Assignment 1

Question 1

This segment of the assignment uses wine quality dataset with 13 variables. Of these variables, 11 serve as the predictor variables and 2 i.e. color and quality serve as the outcome variables. There are two types of wine color - red and white and seven qualities of wine.

The flow of the assignment is as follows:

1. Unnormalized dataset is produced.
2. Pairplot for the data with color and quality as labels is plotted
3. The dataset is normalized using the z-score normalization
4. Pairplot for the data with color and quality as labels is plotted
5. The normalized dataset is split into train and test set(using color and quality of data as outcomes respectively)
6. KNN model is implemented on the dataset with different sets of weighting and distance schemes
7. PCA and LDA are implemented for dimensionality reduction

In [1]:
```
# 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 [2]:
```
#Columns/Features
D = ['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar', 'c
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])
```

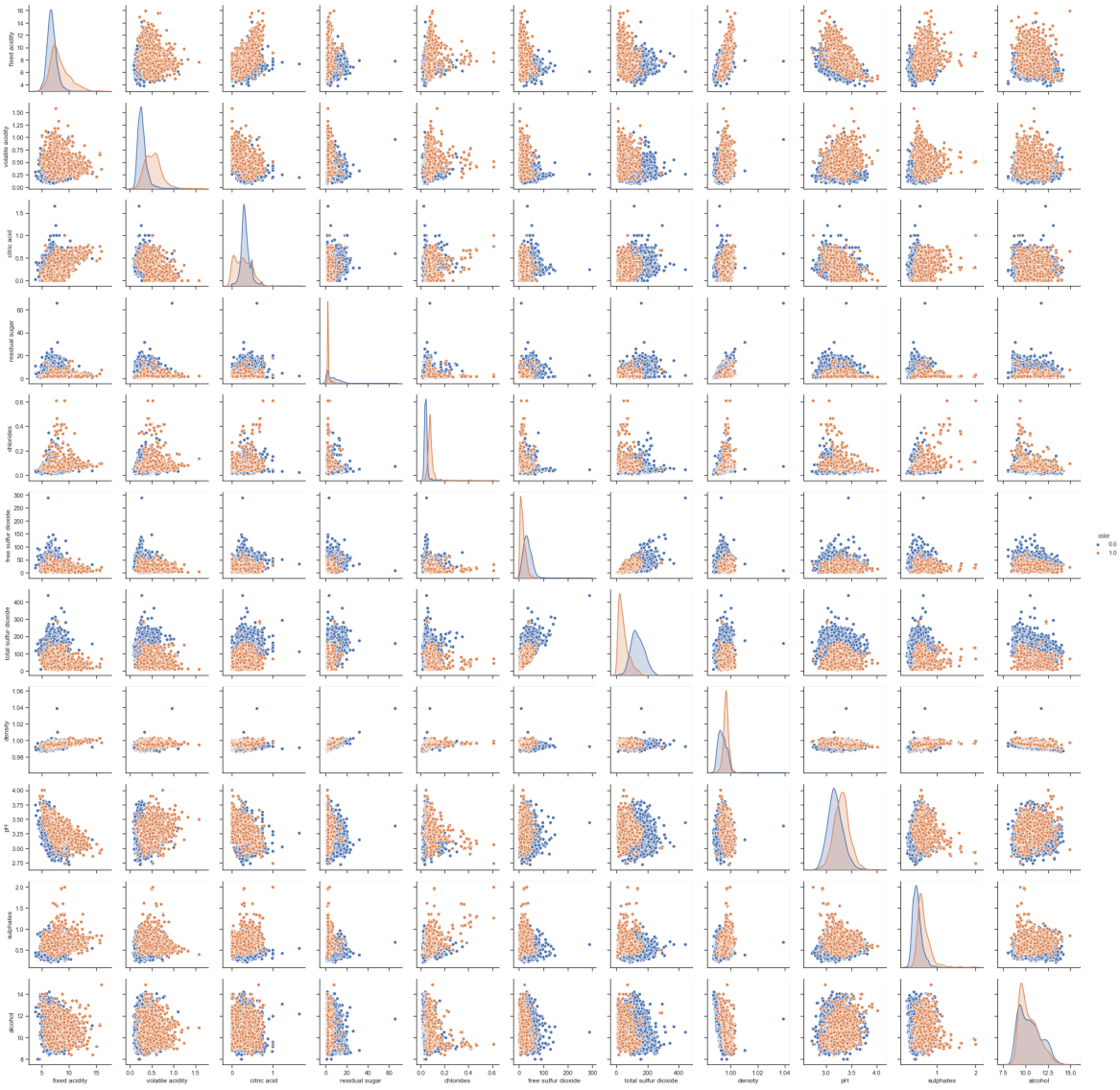
The above dataset contains data for 11 variables in D with quality data in L and color data in C.

Plotting the dataset

To understand the relationship between the variables, a pairplot is plotted. In the following pairplot, the color has been chosen as the label and the data is plotted before normalization.

In [4]:
```
sns.pairplot(wine,hue="color",vars=D)
```

Out[4]:
```
<seaborn.axisgrid.PairGrid at 0x20fd397fb08>
```

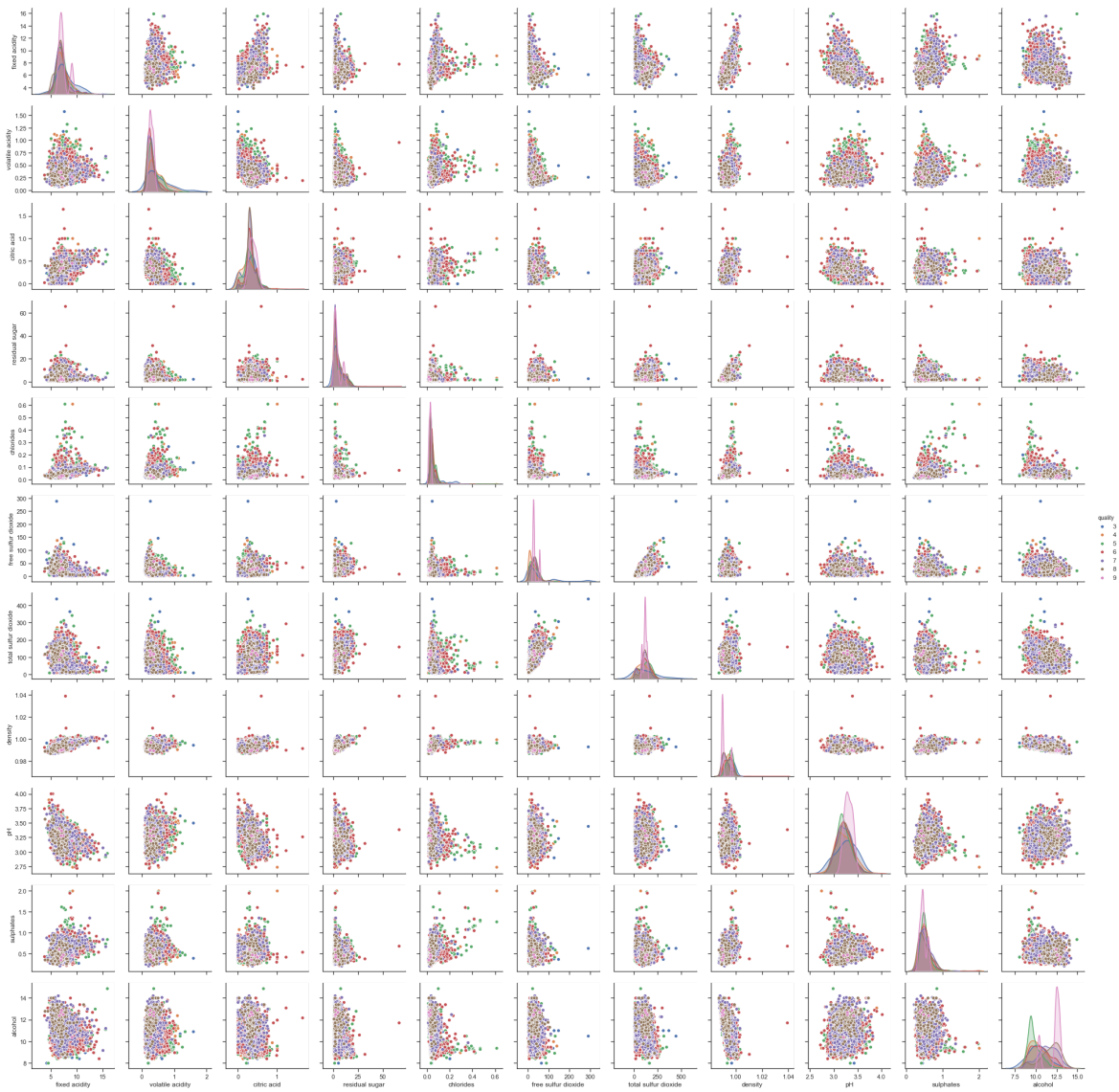


From the plot, it can be seen that none of the variables provided complete distinction between the two classes of wine color. Still variables such as total sulphur dioxide, chlorides, volatile acidity can be used for maximum accuracy.

In the following pairplot, the quality has been chosen as the label and the data is plotted before normalization.

In [5]:
```
sns.pairplot(wine,hue="quality",vars=D)
```

Out[5]: <seaborn.axisgrid.PairGrid at 0x20fdb42ab88>



From the above two plots, we can see that the one with wine quality does not provide any useful information because there are seven classes for distinction with a lot of overlap. Yet, if need be, variables such as alcohol, free sulphur dioxide can be used as points for distinction

Data Normalization

In the next step, we normalize the data set using z-score normalization and then plot the data with respect to colr and quality respectively.

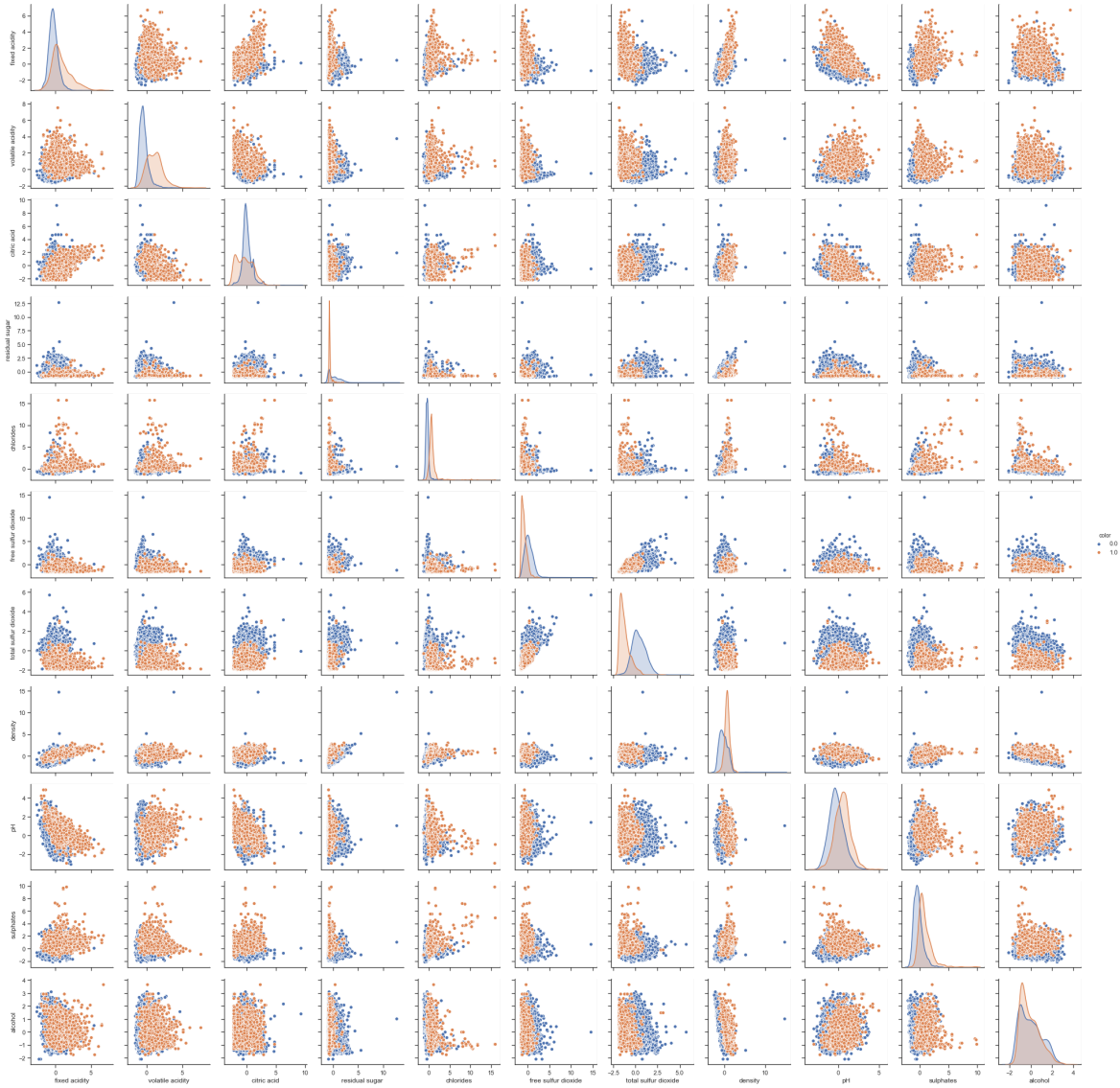
In [6]:
```
from sklearn.preprocessing import StandardScaler
scalar=StandardScaler()
scalar.fit(wine[D])
wine_normalized = scalar.transform(wine[D])
wine_normalized = pd.DataFrame.from_records(wine_normalized,columns = D)
```

In [7]:
```
df=pd.DataFrame(wine_normalized)
x=wine[C].values.tolist()
y=wine[L].values.tolist()
df['color'] = x
df['quality'] = y
```

The data is normalized and the pairplots using color and quality as labels is plotted again to understand the difference between normalized and unnormalized data.

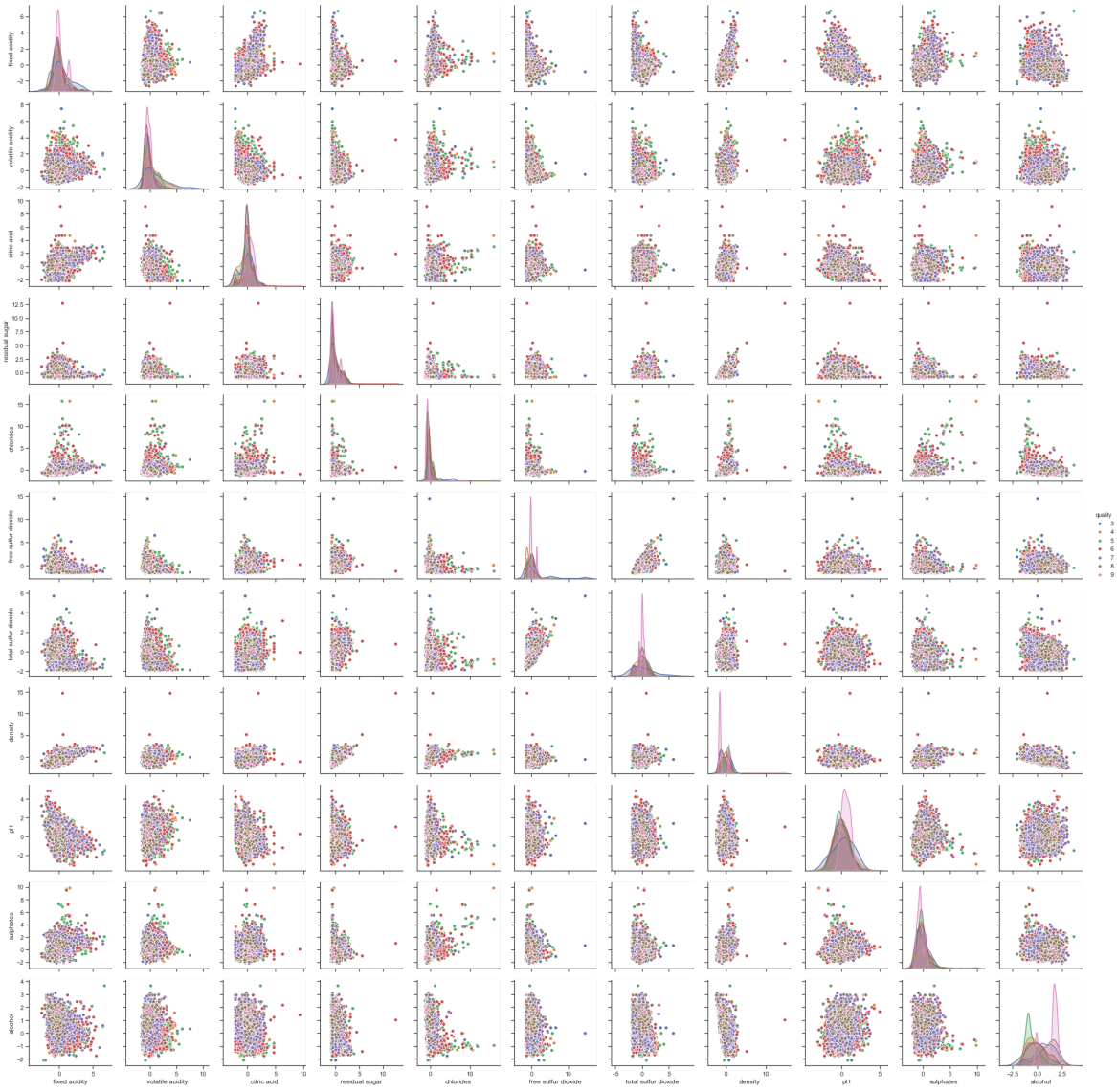
In [8]:
```
sns.pairplot(df,hue="color",vars=D)
```

Out[8]: <seaborn.axisgrid.PairGrid at 0x20fe37b6c08>



In [9]:
```
sns.pairplot(df,hue="quality",vars=D)
```

Out[9]: <seaborn.axisgrid.PairGrid at 0x20fe9f31088>



Upon normalization, we do not observe any change in the distribution, but the scale onto which the data is distributed changes. Therefore, it can be said that the data is standardized but its relative distribution with respect to its neighbours remain the same as a result of which there is not visual difference. Again, pairplot with quality as label does not impart any useful information and hence can be ignored.

Splitting the data into train and test sets

Once the data has been normalized, it is now tested for parameters such as weights and distances.

1. The data is split into train and test sets. The training data is trained on the K nearest neighbour classifier model and the results are tested on the test data.

2. The dataset is tested twice, once with color as the output and the other with quality as the output.
3. Different combination of parameters are used to analyze which set maximizes the accuracy of the model.

In [146]:
```
# classify color of wine with all features
X = df[D].values
y = np.ravel(df[[L]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rand
```

Classification with different weighting schemes

Quality

In the following subset, the dataset is using quality as a label and the outcomes are tested for differnt weight and distance metrics.

In [148]:
```
n_neighborslist = list(range(1,51))
col_names=['uniform', 'manhattan distance', 'euclidean distance','braycurtis'
accarray = np.zeros((len(n_neighborslist),5))

#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)
#case 1 with uniform istribution
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='uniform')
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[0]] = accscore

#case 2 with distance based weight and manhattan distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[1]] = accscore

#case 3 with distance based weight and euclidean distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[2]] = accscore

#case 4 with distance based weight and inf dist
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[3]] = accscore

#case 5 with mahalanobis distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k,algorithm='brute', w
                                           metric='mahalanobis',metric_params
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[4]] = accscore
acc.describe()
acc
```

Out[148]:

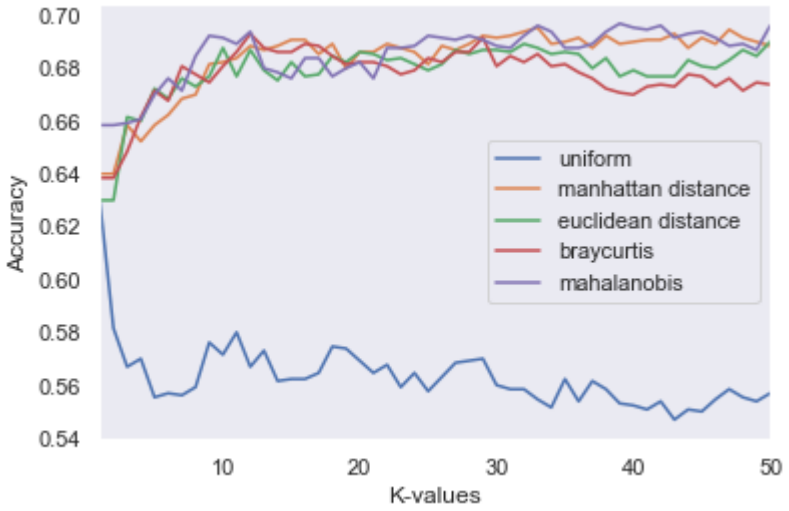
	uniform	manhattan distance	euclidean distance	braycurtis	mahalanobis
0	0.000000	0.000000	0.000000	0.000000	0.000000
1	0.630000	0.640000	0.630000	0.638462	0.658462

	uniform	manhattan distance	euclidean distance	braycurtis	mahalanobis
2	0.581538	0.640000	0.630000	0.638462	0.658462
3	0.566923	0.658462	0.661538	0.648462	0.659231
4	0.570000	0.652308	0.660000	0.661538	0.660769
5	0.555385	0.658462	0.672308	0.671538	0.670000
6	0.556923	0.662308	0.668462	0.667692	0.676154
7	0.556154	0.668462	0.676154	0.680769	0.671538
8	0.559231	0.670000	0.673077	0.677692	0.684615
9	0.576154	0.681538	0.676923	0.674615	0.692308
10	0.571538	0.682308	0.687692	0.680769	0.691538
11	0.580000	0.683846	0.676923	0.686154	0.689231
12	0.566923	0.688462	0.686923	0.693077	0.693846
13	0.573077	0.686923	0.679231	0.687692	0.680000
14	0.561538	0.688462	0.675385	0.686154	0.678462
15	0.562308	0.690769	0.682308	0.686154	0.676154
16	0.562308	0.690769	0.676923	0.689231	0.683846
17	0.564615	0.685385	0.677692	0.688462	0.683846
18	0.574615	0.689231	0.684615	0.684615	0.676923
19	0.573846	0.680000	0.682308	0.681538	0.680000
20	0.569231	0.686154	0.686154	0.682308	0.682308
21	0.564615	0.686154	0.685385	0.682308	0.676154
22	0.567692	0.689231	0.683077	0.680769	0.687692
23	0.559231	0.687692	0.683846	0.677692	0.687692
24	0.564615	0.686154	0.681538	0.679231	0.688462
25	0.557692	0.681538	0.679231	0.683846	0.692308
26	0.563077	0.688462	0.681538	0.682308	0.691538
27	0.568462	0.686923	0.686923	0.686154	0.690769
28	0.569231	0.689231	0.685385	0.686154	0.692308
29	0.570000	0.692308	0.686923	0.691538	0.690769
30	0.560000	0.691538	0.686923	0.680769	0.688462
31	0.558462	0.692308	0.686154	0.684615	0.687692
32	0.558462	0.693846	0.689231	0.682308	0.692308
33	0.554615	0.695385	0.687692	0.685385	0.696154
34	0.551538	0.689231	0.685385	0.680769	0.693846
35	0.562308	0.690000	0.686154	0.681538	0.687692
36	0.553846	0.691538	0.685385	0.678462	0.687692
37	0.561538	0.687692	0.680000	0.676154	0.689231

	uniform	manhattan distance	euclidean distance	braycurtis	mahalanobis
38	0.558462	0.692308	0.683846	0.672308	0.693846
39	0.553077	0.689231	0.676923	0.670769	0.696923
40	0.552308	0.690000	0.679231	0.670000	0.695385
41	0.550769	0.690769	0.676923	0.673077	0.694615
42	0.553846	0.690769	0.676923	0.673846	0.696154
43	0.546923	0.693077	0.676923	0.673077	0.691538
44	0.550769	0.687692	0.683077	0.677692	0.693077
45	0.550000	0.691538	0.680769	0.676923	0.693846
46	0.554615	0.689231	0.680000	0.673077	0.691538
47	0.558462	0.694615	0.683077	0.676154	0.688462
48	0.555385	0.691538	0.686923	0.671538	0.689231
49	0.553846	0.690000	0.684615	0.674615	0.686923
50	0.556923	0.688462	0.690000	0.673846	0.696154

In [149]:
```
r=acc.drop(0)
r.plot()
plt.xlabel('K-values')
plt.grid()
plt.ylabel('Accuracy')
```

Out[149]: Text(0, 0.5, 'Accuracy')



When quality is choosen as the output parameter, we observe that mahalanobis metric performs better than all the other distance and weight metrics. Maximum accuracy was obtained by the manhattan distance metric at k = 12 beyond which there are some fluctuations in the value of accuracy.

Note: All the tests are performed on the normalized data

Color

In [150]:

```
# classify color of wine with all features
X = df[D].values
y = np.ravel(df[[C]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rand
```

In [153]:

```
n_neighborslist = list(range(1,51))
col_names=['uniform', 'manhattan distance', 'euclidean distance','canberra','
accarray = np.zeros((len(n_neighborslist),5))

#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)
#case 1 with uniform istribution
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='uniform')
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[0]] = accscore

#case 2 with distance based weight and manhattan distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[1]] = accscore

#case 3 with distance based weight and euclidean distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[2]] = accscore

#case 4 with distance based weight and inf dist
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[3]] = accscore

#case 5 with mahalanobis distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k,algorithm='brute', w
                                           metric='mahalanobis',metric_params
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[4]] = accscore
acc.describe()
acc
```

Out[153]:

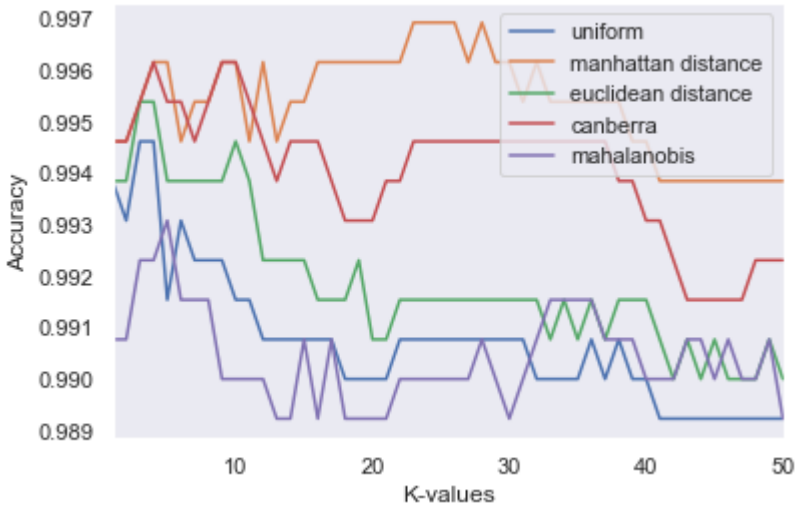
	uniform	manhattan distance	euclidean distance	canberra	mahalanobis
0	0.000000	0.000000	0.000000	0.000000	0.000000
1	0.993846	0.994615	0.993846	0.994615	0.990769
2	0.993077	0.994615	0.993846	0.994615	0.990769

	uniform	manhattan distance	euclidean distance	canberra	mahalanobis
3	0.994615	0.995385	0.995385	0.995385	0.992308
4	0.994615	0.996154	0.995385	0.996154	0.992308
5	0.991538	0.996154	0.993846	0.995385	0.993077
6	0.993077	0.994615	0.993846	0.995385	0.991538
7	0.992308	0.995385	0.993846	0.994615	0.991538
8	0.992308	0.995385	0.993846	0.995385	0.991538
9	0.992308	0.996154	0.993846	0.996154	0.990000
10	0.991538	0.996154	0.994615	0.996154	0.990000
11	0.991538	0.994615	0.993846	0.995385	0.990000
12	0.990769	0.996154	0.992308	0.994615	0.990000
13	0.990769	0.994615	0.992308	0.993846	0.989231
14	0.990769	0.995385	0.992308	0.994615	0.989231
15	0.990769	0.995385	0.992308	0.994615	0.990769
16	0.990769	0.996154	0.991538	0.994615	0.989231
17	0.990769	0.996154	0.991538	0.993846	0.990769
18	0.990000	0.996154	0.991538	0.993077	0.989231
19	0.990000	0.996154	0.992308	0.993077	0.989231
20	0.990000	0.996154	0.990769	0.993077	0.989231
21	0.990000	0.996154	0.990769	0.993846	0.989231
22	0.990769	0.996154	0.991538	0.993846	0.990000
23	0.990769	0.996923	0.991538	0.994615	0.990000
24	0.990769	0.996923	0.991538	0.994615	0.990000
25	0.990769	0.996923	0.991538	0.994615	0.990000
26	0.990769	0.996923	0.991538	0.994615	0.990000
27	0.990769	0.996154	0.991538	0.994615	0.990000
28	0.990769	0.996923	0.991538	0.994615	0.990769
29	0.990769	0.996154	0.991538	0.994615	0.990000
30	0.990769	0.996154	0.991538	0.994615	0.989231
31	0.990769	0.995385	0.991538	0.994615	0.990000
32	0.990000	0.996154	0.991538	0.994615	0.990769
33	0.990000	0.995385	0.990769	0.994615	0.991538
34	0.990000	0.995385	0.991538	0.994615	0.991538
35	0.990000	0.995385	0.990769	0.994615	0.991538
36	0.990769	0.995385	0.991538	0.994615	0.991538
37	0.990000	0.995385	0.990769	0.994615	0.990769
38	0.990769	0.995385	0.991538	0.993846	0.990769

	uniform	manhattan distance	euclidean distance	canberra	mahalanobis
39	0.990000	0.994615	0.991538	0.993846	0.990769
40	0.990000	0.994615	0.991538	0.993077	0.990000
41	0.989231	0.993846	0.990769	0.993077	0.990000
42	0.989231	0.993846	0.990000	0.992308	0.990000
43	0.989231	0.993846	0.990769	0.991538	0.990769
44	0.989231	0.993846	0.990000	0.991538	0.990769
45	0.989231	0.993846	0.990769	0.991538	0.990000
46	0.989231	0.993846	0.990000	0.991538	0.990769
47	0.989231	0.993846	0.990000	0.991538	0.990000
48	0.989231	0.993846	0.990000	0.992308	0.990000
49	0.989231	0.993846	0.990769	0.992308	0.990769
50	0.989231	0.993846	0.990000	0.992308	0.989231

In [154]:
```
r=acc.drop(0)
r.plot()
plt.xlabel('K-values')
plt.grid()
plt.ylabel('Accuracy')
```

Out[154]: Text(0, 0.5, 'Accuracy')



When color is choosen as the output parameter, we observe that manhattan metric performs better than all the other distance and weight metrics. Maximum accuracy was obtained by the manhattan distance metric at k = 23

Note: All the tests are performed on the normalized data

Feature Selection

Out of all the 11 features available, there are a few features that affect the performance of the

model. To increase the accuracy, we select a set of features that maximize the accuracy of the model. To do so, we have implemented, 3 methods as shown below.

1. SelectKBest
2. feature_importances from ExtraTreesClassifier
3. Heatmap for the correlation

The results produced were tested on the model to see which ones maximize the accuracy.

SelectKBest is used for finding the best parameters using color and quality as target variables respectively

Color

In [108]:
```
from sklearn.feature_selection import mutual_info_classif
from sklearn.feature_selection import SelectKBest
X = df[D]  #independent columns
y = df[C]    #target column
bestfeatures = SelectKBest(score_func=mutual_info_classif)
fit = bestfeatures.fit(X,y)
dfscores = pd.DataFrame(fit.scores_)
dfcolumns = pd.DataFrame(X.columns)
#concat two dataframes for better visualization
featureScores = pd.concat([dfcolumns,dfscores],axis=1)
featureScores.columns = ['Specs','Score']  #naming the dataframe columns
print(featureScores.nlargest(11,'Score'))
#https://machinelearningmastery.com/feature-selection-machine-learning-pythor
```

```
                  Specs     Score
4             chlorides  0.351686
6   total sulfur dioxide  0.351131
1       volatile acidity  0.237107
3         residual sugar  0.208091
5    free sulfur dioxide  0.158162
7                density  0.150197
9              sulphates  0.139719
0          fixed acidity  0.130465
2            citric acid  0.111888
8                     pH  0.060464
10              alcohol  0.026363
```

Quality

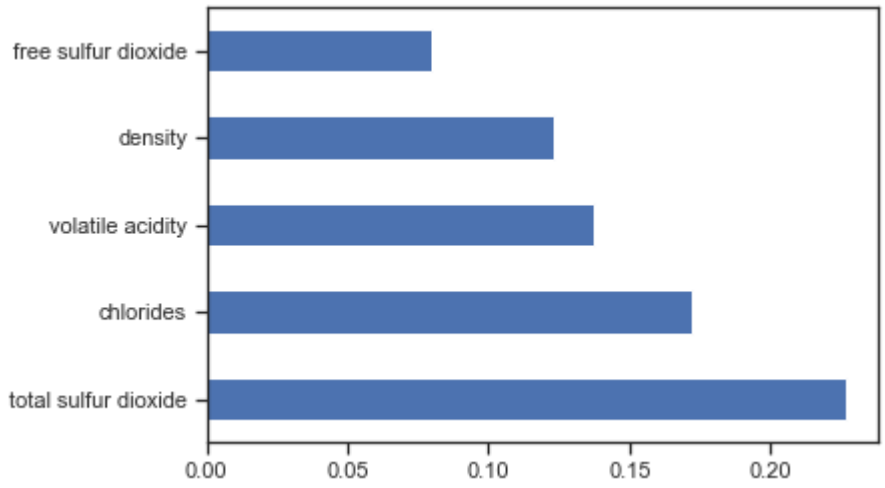
In [109]:
```
from sklearn.feature_selection import mutual_info_classif
from sklearn.feature_selection import SelectKBest
X = df[D]  #independent columns
y = df[L]    #target column
bestfeatures = SelectKBest(score_func=mutual_info_classif)
fit = bestfeatures.fit(X,y)
dfscores = pd.DataFrame(fit.scores_)
dfcolumns = pd.DataFrame(X.columns)
#concat two dataframes for better visualization
featureScores = pd.concat([dfcolumns,dfscores],axis=1)
featureScores.columns = ['Specs','Score']  #naming the dataframe columns
print(featureScores.nlargest(11,'Score'))  #print 5 best features
```

```
                  Specs     Score
7               density  0.156803
10              alcohol  0.156582
4             chlorides  0.070062
6   total sulfur dioxide  0.068807
3         residual sugar  0.066848
2            citric acid  0.057211
1       volatile acidity  0.055835
5    free sulfur dioxide  0.035079
9              sulphates  0.025753
0          fixed acidity  0.022164
8                     pH  0.012822
```

Color

In [110]:
```
from sklearn.ensemble import ExtraTreesClassifier
import matplotlib.pyplot as plt
X = df[D]  #independent columns
y = df[C]    #target column
model = ExtraTreesClassifier()
model.fit(X,y)
print(model.feature_importances_) #use inbuilt class feature_importances of t
#plot graph of feature importances for better visualization
feat_importances = pd.Series(model.feature_importances_, index=X.columns)
feat_importances.nlargest(5).plot(kind='barh')
plt.show()
#https://machinelearningmastery.com/feature-selection-machine-learning-pythor
```

```
C:\Users\hp\Anaconda3\lib\site-packages\sklearn\ensemble\forest.py:245: Fut
ureWarning: The default value of n_estimators will change from 10 in versio
n 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)

[0.06048762 0.13704439 0.03100076 0.05837763 0.17252491 0.07975415
 0.22657226 0.12298942 0.04155226 0.05131578 0.01838082]
```

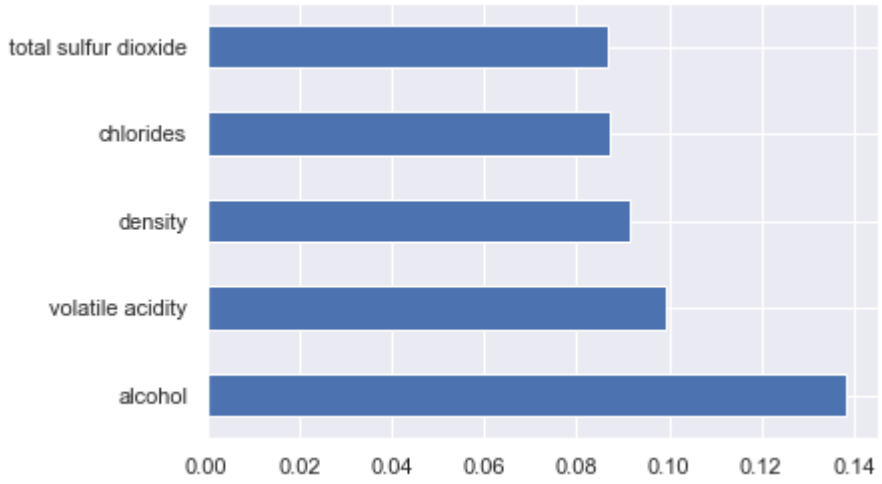


Quality

In [155]:

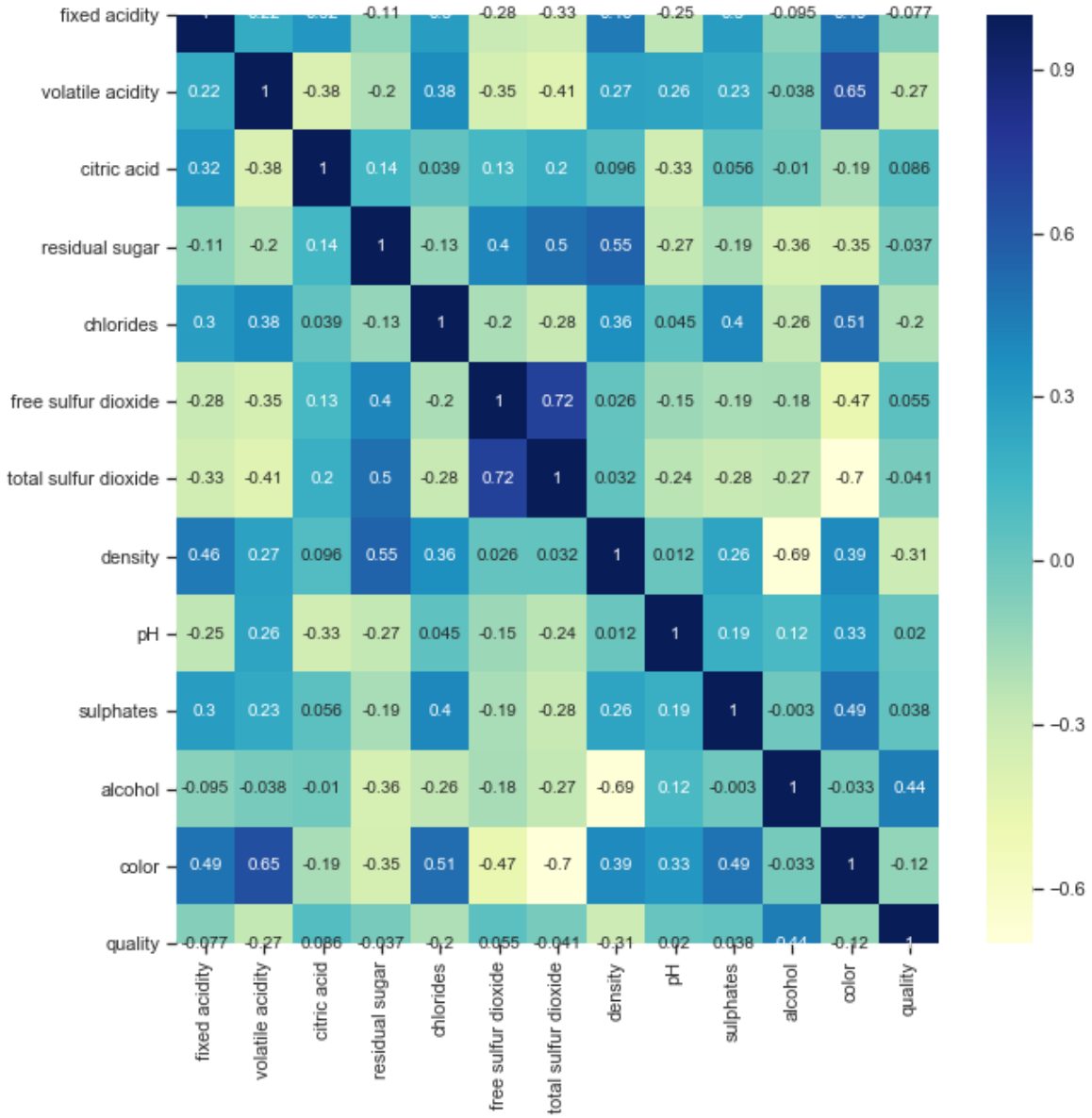
```
from sklearn.ensemble import ExtraTreesClassifier
import matplotlib.pyplot as plt
X = df[D]  #independent columns
y = df[L]    #target column i.e price range
model = ExtraTreesClassifier()
model.fit(X,y)
print(model.feature_importances_) #use inbuilt class feature_importances of t
#plot graph of feature importances for better visualization
feat_importances = pd.Series(model.feature_importances_, index=X.columns)
feat_importances.nlargest(5).plot(kind='barh')
plt.show()
#https://machinelearningmastery.com/feature-selection-machine-learning-pythor
```

```
C:\Users\hp\Anaconda3\lib\site-packages\sklearn\ensemble\forest.py:245: Fut
ureWarning: The default value of n_estimators will change from 10 in versio
n 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)

[0.07968941 0.09933058 0.08316521 0.08306827 0.08714547 0.08528123
 0.08662047 0.09152929 0.0817849  0.08425492 0.13813025]
```



In [112]:
```
corr=df.corr()
cor=corr.index
plt.figure(figsize=(10,10))
sns.heatmap(df[cor].corr(), annot=True,cmap="YlGnBu")
#https://towardsdatascience.com/feature-selection-techniques-in-machine-learr
```

Out[112]: <matplotlib.axes._subplots.AxesSubplot at 0x20f819dde48>



Subset of 4 features, target as color

In [113]:

```
X = df[['chlorides','density','total sulfur dioxide','residual sugar']].value
y = np.ravel(df[[C]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rand
```

In [114]:
```
n_neighborslist = list(range(1,51))
col_names=['uniform', 'manhattan distance', 'euclidean distance','infinte dis
accarray = np.zeros((len(n_neighborslist),5))

#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)
#case 1 with uniform istribution
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='uniform')
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[0]] = accscore

#case 2 with distance based weight and manhattan distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[1]] = accscore

#case 3 with distance based weight and euclidean distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[2]] = accscore

#case 4 with distance based weight and inf dist
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[3]] = accscore

#case 5 with mahalanobis distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k,algorithm='brute', w
                                           metric='mahalanobis',metric_params
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[4]] = accscore
acc.describe()
acc
```

Out[114]:

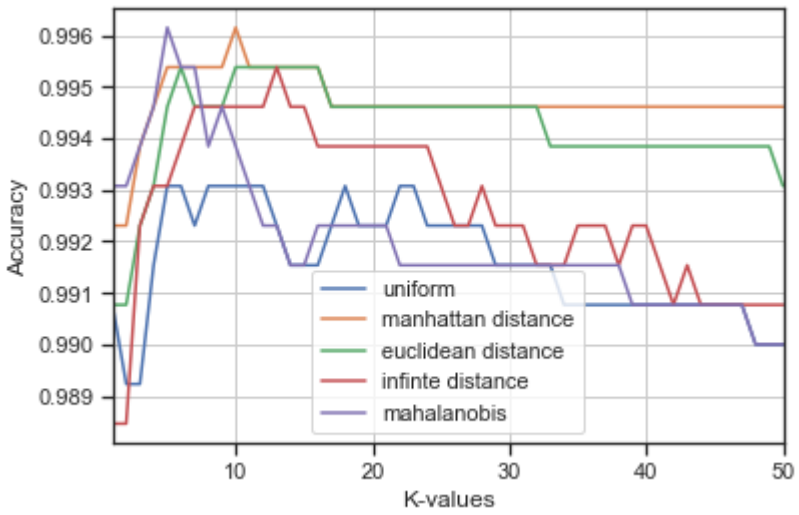
	uniform	manhattan distance	euclidean distance	infinte distance	mahalanobis
0	0.000000	0.000000	0.000000	0.000000	0.000000
1	0.990769	0.992308	0.990769	0.988462	0.993077

	uniform	manhattan distance	euclidean distance	infinte distance	mahalanobis
2	0.989231	0.992308	0.990769	0.988462	0.993077
3	0.989231	0.993846	0.992308	0.992308	0.993846
4	0.991538	0.994615	0.993077	0.993077	0.994615
5	0.993077	0.995385	0.994615	0.993077	0.996154
6	0.993077	0.995385	0.995385	0.993846	0.995385
7	0.992308	0.995385	0.994615	0.994615	0.995385
8	0.993077	0.995385	0.994615	0.994615	0.993846
9	0.993077	0.995385	0.994615	0.994615	0.994615
10	0.993077	0.996154	0.995385	0.994615	0.993846
11	0.993077	0.995385	0.995385	0.994615	0.993077
12	0.993077	0.995385	0.995385	0.994615	0.992308
13	0.992308	0.995385	0.995385	0.995385	0.992308
14	0.991538	0.995385	0.995385	0.994615	0.991538
15	0.991538	0.995385	0.995385	0.994615	0.991538
16	0.991538	0.995385	0.995385	0.993846	0.992308
17	0.992308	0.994615	0.994615	0.993846	0.992308
18	0.993077	0.994615	0.994615	0.993846	0.992308
19	0.992308	0.994615	0.994615	0.993846	0.992308
20	0.992308	0.994615	0.994615	0.993846	0.992308
21	0.992308	0.994615	0.994615	0.993846	0.992308
22	0.993077	0.994615	0.994615	0.993846	0.991538
23	0.993077	0.994615	0.994615	0.993846	0.991538
24	0.992308	0.994615	0.994615	0.993846	0.991538
25	0.992308	0.994615	0.994615	0.993077	0.991538
26	0.992308	0.994615	0.994615	0.992308	0.991538
27	0.992308	0.994615	0.994615	0.992308	0.991538
28	0.992308	0.994615	0.994615	0.993077	0.991538
29	0.991538	0.994615	0.994615	0.992308	0.991538
30	0.991538	0.994615	0.994615	0.992308	0.991538
31	0.991538	0.994615	0.994615	0.992308	0.991538
32	0.991538	0.994615	0.994615	0.991538	0.991538
33	0.991538	0.994615	0.993846	0.991538	0.991538
34	0.990769	0.994615	0.993846	0.991538	0.991538
35	0.990769	0.994615	0.993846	0.992308	0.991538
36	0.990769	0.994615	0.993846	0.992308	0.991538
37	0.990769	0.994615	0.993846	0.992308	0.991538

	uniform	manhattan distance	euclidean distance	infinte distance	mahalanobis
38	0.990769	0.994615	0.993846	0.991538	0.991538
39	0.990769	0.994615	0.993846	0.992308	0.990769
40	0.990769	0.994615	0.993846	0.992308	0.990769
41	0.990769	0.994615	0.993846	0.991538	0.990769
42	0.990769	0.994615	0.993846	0.990769	0.990769
43	0.990769	0.994615	0.993846	0.991538	0.990769
44	0.990769	0.994615	0.993846	0.990769	0.990769
45	0.990769	0.994615	0.993846	0.990769	0.990769
46	0.990769	0.994615	0.993846	0.990769	0.990769
47	0.990769	0.994615	0.993846	0.990769	0.990769
48	0.990000	0.994615	0.993846	0.990769	0.990000
49	0.990000	0.994615	0.993846	0.990769	0.990000
50	0.990000	0.994615	0.993077	0.990769	0.990000

In [115]:
```
r=acc.drop(0)
r.plot()
plt.xlabel('K-values')
plt.grid()
plt.ylabel('Accuracy')
```

Out[115]: Text(0, 0.5, 'Accuracy')



With the smaller set of features, we observe an increase in the overall accuracy of the model for all k values. Also, we observe that the maximum accuarcy is obtained at a smaller k value. When all the features were taken into consideration, the maximum accurracywas obtained at k=23 with manhattan distance. But in this case, we observe that mahalanobis distance can be used to obtained the maximum accuaracy the model at k=5. This saves the computation time and complexity of the model. Manhattan also produces the same accuarcy but at k=10.

One observation is that the maximum accuracy for all features was higher by 0.0007 compared to our result. But to obtain that accuarcy, the model had to compare 23 neighbours which made the model complex. But the overall accuracy of the current model is higher and produces better results for any k value when compared to manhattan distance in the previous case.

Subset of 4 features, target as quality

In [219]:
```
X = df[['chlorides','density','alcohol','residual sugar']].values
y = np.ravel(df[[L]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ranc
```

In [220]:
```
n_neighborslist = list(range(1,51))
col_names=['uniform', 'manhattan distance', 'euclidean distance','infinte dis
accarray = np.zeros((len(n_neighborslist),5))

#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)
#case 1 with uniform istribution
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='uniform')
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[0]] = accscore

#case 2 with distance based weight and manhattan distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[1]] = accscore

#case 3 with distance based weight and euclidean distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[2]] = accscore

#case 4 with distance based weight and inf dist
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[3]] = accscore

#case 5 with mahalanobis distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k,algorithm='brute', w
                                           metric='mahalanobis',metric_params
    neigh.fit(X_train, y_train)
    y_pred = neigh.predict(X_test)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[4]] = accscore
acc.describe()
acc
```

Out[220]:

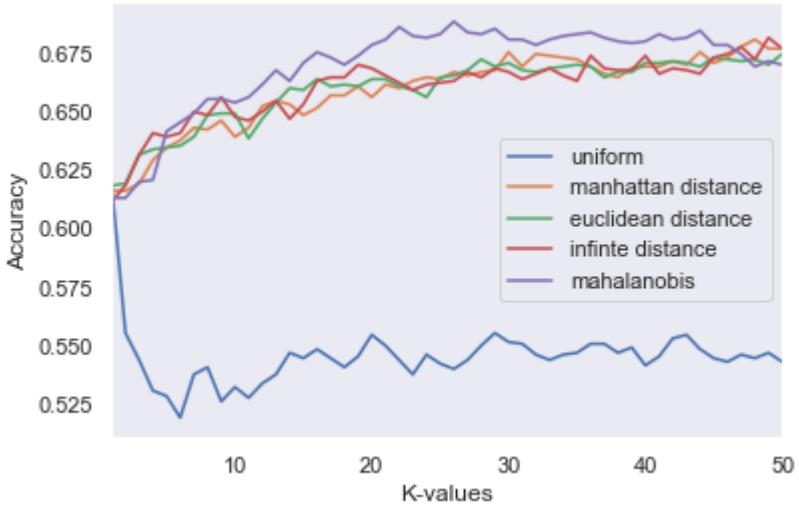
	uniform	manhattan distance	euclidean distance	infinte distance	mahalanobis
0	0.000000	0.000000	0.000000	0.000000	0.000000
1	0.618462	0.616154	0.618462	0.611538	0.613077

	uniform	manhattan distance	euclidean distance	infinte distance	mahalanobis
2	0.555385	0.616154	0.619231	0.618462	0.613077
3	0.543846	0.619231	0.631538	0.631538	0.620000
4	0.530769	0.629231	0.633846	0.640769	0.620769
5	0.528462	0.634615	0.634615	0.639231	0.641538
6	0.519231	0.637692	0.635385	0.640769	0.645385
7	0.537692	0.643077	0.639231	0.650000	0.649231
8	0.540769	0.642308	0.648462	0.648462	0.655385
9	0.526154	0.646154	0.649231	0.656154	0.655385
10	0.532308	0.639231	0.649231	0.647692	0.653846
11	0.527692	0.643077	0.638462	0.646154	0.656154
12	0.533846	0.652308	0.646923	0.650000	0.661538
13	0.537692	0.654615	0.653846	0.654615	0.667692
14	0.546923	0.653077	0.660000	0.646923	0.663077
15	0.544615	0.648462	0.659231	0.653077	0.670769
16	0.548462	0.651538	0.663846	0.663077	0.675385
17	0.544615	0.656923	0.660769	0.664615	0.673077
18	0.540769	0.656923	0.661538	0.664615	0.670000
19	0.545385	0.660769	0.660769	0.670000	0.673846
20	0.554615	0.656154	0.663846	0.668462	0.678462
21	0.550000	0.661538	0.663846	0.665385	0.680769
22	0.543846	0.660000	0.660769	0.662308	0.686154
23	0.537692	0.663077	0.659231	0.659231	0.682308
24	0.546154	0.664615	0.656154	0.661538	0.681538
25	0.542308	0.663846	0.664615	0.662308	0.683077
26	0.540000	0.666923	0.665385	0.663077	0.688462
27	0.543846	0.665385	0.667692	0.666923	0.683846
28	0.550000	0.666923	0.672308	0.664615	0.683077
29	0.555385	0.667692	0.669231	0.668462	0.685385
30	0.551538	0.675385	0.670769	0.666923	0.680769
31	0.550769	0.669231	0.667692	0.663846	0.680769
32	0.546154	0.674615	0.666923	0.666154	0.678462
33	0.543846	0.673846	0.668462	0.668462	0.680769
34	0.546154	0.673077	0.669231	0.665385	0.682308
35	0.546923	0.672308	0.670000	0.663077	0.683077
36	0.550769	0.669231	0.669231	0.673846	0.683846
37	0.550769	0.666154	0.664615	0.668462	0.681538

	uniform	manhattan distance	euclidean distance	infinte distance	mahalanobis
38	0.546923	0.664615	0.666923	0.667692	0.680000
39	0.549231	0.668462	0.666923	0.667692	0.679231
40	0.541538	0.669231	0.670769	0.673846	0.680000
41	0.545385	0.669231	0.670769	0.666154	0.683077
42	0.553077	0.671538	0.671538	0.668462	0.680769
43	0.554615	0.670000	0.670769	0.667692	0.681538
44	0.548462	0.675385	0.669231	0.666154	0.684615
45	0.544615	0.670769	0.673077	0.673077	0.678462
46	0.543077	0.673077	0.672308	0.674615	0.678462
47	0.546154	0.677692	0.671538	0.677692	0.674615
48	0.544615	0.680769	0.672308	0.672308	0.669231
49	0.546923	0.676923	0.670000	0.681538	0.671538
50	0.543077	0.676923	0.674615	0.676923	0.670000

In [221]:
```
r=acc.drop(0)
r.plot()
plt.xlabel('K-values')
plt.grid()
plt.ylabel('Accuracy')
```

Out[221]: Text(0, 0.5, 'Accuracy')



With the smaller set of features, we observe a decrease in the overall accuracy of the model for all k values which indicates the dependency of quality data on more than four variables. Yet we were able to find a subset of four that maximizes the accuracy. Also, we observe that the maximum accuarcy is obtained at a smaller k value. When all the features were taken into consideration, the maximum accurracy was obtained at k=39 with mahalanobis metric. But in this case, we observe that mahalanobis distance can be used to obtained the maximum accuaracy the model at k=26. This saves the computation time and complexity of the model.

One observation is that the maximum accuracy for all features result was higher by ~0.0085 compared to our result. But to obtain that accuarcy, the model had to compare 39 neighbours which made the model complex. The overall accuracy remains low when compared to results with all features, which indicates the dependency of accuracy on more than four features.

(Based on the subset of four features obtained for the feature selection tests, each of the others were added along to see which one increases the accuracy further. None of the features individually increases the accuracy which indicates that the features are interdependent)

PCA with color as outcome, all features

In [119]:
```
#PCA
from sklearn.decomposition import PCA

# classify color of wine with all features
X = df[D].values
y = np.ravel(df[[C]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rand

pca = PCA()
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)
```

In [120]:
```
n_neighborslist = list(range(1,51))
col_names=['uniform', 'manhattan distance', 'euclidean distance','infinte','m
accarray = np.zeros((len(n_neighborslist),5))

#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)
#case 1 with uniform istribution
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='uniform')
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[0]] = accscore

#case 2 with distance based weight and manhattan distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[1]] = accscore

#case 3 with distance based weight and euclidean distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[2]] = accscore

#case 4 with distance based weight and inf dist
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[3]] = accscore

#case 5 with mahalanobis distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k,algorithm='brute', w
                                           metric='mahalanobis',metric_params
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[4]] = accscore
acc.describe()
acc
```

Out[120]:

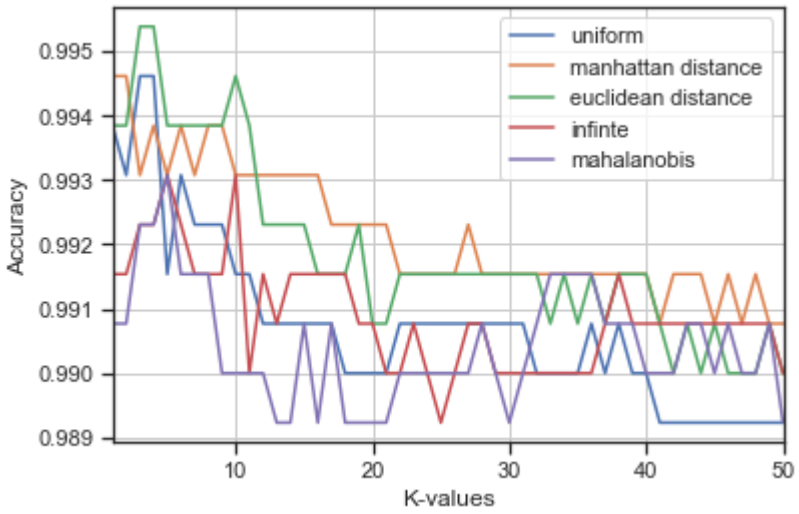
	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
0	0.000000	0.000000	0.000000	0.000000	0.000000
1	0.993846	0.994615	0.993846	0.991538	0.990769
2	0.993077	0.994615	0.993846	0.991538	0.990769

	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
3	0.994615	0.993077	0.995385	0.992308	0.992308
4	0.994615	0.993846	0.995385	0.992308	0.992308
5	0.991538	0.993077	0.993846	0.993077	0.993077
6	0.993077	0.993846	0.993846	0.992308	0.991538
7	0.992308	0.993077	0.993846	0.991538	0.991538
8	0.992308	0.993846	0.993846	0.991538	0.991538
9	0.992308	0.993846	0.993846	0.991538	0.990000
10	0.991538	0.993077	0.994615	0.993077	0.990000
11	0.991538	0.993077	0.993846	0.990000	0.990000
12	0.990769	0.993077	0.992308	0.991538	0.990000
13	0.990769	0.993077	0.992308	0.990769	0.989231
14	0.990769	0.993077	0.992308	0.991538	0.989231
15	0.990769	0.993077	0.992308	0.991538	0.990769
16	0.990769	0.993077	0.991538	0.991538	0.989231
17	0.990769	0.992308	0.991538	0.991538	0.990769
18	0.990000	0.992308	0.991538	0.991538	0.989231
19	0.990000	0.992308	0.992308	0.990769	0.989231
20	0.990000	0.992308	0.990769	0.990769	0.989231
21	0.990000	0.992308	0.990769	0.990000	0.989231
22	0.990769	0.991538	0.991538	0.990000	0.990000
23	0.990769	0.991538	0.991538	0.990769	0.990000
24	0.990769	0.991538	0.991538	0.990000	0.990000
25	0.990769	0.991538	0.991538	0.989231	0.990000
26	0.990769	0.991538	0.991538	0.990000	0.990000
27	0.990769	0.992308	0.991538	0.990769	0.990000
28	0.990769	0.991538	0.991538	0.990769	0.990769
29	0.990769	0.991538	0.991538	0.990000	0.990000
30	0.990769	0.991538	0.991538	0.990000	0.989231
31	0.990769	0.991538	0.991538	0.990000	0.990000
32	0.990000	0.991538	0.991538	0.990000	0.990769
33	0.990000	0.991538	0.990769	0.990000	0.991538
34	0.990000	0.991538	0.991538	0.990000	0.991538
35	0.990000	0.991538	0.990769	0.990000	0.991538
36	0.990769	0.991538	0.991538	0.990000	0.991538
37	0.990000	0.991538	0.990769	0.990769	0.990769
38	0.990769	0.991538	0.991538	0.991538	0.990769

	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
39	0.990000	0.991538	0.991538	0.990769	0.990769
40	0.990000	0.991538	0.991538	0.990769	0.990000
41	0.989231	0.990769	0.990769	0.990769	0.990000
42	0.989231	0.991538	0.990000	0.990769	0.990000
43	0.989231	0.991538	0.990769	0.990769	0.990769
44	0.989231	0.991538	0.990000	0.990769	0.990769
45	0.989231	0.990769	0.990769	0.990769	0.990000
46	0.989231	0.991538	0.990000	0.990769	0.990769
47	0.989231	0.990769	0.990000	0.990769	0.990000
48	0.989231	0.991538	0.990000	0.990769	0.990000
49	0.989231	0.990769	0.990769	0.990769	0.990769
50	0.989231	0.990769	0.990000	0.990000	0.989231

In [121]:
```
r=acc.drop(0)
r.plot()
plt.xlabel('K-values')
plt.grid()
plt.ylabel('Accuracy')
```

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



In this case, we observe that euclidean performs the best when compared to the other metrics. Also, it only required k=3 to reach its maximum accuracy. This reduces the computation time and less number of comparisions would be required to reach maximum accuracy.

When compared to the results before

1. For Euclidean distance, the maximum accuracy obtained with normalized data containing all features is the same as the accuracy obtained with pca containing all data.
2. The maximum accuracy was observed for manhattan distance in the previous cases which seemed to have dropped for this model.

3. We observe a gradual drop in the accuracies as the k value increases. Since each PCA component contains information corresponding to all the features, therefore, first few components lead to maximum accuracy of the model.

Dimensionality reduction using PCA led to a drop in the accuracy of the model.This may be due to loss of some information at the time of tranformation. But the drop in value is by a very small number whose significance would be determined when a larger data set is put to test. Verifying it with cross validation set can improvise the performance of the model.

PCA with 5 components

In [194]:
```
from sklearn.decomposition import PCA

# classify color of wine with all features
X = df[D].values
y = np.ravel(df[[C]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rand

pca = PCA(n_components=5)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)
```

In [195]:
```
n_neighborslist = list(range(1,51))
col_names=['uniform', 'manhattan distance', 'euclidean distance','infinte','m
accarray = np.zeros((len(n_neighborslist),5))

#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)
#case 1 with uniform distribution
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='uniform')
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[0]] = accscore

#case 2 with distance based weight and manhattan distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[1]] = accscore

#case 3 with distance based weight and euclidean distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[2]] = accscore

#case 4 with distance based weight and inf dist
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[3]] = accscore

#case 5 with mahalanobis distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k,algorithm='brute', w
                                           metric='mahalanobis',metric_params
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[4]] = accscore
acc.describe()
acc
```

Out[195]:

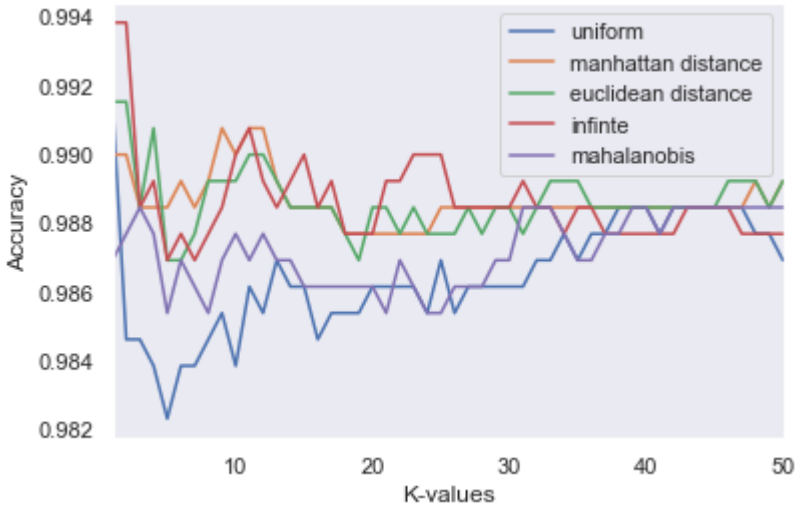
	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
0	0.000000	0.000000	0.000000	0.000000	0.000000
1	0.991538	0.990000	0.991538	0.993846	0.986923
2	0.984615	0.990000	0.991538	0.993846	0.987692

	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
3	0.984615	0.988462	0.988462	0.988462	0.988462
4	0.983846	0.988462	0.990769	0.989231	0.987692
5	0.982308	0.988462	0.986923	0.986923	0.985385
6	0.983846	0.989231	0.986923	0.987692	0.986923
7	0.983846	0.988462	0.987692	0.986923	0.986154
8	0.984615	0.989231	0.989231	0.987692	0.985385
9	0.985385	0.990769	0.989231	0.988462	0.986923
10	0.983846	0.990000	0.989231	0.990000	0.987692
11	0.986154	0.990769	0.990000	0.990769	0.986923
12	0.985385	0.990769	0.990000	0.989231	0.987692
13	0.986923	0.989231	0.989231	0.988462	0.986923
14	0.986154	0.988462	0.988462	0.989231	0.986923
15	0.986154	0.988462	0.988462	0.990000	0.986154
16	0.984615	0.988462	0.988462	0.988462	0.986154
17	0.985385	0.988462	0.988462	0.989231	0.986154
18	0.985385	0.987692	0.987692	0.987692	0.986154
19	0.985385	0.987692	0.986923	0.987692	0.986154
20	0.986154	0.987692	0.988462	0.987692	0.986154
21	0.986154	0.987692	0.988462	0.989231	0.985385
22	0.986154	0.987692	0.987692	0.989231	0.986923
23	0.986154	0.987692	0.988462	0.990000	0.986154
24	0.985385	0.987692	0.987692	0.990000	0.985385
25	0.986923	0.988462	0.987692	0.990000	0.985385
26	0.985385	0.988462	0.987692	0.988462	0.986154
27	0.986154	0.988462	0.988462	0.988462	0.986154
28	0.986154	0.988462	0.987692	0.988462	0.986154
29	0.986154	0.988462	0.988462	0.988462	0.986923
30	0.986154	0.988462	0.988462	0.988462	0.986923
31	0.986154	0.988462	0.987692	0.989231	0.988462
32	0.986923	0.988462	0.988462	0.988462	0.988462
33	0.986923	0.988462	0.989231	0.988462	0.988462
34	0.987692	0.988462	0.989231	0.987692	0.987692
35	0.986923	0.988462	0.989231	0.988462	0.986923
36	0.987692	0.988462	0.988462	0.988462	0.986923
37	0.987692	0.988462	0.988462	0.987692	0.987692
38	0.988462	0.988462	0.988462	0.987692	0.987692

	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
39	0.988462	0.988462	0.988462	0.987692	0.988462
40	0.988462	0.988462	0.988462	0.987692	0.988462
41	0.987692	0.988462	0.988462	0.987692	0.987692
42	0.988462	0.988462	0.988462	0.987692	0.988462
43	0.988462	0.988462	0.988462	0.988462	0.988462
44	0.988462	0.988462	0.988462	0.988462	0.988462
45	0.988462	0.988462	0.988462	0.988462	0.988462
46	0.988462	0.988462	0.989231	0.988462	0.988462
47	0.988462	0.988462	0.989231	0.987692	0.988462
48	0.987692	0.989231	0.989231	0.987692	0.988462
49	0.987692	0.988462	0.988462	0.987692	0.988462
50	0.986923	0.989231	0.989231	0.987692	0.988462

In [196]:
```
r=acc.drop(0)
r.plot()
plt.xlabel('K-values')
plt.grid()
plt.ylabel('Accuracy')
```

Out[196]: Text(0, 0.5, 'Accuracy')



When the number of components were reduced to 5, we observe that as the distance increases(p value in the p-norm), the accuracy of the model increases as well.The maximum accuracy obtained is for p=infinite. The overall accuracy of the model is low when compared to the results obtained when all pca components were used. But using only 5 components we were able to obtain accuracy ~95% of the previous case. This leads to reducing the complexity and saves time.

PCA with color as outcome, selected features

In [222]:

```
from sklearn.decomposition import PCA

# classify color of wine with all features
X = df[['chlorides','total sulfur dioxide','residual sugar','density']].value
y = np.ravel(df[[C]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rand

pca = PCA()
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)
```

In [223]:

```
n_neighborslist = list(range(1,51))
col_names=['uniform', 'manhattan distance', 'euclidean distance','infinte','m
accarray = np.zeros((len(n_neighborslist),5))

#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)
#case 1 with uniform distribution
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='uniform')
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[0]] = accscore

#case 2 with distance based weight and manhattan distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[1]] = accscore

#case 3 with distance based weight and euclidean distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[2]] = accscore

#case 4 with distance based weight and inf dist
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[3]] = accscore

#case 5 with mahalanobis distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k,algorithm='brute', w
                                           metric='mahalanobis',metric_params
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[4]] = accscore
acc.describe()
acc
```

Out[223]:

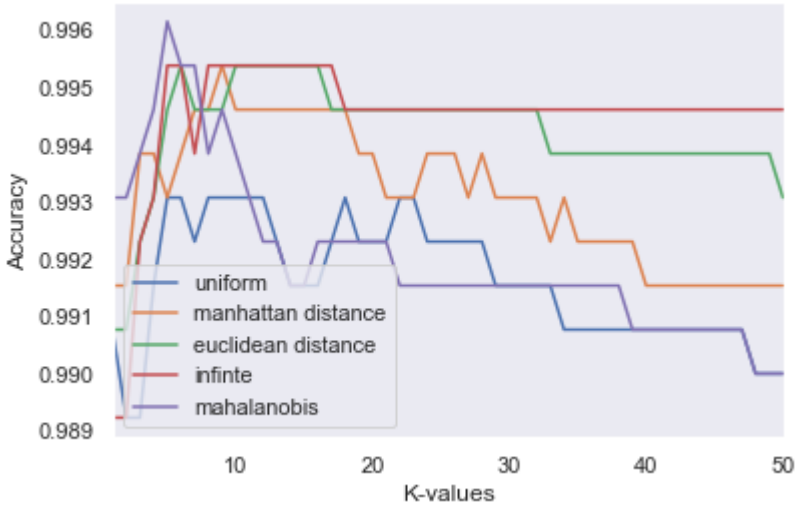
	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
0	0.000000	0.000000	0.000000	0.000000	0.000000
1	0.990769	0.991538	0.990769	0.989231	0.993077

	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
2	0.989231	0.991538	0.990769	0.989231	0.993077
3	0.989231	0.993846	0.992308	0.992308	0.993846
4	0.991538	0.993846	0.993077	0.993077	0.994615
5	0.993077	0.993077	0.994615	0.995385	0.996154
6	0.993077	0.993846	0.995385	0.995385	0.995385
7	0.992308	0.994615	0.994615	0.993846	0.995385
8	0.993077	0.994615	0.994615	0.995385	0.993846
9	0.993077	0.995385	0.994615	0.995385	0.994615
10	0.993077	0.994615	0.995385	0.995385	0.993846
11	0.993077	0.994615	0.995385	0.995385	0.993077
12	0.993077	0.994615	0.995385	0.995385	0.992308
13	0.992308	0.994615	0.995385	0.995385	0.992308
14	0.991538	0.994615	0.995385	0.995385	0.991538
15	0.991538	0.994615	0.995385	0.995385	0.991538
16	0.991538	0.994615	0.995385	0.995385	0.992308
17	0.992308	0.994615	0.994615	0.995385	0.992308
18	0.993077	0.994615	0.994615	0.994615	0.992308
19	0.992308	0.993846	0.994615	0.994615	0.992308
20	0.992308	0.993846	0.994615	0.994615	0.992308
21	0.992308	0.993077	0.994615	0.994615	0.992308
22	0.993077	0.993077	0.994615	0.994615	0.991538
23	0.993077	0.993077	0.994615	0.994615	0.991538
24	0.992308	0.993846	0.994615	0.994615	0.991538
25	0.992308	0.993846	0.994615	0.994615	0.991538
26	0.992308	0.993846	0.994615	0.994615	0.991538
27	0.992308	0.993077	0.994615	0.994615	0.991538
28	0.992308	0.993846	0.994615	0.994615	0.991538
29	0.991538	0.993077	0.994615	0.994615	0.991538
30	0.991538	0.993077	0.994615	0.994615	0.991538
31	0.991538	0.993077	0.994615	0.994615	0.991538
32	0.991538	0.993077	0.994615	0.994615	0.991538
33	0.991538	0.992308	0.993846	0.994615	0.991538
34	0.990769	0.993077	0.993846	0.994615	0.991538
35	0.990769	0.992308	0.993846	0.994615	0.991538
36	0.990769	0.992308	0.993846	0.994615	0.991538
37	0.990769	0.992308	0.993846	0.994615	0.991538

	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
38	0.990769	0.992308	0.993846	0.994615	0.991538
39	0.990769	0.992308	0.993846	0.994615	0.990769
40	0.990769	0.991538	0.993846	0.994615	0.990769
41	0.990769	0.991538	0.993846	0.994615	0.990769
42	0.990769	0.991538	0.993846	0.994615	0.990769
43	0.990769	0.991538	0.993846	0.994615	0.990769
44	0.990769	0.991538	0.993846	0.994615	0.990769
45	0.990769	0.991538	0.993846	0.994615	0.990769
46	0.990769	0.991538	0.993846	0.994615	0.990769
47	0.990769	0.991538	0.993846	0.994615	0.990769
48	0.990000	0.991538	0.993846	0.994615	0.990000
49	0.990000	0.991538	0.993846	0.994615	0.990000
50	0.990000	0.991538	0.993077	0.994615	0.990000

In [224]:
```
r=acc.drop(0)
r.plot()
plt.xlabel('K-values')
plt.grid()
plt.ylabel('Accuracy')
```

Out[224]: Text(0, 0.5, 'Accuracy')



PCA with quality as outcome, all features

In [125]:

```
#total sulphur dioxide, volatile acidity, chlorides, fixed acidity - best fou
#PCA
from sklearn.decomposition import PCA

# classify color of wine with all features
X = df[D].values
y = np.ravel(df[[L]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rand

pca = PCA()
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)
```

In [126]:
```
n_neighborslist = list(range(1,51))
col_names=['uniform', 'manhattan distance', 'euclidean distance','infinte','m
accarray = np.zeros((len(n_neighborslist),5))

#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)
#case 1 with uniform istribution
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='uniform')
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[0]] = accscore

#case 2 with distance based weight and manhattan distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[1]] = accscore

#case 3 with distance based weight and euclidean distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[2]] = accscore

#case 4 with distance based weight and inf dist
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[3]] = accscore

#case 5 with mahalanobis distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k,algorithm='brute', w
                                           metric='mahalanobis',metric_params
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[4]] = accscore
acc.describe()
acc
```

Out[126]:

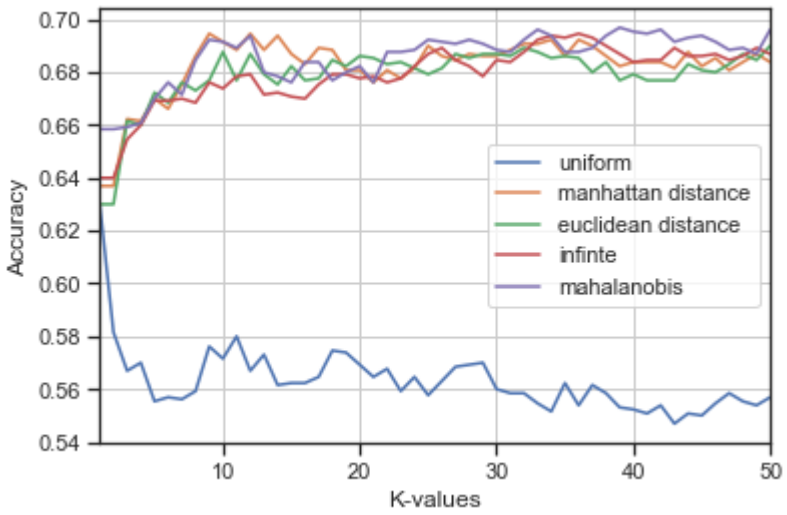
	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
0	0.000000	0.000000	0.000000	0.000000	0.000000
1	0.630000	0.636923	0.630000	0.640000	0.658462

	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
2	0.581538	0.636923	0.630000	0.640000	0.658462
3	0.566923	0.662308	0.661538	0.654615	0.659231
4	0.570000	0.661538	0.660000	0.660000	0.660769
5	0.555385	0.670000	0.672308	0.669231	0.670000
6	0.556923	0.666154	0.668462	0.669231	0.676154
7	0.556154	0.675385	0.676154	0.670000	0.671538
8	0.559231	0.686154	0.673077	0.668462	0.684615
9	0.576154	0.694615	0.676923	0.676154	0.692308
10	0.571538	0.691538	0.687692	0.673846	0.691538
11	0.580000	0.688462	0.676923	0.678462	0.689231
12	0.566923	0.694615	0.686923	0.679231	0.693846
13	0.573077	0.688462	0.679231	0.671538	0.680000
14	0.561538	0.693846	0.675385	0.672308	0.678462
15	0.562308	0.686923	0.682308	0.670769	0.676154
16	0.562308	0.683077	0.676923	0.670000	0.683846
17	0.564615	0.689231	0.677692	0.675385	0.683846
18	0.574615	0.688462	0.684615	0.679231	0.676923
19	0.573846	0.680769	0.682308	0.679231	0.680000
20	0.569231	0.680769	0.686154	0.677692	0.682308
21	0.564615	0.676154	0.685385	0.678462	0.676154
22	0.567692	0.680769	0.683077	0.676154	0.687692
23	0.559231	0.677692	0.683846	0.677692	0.687692
24	0.564615	0.681538	0.681538	0.682308	0.688462
25	0.557692	0.690000	0.679231	0.686923	0.692308
26	0.563077	0.686154	0.681538	0.689231	0.691538
27	0.568462	0.684615	0.686923	0.684615	0.690769
28	0.569231	0.686923	0.685385	0.682308	0.692308
29	0.570000	0.686154	0.686923	0.678462	0.690769
30	0.560000	0.686154	0.686923	0.684615	0.688462
31	0.558462	0.688462	0.686154	0.683846	0.687692
32	0.558462	0.690769	0.689231	0.687692	0.692308
33	0.554615	0.690769	0.687692	0.692308	0.696154
34	0.551538	0.692308	0.685385	0.693846	0.693846
35	0.562308	0.686154	0.686154	0.693077	0.687692
36	0.553846	0.692308	0.685385	0.694615	0.687692
37	0.561538	0.690000	0.680000	0.693077	0.689231

	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
38	0.558462	0.686154	0.683846	0.690000	0.693846
39	0.553077	0.682308	0.676923	0.686923	0.696923
40	0.552308	0.683846	0.679231	0.683846	0.695385
41	0.550769	0.683846	0.676923	0.684615	0.694615
42	0.553846	0.683846	0.676923	0.684615	0.696154
43	0.546923	0.681538	0.676923	0.689231	0.691538
44	0.550769	0.687692	0.683077	0.686154	0.693077
45	0.550000	0.682308	0.680769	0.686154	0.693846
46	0.554615	0.685385	0.680000	0.686923	0.691538
47	0.558462	0.680769	0.683077	0.684615	0.688462
48	0.555385	0.683846	0.686923	0.686154	0.689231
49	0.553846	0.686923	0.684615	0.689231	0.686923
50	0.556923	0.683846	0.690000	0.686923	0.696154

In [127]:
```
r=acc.drop(0)
r.plot()
plt.xlabel('K-values')
plt.grid()
plt.ylabel('Accuracy')
```

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



We observe a tussle between the manhattan and the mahalanobis distance for smaller k values, both having similar accuarcies, But for majority of the k values, we see that mahalanobis has the dominance. The model acquires its maximum accuracy using manhattan as the distance metric at k = 9. This accuracy is higher than the one otained before PCA. The overall accuries obtained are higher as well.

PCA with 5 components, quality

In [197]:

```
#total sulphur dioxide, volatile acidity, chlorides, fixed acidity - best fou
#PCA
from sklearn.decomposition import PCA

# classify color of wine with all features
X = df[D].values
y = np.ravel(df[[L]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rand

pca = PCA(n_components=5)
X_train_pca = pca.fit_transform(X_train,y_train)
X_test_pca = pca.transform(X_test)
```

In [198]:
```
n_neighborslist = list(range(1,51))
col_names=['uniform', 'manhattan distance', 'euclidean distance','infinte','m
accarray = np.zeros((len(n_neighborslist),5))

#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)
#case 1 with uniform istribution
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='uniform')
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[0]] = accscore

#case 2 with distance based weight and manhattan distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[1]] = accscore

#case 3 with distance based weight and euclidean distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[2]] = accscore

#case 4 with distance based weight and inf dist
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[3]] = accscore

#case 5 with mahalanobis distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k,algorithm='brute', w
                                           metric='mahalanobis',metric_params
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[4]] = accscore
acc.describe()
acc
```

Out[198]:

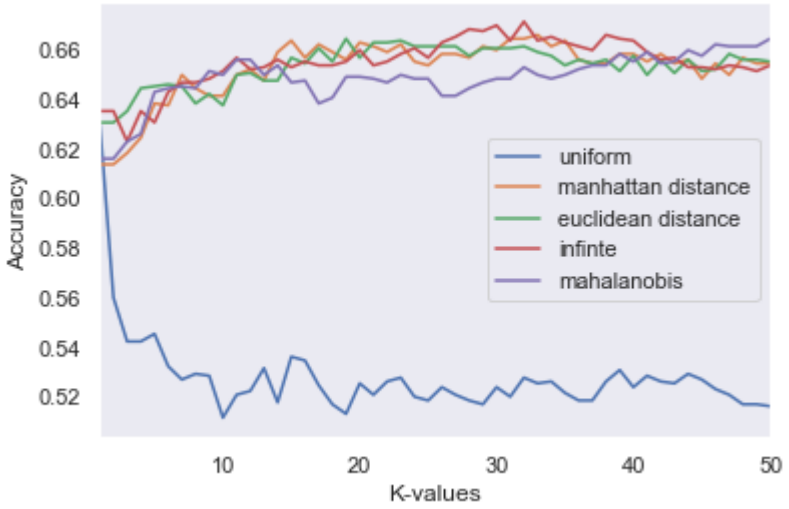
	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
0	0.000000	0.000000	0.000000	0.000000	0.000000
1	0.630769	0.613846	0.630769	0.635385	0.616154

	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
2	0.560000	0.613846	0.630769	0.635385	0.616154
3	0.542308	0.618462	0.635385	0.623077	0.623077
4	0.542308	0.624615	0.644615	0.635385	0.626154
5	0.545385	0.638462	0.645385	0.630769	0.643077
6	0.532308	0.637692	0.646154	0.643077	0.644615
7	0.526923	0.650000	0.645385	0.646923	0.645385
8	0.529231	0.644615	0.638462	0.646923	0.644615
9	0.528462	0.641538	0.642308	0.648462	0.651538
10	0.511538	0.641538	0.637692	0.651538	0.650000
11	0.520769	0.650000	0.650000	0.656923	0.656154
12	0.522308	0.652308	0.650769	0.652308	0.656154
13	0.531538	0.647692	0.647692	0.653077	0.650000
14	0.517692	0.659231	0.647692	0.656154	0.653846
15	0.536154	0.663846	0.656923	0.653077	0.646923
16	0.534615	0.656923	0.654615	0.655385	0.647692
17	0.524615	0.662308	0.660769	0.653846	0.638462
18	0.516923	0.659231	0.655385	0.653846	0.640769
19	0.513077	0.656154	0.664615	0.655385	0.649231
20	0.525385	0.663077	0.656923	0.660000	0.649231
21	0.520769	0.661538	0.663077	0.653846	0.648462
22	0.526154	0.659231	0.663077	0.655385	0.646923
23	0.527692	0.662308	0.663846	0.658462	0.650000
24	0.520000	0.655385	0.661538	0.660769	0.648462
25	0.518462	0.653846	0.661538	0.656923	0.648462
26	0.523846	0.658462	0.661538	0.663077	0.641538
27	0.520769	0.658462	0.661538	0.665385	0.641538
28	0.518462	0.656923	0.657692	0.668462	0.644615
29	0.516923	0.661538	0.660769	0.667692	0.646923
30	0.523846	0.660000	0.660769	0.670000	0.648462
31	0.520000	0.664615	0.660769	0.663846	0.648462
32	0.527692	0.664615	0.661538	0.671538	0.653077
33	0.525385	0.666154	0.659231	0.663846	0.650000
34	0.526154	0.661538	0.657692	0.665385	0.648462
35	0.521538	0.663846	0.653846	0.663077	0.650000
36	0.518462	0.656154	0.656154	0.661538	0.652308
37	0.518462	0.654615	0.654615	0.660000	0.653846

	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
38	0.526154	0.654615	0.656154	0.666154	0.653846
39	0.530769	0.658462	0.651538	0.664615	0.658462
40	0.523846	0.658462	0.657692	0.663846	0.655385
41	0.528462	0.655385	0.650000	0.659231	0.659231
42	0.526154	0.658462	0.656154	0.656154	0.654615
43	0.525385	0.654615	0.650769	0.656923	0.655385
44	0.529231	0.656154	0.656154	0.653077	0.660000
45	0.526923	0.648462	0.651538	0.653077	0.657692
46	0.523077	0.654615	0.652308	0.652308	0.662308
47	0.520769	0.650000	0.658462	0.653846	0.661538
48	0.516923	0.656923	0.656154	0.653077	0.661538
49	0.516923	0.654615	0.656154	0.651538	0.661538
50	0.516154	0.654615	0.655385	0.653846	0.664615

In [199]:
```
r=acc.drop(0)
r.plot()
plt.xlabel('K-values')
plt.grid()
plt.ylabel('Accuracy')
```

Out[199]: Text(0, 0.5, 'Accuracy')



When the number of components were reduced to 5, we observed that as the distance increases(p value in the p-norm), the accuracy of the model increases as well.The maximum accuracy obtained is for p=infinite for majority of the cases. The overall accuracy of the model is low when compared to the results obtained when all pca components were used. But using only 5 components we were able to obtain accuracy ~95% of the previous case. This leads to reducing the complexity and saves time.

PCA selected features, quality

In [225]:

```
#total sulphur dioxide, volatile acidity, chlorides, fixed acidity - best fou
#PCA
from sklearn.decomposition import PCA

# classify color of wine with all features
X = df[['density', 'alcohol', "chlorides", 'residual sugar']].values
y = np.ravel(df[[L]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rand

pca = PCA()
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)
```

In [226]:
```
n_neighborslist = list(range(1,51))
col_names=['uniform', 'manhattan distance', 'euclidean distance','infinte','m
accarray = np.zeros((len(n_neighborslist),5))

#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)
#case 1 with uniform istribution
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='uniform')
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[0]] = accscore

#case 2 with distance based weight and manhattan distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[1]] = accscore

#case 3 with distance based weight and euclidean distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[2]] = accscore

#case 4 with distance based weight and inf dist
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[3]] = accscore

#case 5 with mahalanobis distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k,algorithm='brute', w
                                           metric='mahalanobis',metric_params
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[4]] = accscore
acc.describe()
acc
```

Out[226]:

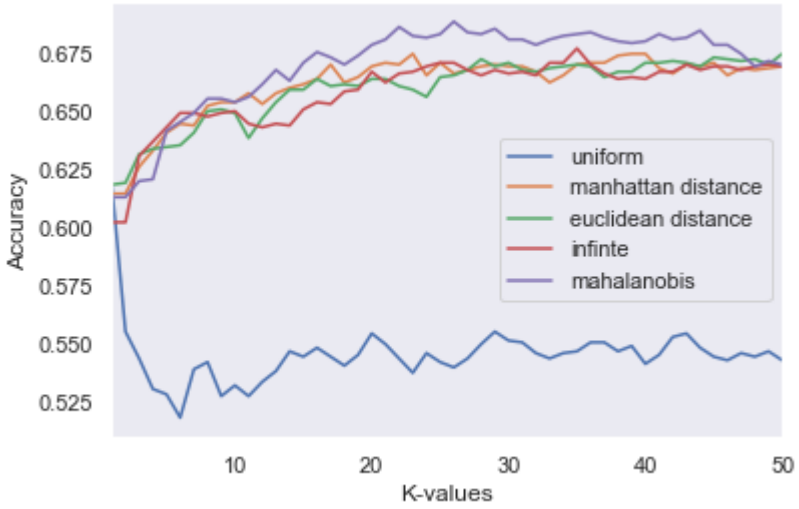
	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
0	0.000000	0.000000	0.000000	0.000000	0.000000
1	0.618462	0.614615	0.618462	0.602308	0.613077

	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
2	0.555385	0.614615	0.619231	0.602308	0.613077
3	0.543846	0.626154	0.631538	0.630769	0.620000
4	0.530769	0.633077	0.633846	0.636923	0.620769
5	0.528462	0.640769	0.634615	0.643077	0.641538
6	0.518462	0.644615	0.635385	0.649231	0.645385
7	0.539231	0.643846	0.640769	0.649231	0.649231
8	0.542308	0.652308	0.650000	0.647692	0.655385
9	0.527692	0.653846	0.650769	0.649231	0.655385
10	0.532308	0.653846	0.649231	0.650000	0.653846
11	0.527692	0.657692	0.638462	0.644615	0.656154
12	0.533846	0.653077	0.646923	0.643077	0.661538
13	0.538462	0.657692	0.653846	0.644615	0.667692
14	0.546923	0.660000	0.659231	0.643846	0.663077
15	0.544615	0.661538	0.659231	0.650769	0.670769
16	0.548462	0.663846	0.663846	0.653846	0.675385
17	0.544615	0.670000	0.660769	0.653077	0.673077
18	0.540769	0.662308	0.661538	0.658462	0.670000
19	0.545385	0.664615	0.660769	0.659231	0.673846
20	0.554615	0.669231	0.663846	0.666923	0.678462
21	0.550000	0.670769	0.663846	0.662308	0.680769
22	0.543846	0.670000	0.660769	0.666154	0.686154
23	0.537692	0.674615	0.659231	0.666923	0.682308
24	0.546154	0.665385	0.656154	0.669231	0.681538
25	0.542308	0.670769	0.664615	0.670769	0.683077
26	0.540000	0.666154	0.665385	0.670769	0.688462
27	0.543846	0.667692	0.667692	0.667692	0.683846
28	0.550000	0.669231	0.672308	0.665385	0.683077
29	0.555385	0.670000	0.669231	0.667692	0.685385
30	0.551538	0.669231	0.670769	0.666154	0.680769
31	0.550769	0.669231	0.667692	0.666923	0.680769
32	0.546154	0.666923	0.666923	0.665385	0.678462
33	0.543846	0.662308	0.668462	0.670769	0.680769
34	0.546154	0.665385	0.669231	0.670769	0.682308
35	0.546923	0.670000	0.670000	0.676923	0.683077
36	0.550769	0.670769	0.669231	0.670000	0.683846
37	0.550769	0.670769	0.664615	0.666154	0.681538

	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
38	0.546923	0.673846	0.666923	0.663846	0.680000
39	0.549231	0.674615	0.666923	0.664615	0.679231
40	0.541538	0.674615	0.670769	0.663846	0.680000
41	0.545385	0.668462	0.670769	0.666923	0.683077
42	0.553077	0.666154	0.671538	0.666923	0.680769
43	0.554615	0.670000	0.670769	0.670000	0.681538
44	0.548462	0.669231	0.669231	0.667692	0.684615
45	0.544615	0.670769	0.673077	0.669231	0.678462
46	0.543077	0.665385	0.672308	0.669231	0.678462
47	0.546154	0.668462	0.671538	0.667692	0.674615
48	0.544615	0.667692	0.672308	0.669231	0.669231
49	0.546923	0.668462	0.670000	0.670000	0.671538
50	0.543077	0.669231	0.674615	0.669231	0.670000

In [227]:
```
r=acc.drop(0)
r.plot()
plt.xlabel('K-values')
plt.grid()
plt.ylabel('Accuracy')
```

Out[227]: Text(0, 0.5, 'Accuracy')



LDA on quality as outcome

In [131]:

```
#total sulphur dioxide, volatile acidity, chlorides, fixed acidity - best fou
#PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# classify color of wine with all features
X = df[D].values
y = np.ravel(df[[L]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rand

lda = LinearDiscriminantAnalysis()
X_train_pca = lda.fit_transform(X_train,y_train)
X_test_pca = lda.transform(X_test)
```

In [132]:
```
n_neighborslist = list(range(1,51))
col_names=['uniform', 'manhattan distance', 'euclidean distance','infinte','m
accarray = np.zeros((len(n_neighborslist),5))

#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)
#case 1 with uniform istribution
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='uniform')
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[0]] = accscore

#case 2 with distance based weight and manhattan distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[1]] = accscore

#case 3 with distance based weight and euclidean distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[2]] = accscore

#case 4 with distance based weight and inf dist
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[3]] = accscore

#case 5 with mahalanobis distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k,algorithm='brute', w
                                           metric='mahalanobis',metric_params
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[4]] = accscore
acc.describe()
acc
```

Out[132]:

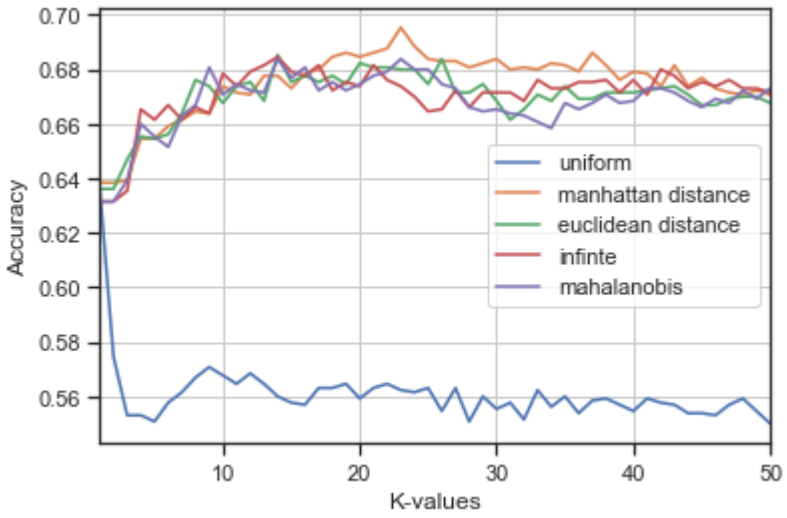
	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
0	0.000000	0.000000	0.000000	0.000000	0.000000
1	0.636154	0.638462	0.636154	0.631538	0.631538

	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
2	0.574615	0.638462	0.636154	0.631538	0.631538
3	0.553077	0.639231	0.646923	0.635385	0.639231
4	0.553077	0.654615	0.655385	0.665385	0.660000
5	0.550769	0.654615	0.654615	0.661538	0.655385
6	0.557692	0.659231	0.656154	0.666923	0.651538
7	0.561538	0.661538	0.663846	0.661538	0.663077
8	0.566923	0.664615	0.676154	0.666154	0.666923
9	0.570769	0.663846	0.673846	0.663846	0.680769
10	0.567692	0.673846	0.667692	0.678462	0.670769
11	0.564615	0.671538	0.673846	0.673846	0.674615
12	0.568462	0.670769	0.675385	0.679231	0.672308
13	0.564615	0.677692	0.668462	0.681538	0.671538
14	0.560000	0.677692	0.685385	0.684615	0.683846
15	0.557692	0.673077	0.675385	0.679231	0.676923
16	0.556923	0.678462	0.677692	0.677692	0.680769
17	0.563077	0.680000	0.675385	0.681538	0.672308
18	0.563077	0.684615	0.677692	0.672308	0.675385
19	0.564615	0.686154	0.674615	0.675385	0.672308
20	0.559231	0.684615	0.682308	0.673846	0.674615
21	0.563077	0.686154	0.680769	0.681538	0.677692
22	0.564615	0.687692	0.680769	0.676154	0.679231
23	0.562308	0.695385	0.680000	0.673846	0.683846
24	0.561538	0.688462	0.680000	0.670000	0.680000
25	0.563077	0.683846	0.674615	0.664615	0.680000
26	0.554615	0.683077	0.683846	0.665385	0.674615
27	0.563077	0.683077	0.671538	0.672308	0.673077
28	0.550769	0.680769	0.671538	0.666154	0.666154
29	0.560000	0.682308	0.674615	0.671538	0.664615
30	0.555385	0.683846	0.668462	0.671538	0.665385
31	0.557692	0.680000	0.661538	0.671538	0.663846
32	0.551538	0.680769	0.665385	0.668462	0.663077
33	0.562308	0.680000	0.670769	0.676154	0.660769
34	0.556154	0.682308	0.668462	0.673077	0.658462
35	0.560000	0.681538	0.673846	0.673077	0.667692
36	0.553846	0.679231	0.669231	0.675385	0.665385
37	0.558462	0.686154	0.669231	0.675385	0.667692

	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
38	0.559231	0.681538	0.671538	0.676154	0.670769
39	0.556923	0.676154	0.671538	0.671538	0.667692
40	0.554615	0.679231	0.671538	0.676154	0.668462
41	0.559231	0.678462	0.672308	0.670769	0.673077
42	0.557692	0.673846	0.673077	0.680000	0.673077
43	0.556923	0.681538	0.673846	0.677692	0.671538
44	0.553846	0.673846	0.670769	0.673077	0.668462
45	0.553846	0.676923	0.666923	0.675385	0.666154
46	0.553077	0.673077	0.666923	0.673846	0.669231
47	0.556923	0.671538	0.669231	0.676154	0.667692
48	0.559231	0.670769	0.670000	0.673077	0.672308
49	0.554615	0.672308	0.670000	0.673077	0.669231
50	0.550000	0.670769	0.667692	0.671538	0.673077

In [133]:
```
r=acc.drop(0)
r.plot()
plt.xlabel('K-values')
plt.grid()
plt.ylabel('Accuracy')
```

Out[133]: Text(0, 0.5, 'Accuracy')



The accuracy obtained for the LDA model is lower compared to that of PCA. The maximum accuracy is obtained by using the manhattan distance which is higher than the maximum accuracy obtained in the previous cases.

In [228]:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# classify color of wine with all features
X = df[['density', 'alcohol', "chlorides", 'residual sugar']].values
y = np.ravel(df[[L]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rand

lda = LinearDiscriminantAnalysis()
X_train_pca = lda.fit_transform(X_train,y_train)
X_test_pca = lda.transform(X_test)
```

In [229]:
```
n_neighborslist = list(range(1,51))
col_names=['uniform', 'manhattan distance', 'euclidean distance','infinte','m
accarray = np.zeros((len(n_neighborslist),5))

#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)
#case 1 with uniform istribution
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='uniform')
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[0]] = accscore

#case 2 with distance based weight and manhattan distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[1]] = accscore

#case 3 with distance based weight and euclidean distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[2]] = accscore

#case 4 with distance based weight and inf dist
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[3]] = accscore

#case 5 with mahalanobis distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k,algorithm='brute', w
                                           metric='mahalanobis',metric_params
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[4]] = accscore
acc.describe()
acc
```

Out[229]:

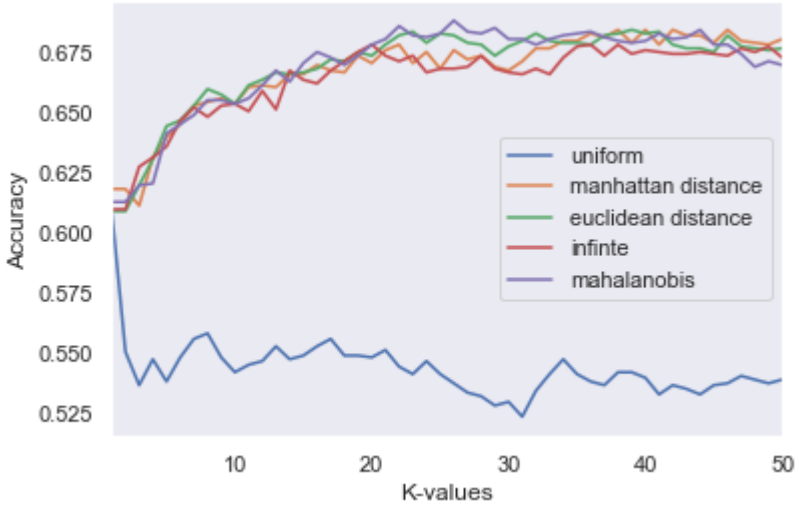
	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
0	0.000000	0.000000	0.000000	0.000000	0.000000
1	0.609231	0.618462	0.609231	0.610000	0.613077
2	0.550769	0.618462	0.609231	0.610000	0.613077

	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
3	0.536923	0.611538	0.620000	0.627692	0.620000
4	0.547692	0.631538	0.630769	0.631538	0.620769
5	0.538462	0.640000	0.644615	0.636154	0.641538
6	0.548462	0.645385	0.646923	0.646923	0.645385
7	0.556154	0.653077	0.653077	0.652308	0.649231
8	0.558462	0.654615	0.660000	0.648462	0.655385
9	0.548462	0.656154	0.657692	0.653077	0.655385
10	0.542308	0.653846	0.653846	0.653846	0.653846
11	0.545385	0.660769	0.661538	0.650769	0.656154
12	0.546923	0.661538	0.663846	0.659231	0.661538
13	0.553077	0.660769	0.666923	0.651538	0.667692
14	0.547692	0.666154	0.666154	0.667692	0.663077
15	0.549231	0.666154	0.666923	0.663846	0.670769
16	0.553077	0.670000	0.668462	0.662308	0.675385
17	0.556154	0.667692	0.672308	0.667692	0.673077
18	0.549231	0.666923	0.671538	0.671538	0.670000
19	0.549231	0.673846	0.675385	0.675385	0.673846
20	0.548462	0.670769	0.673846	0.678462	0.678462
21	0.551538	0.676154	0.678462	0.673846	0.680769
22	0.544615	0.678462	0.682308	0.671538	0.686154
23	0.541538	0.670769	0.683846	0.673846	0.682308
24	0.546923	0.675385	0.679231	0.666923	0.681538
25	0.541538	0.668462	0.683077	0.668462	0.683077
26	0.537692	0.676154	0.682308	0.668462	0.688462
27	0.533846	0.672308	0.679231	0.669231	0.683846
28	0.532308	0.673846	0.678462	0.673846	0.683077
29	0.528462	0.669231	0.673846	0.668462	0.685385
30	0.530000	0.667692	0.677692	0.666923	0.680769
31	0.523846	0.671538	0.680000	0.666154	0.680769
32	0.534615	0.676923	0.683077	0.668462	0.678462
33	0.541538	0.676923	0.680000	0.666154	0.680769
34	0.547692	0.680000	0.679231	0.673077	0.682308
35	0.541538	0.680000	0.679231	0.677692	0.683077
36	0.538462	0.683077	0.678462	0.678462	0.683846
37	0.536923	0.681538	0.682308	0.673846	0.681538
38	0.542308	0.684615	0.683077	0.678462	0.680000

	uniform	manhattan distance	euclidean distance	infinte	mahalanobis
39	0.542308	0.679231	0.684615	0.674615	0.679231
40	0.540000	0.684615	0.683077	0.676154	0.680000
41	0.533077	0.678462	0.683846	0.675385	0.683077
42	0.536923	0.684615	0.678462	0.674615	0.680769
43	0.535385	0.682308	0.676923	0.674615	0.681538
44	0.533077	0.682308	0.676923	0.675385	0.684615
45	0.536923	0.679231	0.675385	0.674615	0.678462
46	0.537692	0.684615	0.682308	0.673846	0.678462
47	0.540769	0.680000	0.677692	0.676923	0.674615
48	0.539231	0.679231	0.676923	0.675385	0.669231
49	0.537692	0.678462	0.676154	0.677692	0.671538
50	0.539231	0.680769	0.676923	0.673077	0.670000

In [230]:
```
r=acc.drop(0)
r.plot()
plt.xlabel('K-values')
plt.grid()
plt.ylabel('Accuracy')
```

Out[230]: Text(0, 0.5, 'Accuracy')



When the number of components were reduced to 5, that the maximum accuracy was obtained for manhattan for majority of the cases. The overall accuracy of the model is low when compared to the results obtained when all lda components were used. But using only 5 components we were able to obtain accuracy ~95% of the previous case. This leads to reducing the complexity and saves time.

LDA on color as target

In [213]:

```
#total sulphur dioxide, volatile acidity, chlorides, fixed acidity - best fou
#PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# classify color of wine with all features
X = df[D].values
y = np.ravel(df[[C]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rand

lda = LinearDiscriminantAnalysis()
X_train_pca = lda.fit_transform(X_train,y_train)
X_test_pca = lda.transform(X_test)
```

In [214]:
```
n_neighborslist = list(range(1,51))
col_names=['uniform', 'manhattan distance', 'euclidean distance','infinte']
accarray = np.zeros((len(n_neighborslist),4))

#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)
#case 1 with uniform istribution
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='uniform')
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[0]] = accscore

#case 2 with distance based weight and manhattan distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[1]] = accscore

#case 3 with distance based weight and euclidean distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[2]] = accscore

#case 4 with distance based weight and inf dist
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[3]] = accscore

acc.describe()
acc
```

Out[214]:

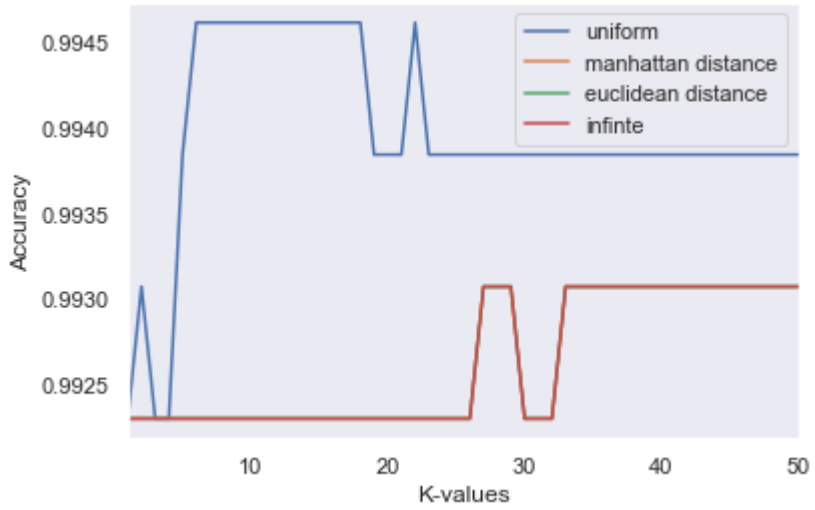
	uniform	manhattan distance	euclidean distance	infinte
0	0.000000	0.000000	0.000000	0.000000
1	0.992308	0.992308	0.992308	0.992308
2	0.993077	0.992308	0.992308	0.992308
3	0.992308	0.992308	0.992308	0.992308
4	0.992308	0.992308	0.992308	0.992308
5	0.993846	0.992308	0.992308	0.992308
6	0.994615	0.992308	0.992308	0.992308
7	0.994615	0.992308	0.992308	0.992308

	uniform	manhattan distance	euclidean distance	infinte
8	0.994615	0.992308	0.992308	0.992308
9	0.994615	0.992308	0.992308	0.992308
10	0.994615	0.992308	0.992308	0.992308
11	0.994615	0.992308	0.992308	0.992308
12	0.994615	0.992308	0.992308	0.992308
13	0.994615	0.992308	0.992308	0.992308
14	0.994615	0.992308	0.992308	0.992308
15	0.994615	0.992308	0.992308	0.992308
16	0.994615	0.992308	0.992308	0.992308
17	0.994615	0.992308	0.992308	0.992308
18	0.994615	0.992308	0.992308	0.992308
19	0.993846	0.992308	0.992308	0.992308
20	0.993846	0.992308	0.992308	0.992308
21	0.993846	0.992308	0.992308	0.992308
22	0.994615	0.992308	0.992308	0.992308
23	0.993846	0.992308	0.992308	0.992308
24	0.993846	0.992308	0.992308	0.992308
25	0.993846	0.992308	0.992308	0.992308
26	0.993846	0.992308	0.992308	0.992308
27	0.993846	0.993077	0.993077	0.993077
28	0.993846	0.993077	0.993077	0.993077
29	0.993846	0.993077	0.993077	0.993077
30	0.993846	0.992308	0.992308	0.992308
31	0.993846	0.992308	0.992308	0.992308
32	0.993846	0.992308	0.992308	0.992308
33	0.993846	0.993077	0.993077	0.993077
34	0.993846	0.993077	0.993077	0.993077
35	0.993846	0.993077	0.993077	0.993077
36	0.993846	0.993077	0.993077	0.993077
37	0.993846	0.993077	0.993077	0.993077
38	0.993846	0.993077	0.993077	0.993077
39	0.993846	0.993077	0.993077	0.993077
40	0.993846	0.993077	0.993077	0.993077
41	0.993846	0.993077	0.993077	0.993077
42	0.993846	0.993077	0.993077	0.993077
43	0.993846	0.993077	0.993077	0.993077

	uniform	manhattan distance	euclidean distance	infinte
44	0.993846	0.993077	0.993077	0.993077
45	0.993846	0.993077	0.993077	0.993077
46	0.993846	0.993077	0.993077	0.993077
47	0.993846	0.993077	0.993077	0.993077
48	0.993846	0.993077	0.993077	0.993077
49	0.993846	0.993077	0.993077	0.993077
50	0.993846	0.993077	0.993077	0.993077

In [215]:
```
r=acc.drop(0)
r.plot()
plt.xlabel('K-values')
plt.grid()
plt.ylabel('Accuracy')
```

Out[215]: Text(0, 0.5, 'Accuracy')



LDA, selected features, color as target

In [231]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# classify color of wine with all features
X = df[['density', 'total sulfur dioxide', "chlorides", 'residual sugar']].va
y = np.ravel(df[[C]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rand

lda = LinearDiscriminantAnalysis()
X_train_pca = lda.fit_transform(X_train,y_train)
X_test_pca = lda.transform(X_test)
```

In [232]:
```
n_neighborslist = list(range(1,51))
col_names=['uniform', 'manhattan distance', 'euclidean distance','infinte']
accarray = np.zeros((len(n_neighborslist),4))

#add multiple plots to same chart, one for each weighting approach
acc=pd.DataFrame(accarray, columns=col_names)
#case 1 with uniform istribution
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='uniform')
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[0]] = accscore

#case 2 with distance based weight and manhattan distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[1]] = accscore

#case 3 with distance based weight and euclidean distance
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[2]] = accscore

#case 4 with distance based weight and inf dist
for k in n_neighborslist:
    neigh = neighbors.KNeighborsClassifier(n_neighbors=k, weights='distance',
    neigh.fit(X_train_pca, y_train)
    y_pred = neigh.predict(X_test_pca)
    accscore = accuracy_score(y_test, y_pred)
    acc.at[k,col_names[3]] = accscore

acc.describe()
acc
```

Out[232]:

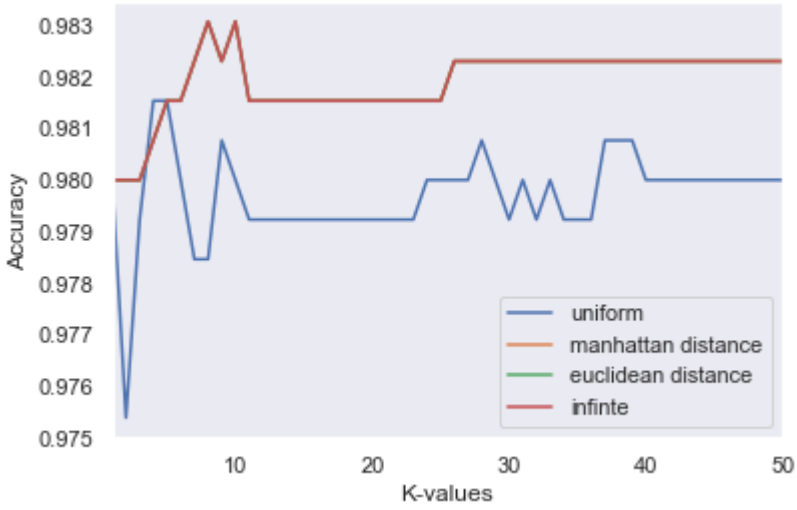
	uniform	manhattan distance	euclidean distance	infinte
0	0.000000	0.000000	0.000000	0.000000
1	0.980000	0.980000	0.980000	0.980000
2	0.975385	0.980000	0.980000	0.980000
3	0.979231	0.980000	0.980000	0.980000
4	0.981538	0.980769	0.980769	0.980769
5	0.981538	0.981538	0.981538	0.981538
6	0.980000	0.981538	0.981538	0.981538

	uniform	manhattan distance	euclidean distance	infinte
7	0.978462	0.982308	0.982308	0.982308
8	0.978462	0.983077	0.983077	0.983077
9	0.980769	0.982308	0.982308	0.982308
10	0.980000	0.983077	0.983077	0.983077
11	0.979231	0.981538	0.981538	0.981538
12	0.979231	0.981538	0.981538	0.981538
13	0.979231	0.981538	0.981538	0.981538
14	0.979231	0.981538	0.981538	0.981538
15	0.979231	0.981538	0.981538	0.981538
16	0.979231	0.981538	0.981538	0.981538
17	0.979231	0.981538	0.981538	0.981538
18	0.979231	0.981538	0.981538	0.981538
19	0.979231	0.981538	0.981538	0.981538
20	0.979231	0.981538	0.981538	0.981538
21	0.979231	0.981538	0.981538	0.981538
22	0.979231	0.981538	0.981538	0.981538
23	0.979231	0.981538	0.981538	0.981538
24	0.980000	0.981538	0.981538	0.981538
25	0.980000	0.981538	0.981538	0.981538
26	0.980000	0.982308	0.982308	0.982308
27	0.980000	0.982308	0.982308	0.982308
28	0.980769	0.982308	0.982308	0.982308
29	0.980000	0.982308	0.982308	0.982308
30	0.979231	0.982308	0.982308	0.982308
31	0.980000	0.982308	0.982308	0.982308
32	0.979231	0.982308	0.982308	0.982308
33	0.980000	0.982308	0.982308	0.982308
34	0.979231	0.982308	0.982308	0.982308
35	0.979231	0.982308	0.982308	0.982308
36	0.979231	0.982308	0.982308	0.982308
37	0.980769	0.982308	0.982308	0.982308
38	0.980769	0.982308	0.982308	0.982308
39	0.980769	0.982308	0.982308	0.982308
40	0.980000	0.982308	0.982308	0.982308
41	0.980000	0.982308	0.982308	0.982308
42	0.980000	0.982308	0.982308	0.982308

	uniform	manhattan distance	euclidean distance	infinte
43	0.980000	0.982308	0.982308	0.982308
44	0.980000	0.982308	0.982308	0.982308
45	0.980000	0.982308	0.982308	0.982308
46	0.980000	0.982308	0.982308	0.982308
47	0.980000	0.982308	0.982308	0.982308
48	0.980000	0.982308	0.982308	0.982308
49	0.980000	0.982308	0.982308	0.982308
50	0.980000	0.982308	0.982308	0.982308

In [233]:
```
r=acc.drop(0)
r.plot()
plt.xlabel('K-values')
plt.grid()
plt.ylabel('Accuracy')
```

Out[233]: Text(0, 0.5, 'Accuracy')

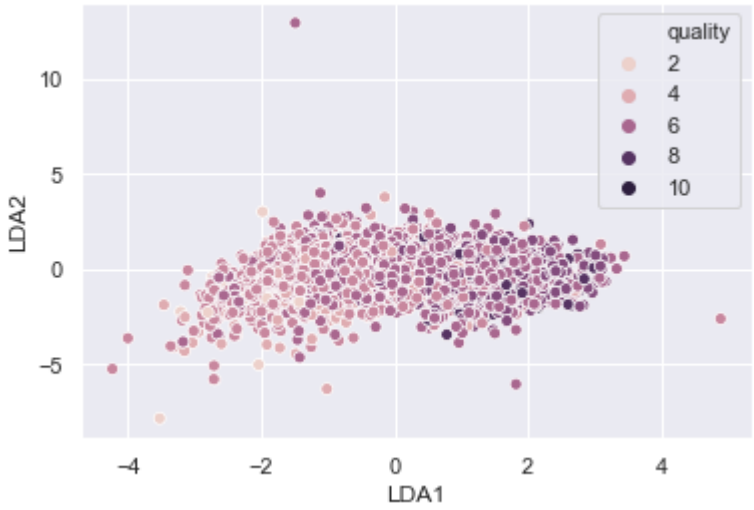


LDA performs better when the number of classes for distinction are many. Since color has only 2 points for seperabilty, therefore all the results for p-norms distance metrics are same as others.

PCA performs better in case where number of samples per class is less. Whereas LDA works better with large dataset having multiple classes; class separability is an important factor while reducing dimensionality.

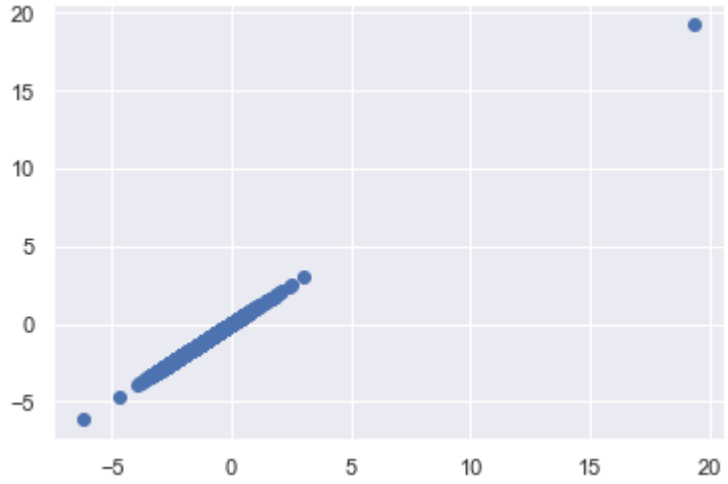
In [141]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# classify color of wine with all features
X = df[D].values
y = np.ravel(df[[L]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ranc

lda = LinearDiscriminantAnalysis(n_components=2)
X_train_lda = lda.fit(X_train,y_train)
X_value= lda.transform(X)

dataframe = pd.DataFrame(data=X_value)
dataframe['quality']=y
dataframe.columns=("LDA1", "LDA2","quality")

import seaborn as sns; sns.set()
import matplotlib.pyplot as plt

sns.scatterplot(x="LDA1", y="LDA2", hue="quality",data=dataframe)
```

Out[141]: <matplotlib.axes._subplots.AxesSubplot at 0x20f8391ee88>



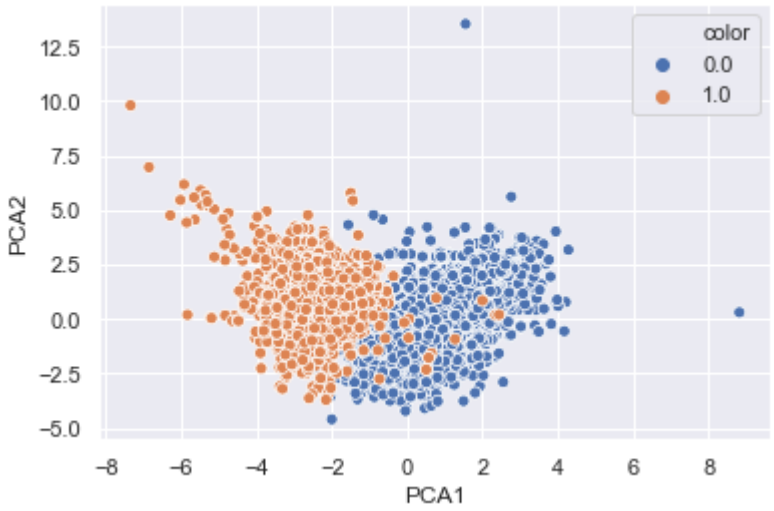
In [218]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# classify color of wine with all features
X = df[D].values
y = np.ravel(df[[C]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rand

lda = LinearDiscriminantAnalysis().fit_transform(X_train, y_train)
plt.scatter(lda[y_train == 0, 0], lda[y_train == 0, 0])
```

Out[218]: <matplotlib.collections.PathCollection at 0x20f86289588>



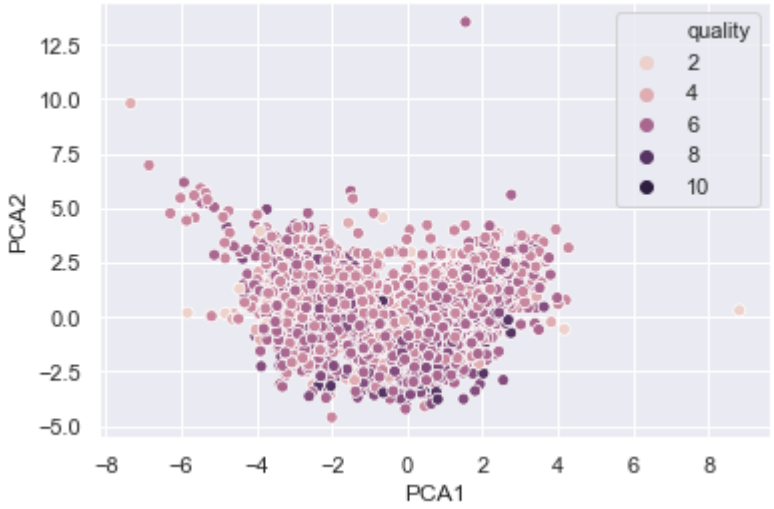
In [143]:
```
from sklearn.decomposition import PCA

# classify color of wine with all features
X = df[D].values
y = np.ravel(df[[C]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ranc

pca = PCA(n_components = 2)
X_train_pca = pca.fit(X_train,y_train)
X_value = pca.transform(X)

dataframe = pd.DataFrame(data=X_value)
dataframe['color']=y
dataframe.columns=("PCA1", "PCA2","color")

import seaborn as sns; sns.set()
import matplotlib.pyplot as plt

sns.scatterplot(x="PCA1", y="PCA2", hue="color",data=dataframe)
```

Out[143]: <matplotlib.axes._subplots.AxesSubplot at 0x20f839c3888>



In [144]:
```
from sklearn.decomposition import PCA

# classify color of wine with all features
X = df[D].values
y = np.ravel(df[[L]])

ran = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rand

pca = PCA(n_components = 2)
X_train_pca = pca.fit(X_train,y_train)
X_value = pca.transform(X)

dataframe = pd.DataFrame(data=X_value)
dataframe['quality']=y
dataframe.columns=("PCA1", "PCA2","quality")

sns.scatterplot(x="PCA1", y="PCA2", hue="quality",data=dataframe)
```

Out[144]: <matplotlib.axes._subplots.AxesSubplot at 0x20f84a0e548>



Analysis

1. k-plots: All the plots required for analysis have been plotted and explained above.
2. Features: This dataset involves 13 variables out which 2 are targets and others are independent variables. a. Using color as the outcome: none of the variables provided complete distinction between the two classes of wine color. Still variables such as total sulfur dioxide, chlorides, volatile acidity can be used for distinction and grouping the data into 2 classes. Total sulfur dioxide and free sulfur dioxide seems to be easy parameters for grouping data. b. Using quality as the outcome: Wine quality does not provide any useful information because there are seven classes for distinction with a lot of overlap. Yet, if need be, variables such as alcohol, free sulphur dioxide can be used as parameters for distinction(as observed from the histogram)
3. Selected features: Three tests were conducted on the dataset separately for color and quality. From the results a subset of 5-6 features were chosen that could improve the performance of the model. a. Color: For color, 'chlorides','density','total sulfur dioxide','residual sugar', 'volatile acidity' and 'fixed acidity' were chosen. Then different combinations were tested before choosing 'chlorides','density','total sulfur dioxide','residual sugar' as they increased the performance of the model. b. Quality: For quality, 'chlorides','density','alcohol','residual sugar','volatile acidity' and 'total sulfur dioxide' were chosen. Then these combinations were tested before choosing 'chlorides', 'density', 'alcohol', 'residual sugar'. Yet these were unable to increase the accuracy of the model and neither did adding a fifth element to these four, which leads us to the conclusion that the left over variables are dependent on each other and that when they all come together, it increases the accuracy of the model.
4. Comparision to PCA and LDA: a. Color: When the results were compared for PCA and LDA, we did not observe any change in the maximum accuracy obtained from PCA, but it reduced for LDA. For PCA, the max accuracy was obtained at smaller k value whereas for LDA the accuracy was less and was obtained at comapratively larger value. b. Quality: There wasn't any significant change when compared for PCA and LDA for quality. The max. accuracy remained the same for all three cases and was obtained at the same k values from mahalnobis distance metric.
5. PCA vs LDA: a. Selecting a subset of features resulted in higher accuracy at smaller k values. This increases the performance and reduces the compuattion complexity and time. b. PCA worked better for color target as the number of classes were small, therefore, for smaller k values we were able to obtain high accuracy whereas LDA worked better for quality target because quality had 7 classes. Using LDA, it was able to obtain the same level of accuracy for smaller value of K. This led to lesser computational complexity while maintaining the accuracy. c. Normalization tends to increase the performance of the model by standardizaing it on the desired scale.
6. Plots: LDA & PCA first 2 components a. Looking at the PCA plot for color target, it can be seen that the variables are easily distinguishable. When compared to pairplot, we were unable to find a pair that complete separates them. PCA makes it simpler with just 2 components b. Looking at the PCA plot for quality target, it can be seen that the variables are slightly distinguishable with a lot of overlap. When compared to pairplot, we were unable to find a pair that complete separates them. With PCA we can seperate a few of them if not all with the first 2 components. c.Looking at the LDA plot for quality target, it can be seen that the variables are distinguishable. When compared to pairplot, we were unable to find a pair that complete separates them. LDA makes it simpler with just 2 components d. Looking at the LDA plot for color target, it just projects one class which does not imply any useful information and hence can be ignored.

Question 2

In [244]:
```
#libraries
from sklearn.decomposition import PCA
import numpy as np
import pandas as pd

data = pd.read_csv("DataB.csv")
```

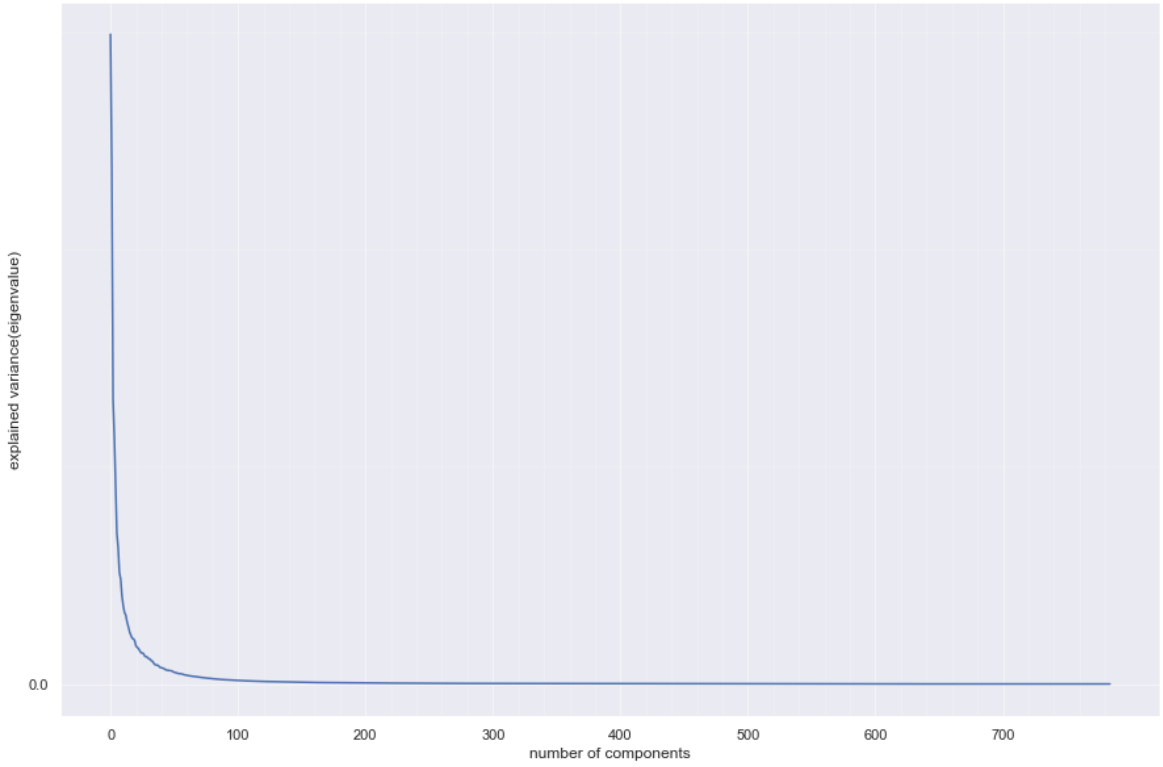
In [245]:
```
#splitting the data into dependent and independent variables
X = data.iloc[:,:-1].values #all columns except the last one
y = data.iloc[:,-1].values #only the last column
```

In [246]:
```
#applying PCA on the dataset
import numpy as np
pca = PCA(random_state=42)
transformed_data = pca.fit_transform(X)

#computing eigenvalues
print("Eigenvectors:",pca.components_)
print("Eigenvalues",pca.explained_variance_)
```

```
Eigenvectors: [[-7.15179787e-01 -3.41186737e-05 -1.90381895e-05 ... -5.23
501700e-06
  -4.25266223e-05 -1.28815021e-05]
 [ 3.14604048e-01  6.40427535e-05  3.54524546e-05 ...  1.59630883e-05
  -5.26104056e-06  1.16214963e-04]
 [ 1.62203522e-01  1.23456596e-06  1.59062509e-05 ...  1.13558926e-06
  -1.06345445e-04 -5.86265097e-05]
 ...
 [ 1.21010333e-04 -6.96189839e-02  5.11526676e-02 ... -8.79294861e-02
   3.48833624e-02 -2.83746395e-02]
 [-1.67108618e-04  2.65092355e-02  1.31931772e-01 ... -6.23608673e-02
  -6.37603998e-02  5.02921792e-02]
 [-1.02404540e-04 -1.01221352e-01  2.73052164e-02 ... -1.51018718e-01
  -5.97551479e-02 -4.91464845e-02]]
Eigenvalues [5.58677817e+05 4.45687427e+05 2.44841589e+05 2.13431674e+05
 1.72629676e+05 1.29741674e+05 1.17369338e+05 9.58290551e+04
 9.01641334e+04 7.57874327e+04 6.73884997e+04 6.13909634e+04
 5.91537110e+04 5.33485122e+04 4.93218152e+04 4.46362200e+04
 4.20482414e+04 3.92453187e+04 3.88929601e+04 3.71893567e+04
```

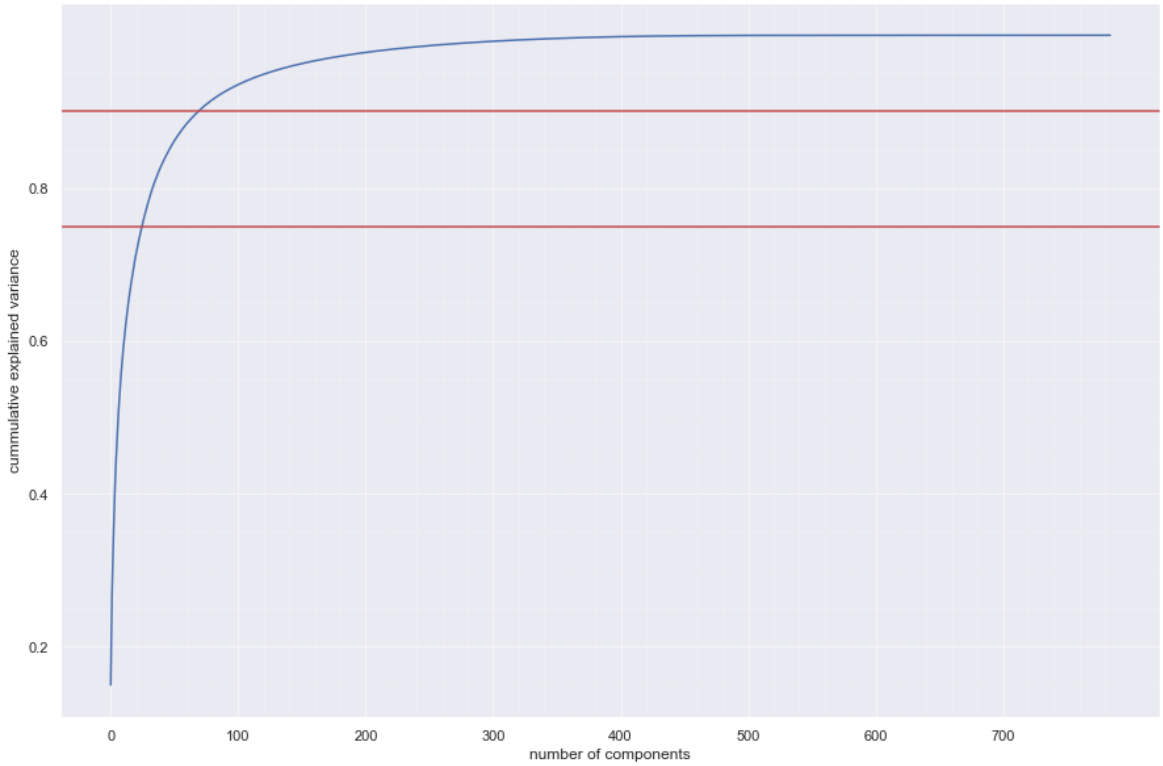
In [247]:

```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(15,10))
ax = plt.subplot(111)

major_ticks_x = np.arange(0, 800, 100)
minor_ticks_x = np.arange(0, 800, 20)
major_ticks_y = np.arange(0, 1, 0.2)
minor_ticks_y = np.arange(0, 1, 0.05)

ax.set_xticks(major_ticks_x)
ax.set_xticks(minor_ticks_x, minor=True)
ax.set_yticks(major_ticks_y)
ax.set_yticks(minor_ticks_y, minor=True)

# And a corresponding grid
ax.grid(which='both')

# Or if you want different settings for the grids:
num_vars=785
sing_vals = np.arange(num_vars) + 1
ax.grid(which='minor', alpha=0.2)
ax.grid(which='major', alpha=0.5)
ax.plot(pca.explained_variance_ratio_)
plt.xlabel('number of components')
plt.ylabel('explained variance(eigenvalue)')
#plt.axhline(y=0.9, color='r', linestyle='-')
#plt.axhline(y=0.75, color='r', linestyle='-')
plt.show()
```



The above plotted scree graph shows the relation between the number of components and the corresponding eigenvalues. To completely understand that how many components would be sufficient, we have plotted a cummulative eigenvalue graph for better understanding.

In [249]:
```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(15,10))
ax = plt.subplot(111)

major_ticks_x = np.arange(0, 800, 100)
minor_ticks_x = np.arange(0, 800, 20)
major_ticks_y = np.arange(0, 1, 0.2)
minor_ticks_y = np.arange(0, 1, 0.05)

ax.set_xticks(major_ticks_x)
ax.set_xticks(minor_ticks_x, minor=True)
ax.set_yticks(major_ticks_y)
ax.set_yticks(minor_ticks_y, minor=True)

# And a corresponding grid
ax.grid(which='both')

# Or if you want different settings for the grids:
num_vars=785
sing_vals = np.arange(num_vars) + 1
ax.grid(which='minor', alpha=0.2)
ax.grid(which='major', alpha=0.5)
ax.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cummulative explained variance')
plt.axhline(y=0.9, color='r', linestyle='-')
plt.axhline(y=0.75, color='r', linestyle='-')
plt.show()
```



From the above plot, the following can be said:

1. For 100% coverage, nearly 300 components would be required.

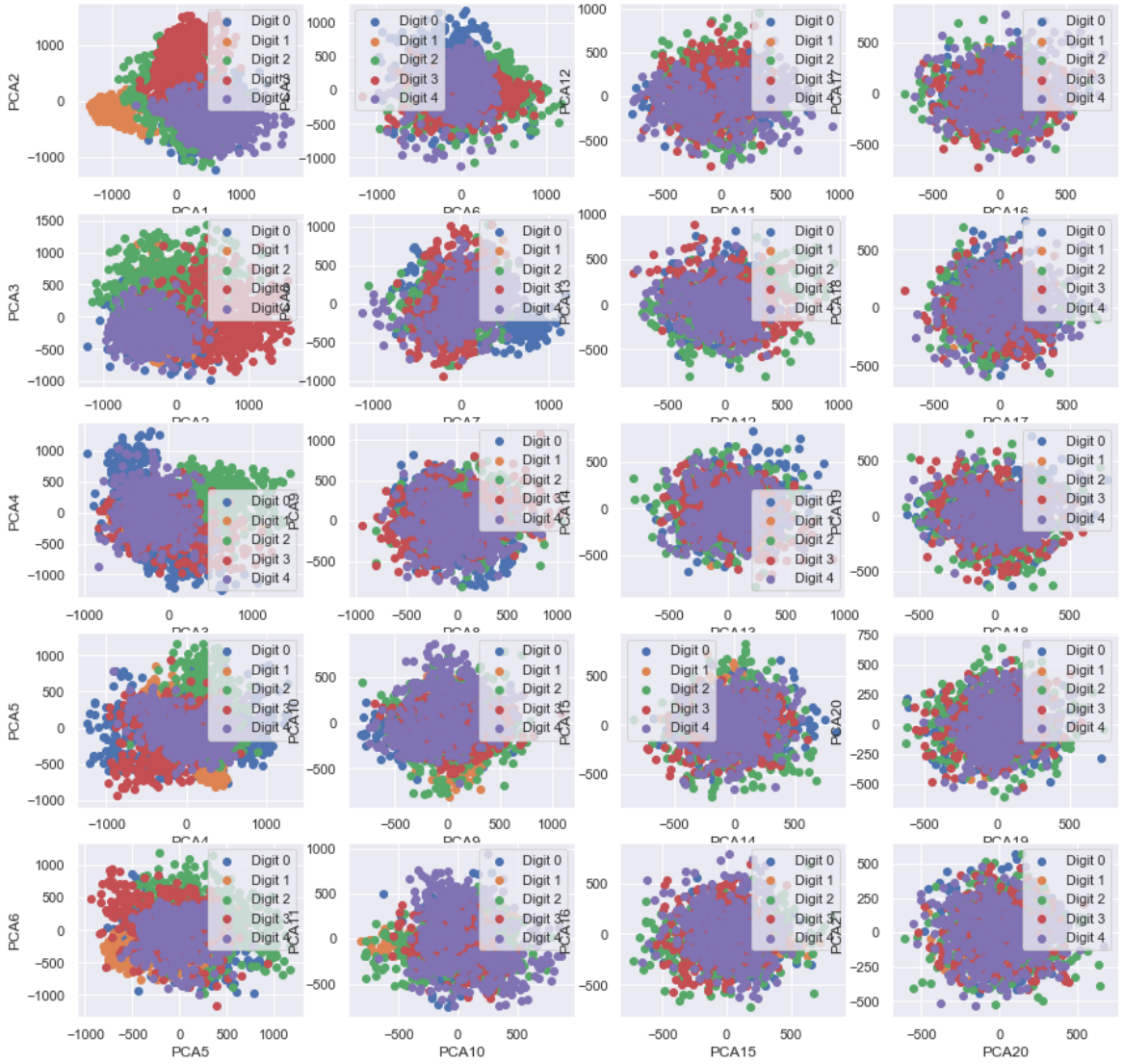
2. 90% coverage could be acheived using nearly 70 components.
3. 75% coverage could be acheived using 22 components.

If we look at the slope of the graph, we can see that for the first 10 components, it is very steep beyond which it starts curving. For choosing a cut-off, 20 components would be an optimal choice as it would cover nearly 73% of the data.

In my view, any value between 10-20 components would be the good choice. This would also be dependent of the test data and the requirement of the accuracy.

In [250]:

```
from sklearn.decomposition import PCA
pca = PCA()
X_pca = pca.fit_transform(X)

fig, axs = plt.subplots(5, 4, figsize=(15,15))

i = 1
for j in range(4):
    for k in range(5):
        for l in range(5):
            axs[k, j].scatter(X_pca[y ==l,i],X_pca[y == l,i+1], label = 'Digi
        xlb = 'PCA'+str(i)
        ylb = 'PCA'+str(i+1)
        axs[k, j].set(xlabel= xlb , ylabel= ylb)
        axs[k, j].legend()
        k = k+1
        i = i+1
```



After looking at the plots, it can be said that

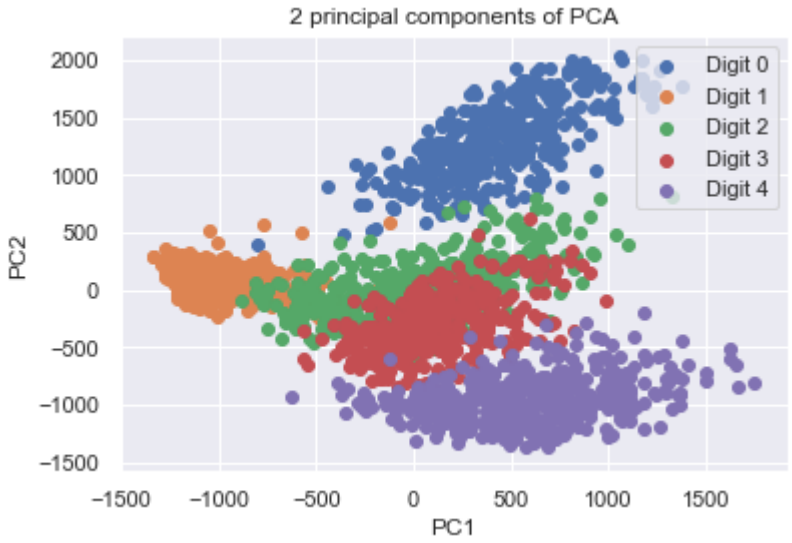
1. Uptil 5 components, the data is visually separable

2. PC 7 seems to be the last plot were a boundry line could separate the data.
3. Beyond PC 15, digit 3 and 4 data dominates the other datapoints which makes it difficult to read it.

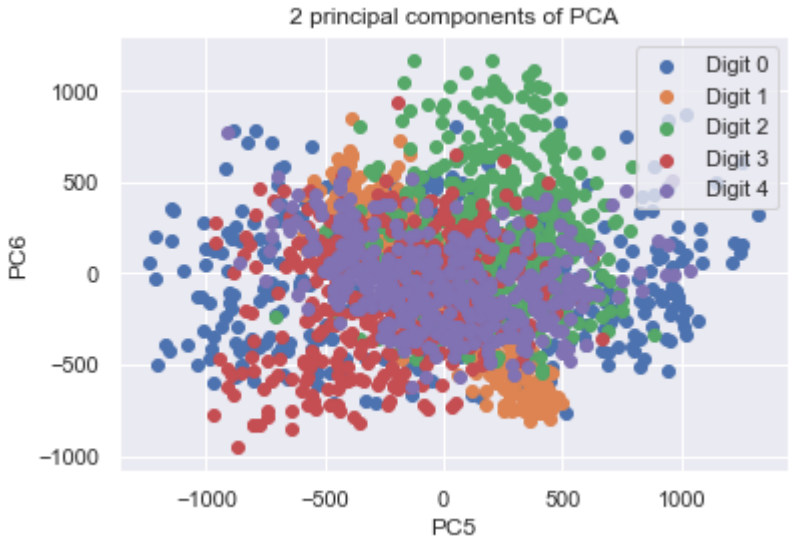
From the visual interpretation, it can be said that 10 components would be sufficient to impart all the useful information because as the number of components would increase, the variance between the data points would start decreasing as a result of which, it would be too difficult to separate them.

Comparing the results from the previous section and understanding the plots of first 20 eigenvalues, we can say that first 10 components seem to imaprt the maximum information and could be sufficient to train the model.

In [251]:
```
for l in range(0,5):
    plt.scatter(X_pca[y ==l,1],X_pca[y == l,0], label = 'Digit '+str(l), c =

plt.title('2 principal components of PCA')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend(loc = "upper right")
plt.show()
```



In [252]:
```
for l in range(5):
    plt.scatter(X_pca[y ==l,4],X_pca[y == l,5], label = 'Digit '+str(l), c =

plt.title('2 principal components of PCA')
plt.xlabel('PC5')
plt.ylabel('PC6')
plt.legend(loc = "upper right")
plt.show()
```



The comparision is as follows:

1. The PCA1 & PCA2 are easily distinguishable and all the classes can be separated. The data has high variance because of which the distinction seems clean.
2. The PCA5 & PCA6 are more messed and difficult to dsitinguish. The data has lower variance compared to PCA1 & PCA2 because of which there is lot of overlap.
3. In the first plot, classes can be classified but in the second plot the points are all over the place and no boundry can separate the. Points for digit 4 seem clustered but even then there is a lot of overlap which may lead to misclassification.

In [253]:

```
%%time
import timeit
def PCA(X,y,n_components):
    X_mean = np.mean(X,axis=0)
    X_centralized = (X - X_mean)
    X_centralized = np.matrix(X_centralized)
    X_centralized = X_centralized.T
    r_m = np.matmul(X_centralized.T,X_centralized)
    l_m = np.matmul(X_centralized,X_centralized.T)

    r_m_val, r_m_vec = np.linalg.eigh(r_m)
    eig_val = np.flip(r_m_val)
    r_m_vec = np.flip(r_m_vec,axis = 1)
    V = r_m_vec[:,0:n_components]

    l_m_val, l_m_vec = np.linalg.eigh(l_m)
    l_m_vec = np.flip(l_m_vec,axis=1)
    U = l_m_vec[:,0:n_components]

    Sigma_vec = np.sqrt(eig_val)
    Sigma_vec = np.nan_to_num(Sigma_vec)
    Sigma_mat = np.diag(Sigma_vec)

    start = timeit.timeit()
    direct_pca_out = np.matmul(U.T,X_centralized)
    end = timeit.timeit()
    running_time =  end - start

    final_pca_out = direct_pca_out.T
    return final_pca_out,running_time

PCA_out,time_taken_PCA = PCA(X,y,2)
```

```
C:\Users\hp\Anaconda3\lib\site-packages\ipykernel_launcher.py:21: RuntimeWa
rning: invalid value encountered in sqrt

Wall time: 5 s
```

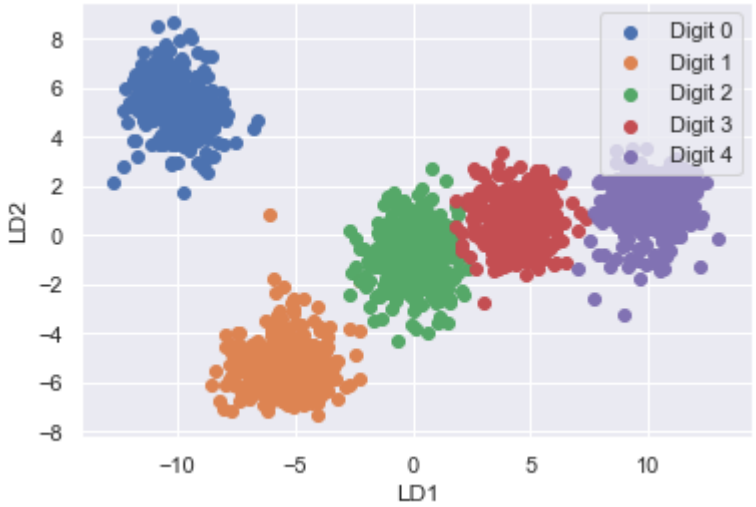
In [254]:
```
%%time

import timeit
def dual_PCA(X,y,n_components):
    X_mean = np.mean(X,axis=0)
    X_centralized = (X - X_mean)
    X_centralized = np.matrix(X_centralized)
    X_centralized = X_centralized.T      ### d*n matrix is X_centralized now
    r_m = np.matmul(X_centralized.T,X_centralized)
    l_m = np.matmul(X_centralized,X_centralized.T)

    r_m_val, r_m_vec = np.linalg.eigh(r_m)
    eig_val = np.flip(r_m_val)
    r_m_vec = np.flip(r_m_vec,axis = 1)
    V = r_m_vec[:,0:n_components]

    l_m_val, l_m_vec = np.linalg.eigh(l_m)
    l_m_vec = np.flip(l_m_vec,axis=1)
    U = l_m_vec[:,0:n_components]
    Sigma_vec = np.sqrt(eig_val)
    #Sigma_vec = np.nan_to_num(Sigma_vec)  # no need to do this part nan to r
    Sigma_vec = Sigma_vec[0:n_components]
    Sigma_mat = np.diag(Sigma_vec)

    start = timeit.timeit()
    dual_pca_out = np.matmul(Sigma_mat,V.T) #  gives p*n traspose it to get t
    end = timeit.timeit()
    final_dual_pca_out = dual_pca_out.T
    running_time = start - end

    return final_dual_pca_out,running_time

dual_PCA_out,time_taken_dual_PCA = dual_PCA(X,y,2)
```

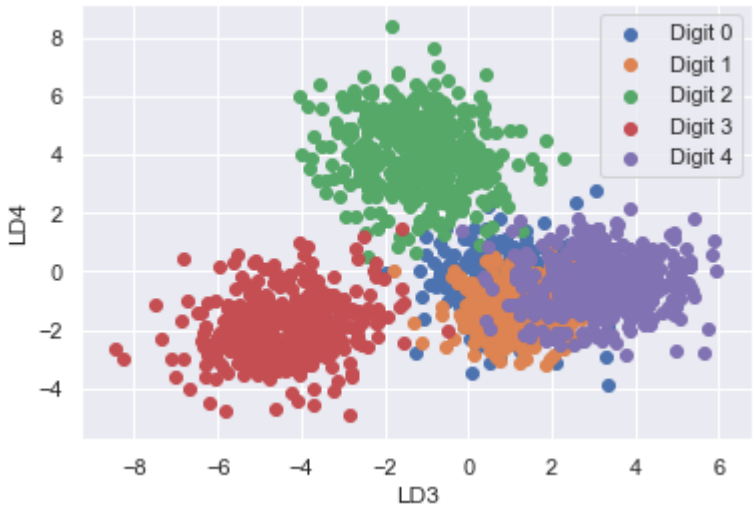
```
C:\Users\hp\Anaconda3\lib\site-packages\ipykernel_launcher.py:19: RuntimeWa
rning: invalid value encountered in sqrt
```

As we can see from the wall time that the time for computation of PCA was lesser than that of dual PCA. This is because the sample size for this model is very large and dual PCA is dependent on the dimensionality and number of samples of the model. In our case, the sample size>dimensionality of the model. Therefore dual PCA is not a good option. Though PCA will be computationally complex yet it seems to be a better option over dual PCA.

In [255]:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
lda = LDA()
lda.fit(X,y)
X_lda = lda.transform(X)

for i in range(3):
    for l in range(5):
        plt.scatter(X_lda[y ==l,i],X_lda[y == l,i+1], label = 'Digit '+str(l)

    plt.xlabel('LD'+str(i+1))
    plt.ylabel('LD'+str(i+2))
    plt.legend(loc = "upper right")
    plt.show()
```





The data is known for 5 classes and it can be seen that for LDA1 vs LDA2, the distinction is quite prevelant. All the data has been classified into labels of known classes. The same can be said for LDA 2 vs LDA 3 where the data can be classified as well with slight overlapping between digits 2 and 3. For the third plot, the overlap is quite prevalent and only digita 2 and 3 can be distinguished. From the plots we can say that first 3 components are sufficient to separate the data and classify it under the defined classes.

LDA vs PCA

1. PCA and LDA are dimensionality reduction techniques with PCA being used mostly on the unsupervised data and LDA being used on supervised data.
2. It can be observed from the graphs that LDA has proved to be a better algorithm for reducing the dimensionality and classifying the data into classes.
3. LDA needed only 3 components for distinction compared to PCA which required 10 components for maximum coverage of data information.
4. LDA knows the labels before hand, therefore classification is an easy task for it but for PCA, it has no idea about the outcome and has to cluster the data based on the available information.
5. Since in our case, the outcomes are known, LDA would be a better model to work with for dimensionality reduction. PCA can be used if we have a dataset for which the outcomes are unknown.

Question 3

In [256]:
```
from sklearn.decomposition import PCA
import numpy as np
import pandas as pd

data = pd.read_csv("DataB.csv")
```

In [257]:
```
X = data.iloc[:,:-1].values #all columns except the last one
y = data.iloc[:,-1].values #only the last column
```

In [258]:
```
%%time
from sklearn.decomposition import KernelPCA
transformer = KernelPCA(n_components=2, kernel='rbf',random_state=42)
X_transformed_PCAKernel = transformer.fit_transform(X)
```

```
Wall time: 1.31 s
```

In [259]:
```
%%time
from sklearn.manifold import Isomap
embedding = Isomap(n_components=2)
X_transformed_Isomap = embedding.fit_transform(X)
```

```
Wall time: 9.37 s
```

In [260]:
```
%%time
from sklearn.manifold import LocallyLinearEmbedding
embedding = LocallyLinearEmbedding(n_components=2,random_state=42)
X_transformed_LocallyLinearEmbedding = embedding.fit_transform(X)
```

```
Wall time: 7 s
```

In [261]:
```
%%time
from sklearn.manifold import SpectralEmbedding
embedding = SpectralEmbedding(n_components=2,random_state=42)
X_transformed_SpectralEmbedding = embedding.fit_transform(X)
```

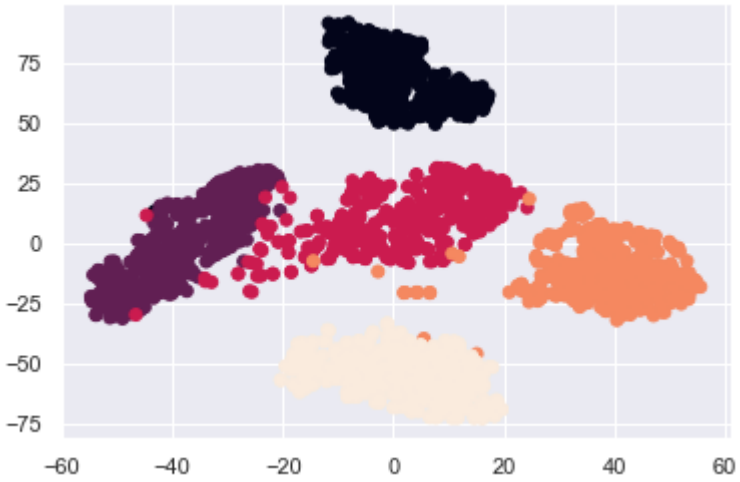
```
Wall time: 9.7 s
```

In [262]:
```
%%time
from sklearn.manifold import TSNE
embedding = TSNE(n_components=2,random_state=42)
X_transformed_TSNE = embedding.fit_transform(X)
```

```
Wall time: 27.8 s
```

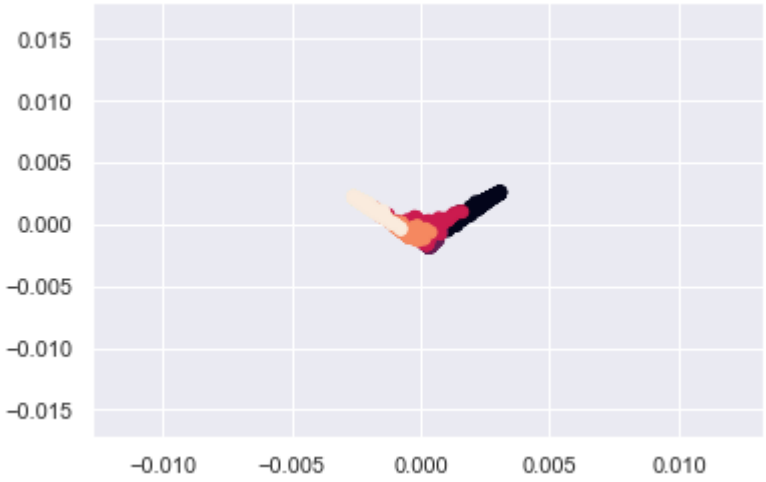
In [263]:
```
plt.scatter(X_transformed_TSNE[:,0],X_transformed_TSNE[:,1],c=y)
```

Out[263]: <matplotlib.collections.PathCollection at 0x20f889a4288>



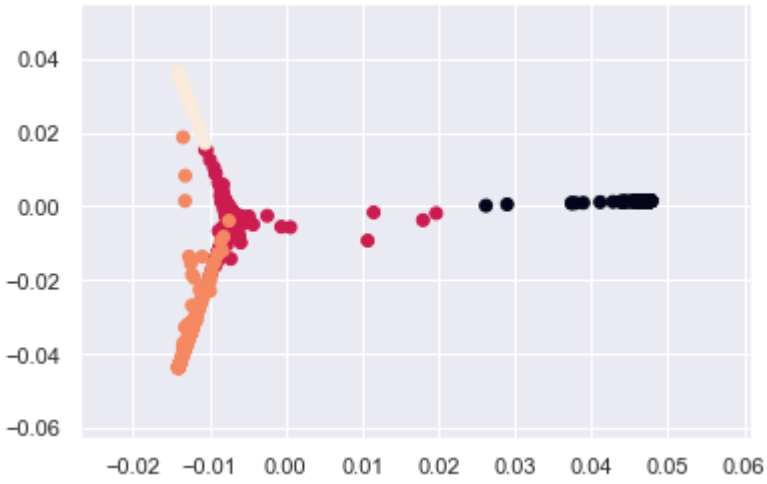
In [264]:
```
_transformed_SpectralEmbedding[:,0],X_transformed_SpectralEmbedding[:,1],c=y)
```

Out[264]: <matplotlib.collections.PathCollection at 0x20f8899a348>



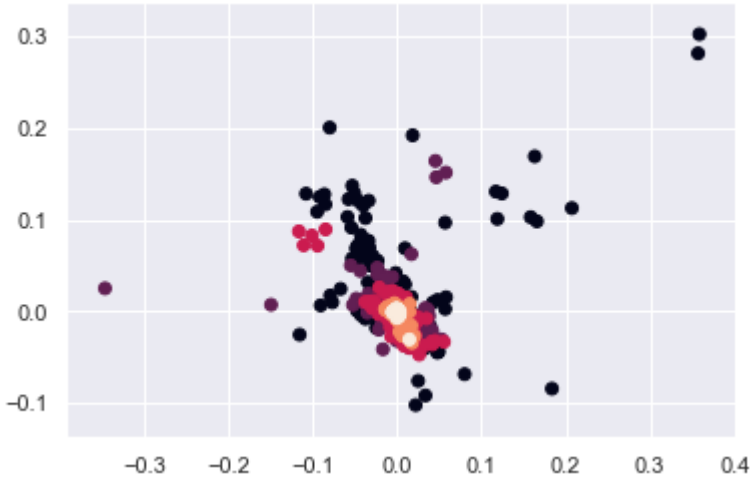
In [265]:
```
ed_LocallyLinearEmbedding[:,0],X_transformed_LocallyLinearEmbedding[:,1],c=y)
```

Out[265]: <matplotlib.collections.PathCollection at 0x20f88913d88>



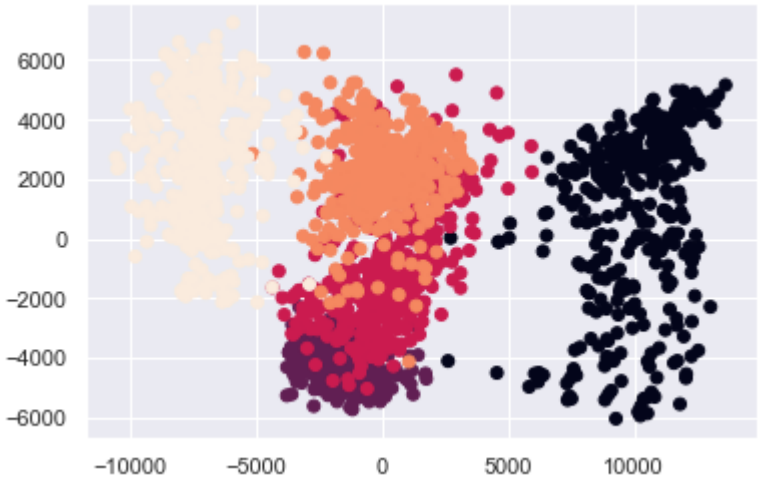
In [266]:
```
plt.scatter(X_transformed_PCAKernel[:,0],X_transformed_PCAKernel[:,1],c=y)
```

Out[266]: <matplotlib.collections.PathCollection at 0x20f88a91f88>



In [267]:
```
plt.scatter(X_transformed_Isomap[:,0],X_transformed_Isomap[:,1],c=y)
```

Out[267]:
```
<matplotlib.collections.PathCollection at 0x20f88a81c48>
```



In [268]:
```
from sklearn import neighbors
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
ran = 42
X_train2, X_test2, y_train2, y_test2 = train_test_split(X_transformed_TSNE, y
```

In [269]:
```
from sklearn.neighbors import KNeighborsClassifier

neigh_color = neighbors.KNeighborsClassifier(n_neighbors=10)
neigh_color.fit(X_train2, y_train2)
y_pred = neigh_color.predict(X_test2)
accuracy_score(y_test2, y_pred)
```

Out[269]:
```
0.9782608695652174
```

In [270]:
```
ran = 42
X_train2, X_test2, y_train2, y_test2 = train_test_split(X_transformed_Isomap,
```

In [271]:
```
from sklearn.neighbors import KNeighborsClassifier

neigh_color = neighbors.KNeighborsClassifier(n_neighbors=10)
neigh_color.fit(X_train2, y_train2)
y_pred = neigh_color.predict(X_test2)
accuracy_score(y_test2, y_pred)
```

Out[271]:
```
0.9130434782608695
```

Analysis

1. From the graphs, we can say tSNE successfully classified the data into groups of respective classes. The other algorithms were unable to properly classify the data. Isomap was also able

to classify the data to some extent but we can see some overlapping. The other three were unable to reproduce all the 5 classes as well.

2. Taking about the time of computation, Kernal PCA was the fastest with 681ms while tSNE was the slowest with 26.2s. The Isomap also computed the results in 7.89s which is faster than tSNE
3. To choose which one is better between these two, KNN was implemented for k=10 for both the cases. tSNE produces an accuracy of 97.8% whereas Isomap produced an accuracy of 91.3%.

From these three observations and results produced, we can conclude that Tsne is the best method.