Automated Machine Learning

Dataset 1: CICIDS2017

A subset of the network traffic data randomly sampled from the CICIDS2017 dataset.

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.

Read the sampled CICIDS2017 dataset

```
df = pd.read_csv("cic_0.01km.csv")
```

```
df
```

Visualize

	Flow Duration int64 -4 - 119999943	Total Length of F... 0 - 719406	Fwd Packet Lengt... 0 - 24820	Fwd Packet Lengt...	Bwd Packet Lengt...	Bwd Packet Lengt...	Flow IAT Mean floa...	F
0	50833	0	0	0	0	0	50833	
1	49	0	0	0	0	0	49	
2	306	6	6	6	6	6	306	
3	63041	65	65	65	124	124	63041	
4	47682	43	43	43	59	59	47682	
5	23896	88	44	44	100	100	7965.333333	
6	343	62	31	31	159	159	114.3333333	
7	25271	164	41	41	57	57	3610.142857	
8	1060	78	39	39	104	104	353.3333333	
9	224	82	41	41	169	169	74.66666667	

28303 rows, showing 10 per page Page 1 of 2831

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,los
s='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
```

```python
# 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
```

```python
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
```

```python
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
```

```
dfh2 = Auto_Feature_Engineering(df)
dfh2
```

Visualize

	Flow Duration float... 0.0 - 1.0	Total Length of F... 0.0 - 1.0	Fwd Packet Lengt... 0.0 - 1.0	Fwd Packet Lengt...	Bwd Packet Lengt...	Flow IAT Mean floa...	Flow IAT Min float64	F
0	0.0004236418538	0	0	0	0	0.0004707129455	0.0004707129455	
1	4.416668617e-7	0	0	0	0	4.907407226e-7	4.907407226e-7	
2	0.000002583334..	0.000008340214..	0.0002417405318	0.001555942266	0.0005158184319	0.000002870370..	0.000002870370..	
3	0.000525375232	0.000090352318..	0.002618855761	0.01685604122	0.01066024759	0.0005837499784	0.0005837499784	
4	0.0003973835088	0.00005977153374	0.001732473811	0.01115091957	0.00507221458	0.0004415370207	0.0004415370207	
5	0.0001991667546	0.0001223231388	0.0017727639	0.01141024329	0.008596973865	0.00007379012072	6.481481241e-8	
6	0.000002891667..	0.00008618221144	0.001248992748	0.008039035042	0.01366918845	0.000001095678...	7.407407133e-8	
7	0.000210625093	0.0002279658496	0.001651893634	0.01063227215	0.004900275103	0.000033464284..	4.629629458e-8	
8	0.000008866670..	0.0001084227821	0.001571313457	0.01011362473	0.00894085282	0.000003308641..	4.814814636e-7	
9	0.000001900000..	0.0001139829248	0.001651893634	0.01063227215	0.01452888583	7.283950348e-7	6.481481241e-8	

28303 rows, showing 10 per page — Page 1 of 2831

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=No
ne,
    learning_rate=None,num_leaves=None,min_child_samples=None,
    neurons=None,epochs=None,patience=None
):
    # fit the model
    if algorithm == 'k-nn':
```

```python
        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_sp
lit),
                                       min_samples_leaf=int(min_samples_lea
f))
    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.

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')
```

Read the sampled IoTID20 dataset

```
df = pd.read_csv("IoT_2020_b_0.01.csv")
```

```
df
```

Visualize

	Flow_ID int64 11 - 64057	Src_IP int64 0 - 57980	Src_Port int64 0 - 64783	Dst_IP int64 5 - 477	Dst_Port int64 0 - 60822	Timestamp int64 2 - 4298	Flow_Duration int64 1 - 99110	T 0
0	12708	25886	53190	200	9020	3760	124	
1	60	25883	60357	110	7760	3649	46189	
2	12695	25883	9020	203	52739	2261	73	
3	12690	25886	52717	200	9020	1963	149	
4	12690	25886	52717	200	9020	1905	83	
5	12446	25883	10000	203	10101	3506	77	
6	12839	25883	5665	205	23	959	31	
7	13671	25883	60121	233	8899	3256	24	
8	47158	9	443	205	51875	2917	73	
9	12446	25883	10000	203	10101	3504	77	

6252 rows, showing 10 per page   Page 1 of 626

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,los
s='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
```

```python
# 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
```

```python
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
```

```python
dfh2 = Auto_Feature_Engineering(df)
dfh2
```

Visualize

	Flow_ID float64 0.0 - 1.0	Src_IP float64 0.0 - 1.0	Src_Port float64 0.0 - 1.0	Dst_IP float64 0.0 - 1.0	Dst_Port float64 0.0 - 1.0	Timestamp float64 0.0 - 1.0	Flow_Duration floa... 0.0 - 1.0	T 0
0	0.1982481342	0.446464298	0.8210487319	0.4131355932	0.1483016014	0.8747672253	0.001241057825	
1	0.0007650751023	0.4464125561	0.9316796073	0.2224576271	0.1275854132	0.8489292365	0.4660323482	
2	0.198045155	0.4464125561	0.1392340583	0.4194915254	0.8671040084	0.5258379888	0.0007264728733	
3	0.1979670862	0.446464298	0.8137474337	0.4131355932	0.1483016014	0.4564711359	0.001493305351	
4	0.1979670862	0.446464298	0.8137474337	0.4131355932	0.1483016014	0.4429702048	0.0008273718835	
5	0.1941573244	0.4464125561	0.1543614837	0.4194915254	0.1660747756	0.8156424581	0.0007668324774	
6	0.200293539	0.4464125561	0.08744578053	0.4237288136	0.0003781526421	0.2227653631	0.0003026970305	
7	0.213284202	0.4464125561	0.9280366763	0.4830508475	0.1463121897	0.7574487896	0.0002320677234	
8	0.7361427724	0.00015522594	0.006838213729	0.4237288136	0.8528986222	0.678538175	0.0007264728733	
9	0.1941573244	0.4464125561	0.1543614837	0.4194915254	0.1660747756	0.8151769088	0.0007668324774	

6252 rows, showing 10 per page — Page 1 of 626

Data Split & Balancing (After Feature Engineering)

```python
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)
```

```python
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)
```

```
▾ LGBMClassifier
LGBMClassifier(learning_rate=0.88427734375, max_depth=28, min_child_samples=40,
               n_estimators=78, num_leaves=251)
```

```
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%

5. 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_sp
lit),
                                       min_samples_leaf=int(min_samples_lea
f))
    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