INTRODUCTION TO DATA SCIENCE

Balance-Scale dataset

Prepared by-

Anirudh Joshi 17ucs023

Under the guidance of-

Prof. Sakthi Balan

Objective

This data set was generated to model psychological experimental results. Each example is classified as having the balance scale tip to the right, tip to the left, or be balanced. The attributes are the left weight, the left distance, the right weight, and the right distance.

Dataset Used: Balance-Scale dataset

- No. of Instances - 625
- No. of Attributes - 5
- No. independent variable - 4
- No. dependent variable - 1

Attribute Information

1. Class Name: 3 (L, B, R)

L represents Left-inclined, R represents Right-inclined and B represents Balanced weight (Categorical data)

2. Left-Weight: 5 (1, 2, 3, 4, 5) - LW

Represents the value of the left-weight (Ordinal data) which may take values from 1 - 5.

3. Left-Distance: 5 (1, 2, 3, 4, 5) - LD

Represents the value of the left-distance from the center of the balance (Ordinal data) which may take values from 1 - 5.

4. Right-Weight: 5 (1, 2, 3, 4, 5) - RW

Represents the value of the left-weight (Ordinal data) which may take values from 1 - 5.

5. Right-Distance: 5 (1, 2, 3, 4, 5) - RD

Represents the value of the left-distance from the center of the balance (Ordinal data) which may take values from 1 - 5.

1. Setting up libraries and dataset

1.1 Importing Libraries

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import math

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

1.2 Importing the dataset

```
#importing the dataset
dataset = pd.read_csv('balance-scale.txt')
dataset.columns = ['Balance','LW', 'LD', 'RW' , 'RD']
```

1.3 Weighing dependant attribute value

```
dataset['Balance'].replace(('L', 'R', 'B'), (-1, 1, 0), inplace = True)
```

1.4 Splitting the attributes

```
X = dataset.iloc[:,1:].values
y = dataset.iloc[:,0:1]
```

1.5 Splitting the dataset

```
#splitting dataset into training set and test set
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size = 0.15,random_state = 1)
```

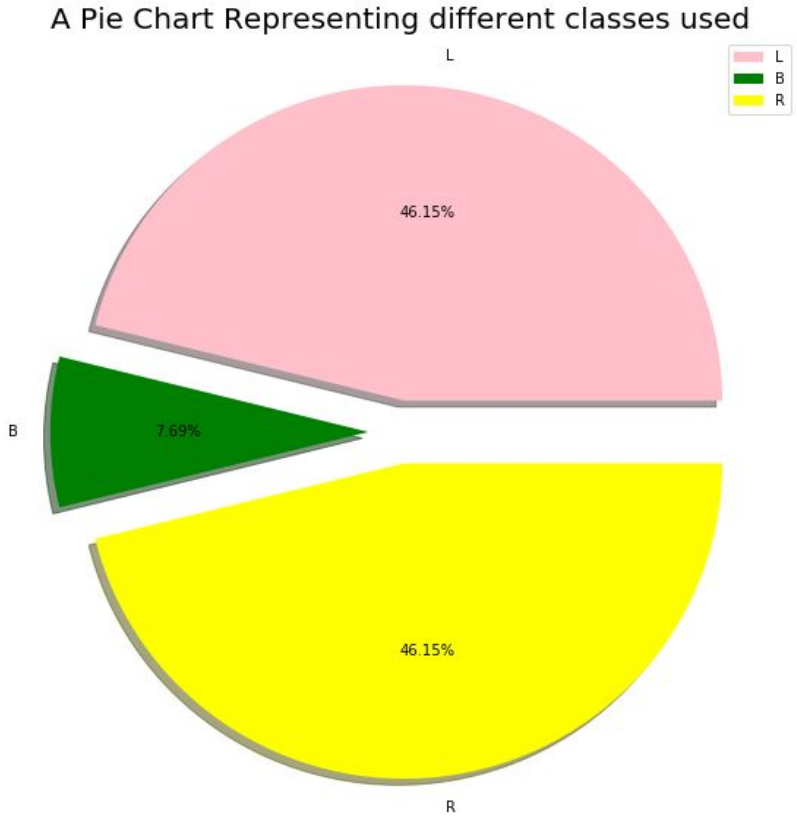
2. Data Visualization

2.1 Initial Rows

```
dataset.head()
```

	Balance	LW	LD	RW	RD
0	2	1	1	1	2
1	2	1	1	1	3
2	2	1	1	1	4
3	2	1	1	1	5
4	2	1	1	2	1

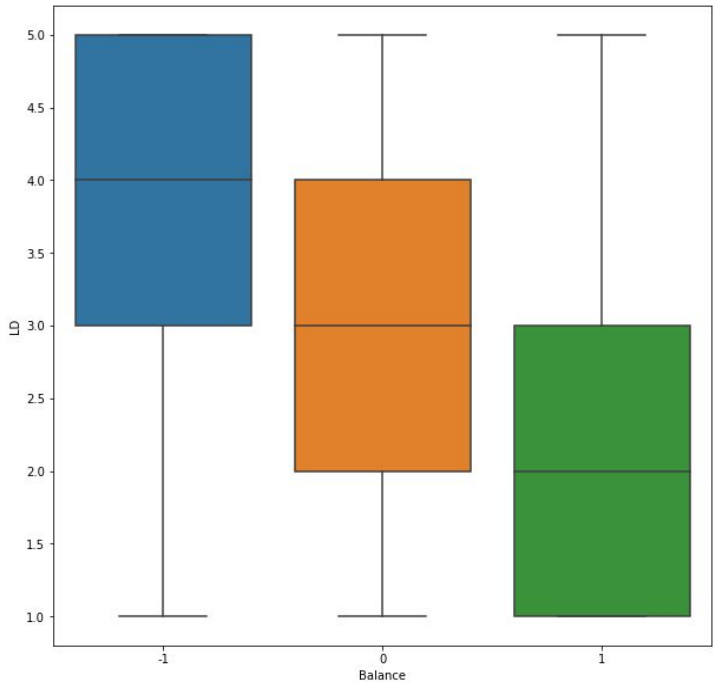
2.2 Data diversity in dependant class

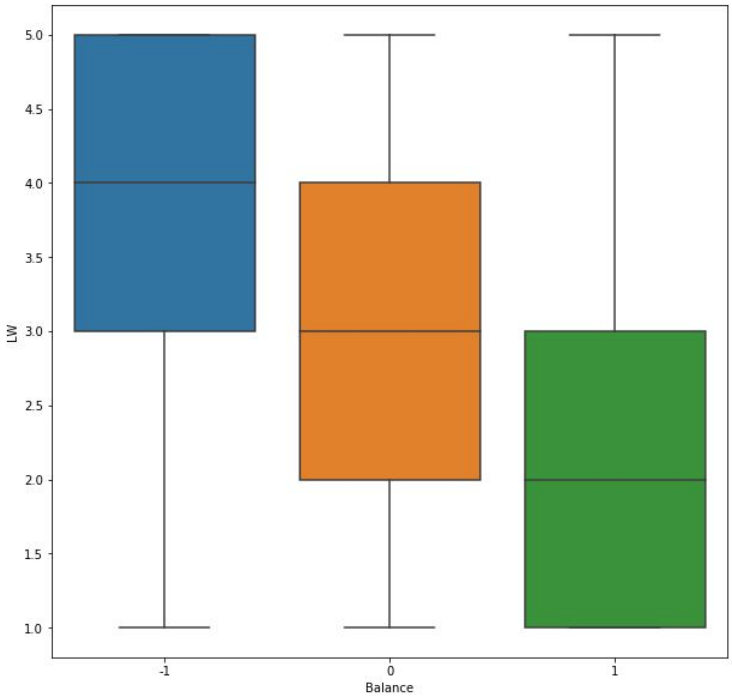


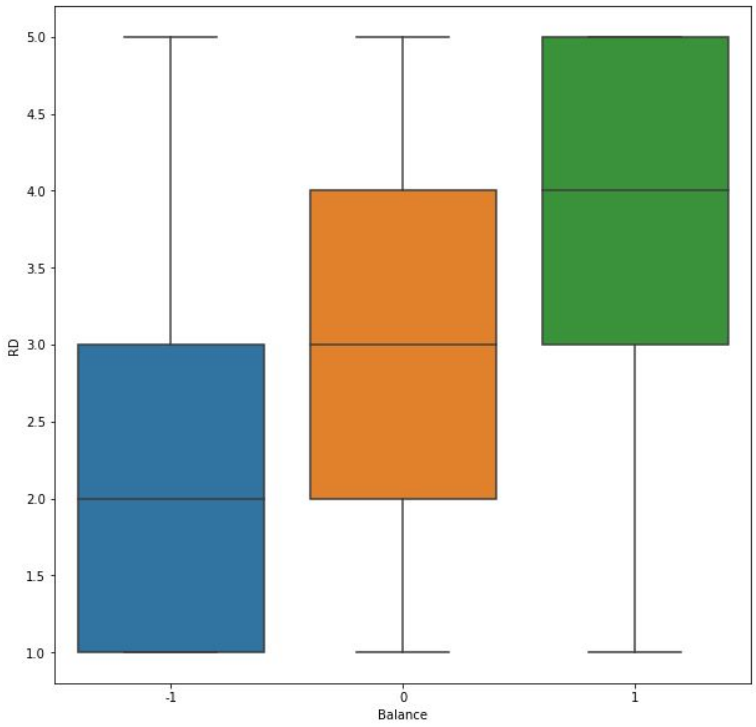
2.3 Correlation Matrix

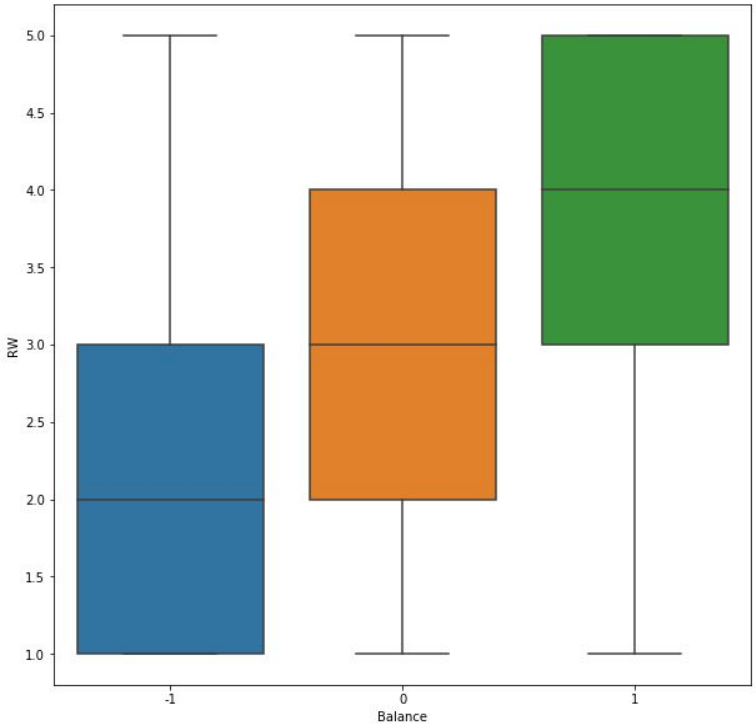
```
import seaborn
corr_matrix=dataset.corr()
fig, ax = plt.subplots(figsize=(12,12))
seaborn.heatmap(corr_matrix,annot=True,linewidths=.5, ax=ax)
```

2.4 Checking for Outliers through Box Plots






RD
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
-1
0
1
Balance


RW
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
-1
0
1
Balance

3 Data Overview

Through the pie charts representing the classes (L, B, R), it is clear that the Balanced class is underrepresented in the dataset. Hence by default, there persists a bias in our training set.

Through the correlation matrix, it is clear that the weight and the distance of either side of the scale are related inversely to each other.

Through the Box plot, it is clear that there are next to no outliers in our data. Also, there is no missing data in the dataset, nor any redundancies in the data. Also, the attributes of Weight and Distance are already scaled and weighted. Hence the data is already pre-processed and ready for future analysis through classification algorithms.

4. Initial Analysis

Through analyzing the data, and the correlation matrix, we could see a clear correlation between the weight and the distances of either side in determining the tipping of the scale. We came up with a mathematical model to predict this classification.

```
In [29]: arr = X_train.tolist()
    ...: ans = []
    ...: ans2 = y_train.values.tolist()
    ...: for i in range(len(arr)):
    ...:     l_w=arr[i][0]*arr[i][1]
    ...:     r_w=arr[i][2]*arr[i][3]
    ...:     if(l_w>r_w):
    ...:         ans.append([-1])
    ...:     elif(l_w==r_w):
    ...:         ans.append([0])
    ...:     else:
    ...:         ans.append([1])
    ...:
    ...: t=len(arr)
    ...: s=0
    ...: for i in range(len(arr)):
    ...:     if(ans2[i]==ans[i]):
    ...:         s+=1
    ...:
    ...: print('Accuracy in percent = ',(s/t)*100)
Accuracy in percent =  100.0
```

After analyzing the data we were able to generate a model that produced 100% accuracy.

5. Classification Algorithms

5.1 Random Forest Classification

```
In [24]: from sklearn.ensemble import RandomForestClassifier
    ...: from sklearn.metrics import accuracy_score as model_score
    ...: rfc = RandomForestClassifier(n_estimators=1000, n_jobs=-1, )
    ...: rfc.fit(X_train, y_train)
    ...: y_pred = rfc.predict(X_test)
    ...: print('Accuracy in percent = ',model_score(y_pred,y_test)*100)
    ...: cm=confusion_matrix(y_test,y_pred)
    ...: print(cm)
Accuracy in percent =  87.2340425531915
[[36  1  0]
 [ 2  0  5]
 [ 2  2 46]]
```

Accuracy turned out to be 87.23% after applying the Random Forest Classification.

5.2 Logistic Regression

```
In [25]: from sklearn.linear_model import LogisticRegression
    ...: from sklearn.metrics import confusion_matrix
    ...: model = LogisticRegression(C = 1)
    ...: model.fit(X_train, y_train)
    ...: y_pred = model.predict(X_test)
    ...: print('Accuracy in percent = ',model_score(y_pred,y_test)*100)
    ...: cm = confusion_matrix(y_test, y_pred)
    ...: print(cm)
Accuracy in percent =  81.91489361702128
[[37  0  0]
 [ 7  0  0]
 [10  0 40]]
```

Accuracy turned out to be 81.91% after applying Logistic Regression.

5.3 Support Vector Machine

```
In [26]: from sklearn.svm import SVC
    ...: classifier = SVC()
    ...: classifier.fit(X_train, y_train)
    ...: y_pred = classifier.predict(X_test)
    ...: print('Accuracy in percent = ',model_score(y_pred,y_test)*100)
    ...: cm=confusion_matrix(y_test,y_pred)
    ...: print(cm)
Accuracy in percent =  91.48936170212765
[[37  0  0]
 [ 5  0  2]
 [ 1  0 49]]
```

Accuracy turned out to be 91.48% after applying the Support Vector Machine.

5.4 K Nearest Neighbors Classifier

```
In [27]: from sklearn.neighbors import KNeighborsClassifier
    ...: clf = KNeighborsClassifier(n_neighbors=20)
    ...: clf.fit(X_train, y_train)
    ...: y_pred = clf.predict(X_test)
    ...: print('Accuracy in percent = ',model_score(y_pred,y_test)*100)
    ...: cm=confusion_matrix(y_test,y_pred)
    ...: print(cm)
Accuracy in percent =  91.48936170212765
[[37  0  0]
 [ 7  0  0]
 [ 1  0 49]]
```

Accuracy turned out to be 91.48% after applying K-nearest neighbors Classifier.

5.5 Decision Tree Model

```
In [28]: from sklearn.tree import DecisionTreeClassifier
    ...: clf = DecisionTreeClassifier(max_depth=6)
    ...: clf.fit(X_train, y_train)
    ...: y_pred = clf.predict(X_test)
    ...: print('Accuracy in percent = ',model_score(y_pred, y_test)*100)
    ...: cm=confusion_matrix(y_test,y_pred)
    ...: print(cm)
Accuracy in percent =  79.7872340425532
[[35  0  2]
 [ 5  0  2]
 [ 7  3 40]]
```

Accuracy turned out to be 79.78% after applying the Decision tree Classifier.

6. Conclusion

After applying 5 classification models, it was noted that our initial analysis through a mathematical model still produced the best results (100% accuracy). However, since such a perfect model was obviously impractical through Machine Learning Classification algorithms, and such drastic results can only be attributed to overfitting. Hence, we have tested our dataset on 5 different classification models. It was seen that the SVC model performed best with an accuracy of 91.48%. It can be attributed to the fact that our data is extremely polarised. Next, we saw that the KNN model performed particularly well for k>8 and best for k=20, tying in it's accuracy with the SVC model for 91.48%. An accuracy of 91.48% is much above the threshold accuracy of 70%, our initial objective. We can conclude that our model satisfactorily classifies the data.