CSC 470: Introduction to Neural Networks

Programming Project #2

Using a Multilayer Perceptron to Predict Diabetes

Background

For this project you will be building a multilayer perceptron to try to predict whether a given individual has diabetes or not. The dataset you will use is an older, but commonly used dataset, the Pima Indians Diabetes Dataset. This project is intended to be a gentle introduction to creating and using a multilayer perceptron, as well as introduce you to a lot of third-party libraries supported by Python that are frequently used in machine learning.

Unlike the first project, you won't be implementing all the neurons in the multilayer perceptron yourself (although that would be an excellent challenge if you are so inclined). As a result, the actual coding for this project will be relatively light, but less intuitive, as you will need to learn some of the APIs for the various libraries.

Rather than state a problem and have you try to assimilate and make sense of all the libraries, which would be a daunting task, I've decided to make this assignment as more of a tutorial. I've taken this particular project from the following book:

Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects, by James Loy (2019), Packt Publishing.

I don't want to make you buy this book for this one project, though, so I'm covering the entire project in this writeup. There are also a few code changes from the book as well.

After you go through the assignment I will have you tweak some things to see how changing certain parameters of the network affects the learning process. Rest assured, not all the projects in the course will be simple tutorial-type projects, but I can't think of a better way to introduce you to all the various libraries without some sort of hands-on learning. The libraries are all quite large, and trying to assimilate the API, then trying to find the functions you need, would most likely lead to a lot of confusion and frustration.

As always, please do not hesitate to email me if you run into any problems!

Installing the Third-Party Libraries

All the third-party libraries used for this project can be installed using Python's package installer (pip). The installation instructions shown below assume you are using pip; if you are using Anaconda to install and manage your packages you will need to refer to the instructions for installing packages through Anaconda. For each package below I list the version I have installed, the URL for the website, a link to the API documentation for the library, and the command-line instructions for installing the package. Please note that pip must be run from the operating system command line, *not* the Python prompt. If pip recommends that you update to a newer version of pip, then do so, using: `pip install --upgrade pip`.

Python's package installer (PIP) will usually try to install/de-install dependencies for each package you try to install to avoid version incompatibilities. If pip does not automatically install a dependency, you will need to note from the install output that gets displayed what the package is, and then install that package before proceeding. pip has a tendency to commandeer the install process, so it's generally best to just let it do its thing. The "`--upgrade`" flag can be used to direct pip to update a package if it is already installed. I have not exhaustively tested all the permutations of package installs, so I can't guarantee what you will see when you install the packages. What I've listed below is the order in which I installed the packages, and so far I have not run into any issues.

NOTE: TensorFlow is currently tested only for Python 3.7-3.10.

NumPy (mathematical functions, linear algebra operations, random numbers)	Version: 1.23.4 URL: https://numpy.org/install/ API: https://numpy.org/doc/stable/reference/ Install command: `pip install numpy`
seaborn (statistical data visualization)	Version: 0.12.0 URL: https://seaborn.pydata.org/installing.html API: https://seaborn.pydata.org/api.html Install command: `pip install seaborn`
matplotlib (data plotting)	Version: 3.5.1 URL: https://matplotlib.org/stable/users/installing.html API: https://matplotlib.org/stable/api/index.html Install command: `pip install matplotlib`
scikit-learn (predictive data analysis)	Version: 1.1.2 URL: https://scikit-learn.org/stable/install.html API: https://scikit-learn.org/stable/modules/classes.html Install command: `pip install scikit-learn`
pandas (data analysis and manipulation)	Version: 1.5.0 URL: https://pandas.pydata.org/ API: https://pandas.pydata.org/docs/reference/index.html Install command: `pip install pandas`
tensorflow (Google's open source platform for machine learning)	Version: 2.10.0 URL (TensorFlow): https://www.tensorflow.org/install API: https://www.tensorflow.org/api_docs/python/tf/all_symbols Install command: `pip install tensorflow`
keras (deep learning framework)	Version: 2.10.0 (installed by the tensorflow package; see above) URL (Keras): https://keras.io/getting_started/ API: https://keras.io/api/

The Dataset

The Pima Indians diabetes dataset was compiled over 30 years ago, and provided by the National Institute of Diabetes and Digestive and Kidney Diseases. The dataset used to be hosted by the UCI Machine Learning Repository., but it is no longer hosted there. It can be found in numerous other places, and from what I can tell it is public domain, so there should be no issues with using it. I have included a copy of the dataset with the assignment. I retrieved this copy from the following URL:

https://networkrepository.com/pima-indians-diabetes.php [1]

You can also find the dataset on Kaggle, but you will need to have a Kaggle login in order to download the dataset.

The dataset, like many datasets in machine learning, is a comma-separated value (.csv) file. There are 9 columns in this dataset, with data collected for the following characteristics (appearing from left to right in the dataset file):

Pregnancies	The number of the patient's previous pregnancies
Glucose	Plasma glucose concentration
Blood Pressure	This is the diastolic blood pressure
Skin Thickness	The skin fold thickness as measured from the triceps area
Insulin	The blood serum insulin concentration
BMI	Body mass index
Diabetes Pedigree Function	This is a summarized score that indicates the genetic predisposition of the patient for diabetes, as extrapolated from the patient's family record for diabetes
Age	The patient's age, in years
Outcome	This is the desired output we want to be able to predict. A value of `1` indicates the patient developed diabetes within 5 years of the initial measurement; a value of `0` indicates the patient did not develop diabetes.

Reading the Dataset

Before we can do anything, we need to load the dataset from the .csv file. in the last project we used Python's built-in file reading capabilities to read the dataset into a 2D list of lists. For this project we're going to load the dataset in such a way as to make it more useful for later steps. Instead of the barebones 2D array we will load the dataset into a `DataFrame` object from the `pandas` library. A `DataFrame` is essentially a matrix, but with a number of useful built-in functions. (We'll see some of these functions later, but feel free to check the `pandas` API on your own to learn more.) We can use `pandas` to read a cvs file into a `DataFrame` using the following syntax:

```
import pandas as pd
df = pd.read_csv('diabetes.csv')
```

Here, we've opted to simply import the entire `pandas` namespace, and use the `read_csv` function to read the .csv file and return a `DataFrame` object.

Exploratory Analysis of the Dataset

Before launching into building a neural network to learn the dataset, it is always a good idea to inspect the data first for any irregularities or inconsistencies that could throw off the learning performance. This type of analysis is often called an **exploratory analysis**, since we're exploring the data to see what we have. For example, are there any outlier data values that might complicate analysis? Are there any data values that are invalid or that do not make sense? Identifying problematic data as a first step will help put us on the right track and enable our neural network to learn as efficiently as possible.

Now, obviously it's going to be difficult to identify problem data just by looking at a bunch of data in a spreadsheet. We might find something, but finding patterns by looking at raw numbers is not a forte for most humans (though a neural network can do this quite easily!). For us to spot problems in the data we need a better way of visualizing the data, such as with graphs, plots, etc. As a first step in the exploratory analysis of the data, we will look at histograms of each of the columns. A histogram is just a bar chart that shows the quantities for each of the different values in a set of data. We can generate a histogram very easily using the `DataFrame` object by calling the `hist` function:

```
df.hist()
```

This function specifically makes a histogram of the `DataFrame`'s columns. This function is entirely dependent on the `matplotlib` library, as it uses a function from that library to create the histogram. The output of this function is interesting, in that the functions returns an actual plot, but the act of returning the plot doesn't make it visible. In order to see the plot we need to call the `plt` function from `matplotlib`. The syntax for this is shown below, and includes the import statement needed to access the `plt` function.

```
from matplotlib import pyplot as plt
plt.show()
```

This shows the histogram in a separate window, which you will probably need to resize in order to view it. Figure 1 shows what the histogram looks like.

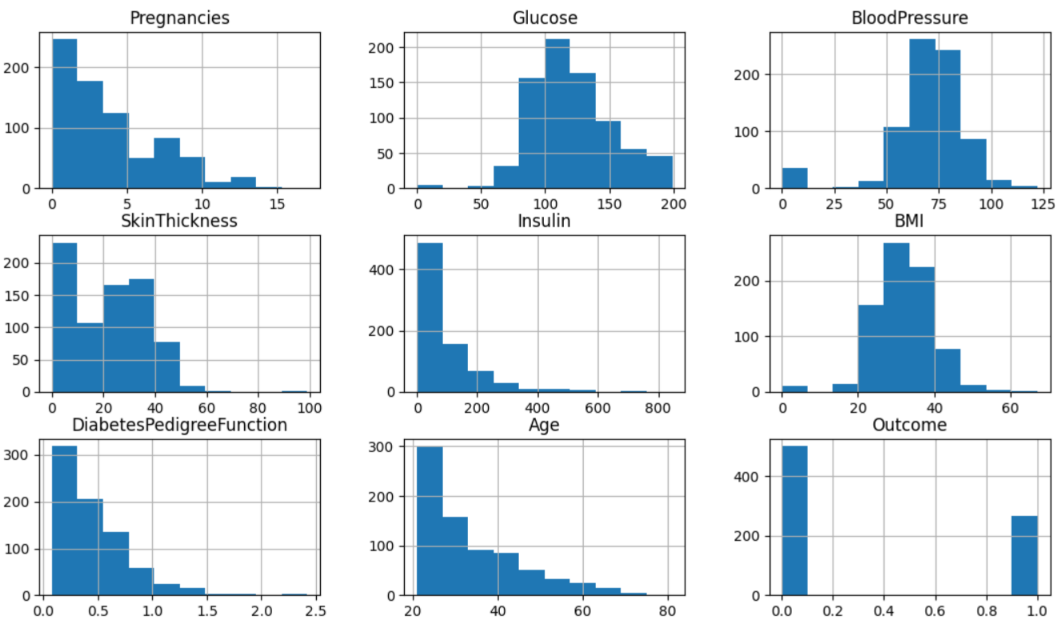


Figure 1. Histogram of the diabetes dataset. Y axis represents the number of patients.

The window in which the histogram appears has a number of tools located in the lower left corner. As drawn, the histogram is very crude, as it is not fine-grained enough to show a separate bar for each discrete data value. By default, the bin size for a histogram is set to 10, which means each bar represents all the patients who had values within a range of 10 values; e.g., the number of patients with values from 0 to <10, from 10 to <20, etc. To get a separate bar for each discrete value, you would need to pass a value for the bins equal to the number of discrete values for each characteristic. For example, for the Glucose characteristic there are 136 discrete values. To see one bar for each value you would set the number of bins to 136, like so:

```
df.hist(bins=136)
```

If you do this you will see a much more detailed and accurate histogram for Glucose. Unfortunately, since the way in which we are calling the `hist()` function causes it to create a histogram for each of the 9 characteristics, optimizing the bin size for one characteristic may make the histograms for one or more of the other characteristics harder to read. For our purposes I will continue to refer to the histogram from Figure 1, with a bin size of 10.

Looking at the histogram, we can glean a lot of information. First, it's important to note that the x-axis represents the value for the particular characteristic, and the y-axis represents the number of patients with a given value. We can see that several of the characteristics appear to be normally distributed, as evidenced by their bell-shaped curves. Other characteristics definitely show skewed distributions. Observations like this can be very important when deciding on the architecture of a neural network, as well as determining initial synaptic weights and the learning parameter value.

Another concern is the distribution of the Outcome characteristic, which shows that 268 of the 768 patients (~35%) had diabetes. The concern here is that if we want to develop a neural network to accurately predict the presence of diabetes, we need to make sure our training date reflect the distributions found in the real world. Otherwise, the learning process will be skewed towards predictions that reflect the training data, not the real world statistics. In this case, according to the World Health Organization (WHO), as of 2014 only about 8.5% of the world's population has diabetes. Clearly the dataset shows diabetes is more prevalent in the Pima Indian population than it is in the world overall. A discrepancy like this doesn't necessarily render invalid either the dataset or any neural network trained with that dataset. However, it does mean that we must take great care in how we interpret the results if we use the network outside the Pima Indian population. We won't be concerned about this particular distribution problem for this asignment.

Of greater concern are the several cases in the data where the data are not consistent with what we would expect. For example, there are 5 patients who had a blood glucose level of 0 which is virtually impossible. The same reasoning applies to the 35 patients with a diastolic blood pressure of 0, and the 11 patients with a BMI of 0.

A final concern about the data is the fact that each of the characteristics has a different scale; i.e., a different range of possible values. This type of situation can cause problems with learning, as variables with larger scales tend to be dominant over variables with smaller scales. To prevent problems with scaling we can normalize the variables to even out the effect each variable has on the learning process.

A separate question that always arises with datasets is whether or not any of the characteristics have a tendency to be better predictors of a certain outcome, in this case the presence of diabetes, than other characteristics. The answer to this question may influence our decisions about assigning initial values for certain synaptic weights, bias values, etc. In the lecture from the previous module we talked about the benefit of allowing prior information to be imparted to a neural network. If we find that certain characteristics are better indicators of diabetes than others, we can make adjustments to allow those characteristics to be given greater importance in the learning process.

In the case of the current dataset, we can use density plots to determine whether a relationship exists between any of the characteristics and the desired output. The seaborn library, which is also based on matplotlib, will enable us to generate the density plots we need. The code for creating the density plots is shown below.

```
import seaborn as sns

# Create a subplot of 3 x 3
figure, axes = plt.subplots(3,3,figsize=(15,15))

# Make sure there is enough padding to allow titles to be seen
figure.tight_layout(pad=5.0)

# Plot a density plot for each variable
for idx, col in enumerate(df.columns):
    ax = plt.subplot(3,3,idx+1)
    ax.yaxis.set_ticklabels([])
    sns.distplot(df.loc[df.Outcome == 0][col], hist=False,
    axlabel= False, kde_kws={'linestyle':'-',
    'color':'black', 'label':"No Diabetes"})
    sns.distplot(df.loc[df.Outcome == 1][col], hist=False,
    axlabel= False, kde_kws={'linestyle':'--',
    'color':'black', 'label':"Diabetes"})
    ax.set_title(col)

# Hide the 9th subplot (bottom right) since the relationship between the
# two outcomes themselves is meaningless
plt.subplot(3,3,9).set_visible(False)

# Show the plot
plt.show()
```

Figure 2 shows the results of the density plots. Again, a great deal of information can be learned from a set of relatively simple graphs. In each plot, the solid line represents patients with *no* diabetes, while the dashed lines represent patients *with* diabetes. In looking for a relationship between each of the characteristics and the presence of diabetes, we are looking for significant differences between the two curves. If both curves overlap each other, it indicates the characteristic's distribution is the same regardless of whether a patient has diabetes. This information is always good to know, but if we want to find a way to predict diabetes from the data, we need to find characteristics whose distribution *changes* in diabetic patients compared to non-diabetic patients. Looking at the plots in Figure 2, we can see such changes in some of the characteristics, the most pronounced being in the number of prior pregnancies, the blood glucose level, and age. Lesser correlations can be found in the body mass index (BMI) and the diabetes pedigree function. Characteristics that do exhibit significant differences between diabetic and non-diabetic patients are good candidates for diabetes indicators, whereas characteristics that behave the same in both populations are poor candidates.

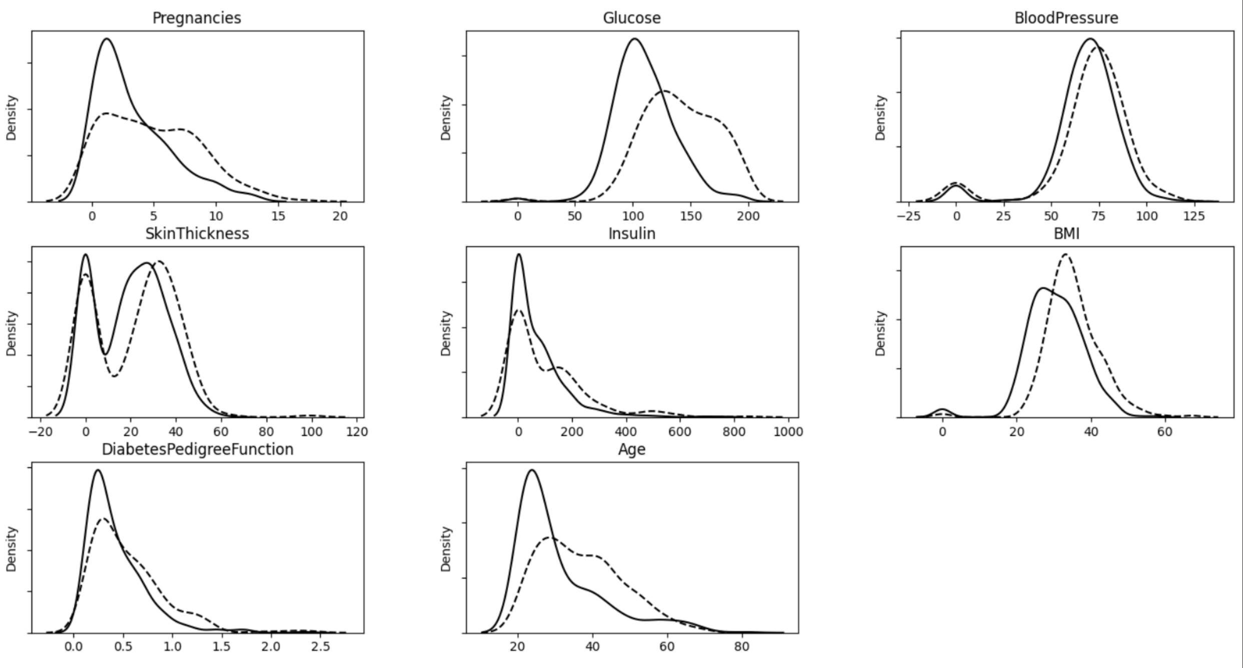


Figure 2. Density plots showing the relationships between the individual characteristics and the presence of diabetes. Solid lines represent no diabetes, while dashed lines represent the presence of diabetes.

Preprocessing the Data

An exploratory analysis of the data will help to clue us into potential problems with the data values, as we've seen. Once we have identified those problems, the next step is figuring out how to deal with them. If we do nothing, it can adversely affect the learning process or even prevent a network from learning from the data at all.

One of the most common problems in a dataset is **missing values**. Missing data can occur from errors in data entry or simply because the data was not available at the time it was collected. The extent of missing data will greatly influence how we design our neural network. The `pandas DataFrame` object has a convenient pair of functions we can use to tell us whether there are any null values in the dataset. Assuming we have already read the dataset into a `DataFrame` object (using `df = pd.read_csv('diabetes.csv')`), we would use the following code to check for null values:

```
print(df.isnull().any())
```

For the diabetes dataset the result that is displayed is:

```
Pregnancies                 False
Glucose                     False
BloodPressure               False
SkinThickness               False
Insulin                     False
BMI                         False
DiabetesPedigreeFunction    False
Age                         False
Outcome                     False
dtype: bool
```

So, the dataset contains no null values, which is good. However, recall that when we looked at the histograms before we found that some of the characteristics had values of 0 when such a value should be impossible for those characteristics. Missing data is often “zeroed out” with default zero values in order to avoid null values, but these zero values are still a problem. The `DataFrame` object has a function, `describe()`, that will display some useful descriptive statistics of the dataset. The syntax for calling this function is:

```
pd.set_option(“display.max_columns”, 500)
print(df.describe(include=’all’))
```

The first statement forces all the columns to be displayed. The default behavior is to truncate the middles of both rows and columns by representing undisplayed data with ellipses, as a means to ensure the data fit on the display. We need to see all the data, though, so we set the option to show the max number of columns. The second statement above calls the `describe()` function and displays the results. The parameter, `include=’all’`, is not always necessary, but it’s never a bad idea to just include it anyway. Including this parameter ensures columns of mixed data types will be displayed. The default is to only display statistics only for numerical data. If we run the above code, we get the following output:

```
       Pregnancies     Glucose  BloodPressure  SkinThickness     Insulin  \
count   768.000000  768.000000     768.000000     768.000000  768.000000
mean      3.845052  120.894531      69.105469      20.536458   79.799479
std       3.369578   31.972618      19.355807      15.952218  115.244002
min       0.000000    0.000000       0.000000       0.000000    0.000000
25%       1.000000   99.000000      62.000000       0.000000    0.000000
50%       3.000000  117.000000      72.000000      23.000000   30.500000
75%       6.000000  140.250000      80.000000      32.000000  127.250000
max      17.000000  199.000000     122.000000      99.000000  846.000000

             BMI  DiabetesPedigreeFunction         Age     Outcome
count  768.000000                768.000000  768.000000  768.000000
mean    31.992578                  0.471876   33.240885    0.348958
std      7.884160                  0.331329   11.760232    0.476951
min      0.000000                  0.078000   21.000000    0.000000
25%     27.300000                  0.243750   24.000000    0.000000
50%     32.000000                  0.372500   29.000000    0.000000
75%     36.600000                  0.626250   41.000000    1.000000
max     67.100000                  2.420000   81.000000    1.000000
```

As you can see, a total of 8 different statistics are displayed for each of the columns:

- the total count of values
- the mean
- the standard deviation
- the minimum value
- cutoff values for three default percentiles: 25%, 50%, and 75%
- the maximum value

To address the problem of zero values we look at the results for the minimum value. We can see the minimum value is 0 for every characteristics except for Age and the Diabetes Pedigree Function. The characteristics where a value of 0 makes the least sense are Glucose, Blood Pressure, Skin Thickness, Insulin, and BMI, as these characteristics would never be 0 for living patients. The reason the zero values are present could be due to a number of reasons, including data entry errors, malfunctioning equipment, or a patient who may have refused to allow measurement of one or more characteristics.

To find out how many zero values there are in each column, we can use the following code:

```
print("Number of rows with 0 values for each variable:")
for col in df.columns:
      missing_rows = df.loc[df[col] == 0].shape[0]
      print(col + ": " + str(missing_rows))
```

The `columns` attribute of the `DataFrame` returns a list of all the columns. The `loc` attribute returns a group of rows or columns by label. In this case, we are retrieving for each column every row containing a value of 0. Finally, the `shape` attribute returns the dimensionality of a `DataFrame` or part of a `DataFrame`. In this case it will just return the count of the rows containing a value of 0, but in other applications it can be used to return the dimensions of a multi-dimensional array. If we run the above code we see the following output:

```
Number of rows with 0 values for each variable:
Pregnancies: 111
Glucose: 5
BloodPressure: 35
SkinThickness: 227
Insulin: 374
BMI: 11
DiabetesPedigreeFunction: 0
Age: 0
Outcome: 500
```

Keeping in mind the dataset only has 768 total rows, the high numbers of zero values for both Skin Thickness and Insulin, two of the characteristics that should never be 0, are definitely a cause for concern. We will assume the zero values for these characteristics represent missing data. The question now is how to deal with those values.

Handling Missing Data

There is no universal way to deal with missing data values. How you deal with missing data depends on the type of data that are missing, and the relationship those data have to the desired output. If an exploratory analysis determines a particular characteristic doesn't seem to correlate with the desired output, it may be possible to simply ignore those values. Often, however, you won't have a clear-cut idea how important a particular characteristic is, and the data will all need to be taken into consideration. Three commonly used methods of dealing with missing data are:

1. **Remove any rows containing missing data.** This is different from ignoring a particular characteristic as I alluded to above. This method eliminates from the sample population any row containing missing data, thereby reducing the size of the dataset. This may be fine if the dataset is large, but if the dataset is small, like the diabetes dataset we're using, eliminating rows may not be a viable option.

2. **Replace the missing values with the median, the mean, or the mode of the values that are present for that characteristic.** Any time you add to a dataset any data that did not come from a valid measurement or observation, you risk introducing an unintentional bias in the data. However, replacing missing data with the mean, median, or mode of the existing values has the benefit of normalizing any bias you might introduce.

3. **Use a separate machine learning model to predict values for the missing data.** For example, you might be able to use regression analysis to predict the missing values. Again, since the replacement values will not have come from valid

observations or measurements, there is some risk of introducing bias.

For this project, we will replace all the missing data for the characteristics, Glucose, Blood Pressure, Skin Thickness, Insulin, and BMI with the mean of the existing values for those characteristics. To do this we'll follow a two-step process:

1. First, replace all the 0 values for those characteristics with the value, `NaN`, which is Python's **Not a Number** value.

2. Second, replace all the `NaN` values for each characteristic with the mean of the existing values for that characteristic.

The code for accomplishing the first step is as follows:

```
import numpy as np

df['Glucose'] = df['Glucose'].replace(0, np.nan)
df['BloodPressure'] = df['BloodPressure'].replace(0, np.nan)
df['SkinThickness'] = df['SkinThickness'].replace(0, np.nan)
df['Insulin'] = df['Insulin'].replace(0, np.nan)
df['BMI'] = df['BMI'].replace(0, np.nan)
```

You can access an entire column by label if the dataset file includes labels, which the diabetes.csv file does. The `replace` function of `DataFrame` allows you to replace all instances of one value in a column with another value. In this case, we are replacing all 0 values in each of the above columns with `NaN`. We can reuse the code we used previously to display the number of rows with 0 values to confirm the 0 values have been replaced. The result looks like this:

```
Number of rows with 0 values for each variable:
Pregnancies: 111
Glucose: 0
BloodPressure: 0
SkinThickness: 0
Insulin: 0
BMI: 0
DiabetesPedigreeFunction: 0
Age: 0
Outcome: 500
```

We now need to replace all the `NaN` values with the means for the existing values. We can do that using the `DataFrame`'s `fillna` function. This function specifically replaces `NaN` values with a specified value. To find the mean for a column we can use the column's `mean()` function. Below is the code for doing this replacement:

```
df['Glucose'] = df['Glucose'].fillna(df['Glucose'].mean())
df['BloodPressure'] = df['BloodPressure'].fillna(df['BloodPressure'].mean())
df['SkinThickness'] = df['SkinThickness'].fillna(df['SkinThickness'].mean())
df['Insulin'] = df['Insulin'].fillna(df['Insulin'].mean())
df['BMI'] = df['BMI'].fillna(df['BMI'].mean())
```

The obvious question here is, why not just replace the 0 values with the mean in the first place? The reason is that the `mean()` function would have included all those 0 values in its computation of the mean, which we don't want. We only want the mean of the non-zero values. If we first replace the 0 values with `NaN`, the `NaN` values are ignored and not included as part of the mean computation.

Data Normalization

We saw from the histograms of the data during the exploratory analysis that the characteristics did not all use a common scale. From the descriptive statistics we obtained from the `describe()` function, we saw the min and max values for each of the characteristics. The max values for Insulin (846) and the Diabetes Pedigree Function (2.42) help to illustrate the wide differences in scale between the characteristics. This is not uncommon, as the numerical values that get reported for a characteristic are determined by the units of the characteristic.

We talked in Module 8 how during back-propagation the weight vector follows a zigzag trajectory as it converges towards a minimum. We want the lengths of the zigs and zags to be as small as possible, and these lengths are influenced heavily by the input values. If we have a dataset where the inputs for some of the columns have a small range of values, but other columns have inputs with significantly larger ranges, the larger inputs will tend to overshadow the smaller inputs by undoing the small zigzagging of the small inputs with a much larger zigzagging. In effect, the characteristics with the larger scales are given greater importance in the learning process. We mentioned one way to compensate for this using a process called **centering**. Centering was the first of three normalization steps (Centering, Decorrelation, Covariance Equalization), and involves subtracting the mean value from each column from each value in the column. This has the effect of shifting the distribution of values for the column so that the center of the distribution lies roughly on the x-axis (if we were to plot the values on a standard x,y coordinate system. The resulting mean is 0 or very close to it.

It is not difficult to write our own code to find the mean of each column and then subtract the mean value from all the values in that column, but since this is such a commonly performed operation, we can take advantage of another library, `scikit-learn`, to do this for us. The preprocessing class of this library has a function called scale() that carries out the first and third normalization steps (it does not perform decorrelation). The following code shows how to do it:

```
from sklearn import preprocessing

# Normalize the data via centering
# Use the scale() function from scikit-learn
print("Centering the data...")
df_scaled = preprocessing.scale(df)

# Result must be converted back to a pandas DataFrame
df_scaled = pd.DataFrame(df_scaled, columns=df.columns)

# Do not want the Outcome column to be scaled, so keep the original
df_scaled['Outcome'] = df['Outcome']
df = df_scaled

print(df.describe().loc[['mean', 'std','max'],].round(2).abs())
```

The `scale()` function works directly on a pandas `DataFrame`, but unfortunately it does not return a `DataFrame`, it returns a matrix of a different format. This matrix can easily be converted back to a `DataFrame`, though, as shown above. The `scale()` function operates on all the columns, but we don't actually want to normalize the `Outcome` column. We therefore replace the scaled `Outcome` column with the original `Outcome` column from the `DataFrame`. The print statement displays for each characteristic the mean, the standard deviation, and the max value. The results are shown on the next page.

```
      Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI
mean         0.00     0.00            0.0           0.00     0.00  0.00
std          1.00     1.00            1.0           1.00     1.00  1.00
max          3.91     2.54            4.1           7.95     8.13  5.04

      DiabetesPedigreeFunction   Age  Outcome
mean                      0.00  0.00     0.35
std                       1.00  1.00     0.48
max                       5.88  4.06     1.00
```

The result is a dataset where the characteristics are now much closer to each other scale-wise, and the target mean value of 0 has been achieved. The max values for the characteristics are also much closer together than they were in the raw data. This will end our preprocessing. Even though we did not do any decorrelation (step 2 of the normalization sequence), we have successfully performed the first and third steps. Now the data are ready to be split into a training set and a testing set.

Generating the Training Set, Validation Set and Testing Set

Previously we have only talked about splitting a dataset into two subsets, a training set and a testing set. There is a third type of subset that is often used, called a **validation set**. The validation set is generated from the training set, and is typically used for tuning the number of hidden layers (**hyperparameter tuning**). For this project we won't use any complex formulas to determine what size to make each of the sets. Instead, we will employ a common 80:20 ratio:

Training Set:	80%	Validation Set:	20% of Training Set
Testing Set:	20%		

First, we split the entire dataset such that 80% of the inputs comprise the training set and 20% of the inputs comprise the testing set. The validation set divides the training set following the same ratio, such that the validation set comprises 20% of the training set.

In order to avoid unintentional bias, we want to have the inputs that go into each of the sets chosen randomly. In the first programming project you used Python's built-in random number function. While there is nothing wrong with doing it this way, to make things easier we can use the `train_test_split` function from the `scikit-learn` library. This function will do all the work of randomly splitting a dataset for us. The code below shows how to use this function to randomly split the diabetes dataset:

```
from sklearn.model_selection import train_test_split

# Split dataset into an input matrix (all columns but Outcome) and Outcome vector
X = df.loc[:, df.columns != 'Outcome']
y = df.loc[:, 'Outcome']

# Split input matrix to create the training set (80%) and testing set (20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Second split on training set to create the validation set (20% of training set)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2)
```

The first step is to separate the dataset into the input matrix and the desired outcome vector, since the desired outcome is not one of the true inputs. We use the standard notation if using an uppercase letter to represent a matrix and a lowercase letter to represent the desired outcome vector. Next, we split the dataset (minus the Outcome column) to randomly choose 80% of the rows for the training set and 20% of the rows for the testing set. Last, we split the *training set* to randomly choose 80% of the training set rows to serve as the final training set, and 20% of the training set rows to serve as the validation set.

Finally, we are ready to turn our attention to designing the multilayer perceptron.

Designing the Multilayer Perceptron

For this project we will design a multilayer perceptron consisting of 4 layers of neurons:

- The input layer
- 2 Hidden layers
- The output layer

Each layer is discussed in more detail next. The number of hidden layers is arbitrary, but we will go with 2 hidden layers, which is very common for many problems.

Input Layer

The input layer is easy to design since we need 1 neuron for each of the 8 inputs (Pregnancies, Glucose, Blood Pressure, Skin Thickness, Insulin, BMI, Diabetes Pedigree Function, and Age). The outcome is not included as one of the inputs. Each input neuron will have one synaptic weight leading into each of the neurons in the first hidden layer.

1st Hidden Layer

Recall from the Module 8 lecture that the number of neurons we should include in each hidden layer is not easy to determine, and choosing an optimal number of hidden neurons typically requires extensive knowledge of and analysis of the input data. Work by Xu and Chen [2] resulted in the following formula for determining the number of hidden neurons:

$$m_1 = C\,(N / (m_0 \log N))^{1/2}$$

where m_1 is the number of hidden neurons, C is a complexity regularization constant, N is the size of the training set, and m_0 is the number of input neurons. There are two problems with using this formula, however:

- There is no easy way to determine the value for the complexity regularization constant, C.
- The number of hidden neurons the formula returns assumes only 1 hidden layer, so if you have multiple hidden layers you still have to determine how many neurons to assign to each layer.

We will arbitrarily use 32 hidden neurons for the first hidden layer. As you will see later, it will be an easy matter to tweak the number of hidden neurons in each layer as we train the network.

2nd Hidden Layer

The number of hidden neurons to use for each hidden layer is as difficult to optimize as the total number of hidden neurons. A lot depends on trial and error, and there is never any guarantee that a configuration that works well for one dataset will work for other datasets. For this project, we will arbitrarily use 16 hidden neurons in the second hidden layer.

Output Layer

The output layer is very easy to figure out. We only need a single neuron, which will receive synaptic weights from each neuron in the second hidden layer. The output layer is unique in that it is the only layer that has direct access to the desired outcome, which it will use to calculate the error signal and compute a weight adjustment to use for its incoming synaptic weights when it starts the back-propagation process. The neuron in this layer is also the one that ultimately returns the predicted output once the network has been trained.

Activation Functions

Each neuron will need an activation function to determine what value the neuron will ultimately send out to the neurons it is connected to. We have talked a lot previously about the advantages of using non-linear sigmoid functions in multilayer perceptrons, especially the logistic function and the hyperbolic tangent function. Despite this, for all the hidden neurons we will use a linear activation function that has been more or less deemed the activation function of choice for the hidden layers of multilayer perceptrons and deep neural networks. This function is known as the **ReLU** function, or the **Rectified Linear Unit** function. Experience has shown that using sigmoid functions for all the neurons of a multilayer network causes what is known as a **vanishing gradient problem**, which inhibits the learning process. The ReLU function does not cause this problem, and allows for faster learning and better performance, in general. The output of ReLU is simple to define:

For a given value, x, ReLU returns x if $x > 0$; otherwise, ReLU returns 0.

For the output layer, however, we will still use a non-linear activation function. Remember that the whole point of using multiple layers is to classify data that are *not* linearly separable, so at some point we will need a non-linear activation function. For this project we will use the standard sigmoid function:

$$S(x) = \frac{1}{1+e^{-x}}$$

The sigmoid function limits the range for all values of x to be between 0 and 1. An activation function based on the sigmoid function will return a 1 if the sigmoid function evaluates to 0.5 or greater, or 0 if the sigmoid function evaluates to less than 0.5.

Building the Multilayer Perceptron

To build the actual multilayer perceptron we will use the Keras library, which has been designed specifically for building machine learning models, and makes creation of a neural network architecture incredibly easy. It interfaces easily with Python, as well as with Google's TensorFlow framework, which we won't be using for this project, but we will see it for a later project. The two primary classes in Keras are the `Model` class and the `Sequential` class. The `Sequential` class groups a linear stack of layers into a `Model`. This architecture reflects the structure of a neural network, which basically consists of a number of layers of neurons arranged such that inputs flow sequentially into the layers from the input layer to the output layer.

To get started, we need to import the `Sequential` class from the Keras library and create a new instance of the `Sequential` class:

```
from keras.models import Sequential

model = Sequential()
```

The input layer will not be explicitly represented as a layer, so the next task is to add the first hidden layer:

```
from keras.layers import Dense

model.add(Dense(32, activation='relu', input_dim=8))
```

We first need to import the `Dense` class, which represents a layer of neurons that are all densely connected to the next layer. In a dense layer each neuron has a synaptic weight to every neuron in the following layer; i.e., there will be no gaps in the connections between the neurons. We then use the `add` function to append a layer to the model. The first argument specifies the number of neurons, which we said would be 32. The second argument specifies each neuron in the layer will use the ReLU activation function. Finally, the last argument specifies the number of inputs each neuron will receive. The number of inputs is the neuron's **dimensionality**. *The single call to the* `add` *function creates the entire first hidden layer*.

Next, we add the second hidden layer:

```
model.add(Dense(16, activation='relu'))
```

The second hidden layer is also a dense layer, and will have 16 neurons. Each neuron in the layer will use the ReLU activation function, just like the first layer. We do not need to specify the dimensionality this time, since Keras will determine the dimensionality automatically based on the size of the previous layer.

Finally, we add the output layer to the stack:

```
model.add(Dense(1, activation='sigmoid'))
```

As we mentioned before, the output layer will only have 1 neuron, but we still use a dense layer since the neuron will receive an input from each neuron in the second hidden layer. Also as we stated, the output neuron will use the non-linear sigmoid activation function.

Believe it or not, it takes only 4 lines of code (not including the import statements) to build the entire multilayer perceptron!

Compiling the Network

Before we can train the network we must define certain parameters for the learning process to follow. For this network we will define the following 3 parameters:

Optimizer

All models must have an optimizer specified. The optimizer is responsible for optimizing the weight vector during training. It will set the initial values for the synaptic weights and learning rate parameters, and it will update those values during learning. For this project we will use the **Adam optimizer** [3], which works well for many problems.

Loss Function

The loss function is also essential. This function is responsible for computing the quantity that is to be minimized during training. In this case, that quantity is the error signal. We will use the **binary cross-entropy function**, as this loss function is well suited to binary classification problems like the one we're trying to solve. Either a patient has diabetes or they do not, there are only those two options.

Metrics

This is the evaluation method we use to evaluate the performance of the network. For this problem we will use the **accuracy** metric, which measures the percentage of correctly classified samples.

Putting it all together, we can use the following code to compile the network:

```
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

The `compile` function can take additional arguments as well, but we will stick to these three for this project.

Training the Network

After compiling the network, it is ready to be trained. We will use the training set and the outcome vector we created previously. The only additional piece of information we need is the number of epochs to use. We'll start with a relatively low number of epochs, 200 to be precise. Training the network requires only a single call to the `fit` function:

```
model.fit(X_train, y_train, epochs=200)
```

If you run this code on a standard CPU you may see a warning like the following:

```
Could not load dynamic library 'nvcuda.dll'; dlerror: nvcuda.dll not found
```

This occurs because TensorFlow is designed to use the GPU associated with Nvidia's CUDA platform, since GPU's are currently significantly faster than most CPUs. The code will still run even if you don't have the optimal GPU installed, it will just run more slowly. For the small datasets we will be using this should not pose a problem, but it would cause problems with much larger and more complex datasets.

If you put all the code I've presented so far together into a single script and run it, when the training starts you will see output that looks something like this:

```
Epoch 1/200
16/16 [==============================] - 0s 931us/step - loss: 0.6760 - accuracy: 0.6171
Epoch 2/200
16/16 [==============================] - 0s 1ms/step - loss: 0.6370 - accuracy: 0.7047
Epoch 3/200
16/16 [==============================] - 0s 933us/step - loss: 0.6045 - accuracy: 0.7515
Epoch 4/200
16/16 [==============================] - 0s 1ms/step - loss: 0.5749 - accuracy: 0.7576
Epoch 5/200
16/16 [==============================] - 0s 997us/step - loss: 0.5457 - accuracy: 0.7658
...
Epoch 195/200
16/16 [==============================] - 0s 1ms/step - loss: 0.1810 - accuracy: 0.9267
Epoch 196/200
16/16 [==============================] - 0s 1ms/step - loss: 0.1794 - accuracy: 0.9246
Epoch 197/200
16/16 [==============================] - 0s 1ms/step - loss: 0.1793 - accuracy: 0.9267
Epoch 198/200
16/16 [==============================] - 0s 931us/step - loss: 0.1768 - accuracy: 0.9287
Epoch 199/200
16/16 [==============================] - 0s 1ms/step - loss: 0.1771 - accuracy: 0.9328
Epoch 200/200
16/16 [==============================] - 0s 1ms/step - loss: 0.1758 - accuracy: 0.9308
```

Here I'm showing the first 5 epochs and the last 5 epochs. For each epoch you can see the length of time required per step of the epoch, the value of the loss metric (binary cross-entropy in our case), and the current accuracy at the end of that epoch. The first value, "16/16" deserves some explanation. We are using 80% of the dataset for the training set, and only 80% of the training set gets used for training; the other 20% of the training set is used for validation. So, the number of samples we are training with is given by:

768 samples * 0.8 * 0.8 = 491 samples

According to the output, it would appear as though only 16 samples are being used. That is not the case, however. By default, if you do not specify a batch size when you call the `fit` function, the function will divide the training set into a number of batches equal to the number of training samples divided by the number of neurons in the first hidden layer, rounded up. Thus, with 491 samples we would get:

491 samples / 32 neurons = 15.34 batches = 16 batches rounded up

All 491 samples are being used in the training, but the output shows the results in terms of batches, not in terms of individual samples. If you want to process the samples individually, set the batch size to 1 when you call the `fit` function, like so:

```
model.fit(X_train, y_train, batch_size=1, epochs=200)
```

The results are shown on the next page. This time you will see "491/491" instead of "16/16". One thing you will notice if you use a batch size of 1 is that the training process will take significantly longer. You may recall in the first project we mentioned how using batches instead of individual samples can speed up the learning process. This is an excellent opportunity to confirm that statement. Another interesting observation is that the accuracy is greatly improved with a batch size of 1 compared to a batch size of 16. Keep in mind that these are just the results for the training set — we actually hope to see very high accuracy at the end of training, in fact, the higher the better.

The next step is to test the trained network to see how well it performs on the testing set.

```
Epoch 1/200
491/491 [==============================] - 1s 879us/step - loss: 0.5687 - accuracy: 0.6864
Epoch 2/200
491/491 [==============================] - 0s 926us/step - loss: 0.4504 - accuracy: 0.7841
Epoch 3/200
491/491 [==============================] - 0s 816us/step - loss: 0.4250 - accuracy: 0.7902
Epoch 4/200
491/491 [==============================] - 0s 808us/step - loss: 0.4094 - accuracy: 0.7943
Epoch 5/200
491/491 [==============================] - 0s 835us/step - loss: 0.3938 - accuracy: 0.8065
...
Epoch 195/200
491/491 [==============================] - 0s 855us/step - loss: 0.0015 - accuracy: 1.0000
Epoch 196/200
491/491 [==============================] - 0s 843us/step - loss: 0.0012 - accuracy: 1.0000
Epoch 197/200
491/491 [==============================] - 0s 892us/step - loss: 0.0012 - accuracy: 1.0000
Epoch 198/200
491/491 [==============================] - 0s 835us/step - loss: 0.0012 - accuracy: 1.0000
Epoch 199/200
491/491 [==============================] - 0s 861us/step - loss: 0.0048 - accuracy: 0.9980
Epoch 200/200
491/491 [==============================] - 0s 914us/step - loss: 0.0915 - accuracy: 0.9695
```

Testing the Network

With a trained network in hand, we need to evaluate the network's performance on inputs it has never seen before, namely the samples in the testing set. We will evaluate the network's performance using three methods:

1. Measuring the testing accuracy
2. Using a confusion matrix
3. Using a Receiver Operating Characteristic (ROC) curve

Measuring the Testing Accuracy

The `Sequential` class has an `evaluate()` function we can use to report the accuracy of the network with respect to a given set of input vectors and their corresponding outputs. The code for doing this is shown below. Note that this code must be executed immediately after the network has been trained. If the script you used to train the network is closed, you will need to train the network again before evaluating it.

```
# Evaluate the accuracy with respect to the training set
scores = model.evaluate(X_train, y_train)
print("Training Accuracy: %.2f%%\n" % (scores[1]*100))

# Evaluate the accuracy with respect to the testing set
scores = model.evaluate(X_test, y_test)
print("Testing Accuracy: %.2f%%\n" % (scores[1]*100))
```

The results of these evaluations will look something like this (the stochastic nature of the training will give slightly different results every time):

```
16/16 [==============================] - 0s 798us/step - loss: 0.1936 - accuracy: 0.9246
Training Accuracy: 92.46%

5/5 [==============================] - 0s 748us/step - loss: 0.7188 - accuracy: 0.7338
Testing Accuracy: 73.38%
```

These results are well within expectations. We would expect the network to perform with high accuracy on the training set, since it was trained specifically to optimize the weight vectors to minimize the error on those specific samples. We would also expect the accuracy to be less for the testing samples, since we can't be sure the samples in the training set are truly representative of the dataset as a whole. Randomly choosing samples for training will help to avoid bias, but there will always be some degree of error. The accuracy of 73.38% for the testing set is actually good, as it means the network should predict correctly about 73% of the time, which is much better than the 50% accuracy we would expect from simply guessing.

The accuracy of the testing set is the value that ultimately determines the value of the network as a classifier. If we aren't satisfied with 73% accuracy, we can re-train the network to see if we can boost the accuracy of the testing set higher. The point is, we don't want to use the accuracy with respect to the training set as the indicator, since a very high accuracy there could be evidence of overfitting, and could result in a lower accuracy when the testing samples are included.

Using a Confusion Matrix

With any neural network that acts as a classifier, there is always the possibility the network will incorrectly classify an input sample. This can happen with any number of output classes. For binary classifiers like the network we've built here, where one class indicates the presence of a disease or some other characteristic, and the other class indicates the absence of that disease or characteristic, we can use a set of 4 terms to indicate the results of a given classification:

- **True positive** — the network correctly predicts a patient **will** likely develop diabetes.
- **True negative** — the network correctly predicts a patient likely **not** develop diabetes.
- **False positive** — the network predicts a patient will likely develop diabetes, when in fact the patient likely will not develop the disease.
- **False negative** — the network predicts the network predicts a patient will likely not develop diabetes, when in fact the patient likely will develop the disease.

Clearly, we want to have high percentages with respect to both true positive and true negative, but low percentages for both false positive and false negative.

While we could just look at the percentages to determine how well the network has done with respect to these outcomes, we can also use what is known as a **confusion matrix** to visualize the results. A confusion matrix plots all the observed outputs against all the desired outputs, creating a table of sorts, where each cell contains the number of outcomes for each observed output/desired output pair. A perfectly accurate network would generate a confusion matrix that would look like an identity matrix. In practice, though, there are misclassifications, and the degree of shading of each cell in the matrix can provide an easy way to visualize how accurate a network is.

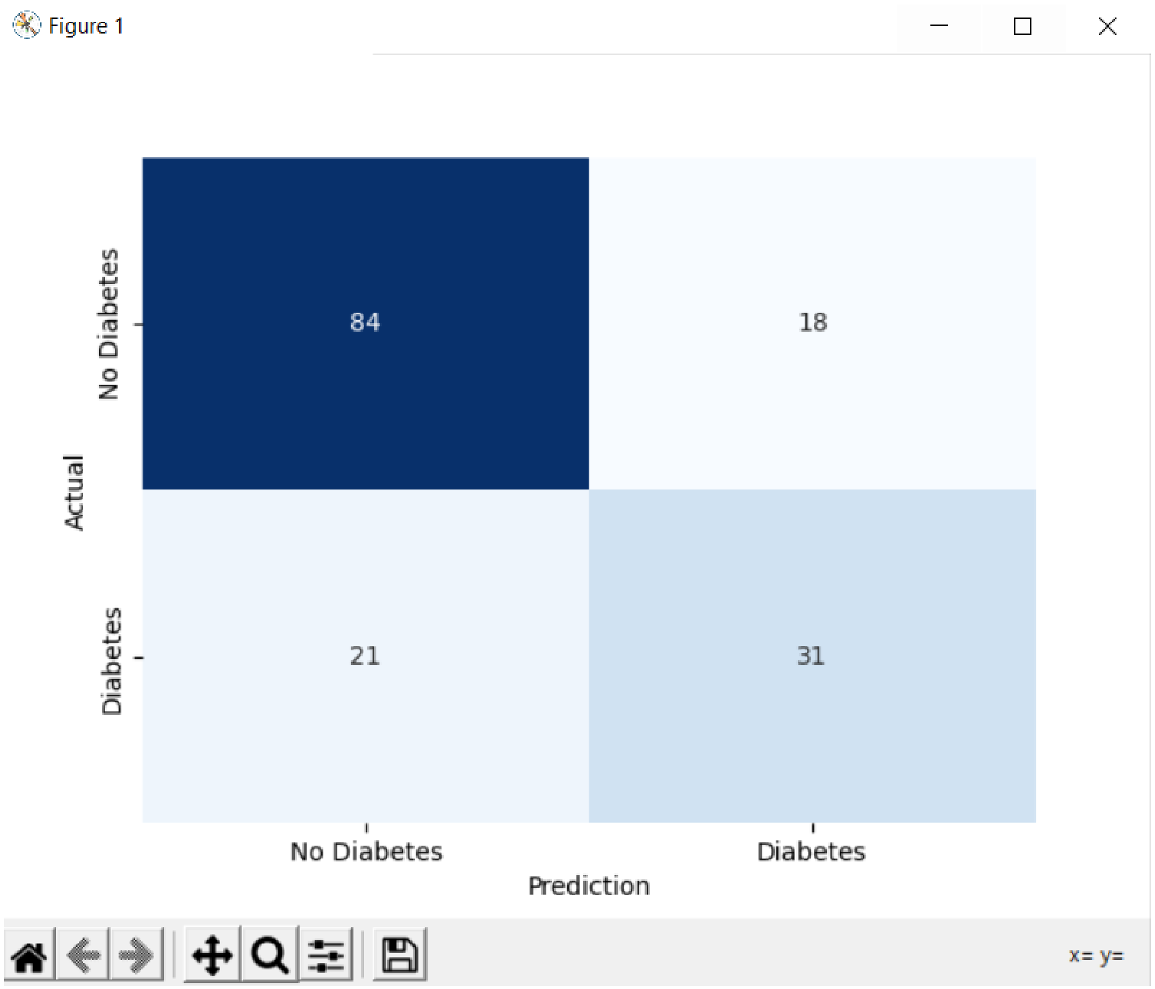
We can use the `confusion_matrix` class from the `scikit-learn` library, and then use the `seaborn` library to help us visualize the resulting matrix. Code for doing this is presented on the next page, using only the testing set samples.

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib
import matplotlib.pyplot as plt

# Construct a confusion matrix
y_test_pred = (model.predict(X_test) > 0.5).astype("int32")
c_matrix = confusion_matrix(y_test, y_test_pred)
ax = sns.heatmap(c_matrix, annot=True,
                 xticklabels=['No Diabetes','Diabetes'],
                 yticklabels=['No Diabetes','Diabetes'],
                 cbar=False, cmap='Blues')
ax.set_xlabel("Prediction")
ax.set_ylabel("Actual")

plt.show()
```

As you can see, drawing the confusion matrix also involves using the plotting capabilities of the `matplotlib` library. The `heatmap` function assigns a different shade of the specified color (blue in this case) to each cell representing a predicted output/desired output pair. The hue of the shading is directly proportional to the number of occurrences in the results, with higher numbers having darker hues and smaller numbers having lighter hues. The "heat" is represented by the darker hues. If you build, train, and test the network we have been discussing so far, and then create a confusion matrix using the above code, you should see something like the following:



For the confusion matrix, only the testing set was used, which consists of 154 samples (768 samples * 0.2 = 153.6 samples). The numbers in each square of the matrix represent the number of each observed output/desired output pair. The sum of all the values should equal the size of the testing set, and indeed it does. Here is how the 4 squares in the matrix map to the 4 outcomes we mentioned before:

Upper left square = True Negatives
Lower right square = True Positives
Upper right square = False Positives
Lower left square = False Negatives

We want higher numbers in the upper left and lower right squares, and this is what we have. Of 154 testing samples, 115 of them were correctly classified (~75%), while 39 were incorrectly classified (~25%).

For a problem with only two classes, the confusion matrix is still helpful, but may be a bit overkill. The confusion matrix is much more helpful the more output classes a problem has.

One final thing I want to mention about the code above is the very last line, which is the one that shows the plot. When a plot is shown, the script will pause until the plot's window is closed.

Using a ROC Curve

For classification problems involving results where you can have true positives and false positives, it is useful to plot the **true positive rate (TPR)** against the **false positive rate (FPR)**, creating what is known as a Receiver Operating Characteristic (ROC) curve. The two rates are calculated as follows:

TPR = True Positives / (True Positives + False Negatives)

FPR = False Positives / (False Positives + True Negatives)

To briefly explain these equations, the TPR is the number of true positives predicted out of all the positives in the dataset. Since false negatives are actually positives that were mistakenly classified, they must be included in the total number of positives. For the FPR, you divide the number of false positives by the sum of the false positives and the true negatives, since false positives should have been classified as negatives.

The `scikit-learn` library provides a convenient function, `roc_curve()`, that will draw the ROC curve for us. First, we need to have the network make its predictions for the testing set and store those results since we need to feed them to the `roc_curve()` function. The code for doing all this is shown below (I've only included the additional needed import statement):

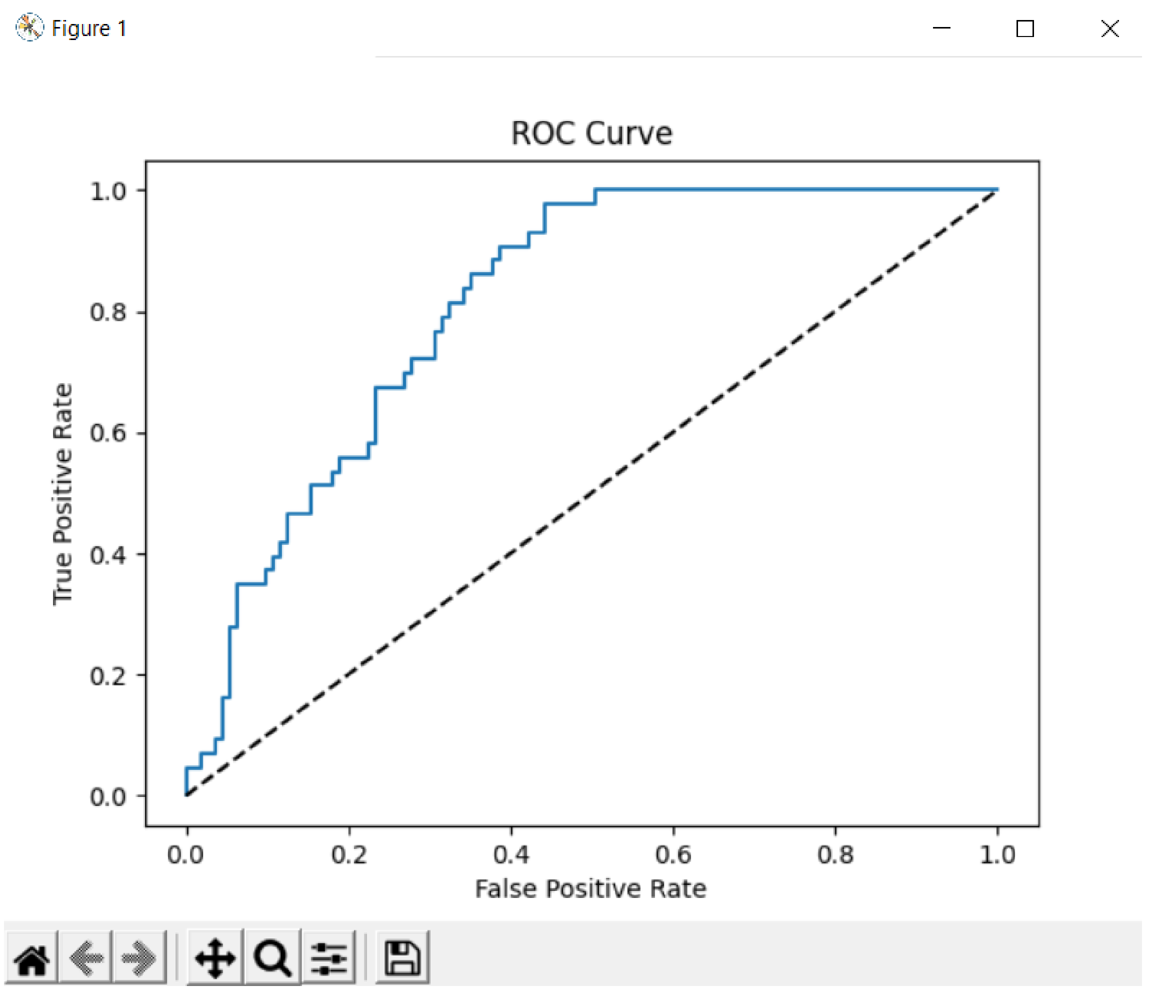
```
from sklearn.metrics import roc_curve

y_test_pred_probs = model.predict(X_test)

FPR, TPR, _ = roc_curve(y_test, y_test_pred_probs)

plt.plot(FPR, TPR)
plt.plot([0,1],[0,1],'--', color='black') #diagonal line
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```

The result should look something like the following. Keep in mind that the results will vary somewhat every time you train the network from scratch.



The dashed line represents the case where the false positive rate equals the true positive rate. In other words, for every true positive you have a false positive. For an accurate network we always want to be above this line, as that indicates the network is finding more true positives than false positives. The area under the ROC curve is directly proportional to the accuracy of the network. The larger the area under the curve (AUC), the more accurate the network is. This also means we want the curve to stay as far away from the dashed line as possible. Any movement in the direction of the dashed line represents a slip in performance. In a very bad network, one that is finding many more false positives than true positives, it is conceivable for the curve to reside entirely to the right of the dashed line. The result shown in the graph above represents a performance that is definitely good, but could be a bit better. The ideal shape would be if the curve formed a right triangle above the dashed line.

What You Need to Do:

As I stated previously, this project is less about problem solving and designing your own network from scratch, and more about introducing you to some of the third-party libraries commonly used in neural networks and machine learning. You can compile all the code presented in this assignment into a Python script (after you have installed all the required libraries) and run it to see all the results I've shown here. *Be sure to place all the import statements at the top of the script.* The project still has some room for improvement, though. Here is what I am asking you to do for this project:

1. **Compile all the code from this project into a single Python script.**

2. **Execute the script to perform all the tasks we covered in this assignment:** (I need to be able to run your script and see the output from these tasks.)

   a. Read the diabetes.csv file into a pandas `DataFrame`.
   b. Check the data for null values and print the result.
   c. Generate descriptive statistics for the dataset and print the result.
   d. Display the number of rows with 0 values for any of the characteristics.
   e. Replace all the 0 values for Glucose, Blood Pressure, Skin Thickness, Insulin, and BMI with the mean value for each of those columns (first replace with NaN, then with the mean).
   f. Normalize the data using the scale function from scikit-learn and print the result.
   g. Generate the training set, testing set, and validation set.
   h. Create the multilayer perceptron (MLP) as specified.
   i. Train the MLP, then test it.
   j. Measure the testing accuracy.
   k. Create and display the confusion matrix.
   l. Create and display the ROC curve.

3. **Create a copy of your script for each of the following:**

   a. Change the number of hidden nodes and/or hidden layers and repeat the above tasks to see how what you did affected the network's ability to learn the dataset. Describe the results.

   b. Look through the API for the `compile()` function of the Keras library, and try out one or more parameters we didn't cover here; or, change the value for one or more of the parameters we did use. Repeat the above tasks to see how the changes affected the learning process. Describe the results. The URL for the API for the `compile()` function is:

      https://keras.io/api/models/model_training_apis/#fit-method

   c. Look through the API for the optimizers available through Keras, and see what happens to the network when you replace the Adam optimizer with some of the other options that are available. Describe the results.

      https://keras.io/api/optimizers/

Important!

You will be submitting **a total of 4 Python scripts**, 1 for the original MLP based on the code I've presented in the assignment, and 3 separate scripts for the 3 items under #3 above. You also need to include a brief description of what you found when you made changes as outlined above.

What You Need to Turn In

Code:

Your 4 Python scripts as described above.

Results:

Brief descriptions of what you saw when you made changes to the network for item #3 above.

Point Distribution

Item	**Points Possible**	**Points**
Script #1 (the original MLP)	10	
Script #2 (variant of the MLP with changes made to the number of hidden layers and/or hidden neurons) plus description of results of changes	10	
Script #3 (variant of the MLP with changes made to the parameters passed to the `compile()` function) plus description of results of changes	10	
Script #4 (variant of the MLP with different optimizers used) plus description of results of changes	10	
Total:	**40**	

References

1. Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization, *AAAI Proceedings*, https://networkrepository.com.

2. Xu, S. and Chen, L. 2009. "Adaptive Higher Order Neural Networks for Effective Data Mining", in *The Sixth International Symposium on Neural Networks (ISNN 2009): Advances in Intelligent and Soft Computing*, p. 169, Springer-Verlag, Berlin Heidelberg.

3. Kingma, D.P. and Ba, J. 2015. Adam: A Method for Stochastic Optimization, Published as a conference paper at the 3*rd International Conference for Learning Representations*, San Diego, 2015.