Predicting Student Admissions

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. Load and visualize the data

To load the data, we will use a very useful data package called Pandas. You can read on Pandas documentation here:

In [1]:

```
import pandas as pd
data = pd.read_csv('student_data.csv')
data.head()
```

Out[1]:

	admit	gre	gpa	rank
0	0	380.0	3.61	3.0
1	1	660.0	3.67	3.0
2	1	800.0	4.00	1.0
3	1	640.0	3.19	4.0
4	0	520.0	2.93	4.0

```
data.shape
```

Out[2]:

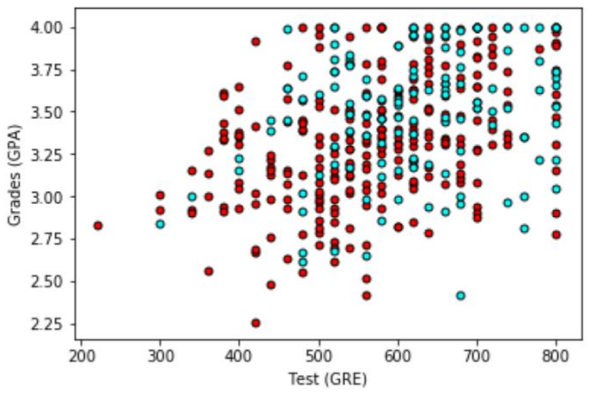
```
(400, 4)
```

Here we can see that the first column is the label y, which corresponds to acceptance/rejection. Namely, a label of 1 means the student got accepted, and a label of 0 means the student got rejected.

When we plot the data, we get the following graphs, which shows that unfortunately, the data is not as nicely separable as we'd hope:

In [3]:

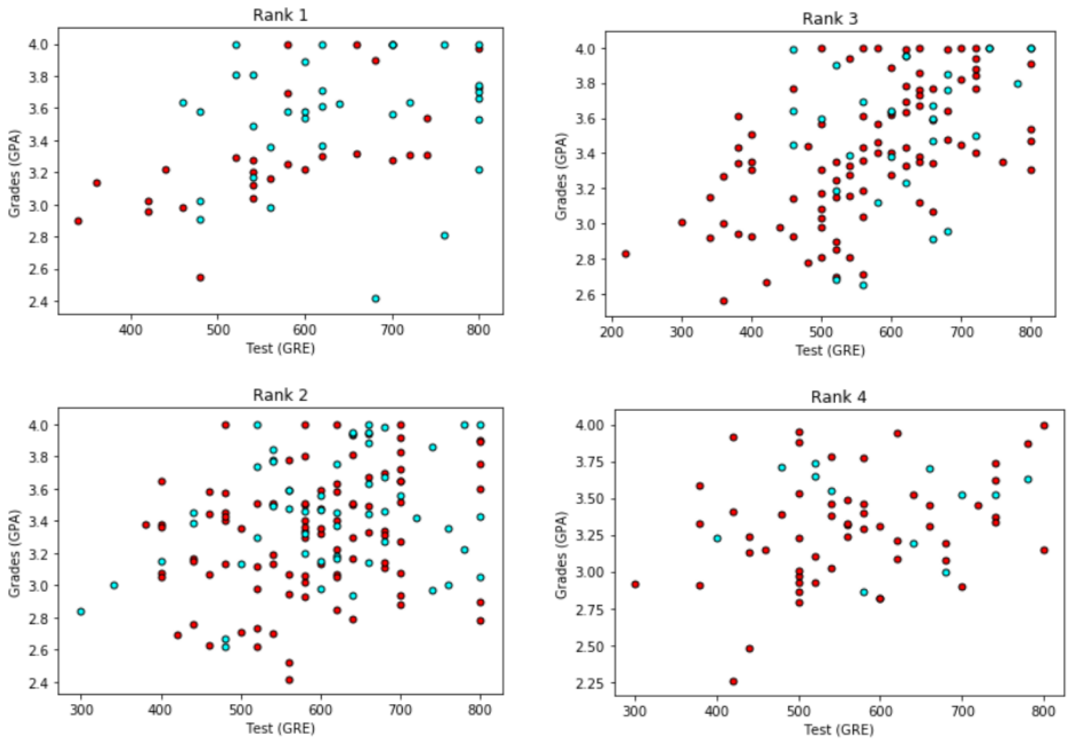
```
import matplotlib.pyplot as plt
import numpy as np
def plot_points(data):
    X = np.array(data[["gre","gpa"]])
    y = np.array(data["admit"])
    admitted = X[np.argwhere(y==1)]
    rejected = X[np.argwhere(y==0)]
    plt.scatter([s[0][0] for s in rejected],
                [s[0][1] for s in rejected],
                s = 25, color = 'red', edgecolor = 'k')
    plt.scatter([s[0][0] for s in admitted],
                [s[0][1] for s in admitted],
                s = 25, color = 'cyan', edgecolor = 'k')
    plt.xlabel('Test (GRE)')
    plt.ylabel('Grades (GPA)')
plot_points(data)
plt.show()
```



The data, based on only GRE and GPA scores, doesn't seem very separable. Maybe if we make a plot for each of the ranks, the boundaries will be more clear.

In [6]:

```
data_rank1 = data[data["rank"]==1]
data_rank2 = data[data["rank"]==2]
data_rank3 = data[data["rank"]==3]
data_rank4 = data[data["rank"]==4]
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()
```



These plots look a bit more linearly separable, although not completely. But it seems that using a multi-layer perceptron with the rank, gre, and gpa as inputs, may give us a decent solution.

2. Process the data

We'll do the following steps to clean up the data for training:

- One-hot encode the rank
- Normalize the gre and the gpa scores, so they'll be in the interval (0,1)
- Split the data into the input X, and the labels y.

In [7]:

```
import keras
from keras.utils import np_utils

# remove NaNs
data = data.fillna(0)

# One-hot encoding the rank
processed_data = pd.get_dummies(data, columns=['rank'])
```

```
Using TensorFlow backend.
```

So what we'll do is, we'll one-hot encode the rank, and our 6 input variables will be:

- Test (GPA)
- Grades (GRE)
- Rank 1
- Rank 2
- Rank 3
- Rank 4.

In [8]:

```
processed_data.head()
```

Out[8]:

	admit	gre	gpa	rank_0.0	rank_1.0	rank_2.0	rank_3.0	rank_4.0
0	0	380.0	3.61	0	0	0	1	0
1	1	660.0	3.67	0	0	0	1	0
2	1	800.0	4.00	0	1	0	0	0
3	1	640.0	3.19	0	0	0	0	1
4	0	520.0	2.93	0	0	0	0	1

The last 4 inputs will be binary variables that have a value of 1 if the student has that rank, or 0 otherwise.

So, first things first, let's notice that the **test scores** have a range of 800, while the **grades** have a range of 4. This is a huge **discrepancy**, and it will affect our training. Normally, the best thing to do is to normalize the scores so they are between 0 and 1. We can do this as follows:

In [9]:

```
# Normalizing the gre and the gpa scores to be in the interval (0,1)
processed_data["gre"] = processed_data["gre"]/800
processed_data["gpa"] = processed_data["gpa"]/4
```

Now, we split our data input into X, and the labels y, and one-hot encode the output, so it appears as two classes (accepted and not accepted).

In [10]:

```
# Splitting the data input into X, and the labels y
X = np.array(processed_data)[:,1:]
X = X.astype('float32')
y = keras.utils.to_categorical(data["admit"],2)
```

In [11]:

```
print(X)
```

```
[[ 0.47499999  0.90249997  0.         ...,  0.          1.          0.        ]
 [ 0.82499999  0.91750002  0.         ...,  0.          1.          0.        ]
 [ 1.          1.          0.         ...,  0.          0.          0.        ]
 ...,
 [ 0.57499999  0.65750003  0.         ...,  1.          0.          0.        ]
 [ 0.875       0.91250002  0.         ...,  1.          0.          0.        ]
 [ 0.75        0.97250003  0.         ...,  0.          1.          0.        ]]
```

In [12]:

```
print("we've made y as CATEGORICAL adding the opposit number as the second argument:\n")
for i in range(5):
    print(y[i])
```

```
we've made y as CATEGORICAL adding the opposit number as the second argument:

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

3. Split the data into training and testing sets

In [13]:

```
# break training set into training and validation sets
(X_train, X_test) = X[50:], X[:50]
(y_train, y_test) = y[50:], y[:50]

# print shape of training set
print('x_train shape:', X_train.shape)

# print number of training, validation, and test images
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')
```

```
x_train shape: (350, 7)
350 train samples
50 test samples
```

4. Define the model architecture

In [14]:

```
# Imports
import numpy as np
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import SGD
from keras.utils import np_utils
```

In [38]:

```
# Building the model
# Note that filling out the empty rank as "0", gave us an extra column, for "Rank 0" students.
# Thus, our input dimension is 7 instead of 6.
```

```
model = Sequential()
model.add(Dense(128, activation='relu', input_shape=(7,)))
model.add(Dropout(.2))
model.add(Dense(64, activation='relu'))
model.add(Dropout(.1))
model.add(Dense(2, activation='softmax'))
```

The error function is given by **categorical_crossentropy**, which is the one we've been using, but there are other options. There are several optimizers which you can choose from, in order to improve your training. Here we use adam, but others that are useful are **rmsprop**. They use a variety of techniques that we'll outline in the following lectures. The model summary will tell us the following:

In [39]:

```
# Compiling the model
model.compile(loss = 'categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 128)               1024
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 64)                8256
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 130
=================================================================
Total params: 9,410.0
Trainable params: 9,410.0
Non-trainable params: 0.0
_________________________________________________________________
```

5. Train the model

In [40]:

```
# Training the model
model.fit(X_train, y_train, epochs=200, batch_size=100, verbose=0)
```

Out[40]:

```
<keras.callbacks.History at 0x19ab34097f0>
```

6. Score the model

In [41]:

```
# Evaluating the model on the training and testing set
score = model.evaluate(X_train, y_train)
print("\n Training Accuracy:", score[1])
score = model.evaluate(X_test, y_test)
print("\n Testing Accuracy:", score[1])
```

```
 32/350 [=>............................] - ETA: 0s
 Training Accuracy: 0.728571430615
32/50 [==================>...........] - ETA: 0s
 Testing Accuracy: 0.660000009537
```

7. Play with parameters!

You can see that we made several decisions in our training. For instance, the number of layers, the sizes of the layers, the number of epochs, etc.

It's your turn to play with parameters! Can you improve the accuracy? The following are other suggestions for these parameters. We'll learn the definitions later in the class:

- Activation function: relu and sigmoid
- Loss function: categorical_crossentropy, mean_squared_error
- Optimizer: rmsprop, adam, ada

In [49]:

```
# Training the model with more epochs
model.fit(X_train, y_train, epochs=500, batch_size=100, verbose=0)
```

Out[49]:

```
<keras.callbacks.History at 0x19ab36cf860>
```

In [50]:

```
# Evaluating the model on the training and testing set
score = model.evaluate(X_train, y_train)
print("\n Training Accuracy:", score[1])
score = model.evaluate(X_test, y_test)
print("\n Testing Accuracy:", score[1])
```

```
 32/350 [=>...........................] - ETA: 0s
 Training Accuracy: 0.828571427209
32/50 [==================>...........] - ETA: 0s
 Testing Accuracy: 0.659999997616
```

In [63]:

```
model_1 = Sequential()
model_1.add(Dense(128, activation='sigmoid', input_shape=(7,)))
model_1.add(Dropout(.2))
model_1.add(Dense(64, activation='sigmoid'))
model_1.add(Dropout(.1))
model_1.add(Dense(2, activation='softmax'))

# Compiling the model
model_1.compile(loss = 'categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

In [64]:

```
# Training the model with more epochs
model_1.fit(X_train, y_train, epochs=200, batch_size=100, verbose=0)

# Evaluating the model on the training and testing set
score = model_1.evaluate(X_train, y_train)
print("\n Training Accuracy:", score[1])
score = model_1.evaluate(X_test, y_test)
print("\n Testing Accuracy:", score[1])
'''Sigmoid function gave more worse results'''
```

```
 32/350 [=>...........................] - ETA: 1s
 Training Accuracy: 0.699999999659
32/50 [==================>...........] - ETA: 0s
 Testing Accuracy: 0.559999997616
```

Out[64]:

```
'Sigmoid function gave more worse results\n'
```

In [62]:

```
model_2 = Sequential()
model_2.add(Dense(128, activation='relu', input_shape=(7,)))
model_2.add(Dropout(.2))
model_2.add(Dense(64, activation='relu'))
model_2.add(Dropout(.1))
model_2.add(Dense(2, activation='softmax'))

# Compiling the model
model_2.compile(loss = 'mean_squared_error', optimizer='adam',
metrics=['accuracy'])

# Training the model with more epochs
model_2.fit(X_train, y_train, epochs=200, batch_size=100, verbose=0)

# Evaluating the model on the training and testing set
score = model_2.evaluate(X_train, y_train)
print("\n Training Accuracy:", score[1])
score = model_2.evaluate(X_test, y_test)
print("\n Testing Accuracy:", score[1])
```

```
 32/350 [=>............................] - ETA: 1s
 Training Accuracy: 0.731428570747
32/50 [==================>...........] - ETA: 0s
 Testing Accuracy: 0.660000009537
```

In [70]:

```
model_3 = Sequential()
model_3.add(Dense(128, activation='relu', input_shape=(7,)))
model_3.add(Dropout(.2))
model_3.add(Dense(64, activation='relu'))
model_3.add(Dropout(.1))
model_3.add(Dense(2, activation='softmax'))

# Compiling the model
model_3.compile(loss = 'mean_squared_error', optimizer='rmsprop',
metrics=['accuracy'])

# Training the model with more epochs
model_3.fit(X_train, y_train, epochs=180, batch_size=100, verbose=0)

# Evaluating the model on the training and testing set
score = model_3.evaluate(X_train, y_train)
print("\n Training Accuracy:", score[1])
score = model_3.evaluate(X_test, y_test)
print("\n Testing Accuracy:", score[1])
'''catgorical_crossentropy gave more worse result'''
```

```
 32/350 [=>............................] - ETA: 1s
 Training Accuracy: 0.720000002044
32/50 [==================>...........] - ETA: 0s
 Testing Accuracy: 0.640000009537
```

In [104]:

```
model_4 = Sequential()
model_4.add(Dense(128, activation='relu', input_shape=(7,)))
model_4.add(Dropout(.2))
model_4.add(Dense(64, activation='relu'))
model_4.add(Dropout(.1))
model_4.add(Dense(2, activation='softmax'))

# Compiling the model
model_4.compile(loss = 'mean_squared_error', optimizer='adam',
metrics=['accuracy'])

# Training the model with more epochs
model_4.fit(X_train, y_train, epochs=395, batch_size=100, verbose=0)

# Evaluating the model on the training and testing set
score = model_4.evaluate(X_train, y_train)
print("\n Training Accuracy:", score[1])
score = model_4.evaluate(X_test, y_test)
print("\n Testing Accuracy:", score[1])
'''categorical_crossentropy gave more worse result
'''
```

```
 32/350 [=>............................] - ETA: 6s
 Training Accuracy: 0.745714287758
32/50 [==================>...........] - ETA: 0s
 Testing Accuracy: 0.679999997616
```

Out[104]:

```
'categorical_crossentropy gave more worse result'
```