Part 3 - Bank Marketing Dataset *(Supervised Classification)*

In [2]:
```
import pandas as pd
import numpy as np
```

In [ ]:
```
data = pd.read_csv('bank.csv',sep=';')
data.head()
```

In [ ]:
```
print('Number of rows:', len(data))
```

In [ ]:
```
data.isna().sum()
# Finding if there's null values
```

In [ ]:
```
data.info()
```

In [ ]:
```
num_col = data._get_numeric_data().columns
cate_col = list(set(data.columns)-set(num_col))
cate_col.remove('y')
nominal_col = ['housing', 'month', 'default', 'marital', 'contact', 'job', 'loan', '
```

In [ ]:
```
ordinal_col = []
for col in cate_col:
    if col not in nominal_col:
        ordinal_col.append(col)
ordinal_col
```

In [ ]:
```
for i in data[cate_col]:
    print(i,data[i].unique())
```

In [ ]:
```
data[num_col].describe()
#pday min should not be -1
```

In [ ]:
```
# checking for erroneous data in dependent data
data['y'].unique()
```

Data Cleaning

In [ ]:
```
data['pdays'] = data['pdays'].replace({-1:0})
#Cleaning the datas
```

In [ ]:
```
data['y'] = data['y'].replace({'no':1, 'yes':0})
# replace string to integer for Classification model
```

Exploratory Data Analytics

In [ ]:
```
y_df = data.pivot_table(values='age', index='y', aggfunc='count')
y_df = y_df.rename({'age':'proportion'},axis=1)
y_df = y_df*100/len(data)
y_df
```

In [ ]:
```
import matplotlib.pyplot as plt
plt.pie(y_df['proportion'], labels=['Subscribed', 'Not Subscribed'], autopct='%1.1f%
          radius = 2)
```

In [ ]:
```
data['y'].value_counts()
```

In [ ]:
```
import seaborn as sns
sns.pairplot(data)
```

In [ ]:
```
import matplotlib.pyplot as plt
corrs = data.corr()
sns.heatmap(corrs, annot=True, fmt='.2f')
plt.show()
```

Reduce skewness in numerical data

In [ ]:
```
skew_limit = 0.75
non_negative_num_col = num_col.drop('balance') # balance has a negative value
skew_cols = (data[non_negative_num_col].skew()
             .sort_values(ascending=False)
             .to_frame()
             .rename(columns={0:'Skew'})
             .query('abs(Skew) > {}'.format(skew_limit)))

skew_cols # very high skewness in features
```

In [ ]:
```
field = "previous"
fig,(ax_org, ax_sqrt) = plt.subplots(1, 2)

data[field].hist(ax=ax_org)

# Apply a log transformation (numpy syntax) to this column
data[field].apply(np.sqrt).hist(ax=ax_sqrt)

# Formatting of titles etc. for each subplot
ax_org.set(title='before', ylabel='frequency', xlabel='value')
ax_sqrt.set(title='after', ylabel='frequency', xlabel='value')
fig.suptitle('Field "{}"'.format(field));
print('pop_orignal skewness: ',data[field].skew())
print('pop_sqrt skewness: ',data[field].apply(np.sqrt).skew())
# fall in skewness after log1p
```

In [ ]:
```
# apply log1p across all numerical columns

for col in data[non_negative_num_col]:
```

```
# drop famrel as skewness increase after log1p
    data[col] = data[col].apply(np.sqrt)
new_skew_cols = (data[skew_cols.index].skew()
             .sort_values(ascending=False)
             .to_frame()
             .rename(columns={0:'Skew'}))

skew_cols['New_skew'] = new_skew_cols
skew_cols['Difference'] = abs(skew_cols['New_skew']) - abs(skew_cols['Skew'])
skew_cols
```

StandardScale numerical data

In [ ]:
```
from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
for col in num_col:
    data[col] = ss.fit_transform(data[[col]])
```

One hot encoding of categorical data

In [ ]:
```
data = pd.get_dummies(data, columns=nominal_col,drop_first=True)
data.head()
# df is the transformed dataest for ML, df is the orignal dataset
```

In [ ]:
```
# drop unknown and other inputs as this are not a reliable input
data = data.drop(['poutcome_unknown', 'poutcome_other', 'contact_unknown', 'job_unkn
```

Transform ordinal data into a numerical form

In [ ]:
```
data['education'].unique()
```

In [ ]:
```
data['education'] = data['education'].replace({'unknown': 0, 'primary':1, 'secondary
```

In [ ]:
```
# As all other numerical data are scaled via StandardScaler, we will MinMax scale th
from sklearn.preprocessing import MinMaxScaler
minm = MinMaxScaler()
data['education'] = minm.fit_transform(data[['education']])
data['education'].head()
```

Stratified Train Test Split

In [ ]:
```
features = list(filter(lambda x: x!= 'y', data.columns))
```

In [ ]:
```
from sklearn.model_selection import StratifiedShuffleSplit

# Get the split indexes
strat_shuf_split = StratifiedShuffleSplit(n_splits=3, test_size = 0.3,
                                          random_state=42)
```

```
train, test = next(strat_shuf_split.split(data[features], data.y))

# Create the dataframes
X_train = data.loc[train, features]
y_train = data.loc[train, 'y']

X_test  = data.loc[test, features]
y_test  = data.loc[test, 'y']
```

Model Evaluation

In [ ]:

```
from sklearn.metrics import precision_recall_fscore_support as score
from sklearn.metrics import confusion_matrix, accuracy_score, roc_auc_score, classif
from sklearn.preprocessing import label_binarize
import seaborn as sns
import matplotlib.pyplot as plt
from timeit import default_timer as timer # Calcuate time elapsed in training model

def classif_report(name, model, threshold=0.5):
    prediction = model.predict_proba(X_test)
    prediction = np.where(prediction[:,1]>=threshold, 1, 0)
    print("\u0332".join(name)+'\n')
    print(classification_report(y_test, prediction))

    # Confusion Matrix
    confuse = confusion_matrix(y_test, prediction)
    names = ['True Neg','False Pos','False Neg','True Pos']
    counts = ["{0:0.0f}".format(value) for value in
                confuse.flatten()]
    percentages = ["{0:.2%}".format(value) for value in
                confuse.flatten()/np.sum(confuse)]
    datalabels = [f"{v1}\n{v2}\n{v3}" for v1, v2, v3 in
                zip(names,counts,percentages)]
    datalabels = np.asarray(datalabels).reshape(2,2)
    axislabels = ['Subscribed', 'Not Subscribed']
    con_mat = sns.heatmap(confuse, annot=datalabels, fmt='', cmap='Blues', annot_kws
    con_mat.set_xticklabels(con_mat.get_xmajorticklabels(), fontsize = 12, verticala
    con_mat.set_yticklabels(con_mat.get_ymajorticklabels(), fontsize = 12, verticala
    plt.xlabel("\nPredictions")
    plt.ylabel("Ground Truth\n")
    plt.show()
```

In [ ]:

```
from sklearn.metrics import roc_curve, precision_recall_curve

def RocCurve(model):
    sns.set_context('talk')
    prediction = model.predict(X_test)

    fig,ax_list = plt.subplots(ncols=2)
    fig.set_size_inches(8, 4)
    # Get the probabilities for each of the two categories
    y_prob = model.predict_proba(X_test)
    auc = roc_auc_score(y_test, y_prob[:,1])

    # Plot the ROC-AUC curve
    aucurve = ax_list[0]

    fpr, tpr, thresholds = roc_curve(y_test, y_prob[:,1])
    aucurve.plot(fpr, tpr, linewidth=5)
    aucurve.plot([0, 1], [0, 1], ls='--', color='black', lw=.3)
```

```
    aucurve.set(xlabel='False-Positive',
           ylabel='True Positive',
           xlim=[-.01, 1.01], ylim=[-.01, 1.01],
           title="AUROC={}".format(round(auc,3)))
    aucurve.grid(True)

    # Plot the precision-recall curve
    prcurve = ax_list[1]

    precision, recall, _ = precision_recall_curve(y_test, y_prob[:,1])
    prcurve.plot(recall, precision, linewidth=5)
    prcurve.set(xlabel='Recall', ylabel='Precision',
           xlim=[-.01, 1.01], ylim=[-.01, 1.01],
           title='Precision-Recall curve')
    prcurve.grid(True)

    plt.tight_layout()
```

In [ ]:

```
from sklearn.metrics import roc_curve, precision_recall_curve

def RocCurve_noproba(model):
    prediction = model.predict(X_test)
    auc = roc_auc_score(y_test, prediction)
```

Dummy Model

In [ ]:

```
from sklearn.dummy import DummyClassifier
dummy = DummyClassifier(strategy="stratified")

start = timer()
dummy.fit(X_train, y_train)

end = timer()
dummy_time = end - start
print('Time Elapsed (sec):', dummy_time)
RocCurve(dummy)
```

In [ ]:

```
classif_report('Dummy Regression', dummy) # threshold based on precision-recall curv
```

Logistics Regression model with ridge regularization

In [ ]:

```
from sklearn.linear_model import LogisticRegressionCV
from sklearn.model_selection import GridSearchCV

start = timer()
logit_reg = LogisticRegressionCV(Cs=5, cv=5, scoring= 'f1').fit(X_train, y_train)

end = timer()
logit_reg_time = end - start
print('Time Elapsed (sec):', logit_reg_time)
RocCurve(logit_reg)
```

In [ ]:

```
logit_threshold = 0.5
classif_report('Logistics Regression', logit_reg, threshold=logit_threshold) # thres
```

K Nearest Neighbour Classification

In [ ]:
```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()
```

In [ ]:
```
param_test = {
    'n_neighbors': [3,4,5],
    'weights': ['distance'],
    'metric': ['euclidean']
}

start = timer()
knn_grid = GridSearchCV(knn, param_grid = param_test, cv=5, scoring= 'f1').fit(X_tra

end = timer()
knn_time = end - start
print('Time Elapsed (sec):', knn_time)
knn_grid.best_params_
knn_threshold = 0.65
```

In [ ]:
```
RocCurve(knn_grid)
```

In [ ]:
```
knn_threshold = 0.65
classif_report('KNN', knn_grid, threshold = knn_threshold) # threshold based on prec
```

In [ ]:
```
param_test = {
    'n_neighbors': [3,4,5],
    'weights': ['uniform', 'distance'],
    'metric': ['manhattan']
}

start = timer()
knn_grid_manhat = GridSearchCV(knn, param_grid = param_test, cv=5, scoring= 'f1').fi

end = timer()
knn_time_manhat = end - start
print('Time Elapsed (sec):', knn_time_manhat)
knn_grid.best_params_
knn_threshold = 0.65
```

In [ ]:
```
RocCurve(knn_grid_manhat)
```

SVC

In [ ]:
```
from sklearn.svm import SVC

SVC = SVC(probability=True)

param_test = {
    'gamma': [.5, 1, 2, 10]
}
```

```
start = timer()
SVC_grid = GridSearchCV(SVC, param_grid = param_test, cv=5, scoring= 'f1').fit(X_tra

end = timer()
SVC_time = end - start
print('Time Elapsed (sec):', SVC_time)
SVC_grid.best_params_
```

In [ ]:
```
RocCurve(SVC_grid)
```

In [ ]:
```
SVC_threshold = 0.7
classif_report('SVC',SVC_grid, threshold=SVC_threshold) # threshold based on precisi
```

Random Foret Classifier

In [ ]:
```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier(verbose=0)
```

In [ ]:
```
RandomFor = RandomForestClassifier(oob_score=True,
                                   random_state=42,
                                   warm_start=True,
                                   n_jobs=-1)
bag_list = list()

for n_trees in [15, 20, 30, 40, 50, 100, 150, 200, 300, 400]:

    # Use this to set the number of trees
    RandomFor.set_params(n_estimators=n_trees, max_depth=27, max_features ='sqrt')

    # Fit the model
    RandomFor.fit(X_train, y_train)

    # Get the oob error
    bag_error = 1 - RandomFor.oob_score_

    # Store it
    bag_list.append(pd.Series({'n_trees': n_trees, 'oob': bag_error}))

rf_oob_df = pd.concat(bag_list, axis=1).T.set_index('n_trees')

rf_oob_df
```

In [ ]:
```
import seaborn as sns

# oob_error with the number of trees
sns.set_context('talk')
sns.set_style('white')

axis = rf_oob_df.plot(legend=False, marker='o', figsize=(14, 7), linewidth=5)
axis.set(ylabel='out-of-bag error');
```

In [ ]:
```
feature = ['auto', 'sqrt']
depth = [int(x) for x in np.linspace(5, 50, num = 3)]
sample_min_split = [2, 5]
sample_min_leaf = [1, 2]
```

```
param_test = {'max_features': feature,
              'max_depth': depth,
              'min_samples_split': sample_min_split,
              'min_samples_leaf': sample_min_leaf}
```

In [ ]:
```
start = timer()
rfc = RandomForestClassifier(n_estimators=150)
rfc_grid = GridSearchCV(rfc, param_grid = param_test, scoring= 'f1').fit(X_train,y_t

end = timer()
rfc_time = end - start
print('Time Elapsed (sec):', rfc_time)
rfc_grid.best_params_
```

In [ ]:
```
RocCurve(rfc_grid)
```

In [ ]:
```
classif_report('Random Forest Classifier',rfc_grid)
```

Extra Tree Classifier

In [ ]:
```
from sklearn.ensemble import ExtraTreesClassifier

# Initialize the random forest estimator
# Note that the number of trees is not setup here
EF = ExtraTreesClassifier(oob_score=True,
                          random_state=42,
                          warm_start=True,
                          bootstrap=True, # sample rows/entries with replacement
                          n_jobs=-1)

bag_list = list()

# Iterate through all of the possibilities for
# number of trees
for n_trees in [15, 20, 30, 40, 50, 100, 150, 200, 300, 400]:

    # Use this to set the number of trees
    EF.set_params(n_estimators=n_trees)
    EF.fit(X_train, y_train)

    # oob error
    oob_error = 1 - EF.oob_score_
    bag_list.append(pd.Series({'n_trees': n_trees, 'oob': oob_error}))

et_oob_df = pd.concat(bag_list, axis=1).T.set_index('n_trees')

et_oob_df
```

In [ ]:
```
start = timer()
EF = ExtraTreesClassifier(oob_score=True, n_estimators=100,
                          random_state=42,
                          warm_start=True,
                          bootstrap=True, # sample rows/entries with replacement
                          n_jobs=-1).fit(X_train, y_train)
end = timer()
```

```
EF_time = end - start
print('Time Elapsed (sec):', EF_time)
```

In [ ]:
```
EF_threshold = 0.75
classif_report('Extra Tree Classifier',EF, EF_threshold)
```

In [ ]:
```
RocCurve(EF)
```

Gradient Boosting

In [ ]:
```
from sklearn.ensemble import GradientBoostingClassifier

error_list = list()

# Iterate through various possibilities for number of trees
tree_list = [15, 20, 30, 40, 50, 100, 150, 200, 300, 400]
for n_trees in tree_list:

    # Initialize the gradient boost classifier
    GBC = GradientBoostingClassifier(n_estimators=n_trees, max_features=5)

    # Fit the model
    print(f'Fitting model with {n_trees} trees')
    GBC.fit(X_train, y_train)
    y_pred = GBC.predict(X_test)

    # Get the error
    error = 1.0 - accuracy_score(y_test, y_pred)

    # Store it
    error_list.append(pd.Series({'n_trees': n_trees, 'error': error}))

GBC_oob_df = pd.concat(error_list, axis=1).T.set_index('n_trees')

GBC_oob_df
```

In [ ]:
```
start = timer()
GBC = GradientBoostingClassifier(n_estimators=150, max_features=5).fit(X_train, y_tr
end = timer()
GBC_time = end - start
print('Time Elapsed (sec):', GBC_time)
```

In [ ]:
```
RocCurve(GBC)
```

In [ ]:
```
GBC_threshold = 0.75
classif_report('Gradient Boosting Classifier',GBC, GBC_threshold)
```

ADABoost

In [ ]:
```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
ABC = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1))
```

In [ ]:
```
from sklearn.ensemble import GradientBoostingClassifier

error_list = list()

# Iterate through various possibilities for number of trees
tree_list = [15, 20, 30, 40, 50, 100, 150, 200, 300, 400]
for n_trees in tree_list:

    # Initialize the ADAboost classifier
    ABC = AdaBoostClassifier(DecisionTreeClassifier(), n_estimators=n_trees, learnin

    # Fit the model
    print(f'Fitting model with {n_trees} trees')
    ABC.fit(X_train, y_train)
    y_pred = ABC.predict(X_test)

    # Get the error
    error = 1.0 - accuracy_score(y_test, y_pred)

    # Store it
    error_list.append(pd.Series({'n_trees': n_trees, 'error': error}))

ABC_oob_df = pd.concat(error_list, axis=1).T.set_index('n_trees')

ABC_oob_df
```

In [ ]:
```
oob_df = pd.concat([rf_oob_df.rename(columns={'oob':'RandomForest'}),
                    et_oob_df.rename(columns={'oob':'ExtraTrees'}),
                    GBC_oob_df.rename(columns={'error':'Gradient Boosted'}),
                    ABC_oob_df.rename(columns={'error':'ADA Boosted'})], axis=1)

oob_df
```

In [ ]:
```
sns.set_context('talk')
sns.set_style('white')

ax = oob_df.plot(marker='o', figsize=(14, 7), linewidth=5)
ax.set(ylabel='out-of-bag error');
```

In [ ]:
```
param_grid = {'learning_rate': [0.001, 0.01, 0.1, 0.3]}

start = timer()
ABC = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1), n_estimators=100)
ABC_grid = GridSearchCV(ABC,
                        param_grid=param_grid,
                        n_jobs=-1,scoring= 'f1').fit(X_train,y_train)

end = timer()
ABC_time = end - start
print('Time Elapsed (sec):', ABC_time)
ABC_grid.best_params_
```

In [ ]:
```
RocCurve(ABC_grid)
```

In [ ]:
```
classif_report('Ada Boost Classifier',ABC_grid)
```

Voting Class

In [ ]:
```
from sklearn.ensemble import VotingClassifier

# The combined model--logistic regression and gradient boosted trees
estimators = [('logit_reg', logit_reg), ('Knn', knn_grid), ('GBC', GBC), ('Extra Tre

start = timer()
# Though it wasn't done here, it is often desirable to train
# this model using an additional hold-out data set and/or with cross validation
VC = VotingClassifier(estimators, voting='soft')
VC = VC.fit(X_train, y_train)

end = timer()
VC_time = end - start
print('Time Elapsed (sec):', VC_time)
```

In [ ]:
```
RocCurve(VC)
```

In [ ]:
```
VC_threshold = 0.75
classif_report('Voting', VC, VC_threshold)
```

Choosing the best model

In [ ]:
```
# Logit, knn, svc have unique threshold values for optimal F1-score
dummy_pred = dummy.predict(X_test)
logit_reg_pred = np.where(logit_reg.predict_proba(X_test)[:,1]>=logit_threshold, 1,
knn_grid_pred = np.where(knn_grid.predict_proba(X_test)[:,1]>=knn_threshold, 1, 0)
SVC_grid_pred = np.where(SVC_grid.predict_proba(X_test)[:,1]>=SVC_threshold, 1, 0)
rfc_grid_pred = rfc_grid.predict(X_test)
EF_pred = np.where(SVC_grid.predict_proba(X_test)[:,1]>=EF_threshold, 1, 0)
GBC_pred = np.where(SVC_grid.predict_proba(X_test)[:,1]>=GBC_threshold, 1, 0)
ABC_grid_pred = ABC_grid.predict(X_test)
VC_pred = np.where(SVC_grid.predict_proba(X_test)[:,1]>=VC_threshold, 1, 0)

model_pred = [dummy_pred, logit_reg_pred, knn_grid_pred, SVC_grid_pred, rfc_grid_pre
time_elapsed = [dummy_time,logit_reg_time, knn_time, SVC_time, rfc_time, EF_time, GB
```

In [ ]:
```
dummy_pred_proba = dummy.predict_proba(X_test)[:,1]
logit_reg_pred_proba = logit_reg.predict_proba(X_test)[:,1]
knn_grid_pred_proba = knn_grid.predict_proba(X_test)[:,1]
SVC_grid_pred_proba = SVC_grid.predict_proba(X_test)[:,1]
rfc_grid_pred_proba = rfc_grid.predict_proba(X_test)[:,1]
EF_pred_proba = EF.predict_proba(X_test)[:,1]
GBC_pred_proba = GBC.predict_proba(X_test)[:,1]
ABC_grid_pred_proba = ABC_grid.predict_proba(X_test)[:,1]
VC_pred_proba = VC.predict_proba(X_test)[:,1]

model_pred_proba = [dummy_pred_proba, logit_reg_pred_proba, knn_grid_pred_proba, SVC
```

In [ ]:
```
AUC = []
for pred in model_pred_proba:
    score = roc_auc_score(y_test, pred)
    AUC.append(round(score,3))
```

```
f1score = []
for pred in model_pred:
    score = f1_score(y_test, pred)
    f1score.append(round(score,3))

precision = []
for pred in model_pred:
    score = precision_score(y_test, pred)
    precision.append(round(score,3))

recall = []
for pred in model_pred:
    score = recall_score(y_test, pred)
    recall.append(round(score,3))

accuracy = []
for pred in model_pred:
    score = accuracy_score(y_test, pred)
    accuracy.append(round(score,3))
```

In [ ]:

```
model_metrics = pd.DataFrame({'AUC':AUC, 'F1':f1score, 'Precision': precision, 'Reca
model_metrics
```

In [ ]: