Project 2: Healthcare Report

By: Nikita Patil

Course: Data Science Capstone

Date: April 6, 2020

Preface

Since this report is quite lengthy, I would like to include some pointers to help the reader navigate through it. The first two section are the Background and Problem Statement. The contents were essentially extracted from the Project Assignment itself as well as the included document in the Course Resources. The following section is the Analysis, where all the problems/requirements proposed are answered. This is broken up into the 5 sections highlighted in the Project Tasks. Specifically, they are: 'Data Exploration – Week 1', 'Data Exploration – Week 2', 'Data Modelling – Week 3', 'Data Modelling – Week 4', and 'Data Reporting – Week 4'. Each of these sections contain solutions to the bullet points in the Project Requirements. Essentially, each individual bullet point (if containing a question) is answered by a section with a heading that is 'underlined'. The actual heading relates more to the solution of the question than the question itself. However, it should be relatively easy to understand which section answers which question since it is written in order of requirements.

The programming language of choice used to accomplish the Analysis is Python 3 through the means of Anaconda's Jupyter Notebook. Python was chosen since no criteria was given pertaining to which language to use and I believe Python to be the most common language utilized by most when it comes to Data Science and Machine Learning applications. The snapshots from Jupyter Notebook contain the code followed by the output. They are labelled sequentially and are merged in with their respective explanation. This is better to follow along than to separate the code, output and explanation since when you see a screen shot and its explanation is physically close to it, the visuals are still fresh in the mind so the writing becomes easier to understand. The explanations are essentially the contents of the 'underlined' sections mentioned above. They are usually at the bottom of the selected figures that pertain to a certain solution.

Finally, a Conclusion section is presented at the end of the report (excluding References). This section highlights final remarks on the process to complete the project, as well as the final results obtained. The References section (last page) details external references utilized using APA format for online resources, presented in the form of footnotes.

Background

This dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. The objective of the dataset is to diagnostically predict whether or not a patient has diabetes, based on certain diagnostic measurements included in the dataset. Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females at least 21 years old of Pima Indian heritage.

Problem Statement

Build a model to accurately predict whether the patients in the dataset have diabetes or not?

Analysis

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

```
data = pd.read_csv('health care diabetes.csv')
data.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Figure 1.

Figure 1 details the libraries that will be used to accomplish the requirements. In addition, it read the dataset csv file in a variable named ‘data’. The first 5 rows are displayed in the figure.

Data Exploration – Week 1

```
plt.hist(data['Glucose'], 25) # 25 bins chosen through trial and error for best visualization
plt.title('Glucose Histogram - Raw Data')
```

```
Text(0.5, 1.0, 'Glucose Histogram - Raw Data')
```

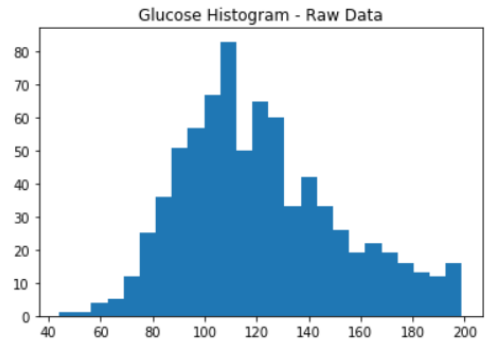

Figure 2.

```
plt.hist(data['BloodPressure'], 25)
plt.title('Blood Pressure Histogram - Raw Data')
```

```
Text(0.5, 1.0, 'Blood Pressure Histogram - Raw Data')
```

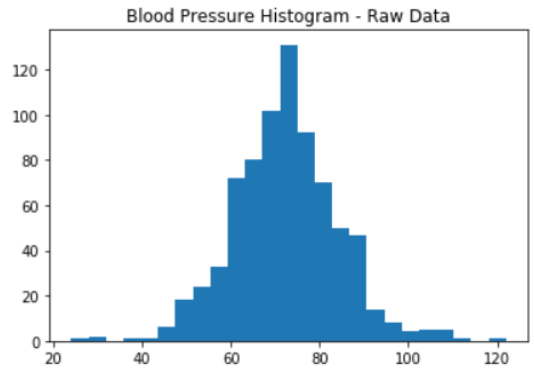

Figure 3.

```
plt.hist(data['SkinThickness'], 25)
plt.title('Skin Thickness Histogram - Raw Data')
```

```
Text(0.5, 1.0, 'Skin Thickness Histogram - Raw Data')
```

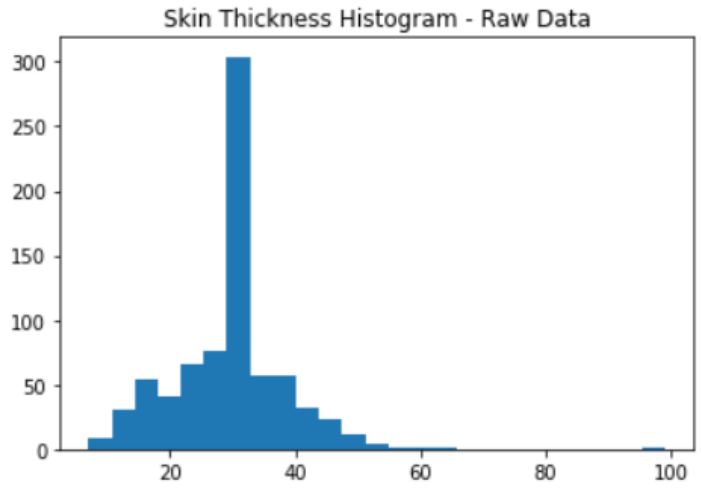

Figure 4.

```
plt.hist(data['Insulin'], 25)
plt.title('Insulin Histogram - Raw Data')
```

```
Text(0.5, 1.0, 'Insulin Histogram - Raw Data')
```

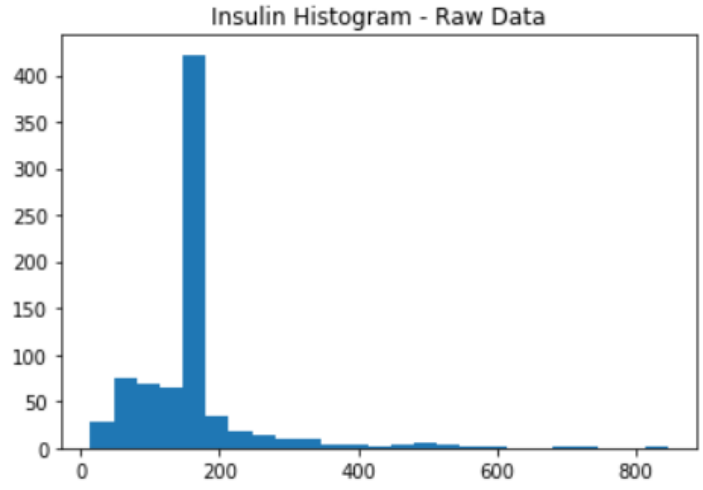

Figure 5.

```
plt.hist(data['BMI'], 25)
plt.title('BMI Histogram - Raw Data')
```

```
Text(0.5, 1.0, 'BMI Histogram - Raw Data')
```

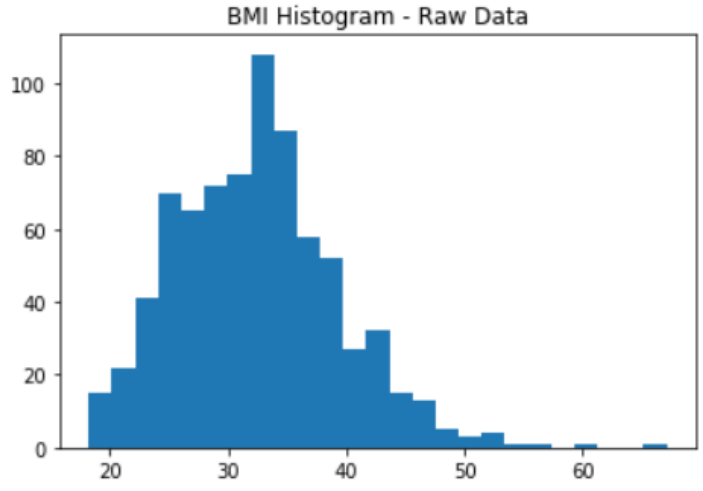

Figure 6.

```
print(data.replace(0, np.NaN).mean()) # calculation of mean by default excludes NaN
```

```
Pregnancies                 4.494673
Glucose                   121.686763
BloodPressure              72.405184
SkinThickness              29.153420
Insulin                   155.548223
BMI                        32.457464
DiabetesPedigreeFunction    0.471876
Age                        33.240885
Outcome                     1.000000
dtype: float64
```

Figure 7.

Histograms Interpretation

The above histograms (Figure 2 – 6) show the distributions of the variables in the dataset that have a non-meaningful zero value [1]. This is due to the fact that having a reported 0 in any of these fields is not physically plausible. Hence, their distributions were analyzed to discern the most reasonable value to impute these missing values with. Also note a meaningful, numeric value is needed to train the prediction models which will be discussed later in the report.

Upon completing the analysis, it was decided that the column/variable mean would be the most appropriate value to impute the missing values with. The mode (most frequent value) was also a viable option, however since some of the distributions were skewed (i.e. Glucose, Insulin and BMI), the column average would give a better representation of all variable values. The variable means were recorded in Figure 7, with 0 values being replaced by NaN (NaN excluded from mean calculations by default). All variables in the dataset are show here, but the only ones that really matter are from Glucose to BMI. These are the average that will be imputed for missing values in the mentioned 5 columns.

```
# change 0 value in 5 specified columns to NaN to use fillna() method
data['Glucose'].replace(0, np.NaN, inplace=True)
data['BloodPressure'].replace(0, np.NaN, inplace=True)
data['SkinThickness'].replace(0, np.NaN, inplace=True)
data['Insulin'].replace(0, np.NaN, inplace=True)
data['BMI'].replace(0, np.NaN, inplace=True)
```

```
data.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148.0	72.0	35.0	NaN	33.6	0.627	50	1
1	1	85.0	66.0	29.0	NaN	26.6	0.351	31	0
2	8	183.0	64.0	NaN	NaN	23.3	0.672	32	1
3	1	89.0	66.0	23.0	94.0	28.1	0.167	21	0
4	0	137.0	40.0	35.0	168.0	43.1	2.288	33	1

Figure 8.

```
# Choose to impute with mean instead of mode since distribution might be skewed and most frequent values
# may not symbolize overall distribution
data['Glucose'].fillna(data['Glucose'].mean(), inplace=True)
data['BloodPressure'].fillna(data['BloodPressure'].mean(), inplace=True)
data['SkinThickness'].fillna(data['SkinThickness'].mean(), inplace=True)
data['Insulin'].fillna(data['Insulin'].mean(), inplace=True)
data['BMI'].fillna(data['BMI'].mean(), inplace=True)
```

```
data.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148.0	72.0	35.00000	155.548223	33.6	0.627	50	1
1	1	85.0	66.0	29.00000	155.548223	26.6	0.351	31	0
2	8	183.0	64.0	29.15342	155.548223	23.3	0.672	32	1
3	1	89.0	66.0	23.00000	94.000000	28.1	0.167	21	0
4	0	137.0	40.0	35.00000	168.000000	43.1	2.288	33	1

Figure 9.

Imputing Column Means for Missing Values

First, all 0s in the Glucose, Blood Pressure, Skin Thickness, Insulin and BMI columns were converted to NaN (as seen in Figure 8). This was done so the fillna() method could be used to impute the missing values with the respected column means (as seen in Figure 9). Note that once the imputation is completed, the columns data types are automatically converted to 'float64 even if they were initially of 'int64' type.

```
data['Glucose'].value_counts().plot.bar()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x273e08dfa08>
```

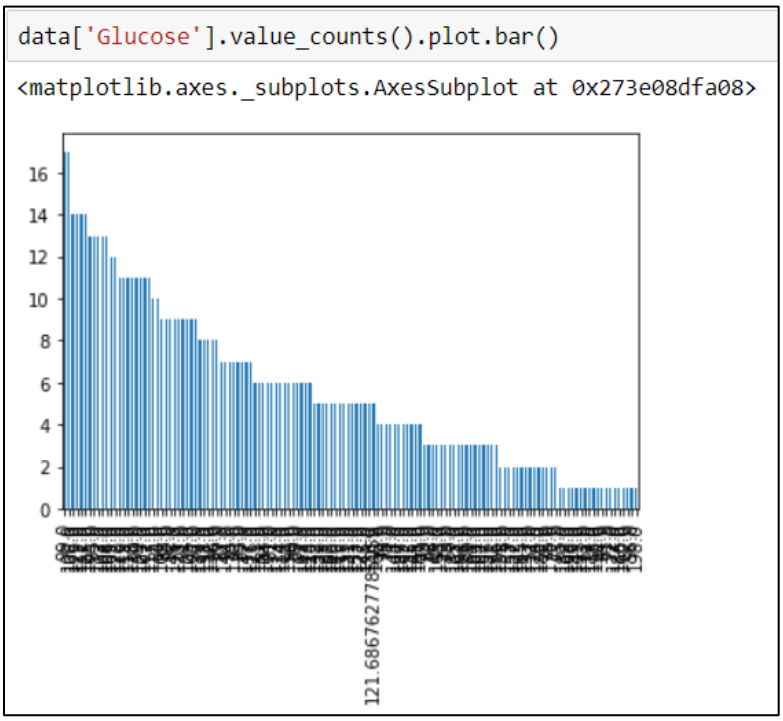

Figure 10.

```
data['BloodPressure'].value_counts().plot.bar()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x273e0cd64c8>
```

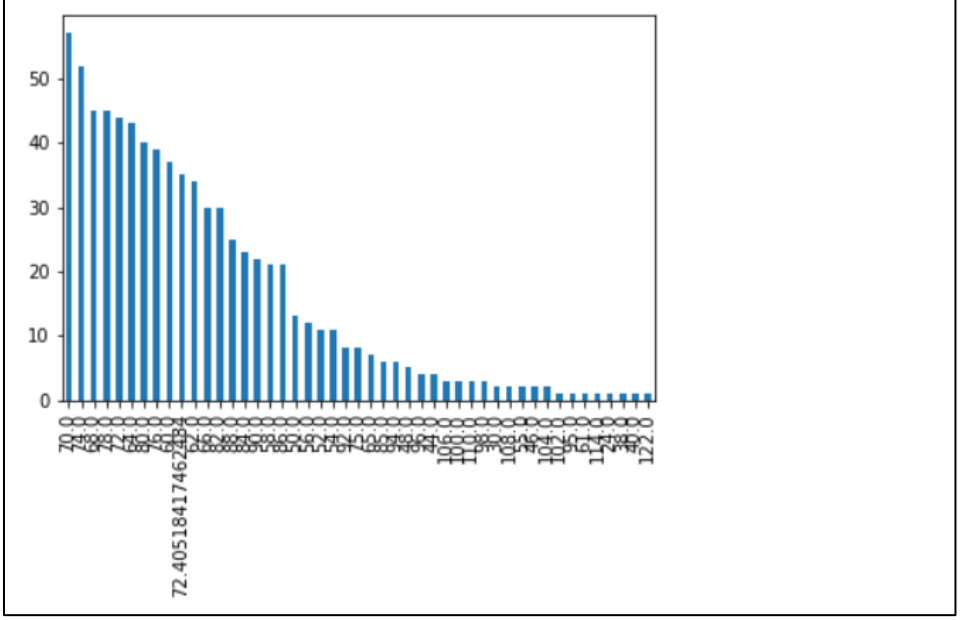

Figure 11.

```
data['SkinThickness'].value_counts().plot.bar()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x273e1e2e148>
```

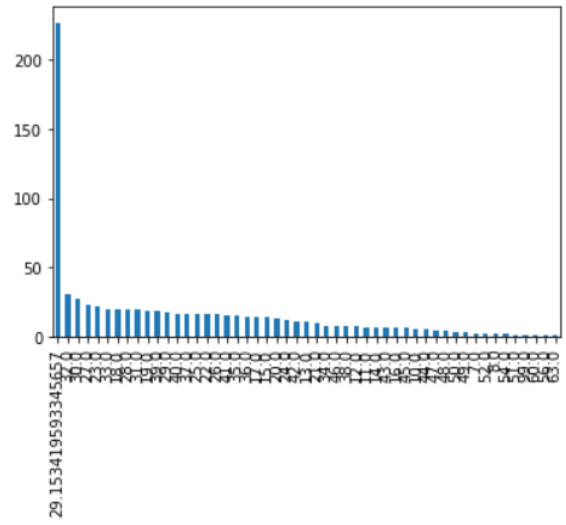

Figure 12.

```
data['Insulin'].value_counts().plot.bar()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x273e1f70e88>
```

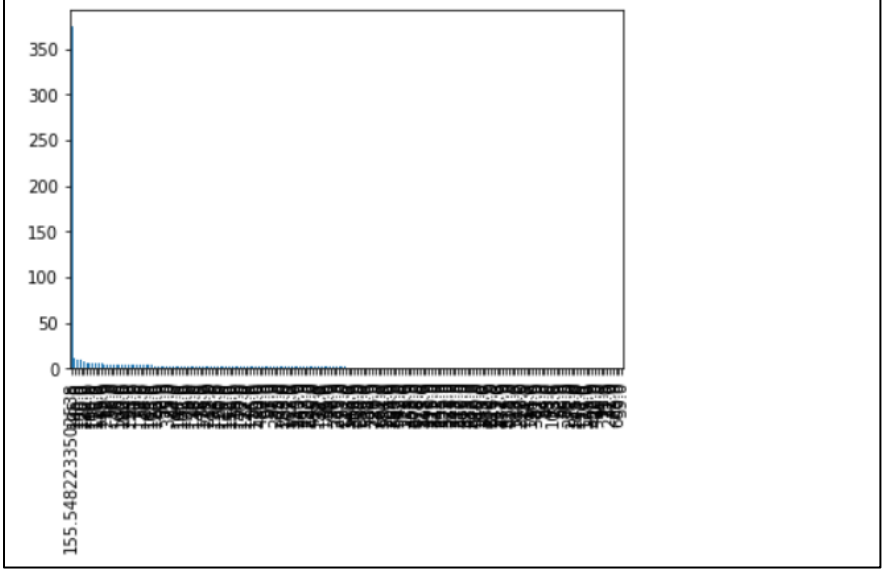

Figure 13.

```
data['BMI'].value_counts().plot.bar()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x273e2404e08>
```

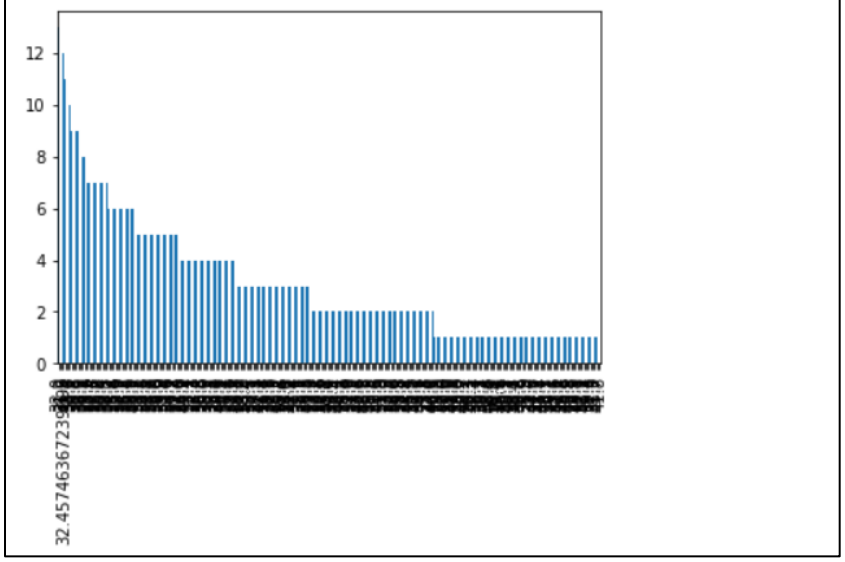

Figure 14.

```
data['Pregnancies'].value_counts().plot.bar()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x273e2cdafc8>
```

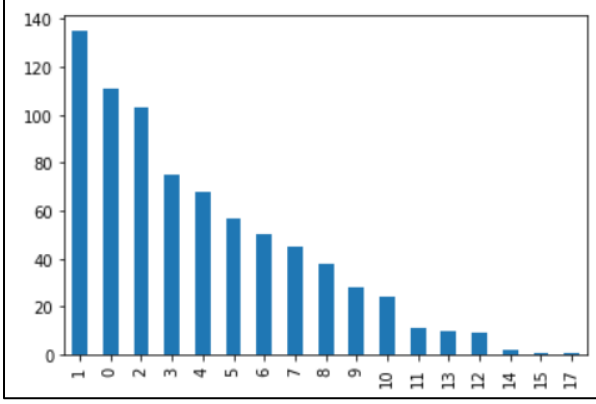

Figure 15.

```
data['DiabetesPedigreeFunction'].value_counts().plot.bar()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x273e2d8dd48>
```

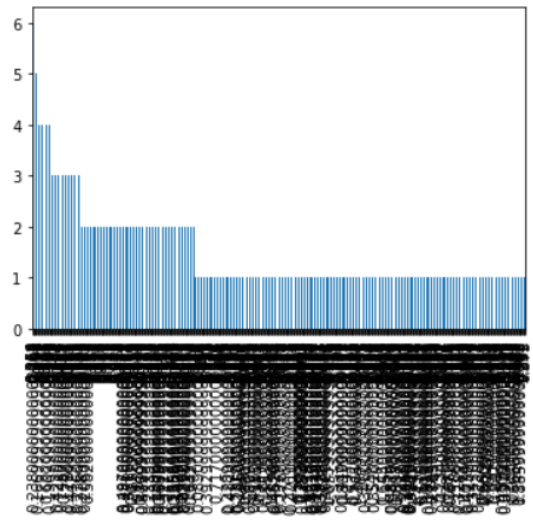

Figure 16.

```
data['Age'].value_counts().plot.bar()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x273e3321fc8>
```

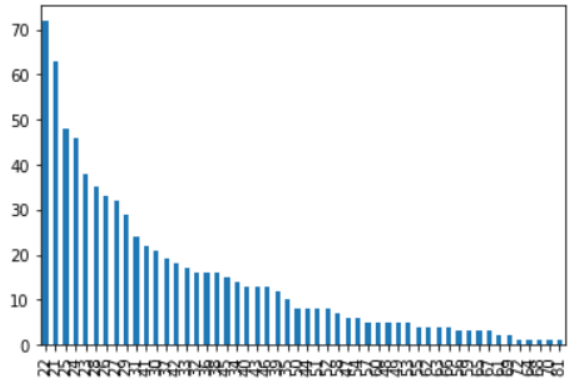

Figure 17.

Count Plots

Figures 10 – 17 display the count plots for the variables: Glucose, Blood Pressure, Skin Thickness, Insulin, BMI, Pregnancies, Diabetes Pedigree Function and Age, respectively. Note that due to the imputation conducted prior, the visuals shown here might not indicate a perfect representation of the 'actual' (original) reported data. An important point to note on the plots with imputed values are the count of the imputed (mean) value. The prior histograms with the most frequent value being 0 translate to the imputed mean value. The plots of the 5 imputed variables also display the mean very clearly, so it can be seen how frequent it occurs in comparison to the rest of values. Some of the plots may also be hard to read (i.e. Diabetes Pedigree Function plot). This is because there are a large range of values contained in the variable. Hence, I included the value counts of each variable in Figures 18 – 25, respectively as same with Figures 10 – 17. This is done to give a better idea of the value counts by numbers instead of lines. Note: if the variable contains a large range of values, only the beginning and end of the range is shown. This serves to provide a more concrete view of the data distribution (after imputation).

```
data['Glucose'].value_counts()
```

```
99.0     17
100.0    17
106.0    14
129.0    14
111.0    14
         ..
186.0     1
160.0     1
67.0      1
56.0      1
198.0     1
Name: Glucose, Length: 136, dtype: int64
```

Figure 18.

```
data['BloodPressure'].value_counts()
```

```
70.000000    57
74.000000    52
68.000000    45
78.000000    45
72.000000    44
64.000000    43
80.000000    40
76.000000    39
60.000000    37
72.405184    35
62.000000    34
66.000000    30
82.000000    30
88.000000    25
84.000000    23
90.000000    22
58.000000    21
86.000000    21
50.000000    13
56.000000    12
52.000000    11
54.000000    11
92.000000     8
75.000000     8
65.000000     7
85.000000     6
94.000000     6
48.000000     5
96.000000     4
44.000000     4
```

Figure 19.

```
data['SkinThickness'].value_counts()
```

```
29.15342    227
32.00000     31
30.00000     27
27.00000     23
23.00000     22
33.00000     20
18.00000     20
28.00000     20
31.00000     19
19.00000     18
39.00000     18
29.00000     17
40.00000     16
37.00000     16
25.00000     16
22.00000     16
26.00000     16
41.00000     15
35.00000     15
36.00000     14
17.00000     14
15.00000     14
20.00000     13
24.00000     12
42.00000     11
```

Figure 20.

```
data['Insulin'].value_counts()
```

```
155.548223    374
105.000000     11
140.000000      9
130.000000      9
120.000000      8
              ...
272.000000      1
41.000000       1
25.000000       1
600.000000      1
59.000000       1
Name: Insulin, Length: 186, dtype: int64
```

Figure 21.

```
data['BMI'].value_counts()
```

```
32.000000    13
31.200000    12
31.600000    12
32.457464    11
32.400000    10
             ..
32.100000     1
52.900000     1
31.300000     1
45.700000     1
41.800000     1
Name: BMI, Length: 248, dtype: int64
```

Figure 22.

```
data['Pregnancies'].value_counts()
```

```
1     135
0     111
2     103
3      75
4      68
5      57
6      50
7      45
8      38
9      28
10     24
11     11
13     10
12      9
14      2
15      1
17      1
Name: Pregnancies, dtype: int64
```

Figure 23.

```
data['DiabetesPedigreeFunction'].value_counts()
```

```
0.254    6
0.258    6
0.259    5
0.238    5
0.207    5
        ..
0.886    1
0.804    1
1.251    1
0.382    1
0.375    1
Name: DiabetesPedigreeFunction, Length: 517, dtype: int64
```

Figure 24.

```
data['Age'].value_counts()
```

```
22    72
21    63
25    48
24    46
23    38
28    35
26    33
27    32
29    29
31    24
41    22
30    21
37    19
42    18
33    17
32    16
36    16
38    16
45    15
```

Figure 25.

Data Types

Since the count plots don't really show the variable's data type, I decided to display them separately using the dtypes property. The data in Figure 26 shows the column data types after the imputation of missing data, while Figure 27 shows before. Both of these indicate the dataset contains int and float types, however the most notable change is the conversion to float64 types on the imputed columns. This is due to the result of mean calculations being of float type, and if any number in an integer column becomes a float, the whole column's data type then becomes float. The target (Outcome) column is also recorded as integer since it only contains 1s and 0s. However, these are actually just class labels for if a patient has diabetes or not. Therefore, this column is actually of binary class type.

```
# after imputation
print(data.dtypes)
```

```
Pregnancies                 int64
Glucose                   float64
BloodPressure             float64
SkinThickness             float64
Insulin                   float64
BMI                       float64
DiabetesPedigreeFunction  float64
Age                         int64
Outcome                     int64
dtype: object
```

Figure 26.

```
# before imputation
print(data.dtypes)
```

```
Pregnancies                 int64
Glucose                     int64
BloodPressure               int64
SkinThickness               int64
Insulin                     int64
BMI                       float64
DiabetesPedigreeFunction  float64
Age                         int64
Outcome                     int64
dtype: object
```

Figure 27.

Data Exploration – Week 2

```
data['Outcome'].value_counts() # 1: 35%, 0: 65%
```

```
0    500
1    268
Name: Outcome, dtype: int64
```

Figure 28.

```
data['Outcome'].value_counts().plot.bar(title='Outcome Counts')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x273e7003e48>
```

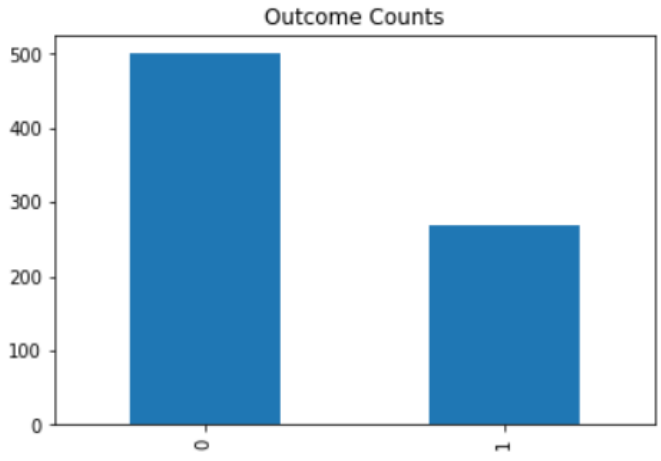

Figure 29.

Balance of Data based on Count of Outcome

According to Figure 28, 500 patients don't have diabetes while 268 do. This difference is visibly noticeable in the count plot of Outcomes in Figure 29. This shows that around 65% of the sample don't have diabetes, while around 35% do. This is a good sign that the majority of people do not have diabetes (according to the sample dataset).

As can be seen from the Outcome distribution, the values are binary (only contains 2 values). Hence, a binary 'classification' model will be used to predict future assessments of diabetes. It will be a supervised classification algorithm since the data contains a label (Outcome variable) and is of the 'class' type. This will be further discussed in the 'Data Modelling – Week 3' section.

```
plt.scatter(data['Pregnancies'], data['Glucose'], marker='o')
plt.title('Glucose vs Pregnancies')
plt.xlabel('Pregnancies')
plt.ylabel('Glucose')
plt.show()
```

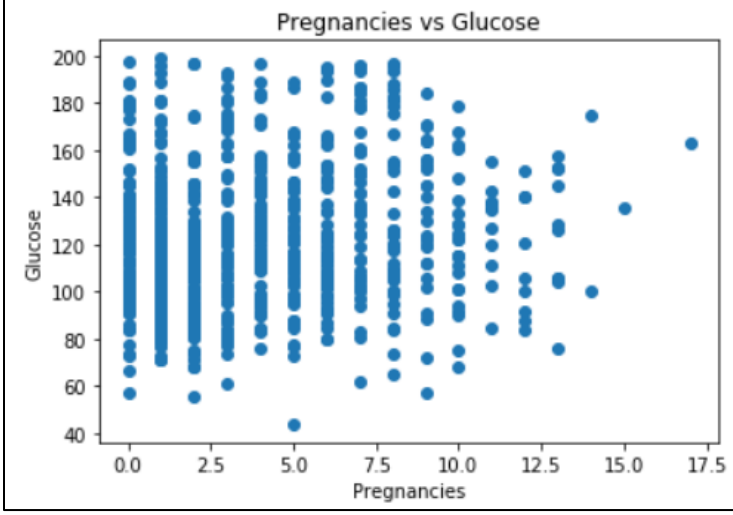

Figure 30.

```
plt.scatter(data['Pregnancies'], data['BloodPressure'], marker='o')
plt.title('Blood Pressure vs Pregnancies')
plt.xlabel('Pregnancies')
plt.ylabel('Blood Pressure')
plt.show()
```

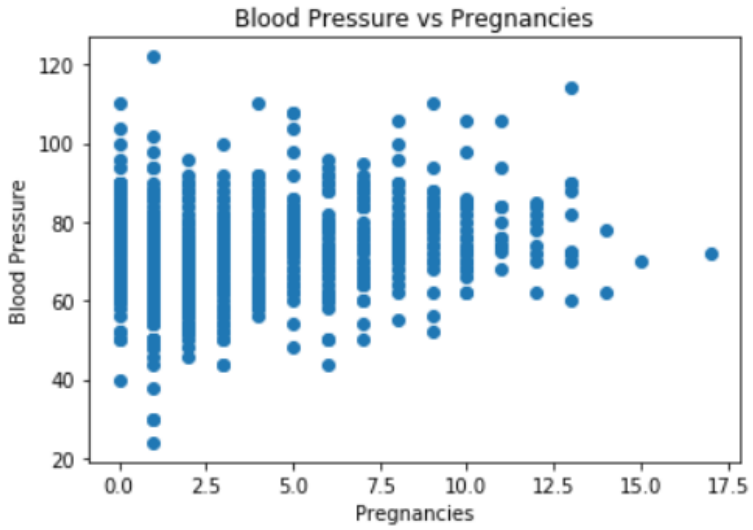

Figure 31.

```
plt.scatter(data['Pregnancies'], data['SkinThickness'], marker='o')
plt.title('Skin Thickness vs Pregnancies')
plt.xlabel('Pregnancies')
plt.ylabel('Skin Thickness')
plt.show()
```

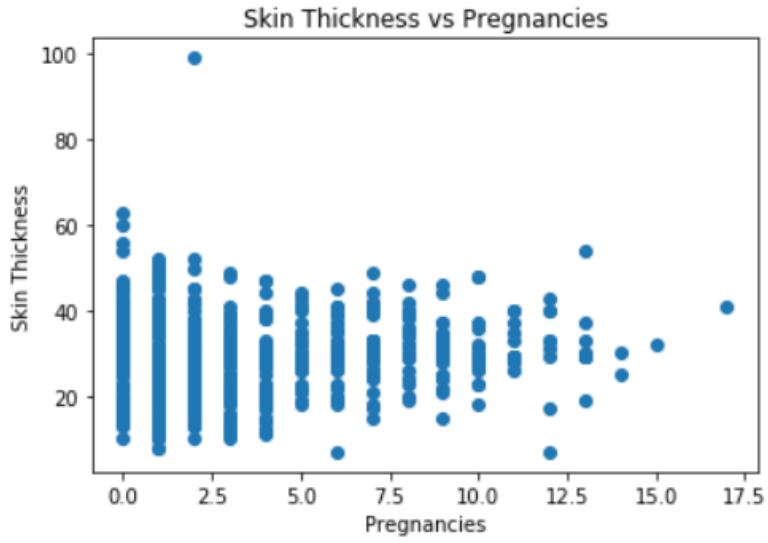

Figure 32.

```
plt.scatter(data['Pregnancies'], data['Insulin'], marker='o')
plt.title('Insulin vs Pregnancies')
plt.xlabel('Pregnancies')
plt.ylabel('Insulin')
plt.show()
```

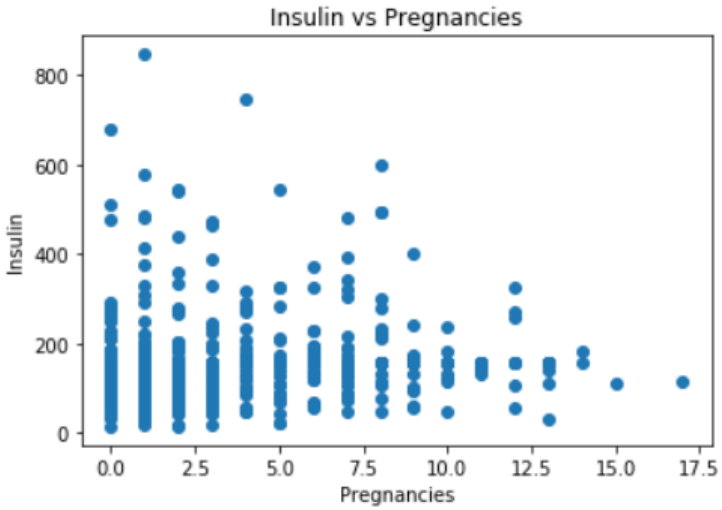

Figure 33.

```
plt.scatter(data['Pregnancies'], data['BMI'], marker='o')
plt.title('BMI vs Pregnancies')
plt.xlabel('Pregnancies')
plt.ylabel('BMI')
plt.show()
```

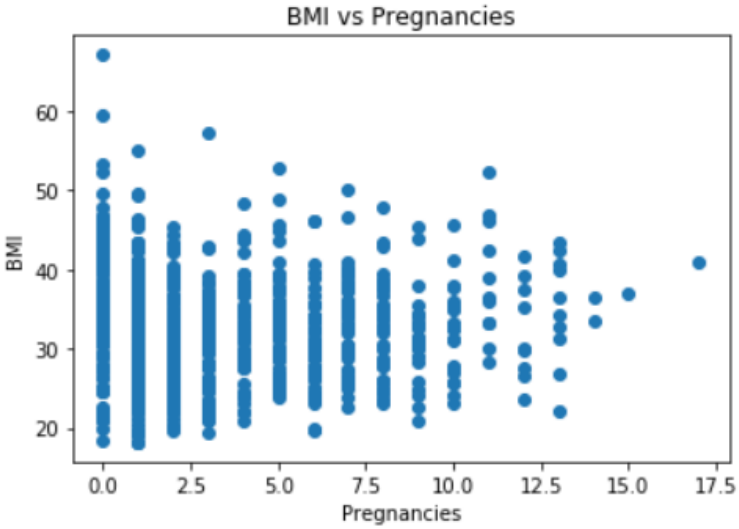

Figure 34.

```
plt.scatter(data['Pregnancies'], data['DiabetesPedigreeFunction'], marker='o')
plt.title('Diabetes Pedigree Function vs Pregnancies')
plt.xlabel('Pregnancies')
plt.ylabel('Diabetes Pedigree Function')
plt.show()
```

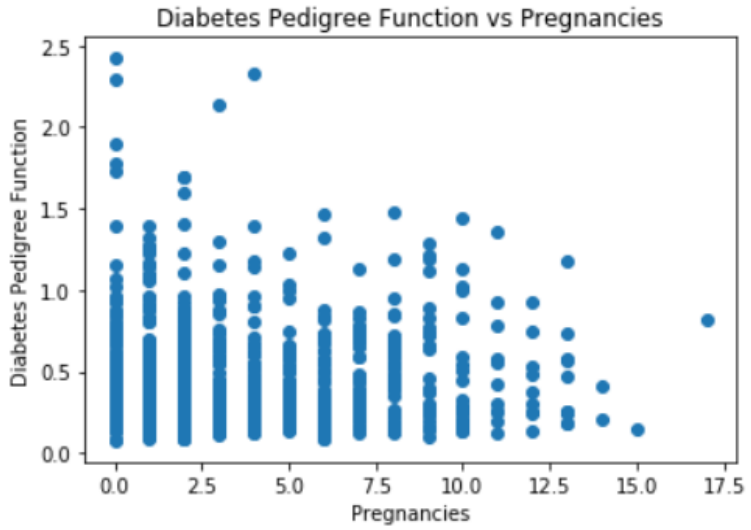

Figure 35.

```
plt.scatter(data['Pregnancies'], data['Age'], marker='o')
plt.title('Age vs Pregnancies')
plt.xlabel('Pregnancies')
plt.ylabel('Age')
plt.show()
```

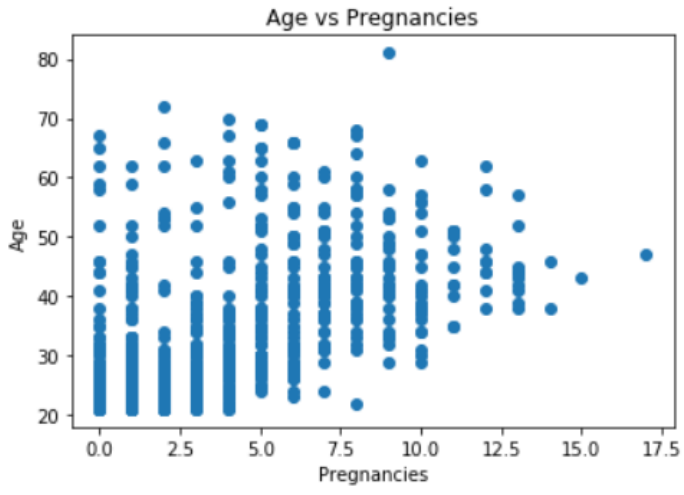

Figure 36.

```
plt.scatter(data['Glucose'], data['BloodPressure'], marker='o')
plt.title('Blood Pressure vs Glucose')
plt.xlabel('Glucose')
plt.ylabel('Blood Pressure')
plt.show()
```

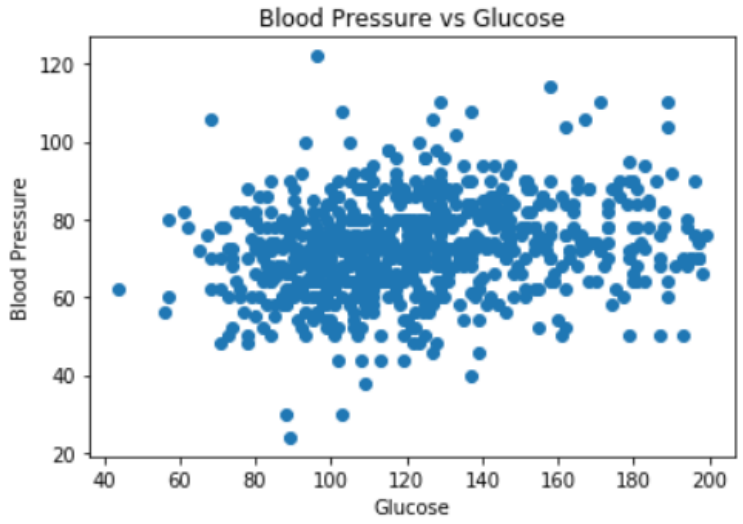

Figure 37.

```
plt.scatter(data['Glucose'], data['SkinThickness'], marker='o')
plt.title('Skin Thickness vs Glucose')
plt.xlabel('Glucose')
plt.ylabel('Skin Thickness')
plt.show()
```

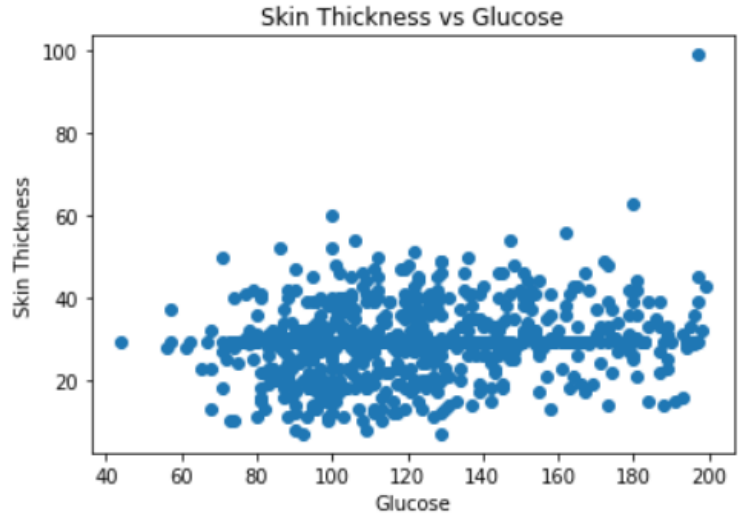

Figure 38.

```
plt.scatter(data['Glucose'], data['Insulin'], marker='o')
plt.title('Insulin vs Glucose')
plt.xlabel('Glucose')
plt.ylabel('Insulin')
plt.show()
```

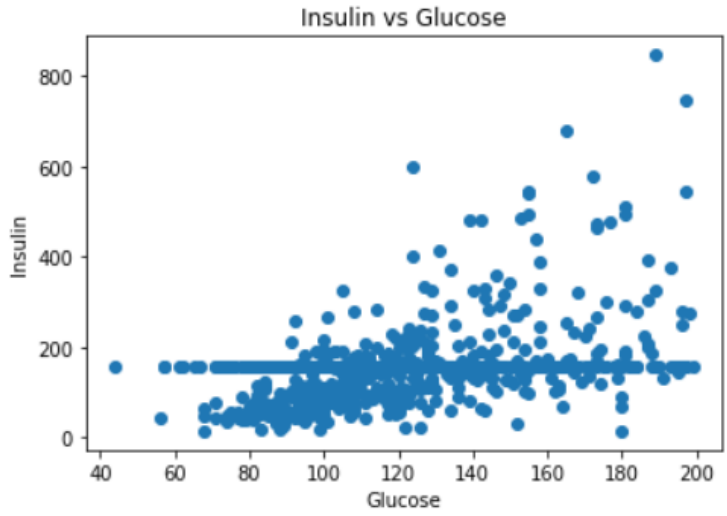

Figure 39.

```
plt.scatter(data['Glucose'], data['BMI'], marker='o')
plt.title('BMI vs Glucose')
plt.xlabel('Glucose')
plt.ylabel('BMI')
plt.show()
```

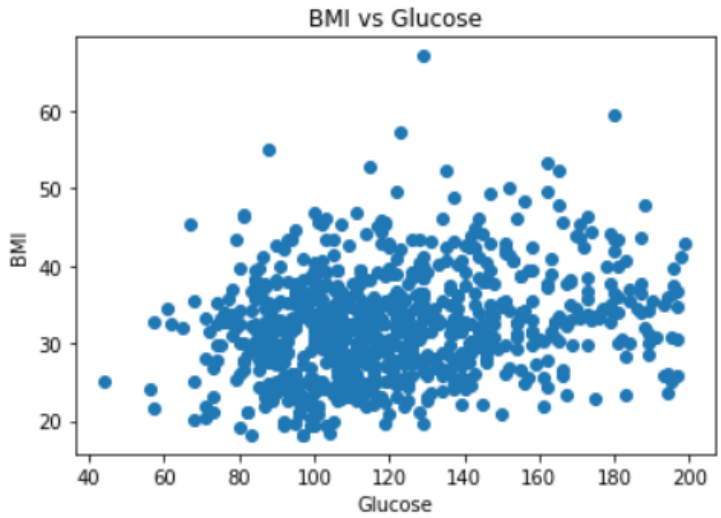

Figure 40.

```
plt.scatter(data['Glucose'], data['DiabetesPedigreeFunction'], marker='o')
plt.title('Diabetes Pedigree Function vs Glucose')
plt.xlabel('Glucose')
plt.ylabel('Diabetes Pedigree Function')
plt.show()
```

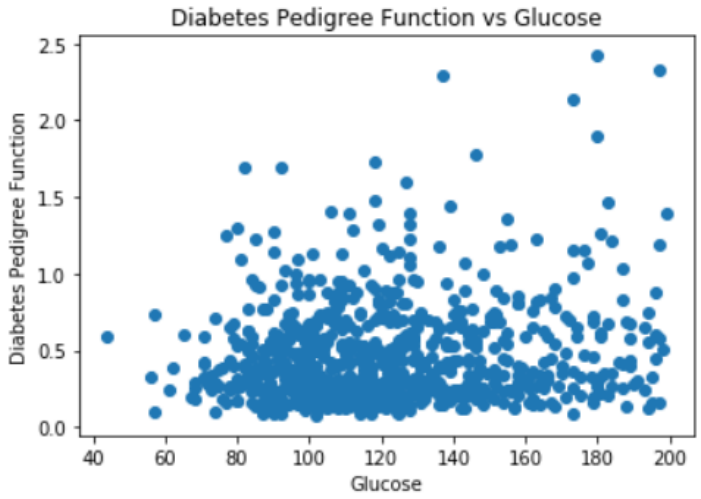

Figure 41.

```
plt.scatter(data['Glucose'], data['Age'], marker='o')
plt.title('Age vs Glucose')
plt.xlabel('Glucose')
plt.ylabel('Age')
plt.show()
```

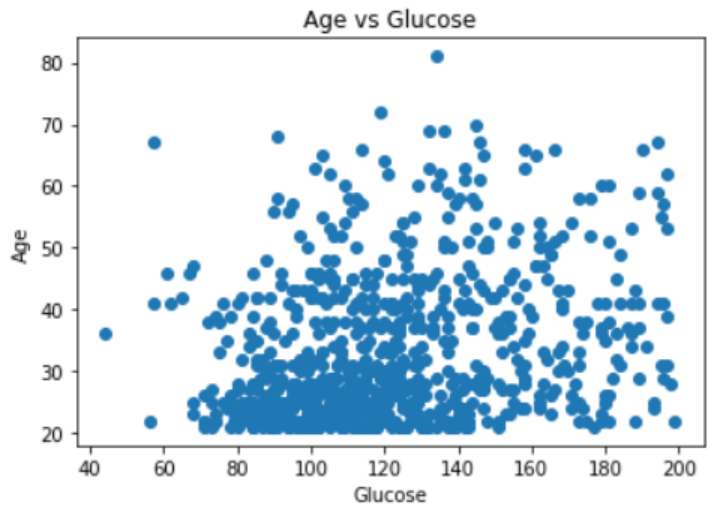

Figure 42.

```
plt.scatter(data['BloodPressure'], data['SkinThickness'], marker='o')
plt.title('Skin Thickness vs Blood Pressure')
plt.xlabel('Blood Pressure')
plt.ylabel('Skin Thickness')
plt.show()
```

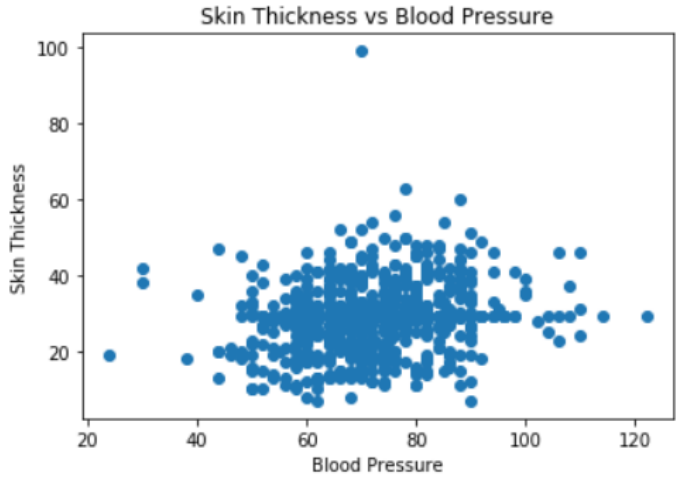

Figure 43.

```
plt.scatter(data['BloodPressure'], data['Insulin'], marker='o')
plt.title('Insulin vs Blood Pressure')
plt.xlabel('Blood Pressure')
plt.ylabel('Insulin')
plt.show()
```

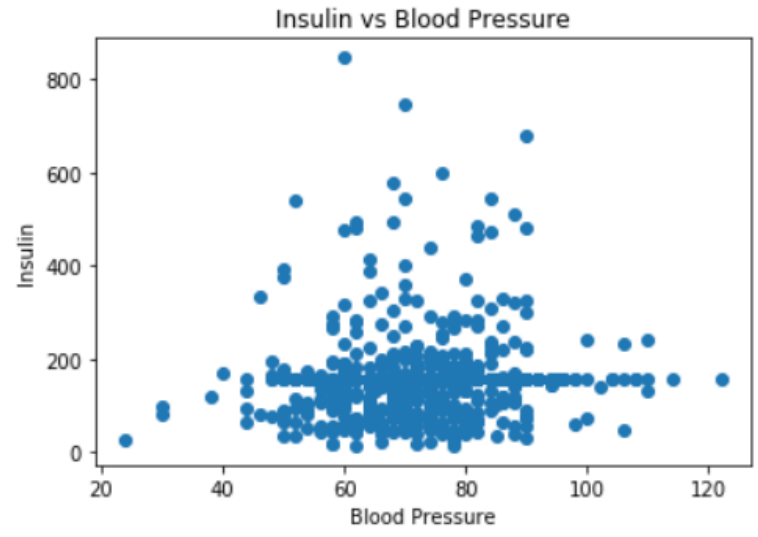

Figure 44.

```
plt.scatter(data['BloodPressure'], data['BMI'], marker='o')
plt.title('BMI vs Blood Pressure')
plt.xlabel('Blood Pressure')
plt.ylabel('BMI')
plt.show()
```

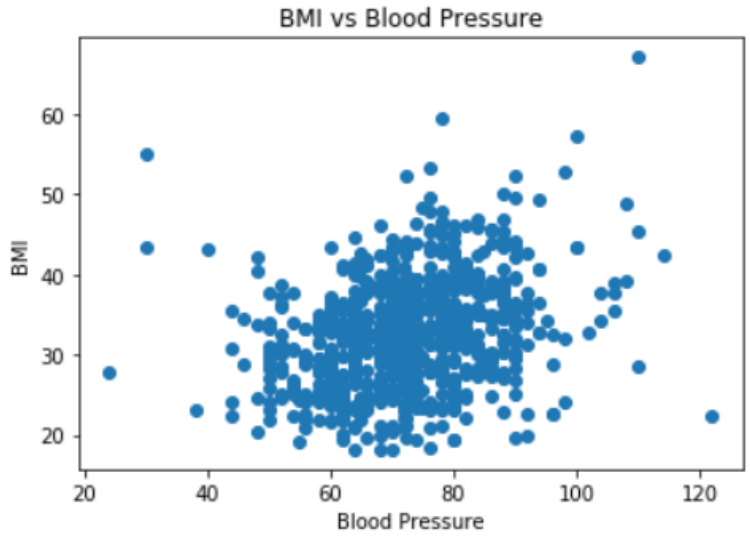

Figure 45.

```
plt.scatter(data['BloodPressure'], data['DiabetesPedigreeFunction'], marker='o')
plt.title('Diabetes Pedigree Function vs Blood Pressure')
plt.xlabel('Blood Pressure')
plt.ylabel('Diabetes Pedigree Function')
plt.show()
```

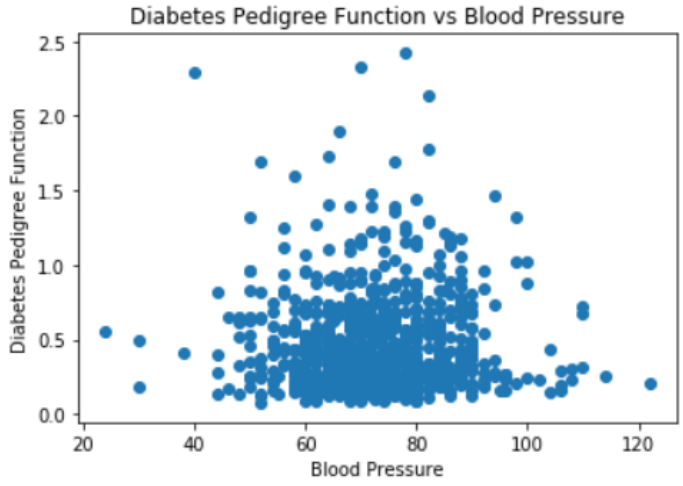

Figure 46.

```
plt.scatter(data['BloodPressure'], data['Age'], marker='o')
plt.title('Age vs Blood Pressure')
plt.xlabel('Blood Pressure')
plt.ylabel('Age')
plt.show()
```

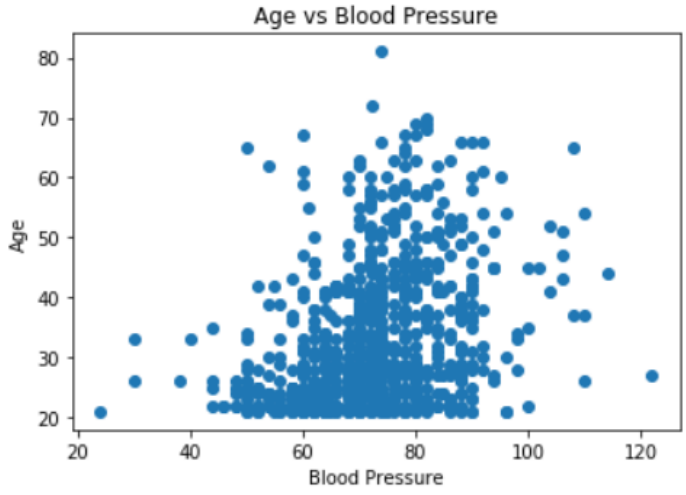

Figure 47.

```
plt.scatter(data['SkinThickness'], data['Insulin'], marker='o')
plt.title('Insulin vs Skin Thickness')
plt.xlabel('Skin Thickness')
plt.ylabel('Insulin')
plt.show()
```

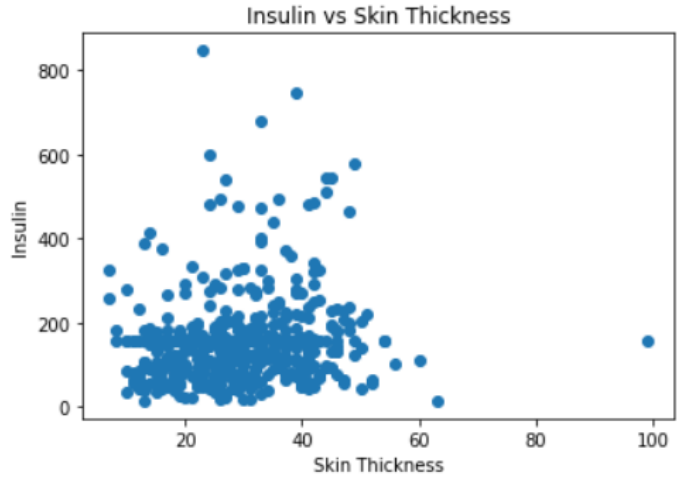

Figure 48.

```
plt.scatter(data['SkinThickness'], data['BMI'], marker='o')
plt.title('BMI vs Skin Thickness')
plt.xlabel('Skin Thickness')
plt.ylabel('BMI')
plt.show()
```

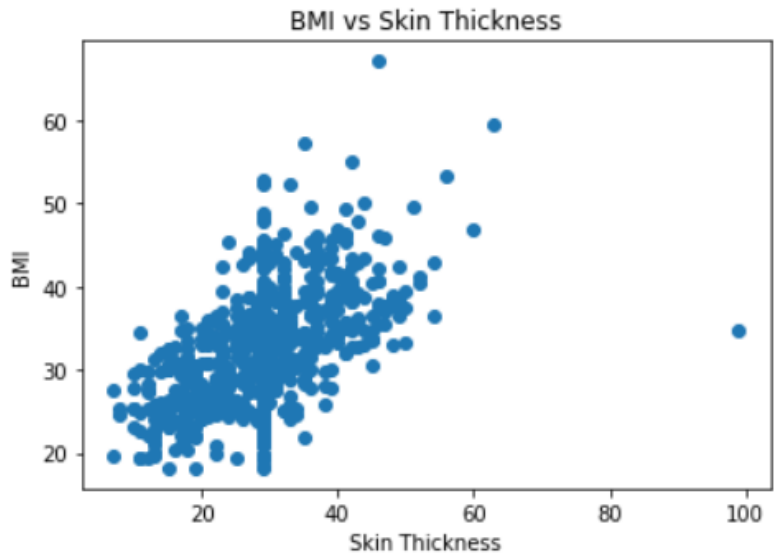

Figure 49.

```
plt.scatter(data['SkinThickness'], data['DiabetesPedigreeFunction'], marker='o')
plt.title('Diabetes Pedigree Function vs Skin Thickness')
plt.xlabel('Skin Thickness')
plt.ylabel('Diabetes Pedigree Function')
plt.show()
```

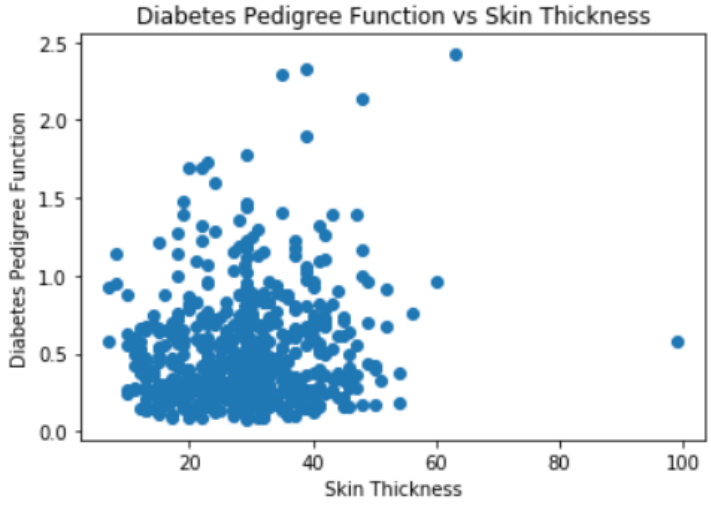

Figure 50.

```
plt.scatter(data['SkinThickness'], data['Age'], marker='o')
plt.title('Age vs Skin Thickness')
plt.xlabel('Skin Thickness')
plt.ylabel('Age')
plt.show()
```

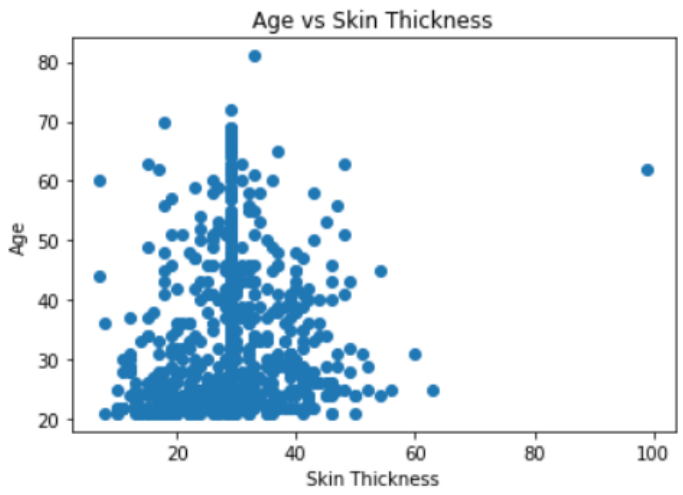

Figure 51.

```
plt.scatter(data['Insulin'], data['BMI'], marker='o')
plt.title('BMI vs Insulin')
plt.xlabel('Insulin')
plt.ylabel('BMI')
plt.show()
```

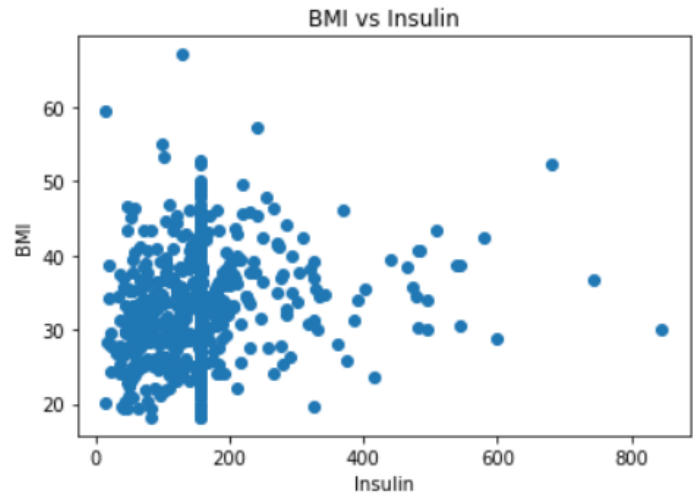

Figure 52.

```
plt.scatter(data['Insulin'], data['DiabetesPedigreeFunction'], marker='o')
plt.title('Diabetes Pedigree Function vs Insulin')
plt.xlabel('Insulin')
plt.ylabel('Diabetes Pedigree Function')
plt.show()
```

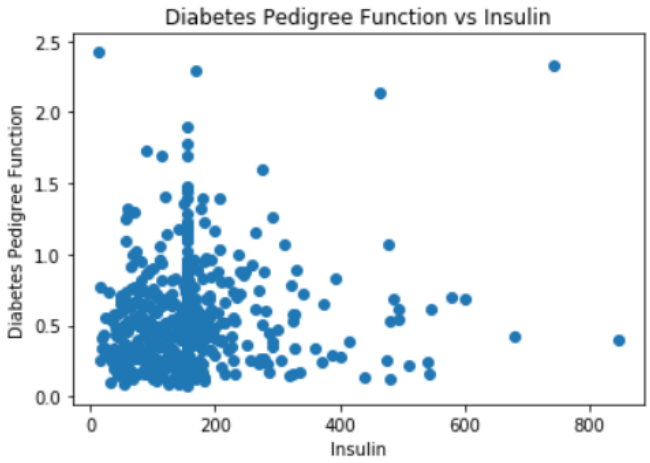

Figure 53.

```
plt.scatter(data['Insulin'], data['Age'], marker='o')
plt.title('Age vs Insulin')
plt.xlabel('Insulin')
plt.ylabel('Age')
plt.show()
```

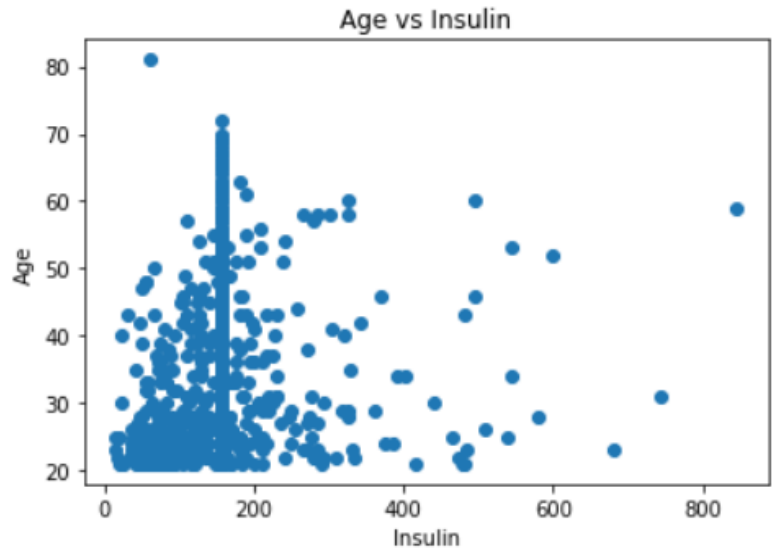

Figure 54.

```
plt.scatter(data['BMI'], data['DiabetesPedigreeFunction'], marker='o')
plt.title('Diabetes Pedigree Function vs BMI')
plt.xlabel('BMI')
plt.ylabel('Diabetes Pedigree Function')
plt.show()
```

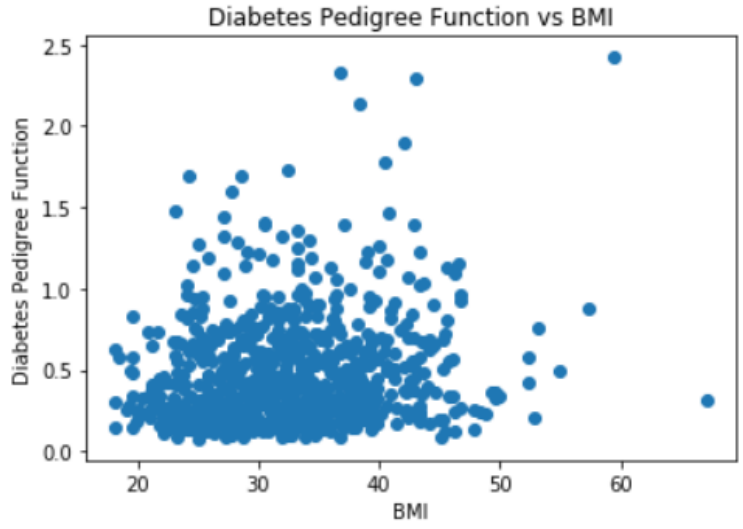

Figure 55.

```
plt.scatter(data['BMI'], data['Age'], marker='o')
plt.title('Age vs BMI')
plt.xlabel('BMI')
plt.ylabel('Age')
plt.show()
```

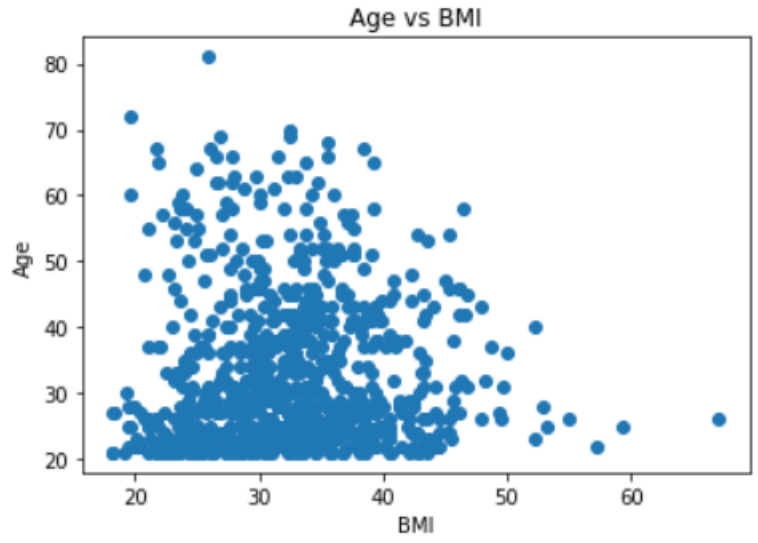

Figure 56.

```
plt.scatter(data['DiabetesPedigreeFunction'], data['Age'], marker='o')
plt.title('Age vs Diabetes Pedigree Function')
plt.xlabel('Diabetes Pedigree Function')
plt.ylabel('Age')
plt.show()
```

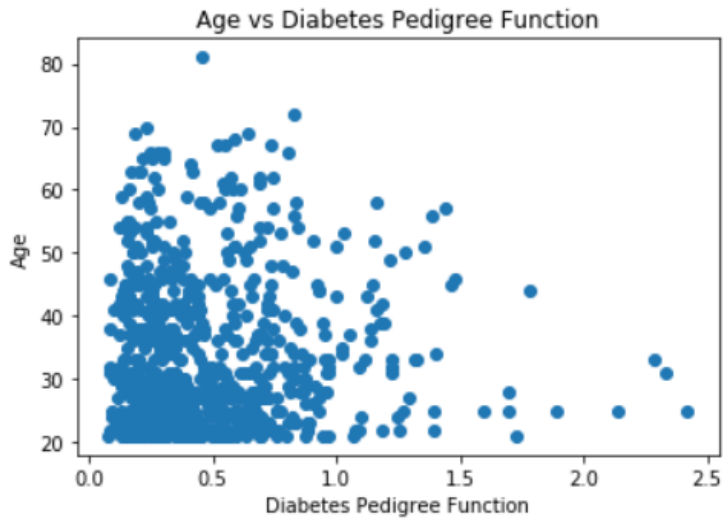

Figure 57.

Scatter Plots

Figures 30 to Figure 57 show the scatter plots highlighting the relationship between each unique pair of variables (excluding the target variable since it would visually have no meaning and thus make no sense). The plots display one of the variables on the x-axis and the other on the y-axis. The axis labels detail the respected variable of the pair.

Majority of the pairs of variables do not have a trend or pattern that can be extracted from their relationship. Here, a trend or pattern is defined as a constant or predictive outcome based on prior data. The only variables where a somewhat definitive trend may be observed is with the pairings of: Insulin vs Glucose, BMI vs Skin Thickness and Age vs Pregnancies. Other pairs of variables have a noticeable pattern but not enough data points conform to it in order to make the trend definitive. Such examples are: Diabetes Pedigree Function vs Glucose, BMI vs Blood Pressure and Diabetes Pedigree Function vs Skin Thickness.

The three plots with a definitive trend are logically sound. The higher your insulin level, the more probable your glucose level is as well. And the thicker your skin is (triceps' fold), the higher your body mass index (BMI) should be. The Age vs Pregnancies plot may be harder to tell, but there is a trend. As well as, it is reasonable to assume the younger someone is (still above 20), the more likely for them to want to have a child and be pregnant than when a female starts going through menopause (>45 years old).

For the lesser definitive trends, there is still logic behind their relationship. There is a high chance that one would consume or require more sugars/glucose if there is a higher amount of diabetic family members

(programmed in their genetics). The blood pressure would also be higher in higher BMI people, since the blood has to apply more force to flow through a body with greater obstacles. Due to genetics, a person with a higher Diabetes Pedigree Function would generally store more fat as well; hence, thicker skin.

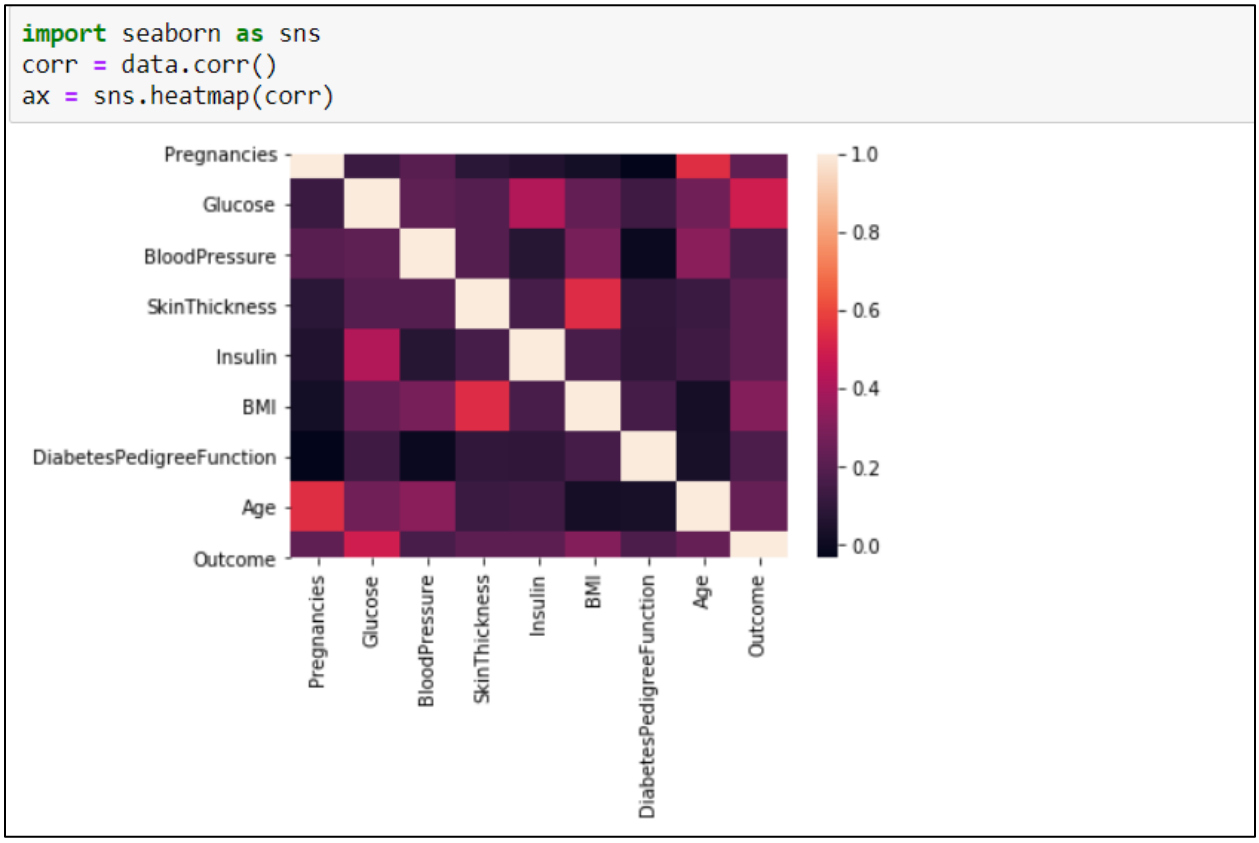

Figure 58.

Correlation Analysis

According to the heatmap in Figure 58, not many variables have a noticeable correlation (as mentioned in 'Scatter Plots' section). As previously mentioned, the Age and Pregnancies, Insulin and Glucose, as well as Skin Thickness and BMI are all correlated pairs. Probably the most important piece of information to take away from this heatmap is what most affects the chances of diabetes (the goal of this project). According to this, the glucose concentration impacts the chance of having diabetes the most. The BMI also slightly affects the Outcome as well. This makes sense since diabetes cause an increased level in glucose. Having a higher BMI also increases chances to have diabetes since the more excess weight one has, the more resistant your muscles and tissues become to your own insulin hormone (leads to diabetes).

Data Modelling – Week 3

```
# features
X = data.drop(columns = ['Outcome'])
X.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
0	6	148.0	72.0	35.00000	155.548223	33.6	0.627	50
1	1	85.0	66.0	29.00000	155.548223	26.6	0.351	31
2	8	183.0	64.0	29.15342	155.548223	23.3	0.672	32
3	1	89.0	66.0	23.00000	94.000000	28.1	0.167	21
4	0	137.0	40.0	35.00000	168.000000	43.1	2.288	33

```
# target
y = data['Outcome'].values
y[0:5]
```

```
array([1, 0, 1, 0, 1], dtype=int64)
```

Figure 59.

Model Building & Incorporating (K-Fold) Cross-Validation

Before discussing the cross-validation technique used, I want to highlight the contents of Figure 59. The features/predictor variables were set to include all columns except the 'Outcome', which is the target variable. Predictor variables were set to 'X' (dataframe), while the target variable was set to 'y' (array).

The construction of the prediction model and algorithm utilized will be further discussed in the next section, however certain points can be made about its general blueprint. Firstly, as previously mentioned, the model will be one such that it predicts a binary classification (0 or 1). The model that classifies the most correct, will achieve the highest accuracy score, and thus become the final model of choice. Note: although I say 'classification' model, a 'classifier' is not the only algorithm that can accomplish this. Essentially, any algorithm that classifies a data point will work (i.e. Logistic Regression Models).

Two validation frameworks were explored: the typical 'holdout' method (train_test_split) and cross-validation. To give a better understanding of why cross-validation was picked over the holdout method, I believe the meaning of 'validation' needs to be discussed first. Validation essentially means evaluating if results obtained correspond to results that were expected. In this way, validation is used to assess the accuracy of the prediction model to predict future classifications of diabetes.

In the standard 'holdout' method (most commonly implemented using the train_test_split module), the validation only depends on one split. So, the accuracy of the model is somewhat biased since it can favour certain values over others. In the cross-validation method however, specifically the k-fold, the data is split into k folds, iterating through a different fold each iteration. This gives a more concise interpretation of the model's accuracy since it uses different segments of the split for training and testing each iteration. Therefore, the model is trained on different data as well as tested on different data, so a higher model competency is achieved. K-Fold cross-validation was chosen since it is the most popular cross-validation

in terms of model building as well as intuitively makes sense. As a side note, in Scikit-learn 0.22.2, the default number of folds were increased from 3 to 5 [3].

```
# KNN Classifier (with k-fold cross-validation, n_neighbors=3)
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score

knn_cv = KNeighborsClassifier(n_neighbors=3)
cv_scores = cross_val_score(knn_cv, X, y, cv=5)

print(cv_scores)
print('cv_scores mean: {}'.format(np.mean(cv_scores)))
```

```
[0.69480519 0.65584416 0.75324675 0.73856209 0.70588235]
cv_scores mean: 0.7096681096681097
```

Figure 60.

```
# SVM with KFold cross validation
from sklearn.model_selection import KFold
from sklearn.svm import SVC

clf = SVC(gamma='auto')
scores = []

kf = KFold(n_splits=5)
X_array = X.to_numpy()

for train_index, test_index in kf.split(X_array):
    print('Train:', train_index, 'Test:', test_index)
    X_train, X_test = X_array[train_index], X_array[test_index]
    y_train, y_test = y[train_index], y[test_index]
    clf.fit(X_train, y_train)
    scores.append(clf.score(X_test, y_test))
```

```
Train: [154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171
 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189
 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225
 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243
 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261
 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279
 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297
 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315
```

Figure 61.

```
scores
```

```
[0.6493506493506493,
 0.5844155844155844,
 0.6298701298701299,
 0.7450980392156863,
 0.6470588235294118]
```

```
print('Average Score: {}'.format(np.mean(scores)))
```

```
Average Score: 0.6511586452762923
```

Figure 62.

```
# Gaussian Naive Bayes Classifier
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
NBScores = []

for train_index, test_index in kf.split(X_array):
    print('Train:', train_index, 'Test:', test_index)
    X_train, X_test = X_array[train_index], X_array[test_index]
    y_train, y_test = y[train_index], y[test_index]

    nbClf = gnb.fit(X_train, y_train)
    y_pred = nbClf.predict(X_test)

    y_correct = (y_test == y_pred).sum()
    records = X_test.shape[0]

    acc_score = y_correct/records

    NBScores.append(acc_score)
```

```
Train: [154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171
 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189
 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225
 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243
 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261
 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279
```

Figure 63.

```
print('Average Score: {}'.format(np.mean(NBScores)))
```

```
Average Score: 0.7474662592309651
```

Figure 64.

```
# Logistic Regression with built-in cross validation (stratified k-fold)
from sklearn.linear_model import LogisticRegressionCV
clf = LogisticRegressionCV(cv=5, random_state=0).fit(X, y)
clf.score(X, y)
```

```
D:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:947: ConvergenceWarning: lbfgs failed to converge.
Increase the number of iterations.
  "of iterations.", ConvergenceWarning)
D:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:947: ConvergenceWarning: lbfgs failed to converge.
Increase the number of iterations.
  "of iterations.", ConvergenceWarning)
D:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:947: ConvergenceWarning: lbfgs failed to converge.
Increase the number of iterations.
  "of iterations.", ConvergenceWarning)
D:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:947: ConvergenceWarning: lbfgs failed to converge.
Increase the number of iterations.
  "of iterations.", ConvergenceWarning)
D:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:947: ConvergenceWarning: lbfgs failed to converge.
Increase the number of iterations.
  "of iterations.", ConvergenceWarning)
D:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:947: ConvergenceWarning: lbfgs failed to converge.
Increase the number of iterations.
  "of iterations.", ConvergenceWarning)
D:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:947: ConvergenceWarning: lbfgs failed to converge.
Increase the number of iterations.
  "of iterations.", ConvergenceWarning)

0.7721354166666666
```

Figure 65.

```
clf.predict(X_array[:5, :])
```

```
array([1, 0, 1, 0, 1], dtype=int64)
```

```
clf.predict_proba(X_array[:5, :])
```

```
array([[0.31281653, 0.68718347],
       [0.95693125, 0.04306875],
       [0.23508854, 0.76491146],
       [0.95199445, 0.04800555],
       [0.24471218, 0.75528782]])
```

Figure 66.

```
# Hyper-tuning KNN Classifier Number of Neighbors (with kfold cv) -> use GridSearchCV
from sklearn.model_selection import GridSearchCV

knn2 = KNeighborsClassifier()

param_grid = {'n_neighbors': np.arange(1,25)} # test #neighbors from 1 to 24

knn_gscv = GridSearchCV(knn2, param_grid, cv=5)

knn_gscv.fit(X, y)

knn_gscv.best_params_ # optimal value for n_neighbors = 23
```

```
{'n_neighbors': 23}
```

```
# when n_neighbors=23 -> improve model accuracy by almost 4%
knn_gscv.best_score_
```

```
0.7486979166666666
```

Figure 67.

```
# to compare the cross validation method, demonstrate 'holdout' method (train_test_split)
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1, stratify=y)

knn = KNeighborsClassifier(n_neighbors=3)

knn.fit(X_train, y_train)
knn.predict(X_test)

knn.score(X_test, y_test) # less accurate than cross validation
```

```
0.7077922077922078
```

Figure 68.

Comparing & Choosing Champion Model

Various models that conduct binary classification were explored and assessed (in addition to the KNN Classifier). These were: KNN Classifier with unoptimized number of neighbors (Figure 60), SVM Classifier (Figures 61, 62), Gaussian Naive Bayes Classifier (Figure 63, 64), Logistic Regression (Figure 65, 66), and another KNN Classifier with optimized number of neighbors (Figure 67). It is important to note that all of these models use some form of k-fold cross-validation during training and testing. I included an example of when only the 'holdout' validation is used to validate the KNN Classifier (Figure 68) to compare the validation. Before I highlight the criteria of choosing a champion model (the most accurate model), I would like to make a few points about all contemplated models.

To start, Figure 60 shows the training of a KNN Classifier (arbitrarily-chosen 3 neighbors) with a cross-validation fold of 5 [2]. Having k=5 essentially means the data is split into 5 segments (4 training and 1

testing). The testing segment is iterated through each data segment once (resulting in 5 iterations). The benefits of this method have already been discussed in the prior section. Since the data (X and y) were trained and tested on the model 5 separate times, it makes sense that there would also be 5 different accuracy scores. These values were stored in an array called 'cv_scores'. An average was taken on these 5 values to attain a single accuracy score, which turned out to be 0.7097 or 70.97%. This is not a bad score, however certain modifications to the model's parameters may be able to improve its accuracy (which will be discussed later on).

The SVM Classifier is implemented in Figure 61 [5]. Since I could not find an implementation of SVM where k-fold cross-validation is built in (or k-fold library accepts an SVM model as a parameter), I had to manually implement it using a for loop. I also needed to convert the X dataframe to a numpy array in order to train the data (y is already in array format). For each iteration (5 iterations with 5 folds), the data was manually split into training and testing sets based on the indices created by k-fold cross-validation. The training was conducted and the accuracy score was appended to an array 'scores' each iteration. The average of these scores came out to be 0.6512 or 65.12% (Figure 62). This is not necessarily a great score; however, this is to be expected of an untuned SVM Classifier. Specifically, the C-parameter (regularization tuner) needs to be manipulated to achieve a good score. But, since the score was already lower than for KNN, it was decided that the fine-tuning won't be conducted and to disregard this model.

The Gaussian Naive Bayes Classifier was explored in Figure 63 [6]. The implementation was similar to that of the SVM Classifier with only a few differences. The obvious difference is the data was trained on the Gaussian Naive Bayes Model, instead of SVM. The other major difference is the accuracy score was manually calculated since I couldn't find an implementation for it. This was done by first calculating the predicted outcomes (y_pred) using the predict() method, then comparing it with y_test (actual outcomes) to see if they were the same or not. The more outcomes that were the same, the higher the score. The average accuracy score turned out to be 0.7475 or 74.75% (Figure 64). This is better than both the KNN Classifier and SVM Classifier tested before. This is expected since the traditional Naive Bayes Theorem is optimized for binary true/false outcomes.

The Logistic Regression model (decided later on as the Champion Model) implementation is shown in Figure 65 [4]. The specific implementation that was used included built-in cross-validation (specifically, stratified k-fold). The data was trained on a Logistic Regression Model with 5 iterations, leading to an average accuracy score of 0.7721 or 77.21%. This is by far the best accuracy score attained so far. Figure 66 gives a sample of the first 5 elements for predicting outcomes based on input data. The predict_proba() method give the likelihoods the outcome would be a certain value (obviously outputting the highest probable one of the pair in the predict() method). As a side note, I wanted to highlight that although logistic regression is not technically a 'classification' algorithm, it does work to classify categorical data (which the Outcome variable is). I believe this algorithm performs well for two reasons: there are not a lot of options/variation for the Outcome (binary 0/1), and also the predictor variables themselves are numeric. Therefore, a regression would be the ideal option.

Since the KNN Classifier was in the 70-percentile range and a hyper-tuning method was readily available for the model parameters, I decided to try it and see if the results attained proved more accurate. This was done in Figure 67 using the GridSearchCV module [2]. Specifically, the tuning of the number of neighbors' parameter used to train the data. The number of neighbors were each tested from a range of 1 to 24. The data was then fitted on the GridSearchCV algorithm, producing the optimal number of neighbors as 23 to obtain the highest accuracy score, which was 0.7487 or 74.87%. Although this is still less than that attained by logistic regression, the optimization gave a 4% increase in accuracy when compared to the previous KNN model.

I included Figure 68 to highlight why cross-validation is better to use than the holdout method. The holdout method is implemented with the common train_test_split module (80% training, 20% testing). The KNN Classifier is trained and tested with the number of neighbors equal to 3 (same as Figure 60) and the score obtained was 70.78%. So not only does the traditional holdout method not give a comprehensive validation of the data, it also results in a lower accuracy score (since that specific split may not have been optimal for testing). This further serves to reassure that cross-validation was the right validation framework to use.

Based on the accuracy scores obtained, the Logistic Regression (with built-in cross-validation) model performed the best in predicting the Outcome variable to classify if a patient has diabetes or not. The accuracy score metric serves to indicate the potency of a model to produce accurate results. The next section will go through various metrics to ascertain the Logistic Regression model as the most efficient and accurate model (compared to models explored above). The comparison of metrics will be between the Logistic Regression model and the (optimized) KNN Classifier model, since that was the model highlighted in the project assignment as well as the second-best model obtained in terms of accuracy.

Data Modelling – Week 4

```
knn_cv = KNeighborsClassifier(n_neighbors=23)
cv_scores = cross_val_score(knn_cv, X, y, cv=5)

print(cv_scores)
print('cv_scores mean: {}'.format(np.mean(cv_scores)))
```

```
[0.74025974 0.68181818 0.74025974 0.77777778 0.80392157]
cv_scores mean: 0.7488074017485783
```

Figure 69.

```
# Classification Report of LogisticRegression CV
clf = LogisticRegressionCV(cv=5, random_state=0).fit(X, y)
y_pred_log = clf.predict(X_array)
```

Figure 70.

```
from sklearn.metrics import classification_report
print(classification_report(y, y_pred_log))
```

```
              precision    recall  f1-score   support

           0       0.79      0.88      0.83       500
           1       0.72      0.57      0.63       268

    accuracy                           0.77       768
   macro avg       0.76      0.72      0.73       768
weighted avg       0.77      0.77      0.76       768
```

Figure 71.

```
from sklearn.model_selection import cross_val_predict
knn_cv = KNeighborsClassifier(n_neighbors=23)
y_pred_knn = cross_val_predict(knn_cv, X, y, cv=5)
```

```
print(classification_report(y, y_pred_knn))
```

```
              precision    recall  f1-score   support

           0       0.78      0.85      0.81       500
           1       0.67      0.56      0.61       268

    accuracy                           0.75       768
   macro avg       0.72      0.70      0.71       768
weighted avg       0.74      0.75      0.74       768
```

Figure 72.

```
from sklearn.metrics import roc_auc_score
print('Area Under ROC Curve for Logistic Regressor: {}'.format(roc_auc_score(y, y_pred_log)))
print('Area Under ROC Curve for KNN Classifier: {}'.format(roc_auc_score(y, y_pred_knn)))
```

```
Area Under ROC Curve for Logistic Regressor: 0.7245820895522388
Area Under ROC Curve for KNN Classifier: 0.7048507462686566
```

```
from sklearn.metrics import average_precision_score
print('Average Prediction Score for Logistic Regressor: {}'.format(average_precision_score(y, y_pred_log)))
print('Average Prediction Score for KNN Classifier: {}'.format(average_precision_score(y, y_pred_knn)))
```

```
Average Prediction Score for Logistic Regressor: 0.5596149141732097
Average Prediction Score for KNN Classifier: 0.5267801616915423
```

Figure 73.

```
from sklearn.metrics import confusion_matrix
print('Logistic Regressor')
print(confusion_matrix(y, y_pred_log))
print('\n')
print('KNN Classifier')
print(confusion_matrix(y, y_pred_knn))
```

```
Logistic Regressor
[[441  59]
 [116 152]]


KNN Classifier
[[425  75]
 [118 150]]
```

Figure 74.

Classification Report & Various Other Metrics

Before we get into exploring the metrics, to confirm the accuracy score of the KNN Classifier when using the optimal number of neighbors is 23, the training and testing of data was conducted from scratch in Figure 69. The mean accuracy scores turned out to be the same at 74.88% as expected (just 0.01% off, but that amount is negligible and probably due to the rounding of intermediate means).

The first metric used to critique the models was the 'Classification Report' [7]. An important point to note is in the case of binary classification, the recall of the positive class is called 'sensitivity', while the recall of the negative class is called 'specificity'. The positive class being 1 and the negative class being 0. Figure 70 assigned prediction values to 'y_pred_log' based on the Logistic Regression model. The classification report in Figure 71 compares the predicted value to the actual result stored in 'y'. The key values to note here are the accuracy measure (Logistic Regression is higher as mentioned previously), as well as recall. The higher the recall is to 1, the better the predictive power. Again, we can see that Logistic Regression beats the KNN Classifier (Figure 72) with 0.88:0.57, as oppose to 0.85:0.56.

Figure 73 details the Area Under Receiver Operating Characteristic Curve (AUC ROC) metric [8]. The closer this value is to 1, the better the model's separability power is (how good it is at distinguishing between classes). The winner of this battle is again Logistic Regression with 0.7246, as opposed to KNN with 0.7049. The average precision score is also indicated in the same figure [9]. The average precision score (similar to the area under a precision-recall curve) signifies how well a model orders the predictions. Logistic Regression is better than KNN with a score of 0.5596, as opposed to 0.5268.

The last metric is the traditional confusion matrix (shown in Figure 74). This highlights the true positives and negatives along with the false positives and negatives [10]. The good thing about this metric is it visually shows you the number of individual records that are classed correctly and incorrectly. The top left shows the true positives, top right the false positives, bottom left the false negatives and bottom right the true negatives. The matrices indicate that Logistic Regression has more true and less false values than

the KNN Classifier, which is what is desired. Hence, Logistic Regression is a more efficient model than the KNN Classifier.

The 'values of parameters' (requirement #9) will be interpreted as metrics used to evaluate model competency. The values of all parameters that were analyzed each indicated a useful aspect of the model. However, I believe the most intuitive and simplest metrics to understand are the accuracy score and the confusion matrix. I concluded this since accuracy is a simple concept to understand, as well as visually interpreting more correct values than false in a confusion matrix is extremely intuitive, even to the common person. Hence, these two specific metrics are understandable to a larger population and hence, easier to prove the model's competency with.

Data Reporting – Week 4

Since the majority of the figures created in Tableau replicate those that were created using Python (except the pie and bubble charts), I will not cover the full details of how the visual was created nor show the specific Sheets used to create them. I will just give brief explanations of the visuals and highlight points I believe deserve more attention. I will however, try to be a bit more detailed when explaining the bubble charts since it is a relatively new concept. The following will consist of a section for each of the required 5 sets of visuals. Note that not all variables are included in the following visuals, since they have been already presented in detail above. The respective sections will further detail this point. Keep in mind also that these visuals were produced using the imputed dataset. Hence the 0 values in the histograms prior for example, will not look the same as the ones produced in this part. This was done so the data relates more to the dataset used for modelling and projections, so management would have a better idea of the current state of affairs and where its projected to go. In addition, it makes logical sense that the presented data should try not have meaningless or missing values in them.

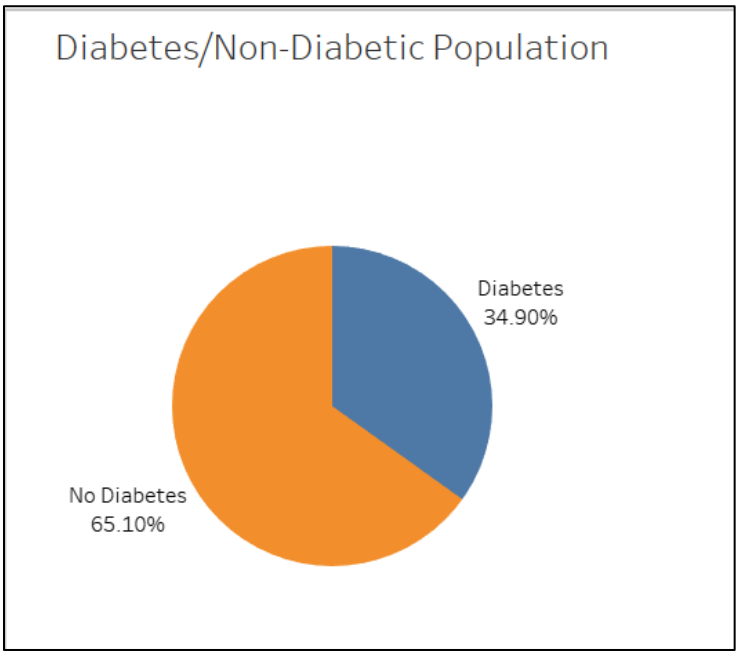

Figure 75.

Diabetes/Non-Diabetes Population (Pie Chart)

In Figure 75, the Dashboard shows the people who have and have not been diagnosed with diabetes as a percent of the total sample population. As stated above as well as in the Project Requirements, 268 of 768 have diabetes, while 500 do not. This is visually represented in the pie chart by 34.9% having it while 65.1% do not. This is immediately clear to the view since they can quickly see the vast size difference between the two proportions. This is valuable in the sense of knowing the current state of the Outcome (goal of the project). Note that the labels 'Diabetes' (1) and 'No Diabetes' (0) were created using an if statement in a Calculated Field. Since the pie chart is a simple and common concept, I feel that no further explanation is required.

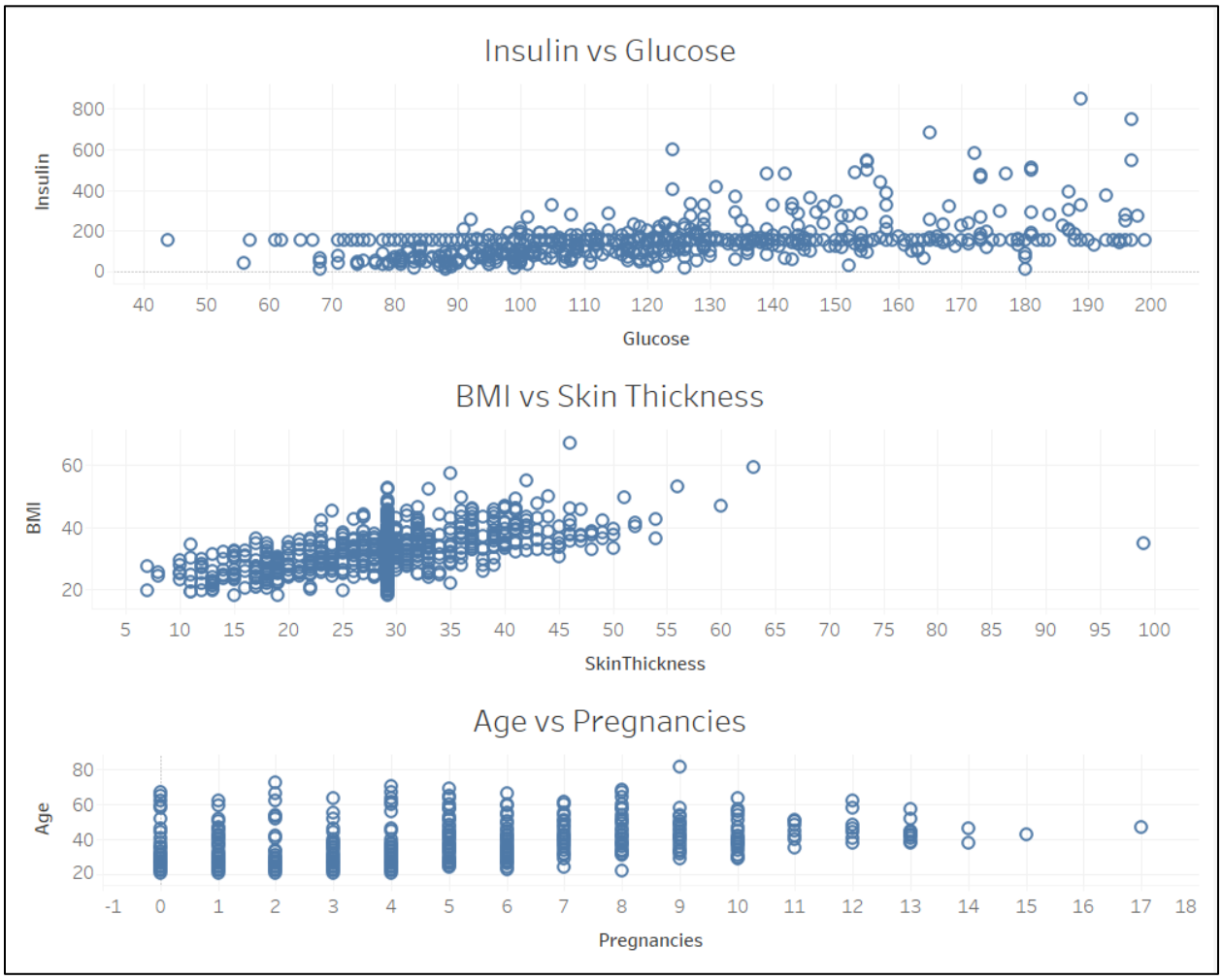

Figure 76.

Scatter Charts Highlighting Among Relevant Variables

Since the scatter plots for all relationships among (predictor) variables were shown in the 'Scatter Plots' section above; in an attempt to not be repetitive, I will only show those I believe to have a decisive relationship through a trend or pattern (Figure 76). As stated before, I believe only 3 pairs of variables have a noticeable, tangible relationship among them: Insulin vs Glucose, BMI vs Skin Thickness, and Age vs Pregnancies. Since these were discussed in great detail previously, I will save the reader the time of going through it again. To quickly highlight however, all three of these relationships follow logical reasoning hence the pattern among each pair is as expected.

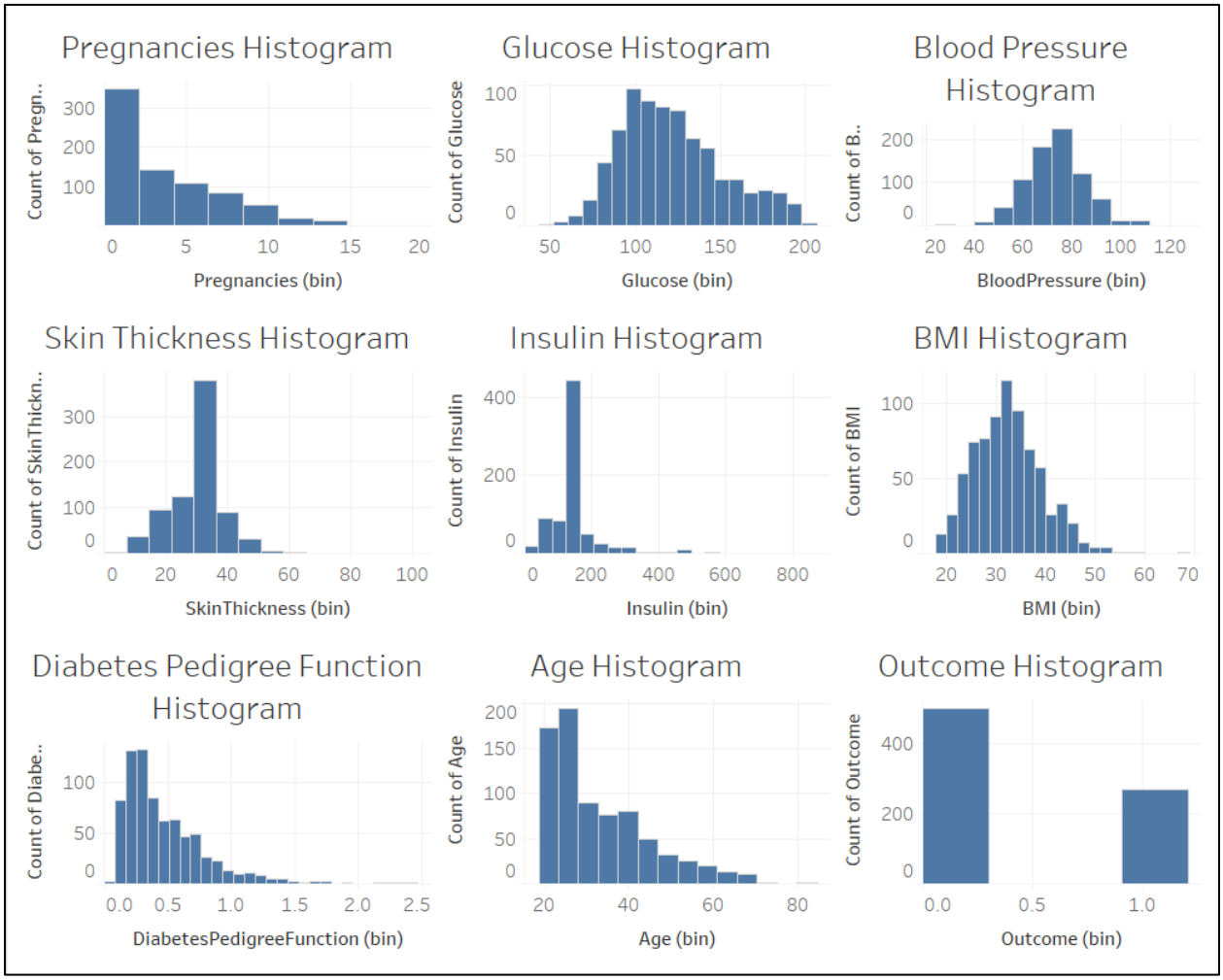

Figure 77.

Histograms for Variable Distribution

All 9 variables (including Outcome) have their histograms on the dashboard. Again, since the concept of these relate to the previous histograms produced, I would only go over them briefly here. As mentioned previously, these histograms do not look like the ones produced prior since these used the imputed

dataset, while the ones in the 'Histograms Interpretation' section used the original dataset. I believe the histograms presented in Figure 77 however give a much better understanding of the current affairs of the population, compared to if the 0 values were used. At least the column average is used so it would not skew the overall interpretation of the data. Apart from that, these histograms appear as expected.

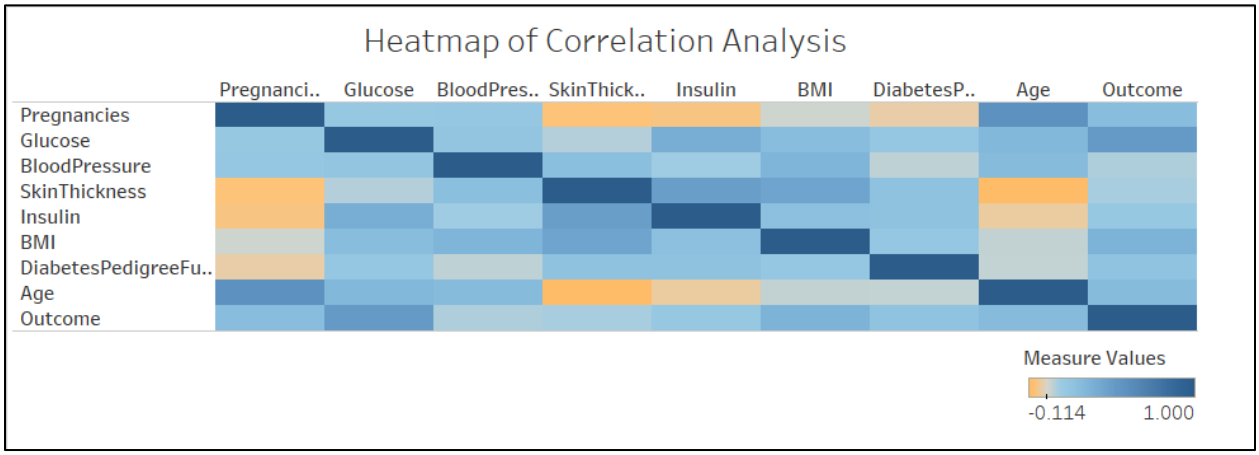

Figure 78.

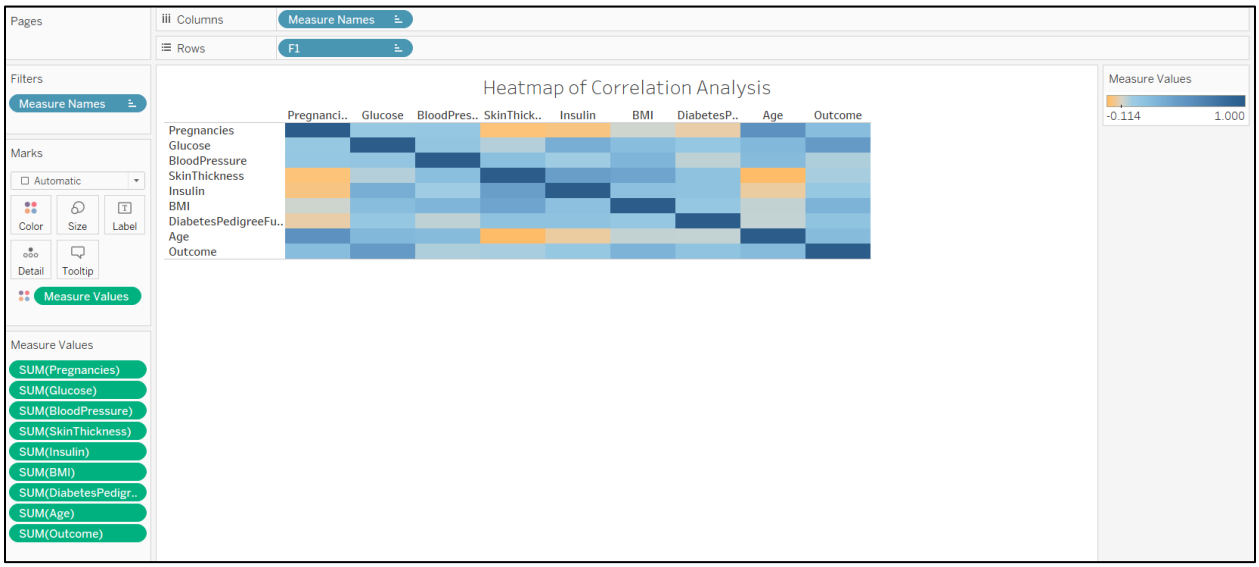

Figure 79.

Heatmap of Correlation Analysis

The heatmap in Figure 78 appears to convey the same information as in the prior heatmap in 'Correlation Analysis'. Before I go through why I included Figure 79, I just want to add that a minor difference here as opposed to the prior is the range of correlation values. The heatmap in Figure 58 had a minimum value of 0, while in Figure 78 the minimum value is -0.114. Hence, it can be interpreted that in Figure 58, if a

correlation was calculated to be below 0, it would automatically be reported as 0 in the heatmap. A negative correlation theoretically means that as one variable increases, the other decreases (an inverse cooperative relationship). This wouldn't really make any major changes to the previously discussed analysis since the negative correlation magnitude is still negligible in comparison to the positive ones, however, it is still interesting to note. Positive correlations are more intuitive anyways since they predict a standard linear relationship. They are also what models use in making predictions since they are usually more potent that negative ones. Other than that minor detail, the analysis of this heatmap is the same as the prior; the same correlations are recorded.

I included Figure 79 essentially to highlight the struggle to obtain a supposedly-simple heatmap. In order to obtain a correlation heatmap, I tried a multitude of failed approaches. I first looked into how Tableau implements its correlation, which I then found out to be through the 'Pearson Correlation Coefficient' using the 'r' factor. I tried applying that to two variables (since with all variables already proved futile) using a Calculated Field, which only resulted in null values (I believe the reason was that none of the variables were actually Dimensions/categorical data). I then thought to remove the automatic aggregation of variables and plot them in a matrix; however, this only produced a scatter plot that requires interpretation for the correlation factor, which is not visually available. I then thought to plot the values while converting one of the variables into a dimension, but that resulted in an unexpected relationship. After many failed attempts such as these, I remembered that the Python function corr() returns a dataframe by default. Hence, I used this to produce a correlation matrix in csv format, and imported it to Tableau (F1 is the auto-generated unique index by Tableau). This provided me with the weighted (correlation) values required to create the required heatmap. Figure 79 basically shows how this visualization was created. Hence, the visual was created in Tableau (as required), but the essential calculation came from Python. I believe this is ok since no restrictions were given pertaining to the use of languages to calculate values for Tableau, and Tableau already has an implementation of 'R' to conduct calculations, so I believe even Tableau themselves expect users to create sophisticated calculations elsewhere, and use Tableau just as a formal visualization tool.

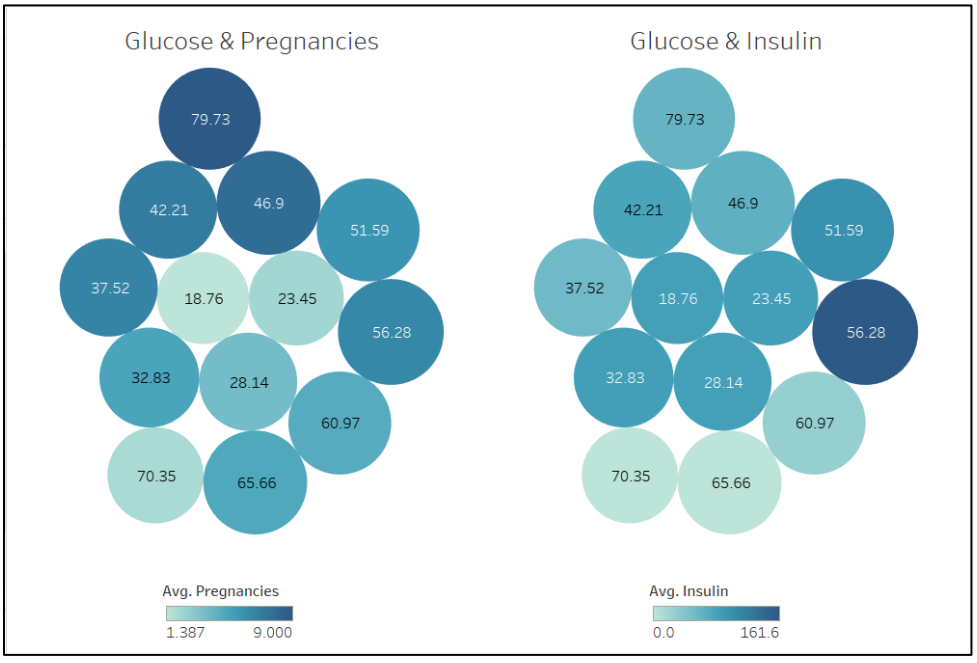

Figure 80.

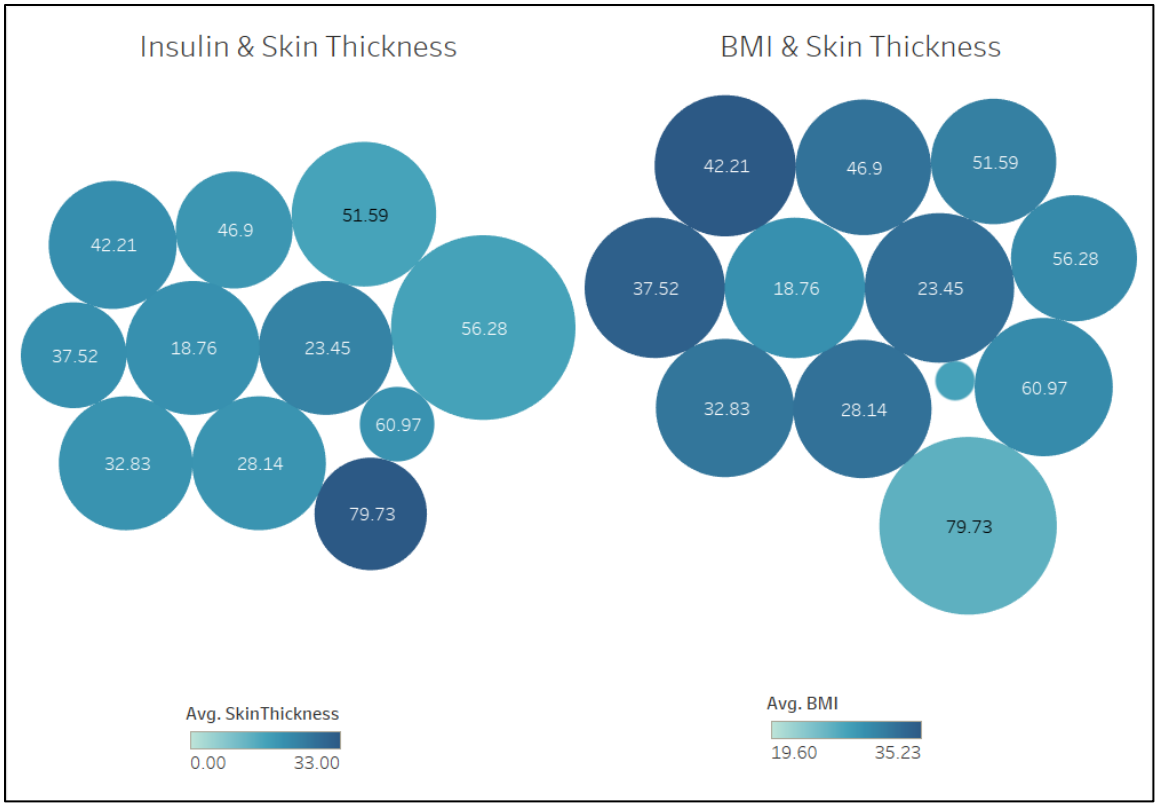

Figure 81.

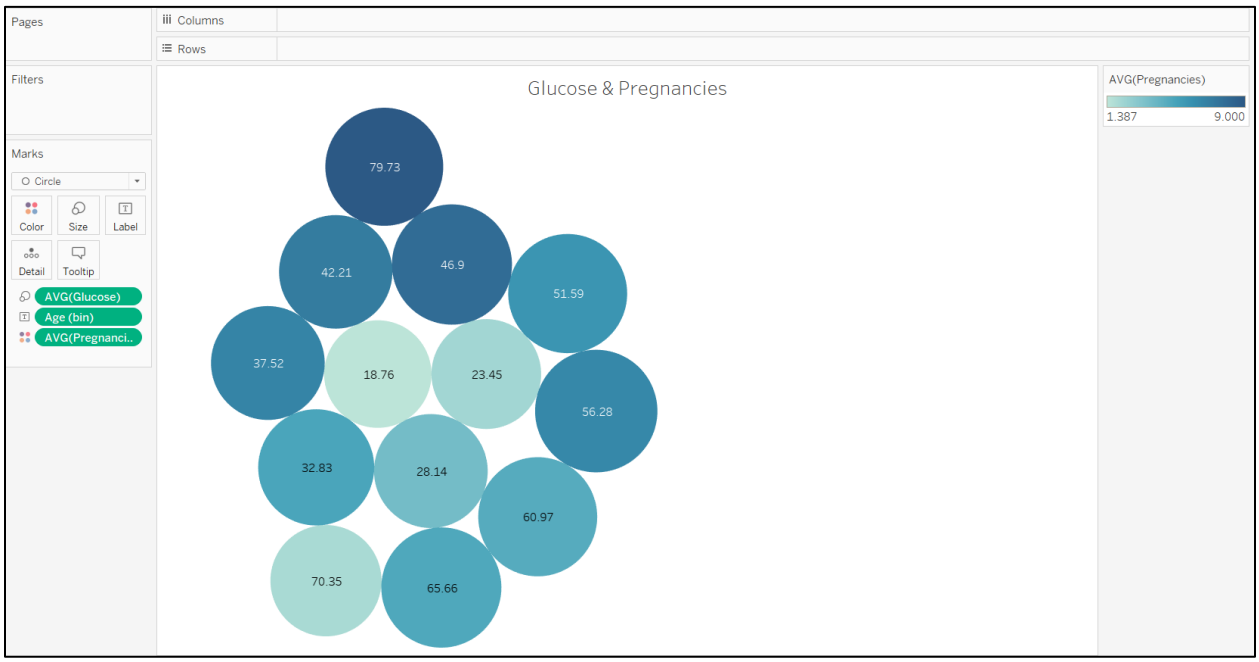

Figure 82.

Exploration of Variables using Bubble Chart (Age Bins)

The size of the bins to segment the Age variable is 5 (as required), however Tableau will only use that as a guideline and will act to optimize the distribution so it is visually plausible. The specific bin is indicated by the number on the circle. Since the bin size is 5 (ideally 20-25, 25-30 etc.), the number would be closet to the beginning of that range. It can be seen that the numbers themselves are differenced by 5 as well, further indicating the bin size.

The bubble charts indicate two values (variables) grouped by Age. Only one or two measures (numeric variables) can be represented on a bubble chart. I decided to have two measures on a single bubble chart so it can also act as a comparison. One variable is indicated by the size of the bubble while the other by its color. I did not included combinations of all variables since I believe it pose no benefit. Instead, I selected specific pairs of variables that I believe have a relationship in order to demonstrate how the bubble chart operates. I also used the average of the measures instead of sum because if certain age groups have more observations than others, an unfair account of the variables would be attained.

I included 4 bubble charts but split them into two different dashboards since attempting to fit all into one made them too clustered and hard to interpret. Figure 80 highlights two bubble charts, both of which pertains to relationships of Glucose. The first is of Glucose & Pregnancies while the other is of Glucose & Insulin. Figure 81 shows the bubble charts for relationships pertaining to Skin Thickness. Specifically, the first one of Insulin & Skin Thickness and the second one of BMI & Skin Thickness.

For the Glucose & Pregnancies chart, the representation is as expected. The colors are darker for age groups between 25 – 55 years old (able to be pregnant), as well as gradual, slight increases in the sizes as age increases (increased glucose levels with age). In terms of the Glucose & Insulin chart, the colors are around the same (darkest around 55 – 60 years old) and lightening drastically in the senior ages. This indicates the insulin content drastically decreases in the older age groups. The Glucose variable would give the same interpretation as the other graph since it's acting on the size aspect again.

The Insulin & Skin Thickness Chart details a steady color (indicating skin thickness) for most of the groups, but has a sudden deepening around the oldest age group of 75 – 80 years old. I believe this to be caused by the deterioration of muscle mass causing the skin to overlap. Since the definition of skin thickness is based on the folds behind the triceps, overlapping skin would drastically increase this measure. Insulin was discussed in the previous paragraph so I won't go over it again. This chart just provides a different visualization of the measure using colors instead of size. The BMI & Skin Thickness chart share a positive linear relationship (as one increases, so does the other). As one gains mass (usually in the form of fat) with no increase in height (increase BMI), the skin thickness would increase with increased fat. The exception to this is the senior group, where the BMI is low but the Skin Thickness is high due to the deterioration of muscle.

I included Figure 82 to show how a bubble chart is constructed. Essentially, the grouping variable is placed on the detail option of the 'Marks Pane' to show the respected group, while a measure on the size option and the other on the color option. In this way, two variables can be represented simultaneously on one visual.

Conclusion

This project provided a very in-depth experience of a real-world scenario when working in the Data Science field. It allowed me to complete all aspects of the Data Science Life Cycle (i.e. Data Exploration, Modelling and Reporting). The instructions were straight-forward and related to real-life requirements.

In terms of the results attained (charts, graphs, arrays, models etc.), there may have been some outliers present. However, these were disregarded when interpreting relationships among variables since they don't contribute to the overall interpretation of the data; they serve no purpose in modelling and reporting to management. The problem presented has formally been solved by building a model (Logistic Regression with Cross-Validation) to predict whether a patient has diabetes or not.

Overall, I am happy with the process I took to accomplish the requirements of this project. Though there were some obstacles, it provided me with valuable troubleshooting skills in which I can use in the future. The results followed logical reasoning and explained the data well.

References

[1] Plotting Histograms with matplotlib and Python (April 1, 2020). *Python for Undergraduate Engineers*. Retrieved from https://pythonforundergradengineers.com/histogram-plots-with-matplotlib-and-python.html

[2] Building a k-Nearest-Neighbors (k-NN) Model with Scikit-learn (April 1, 2020). *towards data science*. Retrieved from https://towardsdatascience.com/building-a-k-nearest-neighbors-k-nn-model-with-scikit-learn-51209555453a

[3] sklearn.model_selection.KFold (April 1, 2020). scikit-learn 0.22.2 documentation. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

[4] sklearn.linear_model.LogisticRegressionCV (April 1, 2020). scikit-learn 0.22.2 documentation. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

[5] sklearn.svm.SVC (April 1, 2020). scikit-learn 0.22.2 documentation. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

[6] 1.9. Naive Bayes (April 1, 2020). scikit-learn 0.22.2 documentation. Retrieved from https://scikit-learn.org/stable/modules/naive_bayes.html

[7] sklearn.metrics.classification_report (April 1, 2020). scikit-learn 0.22.2 documentation. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html

[8] sklearn.metrics.roc_auc_score (April 1, 2020). scikit-learn 0.22.2 documentation. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html#sklearn.metrics.roc_auc_score

[9] sklearn.metrics.average_precision_score (April 1, 2020). scikit-learn 0.22.2 documentation. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html#sklearn.metrics.average_precision_score

[10] sklearn.metrics.confusion_matrix (April 1, 2020). scikit-learn 0.22.2 documentation. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html