

**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. **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.

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.

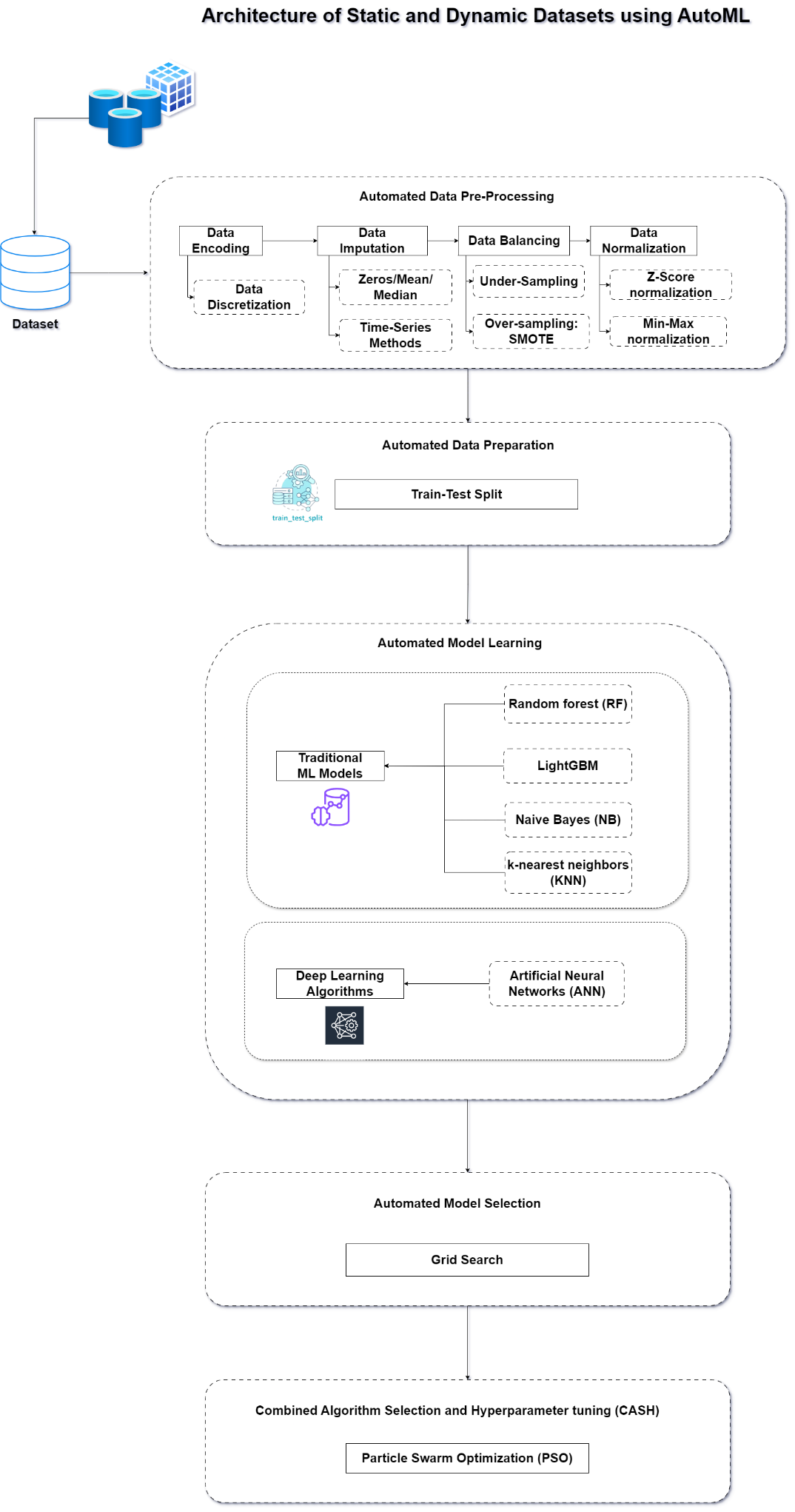
**Cloud-based Data Science Notebook**

A logo with blue and black text

Description automatically generatedDeepnote:

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:**



**COMPLETE DESIGN:**

Static dataset – Intrusion Detection Evaluation Dataset (CICIDS2017)

A diagram of a company

Description automatically generated with medium confidence

Dynamic dataset – IoT Data Analytics in Dynamic Environments

A diagram of a cloud computing system

Description automatically generated

**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 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.
  1. 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

* 1. 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.

* 1. 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.

* 1. 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.

* 1. 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

    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)\*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

**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)\*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

**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%

**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)

A close-up of a computer code

Description automatically generated

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

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

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

Accuracy:0.9993601278976818

clf.cv\_results\_

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

**SCREENSHOTS**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

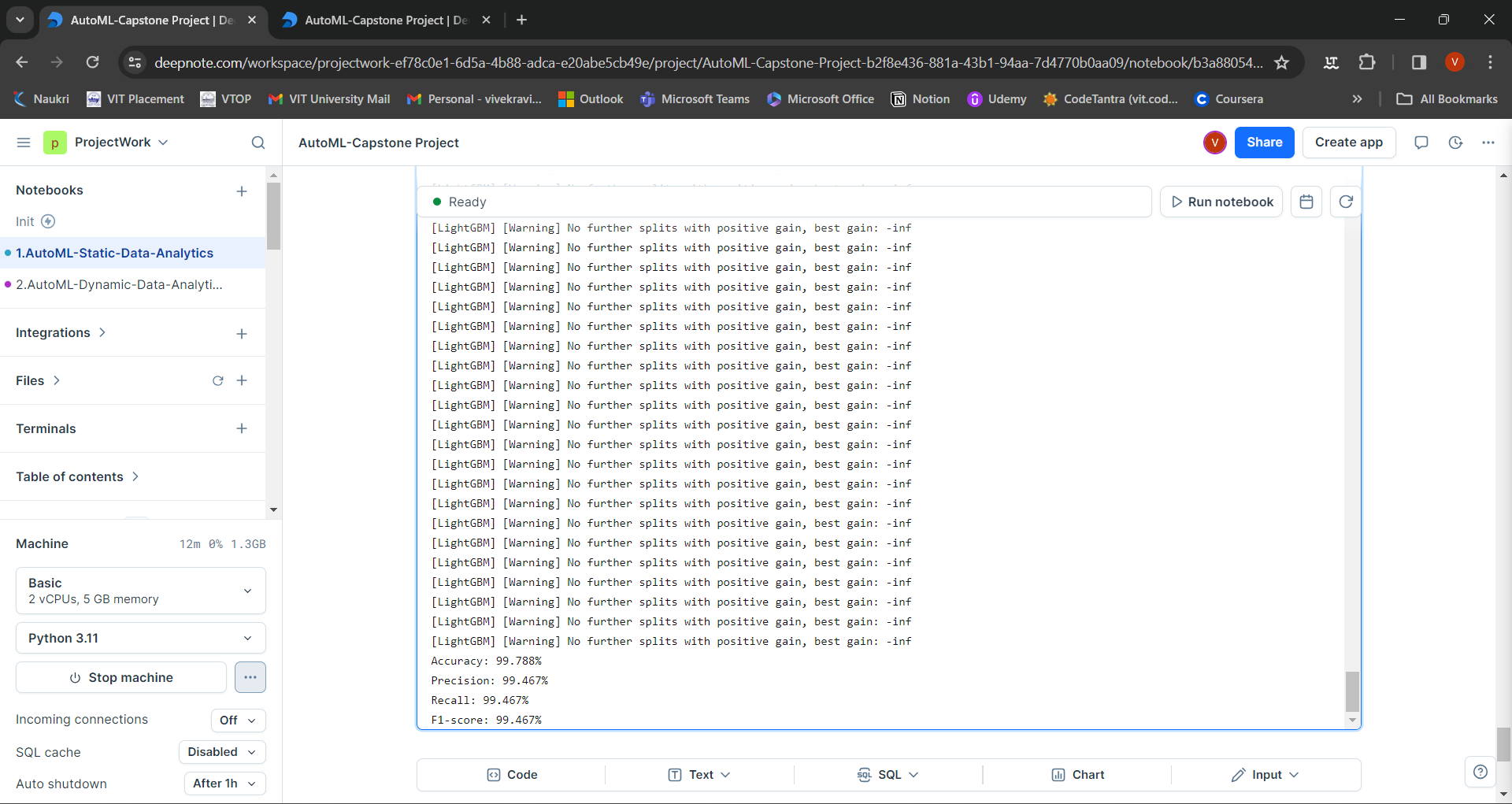
Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

****

**PERFORMANCE METRICS**

**Accuracy**

Accuracy is the most basic metric, de ned 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):

**Acc = TP +TN / TP +TN + FP + FN**

**Static Dataset**

|  |  |
| --- | --- |
| Model Accuracy | 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 Accuracy | 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.

**Precision = TP / TP + FP**

|  |  |
| --- | --- |
| 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 % |

**Static Dataset**

**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.

**Recall = TP / TP + FN**

**Static Dataset**

|  |  |
| --- | --- |
| Model Recall | 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 Recall | 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.

**F1 = 2 x TP / 2 x TP + FP + FN**

**Static Dataset**

|  |  |
| --- | --- |
| Model F1 score | 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 F1 score | 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.