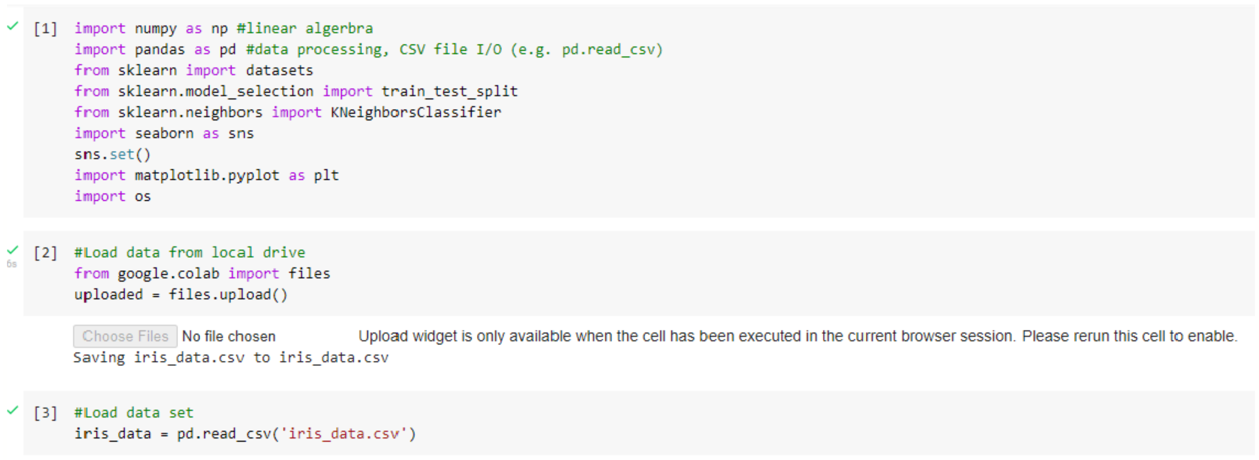
**Sanaz Salari**

**Q:** Select a challenging high-dimensional dataset from your organization or a public source (e.g., Kaggle.com) and explore the data (through EDA as well as dimensionality reduction using PCA, *t*-SNE, and UMAP) and study the impact of feature selection methods (both filters and wrappers) on the performance (both effectiveness and robustness) of the resulting machine learning model.Pair plot map:

**Answer:** I will do standard methods of feature extraction containing PCS, t-SNE and UMAP and then some demonstrations on how to implement feature selection methods in Python. To do that, I use Iris flower dataset.

## 1. import required libraries and load iris_data from drive on google colab.



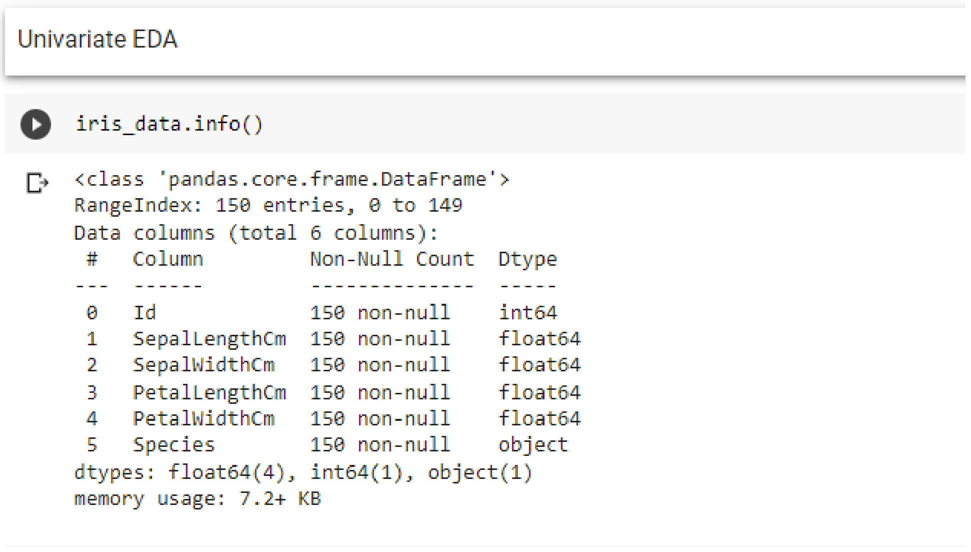
```
[1] import numpy as np #linear algerbra
    import pandas as pd #data processing, CSV file I/O (e.g. pd.read_csv)
    from sklearn import datasets
    from sklearn.model_selection import train_test_split
    from sklearn.neighbors import KNeighborsClassifier
    import seaborn as sns
    sns.set()
    import matplotlib.pyplot as plt
    import os
```

```
[2] #Load data from local drive
    from google.colab import files
    uploaded = files.upload()
```

```
Choose Files  No file chosen          Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
Saving iris_data.csv to iris_data.csv
```

```
[3] #Load data set
    iris_data = pd.read_csv('iris_data.csv')
```

## 2. Univariate EDA



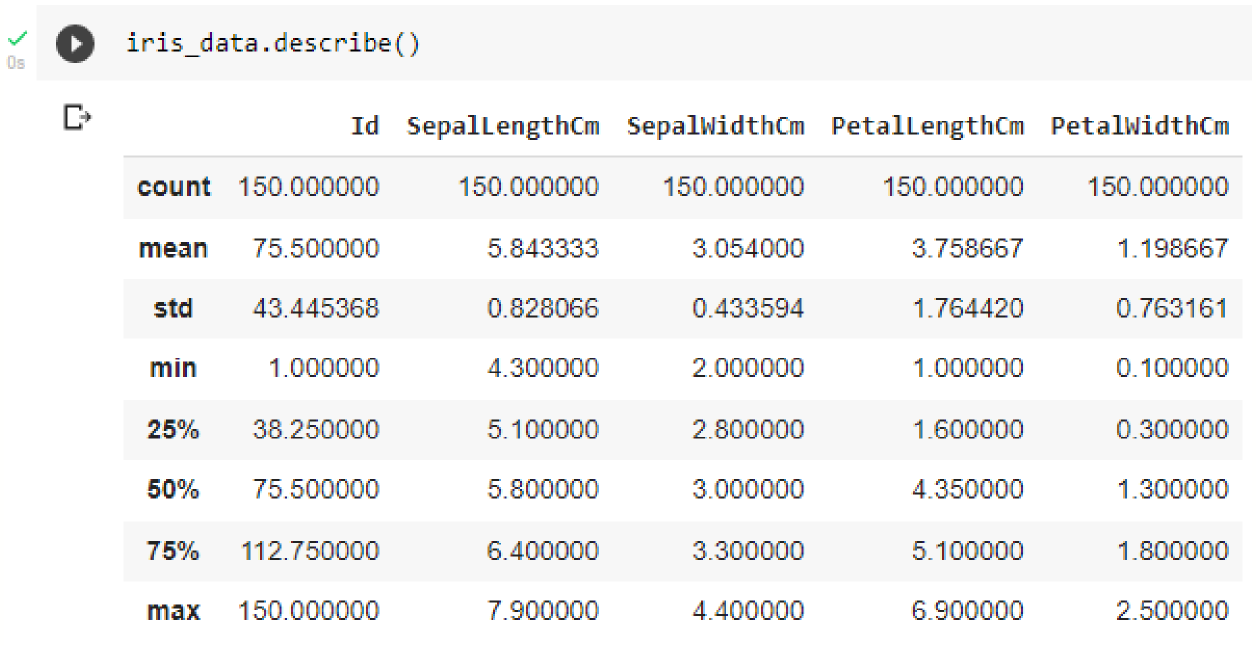
Univariate EDA

```
iris_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 6 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Id             150 non-null    int64
 1   SepalLengthCm  150 non-null    float64
 2   SepalWidthCm   150 non-null    float64
 3   PetalLengthCm  150 non-null    float64
 4   PetalWidthCm   150 non-null    float64
 5   Species        150 non-null    object
dtypes: float64(4), int64(1), object(1)
memory usage: 7.2+ KB
```

This dataset has 6 columns, 4 variables as input features named ‘SepalLengthCm’, ‘SepalWidthCm’, ‘PetalLengthCm’, and ‘PetalWidthCm’ and the output columns as “Species’. This is a classification problem with three classes called ‘Iris_setosa’, ‘Iris_Versicolor’, and ‘Iris-virginica’.

**Inspect data to see if normalization is needed or not:**

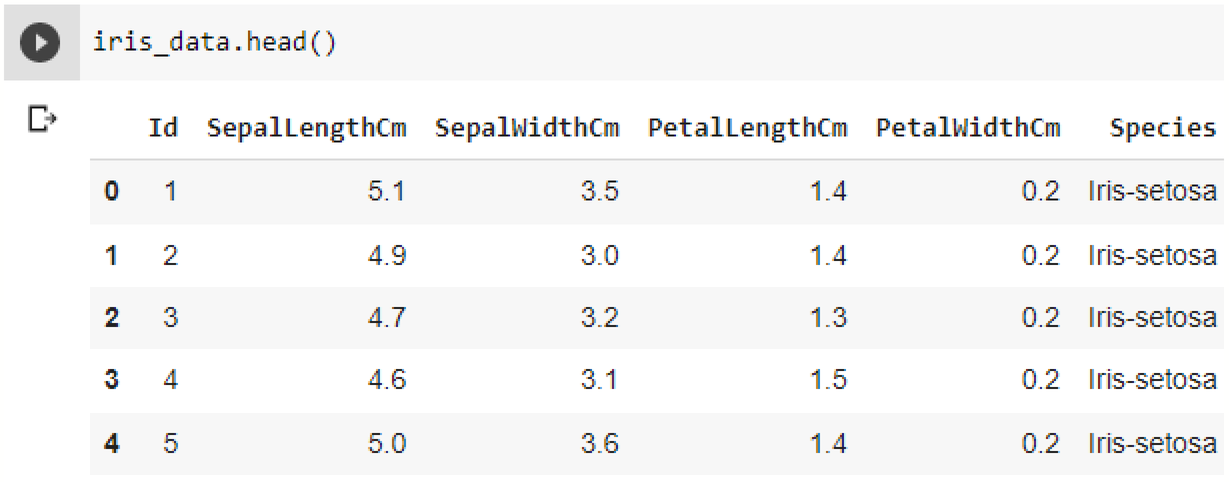


```python
iris_data.describe()
```

| | Id | SepalLengthCm | SepalWidthCm | PetalLengthCm | PetalWidthCm |
|---|---|---|---|---|---|
| **count** | 150.000000 | 150.000000 | 150.000000 | 150.000000 | 150.000000 |
| **mean** | 75.500000 | 5.843333 | 3.054000 | 3.758667 | 1.198667 |
| **std** | 43.445368 | 0.828066 | 0.433594 | 1.764420 | 0.763161 |
| **min** | 1.000000 | 4.300000 | 2.000000 | 1.000000 | 0.100000 |
| **25%** | 38.250000 | 5.100000 | 2.800000 | 1.600000 | 0.300000 |
| **50%** | 75.500000 | 5.800000 | 3.000000 | 4.350000 | 1.300000 |
| **75%** | 112.750000 | 6.400000 | 3.300000 | 5.100000 | 1.800000 |
| **max** | 150.000000 | 7.900000 | 4.400000 | 6.900000 | 2.500000 |

As we see, the distribution of each variable regarding mean, min and max values need to be normalized.

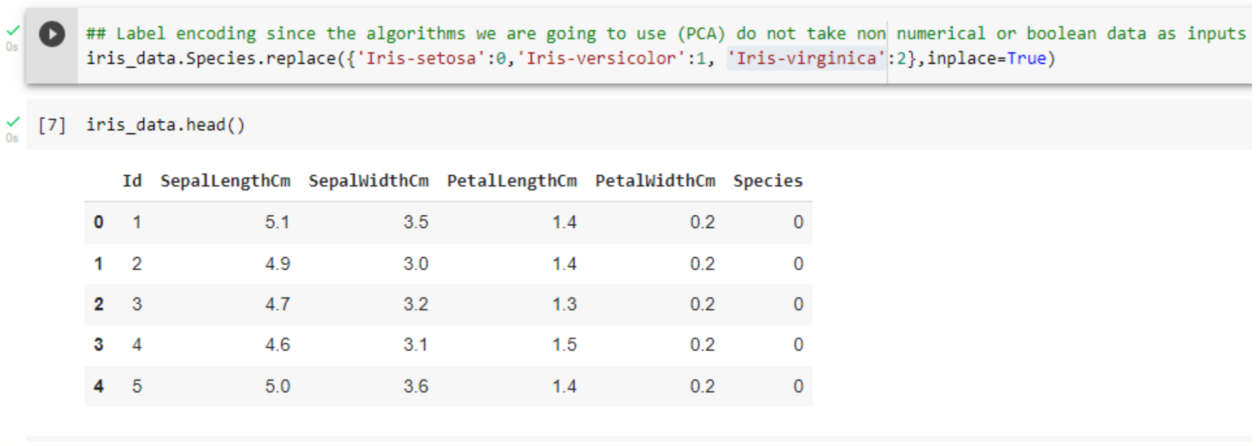
**Let’s see what the five rows of a dataset is:**



```python
iris_data.head()
```

| | Id | SepalLengthCm | SepalWidthCm | PetalLengthCm | PetalWidthCm | Species |
|---|---|---|---|---|---|---|
| **0** | 1 | 5.1 | 3.5 | 1.4 | 0.2 | Iris-setosa |
| **1** | 2 | 4.9 | 3.0 | 1.4 | 0.2 | Iris-setosa |
| **2** | 3 | 4.7 | 3.2 | 1.3 | 0.2 | Iris-setosa |
| **3** | 4 | 4.6 | 3.1 | 1.5 | 0.2 | Iris-setosa |
| **4** | 5 | 5.0 | 3.6 | 1.4 | 0.2 | Iris-setosa |

**Note: PCA does not take non-numerical or boolean data inputs. So, we need label encoding:**



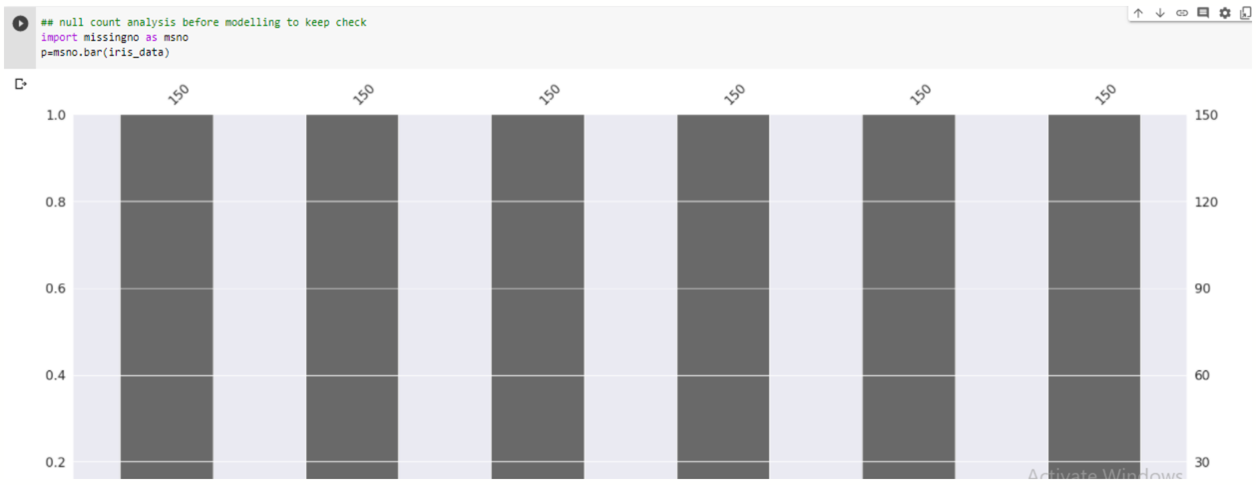
```
## Label encoding since the algorithms we are going to use (PCA) do not take non numerical or boolean data as inputs
iris_data.Species.replace({'Iris-setosa':0,'Iris-versicolor':1, 'Iris-virginica':2},inplace=True)
```

```
[7] iris_data.head()
```

| | Id | SepalLengthCm | SepalWidthCm | PetalLengthCm | PetalWidthCm | Species |
|---|---|---|---|---|---|---|
| 0 | 1 | 5.1 | 3.5 | 1.4 | 0.2 | 0 |
| 1 | 2 | 4.9 | 3.0 | 1.4 | 0.2 | 0 |
| 2 | 3 | 4.7 | 3.2 | 1.3 | 0.2 | 0 |
| 3 | 4 | 4.6 | 3.1 | 1.5 | 0.2 | 0 |
| 4 | 5 | 5.0 | 3.6 | 1.4 | 0.2 | 0 |

As we see, the label encoding was done and classes were changed from string values to int values as 0,1,2.

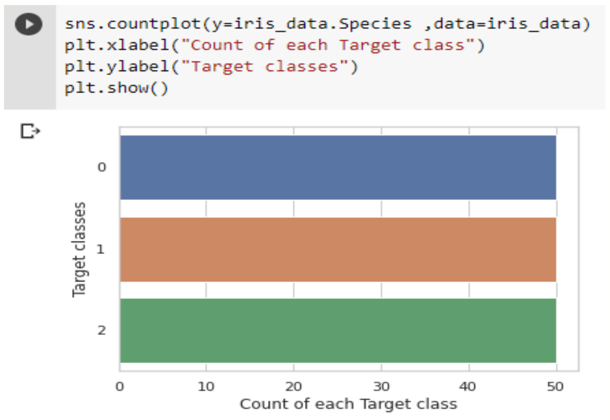
**null count analysis before modelling to keep check:**



```
## null count analysis before modelling to keep check
import missingno as msno
p=msno.bar(iris_data)
```

So, no missing data. There are no null values in none of the dataset's columns. Each of them has 150 samples with not null values.

**To check whether we have class imbalance, we have the following:**



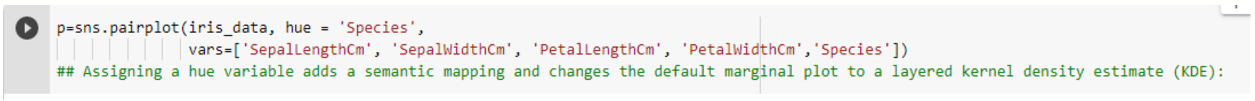
```
sns.countplot(y=iris_data.Species ,data=iris_data)
plt.xlabel("Count of each Target class")
plt.ylabel("Target classes")
plt.show()
```

No class imbalance, all target classes have equal number of rows (50 each).

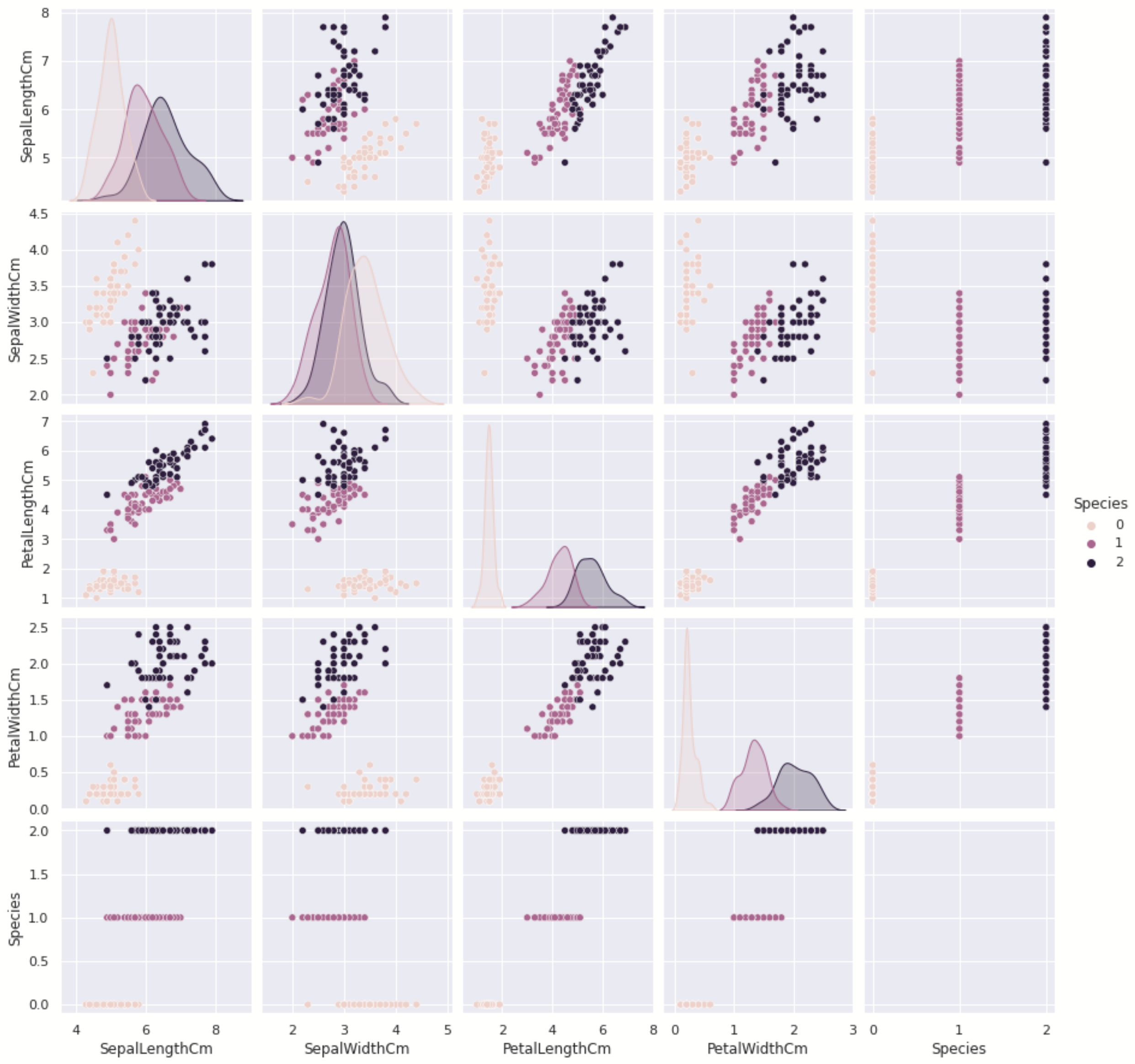
**Early Insights:**

1. 150 rows
2. 4 Independent variables to act as factors
3. All have same units of measurement (cm)

**Bivariate EDA:**

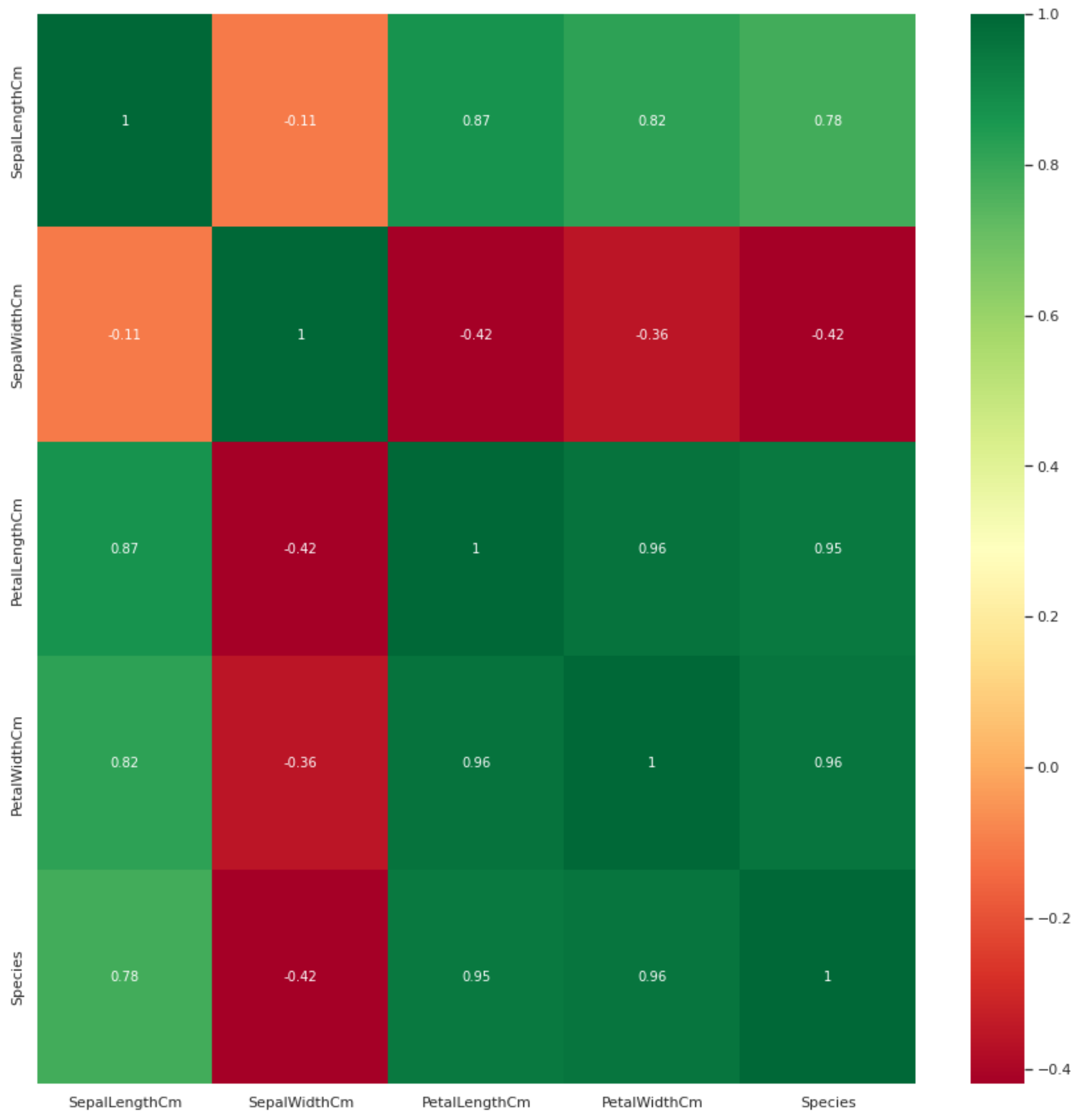


```
p=sns.pairplot(iris_data, hue = 'Species',
               vars=['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm','Species'])
## Assigning a hue variable adds a semantic mapping and changes the default marginal plot to a layered kernel density estimate (KDE):
```



As we see in this figure, there is a positive relationship between 'SepalLengthCm' and 'PetalLength' for classes 1,2 but not class 0. In other words, there is no clear relationship between the mentioned variables in class 0. Each feature has a normal distribution for each class, e.g., Petal Length has a normal distribution in three species. Also, the three classes have no clear relationship with each other. All independent variables have no clear relationship with output classes and there is a non-linear relationship.

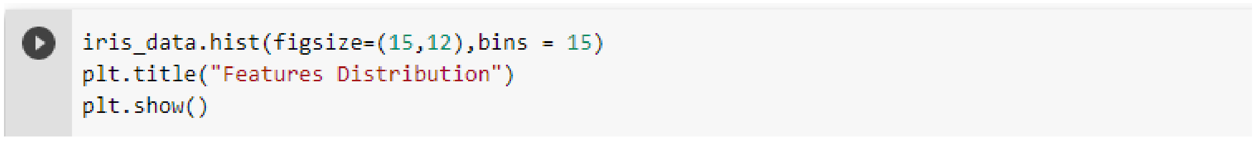
```
plt.figure(figsize=(15,15))
p=sns.heatmap(iris_data.corr(), annot=True,cmap='RdYlGn')
```



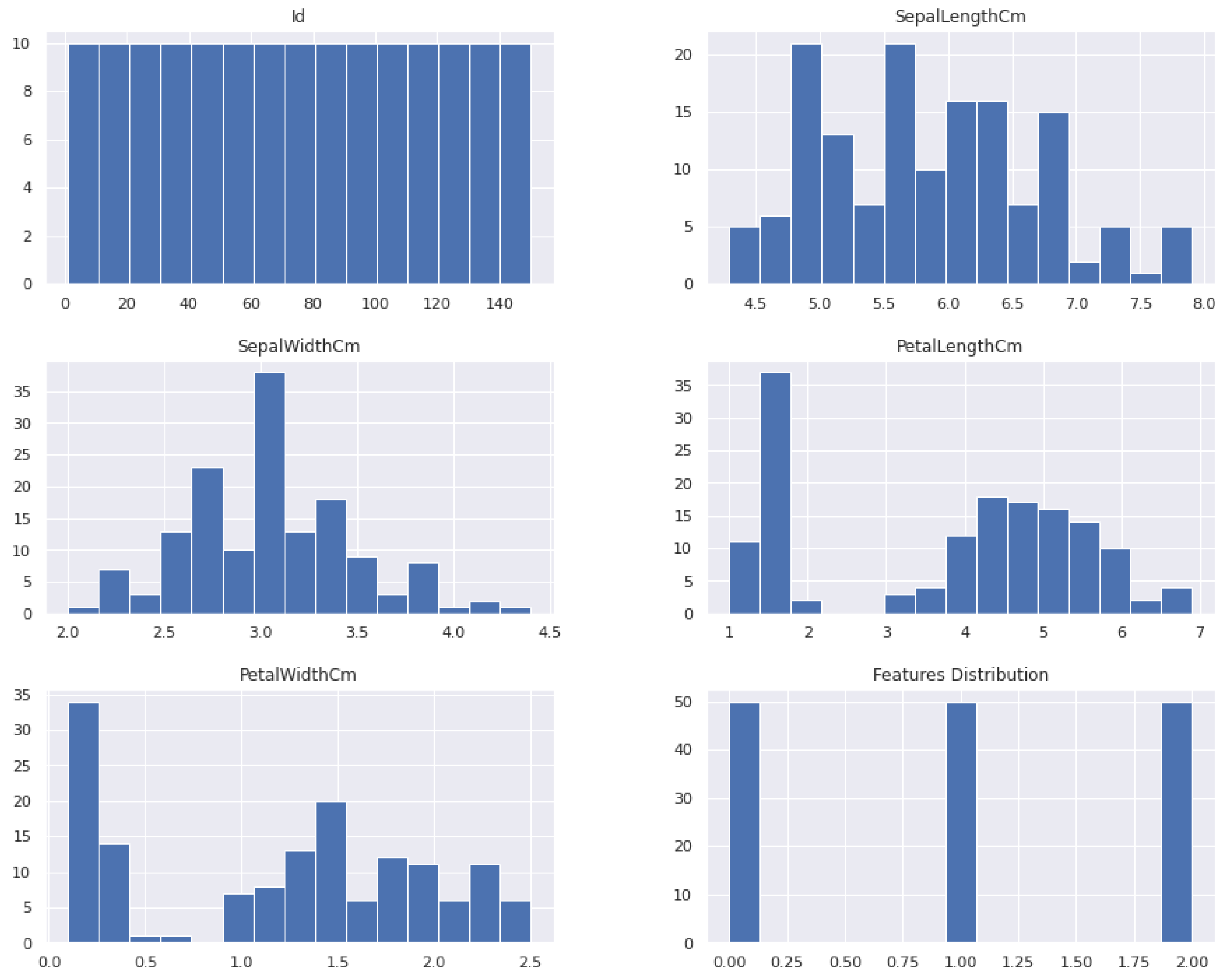
One of the biggest aims of these sort of plots and EDAs are to identify features that are not much helpful in explaining the target outcome. As we see in the above confusion matrix, 'SepalWidthCm' has the least relationship with other features and what's more important is its least influence on the 'Species' target. and seems to be less relevant in explaining the target class as compared to the other features.

So, this feature would be a candidate for removing in feature selection.

Check histogram distribution for each feature:



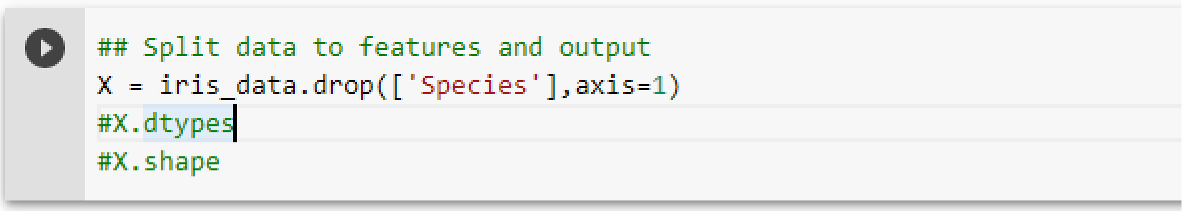
```
iris_data.hist(figsize=(15,12),bins = 15)
plt.title("Features Distribution")
plt.show()
```



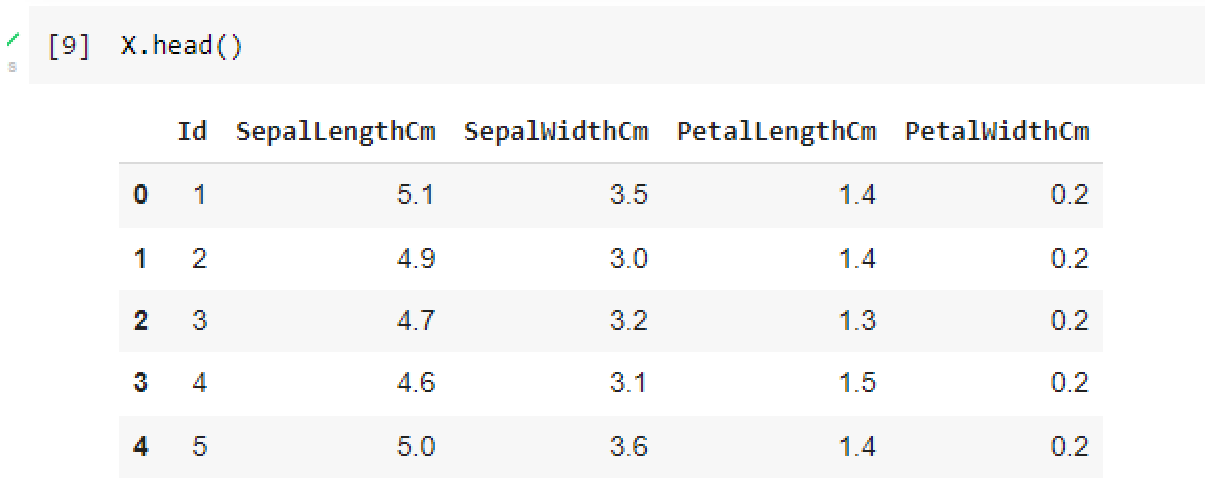
The histogram distributions don't seem to be ideal.

1. Without PCA

Split data to features and output



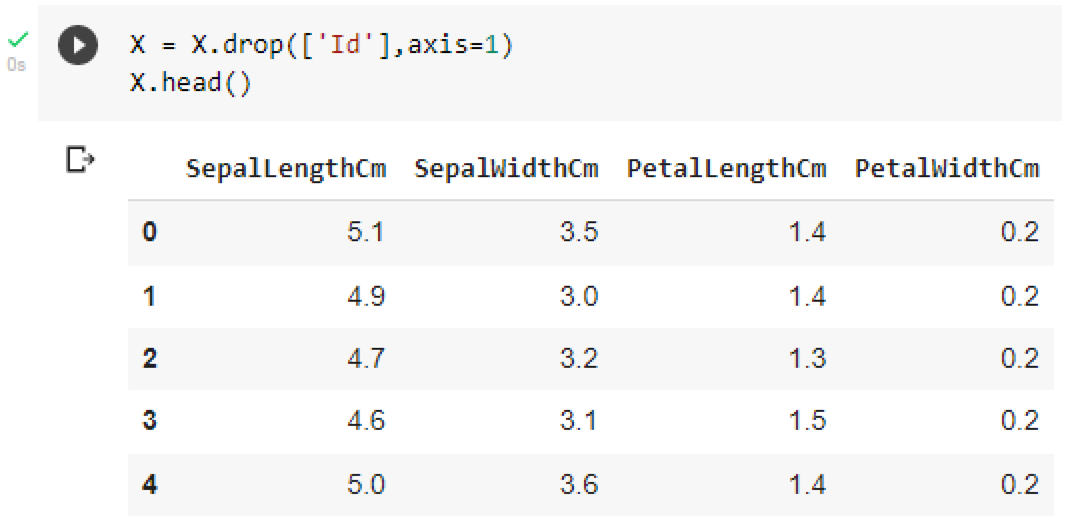
```python
## Split data to features and output
X = iris_data.drop(['Species'],axis=1)
#X.dtypes
#X.shape
```



```python
[9] X.head()
```

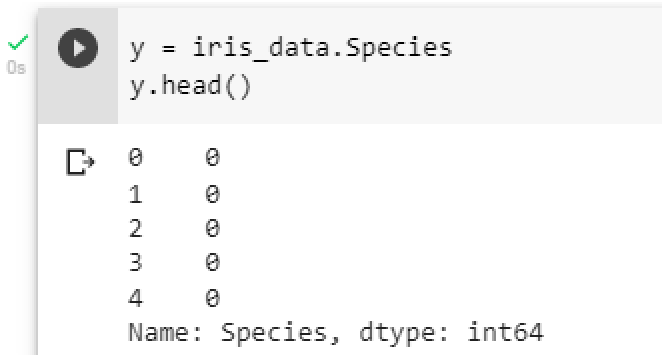
| | Id | SepalLengthCm | SepalWidthCm | PetalLengthCm | PetalWidthCm |
|---|---|---|---|---|---|
| **0** | 1 | 5.1 | 3.5 | 1.4 | 0.2 |
| **1** | 2 | 4.9 | 3.0 | 1.4 | 0.2 |
| **2** | 3 | 4.7 | 3.2 | 1.3 | 0.2 |
| **3** | 4 | 4.6 | 3.1 | 1.5 | 0.2 |
| **4** | 5 | 5.0 | 3.6 | 1.4 | 0.2 |

Drop the ‘Id’ column since it does not a required data:



```python
X = X.drop(['Id'],axis=1)
X.head()
```

| | SepalLengthCm | SepalWidthCm | PetalLengthCm | PetalWidthCm |
|---|---|---|---|---|
| **0** | 5.1 | 3.5 | 1.4 | 0.2 |
| **1** | 4.9 | 3.0 | 1.4 | 0.2 |
| **2** | 4.7 | 3.2 | 1.3 | 0.2 |
| **3** | 4.6 | 3.1 | 1.5 | 0.2 |
| **4** | 5.0 | 3.6 | 1.4 | 0.2 |



```python
y = iris_data.Species
y.head()
```

```
0    0
1    0
2    0
3    0
4    0
Name: Species, dtype: int64
```

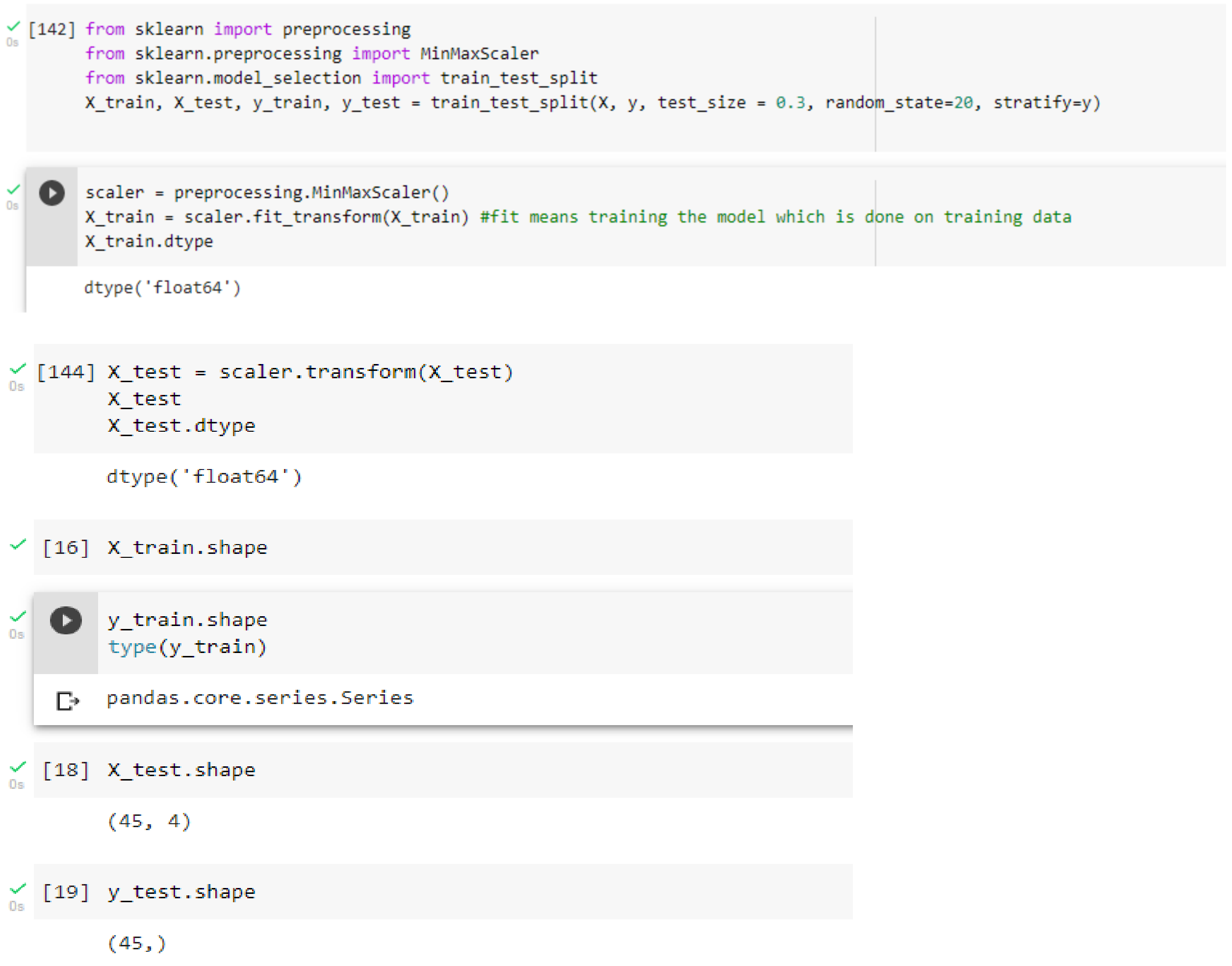
## Scaling features to a range:

A standardization is scaling features to lie between a given minimum and maximum value, often between zero and one, or so that the maximum absolute value of each feature is scaled to unit size. This can be achieved using MinMaxScaler. The motivation to use this scaling includes robustness to very small standard deviations of features and preserving zero entries in sparse data.

First, split features and labels to train and test ones, then standardize all of them.

Here is scaling an iris data matrix to the [0, 1] range:

The instance of the transformer can then be applied to both train and some new test data unseen during the fit call: the same scaling and shifting operations will be applied to be consistent with the transformation performed on the train data:



```
[142] from sklearn import preprocessing
      from sklearn.preprocessing import MinMaxScaler
      from sklearn.model_selection import train_test_split
      X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state=20, stratify=y)
```

```
scaler = preprocessing.MinMaxScaler()
X_train = scaler.fit_transform(X_train) #fit means training the model which is done on training data
X_train.dtype
```

```
dtype('float64')
```

```
[144] X_test = scaler.transform(X_test)
      X_test
      X_test.dtype
```

```
dtype('float64')
```

```
[16] X_train.shape
```

```
y_train.shape
type(y_train)
```

```
pandas.core.series.Series
```

```
[18] X_test.shape
```

```
(45, 4)
```

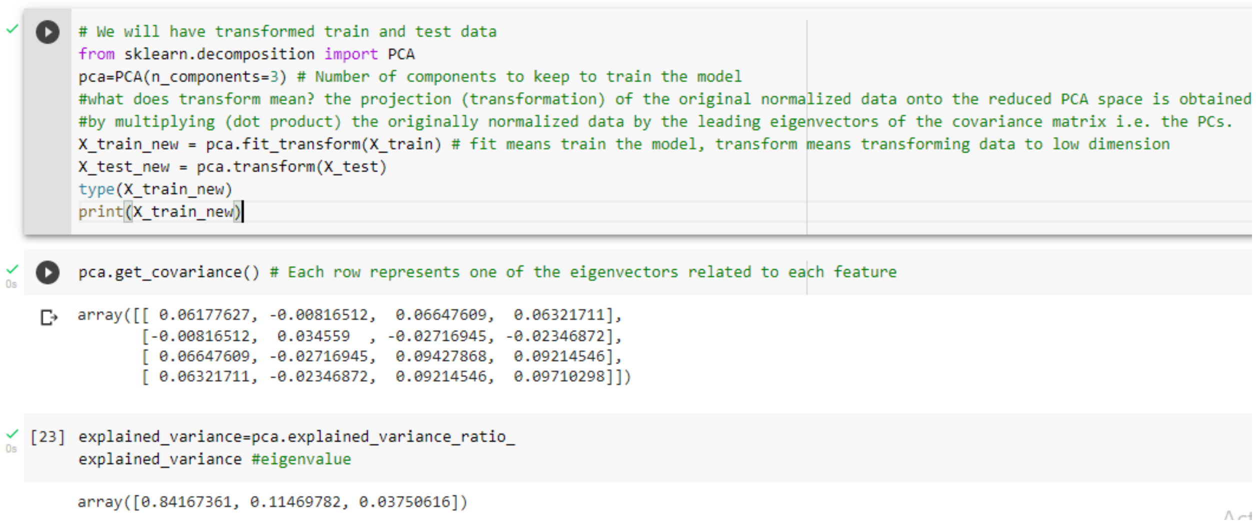
```
[19] y_test.shape
```

```
(45,)
```

## 2. With PCA

First, we will have transformed train and test data. Keep three components to train the model. Transform is
the projection (transformation) of the original normalized data onto the reduced PCA space is obtained

by multiplying (dot product) the originally normalized data by the leading eigenvectors of the covariance matrix i.e. the PCs. Fit means training the model and transform means transforming data to low dimensions. The covariance matrix is symmetric and each row of it represents the eigenvector related to each feature. The explained variance of the three first components is defined with their eigenvalues.



```
# We will have transformed train and test data
from sklearn.decomposition import PCA
pca=PCA(n_components=3) # Number of components to keep to train the model
#what does transform mean? the projection (transformation) of the original normalized data onto the reduced PCA space is obtained
#by multiplying (dot product) the originally normalized data by the leading eigenvectors of the covariance matrix i.e. the PCs.
X_train_new = pca.fit_transform(X_train) # fit means train the model, transform means transforming data to low dimension
X_test_new = pca.transform(X_test)
type(X_train_new)
print(X_train_new)
```

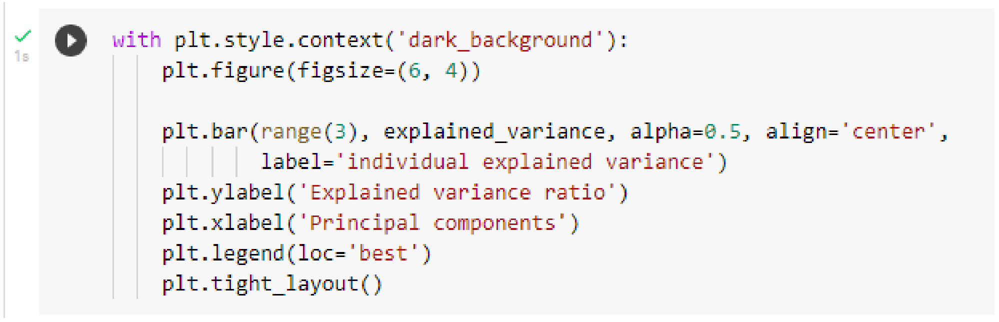
```
pca.get_covariance() # Each row represents one of the eigenvectors related to each feature
```

```
array([[ 0.06177627, -0.00816512,  0.06647609,  0.06321711],
       [-0.00816512,  0.034559  , -0.02716945, -0.02346872],
       [ 0.06647609, -0.02716945,  0.09427868,  0.09214546],
       [ 0.06321711, -0.02346872,  0.09214546,  0.09710298]])
```

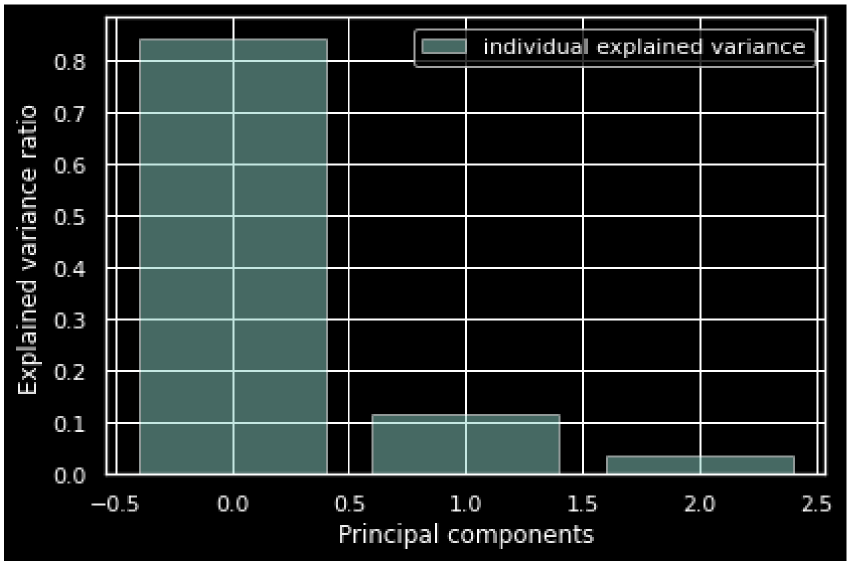
```
[23] explained_variance=pca.explained_variance_ratio_
     explained_variance #eigenvalue
```

```
array([0.84167361, 0.11469782, 0.03750616])
```

To see the variance explained with each feature more clear we plot the following figure:

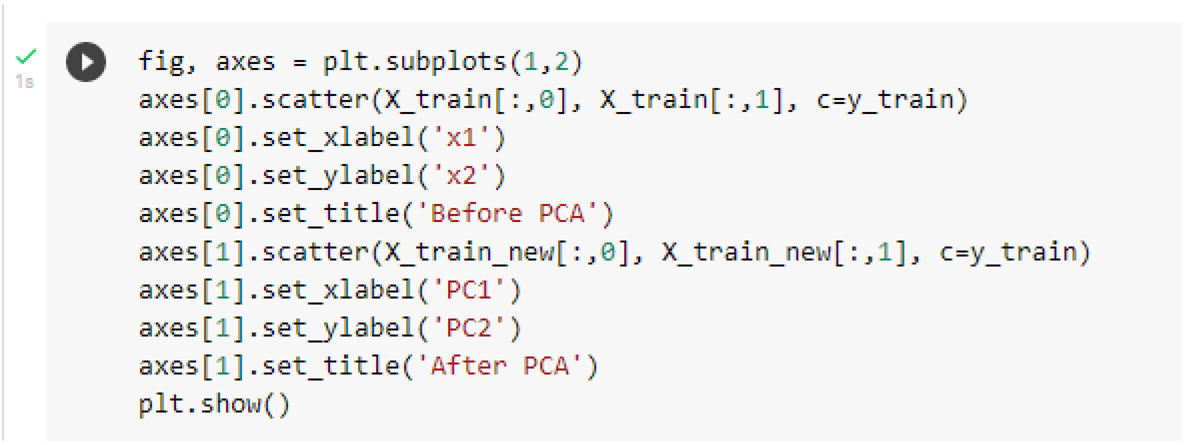


```
with plt.style.context('dark_background'):
    plt.figure(figsize=(6, 4))

    plt.bar(range(3), explained_variance, alpha=0.5, align='center',
            label='individual explained variance')
    plt.ylabel('Explained variance ratio')
    plt.xlabel('Principal components')
    plt.legend(loc='best')
    plt.tight_layout()
```

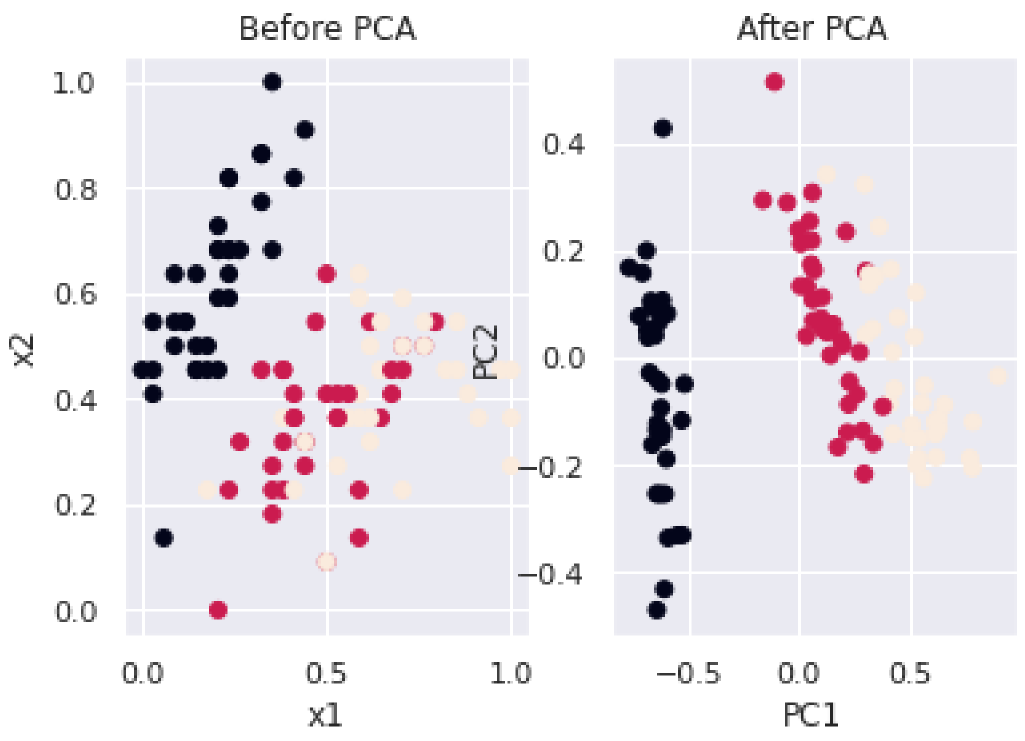


As we see in the above plot, the three first components explain up to 96% variance.

To compare distribution of classes before using PCA and after it, write the following code:

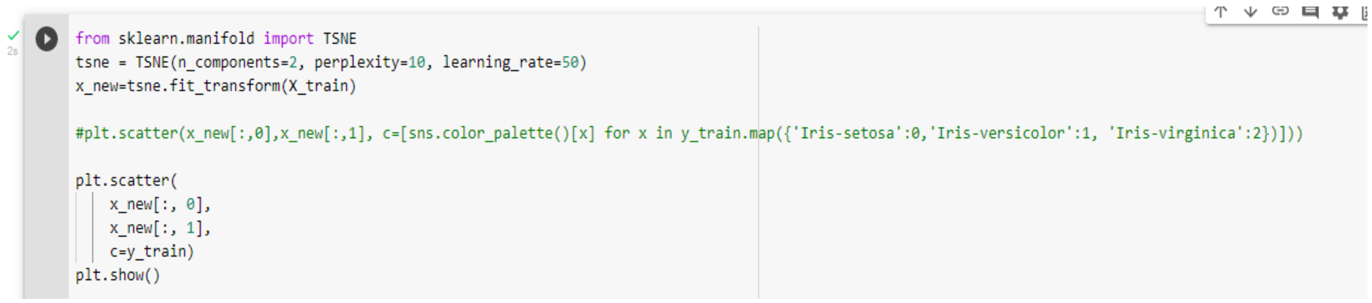


```
fig, axes = plt.subplots(1,2)
axes[0].scatter(X_train[:,0], X_train[:,1], c=y_train)
axes[0].set_xlabel('x1')
axes[0].set_ylabel('x2')
axes[0].set_title('Before PCA')
axes[1].scatter(X_train_new[:,0], X_train_new[:,1], c=y_train)
axes[1].set_xlabel('PC1')
axes[1].set_ylabel('PC2')
axes[1].set_title('After PCA')
plt.show()
```

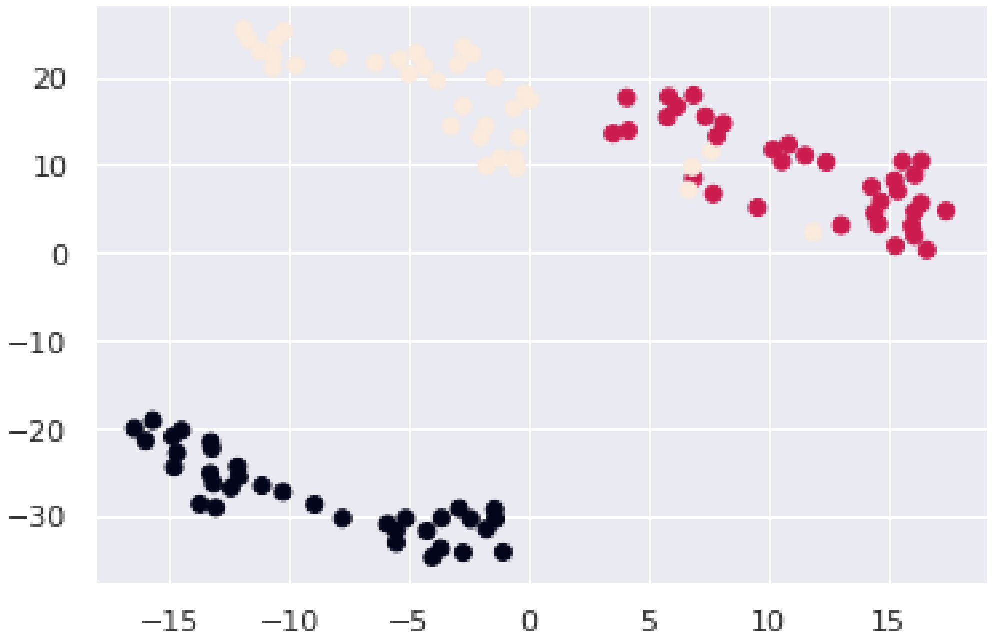


We see by using PCA, the feature extraction leads to classifying three classes more separately than before. PCA can also help us gain insight into the classification power of our data. Also, PCA is very useful to speed up the computation by reducing the dimensionality of the data. In the above figures, I decided to keep only two dimensions, so I used a scatter plot to show the variance of data according to each coordinate. Plus, when we have high dimensionality with the high correlated variables of one another, the PCA can improve the accuracy of the classification model.

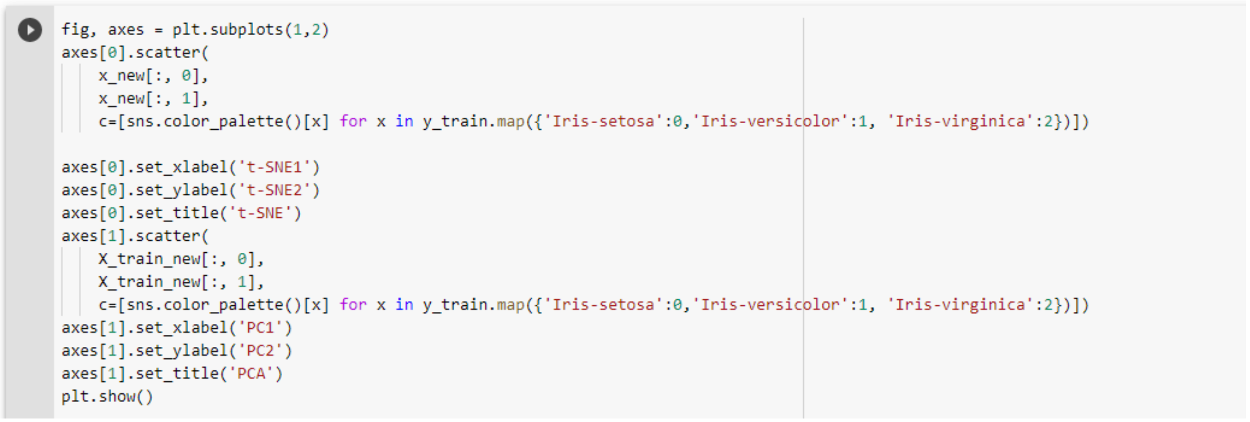
## 2. With t-SNE



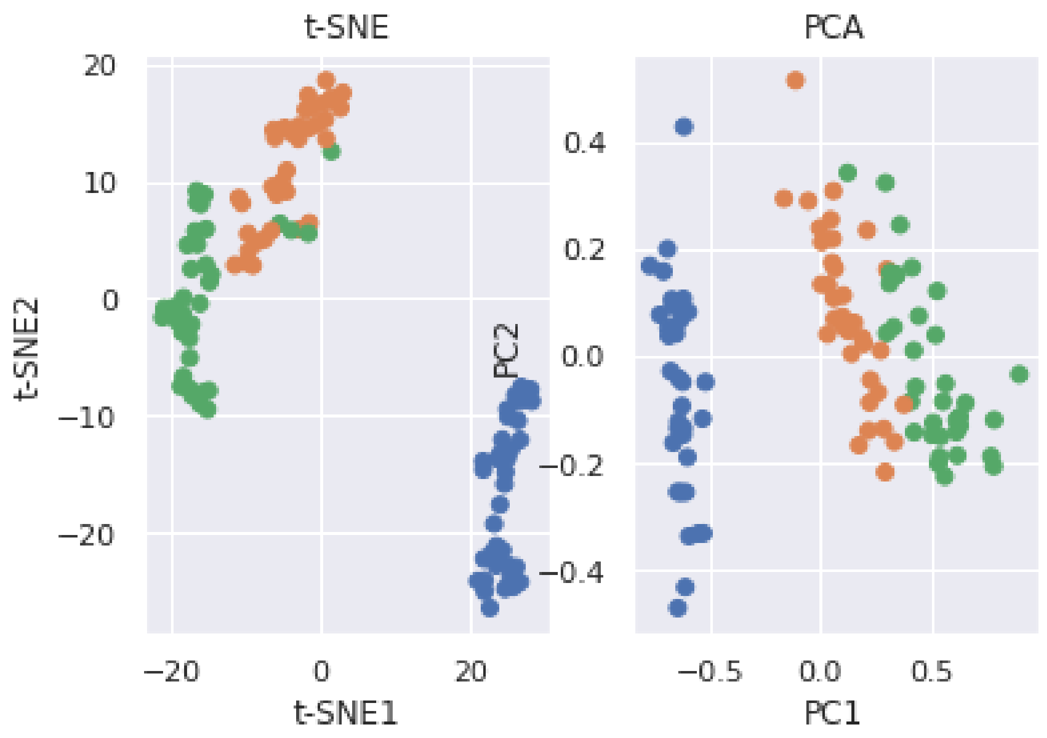
```
from sklearn.manifold import TSNE
tsne = TSNE(n_components=2, perplexity=10, learning_rate=50)
x_new=tsne.fit_transform(X_train)

#plt.scatter(x_new[:,0],x_new[:,1], c=[sns.color_palette()[x] for x in y_train.map({'Iris-setosa':0,'Iris-versicolor':1, 'Iris-virginica':2})])

plt.scatter(
    x_new[:, 0],
    x_new[:, 1],
    c=y_train)
plt.show()
```



To compare using PCS and t-SNE method:

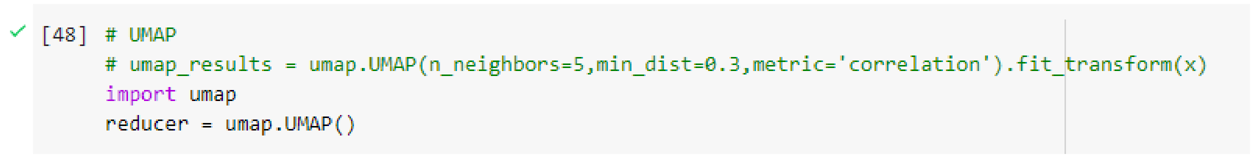


```
fig, axes = plt.subplots(1,2)
axes[0].scatter(
    x_new[:, 0],
    x_new[:, 1],
    c=[sns.color_palette()[x] for x in y_train.map({'Iris-setosa':0,'Iris-versicolor':1, 'Iris-virginica':2})])

axes[0].set_xlabel('t-SNE1')
axes[0].set_ylabel('t-SNE2')
axes[0].set_title('t-SNE')
axes[1].scatter(
    X_train_new[:, 0],
    X_train_new[:, 1],
    c=[sns.color_palette()[x] for x in y_train.map({'Iris-setosa':0,'Iris-versicolor':1, 'Iris-virginica':2})])
axes[1].set_xlabel('PC1')
axes[1].set_ylabel('PC2')
axes[1].set_title('PCA')
plt.show()
```



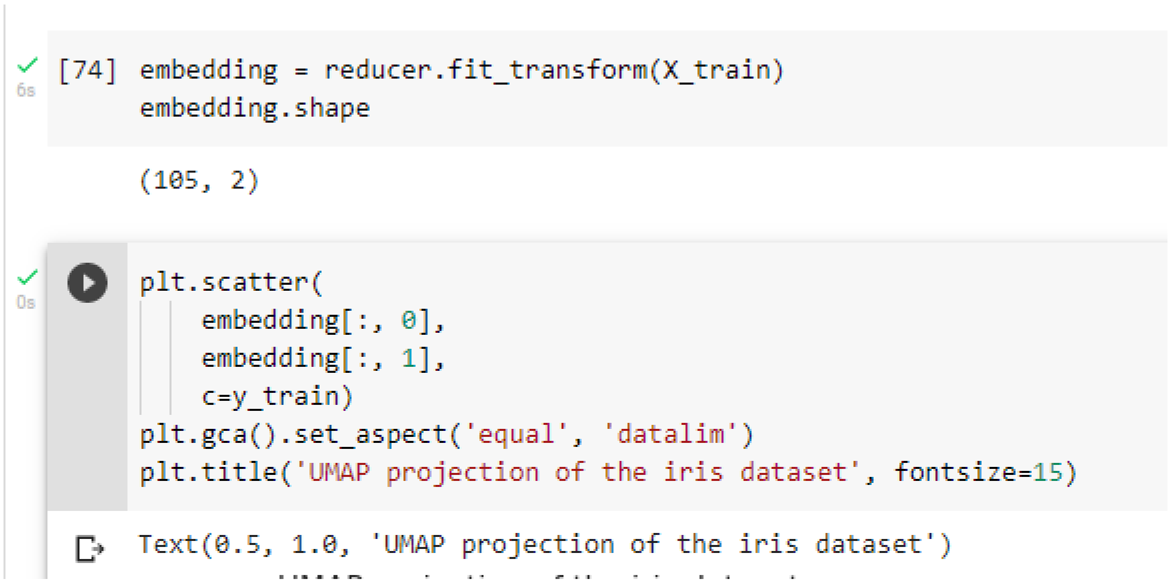
As we see in the above figures, t-SNE performs better than PCA. t-SNE is something called nonlinear dimensionality reduction. What that means is this algorithm allows us to separate data that cannot be separated by any straight line.

### 3. With UMAP



```
[48] # UMAP
     # umap_results = umap.UMAP(n_neighbors=5,min_dist=0.3,metric='correlation').fit_transform(x)
     import umap
     reducer = umap.UMAP()
```

Now we need to train our reducer, letting it learn about the manifold. For this UMAP follows the sklearn API and has a method fit which we pass the data we want the model to learn from. Since, at the end of the day, we are going to want to reduce the representation of the data we will use, instead, the fit_transform method which first calls fit and then returns the transformed data as a NumPy array.

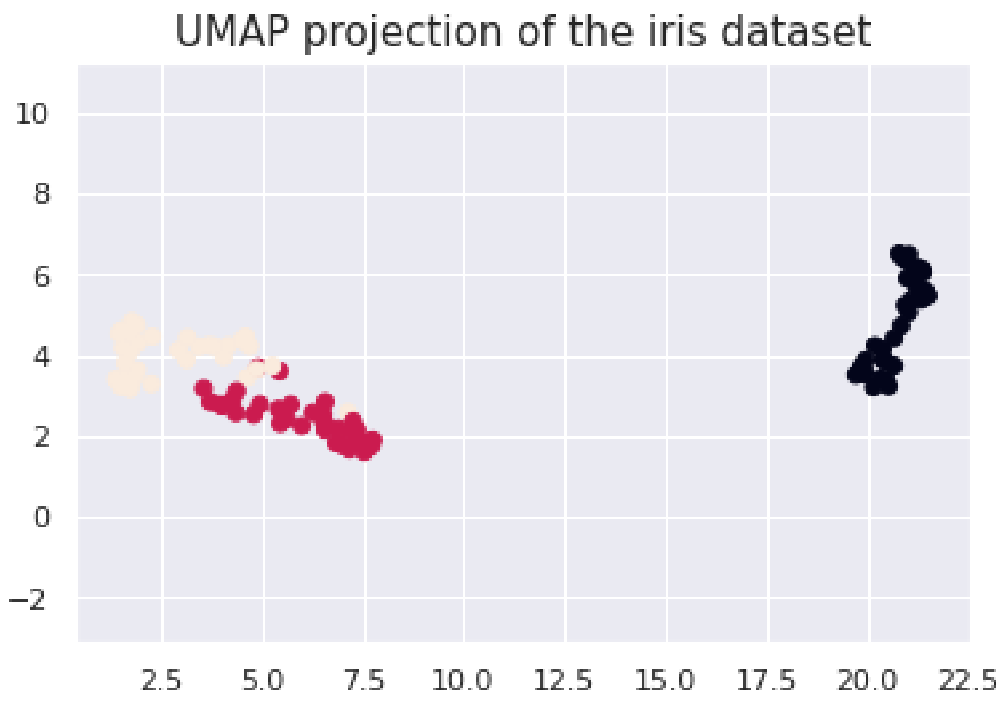


```
[74] embedding = reducer.fit_transform(X_train)
     embedding.shape
```

```
(105, 2)
```

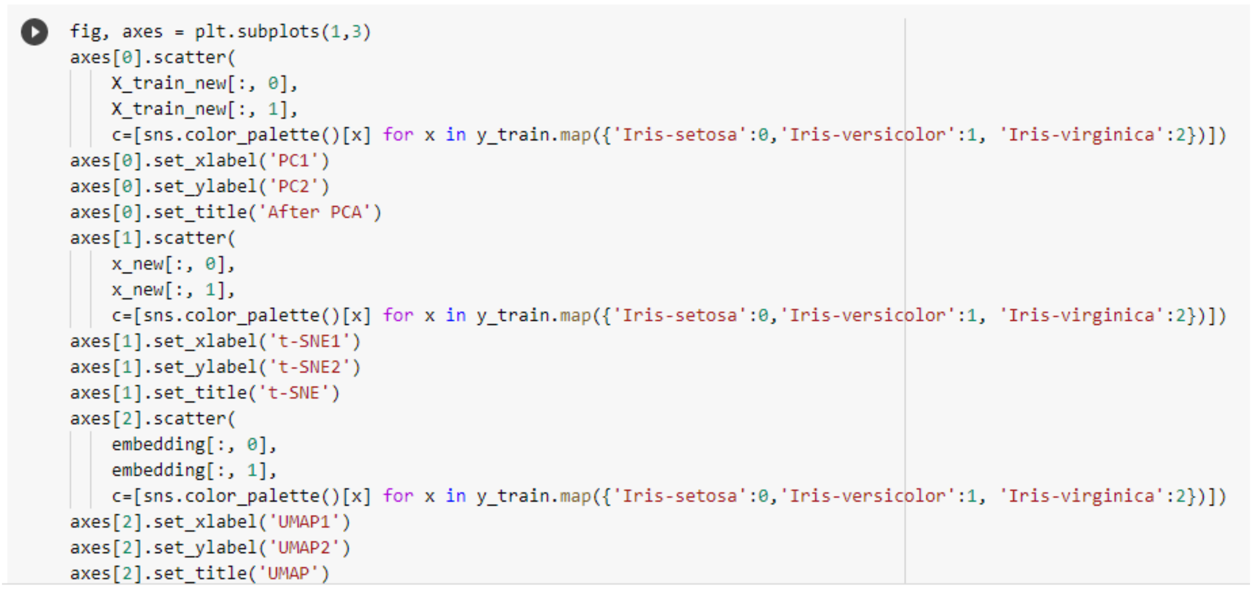
```
plt.scatter(
    embedding[:, 0],
    embedding[:, 1],
    c=y_train)
plt.gca().set_aspect('equal', 'datalim')
plt.title('UMAP projection of the iris dataset', fontsize=15)
```

```
Text(0.5, 1.0, 'UMAP projection of the iris dataset')
```

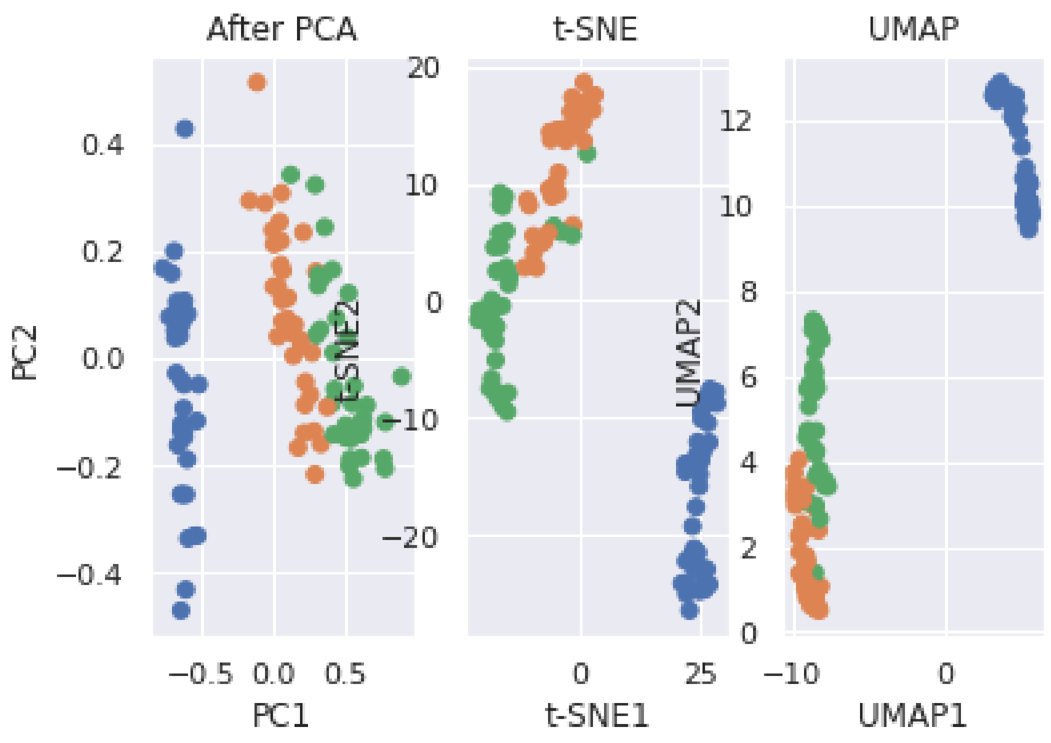


The result is an array with 105 samples of training, but only two feature columns (instead of the four we started with). This is because, by default, UMAP reduces down to 2D. Each row of the array is a 2-dimensional representation of the corresponding penguin. Thus, we can plot the embedding as a standard scatterplot and color by the target array (since it applies to the transformed data which is in the same order as the original).

To compare the three methods, PCA, t-SNE and UMAP:



```
fig, axes = plt.subplots(1,3)
axes[0].scatter(
    X_train_new[:, 0],
    X_train_new[:, 1],
    c=[sns.color_palette()[x] for x in y_train.map({'Iris-setosa':0,'Iris-versicolor':1, 'Iris-virginica':2})])
axes[0].set_xlabel('PC1')
axes[0].set_ylabel('PC2')
axes[0].set_title('After PCA')
axes[1].scatter(
    x_new[:, 0],
    x_new[:, 1],
    c=[sns.color_palette()[x] for x in y_train.map({'Iris-setosa':0,'Iris-versicolor':1, 'Iris-virginica':2})])
axes[1].set_xlabel('t-SNE1')
axes[1].set_ylabel('t-SNE2')
axes[1].set_title('t-SNE')
axes[2].scatter(
    embedding[:, 0],
    embedding[:, 1],
    c=[sns.color_palette()[x] for x in y_train.map({'Iris-setosa':0,'Iris-versicolor':1, 'Iris-virginica':2})])
axes[2].set_xlabel('UMAP1')
axes[2].set_ylabel('UMAP2')
axes[2].set_title('UMAP')
```

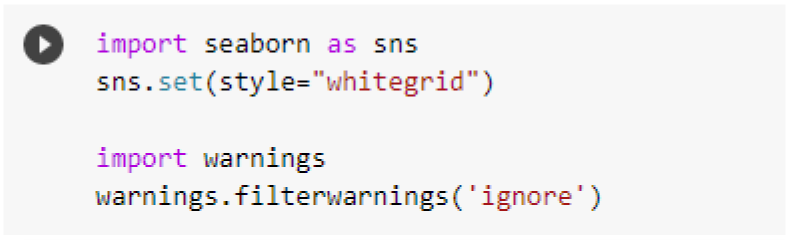


As we were explained, the performance of UMAP is better than PCA and t-SNE methods in visualizing separated labels.

## 4. Feature Selection

Note: We should be selecting features using TRAINING Dataset and NOT FULL Dataset.

In machine learning, feature selection is the process of choosing a subset of input features that contribute the most to the output feature for use in model construction.



```python
import seaborn as sns
sns.set(style="whitegrid")

import warnings
warnings.filterwarnings('ignore')
```

## 4.1. Filter Methods

In filter methods, features are selected independently from any machine algorithms. Filter methods generally use a specific criteria, such as scores in statistical test and variances, to rank the importance of individual features. I will discuss two types of filter selection methods:

ANOVA F-value

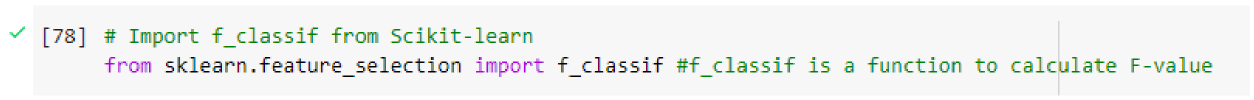
Mutual Information

### 4.1.1 ANOVA F-value

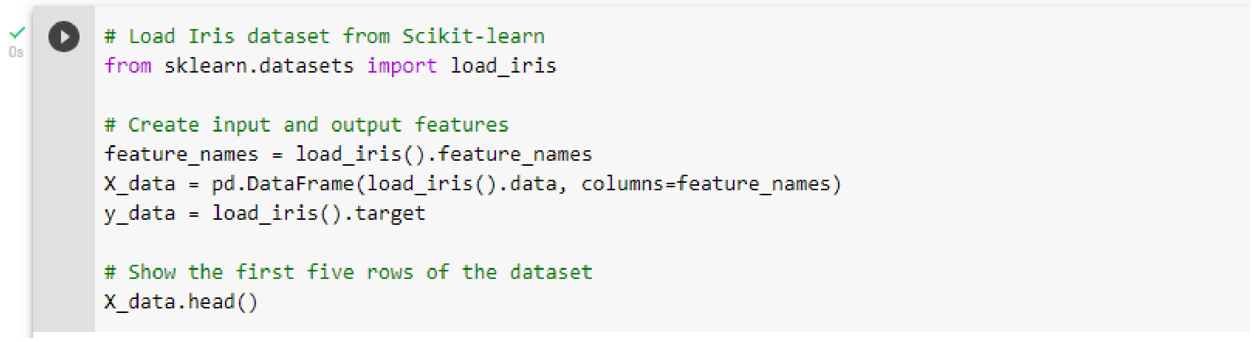
ANOVA F-value method estimates the degree of linearity between the input feature (i.e., predictor) and the output feature. A high F-value indicates high degree of linearity and a low F-value indicates low degree of linearity. We can use Scikit-learn to calculate ANOVA F-value. First, we need to load the library. Scikit-learn has two functions to calculate F-value:

f_classif, which calculate F-value between input and output feature for classification task

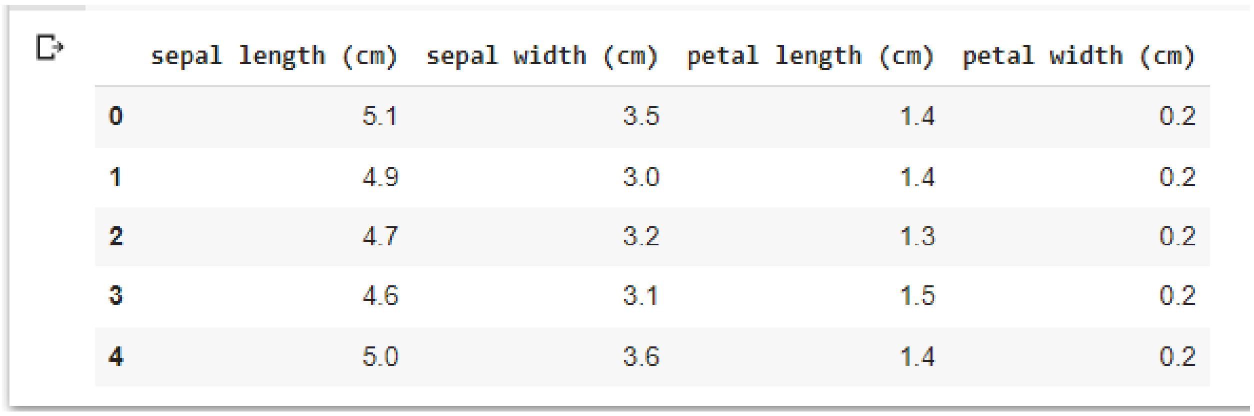
We will use f_classif because the Iris dataset entails classification task.



```python
[78] # Import f_classif from Scikit-learn
     from sklearn.feature_selection import f_classif #f_classif is a function to calculate F-value
```

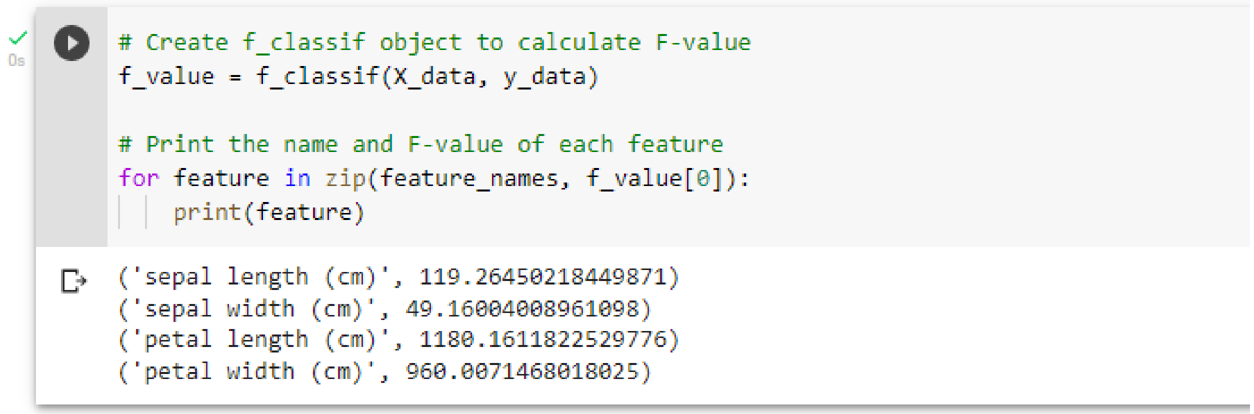


```python
# Load Iris dataset from Scikit-learn
from sklearn.datasets import load_iris

# Create input and output features
feature_names = load_iris().feature_names
X_data = pd.DataFrame(load_iris().data, columns=feature_names)
y_data = load_iris().target

# Show the first five rows of the dataset
X_data.head()
```



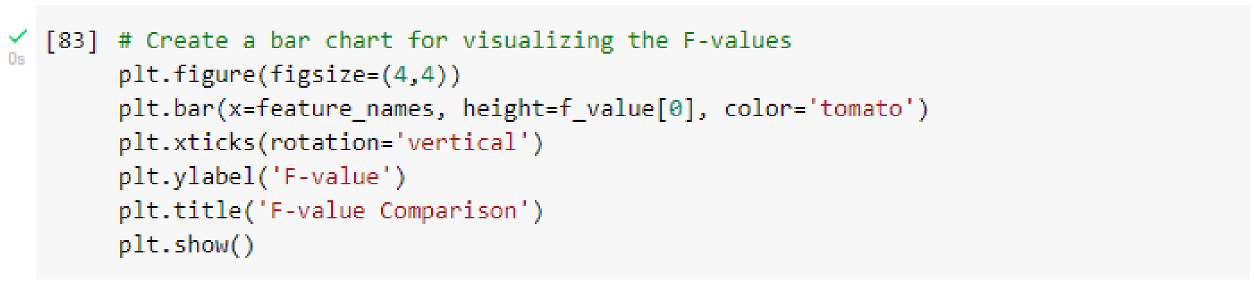
| | sepal length (cm) | sepal width (cm) | petal length (cm) | petal width (cm) |
|---|---|---|---|---|
| 0 | 5.1 | 3.5 | 1.4 | 0.2 |
| 1 | 4.9 | 3.0 | 1.4 | 0.2 |
| 2 | 4.7 | 3.2 | 1.3 | 0.2 |
| 3 | 4.6 | 3.1 | 1.5 | 0.2 |
| 4 | 5.0 | 3.6 | 1.4 | 0.2 |

Then, we calculate F-value for each input feature in the Iris dataset by calling the following and creating a bar chart for visualizing the F-values

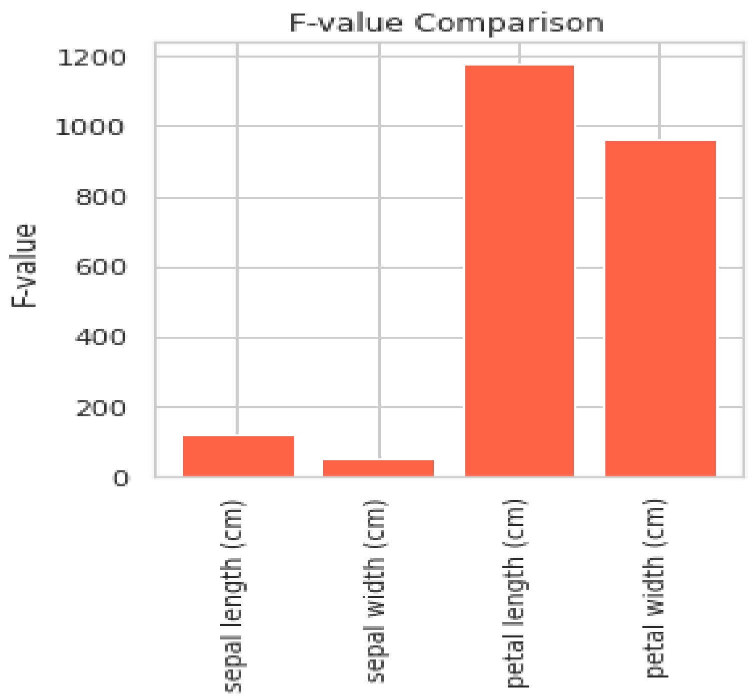


```python
# Create f_classif object to calculate F-value
f_value = f_classif(X_data, y_data)

# Print the name and F-value of each feature
for feature in zip(feature_names, f_value[0]):
    print(feature)
```

```
('sepal length (cm)', 119.26450218449871)
('sepal width (cm)', 49.16004008961098)
('petal length (cm)', 1180.1611822529776)
('petal width (cm)', 960.0071468018025)
```



```python
[83] # Create a bar chart for visualizing the F-values
     plt.figure(figsize=(4,4))
     plt.bar(x=feature_names, height=f_value[0], color='tomato')
     plt.xticks(rotation='vertical')
     plt.ylabel('F-value')
     plt.title('F-value Comparison')
     plt.show()
```



**4.1.2. Mutual Information**

Mutual information (MI) measures the dependence of one variable to another by quantifying the amount of information obtained about one feature, through the other feature. MI is symmetric and non-negative, and is zero if and only if the input and output feature are independent. Unlike ANOVA F-value, mutual information can capture non-linear relationships between input and output feature. We can use Scikit-learn to calculate MI. Scikit-learn has two functions to calculate MI:

mutual_info_classif, which calculate MI for classification task

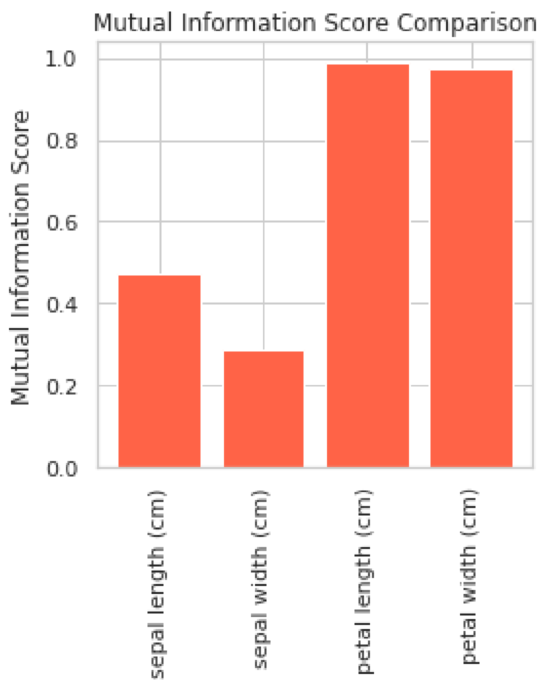
We will use mutual_info_classif because the Iris dataset entails a classification task.

```
[84] # Import mutual_info_classif from Scikit-learn
     from sklearn.feature_selection import mutual_info_classif
```

```
# Create mutual_info_classif object to calculate mutual information
MI_score = mutual_info_classif(X_data, y_data, random_state=0)

# Print the name and mutual information score of each feature
for feature in zip(feature_names, MI_score):
    print(feature)
```

```
('sepal length (cm)', 0.4738732342805525)
('sepal width (cm)', 0.2860722769917176)
('petal length (cm)', 0.9895851732491787)
('petal width (cm)', 0.9749379656705233)
```

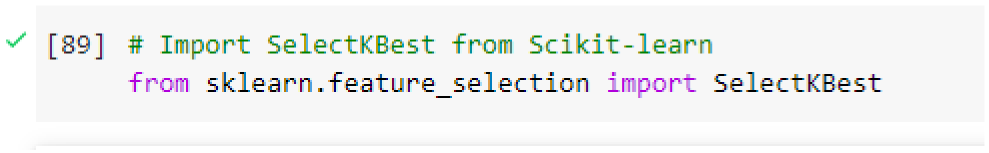
```
[88] # Create a bar chart for visualizing the mutual information scores
     plt.figure(figsize=(4,4))
     plt.bar(x=feature_names, height=MI_score, color='tomato')
     plt.xticks(rotation='vertical')
     plt.ylabel('Mutual Information Score')
     plt.title('Mutual Information Score Comparison')
```



### 4.3. Using Selector Object for Selecting Features

We can use SelectKBest from Scikit-learn to select features according to the k highest scores, determined by a filter method.

First, we need to import SelectKBest.



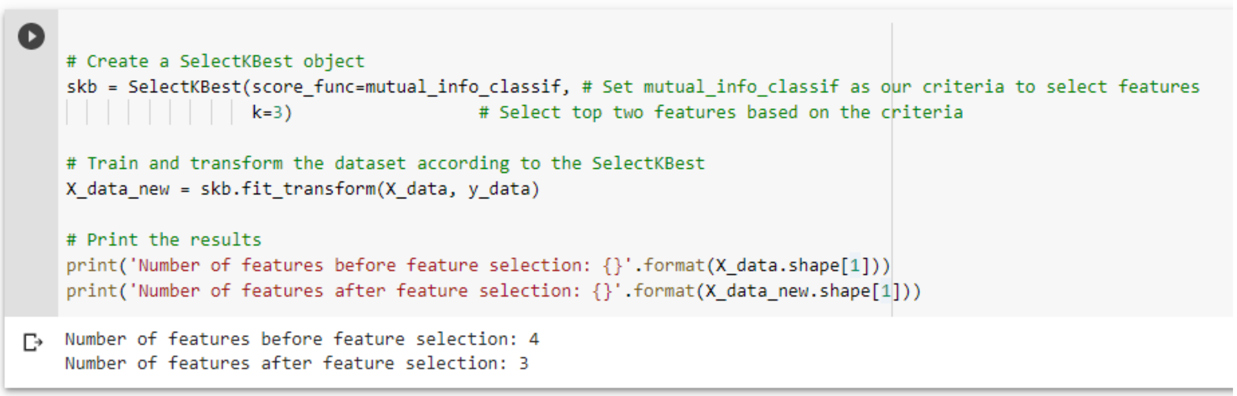
```
[89] # Import SelectKBest from Scikit-learn
     from sklearn.feature_selection import SelectKBest
```

SelectKBest has two important parameters:

score_func: the filter function that is used for feature selection

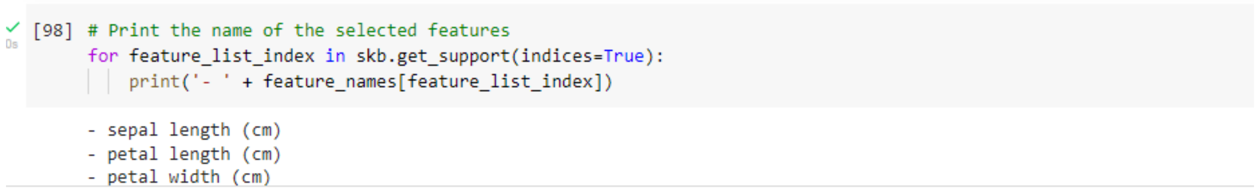
k: the number of top features to select

Let's demonstrate SelectKBest by using ANOVA F-value as our filter method. We will select the top two features based on the ANOVA F-value.



```
# Create a SelectKBest object
skb = SelectKBest(score_func=mutual_info_classif, # Set mutual_info_classif as our criteria to select features
                  k=3)                  # Select top two features based on the criteria

# Train and transform the dataset according to the SelectKBest
X_data_new = skb.fit_transform(X_data, y_data)

# Print the results
print('Number of features before feature selection: {}'.format(X_data.shape[1]))
print('Number of features after feature selection: {}'.format(X_data_new.shape[1]))
```

```
Number of features before feature selection: 4
Number of features after feature selection: 3
```



```
[98] # Print the name of the selected features
     for feature_list_index in skb.get_support(indices=True):
         print('- ' + feature_names[feature_list_index])
```

```
- sepal length (cm)
- petal length (cm)
- petal width (cm)
```

As we see, three features that have the most influence are sepal length, petal length and petal width.

## 5. Wrapper Methods

### 5.1. Exhaustive Feature Selection (EFS)

EFS finds the best subset of features by evaluating all feature combinations. Suppose we have a dataset with three features. EFS will evaluate the following feature combinations:

feature_1

feature_2

feature_3

feature_1 and feature_2

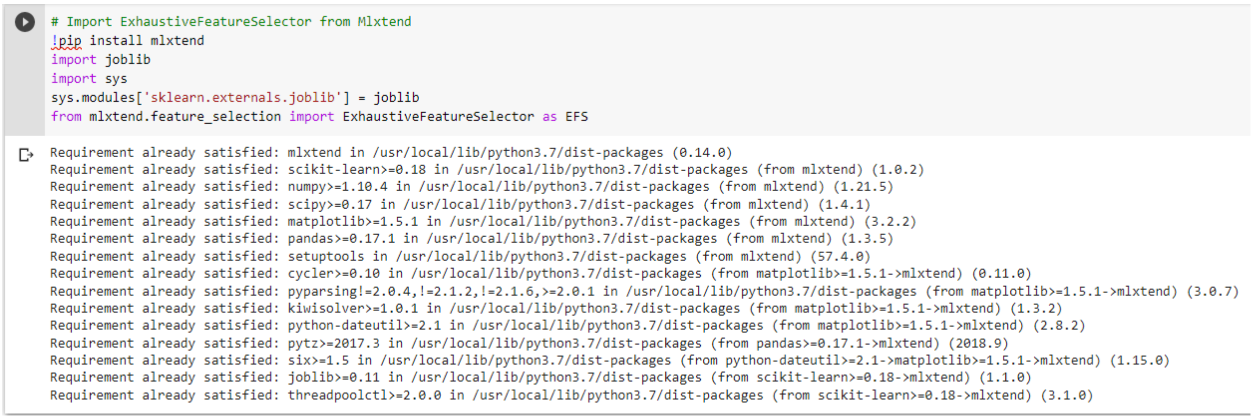
feature_1 and feature_3

feature_2 and feature_3

feature_1, feature_2, and feature_3

EFS selects a subset that generates the best performance (e.g., accuracy, precision, recall, etc.) of the model being considered.

Mlxtend provides ExhaustiveFeatureSelector function to perform EFS.



```
# Import ExhaustiveFeatureSelector from Mlxtend
!pip install mlxtend
import joblib
import sys
sys.modules['sklearn.externals.joblib'] = joblib
from mlxtend.feature_selection import ExhaustiveFeatureSelector as EFS
```

```
Requirement already satisfied: mlxtend in /usr/local/lib/python3.7/dist-packages (0.14.0)
Requirement already satisfied: scikit-learn>=0.18 in /usr/local/lib/python3.7/dist-packages (from mlxtend) (1.0.2)
Requirement already satisfied: numpy>=1.10.4 in /usr/local/lib/python3.7/dist-packages (from mlxtend) (1.21.5)
Requirement already satisfied: scipy>=0.17 in /usr/local/lib/python3.7/dist-packages (from mlxtend) (1.4.1)
Requirement already satisfied: matplotlib>=1.5.1 in /usr/local/lib/python3.7/dist-packages (from mlxtend) (3.2.2)
Requirement already satisfied: pandas>=0.17.1 in /usr/local/lib/python3.7/dist-packages (from mlxtend) (1.3.5)
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-packages (from mlxtend) (57.4.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-packages (from matplotlib>=1.5.1->mlxtend) (0.11.0)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib>=1.5.1->mlxtend) (3.0.7)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib>=1.5.1->mlxtend) (1.3.2)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib>=1.5.1->mlxtend) (2.8.2)
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.7/dist-packages (from pandas>=0.17.1->mlxtend) (2018.9)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from python-dateutil>=2.1->matplotlib>=1.5.1->mlxtend) (1.15.0)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.7/dist-packages (from scikit-learn>=0.18->mlxtend) (1.1.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.7/dist-packages (from scikit-learn>=0.18->mlxtend) (3.1.0)
```

EFS has five important parameters:

estimator: the classifier that we intend to train

min_features: the minimum number of features to select

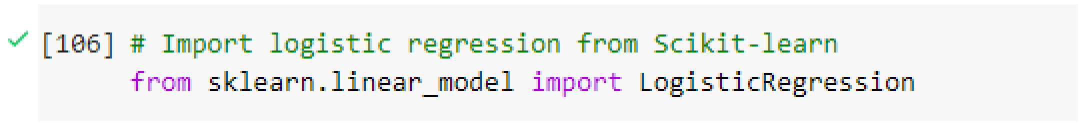
max_features: the maximum number of features to select

scoring: the metric to use to evaluate the classifier

cv: the number of cross-validations to perform

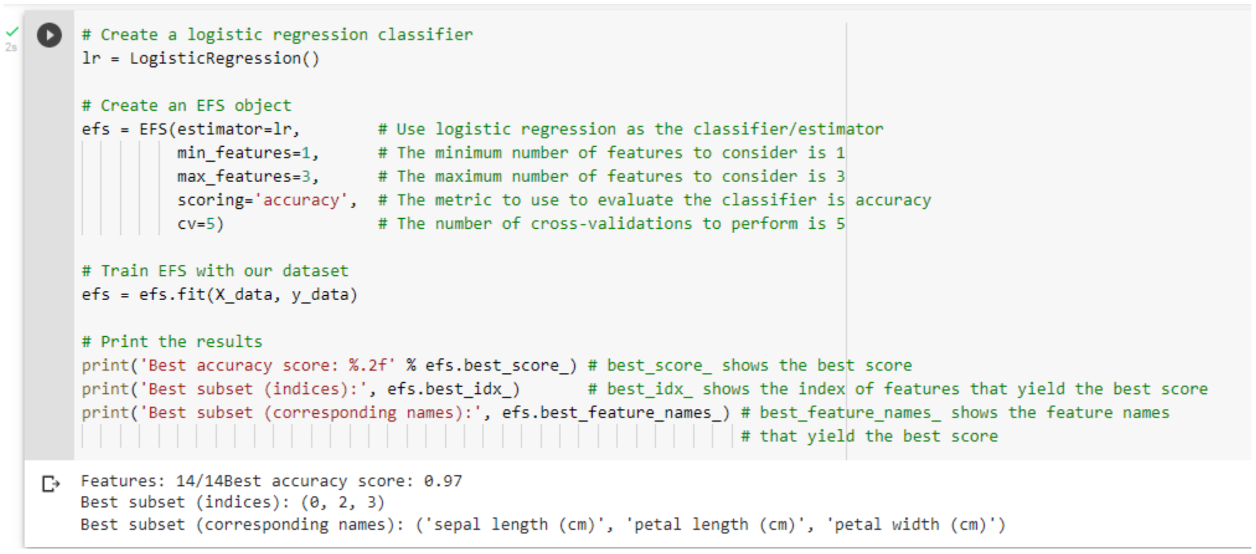
In this example, we use logistic regression as our classifier/estimator.

Import logistic regression as a machine learning model to train and evaluate the performance of wrapper methods:



```
[106] # Import logistic regression from Scikit-learn
      from sklearn.linear_model import LogisticRegression
```

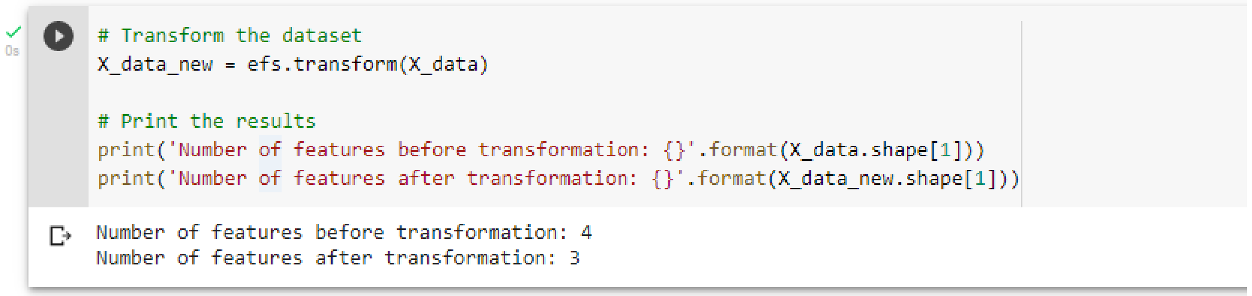
Then, we perform EFS by calling the following:



```
# Create a logistic regression classifier
lr = LogisticRegression()

# Create an EFS object
efs = EFS(estimator=lr,        # Use logistic regression as the classifier/estimator
          min_features=1,      # The minimum number of features to consider is 1
          max_features=3,      # The maximum number of features to consider is 3
          scoring='accuracy',  # The metric to use to evaluate the classifier is accuracy
          cv=5)                # The number of cross-validations to perform is 5

# Train EFS with our dataset
efs = efs.fit(X_data, y_data)

# Print the results
print('Best accuracy score: %.2f' % efs.best_score_) # best_score_ shows the best score
print('Best subset (indices):', efs.best_idx_)       # best_idx_ shows the index of features that yield the best score
print('Best subset (corresponding names):', efs.best_feature_names_) # best_feature_names_ shows the feature names
                                                                     # that yield the best score
```

```
Features: 14/14Best accuracy score: 0.97
Best subset (indices): (0, 2, 3)
Best subset (corresponding names): ('sepal length (cm)', 'petal length (cm)', 'petal width (cm)')
```

Then transform the dataset and print the results:

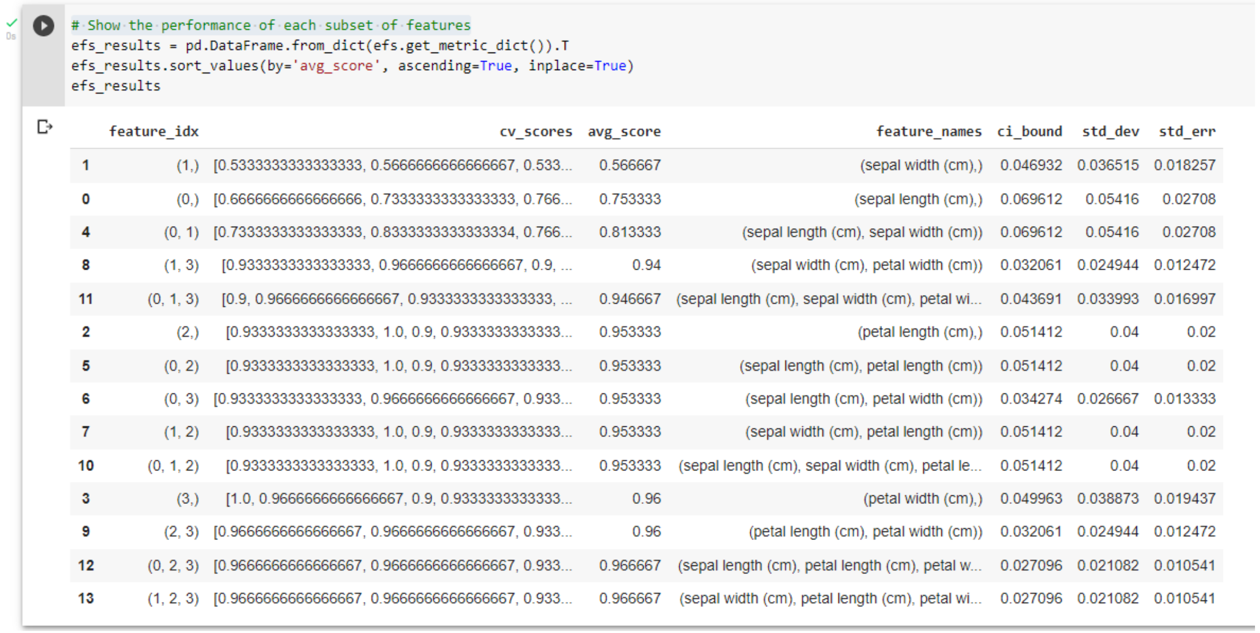


```
# Transform the dataset
X_data_new = efs.transform(X_data)

# Print the results
print('Number of features before transformation: {}'.format(X_data.shape[1]))
print('Number of features after transformation: {}'.format(X_data_new.shape[1]))
```

```
Number of features before transformation: 4
Number of features after transformation: 3
```

The number of features before the transformation was 4 and after transformation is 3.

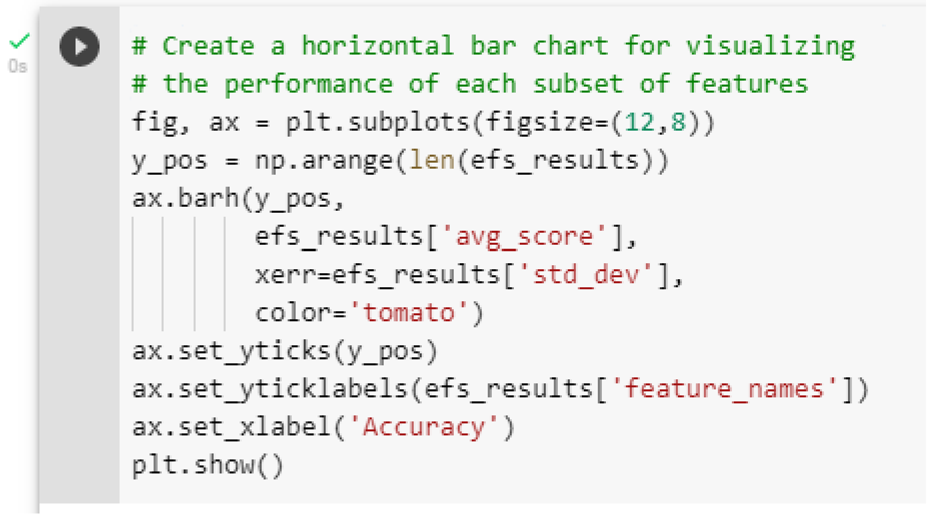
Show the performance of each subset of features:



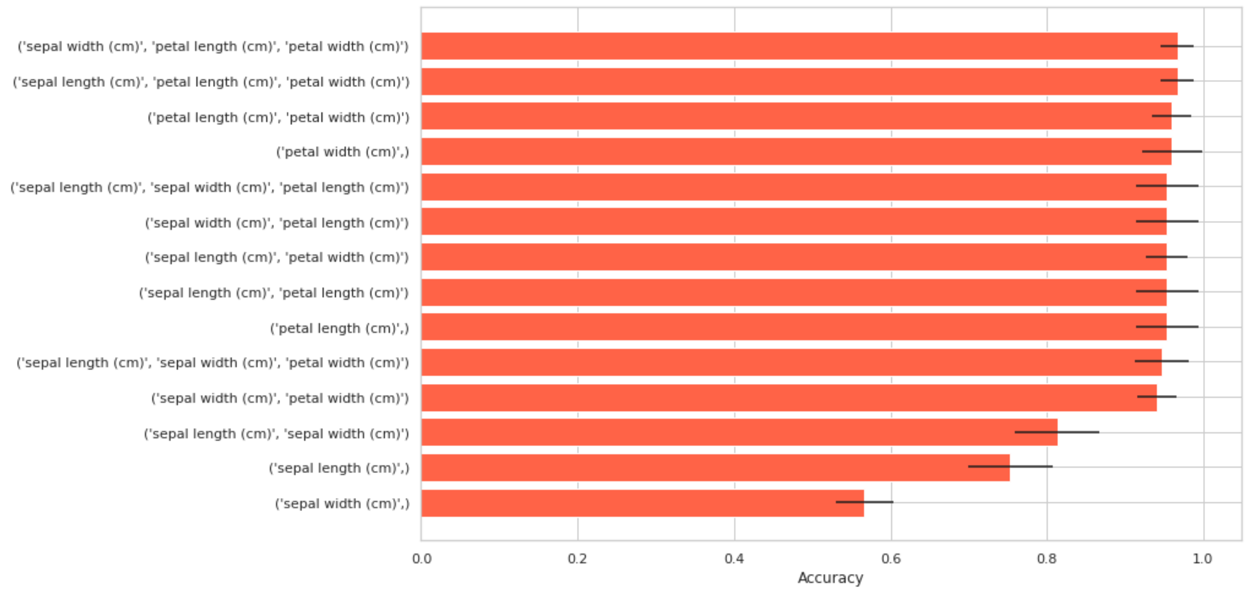
```
# Show the performance of each subset of features
efs_results = pd.DataFrame.from_dict(efs.get_metric_dict()).T
efs_results.sort_values(by='avg_score', ascending=True, inplace=True)
efs_results
```

| | feature_idx | cv_scores | avg_score | feature_names | ci_bound | std_dev | std_err |
|---|---|---|---|---|---|---|---|
| **1** | (1,) | [0.5333333333333333, 0.5666666666666667, 0.533... | 0.566667 | (sepal width (cm),) | 0.046932 | 0.036515 | 0.018257 |
| **0** | (0,) | [0.6666666666666666, 0.7333333333333333, 0.766... | 0.753333 | (sepal length (cm),) | 0.069612 | 0.05416 | 0.02708 |
| **4** | (0, 1) | [0.7333333333333333, 0.8333333333333334, 0.766... | 0.813333 | (sepal length (cm), sepal width (cm)) | 0.069612 | 0.05416 | 0.02708 |
| **8** | (1, 3) | [0.9333333333333333, 0.9666666666666667, 0.9, ... | 0.94 | (sepal width (cm), petal width (cm)) | 0.032061 | 0.024944 | 0.012472 |
| **11** | (0, 1, 3) | [0.9, 0.9666666666666667, 0.9333333333333333, ... | 0.946667 | (sepal length (cm), sepal width (cm), petal wi... | 0.043691 | 0.033993 | 0.016997 |
| **2** | (2,) | [0.9333333333333333, 1.0, 0.9, 0.9333333333333... | 0.953333 | (petal length (cm),) | 0.051412 | 0.04 | 0.02 |
| **5** | (0, 2) | [0.9333333333333333, 1.0, 0.9, 0.9333333333333... | 0.953333 | (sepal length (cm), petal length (cm)) | 0.051412 | 0.04 | 0.02 |
| **6** | (0, 3) | [0.9333333333333333, 0.9666666666666667, 0.933... | 0.953333 | (sepal length (cm), petal width (cm)) | 0.034274 | 0.026667 | 0.013333 |
| **7** | (1, 2) | [0.9333333333333333, 1.0, 0.9, 0.9333333333333... | 0.953333 | (sepal width (cm), petal length (cm)) | 0.051412 | 0.04 | 0.02 |
| **10** | (0, 1, 2) | [0.9333333333333333, 1.0, 0.9, 0.9333333333333... | 0.953333 | (sepal length (cm), sepal width (cm), petal le... | 0.051412 | 0.04 | 0.02 |
| **3** | (3,) | [1.0, 0.9666666666666667, 0.9, 0.9333333333333... | 0.96 | (petal width (cm),) | 0.049963 | 0.038873 | 0.019437 |
| **9** | (2, 3) | [0.9666666666666667, 0.9666666666666667, 0.933... | 0.96 | (petal length (cm), petal width (cm)) | 0.032061 | 0.024944 | 0.012472 |
| **12** | (0, 2, 3) | [0.9666666666666667, 0.9666666666666667, 0.933... | 0.966667 | (sepal length (cm), petal length (cm), petal w... | 0.027096 | 0.021082 | 0.010541 |
| **13** | (1, 2, 3) | [0.9666666666666667, 0.9666666666666667, 0.933... | 0.966667 | (sepal width (cm), petal length (cm), petal wi... | 0.027096 | 0.021082 | 0.010541 |

Create a horizontal bar chart for visualizing the performance of each subset of features:



```
# Create a horizontal bar chart for visualizing
# the performance of each subset of features
fig, ax = plt.subplots(figsize=(12,8))
y_pos = np.arange(len(efs_results))
ax.barh(y_pos,
        efs_results['avg_score'],
        xerr=efs_results['std_dev'],
        color='tomato')
ax.set_yticks(y_pos)
ax.set_yticklabels(efs_results['feature_names'])
ax.set_xlabel('Accuracy')
plt.show()
```



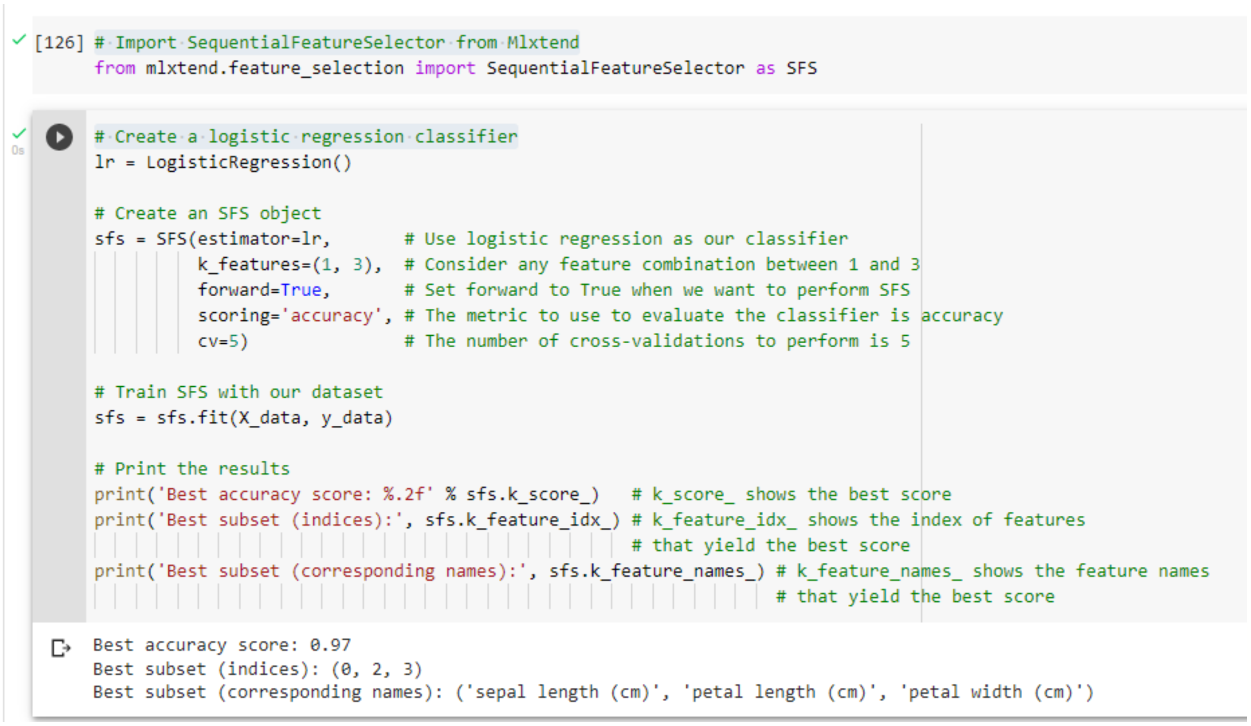
As we see, the combination of three features with indices 1,2,3 has the most average score as a wrapper performance criterion.

### 5.2. Sequential Forward Selection (SFS)

SFS finds the best subset of feature by adding a feature that best improves the model at each iteration.

Import SequentialFeatureSelector from Mlxtend and create a logistic regression classifier.

Then create an SFS object and train it with the dataset and print the results:

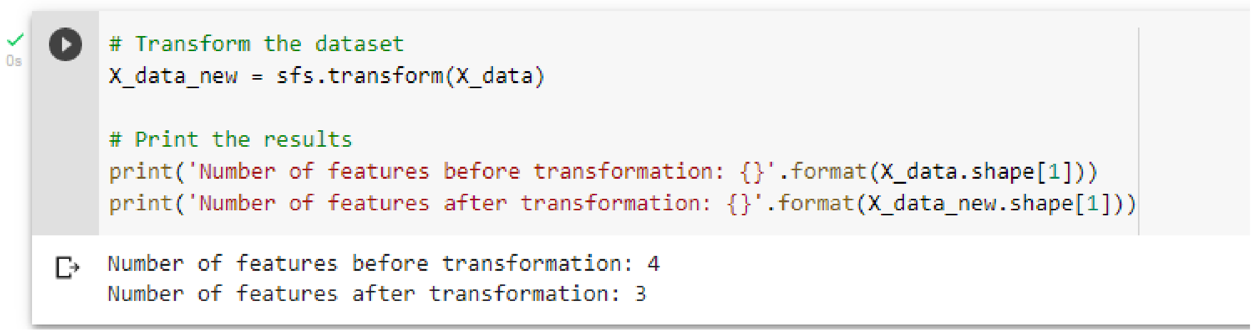


```
[126] # Import SequentialFeatureSelector from Mlxtend
      from mlxtend.feature_selection import SequentialFeatureSelector as SFS
```

```
# Create a logistic regression classifier
lr = LogisticRegression()

# Create an SFS object
sfs = SFS(estimator=lr,       # Use logistic regression as our classifier
          k_features=(1, 3),  # Consider any feature combination between 1 and 3
          forward=True,       # Set forward to True when we want to perform SFS
          scoring='accuracy', # The metric to use to evaluate the classifier is accuracy
          cv=5)               # The number of cross-validations to perform is 5

# Train SFS with our dataset
sfs = sfs.fit(X_data, y_data)

# Print the results
print('Best accuracy score: %.2f' % sfs.k_score_)   # k_score_ shows the best score
print('Best subset (indices):', sfs.k_feature_idx_) # k_feature_idx_ shows the index of features
                                                    # that yield the best score
print('Best subset (corresponding names):', sfs.k_feature_names_) # k_feature_names_ shows the feature names
                                                                  # that yield the best score
```

```
Best accuracy score: 0.97
Best subset (indices): (0, 2, 3)
Best subset (corresponding names): ('sepal length (cm)', 'petal length (cm)', 'petal width (cm)')
```

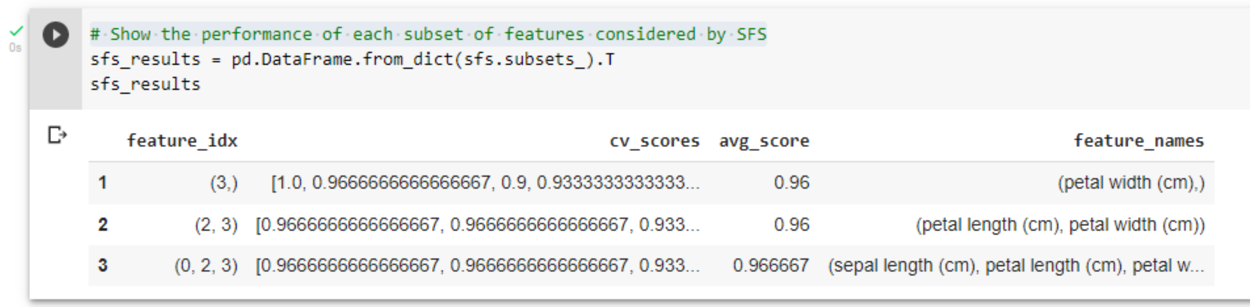
Then, transform the dataset and print the results:



```
# Transform the dataset
X_data_new = sfs.transform(X_data)

# Print the results
print('Number of features before transformation: {}'.format(X_data.shape[1]))
print('Number of features after transformation: {}'.format(X_data_new.shape[1]))
```

```
Number of features before transformation: 4
Number of features after transformation: 3
```

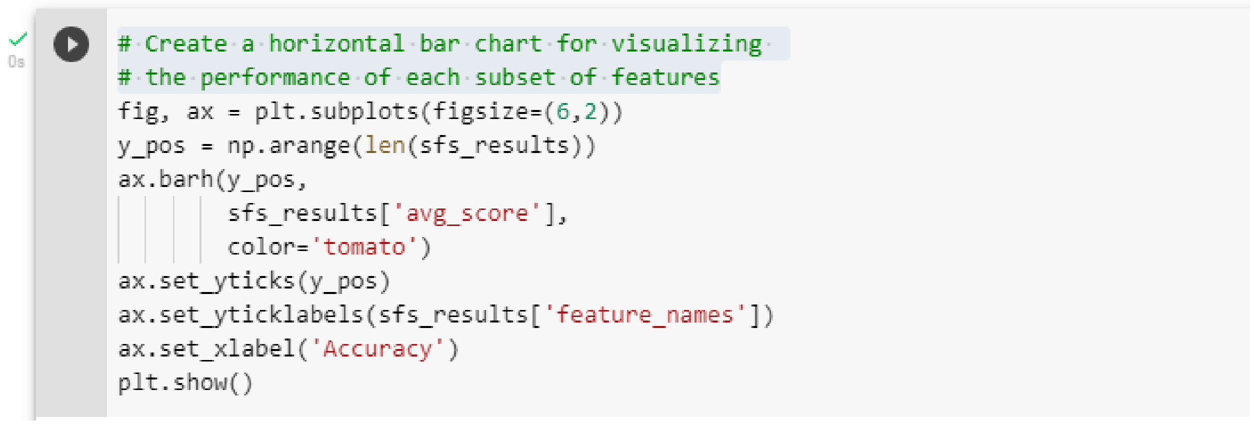
Show the performance of each subset of features considered by SFS:



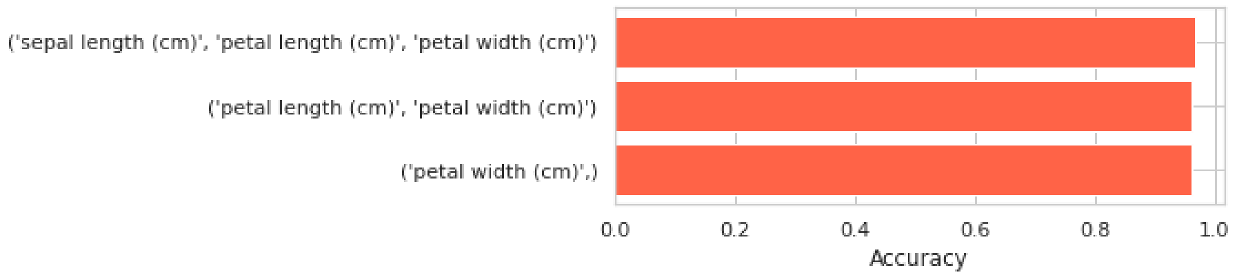
```
# Show the performance of each subset of features considered by SFS
sfs_results = pd.DataFrame.from_dict(sfs.subsets_).T
sfs_results
```

| | feature_idx | cv_scores | avg_score | feature_names |
|---|---|---|---|---|
| 1 | (3,) | [1.0, 0.9666666666666667, 0.9, 0.9333333333333... | 0.96 | (petal width (cm),) |
| 2 | (2, 3) | [0.9666666666666667, 0.9666666666666667, 0.933... | 0.96 | (petal length (cm), petal width (cm)) |
| 3 | (0, 2, 3) | [0.9666666666666667, 0.9666666666666667, 0.933... | 0.966667 | (sepal length (cm), petal length (cm), petal w... |

Create a horizontal bar chart for visualizing the performance of each subset of features:

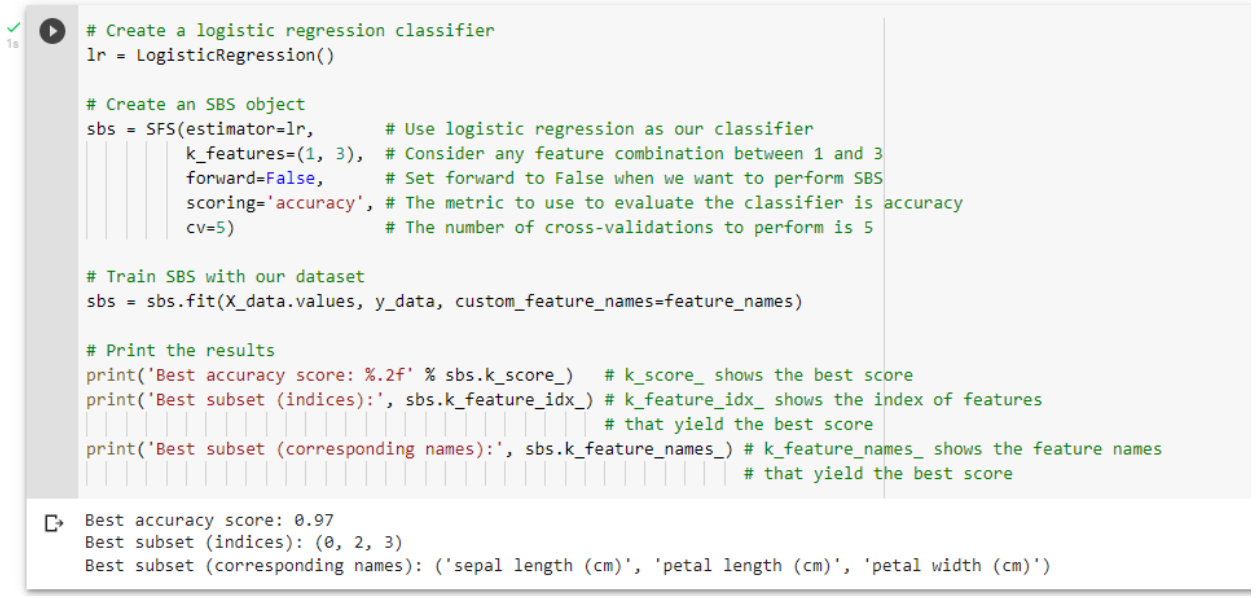


```python
# Create a horizontal bar chart for visualizing
# the performance of each subset of features
fig, ax = plt.subplots(figsize=(6,2))
y_pos = np.arange(len(sfs_results))
ax.barh(y_pos,
        sfs_results['avg_score'],
        color='tomato')
ax.set_yticks(y_pos)
ax.set_yticklabels(sfs_results['feature_names'])
ax.set_xlabel('Accuracy')
plt.show()
```



## 5.3. Sequential Backward Selection (SBS)

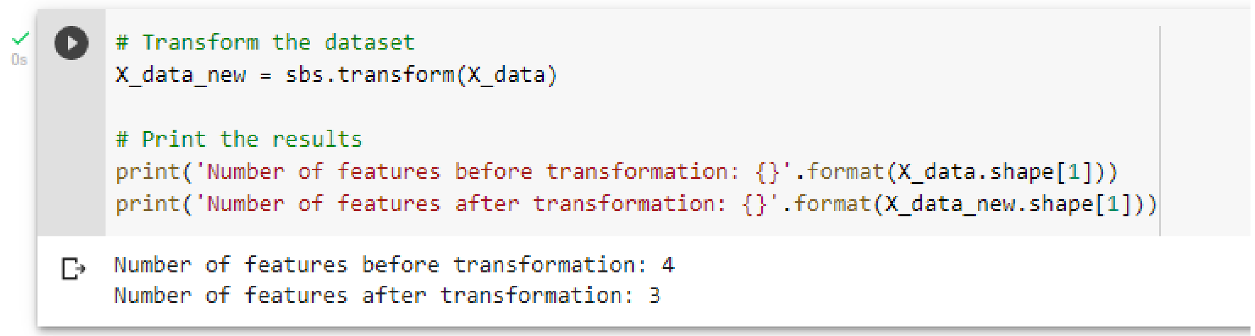
SBS is the opposite of SFS. SBS starts with all features and removes the feature that has the least importance to the model at each iteration.



```python
# Create a logistic regression classifier
lr = LogisticRegression()

# Create an SBS object
sbs = SFS(estimator=lr,       # Use logistic regression as our classifier
          k_features=(1, 3),  # Consider any feature combination between 1 and 3
          forward=False,      # Set forward to False when we want to perform SBS
          scoring='accuracy', # The metric to use to evaluate the classifier is accuracy
          cv=5)               # The number of cross-validations to perform is 5

# Train SBS with our dataset
sbs = sbs.fit(X_data.values, y_data, custom_feature_names=feature_names)

# Print the results
print('Best accuracy score: %.2f' % sbs.k_score_)   # k_score_ shows the best score
print('Best subset (indices):', sbs.k_feature_idx_) # k_feature_idx_ shows the index of features
                                                    # that yield the best score
print('Best subset (corresponding names):', sbs.k_feature_names_) # k_feature_names_ shows the feature names
                                                                  # that yield the best score
```

```
Best accuracy score: 0.97
Best subset (indices): (0, 2, 3)
Best subset (corresponding names): ('sepal length (cm)', 'petal length (cm)', 'petal width (cm)')
```

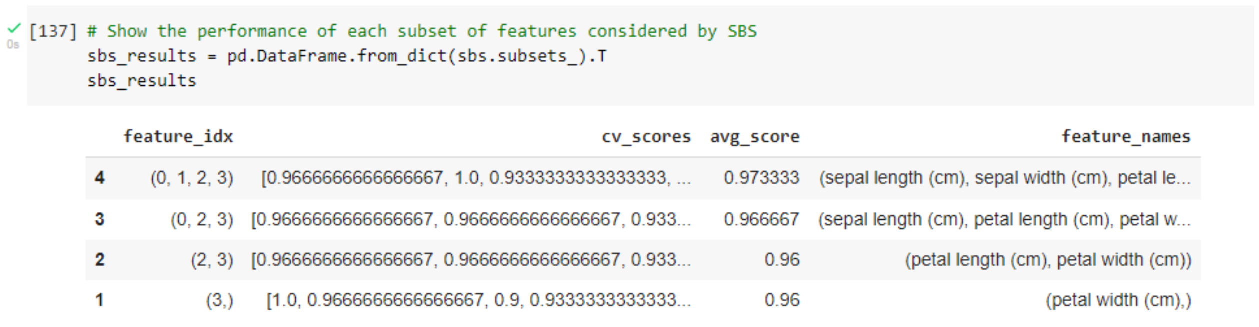
We can transform the dataset into a new dataset containing only the subset of features that generates the best score by using transform method.



```python
# Transform the dataset
X_data_new = sbs.transform(X_data)

# Print the results
print('Number of features before transformation: {}'.format(X_data.shape[1]))
print('Number of features after transformation: {}'.format(X_data_new.shape[1]))
```

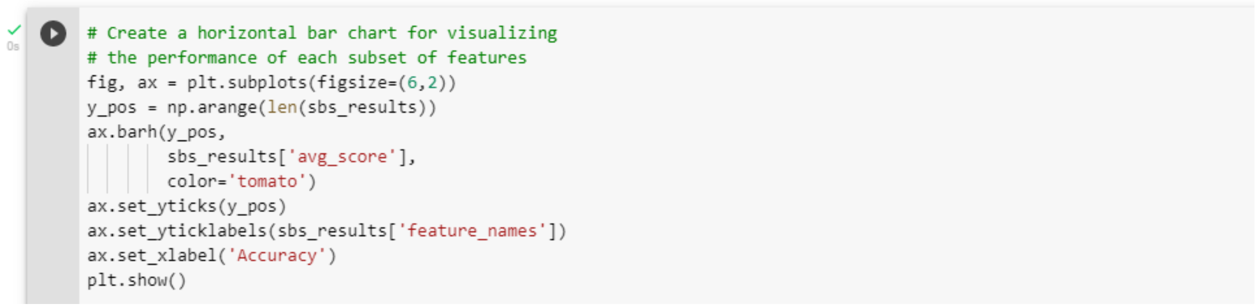
```
Number of features before transformation: 4
Number of features after transformation: 3
```

We can see the performance of each subset of features considered by SFS by calling subsets_.

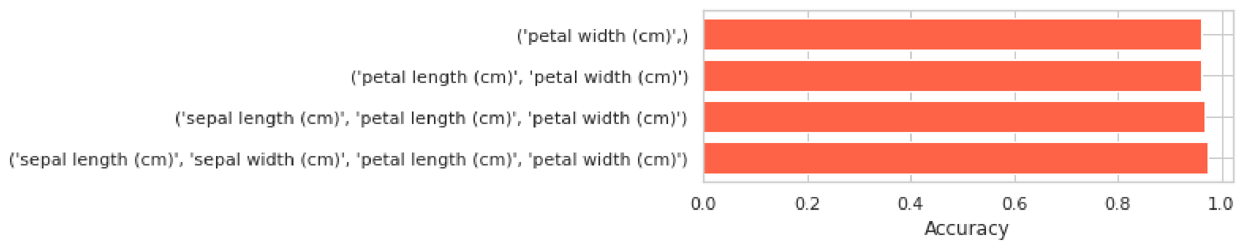


```python
[137] # Show the performance of each subset of features considered by SBS
      sbs_results = pd.DataFrame.from_dict(sbs.subsets_).T
      sbs_results
```

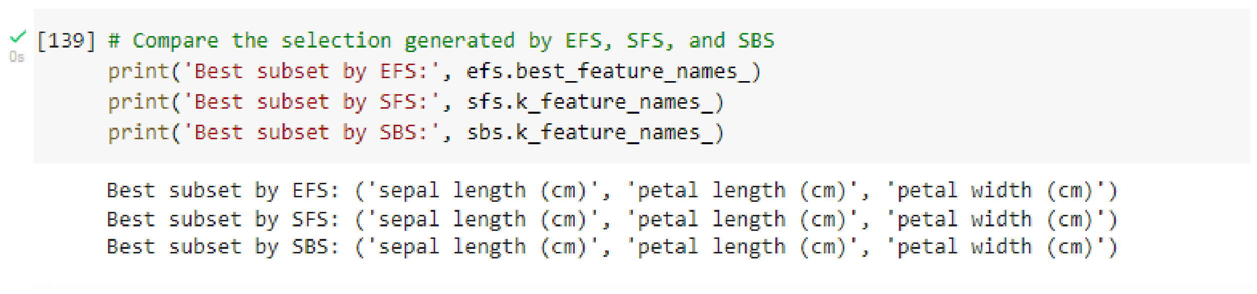
| | feature_idx | cv_scores | avg_score | feature_names |
|---|---|---|---|---|
| **4** | (0, 1, 2, 3) | [0.9666666666666667, 1.0, 0.9333333333333333, ... | 0.973333 | (sepal length (cm), sepal width (cm), petal le... |
| **3** | (0, 2, 3) | [0.9666666666666667, 0.9666666666666667, 0.933... | 0.966667 | (sepal length (cm), petal length (cm), petal w... |
| **2** | (2, 3) | [0.9666666666666667, 0.9666666666666667, 0.933... | 0.96 | (petal length (cm), petal width (cm)) |
| **1** | (3,) | [1.0, 0.9666666666666667, 0.9, 0.9333333333333... | 0.96 | (petal width (cm),) |



```python
# Create a horizontal bar chart for visualizing
# the performance of each subset of features
fig, ax = plt.subplots(figsize=(6,2))
y_pos = np.arange(len(sbs_results))
ax.barh(y_pos,
        sbs_results['avg_score'],
        color='tomato')
ax.set_yticks(y_pos)
ax.set_yticklabels(sbs_results['feature_names'])
ax.set_xlabel('Accuracy')
plt.show()
```



**Let's compare the selection generated by EFS, SFS, and SBS.**



```
[139] # Compare the selection generated by EFS, SFS, and SBS
      print('Best subset by EFS:', efs.best_feature_names_)
      print('Best subset by SFS:', sfs.k_feature_names_)
      print('Best subset by SBS:', sbs.k_feature_names_)
```

```
Best subset by EFS: ('sepal length (cm)', 'petal length (cm)', 'petal width (cm)')
Best subset by SFS: ('sepal length (cm)', 'petal length (cm)', 'petal width (cm)')
Best subset by SBS: ('sepal length (cm)', 'petal length (cm)', 'petal width (cm)')
```

In this scenario, selecting the best combination of features out of the 3 available features in the Iris s et, we end up with similar results regardless of which selection algorithms we used. In other cases w ith larger dataset and higher number of features, the selection is highly likely to be different for each selection algorithm.