StudentAdmissions

September 28, 2018

1 Predicting Student Admissions with Neural Networks

In this notebook, we predict student admissions to graduate school at UCLA based on three pieces of data: - GRE Scores (Test) - GPA Scores (Grades) - Class rank (1-4)

The dataset originally came from here: http://www.ats.ucla.edu/

1.1 Loading the data

To load the data and format it nicely, we will use two very useful packages called Pandas and Numpy. You can read on the documentation here: - https://pandas.pydata.org/pandas-docs/stable/ - https://docs.scipy.org/

In [6]:
```
# Importing pandas and numpy
import pandas as pd
import numpy as np

# Reading the csv file into a pandas DataFrame
data = pd.read_csv('student_data.csv')

# Printing out the first 10 rows of our data
data[:10]
```

Out[6]:
```
   admit  gre   gpa  rank
0      0  380  3.61     3
1      1  660  3.67     3
2      1  800  4.00     1
3      1  640  3.19     4
4      0  520  2.93     4
5      1  760  3.00     2
6      1  560  2.98     1
7      0  400  3.08     2
8      1  540  3.39     3
9      0  700  3.92     2
```

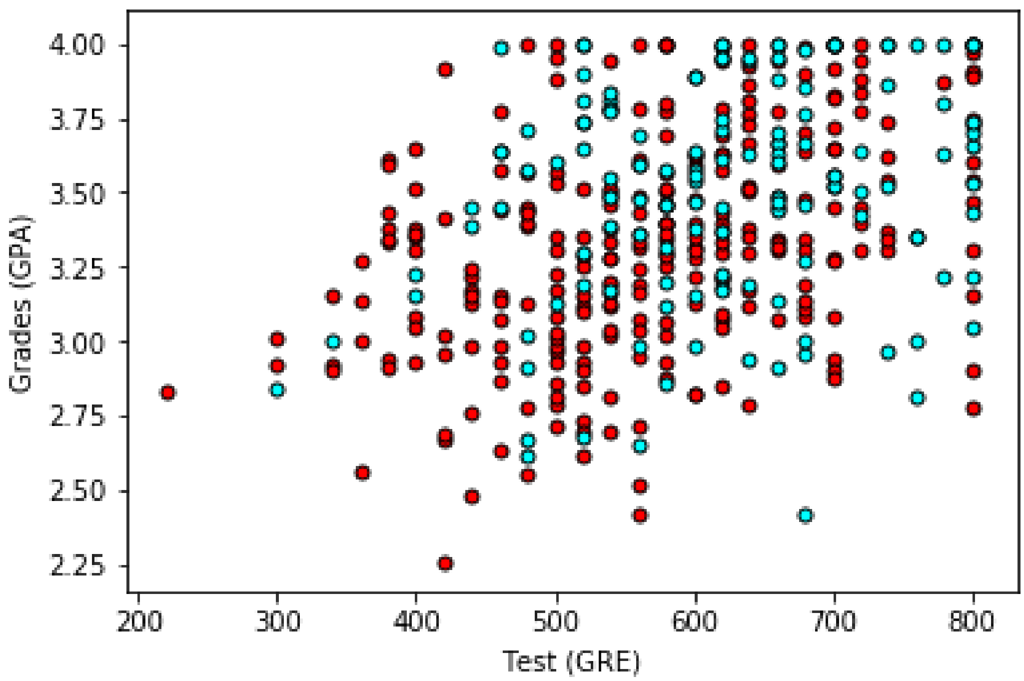
1.2 Plotting the data

First let's make a plot of our data to see how it looks. In order to have a 2D plot, let's ingore the rank.

```
In [7]: # Importing matplotlib
        import matplotlib.pyplot as plt

        # Function to help us plot
        def plot_points(data):
            X = np.array(data[["gre","gpa"]])
            y = np.array(data["admit"])
            admitted = X[np.argwhere(y==1)]
            rejected = X[np.argwhere(y==0)]
            plt.scatter([s[0][0] for s in rejected], [s[0][1] for s in rejected], s = 25, color
            plt.scatter([s[0][0] for s in admitted], [s[0][1] for s in admitted], s = 25, color
            plt.xlabel('Test (GRE)')
            plt.ylabel('Grades (GPA)')

        # Plotting the points
        plot_points(data)
        plt.show()
```

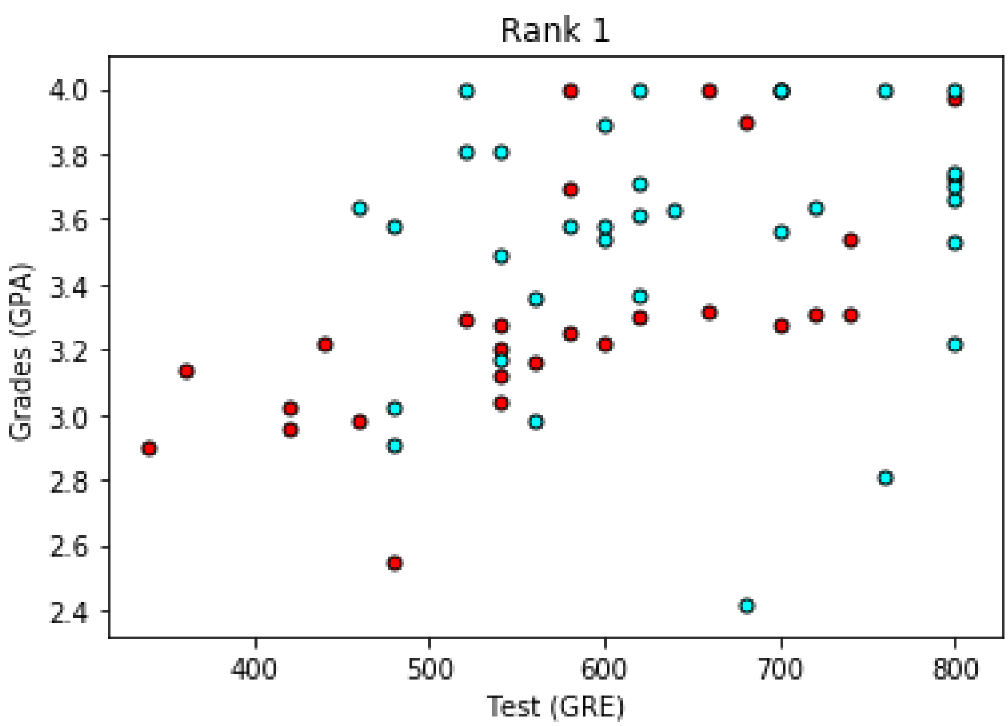


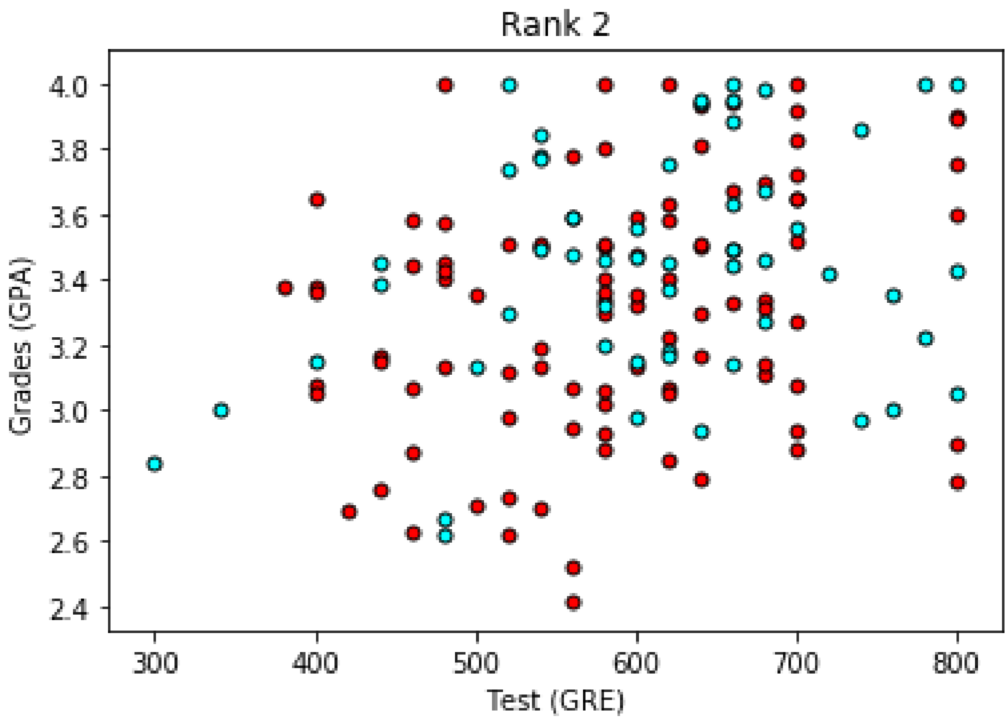
Roughly, it looks like the students with high scores in the grades and test passed, while the ones with low scores didn't, but the data is not as nicely separable as we hoped it would. Maybe it would help to take the rank into account? Let's make 4 plots, each one for each rank.

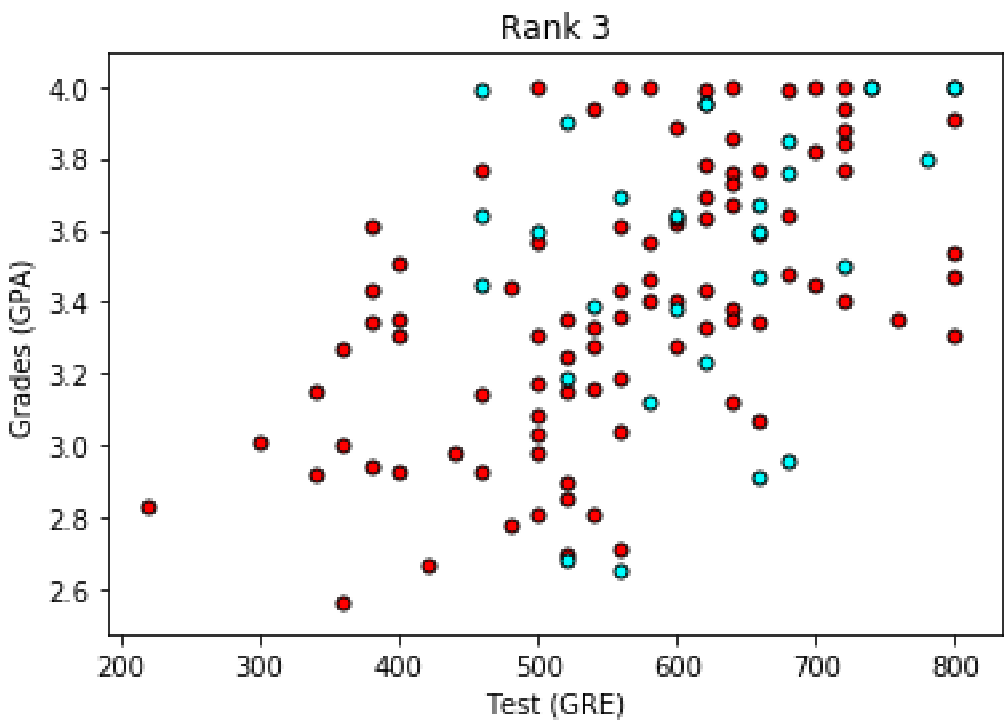
```
In [8]: # Separating the ranks
        data_rank1 = data[data["rank"]==1]
        data_rank2 = data[data["rank"]==2]
        data_rank3 = data[data["rank"]==3]
```

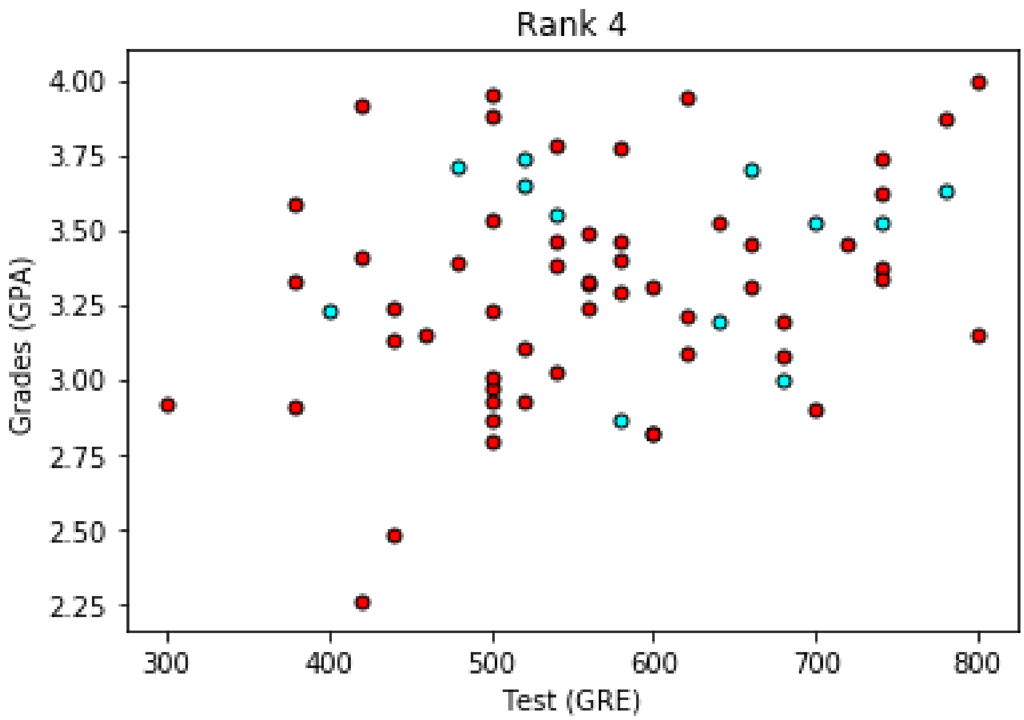
```python
data_rank4 = data[data["rank"]==4]

# Plotting the graphs
plot_points(data_rank1)
plt.title("Rank 1")
plt.show()
plot_points(data_rank2)
plt.title("Rank 2")
plt.show()
plot_points(data_rank3)
plt.title("Rank 3")
plt.show()
plot_points(data_rank4)
plt.title("Rank 4")
plt.show()
```




Rank 2
Grades (GPA)
Test (GRE)
4.0
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
300
400
500
600
700
800


Rank 3
Grades (GPA)
Test (GRE)
4.0
3.8
3.6
3.4
3.2
3.0
2.8
2.6
200
300
400
500
600
700
800



This looks more promising, as it seems that the lower the rank, the higher the acceptance rate. Let's use the rank as one of our inputs. In order to do this, we should one-hot encode it.

1.3 One-hot encoding the rank

Use the `get_dummies` function in Pandas in order to one-hot encode the data.

In [10]:
```
# TODO:  Make dummy variables for rank
one_hot_data = pd.concat([data, pd.get_dummies(data['rank'], prefix='rank')], axis=1)

# TODO: Drop the previous rank column
one_hot_data = one_hot_data.drop('rank', axis=1)

# Print the first 10 rows of our data
one_hot_data[:10]
```

Out[10]:

	admit	gre	gpa	rank_1	rank_2	rank_3	rank_4
0	0	380	3.61	0	0	1	0
1	1	660	3.67	0	0	1	0
2	1	800	4.00	1	0	0	0
3	1	640	3.19	0	0	0	1
4	0	520	2.93	0	0	0	1
5	1	760	3.00	0	1	0	0
6	1	560	2.98	1	0	0	0
7	0	400	3.08	0	1	0	0

```
8       1  540   3.39         0         0         1         0
9       0  700   3.92         0         1         0         0
```

1.4 Scaling the data

The next step is to scale the data. We notice that the range for grades is 1.0-4.0, whereas the range for test scores is roughly 200-800, which is much larger. This means our data is skewed, and that makes it hard for a neural network to handle. Let's fit our two features into a range of 0-1, by dividing the grades by 4.0, and the test score by 800.

In [13]:
```python
# Making a copy of our data
processed_data = one_hot_data[:]

# TODO: Scale the columns
processed_data['gre'] = processed_data['gre']/800
processed_data['gpa'] = processed_data['gpa']/4.0
processed_data[:10]

# Printing the first 10 rows of our procesed data
processed_data[:10]
```

Out[13]:

	admit	gre	gpa	rank_1	rank_2	rank_3	rank_4
0	0	0.475	0.9025	0	0	1	0
1	1	0.825	0.9175	0	0	1	0
2	1	1.000	1.0000	1	0	0	0
3	1	0.800	0.7975	0	0	0	1
4	0	0.650	0.7325	0	0	0	1
5	1	0.950	0.7500	0	1	0	0
6	1	0.700	0.7450	1	0	0	0
7	0	0.500	0.7700	0	1	0	0
8	1	0.675	0.8475	0	0	1	0
9	0	0.875	0.9800	0	1	0	0

1.5 Splitting the data into Training and Testing

In order to test our algorithm, we'll split the data into a Training and a Testing set. The size of the testing set will be 10% of the total data.

In [14]:
```python
sample = np.random.choice(processed_data.index, size=int(len(processed_data)*0.9), repl
train_data, test_data = processed_data.iloc[sample], processed_data.drop(sample)

print("Number of training samples is", len(train_data))
print("Number of testing samples is", len(test_data))
print(train_data[:10])
print(test_data[:10])
```

```
Number of training samples is 360
Number of testing samples is 40
     admit    gre     gpa  rank_1  rank_2  rank_3  rank_4
```

```
326       0  0.850  0.8275       0       1       0       0
72        0  0.600  0.8475       0       0       0       1
327       1  0.700  0.8700       0       1       0       0
305       0  0.725  0.8650       0       0       0       1
299       0  0.900  0.8500       0       0       1       0
215       1  0.825  0.7275       0       0       1       0
13        0  0.875  0.7700       0       1       0       0
373       1  0.775  0.8425       1       0       0       0
361       1  0.675  0.8725       1       0       0       0
84        1  0.625  0.9000       0       0       1       0
     admit    gre     gpa  rank_1  rank_2  rank_3  rank_4
0        0  0.475  0.9025       0       0       1       0
10       0  1.000  1.0000       0       0       0       1
15       0  0.600  0.8600       0       0       1       0
34       0  0.450  0.7850       1       0       0       0
39       1  0.650  0.6700       0       0       1       0
41       1  0.725  0.8300       0       1       0       0
102      0  0.475  0.8325       0       0       0       1
108      0  0.500  0.7325       0       0       1       0
120      1  0.650  0.9350       0       1       0       0
134      0  0.700  0.7375       0       1       0       0
```

1.6 Splitting the data into features and targets (labels)

Now, as a final step before the training, we'll split the data into features (X) and targets (y).

In [15]:
```
features = train_data.drop('admit', axis=1)
targets = train_data['admit']
features_test = test_data.drop('admit', axis=1)
targets_test = test_data['admit']

print(features[:10])
print(targets[:10])
```

```
       gre     gpa  rank_1  rank_2  rank_3  rank_4
326  0.850  0.8275       0       1       0       0
72   0.600  0.8475       0       0       0       1
327  0.700  0.8700       0       1       0       0
305  0.725  0.8650       0       0       0       1
299  0.900  0.8500       0       0       1       0
215  0.825  0.7275       0       0       1       0
13   0.875  0.7700       0       1       0       0
373  0.775  0.8425       1       0       0       0
361  0.675  0.8725       1       0       0       0
84   0.625  0.9000       0       0       1       0
326    0
72     0
```

```
327    1
305    0
299    0
215    1
13     0
373    1
361    1
84     1
Name: admit, dtype: int64
```

1.7 Training the 2-layer Neural Network

The following function trains the 2-layer neural network. First, we'll write some helper functions.

```
In [17]: # Activation (sigmoid) function
         def sigmoid(x):
             return 1 / (1 + np.exp(-x))
         def sigmoid_prime(x):
             return sigmoid(x) * (1-sigmoid(x))
         def error_formula(y, output):
             return - y*np.log(output) - (1 - y) * np.log(1-output)
```

2 Backpropagate the error

Now it's your turn to shine. Write the error term. Remember that this is given by the equation

$$-(y - \hat{y})\sigma'(x)$$

```
In [18]: def error_term_formula(y, output):
             return (y-output) * output * (1 - output)
```

```
In [19]: # Neural Network hyperparameters
         epochs = 1000
         learnrate = 0.5

         # Training function
         def train_nn(features, targets, epochs, learnrate):

             # Use to same seed to make debugging easier
             np.random.seed(42)

             n_records, n_features = features.shape
             last_loss = None

             # Initialize weights
             weights = np.random.normal(scale=1 / n_features**.5, size=n_features)
```

```python
    for e in range(epochs):
        del_w = np.zeros(weights.shape)
        for x, y in zip(features.values, targets):
            # Loop through all records, x is the input, y is the target

            # Activation of the output unit
            #   Notice we multiply the inputs and the weights here
            #   rather than storing h as a separate variable
            output = sigmoid(np.dot(x, weights))

            # The error, the target minus the network output
            error = error_formula(y, output)

            # The error term
            #   Notice we calulate f'(h) here instead of defining a separate
            #   sigmoid_prime function. This just makes it faster because we
            #   can re-use the result of the sigmoid function stored in
            #   the output variable
            error_term = error_term_formula(y, output)

            # The gradient descent step, the error times the gradient times the inputs
            del_w += error_term * x

        # Update the weights here. The learning rate times the
        # change in weights, divided by the number of records to average
        weights += learnrate * del_w / n_records

        # Printing out the mean square error on the training set
        if e % (epochs / 10) == 0:
            out = sigmoid(np.dot(features, weights))
            loss = np.mean((out - targets) ** 2)
            print("Epoch:", e)
            if last_loss and last_loss < loss:
                print("Train loss: ", loss, "  WARNING - Loss Increasing")
            else:
                print("Train loss: ", loss)
            last_loss = loss
            print("=========")
    print("Finished training!")
    return weights

weights = train_nn(features, targets, epochs, learnrate)
```

```
Epoch: 0
Train loss:  0.27292410091
=========
Epoch: 100
Train loss:  0.210936492609
```

```
=========
Epoch: 200
Train loss:  0.208354414776
=========
Epoch: 300
Train loss:  0.207228553927
=========
Epoch: 400
Train loss:  0.206654422369
=========
Epoch: 500
Train loss:  0.206301721151
=========
Epoch: 600
Train loss:  0.206045685612
=========
Epoch: 700
Train loss:  0.205836051276
=========
Epoch: 800
Train loss:  0.205651078968
=========
Epoch: 900
Train loss:  0.205480687812
=========
Finished training!
```

2.1 Calculating the Accuracy on the Test Data

```
In [20]: # Calculate accuracy on test data
         tes_out = sigmoid(np.dot(features_test, weights))
         predictions = tes_out > 0.5
         accuracy = np.mean(predictions == targets_test)
         print("Prediction accuracy: {:.3f}".format(accuracy))
```

```
Prediction accuracy: 0.700
```