National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Omkar Doke
Student ID: x18179525

School of Computing
National College of Ireland

Supervisor: Dr. Muhammad Iqbal

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Omkar Doke
Student ID:	x18179525
Programme:	Data Analytics
Year:	2020
Module:	MSc Research Project
Supervisor:	Dr. Muhammad Iqbal
Submission Due Date:	17/08/2020
Project Title:	Configuration Manual
Word Count:	1744
Page Count:	19

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:	10th August 2020

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
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.	☐

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

Omkar Doke
x18179525

1 Introduction

This configuration manual presents the software and hardware requirements along with the details of programming codes written for model implementation in research project:
"Data Mining for Enhancing Silicon Wafer Fabrication"

2 System Configuration

2.1 Hardware Specifications

Table 1 represents hardware specification of the system on which the research was accomplished.

Table 1: Hardware Specification

RAM	8 GB
Processor	Intel i7 8550U
Speed	1.99 GHz
Operating System	Windows 10, 64 Bit
Storage	1 TB HDD
GPU	NVIDIA GeForce MX150

2.2 Software Specifications

- **Microsoft Excel 2019:**
  Both the data-sets used in the research were downloaded and stored in csv (comma separated values) in excel. It was used for quick evaluation and exploratory plot.

- **Jupyter Notebook from Anaconda Distribution:**
  Anaconda Navigator is an open source software downloaded from the anaconda distribution website [1] . It supports jupyter notebooks to implement machine learning models on research data. Latest version of jupyter notebook (version 5.7.4) was used in the research for data preprocessing, exploratory data analysis (EDA), manipulation of data, transformation and implementation of models.

[1] `https://www.anaconda.com/products/individual`

3 Development of Project

Python programming was used to accomplish the research in various phases viz. data pre-processing of both the data-sets, EDA, merging of both the data-sets, addressing class imbalance and normalization of data to overcome the impact of outliers. It was followed by splitting data into train and test set for predictive modelling using classification-based machine learning algorithms and their cross validation using stratified K-fold validation technique. Sk-Learn (scikit-learn) and Keras were primary libraries used along with numpy, panda, matplotlib for executing the code.

3.1 Data Preparation

Both data-sets[2] downloaded from different websites[3] have been uploaded onto jupyter notebook in csv format. Following sections provide a detail insight of data-processing, EDA, feature engineering, dimensionality reduction performed on both data-sets followed by merging of data-sets for implementation and evaluation of models after addressing class imbalance.

3.1.1 UCI SEMCOM Dataset

Pre-processing of UCI SEMCOM dataset involves handling missing values. UCI SEMCOM dataset consists of 591 attributes with 27 attributes having more than 50% of missing values which were dropped as it didn't lead to data loss. Apart from that, attributes with zero variance (i.e. no effect of dependent variable) were dropped as their presence or absence didn't have any impact on research. Attributes with less than 50% of missing values were imputed with median as the attributes had outliers and data has skew symmetric distribution. Thereafter, dataset was normalized using MinMaxScaler library for scaling because attributes consisted of outliers as well as the attribute values were in different range. The dependent variable of UCI SEMCOM dataset consists of pass category defined as '-1' and fail category as '+1'. Code for preprocessing of UCI SEMCOM dataset is highlighted in Figure 1.

3.1.2 WAFER Dataset

Pre-processing of WAFER dataset involves handling missing values. WAFER dataset consists of 154 attributes with no attributes having more than 50% of missing values thereby none of the attributes were dropped. Also, when checked for impact of attributes on dependent variable, it was found that none of the attributes had zero variance. Attributes with less than 50% of missing values were imputed with mean as the attributes didn't have outliers. Thereafter, dataset was normalized using MinMaxScaler library for scaling. The dependent variable of WAFER dataset consisted of pass category defined as '+1' and fail category as '-1'. To have standardized definition of pass and fail classes in dependent variable, we interchanged the designation for WAFER dataset thereby assigning '-1' to pass class and '+1' to fail class. Code for preprocessing of WAFER dataset is highlighted in Figure 2.

[2] `http://www.timeseriesclassification.com/description.php?Dataset=Wafer`
[3] `https://archive.ics.uci.edu/ml/datasets/SECOM`

Data Cleaning

```
## Dropping 1st row
Wafer_fabrication_df = Wafer_fabrication_df.drop(Wafer_fabrication_df.index[0])

##Removing Columns with more than 50% NaN Values
cols = Wafer_fabrication_df.columns[Wafer_fabrication_df.isnull().mean()>0.5]
Wafer_df = Wafer_fabrication_df.drop(cols, axis=1)

## Dropping 1 column with date an time
Wafer_df = Wafer_df.drop(columns= [0], axis = 1)

##Dropping columns with 0 varainace
Wafer_df = Wafer_df.loc[:,Wafer_df.apply(pd.Series.nunique) != 1]
```

```
## Looking for outlier

plt.rcParams['figure.figsize'] = (18, 16)

plt.subplot(2, 2, 1)
sns.boxplot(Wafer_df[1], color = 'blue')
plt.title('First Sensor Measurements', fontsize = 20)

plt.subplot(2, 2, 2)
sns.boxplot(Wafer_df[20], color = 'darkred')
plt.title('20th Sensor Measurements', fontsize = 20)

plt.subplot(2, 2, 3)
sns.boxplot(Wafer_df[201], color = 'green')
plt.title('201th Sensor Measurements', fontsize = 20)

plt.subplot(2, 2, 4)
sns.boxplot(Wafer_df[578], color = 'orange')
plt.title('578th Sensor Measurements', fontsize = 20)

plt.show()
```

```
## Since their are outlier's in the columns, we are replacing missing values with median instead of mean
##Imputing missing values in columns with less than 50% NaN by the median of that column

Wafer_df.fillna(Wafer_df.median(), inplace= True)

## Separating dependent and independent variables and then Normalizing the independent variables (features)

# Separating out the features
x = Wafer_df.loc[:, :590]

# Separating out the target
y = Wafer_df.loc[:,591]

# Normalizing the features
norm = MinMaxScaler().fit(x)
x = norm.transform(x)
```

Figure 1: Data Cleaning of UCI SEMCOM Dataset

```
Wafer_fabrication2 = pd.read_csv('D:/NCI - Research Project/Data/Wafer/csv_result-Wafer.csv', header = None)
Wafer_fabrication2_df = pd.DataFrame(Wafer_fabrication2)

## Dropping 1st row
Wafer_fabrication2_df = Wafer_fabrication2_df.drop(Wafer_fabrication2_df.index[0])

##Removing Columns with more than 50% NaN Values
cols = Wafer_fabrication2_df.columns[Wafer_fabrication2_df.isnull().mean()>0.5]
Wafer2_df = Wafer_fabrication2_df.drop(cols, axis=1)

## Dropping 1st column with seqence number
Wafer2_df = Wafer2_df.drop(columns= [0], axis = 1)

##Dropping columns with 0 varainace
Wafer2_df = Wafer2_df.loc[:,Wafer2_df.apply(pd.Series.nunique) != 1]
```

```
## Looking for outlier

plt.rcParams['figure.figsize'] = (18, 16)

plt.subplot(2, 2, 1)
sns.boxplot(Wafer2_df[1], color = 'blue')
plt.title('First Sensor Measurements', fontsize = 20)

plt.subplot(2, 2, 2)
sns.boxplot(Wafer2_df[10], color = 'darkred')
plt.title('10th Sensor Measurements', fontsize = 20)

plt.subplot(2, 2, 3)
sns.boxplot(Wafer2_df[142], color = 'green')
plt.title('142th Sensor Measurements', fontsize = 20)

plt.subplot(2, 2, 4)
sns.boxplot(Wafer2_df[150], color = 'orange')
plt.title('150th Sensor Measurements', fontsize = 20)

plt.show()
```

```
##Since there are no outliers as per the boxplot, we are replacing the missing values with mean
##Imputing missing values in columns with less than 50% NaN by the mean of that column
Wafer2_df.fillna(Wafer2_df.mean(), inplace= True)

## Placing +1 for fail cases and -1 for Pass cases
Wafer2_df[153].replace({1: 0, -1: 1}, inplace=True)
Wafer2_df[153].replace({0: -1}, inplace=True)
Wafer2_df.head()

## Separating dependent and independent variables and then Normalizing the independent variables (features)
# Separating out the features
x2 = Wafer2_df.loc[:, :152]

# Separating out the target
y2 = Wafer2_df.loc[:,153]

# Normalizing the features
norm = MinMaxScaler().fit(x2)
x2 = norm.transform(x2)
```

Figure 2: Data Cleaning of WAFER Dataset

3.2 Dimensionality Reduction on both data-sets

3.2.1 UCI SEMCOM Dataset

Feature extraction was performed using principle component analysis (PCA) technique to extract top components explaining 80% variance of the data. PCA was applied to extract 250 components from 447 attributes. Then after, variance ratio was calculated and plotted for principal components which led to the selection of top 100 components as they explained more that 80% variance of data. Figure 3 represents the code for implementation of PCA on UCI SEMCOM data for extracting principle components.

Applying PCA

```
## Applying PCA to get top Principle components representing maximum variance in data
pca = PCA(n_components=250)
principalComponents = pca.fit_transform(x)
columns = ['pca_%i' % i for i in range(250)]
principalDf = pd.DataFrame(data = principalComponents, columns = columns)

## Identifying number of PCA's that explain maximum variance of data (we are attempting to find for 98-99% or more)
pca.fit(x)
variance = pca.explained_variance_ratio_ #calculate variance ratios
var=np.cumsum(np.round(pca.explained_variance_ratio_, decimals=3)*100)
var #cumulative sum of variance explained with [n] features

##Plotting PCA's against variance to identify the PCA's to be selected
fig= plt.figure(figsize=(8,5))
plt.ylabel('% Variance Explained')
plt.xlabel("Number of PCA's")
plt.title('PCA Analysis')
plt.ylim(5,100.5)
plt.style.context('seaborn-blackgrid')
plt.grid(True)
plt.plot(var)
```

```
##Thus from th previous plot, selecting first 100 PCA for this research as they explain more than 80% variance of data
principalDf.drop(principalDf.iloc[:,100:250], axis = 1, inplace = True)

##Concatinating Dependent Variable to the dataframe
FinalDf = pd.concat([principalDf, y], axis = 1)

##Renumbering last column count
FinalDf.rename(columns={591: "Pass/Fail"}, inplace= True)

##Shifting last column up by 1 row
FinalDf['Pass/Fail'] = FinalDf['Pass/Fail'].shift(-1)

##Dropping the last row
FinalDf = FinalDf[:-1]
```

Figure 3: Feature Extraction using PCA on UCI SEMCOM Data

Feature selection was performed using Analysis of Variance (ANOVA) technique to select top features explaining more that 80% variance in data. Initially number of features were gradually reduced to identify feature count for which models provide optimum performance, however different models provided optimum performance for different feature count. Then after, top 100 features were selected from the data to compare model's performance with that of feature extraction technique. Code for feature selection using ANOVA is highlighted in Figure 4.

Applying ANOVA

```
##Select Features With Best ANOVA F-Values
## Create an SelectKBest object to select features with two best ANOVA F-Values
fvalue_selector = SelectKBest(f_classif, k=100)

## Apply the SelectKBest object to the features and target
X_kbest = fvalue_selector.fit_transform(x, y)
FS_1Df = pd.DataFrame(X_kbest)

##Concatinating Dependent Variable to the dataframe
FS_1Df = pd.concat([FS_1Df, y], axis = 1)

##Renumbering last column count
FS_1Df.rename(columns={591: "Pass/Fail"}, inplace= True)

##Shifting last column up by 1 row
FS_1Df['Pass/Fail'] = FS_1Df['Pass/Fail'].shift(-1)

##Dropping the last row
FS_1Df = FS_1Df[:-1]
FS_1Df.head()
```

Figure 4: Feature Selection using ANOVA on UCI SEMCOM Data

3.2.2 WAFER Dataset

Feature selection was performed using Analysis of Variance (ANOVA) technique to select top features explaining more that 80% variance in data. Initially number of features were gradually reduced to identify feature count for which models provide optimum performance, however different models provided optimum performance for different feature count. Then after, top 100 features were selected from the data to compare model's performance with that of feature extraction technique. Code for feature selection using ANOVA is highlighted in Figure 5.

Applying ANOVA

```
##Select Features With Best ANOVA F-Values
## Create an SelectKBest object to select features with two best ANOVA F-Values
fvalue_selector = SelectKBest(f_classif, k=100)

## Apply the SelectKBest object to the features and target
X2_kbest = fvalue_selector.fit_transform(x2, y2)
FS_2Df = pd.DataFrame(X2_kbest)

##Concatinating Dependent Variable to the dataframe
FS_2Df = pd.concat([FS_2Df, y2], axis = 1)

##Renumbering last column count
FS_2Df.rename(columns={153: "Pass/Fail"}, inplace= True)

##Shifting last column up by 1 row
FS_2Df['Pass/Fail'] = FS_2Df['Pass/Fail'].shift(-1)

##Dropping the last row
FS_2Df = FS_2Df[:-1]
FS_2Df.head()
```

Figure 5: Feature Selection using ANOVA on WAFER Data

Feature extraction was performed using principle component analysis (PCA) technique to

extract top components explaining 80% variance of the data. PCA was applied to extract 150 components from attributes. Then after, variance ratio was calculated and plotted for principal components which led to the selection of top 100 components as they explained more that 80% variance of data. Figure 6 represents the code for implementation of PCA on UCI SEMCOM data for extracting principle components.

Applying PCA

```
## Applying PCA to get top Principle components representing maximum variance in data
pca2 = PCA(n_components=150)
principalComponents2 = pca2.fit_transform(x2)
columns = ['pca_%i' % i for i in range(150)]
principal2_Df = pd.DataFrame(data = principalComponents2, columns = columns)

## Identifying number of PCA's that explain maximum variance of data (we are attempting to find for 98-99% or more)
pca2.fit(x2)
variance2 = pca2.explained_variance_ratio_ #calculate variance ratios
var2=np.cumsum(np.round(pca2.explained_variance_ratio_, decimals=3)*100)
var2 #cumulative sum of variance explained with [n] features

##Plotting PCA's against variance to identify the PCA's to be selected
fig= plt.figure(figsize=(8,5))
plt.ylabel('% Variance Explained')
plt.xlabel("Number of PCA's")
plt.title('PCA Analysis')
plt.ylim(5,100.5)
plt.style.context('seaborn-blackgrid')
plt.grid(True)
plt.plot(var2)
```

```
##Thus from th previous plot, selecting first 100 PCA for this research
principal2_Df.drop(principal2_Df.iloc[:,100:150], axis = 1, inplace = True)

##Concatinating Dependent Variable to the dataframe
Final2_Df = pd.concat([principal2_Df, y2], axis = 1)

##Renumbering last column count
Final2_Df.rename(columns={153: "Pass/Fail"}, inplace= True)

##Shifting last column up by 1 row
Final2_Df['Pass/Fail'] = Final2_Df['Pass/Fail'].shift(-1)

##Dropping the last row
Final2_Df = Final2_Df[:-1]
```

Figure 6: Feature Extraction using PCA on WAFER Data

3.3 Merging of data

3.3.1 Merging of feature extracted data frames

Two data frames are created of principle components extracted from both data-sets which are then merged as both had same number of columns. The pass category which was initially assigned '-1' label was reassigned with label '0' and descriptive analysis was performed on final merged dataset. Code in Figure 7 shows how feature extracted PCA data frames from both data-sets were merged together.

3.3.2 Merging of feature selected data frames

Two data frames are created of feature's selected from both data-sets which are then merged as both had same number of columns. The pass category which was initially assigned '-1' label was reassigned with label '0'. Code in Figure 8 shows how feature selected data frames from both data-sets were merged together.

Merging Two PCA Data Frames

```
##Merging data frames
Wafer = pd.concat([FinalDf, Final2_Df], ignore_index=True)

## Changing -1 to 0
Wafer['Pass/Fail'].replace({-1.0: 0.0}, inplace=True)

##Statistical Values of Each Column
Wafer_des = Wafer.describe()
Wafer_des
```

Figure 7: Merging two PCA Data Frames

Merging Two Feature Selected Data Frames

```
##Merging data frames
FS_Wafer = pd.concat([FS_1Df, FS_2Df], ignore_index=True)

## Changing -1 to 0
FS_Wafer['Pass/Fail'].replace({-1.0: 0.0}, inplace=True)
```

Figure 8: Merging two Feature Selected Data Frames

3.4 Splitting the data into Train and Test set

After merging, both the data-set were split into train and test part in 75:25 ratio respectively. Models were trained on train set and evaluated on test set. Their performance was cross validated using stratified K-fold validation technique. Figure 9 illustrates the code for train test split of final dataset.

```
##Again separating the dependent and independent variables from FinalDf
x = Wafer.iloc[:,:100]
y = Wafer.iloc[:, 100]

#Getting the shapes of new data sets x and y
print("Shape of x:", x.shape)
print("Shape of y:", y.shape)

##Splitting the data into train and test sets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.25, random_state = 0)

# gettiing the shapes
print("Shape of x_train: ", x_train.shape)
print("Shape of x_test: ", x_test.shape)
print("Shape of y_train: ", y_train.shape)
print("Shape of y_test: ", y_test.shape)
```

```
Shape of x: (8731, 100)
Shape of y: (8731,)
Shape of x_train:  (6548, 100)
Shape of x_test:  (2183, 100)
Shape of y_train:  (6548,)
Shape of y_test:  (2183,)
```

Figure 9: Train Test Split of Merged Data

3.5 Addressing Class Imbalance

3.5.1 Oversampling of Feature Selected and Feature Extracted Data

After splitting the data, major class imbalance was observed in train set with fail class contributing approximately 10% of entire data. This was then addressed using Synthetic Minority Over-Sampling Technique (SMOTE) wherein the minority class was oversampled to 50% to that of majority class in both features extracted and features selected data. Figure 10 represents the code for oversampling of minority class using SMOTE.

Oversampling using SMOTE of the Fail Cases

```
SM = SMOTE(sampling_strategy= 0.50, random_state= None)
x_train_os, y_train_os = SM.fit_sample(x_train, y_train)

print("Shape of x_train_os: ", x_train_os.shape)
print("Shape of y_train_os: ", y_train_os.shape)
```

Figure 10: Oversampling of Train Set using SMOTE

3.5.2 Random Sampling of Feature Selected Data

In another experiment, class imbalance of feature selected data was address by random oversampling of minority class along with random under-sampling of majority class. 3 different rations of oversampling and under-sampling respectively were experimented viz. 40:60, 45:55 and 50:50. Figure 11 represents the code for sampling of majority and minority class using random sampling.

40:60 Sampling Ratio

```
## Oversample
oversample = RandomOverSampler(sampling_strategy=0.40)
x_train_ros_1, y_train_ros_1 = oversample.fit_sample(x_train, y_train)

## Undersample
undersample = RandomUnderSampler(sampling_strategy=0.60)
x_train_bd, y_train_bd= undersample.fit_sample(x_train_ros_1, y_train_ros_1)
```

45:55 Sampling Ratio

```
## Oversample
oversample = RandomOverSampler(sampling_strategy=0.45)
x_train_ros_1, y_train_ros_1 = oversample.fit_sample(x_train, y_train)

## Undersample
undersample = RandomUnderSampler(sampling_strategy=0.55)
x_train_bd, y_train_bd= undersample.fit_sample(x_train_ros_1, y_train_ros_1)
```

50:50 Sampling Ratio

```
## Oversample
oversample = RandomOverSampler(sampling_strategy=0.50)
x_train_ros_1, y_train_ros_1 = oversample.fit_sample(x_train, y_train)

## Undersample
undersample = RandomUnderSampler(sampling_strategy=0.50)
x_train_bd, y_train_bd= undersample.fit_sample(x_train_ros_1, y_train_ros_1)
```

Figure 11: Random Sampling of Train Set in Feature Selected Data

3.6 Model Implementation and Cross Validation

Various classification models viz. Decision Tree, Logistic Regression, XGBoost, Random Forest, SVM-Linear, SVM-RBF, Naïve Bayes, KNN and basic Neural Network were implemented on pre-processed and feature engineered data. Their performance was evaluated for precision and accuracy. The accuracy of each model was further cross validated using stratified K-fold validation.

3.6.1 Decision Tree

Braha and Shmilovici (2002) used Decision Tree (DT) in their research and achieved an accuracy of 77%. DT was implemented using default parameters and was cross validated using K-fold validation with folds ranging from 10 to 50. Code for DT is illustrated in Figure 12 whereas Figure 13 represents the K-Fold validation of DT.

Decision Tree and its Confusion Matrix

```
## Decision Tree
DT = DecisionTreeClassifier()

## Training DT
DT = DT.fit(x_train_fos, y_train_fos)

## Predicting response on Test
y_pred = DT.predict(x_test)

DT_A2 = metrics.accuracy_score(y_test, y_pred)*100
DT_R2 = metrics.recall_score(y_test, y_pred)*100
DT_P2 = metrics.precision_score(y_test, y_pred)*100
DT_F2 = metrics.f1_score(y_test, y_pred)*100

print('Accuracy: %.2f%% ' % DT_A2)
print("Recall_Accuracy: %.2f%%" % DT_R2)
print("Precision_Accuracy: %.2f%%" % DT_P2)
print("F1 Score: %.2f%%" % DT_F2 )
```

```
Accuracy: 96.98%
Recall_Accuracy: 87.50%
Precision_Accuracy: 83.76%
F1 Score: 85.59%
```

```
cm = confusion_matrix(y_test, y_pred)
TP_DT_2 = cm[1][1]

plt.rcParams['figure.figsize'] = (5, 5)
#sns.set(style = 'dark', font_scale = 1.4)
sns.heatmap(cm, annot = True, annot_kws = {"size": 15})

print(confusion_matrix(y_test, y_pred))
```

```
[[1921   38]
 [  28  196]]
```

Figure 12: Code for Decision Tree

Decision Tree with Stratified K-Fold

```
## Decision Tree for k = 10
skfold = StratifiedKFold(n_splits = 10,random_state=None)
model_skfold = DecisionTreeClassifier()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
DT_1 = (results_skfold.mean()*100)
print("Accuracy when k is 10 : %.2f%%" % DT_1)

## Decision Tree for k = 20
skfold = StratifiedKFold(n_splits = 20,random_state=None)
model_skfold = DecisionTreeClassifier()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
DT_2 = (results_skfold.mean()*100)
print("Accuracy when k is 20 : %.2f%%" % DT_2)

## Decision Tree for K = 30
skfold = StratifiedKFold(n_splits = 30,random_state=None)
model_skfold = DecisionTreeClassifier()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
DT_3 = (results_skfold.mean()*100)
print("Accuracy when k is 30 : %.2f%%" % DT_3)

## Decision Tree for K = 50
skfold = StratifiedKFold(n_splits = 50,random_state=None)
model_skfold = DecisionTreeClassifier()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
DT_4 = (results_skfold.mean()*100)
print("Accuracy when k is 50 : %.2f%%" % DT_4)
```

```
Accuracy when k is 10 : 92.53%
Accuracy when k is 20 : 95.41%
Accuracy when k is 30 : 95.55%
Accuracy when k is 50 : 96.47%
```

Figure 13: K-Fold Validation of Decision Tree

3.6.2 Logistic Regression

Logistic Regression was implemented using default parameters and was cross validated using K-fold validation with folds ranging from 10 to 50. Code for Logistic Regression is illustrated in Figure 14 whereas Figure 15 represents its K-Fold validation.

Logistic Regression & its Confusion Matrix

```
## LOgistic Regression
logreg = LogisticRegression(random_state= 0)

## Training Model
LogReg = logreg.fit(x_train_fos, y_train_fos)

## Predicting response
y_predLog = LogReg.predict(x_test)

LogReg_A2 = metrics.accuracy_score(y_test, y_predLog)*100
LogReg_R2 = metrics.recall_score(y_test, y_predLog)*100
LogReg_P2 = metrics.precision_score(y_test, y_predLog)*100
LogReg_F2 = metrics.f1_score(y_test, y_predLog)*100

print("Accuracy: %.2f%%" % LogReg_A2)
print("Recall_Accuracy: %.2f%%" % LogReg_R2)
print("Precision_Accuracy: %.2f%%" % LogReg_P2)
print("F1 Score: %.2f%%" % LogReg_F2)
```

```
cm = confusion_matrix(y_test, y_predLog)
TP_LogReg_2 = cm[1][1]

plt.rcParams['figure.figsize'] = (5, 5)
#sns.set(style = 'dark', font_scale = 1.4)
sns.heatmap(cm, annot = True, annot_kws = {"size": 15})

print(confusion_matrix(y_test, y_predLog))
```

Figure 14: Code for Logistic Regression

Logistic Regression with Stratified K-Fold

```
## Logistic Regression for k = 10
skfold = StratifiedKFold(n_splits = 10,random_state=None)
model_skfold = LogisticRegression()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
LogReg_1 = (results_skfold.mean()*100)
print("Accuracy when k is 10 : %.2f%%" % LogReg_1)

## Logistic Regression for k = 20
skfold = StratifiedKFold(n_splits = 20,random_state=None)
model_skfold = LogisticRegression()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
LogReg_2 = (results_skfold.mean()*100)
print("Accuracy when k is 20 : %.2f%%" % LogReg_2)

## Logistic Regression for K = 30
skfold = StratifiedKFold(n_splits = 30,random_state=None)
model_skfold = LogisticRegression()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
LogReg_3 = (results_skfold.mean()*100)
print("Accuracy when k is 30 : %.2f%%" % LogReg_3)

## Logistic Regression for K = 50
skfold = StratifiedKFold(n_splits = 50,random_state=None)
model_skfold = LogisticRegression()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
LogReg_4 = (results_skfold.mean()*100)
print("Accuracy when k is 50 : %.2f%%" % LogReg_4)
```

```
C:\Users\Omkar Doke\Anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:940: ConvergenceWarning: lbfgs failed to c
onverge (status=1):
```

Figure 15: K-Fold Validation of Logistic Regression

Model's precision failure was further studied for its threshold for classification of its probabilities. It was then adjusted after plotting the histogram plot and the model was re-implemented which saw further reduction in precision. Figure 16 shows code for experiment with logistic regression.

Understanding why True Positive is less and how to adjust the threshold of classification for that

```
# print the first 25 true and predicted responses
print('True:', y_test.values[0:100])
print('Predicted:', y_predLog[0:100])
```

```
# print the first 10 predicted probabilities of class membership
LogReg.predict_proba(x_test)[79:100]

# print the first 10 predicted probabilities for class 1
LogReg.predict_proba(x_test)[0:10, 1]

# store the predicted probabilities for class 1
y_pred_prob = LogReg.predict_proba(x_test)[:, 1]
y_pred_prob_df = pd.DataFrame(y_pred_prob)
```

```
# histogram of predicted probabilities
# 8 bins
plt.hist(y_pred_prob, bins=5)

# x-axis limit from 0 to 1
plt.xlim(0,1)
plt.title('Histogram of predicted probabilities')
plt.xlabel('Predicted probability of wafer test')
plt.ylabel('Frequency')
```

```
# predict wafer fail if the predicted probability is greater than 0.4, it will return 1 for all values above 0.2 and 0 other
# results are 2D so we slice out the first column

y_pred_class = pd.DataFrame(binarize(y_pred_prob_df, 0.2))

LogReg_B_A1 = metrics.accuracy_score(y_test, y_pred_class)*100
LogReg_B_R1 = metrics.recall_score(y_test, y_pred_class)*100
LogReg_B_P1 = metrics.precision_score(y_test, y_pred_class)*100
LogReg_B_F1 = metrics.f1_score(y_test, y_pred_class)*100

print("Accuracy: %.2f%%" % LogReg_B_A1)
print("Recall_Accuracy: %.2f%%" % LogReg_B_R1)
print("Precision_Accuracy: %.2f%%" % LogReg_B_P1)
print("F1 Score: %.2f%%" % LogReg_B_F1)
```

```
m = confusion_matrix(y_test, y_pred_class)
TP_LogReg_B = cm[1][1]

plt.rcParams['figure.figsize'] = (5, 5)
#sns.set(style = 'dark', font_scale = 1.4)
sns.heatmap(cm, annot = True, annot_kws = {"size": 15})

print(confusion_matrix(y_test, y_pred_class))
```

Figure 16: Understanding Poor Performance of Logistic Regression

3.6.3 XGBoost

XGBoost was implemented using default parameters and was cross validated using K-fold validation with folds ranging from 10 to 50. Code for XGBoost is illustrated in Figure 17 whereas Figure 18 represents its K-Fold validation.

XGB Classifier and its CM

```
## XGB Boost
XGB = XGBClassifier()

## Training Model
XGB = XGB.fit(x_train_fos, y_train_fos)

## Predicting response on Test
y_pred = XGB.predict(x_test)

XGB_A2 = metrics.accuracy_score(y_test, y_pred)*100
XGB_R2 = metrics.recall_score(y_test, y_pred)*100
XGB_P2 = metrics.precision_score(y_test, y_pred)*100
XGB_F2 = metrics.f1_score(y_test, y_pred)*100

print("Accuracy: %.2f%%" % XGB_A2)
print("Recall_Accuracy: %.2f%%" % XGB_R2)
print("Precision_Accuracy: %.2f%%" % XGB_P2)
print("F1 Score: %.2f%%" %XGB_F2)
```

```
Accuracy: 98.44%
Recall_Accuracy: 88.84%
Precision_Accuracy: 95.67%
F1 Score: 92.13%
```

```
cm = confusion_matrix(y_test, y_pred)
TP_XGB_2 = cm[1][1]

plt.rcParams['figure.figsize'] = (5, 5)
#sns.set(style = 'dark', font_scale = 1.4)
sns.heatmap(cm, annot = True, annot_kws = {"size": 15})

print(confusion_matrix(y_test, y_pred))
```

```
[[1950    9]
 [  25  199]]
```

Figure 17: Code for XGBoost Classifier

XGB with Stratified K-Fold

```
## XGB for k = 10
skfold = StratifiedKFold(n_splits = 10,random_state=None)
model_skfold = XGBClassifier()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
XGB_1 = (results_skfold.mean()*100)
print("Accuracy when k is 10 : %.2f%%" % XGB_1)

## XGB for k = 20
skfold = StratifiedKFold(n_splits = 20,random_state=None)
model_skfold = XGBClassifier()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
XGB_2 = (results_skfold.mean()*100)
print("Accuracy when k is 20 : %.2f%%" % XGB_2)

## XGB for K = 30
skfold = StratifiedKFold(n_splits = 30,random_state=None)
model_skfold = XGBClassifier()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
XGB_3 = (results_skfold.mean()*100)
print("Accuracy when k is 30 : %.2f%%" % XGB_3)

## XGB for K = 50
skfold = StratifiedKFold(n_splits = 50,random_state=None)
model_skfold = XGBClassifier()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
XGB_4 = (results_skfold.mean()*100)
print("Accuracy when k is 50 : %.2f%%" % XGB_4)
```

```
Accuracy when k is 10 : 92.78%
Accuracy when k is 20 : 97.17%
Accuracy when k is 30 : 98.05%
Accuracy when k is 50 : 98.27%
```

Figure 18: K-Fold Validation of XGBoost

3.6.4 Random Forest

Random Forest was implemented using default parameters and was cross validated using K-fold validation with folds ranging from 10 to 50. Code for RF is illustrated in Figure 19 whereas Figure 20 represents its K-Fold validation.

Random Forest and its CM

```
## Random Forest
RF = RandomForestClassifier()

## Training Model
RF = RF.fit(x_train_fos, y_train_fos)

## Predicting response on Test
y_pred = RF.predict(x_test)

RF_A2 = metrics.accuracy_score(y_test, y_pred)*100
RF_R2 = metrics.recall_score(y_test, y_pred)*100
RF_P2 = metrics.precision_score(y_test, y_pred)*100
RF_F2 = metrics.f1_score(y_test, y_pred)*100

print("Accuracy: %.2f%%" % RF_A2)
print("Recall_Accuracy: %.2f%%" % RF_R2)
print("Precision_Accuracy: %.2f%%" % RF_P2)
print("F1 Score: %.2f%%" % RF_F2)
```

```
Accuracy: 98.40%
Recall_Accuracy: 87.50%
Precision_Accuracy: 96.55%
F1 Score: 91.80%
```

```
cm = confusion_matrix(y_test, y_pred)
TP_RF_2 = cm[1][1]

plt.rcParams['figure.figsize'] = (5, 5)
#sns.set(style = 'dark', font_scale = 1.4)
sns.heatmap(cm, annot = True, annot_kws = {"size": 15})

print(confusion_matrix(y_test, y_pred))
```

```
[[1952    7]
 [  28  196]]
```

Figure 19: Code for Random Forest

Random Forest with Stratified K-Fold

```
## Random Forest for k = 10
skfold = StratifiedKFold(n_splits = 10,random_state=None)
model_skfold = RandomForestClassifier()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
RF_1 = (results_skfold.mean()*100)
print("Accuracy when k is 10 : %.2f%%" % RF_1)

## Random Forest for k = 20
skfold = StratifiedKFold(n_splits = 20,random_state=None)
model_skfold = RandomForestClassifier()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
RF_2 = (results_skfold.mean()*100)
print("Accuracy when k is 20 : %.2f%%" % RF_2)

## Random Forest for K = 30
skfold = StratifiedKFold(n_splits = 30,random_state=None)
model_skfold = RandomForestClassifier()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
RF_3 = (results_skfold.mean()*100)
print("Accuracy when k is 30 : %.2f%%" % RF_3)

## Random Forest for K = 50
skfold = StratifiedKFold(n_splits = 50,random_state=None)
model_skfold = RandomForestClassifier()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
RF_4 = (results_skfold.mean()*100)
print("Accuracy when k is 50 : %.2f%%" % RF_4)
```

```
Accuracy when k is 10 : 94.01%
Accuracy when k is 20 : 97.50%
Accuracy when k is 30 : 98.14%
Accuracy when k is 50 : 98.28%
```

Figure 20: K-Fold Validation of Random Forest

3.6.5 SVM-Linear

Yu et al. (2017) used SVM-Linear in their research and achieved a F1 Score of 90%. SVM was implemented with 'Linear' Kernel using default parameters and was cross validated using K-fold validation with folds ranging from 10 to 50. Code for SVM-Linear is illustrated in Figure 21 whereas Figure 22 represents its K-Fold validation.

SVM-Linear and its Confusion Matrix

```
## SVM Linear
SVM= svm.SVC(kernel = 'linear')

## Training Model
SVM = SVM.fit(x_train_fos, y_train_fos)

## Predicting response on Test
y_pred = SVM.predict(x_test)

SVM_A2 = metrics.accuracy_score(y_test, y_pred)*100
SVM_R2 = metrics.recall_score(y_test, y_pred)*100
SVM_P2 = metrics.precision_score(y_test, y_pred)*100
SVM_F2 = metrics.f1_score(y_test, y_pred)*100

print("Accuracy: %.2f%%" % SVM_A2)
print("Recall_Accuracy: %.2f%%" % SVM_R2)
print("Precision_Accuracy: %.2f%%" % SVM_P2)
print("F1 Score: %.2f%%" % SVM_F2)
```

```
Accuracy: 92.81%
Recall_Accuracy: 58.93%
Precision_Accuracy: 67.01%
F1 Score: 62.71%
```

```
cm = confusion_matrix(y_test, y_pred)
TP_SVM_2 = cm[1][1]

plt.rcParams['figure.figsize'] = (5, 5)
#sns.set(style = 'dark', font_scale = 1.4)
sns.heatmap(cm, annot = True, annot_kws = {"size": 15})

print(confusion_matrix(y_test, y_pred))
```

```
[[1894   65]
 [  92  132]]
```

Figure 21: Code for SVM-Linear

SVM Linear with Stratified K-Fold

```
## SVM Linear for k = 10
skfold = StratifiedKFold(n_splits = 10,random_state=None)
model_skfold = svm.SVC(kernel= 'linear')
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
SVM_1 = (results_skfold.mean()*100)
print("Accuracy when k is 10 : %.2f%%" % SVM_1)

## SVM Linear for k = 20
skfold = StratifiedKFold(n_splits = 20,random_state=None)
model_skfold = svm.SVC(kernel= 'linear')
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
SVM_2 = (results_skfold.mean()*100)
print("Accuracy when k is 20 : %.2f%%" % SVM_2)

## SVM Linear for K = 30
skfold = StratifiedKFold(n_splits = 30,random_state=None)
model_skfold = svm.SVC(kernel= 'linear')
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
SVM_3 = (results_skfold.mean()*100)
print("Accuracy when k is 30 : %.2f%%" % SVM_3)

## SVM Linear for K = 50
skfold = StratifiedKFold(n_splits = 50,random_state=None)
model_skfold = svm.SVC(kernel= 'linear')
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
SVM_4 = (results_skfold.mean()*100)
print("Accuracy when k is 50 : %.2f%%" % SVM_4)
```

```
Accuracy when k is 10 : 77.85%
Accuracy when k is 20 : 82.89%
Accuracy when k is 30 : 83.95%
Accuracy when k is 50 : 84.39%
```

Figure 22: K-Fold Validation of SVM-Linear

3.6.6 SVM-RBF

Adly et al. (2015) used SVM-RBF in their research and achieved an accuracy of 87.5%. SVM was implemented with 'Radial Basis Function' Kernel using default parameters and was cross validated using K-fold validation with folds ranging from 10 to 50. Code for SVM-RBF is illustrated in Figure 23 whereas Figure 24 represents its K-Fold validation.

SVM-RBF and its COnfusion Matrix

```
## SVM RBF
SVM= svm.SVC(kernel = 'rbf')

## Training Model
SVM = SVM.fit(x_train_fos, y_train_fos)

## Predicting response on Test
y_pred = SVM.predict(x_test)

SVM_RBF_A2 = metrics.accuracy_score(y_test, y_pred)*100
SVM_RBF_R2 = metrics.recall_score(y_test, y_pred)*100
SVM_RBF_P2 = metrics.precision_score(y_test, y_pred)*100
SVM_RBF_F2 = metrics.f1_score(y_test, y_pred)*100

print("Accuracy: %.2f%%" % SVM_RBF_A2)
print("Recall_Accuracy: %.2f%%" % SVM_RBF_R2)
print("Precision_Accuracy: %.2f%%" % SVM_RBF_P2)
print("F1 Score: %.2f%%" % SVM_RBF_F2)
```

```
Accuracy: 97.76%
Recall_Accuracy: 87.50%
Precision_Accuracy: 90.32%
F1 Score: 88.89%
```

```
cm = confusion_matrix(y_test, y_pred)
TP_SVM_RBF_2 = cm[1][1]

plt.rcParams['figure.figsize'] = (5, 5)
#sns.set(style = 'dark', font_scale = 1.4)
sns.heatmap(cm, annot = True, annot_kws = {"size": 15})

print(confusion_matrix(y_test, y_pred))
```

```
[[1938   21]
 [  28  196]]
```

Figure 23: Code for SVM-RBF

SVM RBF with Stratified K-Fold

```
## SVM RBF for k = 10
skfold = StratifiedKFold(n_splits = 10,random_state=None)
model_skfold = svm.SVC(kernel= 'rbf')
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
SVM_RBF_1 = (results_skfold.mean()*100)
print("Accuracy when k is 10 : %.2f%%" % SVM_RBF_1)

## SVM RBF for k = 20
skfold = StratifiedKFold(n_splits = 20,random_state=None)
model_skfold = svm.SVC(kernel= 'rbf')
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
SVM_RBF_2 = (results_skfold.mean()*100)
print("Accuracy when k is 20 : %.2f%%" % SVM_RBF_2)

## SVM RBF for K = 30
skfold = StratifiedKFold(n_splits = 30,random_state=None)
model_skfold = svm.SVC(kernel= 'rbf')
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
SVM_RBF_3 = (results_skfold.mean()*100)
print("Accuracy when k is 30 : %.2f%%" % SVM_RBF_3)

## SVM RBF for K = 50
skfold = StratifiedKFold(n_splits = 50,random_state=None)
model_skfold = svm.SVC(kernel= 'rbf')
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
SVM_RBF_4 = (results_skfold.mean()*100)
print("Accuracy when k is 50 : %.2f%%" % SVM_RBF_4)
```

```
Accuracy when k is 10 : 88.88%
Accuracy when k is 20 : 93.48%
Accuracy when k is 30 : 94.63%
Accuracy when k is 50 : 95.11%
```

Figure 24: K-Fold Validation of SVM-RBF

3.6.7 Naive Bayes

Naïve Bayes was implemented using default parameters and was cross validated using K-fold validation with folds ranging from 10 to 50. Code for NB is illustrated in Figure 25 whereas Figure 26 represents its K-Fold validation.

Naive Bayes and its Confusion Matrix

```
## Naive Bayes
NB = GaussianNB()

## Training Model
NB = NB.fit(x_train_fos, y_train_fos)

## Predicting response on Test
y_pred = NB.predict(x_test)

NB_A2 = metrics.accuracy_score(y_test, y_pred)*100
NB_R2 = metrics.recall_score(y_test, y_pred)*100
NB_P2 = metrics.precision_score(y_test, y_pred)*100
NB_F2 = metrics.f1_score(y_test, y_pred)*100

print("Accuracy: %.2f%%" % NB_A2)
print("Recall_Accuracy: %.2f%%" % NB_R2)
print("Precision_Accuracy: %.2f%%" % NB_P2)
print("F1 Score: %.2f%%" % NB_F2)
```

```
Accuracy: 72.47%
Recall_Accuracy: 58.04%
Precision_Accuracy: 20.41%
F1 Score: 30.20%
```

```
cm = confusion_matrix(y_test, y_pred)
TP_NB_2 = cm[1][1]

plt.rcParams['figure.figsize'] = (5, 5)
#sns.set(style = 'dark', font_scale = 1.4)
sns.heatmap(cm, annot = True, annot_kws = {"size": 15})

print(confusion_matrix(y_test, y_pred))
```

```
[[1452  507]
 [  94  130]]
```

Figure 25: Code for Naïve Bayes

Naive Bayes with Stratified K-Fold

```
## Naive Bayes for k = 10
skfold = StratifiedKFold(n_splits = 10,random_state=None)
model_skfold = GaussianNB()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
NB_1 = (results_skfold.mean()*100)
print("Accuracy when k is 10 : %.2f%%" % NB_1)

## Naive Bayes for k = 20
skfold = StratifiedKFold(n_splits = 20,random_state=None)
model_skfold = GaussianNB()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
NB_2 = (results_skfold.mean()*100)
print("Accuracy when k is 20 : %.2f%%" % NB_2)

## Naive Bayes for K = 30
skfold = StratifiedKFold(n_splits = 30,random_state=None)
model_skfold = GaussianNB()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
NB_3 = (results_skfold.mean()*100)
print("Accuracy when k is 30 : %.2f%%" % NB_3)

## Naive Bayes for K = 50
skfold = StratifiedKFold(n_splits = 50,random_state=None)
model_skfold = GaussianNB()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
NB_4 = (results_skfold.mean()*100)
print("Accuracy when k is 50 : %.2f%%" % NB_4)
```

```
Accuracy when k is 10 : 58.03%
Accuracy when k is 20 : 61.88%
Accuracy when k is 30 : 65.34%
Accuracy when k is 50 : 67.67%
```

Figure 26: K-Fold Validation of Naïve Bayes

3.6.8 KNN

Chien et al. (2012) used KNN in their research and achieved an accuracy of 75%. KNN was implemented using default parameters and was cross validated using K-fold validation with folds ranging from 10 to 50. Code for KNN is illustrated in Figure 27 whereas Figure 28 represents its K-Fold validation.

KNN and its Confusion Matrix

```
## KNN
KNN = KNeighborsClassifier()

## Training Model
KNN = KNN.fit(x_train_fos, y_train_fos)

## Predicting response on Test
y_pred = KNN.predict(x_test)

KNN_A2 = metrics.accuracy_score(y_test, y_pred)*100
KNN_R2 = metrics.recall_score(y_test, y_pred)*100
KNN_P2 = metrics.precision_score(y_test, y_pred)*100
KNN_F2 = metrics.f1_score(y_test, y_pred)*100

print("Accuracy: %.2f%%" % KNN_A2)
print("Recall_Accuracy: %.2f%%" % KNN_R2)
print("Precision_Accuracy: %.2f%%" % KNN_P2)
print("F1 Score: %.2f%%" % KNN_F2)
```

```
Accuracy: 94.96%
Recall_Accuracy: 89.29%
Precision_Accuracy: 69.93%
F1 Score: 78.43%
```

```
cm = confusion_matrix(y_test, y_pred)
TP_KNN_2 = cm[1][1]

plt.rcParams['figure.figsize'] = (5, 5)
#sns.set(style = 'dark', font_scale = 1.4)
sns.heatmap(cm, annot = True, annot_kws = {"size": 15})

print(confusion_matrix(y_test, y_pred))
```

```
[[1873   86]
 [  24  200]]
```

Figure 27: Code for KNN

KNN with Stratified K-Fold

```
## KNN for k = 10
skfold = StratifiedKFold(n_splits = 10,random_state=None)
model_skfold = KNeighborsClassifier()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
KNN_1 = (results_skfold.mean()*100)
print("Accuracy when k is 10 : %.2f%%" % KNN_1)

## KNN for k = 20
skfold = StratifiedKFold(n_splits = 20,random_state=None)
model_skfold = KNeighborsClassifier()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
KNN_2 = (results_skfold.mean()*100)
print("Accuracy when k is 20 : %.2f%%" % KNN_2)

## KNN for K = 30
skfold = StratifiedKFold(n_splits = 30,random_state=None)
model_skfold = KNeighborsClassifier()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
KNN_3 = (results_skfold.mean()*100)
print("Accuracy when k is 30 : %.2f%%" % KNN_3)

## KNN for K = 50
skfold = StratifiedKFold(n_splits = 50,random_state=None)
model_skfold = KNeighborsClassifier()
results_skfold = cross_val_score(model_skfold, a_fos, b_fos, cv = skfold)
KNN_4 = (results_skfold.mean()*100)
print("Accuracy when k is 50 : %.2f%%" % KNN_4)
```

```
Accuracy when k is 10 : 91.12%
Accuracy when k is 20 : 93.89%
Accuracy when k is 30 : 94.84%
Accuracy when k is 50 : 95.17%
```

Figure 28: K-Fold Validation of KNN

3.6.9 Neural Network

Fernandes et al. (2020) used KNN in their research and achieved an accuracy of 89.64%. Basic Neural Network was designed and implemented. It was tested for epochs 25 and 50 with constant batch size of 60. Code for design, training and implementation of NN is illustrated in Figure 29.

Neural Network & its Confusion Matrix

```
##Neural Network
NN = Sequential()
NN.add(Dense(51, input_dim = 100, activation = 'relu'))
NN.add(Dense(27, activation = 'relu'))
NN.add(Dense(15, activation = 'relu'))
NN.add(Dense(9, activation = 'relu'))
NN.add(Dense(6, activation = 'relu'))
NN.add(Dense(2, activation = 'sigmoid'))
NN.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
## Training Neural Network
Neural_N3 = NN.fit(x_train_fos, y_train_fos, epochs=25, batch_size= 60)
```

```
## Testing Neural Network
y_pred = NN.predict(x_test)

#Converting predictions to label
pred = list()
for i in range(len(y_pred)):
    pred.append(np.argmax(y_pred[i]))
NN_3 = accuracy_score(pred,y_test)*100
print('Accuracy is: %.2f%%' % NN_3)
```

```
## Confusion Matrix for Neural Network
cm = confusion_matrix(y_test, pred)
TP_NN_3 = cm[1][1]
plt.rcParams['figure.figsize'] = (5, 5)
#sns.set(style = 'dark', font_scale = 1.4)
sns.heatmap(cm, annot = True, annot_kws = {"size": 15})
print(confusion_matrix(y_test, pred))
```

Figure 29: Code for Neural Network

References

Adly, F., Yoo, P. D., Muhaidat, S., Al-Hammadi, Y., Lee, U. and Ismail, M. (2015). Randomized general regression network for identification of defect patterns in semiconductor wafer maps, *IEEE Transactions on Semiconductor Manufacturing* **28**(2): 145–152.

Braha, D. and Shmilovici, A. (2002). Data mining for improving a cleaning process in the semiconductor industry, *IEEE Transactions on Semiconductor Manufacturing* **15**(1): 91–101.

Chien, C.-F., Hsu, C.-Y. and Chen, P.-N. (2012). Semiconductor fault detection and classification for yield enhancement and manufacturing intelligence, *Flexible Services and Manufacturing Journal* **25**.

Fernandes, S., Antunes, M., Santiago, A., Barraca, J., Gomes, D. and Aguiar, R. (2020). Forecasting appliances failures: A machine-learning approach to predictive maintenance, *MDPI Journals* .

Yu, C., Chien, C. and Kuo, C. (2017). Exploit the value of production data to discover opportunities for saving power consumption of production tools, *IEEE Transactions on Semiconductor Manufacturing* **30**(4): 345–350.