

Configuration Manual

MSc Research Project
Data Analytics

Girish Jagwani
Student ID: x18136371

School of Computing
National College of Ireland

Supervisor: Dr. Catherine Mulwa

National College of Ireland
MSc Project Submission Sheet
School of Computing



Student Name: Girish Jagwani

Student ID: x18136371

Programme: MSc. Data Analytics

Year: 2019-20

Module: MSc. Research Project

Lecturer: Dr. Catherine Mulwa

Submission

Due Date: 13-12-2019

Project Title: Identifying the Patients at Risk of Stroke Using Anomaly Detection Based Classification Approach

Word Count: **Page Count:**

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature

Date:

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	<input type="checkbox"/>
Attach a Moodle submission receipt of the online project submission , to each project (including multiple copies).	<input type="checkbox"/>
You must ensure that you retain a HARD COPY of the project , both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	<input type="checkbox"/>

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Girish Jagwani
Student ID: x18136371

1 Introduction

The aim of this document is to provide a walkthrough and thus enable the user to setup this ICT solution on any suitable machine and produce the desired outcomes. This document, therefore, includes the entire process of environment setup along with the required hardware and software specifications. It also includes the snapshots of code to guide the user throughout implementation and the visualisations of the exploratory data analysis that are not added as part of the technical report.

The rest of this report is structured as follows: Chapter 2 discusses the Environment Configurations, Chapter 3 discusses the Implementation, Chapter 4 illustrates the sample outputs of the implementation of this ICT solution and Chapter 5 is the Appendix for providing a walkthrough to install the necessary software.

2 Environment Configurations

This chapter mainly discusses the overall environment configurations that were used while implementing this ICT solution. This includes Hardware Configurations, Software Configurations and Python packages and libraries used.

2.1 Hardware Configurations

This section discusses the specifications of the hardware, i.e., the machine used for the implementation of this ICT solution. In this case, as displayed in Figure 1, a laptop with 64-bit Microsoft Windows 10 operating system, 1.80 GHz processor and 8 GB Ram was used.

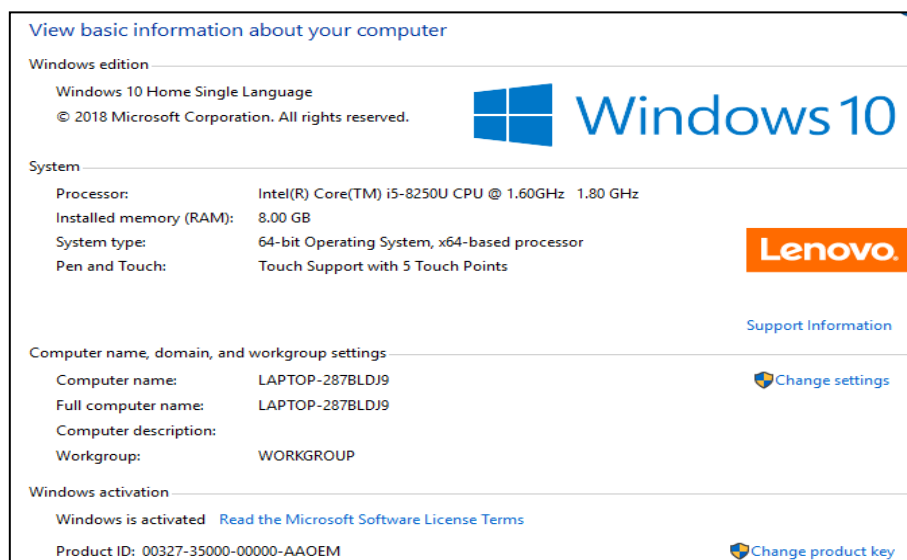


Figure 1 : Hardware Configuration

2.2 Software Configurations

This section discusses the specifications of the software that were used as part of the implementation of this ICT solution. The key software that were used are:

- Anaconda
- Spyder IDE

2.2.1 Anaconda

Anaconda is an open-source platform for data science with Python and R. Python was used as part of the implementation of this ICT solution with Anaconda. Figure 2 enlightens the specifications of the Anaconda used to implement this ICT Solution.

```
(base) PS C:\Users\girish> conda info

active environment : base
active env location : C:\ProgramData\Anaconda3
shell level        : 1
user config file   : C:\Users\girish\.condarc
populated config files : C:\Users\girish\.condarc
conda version      : 4.7.12
conda-build version : 3.18.9
python version     : 3.7.4.final.0
virtual packages   : __cuda=10.0
```

Figure 2 : Anaconda Specifications

2.2.2 Spyder

Spyder is an Integrated Development Environment (IDE) that was used to write the implementation scripts in Python. Spyder IDE was launched using the Anaconda Navigator that is available after the installation of Anaconda (Section 2.2.1). Figure 3 shows the launch card for Spyder IDE version 3.3.6 in Anaconda Navigator.

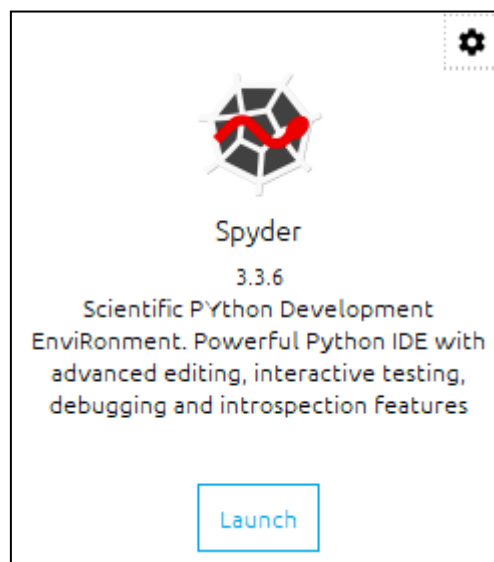


Figure 3 : Spyder IDE

2.3 Python Packages Used

The packages and libraries that are used as part of the implementation of this ICT solution are mentioned in Table 1 below.

Table 1 : Python Packages and Libraries Used for Stroke Detection

impyute.imputation.cs	sklearn.naive_bayes	xgboost
sklearn.model_selection	sklearn.svm	imblearn.over_sampling
sklearn.preprocessing	sklearn.neighbors	imblearn.under_sampling
sklearn.utils.class_weight	sklearn.tree	imblearn.combine
sklearn.metrics	matplotlib.pyplot	keras.models
sklearn.ensemble	seaborn	keras.layers
sklearn.linear_model	math	keras.wrappers.scikit_learn

3 Implementation

This section provides a walkthrough of the Python script that was written as part of the implementation of this ICT Solution.

3.1 Reading the Dataset

After loading the packages (Section 2.3), the first important step is to import the dataset which is a .csv file into the Python environment. This was done using read_csv() function of pandas. The df data frame was created to store the data obtained. Figure 4 shows the code for the same.

The set_option() function was used to set appropriate column width and make all columns visible while viewing the outputs.

```
#-----Reading CSV-----#
#-----#
#-----#
#Reading the dataset
df = pd.read_csv('C:/Users/girish/Downloads/Studies/NCI/3. Semester 3/5. Python/1. Dataset/Dataset.csv')
#To view all columns of data
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 1000)
```

Figure 4 : Reading CSV for Stroke Detection in Python

3.2 Exploratory Data Analysis

This step includes the activities that help to understand the data in a better way. Those are as follows:

1. Identifying the Null Values in the Dataset

After storing the dataset into a pandas dataframe df, the next step that was to identify the null values in the dataset. This was done using the .isnull() function of pandas that return True for null values and False otherwise. A “for” loop was then used to check the columns with True values present and print the respective columns with the count of null values. Figure 5 shows the code for the same and Figure 6 shows the output received.

```
#-----Null Values-----#
#-----#
#-----#
missing_data = df.isnull()
for i in missing_data.columns.values.tolist():
    if True in missing_data[i].value_counts().index.tolist():
        print('*****',i,'*****')
        print(missing_data[i].value_counts())
        print("*****")
        print()
```

Figure 5 : Identifying Null values for Stroke Detection

```

***** bmi *****
False    41938
True      1462
Name: bmi, dtype: int64
*****

***** smoking_status *****
False    30108
True     13292
Name: smoking_status, dtype: int64
*****

```

Figure 6 : Output of Null Values Identified

2. Exploring the Distribution of Categorical Variables

The data distribution of the Categorical variables was identified using the `.value_counts()` function of pandas. A for loop was used with all categorical columns passed as parameters. Figure 7 shows the code for the same and Figure 8 shows the sample of the output received.

```

#-----Data Distribution-----#
#
#
for i in df.columns[[1,3,4,5,6,7,10,11]]:
    print('*****',i,'*****')
    print(df[i].value_counts())
    print('*'*30)
    print()

```

Figure 7 : Exploring Data Distribution for Stroke Detection in Python

```

***** smoking_status *****
never smoked    16053
formerly smoked  7493
smokes          6562
Name: smoking_status, dtype: int64
*****

***** stroke *****
0    42617
1     783
Name: stroke, dtype: int64
*****

```

Figure 8 : Output of Data Distribution

3. Exploring using Data Visualisations

After getting an overview of the data, Data Visualisations were created for better interpretation. pandas function `.value_count()` was used to first store the data distribution into respective variables (Figure 9). These variables were then used for plotting purposes.

```

df_hypertension = df["hypertension"].value_counts()
df_heartdisease = df["heart_disease"].value_counts()
df_stroke = df["stroke"].value_counts()
df_gender = df["gender"].value_counts()
df_evermarried = df["ever_married"].value_counts()
df_worktype = df["work_type"].value_counts()
df_residencetype = df["Residence_type"].value_counts()
df_smokingstatus = df["smoking_status"].value_counts()

```

Figure 9: Storing Data Distribution in Variables

Figure 10 captures the code for some of the visualisations that were created to evaluate the data balance and distribution.

```

#Stroke Barplot
plt.figure(figsize=(8,6))
ax = sns.barplot(x=df_stroke.index, y=df_stroke)
ax.set_title("Stroke")
ax.set(xlabel="", ylabel='Number of Records')

#BMI Distribution Plot
plt.figure(figsize=(8,6))
ax = sns.distplot(df["bmi"].dropna(), color="crimson")
ax.set_title("Body Mass Index Distribution")
ax.set(xlabel="", ylabel="")

#Average Glucose Level Plot
plt.figure(figsize=(8,6))
ax = sns.distplot(df["avg_glucose_level"], color="darkmagenta")
ax.set_title("Average Glucose Level Distribution")
ax.set(xlabel="", ylabel="")

#Gender Barplot
plt.figure(figsize=(8,6))
ax = sns.barplot(x=df_gender.index, y=df_gender)
ax.set_title("Gender")
ax.set(xlabel="", ylabel="Number of Records")

```

Figure 10 : Plotting Data Distribution

The respective Output is shown in Figure 11 to Figure 14.

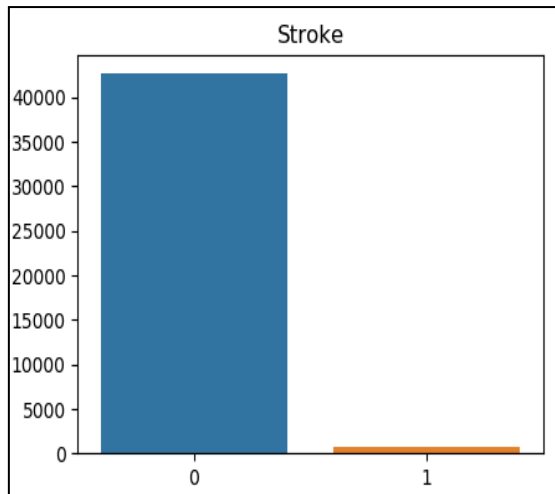


Figure 11 : Bar Plot for Stroke

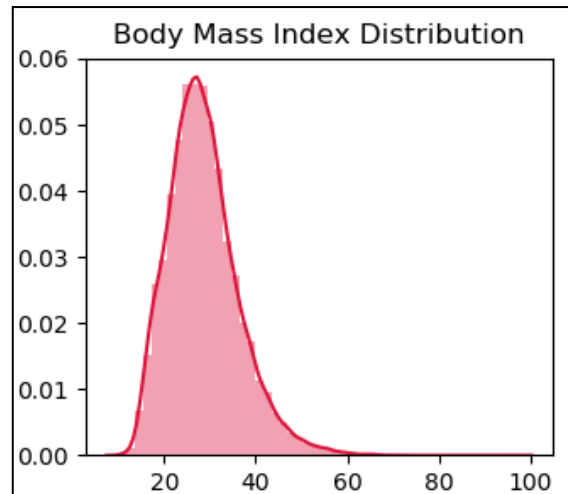


Figure 12 : Distribution Plot for BMI

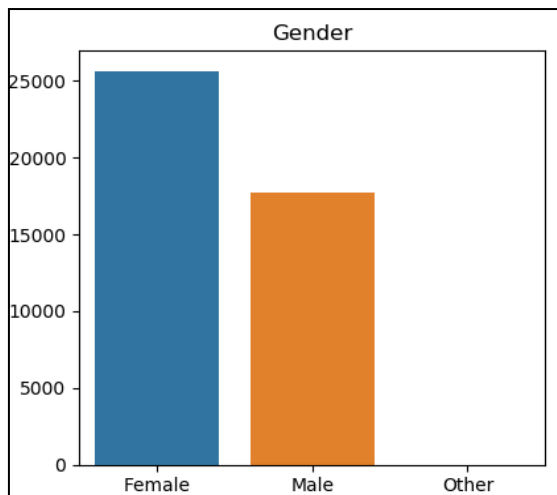


Figure 13 : Bar Plot for Gender

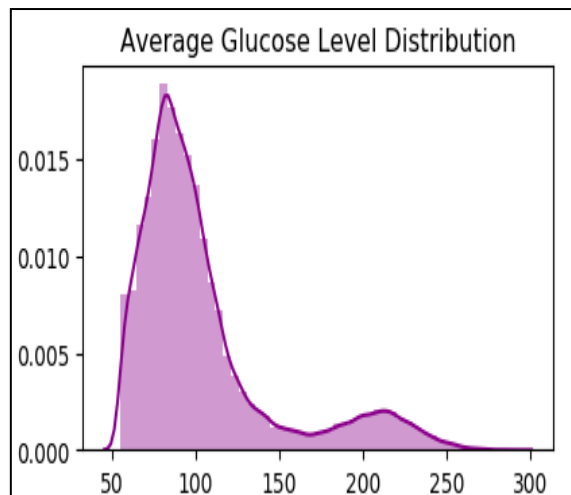


Figure 14 : Distribution Plot for Average Glucose Level

Similarly, Figure 15 captures the code for some of the visualisations that were created to evaluate the relationship between different variables.

```
#Age vs Stroke Distance Plote
plt.figure(figsize=(8,6))
sns.distplot(df.loc[df['stroke'] == 0]['age'],norm_hist=True, color="green", bins=15).set_title("Age vs Stroke")
sns.distplot(df.loc[df['stroke'] == 1]['age'],norm_hist=True, color="red", bins=15)

#BMI vs Stroke
plt.figure(figsize=(8,6))
sns.distplot(df.loc[df['stroke'] == 0]['bmi'], norm_hist=True, color="green", bins=15).set_title("Body Mass Index vs Stroke")
sns.distplot(df.loc[df['stroke'] == 1]['bmi'], norm_hist=True, color="red", bins=15)

#Glucose Level vs Stroke
plt.figure(figsize=(8,6))
sns.distplot(df.loc[df['stroke'] == 0]['avg_glucose_level'], norm_hist=True, color="green", bins=15).set_title("Average Glucos")
sns.distplot(df.loc[df['stroke'] == 1]['avg_glucose_level'], norm_hist=True, color="red", bins=15)

#**Gender vs Age vs Stroke violin plot
ax = sns.violinplot(x = 'stroke', y = 'age', hue = 'gender', data=df, palette= "Set1", xlabel="x")
ax.set_title("Gender vs Age vs Stroke")
ax.set(xlabel="Stroke", ylabel="Age")
```

Figure 15 : Plotting Relationship Charts

The respective Output is shown in Figure 16 to Figure 19.

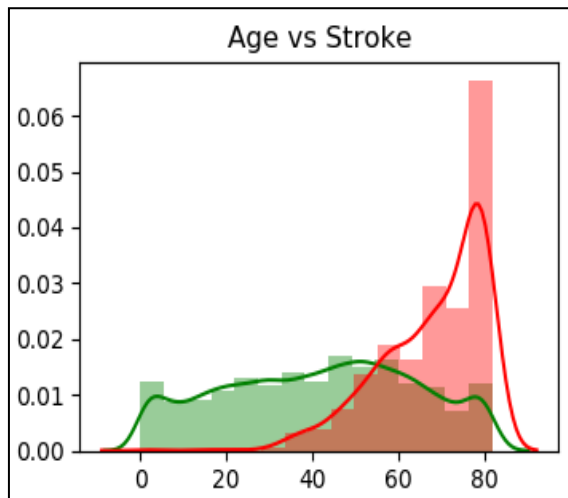


Figure 16 : Age vs Stroke Distribution

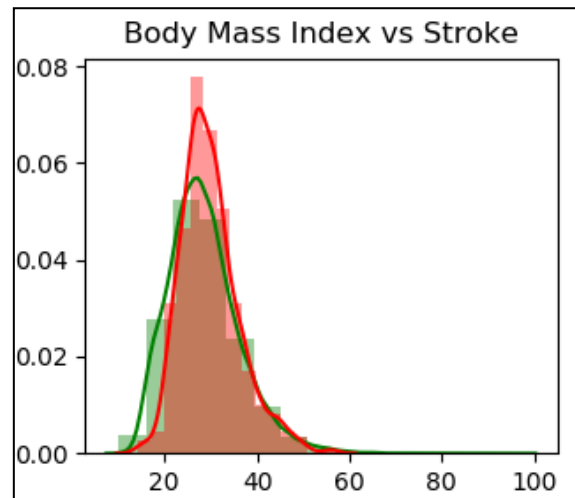


Figure 17 : BMI vs Stroke Distribution

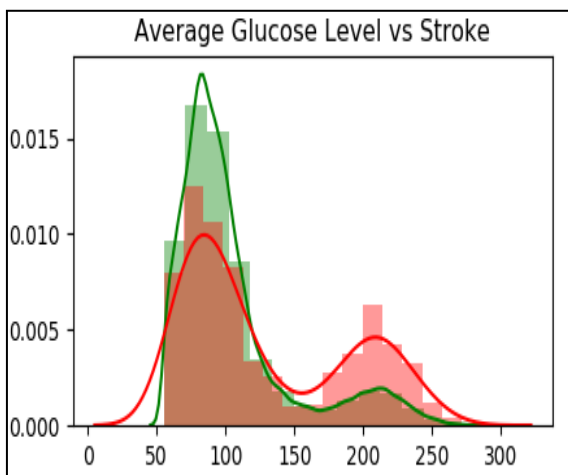


Figure 18 : Average Glucose Level vs Stroke Distribution

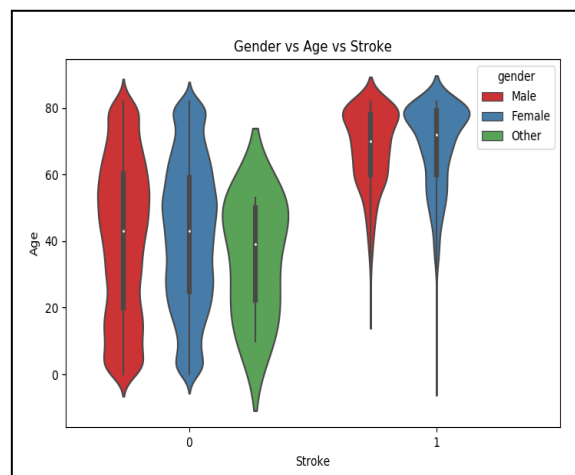


Figure 19 : Gender vs Age vs Stroke Violin Plot

3.3 Data Pre-processing and Feature Selection

This step helps to make data suitable for the processing by the machine learning models. The activities done as part of this step include:

1. Processing Null Values

The Null data identified as part of exploratory data analysis, cannot be used for the implementation of the machine learning models. This data was therefore imputed. Figure 20 shows the code for the imputation of the smoking_status. All the null values for smoking_status with age below 18 years were tagged as never smoked (as the minimum age of smoking is 18 years). Whereas for the rest of the cases, it was tagged unknown.

```
#-----Data Pre-processing-----#
#-----#
#-----#
#-----#
#-----smoking_status-----#
#-----#
#Smoking_status: filling the missing values
df.loc[(df['age']<18) & (df['smoking_status'].isna()), 'smoking_status'] = 'never smoked' #as the age is below 18
df.loc[(df['age']>=18) & (df['smoking_status'].isna()), 'smoking_status'] = 'Unknown'

#Smoking_status data distribution
print('Null Values:', df.smoking_status.isnull().sum())
print(df.groupby('smoking_status').size())
```

Figure 20 : Imputing Null Values for smoking_status

Figure 21 shows the code for the imputation of the BMI using Multiple Imputation by Chained Equation (MICE) technique with the help of mice() function of impute.imputation .cs package.

```
#-----IMPUTATION USING MICE-----#
#-----#
#-----#
#primary dataframe after encoding
df1 = df[['age', 'avg_glucose_level', 'bmi', 'gender_code', 'hypertension_code', 'heartdisease_code', 'evermarried_code', 'worktype_code']]

#Imputing BMI using Mice
imputed_training=mice(df1.values)

#Replacing BMI of df1 with BMI imputed using Mice
df1['bmi']=imputed_training[:,2]

#Verifying Null values after Imputation
print('Null Values:', df1.bmi.isnull().sum())
```

Figure 21 : Imputing Null values for BMI

2. Data Encoding

The Categorical variables are meant to be encoded as integers for the machine learning model to understand, this was achieved using LabelEncoder() method of sklearn. Pre-processing package. Figure 22 shows the code for the encoding of the categorical variables.

```
#-----Data Encoding-----#
#-----#
#-----#
lbl_enc = LabelEncoder() #from sklearn.preprocessing

#Creating Encoders
df["gender_code"] = lbl_enc.fit_transform(df["gender"])
df["hypertension_code"] = lbl_enc.fit_transform(df["hypertension"])
df["heartdisease_code"] = lbl_enc.fit_transform(df["heart_disease"])
df["evermarried_code"] = lbl_enc.fit_transform(df["ever_married"])
df["worktype_code"] = lbl_enc.fit_transform(df["work_type"])
df["residencetype_code"] = lbl_enc.fit_transform(df["Residence_type"])
df["smokingstatus_code"] = lbl_enc.fit_transform(df["smoking_status"])
```

Figure 22 : Encoding Categorical Variables

3. Feature Selection

Feature Selection is necessary to avoid the model fitting issues. As part of feature selection, the redundant un-encoded classification features were removed along with the id column.

Correlation among the features was then evaluated using corr() function, followed by the heatmap() function of seaborn for its plotting. This was to assure the absence of multicollinearity (at a threshold of 0.7) as it is one of the important assumptions for certain machine learning models like Logistic Regression. Figure 23 shows the code for the same.

```
#-----Feature Selection-----#
#-----#
#-----#
#-----#
#-----Separating the Dependent and Independent Variables-----#
#-----#
#Independent Columns
x = df1[['age', 'avg_glucose_level', 'bmi', 'gender_code', 'hypertension_code', 'heartdisease_code', 'ever_married']]
#Dependent Columns
y = df1['stroke']

#-----#
#-----Correlation-----#
#-----#
#Using Pearson Correlation
plt.figure(figsize=(14,10))
cor = x.corr().round(4) #correlation identified and rounded off upto 4 digits
labels = ['Age', 'Glucose\nLevel', 'BMI', 'Gender', 'Hyper\nTension', 'Heart\nDisease', 'Ever\nMarried']
ax = sns.heatmap(cor, annot=True, cmap=plt.cm.Red, xticklabels=labels, yticklabels=labels, linewidths=0.5,
bottom, top = ax.get_ylim()) #Getting vertical limits
ax.set_ylim(bottom + 0.5, top - 0.5) #To avoid heat map from trimming over edges
ax.axvline(x=0, color='k', linewidth=0.2) #To draw borders to heatmap
ax.axvline(x=cor.shape[0], color='k', linewidth=0.2) #To draw borders to heatmap
ax.axhline(y=0, color='k', linewidth=0.2) #To draw borders to heatmap
ax.axhline(y=cor.shape[1], color='k', linewidth=0.2) #To draw borders to heatmap
ax.set_xticklabels(ax.get_xticklabels(), rotation=0) #To avoid auto rotation of axis labels
ax.set_yticklabels(ax.get_yticklabels(), rotation=0) #To avoid auto rotation of axis labels
plt.show() #Showing the plot
```

Figure 23 : Feature Selection for Stroke Detection in Python

4. Train and Test data preparation

The data obtained after feature selection was then divided into a respective stratified ratio of 70:30 for Training and Testing of the models. This was done using the train_test_split() function of sklearn.model_selection package. Figure 24 illustrates the code for the same.

```
#-----Sampling of Test and Train Datasets-----#
#-----#
#Stratified sampling(Stratify = y) in the proportion of 70:30 using test_train_split to preserve the class balance
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=10, stratify=y)

#Verifying the counts
y_train.value_counts()
y_test.value_counts()
```

Figure 24 : Test and Train data preparation

5. Scaling of the Data

Once the Test and Train data is split, the immediate step was to scale the data. It was achieved using StandardScaler() function of sklearn.preprocessing package. Scaling is performed after Test and Train data split to avoid any impact of Test data values on the Training data values and keeping the Train set completely unaware of the Test set values. Figure 25 shows the code for the same.

```

#-----Scaling of data-----#
#-----#
#-----#
x_test_backup = x_test.copy(deep=True) #backup for final output

from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
continuous_variables = ['age', 'avg_glucose_level', 'bmi'] #customising labels for better representation of heat map
x_train[continuous_variables] = ss.fit_transform(x_train[continuous_variables])
x_test[continuous_variables] = ss.transform(x_test[continuous_variables])

```

Figure 25 : Scaling of Stroke Data

By the end of this chapter, it can now be concluded that the dataset is ready for the implementation of different sampling techniques, followed by the implementation of the machine learning models.

3.4 Implementation of the Data Sampling Techniques

To handle the class imbalance observed in the dataset, 3 different data sampling techniques were chosen for the implementation, i.e., SMOTE, Tomek Links and SMOTE + Tomek. The SMOTE technique was implemented using SMOTE() function of imblearn.over_sampling package, the Tomek Links technique was implemented using TomekLinks() function of imblearn.under_sampling package, whereas SMOTE + Tomek was implemented using SMOTETomek() function of imblearn.combine package. The code for the same is shown in Figure 26.

All the 3 sampling techniques were implemented only on the Training data and not the Testing data, to maintain the integrity of the data.

```

#-----SMOTE Oversampling-----#
#-----#
#-----#
sm = SMOTE(random_state=10, ratio = 1.0)
x_smote_resampled, y_smote_resampled = sm.fit_sample(x_train, y_train)
print(sorted(Counter(y_smote_resampled).items()))

#-----Tomek Links Undersampling-----#
#-----#
#-----#
# TomekLinks
tm = TomekLinks(random_state=10, return_indices = True, ratio='majority') #sampling strategy
x_tomek_resampled, y_tomek_resampled, it_tomek_resampled = tm.fit_sample(x_train, y_train)
print(sorted(Counter(y_tomek_resampled).items()))

#-----SMOTE + TomekLinks Combined Sampling-----#
#-----#
#-----#
smt = SMOTETomek(random_state=10)
x_smotetomek_resampled, y_smotetomek_resampled = smt.fit_resample(x_train, y_train)
print(sorted(Counter(y_smotetomek_resampled).items()))

#-----XXX-----XXX-----XXX-----#
#-----XXX-----End of Sampling-----XXX-----#
#-----XXX-----XXX-----XXX-----#

```

Figure 26 : Implementation of Sampling Techniques

3.5 Implementation of the Machine Learning Models

On successful implementation of the sampling techniques, data is ready for the implementation of the machine learning models.

3.5.1 Pre-Model Execution Steps

Prior to the implementation of the machine learning models, few pre-model execution steps illustrated as part of Figure 27 were performed. The code for pre-model execution steps consists of the conversion of the test data from pandas to Numpy using .to_numpy() function.

This is followed by the calculation of the class weights using `compute_class_weight()` package of `sklearn.utils.class_weight` the package, used as an input while implementing models like XGBoost and Neural Network.

In the end, a function `eval_matrix` was created that accepts actual test outcome and the predicted test outcome as input and calculates the value for all the performance metrics. This function was then used for the evaluation of the implemented models. The values of evaluation metrics were then stored into the Result dataframe.

```

#-----Pre-Model Execution Steps-----#
#-----#
#-----#
#converting test set to numpy
x_test_np = x_test.to_numpy()
y_test_np = y_test.to_numpy()

#Calculating weights
from sklearn.utils.class_weight import compute_class_weight
import numpy as np
weights = compute_class_weight('balanced', np.unique(df1['stroke']), df1['stroke'])

#Pre-execution step to automate implementation
x_train_sample = [x_smote_resampled, x_tomek_resampled, x_smotetomek_resampled]
y_train_sample = [y_smote_resampled, y_tomek_resampled, y_smotetomek_resampled]
sample_name = ['SMOTE', 'TOMEKLINKS', 'SMOTETOMEK']
column_names = ['Model_Name', 'Sample_Name', 'Accuracy', 'Sensitivity (TPR)', 'Specificity (TNR)', 'AUROC', 'TP', 'FN', 'FP', 'TN']
Results = pd.DataFrame(columns = column_names)

#Function for calculating evaluation matrix
def eval_matrix(y_test, y_pred):
    #evaluation code
    #cm = confusion_matrix(y_test, y_pred, Labels=[1,0])
    Accuracy = round(accuracy_score(y_test, y_pred),2)
    TP,FN,FP,TN = confusion_matrix(y_test, y_pred, labels=[1, 0]).ravel()
    Sensitivity = round(TP / (TP+FN),2)
    Specificity = round(TN / (TN+FP),2)
    AUC = round(roc_auc_score(y_test_np, y_pred),2) #Higher the AUC, better the model is at distinguishing between 0 and 1.
    return Accuracy, Sensitivity, Specificity, AUC, TP, FN, FP, TN

```

Figure 27 : Implementation of pre-model Execution Steps

3.5.2 Implementation and Evaluation of XGBoost

XGBoost model was implemented using the `XGBClassifier()` function of `xgboost` package. The parameters were then tuned to attain the optimum results. Figure 28 shows the code for the implementation of the XGBoost model for each of the 3 different training data samples followed by its evaluation using the `eval_matrix` function.

```

#-----XGBOOST-----#
#-----#
#-----#
#pip install xgboost #for installation
from xgboost import XGBClassifier #Importing XGBoost
from sklearn.metrics import accuracy_score, confusion_matrix, roc_auc_score
from xgboost import plot_importance #To plot important features
import math

#Model
xgbc = XGBClassifier(max_depth=4, objective= 'binary:logistic', scale_pos_weight=math.sqrt(w
#xgbc = XGBClassifier( objective= 'binary:logistic', scale_pos_weight=weights[1]) #73,76,73,
for i in range(0,3):
    xgbc.fit(x_train_sample[i], y_train_sample[i])
    # make predictions for test data
    y_pred_xgbc = xgbc.predict(x_test_np)
    # print(sorted(Counter(y_pred).items()))
    # print(sorted(Counter(y_test_np).items()))
    evaluation = eval_matrix(y_test_np, y_pred_xgbc)
    xgboost_results = pd.DataFrame(['XGBoost', sample_name[i], evaluation[0], evaluation[1]
Results = Results.append(xgboost_results, ignore_index=True)
#print('Confusion Matrix: \n', confusion_matrix(y_test_np, y_pred, Labels=[1, 0])) #for
plot_importance(xgbc, title="XGBoost "+sample_name[i])
print("Execution Completed for: Voting XGBoost - " + sample_name[i])
print(Results)

```

Figure 28 : Implementation and Evaluation of XGBoost

3.5.3 Implementation and Evaluation of Random Forest

Random Forest model was implemented using the RandomForestClassifier() function of sklearn.ensemble package. The parameter tuning was then done further to attain the optimum results. Figure 29 shows the code for the implementation of the Random Forest model for each of the 3 different training data samples followed by its evaluation using the eval_matrix function.

```
#-----Random Forest-----#
#-----#
RFindices = pd.DataFrame() #to store Important Features
from sklearn.ensemble import RandomForestClassifier
# create model
rfc = RandomForestClassifier(n_estimators=100, max_depth=2, random_state=0, class_weight='balanced')
# fit the model
for i in range(0,3):
    rfc.fit(x_train_sample[i], y_train_sample[i])
    y_pred_rfc = rfc.predict(x_test_np) #Predict Output
# Evaluation Metrics
# print(Counter(y_pred_rf).items())
# print(sorted(Counter(y_test_np).items()))
evaluation = eval_matrix(y_test_np, y_pred_rfc)
rfc_results = pd.DataFrame([[ 'Random Forest', sample_name[i], evaluation[0], evaluation[1], evaluation[2] ]])
#print('Confusion Matrix: \n', confusion_matrix(y_test_np, y_pred, Labels=[1, 0])) #for representation
Results = Results.append(rfc_results, ignore_index=True)
#feature importances = pd.DataFrame(rfc.feature_importances_, index = x_train.columns, columns=['feature', 'importance'])
print("Execution Completed for: Voting Random Forest - " + sample_name[i])
##feature Importance
RFindices[i] = np.argsort(rfc.feature_importances_)[::-1]
print(Results)
```

Figure 29 : Implementation and Evaluation of Random Forest

3.5.4 Implementation and Evaluation of Logistic Regression

Logistic Regression model was implemented using the LogisticRegression() function of sklearn.linear_model package. The penalty was set to 'l2' and class_weight was set to balanced. Figure 30 shows the code for the implementation of the Logistic Regression model for each of the 3 different training data samples followed by its evaluation using the eval_matrix function.

```
#-----Logistic Regression-----#
#-----#
from sklearn.linear_model import LogisticRegression
# create model
lrc = LogisticRegression(penalty = 'l2', class_weight='balanced', random_state=0)
#fit the model
for i in range(0,3):
    lrc.fit(x_train_sample[i], y_train_sample[i])
    y_pred_lrc= lrc.predict(x_test_np)
# Evaluation Metrics
# print(Counter(y_pred_lrc).items())
# print(sorted(Counter(y_test_np).items()))
evaluation = eval_matrix(y_test_np, y_pred_lrc)
lrc_results = pd.DataFrame([[ 'Logistic Regression', sample_name[i], evaluation[0], evaluation[1], evaluation[2] ]])
#print('Confusion Matrix: \n', confusion_matrix(y_test_np, y_pred, Labels=[1, 0])) #for representation
Results = Results.append(lrc_results, ignore_index=True)
print("Execution Completed for: Logistic Regression - " + sample_name[i])
print(Results)
```

Figure 30 : Implementation and Evaluation of Logistic Regression

3.5.5 Implementation and Evaluation of Naïve Bayes

Gaussian Naïve Bayes model was implemented using the GaussianNB() function of sklearn.naive_bayes package. Figure 31 shows the code for the implementation of the Naïve Bayes model for each of the 3 different training data samples followed by its evaluation using the eval_matrix function.

```
#-----#
#-----Naive Bayes-----#
#-----#
from sklearn.naive_bayes import GaussianNB
# create model
gnbc = GaussianNB()
#fit the model
for i in range(0,3):
    gnbc.fit(x_train_sample[i], y_train_sample[i])
    y_pred_gnbc = gnbc.predict(x_test_np) #Predict Output
    evaluation = eval_matrix(y_test_np, y_pred_gnbc)
    gnbc_results = pd.DataFrame(['Naive Bayes', sample_name[i], evaluation[0], evaluation[1]])
    Results = Results.append(gnbc_results, ignore_index=True)
    print("Execution Completed for: Naive Bayes - " + sample_name[i])
print(Results)
```

Figure 31 : Implementation and Evaluation of Naive Bayes

3.5.6 Implementation and Evaluation of Support Vector Classifier

Support Vector Classifier (SVC) model was implemented using the SVC() function of sklearn.svm package. The parameter tuning was then done further to attain the optimum results. Figure 32 shows the code for the implementation of the Support Vector Classifier model for each of the 3 different training data samples followed by its evaluation using the eval_matrix function.

```
#-----#
#-----SVC-----#
#-----#
from sklearn.svm import SVC
svc = SVC(kernel= 'sigmoid', max_iter=12000, C=25, gamma=0.03, probability=True) #degree=40,
#fit the model
for i in range(0,3):
    svc.fit(x_train_sample[i], y_train_sample[i])
    y_pred_svc = svc.predict(x_test_np) #Predict Output
# Evaluation Metrics
# print(Counter(y_pred_rf).items())
# print(sorted(Counter(y_test_np).items()))
evaluation = eval_matrix(y_test_np, y_pred_svc)
svc_results = pd.DataFrame(['SVC', sample_name[i], evaluation[0], evaluation[1], evaluation[2],
# print('Confusion Matrix: \n', confusion_matrix(y_test_np, y_pred, labels=[1, 0])) #for represent
Results = Results.append(svc_results, ignore_index=True)
    print("Execution Completed for: Support Vector Classifier - " + sample_name[i])
print(Results)
```

Figure 32 : Implementation and Evaluation of Support Vector Classifier

3.5.7 Implementation and Evaluation of K-Nearest Neighbors (KNN)

K-Nearest Neighbors model was implemented using the KNeighborsClassifier() function of sklearn.neighbors package. The best-balanced results were achieved with the value for n_neighbors as 70. Figure 33 shows the code for the implementation of the KNN model for each of the 3 different training data samples followed by its evaluation using the eval_matrix function.

```

#-----KNN-----#
#-----#
#-----#
from sklearn.neighbors import KNeighborsClassifier

knc = KNeighborsClassifier(n_neighbors=70)
for i in range(0,3):
    knc.fit(x_train_sample[i], y_train_sample[i])
    y_pred_knc = knc.predict(x_test_np) #Predict Output
# Evaluation Metrics
# print(Counter(y_pred_rf).items())
# print(sorted(Counter(y_test_np).items()))
evaluation = eval_matrix(y_test_np, y_pred_knc)
knc_results = pd.DataFrame(['KNN', sample_name[i], evaluation[0], evaluation[1], evaluation[2],
# print('Confusion Matrix: \n', confusion_matrix(y_test_np, y_pred, labels=[1, 0])) #for representation as i
Results = Results.append(knc_results, ignore_index=True)
print("Execution Completed for: KNN - " + sample_name[i])
print(Results)

```

Figure 33 : Implementation and Evaluation of K-Nearest Neighbors

3.5.8 Implementation and Evaluation of Decision Tree Classifier

Decision Tree Classifier model was implemented using the DecisionTreeClassifier() function of sklearn.tree package. The parameter tuning was then done further to attain the optimum results. Figure 34 shows the code for the implementation of the Decision Tree Classifier model for each of the 3 different training data samples followed by its evaluation using the eval_matrix function.

```

#-----Decision Tree-----#
#-----#
#-----#
#Decision Tree Classifier
from sklearn.tree import DecisionTreeClassifier
dtc = DecisionTreeClassifier(criterion='entropy',max_depth= 10,random_state=0, class_weight='balanced')
for i in range(0,3):
    dtc.fit(x_train_sample[i], y_train_sample[i])
    y_pred_dtc = dtc.predict(x_test_np) #Predict Output
# Evaluation Metrics
# print(Counter(y_pred_rf).items())
# print(sorted(Counter(y_test_np).items()))
evaluation = eval_matrix(y_test_np, y_pred_dtc)
dtc_results = pd.DataFrame(['Decision Tree', sample_name[i], evaluation[0], evaluation[1], evaluation[2],
# print('Confusion Matrix: \n', confusion_matrix(y_test_np, y_pred, labels=[1, 0])) #for representation as i
Results = Results.append(dtc_results, ignore_index=True)
print("Execution Completed for: Decision Tree - " + sample_name[i])
print(Results)

```

Figure 34 : Implementation and Evaluation of Decision Tree Classifier

3.5.9 Implementation and Evaluation of AdaBoost

AdaBoost Classifier model was implemented using the AdaBoostClassifier() function of sklearn.ensemble package. The parameter tuning was then done further to attain the optimum results. Figure 35 shows the code for the implementation of the AdaBoost Classifier model for each of the 3 different training data samples followed by its evaluation using the eval_matrix function.

```

#-----#
#-----AdaBoostClassifier-----#
#-----#
from sklearn.ensemble import AdaBoostClassifier
abc = AdaBoostClassifier(n_estimators=5, learning_rate=1.0, random_state=0)
for i in range(0,3):
    abc.fit(x_train_sample[i], y_train_sample[i])
    y_pred_abc = abc.predict(x_test_np) #Predict Output
# Evaluation Metrics
# print(Counter(y_pred_rf).items())
# print(sorted(Counter(y_test_np).items()))
evaluation = eval_matrix(y_test_np, y_pred_abc)
abc_results = pd.DataFrame(['AdaBoost Classifier', sample_name[i], evaluation[0], evaluation[1],
# print('Confusion Matrix: \n', confusion_matrix(y_test_np, y_pred, labels=[1, 0])) #for representa
Results = Results.append(abc_results, ignore_index=True)
print("Execution Completed for: Voting Classifier - " + sample_name[i])
print(Results)

```

Figure 35 : Implementation and Evaluation of AdaBoost

3.5.10 Implementation and Evaluation of Neural Network

Neural Network Model was implemented using the KerasClassifier() function of the keras.wrappers.scikit_learn package. A function was created that defined the structure of the model. It was then passed as a parameter for the KerasClassifier() function. The parameter tuning was then done further to attain the optimum results. Figure 36 shows the code for the implementation of the Neural Network model for each of the 3 different training data samples followed by its evaluation using the eval_matrix function.

```

#Create Model with KerasClassifier wrapper for scikit_learn
def create_model():
    nnc = Sequential()
    nnc.add(Dense(30, input_dim=10, activation='relu'))
    nnc.add(Dense(20, activation='relu'))
    nnc.add(Dense(1, activation='sigmoid'))
    # Compile model
    nnc.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return nnc

# Compile model
nnc = KerasClassifier(build_fn=create_model, verbose=0)
# Fit the model
for i in range(0,3):
    history_nnc = nnc.fit(x_train_sample[i], y_train_sample[i], validation_split=0.33, epochs=1000, batch_size=2000, class_weight = {0: weights[0], 1: weights[1]})
    # evaluate the model
    #y_pred_nnc = nnc.predict_classes(x_test_np) #only for Keras
    #y_pred_nnc = y_pred_nnc.reshape(-1) #only for Keras
    # print(Counter(y_pred_nnc).items())
    # print(sorted(Counter(y_test_np).items()))
    y_pred_nnc = nnc.predict(x_test_np) #for scikit
    evaluation = eval_matrix(y_test_np, y_pred_nnc) #forscikit
    nnc_results = pd.DataFrame(['Neural Network', sample_name[i], evaluation[0], evaluation[1], evaluation[2], evaluation[3], evaluation[4], evaluation[5], evaluat
    #print('Confusion Matrix: \n', confusion_matrix(y_test_np, y_pred, labels=[1, 0])) #for representation as it is inverse in python.(https://scikit-learn.org/stab
    Results = Results.append(nnc_results, ignore_index=True)
    print("Execution Completed for: Neural Network - " + sample_name[i])
print(Results)

```

Figure 36 : Implementation and Evaluation of Neural Network

3.5.11 Implementation and Evaluation of Ensemble Voting Classifier

After developing all the 9 machine learning models, they were ensembled together as predictors for an Ensemble Voting Classifier, that with the help of probabilities provided by each of these models predict the outcome of the dependent variable. The implementation of Ensemble Voting Classifier is a robust approach as it creates an additional layer of assessment before making the actual decision. The voting classifier is then again created each of the 3 sampling techniques and the results are further evaluated using eval_matrix function.

As Ensemble Voting Classifier is the ultimate goal of this project, the predictions obtained by this model along with its features, are exported to .csv files using .to_csv() function of pandas. The evaluations results obtained for all the 30 different combinations of machine learning





 Output Results	Microsoft Excel Comma Separated Values File	2 KB
 Prediction_Using_SMOTE	Microsoft Excel Comma Separated Values File	327 KB
 Prediction_Using_SMOTETOMEK	Microsoft Excel Comma Separated Values File	327 KB
 Prediction_Using_TOMEKLINKS	Microsoft Excel Comma Separated Values File	327 KB

Figure 39 : .csv Files Generated

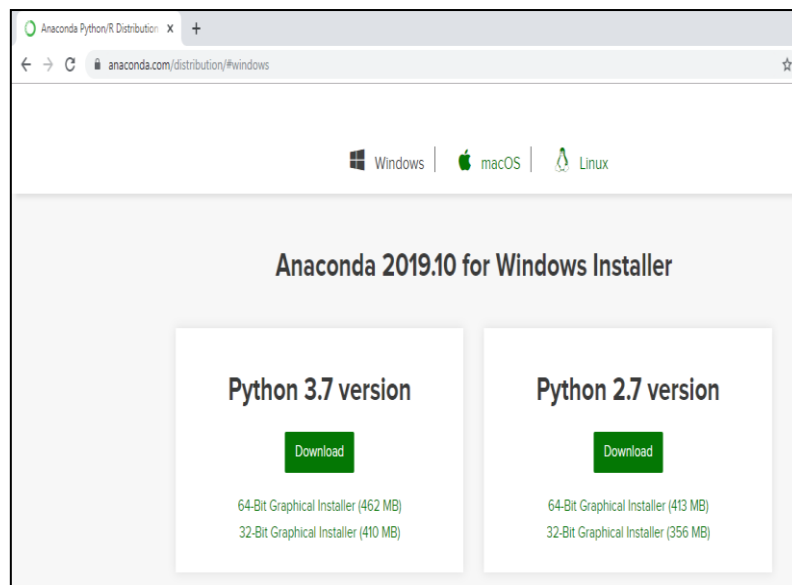
Model_Name	Sample_Name	Accuracy	Sensitivity (TPR)	Specificity (TNR)	AUC	TP	FN	FP	TN
0 XGBoost	SMOTE	0.68	0.78	0.68	0.73	122	35	2711	5812
1 XGBoost	TOMEKLINKS	0.98	0.04	1	0.52	6	151	24	8499
2 XGBoost	SMOTETOMEK	0.69	0.78	0.69	0.73	122	35	2672	5851
3 Random Forest	SMOTE	0.72	0.83	0.72	0.78	131	26	2361	6162
4 Random Forest	TOMEKLINKS	0.66	0.91	0.66	0.78	143	14	2938	5585
5 Random Forest	SMOTETOMEK	0.72	0.83	0.72	0.78	131	26	2383	6140
6 Logistic Regression	SMOTE	0.73	0.85	0.73	0.79	133	24	2300	6223
7 Logistic Regression	TOMEKLINKS	0.73	0.85	0.73	0.79	133	24	2314	6209
8 Logistic Regression	SMOTETOMEK	0.73	0.85	0.73	0.79	134	23	2301	6222
9 Naive Bayes	SMOTE	0.7	0.85	0.69	0.77	134	23	2611	5912
10 Naive Bayes	TOMEKLINKS	0.91	0.4	0.92	0.66	63	94	712	7811
11 Naive Bayes	SMOTETOMEK	0.7	0.85	0.69	0.77	134	23	2612	5911
12 SVC	SMOTE	0.7	0.68	0.7	0.69	107	50	2594	5929
13 SVC	TOMEKLINKS	0.97	0.06	0.98	0.52	10	147	137	8386
14 SVC	SMOTETOMEK	0.7	0.69	0.7	0.69	108	49	2590	5933
15 KNN	SMOTE	0.72	0.71	0.72	0.72	112	45	2356	6167
16 KNN	TOMEKLINKS	0.98	0	1	0.5	0	157	0	8523
17 KNN	SMOTETOMEK	0.72	0.71	0.72	0.72	112	45	2356	6167
18 Decision Tree	SMOTE	0.75	0.7	0.75	0.73	110	47	2125	6398
19 Decision Tree	TOMEKLINKS	0.79	0.62	0.79	0.71	98	59	1752	6771
20 Decision Tree	SMOTETOMEK	0.75	0.7	0.75	0.73	110	47	2126	6397
21 AdaBoost Classifier	SMOTE	0.72	0.82	0.72	0.77	129	28	2414	6109
22 AdaBoost Classifier	TOMEKLINKS	0.98	0	1	0.5	0	157	0	8523
23 AdaBoost Classifier	SMOTETOMEK	0.72	0.82	0.72	0.77	129	28	2414	6109
24 Neural Network	SMOTE	0.73	0.71	0.73	0.72	112	45	2300	6223
25 Neural Network	TOMEKLINKS	0.79	0.59	0.8	0.69	92	65	1728	6795
26 Neural Network	SMOTETOMEK	0.74	0.73	0.74	0.73	114	43	2233	6290
27 Voting Classifier Model	SMOTE	0.74	0.84	0.74	0.79	132	25	2197	6326
28 Voting Classifier Model	TOMEKLINKS	0.97	0.13	0.99	0.56	21	136	108	8415

Figure 40 : Sample Output of Model evaluation in .csv File

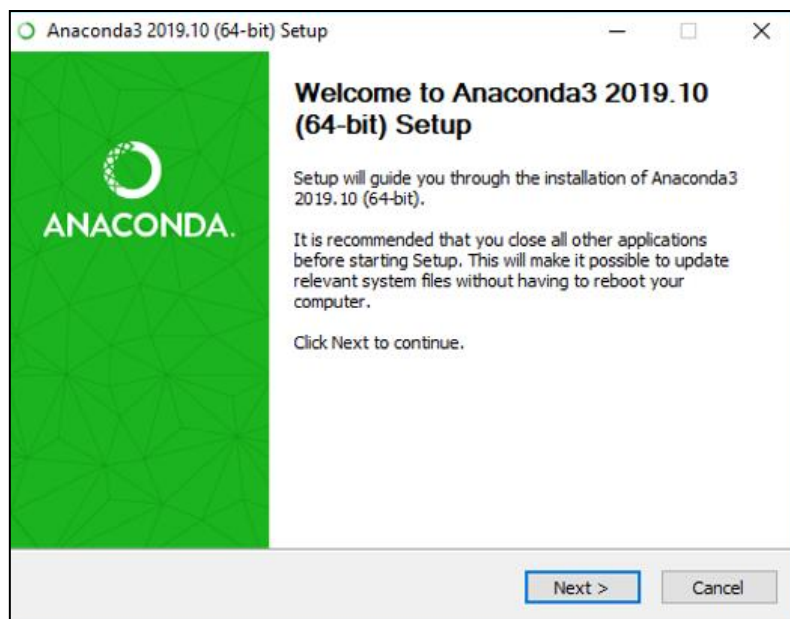
5 Appendix

5.1 Installation of Anaconda

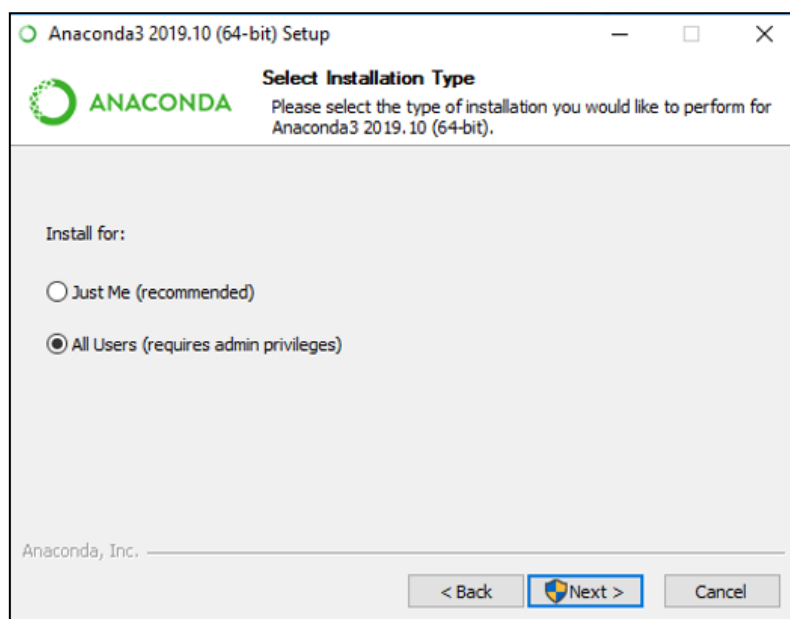
1. Go to <https://www.anaconda.com/distribution/#windows> and download Anaconda Windows Installer with Python 3.7 Version.



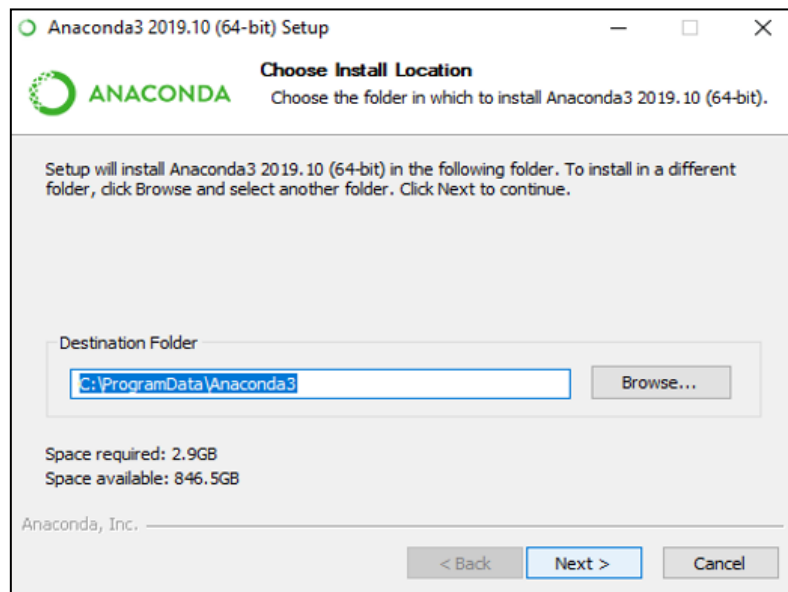
2. On Double-clicking the downloaded file, setup window will appear. Click on the “Next” button.



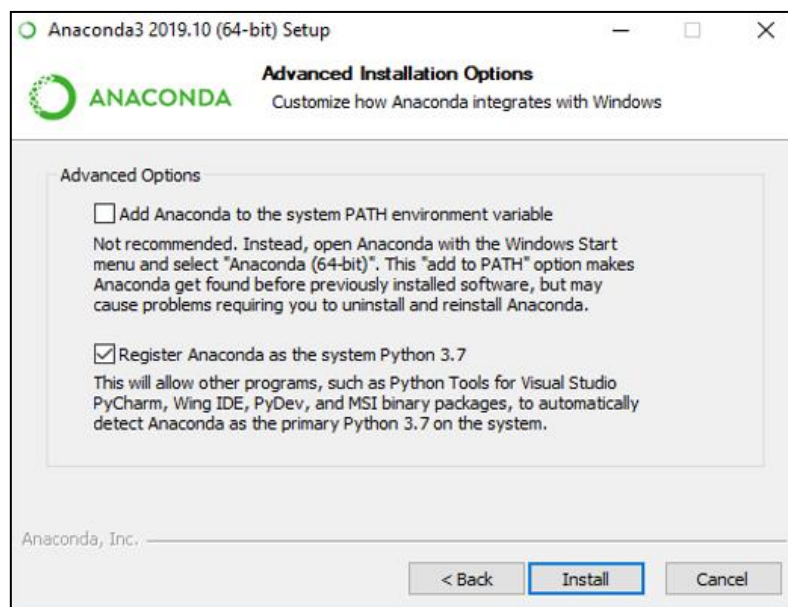
3. Select the appropriate value for “Install for” option based on the type of machine and users. Click on the “Next” button.



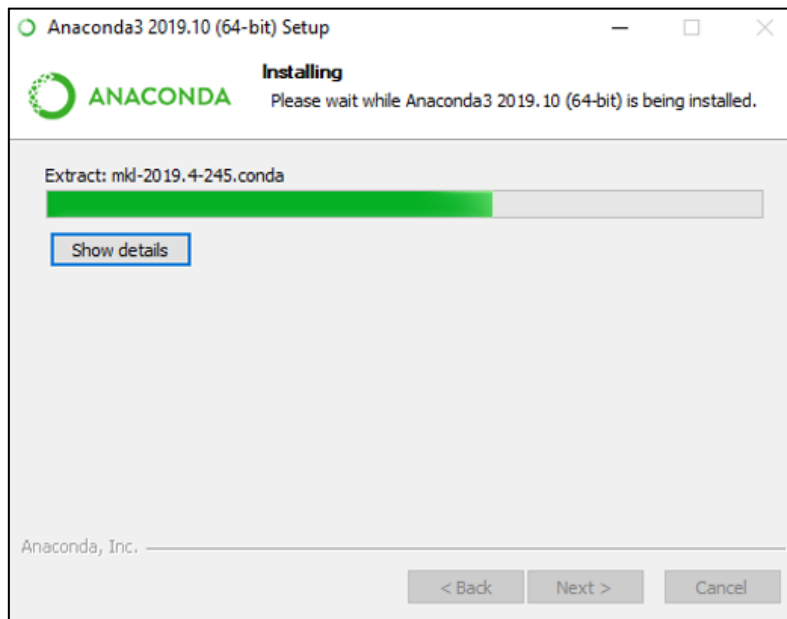
4. Browse the “Destination Folder” if you want to install Anaconda to any specific folder. Click on the “Next” button.



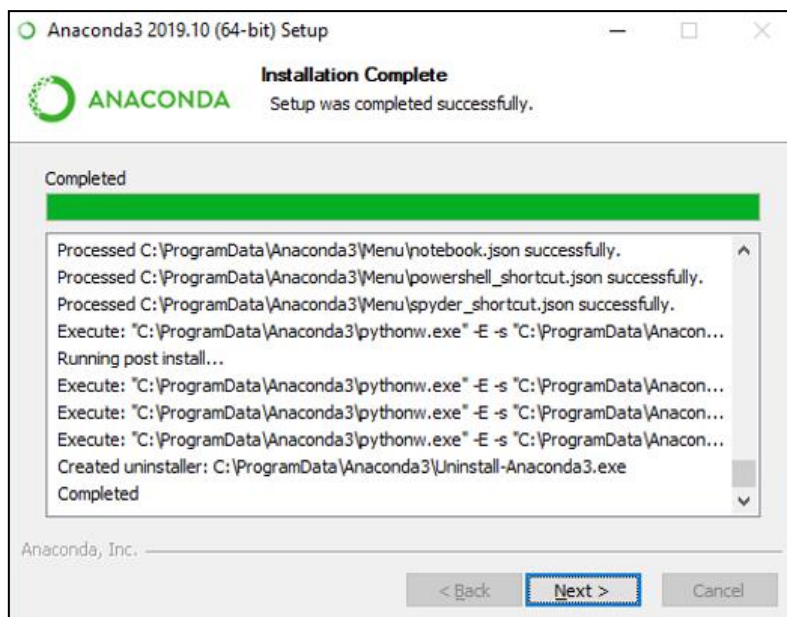
5. Tick the “Register Anaconda as the system Python 3.7” option and click on the “Install” button.



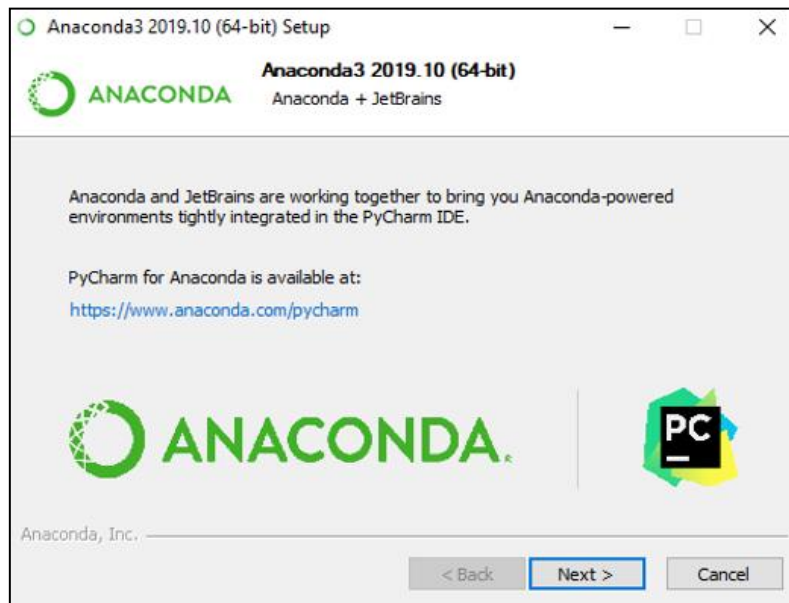
6. The installation process will now continue for approximately 5 minutes.



7. On the successful Installation, Installation complete window will appear. Click on the “Next” button.



8. An Information window will appear. Click on the “Next” button.



9. Thanks for installing window will appear. Click on the “Finish” button to complete your installation.

