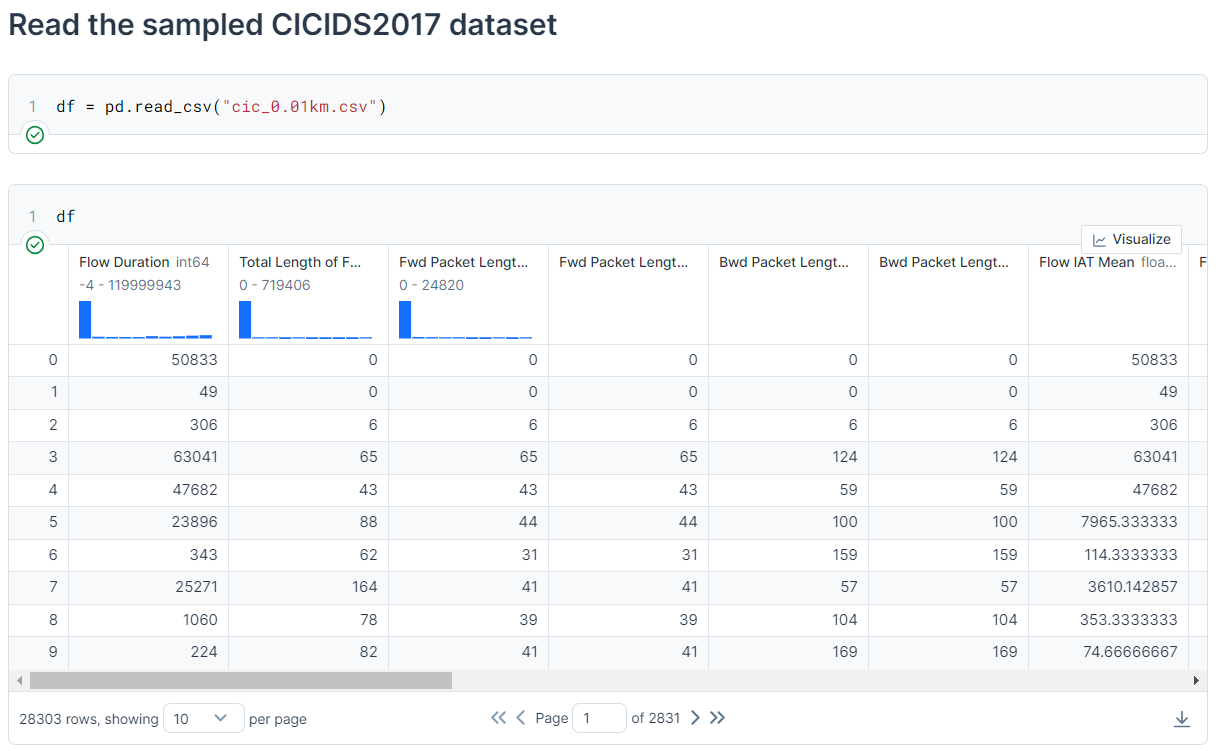
**Automated Machine Learning**

**Dataset 1: CICIDS2017**

A subset of the network traffic data randomly sampled from the [CICIDS2017 dataset](https://www.unb.ca/cic/datasets/ids-2017.html).

The Canadian Institute for Cybersecurity Intrusion Detection System 2017 (CICIDS2017) dataset has the most updated network threats. The CICIDS2017 dataset is close to real-world network data since it has a large amount of network traffic data, a variety of network features, various types of attacks, and highly imbalanced classes.



**1. Automated Data Pre-Processing**

**Automated Transformation/Encoding**

Automatically identify and transform string/text features into numerical features to make the data more readable by ML models

# Define the automated data encoding function

def Auto\_Encoding(df):

    cat\_features=[x for x in df.columns if df[x].dtype=="object"] ## Find string/text features

    le=LabelEncoder()

    for col in cat\_features:

        if col in df.columns:

            i = df.columns.get\_loc(col)

            # Transform to numerical features

            df.iloc[:,i] = df.apply(lambda i:le.fit\_transform(i.astype(str)), axis=0, result\_type='expand')

    return df

df=Auto\_Encoding(df)

**Automated Imputation**

Detect and impute missing values to improve data quality

# Define the automated data imputation function

def Auto\_Imputation(df):

    if df.isnull().values.any() or np.isinf(df).values.any(): # if there is any empty or infinite values

        df.replace([np.inf, -np.inf], np.nan, inplace=True)

        df.fillna(0, inplace = True)  # Replace empty values with zeros; there are other imputation methods discussed in the paper

    return df

df=Auto\_Imputation(df)

**Automated normalization**

Normalize the range of features to a similar scale to improve data quality

def Auto\_Normalization(df):

    stat, p = shapiro(df)

    print('Statistics=%.3f, p=%.3f' % (stat, p))

    # interpret

    alpha = 0.05

    numeric\_features = df.drop(['Labelb'],axis = 1).dtypes[df.dtypes != 'object'].index

    # The selection strategy is based on the following article:

    # https://medium.com/@kumarvaishnav17/standardization-vs-normalization-in-machine-learning-3e132a19c8bf

    # Check if the data distribution follows a Gaussian/normal distribution

    # If so, select the Z-score normalization method; otherwise, select the min-max normalization

    # Details are in the paper

    if p > alpha:

        print('Sample looks Gaussian (fail to reject H0)')

        df[numeric\_features] = df[numeric\_features].apply(

            lambda x: (x - x.mean()) / (x.std()))

        print('Z-score normalization is automatically chosen and used')

    else:

        print('Sample does not look Gaussian (reject H0)')

        df[numeric\_features] = df[numeric\_features].apply(

            lambda x: (x - x.min()) / (x.max()-x.min()))

        print('Min-max normalization is automatically chosen and used')

    return df

df=Auto\_Normalization(df)

**Train-test split**

Split the dataset into the training and the test set

X = df.drop(['Labelb'],axis=1)

y = df['Labelb']

# Here we used the 80%/20% split, it can be changed based on specific tasks

#X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y, train\_size = 0.8, test\_size = 0.2, shuffle=False,random\_state = 0)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y, train\_size = 0.8, test\_size = 0.2,random\_state = 0)

**Automated data balancing**

Generate minority class samples to solve class-imbalance and improve data quality.  
Synthetic Minority Over-sampling Technique (SMOTE) method is used.

pd.Series(y\_train).value\_counts()

Labelb

0 18126

1 4516

Name: count, dtype: int64

# For binary data (can be modified for multi-class data with same logic)

def Auto\_Balancing(X\_train, y\_train):

    number0 = pd.Series(y\_train).value\_counts().iloc[0]

    number1 = pd.Series(y\_train).value\_counts().iloc[1]

    if number0 > number1:

        nlarge = number0

    else:

        nlarge = number1

    # evaluate whether the incoming dataset is imbalanced (the abnormal/normal ratio is smaller than a threshold (e.g., 50%))

    if (number1/number0 > 1.5) or (number0/number1 > 1.5):

        smote=SMOTE(n\_jobs=-1,sampling\_strategy={0:nlarge, 1:nlarge})

        X\_train, y\_train = smote.fit\_resample(X\_train, y\_train)

    return X\_train, y\_train

X\_train, y\_train = Auto\_Balancing(X\_train, y\_train)

pd.Series(y\_train).value\_counts()

Labelb

0 18126

1 18126

Name: count, dtype: int64

**Model learning (for Comparison)**

%%time

lg = lgb.LGBMClassifier(verbose = -1)

lg.fit(X\_train,y\_train)

t1=time.time()

predictions = lg.predict(X\_test)

t2=time.time()

print("Accuracy: "+str(round(accuracy\_score(y\_test,predictions),5)\*100)+"%")

print("Precision: "+str(round(precision\_score(y\_test,predictions),5)\*100)+"%")

print("Recall: "+str(round(recall\_score(y\_test,predictions),5)\*100)+"%")

print("F1-score: "+str(round(f1\_score(y\_test,predictions),5)\*100)+"%")

print("Time: "+str(round((t2-t1)/len(y\_test)\*1000000,5)))

Accuracy: 99.788%

Precision: 99.37899999999999%

Recall: 99.556%

F1-score: 99.467%

Time: 2.93241

CPU times: user 548 ms, sys: 5.26 ms, total: 554 ms

Wall time: 587 ms

%%time

rf = RandomForestClassifier()

rf.fit(X\_train,y\_train)

t1=time.time()

predictions = rf.predict(X\_test)

t2=time.time()

print("Accuracy: "+str(round(accuracy\_score(y\_test,predictions),5)\*100)+"%")

print("Precision: "+str(round(precision\_score(y\_test,predictions),5)\*100)+"%")

print("Recall: "+str(round(recall\_score(y\_test,predictions),5)\*100)+"%")

print("F1-score: "+str(round(f1\_score(y\_test,predictions),5)\*100)+"%")

print("Time: "+str(round((t2-t1)/len(y\_test)\*1000000,5)))

Accuracy: 99.717%

Precision: 99.465%

Recall: 99.111%

F1-score: 99.288%

Time: 9.41595

CPU times: user 3.35 s, sys: 9.49 ms, total: 3.35 s

Wall time: 3.41 s

%%time

nb = GaussianNB()

nb.fit(X\_train,y\_train)

t1=time.time()

predictions = nb.predict(X\_test)

t2=time.time()

print("Accuracy: "+str(round(accuracy\_score(y\_test,predictions),5)\*100)+"%")

print("Precision: "+str(round(precision\_score(y\_test,predictions),5)\*100)+"%")

print("Recall: "+str(round(recall\_score(y\_test,predictions),5)\*100)+"%")

print("F1-score: "+str(round(f1\_score(y\_test,predictions),5)\*100)+"%")

print("Time: "+str(round((t2-t1)/len(y\_test)\*1000000,5)))

Accuracy: 75.358%

Precision: 44.507999999999996%

Recall: 97.244%

F1-score: 61.065999999999995%

Time: 0.47991

CPU times: user 22.2 ms, sys: 0 ns, total: 22.2 ms

Wall time: 29.2 ms

%%time

knn = KNeighborsClassifier()

knn.fit(X\_train,y\_train)

t1=time.time()

predictions = knn.predict(X\_test)

t2=time.time()

print("Accuracy: "+str(round(accuracy\_score(y\_test,predictions),5)\*100)+"%")

print("Precision: "+str(round(precision\_score(y\_test,predictions),5)\*100)+"%")

print("Recall: "+str(round(recall\_score(y\_test,predictions),5)\*100)+"%")

print("F1-score: "+str(round(f1\_score(y\_test,predictions),5)\*100)+"%")

print("Time: "+str(round((t2-t1)/len(y\_test)\*1000000,5)))

Accuracy: 98.834%

Precision: 95.844%

Recall: 98.4%

F1-score: 97.10499999999999%

Time: 164.67113

CPU times: user 900 ms, sys: 0 ns, total: 900 ms

Wall time: 943 ms

import tensorflow as tf

from keras.layers import Input,Dense,Dropout,BatchNormalization,Activation

from keras import Model

import keras.backend as K

import keras.callbacks as kcallbacks

from keras import optimizers

from keras.optimizers import Adam

from sklearn.model\_selection import GridSearchCV

from keras.wrappers.scikit\_learn import KerasClassifier

from keras.callbacks import EarlyStopping

def ANN(optimizer = 'sgd',neurons=16,batch\_size=1024,epochs=80,activation='relu',patience=8,loss='binary\_crossentropy'):

    K.clear\_session()

    inputs=Input(shape=(X.shape[1],))

    x=Dense(1000)(inputs)

    x=BatchNormalization()(x)

    x=Activation('relu')(x)

    x=Dropout(0.3)(x)

    x=Dense(256)(inputs)

    x=BatchNormalization()(x)

    x=Activation('relu')(x)

    x=Dropout(0.25)(x)

    x=Dense(2,activation='softmax')(x)

    model=Model(inputs=inputs,outputs=x,name='base\_nlp')

    model.compile(optimizer='adam',loss='categorical\_crossentropy')

#     model.compile(optimizer=Adam(lr = 0.01),loss='categorical\_crossentropy',metrics=['accuracy'])

    early\_stopping = EarlyStopping(monitor="loss", patience = patience)# early stop patience

    history = model.fit(X, pd.get\_dummies(y).values,

              batch\_size=batch\_size,

              epochs=epochs,

              callbacks = [early\_stopping],

              verbose=0) #verbose set to 1 will show the training process

    return model

%%time

ann = KerasClassifier(build\_fn=ANN, verbose=0)

ann.fit(X\_train,y\_train)

predictions = ann.predict(X\_test)

print("Accuracy: "+str(round(accuracy\_score(y\_test,predictions),5)\*100)+"%")

print("Precision: "+str(round(precision\_score(y\_test,predictions),5)\*100)+"%")

print("Recall: "+str(round(recall\_score(y\_test,predictions),5)\*100)+"%")

print("F1-score: "+str(round(f1\_score(y\_test,predictions),5)\*100)+"%")

print("Time: "+str(round((t2-t1)/len(y\_test)\*1000000,5)))

Accuracy: 94.559%

Precision: 81.207%

Recall: 94.489%

F1-score: 87.346%

Time: 164.67113

CPU times: user 27.2 s, sys: 3.42 s, total: 30.6 s

Wall time: 31 s

**2. Automated Feature Engineering**

Feature selection method 1: **Information Gain (IG)**, used to remove irrelevant features to improve model efficiency  
Feature selection method 2: **Pearson Correlation**, used to remove redundant features to improve model efficiency and accuracy

# Remove irrelevant features and select important features

def Feature\_Importance\_IG(data):

    features = data.drop(['Labelb'],axis=1).values  # "Label" should be changed to the target class variable name if different

    labels = data['Labelb'].values

    # Extract feature names

    feature\_names = list(data.drop(['Labelb'],axis=1).columns)

    # Empty array for feature importances

    feature\_importance\_values = np.zeros(len(feature\_names))

    model = lgb.LGBMRegressor(verbose = -1)

    model.fit(features, labels)

    feature\_importances = pd.DataFrame({'feature': feature\_names, 'importance': model.feature\_importances\_})

    # Sort features according to importance

    feature\_importances = feature\_importances.sort\_values('importance', ascending = False).reset\_index(drop = True)

    # Normalize the feature importances to add up to one

    feature\_importances['normalized\_importance'] = feature\_importances['importance'] / feature\_importances['importance'].sum()

    feature\_importances['cumulative\_importance'] = np.cumsum(feature\_importances['normalized\_importance'])

    cumulative\_importance=0.90 # Only keep the important features with cumulative importance scores>=90%. It can be changed.

    # Make sure most important features are on top

    feature\_importances = feature\_importances.sort\_values('cumulative\_importance')

    # Identify the features not needed to reach the cumulative\_importance

    record\_low\_importance = feature\_importances[feature\_importances['cumulative\_importance'] > cumulative\_importance]

    to\_drop = list(record\_low\_importance['feature'])

#     print(feature\_importances.drop(['importance'],axis=1))

    return to\_drop

# Remove redundant features

def Feature\_Redundancy\_Pearson(data):

    correlation\_threshold=0.90 # Only remove features with the redundancy>90%. It can be changed

    features = data.drop(['Labelb'],axis=1)

    corr\_matrix = features.corr()

    # Extract the upper triangle of the correlation matrix

    upper = corr\_matrix.where(np.triu(np.ones(corr\_matrix.shape), k = 1).astype(np.bool))

    # Select the features with correlations above the threshold

    # Need to use the absolute value

    to\_drop = [column for column in upper.columns if any(upper[column].abs() > correlation\_threshold)]

    # Dataframe to hold correlated pairs

    record\_collinear = pd.DataFrame(columns = ['drop\_feature', 'corr\_feature', 'corr\_value'])

    # Iterate through the columns to drop

    for column in to\_drop:

        # Find the correlated features

        corr\_features = list(upper.index[upper[column].abs() > correlation\_threshold])

        # Find the correlated values

        corr\_values = list(upper[column][upper[column].abs() > correlation\_threshold])

        drop\_features = [column for \_ in range(len(corr\_features))]

        # Record the information (need a temp df for now)

        temp\_df = pd.DataFrame.from\_dict({'drop\_feature': drop\_features,

                                         'corr\_feature': corr\_features,

                                         'corr\_value': corr\_values})

        record\_collinear = record\_collinear.append(temp\_df, ignore\_index = True)

#     print(record\_collinear)

    return to\_drop

def Auto\_Feature\_Engineering(df):

    drop1 = Feature\_Importance\_IG(df)

    dfh1 = df.drop(columns = drop1)

    drop2 = Feature\_Redundancy\_Pearson(dfh1)

    dfh2 = dfh1.drop(columns = drop2)

    return dfh2

def Feature\_Redundancy\_Pearson(data):

    correlation\_threshold=0.90 # Only remove features with the redundancy>90%. It can be changed

    features = data.drop(['Labelb'],axis=1)

    corr\_matrix = features.corr()

    # Extract the upper triangle of the correlation matrix

    upper = corr\_matrix.where(np.triu(np.ones(corr\_matrix.shape), k = 1).astype(np.bool))

    # Select the features with correlations above the threshold

    # Need to use the absolute value

    to\_drop = [column for column in upper.columns if any(upper[column].abs() > correlation\_threshold)]

    # Dataframe to hold correlated pairs

    record\_collinear = pd.DataFrame(columns = ['drop\_feature', 'corr\_feature', 'corr\_value'])

    # Iterate through the columns to drop

    for column in to\_drop:

        # Find the correlated features

        corr\_features = list(upper.index[upper[column].abs() > correlation\_threshold])

        # Find the correlated values

        corr\_values = list(upper[column][upper[column].abs() > correlation\_threshold])

        drop\_features = [column for \_ in range(len(corr\_features))]

        # Record the information (need a temp df for now)

        temp\_df = pd.DataFrame.from\_dict({'drop\_feature': drop\_features,

                                         'corr\_feature': corr\_features,

                                         'corr\_value': corr\_values})

        record\_collinear = pd.concat([record\_collinear, temp\_df], ignore\_index = True)

    return to\_drop

def Auto\_Feature\_Engineering(df):

    drop1 = Feature\_Importance\_IG(df)

    dfh1 = df.drop(columns = drop1)

    drop2 = Feature\_Redundancy\_Pearson(dfh1)

    dfh2 = dfh1.drop(columns = drop2)

    return dfh2

A screenshot of a computer

Description automatically generated

**Data Split & Balancing (After Feature Engineering)**

X = dfh2.drop(['Labelb'],axis=1)

y = dfh2['Labelb']

#X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y, train\_size = 0.8, test\_size = 0.2, shuffle=False,random\_state = 0)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y, train\_size = 0.8, test\_size = 0.2,random\_state = 0)

X\_train, y\_train = Auto\_Balancing(X\_train, y\_train)

**3. Automated Model Selection**

Select the best-performing model among five common machine learning models (Naive Bayes, KNN, random forest, LightGBM, and ANN/MLP) by evaluating their learning performance

**Method 1: Grid Search**

# Create a pipeline

pipe = Pipeline([('classifier', GaussianNB())])

# Create space of candidate learning algorithms and their hyperparameters

search\_space = [{'classifier': [GaussianNB()]},

                {'classifier': [KNeighborsClassifier()]},

                {'classifier': [RandomForestClassifier()]},

                {'classifier': [lgb.LGBMClassifier(verbose = -1)]},

                {'classifier': [KerasClassifier(build\_fn=ANN, verbose=0)]},

                 ]

clf = GridSearchCV(pipe, search\_space, cv=5, verbose=0)

clf.fit(X, y)

print("Best Model:"+ str(clf.best\_params\_))

print("Accuracy:"+ str(clf.best\_score\_))

Best Model:{'classifier': LGBMClassifier(verbose=-1)}

Accuracy:0.9843838600604344

clf.cv\_results\_

LightGBM model is the best performing machine learning model, and the best cross-validation accuracy is 98.438%

**Method 2: Bayesian Optimization with Tree Parzen Estimator (BO-TPE)**

from hyperopt import hp, fmin, tpe, STATUS\_OK, Trials

from sklearn.model\_selection import cross\_val\_score, StratifiedKFold

# Define the objective function

def objective(params):

    classifier\_type = params['type']

    del params['type']

    if classifier\_type == 'nb':

        clf = GaussianNB()

    elif classifier\_type == 'knn':

        clf = KNeighborsClassifier()

    elif classifier\_type == 'rf':

        clf = RandomForestClassifier()

    elif classifier\_type == 'lgb':

        clf = lgb.LGBMClassifier(verbose = -1)

    elif classifier\_type == 'ann':

        clf = KerasClassifier(build\_fn=ANN, verbose=0)

    else:

        return 0

    clf.fit(X\_train,y\_train)

    predictions = clf.predict(X\_test)

    score = accuracy\_score(y\_test,predictions)

    return {'loss':-score, 'status': STATUS\_OK }

# Define the hyperparameter configuration space

space = hp.choice('classifier\_type', [{'type': 'nb'},{'type': 'knn'},{'type': 'rf'},{'type': 'lgb'},{'type': 'ann'},])

# Detect the optimal hyperparameter values

best = fmin(fn=objective,

            space=space,

            algo=tpe.suggest,

            max\_evals=10)

print("Hyperopt estimated optimum {}".format(best))

Classifier type 3 is the LightGBM model, and the best hold-out accuracy is 99.806%

**4. Hyperparameter Optimization**

Optimize the best performing machine learning model (lightGBM) by tuning its hyperparameters

**Cross validation**

from hyperopt import hp, fmin, tpe, STATUS\_OK, Trials

from sklearn.model\_selection import cross\_val\_score, StratifiedKFold

# Define the objective function

def objective(params):

    params = {

        'n\_estimators': int(params['n\_estimators']),

        'max\_depth': int(params['max\_depth']),

        'learning\_rate': abs(float(params['learning\_rate'])),

        "num\_leaves": int(params['num\_leaves']),

        "min\_child\_samples": int(params['min\_child\_samples']),

    }

    clf = lgb.LGBMClassifier( \*\*params)

    score = cross\_val\_score(clf, X, y, scoring='accuracy', cv=StratifiedKFold(n\_splits=5)).mean()

    return {'loss':-score, 'status': STATUS\_OK }

# Define the hyperparameter configuration space

space = {

    'n\_estimators': hp.quniform('n\_estimators', 50, 500, 20),

    'max\_depth': hp.quniform('max\_depth', 5, 50, 1),

    "learning\_rate":hp.uniform('learning\_rate', 0, 1),

    "num\_leaves":hp.quniform('num\_leaves',100,2000,100),

    "min\_child\_samples":hp.quniform('min\_child\_samples',10,50,5),

}

# Detect the optimal hyperparameter values

best = fmin(fn=objective,

            space=space,

            algo=tpe.suggest,

            max\_evals=20)

print("LightGBM: Hyperopt estimated optimum {}".format(best))

LightGBM: Hyperopt estimated optimum {'learning\_rate': 0.28795394018630416, 'max\_depth': 25.0, 'min\_child\_samples': 30.0, 'n\_estimators': 400.0, 'num\_leaves': 1500.0}

%%time

clf = lgb.LGBMClassifier(max\_depth=14, learning\_rate=  0.4765834961973211, n\_estimators = 480,

                         num\_leaves = 600, min\_child\_samples = 25)

clf.fit(X,y)

scores = cross\_val\_score(clf, X, y, cv=5,scoring='accuracy')

print("Accuracy: "+ str(round(scores.mean(),5)\*100)+"%")

scores = cross\_val\_score(clf, X, y, cv=5,scoring='precision')

print("Precision: "+ str(round(scores.mean(),5)\*100)+"%")

scores = cross\_val\_score(clf, X, y, cv=5,scoring='recall')

print("Recall: "+ str(round(scores.mean(),5)\*100)+"%")

scores = cross\_val\_score(clf, X, y, cv=5,scoring='f1')

print("F1-score: "+ str(round(scores.mean(),5)\*100)+"%")

F1-score: 95.806%

CPU times: user 24.8 s, sys: 21 ms, total: 24.8 s

Wall time: 25.5 s

After hyperparameter optimization, the cross-validation accuracy has been improved from 98.438% to 98.477%

**Hold-out validation**

from hyperopt import hp, fmin, tpe, STATUS\_OK, Trials

from sklearn.model\_selection import cross\_val\_score, StratifiedKFold

# Define the objective function

def objective(params):

    params = {

        'n\_estimators': int(params['n\_estimators']),

        'max\_depth': int(params['max\_depth']),

        'learning\_rate': abs(float(params['learning\_rate'])),

        "num\_leaves": int(params['num\_leaves']),

        "min\_child\_samples": int(params['min\_child\_samples']),

    }

    clf = lgb.LGBMClassifier( \*\*params)

    clf.fit(X\_train,y\_train)

    predictions = clf.predict(X\_test)

    score = accuracy\_score(y\_test,predictions)

    return {'loss':-score, 'status': STATUS\_OK }

# Define the hyperparameter configuration space

space = {

    'n\_estimators': hp.quniform('n\_estimators', 50, 500, 20),

    'max\_depth': hp.quniform('max\_depth', 5, 50, 1),

    "learning\_rate":hp.uniform('learning\_rate', 0, 1),

    "num\_leaves":hp.quniform('num\_leaves',100,2000,100),

    "min\_child\_samples":hp.quniform('min\_child\_samples',10,50,5),

}

# Detect the optimal hyperparameter values

best = fmin(fn=objective,

            space=space,

            algo=tpe.suggest,

            max\_evals=50)

print("LightGBM: Hyperopt estimated optimum {}".format(best))

LightGBM: Hyperopt estimated optimum {'learning\_rate': 0.6662289706936085, 'max\_depth': 11.0, 'min\_child\_samples': 50.0, 'n\_estimators': 120.0, 'num\_leaves': 200.0}

%%time

clf = lgb.LGBMClassifier(max\_depth=35, learning\_rate= 0.7925617918030913, n\_estimators = 200,

                         num\_leaves = 200, min\_child\_samples = 25)

clf.fit(X\_train,y\_train)

predictions = clf.predict(X\_test)

print("Accuracy: "+str(round(accuracy\_score(y\_test,predictions),5)\*100)+"%")

print("Precision: "+str(round(precision\_score(y\_test,predictions),5)\*100)+"%")

print("Recall: "+str(round(recall\_score(y\_test,predictions),5)\*100)+"%")

print("F1-score: "+str(round(f1\_score(y\_test,predictions),5)\*100)+"%")

Accuracy: 99.753%

Precision: 99.29%

Recall: 99.467%

F1-score: 99.378%

CPU times: user 730 ms, sys: 7.03 ms, total: 737 ms

Wall time: 773 ms

**5. Combined Algorithm Selection and Hyperparameter tuning (CASH)**

CASH is the process of combining the two AutoML procedures: model selection and hyperparameter optimization.

**Method: Particle Swarm Optimization (PSO)**

import optunity

import optunity.metrics

search = {'algorithm': {'k-nn': {'n\_neighbors': [3, 10]},

                        'naive-bayes': None,

                        'random-forest': {

                                'n\_estimators': [50, 500],

                                'max\_features': [5, 12],

                                'max\_depth': [5,50],

                                "min\_samples\_split":[2,11],

                                "min\_samples\_leaf":[1,11]},

                        'lightgbm': {

                                'n\_estimators': [50, 500],

                                'max\_depth': [5, 50],

                                'learning\_rate': (0, 1),

                                "num\_leaves":[100, 2000],

                                "min\_child\_samples":[10, 50],

                                    },

                        'ann': {

                                'neurons': [10, 100],

                                'epochs': [20, 50],

                                'patience': [3, 20],

                                }

                        }

         }

def performance(

                algorithm, n\_neighbors=None,

    n\_estimators=None, max\_features=None,max\_depth=None,min\_samples\_split=None,min\_samples\_leaf=None,

    learning\_rate=None,num\_leaves=None,min\_child\_samples=None,

    neurons=None,epochs=None,patience=None

):

    # fit the model

    if algorithm == 'k-nn':

        model = KNeighborsClassifier(n\_neighbors=int(n\_neighbors))

    elif algorithm == 'naive-bayes':

        model = GaussianNB()

    elif algorithm == 'random-forest':

        model = RandomForestClassifier(n\_estimators=int(n\_estimators),

                                       max\_features=int(max\_features),

                                       max\_depth=int(max\_depth),

                                       min\_samples\_split=int(min\_samples\_split),

                                       min\_samples\_leaf=int(min\_samples\_leaf))

    elif algorithm == 'lightgbm':

        model = lgb.LGBMClassifier(n\_estimators=int(n\_estimators),

                                   max\_depth=int(max\_depth),

                                   learning\_rate=float(learning\_rate),

                                   num\_leaves=int(num\_leaves),

                                   min\_child\_samples=int(min\_child\_samples),

                                  )

    elif algorithm == 'ann':

        model = KerasClassifier(build\_fn=ANN, verbose=0,

                               neurons=int(neurons),

                                epochs=int(epochs),

                                patience=int(patience)

                               )

    else:

        raise ArgumentError('Unknown algorithm: %s' % algorithm)

# predict the test set

    model.fit(X\_train,y\_train)

    prediction = model.predict(X\_test)

    score = accuracy\_score(y\_test,prediction)

    return score

# Run the CASH process

optimal\_configuration, info, \_ = optunity.maximize\_structured(performance,

                                                              search\_space=search,

                                                              num\_evals=50)

print(optimal\_configuration)

print(info.optimum)

{'algorithm': 'lightgbm', 'epochs': None, 'neurons': None, 'patience': None, 'n\_neighbors': None, 'learning\_rate': 0.32638671874999997, 'max\_depth': 22.041113281250006, 'min\_child\_samples': 40.58548085623468, 'n\_estimators': 256.0068359375, 'num\_leaves': 1292.5494645058002, 'max\_features': None, 'min\_samples\_leaf': None, 'min\_samples\_split': None}

0.9978802331743508

%%time

clf = lgb.LGBMClassifier(max\_depth=24, learning\_rate= 0.25474609375, n\_estimators = 419,

                         num\_leaves = 1463, min\_child\_samples = 16)

clf.fit(X\_train,y\_train)

predictions = clf.predict(X\_test)

print("Accuracy: "+str(round(accuracy\_score(y\_test,predictions),5)\*100)+"%")

print("Precision: "+str(round(precision\_score(y\_test,predictions),5)\*100)+"%")

print("Recall: "+str(round(recall\_score(y\_test,predictions),5)\*100)+"%")

print("F1-score: "+str(round(f1\_score(y\_test,predictions),5)\*100)+"%")

Accuracy: 99.77000000000001%

Precision: 99.291%

Recall: 99.556%

F1-score: 99.423%

CPU times: user 2.23 s, sys: 0 ns, total: 2.23 s

Wall time: 2.29 s

LightGBM with the above hyperparameter values is identified as the optimal model

**Automated Machine Learning**

**Dataset 2: IoTID20**

A subset of the IoT network traffic data randomly sampled from the [IoTID20 dataset](https://sites.google.com/view/iot-network-intrusion-dataset/home).

IoTID20 dataset was created by using normal and attack virtual machines as network platforms, simulating IoT services with the node-red tool, and extracting features with the Information Security Center of Excellence (ISCX) flow meter program. A typical smart home environment was established for generating this dataset using five IoT devices or services: a smart fridge, a smart thermostat, motion-activated lights, a weather station, and a remotely-activated garage door. Thus, the traffic data samples of normal and abnormal IoT devices are collected in Pcap files.

**Import libraries**

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split,cross\_val\_score

import lightgbm as lgb

from sklearn.metrics import classification\_report,confusion\_matrix,accuracy\_score, precision\_score, recall\_score, f1\_score

from sklearn.preprocessing import LabelEncoder

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import classification\_report,confusion\_matrix,accuracy\_score

from sklearn.neighbors import KNeighborsClassifier

from sklearn.svm import SVC

from sklearn.naive\_bayes import GaussianNB

from sklearn.pipeline import Pipeline

from sklearn.model\_selection import GridSearchCV

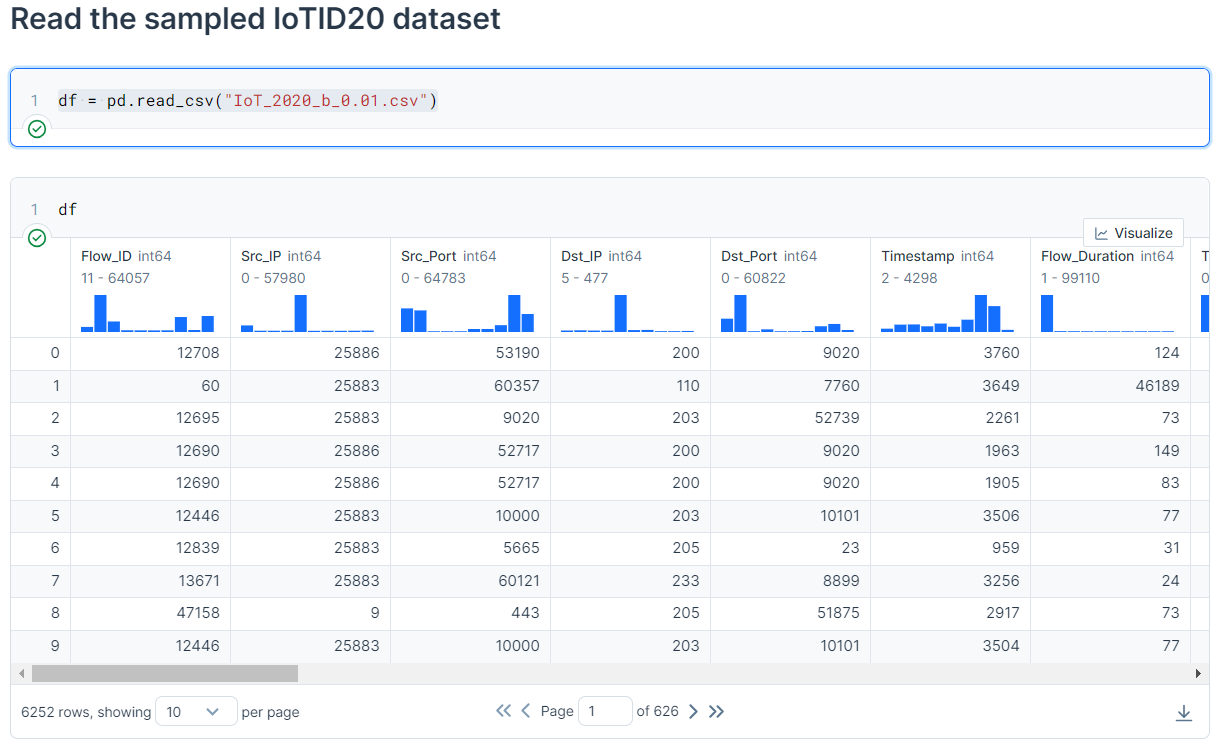
from scipy.stats import shapiro

from imblearn.over\_sampling import SMOTE

import time

import warnings

warnings.filterwarnings('ignore')



**1. Automated Data Pre-Processing**

**Automated Transformation/Encoding**

Automatically identify and transform string/text features into numerical features to make the data more readable by ML models

# Define the automated data encoding function

def Auto\_Encoding(df):

    cat\_features=[x for x in df.columns if df[x].dtype=="object"] ## Find string/text features

    le=LabelEncoder()

    for col in cat\_features:

        if col in df.columns:

            i = df.columns.get\_loc(col)

            # Transform to numerical features

            df.iloc[:,i] = df.apply(lambda i:le.fit\_transform(i.astype(str)), axis=0, result\_type='expand')

    return df

df=Auto\_Encoding(df)

**Automated Imputation**

Detect and impute missing values to improve data quality

# Define the automated data imputation function

def Auto\_Imputation(df):

    if df.isnull().values.any() or np.isinf(df).values.any(): # if there is any empty or infinite values

        df.replace([np.inf, -np.inf], np.nan, inplace=True)

        df.fillna(0, inplace = True)  # Replace empty values with zeros; there are other imputation methods discussed in the paper

    return df

df=Auto\_Imputation(df)

**Automated normalization**

Normalize the range of features to a similar scale to improve data quality

def Auto\_Normalization(df):

    stat, p = shapiro(df)

    print('Statistics=%.3f, p=%.3f' % (stat, p))

    # interpret

    alpha = 0.05

    numeric\_features = df.drop(['Label'],axis = 1).dtypes[df.dtypes != 'object'].index

    # The selection strategy is based on the following article:

    # https://medium.com/@kumarvaishnav17/standardization-vs-normalization-in-machine-learning-3e132a19c8bf

    # Check if the data distribution follows a Gaussian/normal distribution

    # If so, select the Z-score normalization method; otherwise, select the min-max normalization

    # Details are in the paper

    if p > alpha:

        print('Sample looks Gaussian (fail to reject H0)')

        df[numeric\_features] = df[numeric\_features].apply(

            lambda x: (x - x.mean()) / (x.std()))

        print('Z-score normalization is automatically chosen and used')

    else:

        print('Sample does not look Gaussian (reject H0)')

        df[numeric\_features] = df[numeric\_features].apply(

            lambda x: (x - x.min()) / (x.max()-x.min()))

        print('Min-max normalization is automatically chosen and used')

    return df

df=Auto\_Normalization(df)

Statistics=0.108, p=0.000

Sample does not look Gaussian (reject H0)

Min-max normalization is automatically chosen and used

**Train-test split**

Split the dataset into the training and the test set

X = df.drop(['Label'],axis=1)

y = df['Label']

# Here we used the 80%/20% split, it can be changed based on specific tasks

#X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y, train\_size = 0.8, test\_size = 0.2, shuffle=False,random\_state = 0)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y, train\_size = 0.8, test\_size = 0.2,random\_state = 0)

**Automated data balancing**

Generate minority class samples to solve class-imbalance and improve data quality.  
Synthetic Minority Over-sampling Technique (SMOTE) method is used.

pd.Series(y\_train).value\_counts()

Label

1 4717

0 284

Name: count, dtype: int64

# For binary data (can be modified for multi-class data with the same logic)

def Auto\_Balancing(X\_train, y\_train):

    number0 = pd.Series(y\_train).value\_counts().iloc[0]

    number1 = pd.Series(y\_train).value\_counts().iloc[1]

    if number0 > number1:

        nlarge = number0

    else:

        nlarge = number1

    # evaluate whether the incoming dataset is imbalanced (the abnormal/normal ratio is smaller than a threshold (e.g., 50%))

    if (number1/number0 > 1.5) or (number0/number1 > 1.5):

        smote=SMOTE(n\_jobs=-1,sampling\_strategy={0:nlarge, 1:nlarge})

        X\_train, y\_train = smote.fit\_resample(X\_train, y\_train)

    return X\_train, y\_train

X\_train, y\_train = Auto\_Balancing(X\_train, y\_train)

pd.Series(y\_train).value\_counts()

Label

1 4717

0 4717

Name: count, dtype: int64

**Model learning (for Comparison)**

%%time

lg = lgb.LGBMClassifier(verbose = -1)

lg.fit(X\_train,y\_train)

t1=time.time()

predictions = lg.predict(X\_test)

t2=time.time()

print("Accuracy: "+str(round(accuracy\_score(y\_test,predictions),5)\*100)+"%")

print("Precision: "+str(round(precision\_score(y\_test,predictions),5)\*100)+"%")

print("Recall: "+str(round(recall\_score(y\_test,predictions),5)\*100)+"%")

print("F1-score: "+str(round(f1\_score(y\_test,predictions),5)\*100)+"%")

print("Time: "+str(round((t2-t1)/len(y\_test)\*1000000,5)))

Accuracy: 99.92%

Precision: 99.91499999999999%

Recall: 100.0%

F1-score: 99.957%

Time: 5.09103

CPU times: user 314 ms, sys: 3.53 ms, total: 318 ms

Wall time: 325 ms

%%time

rf = RandomForestClassifier()

rf.fit(X\_train,y\_train)

t1=time.time()

predictions = rf.predict(X\_test)

t2=time.time()

print("Accuracy: "+str(round(accuracy\_score(y\_test,predictions),5)\*100)+"%")

print("Precision: "+str(round(precision\_score(y\_test,predictions),5)\*100)+"%")

print("Recall: "+str(round(recall\_score(y\_test,predictions),5)\*100)+"%")

print("F1-score: "+str(round(f1\_score(y\_test,predictions),5)\*100)+"%")

print("Time: "+str(round((t2-t1)/len(y\_test)\*1000000,5)))

Accuracy: 99.83999999999999%

Precision: 99.83%

Recall: 100.0%

F1-score: 99.91499999999999%

Time: 12.99924

CPU times: user 1.01 s, sys: 0 ns, total: 1.01 s

Wall time: 1.03 s

%%time

nb = GaussianNB()

nb.fit(X\_train,y\_train)

t1=time.time()

predictions = nb.predict(X\_test)

t2=time.time()

print("Accuracy: "+str(round(accuracy\_score(y\_test,predictions),5)\*100)+"%")

print("Precision: "+str(round(precision\_score(y\_test,predictions),5)\*100)+"%")

print("Recall: "+str(round(recall\_score(y\_test,predictions),5)\*100)+"%")

print("F1-score: "+str(round(f1\_score(y\_test,predictions),5)\*100)+"%")

print("Time: "+str(round((t2-t1)/len(y\_test)\*1000000,5)))

Accuracy: 69.624%

Precision: 99.874%

Recall: 67.717%

F1-score: 80.711%

Time: 2.00245

CPU times: user 14.5 ms, sys: 0 ns, total: 14.5 ms

Wall time: 19.5 ms

%%time

knn = KNeighborsClassifier()

knn.fit(X\_train,y\_train)

t1=time.time()

predictions = knn.predict(X\_test)

t2=time.time()

print("Accuracy: "+str(round(accuracy\_score(y\_test,predictions),5)\*100)+"%")

print("Precision: "+str(round(precision\_score(y\_test,predictions),5)\*100)+"%")

print("Recall: "+str(round(recall\_score(y\_test,predictions),5)\*100)+"%")

print("F1-score: "+str(round(f1\_score(y\_test,predictions),5)\*100)+"%")

print("Time: "+str(round((t2-t1)/len(y\_test)\*1000000,5)))

Accuracy: 98.881%

Precision: 99.82799999999999%

Recall: 98.978%

F1-score: 99.401%

Time: 64.61487

CPU times: user 87.2 ms, sys: 0 ns, total: 87.2 ms

Wall time: 90.2 ms

import tensorflow as tf

from keras.layers import Input,Dense,Dropout,BatchNormalization,Activation

from keras import Model

import keras.backend as K

import keras.callbacks as kcallbacks

from keras import optimizers

from keras.optimizers import Adam

from sklearn.model\_selection import GridSearchCV

from keras.wrappers.scikit\_learn import KerasClassifier

from keras.callbacks import EarlyStopping

def ANN(optimizer = 'sgd',neurons=16,batch\_size=1024,epochs=80,activation='relu',patience=8,loss='binary\_crossentropy'):

    K.clear\_session()

    inputs=Input(shape=(X.shape[1],))

    x=Dense(1000)(inputs)

    x=BatchNormalization()(x)

    x=Activation('relu')(x)

    x=Dropout(0.3)(x)

    x=Dense(256)(inputs)

    x=BatchNormalization()(x)

    x=Activation('relu')(x)

    x=Dropout(0.25)(x)

    x=Dense(2,activation='softmax')(x)

    model=Model(inputs=inputs,outputs=x,name='base\_nlp')

    model.compile(optimizer='adam',loss='categorical\_crossentropy')

#     model.compile(optimizer=Adam(lr = 0.01),loss='categorical\_crossentropy',metrics=['accuracy'])

    early\_stopping = EarlyStopping(monitor="loss", patience = patience)# early stop patience

    history = model.fit(X, pd.get\_dummies(y).values,

              batch\_size=batch\_size,

              epochs=epochs,

              callbacks = [early\_stopping],

              verbose=0) #verbose set to 1 will show the training process

    return model

%%time

ann = KerasClassifier(build\_fn=ANN, verbose=0)

ann.fit(X\_train,y\_train)

predictions = ann.predict(X\_test)

print("Accuracy: "+str(round(accuracy\_score(y\_test,predictions),5)\*100)+"%")

print("Precision: "+str(round(precision\_score(y\_test,predictions),5)\*100)+"%")

print("Recall: "+str(round(recall\_score(y\_test,predictions),5)\*100)+"%")

print("F1-score: "+str(round(f1\_score(y\_test,predictions),5)\*100)+"%")

print("Time: "+str(round((t2-t1)/len(y\_test)\*1000000,5)))

Accuracy: 97.682%

Precision: 99.739%

Recall: 97.785%

F1-score: 98.753%

Time: 64.61487

CPU times: user 7.9 s, sys: 1.25 s, total: 9.15 s

Wall time: 9.37 s

**2. Automated Feature Engineering**

Feature selection method 1: **Information Gain (IG)**, used to remove irrelevant features to improve model efficiency  
Feature selection method 2: **Pearson Correlation**, used to remove redundant features to improve model efficiency and accuracy

# Remove irrelevant features and select important features

def Feature\_Importance\_IG(data):

    features = data.drop(['Label'],axis=1).values  # "Label" should be changed to the target class variable name if different

    labels = data['Label'].values

    # Extract feature names

    feature\_names = list(data.drop(['Label'],axis=1).columns)

    # Empty array for feature importances

    feature\_importance\_values = np.zeros(len(feature\_names))

    model = lgb.LGBMRegressor(verbose = -1)

    model.fit(features, labels)

    feature\_importances = pd.DataFrame({'feature': feature\_names, 'importance': model.feature\_importances\_})

    # Sort features according to importance

    feature\_importances = feature\_importances.sort\_values('importance', ascending = False).reset\_index(drop = True)

    # Normalize the feature importances to add up to one

    feature\_importances['normalized\_importance'] = feature\_importances['importance'] / feature\_importances['importance'].sum()

    feature\_importances['cumulative\_importance'] = np.cumsum(feature\_importances['normalized\_importance'])

    cumulative\_importance=0.90 # Only keep the important features with cumulative importance scores>=90%. It can be changed.

    # Make sure most important features are on top

    feature\_importances = feature\_importances.sort\_values('cumulative\_importance')

    # Identify the features not needed to reach the cumulative\_importance

    record\_low\_importance = feature\_importances[feature\_importances['cumulative\_importance'] > cumulative\_importance]

    to\_drop = list(record\_low\_importance['feature'])

#     print(feature\_importances.drop(['importance'],axis=1))

    return to\_drop

# Remove redundant features

def Feature\_Redundancy\_Pearson(data):

    correlation\_threshold=0.90 # Only remove features with the redundancy>90%. It can be changed

    features = data.drop(['Label'],axis=1)

    corr\_matrix = features.corr()

    # Extract the upper triangle of the correlation matrix

    upper = corr\_matrix.where(np.triu(np.ones(corr\_matrix.shape), k = 1).astype(np.bool))

    # Select the features with correlations above the threshold

    # Need to use the absolute value

    to\_drop = [column for column in upper.columns if any(upper[column].abs() > correlation\_threshold)]

    # Dataframe to hold correlated pairs

    record\_collinear = pd.DataFrame(columns = ['drop\_feature', 'corr\_feature', 'corr\_value'])

    # Iterate through the columns to drop

    for column in to\_drop:

        # Find the correlated features

        corr\_features = list(upper.index[upper[column].abs() > correlation\_threshold])

        # Find the correlated values

        corr\_values = list(upper[column][upper[column].abs() > correlation\_threshold])

        drop\_features = [column for \_ in range(len(corr\_features))]

        # Record the information (need a temp df for now)

        temp\_df = pd.DataFrame.from\_dict({'drop\_feature': drop\_features,

                                         'corr\_feature': corr\_features,

                                         'corr\_value': corr\_values})

        record\_collinear = record\_collinear.append(temp\_df, ignore\_index = True)

#     print(record\_collinear)

    return to\_drop

def Auto\_Feature\_Engineering(df):

    drop1 = Feature\_Importance\_IG(df)

    dfh1 = df.drop(columns = drop1)

    drop2 = Feature\_Redundancy\_Pearson(dfh1)

    dfh2 = dfh1.drop(columns = drop2)

    return dfh2



**Data Split & Balancing (After Feature Engineering)**

X = dfh2.drop(['Label'],axis=1)

y = dfh2['Label']

#X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y, train\_size = 0.8, test\_size = 0.2, shuffle=False,random\_state = 0)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y, train\_size = 0.8, test\_size = 0.2,random\_state = 0)

X\_train, y\_train = Auto\_Balancing(X\_train, y\_train)

**3. Automated Model Selection**

Select the best-performing model among five common machine learning models (Naive Bayes, KNN, random forest, LightGBM, and ANN/MLP) by evaluating their learning performance

**Method 1: Grid Search**

# Create a pipeline

pipe = Pipeline([('classifier', GaussianNB())])

# Create space of candidate learning algorithms and their hyperparameters

search\_space = [{'classifier': [GaussianNB()]},

                {'classifier': [KNeighborsClassifier()]},

                {'classifier': [RandomForestClassifier()]},

                {'classifier': [lgb.LGBMClassifier(verbose = -1)]},

                {'classifier': [KerasClassifier(build\_fn=ANN, verbose=0)]},

                 ]

clf = GridSearchCV(pipe, search\_space, cv=5, verbose=0)

clf.fit(X, y)

A close-up of a computer code

Description automatically generated

print("Best Model:"+ str(clf.best\_params\_))

print("Accuracy:"+ str(clf.best\_score\_))

Best Model:{'classifier': LGBMClassifier(verbose=-1)}

Accuracy:0.9993601278976818

clf.cv\_results\_

LightGBM model is the best performing machine learning model, and the best cross-validation accuracy is 99.936%

**Method 2: Bayesian Optimization with Tree Parzen Estimator (BO-TPE)**

from hyperopt import hp, fmin, tpe, STATUS\_OK, Trials

from sklearn.model\_selection import cross\_val\_score, StratifiedKFold

# Define the objective function

def objective(params):

    classifier\_type = params['type']

    del params['type']

    if classifier\_type == 'nb':

        clf = GaussianNB()

    elif classifier\_type == 'knn':

        clf = KNeighborsClassifier()

    elif classifier\_type == 'rf':

        clf = RandomForestClassifier()

    elif classifier\_type == 'lgb':

        clf = lgb.LGBMClassifier(verbose = -1)

    elif classifier\_type == 'ann':

        clf = KerasClassifier(build\_fn=ANN, verbose=0)

    else:

        return 0

    clf.fit(X\_train,y\_train)

    predictions = clf.predict(X\_test)

    score = accuracy\_score(y\_test,predictions)

    return {'loss':-score, 'status': STATUS\_OK }

# Define the hyperparameter configuration space

space = hp.choice('classifier\_type', [{'type': 'nb'},{'type': 'knn'},{'type': 'rf'},{'type': 'lgb'},{'type': 'ann'},])

# Detect the optimal hyperparameter values

best = fmin(fn=objective,

            space=space,

            algo=tpe.suggest,

            max\_evals=10)

print("Hyperopt estimated optimum {}".format(best))

Hyperopt estimated optimum {'classifier\_type': 3}

Classifier type 3 is the LightGBM model, and the best hold-out accuracy is 100.0%

**4. Hyperparameter Optimization**

Optimize the best performing machine learning model (lightGBM) by tuning its hyperparameters

**Cross validation**

from hyperopt import hp, fmin, tpe, STATUS\_OK, Trials

from sklearn.model\_selection import cross\_val\_score, StratifiedKFold

# Define the objective function

def objective(params):

    params = {

        'n\_estimators': int(params['n\_estimators']),

        'max\_depth': int(params['max\_depth']),

        'learning\_rate': abs(float(params['learning\_rate'])),

        "num\_leaves": int(params['num\_leaves']),

        "min\_child\_samples": int(params['min\_child\_samples']),

    }

    clf = lgb.LGBMClassifier( \*\*params)

    score = cross\_val\_score(clf, X, y, scoring='accuracy', cv=StratifiedKFold(n\_splits=5)).mean()

    return {'loss':-score, 'status': STATUS\_OK }

# Define the hyperparameter configuration space

space = {

    'n\_estimators': hp.quniform('n\_estimators', 50, 500, 20),

    'max\_depth': hp.quniform('max\_depth', 5, 50, 1),

    "learning\_rate":hp.uniform('learning\_rate', 0, 1),

    "num\_leaves":hp.quniform('num\_leaves',100,2000,100),

    "min\_child\_samples":hp.quniform('min\_child\_samples',10,50,5),

}

# Detect the optimal hyperparameter values

best = fmin(fn=objective,

            space=space,

            algo=tpe.suggest,

            max\_evals=20)

print("LightGBM: Hyperopt estimated optimum {}".format(best))

LightGBM: Hyperopt estimated optimum {'learning\_rate': 0.5323259090349739, 'max\_depth': 38.0, 'min\_child\_samples': 20.0, 'n\_estimators': 180.0, 'num\_leaves': 400.0}

%%time

clf = lgb.LGBMClassifier(max\_depth=16, learning\_rate=  0.5636571315681871, n\_estimators = 180,

                         num\_leaves = 1800, min\_child\_samples = 50)

clf.fit(X,y)

scores = cross\_val\_score(clf, X, y, cv=5,scoring='accuracy')

print("Accuracy: "+ str(round(scores.mean(),5)\*100)+"%")

scores = cross\_val\_score(clf, X, y, cv=5,scoring='precision')

print("Precision: "+ str(round(scores.mean(),5)\*100)+"%")

scores = cross\_val\_score(clf, X, y, cv=5,scoring='recall')

print("Recall: "+ str(round(scores.mean(),5)\*100)+"%")

scores = cross\_val\_score(clf, X, y, cv=5,scoring='f1')

print("F1-score: "+ str(round(scores.mean(),5)\*100)+"%")

F1-score: 99.983%

CPU times: user 3.14 s, sys: 81.3 ms, total: 3.22 s

Wall time: 3.55 s

After hyperparameter optimization, the cross-validation accuracy has been improved from 99.936%% to 99.968%

**Hold-out validation**

from hyperopt import hp, fmin, tpe, STATUS\_OK, Trials

from sklearn.model\_selection import cross\_val\_score, StratifiedKFold

# Define the objective function

def objective(params):

    params = {

        'n\_estimators': int(params['n\_estimators']),

        'max\_depth': int(params['max\_depth']),

        'learning\_rate': abs(float(params['learning\_rate'])),

        "num\_leaves": int(params['num\_leaves']),

        "min\_child\_samples": int(params['min\_child\_samples']),

    }

    clf = lgb.LGBMClassifier( \*\*params)

    clf.fit(X\_train,y\_train)

    predictions = clf.predict(X\_test)

    score = accuracy\_score(y\_test,predictions)

    return {'loss':-score, 'status': STATUS\_OK }

# Define the hyperparameter configuration space

space = {

    'n\_estimators': hp.quniform('n\_estimators', 50, 500, 20),

    'max\_depth': hp.quniform('max\_depth', 5, 50, 1),

    "learning\_rate":hp.uniform('learning\_rate', 0, 1),

    "num\_leaves":hp.quniform('num\_leaves',100,2000,100),

    "min\_child\_samples":hp.quniform('min\_child\_samples',10,50,5),

}

# Detect the optimal hyperparameter values

best = fmin(fn=objective,

            space=space,

            algo=tpe.suggest,

            max\_evals=50)

print("LightGBM: Hyperopt estimated optimum {}".format(best))

LightGBM: Hyperopt estimated optimum {'learning\_rate': 0.8552679928489205, 'max\_depth': 37.0, 'min\_child\_samples': 15.0, 'n\_estimators': 380.0, 'num\_leaves': 1800.0}

%%time

clf = lgb.LGBMClassifier(max\_depth=45, learning\_rate= 0.17566405992887468, n\_estimators = 300,

                         num\_leaves = 400, min\_child\_samples = 45)

clf.fit(X\_train,y\_train)

predictions = clf.predict(X\_test)

print("Accuracy: "+str(round(accuracy\_score(y\_test,predictions),5)\*100)+"%")

print("Precision: "+str(round(precision\_score(y\_test,predictions),5)\*100)+"%")

print("Recall: "+str(round(recall\_score(y\_test,predictions),5)\*100)+"%")

print("F1-score: "+str(round(f1\_score(y\_test,predictions),5)\*100)+"%")

Accuracy: 100.0%

Precision: 100.0%

Recall: 100.0%

F1-score: 100.0%

CPU times: user 514 ms, sys: 16.6 ms, total: 530 ms

Wall time: 790 ms

After hyperparameter optimization, the hold-out accuracy has been improved from 100.0% to 100.0%

**4. Combined Algorithm Selection and Hyperparameter tuning (CASH)**

CASH is the process of combining the two AutoML procedures: model selection and hyperparameter optimization.

import optunity

import optunity.metrics

search = {'algorithm': {'k-nn': {'n\_neighbors': [3, 10]},

                        'naive-bayes': None,

                        'random-forest': {

                                'n\_estimators': [20, 100],

                                'max\_features': [5, 12],

                                'max\_depth': [5,50],

                                "min\_samples\_split":[2,11],

                                "min\_samples\_leaf":[1,11]},

                        'lightgbm': {

                                'n\_estimators': [20, 100],

                                'max\_depth': [5, 50],

                                'learning\_rate': (0, 1),

                                "num\_leaves":[100, 2000],

                                "min\_child\_samples":[10, 50],

                                    },

                        'ann': {

                                'neurons': [10, 100],

                                'epochs': [20, 50],

                                'patience': [3, 20],

                                }

                        }

         }

def performance(

                algorithm, n\_neighbors=None,

    n\_estimators=None, max\_features=None,max\_depth=None,min\_samples\_split=None,min\_samples\_leaf=None,

    learning\_rate=None,num\_leaves=None,min\_child\_samples=None,

    neurons=None,epochs=None,patience=None

):

    # fit the model

    if algorithm == 'k-nn':

        model = KNeighborsClassifier(n\_neighbors=int(n\_neighbors))

    elif algorithm == 'naive-bayes':

        model = GaussianNB()

    elif algorithm == 'random-forest':

        model = RandomForestClassifier(n\_estimators=int(n\_estimators),

                                       max\_features=int(max\_features),

                                       max\_depth=int(max\_depth),

                                       min\_samples\_split=int(min\_samples\_split),

                                       min\_samples\_leaf=int(min\_samples\_leaf))

    elif algorithm == 'lightgbm':

        model = lgb.LGBMClassifier(n\_estimators=int(n\_estimators),

                                   max\_depth=int(max\_depth),

                                   learning\_rate=float(learning\_rate),

                                   num\_leaves=int(num\_leaves),

                                   min\_child\_samples=int(min\_child\_samples),

                                  )

    elif algorithm == 'ann':

        model = KerasClassifier(build\_fn=ANN, verbose=0,

                               neurons=int(neurons),

                                epochs=int(epochs),

                                patience=int(patience)

                               )

    else:

        raise ArgumentError('Unknown algorithm: %s' % algorithm)

# predict the test set

    model.fit(X\_train,y\_train)

    prediction = model.predict(X\_test)

    score = accuracy\_score(y\_test,prediction)

    return score

# Run the CASH process

optimal\_configuration, info, \_ = optunity.maximize\_structured(performance,

                                                              search\_space=search,

                                                              num\_evals=50)

print(optimal\_configuration)

print(info.optimum)

{'algorithm': 'random-forest', 'epochs': None, 'neurons': None, 'patience': None, 'n\_neighbors': None, 'learning\_rate': None, 'max\_depth': 22.05078125, 'min\_child\_samples': None, 'n\_estimators': 75.9375, 'num\_leaves': None, 'max\_features': 6.28515625, 'min\_samples\_leaf': 7.7578125, 'min\_samples\_split': 9.41796875}

1.0

%%time

clf = lgb.LGBMClassifier(max\_depth=28, learning\_rate= 0.88427734375, n\_estimators = 78,

                         num\_leaves = 251, min\_child\_samples = 40)

clf.fit(X\_train,y\_train)

predictions = clf.predict(X\_test)

print("Accuracy: "+str(round(accuracy\_score(y\_test,predictions),5)\*100)+"%")

print("Precision: "+str(round(precision\_score(y\_test,predictions),5)\*100)+"%")

print("Recall: "+str(round(recall\_score(y\_test,predictions),5)\*100)+"%")

print("F1-score: "+str(round(f1\_score(y\_test,predictions),5)\*100)+"%")

Accuracy: 100.0%

Precision: 100.0%

Recall: 100.0%

F1-score: 100.0%

CPU times: user 122 ms, sys: 0 ns, total: 122 ms

Wall time: 128 ms

LightGBM with the above hyperparameter values is identified as the optimal model

