Life-Expectancy Prediction

Phase-2

By Akshay Hari
(AM.EN.U4CSE19104)

1. Problem Description

Life expectancy has become one of the most frequently used health status indicators. It is a measure that **summarizes the mortality of a country**, allowing us to compare it by generation and analyze trends.

We aim to train different models that are capable of learning the important features which lead to life expectancy rates. The model chosen is the one that can predict the life expectancy rates with the highest accuracy.

2. Dataset Description

Dataset consists of the average **life expectancy** rates of 15 different countries over 217 years (1802-2016). It contains 3 features, which include the **Country Name**, **Year,** and the corresponding **Life Expectancy Rates.**

Countries Life Expectancy (Kaggle)	Link

3. Data Preparation

The dataset consists of several countries which are encoded in string format. These were replaced with corresponding label values using the **Label Encoding** method.

```
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
df['labels'] = label_encoder.fit_transform(df.loc[:, 'Entity'])
```

	Entity	Year	Life expectancy	labels
0	Australia	1802	34.049999	0
1	Australia	1803	34.049999	0
2	Australia	1804	34.049999	0
3	Australia	1805	34.049999	0
4	Australia	1806	34.049999	0

	Entity	Year	Life expectancy	labels
1300	India	1800	25.442400	6
1301	India	1801	25.442400	6
1302	India	1802	25.000000	6
1303	India	1803	24.000000	6
1304	India	1804	23.500000	6

Some rows within the dataset contains **NaN** parameters, these had to be avoided using the **dropna()** method which removes all rows with NaN parameter values.

```
df = df.dropna(axis=0)
```

Some features within the dataset have vastly varying values, therefore those values had to be standardized using the StandardScale method.

```
from sklearn import preprocessing, linear_model,
model_selection, metrics

scaler = preprocessing.StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
array([[ 0.75519552,  1.60575071],
       [-1.4767477 , -0.25752521],
       [-1.06224396, -0.4904347 ],
       ...,
       [ 1.64797281, -0.4904347 ],
       [ 1.64797281, -0.95625368],
       [ 0.48417385,  1.60575071]])
```

4. Summarization

Dataset can be summarized into three columns containing the **Entity, Years,** and the corresponding **Life Expectancy** rate. The entire data consists of about 3253 entries which are non-null. The feature '**Entity'** follows a string data type, '**Year'** follows integer format, and '**Life Expectancy'** follows the floating-point format.

Dataset dimensions consist of 3253 rows and 3 columns. (3253 x 3)

```
df.info()
```

```
RangeIndex: 3253 entries, 0 to 3252
Data columns (total 3 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   Entity           3253 non-null   object 
 1   Year             3253 non-null   int64  
 2   Life expectancy  3253 non-null   float64
```

The describe() method is used for calculating some statistical data like **percentile, mean**, and **std** of the numerical values of the dataset. The statistical summary indicates that the dataset consists of 15 different unique countries.

```
df.describe(include='all')
```

	Entity	Year	Life expectancy
count	3253	3253.000000	3253.000000
unique	15	NaN	NaN
top	Canada	NaN	NaN
freq	217	NaN	NaN
mean	NaN	1908.066093	48.680380
std	NaN	62.613962	17.965669
min	NaN	1800.000000	8.108836
25%	NaN	1854.000000	32.000000
50%	NaN	1908.000000	41.880001
75%	NaN	1962.000000	66.820000
max	NaN	2016.000000	83.940002

- **Breaking down each of the column class variables** :

```
df['Entity']
```

```
0           Australia
1           Australia
2           Australia
3           Australia
4           Australia
              ...
3248    United States
3249    United States
3250    United States
3251    United States
3252    United States
Name: Entity, Length: 3253, dtype: object
```

```
df['Year']
```

```
0       1802
1       1803
2       1804
3       1805
4       1806
        ...
3248    2012
3249    2013
3250    2014
3251    2015
3252    2016
Name: Year, Length: 3253, dtype: int64
```

```
df['Life expectancy']
```

```
0       34.049999
1       34.049999
2       34.049999
3       34.049999
4       34.049999
          ...
3248    78.940002
3249    78.959999
3250    78.940002
3251    78.870003
3252    78.860001
Name: Life expectancy, Length: 3253, dtype: float64
```

5. Data Visualization

```
df['Year'].unique()
```

```
array([1802, 1803, 1804, 1805, 1806, 1807, 1808, 1809, 1810, 1811, 1812,
       1813, 1814, 1815, 1816, 1817, 1818, 1819, 1820, 1821, 1822, 1823,
       1824, 1825, 1826, 1827, 1828, 1829, 1830, 1831, 1832, 1833, 1834,
       1835, 1836, 1837, 1838, 1839, 1840, 1841, 1842, 1843, 1844, 1845,
       1846, 1847, 1848, 1849, 1850, 1851, 1852, 1853, 1854, 1855, 1856,
       1857, 1858, 1859, 1860, 1861, 1862, 1863, 1864, 1865, 1866, 1867,
       1868, 1869, 1870, 1871, 1872, 1873, 1874, 1875, 1876, 1877, 1878,
       1879, 1880, 1881, 1882, 1883, 1884, 1885, 1886, 1887, 1888, 1889,
       1890, 1891, 1892, 1893, 1894, 1895, 1896, 1897, 1898, 1899, 1900,
       1901, 1902, 1903, 1904, 1905, 1906, 1907, 1908, 1909, 1910, 1911,
       1912, 1913, 1914, 1915, 1916, 1917, 1918, 1919, 1920, 1921, 1922,
       1923, 1924, 1925, 1926, 1927, 1928, 1929, 1930, 1931, 1932, 1933,
       1934, 1935, 1936, 1937, 1938, 1939, 1940, 1941, 1942, 1943, 1944,
       1945, 1946, 1947, 1948, 1949, 1950, 1951, 1952, 1953, 1954, 1955,
       1956, 1957, 1958, 1959, 1960, 1961, 1962, 1963, 1964, 1965, 1966,
       1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977,
       1978, 1979, 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988,
       1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999,
       2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010,
       2011, 2012, 2013, 2014, 2015, 2016, 1800, 1801])
```

- Dataset consists of data between the years 1800 and 2016.

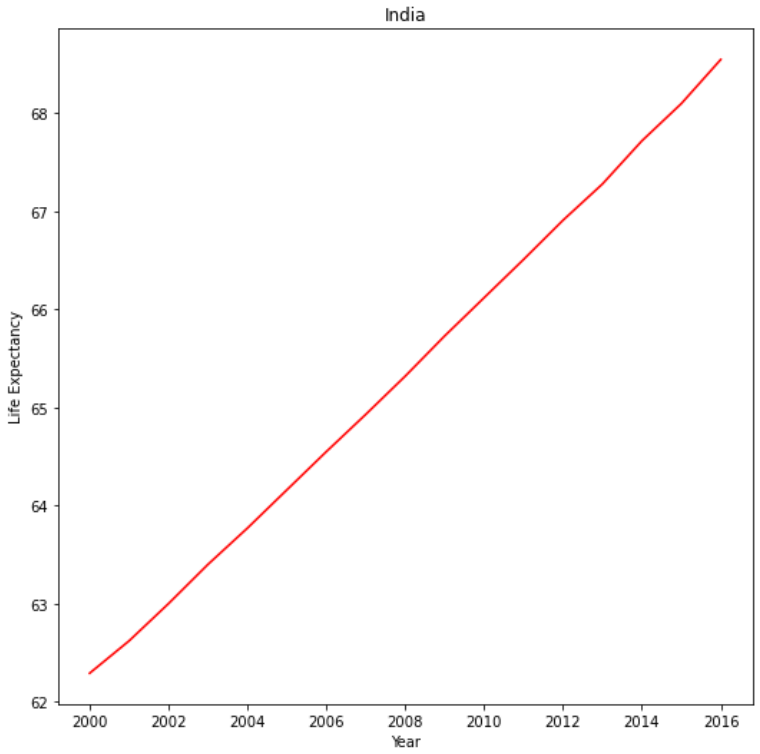
```
df['Entity'].unique()
```

```
array(['Australia', 'Brazil', 'Canada', 'China', 'France', 'Germany',
       'India', 'Italy', 'Japan', 'Mexico', 'Russia', 'Spain',
       'Switzerland', 'United Kingdom', 'United States'], dtype=object)
```

- Also, there are 15 different countries in the dataset.

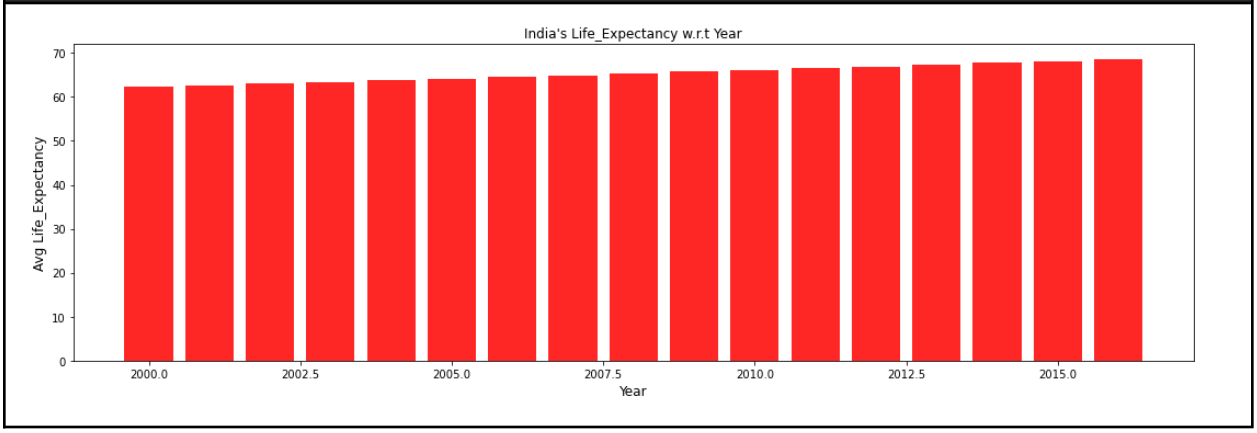
❖ **Graph Depicting Linear Increase Of Life Expectancy Rates in India**

```
plt.plot(yrs,lex, color='red')
plt.xlabel('Year')
plt.ylabel('Life Expectancy')
plt.title("India")
plt.show()
```



- We can observe that there is a **Linear Increase** Of Life Expectancy rates over the past few years.

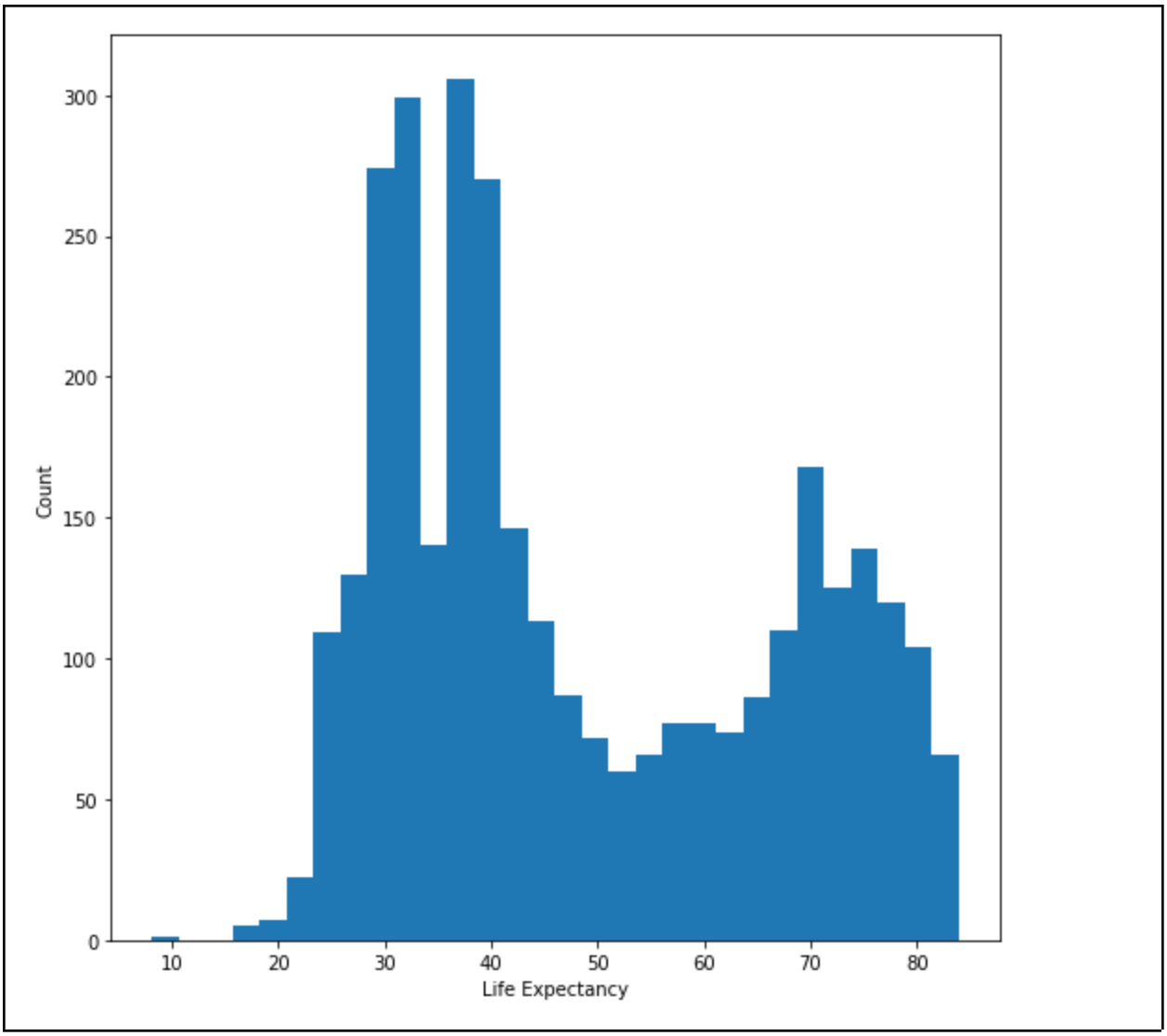
```
plt.figure(figsize=(15,5))
plt.bar(yrs,lex,color='red',alpha=0.85)
plt.xlabel("Year",fontsize=12)
plt.ylabel("Avg Life_Expectancy",fontsize=12)
plt.title("India's Life_Expectancy w.r.t Year")
plt.show()
```



- India's Life Expectancy rates have improved over the years 2000-2016, with an average rate of 60.

❖ **Analyzing The Data**

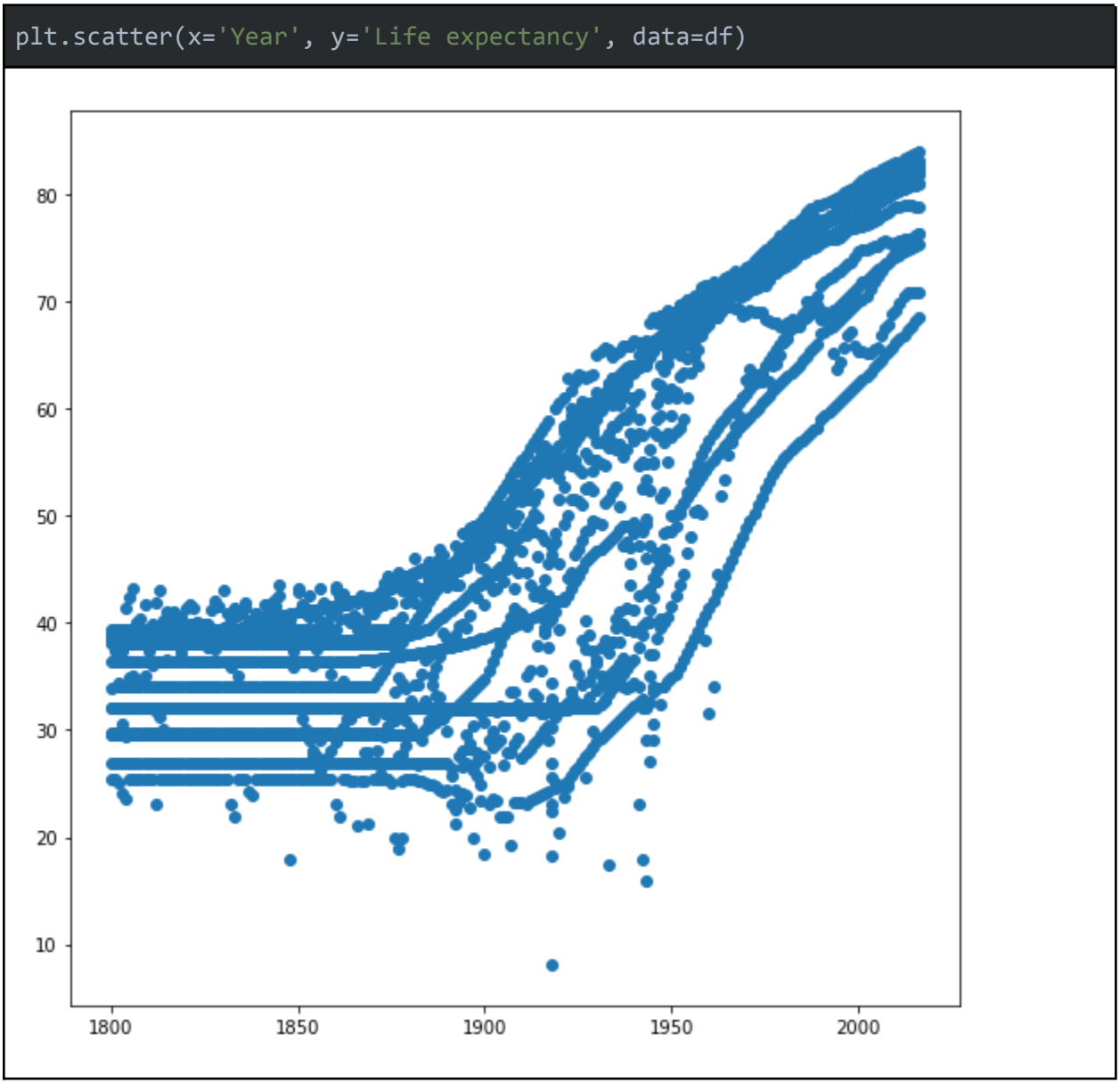
```
plt.hist(df['Life expectancy'], density=False, bins=30)
plt.ylabel('Count')
plt.xlabel('Life Expectancy');
```



- It is evident from the histogram that the dataset follows a right-skewed distribution. Even though the median score is almost 42 years, distribution density increases at around 50 years. There are quite a lot of life expectancy rates bigger than the median value of life expectancy.

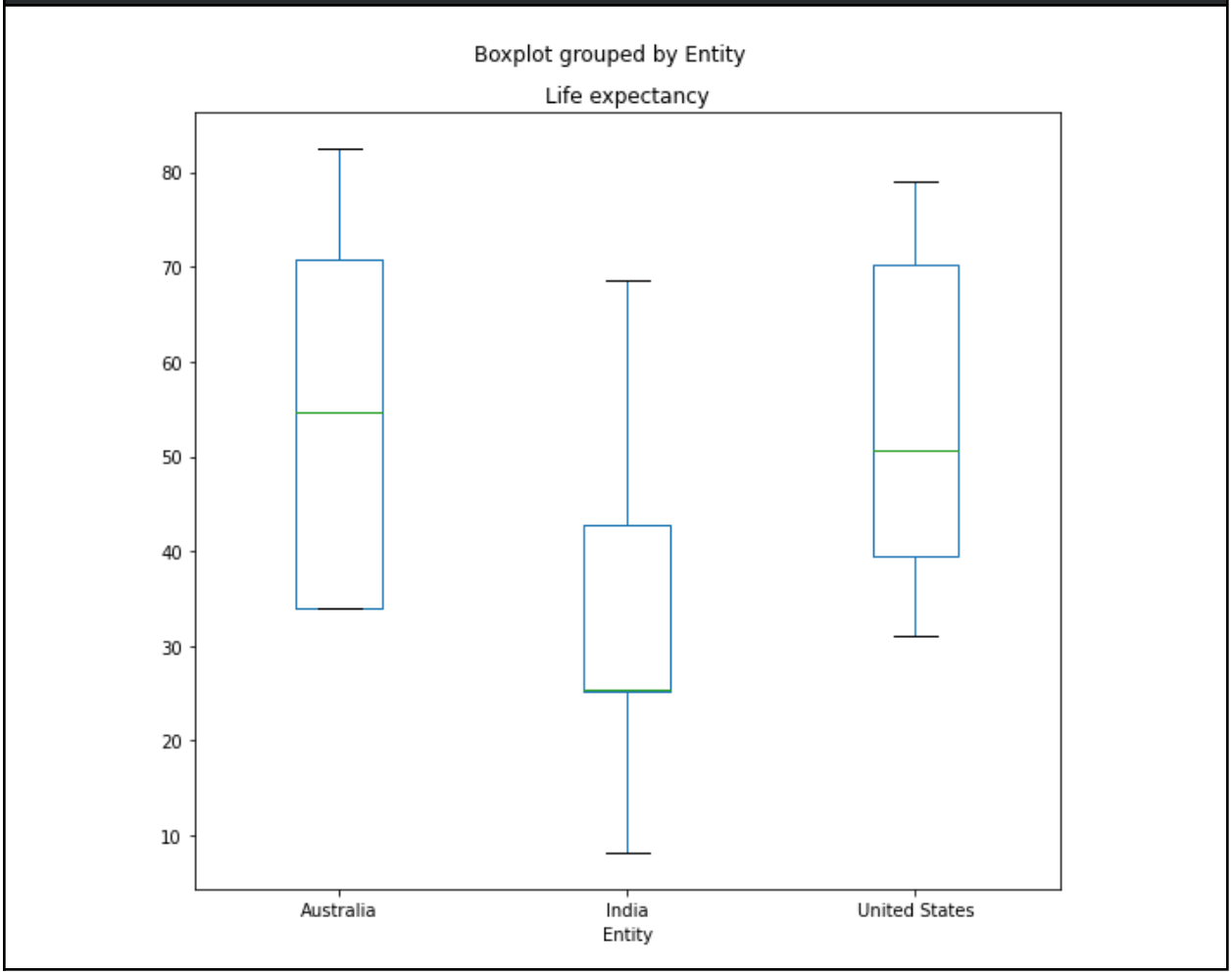
❖ **Scatterplot - Life Expectancy Of Countries corresponding to the years**

```
plt.scatter(x='Year', y='Life expectancy', data=df)
```



❖ **Box Plot Depicting Life Expectancy Rates of 3 countries**

```
df[df.Entity.isin(['Australia','India','United
States'])].boxplot(by='Entity',
                   column=['Life expectancy'],
                   grid=False)
```



- The boxplot representation of Australia depicts that the average life expectancy rates of the country were evenly spread out over the years, with the maximum at the range of 80 - 83 and the minimum at the range of 34-36.

- The boxplot representation of India depicts that the average life expectancy rates of the country fall in the lower end of the IQR, with the maximum at the range of 67 - 69 and the minimum at the range of 8-10.
- The boxplot representation of the United States depicts that the average life expectancy rates of the country fall in the upper end of the IQR, with the maximum at the range of 77-79 and the minimum at the range of 30-33.

6. Python Packages

- **Numpy**: NumPy is a Python library used for working with arrays.
- **Pandas**: Pandas is an open-source Python package that is most widely used for data science/data analysis and machine learning tasks. It is used for cleaning, transforming, manipulating, and analysis of data.
- **Matplotlib**: Matplotlib is a cross-platform, data visualization and graphical plotting library for Python and its numerical extension NumPy.
- **Sklearn**: Scikit-learn (Sklearn) is the most useful and robust library for machine learning in Python. It provides a selection of efficient tools for machine learning and statistical modeling including classification, regression, clustering, and dimensionality reduction via a consistent interface in Python.

7. Learning Algorithms

- **K-Neighbor-Regressor (KNR)**: The KNR algorithm finds the K closest neighbors of a given test data point by calculating the **Euclidean Distance Measure** with all the other training data samples. In the case of regression, the mean value of the labels containing the indices of K closest distance samples is used to predict the Life Expectancy Rates.

```
class KNeighbourRegressor():

      def __init__(self,k):
      self.k = k

      def fit(self,X,y):

      self.X_train = X
      self.y_train = y


      def predictions(self,x):

      distances = [euclid_distance(x,x_train) for x_train in
self.X_train]

      k_indices = np.argsort(distances)[:self.k]
      k_nearest_labels = [self.y_train[i] for i in k_indices]

      y_pred = np.mean(k_nearest_labels)

      return y_pred


      def predict(self,X):

      predicted_labels = [self.predictions(x) for x in X]

      return np.array(predicted_labels)
```

Function To Calculate Euclidean Distances

```
def euclid_distance(a,b):

      dist = (a-b)**2
      return np.sqrt(np.sum(dist))
```

❖ **K-Fold Cross Validation For KNR - Algorithm :**

Implementation Of K-Fold Cross-Validation

```
def kfold(x,y,model):

      model_accuracies = []

      kf = KFold(n_splits = 5)

      for train_index, test_index in kf.split(x):

      X_train, X_valid = x[train_index], x[test_index]
      Y_train, Y_valid = y[train_index], y[test_index]

      model.fit(X_train, Y_train)
      Y_pred = model.predict(X_valid)

      score = metrics.r2_score(Y_valid, Y_pred)
      model_accuracies += [[model, score]]

      return sorted(model_accuracies, key=lambda a: a[1])[-1]
```

```
KNN = KNeighbourRegressor(5)
best_model, best_out = kfold(X_train,y_train,KNN)
print(best_model,best_out)
```

```
<__main__.KNeighbourRegressor object at 0x7fa160b663a0> 0.9949230858291616
```

The highest cross-validation accuracy for K-Neighbor-Regressor was calculated using the **r2_score()** function. The best model among the splits gave an accuracy of 99.4% for the validation set.

- **Linear Regression (LR)**: The Linear Regression algorithm attempts to model the relationship between the two variables by fitting a linear equation to the observed data. Here we aim to find a best-fit-line (y = mx+b) that can fit the data by minimizing the cost function. To obtain the minimum loss function, optimum values of the slope and y-intercept are required. This can be achieved by using a gradient descent optimization method, which obtains the values by finding the gradients of the weights and bias and updating it accordingly in such a way that the cost function is always minimum.

Implementation Of Linear Regression

```
class LinearRegression:

     def __init__(self,X,Y):

     self.X_train = X
     self.y_train = Y

     def gradient_descent(self,x, y, iter = 100, lr = 0.08):

     m, n, b = np.zeros(x.shape[1]), (-2*lr)/x.shape[0], 0

     for _ in range(iter):
          yd = [j-(sum(m*i)+b) for i, j in zip(x, y)]
          m -= n*sum([i*j for i,j in zip(x, yd)])
          b -= n*sum(yd)

     return m, b

     def fit(self,x, y):

     coef,intercept = self.gradient_descent(x, y)
     return coef,intercept
```

```
    def predict(self,x):

    m,b = self.fit(self.X_train,self.y_train)
    return np.dot(x,m) + b
```

❖ **K-Fold Cross Validation For LR - Algorithm :**

K-Fold Cross Validation For Linear Regression

```
LR = LinearRegression(X_train,y_train)
best_model, best_out = kfold(X_train,y_train,LR)
print(best_model,best_out)
```

```
LR = LinearRegression(X_train,y_train)
best_model, best_out = kfold(X_train,y_train,LR)
print(best_model,best_out)
```

```
<__main__.LinearRegression object at 0x7fa160884af0> 0.7378931401616659
```

- **Support Vector Regression (SVR)**: In Support Vector Regression, the main objective is to find an optimal hyperplane that can distinctly classify the data points with minimum error. It tries to fit the best line within a threshold value, which is the distance between the hyperplane and boundary line. We form a decision boundary within the threshold value from the original hyperplane, such that data points closest to the hyperplane (**support vectors**) are within that boundary line. We take all those points within the decision boundary having the least error rate, and try to fit our model.

Implementation Of Support Vector Regression

```
from sklearn.svm import SVR
svr = SVR()
```

❖ **K-Fold Cross Validation For SVR - Algorithm :**

K-Fold Cross Validation For Support Vector Regression

```
best_model, best_out = kfold(X_train,y_train,svr)
print(best_model,best_out)
```

```
best_model, best_out = kfold(X_train,y_train,svr)
print(best_model,best_out)
```

```
SVR() 0.8339060166361152
```

8. Evaluating The Models Using Metrics

Function for calculating Mean Absolute Error

```
def MAE(Y_actual,Y_Predicted):
      mae = np.mean(np.abs((Y_actual -
Y_Predicted)/Y_actual))*100
      return mae
```

❖ **K-Neighbors-Regressor :**

Testing the accuracy for KNR Algorithm

```
y_pred_KNN = KNN.predict(X_test)
metrics.r2_score(y_test, y_pred_KNN)
```

```
metrics.r2_score(y_test, y_pred_KNN)
```

```
0.9916390572524313
```

```
KNN_MAE = MAE(y_test,y_pred_KNN)
Accuracy_KNN = 100 - KNN_MAE
print("Mean Absolute Error: ",KNN_MAE)
print('Accuracy of KNN model: {:0.2f}%.'.format(Accuracy_KNN))
```

```
Mean Absolute Error:  1.7614748468969457
Accuracy of KNN model: 98.24%.
```

KNR Algorithm gives an r2 score of 0.99 and a Mean Absolute Error of 1.76 with an accuracy of 98.34% for the test dataset (X_test).

- **Linear Regression (LR)**:

Testing the accuracy for Linear Regression

```
y_pred_LR = LR.predict(X_test)
metrics.r2_score(y_test, y_pred_LR)
```

```
metrics.r2_score(y_test, y_pred_LR)
```

```
0.699620078031356
```

```
LR_MAE = MAE(y_test,y_pred_LR)
Accuracy_LR = 100 - LR_MAE
print("MAE: ",LR_MAE)
print('Accuracy of LR model: {:0.2f}%.'.format(Accuracy_LR))
```

```
MAE:  20.417826290376308
Accuracy of LR model: 79.58%.
```

Linear Regression gives an r2 score of 0.69 and a Mean Absolute Error of 20.41 with an accuracy of 79.58% for the test dataset (X_test).

- **Support Vector Regression (SVR)**:

Testing the accuracy for Support Vector Regression (SVR)

```
y_pred_SVR = svr.predict(X_test)
metrics.r2_score(y_test, y_pred_SVR)
```

```
metrics.r2_score(y_test, y_pred_SVR)
```

```
0.8018846132020336
```

```
SVR_MAE = MAE(y_test,y_pred_SVR)
Accuracy_SVR = 100 - SVR_MAE
print("MAE: ",SVR_MAE)
print('Accuracy of SVR model: {:0.2f}%.'.format(Accuracy_SVR))
```

```
MAE:  13.853596851533279
Accuracy of SVR model: 86.15%.
```

Support Vector Regression gives an r2 score of 0.80 and a Mean Absolute Error of 13.85 with an accuracy of 86.15% for the test dataset (X_test).

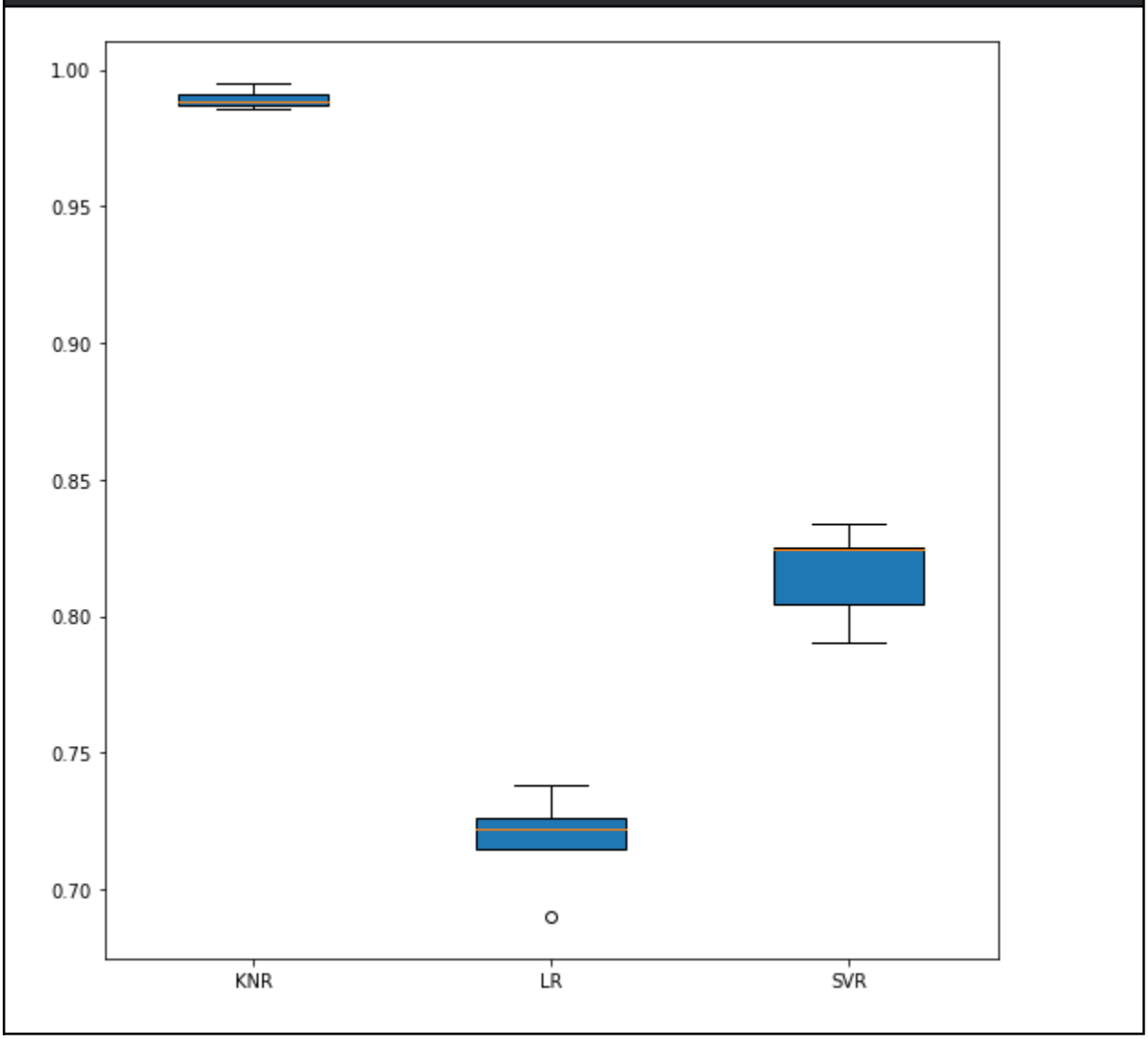
Conclusion :

From the three different models, it can be inferred that the KNN model gives the highest accuracy and performs much better when compared to the other models.

❖ **Boxplot Representing Accuracy Rates**

```
plt.rcParams["figure.figsize"] = [7.50, 7.50]
plt.rcParams["figure.autolayout"] = True

box_plot_data=[box_KNN,box_LR,box_SVR]
plt.boxplot(box_plot_data,patch_artist=True,labels=['KNR','LR','SVR']
,widths=(0.5, 0.5, 0.5))
plt.show()
```



- The accuracy rates of KNR as represented by the boxplot suggest that most of the validation scores are clustered near to the median within the range 0.98 - 0.99. It also depicts that the maximum value lies in the range 0.99 - 0.995 and the minimum value lies in the range 0.98 - 0.99.
- The accuracy rates of LR as represented by the boxplot suggest that most of the validation scores are spread out within the range 0.71 - 0.74. It also depicts that the maximum value lies in the range 0.72 - 0.73. However, it is observed that the minimum value is an outlier that lies outside the IQR with an accuracy rate of 0.69.
- The accuracy rates of SVR as represented by the boxplot suggest that most of the validation scores are spread out within the range 0.79 - 0.83 with most of the values falling in the lower end of IQR. It also depicts that the maximum value lies in the range 0.82 - 0.83 and the minimum value lies in the range 0.79-0.8.

❖ **KNR Model Accuracy with Varying Parameters**

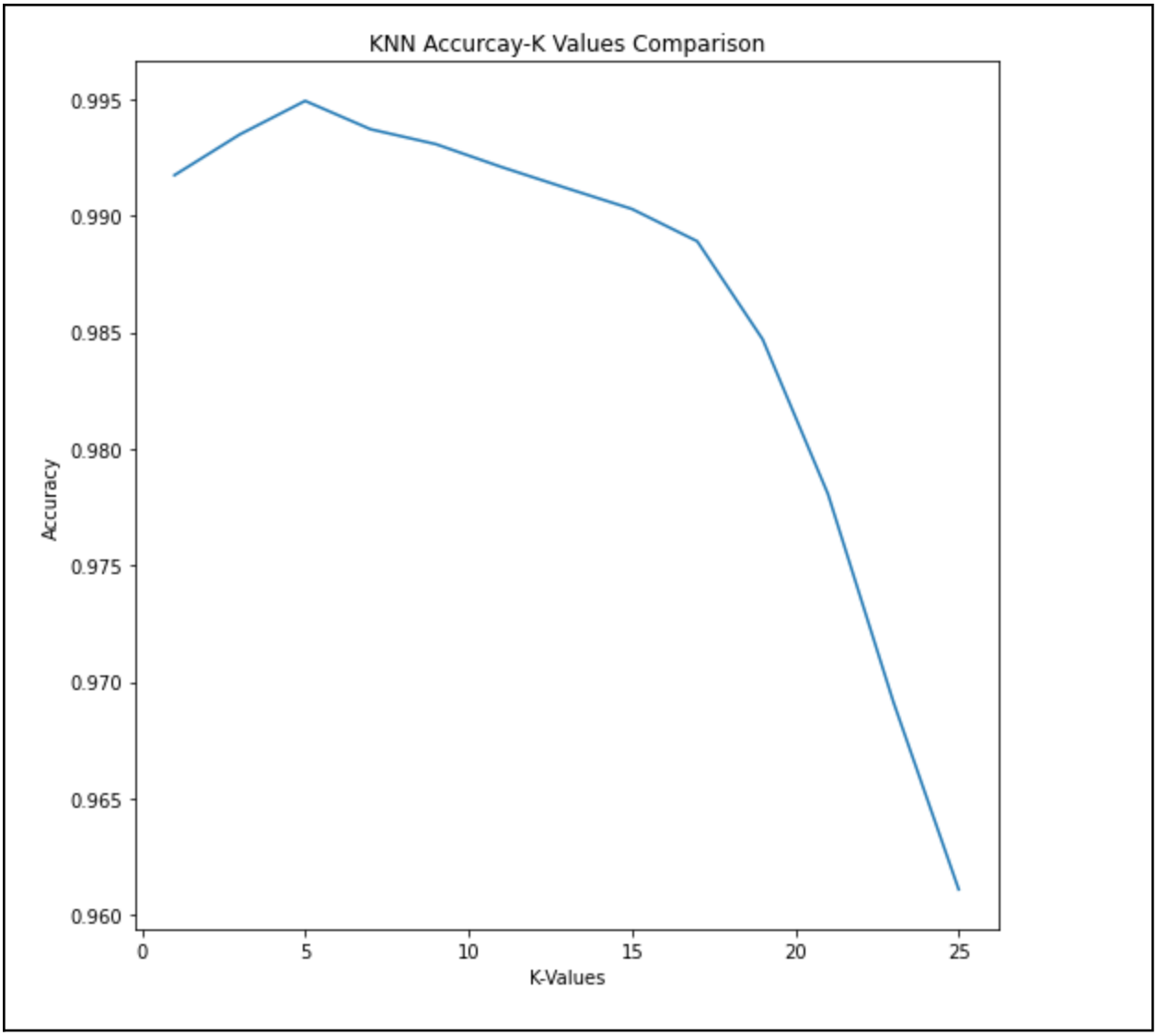
```
acc_score = []
k_values = []

for k in range(1,26,2):

     KNN = KNeighbourRegressor(k)
     _,best_out = kfold(X_train,y_train,KNN)
     acc_score.append(best_out)
     k_values.append(k)
```

```
plt.plot(k_values,acc_score)

plt.xlabel('K-Values')
plt.ylabel('Accuracy')
plt.title('KNN Accurcay-K Values Comparison')
plt.show()
```



It can be inferred from the graph that, as the k-value increases, the accuracy decreases. The parameter value i.e **k = 5**, gives the highest accuracy among all the other k-values.