```python
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
```

```python
wine = load_wine()
X = pd.DataFrame(wine.data, columns=wine.feature_names)
y = pd.Categorical.from_codes(wine.target, wine.target_names)
```

```python
X.shape
```

```
(178, 13)
```

```python
X.head()
```

	alcohol	malic_acid	ash	alcalinity_of_ash	magnesium	total_phenols	flavanoids	nonflavanoid_phenols	pr
0	14.23	1.71	2.43	15.6	127.0	2.80	3.06	0.28	
1	13.20	1.78	2.14	11.2	100.0	2.65	2.76	0.26	
2	13.16	2.36	2.67	18.6	101.0	2.80	3.24	0.30	
3	14.37	1.95	2.50	16.8	113.0	3.85	3.49	0.24	
4	13.24	2.59	2.87	21.0	118.0	2.80	2.69	0.39	

```python
wine.target_names
```

```
array(['class_0', 'class_1', 'class_2'], dtype='<U7')
```

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

Linear Discriminant Analysis can be broken up into the following steps: Compute the within class and between class scatter matrices Compute the eigenvectors and corresponding eigenvalues for the scatter matrices Sort the eigenvalues and select the top k Create a new matrix containing eigenvectors that map to the k eigenvalues Obtain the new features (i.e. LDA components) by taking the dot product of the data and the matrix from step 4. For every class, we create a vector with the means of each feature.

```python
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
```

	class_0	class_1	class_2
alcohol	13.744746	12.278732	13.153750
malic_acid	2.010678	1.932676	3.333750
ash	2.455593	2.244789	2.437083
alcalinity_of_ash	17.037288	20.238028	21.416667
magnesium	106.338983	94.549296	99.312500
total_phenols	2.840169	2.258873	1.678750
flavanoids	2.982373	2.080845	0.781458
nonflavanoid_phenols	0.290000	0.363662	0.447500
proanthocyanins	1.899322	1.630282	1.153542
color_intensity	5.528305	3.086620	7.396250
hue	1.062034	1.056282	0.682708
od280/od315_of_diluted_wines	3.157797	2.785352	1.683542
proline	1115.711864	519.507042	629.895833

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
```

```
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)
```

```
eigen_values, eigen_vectors = np.linalg.eig(np.linalg.inv(within_class_scatter_matrix).dot(between_class_scat
```

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])
```

```
9.884546449232964
2.9033610617160606
6.285916968291436e-16
6.285916968291436e-16
5.979482586809227e-16
5.345289989557e-16
5.345289989557e-16
2.5624197998855253e-16
2.5624197998855253e-16
2.4778227778444637e-16
5.049704088349899e-17
1.0023889228649853e-17
0.0
```

It is 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))
```

```
Explained Variance
Eigenvector 0: 0.772960426932225
Eigenvector 1: 0.22703957306777506
Eigenvector 2: 4.915516446228146e-17
Eigenvector 3: 4.915516446228146e-17
Eigenvector 4: 4.6758882027333914e-17
Eigenvector 5: 4.179956716906465e-17
Eigenvector 6: 4.179956716906465e-17
Eigenvector 7: 2.0037834944392412e-17
Eigenvector 8: 2.0037834944392412e-17
Eigenvector 9: 1.9376295736600758e-17
Eigenvector 10: 3.948811863102963e-18
Eigenvector 11: 7.838568757294643e-19
Eigenvector 12: 0.0
```

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
w_matrix
```

```
array([[-4.1562e-02,  2.0879e-01],
       [-1.6684e-02,  3.6500e-03],
       [-4.4776e-03,  4.5911e-01],
       [ 7.5359e-03, -2.4950e-02],
       [ 6.4335e-03,  3.1071e-03],
       [ 9.7213e-02,  1.9083e-01],
```

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.

```
       [illegible]
       [illegible]
```

```
X_lda = np.array(X.dot(w_matrix))
X_lda
```

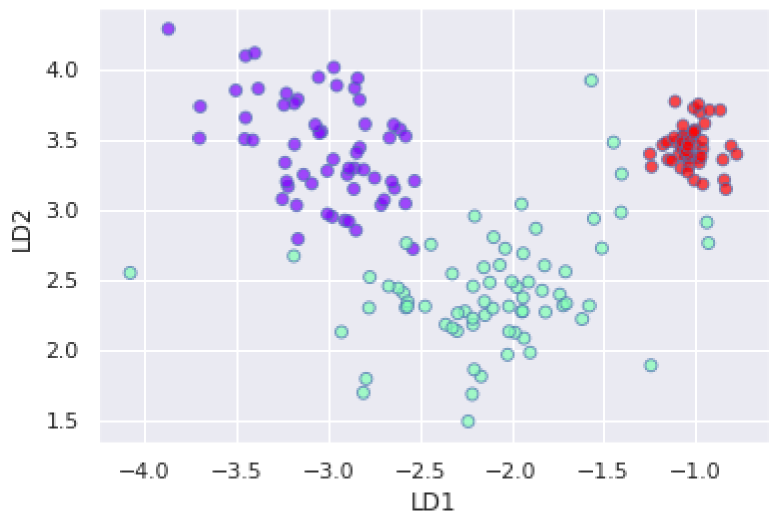
```
[illegible]
[-2.6703,  2.4536],
[-4.0763,  2.5482],
[-2.1484,  2.3438],
[-2.5665,  2.342 ],
[-2.8082,  1.6951],
[-2.5682,  2.3082],
[-2.7733,  2.5182],
[-2.2102,  2.4526],
[-2.3226,  2.1558],
[-2.0354,  2.7213],
[-1.0681,  3.5209],
[-1.1498,  3.3567],
[-1.08  ,  3.2958],
[-1.0542,  3.4031],
[-1.033 ,  3.3645],
[-1.1067,  3.4793],
[-1.0068,  3.5505],
[-1.0075,  3.2089],
[-1.0538,  3.3798],
[-0.985 ,  3.3684],
[-1.0918,  3.3882],
[-1.1791,  3.4601],
[-1.0039,  3.4614],
[-1.2402,  3.3053],
[-1.0357,  3.3117],
[-1.0564,  3.4578],
[-0.8066,  3.4534],
[-1.0643,  3.5227],
[-1.0027,  3.5641],
[-1.1572,  3.4807],
[-1.1166,  3.5132],
[-0.9805,  3.3335],
[-0.8516,  3.3555],
[-1.076 ,  3.4411],
[-0.8448,  3.2103],
[-1.073 ,  3.4821],
[-1.039 ,  3.3952],
[-1.0692,  3.5956],
[-1.2492,  3.3961],
[-1.1306,  3.3489],
[-1.0532,  3.4146],
[-0.9758,  3.6891],
[-1.0432,  3.267 ],
[-1.0349,  3.4871],
[-0.9621,  3.4335],
[-0.9611,  3.1801],
[-0.9632,  3.4894],
[-0.9695,  3.378 ],
[-1.0425,  3.4522],
[-0.951 ,  3.6104],
[-0.8351,  3.1463],
[-0.7765,  3.3949],
[-1.0098,  3.7191],
[-1.1122,  3.7663],
[-1.0118,  3.5502],
[-0.8661,  3.7044],
[-0.9247,  3.7037],
[-0.985 ,  3.7465]])
```

Matplotlib cannot 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'])
```

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

```
<matplotlib.collections.PathCollection at 0x7f3d7323ad68>
```



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 LinearDiscriminantAnalysis
lda = LinearDiscriminantAnalysis()
X_lda = lda.fit_transform(X, y)
```

```
lda.explained_variance_ratio_
```

```
array([0.6875, 0.3125])
```

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