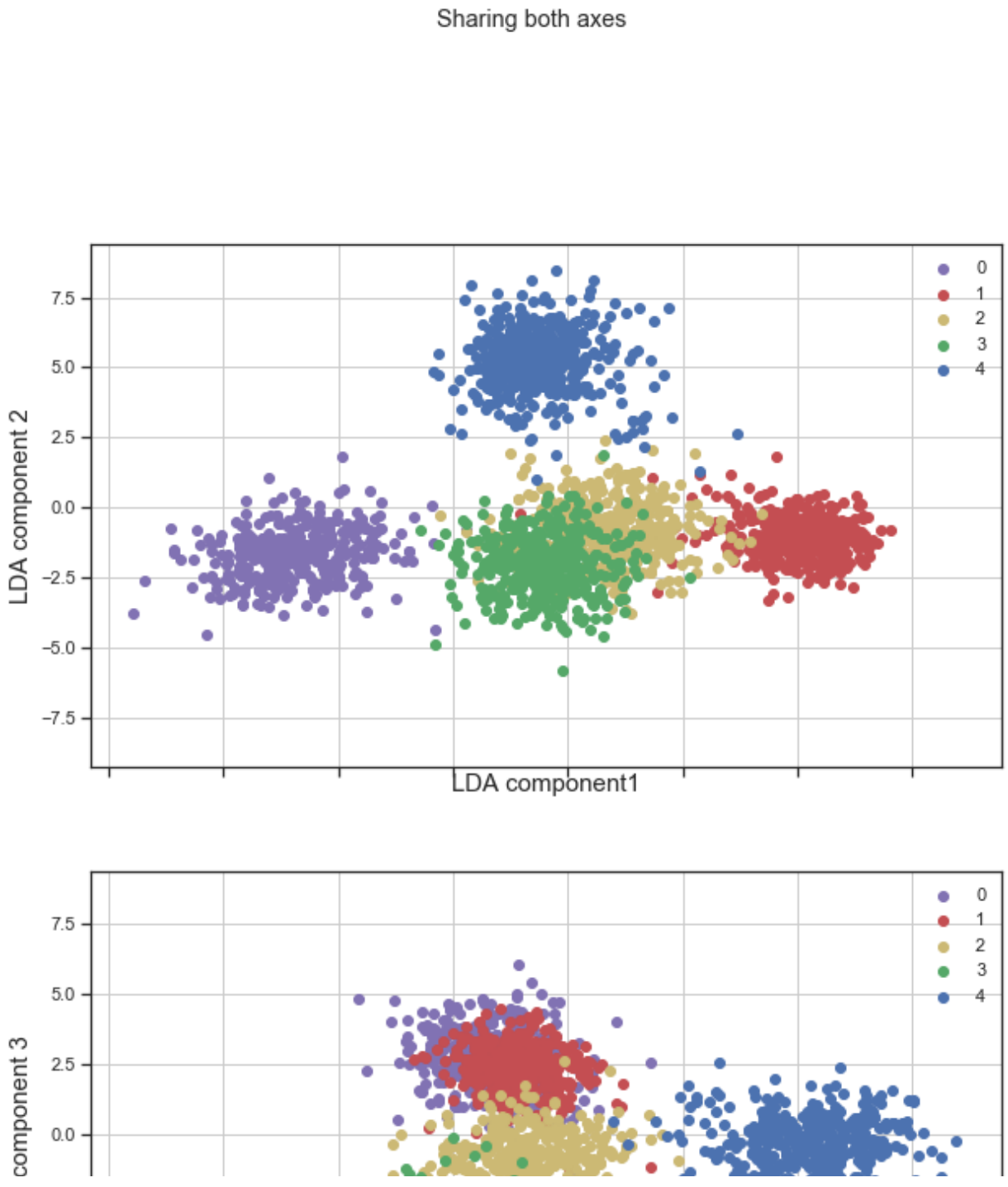
In [167]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
lda = LinearDiscriminantAnalysis(n_components= 4)
ldaComponents =lda.fit_transform(X_lda_df, np.ravel(y_lda_np))
np.shape(ldaComponents)
```

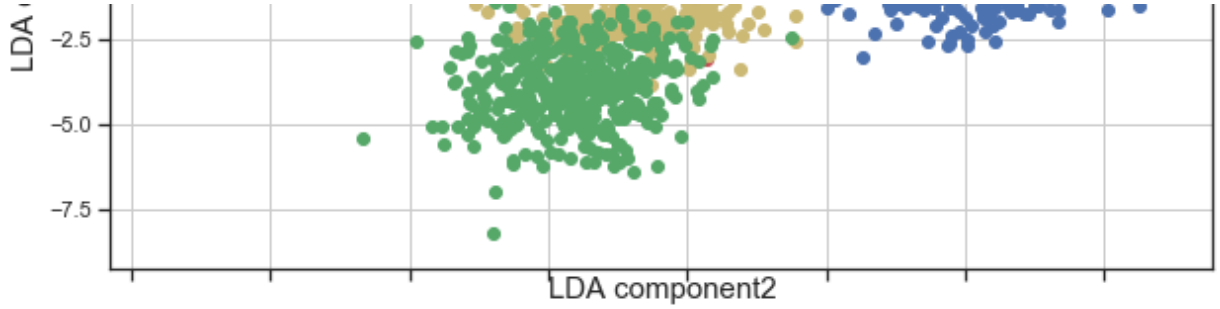
Out[167]: (2066, 4)

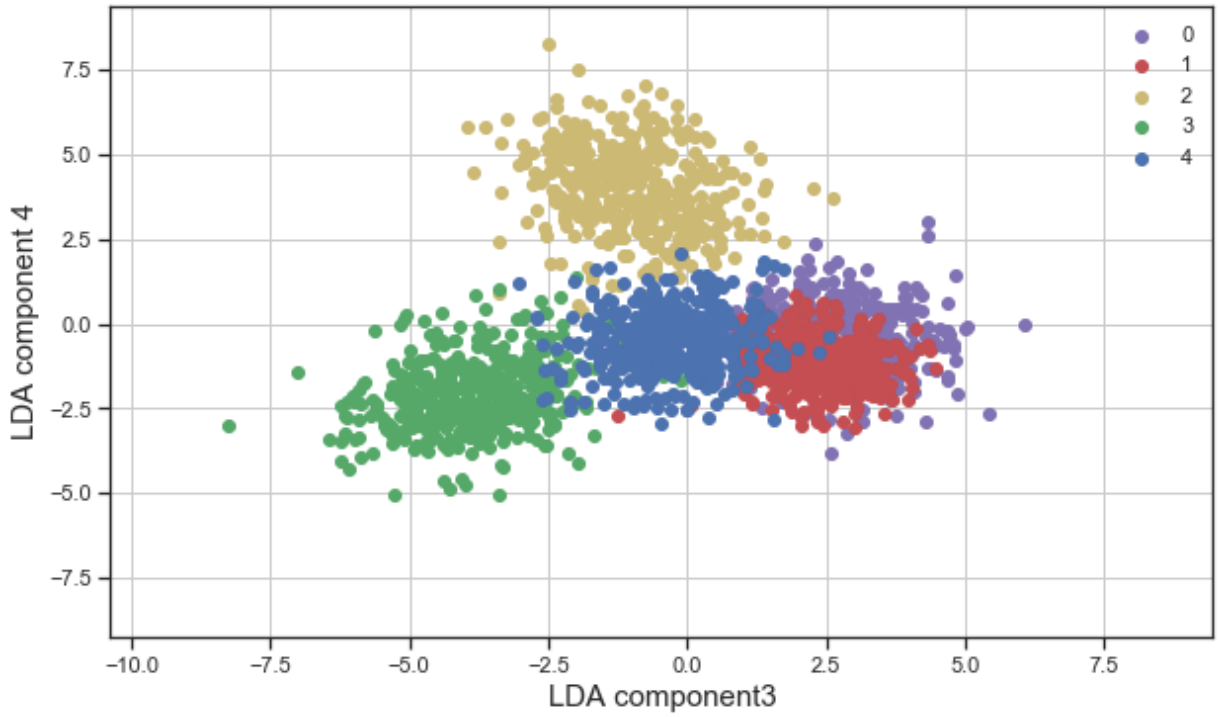
In [168]:
```
lda_1d_c = pd.DataFrame(data = ldaComponents, columns= ['LDA component 1','LDA c
lda_1d_final_c = pd.DataFrame.join(lda_1d_c, y_lda_df)
lda_1d_final_c.rename_axis('index')
lda_1d_final_c =lda_1d_final_c.assign(new_index = lambda z: z.index)
lda_1d_final_c.shape
```

Out[168]: (2066, 6)

In [169]:
```
fig,axs = plt.subplots(3, sharex=True, sharey=True, figsize =(10,20))
fig.suptitle('Sharing both axes')
for i in range(0,3):
    axs[int(i)].set_ylabel('LDA component '+ str(i+2), fontsize = 15)
    axs[int(i)].set_xlabel('LDA component' + str(i+1), fontsize = 15)
    class_colors = [0,1,2,3,4]
    colors = ['m','r','y','g','b']
    for class_color, color in zip(class_colors,colors):
        indicesTokeep = lda_1d_final_c['class'] == class_color
        axs[int(i)].scatter(x = lda_1d_final_c.loc[indicesTokeep,'LDA component
                  , c = color)
        axs[int(i)].legend(class_colors)
        axs[int(i)].grid()
```







It can be visualized from the above plots that different classes can be distinct in different directions. Following are the different directions responsible for different classes;
LDA component 1: This direction is responsible for the separation of the digit 0 and digits 1 as it can be observed from cluster spread along the x-axis of the first graph.
LDA component 2: if projected data on direction 2 than except digit 4 all the other classes are collapsed near to each other. So, the only digit 2 can be classified along direction 2.
LDA component 3: Direction 3 separates the MNIST digit 3 when projected on it
LDA component 4: when data are projected along direction 4 then digit 2 class seems to have separability from the rest of the class.

2) Compare results of LDA with results obatined using PCA

It can be visualized from the above plots, Projection on principal component 1 and principal component 2 provide good visualization of different classes, but LDA outperforms PCA when it comes to comprehending the separability of classes in lower dimensions. Comparing these results of PCA and LDA it can be said that LDA tries to attain maximum separability of classes across different directions while principal components in PCA are in the direction of the maximum variance.

2.3.2 Theoricitical Question

We can consider the total scatter as the summation of the within and between scatters: $\mathbf{S_T = S_W + S_B \implies S_B = S_T - S_W}$. By substituting this into the Fisher criterion, the FDA optimization can be slightly modified to:

Here d is the dimensions of datapoints and p is the dimension of projection space The optimization equation is equivalent to:

$$\begin{aligned} \underset{U}{\text{maximize}} \quad & tr(U^T S_T U) \\ \text{subject to} \quad & U^T S_W U = I. \end{aligned}$$

Using Lagrange multiplier, we have:

$$\mathcal{L} = tr(U^T S_T U) - tr(\Lambda^T (U^T S_W U - I))$$

where $\Lambda \in \mathbb{R}^{d \times d}$ is a diagonal entries are the Lagrange multipliers. Equating the derivative of $\mathcal{L}$ to zero gives:

$$\mathbb{R}^{d \times p} \ni \frac{\partial L}{\partial U} = 2S_T U - 2S_W U\Lambda = 0$$

$$\implies 2S_T U = 2S_W U\Lambda$$

$$\implies S_T U = S_W U\Lambda$$

$$\implies S_W^{-1} S_T U = U\Lambda$$

Which is a generalized eigenvalue problem (S_T, S_W). The columns of U are the eigenvectors sorted by largest to smallest eigenvalues (because the optimization is maximization) and the diagonal entries of Λ are the corresponding eigenvalues. The columns of U are referred to as the Fisher directions or Fisher axes.

The FDA directions can be obtained by the generalized eigenvalue problem ($\mathbf{S_T}$,$\mathbf{S_W}$). By comparing the equations, it shows that PCA captures the orthonormal directions with the maximum variance of data. However, the FDA has the same goal but also it requires the manipulated directions to be orthonormal. This manipulation is done by the within scatter which makes sense because the within scatters make use of the class labels. This comparison gives a hint for the connection between PCA and FDA. From question 2 in a practical question, it is clearly seen that PCA intermingles the classes. There is not a cut point for the dimensions. LDA gives good clear cut dimensions since it considers labels in the data. Suppose there are two different clusters with opposite labels, but still they are placed very near to each other. Most of the data variation in the direction of these clusters. These clusters would be projected onto the direction of the greatest variety of data and it results in the formation of a single cluster of data. So PCA mixes up the clusters without considering the labels. FDA projects the data onto a direction that is orthogonal to

the direction of the greatest variation of the data. This direction is in the least variation of the data. These two clusters would then be nearly perfectly separated from each other because of taking into account of their labels.

References :

1) B. Ghojogh, M. N. Samad, S. A. Mashhadi, T. Kapoor, W. Ali, F. Karray and M. Crowley, "Feature Selection and Feature Extraction in Pattern Analysis: A Literature Review", arXiv:1905.02845v1, 7 May 2019

2) B. Ghojogh,M. Crowley, "Unsupervised and Supervised Principal Component Analysis: Tutorial", arXiv:1906.03148v1, 1 Jun 2019

3) B. Ghojogh, F. Karray and M. Crowley, "Fisher and Kernel Fisher Discriminant Analysis: Tutorial", arXiv:1906.09436v1, 22 Jun 2019

Q 3 Nonlinear Dimensionality Reduction

3.1 Dataset

In [170]:
```
DataB_features =X1
import time
col_name_features = ["Feature "+str(i) for i in range(1,785)]
DataB_features_scaled_df = pd.DataFrame(data = DataB_features, columns = col_name
X_KERNEL_PCA_df = DataB_features
y_KERNEL_PCA_df = DataB_class
```

3.2 Practical Questions

3.2.1 Different Embedding Marks

1) Kernel PCA

In [171]:
```
X_KERNEL_PCA_df = DataB_features
y_KERNEL_PCA_df = DataB_class

from sklearn.decomposition import KernelPCA
start = time.time()
transformer = KernelPCA(n_components=2, kernel='rbf',random_state =42)
KernelPCA_components = transformer.fit_transform(X_KERNEL_PCA_df)
end = time.time()
KernelPCA_components_df = pd.DataFrame(data = KernelPCA_components, columns= ['K
KernelPCA_components_final_df = pd.DataFrame.join(KernelPCA_components_df,y_KERNE
KernelPCA_components_final_df.rename_axis('index')
KernelPCA_components_final_df = KernelPCA_components_final_df.assign(new_index =
print("The time taken by KPCA is :",end - start)
KernelPCA_components_final_df.head()
```

```
The time taken by KPCA is : 0.7659530639648438

C:\Users\aksha\Anaconda3\lib\site-packages\sklearn\utils\extmath.py:516: Runtim
eWarning: invalid value encountered in multiply
  v *= signs[:, np.newaxis]
```

Out[171]:

	KernelPCA_component 1	KernelPCA_component 2	class	new_index
0	0.044795	-0.028970	0	0
1	-0.002986	0.001198	0	1
2	-0.013929	-0.009832	0	2
3	-0.004760	0.002459	0	3
4	0.010747	-0.004858	0	4

(2)Isomap

In [172]:
```
X_ISOMAP_df = DataB_features
y_ISOMAP_df = DataB_class
```

In [173]:
```
from sklearn.manifold import Isomap
start = time.time()
transformer = Isomap(n_components=2)
ISOMAP_components = transformer.fit_transform(X_ISOMAP_df)
end = time.time()
ISOMAP_components_df = pd.DataFrame(data = ISOMAP_components, columns= ['ISOMAP_
ISOMAP_components_final_df = pd.DataFrame.join(ISOMAP_components_df,y_ISOMAP_df)
ISOMAP_components_final_df.rename_axis('index')
ISOMAP_components_final_df = ISOMAP_components_final_df.assign(new_index = lambd
print("The time taken by ISOMAP is :",end - start)
ISOMAP_components_final_df.head()
```

```
The time taken by ISOMAP is : 7.3601460456848145
```

Out[173]:

	ISOMAP_component 1	ISOMAP_component 2	class	new_index
0	8706.049952	-344.127405	0	0
1	8858.821154	-333.866129	0	1
2	3954.252089	-926.795027	0	2
3	5791.867059	-643.647584	0	3
4	10700.083472	281.608819	0	4

(3)LLE

In [174]:
```
X_LLE_df = DataB_features
y_LLE_df = DataB_class
```

In [175]:
```
from sklearn.manifold import LocallyLinearEmbedding
start = time.time()
transformer = LocallyLinearEmbedding(n_components=2,random_state =42)
LLE_components = transformer.fit_transform(X_LLE_df)
end = time.time()
LLE_components_df = pd.DataFrame(data = LLE_components, columns= ['LLE_component
LLE_components_final_df = pd.DataFrame.join(LLE_components_df,y_LLE_df)
LLE_components_final_df.rename_axis('index')
LLE_components_final_df = LLE_components_final_df.assign(new_index = lambda z : 
print("The time taken by LLE is :",end - start)
LLE_components_final_df.head()
```

```
The time taken by LLE is : 6.125690937042236
```

Out[175]:

	LLE_component 1	LLE_component 2	class	new_index
0	-0.045258	-0.000862	0	0
1	-0.045060	-0.000827	0	1
2	-0.044573	-0.000925	0	2
3	-0.044700	-0.000871	0	3
4	-0.046829	-0.001124	0	4

(4) Laplacian Eigenmap

In [176]:
```
# (4) Laplacian Eigenmap
X_LEM_df = DataB_features
y_LEM_df = DataB_classX_LEM_df = DataB_features
y_LEM_df = DataB_class
```

In [177]:
```
from sklearn.manifold import SpectralEmbedding
start = time.time()
transformer = SpectralEmbedding(n_components=2,random_state =42)
LEM_components = transformer.fit_transform(X_LEM_df)
end = time.time()
LEM_components_df = pd.DataFrame(data = LEM_components, columns= ['LEM_component
LEM_components_final_df = pd.DataFrame.join(LEM_components_df,y_LEM_df)
LEM_components_final_df.rename_axis('index')
LEM_components_final_df = LEM_components_final_df.assign(new_index = lambda z : 
print("The time taken by LLE is :",end - start)
LEM_components_final_df.head()
```

```
The time taken by LLE is : 6.960597276687622
```

Out[177]:

	LEM_component 1	LEM_component 2	class	new_index
0	0.003748	-0.000115	0	0
1	0.003848	-0.000155	0	1
2	0.002052	-0.000082	0	2
3	0.002793	0.000009	0	3
4	0.004163	-0.000274	0	4

(5) t-SNE

In [178]:
```
X_TSNE_df = DataB_features
y_TSNE_df = DataB_class
```

In [179]:
```
from sklearn.manifold import TSNE
start = time.time()
transformer = TSNE(n_components=2,random_state =42)
TSNE_components = transformer.fit_transform(X_TSNE_df)
end = time.time()
TSNE_components_df = pd.DataFrame(data = TSNE_components, columns= ['TSNE_compone
TSNE_components_final_df = pd.DataFrame.join(TSNE_components_df,y_LEM_df)
TSNE_components_final_df.rename_axis('index')
TSNE_components_final_df = TSNE_components_final_df.assign(new_index = lambda z
print("The time taken by t-sne is :",end - start)
TSNE_components_final_df.head()
```

```
The time taken by t-sne is : 17.500461101531982
```

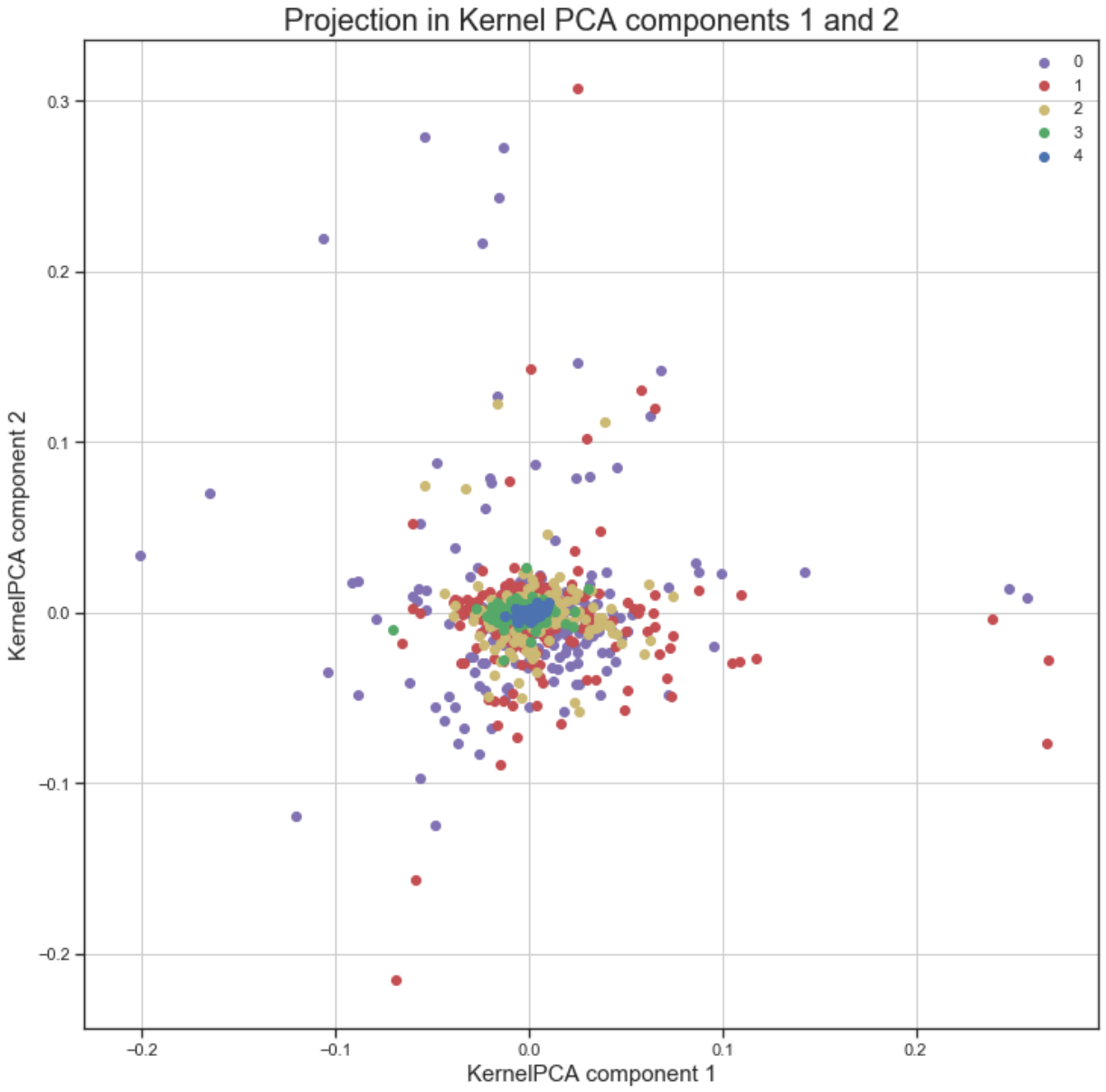
Out[179]:

	TSNE_component 1	TSNE_component 2	class	new_index
0	-53.996166	-7.566008	0	0
1	-54.534958	-8.797090	0	1
2	-43.846584	-0.181216	0	2
3	-44.933132	-3.096598	0	3
4	-60.373814	-7.939509	0	4

3.2 Plot and Compare

(1) Kernel PCA

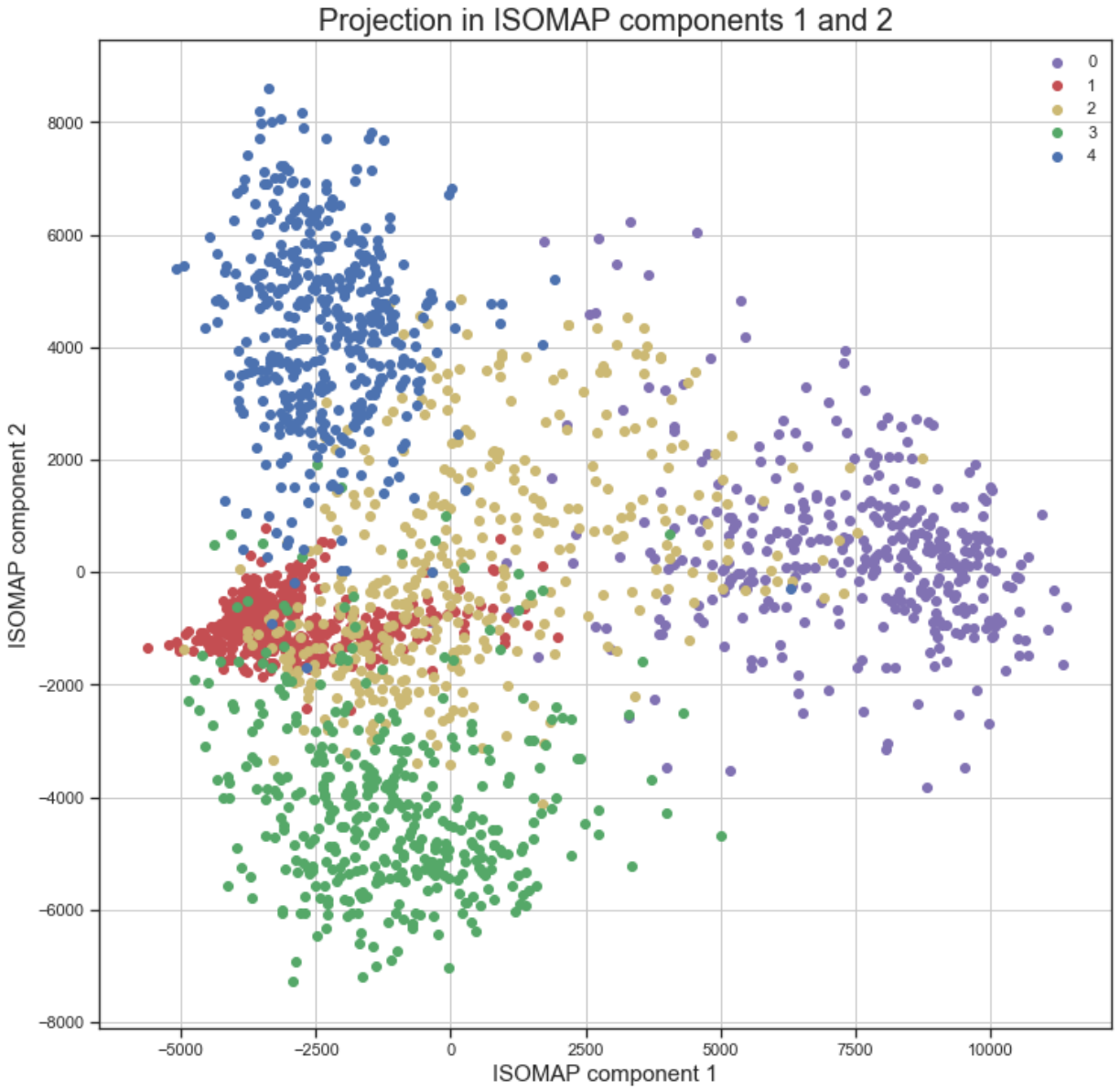
In [180]:
```
figure = plt.figure(figsize = (12,12))
ax = figure.add_subplot(1,1,1)
ax.set_ylabel('KernelPCA component 2', fontsize = 15)
ax.set_xlabel('KernelPCA component 1', fontsize = 15)
ax.set_title('Projection in Kernel PCA components 1 and 2', fontsize = 20)
class_colors = [0,1,2,3,4]
colors = ['m','r','y','g','b']
for class_color, color in zip(class_colors,colors):
    indicesTokeep = KernelPCA_components_final_df['class'] == class_color
    ax.scatter(x = KernelPCA_components_final_df.loc[indicesTokeep,'KernelPCA_co
               , c = color)
ax.legend(class_colors)
ax.grid()
```



- The kernel PCA here is using RBF kernel, since the kernel does not know much about the manifold on which the data lies, its performance on MNIST data seems poor.
- Kernel PCA tends to perform good on theoretical analysis, as the kernel to be used is decided beforehand and is independent of properties of data.
- As we can see from the above plot,Kernel PCA is not able to differentiate among different classes.

(2) Isomap

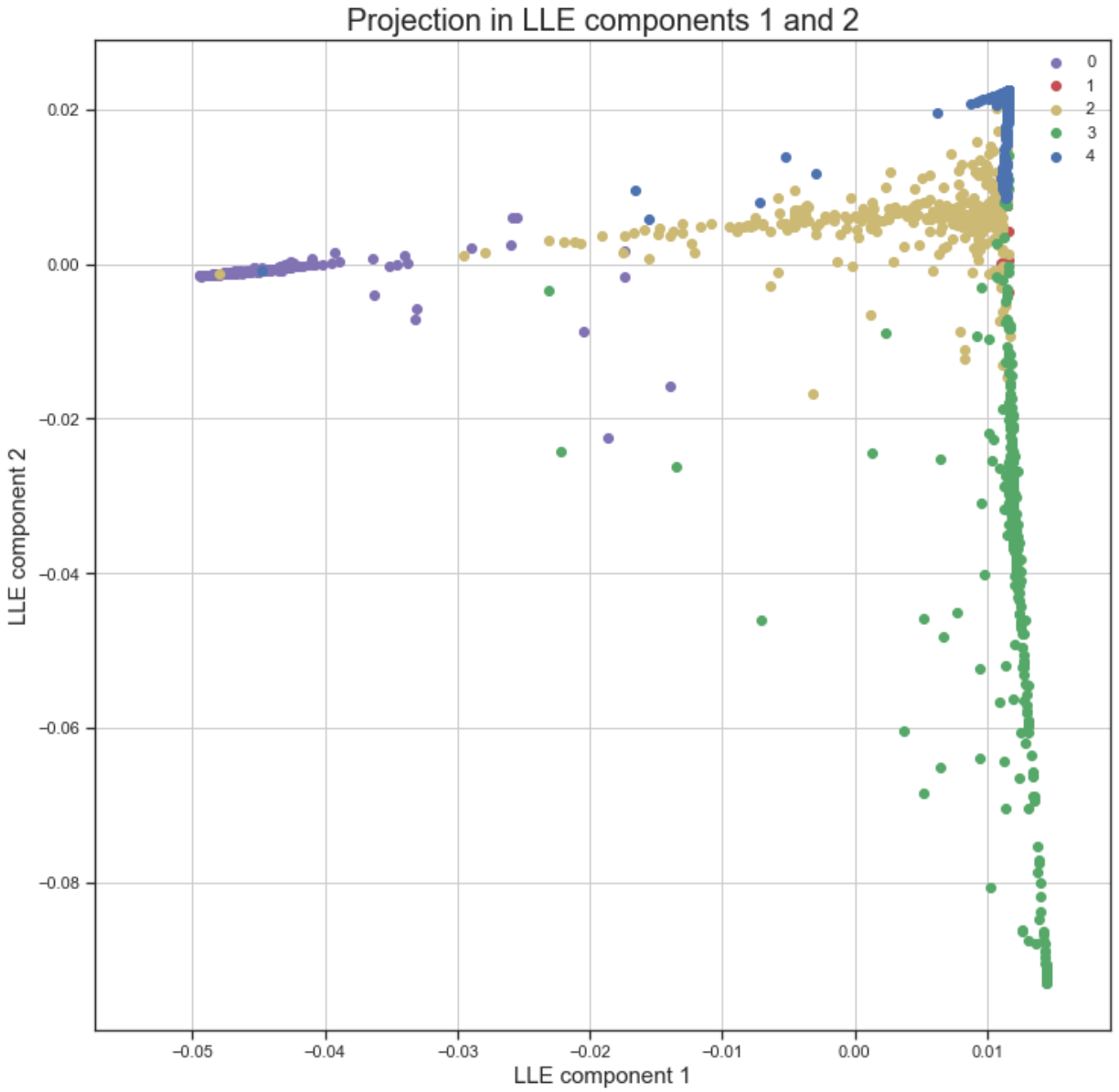
In [181]:
```
figure = plt.figure(figsize = (12,12))
ax = figure.add_subplot(1,1,1)
ax.set_ylabel('ISOMAP component 2', fontsize = 15)
ax.set_xlabel('ISOMAP component 1', fontsize = 15)
ax.set_title('Projection in ISOMAP components 1 and 2', fontsize = 20)
class_colors = [0,1,2,3,4]
colors = ['m','r','y','g','b']
for class_color, color in zip(class_colors,colors):
    indicesTokeep = ISOMAP_components_final_df['class'] == class_color
    ax.scatter(x = ISOMAP_components_final_df.loc[indicesTokeep,'ISOMAP_componen
               , c = color)
ax.legend(class_colors)
ax.grid()
```



- Isomap uses a k-nearest neighbor graph(to use geodesic distance, here k =5) and applies MDS on that graph, thus. It tries to use local distances among nearby points and hence is usually able to unfold the manifold well.
- from the above projections we can see it looks like "octopus" shaped, which can be expected from the theoretical background as it tries to connect a KNN graph.
- It can be seen from the above graph, points 2,3 and 1 share some overlap. In addition to this classes, 4,0 and 3 look well separated.

(3) LLE

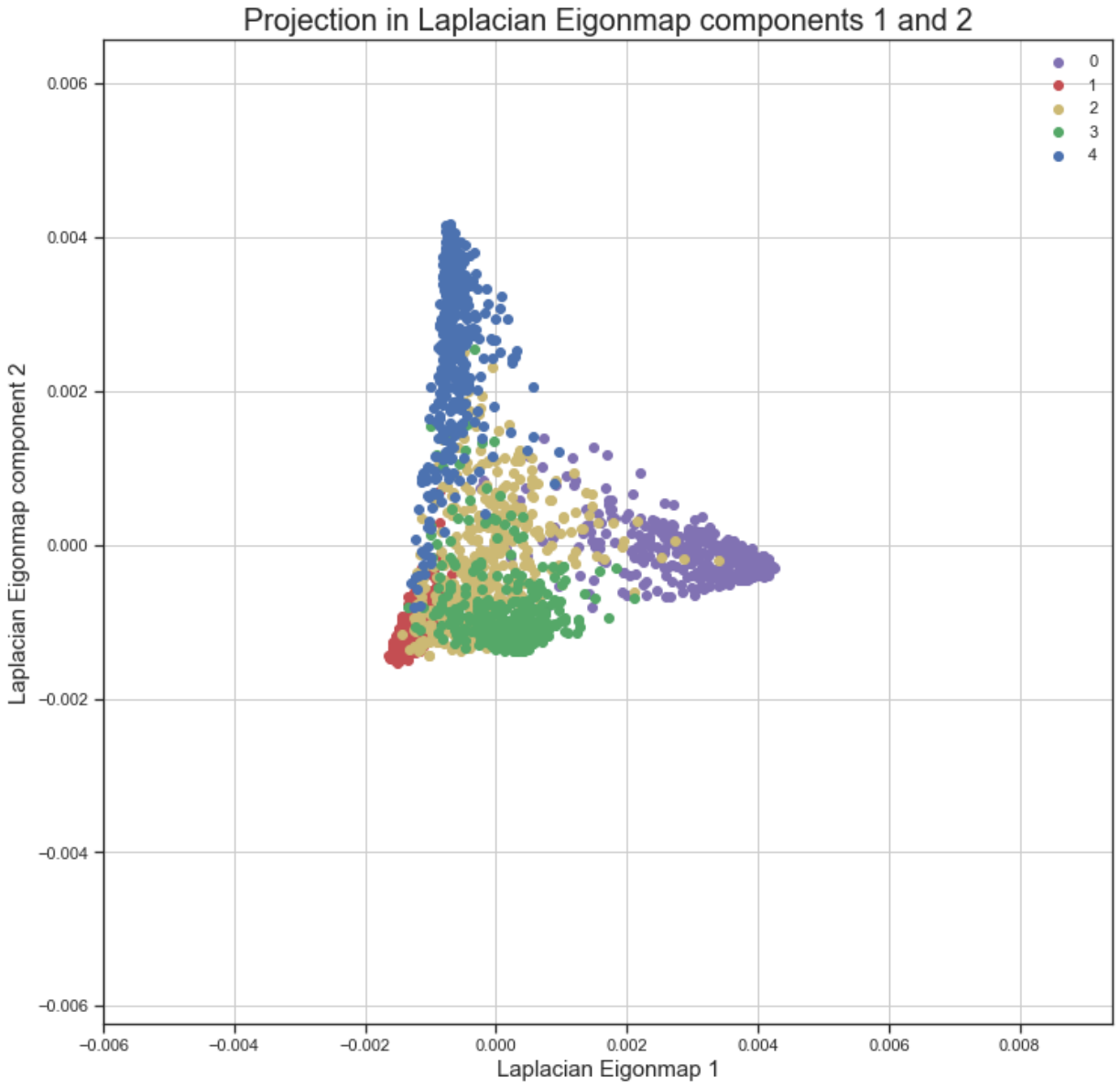
In [182]:
```
figure = plt.figure(figsize = (12,12))
ax = figure.add_subplot(1,1,1)
ax.set_ylabel('LLE component 2', fontsize = 15)
ax.set_xlabel('LLE component 1', fontsize = 15)
ax.set_title('Projection in LLE components 1 and 2', fontsize = 20)
class_colors = [0,1,2,3,4]
colors = ['m','r','y','g','b']
for class_color, color in zip(class_colors,colors):
    indicesTokeep = LLE_components_final_df['class'] == class_color
    ax.scatter(x = LLE_components_final_df.loc[indicesTokeep,'LLE_component 1'],y
               , c = color)
ax.legend(class_colors)
ax.grid()
```



- The result of LLE is almost symmetric it can be attributed as optimization of uses the constraint of unit covariance. (It assumes the dense distribution of data points in original space, and does not perform well in presence of outliers in the original data)
- Classes 0,2,3 and 4 looks well separated from each other, whereas there is substantial overlap among class 1 and classes 2,3 and 4.
- LLE is sensitive to outliers and noise. Datasets have a varying density and it is not always possible to have a smooth manifold. In these cases, LLE gives a poor result. (Reference: https://blog.paperspace.com/dimension-reduction-with-lle/ (https://blog.paperspace.com/dimension-reduction-with-lle/))

(4) Laplacian eigonmap

In [183]:
```
figure = plt.figure(figsize = (12,12))
ax = figure.add_subplot(1,1,1)
ax.set_ylabel('Laplacian Eigonmap component 2', fontsize = 15)
ax.set_xlabel('Laplacian Eigonmap 1', fontsize = 15)
ax.set_title('Projection in Laplacian Eigonmap components 1 and 2', fontsize = 2
class_colors = [0,1,2,3,4]
colors = ['m','r','y','g','b']
for class_color, color in zip(class_colors,colors):
    indicesTokeep = LEM_components_final_df['class'] == class_color
    ax.scatter(x = LEM_components_final_df.loc[indicesTokeep,'LEM_component 1'],y
               , c = color)
ax.legend(class_colors)
ax.grid()
```

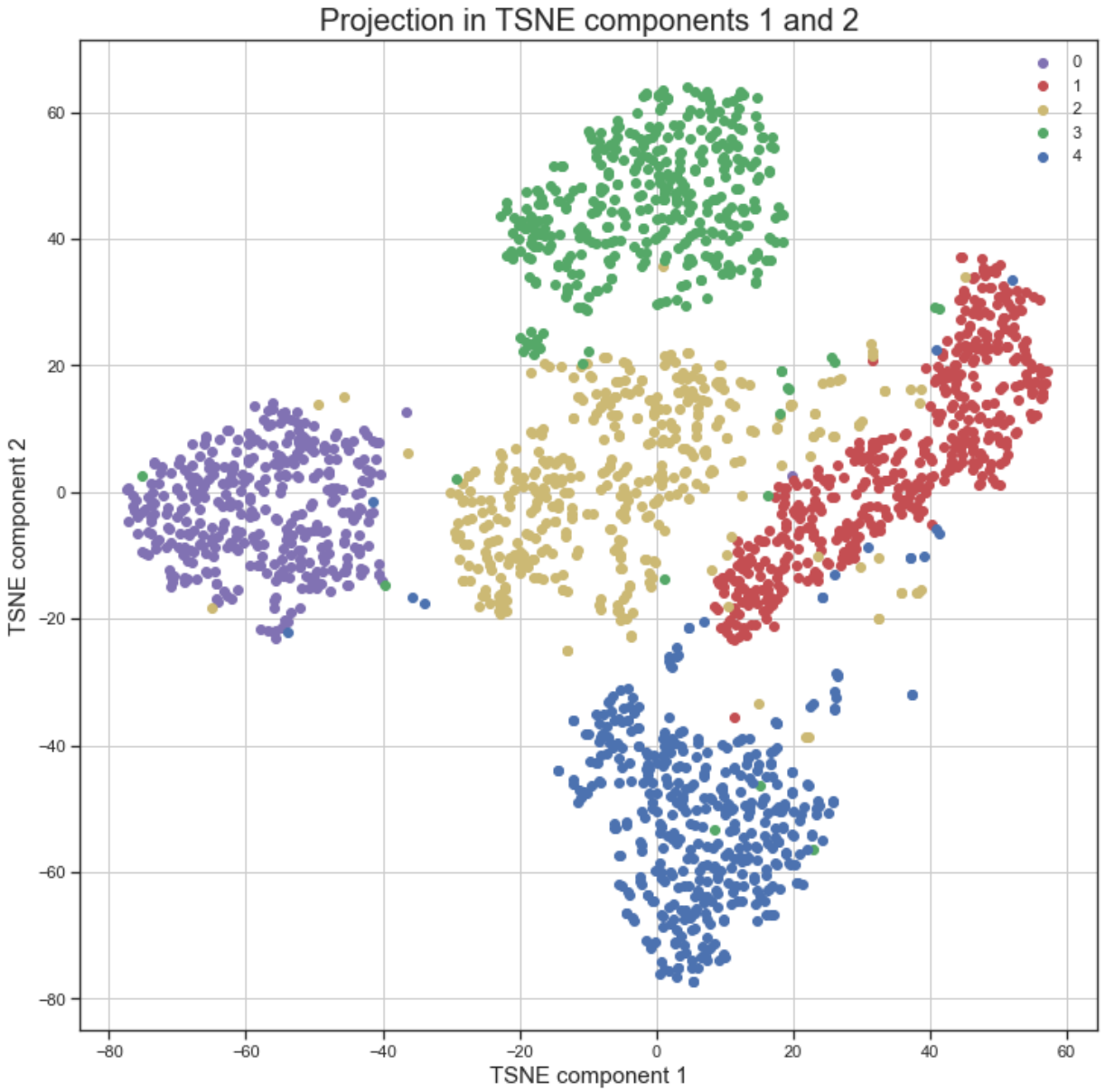


From the above plot and theoretical background on Laplacian eigenmaps, we have the following observations :

- Datapoints sharing similar features look very close to each other in the lower dimensional space, this can be attributed to one-directional cost function used for the Laplacian eigenmaps algorithm, which penalizes cost if we do not maintain the high similarity(in projected space) among the most similar points(from the original space).
- The cluster of data looks good and compact.
- it can be visualized that classes 0,3 and 4 looks separated from the rest of the classes.

(5)T-sne

In [184]:
```
figure = plt.figure(figsize = (12,12))
ax = figure.add_subplot(1,1,1)
ax.set_ylabel('TSNE component 2', fontsize = 15)
ax.set_xlabel('TSNE component 1', fontsize = 15)
ax.set_title('Projection in TSNE components 1 and 2', fontsize = 20)
class_colors = [0,1,2,3,4]
colors = ['m','r','y','g','b']
for class_color, color in zip(class_colors,colors):
    indicesTokeep = TSNE_components_final_df['class'] == class_color
    ax.scatter(x = TSNE_components_final_df.loc[indicesTokeep,'TSNE_component 1'
               , c = color)
ax.legend(class_colors)
ax.grid()
```



As expected, t-SNE separates classes well, following are the observations from the above plot and theory :

- Most Classes form spherical/ t-distribution patches, which can be attributed to the use of t distribution in the theoretical framework for t-SNE.
- Classes are separated well, i.e. there is a significant distance among the patches for different classes, thus it can be understood as t-SNE deals with the probabilities and also these are further multiplied(probabilities) by a factor of 4.
- There is very little overlap shared among different categories.
- all classes; 0,1,2,3,4 looks well separated in t-SNE transformed 2-d plot with very few data points outside the class patches.

Comparison among Manifold methods:

1. Most of the manifold learning methods such as isomap, LLE, LE and t-sne are able to differentiate between different classes of MNIST data except RBF kernel-based Kernel PCA. This can be attributed to the fact that the kernel used by KPCA(here in this question) is Radial Basis function and is independent of data features, whereas the rest of techniques used here use a data-driven kernel and hence are comparatively more accurate for visualizing different classes present in the data.
2. From time calculation performed above, it can be seen that KPCA(approx 0.8 seconds) takes almost one-tenth of time compared to the time taken by isomap (approx 7 seconds), LLE (approx 6 seconds), LE (approx 7 seconds). Furthermore, t-SNE(approx 18 seconds) takes almost as much as thrice time compared to isomap, LLE and LE. (Time taken by ISOMAP and LLE depends on the number of neighbors, here these algorithms are performed using k = 5)
3. t-SNE vs KPCA: A) Time taken by t-SNE(approx 18 seconds) is substantially more than time taken by KPCA (approx 0.8 seconds), as t-SNE based on iteratively placing points together on lower dimension space(by converting pairwise distances to probabilities) and does not have a convex cost function, Whereas KPCA has an optimized solution (based on the use of Kernel). B) t-SNE is able to differentiate among the different classes of data well as can be seen from

the above plots. On the other hand, KPCA does not seem to be able to distinguish the classes and the projected data looks scattered for this dataset. C) t-SNE is able to maintain the distance between various class patches (as t-SNE uses T distribution to reduce overlap), whereas the classes in KPCA projected space to exhibit a lot of overlap.

D. Trade-off while deciding the best methods :

Time: Time taken by these algorithms is diffrenent if computational power is limited, a method which takes less time to execute would be preferred. Therefore in such a scenario, Isomap, LE, and LLE would be better choice.
Visualization / seprability: The best visualization in terms of separability is provided by t-SNE and hence, in situations where it is more important to visualize irrespective of the computational time taken by the algorithm, t-SNE would be the most preferred method.

In [ ]: