# BOSTON DATASET

```
pip install scikit-learn==1.1.3
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting scikit-learn==1.1.3
  Downloading scikit_learn-1.1.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (30.5 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 30.5/30.5 MB 20.9 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.10/dist-packages (from scikit-learn==1.1.3) (1.22.4)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.10/dist-packages (from scikit-learn==1.1.3) (1.10.1)
Requirement already satisfied: joblib>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn==1.1.3) (1.2.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn==1.1.3) (3.1.0)
Installing collected packages: scikit-learn
  Attempting uninstall: scikit-learn
    Found existing installation: scikit-learn 1.2.2
    Uninstalling scikit-learn-1.2.2:
      Successfully uninstalled scikit-learn-1.2.2
Successfully installed scikit-learn-1.1.3
```

```
import numpy as np
import pandas as pd
# Importing the Boston Housing dataset from the sklearn
from sklearn.datasets import load_boston
boston = load_boston()
#Converting the data into pandas dataframe
data = pd.DataFrame(boston.data)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/utils/deprecation.py:87: FutureWarning: Function load_boston is deprecated; `load_bostor

    The Boston housing prices dataset has an ethical problem. You can refer to
    the documentation of this function for further details.

    The scikit-learn maintainers therefore strongly discourage the use of this
    dataset unless the purpose of the code is to study and educate about
    ethical issues in data science and machine learning.

    In this special case, you can fetch the dataset from the original
    source::

        import pandas as pd
        import numpy as np

        data_url = "http://lib.stat.cmu.edu/datasets/boston"
        raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
        data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
        target = raw_df.values[1::2, 2]

    Alternative datasets include the California housing dataset (i.e.
    :func:`~sklearn.datasets.fetch_california_housing`) and the Ames housing
    dataset. You can load the datasets as follows::

        from sklearn.datasets import fetch_california_housing
        housing = fetch_california_housing()

    for the California housing dataset and::

        from sklearn.datasets import fetch_openml
        housing = fetch_openml(name="house_prices", as_frame=True)

    for the Ames housing dataset.
  warnings.warn(msg, category=FutureWarning)
```

```
data.head()
#Adding the feature names to the dataframe
data.columns = boston.feature_names
#Adding the target variable to the dataset
data['PRICE'] = boston.target
#Looking at the data with names and target variable
data.head(n=10)
```

| | CRIM | ZN | INDUS | CHAS | NOX | RM | AGE | DIS | RAD | TAX | PTRATIO | B | LSTAT | PRICE |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.00632 | 18.0 | 2.31 | 0.0 | 0.538 | 6.575 | 65.2 | 4.0900 | 1.0 | 296.0 | 15.3 | 396.90 | 4.98 | 24.0 |
| **1** | 0.02731 | 0.0 | 7.07 | 0.0 | 0.469 | 6.421 | 78.9 | 4.9671 | 2.0 | 242.0 | 17.8 | 396.90 | 9.14 | 21.6 |
| **2** | 0.02729 | 0.0 | 7.07 | 0.0 | 0.469 | 7.185 | 61.1 | 4.9671 | 2.0 | 242.0 | 17.8 | 392.83 | 4.03 | 34.7 |
| **3** | 0.03237 | 0.0 | 2.18 | 0.0 | 0.458 | 6.998 | 45.8 | 6.0622 | 3.0 | 222.0 | 18.7 | 394.63 | 2.94 | 33.4 |
| **4** | 0.06905 | 0.0 | 2.18 | 0.0 | 0.458 | 7.147 | 54.2 | 6.0622 | 3.0 | 222.0 | 18.7 | 396.90 | 5.33 | 36.2 |
| **5** | 0.02985 | 0.0 | 2.18 | 0.0 | 0.458 | 6.430 | 58.7 | 6.0622 | 3.0 | 222.0 | 18.7 | 394.12 | 5.21 | 28.7 |
| **6** | 0.08829 | 12.5 | 7.87 | 0.0 | 0.524 | 6.012 | 66.6 | 5.5605 | 5.0 | 311.0 | 15.2 | 395.60 | 12.43 | 22.9 |

```
#Shape of the data
print(data.shape)
#Checking the null values in the dataset
data.isnull().sum()
data.describe()
```

```
(506, 14)
```

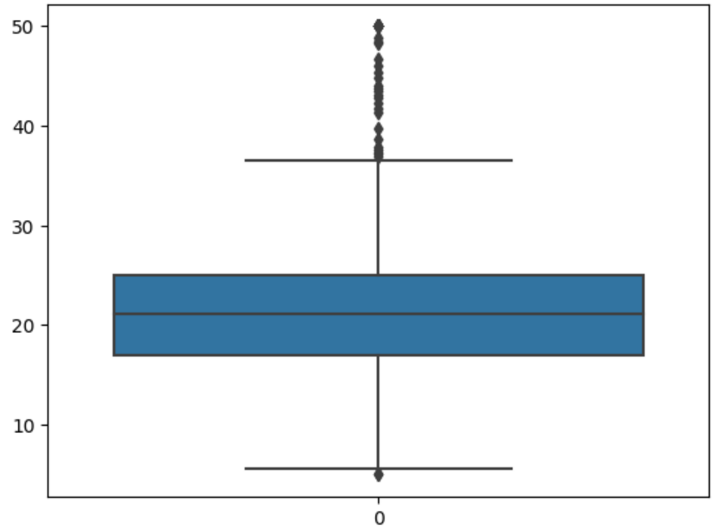
| | CRIM | ZN | INDUS | CHAS | NOX | RM | AGE | DIS | RAD | TAX | PTRATIO | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **count** | 506.000000 | 506.000000 | 506.000000 | 506.000000 | 506.000000 | 506.000000 | 506.000000 | 506.000000 | 506.000000 | 506.000000 | 506.000000 | 50 |
| **mean** | 3.613524 | 11.363636 | 11.136779 | 0.069170 | 0.554695 | 6.284634 | 68.574901 | 3.795043 | 9.549407 | 408.237154 | 18.455534 | 35 |
| **std** | 8.601545 | 23.322453 | 6.860353 | 0.253994 | 0.115878 | 0.702617 | 28.148861 | 2.105710 | 8.707259 | 168.537116 | 2.164946 | 9 |
| **min** | 0.006320 | 0.000000 | 0.460000 | 0.000000 | 0.385000 | 3.561000 | 2.900000 | 1.129600 | 1.000000 | 187.000000 | 12.600000 | |
| **25%** | 0.082045 | 0.000000 | 5.190000 | 0.000000 | 0.449000 | 5.885500 | 45.025000 | 2.100175 | 4.000000 | 279.000000 | 17.400000 | 37 |
| **50%** | 0.256510 | 0.000000 | 9.690000 | 0.000000 | 0.538000 | 6.208500 | 77.500000 | 3.207450 | 5.000000 | 330.000000 | 19.050000 | 39 |
| **75%** | 3.677083 | 12.500000 | 18.100000 | 0.000000 | 0.624000 | 6.623500 | 94.075000 | 5.188425 | 24.000000 | 666.000000 | 20.200000 | 39 |
| **max** | 88.976200 | 100.000000 | 27.740000 | 1.000000 | 0.871000 | 8.780000 | 100.000000 | 12.126500 | 24.000000 | 711.000000 | 22.000000 | 39 |

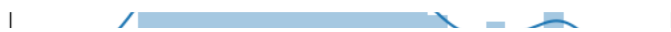
```
import seaborn as sns
sns.distplot(data.PRICE)
```

```
<ipython-input-6-5528f5f0d0e9>:2: UserWarning:
```

```
#Distribution using box plot
sns.boxplot(data.PRICE)
```

```
<Axes: >
```



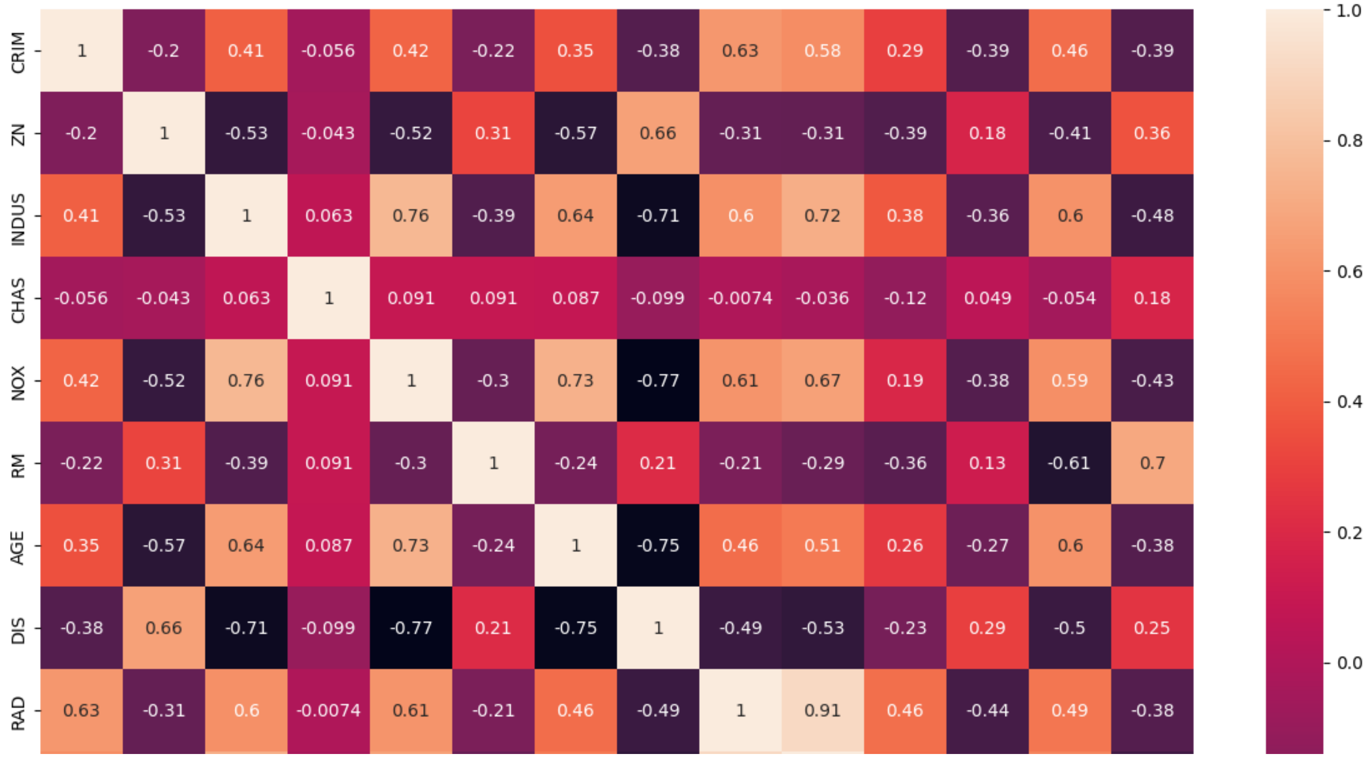


```
correlation = data.corr()
correlation.loc['PRICE']
```

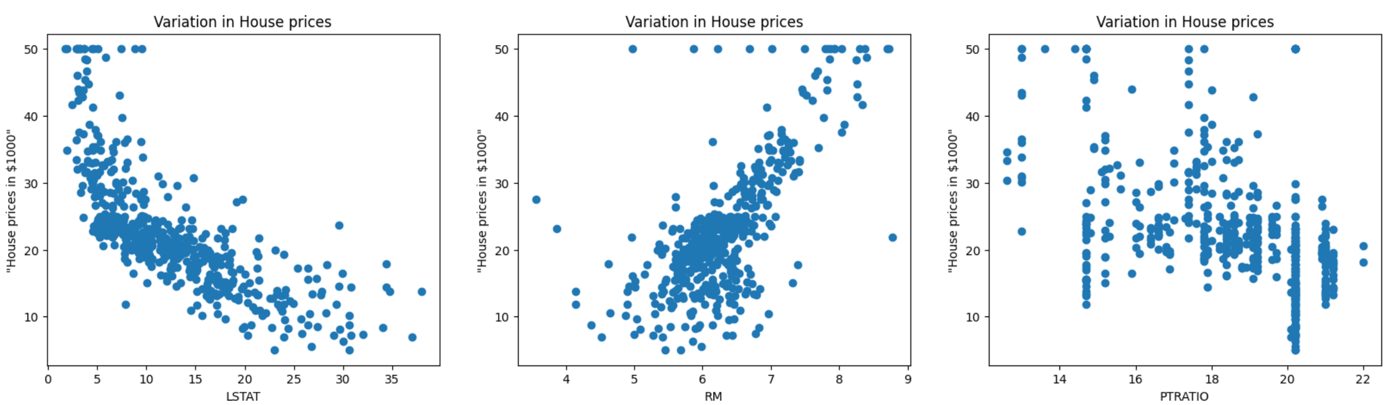
```
CRIM      -0.388305
ZN         0.360445
INDUS     -0.483725
CHAS       0.175260
NOX       -0.427321
RM         0.695360
AGE       -0.376955
DIS        0.249929
RAD       -0.381626
TAX       -0.468536
PTRATIO   -0.507787
B          0.333461
LSTAT     -0.737663
PRICE      1.000000
Name: PRICE, dtype: float64
```

```
import matplotlib.pyplot as plt
fig,axes = plt.subplots(figsize=(15,12))
sns.heatmap(correlation,square = True,annot = True)
# By looking at the correlation plot LSAT is negatively correlated with -0.75 and RM is positively correlated to the price and PTRATIO is cor
```

```
<Axes: >
```




```
# Checking the scatter plot with the most correlated features
plt.figure(figsize = (20,5))
features = ['LSTAT','RM','PTRATIO']
for i, col in enumerate(features):
  plt.subplot(1, len(features) , i+1)
  x = data[col]
  y = data.PRICE
  plt.scatter(x, y, marker='o')
  plt.title("Variation in House prices")
  plt.xlabel(col)
  plt.ylabel('"House prices in $1000"')
```




```
X = data.iloc[:,:-1]
y= data.PRICE
```

```
import keras
from keras.layers import Dense, Activation,Dropout
from keras.models import Sequential
model = Sequential()
model.add(Dense(128,activation = 'relu',input_dim =13))
model.add(Dense(64,activation = 'relu'))
model.add(Dense(32,activation = 'relu'))
model.add(Dense(16,activation = 'relu'))
model.add(Dense(1))
```

```
#model.compile(optimizer='adam', loss='mse', metrics=['mae'])
model.compile(optimizer = 'adam',loss ='mean_squared_error',metrics=['mae'])
```

```
!pip install ann_visualizer
!pip install graphviz
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting ann_visualizer
  Downloading ann_visualizer-2.5.tar.gz (4.7 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: ann_visualizer
  Building wheel for ann_visualizer (setup.py) ... done
  Created wheel for ann_visualizer: filename=ann_visualizer-2.5-py3-none-any.whl size=4167 sha256=31ee98ab933ffbaf3897b525d37b8363c2c4e6
  Stored in directory: /root/.cache/pip/wheels/6e/0f/ae/f5dba91db71b1b32bf03d0ad18c32e86126093aba5ec6b6488
Successfully built ann_visualizer
Installing collected packages: ann_visualizer
Successfully installed ann_visualizer-2.5
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: graphviz in /usr/local/lib/python3.10/dist-packages (0.20.1)
```

```
# Make predictions on new data
import sklearn
new_data = sklearn.preprocessing.StandardScaler().fit_transform(([[100.1, 10.0,5.0, 0, 0.4, 6.0, 50, 6.0, 1, 400, 20, 300, 10]]))
prediction = model.predict(new_data)
print("Predicted house price:", prediction)
```

```
1/1 [==============================] - 0s 35ms/step
Predicted house price: [[0.]]
```

# IMDB Dataset

```
#Importing the pandas for data processing and numpy for numerical
import numpy as np
import pandas as pd
```

```
#loading imdb data with most frequent 10000 words

from keras.datasets import imdb
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=10000) # you may take top 10,000 word frequently review of movies
#consolidating data for EDA(exploratory data analysis: involves gathering all the relevant data into one place and preparing it for analysis
data = np.concatenate((X_train, X_test), axis=0)
label = np.concatenate((y_train, y_test), axis=0)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz
17464789/17464789 [==============================] - 0s 0us/step
```

```
print("Review is ",X_train[5])
print("Review is ",y_train[5])
```

```
Review is  [1, 778, 128, 74, 12, 630, 163, 15, 4, 1766, 7982, 1051, 2, 32, 85, 156, 45, 40, 148, 139, 121, 664, 665, 10, 10, 1361, 173,
Review is  0
```

```
vocab=imdb.get_word_index() #The code you provided retrieves the word index for the IMDB dataset
print(vocab)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb_word_index.json
1641221/1641221 [==============================] - 0s 0us/step
{'fawn': 34701, 'tsukino': 52006, 'nunnery': 52007, 'sonja': 16816, 'vani': 63951, 'woods': 1408, 'spiders': 16115, 'hanging': 2345, 'wc
```

```
data #data is a numpy array that contains all the text data from the IMDB dataset, both the training and testing sets.
```

```
array([list([1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 670, 2,
9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 2, 336, 385, 39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147, 2025, 19,
14, 22, 4, 1920, 4613, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38, 76, 15, 13, 1247, 4, 22, 17, 515, 17, 12, 16, 626, 18, 2, 5, 62, 386,
12, 8, 316, 8, 106, 5, 4, 2223, 5244, 16, 480, 66, 3785, 33, 4, 130, 12, 16, 38, 619, 5, 25, 124, 51, 36, 135, 48, 25, 1415, 33, 6, 22,
12, 215, 28, 77, 52, 5, 14, 407, 16, 82, 2, 8, 4, 107, 117, 5952, 15, 256, 4, 2, 7, 3766, 5, 723, 36, 71, 43, 530, 476, 26, 400, 317,
46, 7, 4, 2, 1029, 13, 104, 88, 4, 381, 15, 297, 98, 32, 2071, 56, 26, 141, 6, 194, 7486, 18, 4, 226, 22, 21, 134, 476, 26, 480, 5,
144, 30, 5535, 18, 51, 36, 28, 224, 92, 25, 104, 4, 226, 65, 16, 38, 1334, 88, 12, 16, 283, 5, 16, 4472, 113, 103, 32, 15, 16, 5345,
```

```
    19, 178, 32]),
           list([1, 194, 1153, 194, 8255, 78, 228, 5, 6, 1463, 4369, 5012, 134, 26, 4, 715, 8, 118, 1634, 14, 394, 20, 13, 119, 954, 189,
    102, 5, 207, 110, 3103, 21, 14, 69, 188, 8, 30, 23, 7, 4, 249, 126, 93, 4, 114, 9, 2300, 1523, 5, 647, 4, 116, 9, 35, 8163, 4, 229, 9,
    340, 1322, 4, 118, 9, 4, 130, 4901, 19, 4, 1002, 5, 89, 29, 952, 46, 37, 4, 455, 9, 45, 43, 38, 1543, 1905, 398, 4, 1649, 26, 6853, 5,
    163, 11, 3215, 2, 4, 1153, 9, 194, 775, 7, 8255, 2, 349, 2637, 148, 605, 2, 8003, 15, 123, 125, 68, 2, 6853, 15, 349, 165, 4362, 98, 5,
    4, 228, 9, 43, 2, 1157, 15, 299, 120, 5, 120, 174, 11, 220, 175, 136, 50, 9, 4373, 228, 8255, 5, 2, 656, 245, 2350, 5, 4, 9837, 131,
    152, 491, 18, 2, 32, 7464, 1212, 14, 9, 6, 371, 78, 22, 625, 64, 1382, 9, 8, 168, 145, 23, 4, 1690, 15, 16, 4, 1355, 5, 28, 6, 52, 154,
    462, 33, 89, 78, 285, 16, 145, 95]),
           list([1, 14, 47, 8, 30, 31, 7, 4, 249, 108, 7, 4, 5974, 54, 61, 369, 13, 71, 149, 14, 22, 112, 4, 2401, 311, 12, 16, 3711, 33,
    75, 43, 1829, 296, 4, 86, 320, 35, 534, 19, 263, 4821, 1301, 4, 1873, 33, 89, 78, 12, 66, 16, 4, 360, 7, 4, 58, 316, 334, 11, 4, 1716,
    43, 645, 662, 8, 257, 85, 1200, 42, 1228, 2578, 83, 68, 3912, 15, 36, 165, 1539, 278, 36, 69, 2, 780, 8, 106, 14, 6905, 1338, 18, 6,
    22, 12, 215, 28, 610, 40, 6, 87, 326, 23, 2300, 21, 23, 22, 12, 272, 40, 57, 31, 11, 4, 22, 47, 6, 2307, 51, 9, 170, 23, 595, 116, 595,
    1352, 13, 191, 79, 638, 89, 2, 14, 9, 8, 106, 607, 624, 35, 534, 6, 227, 7, 129, 113]),
           ...,
           list([1, 13, 1408, 15, 8, 135, 14, 9, 35, 32, 46, 394, 20, 62, 30, 5093, 21, 45, 184, 78, 4, 1492, 910, 769, 2290, 2515, 395,
    4257, 5, 1454, 11, 119, 2, 89, 1036, 4, 116, 218, 78, 21, 407, 100, 30, 128, 262, 15, 7, 185, 2280, 284, 1842, 2, 37, 315, 4, 226, 20,
    272, 2942, 40, 29, 152, 60, 181, 8, 30, 50, 553, 362, 80, 119, 12, 21, 846, 5518]),
           list([1, 11, 119, 241, 9, 4, 840, 20, 12, 468, 15, 94, 3684, 562, 791, 39, 4, 86, 107, 8, 97, 14, 31, 33, 4, 2960, 7, 743, 46,
    1028, 9, 3531, 5, 4, 768, 47, 8, 79, 90, 145, 164, 162, 50, 6, 501, 119, 7, 9, 4, 78, 232, 15, 16, 224, 11, 4, 333, 20, 4, 985, 200, 5,
    2, 5, 9, 1861, 8, 79, 357, 4, 20, 47, 220, 57, 206, 139, 11, 12, 5, 55, 117, 212, 13, 1276, 92, 124, 51, 45, 1188, 71, 536, 13, 520,
    14, 20, 6, 2302, 7, 470]),
           list([1, 6, 52, 7465, 430, 22, 9, 220, 2594, 8, 28, 2, 519, 3227, 6, 769, 15, 47, 6, 3482, 4067, 8, 114, 5, 33, 222, 31, 55,
    184, 704, 5586, 2, 19, 346, 3153, 5, 6, 364, 350, 4, 184, 5586, 9, 133, 1810, 11, 5417, 2, 21, 4, 7298, 2, 570, 50, 2005, 2643, 9, 6,
    1249, 17, 6, 2, 2, 21, 17, 6, 1211, 232, 1138, 2249, 29, 266, 56, 96, 346, 194, 308, 9, 194, 21, 29, 218, 1078, 19, 4, 78, 173, 7, 27,
    2, 5698, 3406, 718, 2, 9, 6, 6907, 17, 210, 5, 3281, 5677, 47, 77, 395, 14, 172, 173, 18, 2740, 2931, 4517, 82, 127, 27, 173, 11, 6,
    392, 217, 21, 50, 9, 57, 65, 12, 2, 53, 40, 35, 390, 7, 11, 4, 3567, 7, 4, 314, 74, 6, 792, 22, 2, 19, 714, 727, 5205, 382, 4, 91,
    6533, 439, 19, 14, 20, 9, 1441, 5805, 1118, 4, 756, 25, 124, 4, 31, 12, 16, 93, 804, 34, 2005, 2643])],
          dtype=object)
```

```
label
X_train.shape
X_test.shape
y_train
y_test
```

```
array([0, 1, 1, ..., 0, 0, 0])
```

```
# Function to perform relevant sequence adding on the data
# Now it is time to prepare our data. We will vectorize every review and fill it with zeros so that it contains exactly 1000 numbers.
# That means we fill every review that is shorter than 500 with zeros.
# We do this because the biggest review is nearly that long and every input for our neural network needs to have the same size.
# We also transform the targets into floats.
# sequences is name of method the review less than 1000 we perform padding overthere

def vectorize(sequences, dimension = 10000):
  # Create an all-zero matrix of shape (len(sequences), dimension)
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1
    return results
```

```
test_x = data[:10000]
test_y = label[:10000]
train_x = data[10000:]
train_y = label[10000:]
```

```
test_x
test_y
train_x
train_y
```

```
array([0, 0, 0, ..., 0, 0, 0])
```

```
print("Categories:", np.unique(label))
print("Number of unique words:", len(np.unique(np.hstack(data))))
# The hstack() function is used to stack arrays in sequence horizontally (column wise).
```

```
Categories: [0 1]
Number of unique words: 9998
```

```
print("Label:", label[0])
print(data[0])
```

```
Label: 1
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 670, 2, 9, 35, 480, 28
```

```
#Adding sequence to data
# Vectorization is the process of converting textual data into numerical vectors and is a process that is usually applied once the text is cl
data = vectorize(data)
label = np.array(label).astype("float32")
```

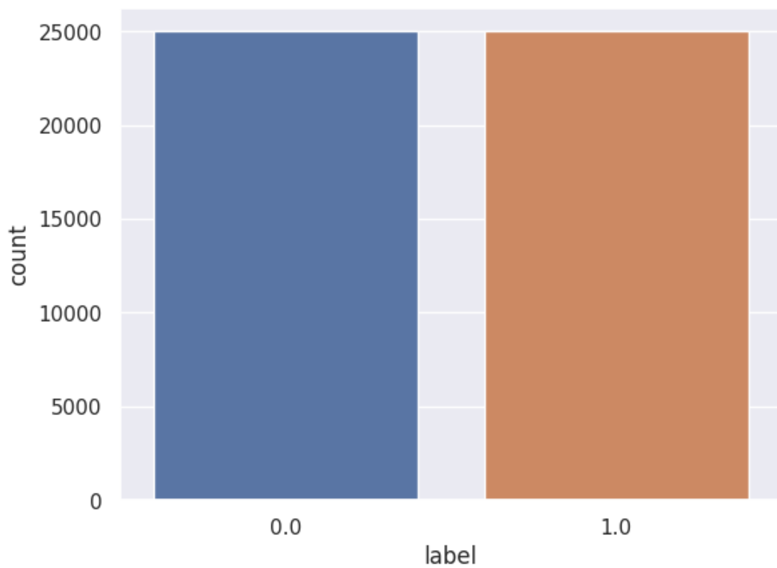
```
data
label
```

```
array([1., 0., 0., ..., 0., 0., 0.], dtype=float32)
```

```
# Let's check distribution of data

# To create plots for EDA(exploratory data analysis)
import seaborn as sns #seaborn is a popular Python visualization library that is built on top of Matplotlib and provides a high-level interfa
sns.set(color_codes=True)
import matplotlib.pyplot as plt # %matplotlib to display Matplotlib plots inline with the notebook
%matplotlib inline
```

```
labelDF=pd.DataFrame({'label':label})
sns.countplot(x='label', data=labelDF)

# For below analysis it is clear that data has equel distribution of sentiments.This will help us building a good model.
```

```
<Axes: xlabel='label', ylabel='count'>
```




## MNIST Dataset

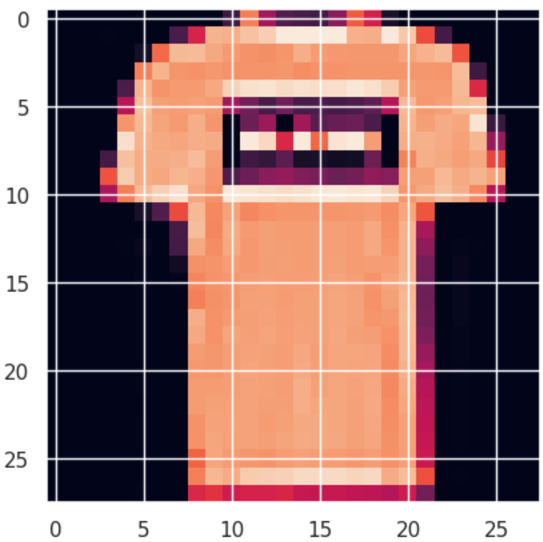
```
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow import keras
import numpy as np

(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()

# There are 10 image classes in this dataset and each class has a mapping corresponding to the following labels:

#0 T-shirt/top
#1 Trouser
#2 pullover
#3 Dress
#4 Coat
#5 sandals
#6 shirt
#7 sneaker
#8 bag
#9 ankle boot
```

```
# https://ml-course.github.io/master/09%20-%20Convolutional%20Neural%20Networks.pdf
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 [==============================] - 0s 0us/step
```

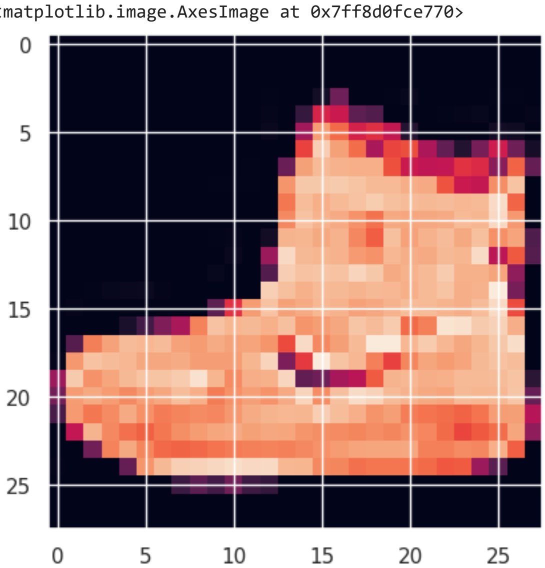
```
plt.imshow(x_train[1])
```

```
<matplotlib.image.AxesImage at 0x7ff8d101d510>
```




```
plt.imshow(x_train[0])
```

```
<matplotlib.image.AxesImage at 0x7ff8d0fce770>
```




```
# Next, we will preprocess the data by scaling the pixel values to be between 0 and 1, and then reshaping the images to be 28x28 pixels.

x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)

# 28, 28 comes from width, height, 1 comes from the number of channels
# -1 means that the length in that dimension is inferred.
# This is done based on the constraint that the number of elements in an ndarray or Tensor when reshaped must remain the same.
```

```
# each image is a row vector (784 elements) and there are lots of such rows (let it be n, so there are 784n elements). So TensorFlow can infe
```

```
x_train.shape
```

```
(60000, 28, 28, 1)
```

```
x_test.shape
```

```
(10000, 28, 28, 1)
```

```
y_train.shape
```

```
(60000,)
```

```
y_test.shape
```

```
(10000,)
```

```
model = keras.Sequential([
    keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
    # 32 filters (default), randomly initialized
    # 3*3 is Size of Filter
    # 28,28,1 size of Input Image
    # No zero-padding: every output 2 pixels less in every dimension
    # in Paramter shwon 320 is value of weights: (3x3 filter weights + 32 bias) * 32 filters
    # 32*3*3=288(Total)+32(bias)= 320


    keras.layers.MaxPooling2D((2,2)),
    # It shown 13 * 13 size image with 32 channel or filter or depth.

    keras.layers.Dropout(0.25),
    # Reduce Overfitting of Training sample drop out 25% Neuron

    keras.layers.Conv2D(64, (3,3), activation='relu'),

    keras.layers.MaxPooling2D((2,2)),
    # It shown 5 * 5 size image with 64 channel or filter or depth.

    keras.layers.Dropout(0.25),

    keras.layers.Conv2D(128, (3,3), activation='relu'),

    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    # 128 Size of Node in Dense Layer
    # 1152*128 = 147584

    keras.layers.Dropout(0.25),
    keras.layers.Dense(10, activation='softmax')
    # 10 Size of Node another Dense Layer
    # 128*10+10 bias= 1290
])
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 dropout (Dropout)           (None, 13, 13, 32)        0

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 dropout_1 (Dropout)         (None, 5, 5, 64)          0
```

```
 conv2d_2 (Conv2D)           (None, 3, 3, 128)         73856

 flatten (Flatten)           (None, 1152)              0

 dense (Dense)               (None, 128)               147584

 dropout_2 (Dropout)         (None, 128)               0

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 241,546
Trainable params: 241,546
Non-trainable params: 0
_________________________________________________________________
```

```
# Compile and Train the Model
# After defining the model, we will compile it and train it on the training data.

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

history = model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))

# 1875 is a number of batches. By default batches contain 32 samles.60000 / 32 = 1875
```

```
Epoch 1/10
1875/1875 [==============================] - 89s 46ms/step - loss: 0.5657 - accuracy: 0.7909 - val_loss: 0.3743 - val_accuracy: 0.8643
Epoch 2/10
1875/1875 [==============================] - 88s 47ms/step - loss: 0.3743 - accuracy: 0.8634 - val_loss: 0.3230 - val_accuracy: 0.8830
Epoch 3/10
1875/1875 [==============================] - 82s 44ms/step - loss: 0.3273 - accuracy: 0.8797 - val_loss: 0.3025 - val_accuracy: 0.8895
Epoch 4/10
1875/1875 [==============================] - 84s 45ms/step - loss: 0.2994 - accuracy: 0.8903 - val_loss: 0.2785 - val_accuracy: 0.8978
Epoch 5/10
1875/1875 [==============================] - 83s 44ms/step - loss: 0.2818 - accuracy: 0.8961 - val_loss: 0.2743 - val_accuracy: 0.8981
Epoch 6/10
1875/1875 [==============================] - 82s 44ms/step - loss: 0.2658 - accuracy: 0.9021 - val_loss: 0.2643 - val_accuracy: 0.8994
Epoch 7/10
1875/1875 [==============================] - 83s 44ms/step - loss: 0.2573 - accuracy: 0.9042 - val_loss: 0.2861 - val_accuracy: 0.8881
Epoch 8/10
1875/1875 [==============================] - 82s 44ms/step - loss: 0.2469 - accuracy: 0.9077 - val_loss: 0.2677 - val_accuracy: 0.9009
Epoch 9/10
1875/1875 [==============================] - 83s 44ms/step - loss: 0.2362 - accuracy: 0.9118 - val_loss: 0.2576 - val_accuracy: 0.9031
Epoch 10/10
1875/1875 [==============================] - 83s 44ms/step - loss: 0.2309 - accuracy: 0.9134 - val_loss: 0.2604 - val_accuracy: 0.9068
```

```
# Finally, we will evaluate the performance of the model on the test data.

test_loss, test_acc = model.evaluate(x_test, y_test)

print('Test accuracy:', test_acc)
```

```
313/313 [==============================] - 3s 11ms/step - loss: 0.2604 - accuracy: 0.9068
Test accuracy: 0.9067999720573425
```