CUSTOMER CHURN PREDICTION

Summary

Customer churn is a financial expression referring to losing of a customer or client when a customer abandons interaction with business or company. Churn rate is defined as the rate of customers leaving a company within a specific time duration. Low churn rate can be an indicator of success for companies so they try to retain as many customers as they can. With the advent of machine learning deep learning techniques, companies can identify potential customers who may stop doing business with them in the near future. Strategies can be applied to retain those costumers. In this notebook, customer churn is predicted for a bank based on different customer attributes including geography, age, gender, income and more. a wide range of predictive algorithms and various performance metrics were considered to achieve the most reliable classifier.

Python functions and data files to run this notebook are in my Github page.

```
import os
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from functions import* # import require functions to run this notebook
from IPython.display import HTML
import pickle
import logging
import matplotlib.patheffects as pe
import matplotlib.ticker as mtick
import warnings
warnings.filterwarnings('ignore
```

Table of Contents

Methodology

A binary classification will be applied Churn_Modelling.csv downloaded from Kaggle: the target feature is Exited (0 and 1). The following steps will be applied:

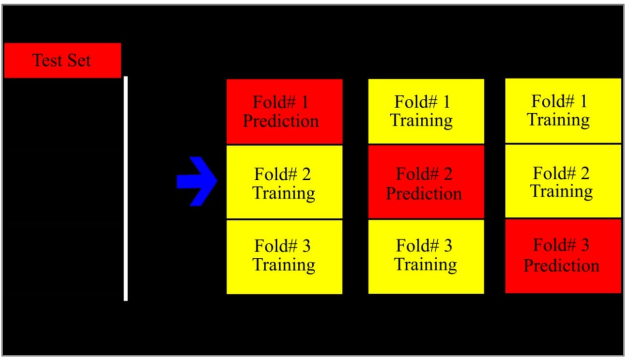
The irrelevant features are removed from data (we do not need customer's name, customer's ID). The linear correlation between features and target are calculated or run Random Forest analysis to rank the importance of features to predict the target.

The data set is divided into 80% training set and 20% test set. The test set is only applied at the end as a never-before-seen dataset to only evaluate the trained models and make sure the models are not overfitted and no information leaks occur (see Figure below).

drawing

Data processing is applied for training set including imputation of missing values, normalization and text handling. The data is clean up and get ready to feed ML algorithms.

Train ML algorithms mentioned above using the training set by implementing a binary classification. Each algorithm is optimized by fine-tuning the hyperparameters. To avoid overfitting, K-fold cross validation is applied. The training set (we do not touch test set) into 3-folds (k=3); each model is trained with 2 folds and apply prediction for 1-folds. The process is repeated for all 3 folds to apply prediction for all training set. This leads to prevent overfitting and have a clean prediction (See Figure below). Therefore, performance of the algorithms can be measured for the training set and compared with other since the same data is fed to each algorithms



The trained models are applied to predict test set to confirm the performances make sure there is no overfitting. The models have not seen the test sets during the training so the resulted performance for test set can be an indication of performance for future prediction of these models. The same data processing for training set should be applied for test set. The same statistics for imputation of training set should be applied for test set.

Data Processing

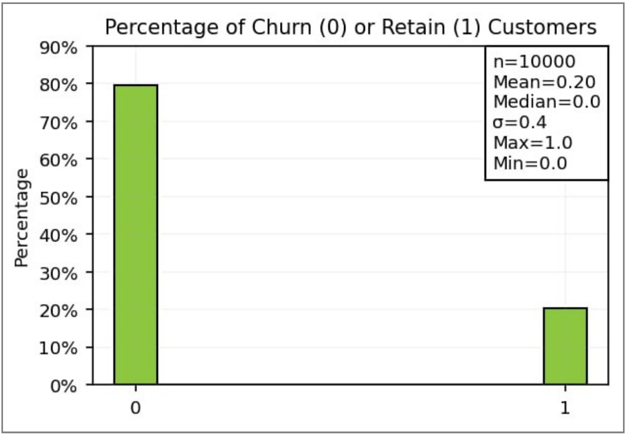
The data set Churn_Modelling.csv can be downloaded from Kaggle named.

```
# Read data 'Churn_Modelling.csv'

df = pd.read_csv('./Data/Churn_Modelling.csv')


# Shuffle the data

np.random.seed(32)

df=df.reindex(np.random.permutation(df.index))

df.reset_index(inplace=True, drop=True) # Reset
index


# Remove 'RowNumber','CustomerId','Surname'
features that are un
```

```
df=df.drop(['RowNumber','CustomerId','Surname'],axis=1,inplace=False)
```

Divide Data into Training set (80%) and Test set (20%)

```
font = {'size'   : 9.5}

plt.rc('font', **font)

fig = plt.subplots(figsize=(10, 3), dpi= 150, facecolor='w', edgecolor='k')

ax1=plt.subplot(1,2,1)

bins = np.array([-0.05,0.05,0.95,1.05])

EDA_plot.histplt (df['Exited'],bins=bins,title='Percentage of Churn (0) or Retain (1) Customers',xlabl="",

        ylabl='Percentage',xlimt=(-0.1,1.1),ylimt=(0,0.9),axt=ax1,x_tick=[0,1],

        scale=1.1,loc=1,font=8,color='#8DC63F')



plt.show()
```

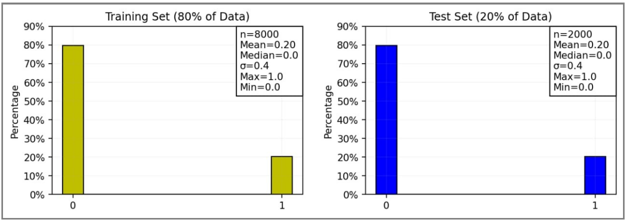



```
# Training and Test

spt = StratifiedShuffleSplit(n_splits=1, test_size=0.2,
random_state=42)

for train_idx, test_idx in spt.split(df, df['Exited']):

    train_set_strat = df.loc[train_idx]

    test_set_strat  = df.loc[test_idx]


train_set_strat.reset_index(inplace=True, drop=True)
# Reset index

test_set_strat.reset_index(inplace=True, drop=True) #
Reset index
```

```
font = {'size'   : 10}

plt.rc('font', **font)


fig = plt.subplots(figsize=(10, 3), dpi= 160,
facecolor='w', edgecolor='k')


ax1=plt.subplot(1,2,1)

val=train_set_strat['Exited']

EDA_plot.histplt(val,bins=bins,title="Training Set
(80% of Data)",xlabl="",

        ylabl='Percentage',xlimt=(-0.1,1.1),ylimt=
(0,0.9),axt=ax1,x_tick=[0,1],

        scale=1.2,loc=1,font=8,color='y')


ax2=plt.subplot(1,2,2)

val=test_set_strat['Exited']

EDA_plot.histplt(val,bins=bins,title="Test Set (20% of
Data)",xlabl="",

        ylabl='Percentage',xlimt=(-0.1,1.1),ylimt=
(0,0.9),axt=ax2,x_tick=[0,1],

        scale=1.2,loc=1,font=8,color='b')
```

plt.subplots_adjust(wspace=0.25)

plt.show()



Text Handeling

'Geography' and 'Gender' features are text that are converted to numbers via one-hot-encoding.

Convert Geography to one-hot-encoding

Geog_1hot=pd.get_dummies(train_set_strat['Geography'],prefix='Geography')

Convert gender to 0 and 1

ordinal_encoder = OrdinalEncoder()

train_set_strat['Gender'] = ordinal_encoder.fit_transform(train_set_strat[['Gender']])

Remove 'Geography'

train_set_strat=train_set_strat.drop(['Geography'],axis=1,inplace=False)

```
train_set_strat=pd.concat([Geog_1hot,train_set_strat], axis=1) # Concatenate rows

train_set_strat
```

	Geography_France	Geography_Germany	Geography_Spain	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
0	1	0	0	614	1.0	50	4	137104.47	1	1	0	127166.49	1
1	1	0	0	699	1.0	40	7	0.00	1	0	1	152876.13	1
2	1	0	0	666	1.0	46	5	123873.19	1	1	1	177844.06	0
3	1	0	0	759	0.0	33	2	0.00	2	1	0	56583.88	0
4	1	0	0	599	1.0	34	8	0.00	2	1	1	174196.68	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
7995	0	0	1	560	0.0	43	4	95140.44	2	1	0	123181.44	1
7996	0	0	1	850	1.0	33	3	100476.46	2	1	1	136539.13	0
7997	0	1	0	621	1.0	47	7	107363.29	1	1	1	66799.28	0

7998	1	0	0	654	0.0	24	8	145081.73	1	1	1	130075.07	0
7999	0	1	0	769	1.0	27	7	188614.07	1	1	0	171344.09	0

8000 rows × 13 columns

	Geography_France	Geography_Germany	Geography_Spain	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
0	1	0	0	614	1.0	50	4	137104.47	1	1	0	127166.49	1
1	1	0	0	699	1.0	40	7	0.00	1	0	1	152876.13	1
2	1	0	0	666	1.0	46	5	123873.19	1	1	1	177844.06	0
3	1	0	0	759	0.0	33	2	0.00	2	1	0	56583.88	0
4	1	0	0	599	1.0	34	8	0.00	2	1	1	174196.68	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
7995	0	0	1	560	0.0	43	4	95140.44	2	1	0	123181.44	1

7996 0 0 1 850 1.0 33 3 100476.46 2
1 1 136539.13 0

7997 0 1 0 621 1.0 47 7 107363.29 1
1 1 66799.28 0

7998 1 0 0 654 0.0 24 8 145081.73 1
1 1 130075.07 0

7999 0 1 0 769 1.0 27 7 188614.07 1
1 0 171344.09 0

8000 rows × 13 columns

Impute Missing Values

Check if there is missing values.

```
train_set_strat.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8000 entries, 0 to 7999
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Geography_France   8000 non-null   uint8
 1   Geography_Germany  8000 non-null   uint8
 2   Geography_Spain    8000 non-null   uint8
```

```
 3   CreditScore       8000 non-null   int64
 4   Gender            8000 non-null   float64
 5   Age               8000 non-null   int64
 6   Tenure            8000 non-null   int64
 7   Balance           8000 non-null   float64
 8   NumOfProducts     8000 non-null   int64
 9   HasCrCard         8000 non-null   int64
 10  IsActiveMember    8000 non-null   int64
 11  EstimatedSalary   8000 non-null   float64
 12  Exited            8000 non-null   int64
dtypes: float64(3), int64(7), uint8(3)
memory usage: 648.6 KB
```

There is no missing value.

```
# Correlation matrix
font = {'size'   : 11}
plt.rc('font', **font)
fig, ax=plt.subplots(figsize=(8, 8), dpi= 110, facecolor='w', edgecolor='k')

df_tmp=train_set_strat.drop(['Geography_France','Geography_Germany','Geography_Spain'], axis=1)
```

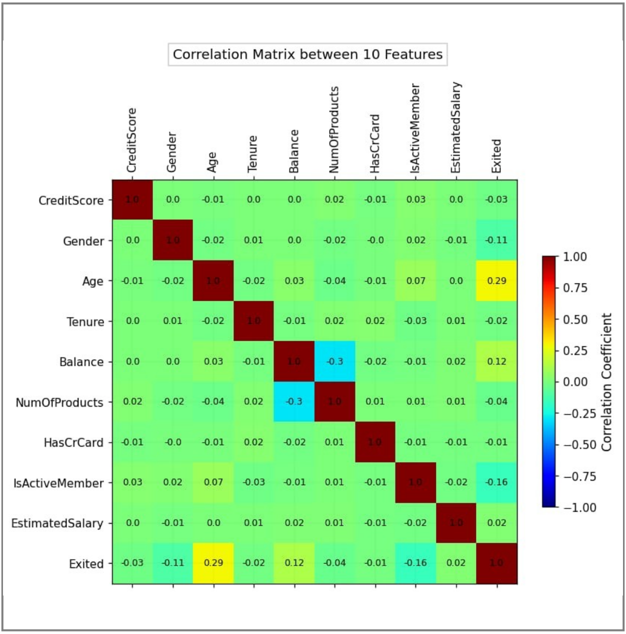
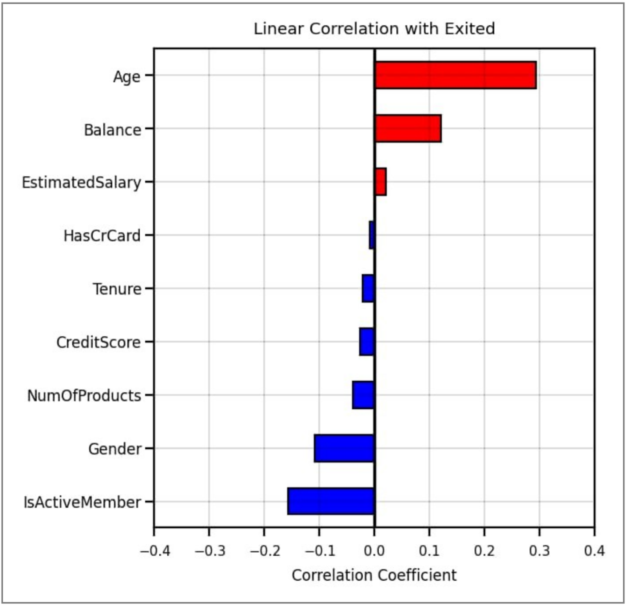


Figure below show correlation between training features and target feature (Exited). Age has positive correlation and IsActiveMember has negative correlation with target (Exited). Other Features are not linearly correlated with the target (Exited).

```
font = {'size'   : 5.5}

plt.rc('font', **font)

ax1 = plt.subplots(figsize=(3.2, 3.5), dpi= 200,
facecolor='w', edgecolor='k')


# Plot correlations of attributes with the last column

target='Exited'

corr=df_tmp.corr()

corr=corr[target].drop([target])

coefs=corr.values

clmns=list(corr.index)

Correlation_plot.corr_bar(coefs,clmns=clmns,yfontsiz
e=6.0,xlim=[-0.4,0.4],

                    title=f'Linear Correlation with
{target}',ymax_vert_lin=30)
```



Standardization

Make training features and target

X_train = train_set_strat.drop("Exited", axis=1)

y_train = train_set_strat["Exited"].values

Divide into two training sets (with and without standization)

clmn=
['Geography_France','Geography_Germany','Geograph
y_Spain',

```
 'Gender','NumOfProducts','HasCrCard','IsActiveMemb
er']

X_train_for_std = X_train.drop(clmn, axis=1)

X_train_not_std =X_train[clmn]

features_colums=list(X_train_for_std.columns)+list(X_
train_not_std.columns)

# Standadization

scaler = StandardScaler()

scaler.fit(X_train_for_std)

# Standardization for training set

df_train_std=scaler.transform(X_train_for_std)

fname = './Trained_Models/scaler.sav'

pickle.dump(scaler, open(fname, 'wb'))

X_train_std=np.concatenate((df_train_std,X_train_not_
std), axis=1)

X_train_std
```

```
array([[-0.37708143,  1.05758327, -0.35609222, ...,  1.
  ,
        1.        ,  0.        ],
       [ 0.50297627,  0.10215877,  0.68069212, ...,  1.        ,
        0.        ,  1.        ],
       [ 0.16130681,  0.67541347, -0.01049744, ...,  1.        ,
        1.        ,  1.        ],
       ...,
       [-0.30460609,  0.77095592,  0.68069212, ...,  1.        ,
        1.        ,  1.        ],
       [ 0.03706337, -1.42652045,  1.0262869 , ...,  1.        ,
        1.        ,  1.        ],
       [ 1.22772967, -1.13989309,  0.68069212, ...,  1.        ,
        1.        ,  0.        ]])
```

```
# Divide training data into smaller training and
validation


# Training set

Training_c=np.concatenate((X_train_std,y_train.resha
pe(-1,1)),axis=1)

Smaller_Training, Validation =
train_test_split(Training_c, test_size=0.2,
random_state=100)
```

Smaller_Training_Target=Smaller_Training[:,-1]

Smaller_Training=Smaller_Training[:,:-1]

Validation set

Validation_Target=Validation[:,-1]

Validation=Validation[:,:-1]

Number of predictors and make empty arrays

predictor= ['Dummy Classifier','Stocastic Gradient Descent','Logistic Regression',

'Support Vector Machine','Decision Trees','Adaptive Gradient Boosting',

'Random Forest','Neural Network']

filename=["" for x in range(len(predictor))]

Creat Empty List for Avergae Accuracy for each regressor

Predict_training=np.zeros(len(predictor))

pre_prob=[]

Train_Accu=np.zeros(len(predictor)); Train_Rec=np.zeros(len(predictor));Train_Pre=np.zeros(len(predictor));Train_Spe=np.zeros(len(predictor))

Test_Accu=np.zeros(len(predictor

Predictive Models

A wide range of machine learning algorithms from basic to advanced methods. A brief explanation of each is as follows:

Stochastic Gradient Descent (SGD) is a good place to start. Gradient descent is a general idea to minimize a cost function by iteratively tweaking parameters. It has a wide range of application to find optimal solutions for many problems. It computes the gradient of the cost function regarding model parameters and updates the parameters through iteration until a global minimum of cost function is reached. Gradient Descent (and mini-batch) can be very slow for large data set: Stochastic Gradient Descent is efficient because it just picks a random instance at every iteration and computes the gradients based only on that single instance.

Logistic Regression (LR) is a simple approach to estimate the probability of a particular class. It calculates a weighted sum of the input features (plus a bias term) and use sigmoid function to estimate probability of each class. It has been commonly used for medical science and health and failure detection in engineering.

Support Vector Machine (SVM) is a powerful algorithm to perform linear or nonlinear classification.It is highly preferred since less computation power is required to produce reliable accuracy.The fundamental idea is to have the largest possible margin between the classes. It predicts theclass of a new instance by computing a decision function with optimum parameters.

Decision Trees (DT) is a reliable ML algorithms and has been widely applied to solve complex and non-linear problems for both classification and regression. It is the fundamental component of Random Forest: it is applied based on a flowchart structure in which each node denotes a test, each branch represents the result of the test, and each leaf node represents a class label.

Random Forest (RF) is among the most versatile and reliable machine learning algorithms for non-linear and complex data. The RF randomly creates and merges multiple decision trees and predicts a class that gets the most votes among all trees. Despite its simplicity, RF is one of the most powerful ML algorithms available today.

Adaptive Boosting (AB) can be applied for any predictor mentioned above to enhance the performance and turn into a stronger learner. The general idea is to use a base classifier and apply on training set first, then corrects base classifier by paying attention to the training instances that are underfitted. This leads to a new classifier focusing more on the hard cases. The process of training a classifier repeated sequentially, each classifier trying to correct its predecessor.

Neural Network is a specific subfield of machine learning for tackling a very complex problem. In comparison with Shallow Neural Network, Deep Neural Network usually involves much more successive layers of representations that are learned from training data. The large network's architecture may lead to some problems such as vanishing/exploding gradients, overfitting, computational cost and slow training. However, these problems can be resolved by tuning some hyperparameters. Deep Neural Network has been recently applied in many disciplines including computer vision, speech recognition, natural language processing, audio recognition and so on.

Ensemble Learning usually applies near the end of project when a few good and promising predictors are built to integrate them into an even stronger predictor. It works by aggregating the predictions of a group of predictors. RF and AB can be categorized as

EL. A very simple EL is hard voting (HV) that aggregates the predictions of each classifier and predicts the class that gets the most votes. Soft voting (SV) is another EL that works by averaging the probability of

each class and predict a class with the highest probability. It often achieves higher performance than HV due to giving more weight to highly confident votes. Instead of using simple functions to aggregate the predictions of all predictors, staking approach (SA) trains a model to perform this aggregation. The final predictor takes these predictions as inputs and makes the final prediction. EL may lead to even better prediction than the best individual if predictors are independent from each other (Géron 2019).

The performance of all seven algorithms are measured for both training set and test set compared with a Dummy Classifier for sanity check. This classifier is useful as a simple baseline to compare with other (real) classifiers. The ML algorithms for this work are retrieved from Scikit-Learn and Tensorflow Libraries.

Performance Measurement

The most common approach for assessment of classification is accuracy, which is calculated by number of true predicted over total number of data. However, accuracy alone may not be practical for performance measurement of classifiers, especially in cease of skewed datasets. Accuracy should be considered along with other metrics. Confusion matrix is a much better way to evaluate the performance of a classifier. The general idea is to consider the number of times instances of negative class are misclassified as positive class and vice versa. Three more metrics Sensitivity, Precision and Specificity can be calculated as well as Accuracy:

Accuracy=$(TP+TN)/(TP+TN+FP+FN)$: Accuracy is simply the fraction of the total samples that is correctly identified.

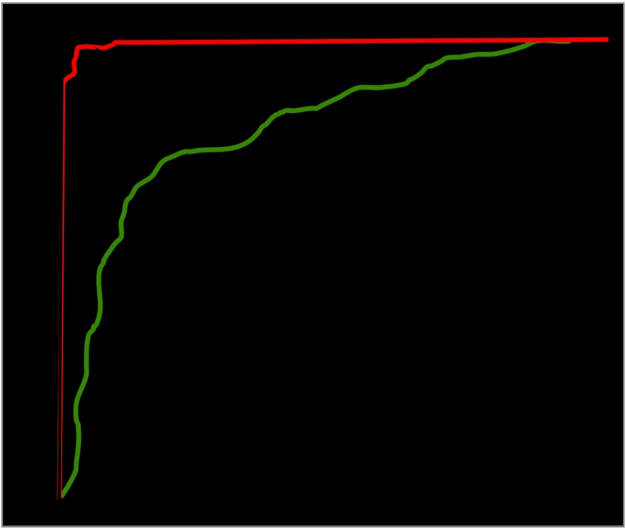
Sensitivity (Recall)= $TP/(TP+FN)$: Sensitivity is the proportion of correct positive predictions to the total positive classes.

Precision= $TP/(TP+FP)$: Precision is the proportion of correct positive prediction to the total predicted values.

Specificity= $TN/(TN+FP)$ Specificity is the true negative rate or the proportion of negatives that are correctly identified.

TP: True Positives, FP: False Positives, FN: False Negatives, TN: True Negatives

Another common approach to measure performance is the receiver operating characteristic (ROC). The ROC curve plots the true positive rate (Sensitivity) against the false positive rate (1-Specificity). Every point on the ROC curve represents a chosen cut-off even though it cannot be seen. For more information and details see ROC. The most common way to compare classifiers is to measure the area under the curve (AUC). A perfect classifier will have a ROC AUC equal to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5 (See Figure below).



Dummy Classifier

dmy_clf = DummyClassifier(random_state=42)

dmy_clf.fit(X_train_std,y_train)

y_train_pred=cross_val_predict(dmy_clf,X_train_std,y_train, cv=3)

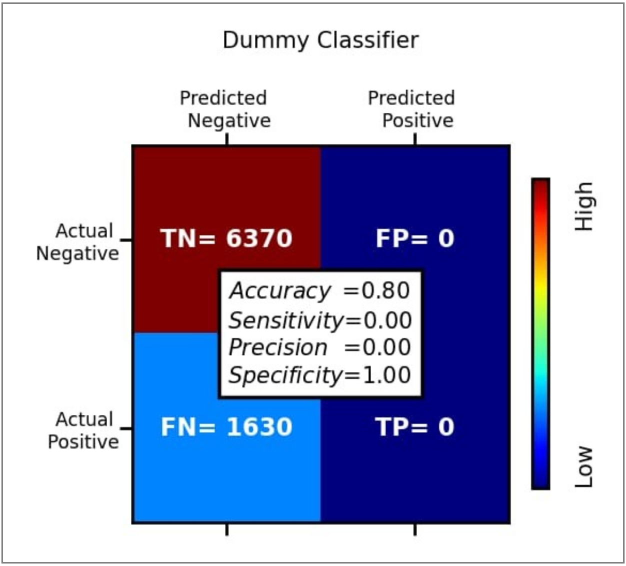
y_train_proba_dc=cross_val_predict(dmy_clf,X_train_std,y_train, cv=3, method='predict_proba')

font = {'size'   : 6}

plt.rc('font', **font)

fig = plt.subplots(figsize=(5, 5), dpi= 250, facecolor='w', edgecolor='k')

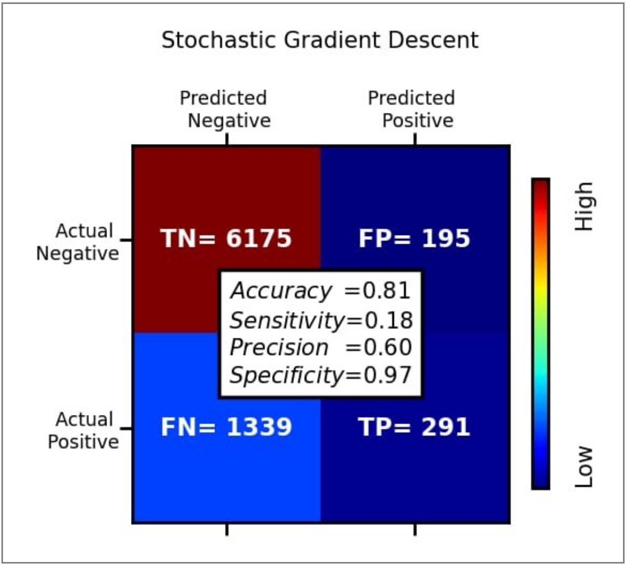
```
ax1=plt.subplot(1,2,1)

i=0

tmp1,tmp2,tmp3,tmp4=prfrmnce_plot.Conf_Matrix(y_
train,y_train_pred.reshape(-1,1),axt=ax1,


t_fontsize=6,x_fontsize=5,y_fontsize=5,title='Dummy
Classifier')


Train_Accu[i]=tmp1; Train_Pre[i]=tmp2;
Train_Rec[i]=tmp3; Train_Spe[i]=tmp4


# save the model to disk

filename[i] = './Trained_Models/dmy_clf.sav'

pickle.dump(dmy_clf, open(
```



Stochastic Gradient Descent

```
sgd_clf = SGDClassifier(random_state=42,loss='log')

sgd_clf.fit(X_train_std,y_train)

y_train_pred=cross_val_predict(sgd_clf,X_train_std,y_train, cv=3)

y_train_proba_sgd=cross_val_predict(sgd_clf,X_train_std,y_train, cv=3, method='predict_proba')

font = {'size'   : 6}

plt.rc('font', **font)
```

```
fig = plt.subplots(figsize=(5, 5), dpi= 250, facecolor='w', edgecolor='k')

ax1=plt.subplot(1,2,1)

i=1

tmp1,tmp2,tmp3,tmp4=prfrmnce_plot.Conf_Matrix(y_train,y_train_pred.reshape(-1,1),axt=ax1,t_fontsize=6, x_fontsize=5,
                              y_fontsize=5,title='Stochastic Gradient Descent')

Train_Accu[i]=tmp1; Train_Pre[i]=tmp2; Train_Rec[i]=tmp3; Train_Spe[i]=tmp4


# save the model to disk

filename[i] = './Trained_Models/sgd_clf.sav'

pickle.dump(sgd_clf, open(
```



Logistic Regression

lr_clf = LogisticRegression(max_iter=
50,C=50,random_state=42)

lr_clf.fit(X_train_std,y_train)

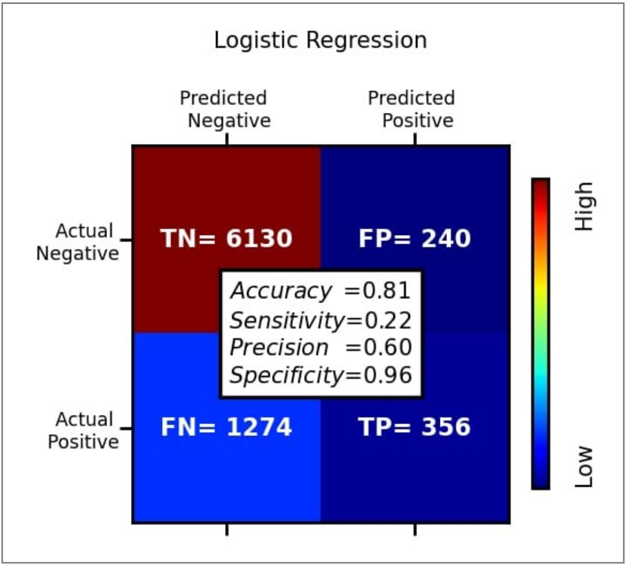
y_train_pred=cross_val_predict(lr_clf,X_train_std,y_tr
ain, cv=3)

y_train_proba_lr=cross_val_predict(lr_clf,X_train_std,
y_train, cv=3, method='predict_proba')

font = {'size'   : 6}

plt.rc('font', **font)

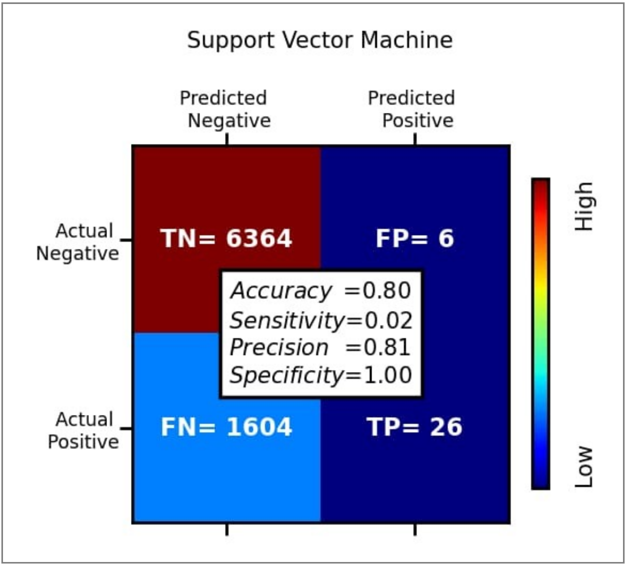
```
fig = plt.subplots(figsize=(5, 5), dpi= 250, facecolor='w', edgecolor='k')

ax1=plt.subplot(1,2,1)

i=2

tmp1,tmp2,tmp3,tmp4=prfrmnce_plot.Conf_Matrix(y_train,y_train_pred.reshape(-1,1),axt=ax1,


t_fontsize=6,x_fontsize=5,y_fontsize=5,title='Logistic Regression')

Train_Accu[i]=tmp1; Train_Pre[i]=tmp2; Train_Rec[i]=tmp3; Train_Spe[i]=tmp4


# save the model to disk

filename[i] = './Trained_Models/lr_clf.sav'

pickle.dump(lr_clf, open(
```



Support Vector Machine

```
svm_clf =
LinearSVC(C=60,loss='hinge',random_state=42)

svm_clf.fit(X_train_std,y_train)

y_train_pred=cross_val_predict(svm_clf,X_train_std,y_
train, cv=3)

y_train_proba_svm=y_train_pred

font = {'size'   : 6}

plt.rc('font', **font)
```

```
fig = plt.subplots(figsize=(5, 5), dpi= 250, facecolor='w',
edgecolor='k')

ax1=plt.subplot(1,2,1)

i=3

tmp1,tmp2,tmp3,tmp4=prfrmnce_plot.Conf_Matrix(y_
train,y_train_pred.reshape(-1,1),axt=ax1,


t_fontsize=6,x_fontsize=5,y_fontsize=5,title='Support
Vector Machine')

Train_Accu[i]=tmp1; Train_Pre[i]=tmp2;
Train_Rec[i]=tmp3; Train_Spe[i]=tmp4


# save the model to disk

filename[i] = './Trained_Models/svm_clf.sav'

pickle.dump(svm_clf, open(
```



Decision Trees

tree_clf = DecisionTreeClassifier(max_depth= 10, min_samples_leaf= 15, min_samples_split=2, random_state=42)

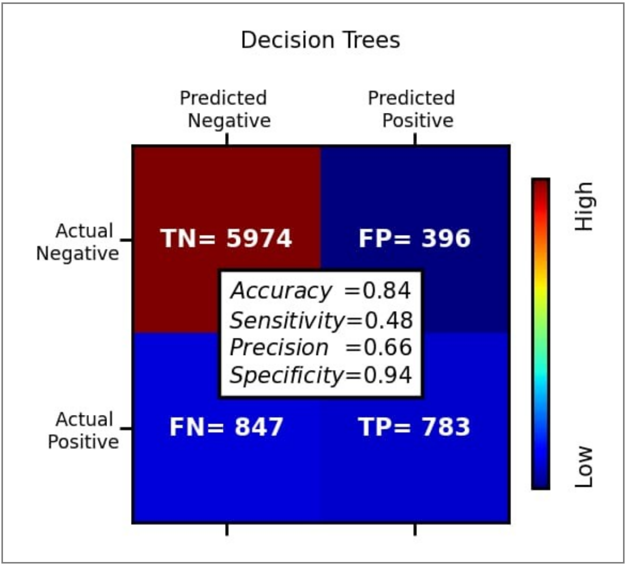
tree_clf.fit(X_train_std,y_train)

y_train_pred=cross_val_predict(tree_clf,X_train_std,y_train, cv=3)

y_train_proba_dt=cross_val_predict(tree_clf,X_train_std,y_train, cv=3, method='predict_proba')

font = {'size'   : 6}

```
plt.rc('font', **font)

fig = plt.subplots(figsize=(5, 5), dpi= 250, facecolor='w', edgecolor='k')

ax1=plt.subplot(1,2,1)

i=4

tmp1,tmp2,tmp3,tmp4=prfrmnce_plot.Conf_Matrix(y_train,y_train_pred.reshape(-1,1),axt=ax1,


t_fontsize=6,x_fontsize=5,y_fontsize=5,title='Decision Trees')

Train_Accu[i]=tmp1; Train_Pre[i]=tmp2; Train_Rec[i]=tmp3; Train_Spe[i]=tmp4


# save the model to disk

filename[i] = './Trained_Models/tree_clf.sav'

pickle.dump(tree_clf, open(filename[i], 'wb'))
```



Flowchart of Decision Tree for Prediction

Flowchart of Decision Tree for Prediction with 3 depths. Training features (X) and target (y) are shown in the figure.

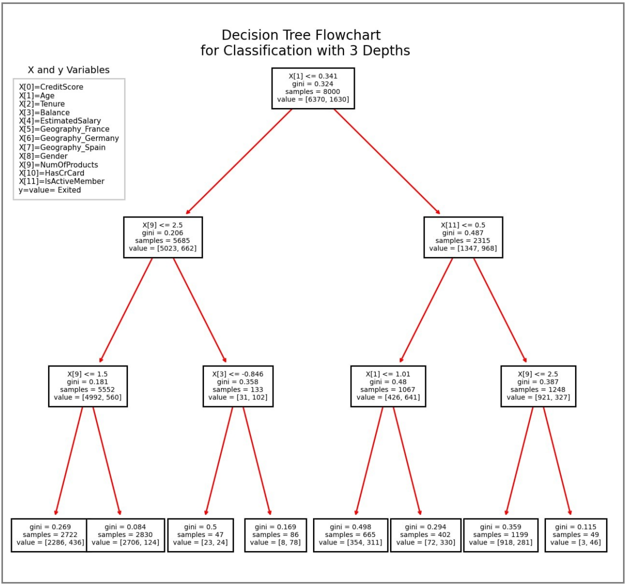
np.random.seed(42)

tree_clf = tree.DecisionTreeClassifier(max_depth=3)

tree_clf.fit(X_train_std,y_train)

fig, axes = plt.subplots(nrows = 1,ncols = 1,figsize = (8,8), dpi=200)

```
out = tree.plot_tree(tree_clf,fontsize=5)
for o in out:
    arrow = o.arrow_patch
    if arrow is not None:
        arrow.set_edgecolor('red')
        arrow.set_linewidth(1)


txt=''
for i in range(len(features_colums)):
    txt+='X'+'['+str(i)+']='+features_colums[i]+'\n'
txt+='y=value= Exited'
plt.text(0.025,0.9, 'X and y Variables',
fontsize=7,bbox=dict(facecolor='white', alpha=0.0))
plt.text(0.01,0.7, txt,
fontsize=5.5,bbox=dict(facecolor='white', alpha=0.2))
fig.suptitle('Decision Tree Flowchart \n for
Classification with 3 Depths', fontsize=10,y=0.86)
plt.show()
```



Adaptive Boosting with Decision Trees

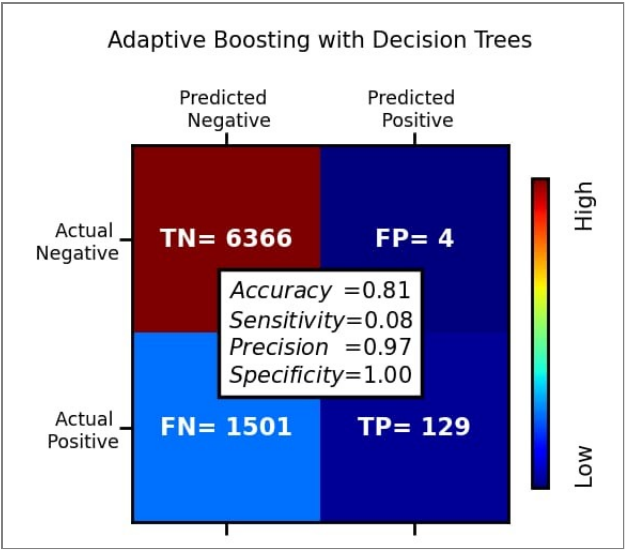
ada_tree = AdaBoostClassifier(n_estimators=100, algorithm= 'SAMME.R', learning_rate= 0.01, random_state=42)

ada_tree.fit(X_train_std,y_train)

y_train_pred=cross_val_predict(ada_tree,X_train_std,y_train, cv=3)

y_train_proba_ada=cross_val_predict(ada_tree,X_train_std,y_train, cv=3, method='predict_proba')

```
font = {'size'   : 6}

plt.rc('font', **font)

fig = plt.subplots(figsize=(5, 5), dpi= 250, facecolor='w', edgecolor='k')

ax1=plt.subplot(1,2,1)

i=5

tmp1,tmp2,tmp3,tmp4=prfrmnce_plot.Conf_Matrix(y_train,y_train_pred.reshape(-1,1),axt=ax1,


t_fontsize=6,x_fontsize=5,y_fontsize=5,title='Adaptive Boosting with Decision Trees')

Train_Accu[i]=tmp1; Train_Pre[i]=tmp2; Train_Rec[i]=tmp3; Train_Spe[i]=tmp4


# save the model to disk

filename[i] = './Trained_Models/ada_tree.sav'

pickle.dump(ada_tree, open
```



Random Forest

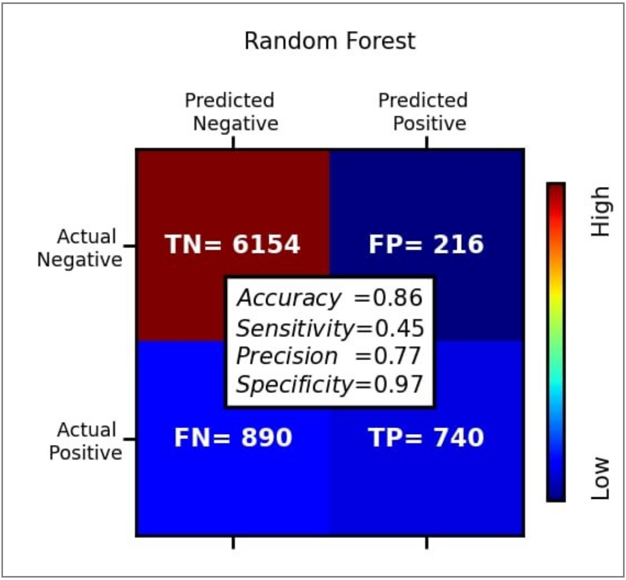
rnd = RandomForestClassifier(n_estimators=50, max_depth= 25, min_samples_split= 20, bootstrap= True, random_state=42)

rnd.fit(X_train_std,y_train)

y_train_pred=cross_val_predict(rnd,X_train_std,y_train, cv=3)

y_train_proba_rnd=cross_val_predict(rnd,X_train_std, y_train, cv=3, method='predict_proba')

```
font = {'size'   : 6}

plt.rc('font', **font)

fig = plt.subplots(figsize=(5, 5), dpi= 250, facecolor='w', edgecolor='k')

ax1=plt.subplot(1,2,1)

i=6

tmp1,tmp2,tmp3,tmp4=prfrmnce_plot.Conf_Matrix(y_train,y_train_pred.reshape(-1,1),axt=ax1,

t_fontsize=6,x_fontsize=5,y_fontsize=5,title='Random Forest')

Train_Accu[i]=tmp1; Train_Pre[i]=tmp2; Train_Rec[i]=tmp3; Train_Spe[i]=tmp4


# save the model to disk

filename[i] = './Trained_Models/rnd.sav'

pickle.dump(rnd, open
```



Random Forest Feature Importance

```
# Plot the importance of features

font = {'size'   : 7}

plt.rc('font', **font)

fig, ax1 = plt.subplots(figsize=(6, 3), dpi= 180, facecolor='w', edgecolor='k')


importance=rnd.feature_importances_
```

```
_=prfrmnce_plot(importance, title=f'Random Forest Algorithm to Rank Importance of Features to Predict Churn Customers',
           ylabel='Random Forest Score'
          ,clmns=features_colums,titlefontsize=10,
          xfontsize=7, yfontsize=8).bargraph(perent=True, select=None,axt=ax1,
          fontsizelable=7.
```

Neural Network

```
model_NN=ANN (input_dim=Smaller_Training.shape[1], activation= 'relu',
             dropout_rate= False, neurons= 100 )


# Early stopping to avoid overfitting

monitor= keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=1e-5,patience=5, mode='auto')

history=model_NN.fit(Smaller_Training,Smaller_Training_Target,batch_size=32,validation_data=
         (Validation,Validation_Target),callbacks=[monitor],verbose=1,epochs=1000)
```

Epoch 1/1000

200/200 [==============================] - 1s 3ms/step - loss: 0.4440 - accuracy: 0.8061 - val_loss: 0.4116 - val_accuracy: 0.8356

Epoch 2/1000

200/200 [==============================] - 0s 2ms/step - loss: 0.3962 - accuracy: 0.8333 - val_loss: 0.3923 - val_accuracy: 0.8394

Epoch 3/1000

200/200 [==============================] - 0s 2ms/step - loss: 0.3706 - accuracy: 0.8477 - val_loss: 0.3713 - val_accuracy: 0.8525

Epoch 4/1000

200/200 [==============================] - 0s 2ms/step - loss: 0.3500 - accuracy: 0.8531 - val_loss: 0.3720 - val_accuracy: 0.8619

Epoch 5/1000

200/200 [==============================] - 0s 2ms/step - loss: 0.3369 - accuracy: 0.8628 - val_loss: 0.3578 - val_accuracy: 0.8550

Epoch 6/1000

200/200 [==============================] - 0s 2ms/step - loss: 0.3270 - accuracy: 0.8678 - val_loss: 0.3498 - val_accuracy: 0.8625

Epoch 7/1000

200/200 [==============================] - 0s 2ms/step - loss: 0.3182 - accuracy: 0.8728 - val_loss: 0.3549 - val_accuracy: 0.8587

Epoch 8/1000

200/200 [==============================] - 0s 2ms/step - loss: 0.3140 - accuracy: 0.8744 - val_loss: 0.3475 - val_accuracy: 0.8594

Epoch 9/1000

200/200 [==============================] - 0s 2ms/step - loss: 0.3041 - accuracy: 0.8780 - val_loss: 0.3460 - val_accuracy: 0.8619

Epoch 10/1000

200/200 [==============================] - 0s 2ms/step - loss: 0.3003 - accuracy: 0.8781 - val_loss: 0.3659 - val_accuracy: 0.8619

Epoch 11/1000

200/200 [==============================] - 0s 2ms/step - loss: 0.2920 - accuracy: 0.8797 - val_loss: 0.3572 - val_accuracy: 0.8556

Epoch 12/1000

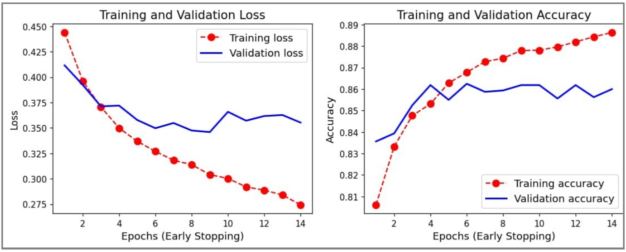
200/200 [==============================] - 0s 2ms/step - loss: 0.2888 - accuracy: 0.8820 - val_loss: 0.3617 - val_accuracy: 0.8619

Epoch 13/1000

200/200 [==============================] - 0s 2ms/step - loss: 0.2841 - accuracy: 0.8844 - val_loss: 0.3628 - val_accuracy: 0.8562

Epoch 14/1000

200/200 [==============================] - 0s 2ms/step - loss: 0.2744 - accuracy: 0.8864 - val_loss: 0.3553 - val_accuracy: 0.8600



Bootstraping (Neural Network)

Bootstrapping is run to achieve reliable number of epochs for neural network.

tf.get_logger().setLevel(logging.ERROR) # Disable tensorflow warning

pred_out_of_smpl= []

y_out_of_smpl   = []

acr_out_of_smpl   = []

```
# Apply 20 bootstraping sampling
n_splits=20
boot =
StratifiedShuffleSplit(n_splits=n_splits,test_size=0.2,
random_state=42)


num = 0
ES_epoches=[]
for train_idx, validation_idx in boot.split(X_train_std,
y_train):
    num+=1
    x_train      = X_train_std[train_idx]
    t_train      = y_train[train_idx]
    x_validation = X_train_std[validation_idx]
    t_validation = y_train[validation_idx]


    y_out_of_smpl.append(t_validation)


    # Call Function
    model,
epoch=BS_ANN(x_train=x_train,y_train=t_train,
```

```
 x_Validation=x_validation,y_Validation=t_validation,
neurons=100, activation= 'relu',
            dropout_rate= False)


    ES_epoches.append(epoch)
    pred = model.predict(x_validation)


    pred=[1 if i >= 0.5 else 0 for i in pred]
    pred_out_of_smpl.append(pred)


    acr=accuracy_score(t_validation, pred)
    reca=recall_score(t_validation, pred) # ==
TP/(TP+FN) )


    acr_out_of_smpl.append(reca)


    # Record this
    print('Sample #'+str(num)+', Mean
Recall='+str(np.round(np.mean(acr_out_of_smpl),4))
          +', S.D.
Recall='+str(np.round(np.sqrt(np.var(acr_out_of_smpl
)),4))
```

+', Epoch='+str(epoch)+', Mean Epoch='+str(int(np.mean(ES_epoches))))

50/50 [==============================] - 0s 899us/step

Sample #1, Mean Recall=0.4448, S.D. Recall=0.0, Epoch=10, Mean Epoch=10

50/50 [==============================] - 0s 857us/step

Sample #2, Mean Recall=0.4525, S.D. Recall=0.0077, Epoch=10, Mean Epoch=10

50/50 [==============================] - 0s 1ms/step

Sample #3, Mean Recall=0.4611, S.D. Recall=0.0138, Epoch=11, Mean Epoch=10

50/50 [==============================] - 0s 857us/step

Sample #4, Mean Recall=0.4609, S.D. Recall=0.012, Epoch=13, Mean Epoch=11

50/50 [==============================] - 0s 899us/step

Sample #5, Mean Recall=0.4663, S.D. Recall=0.0152, Epoch=15, Mean Epoch=11

50/50 [==============================] - 0s 898us/step

Sample #6, Mean Recall=0.4729, S.D. Recall=0.0203, Epoch=9, Mean Epoch=11

50/50 [==============================] - 0s 857us/step

Sample #7, Mean Recall=0.4667, S.D. Recall=0.0242, Epoch=9, Mean Epoch=11

50/50 [==============================] - 0s 939us/step

Sample #8, Mean Recall=0.4609, S.D. Recall=0.0273, Epoch=15, Mean Epoch=11

50/50 [==============================] - 0s 980us/step

Sample #9, Mean Recall=0.47, S.D. Recall=0.0365, Epoch=13, Mean Epoch=11

50/50 [==============================] - 0s 959us/step

Sample #10, Mean Recall=0.4687, S.D. Recall=0.0348, Epoch=11, Mean Epoch=11

50/50 [==============================] - 0s 1ms/step

Sample #11, Mean Recall=0.4702, S.D. Recall=0.0335, Epoch=11, Mean Epoch=11

50/50 [==============================] - 0s 1ms/step

Sample #12, Mean Recall=0.4783, S.D. Recall=0.0419, Epoch=13, Mean Epoch=11

50/50 [==============================] - 0s 837us/step

Sample #13, Mean Recall=0.4792, S.D. Recall=0.0404, Epoch=11, Mean Epoch=11

50/50 [==============================] - 0s 837us/step

Sample #14, Mean Recall=0.4783, S.D. Recall=0.039, Epoch=12, Mean Epoch=11

50/50 [==============================] - 0s 918us/step

Sample #15, Mean Recall=0.4736, S.D. Recall=0.0416, Epoch=9, Mean Epoch=11

50/50 [==============================] - 0s 837us/step

Sample #16, Mean Recall=0.4751, S.D. Recall=0.0407, Epoch=12, Mean Epoch=11

50/50 [==============================] - 0s 899us/step

Sample #17, Mean Recall=0.478, S.D. Recall=0.0411, Epoch=12, Mean Epoch=11

50/50 [==============================] - 0s 837us/step

Sample #18, Mean Recall=0.4761, S.D. Recall=0.0407, Epoch=10, Mean Epoch=11

50/50 [==============================] - 0s 852us/step

Sample #19, Mean Recall=0.4839, S.D. Recall=0.0514, Epoch=14, Mean Epoch=11

50/50 [==============================] - 0s 898us/step

Sample #20, Mean Recall=0.4888, S.D. Recall=0.0545, Epoch=9, Mean Epoch=11

Call Function with fined-tune numebr of neurons

model_FNN=ANN (input_dim=X_train_std.shape[1], activation= 'relu',

dropout_rate= False, neurons= 100)

Early stopping to avoid overfitting

history=model_FNN.fit(X_train_std, y_train,batch_size=32,verbose=1,epochs=11)

Epoch 1/11

250/250 [==============================] - 1s 1ms/step - loss: 0.4393 - accuracy: 0.8040

Epoch 2/11

250/250 [==============================] - 0s 1ms/step - loss: 0.3931 - accuracy: 0.8365

Epoch 3/11

250/250 [==============================] - 0s 1ms/step - loss: 0.3624 - accuracy: 0.8516

Epoch 4/11

250/250 [==============================] - 0s 1ms/step - loss: 0.3447 - accuracy: 0.8593

Epoch 5/11

250/250 [==============================] - 0s 1ms/step - loss: 0.3349 - accuracy: 0.8655

Epoch 6/11

250/250 [==============================] - 0s 1ms/step - loss: 0.3252 - accuracy: 0.8679

Epoch 7/11

250/250 [==============================] - 0s 1ms/step - loss: 0.3197 - accuracy: 0.8673

Epoch 8/11

250/250 [==============================] - 0s 1ms/step - loss: 0.3152 - accuracy: 0.8705

Epoch 9/11

250/250 [==============================] - 0s 1ms/step - loss: 0.3118 - accuracy: 0.8698

Epoch 10/11

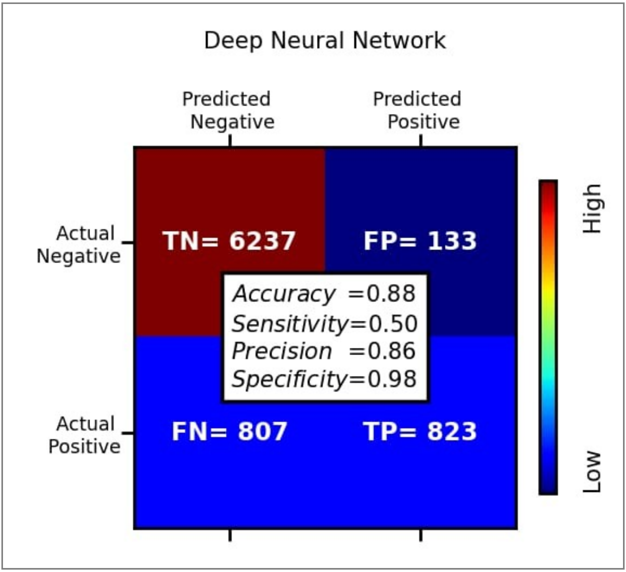
250/250 [==============================] - 0s 1ms/step - loss: 0.3039 - accuracy: 0.8754

```
Epoch 11/11

250/250 [==============================] - 0s 1ms/step - loss: 0.3013 - accuracy: 0.8754

font = {'size'   : 6}

plt.rc('font', **font)

fig = plt.subplots(figsize=(5, 5), dpi= 250, facecolor='w', edgecolor='k')

ax1=plt.subplot(1,2,1)

i=7

pred = model_FNN.predict(X_train_std)

y_train_proba_ai=pred

y_train_pred=np.array([1 if i >= 0.5 else 0 for i in pred])

tmp1,tmp2,tmp3,tmp4=prfrmnce_plot.Conf_Matrix(y_train,y_train_pred.reshape(-1,1),axt=ax1,

    t_fontsize=6,x_fontsize=5,y_fontsize=5,title='Deep Neural Network')


Train_Accu[i]=tmp1; Train_Pre[i]=tmp2; Train_Rec[i]=tmp3; Train_Spe[i]=tmp4


# save the model to disk
```

filename[i] = './Trained_Models/'

model_FNN.save(os.path.join(filename[i],"NN_model.h5"))

250/250 [==============================] - 0s 839us/step



ROC Chart for Training set

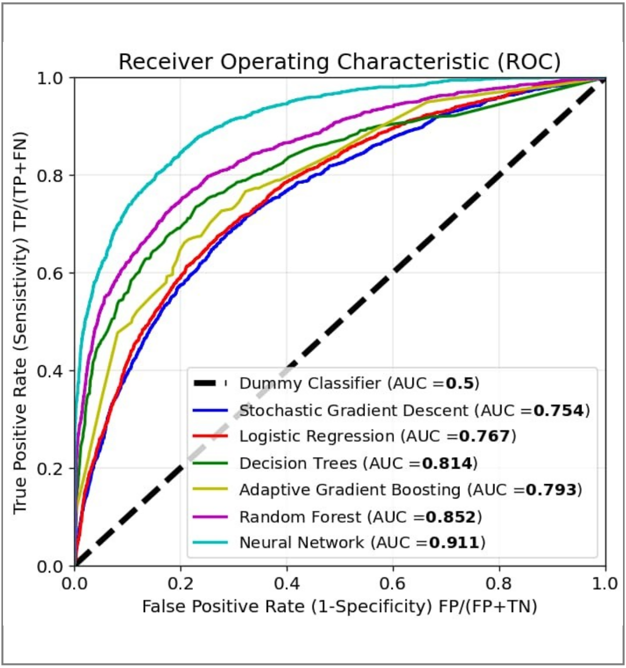
The receiver operating characteristic (ROC) curve is another common tool used with binary classifiers. It is very similar to the precision/recall curve, but instead of plotting precision versus recall, the ROC curve plots the true positive rate (another name for recall) against the false positive rate. The FPR is the ratio of negative instances that are incorrectly classified as positive. It is equal to one minus the true negative rate, which is the ratio of negative instances that are correctly classified as negative. The TNR is also called specificity. Hence the ROC curve plots sensitivity (recall) versus 1 – specificity.

For more information and details about ROC, see ROC

Thus every point on the ROC curve represents a chosen cut-off even though you cannot see this cut-off. What you can see is the true positive fraction and the false positive fraction that you will get when you choose this cut-off.

```
font = {'size' : 12}

plt.rc('font', **font)

fig,ax = plt.subplots(figsize=(6.5,6), dpi= 120,
facecolor='w', edgecolor='k')


prediction_prob=[y_train_proba_dc,
y_train_proba_sgd,y_train_proba_lr,y_train_proba_dt,
```

```
 y_train_proba_ada,y_train_proba_rnd,y_train_proba_
ai]

predictor_name=['Dummy Classifier','Stochastic
Gradient Descent','Logistic Regression','Decision
Trees',

                'Adaptive Gradient Boosting','Random
Forest','Neural Network']


prfrmnce_plot.AUC(prediction_prob,np.array(y_train)
, n_algorithm=len(predictor_name),
label=predictor_name

       ,title='Receiver
```



The best model is neural network with AUC=0.91 followed by Random Forest (AUC=0.85). Lets compare accuracy, sensitivity, specificity and precision of the classifiers.

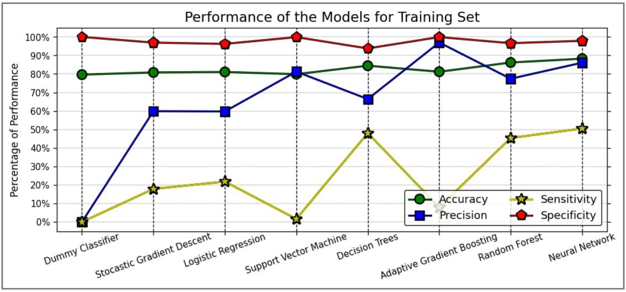
Performance of the Models on Training Set

font = {'size'   :9 }

plt.rc('font', **font)

```
ax2 = ax1.twinx()


fig, ax1 = plt.subplots(figsize=(10, 3.7), dpi= 120,
facecolor='w', edgecolor='k')


ax1.plot(predictor,Train_Accu*100,'go-
',linewidth=1,path_effects=[pe.Stroke(linewidth=2,
foreground='k'), pe.Normal()],

 markersize=9,label='Accuracy',markeredgecolor='k')
ax1.plot(predictor,Train_Pre*100,'bs-
',linewidth=1,path_effects=[pe.Stroke(linewidth=2,
foreground='k'), pe.Normal()],

 markersize=8.5,label='Precision',markeredgecolor='k'
)
ax1.plot(predictor,Train_Rec*100,'y*-
',linewidth=2,path_effects=[pe.Stroke(linewidth=2,
foreground='k'), pe.Normal()],

 markersize=12,label='Sensitivity',markeredgecolor='k
')
ax1.plot(predictor,Train_Spe*100,'rp-
',linewidth=1,path_effects=[pe.Stroke(linewidth=2,
foreground='k'), pe.Normal()],
```

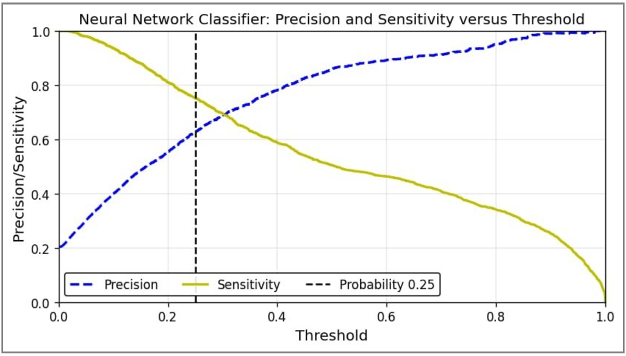
```
 markersize=10,label='Specificity',markeredgecolor='k
')

ax1.set_xticks(np.arange(len(predictor)))

ax1.set_xticklabels(predictor,y=0.03, rotation=19)

ax1.xaxis.grid(color='k', linestyle='--', linewidth=0.8)

ax1.yaxis.grid(color='k', linestyle='--', linewidth=0.2)

plt.ylabel('Percentage of Performance',fontsize=10)

plt.title('Performance of the Models for Training
Set',fontsize=14)

legend=plt.legend(ncol=2,loc=4,fontsize='11',framealp
ha =0.8)

legend.get_frame().set_edgecolor("black")


ax1.yaxis.set_major_formatter(mtick.PercentFormatte
r())

plt.yticks([0,10,20,30,40,50,60,70,80,90,100],y=0,
fontsize=9)

ax1.yaxis.set_ticks_position('both')


#plt.ylim(0.55,0.75)

plt.show()
```



Sensitivity is very low for all classifiers that lead to very high false negatives. This is problematic for customer churn since we want to minimize false negatives prediction. We can change the threshold to decrease precision that leads to increase sensitivity. See the Figure below for the plot of precision and sensitivity for each threshold for Neural Network. By decreasing the threshold from 0.5 to 0.29, sensitivity will increase.

```
font = {'size' : 10}

plt.rc('font', **font)

fig = plt.subplots(figsize=(8,4), dpi= 120, facecolor='w', edgecolor='k')


pred=y_train_proba_ai
```

```
precisions, sensitivity, thresholds =
precision_recall_curve(y_train, pred)

threshold_ = 0.25

plot_precision_recall_vs_threshold(precisions,
sensitivity, thresholds,x=threshold_)

plt.show()
```



Now calculate the sensitivity and precision again with threshold 0.25. Sensitivity increases to 0.72 but precision decreases to 0.65. This threshold will apply for test set and future data.

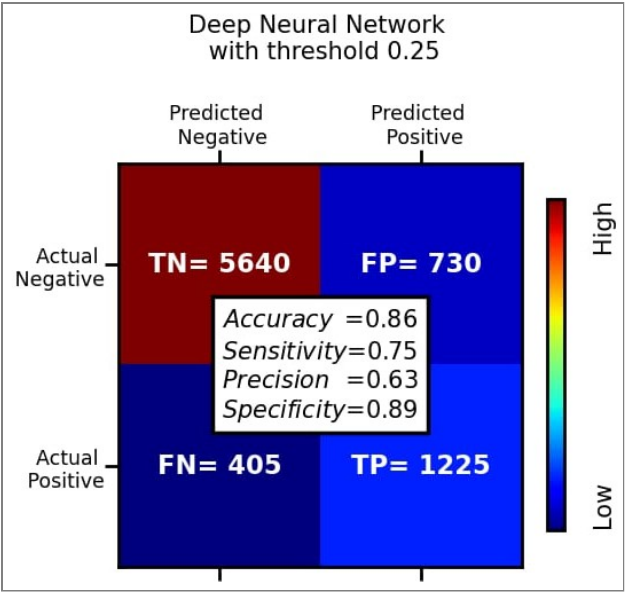
```
font = {'size'   : 6}

plt.rc('font', **font)
```

fig = plt.subplots(figsize=(5, 5), dpi= 250, facecolor='w', edgecolor='k')

ax1=plt.subplot(1,2,1)

y_p=np.array([1 if i >= threshold_ else 0 for i in pred])

_,_,_,_=prfrmnce_plot.Conf_Matrix(y_train,y_p.reshape(-1,1),axt=ax1,



Test Set

Process the test data using the same approach and statistics from training set.

```
# Data Processing

# Convert Geography to one-hot-encoding
Geog_1hot=pd.get_dummies(test_set_strat['Geography'],prefix='Geography')

# Convert gender to 0 and 1
ordinal_encoder = OrdinalEncoder()
test_set_strat['Gender'] = ordinal_encoder.fit_transform(test_set_strat[['Gender']])

# Remove 'Geography'
test_set_strat=test_set_strat.drop(['Geography'],axis=1, inplace=False)
test_set_strat=pd.concat([Geog_1hot,test_set_strat], axis=1) # Concatenate rows

# Standardize data
X_test = test_set_strat.drop("Exited", axis=1)
y_test = test_set_strat["Exited"].values
#
```

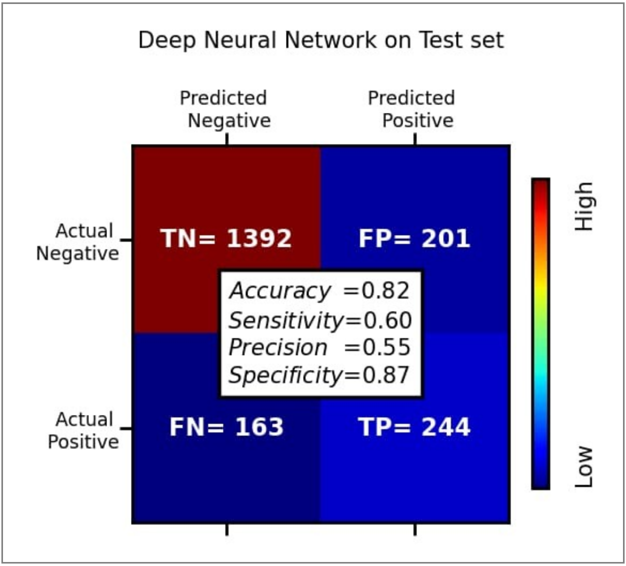
```
clmn=
['Geography_France','Geography_Germany','Geograph
y_Spain',

 'Gender','NumOfProducts','HasCrCard','IsActiveMemb
er']

X_test_for_std =  X_test.drop(clmn, axis=1)

X_test_not_std =  X_test[clmn]

features_colums=list(X_test_for_std.columns)+list(X_te
st_not_std.columns)

#

fname = './Trained_Models/scaler.sav'

scaler_model = pickle.load(open(fname, 'rb'))

#

df_test_std=scaler_model.transform(X_test_for_std)

X_test_std=np.concatenate
```

Prediction on Never Seen before Data

Load the trained Neural Network model and apply for prediction of test set.

Now apply threshold that we applied for training set. The Figure below shows the calculated metrics for test sets.

```
filename = './Trained_Models'

loaded_model =
keras.models.load_model(os.path.join(filename,"NN_
model.h5"))

pred=loaded_model.predict(X_test_std)

63/63 [==============================] - 0s
903us/step

font = {'size'   : 6}

plt.rc('font', **font)

fig = plt.subplots(figsize=(5, 5), dpi= 250, facecolor='w',
edgecolor='k')

ax1=plt.subplot(1,2,1)

y_p=np.array([1 if i >= threshold_ else 0 for i in pred])

_,_,_,_=prfrmnce_plot.Conf_Matrix(y_test,y_p.reshape(
-1,1),axt=ax1,

          t_fontsize=6,
```



The sensitivity and precision have been reduced for test set because of the model has not seen the data during training. The performance can be enhanced by increasing more training data

Train Final Model with all Data

The final step is training the best model (Neural Network) with entire data to take advantage of all data data to train a better model. The same hyper parameters should be applied.

```
# Data Processing

# Read data 'Churn_Modelling.csv'
df = pd.read_csv('./Data/Churn_Modelling.csv')

# Remove 'RowNumber','CustomerId','Surname'
features that are un
df=df.drop(['RowNumber','CustomerId','Surname'],axi
s=1,inplace=False)

# Shuffle the data
np.random.seed(32)
df=df.reindex(np.random.permutation(df.index))
df.reset_index(inplace=True, drop=True) # Reset
index

# Convert Geography to one-hot-encoding
Geog_1hot=pd.get_dummies(df['Geography'],prefix='G
eography')

# Convert gender to 0 and 1
```

```
ordinal_encoder = OrdinalEncoder()
df['Gender'] =
ordinal_encoder.fit_transform(df[['Gender']])


# Remove 'Geography'
df_set_strat=df.drop(['Geography'],axis=1,inplace=Fals
e)
df_set_strat=pd.concat([Geog_1hot,df_set_strat],
axis=1) # Concatenate rows


# Standardize data
X_df = df_set_strat.drop("Exited", axis=1)
y_df = df_set_strat["Exited"].values
#
clmn=
['Geography_France','Geography_Germany','Geograph
y_Spain',

 'Gender','NumOfProducts','HasCrCard','IsActiveMemb
er']
X_for_std =  X_df.drop(clmn, axis=1)
X_not_std =  X_df[clmn]
features_colums=list(X_for_std.columns)+list(clmn)
```

```
#

scaler = StandardScaler()

scaler.fit(X_for_std)


df_std=scaler.transform(X_for_std)

X_test_std=np.concatenate((X_for_std,X_not_std),
axis=1)

# Retain Neural Network with entire data

Final_Model=ANN (input_dim=X_for_std.shape[1],
activation= 'relu',

             dropout_rate= False, neurons= 100 )

history=Final_Model.fit(X_for_std,
y_df,batch_size=32,verbose=1,epochs=11)

Epoch 1/11

313/313 [==============================] - 1s
2ms/step - loss: 88.0091 - accuracy: 0.6745

Epoch 2/11

313/313 [==============================] - 0s
2ms/step - loss: 12.5444 - accuracy: 0.6775

Epoch 3/11

313/313 [==============================] - 0s
2ms/step - loss: 5.0050 - accuracy: 0.6806

Epoch 4/11
```

313/313 [==============================] - 0s
2ms/step - loss: 3.1567 - accuracy: 0.6871

Epoch 5/11

313/313 [==============================] - 0s
2ms/step - loss: 1.7455 - accuracy: 0.6909

Epoch 6/11

313/313 [==============================] - 0s
1ms/step - loss: 1.6542 - accuracy: 0.6924

Epoch 7/11

313/313 [==============================] - 1s
2ms/step - loss: 0.7191 - accuracy: 0.7439

Epoch 8/11

313/313 [==============================] - 0s
2ms/step - loss: 0.6707 - accuracy: 0.7428

Epoch 9/11

313/313 [==============================] - 1s
2ms/step - loss: 0.6263 - accuracy: 0.7584

Epoch 10/11

313/313 [==============================] - 1s
2ms/step - loss: 0.6046 - accuracy: 0.7596

Epoch 11/11

313/313 [==============================] - 0s
2ms/step - loss: 0.6500 - accuracy: 0.7458

Save final model to disk

```
filename = './Trained_Models/'

model_FNN.save(os.path.join
```