Projecto 02 Team 8

November 17, 2021

1 CASE STUDY: Happiness

1.1 Team 8 Members

- Alejandro Bañuelos A01244596
- Eduardo López Goerne Salgado A00570852

1. This practice is a guided case study in Google Colab. We'll review the concepts of unsupervised and supervised learning.

2. Attend the teacher's instructions for the development of the practice.

3. Data set for the case study is: hapiness.csv Download hapiness.csv

4. PART ONE

   4.1 Load the file

   4.2 Turn categorical columns into ordinal columns

   4.3 Standardize the data. Justify the method used.

   4.4 Use elbow method to define k clusters

   4.5 Apply k-means analysis (used the k from elbow method)

   4.6 Give your conclusions about the number of clusters formed (features, similarities, etc)

5. PART TWO Classification analysis

   5.1 Target Y (ClassHapiness) X (Rest variables)- Obtain training and testing set (80-20)

   5.2 Perform LogisticRegression, KNN, and Decision Tree

   5.3 Perform confusion matrix for all results

   5.4 Select one method to give an interpretation of the confusion matrix

6. PART THREE Regression Analysis

   6.1 Target Y (Happiness Score) X (Rest variables)- Obtain training and testing set (80-20)

   6.2 Perform MultipleLinearRegression, DecisionTreeRegression, SupporVectorRegression

   6.3 Obtain R2, Mean Squared Error, Root Mean Squared Error, Mean Absolute Error for all methods

   6.4 Select one method to give an interpretation of the resul metrics

7. Deliver a notebook jupyter code with the conclusions included.

```
[ ]: import pandas as pd
     from sklearn.preprocessing import StandardScaler
     import seaborn as sns
     import numpy as np
     from sklearn.preprocessing import LabelEncoder
     from sklearn.preprocessing import MinMaxScaler
     import matplotlib.pyplot as plt
     from sklearn.decomposition import PCA
     import tensorflow as tf
```

```
[ ]: ## Graph fonts ##
     fontT = {'family' : 'Arial',
             'weight' : 'bold',
             'size'   : 20}

     fontL = {'family' : 'Arial',
             'weight' : 'bold',
             'size'   : 18}

     fontAx = {'family' : 'Arial',
             'weight' : 'normal',
             'size'   : 10}

     fontTix = {'family' : 'Arial',
             'weight' : 'normal',
             'size'   : 6}
```

1.2 Part One Reading and Scaling the Data

```
[ ]: df = pd.read_csv("hapiness.csv")
     df = df.set_index('Country')
     df.describe()
```

```
[ ]:        Happiness Rank  Happiness Score  Standard Error  \
     count      158.000000       158.000000      158.000000
     mean        79.493671         5.375734        0.047885
     std         45.754363         1.145010        0.017146
     min          1.000000         2.839000        0.018480
     25%         40.250000         4.526000        0.037268
     50%         79.500000         5.232500        0.043940
     75%        118.750000         6.243750        0.052300
     max        158.000000         7.587000        0.136930

            Economy (GDP per Capita)      Family  Health (Life Expectancy)  \
     count                158.000000  158.000000                158.000000
```

```
mean                   0.846137    0.991046                  0.630259
std                    0.403121    0.272369                  0.247078
min                    0.000000    0.000000                  0.000000
25%                    0.545808    0.856823                  0.439185
50%                    0.910245    1.029510                  0.696705
75%                    1.158448    1.214405                  0.811013
max                    1.690420    1.402230                  1.025250

          Freedom  Trust (Government Corruption)  Generosity  \
count  158.000000                     158.000000  158.000000
mean     0.428615                       0.143422    0.237296
std      0.150693                       0.120034    0.126685
min      0.000000                       0.000000    0.000000
25%      0.328330                       0.061675    0.150553
50%      0.435515                       0.107220    0.216130
75%      0.549092                       0.180255    0.309882
max      0.669730                       0.551910    0.795880

       Dystopia Residual  ClassHapiness
count         158.000000     158.000000
mean            2.098977       0.278481
std             0.553550       0.449677
min             0.328580       0.000000
25%             1.759410       0.000000
50%             2.095415       0.000000
75%             2.462415       1.000000
max             3.602140       1.000000
```

[ ]:

```
df.head()
```

[ ]:

```
                     Region  Happiness Rank  Happiness Score  Standard Error  \
Country
Switzerland  Western Europe               1            7.587         0.03411
Iceland      Western Europe               2            7.561         0.04884
Denmark      Western Europe               3            7.527         0.03328
Norway       Western Europe               4            7.522         0.03880
Canada        North America               5            7.427         0.03553

             Economy (GDP per Capita)   Family  Health (Life Expectancy)  \
Country
Switzerland                   1.39651  1.34951                   0.94143
Iceland                       1.30232  1.40223                   0.94784
Denmark                       1.32548  1.36058                   0.87464
Norway                        1.45900  1.33095                   0.88521
Canada                        1.32629  1.32261                   0.90563

             Freedom  Trust (Government Corruption)  Generosity  \
```

```
Country
Switzerland  0.66557                        0.41978     0.29678
Iceland      0.62877                        0.14145     0.43630
Denmark      0.64938                        0.48357     0.34139
Norway       0.66973                        0.36503     0.34699
Canada       0.63297                        0.32957     0.45811

             Dystopia Residual  ClassHapiness
Country
Switzerland            2.51738              1
Iceland                2.70201              1
Denmark                2.49204              1
Norway                 2.46531              1
Canada                 2.45176              1
```

[ ]:

```
## Encoding Variables##
le= LabelEncoder()
df['Region']=le.fit_transform(df["Region"])
column_names = df.columns
df.head()
```

[ ]:

```
             Region  Happiness Rank  Happiness Score  Standard Error  \
Country
Switzerland       9               1            7.587         0.03411
Iceland           9               2            7.561         0.04884
Denmark           9               3            7.527         0.03328
Norway            9               4            7.522         0.03880
Canada            5               5            7.427         0.03553

             Economy (GDP per Capita)   Family  Health (Life Expectancy)  \
Country
Switzerland                   1.39651  1.34951                   0.94143
Iceland                       1.30232  1.40223                   0.94784
Denmark                       1.32548  1.36058                   0.87464
Norway                        1.45900  1.33095                   0.88521
Canada                        1.32629  1.32261                   0.90563

             Freedom  Trust (Government Corruption)  Generosity  \
Country
Switzerland  0.66557                        0.41978     0.29678
Iceland      0.62877                        0.14145     0.43630
Denmark      0.64938                        0.48357     0.34139
Norway       0.66973                        0.36503     0.34699
Canada       0.63297                        0.32957     0.45811

             Dystopia Residual  ClassHapiness
Country
```

```
Switzerland              2.51738              1
Iceland                  2.70201              1
Denmark                  2.49204              1
Norway                   2.46531              1
Canada                   2.45176              1
```

[ ]:
```
## Scales ##
scaling_procedure_1 = MinMaxScaler(feature_range= (0,1))
```

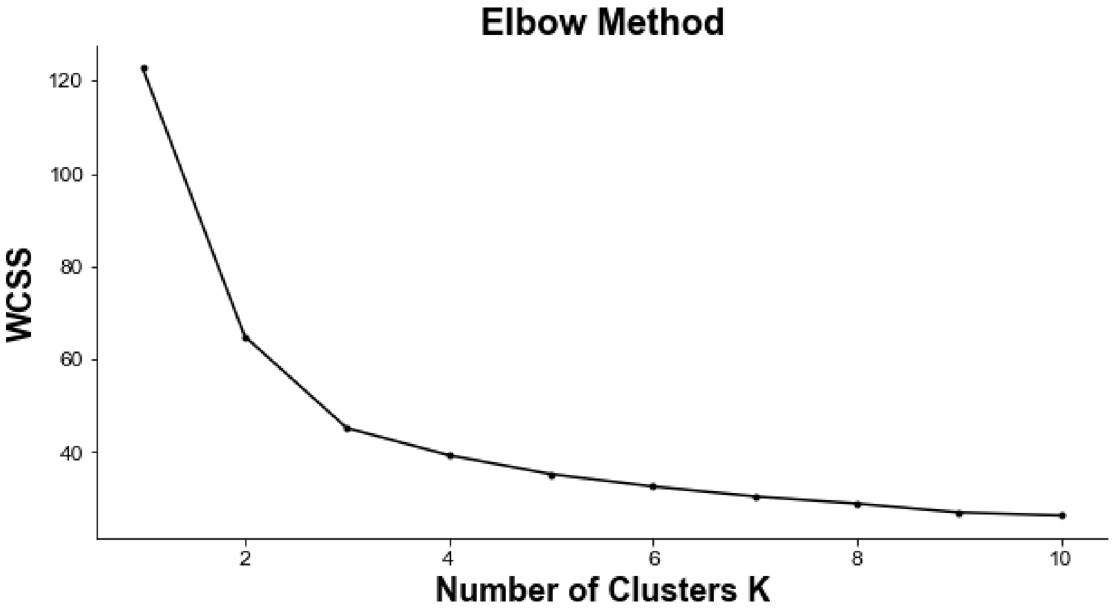
[ ]:
```
## Scaled Data ##
df_scaled = scaling_procedure_1.fit_transform(df)
df_scaled = pd.DataFrame(df_scaled, columns = column_names)
df_scaled.head()
```

[ ]:
```
     Region  Happiness Rank  Happiness Score  Standard Error  \
0  1.000000        0.000000         1.000000        0.131954
1  1.000000        0.006369         0.994524        0.256311
2  1.000000        0.012739         0.987363        0.124947
3  1.000000        0.019108         0.986310        0.171549
4  0.555556        0.025478         0.966302        0.143943

   Economy (GDP per Capita)    Family  Health (Life Expectancy)   Freedom  \
0                  0.826132  0.962403                  0.918244  0.993789
1                  0.770412  1.000000                  0.924496  0.938841
2                  0.784113  0.970297                  0.853099  0.969615
3                  0.863099  0.949167                  0.863409  1.000000
4                  0.784592  0.943219                  0.883326  0.945112

   Trust (Government Corruption)  Generosity  Dystopia Residual  ClassHapiness
0                       0.760595    0.372895           0.668630            1.0
1                       0.256292    0.548198           0.725030            1.0
2                       0.876175    0.428947           0.660889            1.0
3                       0.661394    0.435983           0.652724            1.0
4                       0.597144    0.575602           0.648584            1.0
```

[ ]:
```
from sklearn.cluster import KMeans
## Create the clusters graph ##
wcss = []

for i in range(1,11):
    kmeans = KMeans(n_clusters = i, max_iter=300)
    kmeans.fit(df_scaled)
    wcss.append(kmeans.inertia_)
```

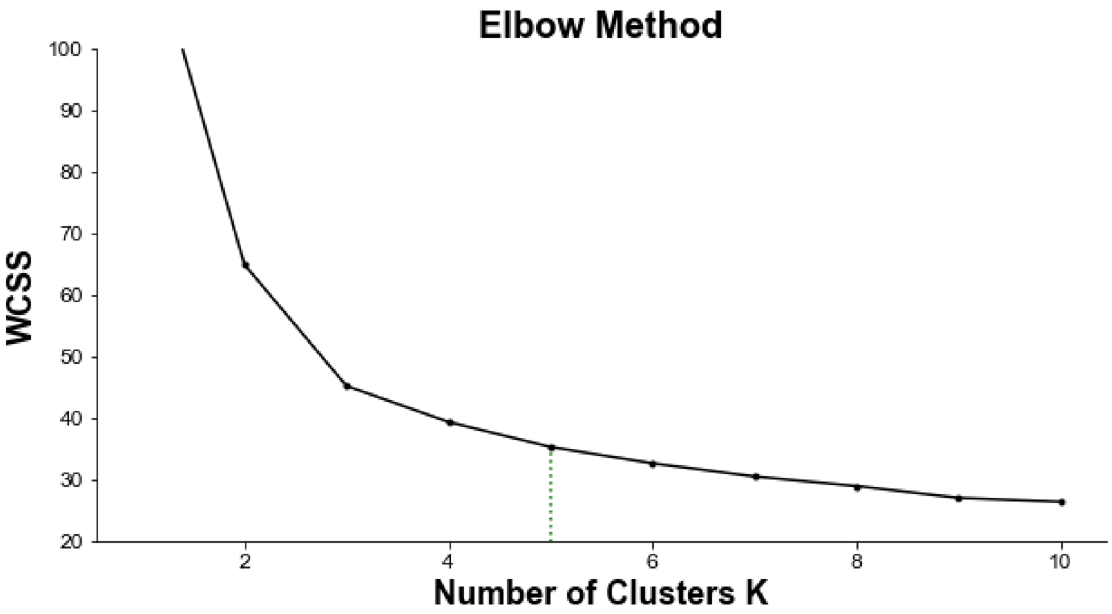
[ ]:
```
## Graph for clusters ##
plt.figure(figsize=(10,5))
plt.plot(range(1,11),wcss, marker=".", c = "k" )
```

```
plt.title("Elbow Method",**fontT)
plt.xlabel("Number of Clusters K",**fontL)
plt.ylabel("WCSS",**fontL)
plt.xticks(fontsize = 12 , family = "Arial")
plt.yticks(fontsize = 12 , family = "Arial")
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
```



```
[ ]: ## Graph for clusters ##
plt.figure(figsize=(10,5))
plt.plot(range(1,11),wcss, marker=".", c = "k" )
plt.title("Elbow Method",**fontT)
plt.xlabel("Number of Clusters K",**fontL)
plt.ylabel("WCSS",**fontL)
plt.xticks(fontsize = 12 , family = "Arial")
plt.yticks(fontsize = 12 , family = "Arial")
plt.ylim(20,100)
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.vlines(x=5, ymin=0, ymax= wcss[4] ,colors='green', ls=':')
```

```
[ ]: <matplotlib.collections.LineCollection at 0x20e640cdeb0>
```



[ ]:

```
kmeans = KMeans(n_clusters = 5, init = 'k-means++', random_state = 42)
kmeans.fit(df_scaled)
print(kmeans.labels_)
```

```
[2 2 2 2 2 2 2 2 3 3 3 3 2 3 3 3 2 2 2 2 2 3 3 2 3 2 3 2 2 3 3 3 3 3 3 2 2
 3 3 3 3 3 3 3 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 4 4 0 0 0 4 0 0 4
 4 0 0 1 4 0 1 0 0 0 1 0 0 4 0 4 1 0 0 1 0 0 1 0 4 0 1 4 0 0 0 0 0 0 1 0 0
 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 4 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1]
```

[ ]:

```
## Cluster to dataset ##
df['Cluster'] = kmeans.labels_
```

[ ]:

```
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import scale
```

[ ]:

```
pca_pipe = make_pipeline(StandardScaler(), PCA())
pca_pipe.fit(df)


modelo_pca = pca_pipe.named_steps['pca']
```

[ ]:

```
PCNames = []
for i in range(1,14):
    nombre = "PC " + str(i)
```

```
    PCNames.append(nombre)
```

[ ]:

```
# Se combierte el array a dataframe para añadir nombres a los ejes.
pd.DataFrame(
    data    = modelo_pca.components_,
    columns = df.columns,
    index   = PCNames
)
```

[ ]:

```
          Region  Happiness Rank  Happiness Score  Standard Error  \
PC 1   0.054120        0.405688        -0.408508        0.105235
PC 2  -0.567788       -0.110449         0.085991        0.018146
PC 3  -0.092015        0.138313        -0.138018       -0.440407
PC 4  -0.214426        0.012076        -0.034342        0.834893
PC 5   0.141066        0.043551        -0.056964       -0.029107
PC 6  -0.406538       -0.073538         0.068900       -0.158281
PC 7  -0.277065        0.038535        -0.064917        0.152845
PC 8   0.400599       -0.091732         0.114097        0.167742
PC 9  -0.419988        0.123826        -0.114404       -0.110550
PC 10 -0.066276       -0.135073         0.110698       -0.016022
PC 11 -0.127468       -0.074224         0.073510       -0.086245
PC 12  0.028189       -0.864668        -0.288824       -0.016645
PC 13 -0.000028       -0.000101        -0.816155       -0.000008

        Economy (GDP per Capita)    Family  Health (Life Expectancy)   Freedom  \
PC 1                   -0.343943 -0.316719                 -0.324738 -0.281418
PC 2                    0.223272  0.136370                  0.230717 -0.278487
PC 3                    0.206463  0.031823                  0.286686  0.096356
PC 4                    0.070603  0.184691                  0.011425  0.228315
PC 5                    0.153493  0.154832                  0.196966 -0.185248
PC 6                   -0.269159 -0.099999                  0.061382  0.046942
PC 7                    0.130711 -0.712204                  0.315868 -0.252575
PC 8                    0.226105  0.113206                  0.115328 -0.697608
PC 9                   -0.182378  0.433251                 -0.345170 -0.413109
PC 10                  -0.126628  0.113211                  0.115980 -0.063098
PC 11                   0.667298 -0.195442                 -0.647941  0.045114
PC 12                  -0.207071 -0.124682                 -0.141221 -0.081620
PC 13                   0.287339  0.194112                  0.176051  0.107368

        Trust (Government Corruption)  Generosity  Dystopia Residual  \
PC 1                        -0.205565   -0.108485          -0.147734
PC 2                        -0.286736   -0.462755           0.089042
PC 3                         0.234025    0.156135          -0.692154
PC 4                         0.045099    0.202285          -0.336640
PC 5                        -0.708215    0.122698          -0.217785
PC 6                        -0.246545    0.743011           0.230940
PC 7                         0.220792    0.017963          -0.003288
```

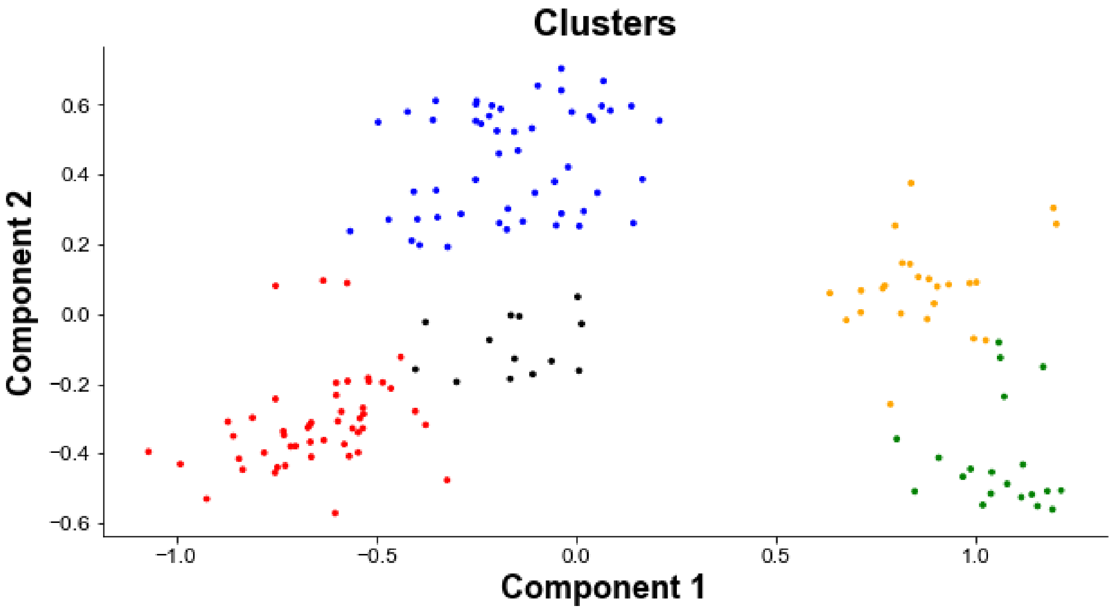
```
PC 8                         0.168096    0.326993            0.042757
PC 9                         0.263233    0.053874           -0.119962
PC 10                        0.313059   -0.025201            0.168804
PC 11                       -0.045490    0.140267            0.016815
PC 12                       -0.051766   -0.055289           -0.276439
PC 13                        0.085546    0.090303            0.394528

       ClassHapiness   Cluster
PC 1       -0.358464 -0.216631
PC 2       -0.138715 -0.367329
PC 3       -0.171383 -0.191540
PC 4       -0.082934 -0.033747
PC 5       -0.054785  0.535173
PC 6       -0.074100 -0.198563
PC 7        0.198500  0.343224
PC 8       -0.031547 -0.296733
PC 9        0.298014  0.311127
PC 10      -0.801969  0.385069
PC 11      -0.186357  0.057423
PC 12       0.020649 -0.015737
PC 13       0.000007 -0.000002
```

```
[ ]: pca = PCA(n_components=2)
     pca_df = pca.fit_transform(df_scaled)
```

```
[ ]: pca_df_table = pd.DataFrame(data = pca_df , columns = ["Component 1" ,␣
      ↪"Component 2"])

     pca_names_df = pca_df_table
     pca_names_df["Cluster"] = df["Cluster"].values
     pca_names_df
```

```
[ ]:     Component 1  Component 2  Cluster
     0       1.213124    -0.505813        2
     1       1.117556    -0.431867        2
     2       1.191166    -0.560488        2
     3       1.178393    -0.507933        2
     4       1.167814    -0.151633        2
     ..           …            …      …
     153    -0.601751    -0.571173        1
     154    -0.842278    -0.415221        1
     155    -0.750699     0.080603        1
     156    -1.068387    -0.395021        1
     157    -0.988804    -0.430668        1

     [158 rows x 3 columns]
```

```
[ ]: ## Graph for clusters ##
     plt.figure(figsize=(10,5))
     plt.title("Clusters",**fontT)
     plt.xlabel("Component 1",**fontL)
     plt.ylabel("Component 2",**fontL)
     plt.xticks(fontsize = 12 , family = "Arial")
     plt.yticks(fontsize = 12 , family = "Arial")
     colors = np.array(["blue", "red","green","orange","black"])
     plt.scatter(x = pca_names_df['Component 1'],y = pca_names_df['Component 2'],␣
      ↪marker=".", c = colors[pca_names_df.Cluster] , s = 30)
     plt.gca().spines['top'].set_visible(False)
     plt.gca().spines['right'].set_visible(False)
```



In this case we can see that we choose to have 5 clusters, we chose 5 clusters because according to the elbow method that is the point in which the WCSS is starting to stabilize. In this case we did PCA analysis to plot our 12-dimension dataset into a 2-D dataset. In this case the first components are high on the happiness rank and the low GDP. The second component relates to economy and are on regions around Europe.

The first cluster with color red are represented by negative values of component 2 and component 1, that means that in this cluster we have members that are low on happiness and high on the GDP. The black cluster that is in the middle isn't represented by either component 1 or 2. The blue cluster on the top represents elements that are high on component two and on the middle of component 1, this means that elements from this cluster are high on economy but don't relate a lot to the region. The yellow cluster is on the right of component 1 and on the middle near 0 of component 2, this means that elements of this cluster are not in Europe and are average on economy. The green cluster is on the right of component 1 and on the bottom 0 of component 2, this means that elements of this cluster are not in Europe and are low on economy.

1.3 PART TWO Classification analysis

```
[ ]: ## Target ##
     y = df_scaled["ClassHapiness"]
     y.head()
```

```
[ ]: 0    1.0
     1    1.0
     2    1.0
     3    1.0
     4    1.0
     Name: ClassHapiness, dtype: float64
```

```
[ ]: ## Rest of Variables ##
     x = df_scaled[column_names.drop(["ClassHapiness"])]
     x.head()
```

```
[ ]:      Region  Happiness Rank  Happiness Score  Standard Error  \
     0  1.000000        0.000000         1.000000        0.131954
     1  1.000000        0.006369         0.994524        0.256311
     2  1.000000        0.012739         0.987363        0.124947
     3  1.000000        0.019108         0.986310        0.171549
     4  0.555556        0.025478         0.966302        0.143943

        Economy (GDP per Capita)    Family  Health (Life Expectancy)   Freedom  \
     0                  0.826132  0.962403                  0.918244  0.993789
     1                  0.770412  1.000000                  0.924496  0.938841
     2                  0.784113  0.970297                  0.853099  0.969615
     3                  0.863099  0.949167                  0.863409  1.000000
     4                  0.784592  0.943219                  0.883326  0.945112

        Trust (Government Corruption)  Generosity  Dystopia Residual
     0                       0.760595    0.372895           0.668630
     1                       0.256292    0.548198           0.725030
     2                       0.876175    0.428947           0.660889
     3                       0.661394    0.435983           0.652724
     4                       0.597144    0.575602           0.648584
```

1.4 Split the data

```
[ ]: from sklearn.model_selection import train_test_split
     from sklearn import metrics
     from sklearn.metrics import classification_report
```

```
[ ]: #Separate train and test data
     x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.
      ↪20,random_state=0)
```

```
[ ]: print("Size of the full data set: ",x.shape)
     print("Size of the training data set: ",x_train.shape)
     print("Size of the test data set: ",x_test.shape)
```

```
Size of the full data set:  (158, 11)
Size of the training data set:  (126, 11)
Size of the test data set:  (32, 11)
```

1.5 Logistic Regression

```
[ ]: from sklearn.linear_model import LogisticRegression
```

```
[ ]: #Fit classifier with train data
     logistic_regression= LogisticRegression()
     logistic_regression.fit(x_train,y_train)
```

```
[ ]: LogisticRegression()
```

```
[ ]: #Predict test data
     y_pred=logistic_regression.predict(x_test)
     print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.9375
```

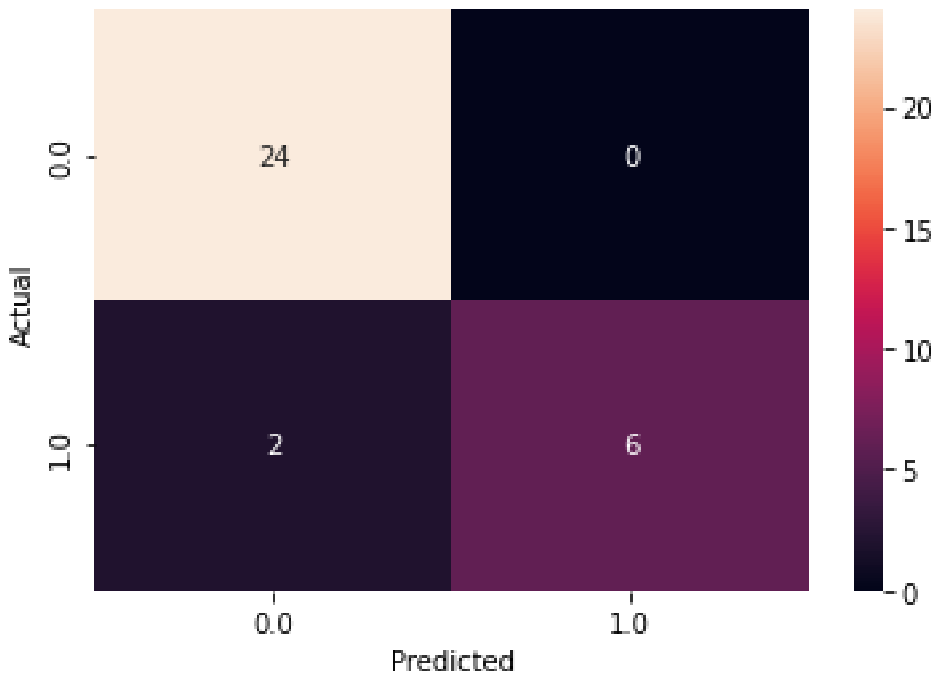
```
[ ]: from sklearn.metrics import confusion_matrix
```

```
[ ]: #Get right and wrong classifications
     cm = confusion_matrix(y_test,y_pred)
     print(cm)
     tn, fp, fn, tp = cm.ravel()
```

```
[[24  0]
 [ 2  6]]
```

```
[ ]: #Pretty print confusion matrix
     cm2 = pd.crosstab(y_test,y_pred,rownames=['Actual'],colnames=['Predicted'])
     sns.heatmap(cm2,annot=True)
```

```
[ ]: <AxesSubplot:xlabel='Predicted', ylabel='Actual'>
```



1.6 KNN

[ ]:
```
from sklearn.neighbors import KNeighborsClassifier

#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=5)

#Train the model using the training sets
knn.fit(x_train, y_train)

#Predict the response for test dataset
y_pred = knn.predict(x_test)
```

[ ]:
```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

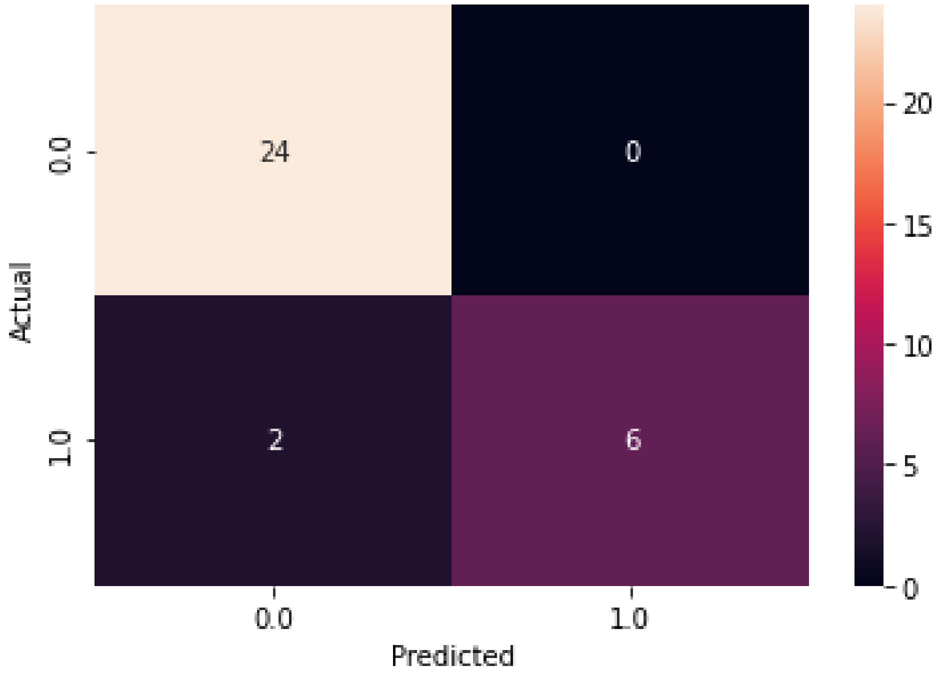
```
Accuracy: 0.9375
```

[ ]:
```
#Get right and wrong classifications
cm = confusion_matrix(y_test,y_pred)
print(cm)
tn, fp, fn, tp = cm.ravel()
```

```
[[24  0]
 [ 2  6]]
```

[ ]:
```
#Pretty print confusion matrix
cm2 = pd.crosstab(y_test,y_pred,rownames=['Actual'],colnames=['Predicted'])
sns.heatmap(cm2,annot=True)
```

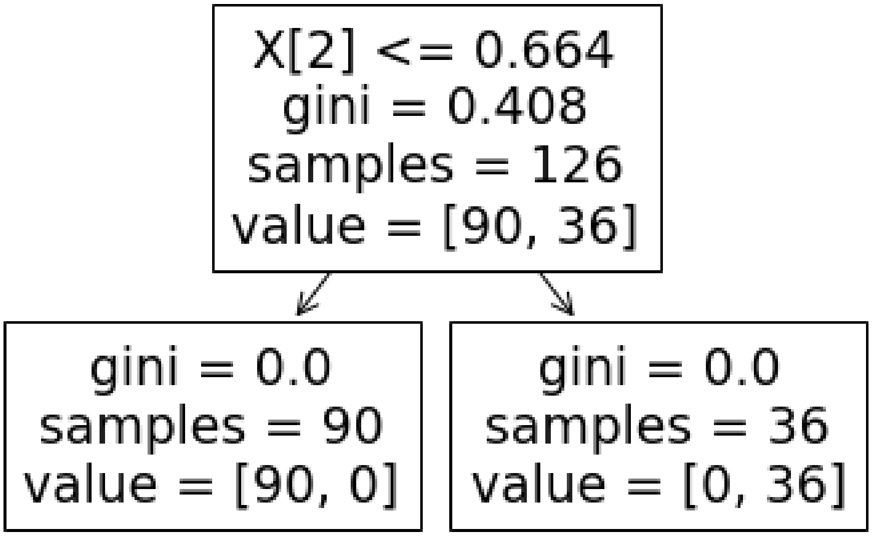
[ ]:
```
<AxesSubplot:xlabel='Predicted', ylabel='Actual'>
```



1.7 Desicion Tree

[ ]:
```
from sklearn import tree

dtc = tree.DecisionTreeClassifier(max_depth = 4,random_state=0)
rfs = RandomForestClassifier(n_estimators=1, random_state=0, max_depth = 3,␣
 ↪bootstrap = False)
dtc.fit(x_train,y_train)
tree.plot_tree(dtc)
```

[ ]:
```
[Text(167.4, 163.07999999999998, 'X[2] <= 0.664\ngini = 0.408\nsamples =
126\nvalue = [90, 36]'),
 Text(83.7, 54.360000000000014, 'gini = 0.0\nsamples = 90\nvalue = [90, 0]'),
 Text(251.10000000000002, 54.360000000000014, 'gini = 0.0\nsamples = 36\nvalue =
[0, 36]')]
```



[ ]:

```
dtpred = dtc.predict(x_test)
print(confusion_matrix(y_test, dtpred))
print(classification_report(y_test, dtpred))
```

```
[[23  1]
 [ 0  8]]
              precision    recall  f1-score   support

         0.0       1.00      0.96      0.98        24
         1.0       0.89      1.00      0.94         8

    accuracy                           0.97        32
   macro avg       0.94      0.98      0.96        32
weighted avg       0.97      0.97      0.97        32
```

[ ]:

```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

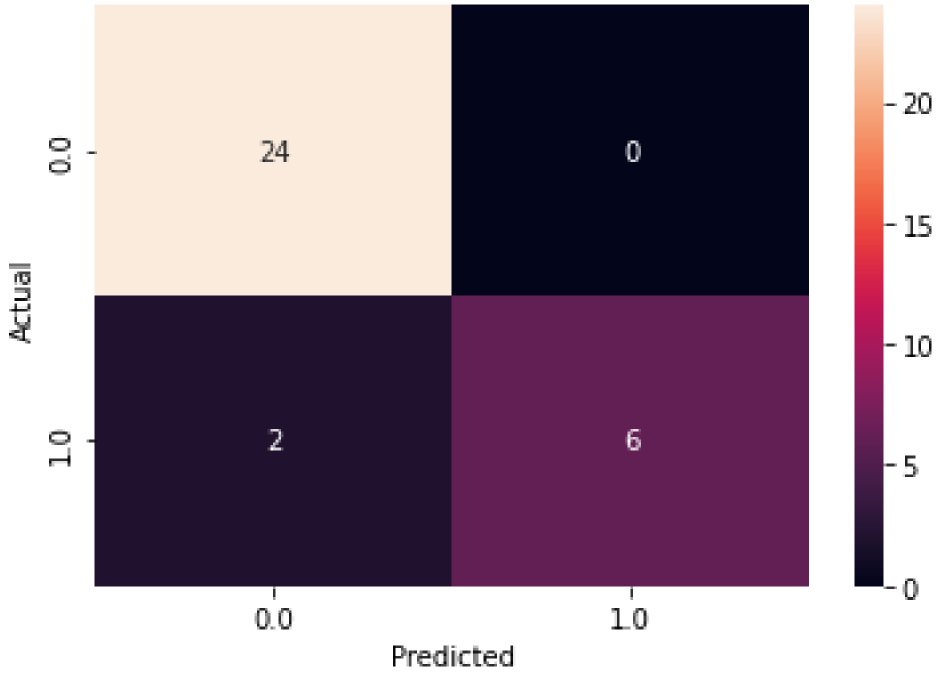
```
Accuracy: 0.9375
```

[ ]:

```
#Pretty print confusion matrix
cm2 = pd.crosstab(y_test,y_pred,rownames=['Actual'],colnames=['Predicted'])
sns.heatmap(cm2,annot=True)
```

[ ]:

```
<AxesSubplot:xlabel='Predicted', ylabel='Actual'>
```



1.8 Random Forrest

```
[ ]: from sklearn.ensemble import RandomForestClassifier

     #Fit classifier with train data
     rf = RandomForestClassifier(n_estimators=20, random_state=0, max_depth = 3)
     rf.fit(x_train,y_train)
```

```
[ ]: RandomForestClassifier(max_depth=3, n_estimators=20, random_state=0)
```

```
[ ]: #Predict test data
     y_pred=rf.predict(x_test)
     print(y_pred)
```

```
[1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1.
 1. 1. 0. 1. 0. 0. 0. 0.]
```

```
[ ]: #Get right and wrong classifications
     cm = confusion_matrix(y_test,y_pred)
     print(cm)
     tn, fp, fn, tp = cm.ravel()
```

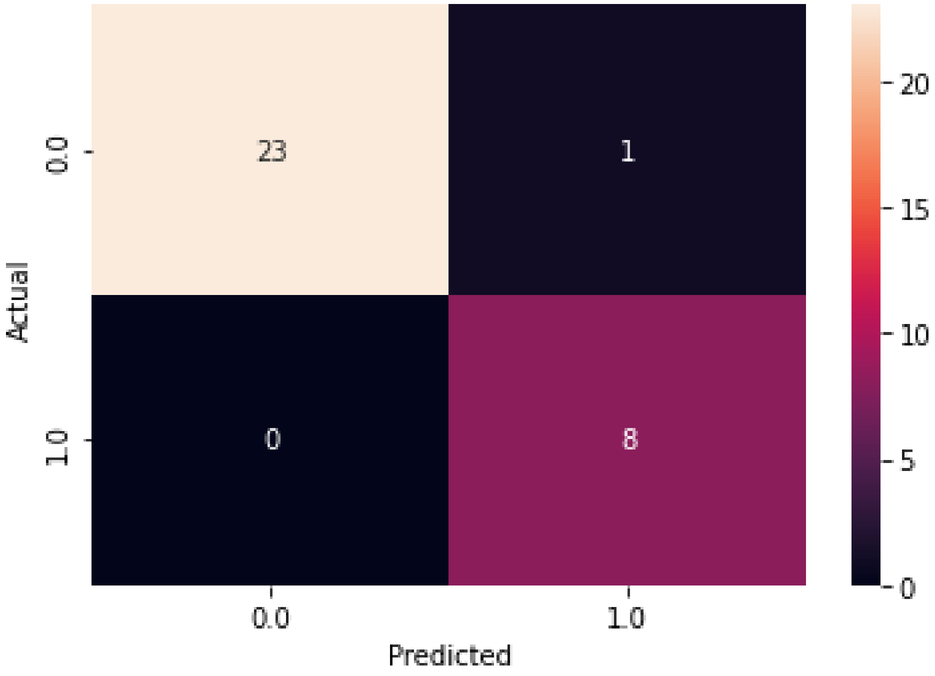
```
[[23  1]
 [ 0  8]]
```

```
[ ]: print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.96875
```

```
[ ]: #Pretty print confusion matrix
cm2 = pd.crosstab(y_test,y_pred,rownames=['Actual'],colnames=['Predicted'])
sns.heatmap(cm2,annot=True)
```

```
[ ]: <AxesSubplot:xlabel='Predicted', ylabel='Actual'>
```



1.9 Tensorflow

```
[ ]: happiness_model = tf.keras.models.Sequential()
##Capa de entrada##
happiness_model.add(tf.keras.layers.Dense(20,input_dim = 11, activation =␣
 ↪"relu"))
happiness_model.add(tf.keras.layers.Dense(10, activation = "relu"))
## Salida ##
happiness_model.add(tf.keras.layers.Dense(1, activation = "sigmoid"))
```

```
[ ]: ##Problemas de una sola salida ##
happiness_model.compile(loss="binary_crossentropy", optimizer="adam" ,␣
 ↪metrics="accuracy")
```

```
[ ]: happiness_model.summary()
```

```
Model: "sequential_15"
_________________________________________________________________
```

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_45 (Dense)             (None, 20)                240
_________________________________________________________________
dense_46 (Dense)             (None, 10)                210
_________________________________________________________________
dense_47 (Dense)             (None, 1)                 11
=================================================================
Total params: 461
Trainable params: 461
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
history = happiness_model.fit(x_train,y_train,epochs= 150, batch_size=␣
↪16,validation_data=(x_test,y_test))
```

```
Epoch 1/150
8/8 [==============================] - 1s 20ms/step - loss: 0.6729 - accuracy:
0.7063 - val_loss: 0.6463 - val_accuracy: 0.7500
Epoch 2/150
8/8 [==============================] - 0s 4ms/step - loss: 0.6480 - accuracy:
0.7143 - val_loss: 0.6215 - val_accuracy: 0.7500
Epoch 3/150
8/8 [==============================] - 0s 4ms/step - loss: 0.6264 - accuracy:
0.7143 - val_loss: 0.5986 - val_accuracy: 0.7500
Epoch 4/150
8/8 [==============================] - 0s 3ms/step - loss: 0.6048 - accuracy:
0.7143 - val_loss: 0.5780 - val_accuracy: 0.7500
Epoch 5/150
8/8 [==============================] - 0s 4ms/step - loss: 0.5856 - accuracy:
0.7143 - val_loss: 0.5585 - val_accuracy: 0.7500
Epoch 6/150
8/8 [==============================] - 0s 3ms/step - loss: 0.5644 - accuracy:
0.7143 - val_loss: 0.5413 - val_accuracy: 0.7500
Epoch 7/150
8/8 [==============================] - 0s 4ms/step - loss: 0.5444 - accuracy:
0.7143 - val_loss: 0.5242 - val_accuracy: 0.7500
Epoch 8/150
8/8 [==============================] - 0s 4ms/step - loss: 0.5231 - accuracy:
0.7143 - val_loss: 0.5067 - val_accuracy: 0.7500
Epoch 9/150
8/8 [==============================] - 0s 3ms/step - loss: 0.4981 - accuracy:
0.7143 - val_loss: 0.4887 - val_accuracy: 0.7500
Epoch 10/150
8/8 [==============================] - 0s 3ms/step - loss: 0.4705 - accuracy:
0.7143 - val_loss: 0.4700 - val_accuracy: 0.7500
Epoch 11/150
8/8 [==============================] - 0s 4ms/step - loss: 0.4459 - accuracy:
```

```
0.7143 - val_loss: 0.4524 - val_accuracy: 0.7500
Epoch 12/150
8/8 [==============================] - 0s 4ms/step - loss: 0.4201 - accuracy:
0.7460 - val_loss: 0.4328 - val_accuracy: 0.7500
Epoch 13/150
8/8 [==============================] - 0s 3ms/step - loss: 0.3994 - accuracy:
0.7540 - val_loss: 0.4183 - val_accuracy: 0.7812
Epoch 14/150
8/8 [==============================] - 0s 4ms/step - loss: 0.3799 - accuracy:
0.7698 - val_loss: 0.4039 - val_accuracy: 0.7812
Epoch 15/150
8/8 [==============================] - 0s 4ms/step - loss: 0.3632 - accuracy:
0.7937 - val_loss: 0.3917 - val_accuracy: 0.8125
Epoch 16/150
8/8 [==============================] - 0s 3ms/step - loss: 0.3483 - accuracy:
0.8492 - val_loss: 0.3808 - val_accuracy: 0.8125
Epoch 17/150
8/8 [==============================] - 0s 4ms/step - loss: 0.3336 - accuracy:
0.8810 - val_loss: 0.3704 - val_accuracy: 0.8438
Epoch 18/150
8/8 [==============================] - 0s 4ms/step - loss: 0.3197 - accuracy:
0.8968 - val_loss: 0.3604 - val_accuracy: 0.8438
Epoch 19/150
8/8 [==============================] - 0s 3ms/step - loss: 0.3068 - accuracy:
0.9048 - val_loss: 0.3513 - val_accuracy: 0.8750
Epoch 20/150
8/8 [==============================] - 0s 4ms/step - loss: 0.2951 - accuracy:
0.9127 - val_loss: 0.3427 - val_accuracy: 0.8750
Epoch 21/150
8/8 [==============================] - 0s 4ms/step - loss: 0.2837 - accuracy:
0.9286 - val_loss: 0.3344 - val_accuracy: 0.8750
Epoch 22/150
8/8 [==============================] - 0s 3ms/step - loss: 0.2729 - accuracy:
0.9365 - val_loss: 0.3263 - val_accuracy: 0.8750
Epoch 23/150
8/8 [==============================] - 0s 4ms/step - loss: 0.2615 - accuracy:
0.9365 - val_loss: 0.3187 - val_accuracy: 0.8750
Epoch 24/150
8/8 [==============================] - 0s 3ms/step - loss: 0.2521 - accuracy:
0.9524 - val_loss: 0.3117 - val_accuracy: 0.8750
Epoch 25/150
8/8 [==============================] - 0s 3ms/step - loss: 0.2413 - accuracy:
0.9603 - val_loss: 0.3042 - val_accuracy: 0.8750
Epoch 26/150
8/8 [==============================] - 0s 4ms/step - loss: 0.2318 - accuracy:
0.9603 - val_loss: 0.2972 - val_accuracy: 0.8438
Epoch 27/150
8/8 [==============================] - 0s 4ms/step - loss: 0.2224 - accuracy:
```

```
0.9603 - val_loss: 0.2905 - val_accuracy: 0.8750
Epoch 28/150
8/8 [==============================] - 0s 3ms/step - loss: 0.2133 - accuracy:
0.9683 - val_loss: 0.2838 - val_accuracy: 0.8438
Epoch 29/150
8/8 [==============================] - 0s 4ms/step - loss: 0.2034 - accuracy:
0.9683 - val_loss: 0.2777 - val_accuracy: 0.8438
Epoch 30/150
8/8 [==============================] - 0s 4ms/step - loss: 0.1959 - accuracy:
0.9683 - val_loss: 0.2718 - val_accuracy: 0.8750
Epoch 31/150
8/8 [==============================] - 0s 3ms/step - loss: 0.1890 - accuracy:
0.9683 - val_loss: 0.2663 - val_accuracy: 0.8750
Epoch 32/150
8/8 [==============================] - 0s 4ms/step - loss: 0.1805 - accuracy:
0.9683 - val_loss: 0.2605 - val_accuracy: 0.8750
Epoch 33/150
8/8 [==============================] - 0s 4ms/step - loss: 0.1738 - accuracy:
0.9683 - val_loss: 0.2556 - val_accuracy: 0.8750
Epoch 34/150
8/8 [==============================] - 0s 4ms/step - loss: 0.1657 - accuracy:
0.9683 - val_loss: 0.2500 - val_accuracy: 0.8750
Epoch 35/150
8/8 [==============================] - 0s 3ms/step - loss: 0.1593 - accuracy:
0.9683 - val_loss: 0.2452 - val_accuracy: 0.8750
Epoch 36/150
8/8 [==============================] - 0s 4ms/step - loss: 0.1535 - accuracy:
0.9762 - val_loss: 0.2407 - val_accuracy: 0.8750
Epoch 37/150
8/8 [==============================] - 0s 4ms/step - loss: 0.1485 - accuracy:
0.9683 - val_loss: 0.2366 - val_accuracy: 0.8750
Epoch 38/150
8/8 [==============================] - 0s 3ms/step - loss: 0.1426 - accuracy:
0.9683 - val_loss: 0.2321 - val_accuracy: 0.9062
Epoch 39/150
8/8 [==============================] - 0s 3ms/step - loss: 0.1373 - accuracy:
0.9762 - val_loss: 0.2278 - val_accuracy: 0.8750
Epoch 40/150
8/8 [==============================] - 0s 4ms/step - loss: 0.1321 - accuracy:
0.9762 - val_loss: 0.2239 - val_accuracy: 0.8750
Epoch 41/150
8/8 [==============================] - 0s 4ms/step - loss: 0.1281 - accuracy:
0.9683 - val_loss: 0.2202 - val_accuracy: 0.9062
Epoch 42/150
8/8 [==============================] - 0s 3ms/step - loss: 0.1240 - accuracy:
0.9762 - val_loss: 0.2167 - val_accuracy: 0.8750
Epoch 43/150
8/8 [==============================] - 0s 4ms/step - loss: 0.1198 - accuracy:
```

```
0.9762 - val_loss: 0.2131 - val_accuracy: 0.9062
Epoch 44/150
8/8 [==============================] - 0s 4ms/step - loss: 0.1163 - accuracy:
0.9762 - val_loss: 0.2096 - val_accuracy: 0.9375
Epoch 45/150
8/8 [==============================] - 0s 3ms/step - loss: 0.1127 - accuracy:
0.9683 - val_loss: 0.2064 - val_accuracy: 0.9375
Epoch 46/150
8/8 [==============================] - 0s 4ms/step - loss: 0.1107 - accuracy:
0.9762 - val_loss: 0.2040 - val_accuracy: 0.8750
Epoch 47/150
8/8 [==============================] - 0s 4ms/step - loss: 0.1082 - accuracy:
0.9683 - val_loss: 0.2005 - val_accuracy: 0.9375
Epoch 48/150
8/8 [==============================] - 0s 3ms/step - loss: 0.1040 - accuracy:
0.9683 - val_loss: 0.1977 - val_accuracy: 0.9375
Epoch 49/150
8/8 [==============================] - 0s 7ms/step - loss: 0.1011 - accuracy:
0.9762 - val_loss: 0.1950 - val_accuracy: 0.9375
Epoch 50/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0994 - accuracy:
0.9683 - val_loss: 0.1921 - val_accuracy: 0.9375
Epoch 51/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0968 - accuracy:
0.9762 - val_loss: 0.1897 - val_accuracy: 0.9375
Epoch 52/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0937 - accuracy:
0.9841 - val_loss: 0.1870 - val_accuracy: 0.9375
Epoch 53/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0917 - accuracy:
0.9762 - val_loss: 0.1844 - val_accuracy: 0.9375
Epoch 54/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0900 - accuracy:
0.9841 - val_loss: 0.1832 - val_accuracy: 0.9062
Epoch 55/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0872 - accuracy:
0.9841 - val_loss: 0.1798 - val_accuracy: 0.9375
Epoch 56/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0855 - accuracy:
0.9841 - val_loss: 0.1769 - val_accuracy: 0.9375
Epoch 57/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0858 - accuracy:
0.9683 - val_loss: 0.1748 - val_accuracy: 0.9375
Epoch 58/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0831 - accuracy:
0.9762 - val_loss: 0.1727 - val_accuracy: 0.9375
Epoch 59/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0803 - accuracy:
```

```
0.9921 - val_loss: 0.1706 - val_accuracy: 0.9375
Epoch 60/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0797 - accuracy:
0.9841 - val_loss: 0.1683 - val_accuracy: 0.9375
Epoch 61/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0764 - accuracy:
0.9921 - val_loss: 0.1682 - val_accuracy: 0.9062
Epoch 62/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0771 - accuracy:
0.9841 - val_loss: 0.1649 - val_accuracy: 0.9062
Epoch 63/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0756 - accuracy:
0.9841 - val_loss: 0.1646 - val_accuracy: 0.9062
Epoch 64/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0724 - accuracy:
0.9841 - val_loss: 0.1597 - val_accuracy: 0.9375
Epoch 65/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0715 - accuracy:
0.9841 - val_loss: 0.1574 - val_accuracy: 0.9375
Epoch 66/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0709 - accuracy:
0.9841 - val_loss: 0.1557 - val_accuracy: 0.9375
Epoch 67/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0694 - accuracy:
0.9921 - val_loss: 0.1543 - val_accuracy: 0.9375
Epoch 68/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0685 - accuracy:
0.9921 - val_loss: 0.1525 - val_accuracy: 0.9375
Epoch 69/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0674 - accuracy:
0.9841 - val_loss: 0.1499 - val_accuracy: 0.9375
Epoch 70/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0658 - accuracy:
1.0000 - val_loss: 0.1492 - val_accuracy: 0.9062
Epoch 71/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0651 - accuracy:
0.9921 - val_loss: 0.1466 - val_accuracy: 0.9375
Epoch 72/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0637 - accuracy:
1.0000 - val_loss: 0.1452 - val_accuracy: 0.9375
Epoch 73/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0629 - accuracy:
1.0000 - val_loss: 0.1439 - val_accuracy: 0.9062
Epoch 74/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0624 - accuracy:
0.9921 - val_loss: 0.1417 - val_accuracy: 0.9375
Epoch 75/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0608 - accuracy:
```

```
0.9921 - val_loss: 0.1393 - val_accuracy: 0.9375
Epoch 76/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0611 - accuracy:
0.9841 - val_loss: 0.1384 - val_accuracy: 0.9375
Epoch 77/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0591 - accuracy:
1.0000 - val_loss: 0.1363 - val_accuracy: 0.9375
Epoch 78/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0581 - accuracy:
1.0000 - val_loss: 0.1346 - val_accuracy: 0.9375
Epoch 79/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0578 - accuracy:
1.0000 - val_loss: 0.1347 - val_accuracy: 0.9375
Epoch 80/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0567 - accuracy:
1.0000 - val_loss: 0.1322 - val_accuracy: 0.9688
Epoch 81/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0557 - accuracy:
1.0000 - val_loss: 0.1306 - val_accuracy: 0.9688
Epoch 82/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0557 - accuracy:
1.0000 - val_loss: 0.1302 - val_accuracy: 0.9375
Epoch 83/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0538 - accuracy:
1.0000 - val_loss: 0.1274 - val_accuracy: 0.9375
Epoch 84/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0544 - accuracy:
1.0000 - val_loss: 0.1266 - val_accuracy: 0.9688
Epoch 85/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0529 - accuracy:
1.0000 - val_loss: 0.1248 - val_accuracy: 0.9688
Epoch 86/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0522 - accuracy:
1.0000 - val_loss: 0.1232 - val_accuracy: 0.9688
Epoch 87/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0518 - accuracy:
0.9921 - val_loss: 0.1214 - val_accuracy: 0.9375
Epoch 88/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0508 - accuracy:
1.0000 - val_loss: 0.1210 - val_accuracy: 0.9688
Epoch 89/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0503 - accuracy:
1.0000 - val_loss: 0.1207 - val_accuracy: 0.9375
Epoch 90/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0512 - accuracy:
0.9921 - val_loss: 0.1173 - val_accuracy: 0.9375
Epoch 91/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0494 - accuracy:
```

```
1.0000 - val_loss: 0.1205 - val_accuracy: 0.9375
Epoch 92/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0504 - accuracy:
1.0000 - val_loss: 0.1155 - val_accuracy: 0.9688
Epoch 93/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0480 - accuracy:
1.0000 - val_loss: 0.1169 - val_accuracy: 0.9375
Epoch 94/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0484 - accuracy:
1.0000 - val_loss: 0.1136 - val_accuracy: 0.9375
Epoch 95/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0466 - accuracy:
1.0000 - val_loss: 0.1133 - val_accuracy: 0.9375
Epoch 96/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0470 - accuracy:
1.0000 - val_loss: 0.1107 - val_accuracy: 0.9688
Epoch 97/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0470 - accuracy:
0.9921 - val_loss: 0.1140 - val_accuracy: 0.9375
Epoch 98/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0451 - accuracy:
1.0000 - val_loss: 0.1093 - val_accuracy: 0.9375
Epoch 99/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0450 - accuracy:
1.0000 - val_loss: 0.1084 - val_accuracy: 0.9375
Epoch 100/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0450 - accuracy:
1.0000 - val_loss: 0.1069 - val_accuracy: 0.9375
Epoch 101/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0440 - accuracy:
1.0000 - val_loss: 0.1050 - val_accuracy: 0.9688
Epoch 102/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0473 - accuracy:
1.0000 - val_loss: 0.1082 - val_accuracy: 0.9375
Epoch 103/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0431 - accuracy:
1.0000 - val_loss: 0.1036 - val_accuracy: 0.9688
Epoch 104/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0441 - accuracy:
1.0000 - val_loss: 0.1033 - val_accuracy: 0.9375
Epoch 105/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0419 - accuracy:
1.0000 - val_loss: 0.1009 - val_accuracy: 0.9688
Epoch 106/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0422 - accuracy:
1.0000 - val_loss: 0.0996 - val_accuracy: 0.9688
Epoch 107/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0417 - accuracy:
```

```
1.0000 - val_loss: 0.1039 - val_accuracy: 0.9375
Epoch 108/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0414 - accuracy:
1.0000 - val_loss: 0.1012 - val_accuracy: 0.9375
Epoch 109/150
8/8 [==============================] - 0s 2ms/step - loss: 0.0423 - accuracy:
1.0000 - val_loss: 0.0964 - val_accuracy: 0.9688
Epoch 110/150
8/8 [==============================] - 0s 5ms/step - loss: 0.0408 - accuracy:
1.0000 - val_loss: 0.0995 - val_accuracy: 0.9375
Epoch 111/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0401 - accuracy:
1.0000 - val_loss: 0.0989 - val_accuracy: 0.9375
Epoch 112/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0396 - accuracy:
1.0000 - val_loss: 0.0983 - val_accuracy: 0.9375
Epoch 113/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0410 - accuracy:
0.9921 - val_loss: 0.0929 - val_accuracy: 0.9688
Epoch 114/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0381 - accuracy:
1.0000 - val_loss: 0.0949 - val_accuracy: 0.9375
Epoch 115/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0392 - accuracy:
1.0000 - val_loss: 0.1016 - val_accuracy: 0.9375
Epoch 116/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0383 - accuracy:
1.0000 - val_loss: 0.0941 - val_accuracy: 0.9375
Epoch 117/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0371 - accuracy:
1.0000 - val_loss: 0.0895 - val_accuracy: 0.9688
Epoch 118/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0379 - accuracy:
0.9921 - val_loss: 0.0887 - val_accuracy: 0.9688
Epoch 119/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0363 - accuracy:
1.0000 - val_loss: 0.0928 - val_accuracy: 0.9375
Epoch 120/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0363 - accuracy:
1.0000 - val_loss: 0.0941 - val_accuracy: 0.9375
Epoch 121/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0373 - accuracy:
1.0000 - val_loss: 0.0892 - val_accuracy: 0.9375
Epoch 122/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0365 - accuracy:
1.0000 - val_loss: 0.0870 - val_accuracy: 0.9688
Epoch 123/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0353 - accuracy:
```

```
1.0000 - val_loss: 0.0901 - val_accuracy: 0.9375
Epoch 124/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0361 - accuracy:
1.0000 - val_loss: 0.0875 - val_accuracy: 0.9375
Epoch 125/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0351 - accuracy:
1.0000 - val_loss: 0.0898 - val_accuracy: 0.9375
Epoch 126/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0349 - accuracy:
1.0000 - val_loss: 0.0864 - val_accuracy: 0.9375
Epoch 127/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0370 - accuracy:
1.0000 - val_loss: 0.0922 - val_accuracy: 0.9375
Epoch 128/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0333 - accuracy:
1.0000 - val_loss: 0.0825 - val_accuracy: 0.9688
Epoch 129/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0341 - accuracy:
1.0000 - val_loss: 0.0821 - val_accuracy: 0.9688
Epoch 130/150
8/8 [==============================] - 0s 2ms/step - loss: 0.0339 - accuracy:
1.0000 - val_loss: 0.0835 - val_accuracy: 0.9375
Epoch 131/150
8/8 [==============================] - 0s 5ms/step - loss: 0.0344 - accuracy:
1.0000 - val_loss: 0.0805 - val_accuracy: 0.9688
Epoch 132/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0347 - accuracy:
1.0000 - val_loss: 0.0910 - val_accuracy: 0.9375
Epoch 133/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0334 - accuracy:
1.0000 - val_loss: 0.0807 - val_accuracy: 0.9375
Epoch 134/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0348 - accuracy:
0.9921 - val_loss: 0.0770 - val_accuracy: 1.0000
Epoch 135/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0326 - accuracy:
1.0000 - val_loss: 0.0832 - val_accuracy: 0.9375
Epoch 136/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0323 - accuracy:
1.0000 - val_loss: 0.0815 - val_accuracy: 0.9375
Epoch 137/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0319 - accuracy:
1.0000 - val_loss: 0.0821 - val_accuracy: 0.9375
Epoch 138/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0316 - accuracy:
1.0000 - val_loss: 0.0800 - val_accuracy: 0.9375
Epoch 139/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0313 - accuracy:
```

```
1.0000 - val_loss: 0.0790 - val_accuracy: 0.9375
Epoch 140/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0311 - accuracy:
1.0000 - val_loss: 0.0766 - val_accuracy: 0.9375
Epoch 141/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0312 - accuracy:
1.0000 - val_loss: 0.0819 - val_accuracy: 0.9375
Epoch 142/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0307 - accuracy:
1.0000 - val_loss: 0.0784 - val_accuracy: 0.9375
Epoch 143/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0304 - accuracy:
1.0000 - val_loss: 0.0763 - val_accuracy: 0.9375
Epoch 144/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0298 - accuracy:
1.0000 - val_loss: 0.0738 - val_accuracy: 0.9375
Epoch 145/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0310 - accuracy:
1.0000 - val_loss: 0.0729 - val_accuracy: 0.9688
Epoch 146/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0296 - accuracy:
1.0000 - val_loss: 0.0745 - val_accuracy: 0.9375
Epoch 147/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0288 - accuracy:
1.0000 - val_loss: 0.0799 - val_accuracy: 0.9375
Epoch 148/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0294 - accuracy:
1.0000 - val_loss: 0.0784 - val_accuracy: 0.9375
Epoch 149/150
8/8 [==============================] - 0s 4ms/step - loss: 0.0297 - accuracy:
1.0000 - val_loss: 0.0740 - val_accuracy: 0.9375
Epoch 150/150
8/8 [==============================] - 0s 3ms/step - loss: 0.0291 - accuracy:
1.0000 - val_loss: 0.0716 - val_accuracy: 0.9375
```

[ ]:

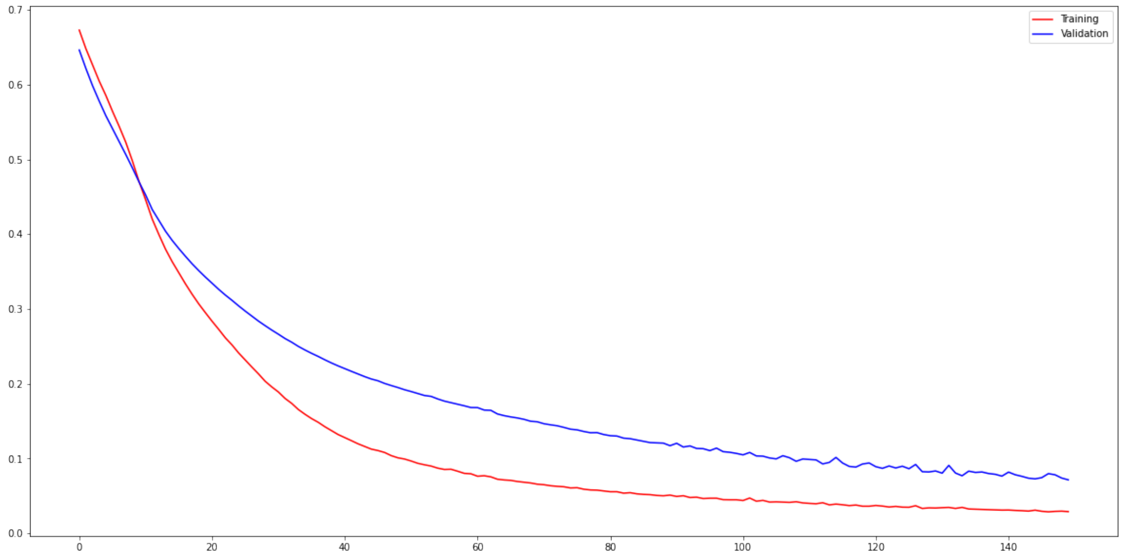
```python
_,accuracy  = happiness_model.evaluate(x_train,y_train)
print("Acuracy: %2f " % (accuracy*100))
```

```
4/4 [==============================] - 0s 2ms/step - loss: 0.0285 - accuracy:
1.0000
Acuracy: 100.000000
```

[ ]:

```python
plt.figure(figsize=(20,10))

plt.plot(history.history['loss'], c="r", label ="Training")
plt.plot(history.history['val_loss'], c="b", label ="Validation")

plt.legend()
```

```
[ ]: <matplotlib.legend.Legend at 0x20e5f29a970>
```



We choose the Logistic Regression method; this method had a high accuracy of 93%. The model predicted 24 negative values correctly and 0 false negative values, this means that the model predicted a 24 correctly when the result was a 0. On the positive side of the model, we chose 2 false positives and 6 true positives that means the model predicted only two times wrong a 1 when it was a 0, on the other side it correctly predicted 6 times a 1 when it was 1. This model has a high accuracy so it can be used to make predictions.

1.10 Analysis

1.11 PART THREE Regression Analysis

```
[ ]: ## Target ##
     y = df_scaled["Happiness Score"]
     y.head()
```

```
[ ]: 0    1.000000
     1    0.994524
     2    0.987363
     3    0.986310
     4    0.966302
     Name: Happiness Score, dtype: float64
```

```
[ ]: ## Rest of Variables ##
     x = df_scaled[column_names.drop(["Happiness Score"])]
     x.head()
```

```
[ ]:      Region  Happiness Rank  Standard Error  Economy (GDP per Capita)  \
     0  1.000000        0.000000        0.131954                  0.826132
     1  1.000000        0.006369        0.256311                  0.770412
     2  1.000000        0.012739        0.124947                  0.784113
     3  1.000000        0.019108        0.171549                  0.863099
     4  0.555556        0.025478        0.143943                  0.784592

          Family  Health (Life Expectancy)   Freedom  \
     0  0.962403                  0.918244  0.993789
     1  1.000000                  0.924496  0.938841
     2  0.970297                  0.853099  0.969615
     3  0.949167                  0.863409  1.000000
     4  0.943219                  0.883326  0.945112

        Trust (Government Corruption)  Generosity  Dystopia Residual  ClassHapiness
     0                       0.760595    0.372895           0.668630            1.0
     1                       0.256292    0.548198           0.725030            1.0
     2                       0.876175    0.428947           0.660889            1.0
     3                       0.661394    0.435983           0.652724            1.0
     4                       0.597144    0.575602           0.648584            1.0
```

1.12 Separating the data set

```
[ ]: #Separate train and test data
     x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.
      ↪20,random_state=0)
```

```
[ ]: print("Size of the full data set: ",x.shape)
     print("Size of the training data set: ",x_train.shape)
     print("Size of the test data set: ",x_test.shape)
```

```
Size of the full data set:  (158, 11)
Size of the training data set:  (126, 11)
Size of the test data set:  (32, 11)
```

1.13 Graph

```
[ ]: df_prueba = pd.read_csv("hapiness.csv")
     df_prueba.columns
```

```
[ ]: Index(['Country', 'Region', 'Happiness Rank', 'Happiness Score',
            'Standard Error', 'Economy (GDP per Capita)', 'Family',
            'Health (Life Expectancy)', 'Freedom', 'Trust (Government Corruption)',
            'Generosity', 'Dystopia Residual', 'ClassHapiness'],
           dtype='object')
```

```
[ ]: countries = df_prueba["Country"]
     countries.head()
```

```
[ ]: 0    Switzerland
     1        Iceland
     2        Denmark
     3         Norway
     4         Canada
     Name: Country, dtype: object
```

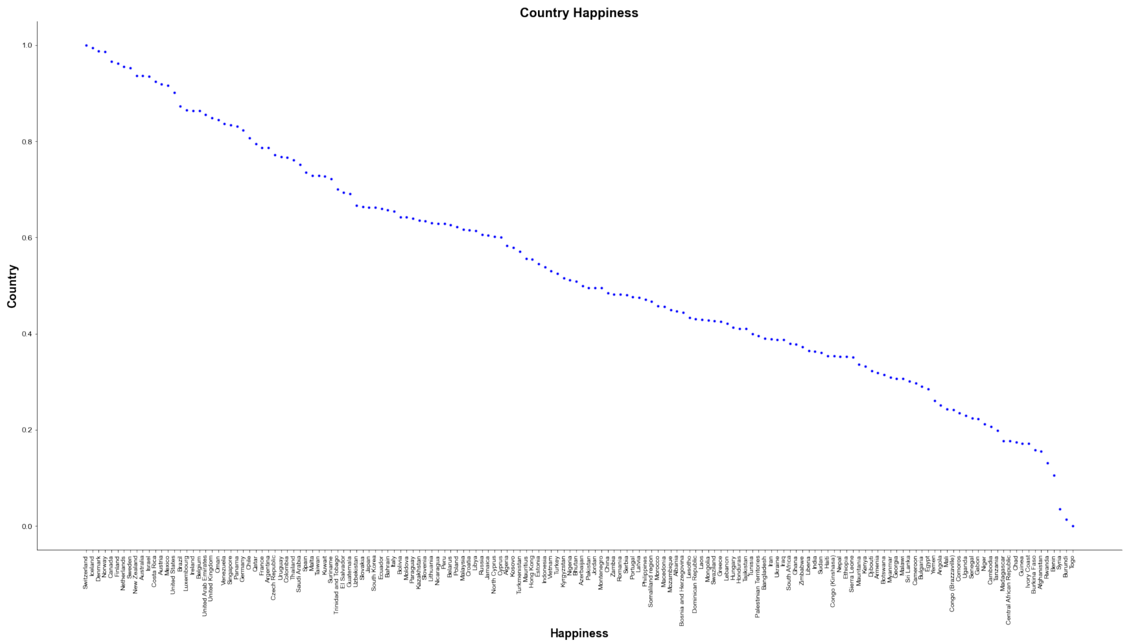
```
[ ]: happiness_real = df_scaled["Happiness Score"]
     happiness_real.head()
```

```
[ ]: 0    1.000000
     1    0.994524
     2    0.987363
     3    0.986310
     4    0.966302
     Name: Happiness Score, dtype: float64
```

```
[ ]: df_country_score  = pd.DataFrame()

     df_country_score["Country"] = countries
     df_country_score["Real Score"] = happiness_real

     df_country_score.head()
```

```
[ ]:        Country  Real Score
     0  Switzerland    1.000000
     1      Iceland    0.994524
     2      Denmark    0.987363
     3       Norway    0.986310
     4       Canada    0.966302
```

```
[ ]: ## Graph ##
     plt.figure(figsize=(30,15))
     plt.title("Country Happiness",**fontT)
     plt.xlabel("Happiness",**fontL)
     plt.ylabel("Country",**fontL)
     plt.xticks(fontsize = 10 , family = "Arial",rotation=90)
     plt.yticks(fontsize = 12 , family = "Arial")
     plt.scatter(x = df_country_score["Country"],y = df_country_score["Real Score"],␣
      ↪marker=".", c = "blue" , s = 30)
     plt.gca().spines['top'].set_visible(False)
     plt.gca().spines['right'].set_visible(False)
```



1.14 Linear Regresion

```
[ ]: # importing module
     from sklearn.linear_model import LinearRegression
     # creating an object of LinearRegression class
     LR = LinearRegression()
     # fitting the training data
     LR.fit(x_train,y_train)
```

```
[ ]: LinearRegression()
```

```
[ ]: y_prediction =  LR.predict(x_test)
     y_prediction
```

```
[ ]: array([0.95294123, 0.66473014, 0.37794277, 0.35285781, 0.60592293,
            0.35134839, 0.35241877, 0.23518144, 0.38891769, 0.45662342,
            0.8313438 , 0.22425289, 0.80679266, 0.61739891, 0.49596037,
            0.43080545, 0.307012  , 0.42901746, 0.222721  , 0.62958716,
            0.6630332 , 0.76151044, 0.62165511, 0.93660224, 0.70106093,
            0.72855103, 0.42513316, 0.85558545, 0.20644062, 0.39069107,
            0.64239061, 0.633671  ])
```

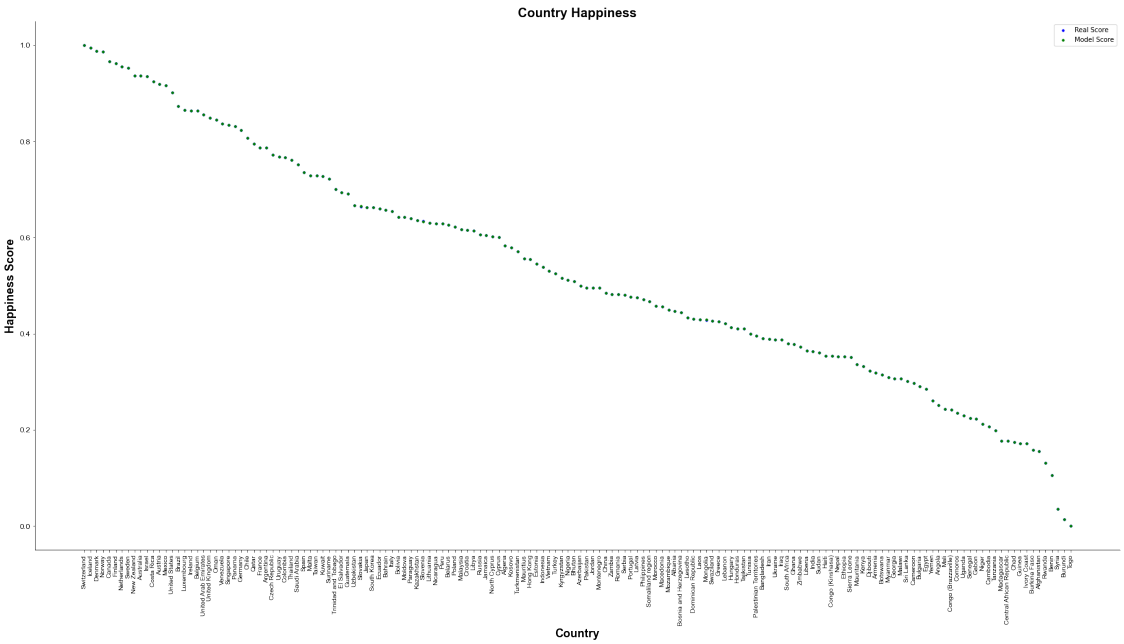
```
[ ]: # importing r2_score module
     from sklearn.metrics import r2_score
     from sklearn.metrics import mean_squared_error
     from sklearn.metrics import mean_absolute_error
```

```


score = r2_score(y_test,y_prediction)
print("R2 score is ",score)
print("Mean squared error is ",mean_squared_error(y_test,y_prediction))
print("Mean absolute error is ",mean_absolute_error(y_test,y_prediction))
print("Root mean squared error is ",np.
 ↪sqrt(mean_squared_error(y_test,y_prediction)))
```

```
R2 score is  0.9999999056811838
Mean squared error is  4.2781186931027415e-09
Mean absolute error is  5.672829785568004e-05
Root mean squared error is  6.540732904730739e-05
```

[ ]:

```
y_prediction =  LR.predict(x)



## Graph ##
plt.figure(figsize=(30,15))
plt.title("Country Happiness",**fontT)
plt.ylabel("Happiness Score",**fontL)
plt.xlabel("Country",**fontL)
plt.xticks(fontsize = 10 , family = "Arial",rotation=90)
plt.yticks(fontsize = 12 , family = "Arial")
plt.scatter(x = df_country_score["Country"],y = df_country_score["Real Score"],␣
 ↪marker=".", c = "blue" , s = 40)
plt.scatter(x = df_country_score["Country"],y = y_prediction, marker=".", c =␣
 ↪"green" , s = 40)
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.legend(['Real Score','Model Score'])
plt.show()
```



1.15 Decision Tree Regression

[ ]:
```
# importing module
from sklearn.tree import DecisionTreeRegressor

# creating an object of DecisionTreeRegressor class
DR = DecisionTreeRegressor()

# fitting the training data
DR.fit(x_train,y_train)
```

[ ]:
```
DecisionTreeRegressor()
```

[ ]:
```
y_prediction =  DR.predict(x_test)
y_prediction
```

[ ]:
```
array([0.95598147, 0.66638585, 0.36352148, 0.35341196, 0.61394271,
       0.33277169, 0.35341196, 0.24220725, 0.41048863, 0.46714406,
       0.83635215, 0.22999158, 0.83635215, 0.60151643, 0.48230834,
       0.43365628, 0.32224094, 0.42122999, 0.26074136, 0.626369  ,
       0.65480202, 0.75231676, 0.626369  , 0.93618366, 0.72788543,
       0.72935973, 0.38711036, 0.86310025, 0.22999158, 0.4100674 ,
       0.64005897, 0.626369  ])
```

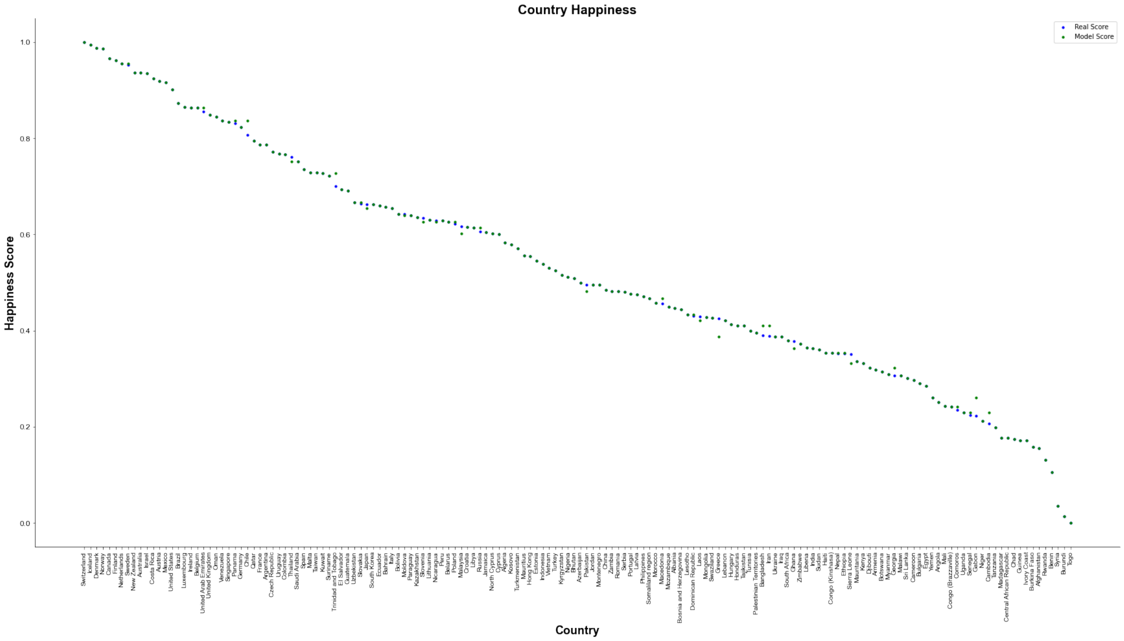
[ ]:
```
# importing r2_score module
from sklearn.metrics import r2_score
```

```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error


score = r2_score(y_test,y_prediction)
print("R2 score is ",score)
print("Mean squared error is ",mean_squared_error(y_test,y_prediction))
print("Mean absolute error is ",mean_absolute_error(y_test,y_prediction))
print("Root mean squared error is ",np.
 ↪sqrt(mean_squared_error(y_test,y_prediction)))
```

```
R2 score is  0.9946608516305063
Mean squared error is  0.00024217342166506185
Mean absolute error is  0.011596988205560241
Root mean squared error is  0.015561922171282758
```

[ ]:

```
y_prediction =  DR.predict(x)



## Graph ##
plt.figure(figsize=(30,15))
plt.title("Country Happiness",**fontT)
plt.ylabel("Happiness Score",**fontL)
plt.xlabel("Country",**fontL)
plt.xticks(fontsize = 10 , family = "Arial",rotation=90)
plt.yticks(fontsize = 12 , family = "Arial")
plt.scatter(x = df_country_score["Country"],y = df_country_score["Real Score"],␣
 ↪marker=".", c = "blue" , s = 40)
plt.scatter(x = df_country_score["Country"],y = y_prediction, marker=".", c =␣
 ↪"green" , s = 40)
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.legend(['Real Score','Model Score'])
plt.show()
```



1.16 Suppor Vector Regression

[ ]:

```
# importing module
from sklearn.svm import SVR

# creating an object of DecisionTreeRegressor class
SVRM = SVR()

# fitting the training data
SVRM.fit(x_train,y_train)
```

[ ]:

```
SVR()
```

[ ]:

```
y_prediction =  SVRM.predict(x_test)
y_prediction
```

[ ]:

```
array([0.89874015, 0.56742973, 0.28507188, 0.27440559, 0.54380783,
       0.3028409 , 0.24962955, 0.16835707, 0.30697517, 0.43981359,
       0.83639048, 0.21607574, 0.83133251, 0.53902106, 0.36056113,
       0.43876695, 0.3405847 , 0.34446699, 0.26500702, 0.54287566,
       0.58702377, 0.79577162, 0.5622068 , 0.82184065, 0.69360267,
       0.77408284, 0.42188278, 0.89397201, 0.2631628 , 0.29126685,
       0.53226329, 0.5711765 ])
```

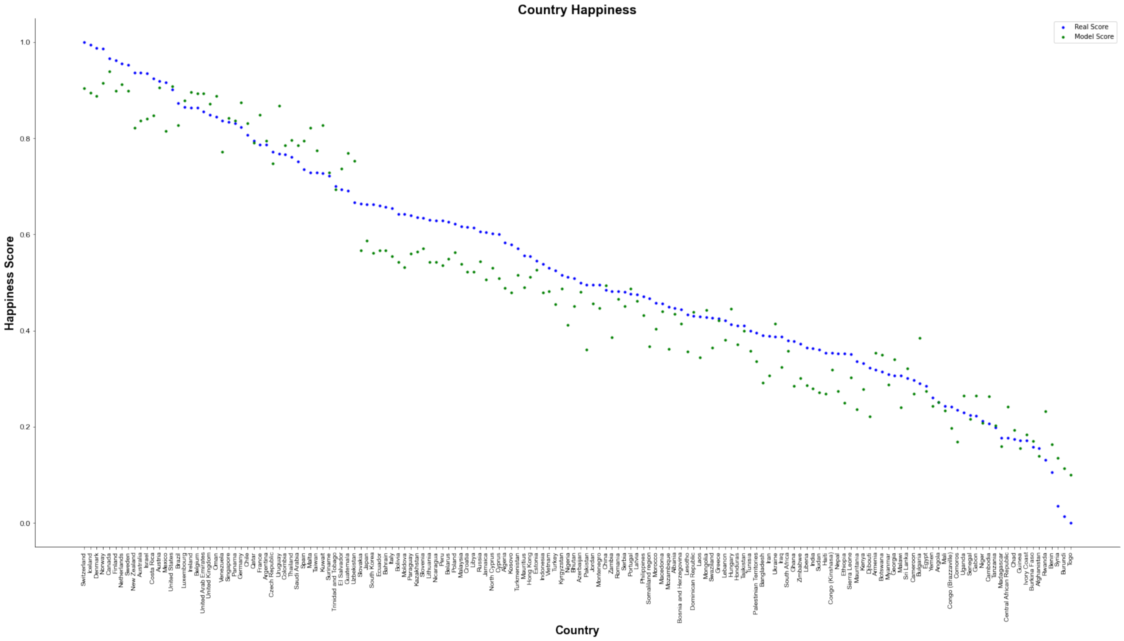
[ ]:

```
# importing r2_score module
from sklearn.metrics import r2_score
```

```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error


score = r2_score(y_test,y_prediction)
print("R2 score is ",score)
print("Mean squared error is ",mean_squared_error(y_test,y_prediction))
print("Mean absolute error is ",mean_absolute_error(y_test,y_prediction))
print("Root mean squared error is ",np.
 ↪sqrt(mean_squared_error(y_test,y_prediction)))
```

```
R2 score is  0.8934603767662862
Mean squared error is  0.0048324308140292
Mean absolute error is  0.05988316388973619
Root mean squared error is  0.06951568753906703
```

```
[ ]: y_prediction =  SVRM.predict(x)


## Graph ##
plt.figure(figsize=(30,15))
plt.title("Country Happiness",**fontT)
plt.ylabel("Happiness Score",**fontL)
plt.xlabel("Country",**fontL)
plt.xticks(fontsize = 10 , family = "Arial",rotation=90)
plt.yticks(fontsize = 12 , family = "Arial")
plt.scatter(x = df_country_score["Country"],y = df_country_score["Real Score"],␣
 ↪marker=".", c = "blue" , s = 40)
plt.scatter(x = df_country_score["Country"],y = y_prediction, marker=".", c =␣
 ↪"green" , s = 40)
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.legend(['Real Score','Model Score'])
plt.show()
```



We choose Multivariable Linear Regression, in this case this model was the best of the models from above in this case the model had a R2 of 99%, this means that the model represented almost all the data perfectly, we can see in the graph from above the model's prediction is almost exact to the real score, so that is why the high R2 of the system. The mean squared error, mean absolute error and mean squared error are really low that shows the model is a very good model, and that the model has a very low error on it.

1.17 Conclusion

While having made an analysis we can determine the different parameters of dataset, we can see that with different models we can achieve different results but with high accuracy, although with machine learning and multiple regression models we can predict how future results will work, we would need a bigger sample to create a better model. In this case we can see that with the enough information of a country we can determine the happiness level of its habitants with high accuracy levels.