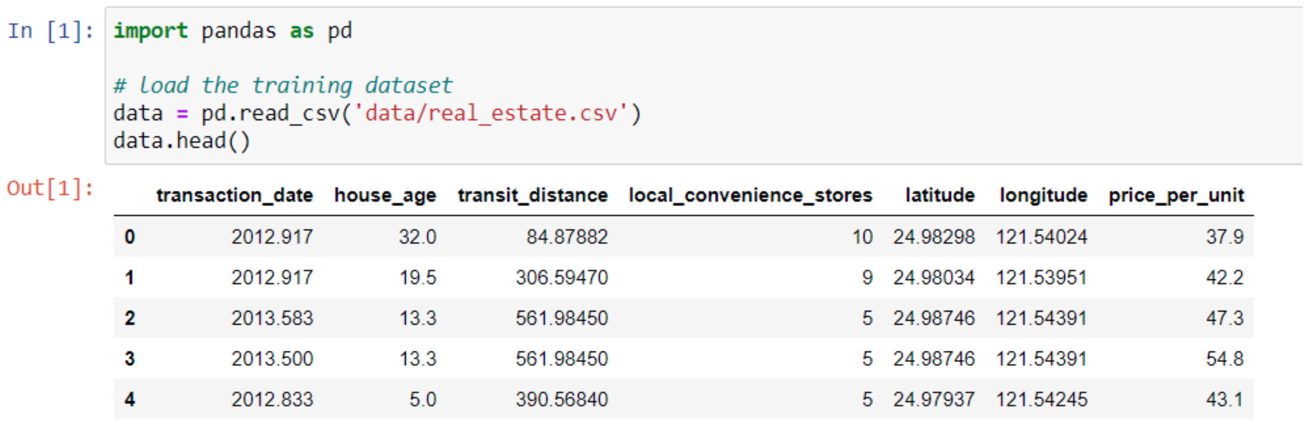
Regression! Classification! & Clustering!

Mahedi Hasan Jisan

Regression is a statistical method that can be used in such scenarios where one feature is dependent on the other features. Regression also identifies the importance of the features, the influences of each other, what can be useful, and what can be ignored. Regression usually works well with numerical datasets. Let's get to the point, I will use the **dataset of real estate sales transactions to predict the price-per-unit of a property based on its features. The price-per-unit in this data is based on a unit measurement of 3.3 square meters.** The dataset can be found **here**!

Let's start with loading the dataset and see the features and labels!



```
In [1]: import pandas as pd

        # load the training dataset
        data = pd.read_csv('data/real_estate.csv')
        data.head()
```

Out[1]:

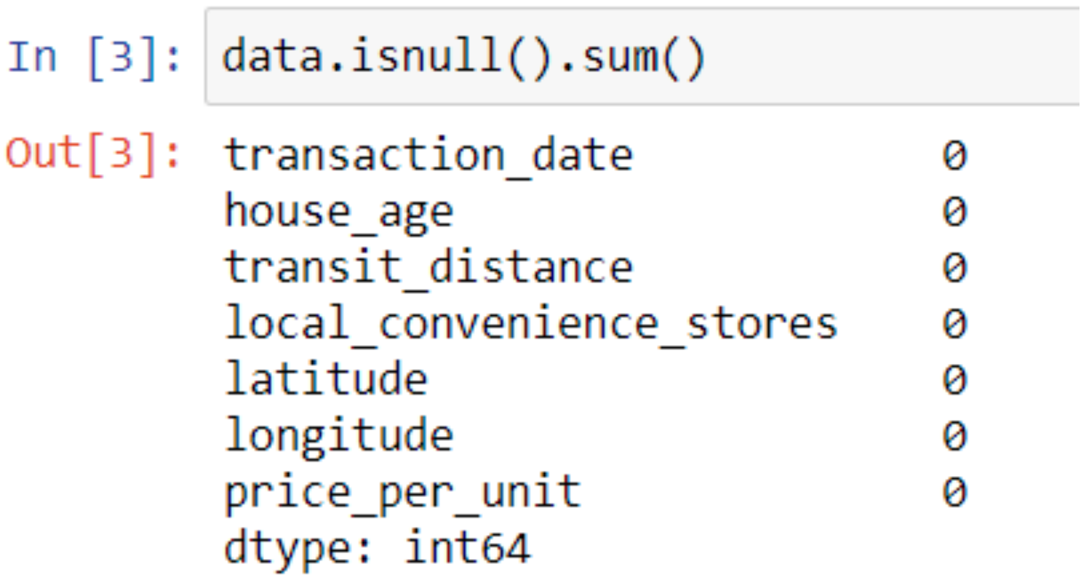
	transaction_date	house_age	transit_distance	local_convenience_stores	latitude	longitude	price_per_unit
0	2012.917	32.0	84.87882	10	24.98298	121.54024	37.9
1	2012.917	19.5	306.59470	9	24.98034	121.53951	42.2
2	2013.583	13.3	561.98450	5	24.98746	121.54391	47.3
3	2013.500	13.3	561.98450	5	24.98746	121.54391	54.8
4	2012.833	5.0	390.56840	5	24.97937	121.54245	43.1

Pandas to Load Dataset!

> *The data consists of the following variables:*

> ***transaction_date*** *— the transaction date (for example, 2013.250=2013 March, 2013.500=2013 June, etc.)*
>
> ***house_age*** *— the house age (in years)*
>
> ***transit_distance*** *— the distance to the nearest light rail station (in meters)*
>
> ***local_convenience_stores*** *— the number of convenience stores within walking distance*
>
> ***latitude*** *— the geographic coordinate, latitude*
>
> ***longitude*** *— the geographic coordinate, longitude*
>
> ***price_per_unit*** *house price of unit area (3.3 square meters)*

The very important job is to look for missing values and outliers in the dataset. Okay, let's do that!



```
In [3]: data.isnull().sum()

Out[3]: transaction_date            0
        house_age                   0
        transit_distance            0
        local_convenience_stores    0
        latitude                    0
        longitude                   0
        price_per_unit              0
        dtype: int64
```

No Missing values!

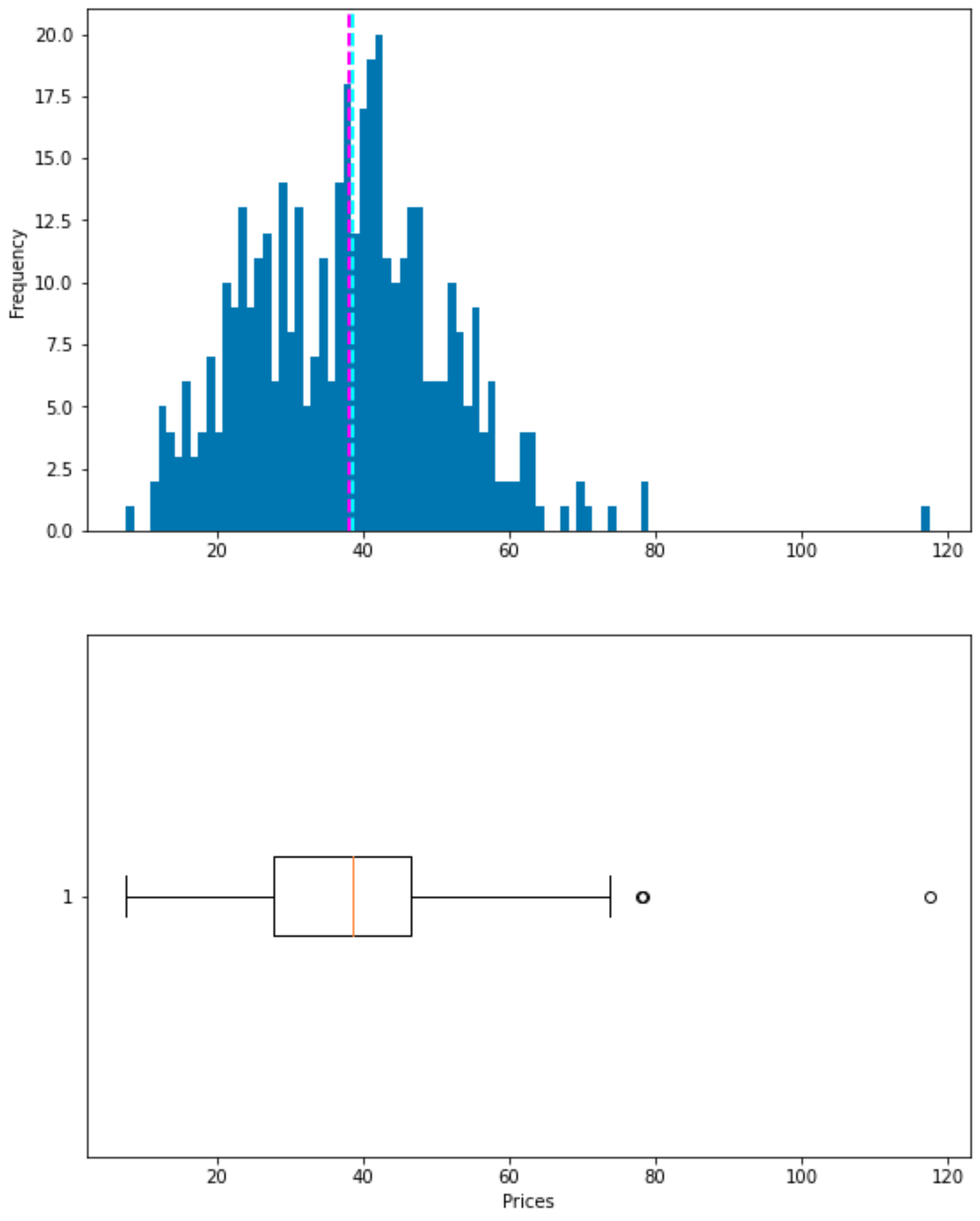
How about outliers!?

```
# let’s look into price distribution

import pandas as pd
import matplotlib.pyplot as plt

# This ensures plots are displayed inline in the Jupyter
notebook
%matplotlib inline

# Get the label column
label = data[‘price_per_unit’]

# Create a figure for 2 subplots (2 rows, 1 column)
fig, ax = plt.subplots(2, 1, figsize = (9,12))

# Plot the histogram
ax[0].hist(label, bins=100)
ax[0].set_ylabel(‘Frequency’)

# Add lines for the mean, median, and mode
ax[0].axvline(label.mean(), color=’magenta’,
linestyle=’dashed’, linewidth=2)
ax[0].axvline(label.median(), color=’cyan’,
linestyle=’dashed’, linewidth=2)

# Plot the boxplot
ax[1].boxplot(label, vert=False)
ax[1].set_xlabel(‘Prices’)
```

> *# Add a title to the Figure*
> *fig.suptitle('Price Distribution')*
>
> *# Show the figure*
> *fig.show()*

The output will be:

Price Distribution

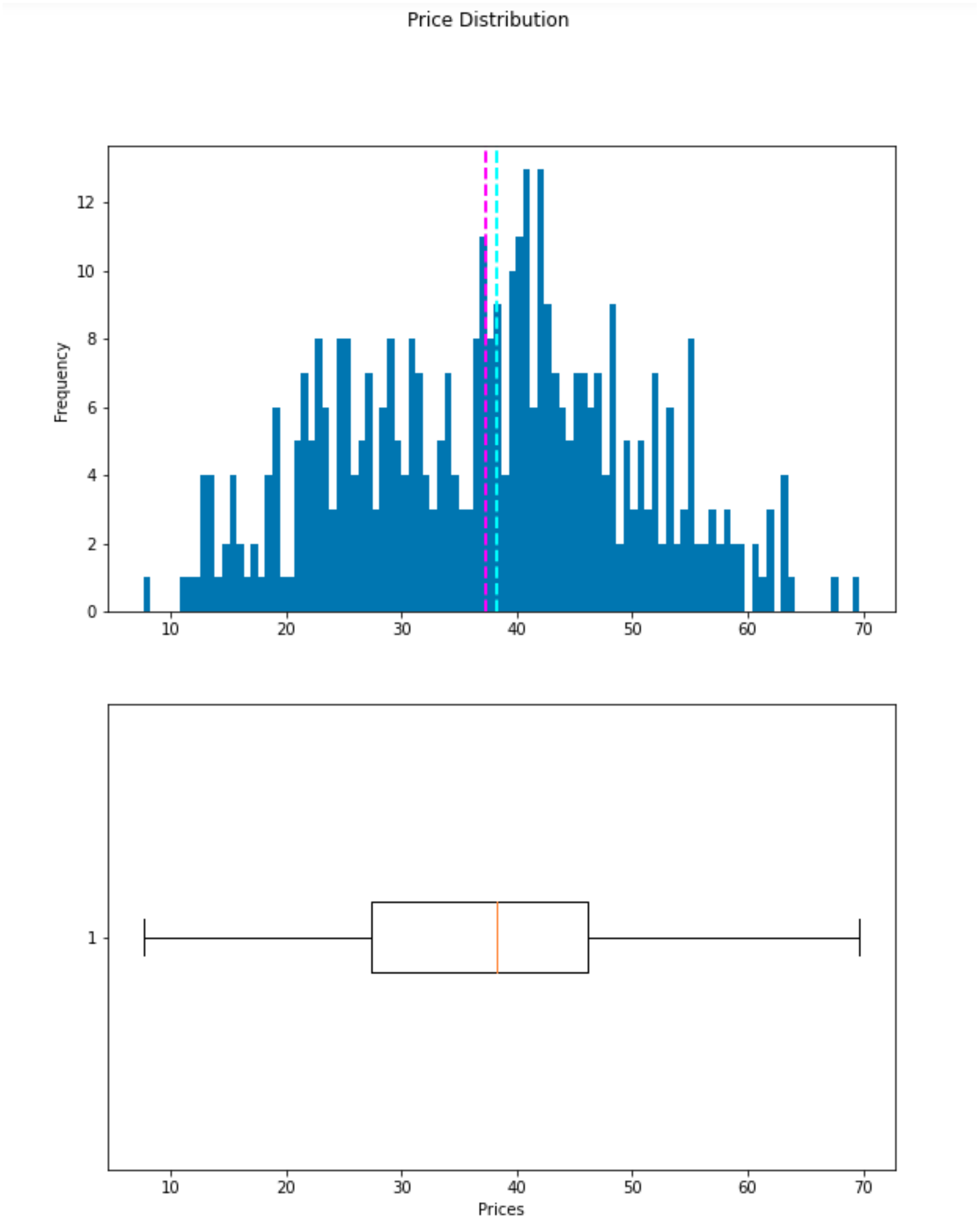


Outliers Exist in Price_per_units!

We can remove the outliers by trimming the dataset. In the above figure, we can see that the highest value on the right side around 70. After 70, those values are considered as outliers. So removing the outliers will look like this:

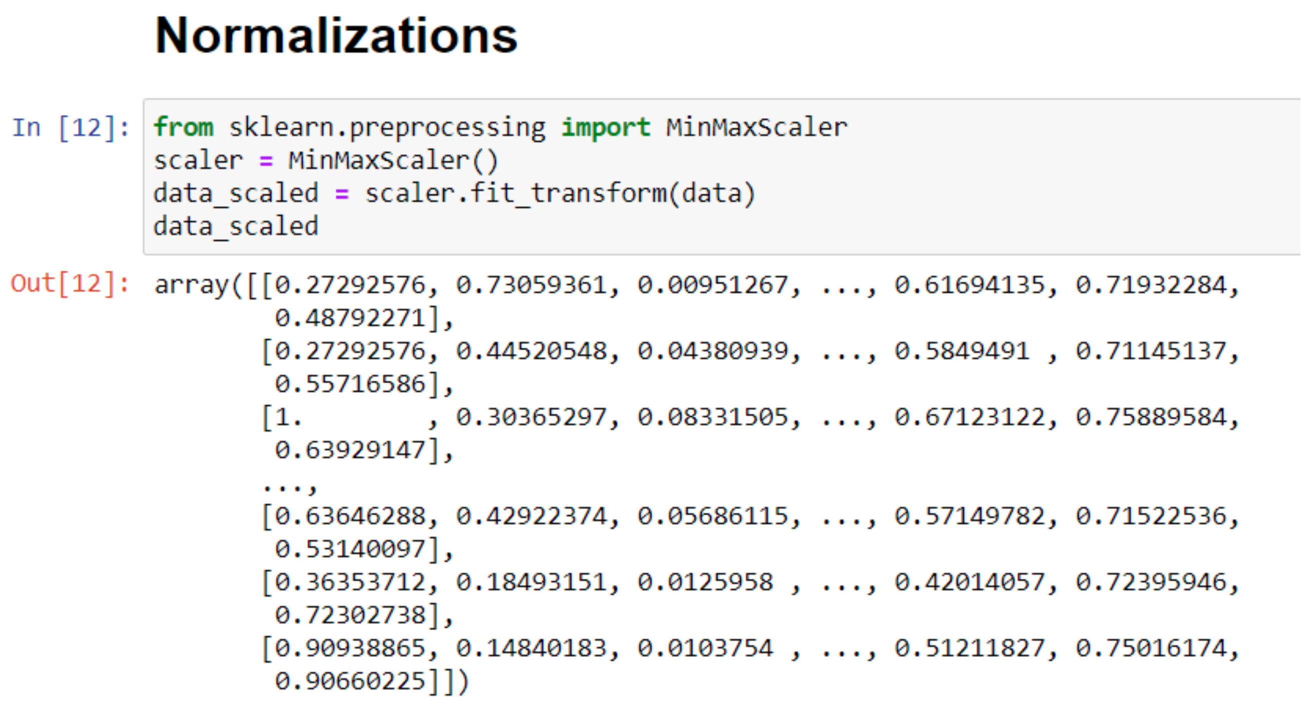
> *# Remove Outliers*
>
> *data = data[data['price_per_unit'] < 70]*

So, if we visualize it again, it will look like this:



Outliers Removed!

Now, I think we did enough with the data to apply regression. However, it is always good practice to normalize the data to keep them in the same space.



Normalizations

```
In [12]: from sklearn.preprocessing import MinMaxScaler
         scaler = MinMaxScaler()
         data_scaled = scaler.fit_transform(data)
         data_scaled

Out[12]: array([[0.27292576, 0.73059361, 0.00951267, ..., 0.61694135, 0.71932284,
                 0.48792271],
                [0.27292576, 0.44520548, 0.04380939, ..., 0.5849491 , 0.71145137,
                 0.55716586],
                [1.        , 0.30365297, 0.08331505, ..., 0.67123122, 0.75889584,
                 0.63929147],
                ...,
                [0.63646288, 0.42922374, 0.05686115, ..., 0.57149782, 0.71522536,
                 0.53140097],
                [0.36353712, 0.18493151, 0.0125958 , ..., 0.42014057, 0.72395946,
                 0.72302738],
                [0.90938865, 0.14840183, 0.0103754 , ..., 0.51211827, 0.75016174,
                 0.90660225]])
```

Normalizing the Data!

After normalizing the data, we need to separate features and labels. We also need to use train_test_split to split the data into training and testing sets.

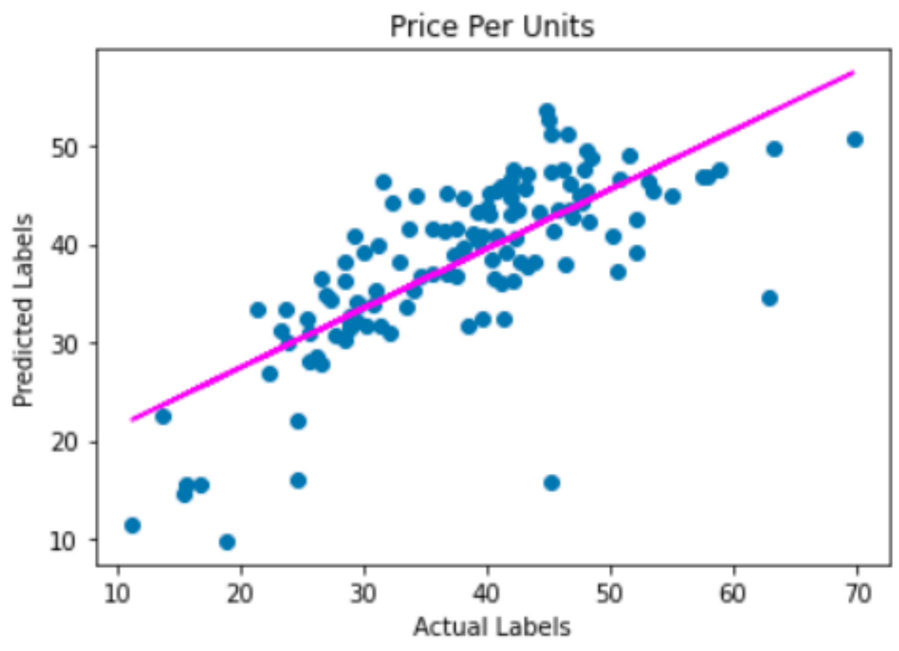
> *# Separate features and labels*
> *X, y = data[['transaction_date', 'house_age', 'transit_distance', 'local_convenience_stores', 'latitude', 'longitude']].values, data['price_per_unit'].values*
>
> *# Split data 70%-30% into training set and test set*
>
> *from sklearn.model_selection import train_test_split*
>
> *X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_state=0)*

We split the dataset into 70% training and 30% testing data. Now, it is time to apply regression algorithms and fit the model with the data.

```
# Train the model
from sklearn.linear_model import LinearRegression

# Fit a linear regression model on the training set
model = LinearRegression().fit(X_train, y_train)

predictions = model.predict(X_test)
```

In this case, we have used the Linear Regression algorithm to fit the model and make the predictions using the testing data. Okay, let’s visualize our regression model based on label test and prediction data.

In [18]:
```python
import matplotlib.pyplot as plt

%matplotlib inline

plt.scatter(y_test, predictions)
plt.xlabel('Actual Labels')
plt.ylabel('Predicted Labels')
plt.title('Price Per Units')
# overlay the regression line
z = np.polyfit(y_test, predictions, 1)
p = np.poly1d(z)
plt.plot(y_test,p(y_test), color='magenta')
plt.show()
```



Pretty good regression model!

> We can also use Mean Squared Error (MSE) and R2 Score to evaluate the model. MSE actually identifies how close a regression line is to a set of points. The equation for MSE is given below:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

MSE	= mean squared error
n	= number of data points
Y_i	= observed values
$\hat{Y}_i$	= predicted values

Taken From WikiPedia!

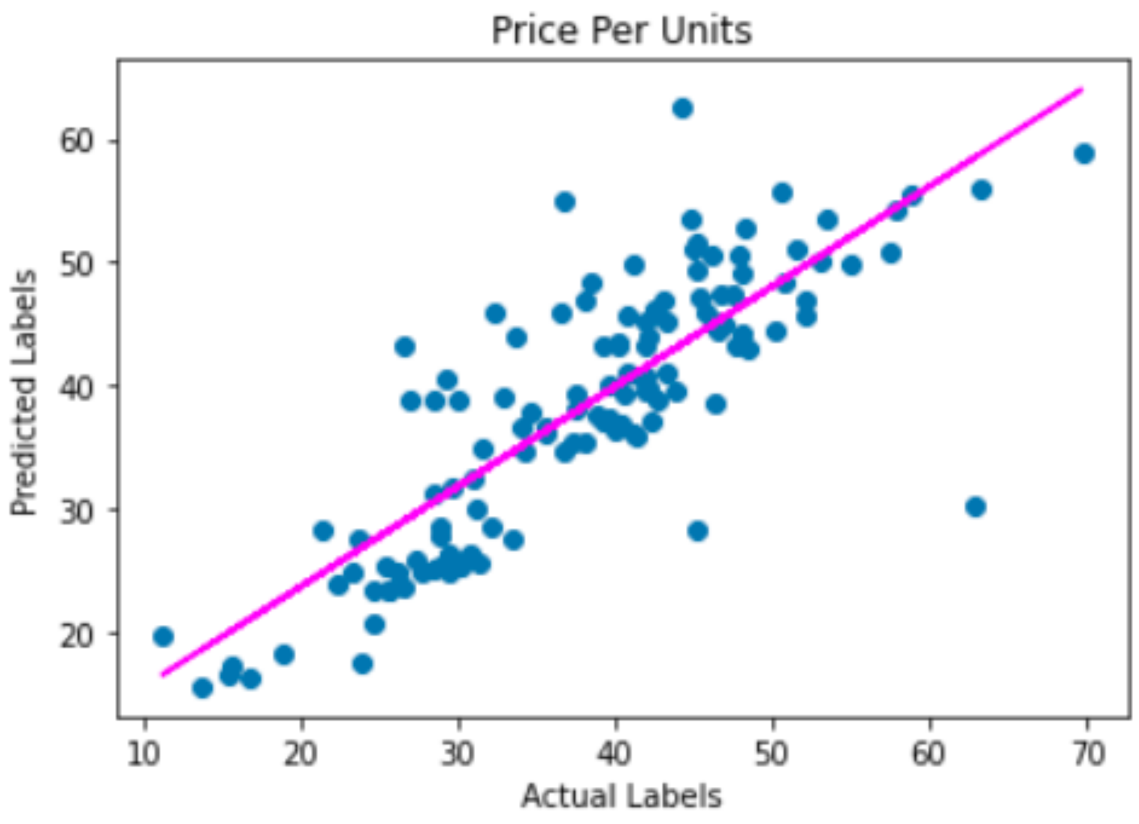
> "**R-squared** (**R2**) is a statistical measure that represents the proportion of the variance for a dependent variable that's explained by an independent variable or variables in a regression model". A detail descriptions can be found here!

```
from sklearn.metrics import mean_squared_error,
r2_score
from sklearn.ensemble import
GradientBoostingRegressor

# Fit a lasso model on the training set
model = GradientBoostingRegressor().fit(X_train,
y_train)
print (model, "\n")

# Evaluate the model using the test data
predictions = model.predict(X_test)
mse = mean_squared_error(y_test, predictions)
print("MSE:", mse)
rmse = np.sqrt(mse)
print("RMSE:", rmse)
```

> *r2 = r2_score(y_test, predictions)*
> *print("R2:", r2)*

The output will be like the following:

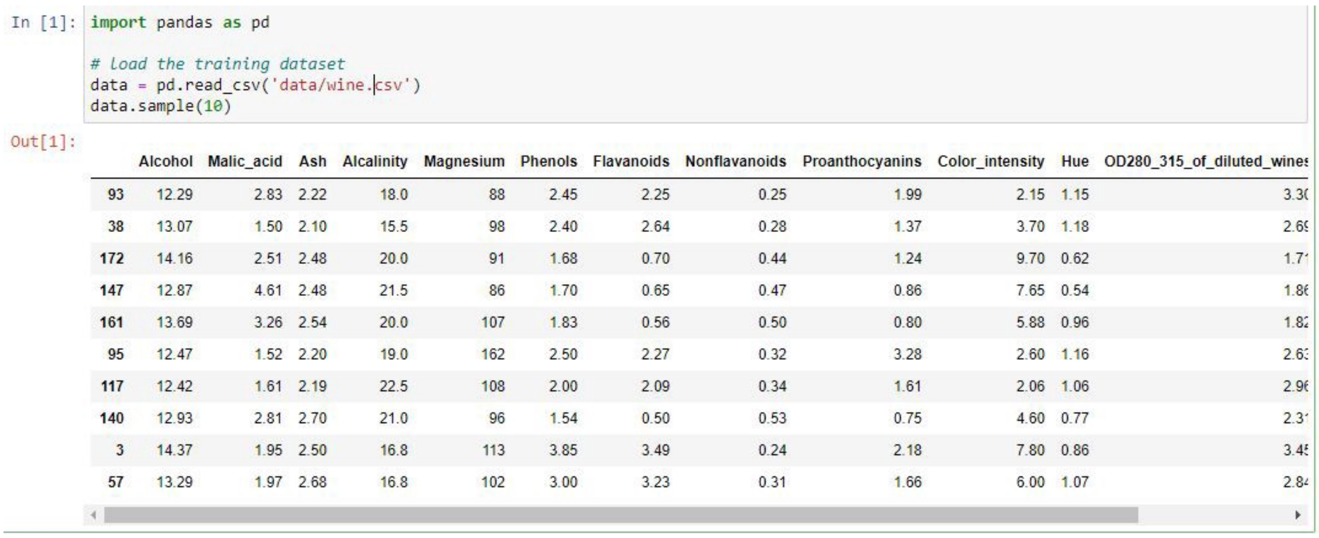
```
MSE: 40.1883830523336
RMSE: 6.3394308145395515
R2: 0.6570707121373265
```



MSE and R2!

Okay, enough about regression. Let's talk about classification now!

Classification usually falls into the category of recognizing something based on a pre-existing training dataset. In the classification problem, each features combined to make a label. Therefore, there can be multiple labels exists in a dataset. For example, recognizing the fingers from the hand would be a classification problem. In this case, I will be using the Wine Classification problem. The dataset can be found here! Let's start by loading the dataset:



```
In [1]: import pandas as pd

        # Load the training dataset
        data = pd.read_csv('data/wine.csv')
        data.sample(10)
```

Out[1]:

	Alcohol	Malic_acid	Ash	Alcalinity	Magnesium	Phenols	Flavanoids	Nonflavanoids	Proanthocyanins	Color_intensity	Hue	OD280_315_of_diluted_wine
93	12.29	2.83	2.22	18.0	88	2.45	2.25	0.25	1.99	2.15	1.15	3.3
38	13.07	1.50	2.10	15.5	98	2.40	2.64	0.28	1.37	3.70	1.18	2.6
172	14.16	2.51	2.48	20.0	91	1.68	0.70	0.44	1.24	9.70	0.62	1.7
147	12.87	4.61	2.48	21.5	86	1.70	0.65	0.47	0.86	7.65	0.54	1.8
161	13.69	3.26	2.54	20.0	107	1.83	0.56	0.50	0.80	5.88	0.96	1.8
95	12.47	1.52	2.20	19.0	162	2.50	2.27	0.32	3.28	2.60	1.16	2.6
117	12.42	1.61	2.19	22.5	108	2.00	2.09	0.34	1.61	2.06	1.06	2.9
140	12.93	2.81	2.70	21.0	96	1.54	0.50	0.53	0.75	4.60	0.77	2.3
3	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.4
57	13.29	1.97	2.68	16.8	102	3.00	3.23	0.31	1.66	6.00	1.07	2.8

Wine Classification Dataset!

And the labels are:

- **0** (*variety A*)
- **1** (*variety B*)
- **2** (*variety C*)

Now, follow the same steps to clean the data and remove the outliers as we have seen in the regression part. Therefore, we have to separate the dataset into a training and testing set.

> *features = ['Alcohol', 'Malic_acid', 'Ash', 'Alcalinity', 'Magnesium', 'Phenols',*
> *'Flavanoids', 'Nonflavanoids', 'Proanthocyanins', 'Color_intensity',*
> *'Hue', 'OD280_315_of_diluted_wines', 'Proline']*
> *label = 'WineVariety'*
> *X, y = data[features].values, data[label].values*

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size = 0.30, random_state=0)
print(‘Training Class: %d\nTestig Class: %d’ %
(X_train.shape[0], X_test.shape[0]))
```

And the output would be:

```
Training Class: 124
Testig Class: 54
```

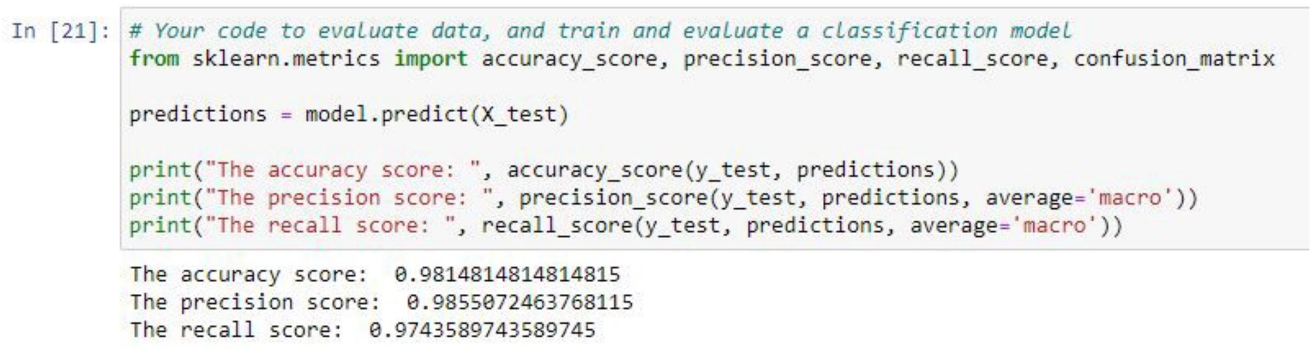
From the above code, it is clear that we have taken 70% of the data as a training set and 30% of the data as a testing set.

Now, I am going to use sklearn pipelining to preprocess the dataset and fed them into a classifier. ***"Pipeline is just an abstract notion, it's not some existing ml algorithm. Often in ML tasks, you need to perform a sequence of different transformations (find set of features, generate new features, select only some good features) of the raw dataset before applying final estimator."*** Also, we can look into the details of pipelining here!

```
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
```

> *feature_columns = [0,1,2,3,4,5,6]*
> *feature_transformer = Pipeline(steps=[(‘scaler’, StandardScaler())])*
>
> *# Create preprocessing steps*
> *preprocessor = ColumnTransformer(transformers=[(‘preprocess’, feature_transformer, feature_columns)])*
>
> *# Create training pipeline*
> *pipeline = Pipeline(steps=[(‘preprocessor’, preprocessor),*
> *(‘regressor’, LogisticRegression(solver=’lbfgs’, multi_class=’auto’))])*
>
> *# fit the pipeline to train a linear regression model on the training set*
> *model = pipeline.fit(X_train, y_train)*

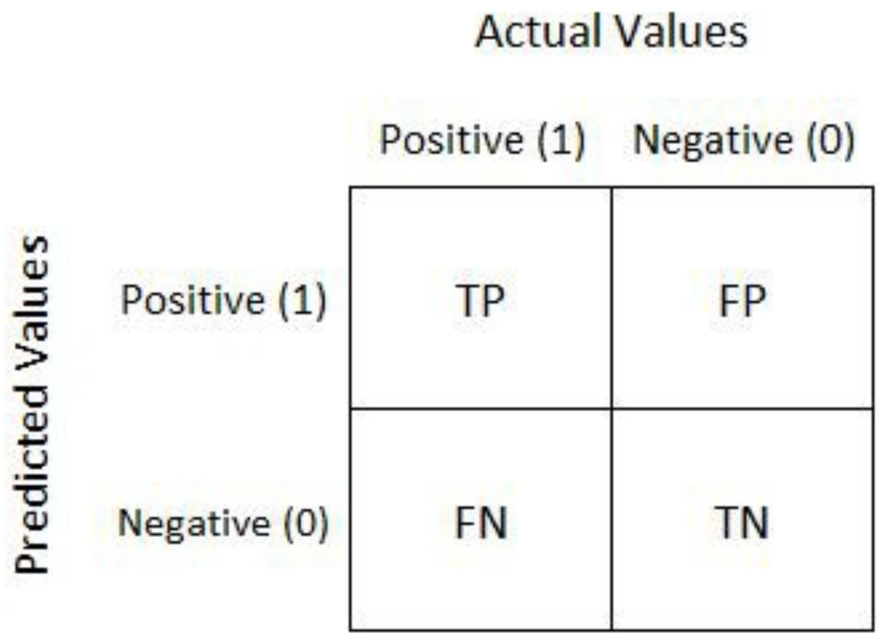
And if we predict the model with the test data, then the model outcome would be:



```
In [21]: # Your code to evaluate data, and train and evaluate a classification model
         from sklearn.metrics import accuracy_score, precision_score, recall_score, confusion_matrix

         predictions = model.predict(X_test)

         print("The accuracy score: ", accuracy_score(y_test, predictions))
         print("The precision score: ", precision_score(y_test, predictions, average='macro'))
         print("The recall score: ", recall_score(y_test, predictions, average='macro'))

         The accuracy score:  0.9814814814814815
         The precision score:  0.9855072463768115
         The recall score:  0.9743589743589745
```

Model Evaluations!

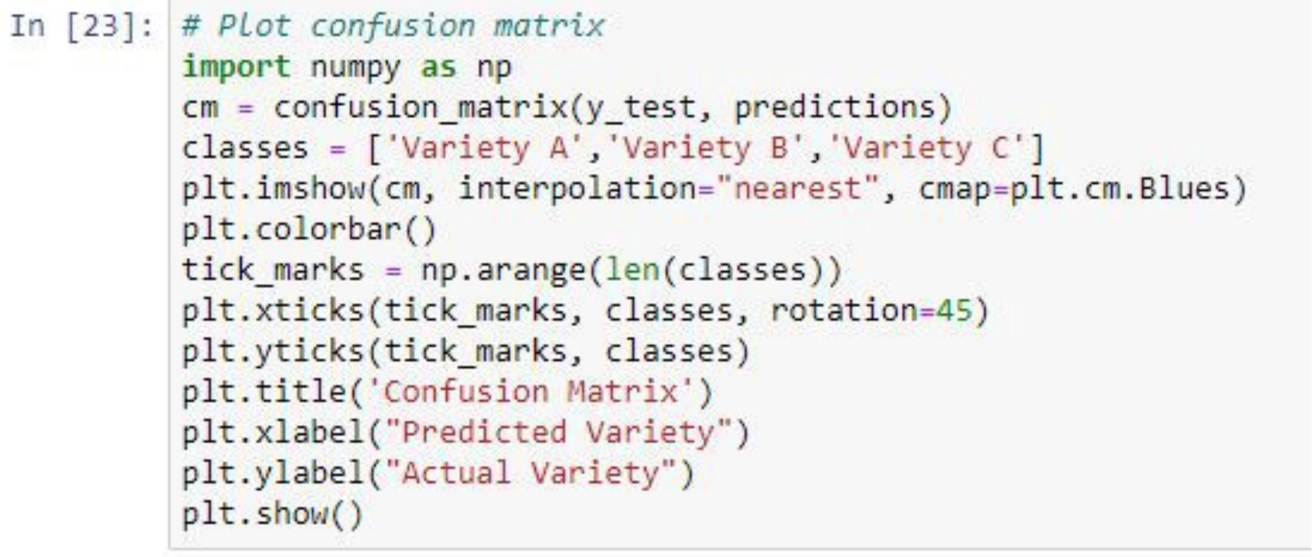
Now, I will show how to evaluate the effectiveness of the model by using a confusion matrix. Confusion Matrix is a performance measurement for a machine learning classification problem where output can be two or more classes. It is a table with 4 different combinations of predicted and actual values. It is useful for measuring Recall, Precision, Accuracy, and AUC-ROC curves.



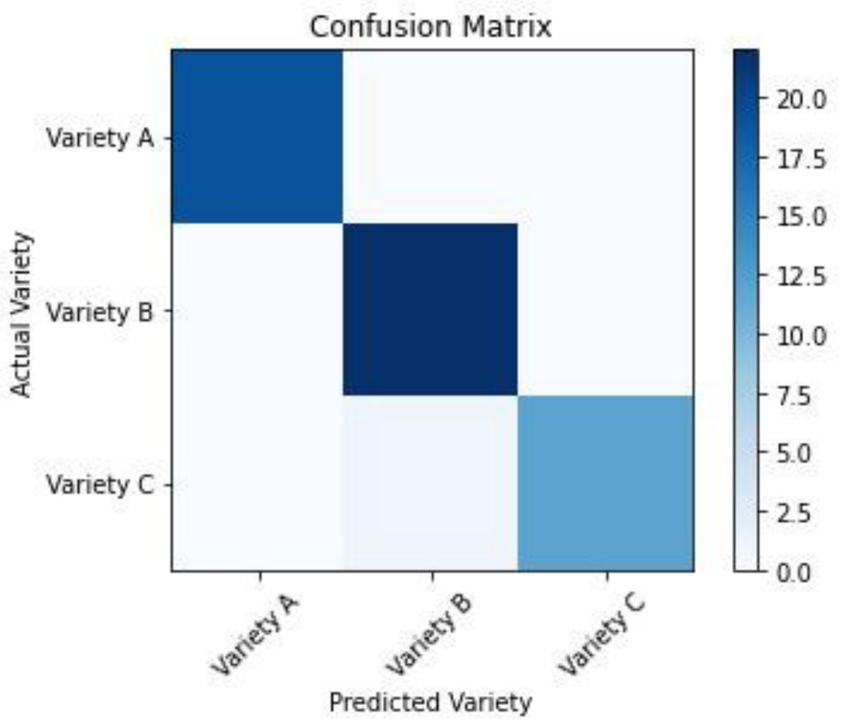
		Actual Values	
		Positive (1)	Negative (0)
Predicted Values	Positive (1)	TP	FP
	Negative (0)	FN	TN

Confusion Matrix Combinations!

In the above figure, TP is true positive which means that your prediction is positive and it is true. FP is false positive which means that your prediction is positive but it's false. FN is false negative which means that your prediction is negative and it's false. Lastly, TN is true negative which means that your prediction is negative and it's true. Also, we can calculate the Recall = TP/(TP+FN), precision = TP/(TP+FP), and F-measures = (2*recall*precision)/ (recall + precision). Okay, let's look at the confusion matrix for our classification problem.

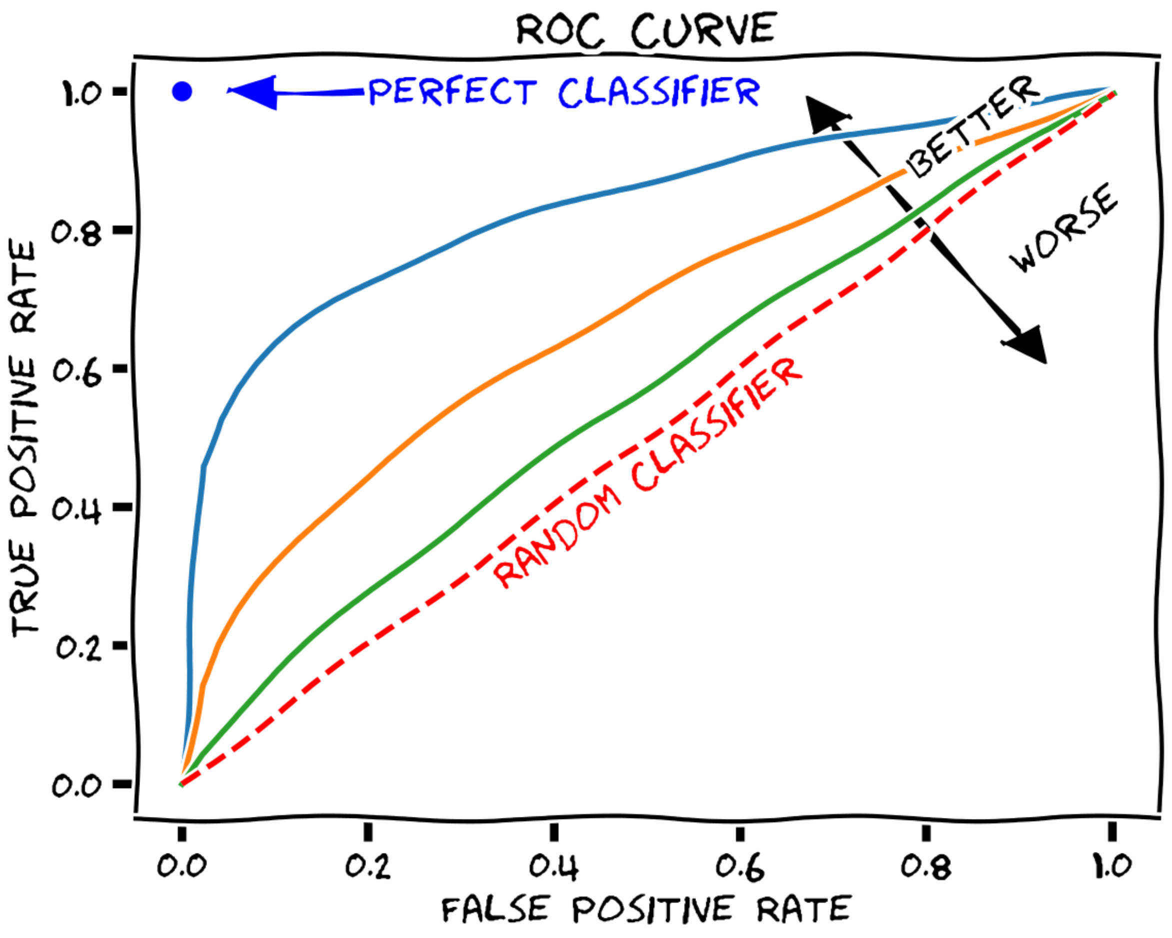


```
In [23]: # Plot confusion matrix
         import numpy as np
         cm = confusion_matrix(y_test, predictions)
         classes = ['Variety A','Variety B','Variety C']
         plt.imshow(cm, interpolation="nearest", cmap=plt.cm.Blues)
         plt.colorbar()
         tick_marks = np.arange(len(classes))
         plt.xticks(tick_marks, classes, rotation=45)
         plt.yticks(tick_marks, classes)
         plt.title('Confusion Matrix')
         plt.xlabel("Predicted Variety")
         plt.ylabel("Actual Variety")
         plt.show()
```



Confusion Matrix!

In addition to the confusion matrix, the roc curve also shows how good your classifier is!



[Taken from Web!]

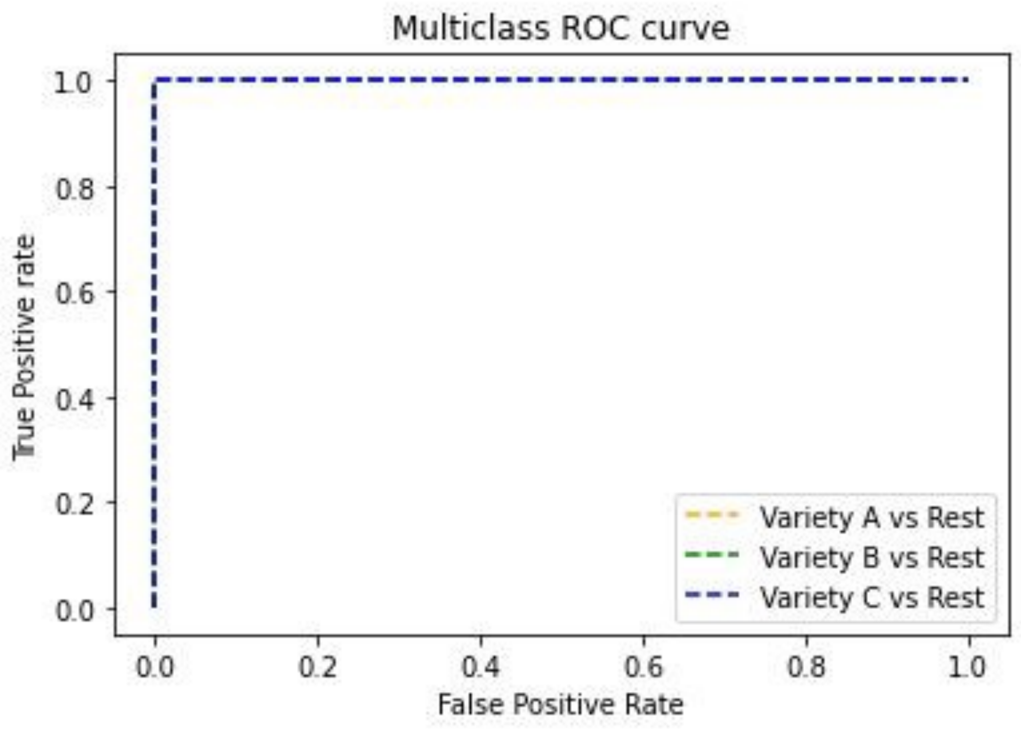
In the above figure, we can clearly understand that what kind of graph will tell us if the classifier is good or not! Well, let's look at our ROC curve!

> *from sklearn.metrics import roc_curve, roc_auc_score*
>
> *# Get class probability scores*
> *probabilities = model.predict_proba(X_test)*
>
> *auc = roc_auc_score(y_test,probabilities, multi_class='ovr')*
> *print('Average AUC:', auc)*
>
> *# Get ROC metrics for each class*
> *fpr = {}*

```
tpr = {}
thresh ={}
for i in range(len(classes)):
fpr[i], tpr[i], thresh[i] = roc_curve(y_test,
probabilities[:,i], pos_label=i)

# Plot the ROC chart
plt.plot(fpr[0], tpr[0], linestyle=' — ',color='orange',
label=classes[0] + ' vs Rest')
plt.plot(fpr[1], tpr[1], linestyle=' — ',color='green',
label=classes[1] + ' vs Rest')
plt.plot(fpr[2], tpr[2], linestyle=' — ',color='blue',
label=classes[2] + ' vs Rest')
plt.title('Multiclass ROC curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive rate')
plt.legend(loc='best')
plt.show()
```

The output would be:

```
Average AUC: 1.0
```

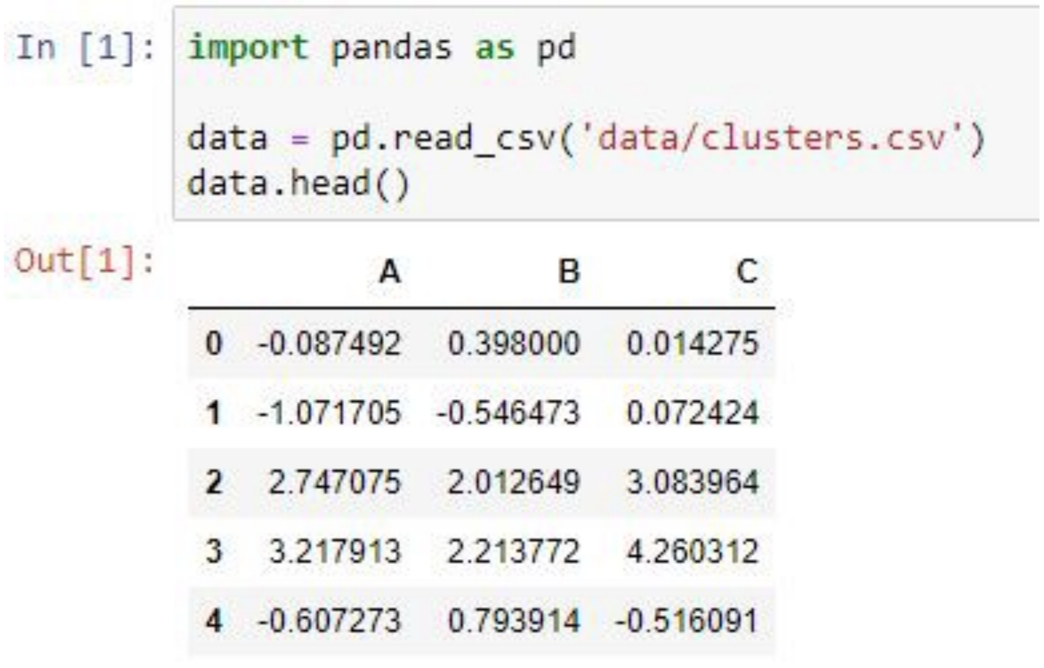


ROC curve of our classification!

Therefore, this classification model is the perfect classifier!

Clustering comes in when we can not do the regression and classification of the data. That being said, clustering is distributing the data to its closest group. ***Clustering is an unsupervised machine learning technique in which you train a model to group similar entities into clusters based on their features.***

In this example, I will be using a pre-defined dataset! You can use any dataset you would like! Let’s load the data first!


```
In [1]: import pandas as pd

        data = pd.read_csv('data/clusters.csv')
        data.head()
```

Out[1]:

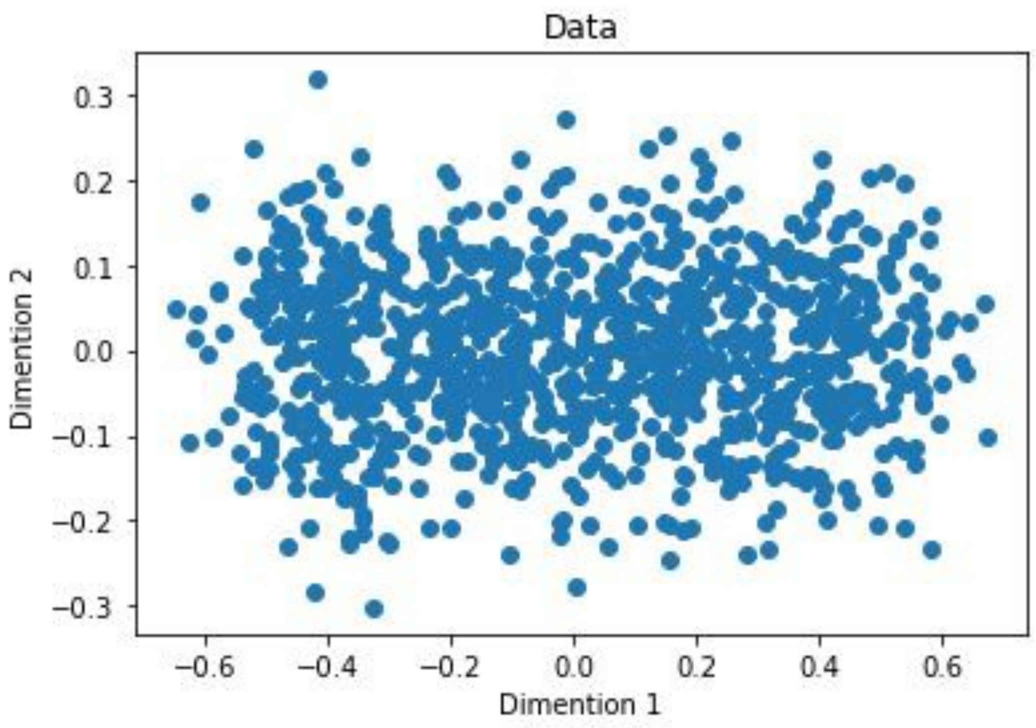
	A	B	C
0	-0.087492	0.398000	0.014275
1	-1.071705	-0.546473	0.072424
2	2.747075	2.012649	3.083964
3	3.217913	2.213772	4.260312
4	-0.607273	0.793914	-0.516091

Dataset!

The challenge is to identify the number of discrete clusters present in the data and create a clustering model that separates the data into that number of clusters. Another challenge is also to visualize the clusters to evaluate the level of separation achieved by your model. First off, let's use PCA to make 2D version visualization. ***"PCA is a dimensionality reduction method that is often used to reduce the dimensionality of large data sets, by transforming a large set of variables into a smaller one that still contains most of the information in the large set."*** Therefore, we will be using it to make the dataset into a 2D version and then visualize it.

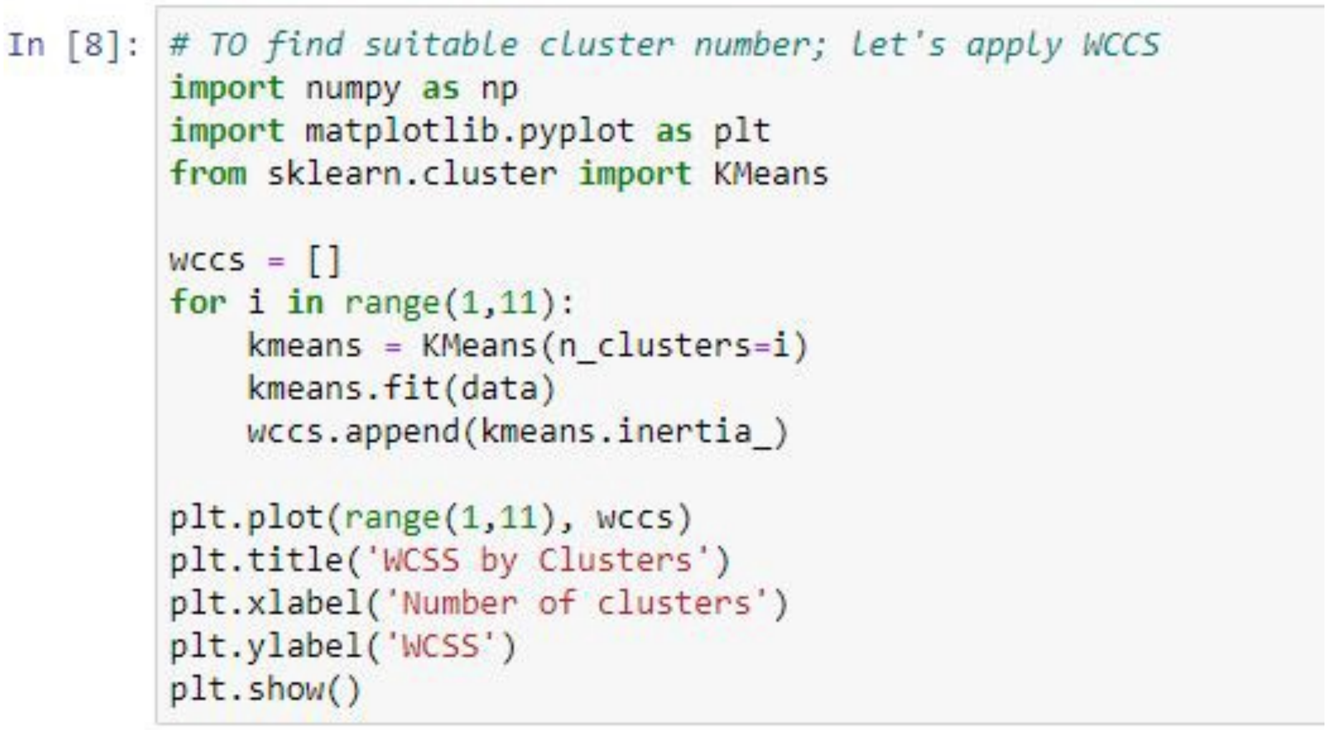
> *from sklearn.preprocessing import MinMaxScaler*
> *from sklearn.decomposition import PCA*
> *import matplotlib.pyplot as plt*
> *%matplotlib inline*
>
> *scaled_features = MinMaxScaler().fit_transform(data)*

```
pca = PCA(n_components=2).fit(scaled_features)
features2D = pca.transform(scaled_features)

plt.scatter(features2D[:,0], features2D[:,1])
plt.xlabel('Dimention 1')
plt.ylabel('Dimention 2')
plt.title('Data')
plt.show()
```

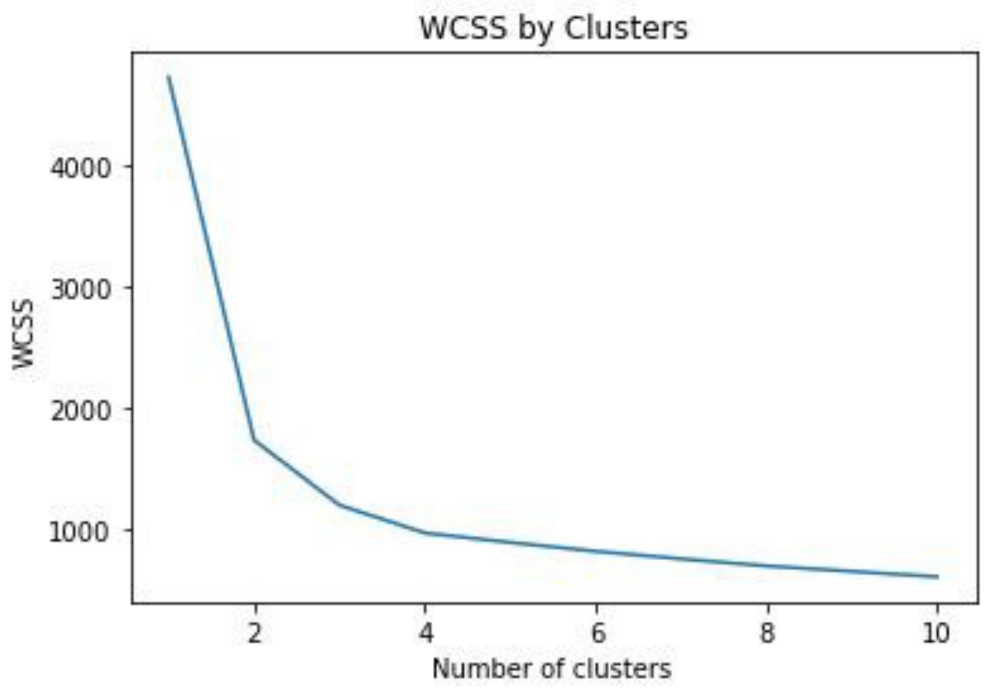


2D visualization of the data using PCA!

We can also run multiple cluster size examples using the KMeans clustering algorithm to find the suitable cluster size. Okay, let's do that!



```
In [8]: # TO find suitable cluster number; let's apply WCCS
        import numpy as np
        import matplotlib.pyplot as plt
        from sklearn.cluster import KMeans

        wccs = []
        for i in range(1,11):
            kmeans = KMeans(n_clusters=i)
            kmeans.fit(data)
            wccs.append(kmeans.inertia_)

        plt.plot(range(1,11), wccs)
        plt.title('WCSS by Clusters')
        plt.xlabel('Number of clusters')
        plt.ylabel('WCSS')
        plt.show()
```



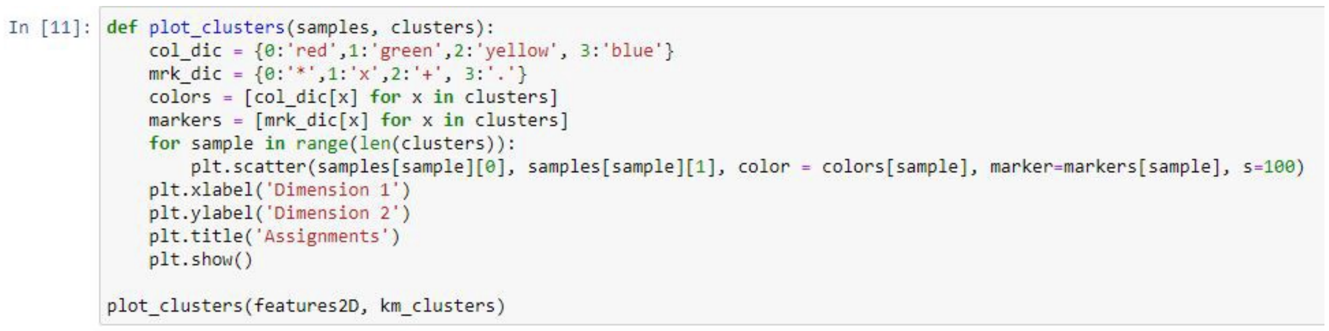
Finding suitable cluster size!

In the above figure, we can see that there are four edges or elbows if you call it. These elbows tell us that there can be four possible clusters that exists in this dataset! Cool, right? 😆

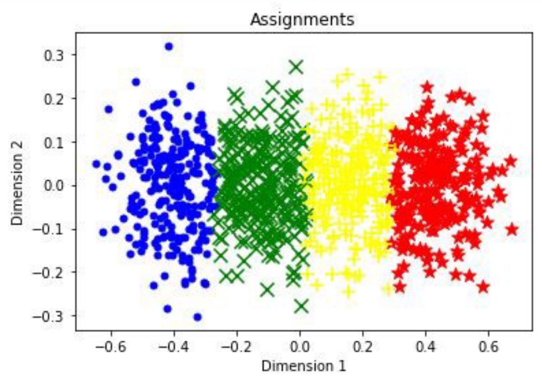
Now, we are going to use the KMeans cluster with 4 cluster sizes!

> *from sklearn.cluster import KMeans*
>
> *model = KMeans(n_clusters=4, init='k-means++', n_init=500, max_iter=1500)*
> *km_clusters = model.fit_predict(data)*

In the above code, n_init means 500 times of training and max_iter 1500 means that in each training, there can be a maximum of 1500 iterations. Now, let's visualize the clusters!



```
In [11]: def plot_clusters(samples, clusters):
             col_dic = {0:'red',1:'green',2:'yellow', 3:'blue'}
             mrk_dic = {0:'*',1:'x',2:'+', 3:'.'}
             colors = [col_dic[x] for x in clusters]
             markers = [mrk_dic[x] for x in clusters]
             for sample in range(len(clusters)):
                 plt.scatter(samples[sample][0], samples[sample][1], color = colors[sample], marker=markers[sample], s=100)
             plt.xlabel('Dimension 1')
             plt.ylabel('Dimension 2')
             plt.title('Assignments')
             plt.show()

         plot_clusters(features2D, km_clusters)
```



Four Clusters!

We have got our clusters right? Yeah looks cool! Clustering can be useful to make the classification as well. Think about it! 😉

That's it for today! Cheers! 😃