ML approach

Decision trees frequently perform well on imbalanced data. In modern machine learning, tree ensembles, such as Random Forest, almost always outperform singular decision trees. Random Forests is selected because its performance compared with other machine learning techniques is the best in terms of classification metrics and low upper bound error probability. The analysis is explained in details in the following sections. The appendix contains complete code to:

- **Create train and test sets.**
- **Create a Random Forest classifier (including setting class weights).**
  To combat severe class imbalance of datasets, Random Forest with class weighting for imbalanced classification will be used.
- **Tune Hyperparameter by Grid Search**
  The exhaustive search for the optimal number of trees and the minimum samples of leaf is used. The minimum samples of leaf of all models for all the datasets are all set 2. The number of trees for Dataset a is 125, while those for other datasets are 250.
- **Evaluate the model using two metrics (accuracy and AUC)**
- **Evaluate the mode's upper bound probability of misclassification**

Methodology for validation

To begin with the ML-based classification, 80 percent of each dataset was used for training, with 5-fold cross-validation, and 20 percent of the datasets has been used for testing. Both linear and nonlinear classifiers have been tested based on their classification metrics and upper bound probability to select the best ML model. Random Forest (RF) will be compared with other nonlinear models, including Penalized Support Vector Machine (SVM), XGBoost (XGB), and Neural Network (NN), and a linear model, Logistic regression (LR), accompanied by SMOTE resampling. The accuracy and AUC are used to measure the classification performance of each model.

Methodology for upper bound estimation for the misclassification error

The probability of the error or misclassification can be calculated with (1) [1]

$$P(error) = \int_{-\infty}^{+\infty} P(error|x)\, P(x)\, dx$$

If we estimate the probability density function of each class, by dividing the error probability into two regions as R_1 and R_2, the probability of error can be written as:

$$\int_{R_1} P(x|Class\ 0)\, P(Class\ 0)\, dx + \int_{R_2} P(x|Class\ 1)\, P(Class\ 1)\, dx$$

Fukunaga, K. (1990) showed that the upper bound error can be calculated using probability density function (PDF)-based distances, which usually rely on mean and variance distance. For

safety concern, it is critical to consider the worst-case scenario. In this case, Chernoff Distance (CD), as a Probability Density Function (PDF)-based distance measure, is selected to test an upper bound for the probability of the next misclassification error [2].

$$P(error) = P(Class\ 0)^{\lambda}\ P(Class\ 1)^{1-\lambda}$$

$$\int_{-\infty}^{+\infty} P(Class\ 0)^{\lambda}\ P(Class\ 1)^{1-\lambda}\ dx$$

Dataset a: Table 1 compares the upper bound error probability estimation based on Chernoff Distance. We observe that XGB model reports the lowest upper bound, followed by RF classifier.

Table 1: Comparison of Chernoff Distance for Dataset a

	LR	SVM	RF	XGB	NN
CD	0. 116673	0.032613	0.018716	0.008076	0.020670

Table 2 represents the accuracy and AUC of each ML model for Dataset a. RF model has the best accuracy and AUC overall.

Table 2: Comparison of classification metrics for Dataset a

	LR	SVM	RF	XGB	NN
Accuracy	0.485	0.895	0.945	0.925	0.87
AUC	0.483	0.878	0.931	0.905	0.914

Dataset b: Similar to the Dataset a, the proposed Chernoff Distance can be applied to the Dataset b. Seeing from Table 3, SVM and RF models have better estimation for upper bound probability for the larger Dataset b.

Table 3: Comparison of Chernoff Distance for Dataset b

	LR	SVM	RF	XGB	NN
CD	0. 002134	0.000859	0.001198	0.001317	0.001770

Table 4 represents the accuracy and AUC of each ML model for Dataset b. RF model has the best accuracy and AUC overall.

Table 4: Comparison of classification metrics for Dataset b

	LR	SVM	RF	XGB	NN
Accuracy	0.859	0.905	0.989	0.982	0.859
AUC	0.860	0.906	0.989	0.982	0.952

Dataset c: The Dataset c has similar statistical characteristics with the Dataset b. Table 5 provides the upper bound error probability of different classifiers. As indicated in the table, the best estimation is related to SVM model. The following RF and XGB models also have relatively low upper bound probability.

Table 5: Comparison of Chernoff Distance for Dataset c

	LR	SVM	RF	XGB	NN
CD	0. 115958	0.000660	0.001186	0.001110	0.010075

Table 6 represents the accuracy and AUC of each ML models for Dataset c. RF model has the best accuracy and AUC overall.

Table 6: Comparison of classification metrics for Dataset c

	LR	SVM	RF	XGB	NN
Accuracy	0.619	0.557	0.994	0.974	0.818
AUC	0.566	0.728	0.983	0.971	0.930

Overall, it is noticeable that RF have the best classification performance and relatively low Chernoff Distance, so it is selected as a reliable safety prediction model.

Scalability to Higher Dimension

The Chernoff Distance can be calculated in a higher dimension as long as mean and variance distances of every dimension is provided. In order to load high-dimension data into ML models, the solution to this flexibility should be considered. The two feasible approaches to adapt ML models to higher dimension are:

1. Increase the size of input layer (limited to Neural Network): since it provides flexibility to adjust input layer without changing hidden layer and output layer, it could be a simple way to allow the model to receive higher dimension of data. However, the model could not be a perfect fit to the new dimension of data since the data can have more noises than before.
2. Dimension reduction: reduce the dimension of data before loaded into the model. This is an efficient way to identify the most important latent variable from the original data. However, concerning safety, we aim to catch as much information as possible. Tiny information loss, particularly about danger detection, can lead to an accident.

Reference

[1] S. Theodoridis and K. Koutroumbas, Pattern Recognition. Elsevier Inc., 2009.
[2] K. Fukunaga, Introduction to statistical pattern recognition. Elsevier, 2013.

Appendix

The following Python code is accessible on my Github
https://github.com/cory1219/Siemens-Challenge-/blob/main/Siemens%20final%20script.py

```
import os
import tempfile
import random
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from imblearn.over_sampling import SMOTE
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
import tensorflow as tf
from tensorflow import keras
from tensorflow.python.keras import backend as K
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from sklearn.metrics import accuracy_score, roc_auc_score
from bayes_opt import BayesianOptimization

#import datasets
random.seed(2021)
df1 = pd.read_excel('trainingdata_a.xls')
df2 = pd.read_excel('trainingdata_b.xls')
df3 = pd.read_excel('trainingdata_c.xls')

#create functions of spliting data for LOG, SVM, RF, XGBoost
def split(df):
    X_df = df[['x_i1', 'x_i2']]
    y_df = df[['l_i']]
    X_train, X_test, y_train, y_test = train_test_split(
        X_df, y_df, test_size=0.2, random_state=2021, stratify = y_df)
    return X_train, X_test, y_train, y_test, y_df

X_train_1, X_test_1, y_train_1, y_test_1, y_df_1 = split(df1) #dataset a
X_train_2, X_test_2, y_train_2, y_test_2, y_df_2 = split(df2) #dataset b
X_train_3, X_test_3, y_train_3, y_test_3, y_df_3 = split(df3) #dataset c
```

```
#Random forest
def rf(X_train, y_train, X_test, y_test):
   rf = RandomForestClassifier(class_weight='balanced') #class weighting
   params_rf = {'n_estimators': [25, 100, 125, 250],
            'min_samples_leaf': [2, 10, 20],
            'random_state': [2021]}
   grid_rf = GridSearchCV(estimator=rf,
                     param_grid=params_rf,
                     scoring='roc_auc',
                     cv=5,
                     verbose=1) #hyperparameter tuning
   grid_rf.fit(X_train, y_train) #train
   print(grid_rf.best_score_)
   best_rf = grid_rf.best_estimator_
   y_pred_rf = best_rf.predict(X_test) #predict
   plot_confusion_matrix(best_rf, X_test, y_test)
   accuracy = accuracy_score(y_test, y_pred_rf) #accuracy
   auc = roc_auc_score(y_test, y_pred_rf) #auc
   parameter = grid_rf.best_params_
   print("Accuracy",accuracy_score(y_test, y_pred_rf))
   print("ROCAUC",roc_auc_score(y_test, y_pred_rf))
   return y_pred_rf, accuracy, auc, parameter

y_pred_rf_1, accuracy_rf_1, auc_rf_1, parameter_rf_1 = rf(X_train_1, y_train_1, X_test_1,
y_test_1) #dataset a
y_pred_rf_2, accuracy_rf_2, auc_rf_2, parameter_rf_2 = rf(X_train_2, y_train_2, X_test_2,
y_test_2) #dataset b
y_pred_rf_3, accuracy_rf_3, auc_rf_3, parameter_rf_3 = rf(X_train_3, y_train_3, X_test_3,
y_test_3) #dataset c

#PDF-based upper bounds
def chernoff_distance(s, means, variances, univariate = False):
   """ Returns the Chernoff Distance, as defined in (3.150), p.98 of
      Fukunaga, 1990, Introduction to Statistical Pattern Recognition, 2nd Edition
   """
   from math import log
   from math import sqrt
   from math import pow

   from numpy.linalg import det
   from numpy.linalg import inv
   from numpy import array

   if univariate:
```

```
        part1 = 0.25 * log(0.25 * (variances[0] / variances[1] + variances[1] / variances[0]) + 0.5)
        part2 = 0.25 * pow(means[0] - means[1], 2) / (variances[0] + variances[1])
        return part1 + part2

    mean_diff = array(means[0]) - array(means[1])
    var_avg = (array(variances[0]) + array(variances[1])) * 0.5
    ln_coeff = 0.5 * log(det(var_avg) / sqrt(det(array(variances[0])) * det(array(variances[1]))))

    return 0.125 * np.matmul(np.matmul(mean_diff.T, inv(var_avg)), mean_diff) + ln_coeff

def upper_bound(X_train, y_train, X_test, y_pred):
    return chernoff_distance(
    0.5,
    [(X_train[y_train.l_i == 1].x_i1.mean(), X_train[y_train.l_i == 1].x_i2.mean()),
     (X_test[y_pred == 1].x_i1.mean(), X_test[y_pred == 1].x_i2.mean())],
    [[X_train[y_train.l_i == 1].x_i1.var(), 0], [0, X_train[y_train.l_i == 1].x_i2.var()]],
     [[X_test[y_pred == 1].x_i1.var(), 0], [0, X_test[y_pred == 1].x_i2.var()]]])

#upper bound of RF
upper_bound(X_train_1, y_train_1, X_test_1, y_pred_rf_1) #dataset a
upper_bound(X_train_2, y_train_2, X_test_2, y_pred_rf_2) #dataset b
upper_bound(X_train_3, y_train_3, X_test_3, y_pred_rf_3) #dataset c

#comparison with other ML models
#Logistic regression with SMOTE
def log_smote(X_train, y_train, X_test, y_test):
    sm = SMOTE(random_state=2021, sampling_strategy = "minority")
    X_train, y_train = sm.fit_resample(X_train, y_train)
    print(y_train.l_i.sum())
    lr_sm = LogisticRegression(random_state=2021).fit(X_train, y_train)
    y_pred_log = lr_sm.predict(X_test)
    #y_pred_prob = lr_sm.predict_proba(X_test)[:,1]
    plot_confusion_matrix(lr_sm, X_test, y_test)
    accuracy = accuracy_score(y_test, y_pred_log)
    auc = roc_auc_score(y_test, y_pred_log)
    print("Accuracy",accuracy_score(y_test, y_pred_log))
    print("ROCAUC",roc_auc_score(y_test, y_pred_log))
    return y_pred_log, accuracy, auc

y_pred_log_1, accuracy_log_1, auc_log_1 = log_smote(X_train_1, y_train_1, X_test_1,
y_test_1)
y_pred_log_2, accuracy_log_2, auc_log_2 = log_smote(X_train_2, y_train_2, X_test_2,
y_test_2)
```

```
y_pred_log_3, accuracy_log_3, auc_log_3 = log_smote(X_train_3, y_train_3, X_test_3,
y_test_3)

upper_bound(X_train_1, y_train_1, X_test_1, y_pred_log_1) #upper bound of log for dataset a
upper_bound(X_train_2, y_train_2, X_test_2, y_pred_log_2) #upper bound of log for dataset b
upper_bound(X_train_3, y_train_3, X_test_3, y_pred_log_3) #upper bound of log for dataset c

#Penalized SVM
def svm_pen(X_train, y_train, X_test, y_test):
   svm_p = SVC(class_weight = 'balanced', probability=True)
   svm_p.fit(X_train, y_train)
   y_pred_svm = svm_p.predict(X_test)
   plot_confusion_matrix(svm_p, X_test, y_test)
   accuracy = accuracy_score(y_test, y_pred_svm)
   auc = roc_auc_score(y_test, y_pred_svm)
   print("Accuracy",accuracy_score(y_test, y_pred_svm))
   print("ROCAUC",roc_auc_score(y_test, y_pred_svm))
   return y_pred_svm, accuracy, auc

y_pred_svm_1, accuracy_svm_1, auc_svm_1 = svm_pen(X_train_1, y_train_1, X_test_1,
y_test_1)
y_pred_svm_2, accuracy_svm_2, auc_svm_2 = svm_pen(X_train_2, y_train_2, X_test_2,
y_test_2)
y_pred_svm_3, accuracy_svm_3, auc_svm_3 = svm_pen(X_train_3, y_train_3, X_test_3,
y_test_3)

upper_bound(X_train_1, y_train_1, X_test_1, y_pred_svm_1) #upper bound of svm for dataset
a
upper_bound(X_train_2, y_train_2, X_test_2, y_pred_svm_2) #upper bound of svm for dataset
b
upper_bound(X_train_3, y_train_3, X_test_3, y_pred_svm_3) #upper bound of svm for dataset
c

#XGBoost
def xgb(X_train, y_train, X_test, y_test, y_df):
   est = (y_df[y_df['l_i'] == 0].count() / y_df[y_df['l_i'] == 1].count()).to_numpy().item()
   xgb = XGBClassifier(scale_pos_weight=est)
   params_xgb = {'n_estimators': [3, 5, 10],
            'num_boost_round': [5, 10, 15],
            'max_depth': [2, 5, 10],
            'eta': [0.001, 0.01, 0.1],
            'random_state': [2021]}
   grid_xgb = GridSearchCV(estimator=xgb,
                param_grid=params_xgb,
```

```
                scoring='roc_auc',
                cv=5,
                verbose=1)
    grid_xgb.fit(X_train, y_train)
    print(grid_xgb.best_score_)
    best_xgb = grid_xgb.best_estimator_
    y_pred_xgb = best_xgb.predict(X_test)
    plot_confusion_matrix(best_xgb, X_test, y_test)
    accuracy = accuracy_score(y_test, y_pred_xgb)
    auc = roc_auc_score(y_test, y_pred_xgb)
    print("Accuracy",accuracy_score(y_test, y_pred_xgb))
    print("ROCAUC",roc_auc_score(y_test, y_pred_xgb))
    return y_pred_xgb, accuracy, auc

y_pred_xgb_1, accuracy_xgb_1, auc_xgb_1 = xgb(X_train_1, y_train_1, X_test_1, y_test_1,
y_df_1)
y_pred_xgb_2, accuracy_xgb_2, auc_xgb_2 = xgb(X_train_2, y_train_2, X_test_2, y_test_2,
y_df_2)
y_pred_xgb_3, accuracy_xgb_3, auc_xgb_3 = xgb(X_train_3, y_train_3, X_test_3, y_test_3,
y_df_3)

upper_bound(X_train_1, y_train_1, X_test_1, y_pred_xgb_1) #upper bound of xgb for dataset a
upper_bound(X_train_2, y_train_2, X_test_2, y_pred_xgb_2) #upper bound of xgb for dataset b
upper_bound(X_train_3, y_train_3, X_test_3, y_pred_xgb_3) #upper bound of xgb for dataset c

#Neural networks weighted
tf.random.set_seed(2021)
def separate(df): #create train, validation, test datasets
    neg, pos = np.bincount(df['l_i'])
    total = neg + pos
    print('Examples:\n Total: {}\n Positive: {} ({:.2f}%)\n'.format(
        total, pos, 100*pos/total))
    cleaned_df = df.copy()
    train_df, test_df = train_test_split(cleaned_df, test_size=0.2, random_state=2021,
stratify=cleaned_df['l_i'])
    base_df, val_df = train_test_split(train_df, test_size=0.2, random_state=2021,
stratify=train_df['l_i'])
    train_labels = np.array(base_df.pop('l_i'))
    val_labels = np.array(val_df.pop('l_i'))
    test_labels = np.array(test_df.pop('l_i'))

    train_features = np.array(base_df[['x_i1', 'x_i2']])
    val_features = np.array(val_df[['x_i1', 'x_i2']])
    test_features = np.array(test_df[['x_i1', 'x_i2']])
```

```
    initial_bias = np.log([pos/neg])
    train_df_labels = train_df[['l_i']]

    weight_for_0 = (1/neg)*(total)/2.0
    weight_for_1 = (1/pos)*(total)/2.0
    class_weight = {0: weight_for_0, 1: weight_for_1}
    class_weight
    print('Weight for class 0: {:.2f}'.format(weight_for_0))
    print('Weight for class 1: {:.2f}'.format(weight_for_1))
    return train_df, test_df, train_labels, train_features, val_features, val_labels, test_labels,
test_features, initial_bias, class_weight, train_df_labels

train_df_1, test_df_1, train_labels_1, train_features_1, val_features_1, val_labels_1,
test_labels_1, test_features_1, initial_bias_1, class_weight_1, train_df_labels_1 = separate(df1)
#dataset a
train_df_2, test_df_2, train_labels_2, train_features_2, val_features_2, val_labels_2,
test_labels_2, test_features_2, initial_bias_2, class_weight_2, train_df_labels_2 = separate(df2)
#dataset b
train_df_3, test_df_3, train_labels_3, train_features_3, val_features_3, val_labels_3,
test_labels_3, test_features_3, initial_bias_3, class_weight_3, train_df_labels_3 = separate(df3)
#dataset c

early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_auc',
    verbose=1,
    patience=10,
    mode='max',
    restore_best_weights=True)

def plot_cm(labels, predictions, p=0.5): #confusion matrix plot
  cm = confusion_matrix(labels, predictions > p)
  plt.figure(figsize=(5,5))
  sns.heatmap(cm, annot=True, fmt="d")
  plt.title('Confusion matrix @{:.2f}'.format(p))
  plt.ylabel('Actual label')
  plt.xlabel('Predicted label')

  print('Legitimate Transactions Detected (True Negatives): ', cm[0][0])
  print('Legitimate Transactions Incorrectly Detected (False Positives): ', cm[0][1])
  print('Fraudulent Transactions Missed (False Negatives): ', cm[1][0])
  print('Fraudulent Transactions Detected (True Positives): ', cm[1][1])
  print('Total Fraudulent Transactions: ', np.sum(cm[1]))
```

```
def nn(train_df, test_df, train_labels, train_features, val_features, val_labels, test_labels,
test_features, initial_bias, class_weight): #classifier
   def get_model_opt(units=16, dropout=0.5):
      output_bias = tf.keras.initializers.Constant(initial_bias)
      model = keras.Sequential([
      keras.layers.Dense(int(units),
                  activation='relu',
                  input_shape=(train_features.shape[-1],)),
      keras.layers.Dropout(dropout),
      keras.layers.Dense(1, activation='sigmoid', bias_initializer=output_bias)])
      return model

   def fit_with(units, dropout, learning_rate, epochs, batch_size):
      model = get_model_opt(units, dropout)

      model.compile(
         optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
         loss=keras.losses.BinaryCrossentropy(),
         metrics=[
            keras.metrics.BinaryAccuracy(name='accuracy'),
            keras.metrics.AUC(name='auc')
            ])

      initial_weights = os.path.join(tempfile.mkdtemp(), 'initial_weights')
      model.save_weights(initial_weights)
      model.load_weights(initial_weights)

      history = model.fit(x=train_features,
                  y=train_labels,
                  epochs=int(epochs),
                  batch_size=int(batch_size),
                  validation_data=(val_features,val_labels),
                  callbacks=[early_stopping],
                  class_weight=class_weight)
      # Evaluate the model with the eval dataset.
      accuracy = history.history['accuracy'][-1]

      K.clear_session()
      tf.compat.v1.reset_default_graph()

      # Return the accuracy.
      return accuracy

   # Bounded region of parameter space
```

```
pbounds = {
   'units': (16,512),
   'dropout': (0.1,0.8),
   'batch_size': (32, 128),
   'epochs': (25, 100),
   'learning_rate': (1e-3, 1e-1)}

optimizer = BayesianOptimization(
f=fit_with,
pbounds=pbounds,
verbose=2,
random_state=2021
)

optimizer.maximize(init_points=10, n_iter=10,)


for i, res in enumerate(optimizer.res):
   print("Iteration {}: \n\t{}".format(i, res))

print(optimizer.max)

def get_nnw_model(units, dropout, learning_rate, epochs, batch_size):
   model = get_model_opt(units, dropout)
   model.compile(
      optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
      loss=keras.losses.BinaryCrossentropy(),
      metrics=[
         keras.metrics.BinaryAccuracy(name='accuracy'),
         keras.metrics.AUC(name='auc')
         ])

   initial_weights = os.path.join(tempfile.mkdtemp(), 'initial_weights')
   model.save_weights(initial_weights)
   model.load_weights(initial_weights)

   model.fit(x=train_features,
           y=train_labels,
           epochs=int(epochs),
           batch_size=int(batch_size),
           validation_data=(val_features,val_labels),
           callbacks=[early_stopping],
           class_weight=class_weight)
   return model
```

```
    model_weighted = get_nnw_model(
        optimizer.max['params']['units'],
        optimizer.max['params']['dropout'],
        optimizer.max['params']['learning_rate'],
        optimizer.max['params']['epochs'],
        optimizer.max['params']['batch_size']
        )
    train_predictions_baseline_weighted = np.where(model_weighted.predict(train_features,
batch_size=128)>=0.5, 1, 0)
    test_predictions_baseline_weighted = np.where(model_weighted.predict(test_features,
batch_size=128)>=0.5, 1, 0)
    baseline_results_weighted = model_weighted.evaluate(test_features, test_labels,
                            batch_size=128, verbose=0)
    for name, value in zip(model_weighted.metrics_names, baseline_results_weighted):
        print(name, ': ', value)
        print()

    plot_cm(test_labels, test_predictions_baseline_weighted)
    return test_predictions_baseline_weighted, baseline_results_weighted

test_predictions_baseline_weighted_1, baseline_results_weighted_1 = nn(train_df_1,
test_df_1, train_labels_1, train_features_1, val_features_1, val_labels_1, test_labels_1,
test_features_1, initial_bias_1, class_weight_1)
test_predictions_baseline_weighted_2, baseline_results_weighted_2 = nn(train_df_2,
test_df_2, train_labels_2, train_features_2, val_features_2, val_labels_2, test_labels_2,
test_features_2, initial_bias_2, class_weight_2)
test_predictions_baseline_weighted_3, baseline_results_weighted_3 = nn(train_df_3,
test_df_3, train_labels_3, train_features_3, val_features_3, val_labels_3, test_labels_3,
test_features_3, initial_bias_3, class_weight_3)

upper_bound(train_df_1, train_df_labels_1, test_df_1, test_predictions_baseline_weighted_1)
#upper bound of nn for dataset a
upper_bound(train_df_2, train_df_labels_2, test_df_2, test_predictions_baseline_weighted_2)
#upper bound of nn for dataset b
upper_bound(train_df_3, train_df_labels_3, test_df_3, test_predictions_baseline_weighted_3)
#upper bound of nn for dataset c
```