Mean Normalization

In machine learning we use large amounts of data to train our models. Some machine learning algorithms may require that the data is *normalized* in order to work correctly. The idea of normalization, also known as *feature scaling*, is to ensure that all the data is on a similar scale, *i.e.* that all the data takes on a similar range of values. For example, we might have a dataset that has values between 0 and 5,000. By normalizing the data we can make the range of values be between 0 and 1.

In this lab, you will be performing a different kind of feature scaling known as *mean normalization*. Mean normalization will scale the data, but instead of making the values be between 0 and 1, it will distribute the values evenly in some small interval around zero. For example, if we have a dataset that has values between 0 and 5,000, after mean normalization the range of values will be distributed in some small range around 0, for example between -3 to 3. Because the range of values are distributed evenly around zero, this guarantees that the average (mean) of all elements will be zero. Therefore, when you perform *mean normalization* your data will not only be scaled but it will also have an average of zero.

To Do:

You will start by importing NumPy and creating a rank 2 ndarray of random integers between 0 and 5,000 (inclusive) with 1000 rows and 20 columns. This array will simulate a dataset with a wide range of values. Fill in the code below

In [1]:
```
# import NumPy into Python
import numpy as np

# Create a 1000 x 20 ndarray with random integers in the half-open inter
val [0, 5001).
X = np.random.randint(0,5001,size=(1000, 20))

# print the shape of X
print("Shape of X is: ", X.shape)
```

```
Shape of X is:  (1000, 20)
```

Now that you created the array we will mean normalize it. We will perform mean normalization using the following equation:

$$\text{Norm\_Col}_i = \frac{\text{Col}_i - \mu_i}{\sigma_i}$$

where Col_i is the ith column of X, μ_i is average of the values in the ith column of X, and σ_i is the standard deviation of the values in the ith column of X. In other words, mean normalization is performed by subtracting from each column of X the average of its values, and then by dividing by the standard deviation of its values. In the space below, you will first calculate the average and standard deviation of each column of X.

In [2]:
```
# Average of the values in each column of X
ave_cols =np.mean(X, axis=0)

# Standard Deviation of the values in each column of X
std_cols = np.std(X, axis=0)
```

If you have done the above calculations correctly, then `ave_cols` and `std_cols`, should both be vectors with shape `(20,)` since X has 20 columns. You can verify this by filling the code below:

In [3]:
```
# Print the shape of ave_cols
print("The shape of ave_cols is: ", ave_cols.shape)

# Print the shape of std_cols
print("The shape of std_cols is: ", std_cols.shape)
```

```
The shape of ave_cols is:  (20,)
The shape of std_cols is:  (20,)
```

You can now take advantage of Broadcasting to calculate the mean normalized version of X in just one line of code using the equation above. Fill in the code below

In [4]:
```
# Mean normalize X
X_norm = (X - ave_cols) / std_cols
```

If you have performed the mean normalization correctly, then the average of all the elements in X_{norm} should be close to zero, and they should be evenly distributed in some small interval around zero. You can verify this by filing the code below:

In [5]:
```
# Print the average of all the values of X_norm
# You can use either the function or a method. So, there are multiple wa
ys to solve.
print("The average of all the values of X_norm is: ")
print(np.mean(X_norm))
print(X_norm.mean())

# Print the average of the minimum value in each column of X_norm
print("The average of the minimum value in each column of X_norm is: ")
print(X_norm.min(axis = 0).mean())
print(np.mean(np.sort(X_norm, axis=0)[0]))

# Print the average of the maximum value in each column of X_norm
print("The average of the maximum value in each column of X_norm is: ")
print(np.mean(np.sort(X_norm, axis=0)[-1]))
print(X_norm.max(axis = 0).mean())
```

```
The average of all the values of X_norm is:
-3.4106051316484806e-17
-3.4106051316484806e-17
The average of the minimum value in each column of X_norm is:
-1.734579449615548
-1.734579449615548
The average of the maximum value in each column of X_norm is:
1.7193057612873717
1.7193057612873717
```

You should note that since X was created using random integers, the above values will vary.

Data Separation

After the data has been mean normalized, it is customary in machine learnig to split our dataset into three sets:

1. A Training Set
2. A Cross Validation Set
3. A Test Set

The dataset is usually divided such that the Training Set contains 60% of the data, the Cross Validation Set contains 20% of the data, and the Test Set contains 20% of the data.

In this part of the lab you will separate `X_norm` into a Training Set, Cross Validation Set, and a Test Set. Each data set will contain rows of `X_norm` chosen at random, making sure that we don't pick the same row twice. This will guarantee that all the rows of `X_norm` are chosen and randomly distributed among the three new sets.

You will start by creating a rank 1 ndarray that contains a random permutation of the row indices of `X_norm`. You can do this by using the `np.random.permutation()` function. The `np.random.permutation(N)` function creates a random permutation of integers from 0 to `N - 1`. Let's see an example:

In [6]:
```
# We create a random permutation of integers 0 to 4
np.random.permutation(5)
```

Out[6]:
```
array([3, 1, 4, 0, 2])
```

To Do

In the space below create a rank 1 ndarray that contains a random permutation of the row indices of `X_norm`. You can do this in one line of code by extracting the number of rows of `X_norm` using the `shape` attribute and then passing it to the `np.random.permutation()` function. Remember the `shape` attribute returns a tuple with two numbers in the form `(rows,columns)`.

In [7]:
```
# Create a rank 1 ndarray that contains a random permutation of the ro
 indices of `X_norm`
row_indices = np.random.permutation(X_norm.shape[0])
```

Now you can create the three datasets using the `row_indices` ndarray to select the rows that will go into each dataset. Rememeber that the Training Set contains 60% of the data, the Cross Validation Set contains 20% of the data, and the Test Set contains 20% of the data. Each set requires just one line of code to create. Fill in the code below

In [8]:
```
# Make any necessary calculations.
# You can save your calculations into variables to use later.

# You have to extract the number of rows in each set using row_indices.
# Note that the row_indices are random integers in a 1-D array.
# Hence, if you use row_indices for slicing, it will NOT give the correc
t result.

# Let's get the count of 60% rows. Since, len(X_norm) has a lenght 1000,
therefore, 60% = 600
sixty = int(len(X_norm) * 0.6)

# Let's get the count of 80% rows
eighty = int(len(X_norm) * 0.8)

# Create a Training Set
# Here row_indices[:sixty] will give you first 600 values, e.g., [93 255
976 505 281 292 977,.....]
# Those 600 values will will be random, because row_indices is a 1-D arr
ay of random integers.
# Next, extract all rows represented by these 600 indices, as X_norm[row
_indices[:sixty], :]
X_train = X_norm[row_indices[:sixty], :]

# Create a Cross Validation Set
X_crossVal = X_norm[row_indices[sixty: eighty], :]

# Create a Test Set
X_test = X_norm[row_indices[eighty: ], :]
```

If you performed the above calculations correctly, then `X_tain` should have 600 rows and 20 columns, `X_crossVal` should have 200 rows and 20 columns, and `X_test` should have 200 rows and 20 columns. You can verify this by filling the code below:

In [9]:
```
# Print the shape of X_train
print(X_train.shape)

# Print the shape of X_crossVal
print(X_crossVal.shape)

# Print the shape of X_test
print(X_test.shape)
```

```
(600, 20)
(200, 20)
(200, 20)
```

In [ ]:
```

```