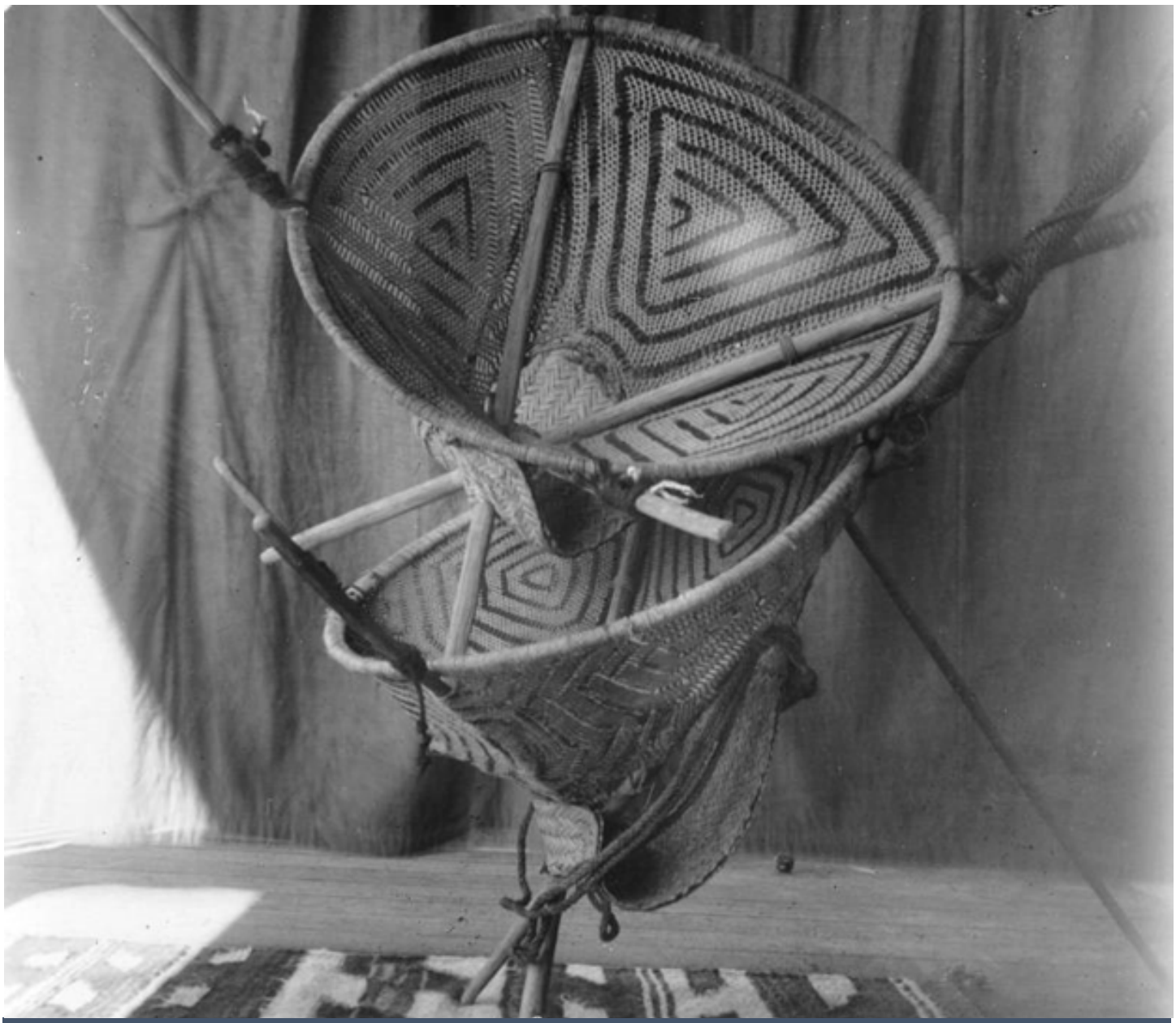


Pima Indians Diabetes

A Study of Imputation Methods and it's Impact on Machine Learning Algorithm Performance

Joshua Rieck

4/5/23

CSC 7810: Data Mining
Algorithms and Applications

Table of Contents

Abstract

This research paper aims to test different imputation methods in filling in missing values using the Pima Indians diabetes dataset, and to evaluate the performance of several machine learning models in predicting diabetes outcomes using the imputed dataset. The dataset contains medical information on Pima Indian women, including several features such as age, body mass index (BMI), and glucose levels. The missing values in the dataset were imputed using different methods, including k-nearest neighbor imputation, linear regression imputation, and random forest imputation. Each imputed dataset was then trained on eight different machine learning models, with cross validation used on each algorithm to find the optimal set of hyper-parameters. The performance of these models was evaluated using area under the curve (AUC) scoring. The study found that imputation methods can significantly impact feature distribution and model performance, with some methods resulting in improved performance and others leading to decreased performance. The KNN model consistently showed improvement across all imputation methods, while the decision tree model had the poorest performance. The best overall performance was achieved by the Keras artificial neural network using random forest imputation, with glucose identified as the most important variable for predicting the diagnosis. The results suggest that careful consideration of imputation methods and learning models is crucial for accurate and effective analysis of data with missing values.

Introduction

When training machine learning models, missing data can create significant challenges in accurately predicting outcomes. As the adage goes, "garbage in, garbage out," emphasizing that the quality of data inputs will dictate the quality of model outputs. Missing data can not only impact the performance evaluation of machine learning models but also result in issues such as overfitting and inaccurate prediction results on newer data.

In the past, two techniques commonly employed to address missing data are deletion of instances with missing values and auto-filling with the mean score within a feature. While these two techniques have offered a quick and simple solution to dealing with missing values, they can have their limitations. For instance, deletion can potentially lead to significant data loss, while filling missing values with the mean can introduce more bias in the dataset, which can skew the feature data distribution, resulting in inaccurate results. However, using machine learning imputation methods can help preserve the data's underlying structure, reduce bias, and make better feature predictions by considering the relationships between all the features in the dataset.

Study Objectives

The objective of this paper is to implement and observe several machine learning imputation methods and test to see if there is any noticeable performance improvement across multiple machine learning algorithms. Each machine learning algorithm will have their hyper-parameters fine-tuned in order to ensure the highest performance result.

Historical Review

According to the World Health Organization, diabetes is a chronic disease that affects the body’s ability to produce and regulate insulin levels within the body that control an individual’s blood glucose (World Health Organization, 2022). Long term diabetes not only affects the pancreas, which produces the insulin hormone, but can also lead to co-morbidities including cardiovascular disease, kidney disease, and nerve damage in limbs and extremities. Complications from diabetes can also lead to amputation and blindness in severe cases (World Health Organization, 2022). This is why early detection is so important for diabetes patients. Early detection allows for healthcare professionals to quickly intervene and put patients on treatment plans that help curb dangerous co-morbidities above and improve quality of life.

The Pima Indians are a Native American tribe located in the Southwest United States, where they have 52,600 acres of tribal lands just outside of Phoenix, AZ (Inter Tribal Council of Arizona, 2023). Nearly 6,000 years ago, the Pima tribe's ancestors settled around the Gila River basin and established an intricate man-made irrigation system that consisted of 500 miles of canals and ditches; with this system, they successfully cultivated the land, supporting a population that reached 50,000 to 60,000 people at its peak (Inter Tribal Council of Arizona, 2023). Early Spanish settlers made contact with the tribe and established trade of new crops and livestock, then in 1846, the US government established ties with the tribe after the Mexican American War (Inter Tribal Council of Arizona, 2023).

Although the Pima tribe treated American settlers who ventured West in search of gold with friendliness, the settlers constructed their own dams and irrigation system upstream on the Gila River in the 1870s and 1880s, which unsurprisingly resulted in a severe famine and widespread starvation among the established Pima tribe living downstream (Inter Tribal Council of Arizona, 2023). The transition from locally sourced food to highly processed canned food provided by the US government lead to the development of high rates of obesity and diabetes amongst the tribe (Inter Tribal Council of Arizona, 2023). In fact, according to a comprehensive global study on diabetes rates among regional populations, the Pima Indian tribe had the highest rate of diabetes recorded. The study revealed that 42% of participants aged 25 and older had diabetes, while 50% of participants aged 35 and older had the disease (Bennett, Burch, & Miller, 1971).

The dataset utilized in this project is composed of de-identified health data from 768 individuals, collected by the National Institute of Diabetes and Digestive and Kidney Diseases. The dataset has been widely used in diabetes research, with one of the earliest papers citing it dating back to 1988. For the purposes of this study, the patient selection criteria were women aged 21 or older with Pima ancestry (UCI Machine Learning, 2023).

Literature Review

There have been many studies that have been conducted using the Pima Indians diabetes dataset throughout the years. Not only have there been countless studies based on various machine learning models and their performance on accurately predicting a diabetes diagnosis, there have been studies that take into consideration what learning models are the most transparent for healthcare professionals to feel confident in a diagnosis prediction. A study completed in 2022 by Chang, V et al proposed that the best learning models to implement in an Internet of Medical Things (IoMT)

environment that give healthcare providers access to the internal decision-making process are decision tree, naïve bayes, and random forest models (Chang, Bailey, Xu, & Sun, 2022).

Other studies that have used this dataset have explored different feature reduction methods to improve prediction performance. One study done in the Journal of Statistics and Mathematical Engineering used Exploratory Factor Analysis (EFA) to combine various features such as BMI and SkinThickness, Insulin and glucose, pregnancies and age into three respective factors for improved predictive performance (Biju & James, 2022). Another study looked at the effects of dimensionality reduction on the data using principal component analysis (PCA) and tested the performance accuracy using a naïve bayes learning model, resulting in a less than 1% improvement (Ozsahin, Mustapha, Mubarak, Ameen, & Uzun, 2022).

There continue to be new studies done by the National Institute of Diabetes and Digestive and Kidney Diseases, located in Phoenix, AZ that has utilized more in-depth patient data collected from the local Pima Indian community. One of these newer studies is exploring how diabetes may lead to an increased risk of cognitive decline and dementia (Biessels & Despa, 2018).

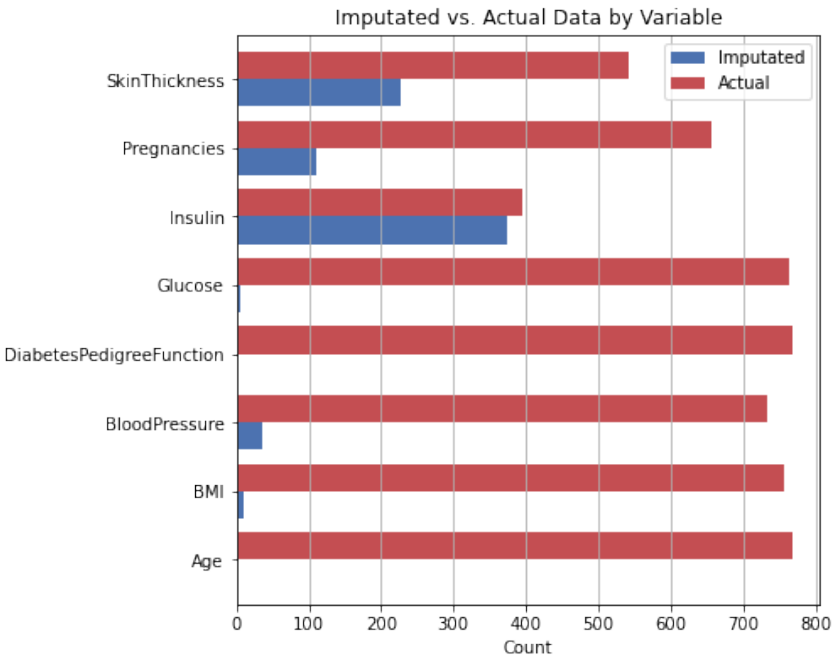
Methodology

Defining the Dataset Variables

The Pima Indians dataset has a total of 768 instances and nine numeric feature variables. The table below lists the feature variables that were collected from each individual, a brief description of what each one means, and the range of observed values for each variable:

Feature Name	Description	Range
Pregnancies	number of times pregnant	[0, 17]
Glucose	Plasma glucose concentration at 2 Hours in an oral glucose tolerance test (GTIT)	[44, 199]
BloodPressure	Diastolic Blood Pressure (mm Hg)	[24, 122]
SkinThickness	Triceps skin fold thickness (mm)	[7, 99]
Insulin	2-Hour Serum insulin (µh/ml)	[14, 846]
BMI	Body mass index [weight in kg/(Height in m)]	[18.2, 67.1]
DiabetesPedigreeFunction	probability of genetic pre-disposition to diabetes	[0.078, 2.42]
Age	Age in years	[21, 81]
Outcome	Binary value indicating non-diabetic /diabetic	[0, 1]

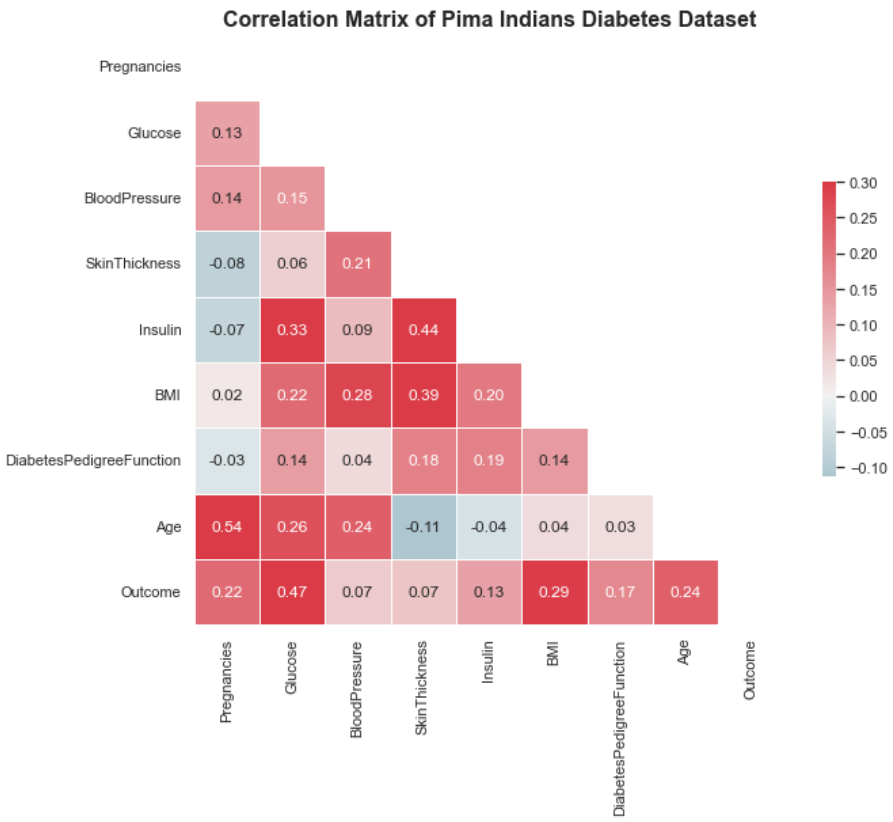
Upon closer inspection of the dataset, it becomes apparent that there are several instances where missing data is represented as a value of 0 instead of a blank or null value. To illustrate this point, the graph and table below present the distribution of each feature variable, separating the counts of values greater than zero from those equal to zero:



Variable	Imputated	Actual
Age	0	768
BMI	11	757
BloodPressure	35	733
DiabetesPedigreeFunction	0	768
Glucose	5	763
Insulin	374	394
Pregnancies	111	657
SkinThickness	227	541

Although it is reasonable to anticipate that some individuals may have no pregnancies at the time of evaluation, it is still possible that some of the zero values for this feature may actually be null. Among the feature variables, insulin had the highest number of null values, with 374 out of 768, while SkinThickness had the second highest number of null values, with 227 out of 768.

Next, we will examine the correlation between each feature variable. This is an essential process that helps to determine whether any of the feature variables are dependent on each other. The independence of features is significant because some of the learning models used in this study, such as Naïve Bayes, assume that all features are independent. Therefore, understanding the correlation between each feature variable is crucial in selecting an appropriate machine learning model.



Based on the correlation graph above, it can be observed that there is no strong correlation between each individual feature variable or between the feature variables and the output variable. As a result, there is no requirement to apply various data pre-processing techniques such as feature selection or feature extraction to reduce the dimensionality of the data.

Defining Imputation Steps

In this experiment, we will compare the performance of three different imputation methods to fill in the missing null values in the dataset: K-Nearest Neighbors Imputer, Linear Regression, and Random Forest Imputer. Each method uses the respective algorithm to input the null values.

KNN and random forest imputation methods assume that the data is non-parametric, while linear regression relies on the assumption that the variable relationship is linear (Faisal, 2018) (Dash, 2022). Consequently, each algorithm handles missing data differently. Linear regression imputation relies on complete individual cases to predict the missing values (Swalin, 2018), but for this dataset, the linear regression model can only rely on 392 out of 768 total instances. In contrast, both KNN and random forest imputers directly handle each null value by looking at the values of all variables across the dataset, instead of limiting themselves to only complete instances (Swalin, 2018).

Data Pre-Processing Steps

After a determination has been made regarding what imputation method (if any) is applied to the dataset for testing, the data will be put through a scaling process using the standard scale. This scaling method places each feature variable's mean at 0 and sets the standard deviation to 1, while preserving the original distribution of each feature. Standard scaling also works better than other scaling methods when it comes to outlier sensitivity, due to the mean and standard deviation being the same across all feature variables.

The dataset will then be split into two subsets using a train-test ratio of 80-20. This ratio is commonly used in most machine learning studies and allows for the learning algorithms to train on more data. The 80-20 split also decreases performance variance and allows for algorithms to have a more stable output.

Defining Machine Learning Algorithms Used

Logistic Regression

Logistic Regression classifier is a standalone binary classification algorithm that uses feature variables to predict outcomes. The algorithm transforms each feature variable into a probability between 0 and 1 using the sigmoid formula, which acts as an activation function:

$$Sigmoid = \frac{1}{1 + e^{-x}}$$

This prediction is then compared what the true classification is using the log-loss function:

$$Log\text{-}Loss = -y_i ln\big(f(x_i)\big) - (1 - y) ln\big(1 - f(x_i)\big)$$

The algorithm adjusts the weights of the prediction model to minimize the loss and produce accurate predictions. This algorithm works well at predicting binary classifiers (Thorn, 2020).

Naïve Bayes

Naïve Bayes classifier is another standalone algorithm that uses Bayes Theorem to predict the probability of an outcome (A), given a set of features (B):

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$

Naïve Bayes assumes that each feature is independent from one another, hence it being naive (Gandhi, 2018). One drawback to this assumption of feature independence is how each feature is correlated to one another. Features that have correlation could lead to performance issues with the model.

Decision Tree

The Decision Tree is another standalone algorithm that uses rule-based logic to create a root node and split the data into various leaf nodes by calculating lowest loss function for each split. Like logistic regression, the decision tree algorithm can use the log-loss function. Other loss functions that can be used are entropy loss:

$$Entropy = -\sum_{i=1}^{n} p_i * \log(p_i)$$

And Gini Index loss:

$$Gini\ Index = 1 - \sum_{i=1}^{n} (p_i)^2$$

Decision trees can handle both categorical and numeric feature variables and is easy to visualize and explain. Decision trees are also good at handling datasets where the relationships between each feature are non-linear. The only drawback to decision trees is that they can overfit training data, which can lead to lower performance on the testing data (Bento, 2021) (scikit learn, 2023). Decision trees also run into performance issues when there is an imbalance in the distribution of classifier data, which can cause issues when nodes are being split (Brownlee, 2020).

K Nearest Neighbor

The K nearest neighbor (KNN) classifier is a standalone algorithm that works with both regression and classification models and assigns a prediction based on both distance and majority vote. There are several different distance metrics that can be used to select the nearest neighbor points for the majority vote. The distance metrics used in this project are the following:

$$Euclidean\ Distance = \sqrt{\sum_{i=1}^{n} (y_i - x_i)^2}$$

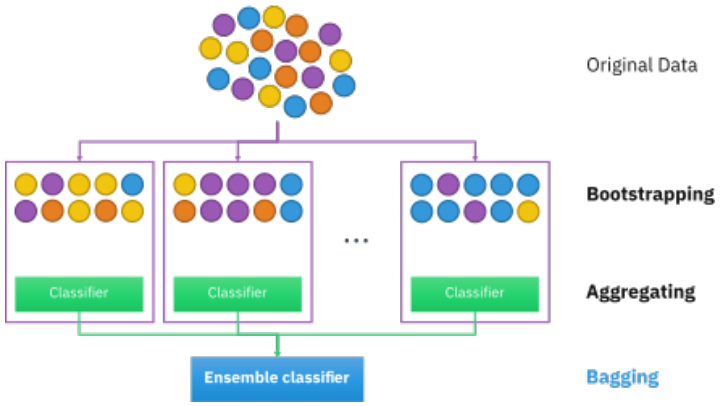
$$Manhattan\ Distance = \left(\sum_{i=1}^{n} |X_i - y_i|\right)$$

$$Minkowski\ Distance = \left(\sum_{i=1}^{n} |X_i - y_i|\right)^{\frac{1}{p}}$$

Another important feature of the KNN classifier is the hyper-parameter n_neighbors, which is used to determine how many nearest neighbors to use for the majority vote.

Random Forest

Random Forest classifier is an ensemble learning algorithm that uses a method called bagging to train a multitude of independent decision tree models to make a prediction using majority vote. Each decision tree takes samples from the original data. These samples can be chosen multiple times across multiple decision trees (this is known as “with replacement”). The graph below from Analytics Vidyha (Sruthi, 2023) provides a good visual representation of how the random forest algorithm works:



Majority Vote

The Majority Vote algorithm is an ensemble learning technique that employs multiple independent machine learning models to predict classifications. In this experiment, a soft voting method will be used as the voting criteria. Unlike the hard voting method, which relies on the highest number of votes, the soft voting method selects the prediction based on the standalone algorithm with the highest overall probability. To conduct this experiment, I will select the top three performing standalone algorithms and keep the hyper-parameters that yield the best performance results.

Gradient Boosting

The Gradient Boosting algorithm is another ensemble learning technique that combines multiple weak models iteratively to create a strong model that corrects errors of previous models. By itself, weak learning models are limited in their ability to make accurate predictions, but their iterative combination can lead to a highly accurate model prediction. Unlike the bagging technique used in random forest, this algorithm uses boosting technique, which improves predictive performance by building upon the

strengths and weaknesses of previous models. Similar to logistic regression, gradient boosting uses log-loss function improve performance. Gradient boosting can also use binomial deviance loss:

$$Deviance = log\left(1 + e^{-y_i f(x_i)}\right)$$

And Exponential Loss:

$$Exponential\ Loss = e^{-y_i f(x_i)}$$

Like the decision trees algorithm, gradient boosting can be susceptible to overfitting, especially if the number of estimators in the model are high (Corporate Finance Institute, 2023).

Deep Learning Artificial Neural Network (ANN)

The artificial neural network (ANN) algorithm consists of interconnected layers and nodes (also known as perceptrons) that stem from an initial input level and use forward and backward propagation techniques to increase model performance. These models can have multiple layers of nodes and can have significant numbers of neural connections. During the training stage, the ANN algorithm will set the initial weights and bias for each node, predict the output (forward propagation), compare the output to the actual prediction using a loss function, then re-adjust the weights by a specified learning rate (backwards propagation). These steps are completed repeatedly a specified number of times, known epochs, until the loss function has been minimized (Seth, 2021). As with Logistic regression, the ANN algorithm uses sigmoid activation. Other activation functions include the ReLU (rectified linear unit) function:

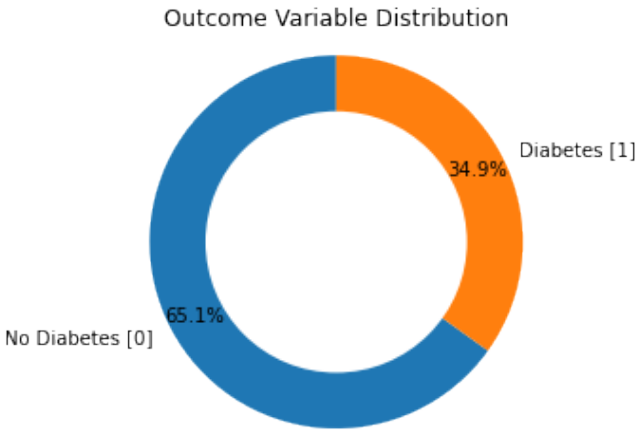
$$ReLU = max(0, x)$$

And the Tanh function:

$$Tanh = \frac{e^{x} - e^{-x}}{e^{x} - e^{-x}}$$

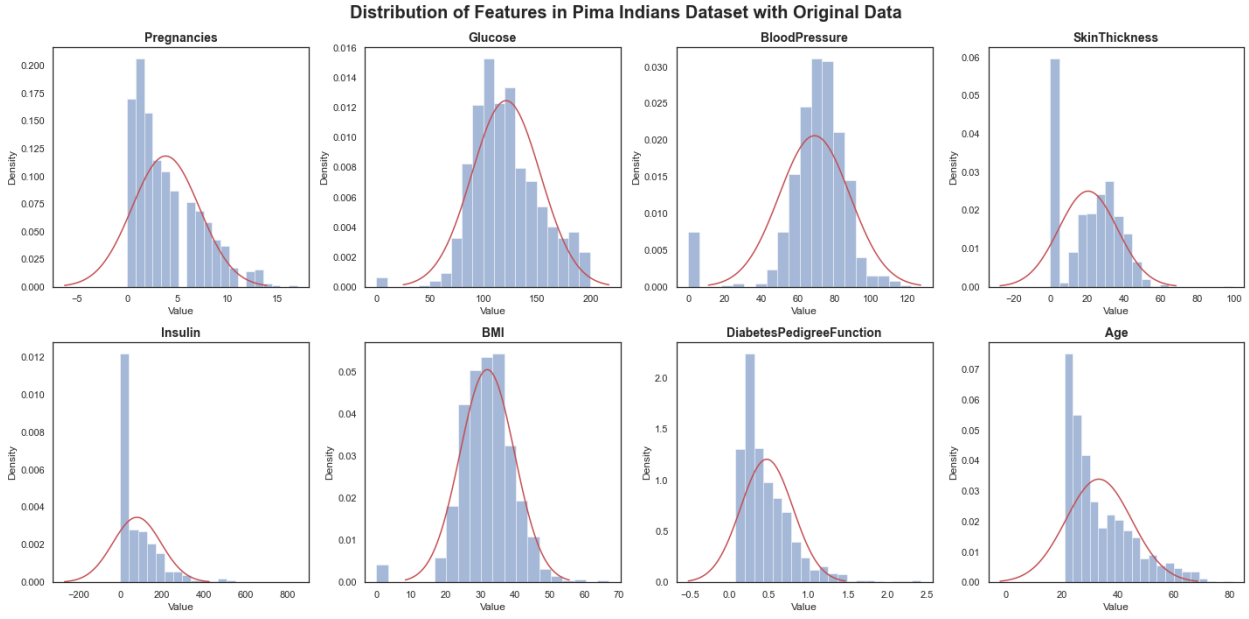
Performance Metrics Used

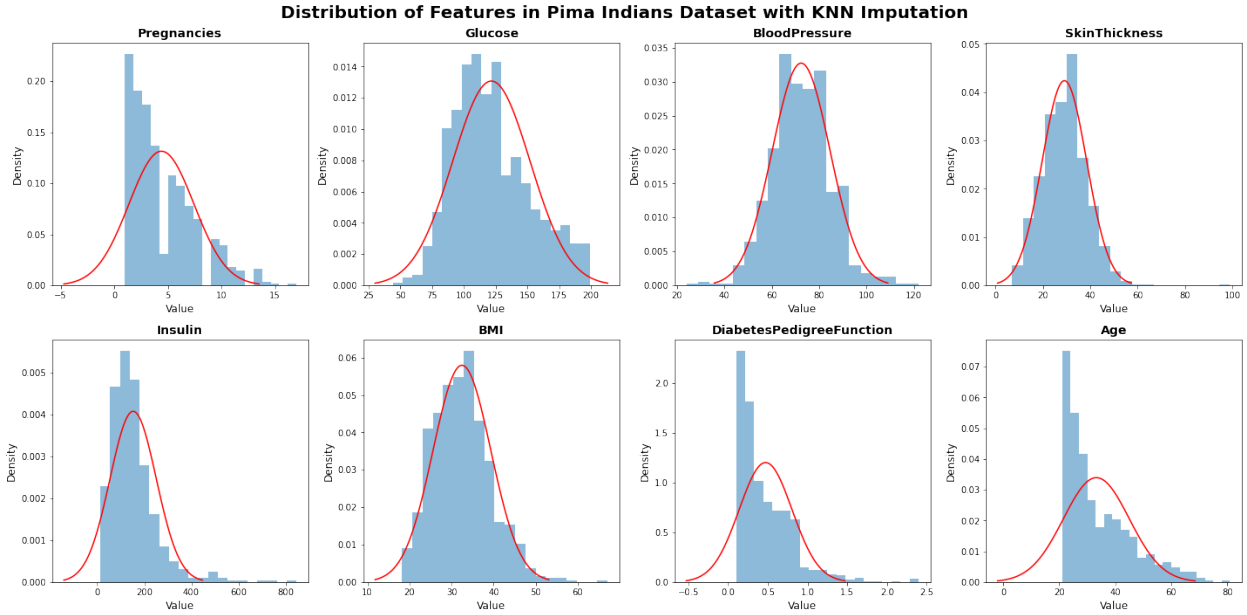
Since the class predictor variable in the Pima Indians diabetes dataset is a binary and is not equally distributed, the best performance metric to use would be the Area Under the Curve score (AUC). The AUC score takes the output of the confusion matrix of a learning algorithm and plots the true positive rate against the false positive rate. The AUC performance metric works has been shown to perform better at distinguishing between binary classes than accuracy score.

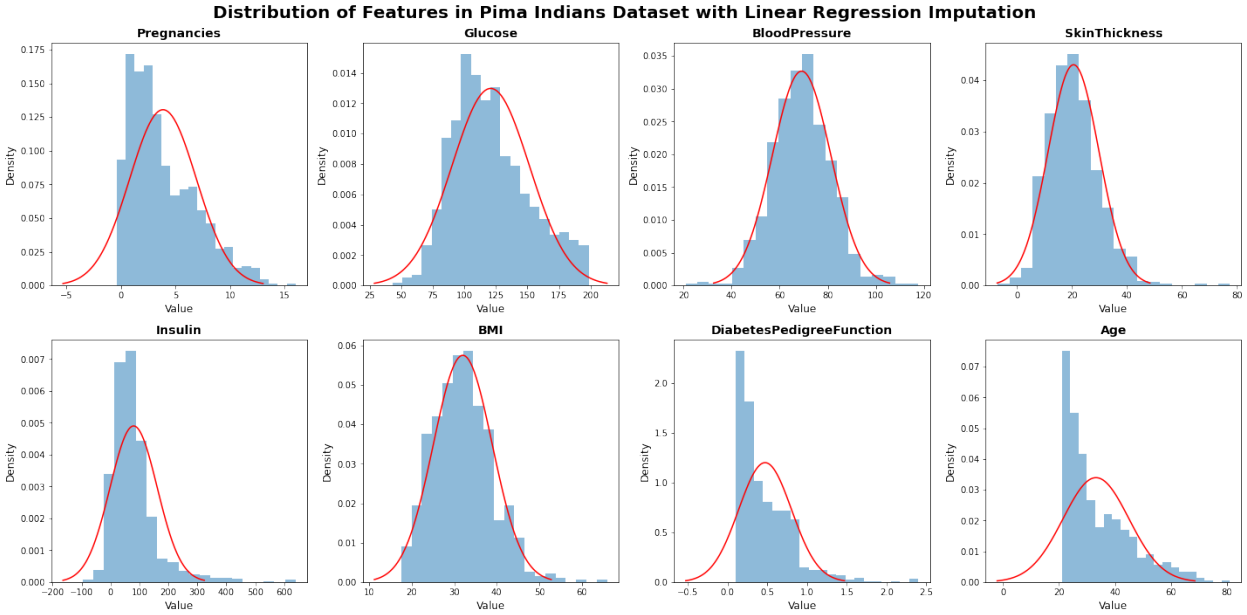


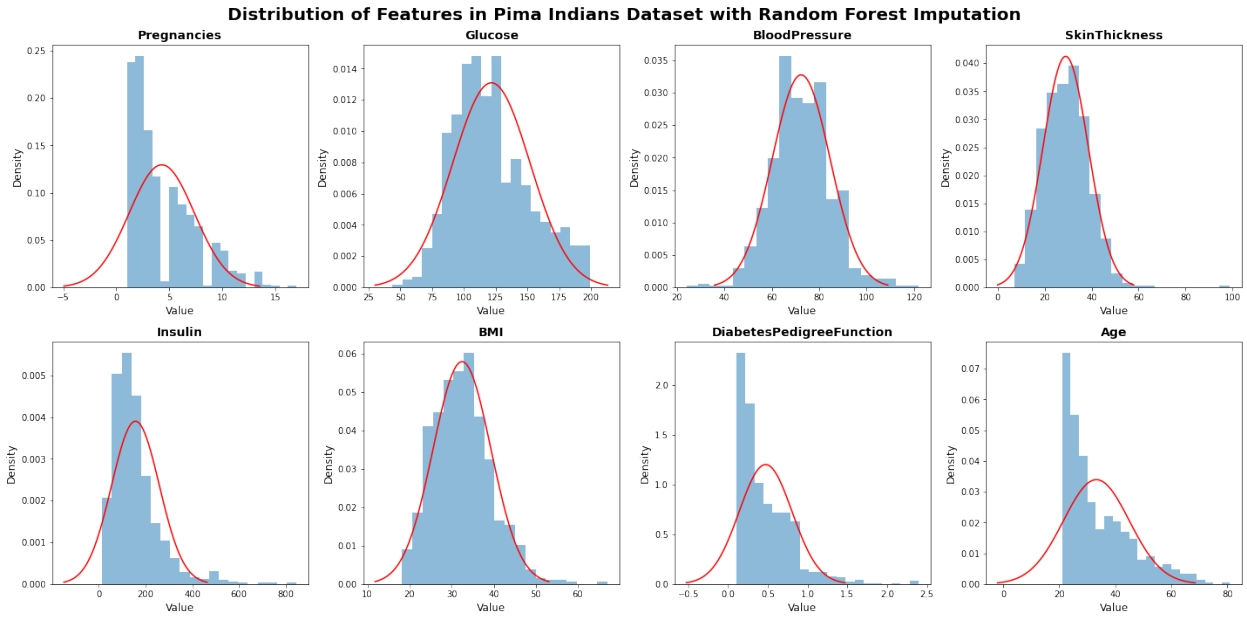
Results of the Experiment

Effects of Imputation on Feature Distribution




Distribution of Features in Pima Indians Dataset with KNN Imputation
Pregnancies
Glucose
BloodPressure
SkinThickness
Insulin
BMI
DiabetesPedigreeFunction
Age
Density
Value


Distribution of Features in Pima Indians Dataset with Linear Regression Imputation
Pregnancies
Glucose
BloodPressure
SkinThickness
Insulin
BMI
DiabetesPedigreeFunction
Age
Density
Value

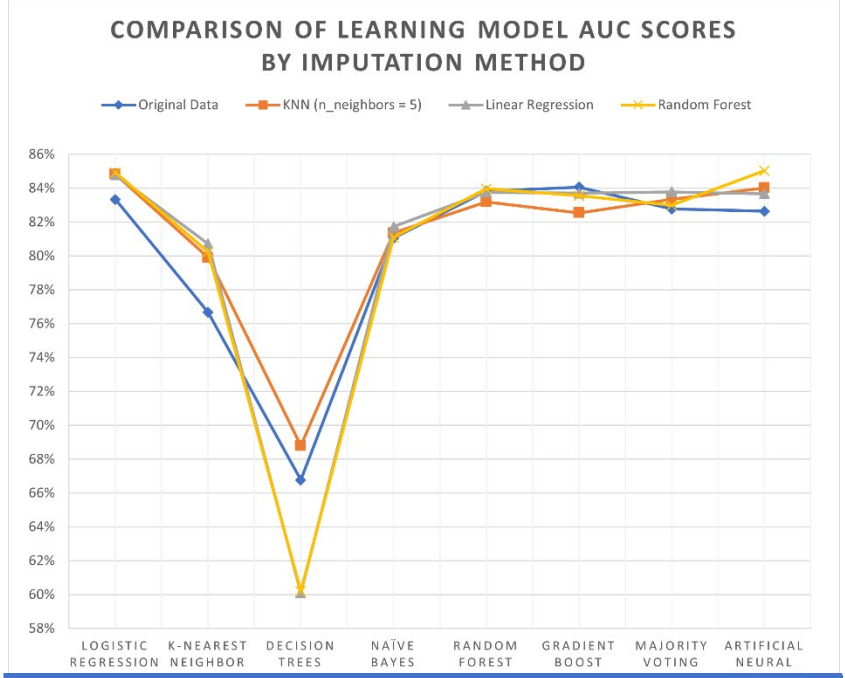


In the original data, shown above, features such as Insulin and Skin Thickness have a pronounced right skewness due to a significant proportion of the null data points being zeroed out. These features become more normalized across all the imputation methods, which can be seen in the charts below the original data distribution. Another unique distribution in the original distribution is on the Age feature. According to the study, there was an artificial age floor placed in the sample collection that created a minimum age of 21. The distribution of this feature skews right, with a significant proportion of patients sampled being around 21-30 years of age. Because the Age feature did not have any null values, the distribution of this variable stays the same across all the imputation methods.

Some other observations that were identified across the three imputation distributions was how each model handled zero values in the Pregnancies feature. It appears that the linear regression imputation method kept a certain percentage of the data points as zero, while KNN and Random Forest kept none and distributed all of the zero values across the distribution. As stated above in the methodology section, due to the null values being zeroed out on the original dataset, it is impossible to determine which patients did have zero pregnancies and which ones had missing data.

Another observation on the linear regression imputation distribution is in the Insulin feature. The left tail of the distribution curve dips into negative values, which is impossible since blood sugar cannot dip below 0.

Imputation Improvement by Learning Model

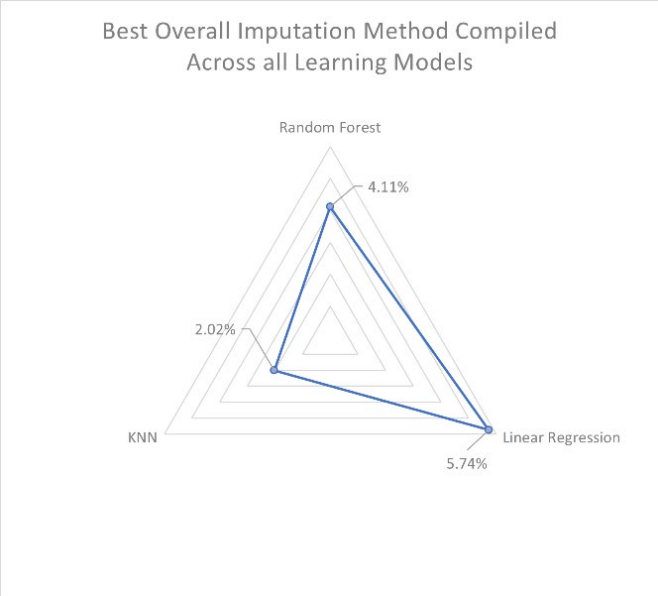


Algorithm	Original Data	KNN (n_neighbors = 5)	Linear Regression	Random Forest
Logistic Regression	83.34%	84.83%	84.79%	84.92%
K-Nearest Neighbor	76.67%	79.90%	80.72%	80.24%
Decision Trees	66.77%	68.79%	60.10%	60.20%
Naïve Bayes	81.05%	81.36%	81.73%	81.10%
Random Forest	83.83%	83.20%	83.76%	83.97%
Gradient Boost	84.07%	82.54%	83.71%	83.56%
Majority Voting	82.77%	83.34%	83.78%	82.99%
Artificial Neural Network	82.64%	84.02%	83.67%	85.03%

The graph above compares the performance of each learning model using the original data and the imputed data. Upon initial inspection, it is evident that the decision tree model had the poorest performance across all imputation methods. As discussed earlier in the methodology section, this could be attributed to the classifier imbalance, which may lead to issues with node splitting.

Upon closer examination of both the graph and the table, it becomes clear that most of the imputation methods resulted in improved performance compared to the original data, with a few exceptions. The decision tree model showed a decrease in performance of over 6.5% with both Linear Regression and Random Forest imputation methods, making the respective models only marginally better than randomly guessing the classifier.

Best Cumulative Imputation Method



Learning Algorithms	Best Imputation by Model	Best Imputation Improvement
Logistic Regression	Random Forest	1.58%
K-Nearest Neighbor	Linear Regression	4.05%
Decision Trees	KNN	2.02%
Naïve Bayes	Linear Regression	0.68%
Random Forest	Random Forest	0.14%
Gradient Boost	Original Data	-0.36%
Majority Voting	Linear Regression	1.01%
Artificial Neural Network	Random Forest	2.39%

One way to determine the best imputation method is to evaluate the improvement percent difference for each learning model using the best imputation method score. Based on the table above, the KNN model using linear regression imputation showed the highest improvement, followed by the ANN learning model using random forest imputation.

The chart above displays the cumulative improvement differential for each imputation method. Linear Regression was the top-performing method, with a cumulative score of 5.74% improvement. Random Forest imputation followed closely behind with a cumulative differential improvement of 4.11%, while KNN imputation had a differential of 2.02%.

Learning Model with Best Average Performance Improvement

Learning Algorithms	KNN	Linear Regression	Random Forest	Average Improvement by Learning model
Logistic Regression	1.49%	1.45%	1.58%	**1.51%**
K-Nearest Neighbor	3.23%	4.05%	3.57%	**3.62%**
Decision Trees	2.02%	-6.67%	-6.57%	**-3.74%**
Naïve Bayes	0.31%	0.68%	0.05%	**0.35%**
Random Forest	-0.63%	-0.07%	0.14%	**-0.19%**
Gradient Boost	-1.53%	-0.36%	-0.51%	**-0.80%**
Majority Voting	0.57%	1.01%	0.22%	**0.60%**
Artificial Neural Network	1.38%	1.03%	2.39%	**1.60%**

The table above displays the average improvement percentage across all learning models used on the dataset. The KNN model showed consistent improvement across all three different imputation methods, with an average improvement of 3.62%. In contrast, the decision tree learning model had an average percent improvement of -3.74%, and both linear and random forest imputation methods performed over 6.5% worse than the original data performance.

One possible reason why linear regression imputation may not improve the performance of a decision tree model is that it assumes a linear relationship between feature variables when filling in null values. However, this assumption may not hold true in many cases, particularly when there are nonlinear relationships between variables. Additionally, the classifier imbalance issue mentioned earlier in this paper can lead to bias that favors the dominant classifier for both linear regression and random forest imputation methods, further impacting the performance of the decision tree model.

Some other learning models that showed negative average improvement across all imputation methods are random forest and gradient boosting.

Best Overall Learning Model Performance

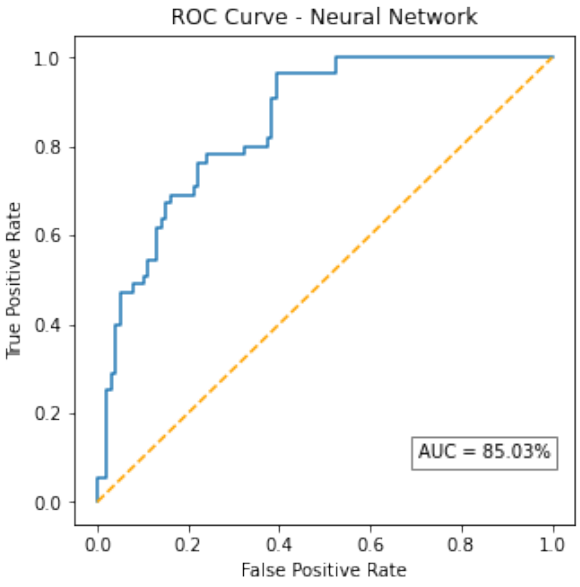
The learning model that had the best overall performance across all imputation methods was the Keras artificial neural network, with an AUC score of 85.03% using the random forest imputation method. The hyper-parameters that were used in cross-validation to fine-tune the model are as follows:

Batch Size:	[16, 32, 64]
Epochs:	[50, 100, 150]
Activation function:	['relu', 'tanh' , 'sigmoid']
Number of Hidden Layers:	[1, 2, 3, 4]
Number of Neurons:	[16, 32, 64]
Optimizer:	['adam', 'rmsprop']

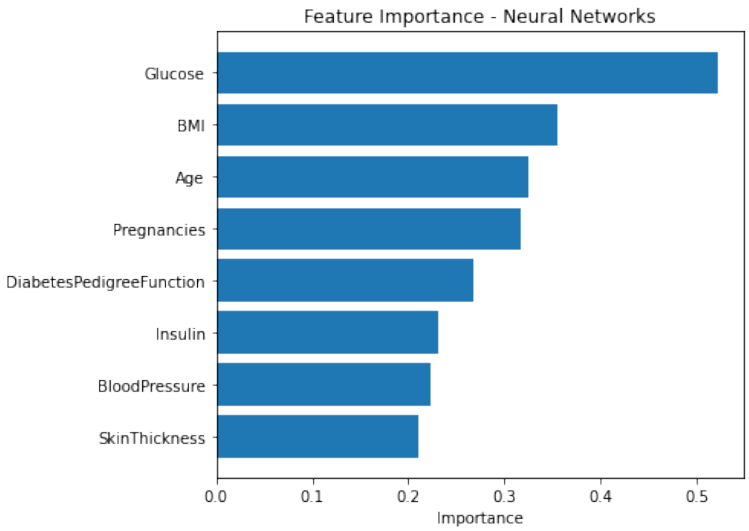
After cross validation was complete using the conditions above, the best combination of hyper-parameters that yielded the highest performance results on the training data was:

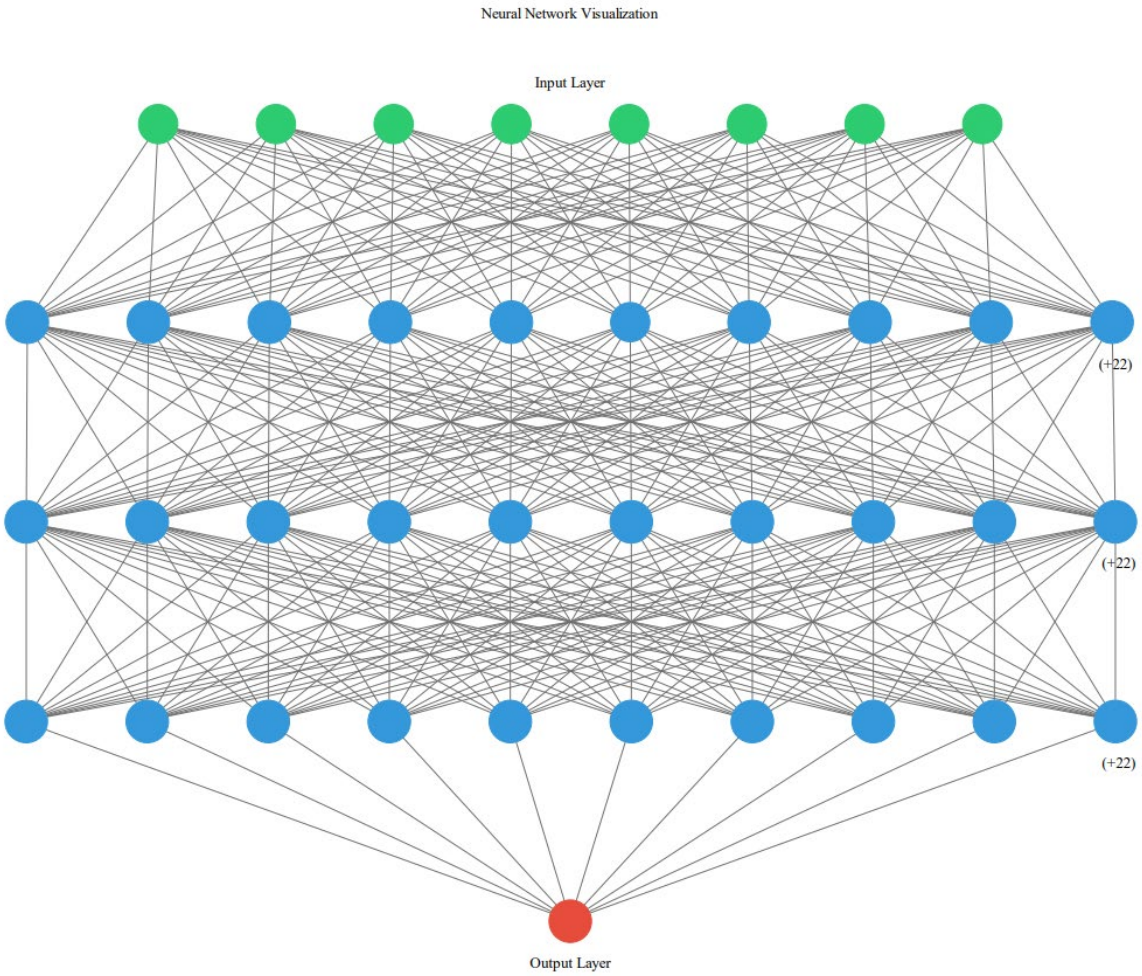
'activation': 'sigmoid', 'batch_size': 64, 'epochs': 150, 'hidden_layers': 3, 'neurons': 32, 'optimizer': 'adam'

Applied to the test data, the following fine-tuned hyper-parameters yielded the resulting ROC curve with accompanying AUC score as well as feature importance graph:



The graph above depicting feature importance shows that the glucose feature is the most crucial variable for predicting the diagnosis using the artificial neural network model. Interestingly, the feature importance ranking displayed in the graph aligns with the ranking obtained through the correlation matrix analysis conducted on the outcome variable, as detailed in the methodology section.




Neural Network Visualization
Input Layer
(+22)
(+22)
(+22)
Output Layer

Best Hyper-Parameters by Learning Model and Imputation Method

Learning Model	Original Data
Logistic Regression	'C': 1.0, 'max_iter': 100, 'penalty': 'l2', 'solver': 'newton-cg'
K-Nearest Neighbor	'metric': 'euclidean', 'n_neighbors': 15, 'weights': 'uniform'
Decision Trees	'criterion': 'log_loss', 'splitter': 'best'
Naïve Bayes	'var_smoothing': 1e-09
Random Forest	'criterion': 'entropy', 'n_estimators': 500, 'n_jobs': -1
Gradient Boost	'learning_rate': 0.1, 'loss': 'exponential', 'n_estimators': 50
Majority Voting	('lr', lr) , ('gnb', gnb) , ('knn', knn)], voting='soft'
Artificial Neural Network	'activation': 'relu', 'batch_size': 16, 'epochs': 50, 'hidden_layers': 1, 'neurons': 64, 'optimizer': 'rmsprop'

Learning Model	KNN Imputation Method
Logistic Regression	'C': 100, 'max_iter': 100, 'penalty': 'none', 'solver': 'newton-cg'
K-Nearest Neighbor	'metric': 'euclidean', 'n_neighbors': 13, 'weights': 'distance'
Decision Trees	'criterion': 'log_loss', 'splitter': 'best'
Naïve Bayes	'var_smoothing': 1e-09
Random Forest	'criterion': 'entropy', 'n_estimators': 100, 'n_jobs': -1
Gradient Boost	'learning_rate': 0.1, 'loss': 'log_loss', 'n_estimators': 50
Majority Voting	('lr', lr) , ('gnb', gnb) , ('knn', knn)], voting='soft'
Artificial Neural Network	'activation': 'relu', 'batch_size': 32, 'epochs': 50, 'hidden_layers': 1, 'neurons': 16, 'optimizer': 'adam'

Learning Model	Linear Regression Imputation Method
Logistic Regression	'C': 100, 'max_iter': 100, 'penalty': 'none', 'solver': 'newton-cg'
K-Nearest Neighbor	'metric': 'euclidean', 'n_neighbors': 17, 'weights': 'distance'
Decision Trees	'criterion': 'gini', 'splitter': 'best'
Naïve Bayes	'var_smoothing': 1e-09
Random Forest	criterion': 'entropy', 'n_estimators': 50, 'n_jobs': -1
Gradient Boost	'learning_rate': 0.01, 'loss': 'exponential', 'n_estimators': 500
Majority Voting	('lr', lr) , ('gnb', gnb) , ('knn', knn)], voting='soft'
Artificial Neural Network	'activation': 'relu', 'batch_size': 32, 'epochs': 50, 'hidden_layers': 1, 'neurons': 32, 'optimizer': 'adam'

Learning Model	Random Forest Imputation Method
Logistic Regression	'C': 100, 'max_iter': 100, 'penalty': 'none', 'solver': 'newton-cg'
K-Nearest Neighbor	'metric': 'euclidean', 'n_neighbors': 19, 'weights': 'distance'
Decision Trees	'criterion': 'entropy', 'splitter': 'random'
Naïve Bayes	'var_smoothing': 1e-09
Random Forest	'criterion': 'gini', 'n_estimators': 50, 'n_jobs': -1
Gradient Boost	'learning_rate': 0.01, 'loss': 'exponential', 'n_estimators': 500
Majority Voting	('lr', lr) , ('gnb', gnb) , ('knn', knn)], voting='soft'
Artificial Neural Network	'activation': 'sigmoid', 'batch_size': 64, 'epochs': 150, 'hidden_layers': 3, 'neurons': 32, 'optimizer': 'adam'

Conclusion

In conclusion, this study aimed to evaluate the effects of imputation methods on feature distribution and the performance of learning models in diabetes prediction. The results indicate that most imputation methods resulted in improved performance compared to the original data, except for the decision tree model and, to a lesser extent, the gradient boosting model. The KNN model consistently showed improvement across all three imputation methods, and had the highest improvement score of all the learning models tested with a 4.05% improvement using linear regression

imputation. The Keras artificial neural network with random forest imputation yielded the highest AUC score of 85.03%, with the glucose feature identified as the most important variable for predicting the diagnosis using the artificial neural network model.

The study also identified unique feature observations in the original data distribution, such as the right skewness in the Insulin and Skin Thickness features due to a significant proportion of the null data points being zeroed out. The Age feature was found to have a right skewed distribution, with a significant proportion of patients being around 21-30 years of age. The study also highlighted how different imputation methods handled zero values in the Pregnancies feature and observed the left tail dipping into negative values in the Insulin feature for the linear regression imputation distribution.

Overall, the study provides insight into the effects of imputation methods on feature distribution and the performance of learning models. The results suggest that the KNN model with linear regression imputation is the best cumulative imputation method, while the Keras artificial neural network with random forest imputation is the best overall learning model for diabetes prediction. However, further research may be needed to evaluate the performance of different learning models and imputation methods on larger datasets and different medical conditions.

References

Bennett, P. H., Burch, T. A., & Miller, M. (1971). Diabetes Mellitus in American (Pima) Indians. *The Lancet*, 125-128.

Bento, C. (2021, June 28). *Decision Tree Classifier explained in real-life: picking a vacation destination*. Retrieved from Towards Data Science: https://towardsdatascience.com/decision-tree-classifier-explained-in-real-life-picking-a-vacation-destination-6226b2b60575

Biessels, G. J., & Despa, F. (2018). Cognitive decline and dementia in diabetes mellitus: mechanisms and clinical implications. *Nature Reviews Endocrinology, Volume 14*, 591-604.

Biju, S. M., & James, K. C. (2022). Some Studies and Inferences on Pima Indian Diabetes Data. *Journal of Statistics and Mathematical Engineering*, 1-9.

Brownlee, J. (2020, August 21). *Cost-Sensitive Decision Trees for Imbalanced Classification.* Retrieved from Machine Learning Mastery: https://machinelearningmastery.com/cost-sensitive-decision-trees-for-imbalanced-classification/#:~:text=The%20decision%20tree%20algorithm%20is,two%20groups%20with%20minimum%20mixing.

Chang, V., Bailey, J., Xu, Q. A., & Sun, Z. (2022, March 24). *Pima Indians diabetes mellitus classification based on machine learning (ML) algorithms*. Retrieved from Neural Computing and Applications: https://doi.org/10.1007/s00521-022-07049-z

Corporate Finance Institute. (2023, January 8). *Gradient Boosting: A method used in building predictive models.* Retrieved from Corporate Finance Institute: https://corporatefinanceinstitute.com/resources/data-science/gradient-boosting/

Dash, S. K. (2022, September 22). *Handling Missing Values with Random Forest.* Retrieved from Analytics Vidhya: https://www.analyticsvidhya.com/blog/2022/05/handling-missing-values-with-random-forest/

Faisal, S. (2018, March 29). 2.3 Nearest Neighbors Method. In *Nearest Neighbor Methods for the Imputation of Missing Values in Low and High-Dimensional Data* (pp. 14-15). Gottingen: Cuvillier Verlag. Retrieved from https://search-ebscohost-com.proxy.lib.wayne.edu/login.aspx?direct=true&db=e000xna&AN=2130272&site=ehost-live&scope=site

Gandhi, R. (2018, May 5). *Naive Bayes Classifier*. Retrieved from Towards Data Science: https://towardsdatascience.com/naive-bayes-classifier-81d512f50a7c

Inter Tribal Council of Arizona. (2023, March 30). *Salt River Pima-Maricopa Indian Community*. Retrieved from Inter Tribal Council of Arizona: https://itcaonline.com/member-tribes/salt-river-pima-maricopa-indian-community/#:~:text=Consisting%20of%2052%2C600%20acres%2C%20the,Tempe%2C%20Fountain%20Hills%20and%20Mesa.

Ozsahin, D. U., Mustapha, M. T., Mubarak, A. S., Ameen, Z. S., & Uzun, B. (2022). Impact of Outliers and Dimensionality Reduction on the Performance of Predictive Models for Medical Disease Diagnosis. *International Conference on Artificial Intelligence in Everything*, 79-86.

scikit learn. (2023). *Decision Trees*. Retrieved from scikit learn: https://scikit-learn.org/stable/modules/tree.html

Seth, N. (2021, June 8). *How does Backward Propagation Work in Neural Networks?* Retrieved from Analytics Vidhya: https://www.analyticsvidhya.com/blog/2021/06/how-does-backward-propagation-work-in-neural-networks/

Sruthi, E. R. (2023, March 24). *Understand Random Forest Algorithms With Examples (Updated 2023)*. Retrieved from Analytics Vidhya: https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/

Swalin, A. (2018, January 30). *How to Handle Missing Data.* Retrieved from Towards Data Science: https://towardsdatascience.com/how-to-handle-missing-data-8646b18db0d4

Tan, P.-N., Steinbach, M., Karpatne, A., & Kumar, V. (2022). 6.10.6 Random Forests. In *Introduction to Data Mining: Second Edition* (pp. 512-514). Uttar Pradesh: Pearson.

Thorn, J. (2020, February 8). *Logistic Regression Explained*. Retrieved from Towards Data Science: https://towardsdatascience.com/logistic-regression-explained-9ee73cede081

UCI Machine Learning. (2023, March 25). *Pima Indians Diabetes Database.* Retrieved from Kaggle: https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database

World Health Organization. (2022, September 16). *Diabetes.* Retrieved from World Health Organization: https://www.who.int/news-room/fact-sheets/detail/diabetes

Appendix

Appendix A: Python Code

In [1]:

```
#!pip install tensorflow
#!pip install ann-visualizer
#!pip install graphviz
#!pip install altair
#!pip install altair_viewer
```

In [120…

```
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings("ignore")

# Data Visualization Packages
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as stats

# Data Pre-Processing Packages
from sklearn.preprocessing import StandardScaler
import random
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import KNNImputer, IterativeImputer
from sklearn.linear_model import LinearRegression
import altair as alt
from sklearn.model_selection import train_test_split , GridSearchCV

# Individual Learning Packages
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB

# Ensemble Learning Packages
from sklearn.ensemble import RandomForestClassifier , GradientBoostingClassifier, RandomForestRegressor, VotingClassifie

# Evaluation Metrics Packages
```

```
from sklearn.metrics import accuracy_score , classification_report, confusion_matrix , roc_curve , roc_auc_score

# Neural Network Packages

from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from keras.utils import plot_model
import json
from ann_visualizer.visualize import ann_viz
```

In [121…

```
url = 'https://raw.githubusercontent.com/jmrieck17/CSC_7810_Final_Project/main/diabetes.csv'
df = pd.read_csv(url)

print(df.head(5))
```

```
   Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0            6      148             72             35        0  33.6
1            1       85             66             29        0  26.6
2            8      183             64              0        0  23.3
3            1       89             66             23       94  28.1
4            0      137             40             35      168  43.1

   DiabetesPedigreeFunction  Age  Outcome
0                     0.627   50        1
1                     0.351   31        0
2                     0.672   32        1
3                     0.167   21        0
4                     2.288   33        1
```

In [122…

```
print('Dataframe Shape:' , df.shape)
print()
print('Column Variables:')
print(list(df.columns))
print()
```

```
print('Unique Variables by Column:')
print(df.nunique())
```

```
Dataframe Shape: (768, 9)

Column Variables:
['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcom
e']

Unique Variables by Column:
Pregnancies                 17
Glucose                    136
BloodPressure               47
SkinThickness               51
Insulin                    186
BMI                        248
DiabetesPedigreeFunction   517
Age                         52
Outcome                      2
dtype: int64
```

In [123…

```
print('Rows with Null Values by Column:')
df.isnull().sum()
```

Out[123…

```
Rows with Null Values by Column:
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

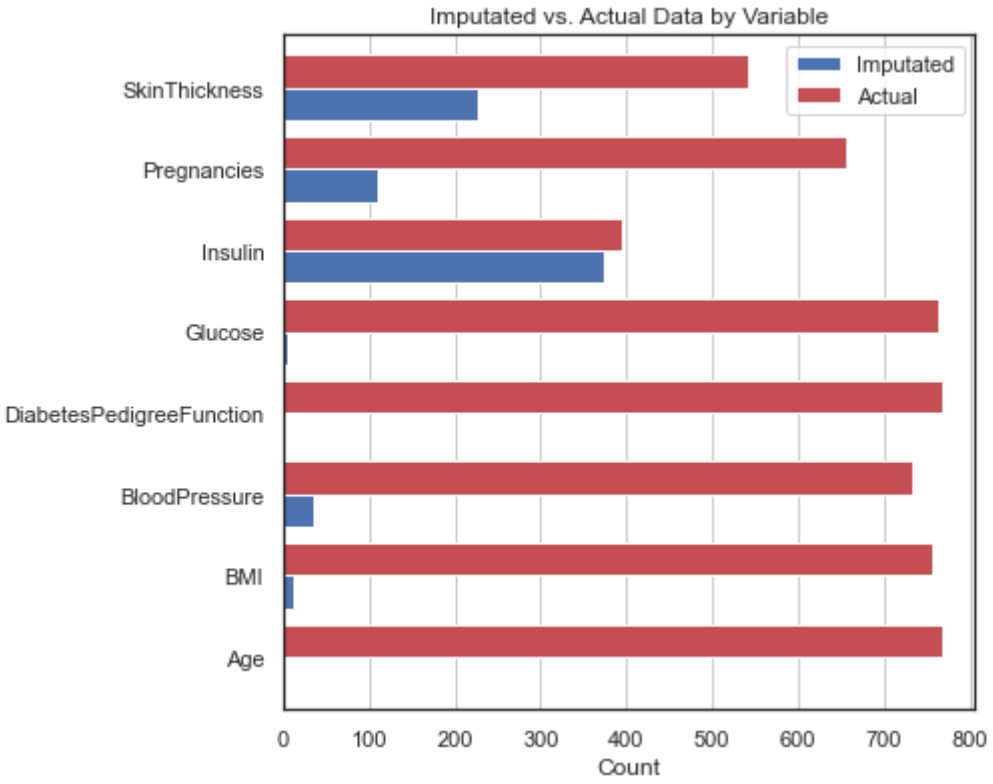
In [124…

```
X = df.drop('Outcome', axis = 1)
y = df.Outcome
```

In [125…

```
count_zero_cols = (X.eq(0)).sum()
count_nonzero_cols = (X.ne(0)).sum()

# combine both counts into one DataFrame
combined_counts = pd.concat([count_zero_cols, count_nonzero_cols], axis=1)
combined_counts.columns = ['Imputated', 'Actual']

# sort the column names alphabetically
combined_counts = combined_counts.sort_index()

# define color palette
colors = ['#4C72B0', '#C44E52']

# plot the results in a horizontal bar chart with sorted index
fig, ax = plt.subplots(figsize=(8, 6))
combined_counts.sort_index().plot(kind='barh', ax=ax, color=colors, width=0.8)

# add a grid
ax.grid(axis='x')

# add space for the axis labels
plt.subplots_adjust(left=0.3, bottom=0.1)

# add title and axis labels
ax.set_title("Imputated vs. Actual Data by Variable")
ax.set_xlabel("Count")

# create a table of counts
table_data = combined_counts.reset_index().values.tolist()
table_data.insert(0, ['Variable', 'Imputated', 'Actual'])
table = ax.table(cellText=table_data, loc='right', bbox=[1.2, 0, 0.5, 1])
table.auto_set_font_size(False)
```

```
table.set_fontsize(12)

# adjust bbox and font size to fit variable names
table.auto_set_column_width(col=list(range(combined_counts.shape[1])))
table.scale(3, 3)
table.auto_set_font_size(False)
table.set_fontsize(8)

plt.show()
```

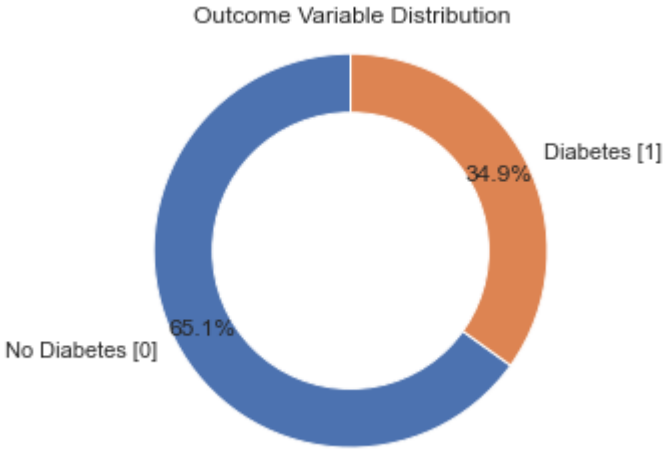


Variable	Imputated	Actual
Age	0	768
BMI	11	757
BloodPressure	35	733
DiabetesPedigreeFunction	0	768
Glucose	5	763
Insulin	374	394
Pregnancies	111	657
SkinThickness	227	541

In [126…

```
counts = df.groupby('Outcome').size()
```

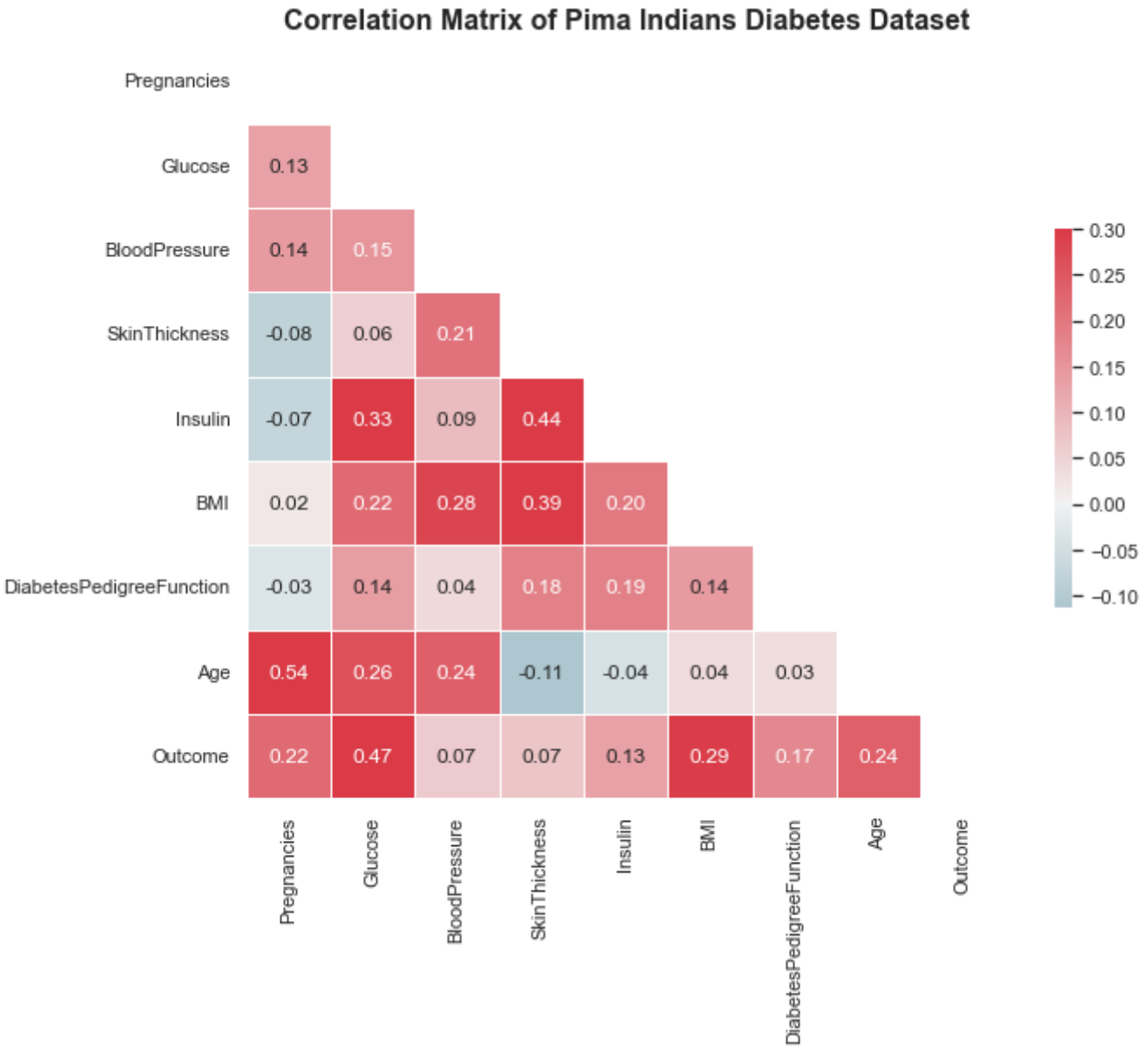
```
fig, ax = plt.subplots()
ax.pie(counts, labels=['No Diabetes [0]', 'Diabetes [1]'], autopct='%1.1f%%', startangle=90, pctdistance=0.85)
centre_circle = plt.Circle((0,0),0.70,fc='white')
fig = plt.gcf()
fig.gca().add_artist(centre_circle)
ax.axis('equal')
plt.title('Outcome Variable Distribution')
plt.show()
```



In [127…

```
corr_matrix = df.corr()

# Set up the plot
sns.set(style="white")
mask = np.zeros_like(corr_matrix, dtype=bool)
mask[np.triu_indices_from(mask)] = True
fig, ax = plt.subplots(figsize=(10, 8))
cmap = sns.diverging_palette(220, 10, as_cmap=True)

# Plot the correlation matrix
sns.heatmap(corr_matrix, mask=mask, cmap=cmap, vmax=.3, center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5}, annot=True, fmt=".2f")
```

```
# Rotate the x-axis labels
plt.xticks(rotation=90)

plt.title("Correlation Matrix of Pima Indians Diabetes Dataset", fontweight="bold", fontsize=16)

plt.show()
```


Correlation Matrix of Pima Indians Diabetes Dataset
Pregnancies
Glucose 0.13
BloodPressure 0.14 0.15
SkinThickness -0.08 0.06 0.21
Insulin -0.07 0.33 0.09 0.44
BMI 0.02 0.22 0.28 0.39 0.20
DiabetesPedigreeFunction -0.03 0.14 0.04 0.18 0.19 0.14
Age 0.54 0.26 0.24 -0.11 -0.04 0.04 0.03
Outcome 0.22 0.47 0.07 0.07 0.13 0.29 0.17 0.24
Pregnancies
Glucose
BloodPressure
SkinThickness
Insulin
BMI
DiabetesPedigreeFunction
Age
Outcome
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10

In [128…

```python
# This wrapper allows for me to choose which imputation method I want to use to fill in missing data

def impute_missing_values(df, method='KNN', n_neighbors=5):

    imputer = None

    if method == 'KNN':
        X = df.values.copy()
        X[X == 0] = np.nan
        imputer = KNNImputer(n_neighbors=n_neighbors)
        X_imputed = imputer.fit_transform(X)
        X_imputed = np.round(X_imputed, decimals=1)
        df_imputed = pd.DataFrame(X_imputed, columns=df.columns)

    elif method == 'Random Forest':
        X = df.values.copy()
        X[X == 0] = np.nan
        estimator = RandomForestRegressor(random_state=0)
        imputer = IterativeImputer(random_state=0, estimator=estimator)
        X_imputed = imputer.fit_transform(X)
        X_imputed = np.round(X_imputed, decimals=1)
        df_imputed = pd.DataFrame(X_imputed, columns=df.columns)

    elif method == 'Linear Regression':
        X = df.values.copy()
        X[X == 0] = np.nan
        imputer = KNNImputer(n_neighbors=n_neighbors)
        X_imputed = imputer.fit_transform(X)
        X_imputed[X == 0] = np.nan
        model = LinearRegression()
        model.fit(X_imputed, df.values)
        X_pred = model.predict(X_imputed)
        X_pred = np.round(X_pred, 1)
        X_pred[X == 0] = 0
        df_imputed = pd.DataFrame(X_pred, columns=df.columns)

    else:
```

```
        raise ValueError("Invalid imputation method. Must be one of ['KNN', 'Random Forest', 'Linear Regression']")

    return df_imputed
```

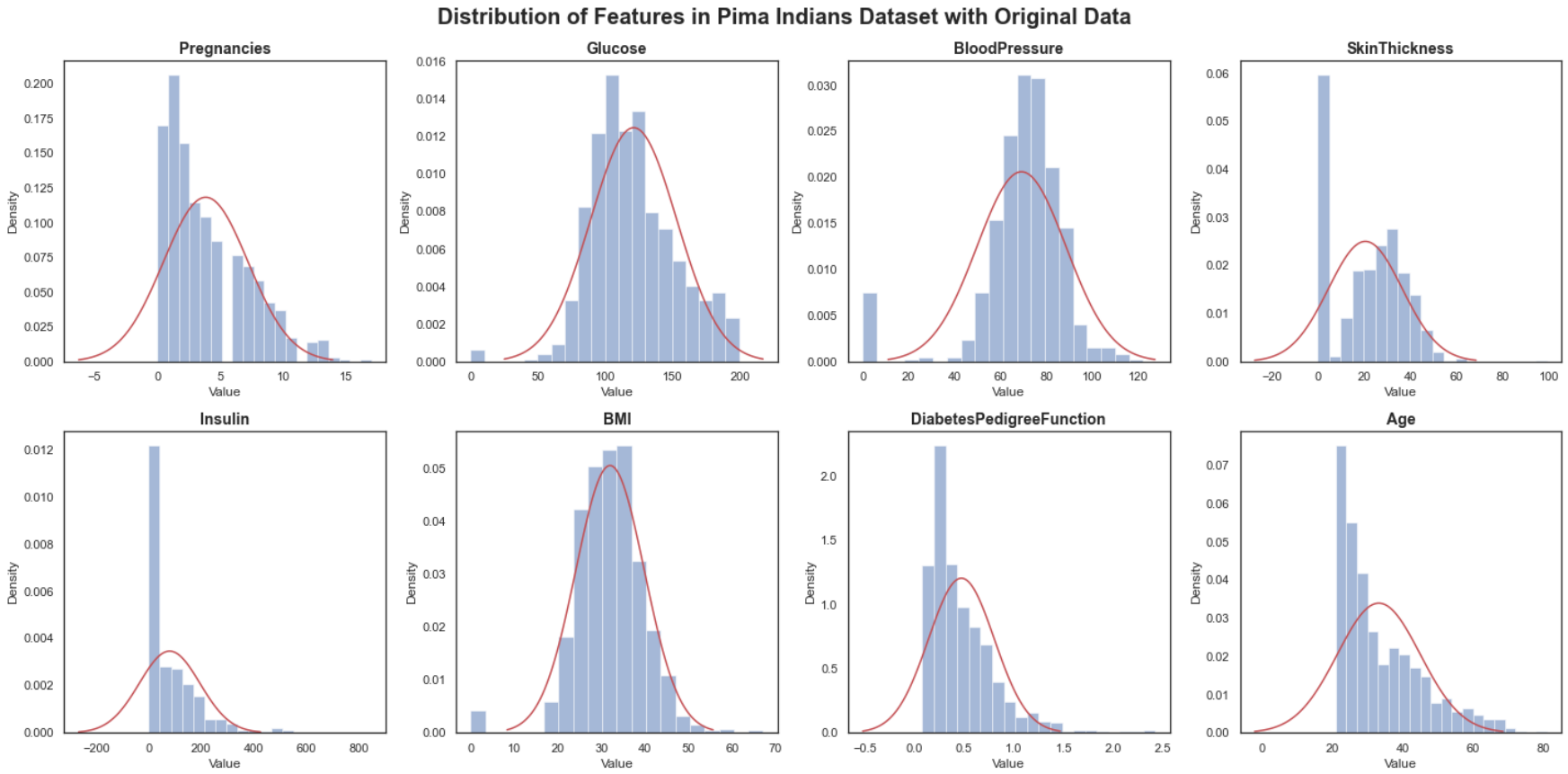
In [129…

```
imputed_X = impute_missing_values(X, method='Linear Regression')
print(imputed_X)
```

```
     Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0            5.7    147.4           69.9           22.8     60.1  33.6
1            0.8     83.8           63.0           21.2      7.2  26.1
2            7.6    183.4           60.4           16.5     73.4  23.3
3            0.6     87.8           61.9           16.9     53.4  27.4
4            4.6    136.4           40.6           40.1    180.6  43.1
..           ...      ...            ...            ...      ...   ...
763         10.1    100.1           73.7           28.3     34.2  32.8
764          1.3    120.9           66.4           20.6    105.6  36.3
765          4.6    120.5           67.8           13.4     38.5  25.6
766          1.1    124.6           58.2           23.9     38.3  30.3
767          0.5     91.8           66.2           24.4     23.3  29.9

     DiabetesPedigreeFunction   Age
0                         0.6  50.0
1                         0.4  31.0
2                         0.7  32.0
3                         0.2  21.0
4                         2.3  33.0
..                        ...   ...
763                       0.2  63.0
764                       0.3  27.0
765                       0.2  30.0
766                       0.3  47.0
767                       0.3  23.0

[768 rows x 8 columns]
```

In [131…

```
fig, axs = plt.subplots(nrows=2, ncols=4, figsize=(20, 10))
axs = axs.flatten()

for i, col in enumerate(imputed_X.columns):
    data = imputed_X[col]
    mean, std = data.mean(), data.std()
    xmin, xmax = mean - 3 * std, mean + 3 * std
    x = np.linspace(xmin, xmax, 100)
    axs[i].hist(data, density=True, bins=20, alpha=0.5)
    axs[i].plot(x, stats.norm.pdf(x, mean, std), 'r')
    axs[i].set_title(col, fontsize=14, fontweight='bold')
    axs[i].set_xlabel('Value', fontsize=12)
    axs[i].set_ylabel('Density', fontsize=12)

plt.suptitle('Distribution of Features in Pima Indians Dataset with Linear Regression', fontsize=20, fontweight='bold')
plt.tight_layout()
plt.show()
```



In [100…

```
# this wrapper allows me to choose whether I want to scale my data
# If using imputation method, use X_imputed, otherwise use X

def scale_data(X, scaling):
    if scaling:
        scaler = StandardScaler()
        X_scaled = scaler.fit_transform(X)
        X_scaled = np.round(X_scaled, decimals=2)
        return pd.DataFrame(X_scaled, columns=X.columns)
```

```
    else:
        return X
```

In [101…

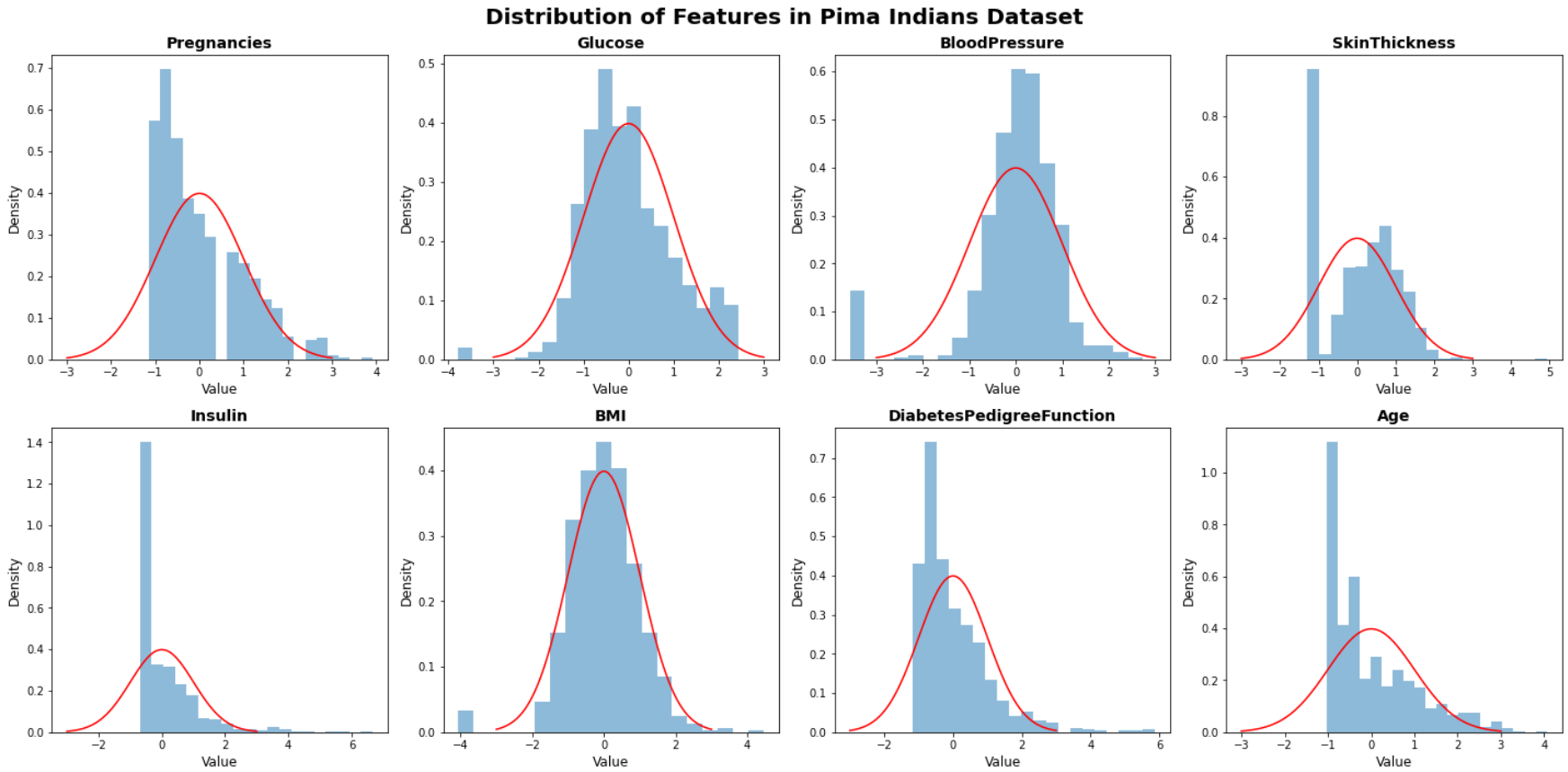
```
X_scaled = scale_data(X, scaling = True)
print(X_scaled)
```

```
     Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0           0.64     0.85           0.15           0.91    -0.69  0.20
1          -0.84    -1.12          -0.16           0.53    -0.69 -0.68
2           1.23     1.94          -0.26          -1.29    -0.69 -1.10
3          -0.84    -1.00          -0.16           0.15     0.12 -0.49
4          -1.14     0.50          -1.50           0.91     0.77  1.41
..           ...      ...            ...            ...      ...   ...
763         1.83    -0.62           0.36           1.72     0.87  0.12
764        -0.55     0.03           0.05           0.41    -0.69  0.61
765         0.34     0.00           0.15           0.15     0.28 -0.74
766        -0.84     0.16          -0.47          -1.29    -0.69 -0.24
767        -0.84    -0.87           0.05           0.66    -0.69 -0.20

     DiabetesPedigreeFunction   Age
0                        0.47  1.43
1                       -0.37 -0.19
2                        0.60 -0.11
3                       -0.92 -1.04
4                        5.48 -0.02
..                        ...   ...
763                     -0.91  2.53
764                     -0.40 -0.53
765                     -0.69 -0.28
766                     -0.37  1.17
767                     -0.47 -0.87

[768 rows x 8 columns]
```

In [102…

```
fig, axs = plt.subplots(nrows=2, ncols=4, figsize=(20, 10))
axs = axs.flatten()
```

```
for i, col in enumerate(X_scaled.columns):
    data = X_scaled[col]
    mean, std = data.mean(), data.std()
    xmin, xmax = mean - 3 * std, mean + 3 * std
    x = np.linspace(xmin, xmax, 100)
    axs[i].hist(data, density=True, bins=20, alpha=0.5)
    axs[i].plot(x, stats.norm.pdf(x, mean, std), 'r')
    axs[i].set_title(col, fontsize=14, fontweight='bold')
    axs[i].set_xlabel('Value', fontsize=12)
    axs[i].set_ylabel('Density', fontsize=12)

plt.suptitle('Distribution of Features in Pima Indians Dataset', fontsize=20, fontweight='bold')
plt.tight_layout()
plt.show()
```



In [103…

```
random.seed(1234)
```

In [104…

```
X_train,X_test,y_train,y_test = train_test_split(X_scaled,y,test_size=0.2 , random_state = 1234)
```

Individual ML Methods

Logistic Regression

In [260…

```
parameters_lr = {
    'penalty' : ['none' , 'l1' , 'l2'],
    'C'       : [100 , 10 , 1.0 , 0.1 , 0.01],
    'solver'  : ['newton-cg' , 'lbfgs' , 'sag' , 'saga'],
    'max_iter' : [100]
}

cv = 5 # so this will split the dataset into 5 kfold validation chunks for the GridSearchCv to run through

lr = LogisticRegression()
clf = GridSearchCV(lr,
                   param_grid = parameters_lr,
                   scoring = 'accuracy',
                   cv = cv)

clf.fit(X_train , y_train)

print("Tuned Hyperparameters :", clf.best_params_)
print("Accuracy: {:.2%}".format(clf.best_score_))
```

```
Tuned Hyperparameters : {'C': 100, 'max_iter': 100, 'penalty': 'none', 'solver': 'newton-cg'}
Accuracy: 78.02%
```

In [261…

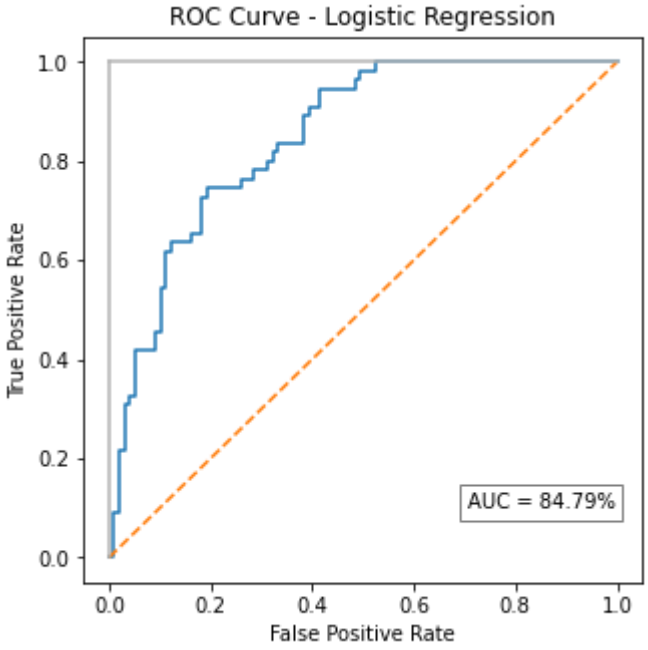
```
lr = LogisticRegression(**clf.best_params_)
lr.fit(X_train , y_train)
predict_lr = lr.predict(X_test)

print("Accuracy: {:.2%}".format(accuracy_score(y_test, predict_lr)))
print('Confusion Matrix:')
print(confusion_matrix(y_test, predict_lr))
print('Classification Report Table:')
print(classification_report(y_test, predict_lr))
```

```
Accuracy: 77.27%
Confusion Matrix:
[[89 10]
 [25 30]]
Classification Report Table:
              precision    recall  f1-score   support

           0       0.78      0.90      0.84        99
           1       0.75      0.55      0.63        55

    accuracy                           0.77       154
   macro avg       0.77      0.72      0.73       154
weighted avg       0.77      0.77      0.76       154
```

In [262…

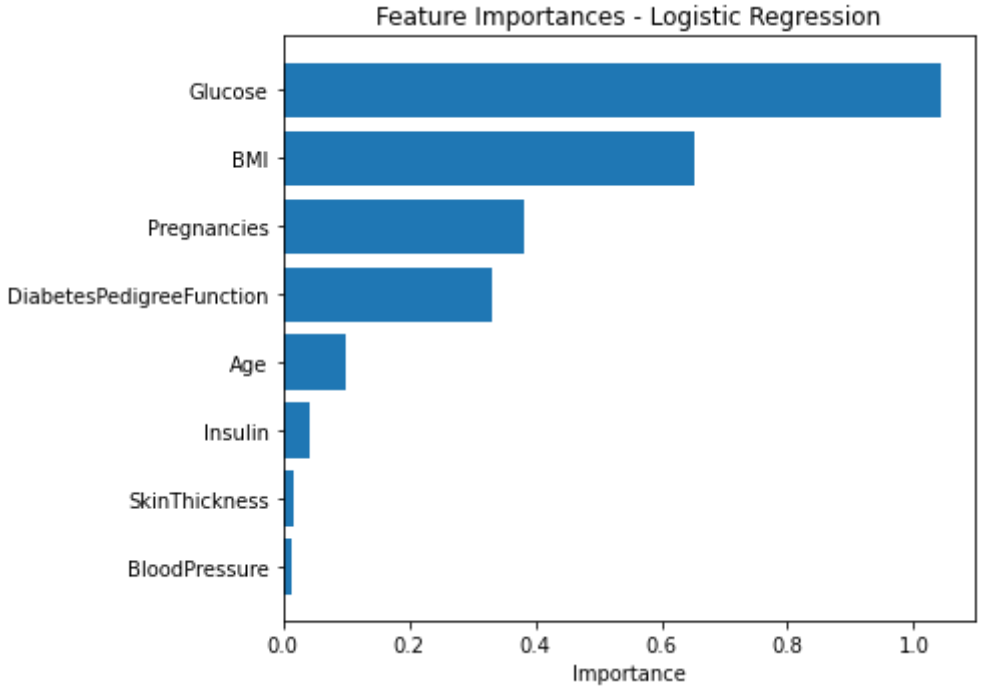
```python
# ROC-AUC Graph

score_lr = lr.predict_proba(X_test)[:,1]

false_positive_ratelr, true_positive_ratelr, thresholdlr = roc_curve(y_test, score_lr)

roc_auc = roc_auc_score(y_test, score_lr)
print('roc_auc_score for Logistic regression: {:.2%}'.format(roc_auc))

plt.subplots(1, figsize=(5,5))
plt.title('ROC Curve - Logistic Regression')
plt.plot(false_positive_ratelr, true_positive_ratelr)
plt.plot([0, 1], ls="--")
plt.plot([0, 0], [1, 0] , c=".7"), plt.plot([1, 1] , c=".7")
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.text(0.85, 0.1, 'AUC = {:.2%}'.format(roc_auc), bbox=dict(facecolor='white', alpha=0.5), ha='center')
plt.show()
```

```
roc_auc_score for Logistic regression: 84.79%
```



In [263…

```
# Feature importance Graph

importances = np.abs(lr.coef_)[0]

indices = np.argsort(importances)[::-1]

plt.figure(figsize=(7, 5))
plt.barh(range(X.shape[1]), importances[indices][::-1])
plt.yticks(range(X.shape[1]), X.columns[indices][::-1])
plt.title('Feature Importances - Logistic Regression')
plt.xlabel('Importance')
plt.tight_layout()
plt.show()
```



Naive Bayes

In [264…

```
random.seed(1234)
```

In [265…

```
parameters_gnb = {
    'var_smoothing': [1e-09, 1e-08, 1e-07, 1e-06]
}

cv = 5

gnb = GaussianNB()
nb = GridSearchCV(gnb,
                  param_grid = parameters_gnb,
```

```
                    scoring = 'accuracy',
                    cv = cv)

nb.fit(X_train , y_train)

print("Tuned Hyperparameters :", nb.best_params_)
print("Accuracy: {:.2%}".format(nb.best_score_))
```

```
Tuned Hyperparameters : {'var_smoothing': 1e-09}
Accuracy: 74.59%
```

In [266…

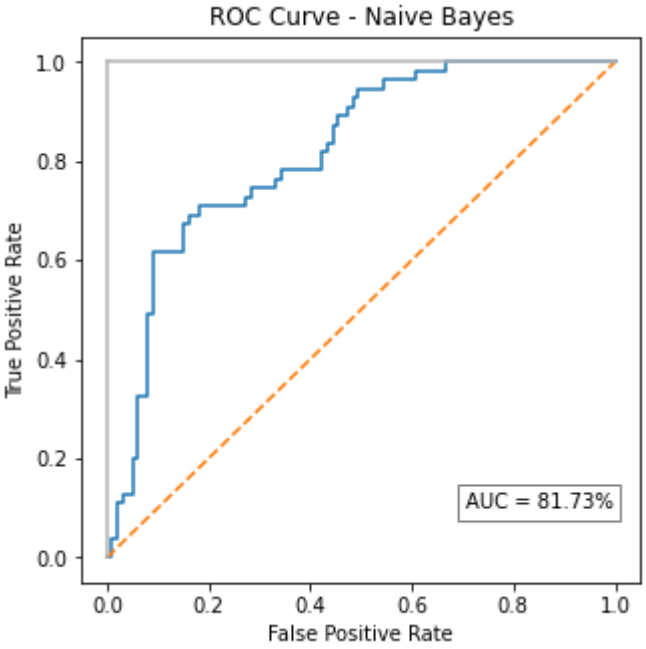
```
gnb = GaussianNB(**nb.best_params_)
gnb.fit(X_train , y_train)
predict_gnb = gnb.predict(X_test)

print("Accuracy: {:.2%}".format(accuracy_score(y_test, predict_gnb)))
print('Confusion Matrix:')
print(confusion_matrix(y_test, predict_gnb))
print('Classification Report Table:')
print(classification_report(y_test, predict_gnb))
```

```
Accuracy: 77.92%
Confusion Matrix:
[[86 13]
 [21 34]]
Classification Report Table:
              precision    recall  f1-score   support

           0       0.80      0.87      0.83        99
           1       0.72      0.62      0.67        55

    accuracy                           0.78       154
   macro avg       0.76      0.74      0.75       154
weighted avg       0.78      0.78      0.77       154
```

In [267…

```
# ROC-AUC Graph

score_gnb = gnb.predict_proba(X_test)[:,1]

false_positive_rategnb, true_positive_rategnb, thresholdgnb = roc_curve(y_test, score_gnb)

roc_auc = roc_auc_score(y_test, score_gnb)
print('roc_auc_score for Naive Bayes: {:.2%}'.format(roc_auc))

plt.subplots(1, figsize=(5,5))
plt.title('ROC Curve - Naive Bayes')
plt.plot(false_positive_rategnb, true_positive_rategnb)
plt.plot([0, 1], ls="--")
plt.plot([0, 0], [1, 0] , c=".7"), plt.plot([1, 1] , c=".7")
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.text(0.85, 0.1, 'AUC = {:.2%}'.format(roc_auc), bbox=dict(facecolor='white', alpha=0.5), ha='center')
plt.show()
```

```
roc_auc_score for Naive Bayes: 81.73%
```



Decision Tree

In [268…

```
random.seed(1234)
```

In [269…

```
parameters_dtree = {
    'criterion' : [ 'gini', 'entropy', 'log_loss' ],
    'splitter'  : ['best', 'random']
}

cv = 5 # so this will split the dataset into 5 kfold validation chunks for the GridSearchCv to run through

dtree = DecisionTreeClassifier()
dtc = GridSearchCV(dtree,
                   param_grid = parameters_dtree,
```

```
                    scoring = 'accuracy',
                    cv = cv)

dtc.fit(X_train , y_train)

print("Tuned Hyperparameters :", dtc.best_params_)
print("Accuracy: {:.2%}".format(dtc.best_score_))
```

```
Tuned Hyperparameters : {'criterion': 'gini', 'splitter': 'best'}
Accuracy: 71.82%
```

In [270…

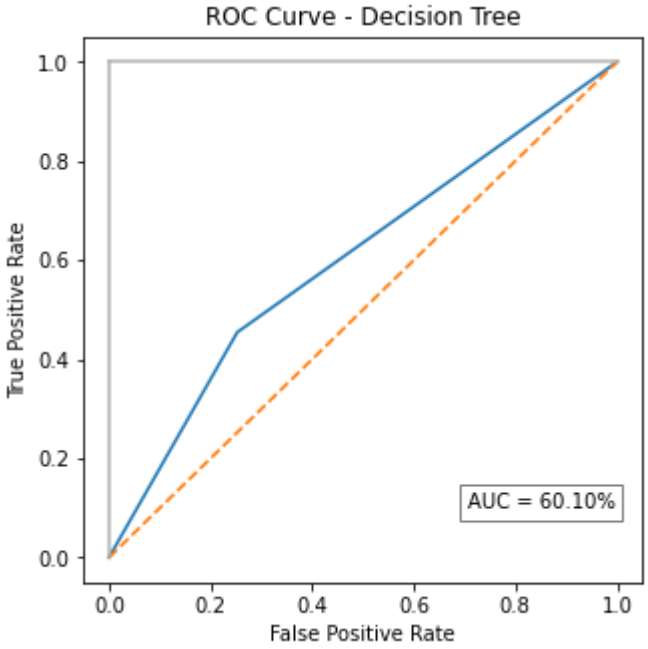
```
dtree = DecisionTreeClassifier(**dtc.best_params_)
dtree.fit(X_train , y_train)
predict_dtc = dtree.predict(X_test)

print("Accuracy: {:.2%}".format(accuracy_score(y_test, predict_dtc)))
print('Confusion Matrix:')
print(confusion_matrix(y_test, predict_dtc))
print('Classification Report Table:')
print(classification_report(y_test, predict_dtc))
```

```
Accuracy: 64.29%
Confusion Matrix:
[[74 25]
 [30 25]]
Classification Report Table:
              precision    recall  f1-score   support

           0       0.71      0.75      0.73        99
           1       0.50      0.45      0.48        55

    accuracy                           0.64       154
   macro avg       0.61      0.60      0.60       154
weighted avg       0.64      0.64      0.64       154
```

In [271…

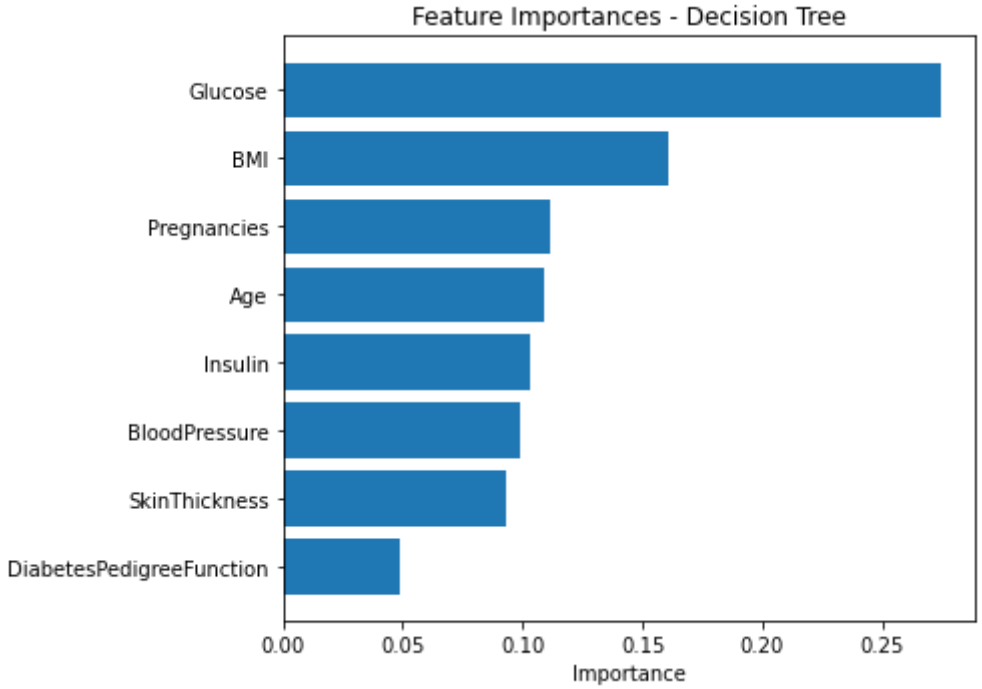
```
# ROC-AUC Graph

score_dtree = dtree.predict_proba(X_test)[:,1]

false_positive_ratedtree, true_positive_ratedtree, thresholddtree = roc_curve(y_test, score_dtree)

roc_auc = roc_auc_score(y_test, score_dtree)
print('roc_auc_score for Decision Tree: {:.2%}'.format(roc_auc))

plt.subplots(1, figsize=(5,5))
plt.title('ROC Curve - Decision Tree')
plt.plot(false_positive_ratedtree, true_positive_ratedtree)
plt.plot([0, 1], ls="--")
plt.plot([0, 0], [1, 0] , c=".7"), plt.plot([1, 1] , c=".7")
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.text(0.85, 0.1, 'AUC = {:.2%}'.format(roc_auc), bbox=dict(facecolor='white', alpha=0.5), ha='center')
plt.show()
```

```
roc_auc_score for Decision Tree: 60.10%
```



In [272…

```
# Feature importance Graph

importances = dtree.feature_importances_

indices = np.argsort(importances)

plt.figure(figsize=(7, 5))
plt.barh(range(X.shape[1]), importances[indices])
plt.yticks(range(X.shape[1]), X.columns[indices])
plt.title('Feature Importances - Decision Tree')
plt.xlabel('Importance')
plt.tight_layout()
plt.show()
```



K-Nearest Neighbor

In [273…

```
random.seed(1234)
```

In [274…

```
parameters_knn = {
    'n_neighbors' : [1,3,5,7,9,11,13,15,17,19,21,23,25,27],
    'weights'       : ['uniform' , 'distance'],
    'metric'  : ['euclidean' , 'manhattan' , 'minkowski']
}

cv = 5

knn = KNeighborsClassifier()
```

```
knneighbors = GridSearchCV(knn,
                   param_grid = parameters_knn,
                   scoring = 'accuracy',
                   cv = cv)

knneighbors.fit(X_train , y_train)

print("Tuned Hyperparameters :", knneighbors.best_params_)
print("Accuracy: {:.2%}".format(knneighbors.best_score_))
```

```
Tuned Hyperparameters : {'metric': 'euclidean', 'n_neighbors': 17, 'weights': 'distance'}
Accuracy: 77.69%
```

In [275…

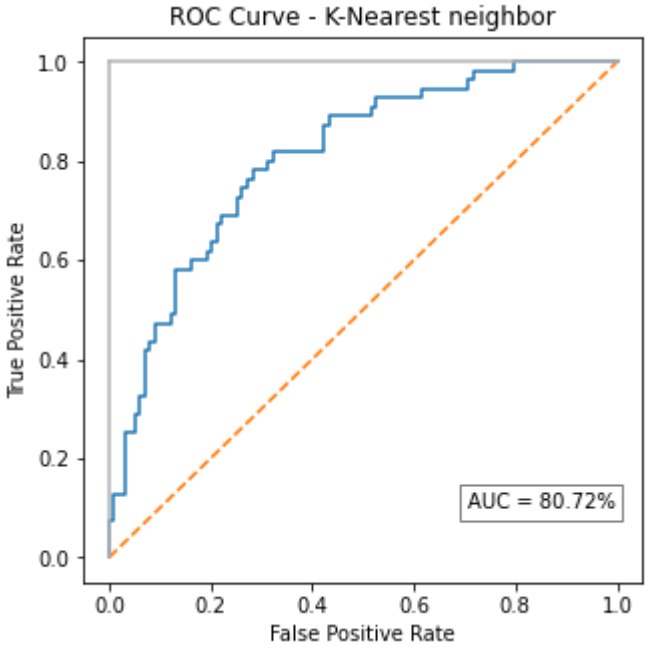
```
knn = KNeighborsClassifier(**knneighbors.best_params_)
knn.fit(X_train , y_train)
predict_knn = knn.predict(X_test)

print("Accuracy: {:.2%}".format(accuracy_score(y_test, predict_knn)))
print('Confusion Matrix:')
print(confusion_matrix(y_test, predict_knn))
print('Classification Report Table:')
print(classification_report(y_test, predict_knn))
```

```
Accuracy: 75.97%
Confusion Matrix:
[[86 13]
 [24 31]]
Classification Report Table:
              precision    recall  f1-score   support

           0       0.78      0.87      0.82        99
           1       0.70      0.56      0.63        55

    accuracy                           0.76       154
   macro avg       0.74      0.72      0.72       154
weighted avg       0.75      0.76      0.75       154
```

In [276…

```
# ROC-AUC Graph

score_knn = knn.predict_proba(X_test)[:,1]

false_positive_rateknn, true_positive_rateknn, thresholdknn = roc_curve(y_test, score_knn)

roc_auc = roc_auc_score(y_test, score_knn)
print('roc_auc_score for K-Nearest Neighbor: {:.2%}'.format(roc_auc))

plt.subplots(1, figsize=(5,5))
plt.title('ROC Curve - K-Nearest neighbor')
plt.plot(false_positive_rateknn, true_positive_rateknn)
plt.plot([0, 1], ls="--")
plt.plot([0, 0], [1, 0] , c=".7"), plt.plot([1, 1] , c=".7")
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.text(0.85, 0.1, 'AUC = {:.2%}'.format(roc_auc), bbox=dict(facecolor='white', alpha=0.5), ha='center')
plt.show()
```

```
roc_auc_score for K-Nearest Neighbor: 80.72%
```



Ensemble Methods

Random Forest Classifier

In [277…

```
random.seed(1234)
```

In [278…

```
parameters_rf = {
    'n_estimators' : [10,50,100,200,500],
    'criterion' : ['gini', 'entropy', 'log_loss'],
    'n_jobs' : [-1]
}

cv = 5
```

```
rf = RandomForestClassifier(random_state = 1234)
rforest = GridSearchCV(rf,
                   param_grid = parameters_rf,
                   scoring = 'accuracy',
                   cv = cv)

rforest.fit(X_train , y_train)

print("Tuned Hyperparameters :", rforest.best_params_)
print("Accuracy: {:.2%}".format(rforest.best_score_))
```

```
Tuned Hyperparameters : {'criterion': 'entropy', 'n_estimators': 50, 'n_jobs': -1}
Accuracy: 77.69%
```

In [279…

```
rf = RandomForestClassifier(**rforest.best_params_)
rf.fit(X_train , y_train)
predict_rf = rf.predict(X_test)

print("Accuracy: {:.2%}".format(accuracy_score(y_test, predict_rf)))
print('Confusion Matrix:')
print(confusion_matrix(y_test, predict_rf))
print('Classification Report Table:')
print(classification_report(y_test, predict_rf))
```

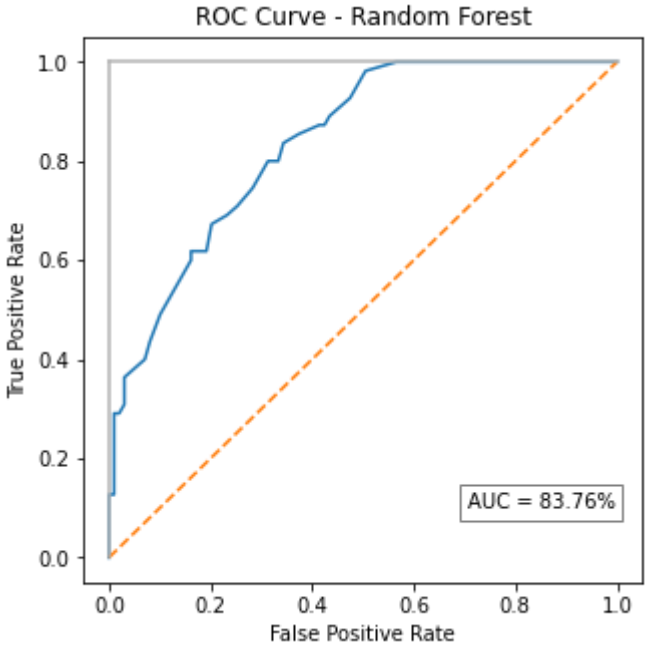
```
Accuracy: 75.32%
Confusion Matrix:
[[85 14]
 [24 31]]
Classification Report Table:
              precision    recall  f1-score   support

           0       0.78      0.86      0.82        99
           1       0.69      0.56      0.62        55

    accuracy                           0.75       154
   macro avg       0.73      0.71      0.72       154
```

```
weighted avg       0.75      0.75      0.75       154
```

In [280…

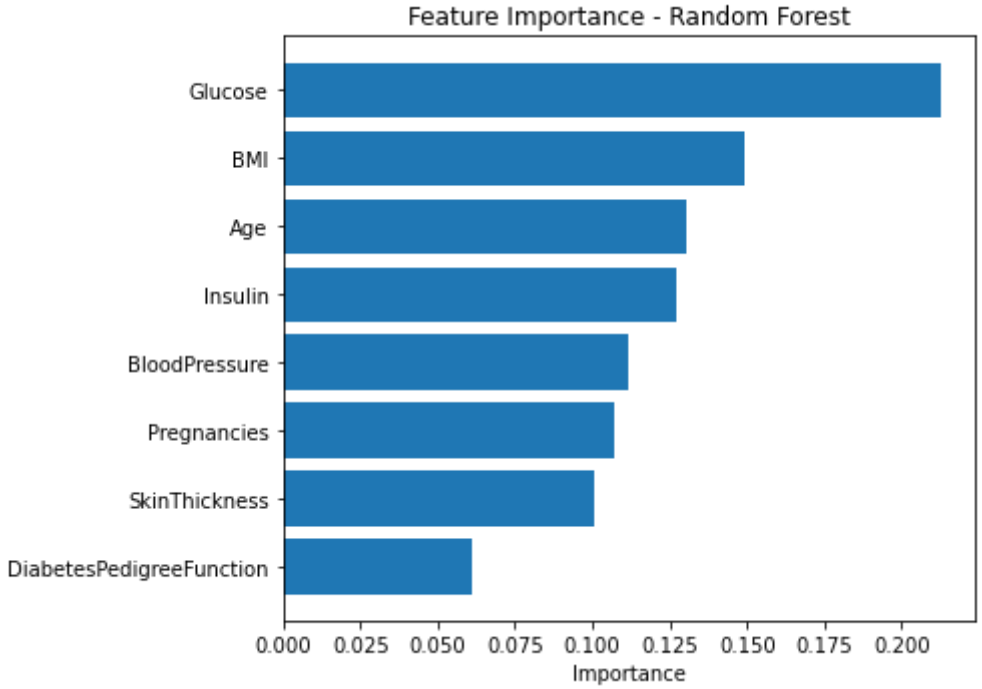
```
# ROC-AUC Graph

score_rf = rf.predict_proba(X_test)[:,1]

false_positive_raterf, true_positive_raterf, thresholdrf = roc_curve(y_test, score_rf)

roc_auc = roc_auc_score(y_test, score_rf)
print('roc_auc_score for Random Forest: {:.2%}'.format(roc_auc))

plt.subplots(1, figsize=(5,5))
plt.title('ROC Curve - Random Forest')
plt.plot(false_positive_raterf, true_positive_raterf)
plt.plot([0, 1], ls="--")
plt.plot([0, 0], [1, 0] , c=".7"), plt.plot([1, 1] , c=".7")
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.text(0.85, 0.1, 'AUC = {:.2%}'.format(roc_auc), bbox=dict(facecolor='white', alpha=0.5), ha='center')
plt.show()
```

```
roc_auc_score for Random Forest: 83.76%
```



In [281…

```
# Feature importance Graph

importances = rf.feature_importances_

indices = np.argsort(importances)

plt.figure(figsize=(7, 5))
plt.barh(range(X.shape[1]), importances[indices])
plt.yticks(range(X.shape[1]), X.columns[indices])
plt.title('Feature Importance - Random Forest')
plt.xlabel('Importance')
plt.tight_layout()
plt.show()
```



Majority Vote Classifier

In [282…

```
random.seed(1234)
```

In [283…

```
# I'm going to use the same hyper-parameters above in my models down below

lr = LogisticRegression(**clf.best_params_)
gnb = GaussianNB(**nb.best_params_)
knn = KNeighborsClassifier(**knneighbors.best_params_)

ensemble = VotingClassifier(estimators=[('lr', lr) , ('gnb', gnb) , ('knn', knn)],
                            voting='soft')
```

```
ensemble.fit(X_train, y_train)
predict_ens_testing = ensemble.predict(X_test)

print("Test Set Accuracy: {:.2%}".format(accuracy_score(y_test, predict_ens_testing)))
print('Test Set Confusion Matrix:')
print(confusion_matrix(y_test, predict_ens_testing))
print('Test Set Classification Report Table:')
print(classification_report(y_test, predict_ens_testing))
```

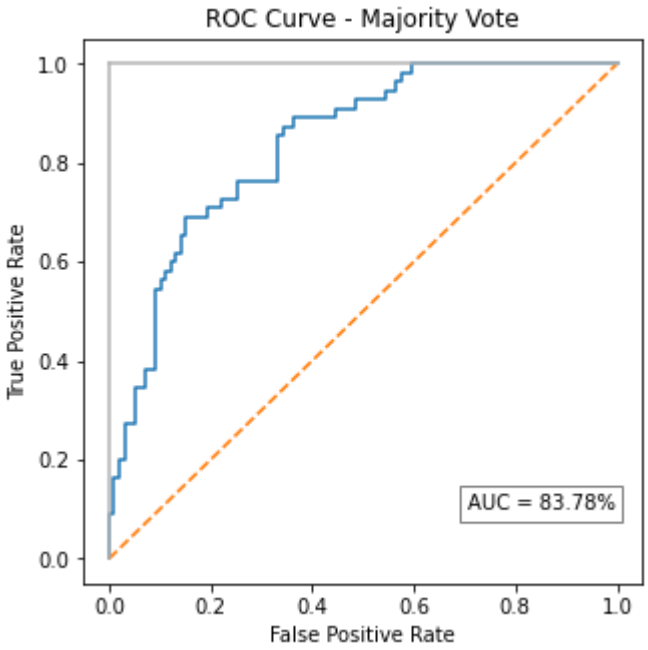
```
Test Set Accuracy: 77.92%
Test Set Confusion Matrix:
[[89 10]
 [24 31]]
Test Set Classification Report Table:
              precision    recall  f1-score   support

           0       0.79      0.90      0.84        99
           1       0.76      0.56      0.65        55

    accuracy                           0.78       154
   macro avg       0.77      0.73      0.74       154
weighted avg       0.78      0.78      0.77       154
```

In [284…

```
# ROC-AUC Graph

score_mv = ensemble.predict_proba(X_test)[:,1]

false_positive_ratemv, true_positive_ratemv, thresholdmv = roc_curve(y_test, score_mv)

roc_auc = roc_auc_score(y_test, score_mv)
print('roc_auc_score for Majority Vote: {:.2%}'.format(roc_auc))

plt.subplots(1, figsize=(5,5))
plt.title('ROC Curve - Majority Vote')
plt.plot(false_positive_ratemv, true_positive_ratemv)
plt.plot([0, 1], ls="--")
```

```
plt.plot([0, 0], [1, 0] , c=".7"), plt.plot([1, 1] , c=".7")
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.text(0.85, 0.1, 'AUC = {:.2%}'.format(roc_auc), bbox=dict(facecolor='white', alpha=0.5), ha='center')
plt.show()
```

```
roc_auc_score for Majority Vote: 83.78%
```



Gradient Boosting Classifier

In [285…

```
random.seed(1234)
```

In [286…

```
parameters_gb = {
    'n_estimators' : [10,50,100,200,500],
    'learning_rate' :[0.001 , 0.01 , 0.1 , 1],
```

```
    'loss' : ['log_loss', 'deviance', 'exponential']
}

cv = 5

gbc = GradientBoostingClassifier()
gradboost = GridSearchCV(gbc,
                   param_grid = parameters_gb,
                   scoring = 'accuracy',
                   cv = cv)

gradboost.fit(X_train , y_train)

print("Tuned Hyperparameters :", gradboost.best_params_)
print("Accuracy: {:.2%}".format(gradboost.best_score_))
```

```
Tuned Hyperparameters : {'learning_rate': 0.01, 'loss': 'exponential', 'n_estimators': 500}
Accuracy: 76.87%
```

In [287…

```
gbc = GradientBoostingClassifier(**gradboost.best_params_)
gbc.fit(X_train , y_train)
predict_gbc = gbc.predict(X_test)

print("Accuracy: {:.2%}".format(accuracy_score(y_test, predict_gbc)))
print('Confusion Matrix:')
print(confusion_matrix(y_test, predict_gbc))
print('Classification Report Table:')
print(classification_report(y_test, predict_gbc))
```

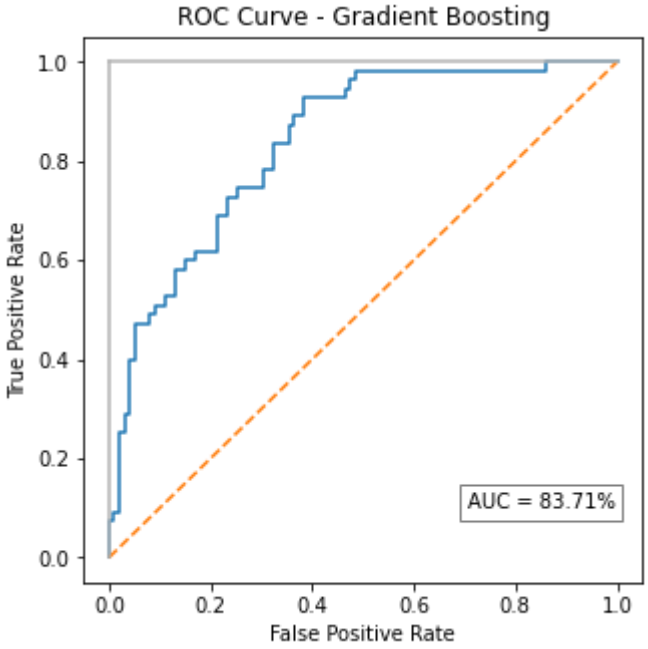
```
Accuracy: 75.32%
Confusion Matrix:
[[86 13]
 [25 30]]
Classification Report Table:
              precision    recall  f1-score   support

           0       0.77      0.87      0.82        99
           1       0.70      0.55      0.61        55
```

```
    accuracy                           0.75       154
   macro avg       0.74      0.71      0.72       154
weighted avg       0.75      0.75      0.75       154
```

In [288…

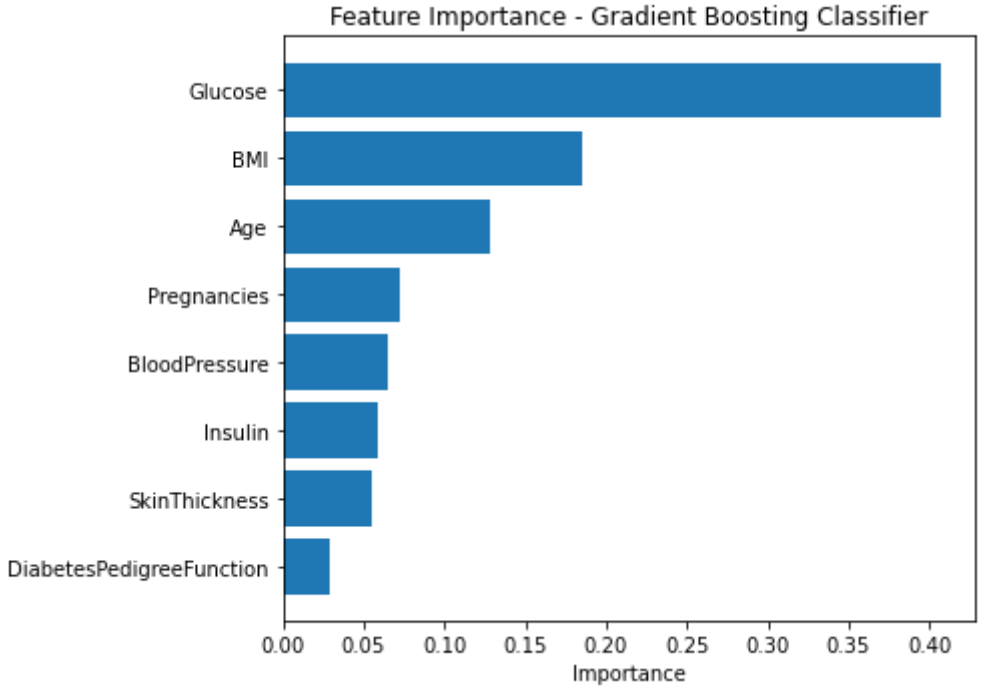
```
# ROC-AUC Graph

score_gbc = gbc.predict_proba(X_test)[:,1]

false_positive_rategbc, true_positive_rategbc, thresholdgbc = roc_curve(y_test, score_gbc)

roc_auc = roc_auc_score(y_test, score_gbc)
print('roc_auc_score for Gradient Boosting: {:.2%}'.format(roc_auc))

plt.subplots(1, figsize=(5,5))
plt.title('ROC Curve - Gradient Boosting')
plt.plot(false_positive_rategbc, true_positive_rategbc)
plt.plot([0, 1], ls="--")
plt.plot([0, 0], [1, 0] , c=".7"), plt.plot([1, 1] , c=".7")
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.text(0.85, 0.1, 'AUC = {:.2%}'.format(roc_auc), bbox=dict(facecolor='white', alpha=0.5), ha='center')
plt.show()
```

```
roc_auc_score for Gradient Boosting: 83.71%
```



In [289…

```
# Feature importance Graph

importances = gbc.feature_importances_
indices = np.argsort(importances)

plt.figure(figsize=(7, 5))
plt.barh(range(X.shape[1]), importances[indices])
plt.yticks(range(X.shape[1]), X.columns[indices])
plt.title('Feature Importance - Gradient Boosting Classifier')
plt.xlabel('Importance')
plt.tight_layout()
plt.show()
```



Neural Network

In [105…

```
random.seed(1234)
```

In [106…

```
# Define the function to create a Keras model
def create_model(activation='relu', hidden_layers=1, neurons=32, optimizer='adam'):
    model = Sequential()
    model.add(Dense(neurons, input_dim=8, activation=activation))
    for i in range(hidden_layers-1):
        model.add(Dense(neurons, activation=activation))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
```

```
    return model

# Create a KerasClassifier object for GridSearchCV
model = KerasClassifier(build_fn=create_model, verbose=0)

# Define the hyperparameters to tune
param_grid = {
    'batch_size': [16, 32, 64],
    'epochs': [50, 100, 150],
    'activation': ['relu', 'tanh' , 'sigmoid'],
    'hidden_layers': [1, 2, 3, 4],
    'neurons': [16, 32, 64],
    'optimizer': ['adam', 'rmsprop']
}

# Create a GridSearchCV object to search for the best hyperparameters
grid = GridSearchCV(estimator=model, param_grid=param_grid, cv=3, n_jobs=-1, scoring='accuracy')

# Fit the GridSearchCV object to the training data
grid_result = grid.fit(X_train , y_train)

# Print the best hyperparameters and the best accuracy score
print("Best Hyperparameters:", grid_result.best_params_)
print("Best Accuracy Score: {:.2%}".format(grid_result.best_score_))
```

```
Best Hyperparameters: {'activation': 'relu', 'batch_size': 16, 'epochs': 50, 'hidden_layers': 1, 'neurons': 64, 'optimize
r': 'rmsprop'}
Best Accuracy Score: 79.97%
```

In [107…

```
# Create a new Keras model with the best hyperparameters
best_params = grid_result.best_params_

model = create_model(
    activation=best_params['activation'],
    hidden_layers=best_params['hidden_layers'],
    neurons=best_params['neurons'],
    optimizer=best_params['optimizer']
)
```

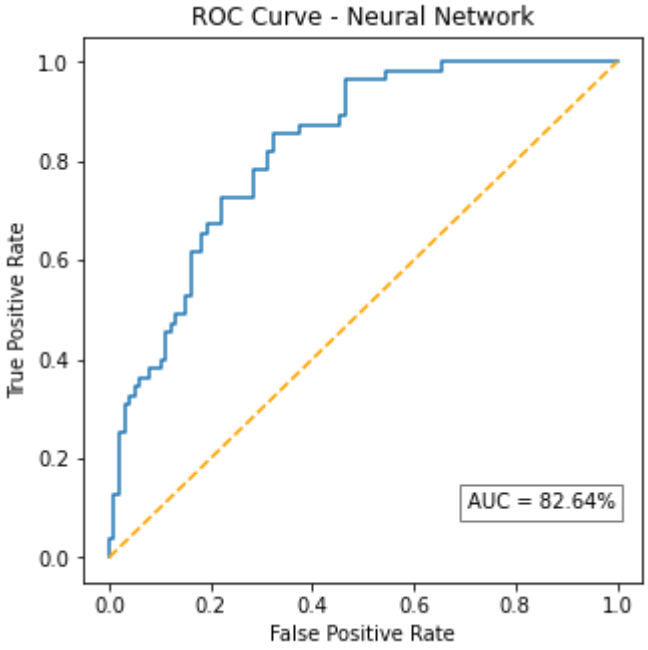
```
# Train the model with the best hyperparameters
model.fit(X_train , y_train, batch_size=best_params['batch_size'], epochs=best_params['epochs'], verbose=0)

# Evaluate the model on the test data
score = model.evaluate(X_test, y_test, verbose=0)
print("Test Loss: {:.4f}".format(score[0]))
print("Test Accuracy: {:.2%}".format(score[1]))
```

```
Test Loss: 0.4930
Test Accuracy: 72.73%
```

In [108…

```
# ROC-AUC Graph

y_pred_proba = model.predict(X_test)

fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
roc_auc = roc_auc_score(y_test, y_pred_proba)

plt.subplots(1, figsize=(5,5))
plt.title('ROC Curve - Neural Network')
plt.plot(fpr, tpr)
plt.plot([0, 1], [0, 1], linestyle='--', color='orange')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.text(0.85, 0.1, 'AUC = {:.2%}'.format(roc_auc), bbox=dict(facecolor='white', alpha=0.5), ha='center')
plt.show()
```

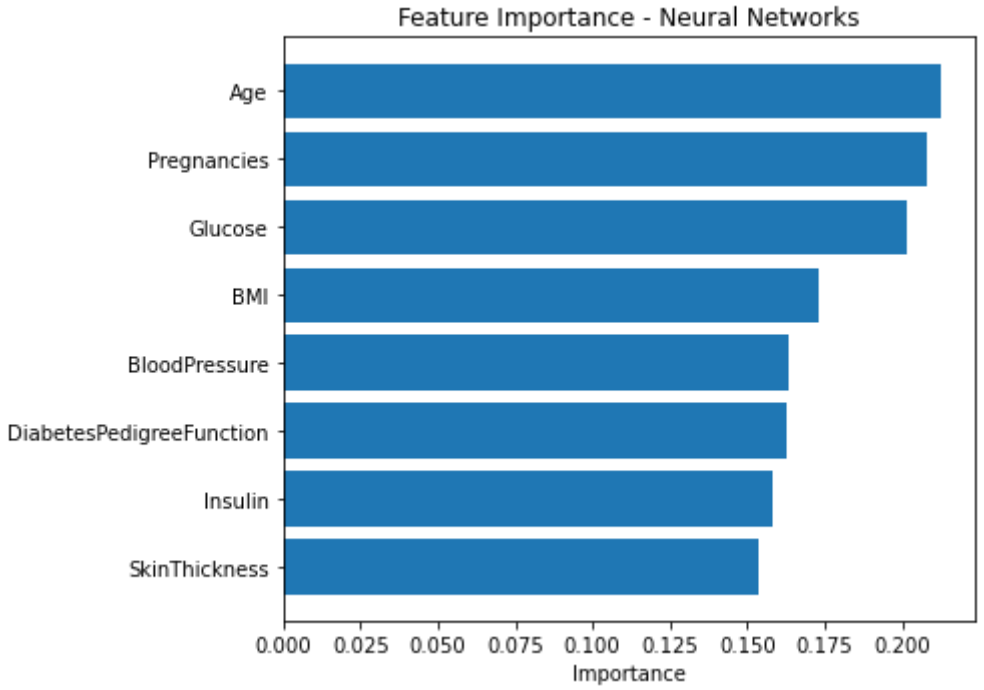
```
5/5 [==============================] - 0s 3ms/step
```



In [109…

```
X_train_df = pd.DataFrame(X_train, columns=X.columns)

# Get the weights from the first layer of the model
weights = model.layers[0].get_weights()[0]

# Calculate the mean absolute weight for each input feature
feature_importance = np.mean(np.abs(weights), axis=1)

# Sort the feature importance values in ascending order
sorted_idx = np.argsort(feature_importance)

# Plot the feature importances
plt.figure(figsize=(7, 5))
plt.barh(range(X_train_df.shape[1]), feature_importance[sorted_idx])
plt.yticks(range(X_train_df.shape[1]), X_train_df.columns[sorted_idx])
plt.xlabel('Importance')
```

```
plt.title('Feature Importance - Neural Networks')
plt.tight_layout()
plt.show()
```



In [110…

```
topology = model.to_json()
print(json.dumps(json.loads(topology), indent=2))
```

```
{
  "class_name": "Sequential",
  "config": {
    "name": "sequential_13",
    "layers": [
      {
        "class_name": "InputLayer",
```

```
  "config": {
    "batch_input_shape": [
      null,
      8
    ],
    "dtype": "float32",
    "sparse": false,
    "ragged": false,
    "name": "dense_32_input"
  }
},
{
  "class_name": "Dense",
  "config": {
    "name": "dense_32",
    "trainable": true,
    "dtype": "float32",
    "batch_input_shape": [
      null,
      8
    ],
    "units": 64,
    "activation": "relu",
    "use_bias": true,
    "kernel_initializer": {
      "class_name": "GlorotUniform",
      "config": {
        "seed": null
      }
    },
    "bias_initializer": {
      "class_name": "Zeros",
      "config": {}
    },
    "kernel_regularizer": null,
    "bias_regularizer": null,
    "activity_regularizer": null,
    "kernel_constraint": null,
```

```
          "bias_constraint": null
        }
      },
      {
        "class_name": "Dense",
        "config": {
          "name": "dense_33",
          "trainable": true,
          "dtype": "float32",
          "units": 1,
          "activation": "sigmoid",
          "use_bias": true,
          "kernel_initializer": {
            "class_name": "GlorotUniform",
            "config": {
              "seed": null
            }
          },
          "bias_initializer": {
            "class_name": "Zeros",
            "config": {}
          },
          "kernel_regularizer": null,
          "bias_regularizer": null,
          "activity_regularizer": null,
          "kernel_constraint": null,
          "bias_constraint": null
        }
      }
    ]
  },
  "keras_version": "2.11.0",
  "backend": "tensorflow"
}
```


In [111…

```
def create_ann_viz_model(best_params, X):
    model = create_model(
```

```
        activation=best_params['activation'],
        hidden_layers=best_params['hidden_layers'],
        neurons=best_params['neurons'],
        optimizer=best_params['optimizer']
    )

    # Train the model with the best hyperparameters
    model.fit(X, y, batch_size=best_params['batch_size'], epochs=best_params['epochs'], verbose=0)

    # Create a new ann_viz model with the updated weights
    ann_viz_model = ann_viz(model, title="Neural Network Visualization")

    return ann_viz_model

ann_viz_model = create_ann_viz_model(best_params, X)
```

In [ ]: