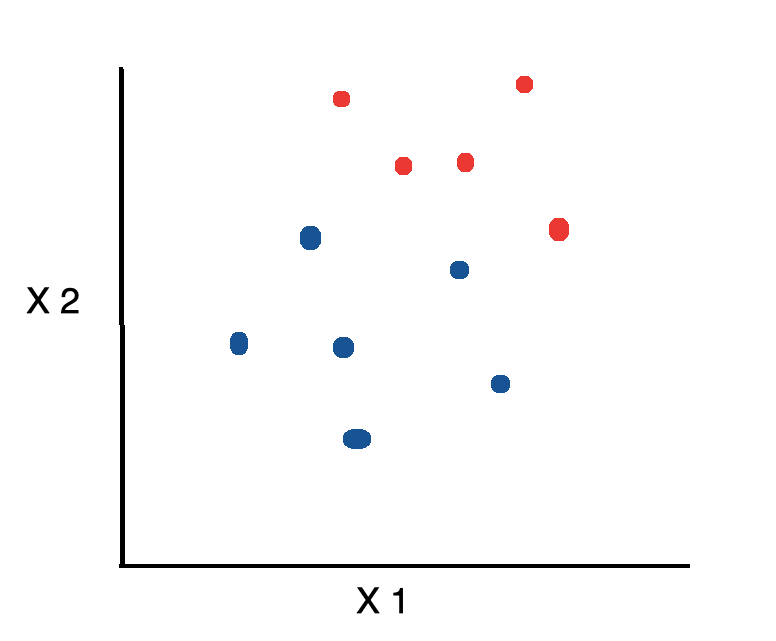
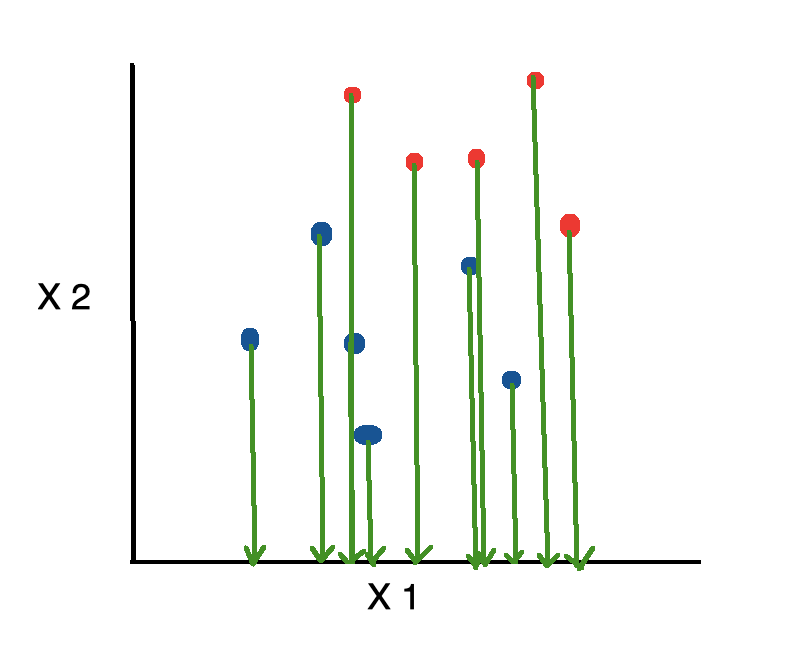
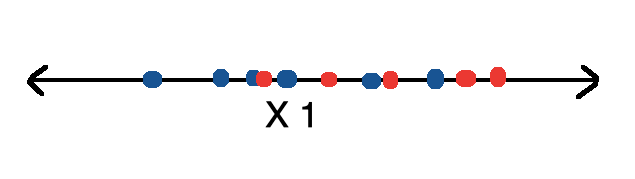
# Linear Discriminant Analysis In Python

Linear Discriminant Analysis (LDA) is a dimensionality reduction technique. As the name implies dimensionality reduction techniques reduce the number of dimensions (i.e. variables) in a dataset while retaining as much information as possible. For instance, suppose that we plotted the relationship between two variables where each color represent a different class.

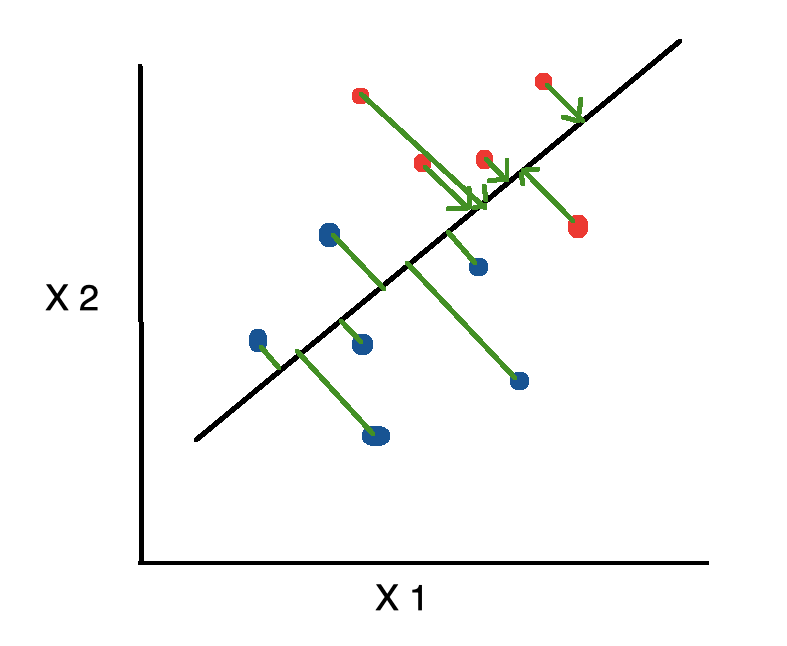


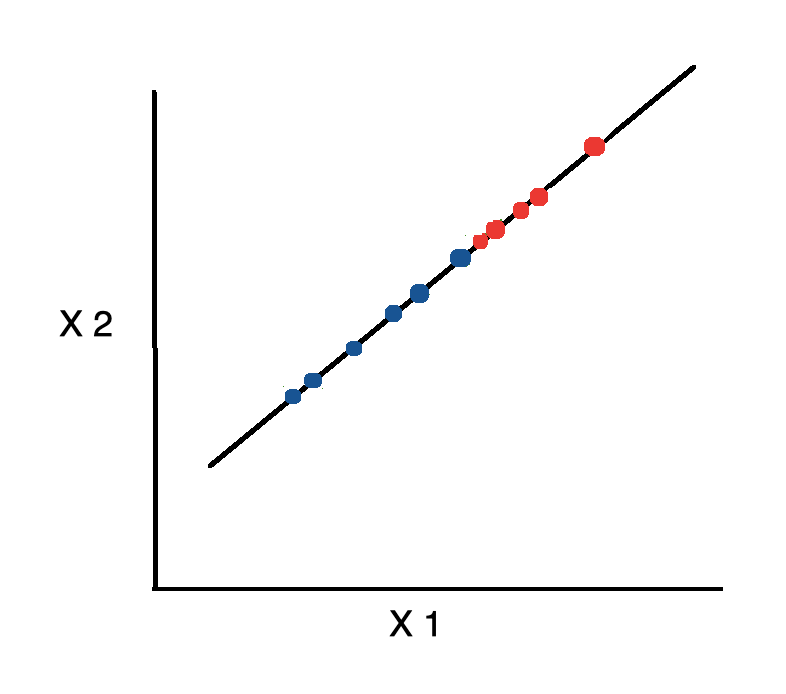
If we’d like to reduce the number of dimensions down to 1, one approach would be to project everything on to the x-axis.

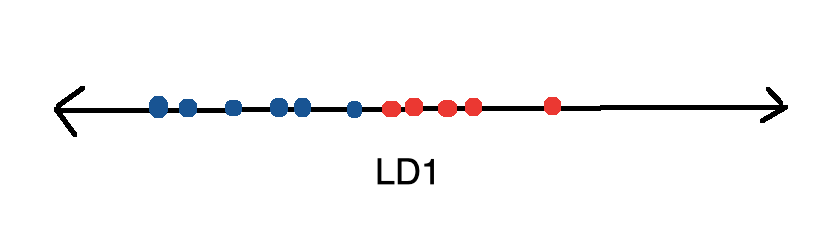




This is bad because it disregards any useful information provided by the second feature. On the other hand, Linear Discriminant Analysis, or LDA, uses the information from both features to create a new axis and projects the data on to the new axis in such a way as to minimizes the variance and maximizes the distance between the means of the two classes.







**Code**

Let’s see how we could go about implementing Linear Discriminant Analysis from scratch using Python. To start, import the following libraries.

from sklearn.datasets import load\_wine  
import pandas as pd  
import numpy as np  
np.set\_printoptions(precision=4)  
from matplotlib import pyplot as plt  
import seaborn as sns  
sns.set()  
from sklearn.preprocessing import LabelEncoder  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import confusion\_matrix

In the proceeding tutorial, we’ll be working with the wine dataset which can be obtained from the UCI machine learning repository. Fortunately, the scitkit-learn library provides a wrapper function for downloading and

wine = load\_wine()X = pd.DataFrame(wine.data, columns=wine.feature\_names)  
y = pd.Categorical.from\_codes(wine.target, wine.target\_names)

The dataset contains 178 rows of 13 columns each.

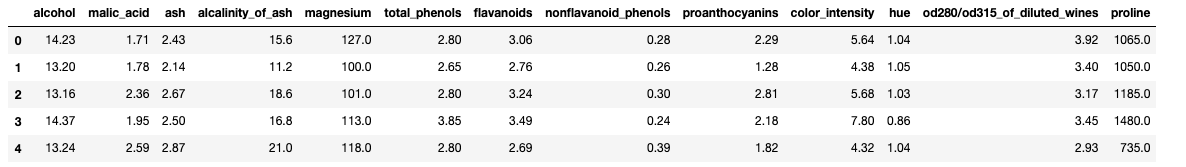
X.shape





The features are composed of various characteristics such as the magnesium and alcohol content of the wine.

X.head()



There are 3 different kinds of wine.

wine.target\_names



We create a DataFrame containing both the features and classes.

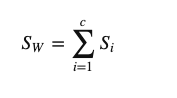
df = X.join(pd.Series(y, name='class'))

**Linear Discriminant Analysis**can be broken up into the following steps:

1. Compute the within class and between class scatter matrices
2. Compute the eigenvectors and corresponding eigenvalues for the scatter matrices
3. Sort the eigenvalues and select the top ***k***
4. Create a new matrix containing eigenvectors that map to the ***k*** eigenvalues
5. Obtain the new features (i.e. LDA components) by taking the dot product of the data and the matrix from step 4

**Within Class Scatter Matrix**

We calculate the *within class scatter matrix*using the following formula.



where ***c*** is the total number of distinct classes and

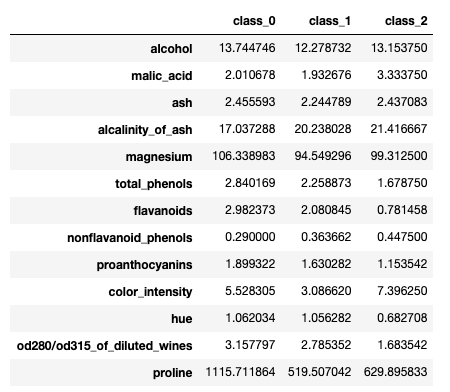




where ***x*** is a sample (i.e. row) and ***n*** is the total number of samples with a given class.

For every class, we create a vector with the means of each feature.

class\_feature\_means = pd.DataFrame(columns=wine.target\_names)for c, rows in df.groupby('class'):  
 class\_feature\_means[c] = rows.mean()class\_feature\_means

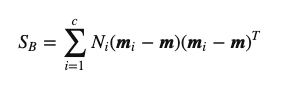


Then, we plug the mean vectors (***mi***) into the equation from before in order to obtain the within class scatter matrix.

within\_class\_scatter\_matrix = np.zeros((13,13))for c, rows in df.groupby('class'):rows = rows.drop(['class'], axis=1)  
   
 s = np.zeros((13,13))for index, row in rows.iterrows():  
 x, mc = row.values.reshape(13,1), class\_feature\_means[c].values.reshape(13,1)  
   
 s += (x - mc).dot((x - mc).T)  
   
 within\_class\_scatter\_matrix += s

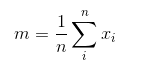
**Between Class Scatter Matrix**

Next, we calculate the *between class scatter matrix*using the following formula.



where





feature\_means = df.mean()between\_class\_scatter\_matrix = np.zeros((13,13))for c in class\_feature\_means:   
 n = len(df.loc[df['class'] == c].index)  
   
 mc, m = class\_feature\_means[c].values.reshape(13,1), feature\_means.values.reshape(13,1)  
   
 between\_class\_scatter\_matrix += n \* (mc - m).dot((mc - m).T)

Then, we solve the generalized eigenvalue problem for

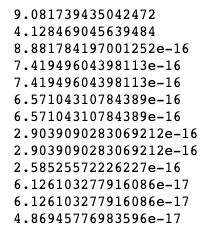


to obtain the linear discriminants.

eigen\_values, eigen\_vectors = np.linalg.eig(np.linalg.inv(within\_class\_scatter\_matrix).dot(between\_class\_scatter\_matrix))

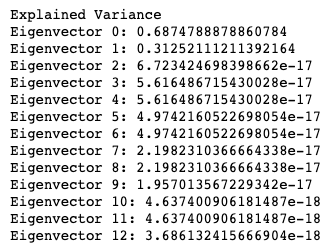
The eigenvectors with the highest eigenvalues carry the most information about the distribution of the data. Thus, we sort the eigenvalues from highest to lowest and select the first ***k*** eigenvectors. In order to ensure that the eigenvalue maps to the same eigenvector after sorting, we place them in a temporary array.

pairs = [(np.abs(eigen\_values[i]), eigen\_vectors[:,i]) for i in range(len(eigen\_values))]pairs = sorted(pairs, key=lambda x: x[0], reverse=True)for pair in pairs:  
 print(pair[0])



Just looking at the values, it’s difficult to determine how much of the variance is explained by each component. Thus, we express it as a percentage.

eigen\_value\_sums = sum(eigen\_values)print('Explained Variance')  
for i, pair in enumerate(pairs):  
 print('Eigenvector {}: {}'.format(i, (pair[0]/eigen\_value\_sums).real))



First, we create a matrix ***W***with the first two eigenvectors.

w\_matrix = np.hstack((pairs[0][1].reshape(13,1), pairs[1][1].reshape(13,1))).real

Then, we save the dot product of ***X*** and ***W*** into a new matrix***Y***.



where ***X*** is a ***n×d*** matrix with ***n*** samples and ***d*** dimensions, and ***Y*** is a ***n×k***matrix with ***n*** samples and ***k*** ( k<n) dimensions. In other words, ***Y*** is composed of the LDA components, or said yet another way, the new feature space.

X\_lda = np.array(X.dot(w\_matrix))

matplotlib can’t handle categorical variables directly. Thus, we encode every class as a number so that we can incorporate the class labels into our plot.

le = LabelEncoder()y = le.fit\_transform(df['class'])

Then, we plot the data as a function of the two LDA components and use a different color for each class.

plt.xlabel('LD1')  
plt.ylabel('LD2')plt.scatter(  
 X\_lda[:,0],  
 X\_lda[:,1],  
 c=y,  
 cmap='rainbow',  
 alpha=0.7,  
 edgecolors='b'  
)



Rather than implementing the Linear Discriminant Analysis algorithm from scratch every time, we can use the predefined LinearDiscriminantAnalysis class made available to us by the scikit-learn library.

from sklearn.discriminant\_analysis import LinearDiscriminantAnalysislda = LinearDiscriminantAnalysis()X\_lda = lda.fit\_transform(X, y)

We can access the following property to obtain the variance explained by each component.

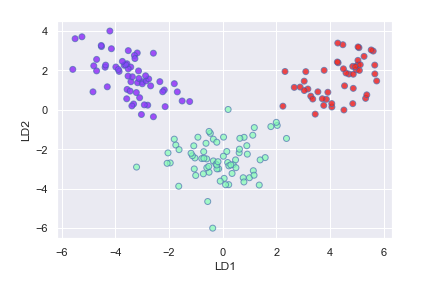
lda.explained\_variance\_ratio\_





Just like before, we plot the two LDA components.

plt.xlabel('LD1')  
plt.ylabel('LD2')plt.scatter(  
 X\_lda[:,0],  
 X\_lda[:,1],  
 c=y,  
 cmap='rainbow',  
 alpha=0.7,  
 edgecolors='b'  
)



Next, let’s take a look at how LDA compares to Principal Component Analysis or PCA. We start off by creating and fitting an instance of the PCA class.

from sklearn.decomposition import PCApca = PCA(n\_components=2)X\_pca = pca.fit\_transform(X, y)

We can access the explained\_variance\_ratio\_ property to view the percentage of the variance explained by each component.

pca.explained\_variance\_ratio\_



As we can see, PCA selected the components which would result in the highest spread (retain the most information) and not necessarily the ones which maximize the separation between classes.

plt.xlabel('PC1')  
plt.ylabel('PC2')plt.scatter(  
 X\_pca[:,0],  
 X\_pca[:,1],  
 c=y,  
 cmap='rainbow',  
 alpha=0.7,  
 edgecolors='b'  
)



Next, let’s see whether we can create a model to classify the using the LDA components as features. First, we split the data into training and testing sets.

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_lda, y, random\_state=1)

Then, we build and train a Decision Tree. After predicting the category of each sample in the test set, we create a confusion matrix to evaluate the model’s performance.

dt = DecisionTreeClassifier()dt.fit(X\_train, y\_train)y\_pred = dt.predict(X\_test)confusion\_matrix(y\_test, y\_pred)



As we can see, the Decision Tree classifier correctly classified everything in the test set.