Linear & Logistic Regression

Qusai Amer

CSCE 633

Homework #2

09/30/2020

COMPUTER SCIENCE
& ENGINEERING
TEXAS A&M UNIVERSITY

Background

The following report is about an application of linear and logistic regressions. These two methods are part of supervised machine learning that are widely used nowadays. Supervised learning takes a training data to estimate what would be the outcome for other data. In this particular example, linear and logistic regressions are used to estimate the combat point for a Pokémon with different features. The features relationship to the outcome and between themselves can be understood via the data exploration. Then Linear Regression is implemented from scratch and applied to a validation set obtained via cross-validation. The RSS, residual sum of squares, error is computed for all the folds before and after regularization. On the other side, logistic regression is implemented using scikit-learn which is a python machine learning library and it's accuracy obtained with and without regularization.

Data Exploration

Initially, all Pokémon features are structured as a data frame using python package Pandas.

```
In [2]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        from scipy.stats import pearsonr
```

```
In [178]: tdata = pd.read_csv('hw2_data.csv')
          tdata.head(5)
```

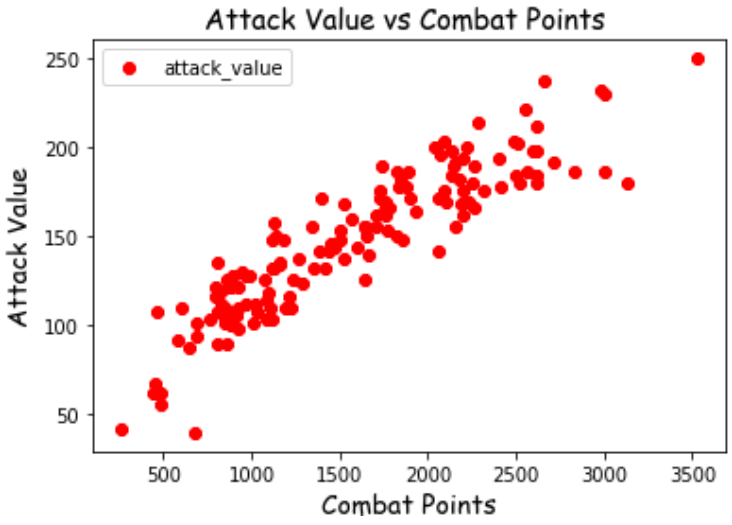
Out[178]:

	name	stamina	attack_value	defense_value	capture_rate	flee_rate	spawn_chance	primary_strength	combat_point
0	Bulbasaur	90	126	126	0.16	0.10	69.0	Grass	1079
1	Ivysaur	120	156	158	0.08	0.07	4.2	Grass	1643
2	Venusaur	160	198	200	0.04	0.05	1.7	Grass	2598
3	Charmander	78	128	108	0.16	0.10	25.3	Fire	962
4	Charmeleon	116	160	140	0.08	0.07	1.2	Fire	1568

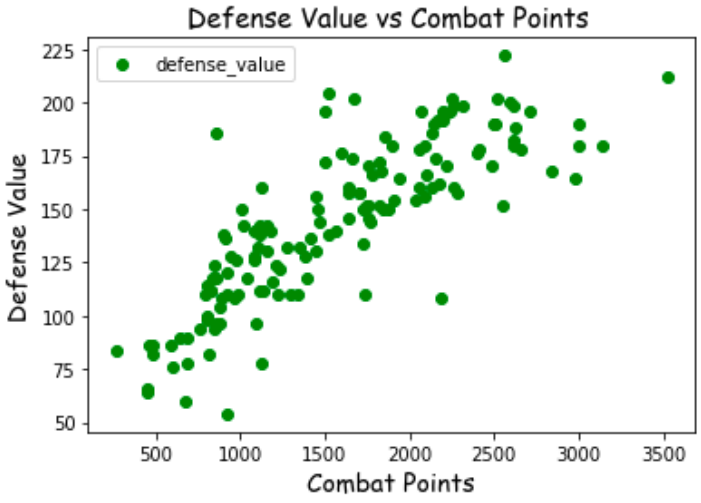
(i) **As seen from Figure(1) all the Pokémon attributes are numerical except primary strength is categorical.**

Then each feature is plotted vs the outcome which is combat point in a scatter plot. For each case the a Pearson's correlation is calculated as well which shows the similarity in the linearity between each feature and the outcome.

```
Pearsons correlation: 0.908
```




```
Pearsons correlation: 0.826
```

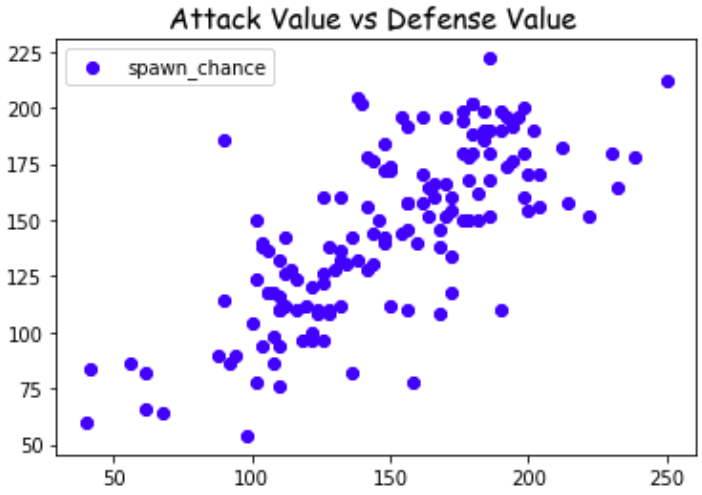


(ii) The scatter plots and the Pearson's coefficients indicate that the most predictive attributes of the Combat Point are **Attack Value and Defense Value**.

Note that code and other scatter plots are attached at the annex below.

Similarly, scatter plots and Pearson's coefficient is found between features.

```
Pearsons correlation: 0.737
```



(iii) **The most correlated variables to each other are the Attack Value and the Defense Value with a Pearson's Coefficient of 0.737. Then comes the Stamina with both the Attack Value and Defense Value with a Pearson's Coefficient of 0.303.** Note that code and other scatter plots are attached at the annex below.

Preprocessing of Categorical Variables

It's vital before implementing linear regression to represent all features numerically. Thus the only categorical feature, primary strength, is converted to numerical by one-hot encoding method. For each primary strength a column is added then if the data sample belongs to a certain column it would get a value of 1 and 0 if not; this can further understood by looking at the table below.

(iv) The following table shows the result of the one hot encoding performed.

capture_rate	flee_rate	spawn_chance	Grass	Fire	Water	...	Electric	Dragon	Rock	Poison	Ground	Fighting	Ghost	Fairy	Psychic	Ice
0.16	0.10	69.00	1	0	0	...	0	0	0	0	0	0	0	0	0	0
0.08	0.07	4.20	1	0	0	...	0	0	0	0	0	0	0	0	0	0
0.04	0.05	1.70	1	0	0	...	0	0	0	0	0	0	0	0	0	0
0.16	0.10	25.30	0	1	0	...	0	0	0	0	0	0	0	0	0	0
0.08	0.07	1.20	0	1	0	...	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
0.16	0.09	1.80	0	0	0	...	0	0	1	0	0	0	0	0	0	0
0.16	0.09	1.60	0	0	0	...	0	0	0	0	0	0	0	0	0	0
0.32	0.09	30.00	0	0	0	...	0	1	0	0	0	0	0	0	0	0
0.08	0.06	2.00	0	0	0	...	0	1	0	0	0	0	0	0	0	0
0.04	0.05	0.11	0	0	0	...	0	1	0	0	0	0	0	0	0	0

Note that code and other scatter plots are attached at the annex below.

(v) Implementing Linear Regression

The objective of LR is to predict the combat point for each Pokémon. The following steps are followed in order to implement the linear regression.

1- Names , categorical variables, and outcome are dropped from the data.
2- A Bias column is added to the front.

```
#Keep only the numeriacal representation of the features
tdata2 = tdata.drop(['name','primary_strength','combat_point'],axis =1)
#Add the bais term as the first column
tdata2['Bais Term'] = 1
cols = list(tdata2.columns.values)
cols.pop(cols.index('Bais Term'))
tdata2 = tdata2[['Bais Term'] + cols]
tdata2.head(100)
```

3- Note that the new parameters number is 22 including 15 via the one-hot encoding, six continuous features, and the bias term.

4- The OLS, ordinary least squares, solution is the found using this equation.

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

```
#Find the ordinary least squares solution
X = tdata2.to_numpy()
y = tdata.combat_point
#yn = y.to_numpy()

w = np.linalg.pinv(X.transpose()@X) @ (X.transpose()@tdata.combat_point)
```

5- The LR model is implemented using the following equations and stored at the variable predict.

Model: $f : \mathbf{x} \rightarrow y \ , \quad f(\mathbf{x}) = w_0 + \sum_{n=1}^{D} w_d x_d = \mathbf{w}^T\mathbf{x}$

6- Cross Validation is implemented using k = 5. The for loop below takes the first fold as a test data and use other folds as its training data and iterate over all five folds. Then the square root of RSS of each fold is computed. Note that the data is converted from being dataframe to array for easier linear algebra operations using asarray().

```
import math

random_index = np.random.permutation(len(X))
for i in range(0,5):
    Xs = []
    Xt = []
    ys = []
    yt = []
    for j in range(0,len(X)):
        if (j%5 ==i):
            Xs.append(X[random_index[j]])
            ys.append(y[random_index[j]])
        else:
            Xt.append(X[random_index[j]])
            yt.append(y[random_index[j]])
    Xs = np.asarray(Xs)
    Xt = np.asarray(Xt)
    ys = np.asarray(ys)
    yt = np.asarray(yt)
    w_fold = np.linalg.pinv(Xt.transpose()@Xt) @ (Xt.transpose()@yt)
    predict = Xt@w_fold.transpose()
    #print(yt)
    RSSList = 0
    for z in range(0,len(Xt)):
        RSS = (yt[z] - predict[z])*(yt[z] - predict[z])
        RSSList += RSS
    #print(RSSt)
    RSS_fold = math.sqrt(RSSList)
    print(RSS_fold)
```

The average square root of the RSS values for all fold is the following: 1138.8.

(vi) LR Regularization

Repeated the same steps but this time a regularization term is included in the w* calculation. Note all the features 22 are used here. Also note that the solution is considered a closed-form solution. ***λ is added to penalize the model and prevent overfitting which ultimately reduces the model weights.***

$$\text{Linear: } \mathbf{w}^* = (\mathbf{X}^T\mathbf{X} + \lambda \mathbf{I}_{D\times D})^{-1}\mathbf{X}^T\mathbf{y}$$

```
#print(np.linalg.pinv(Xt.transpose()@Xt))
w_foldr = np.linalg.pinv(Xt.transpose()@Xt + r_term*np.identity(22)) @ (Xt.transpose()@yt)
```

Many $\boldsymbol{\lambda}$ values are tested between 0, which gave same results to the case without regularization, and +infinity which made the weights equal to zero. The following tble shows how the tunning process of $\boldsymbol{\lambda}$.

$\boldsymbol{\lambda}$	Average sqr(RSS_fold)	w*
0	1138.8	[-1245 4. 11. 5. -53. 316. 0. -97. -91 -61. 4. -114 -167. 19. -128. -54. -27. -55 -228. -3. -244. 0]
0.3	1194	[-1090 3 11 5 -133 83 0 -70 -102 -34 -17 -125 -160 -4 -81 -32 -60. -91 -184 5 -136 0]
0.5	1260	[-1005. 4. 10. 5. -262. 63. 0. -83. -83. -42. -28. -88. -150. 74. -87. -27. -77. -68. -120. -2. -148. -77.]
0.9	1300	[-767. 5. 9. 3. -299. -20. 0. -31. 3. -29. -40. -110. -64. 45. -44. -45. -82. -98. -98. -58. -57. -58.]
1.5	1330	[-731 5 8 4 -318 -40 0 -50 -16 -26 -58 -123 -50 88 -55 -67 -88 -51 -79 -49 -65 -41]

An optimal value was found to be 0.3 that minimize the model's weights but in the same time doesn't restrict the emphasis given to minimizing the RSS.

(vii) **(Bonus)**

The linear regression is now implemented with different features compensations.

1- Using only two features

Using this time only two features, Attack Value and Defense Value. Those two features are chosen because of the fact that they have the closest linearity based on the previous visualization analysis. All the LR implementation steps are repeated but this time w has only two parameter.

```
#Find the ordinary least squares solution
tdata3 = tdata2[['attack_value','defense_value']]

XX = tdata3.to_numpy()
```

Although only two features are used note that the RSS values are relatively low. This is mainly because, the Attack and Defense Values are the most representative features out of the 22 according to the data exploration.

The average square root RSS of all fold is **3618**. It makes sense that it's higher than the case with the 22 features as more data generates a better model.

2- Using four features

This time four features are used to implement the linear regression.

```
#Using four features
tdata3 = tdata2[['stamina','attack_value','defense_value', 'capture_rate']]

XX = tdata3.to_numpy()
```

The average square root RSS of all fold is **1995**. Now using four features is still not as good as the main case with 22 features but still provides an appropriate representation, model.

RSS for each fold values are found in the annex below.

(viii) Logistic Regression

Scikit Learn is used to impalement the logistic regression. Initially the output values are converted to a binary array that has a value of 1 it's value is larger than the mean, 1577, and 0 if the value is lower than the mean.

```
ym = y.mean()
yy = y.to_numpy()
yb = np.where(yy>ym,1,0)
print(yy)
yb
```

```
[2057  647 1668  923  863 1643 1097 1015 1176 1190  460 1526  797 1849
 2319 1079 1736 1232 1082 2145  884 1877 2620 1160 1125 2281  481 1465
  894  882 2502  845 1505 1779  830 2403  679 2708 2976 1140 1643 1657
 2662 1450  926 2414 2560 2548 1013  810  861 2215 2236 1823 2161  924
 1117  911 1527 1454 2197  488 1857 2598 2612 1594 1760 2088 2134 1350
 2836 1209 1414 1114  604 1759 2259  855 1127 2523  691 1112 1826 1107
 2199  944  761 1903 1272  897  804 1568 1638  990 2492 1390 1227 2106
 1728  264 1293  800 2621 1084 3005 2510 1836 1701  972 1893 2192 1725
 2203 2067 3525 3135 2042 1758  585  801  446  828  962 1382 2093 1752
  684 2058 2137 1503  837 1344  849 1703 2266 2249  452 2615 3002 1156
 1935 1773 1443 2180 1036 2155]

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

Then Scikit Learn is used to split the data into test and train subsets. Note that the split fraction Is 20% subtest set and 80% train subset. Also note that the outcome, combat point, is similarly split.

```
X_train,X_test,y_train,y_test=train_test_split(X,yb,test_size=0.20, random_state=0)
```

In this case the logistic regression does not include regularization. Thus, penalty is specified as 'none' :

```
#Logitic Regression using Sklearn

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split


X_train,X_test,y_train,y_test=train_test_split(X,yb,test_size=0.20, random_state=0)

model = LogisticRegression(solver='lbfgs',tol=0.0001,penalty='none', max_iter=1000)
model.fit(X_train, y_train)
pred = model.predict(X_test)
score = model.score(X_test, y_test)
print(round(score,2))
```

```
0.97
```

Accuracy = 0.97

Note that the accuracy is high because there is a chance that the model is overfitting the data. Thus, in the next section, regularization will be used to optimize my model.

(xi) Logistic Regression with Regularization and Cross Validation

Logistic Regression is also implemented using Scikit Learn python package. This time, regularization, l2norm, is used as well as cross validation. The **cross validation** used here is similar to the one used in the previous linear regression but this time the model is found using the sklearn library.

```
from sklearn.linear_model import LogisticRegression

X_train,X_test,y_train,y_test=train_test_split(X,yb,test_size=0.20,random_state=0)

random_index = np.random.permutation(len(X_train))

for i in range(0,5):
    XXs = []
    XXt = []
    ys = []
    yt = []
    r_term = 0

    for j in range(0,len(X_train)):
        if (j%5 ==i):
            XXs.append(X_train[random_index[j]])
            ys.append(y_train[random_index[j]])
        else:
            XXt.append(X_train[random_index[j]])
            yt.append(y_train[random_index[j]])
    XXs = np.asarray(XXs)
    XXt = np.asarray(XXt)
    ys = np.asarray(ys)
    yt = np.asarray(yt)

    #print(np.linalg.pinv(Xt.transpose()@Xt))


    model = LogisticRegression(solver='saga',tol=0.0001,penalty='l2',C=0.05, max_iter=10000)
    #C = Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller value
    model.fit(XXt, yt)
    pred = model.predict(XXs)
    score = model.score(XXs, ys)
    print(round(score,3))
```

Note that the C inside LogisticRegression strong is the regularization and its parameter.

λ	Average Accuracy (%) of all folds
0.5	95%
1	88&
2	82%

The regularization parameter is tunned in the cross validation to be 1.25 which equals C=0.8. Then it is used for **the test data.**

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split


X_train,X_test,y_train,y_test=train_test_split(X,yb,test_size=0.20, random_state=0)

model = LogisticRegression(solver='saga',tol=0.0001,penalty='l2',C=0.8, max_iter=1000)
model.fit(X_train, y_train)
pred = model.predict(X_test)
score = model.score(X_test, y_test)
print(round(score,2))
```

```
0.93
```

Accuracy = 0.93

Note that the accuracy here is lower than the case without regularization which makes sense as the model now is more flexible and dose not overfit.