



VIT[®]
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF COMPUTER SCIENCE ENGINEERING AND INFORMATION SYSTEMS

Winter Semester – 2023-24

M.Tech (SE)

SWE1904 - Capstone Project

2nd Review

Register Number	19MIS0184
Student Name	VIVEK R
Project Code (Course Code)	SWE1904
Project Domain (Capstone Project)	DATA SCIENCE
Project Title (Capstone Project)	Comprehensive approach of Static and Dynamic Data Analytics using AutoML
Guide Name	Dr. CHADRASEGAR T

Comprehensive approach of Static and Dynamic Data Analytics using AutoML

PROPOSED METHODOLOGY:

The proposed methodology involves a comprehensive approach to leveraging Automated Machine Learning (AutoML) for both static intrusion detection and dynamic Internet of Things (IoT) data analytics tasks. It begins with a thorough exploratory data analysis and preprocessing phase, addressing issues such as class imbalance, redundant records, and missing values in the CICIDS2017 intrusion detection dataset. Automated feature engineering and selection techniques will be employed to extract relevant features from the raw data, tailored for the respective tasks. Subsequently, AutoML frameworks will be utilized to automate the process of selecting appropriate machine learning algorithms and optimizing their hyperparameters, with a focus on maximizing performance metrics like accuracy, precision, recall, and F1-score.

To handle the dynamic nature of IoT data streams, techniques for detecting concept drift will be implemented, enabling the monitoring of performance metrics, distribution statistics, or leveraging dedicated drift detection algorithms. Automated model updating procedures will be developed to retrain or adapt the models when concept drift is detected, ensuring their continued accuracy and relevance over time.

The performance of the developed AutoML models will be rigorously evaluated using appropriate evaluation metrics and protocols. A comparative analysis will be conducted, contrasting the AutoML models' performance with traditional machine learning approaches that require manual intervention for model development. Finally, the feasibility of deploying the AutoML models in real-world scenarios will be investigated, considering factors like computational resources, scalability, and integration with existing systems. Recommendations and guidelines for effectively utilizing AutoML techniques in intrusion detection systems and IoT data analytics applications will be provided.

TECHNOLOGY USED:

Programming Language

Python:

Python a high-level, interpreted programming language known for its simplicity and readability serves as the primary programming language for developing the project. Its versatility and extensive ecosystem of libraries make it well-suited for tasks ranging from data preprocessing to model deployment.



Libraries

Data Analysis and Machine Learning Libraries:

1. **pandas:** For data manipulation & analysis, used working with structured (tabular) data.
2. **numpy:** Core package for scientific computing, handling multi-dimensional arrays and mathematical functions.
3. **scikit-learn:** Simple & efficient tools for machine learning tasks like classification, regression, clustering, and preprocessing.
4. **lightgbm:** A gradient boosting framework that uses tree-based learning algorithms and provides high-performance, distributed, and efficient implementations
5. **imblearn:** A library for imbalanced datasets, providing methods to over-sample or under-sample datasets for classification tasks.
6. **scipy:** A library for scientific and technical computing in Python, providing many user-friendly and efficient numerical routines, such as routines for numerical integration, interpolation, optimization, linear algebra, and statistics.

Deep Learning Libraries:

1. **tensorflow:** An open-source library for numerical computation and large-scale machine learning, developed by Google Brain.
2. **keras:** A high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano.

Hyperparameter Optimization Libraries:

1. **hyperopt:** Serial and parallel optimization of black-box functions, featuring global and Bayesian optimization algorithms..
2. **optunity:** Optimizes user-defined Python functions, providing solvers including particle swarm optimization (PSO) and other metaheuristics

warnings: A built-in Python module for handling warning messages in Python.

Cloud-based Data Science Notebook

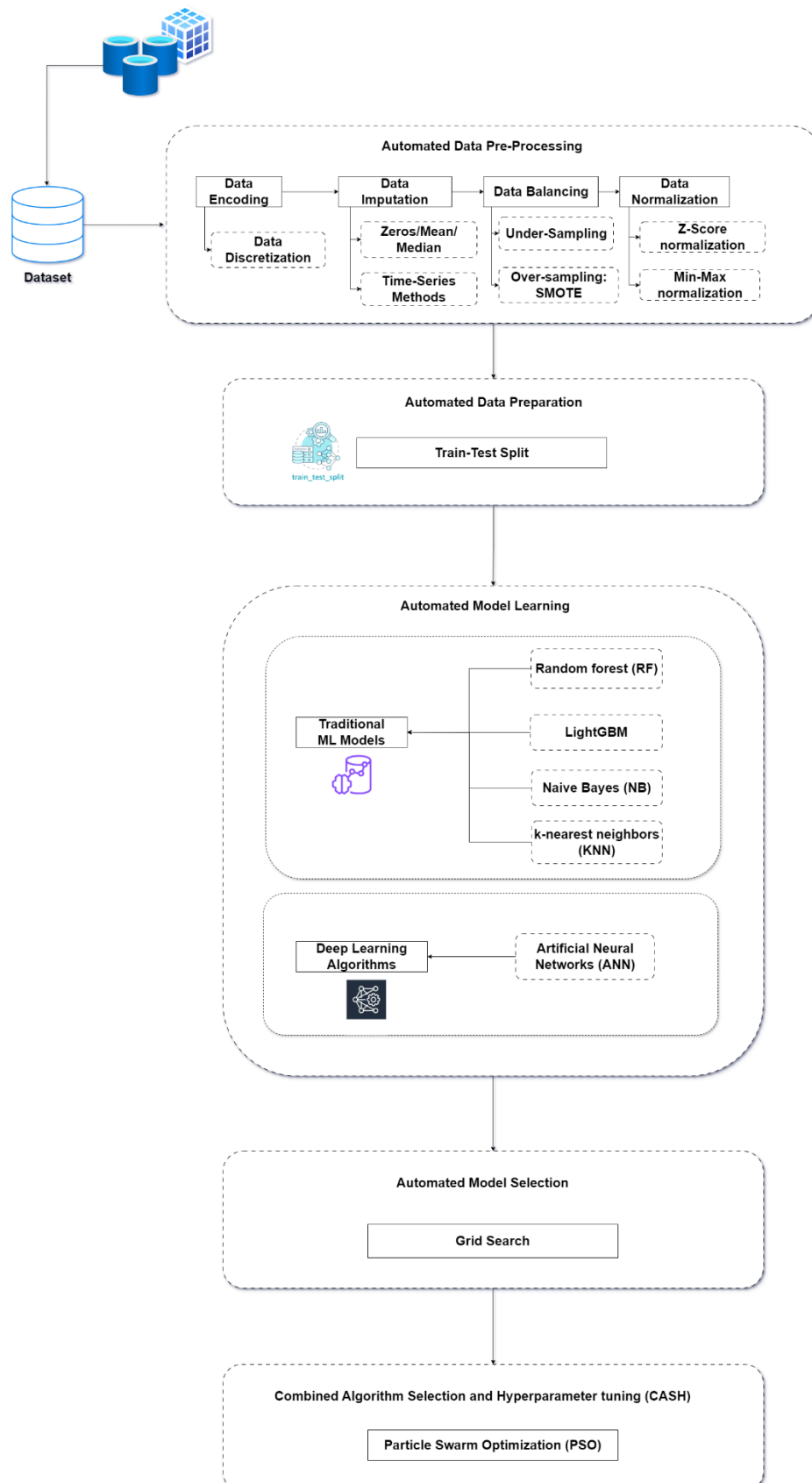
Deeptime:

A cloud-based data science notebook environment that allows you to write and execute code, visualize data, and collaborate with others. It was likely used as the development and execution environment for this project.



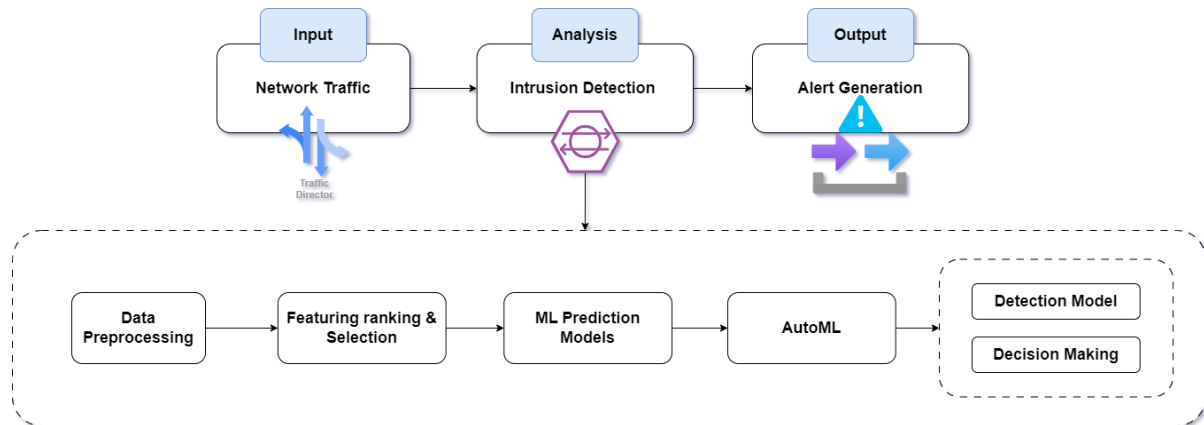
ARCHITECTURE:

Architecture of Static and Dynamic Datasets using AutoML

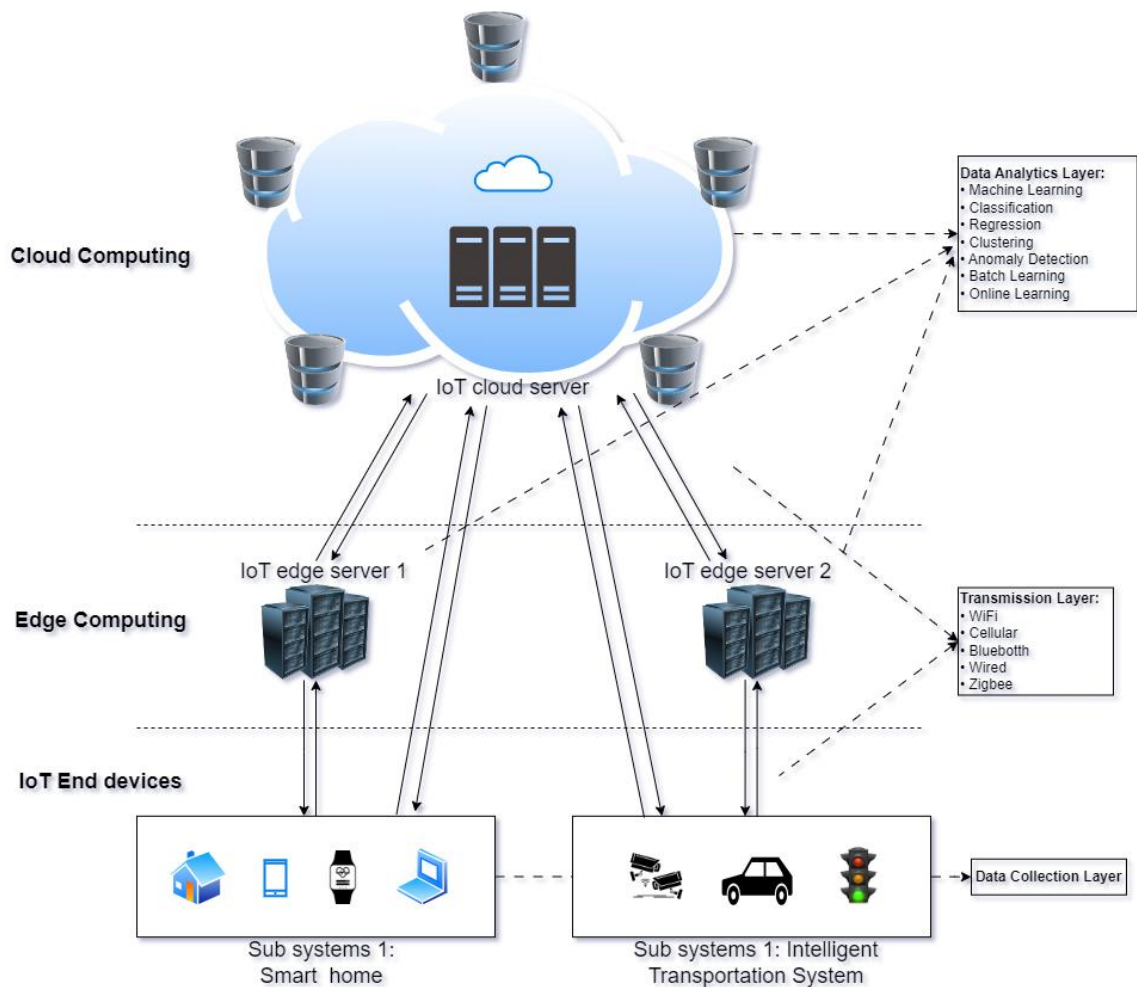


COMPLETE DESIGN:

Static dataset – Intrusion Detection Evaluation Dataset (CICIDS2017)



Dynamic dataset – IoT Data Analytics in Dynamic Environments



PROJECT MODULE DESCRIPTION:

1. Automated Data Preprocessing Module

This module handles the initial data preparation tasks in an automated manner. It consists of the following sub-modules:

A. Automated Transformation/Encoding

- This sub-module automatically identifies and transforms string/text features into numerical features, making the data more readable for machine learning models. Functionality: It uses the LabelEncoder from scikit-learn to encode categorical features.

B. Automated Imputation

- This sub-module detects and imputes missing values in the dataset to improve data quality. Functionality: It replaces infinite values with NaN and then fills NaN values with zeros. However, it can be modified to use other imputation techniques.

C. Automated Normalization

- This sub-module normalizes the range of features to a similar scale, based on the data distribution. Functionality: It uses the Shapiro-Wilk test to determine if the data follows a Gaussian distribution. If so, it applies Z-score normalization; otherwise, it applies min-max normalization.

D. Train-Test Split

- This sub-module splits the dataset into training and test sets. Functionality: It uses scikit-learn's train_test_split function to create an 80/20 train-test split by default, but this can be modified.

E. Automated Data Balancing

- This sub-module generates minority class samples to address class imbalance and improve data quality. Functionality: It uses the Synthetic Minority Over-sampling Technique (SMOTE) from the imbalanced-learn library to oversample the minority class.

2. Automated Model Learning Module

This module performs automated training and evaluation of several machine learning models for comparison purposes.

A. Model Training

- Trains the following machine learning models on the preprocessed training data: Naive Bayes, K-Nearest Neighbors, Random Forest, LightGBM, and Artificial Neural Network (ANN). Functionality: It uses the respective classes from scikit-learn, LightGBM, and Keras libraries to instantiate and fit the models on the training data.

i. Naive Bayes

Description: Naive Bayes is a probabilistic classifier based on applying Bayes' theorem with the assumption of independence between features.

Implementation: The GaussianNB class from scikit-learn is used

ii. K-Nearest Neighbors (KNN)

Description: KNN is a non-parametric algorithm that classifies a data point based on the majority class among its k nearest neighbors in the feature space.

Implementation: The KNeighborsClassifier class from scikit-learn is used to instantiate the KNN classifier.

iii. Random Forest

Description: Random Forest is an ensemble learning method that constructs multiple decision trees and combines their predictions by averaging or majority voting.

Implementation: The RandomForestClassifier class from scikit-learn is used to instantiate the Random Forest classifier.

iv. LightGBM

Description: LightGBM (Light Gradient Boosting Machine) is a gradient boosting framework that uses tree-based learning algorithms to build efficient and high-performance models.

Implementation: The LGBMClassifier class from the LightGBM library is used to instantiate the LightGBM classifier.

v. Artificial Neural Network (ANN)

Description: An Artificial Neural Network (ANN) or Multi-Layer Perceptron (MLP) is a type of feedforward neural network that consists of an input layer, one or more hidden layers, and an output layer, inspired by the biological neural networks in the human brain.

Implementation: A custom function ANN is defined, which creates a sequential model using the Keras library. The model architecture consists of an input layer, two dense hidden layers with ReLU activation and dropout, and an output layer with softmax activation for binary classification.

Training: The KerasClassifier from scikit-learn is used to wrap the custom ANN function. The fit method is called on the KerasClassifier instance, passing the training data (X_train and y_train) to train the ANN model using backpropagation and the specified hyperparameters (e.g., batch size, epochs, early stopping patience).

B. Model Evaluation

- Evaluates the trained models on the test data and reports various performance metrics. Functionality: It uses the trained models to make predictions on the test data and calculates the following metrics using scikit-learn's functions:
 - Accuracy
 - Precision
 - Recall
 - F1-score
 - Time (time taken to make predictions on the test set)

3. Automated Model Selection Module

This module selects the best-performing machine learning model among five common models: Naive Bayes, K-Nearest Neighbors, Random Forest, LightGBM, and Artificial Neural Network (ANN).

Grid Search

- Description: This method performs an exhaustive search over the specified hyperparameter values for each model.
- Functionality: It uses scikit-learn's GridSearchCV to evaluate the performance of each model using 5-fold cross-validation.

4. Combined Algorithm Selection and Hyperparameter Tuning (CASH) Module

This module combines the processes of model selection and hyperparameter optimization into a single step.

Particle Swarm Optimization (PSO)

- Description: This method uses Particle Swarm Optimization (PSO) to simultaneously select the best machine learning model and tune its hyperparameters.
- Functionality: It defines a performance function that trains and evaluates each model with different hyperparameter values on the hold-out test set. The optunity library is used to perform the CASH process using PSO.

IMPLEMENTATION (70%):

AutoML – 1:Static Dataset - CICIDS2017

1. Data Pre-Processing

i. 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)
```

ii. 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)
```

iii. Normalization

```
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)

```

iv. 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)

```

v. 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
```

2. Model learning

LGBM Classifier Algorithm

```
%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)*100000,5)))
```

```
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
```

Random Forest Algorithm

```
%%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
```

Naive Bayes Algorithm

```
%%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
```

K-Nearest Neighbor (KNN) Algorithm

```
%%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)*100000,5)))
```

```
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
```

KerasClassifier Algorithm

```
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

```

3. 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: 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%

4. 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

AutoML – 2:Dynamic Dataset - IoTID20

1. Data Pre-Processing

i. 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)
```

ii. 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)
```

iii. 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
```

iv. 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)
```

v. 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
```

2. Model learning

LGBM Classifier Algorithm

```
%%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
```

Random Forest Algorithm

```
%%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
```

Naive Bayes Algorithm

```
%%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
```

K-Nearest Neighbor (KNN) Algorithm

```
%%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
```

KerasClassifier Algorithm

```
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
```

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: 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%

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_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=
                                                                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)+"%")

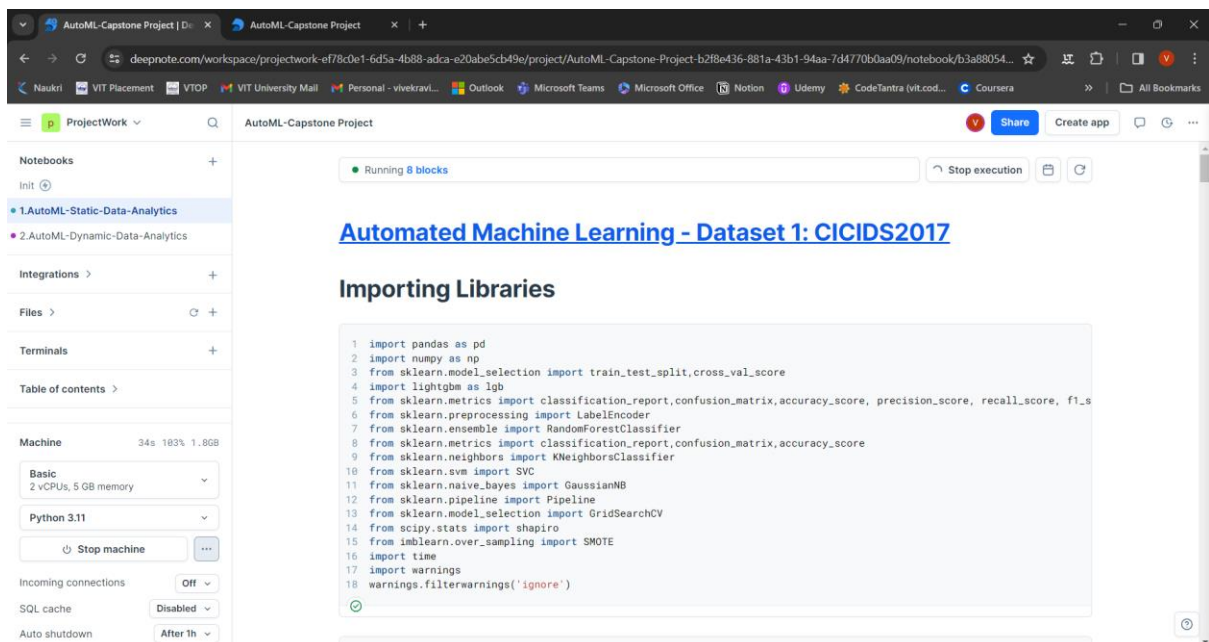
```

```
print("Recall: "+str(round(recall_score(y_test,predictions),5)*100)+"%")
print("F1-score: "+str(round(f1_score(y_test,predictions),5)*100)+"%")
```

Accuracy: 99.92%
Precision: 100.0%
Recall: 99.91499999999999%
F1-score: 99.957%
CPU times: user 152 ms, sys: 3.32 ms, total: 156 ms
Wall time: 163 ms

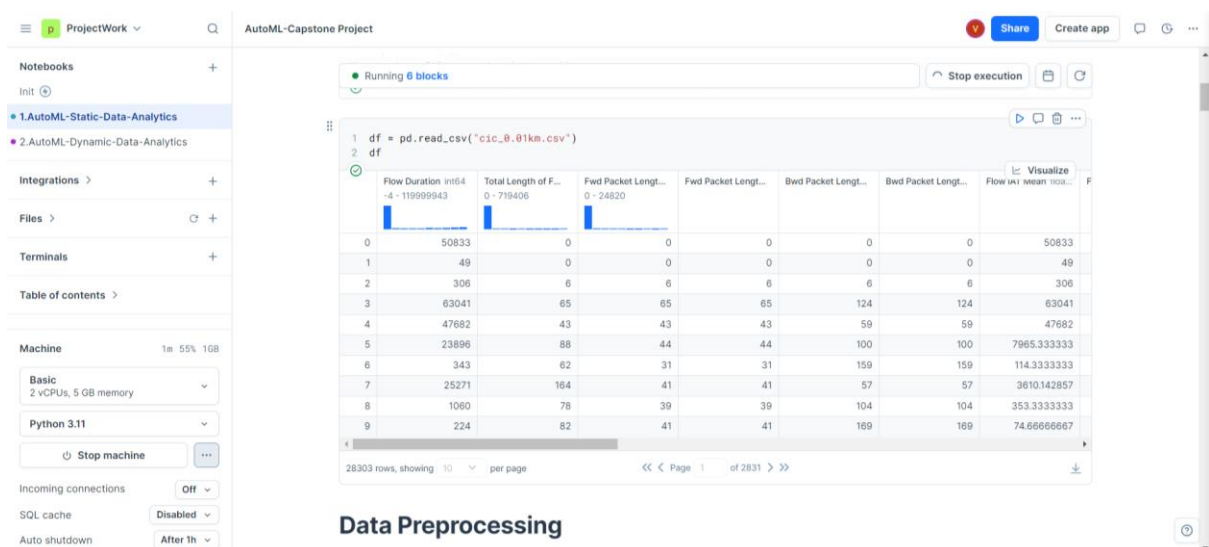
LightGBM with the above hyperparameter values is identified as the optimal model

SCREENSHOTS



The screenshot shows the AutoML-Capstone Project interface. The notebook is titled "Automated Machine Learning - Dataset 1: CICIDS2017" and is in the "Importing Libraries" section. The code block shows the following imports:

```
1 import pandas as pd
2 import numpy as np
3 from sklearn.model_selection import train_test_split, cross_val_score
4 import lightgbm as lgb
5 from sklearn.metrics import classification_report, confusion_matrix, accuracy_score, precision_score, recall_score, f1_score
6 from sklearn.preprocessing import LabelEncoder
7 from sklearn.ensemble import RandomForestClassifier
8 from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
9 from sklearn.neighbors import KNeighborsClassifier
10 from sklearn.svm import SVC
11 from sklearn.naive_bayes import GaussianNB
12 from sklearn.pipeline import Pipeline
13 from sklearn.model_selection import GridSearchCV
14 from scipy.stats import shapiro
15 from imblearn.over_sampling import SMOTE
16 import time
17 import warnings
18 warnings.filterwarnings('ignore')
```



The screenshot shows the AutoML-Capstone Project interface. The notebook is titled "Data Preprocessing" and displays a table of data. The table has the following columns: Flow Duration, Total Length of Flow, Fwd Packet Length, Fwd Packet Length, Bwd Packet Length, Bwd Packet Length, and Flow Latency. The table shows 28303 rows of data.

	Flow Duration	Total Length of Flow	Fwd Packet Length	Fwd Packet Length	Bwd Packet Length	Bwd Packet Length	Flow Latency
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	114.333333
7	25271	164	41	41	57	57	3610.142857
8	1060	78	39	39	104	104	353.333333
9	224	82	41	41	169	169	74.66666667

PERFORMANCE METRICS

Accuracy

Accuracy is the most basic metric, defined as the proportion of correctly categorized test instances to the total number of test instances. It is applicable to the majority of classification problems but is less useful when dealing with imbalanced datasets. Accuracy can be calculated by using True Positives (TPs), True Negatives (TNs), False Positives (FPs), and False Negatives (FNs):

$$\text{Acc} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

Static Dataset

Model Precision	Percentage
LGBM Classifier	99.753 %
Random Forest	75.729 %
Naive Bayes	98.728 %
k-nearest neighbors (KNN)	92.475 %
KerasClassifier Model	99.753 %

Dynamic Dataset

Model Precision	Percentage
LGBM Classifier	99.92%
Random Forest	99.839
Naive Bayes	70.184%
k-nearest neighbors (KNN)	99.280
KerasClassifier Model	92.475 %

Precision

Precision is the metric used to quantify the correctness of classification. Precision indicates the ratio of correct positive classifications to expected positive classifications. The larger the proportion, the more accurate the model, indicating that it is more capable of correctly identifying the positive class.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Static Dataset

Model Precision	Percentage
LGBM Classifier	99.378 %
Random Forest	99.554 %
Naive Bayes	44.891 %
k-nearest neighbors (KNN)	95.584 %
KerasClassifier Model	73.378 %

Dynamic Dataset

Model Precision	Percentage
LGBM Classifier	99.914
Random Forest	99.83%
Naive Bayes	99.875%
k-nearest neighbors (KNN)	99.744%
KerasClassifier Model	92.475 %

Recall

Recall is a measure of the percentage of accurately recognized positive instances to the total number of positive instances.

$$\text{Recall} = \text{TP} / \text{TP} + \text{FN}$$

Static Dataset

Model Precision	Percentage
LGBM Classifier	99.788 %
Random Forest	99.753 %
Naive Bayes	97.244 %
k-nearest neighbors (KNN)	98.133 %
KerasClassifier Model	97.511 %

Dynamic Dataset

Model Precision	Percentage
LGBM Classifier	100.0%
Random Forest	100.0%
Naive Bayes	68.313%
k-nearest neighbors (KNN)	99.489%
KerasClassifier Model	92.475 %

F1 score

The F1 score is calculated as the harmonic mean of the Recall and Precision scores, therefore balancing their respective strengths.

$$\text{F1} = 2 \times \text{TP} / 2 \times \text{TP} + \text{FP} + \text{FN}$$

Static Dataset

Model Precision	Percentage
LGBM Classifier	99.788 %
Random Forest	99.753 %
Naive Bayes	61.426 %
k-nearest neighbors (KNN)	96.842%
KerasClassifier Model	83.740%

Dynamic Dataset

Model Precision	Percentage
LGBM Classifier	99.957%
Random Forest	99.914
Naive Bayes	81.133%
k-nearest neighbors (KNN)	99.616%
KerasClassifier Model	92.475 %

REFERENCES:

- [1] Yang, L., & Shami, A. (2022). IoT data analytics in dynamic environments: From an automated machine learning perspective. *Engineering Applications of Artificial Intelligence*, 116, 105366.
- [2] Singh, A., Amutha, J., Nagar, J., Sharma, S., & Lee, C. C. (2022). AutoML-ID: Automated machine learning model for intrusion detection using wireless sensor network. *Scientific Reports*, 12(1), 9074.
- [3] Lindstedt, H. (2022). Methods for network intrusion detection : Evaluating rule-based methods and machine learning models on the CIC-IDS2017 dataset (Dissertation). Retrieved from <https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-479347>
- [4] Garouani, M., Ahmad, A., Bouneffa, M., & Hamlich, M. (2022). AMLBID: an auto-explained automated machine learning tool for big industrial data. *SoftwareX*, 17, 100919.
- [5] He, Y., Lin, J., Liu, Z., Wang, H., Li, L. J., & Han, S. (2018). Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European conference on computer vision (ECCV)* (pp. 784-800).
- [6] He, X., Zhao, K., & Chu, X. (2021). AutoML: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212, 106622.
- [7] Lee, J., Ahn, S., Kim, H., & Lee, J. R. (2022). Dynamic Hyperparameter Allocation under Time Constraints for Automated Machine Learning. *Intelligent Automation & Soft Computing*, 31(1).
- [8] Wever, M., Tornede, A., Mohr, F., & Hüllermeier, E. (2021). AutoML for multi-label classification: Overview and empirical evaluation. *IEEE transactions on pattern analysis and machine intelligence*, 43(9), 3037-3054.
- [9] Celik, B., Singh, P., & Vanschoren, J. (2023). Online automl: An adaptive automl framework for online learning. *Machine Learning*, 112(6), 1897-1921.
- [10] Zhang, S., Gong, C., Wu, L., Liu, X., & Zhou, M. (2023). AutoML-GPT: Automatic Machine Learning with GPT. *arXiv preprint arXiv:2305.02499*.