UNIVERSITI TEKNIKAL MALAYSIA MELAKA	**NEURAL NETWORK**	**Mini Project by group (30%)**
BITI 3133	**SEMESTER 2**	**SESI 2022 / 2023**

Predicting Automotive Vehicles Engine Health

NAME	MATRIC NO
KEE CHIN YEW	B032110144
MUHAMMAD FAKHRUL HAZWAN	B032110357
SAIFUL ADLI	B032110275
NABIL SYAZANI BIN MOHD NASIR	B032110353

CONTENT

1. INTRODUCTION

Automotive engines are one of the important parts in vehicles. The engine is like the brain of vehicles, supplying necessary power to make vehicles function. Hence, the periodic maintenance of the engine is needed to help vehicles to function with the best performance. Automotive vehicles engine health allows us to determine the condition of the engine for deciding whether the engine needs maintenance or not.

Automotive vehicles engines have many features and details need to take care to ensure there is no anything that would cause permanent damage to the engine. Most of the automotive vehicles' engines are internal combustion engines. This engine uses a technique that ignition and combustion of the fuel occurs inside the engine. After that, the energy will be converted from the combustion to work. The engine consists of a fixed cylinder and a moving piston which will be pushed by combustion gases to rotate the crankshaft. Eventually, this motion will drive the vehicle’s wheels through the engine.

Based on the design and the technique of automotive vehicles engines, we cannot directly decide the condition of the engine. Some of the engines may be working well currently but the engines could have some serious risks that are hard to discover. Inaccurate estimation of engine condition may cause accidents in future. However, there are some features that allow us to predict the condition of the engine. The work done by the engine, coolant in the engine and lubricant oil of the engine can be the important variables for determining the condition of the engine because these features show the performance of the engine.

The main contributions of this research work includes:

- We mainly addressed the main issue which is the prediction of engine condition.
- We developed an Artificial Neural Network (ANN) method to classify the condition of the engine in order to predict the requirement of maintenance of the engine.
- We used Manual Grid Search to find out the best parameter for ANN model to obtain the most accurate prediction.

2. SAMPLE DATA

The data is obtained from Kaggle which is named Automotive Vehicles Engine Health Dataset. The dataset consists of 19535 instances and 7 attributes. The attributes are Engine rpm, Lub oil pressure, Fuel pressure, Coolant pressure, lub oil temp, Coolant temp and Engine Condition. The target attribute is Engine Condition. For better data processing in our predicting model, we modified the attributes' name as the name listed below.

Data dictionary:

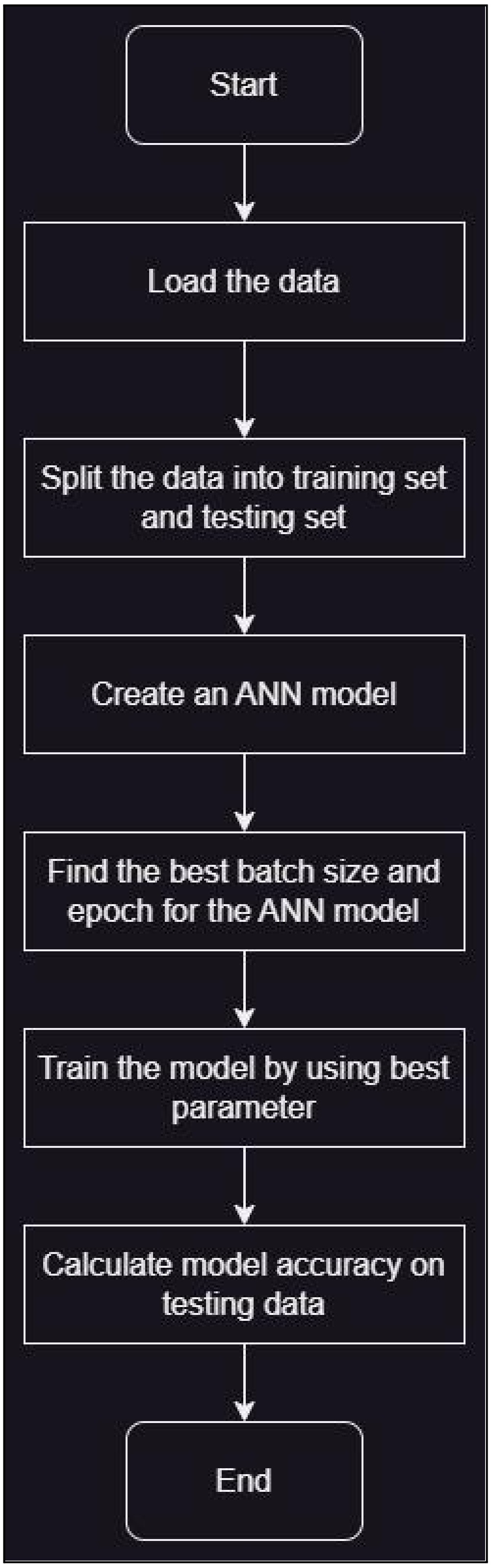
Attribute Name	Attribute Name Changed	Description
Engine rpm	Engine_rpm	This is the number of revolutions per minute of wheels done by the work of the engine. The unit is rpm.
Lub oil pressure	Lub_oil_pressure	This is the pressure of the lubricant oil in the engine to optimise the function of moving parts in the engine. The oil pressure ensures the oil reaches everywhere needed in the engine. The unit is PSI
Fuel pressure	Fuel_pressure	This is the fuel pressure in the engine. High fuel pressure will pass too much fuel into the engine and cause damage. The unit is PSI.
Coolant pressure	Coolant_pressure	This is the coolant pressure in the engine. The pressure helps to prevent coolant in the engine from boiling off. The unit is PSI.
lub oil temp	lub_oil_temp	This is the temperature of the lubricant oil in the engine. The temperature affects the viscosity of the lubricant oil. The unit is degree.
Coolant temp	Coolant_temp	This is the temperature of the coolant. High temperature means the coolant probably boiled

		off and the engine overheated. The unit is degree.
Engine Condition	Engine_Condition	This is the attribute to store the condition of the engine. The value is 1 which indicates acceptable and 0 which indicates need for maintenance or repair.

Sample of 10 data:

	Engine_rpm	Lub_oil_pressure	Fuel_pressure	Coolant_pressure	lub_oil_temp	Coolant_temp	Engine_Condition
0	700	2.493592	11.790927	3.178981	84.144163	81.632187	1
1	876	2.941606	16.193866	2.464504	77.640934	82.445724	0
2	520	2.961746	6.553147	1.064347	77.752266	79.645777	1
3	473	3.707835	19.510172	3.727455	74.129907	71.774629	1
4	619	5.672919	15.738871	2.052251	78.396989	87.000225	0
5	1221	3.989226	6.679231	2.214250	76.401152	75.669818	0
6	716	3.568896	5.312266	2.461067	83.646589	79.792411	1
7	729	3.845166	10.191126	2.362998	77.921202	71.671761	1
8	845	4.877239	3.638269	3.525604	76.301626	70.496024	0
9	824	3.741228	7.626214	1.301032	77.066520	85.143297	0

3. FLOWCHART


Start
Load the data
Split the data into training set and testing set
Create an ANN model
Find the best batch size and epoch for the ANN model
Train the model by using best parameter
Calculate model accuracy on testing data
End

4. LEARNING PROCESS

1. Load the data into workspace

   The dataset is imported into workspace for training model. The dataset used is Automotive Vehicles Engine Health Dataset and the dataset file type is .csv.

2. Split the data into training set and testing set

   The target variable and predictors are defined. Target variable is Engine Condition and Predictors are Engine rpm, Lub oil pressure, Fuel pressure, Coolant pressure, lub oil temp and Coolant temp. The predictors are standardised for the prediction. The dataset is splitted into a training set and testing set with a ratio 8:2.

3. Create an ANN model

   Define the parameter of the model:

   - For the input layer, the input dimension is 6 because there are 6 features in the data, the number of nodes in the layer is 7 which is equal to the number of features and an additional 1 node for bias. The activation function used is the Relu function. This function will change all negative values to 0 and remain the same values as the output if the value is positive value.
   - For the output layer, there is only one node in the layer since the target variable has only one. The activation function used is the Sigmoid function. This function transforms the values to the value between 0 and 1.
   - For the hidden layer, the layer has only one layer with 6 nodes in the layer. The number of modes is decided by using the mean of the nodes in the input and output layers. The activation function used in the hidden layer also is the Relu function.

   Create a classifier with the defined parameter. Compile the classifier with these parameters, optimizer = 'adam', loss = 'binary_crossentropy' and metrics = ['accuracy']. After that, fit the neural network on the training data with batch size = 10 and epochs = 10.

4. Find the best batch size and epoch for the ANN model

   For obtaining the best accuracy, use Manual Grid Search to find out the best parameter for the model. The batch size and epoch are defined which are [5, 10, 15, 20] and [5, 10, 50, 100] respectively to find out which combination provides the highest accuracy.

5. Train the model by using best parameter

   Use the parameter obtained in the previous step to train the model with the training set.

6. Calculate the model accuracy on test data

   Make predictions on the testing data with the model. Define the probability threshold value which is 0.5 to classify the engine whether it needs to have maintenance or not. After that, generate new predictions by applying the probability threshold value. At last, calculate the precision, recall, f1-score and accuracy from the result.

5. STEP BY STEP LEARNING

Step 1: Import the required libraries

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense
from sklearn import metrics
```

Step 2: Initialise the variables and import the dataset

```
bbatch = 0
bepoch = 0
baccuracy = 0
minaccuracy = 1

# To remove the scientific notation from numpy arrays
np.set_printoptions(suppress=True)

df = pd.read_csv('C:/Users/HP/Downloads/engine_data.csv')
```

Step 3: Separate the target variable and predictor variables and define it as x and y

```
# Separate Target Variable and Predictor Variables
TargetVariable = ['Engine_Condition']
Predictors = ['Engine_rpm', 'Lub_oil_pressure', 'Fuel_pressure',
              'Coolant_pressure', 'lub_oil_temp', 'Coolant_temp']

X = df[Predictors].values
y = df[TargetVariable].values
```

Step 4: Standardise the data

```
# Standardization of data
PredictorScaler = StandardScaler()

# Storing the fit object for later reference
PredictorScalerFit = PredictorScaler.fit(X)

# Generating the standardized values of X and y
X = PredictorScalerFit.transform(X)
```

Step 5: Split the dataset into a training and testing set

```
# Split the data into training and testing set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step 6: Display the shape of the training and testing set

```
# Quick sanity check with the shapes of Training and Testing datasets
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

Step 7: Create an ANN model with the parameters

```
classifier = Sequential()
# Defining the Input layer
classifier.add(Dense(units=7, input_dim=6, kernel_initializer='uniform', activation='relu'))
# Defining the Hidden layer
classifier.add(Dense(units=4, kernel_initializer='uniform', activation='relu'))
# Defining the Output layer
classifier.add(Dense(units=1, kernel_initializer='uniform', activation='sigmoid'))
classifier.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# fitting the Neural Network on the training data
survivalANN_Model = classifier.fit(X_train, y_train, batch_size=10, epochs=10, verbose=1)
```

Step 8: Create a function for finding best parameters by using Manual Grid Search

```
# Defining a function for finding best hyperparameters
def FunctionFindBestParams(X_train, y_train):
```

Step 9: define the parameters to test in the function

```
    # Defining the list of hyper parameters to try
    TrialNumber = 0
    batch_size_list = [5, 10, 15, 20]
    epoch_list = [5, 10, 50, 100]

    SearchResultsData = pd.DataFrame(columns=['TrialNumber', 'Parameters',
'Accuracy'])
```

Step 10: Make a loop for making different parameter combinations and find the model accuracy with the parameters

```
    for batch_size_trial in batch_size_list:
        for epochs_trial in epoch_list:
            TrialNumber += 1

            # Creating the classifier ANN model
            classifier = Sequential()
                                        classifier.add(Dense(units=7,   input_dim=6,
kernel_initializer='uniform', activation='relu'))
                 classifier.add(Dense(units=4, kernel_initializer='uniform',
activation='relu'))
                 classifier.add(Dense(units=1, kernel_initializer='uniform',
activation='sigmoid'))
            classifier.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

                      survivalANN_Model  =  classifier.fit(X_train,  y_train,
batch_size=batch_size_trial, epochs=epochs_trial,
                                                   verbose=0)
            # Fetching the accuracy of the training
            Accuracy = survivalANN_Model.history['accuracy'][-1]

            # printing the results of the current iteration
                              print(TrialNumber,  'Parameters:',  'batch_size:',
batch_size_trial, '-', 'epochs:', epochs_trial,
                 'Accuracy:', Accuracy)
```

Step 11: Store the parameters and accuracy into variables

```
            global bbatch
            global bepoch
            global baccuracy
            global minaccuracy

            if float(Accuracy) < float(minaccuracy):
                minaccuracy = Accuracy

            if float(Accuracy) > float(baccuracy):
                bbatch = batch_size_trial
                bepoch = epochs_trial
                baccuracy = Accuracy

                        SearchResultsData  =  pd.concat([SearchResultsData,
pd.DataFrame(data=[[TrialNumber, 'batch_size' +

str(batch_size_trial) + '-' + 'epoch' +

str(epochs_trial), Accuracy]],

columns=['TrialNumber', 'Parameters',

'Accuracy'])])
   return (SearchResultsData)
```

Step 12: Call the function to find the best parameters

```
# Calling the function
ResultsData = FunctionFindBestParams(X_train, y_train)
```

Step 13: Display the parameters with its accuracy and visualise it

```
# Printing the best parameter
print(ResultsData.sort_values(by='Accuracy', ascending=False).head(1))
# Visualizing the results
plt.figure(figsize=(12, 5))
plt.plot(ResultsData.Parameters, ResultsData.Accuracy, label='Accuracy')
plt.xticks(rotation=20)
plt.ylim([round(minaccuracy*0.8, 2), round(baccuracy*1.2, 2)])
plt.show()
```

Step 14: Train the model with the best parameters

```
# Training the model with best hyperparameters
classifier.fit(X_train,    y_train,    batch_size=bbatch,    epochs=bepoch,
verbose=1)
```

Step 15: Make prediction on the testing set by using the model

```
# Predictions on testing data
Predictions = classifier.predict(X_test)
# Scaling the test data back to original scale
Test_Data = PredictorScalerFit.inverse_transform(X_test)
# Generating a data frame for analyzing the test data
TestingData = pd.DataFrame(data=Test_Data, columns=Predictors)
TestingData['Condition'] = y_test
TestingData['PredictedProb'] = Predictions
```

Step 16: Define the probability threshold value

```
# Defining the probability threshold
def probThreshold(inpProb):
    if inpProb > 0.5:
        return (1)
    else:
        return (0)
```

Step 17: Reassign the value into output after applying the probability threshold

```
#  Generating  predictions  on  the  testing  data  by  applying  probability
threshold
TestingData['Predicted']                                                    =
TestingData['PredictedProb'].apply(probThreshold)
```

Step 18: Display the results and the confusion matrix

```
print(TestingData.head())
print('\nTesting Accuracy Results: ')
print(metrics.classification_report(TestingData['Condition'],
TestingData['Predicted']))
print(metrics.confusion_matrix(TestingData['Condition'],
TestingData['Predicted']))
```

6. ANALYSIS

In this project, we used python to develop the predicting model to determine the engine condition.Below is the shape for the training set of x, training set of y, testing set of x and testing set of y.

```
(15628, 6)
(15628, 1)
(3907, 6)
(3907, 1)
```

Before we predict the engine condition, we need to find out the best parameters for our predicting model. Hence, the parameters are found by using Manual Grid Search.

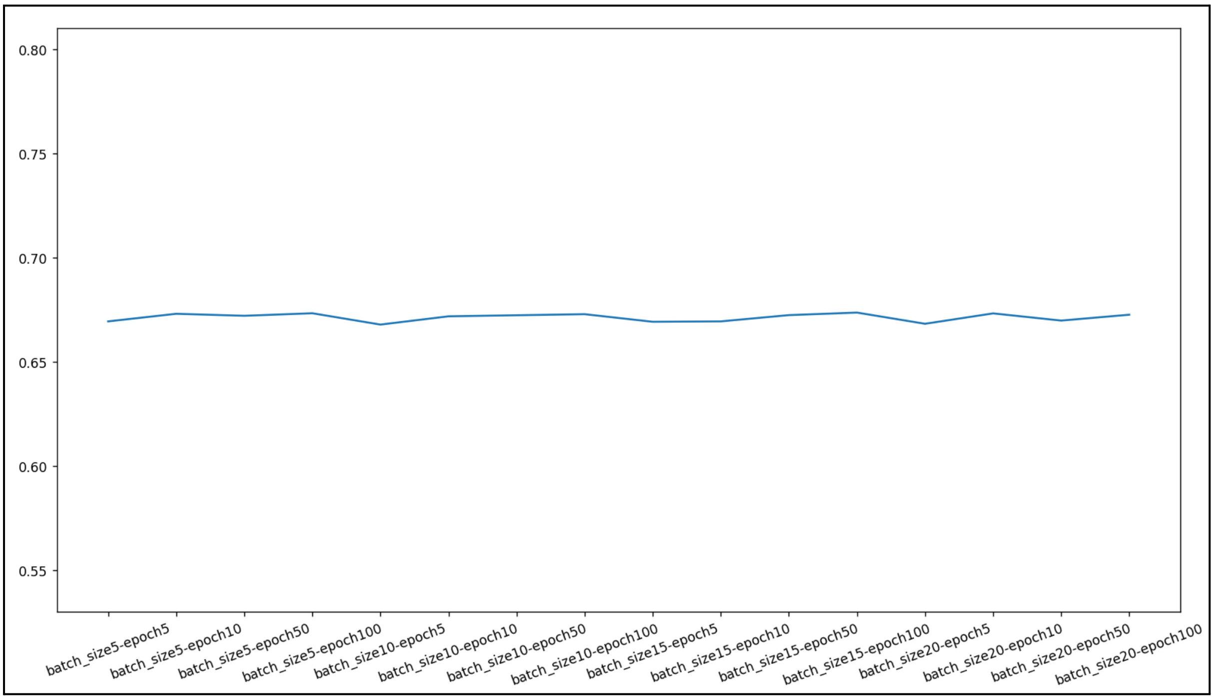
```
1 Parameters: batch_size: 5 - epochs: 5 Accuracy: 0.6693114638328552
2 Parameters: batch_size: 5 - epochs: 10 Accuracy: 0.6729587912559509
3 Parameters: batch_size: 5 - epochs: 50 Accuracy: 0.6719989776611328
4 Parameters: batch_size: 5 - epochs: 100 Accuracy: 0.6732147336006165
5 Parameters: batch_size: 10 - epochs: 5 Accuracy: 0.6677758097648621
6 Parameters: batch_size: 10 - epochs: 10 Accuracy: 0.6717430353164673
7 Parameters: batch_size: 10 - epochs: 50 Accuracy: 0.6722549200057983
8 Parameters: batch_size: 10 - epochs: 100 Accuracy: 0.6727668046951294
9 Parameters: batch_size: 15 - epochs: 5 Accuracy: 0.6691195368766785
10 Parameters: batch_size: 15 - epochs: 10 Accuracy: 0.6693114638328552
11 Parameters: batch_size: 15 - epochs: 50 Accuracy: 0.6723189353942871
12 Parameters: batch_size: 15 - epochs: 100 Accuracy: 0.6735346913337708
13 Parameters: batch_size: 20 - epochs: 5 Accuracy: 0.6681597232818604
14 Parameters: batch_size: 20 - epochs: 10 Accuracy: 0.6731507778167725
15 Parameters: batch_size: 20 - epochs: 50 Accuracy: 0.6696954369544983
16 Parameters: batch_size: 20 - epochs: 100 Accuracy: 0.6725108623504639
```

From the results, we can see that the batch size = 15 and epochs = 100 are the best parameters as they have the highest accuracy in the model.

	TrialNumber	Parameters	Accuracy
0	12	batch_size15-epoch100	0.673535

Below is the line graph to show the accuracy of the model prediction by using these parameters.

Graph Parameters Against Accuracy



After obtaining the parameters, the model is trained by using the parameters, then predict the engine condition from the testing set. The probability threshold value is set as 0.5, hence the predicted value will be 1 if the predicted probability value is higher than or equal to 0.5, otherwise will be 0. Here are some samples from the table of prediction.

```
   Engine_rpm  Lub_oil_pressure  ...  PredictedProb  Predicted
0       682.0          2.391656  ...       0.612737          1
1       605.0          5.466877  ...       0.857289          1
2       658.0          3.434232  ...       0.589744          1
3       749.0          2.094656  ...       0.521460          1
4       676.0          3.538228  ...       0.726733          1

[5 rows x 9 columns]
```

```
Testing Accuracy Results:
              precision    recall  f1-score   support

           0       0.55      0.39      0.46      1459
           1       0.69      0.81      0.75      2448

    accuracy                           0.65      3907
   macro avg       0.62      0.60      0.60      3907
weighted avg       0.64      0.65      0.64      3907
```

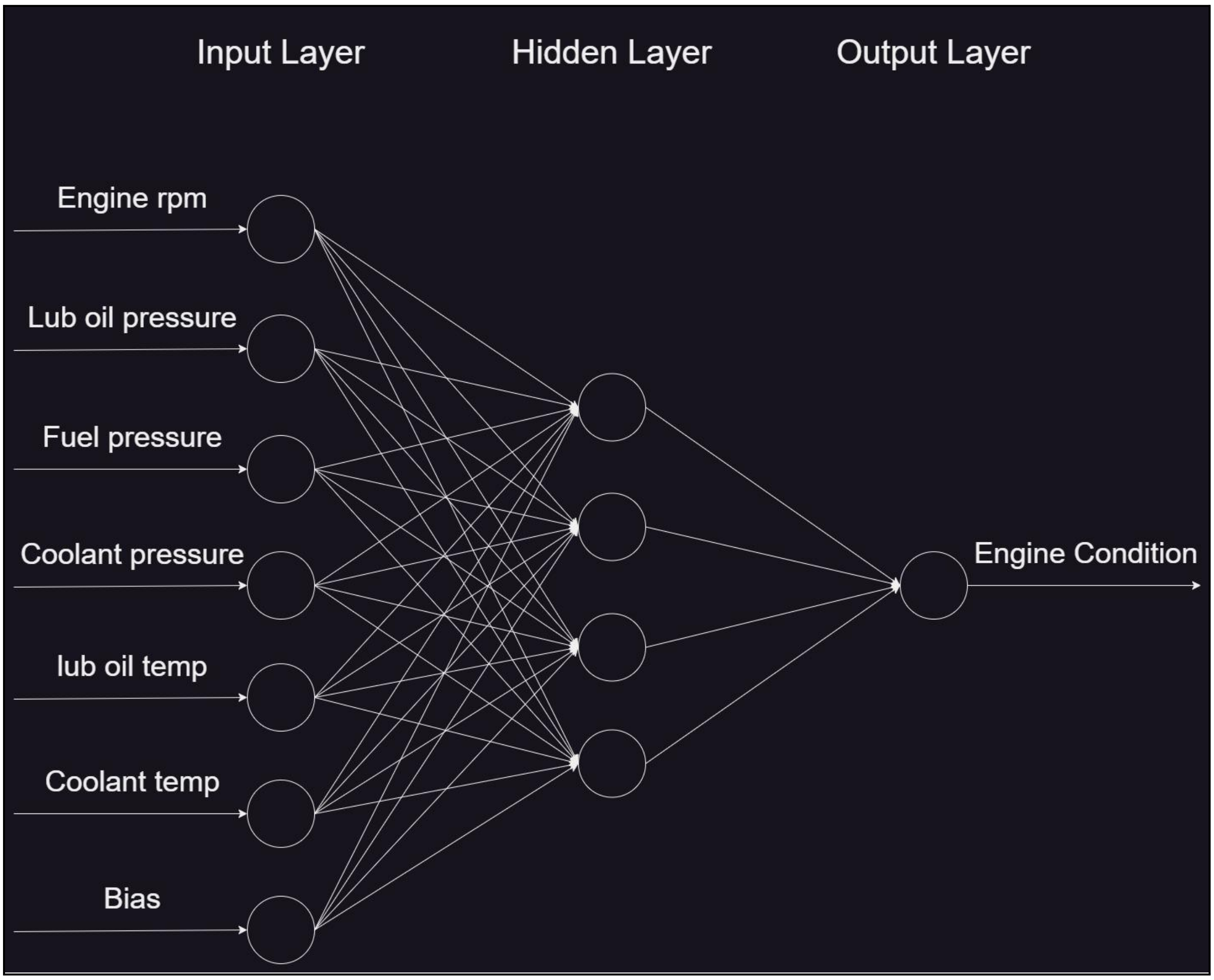
Precision is the value of the total number of correct positive predictions made and recall is the value of the total number of positive cases that are correctly predicted. From the testing accuracy results, precision and recall of 1 are higher than 0 which are 0.69 and 0.81 respectively. In short, the accuracy is 0.65 with 3907 of samples.

```
[[ 574  885]
 [ 469 1979]]
```

This is the confusion matrix for the results. We can see that true positive is 574, false positive is 885, false negative is 469 and true negative is 1979 from 3907 sample data.

7. DECISION / BUSINESS INTELLIGENCE MODEL

ANN Business Intelligence Model



8. CONCLUSION

Automatic vehicle engines can be used in many fields to provide power to the connected machine. However, the condition of the engine is hard to predict if the person is not having the skills and knowledge about the engine. This model helps them to predict the condition of the engine from some measurable variables. The model uses Manual Grid Search to find out the best parameters for the prediction to improve the accuracy. This is important because incorrect prediction may cause permanent damage to the engine due to late maintenance. After that, an artificial neural network (ANN) is used to predict the condition of the engine. From the results of the prediction, the accuracy of the prediction is about 65%. This accuracy is acceptable but still has a lot of improvement space. However, the recall of the 1 is 81%, which means most of the engines that need maintenance are predicted correctly. Although the accuracy of the model is not high enough, the model still can ensure most of the engine that is having problems from getting timely maintenance.

9. REFERENCES

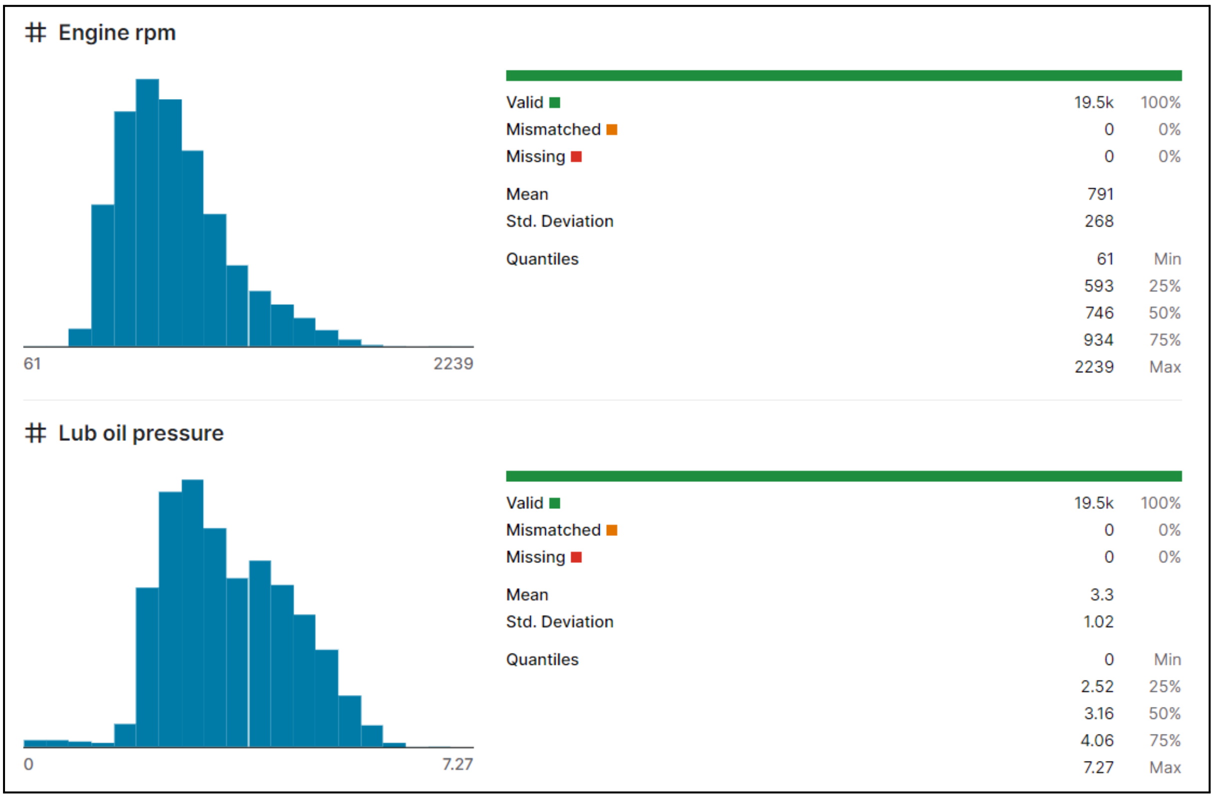
1. Modi, P. (2023, April 5). Automotive Vehicles Engine Health Dataset. Kaggle. https://www.kaggle.com/datasets/parvmodi/automotive-vehicles-engine-health-dataset
2. Vehicle Technologies Office. (2013, November 22). Internal Combustion Engine Basics. Energy.gov. https://www.energy.gov/eere/vehicles/articles/internal-combustion-engine-basics#:~:text=The%20engine%20consists%20of%20a,motion%20drives%20the%20vehicle's%20wheels
3. Rymax Lubricants. (2020, September 9). What is high oil pressure and how to solve it?. Rymax Lubricants. https://www.rymax-lubricants.com/updates/what-is-high-oil-pressure-and-how-to-solve-it/
4. Farukh Hashmi. (2021, November 7). How to use Artificial Neural Networks for classification in python?. Thinking Neurons. https://thinkingneuron.com/how-to-use-artificial-neural-networks-for-classification-in-python/
5. David Landup. (2023, May 17) How to Set Axis Range (xlim, ylim) in Matplotlib. Stack Abuse. https://stackabuse.com/how-to-set-axis-range-xlim-ylim-in-matplotlib/
6. Rob Hyndman. (2010, July 20) How to choose the number of hidden layers and nodes in a feedforward neural network?. Stack Exchange. https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw
7. Teemu Kanstrén. (2020, September 20). A Look at Precision, Recall, and F1-Score. Towards Data Science. https://towardsdatascience.com/a-look-at-precision-recall-and-f1-score-36b5fd0dd3ec
8. Guofeng Ma, Ying Liu, and Shanshan Shang. (2019 September 11). A Building Information Model (BIM) and Artificial Neural Network (ANN) Based System for Personal Thermal Comfort Evaluation and Energy Efficient Design of Interior Space. MDPI. https://www.mdpi.com/2071-1050/11/18/4972

10. APPENDIX

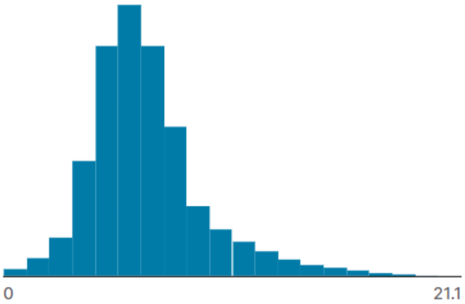
User Manual

1. Unzip the file “Project Code”.
2. Open folder “Project Code”, open “project.py” using PyCharm Community Edition 2022.2.3 or above.
3. Run the program to get the results.
4. While the program is running, line graph for the model accuracy will pop out, closing it to continue the run.
5. The output in the terminal at bottom is the results of the model.
6. If you want to change the ratio of splitting training and testing data, go to Line 38 and change the test_size value to any number from 0 to 1.
7. If you want to change the parameters, go to Line 61 to change the batch_size_list value with any number greater than 0 to change the size of the data batch, and Line 62 to change the epoch_list value to with any number greater than 0 to change the number of iteration.

Dataset

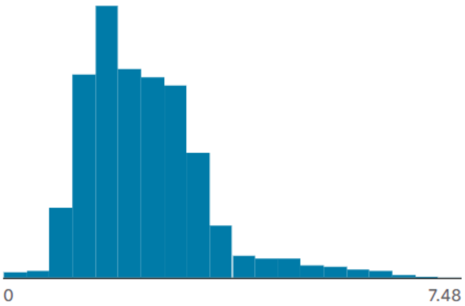


Fuel pressure



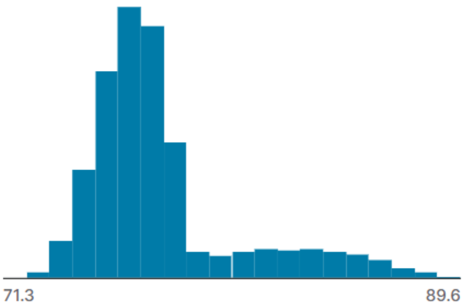
Valid	19.5k	100%
Mismatched	0	0%
Missing	0	0%
Mean	6.66	
Std. Deviation	2.76	
Quantiles	0	Min
	4.92	25%
	6.2	50%
	7.74	75%
	21.1	Max

Coolant pressure



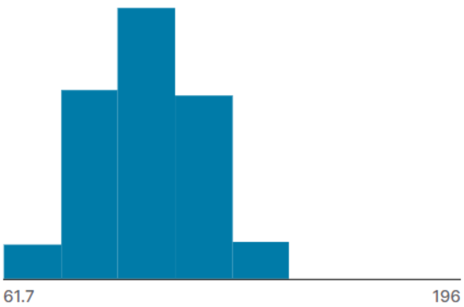
Valid	19.5k	100%
Mismatched	0	0%
Missing	0	0%
Mean	2.34	
Std. Deviation	1.04	
Quantiles	0	Min
	1.6	25%
	2.17	50%
	2.85	75%
	7.48	Max

lub oil temp



Valid	19.5k	100%
Mismatched	0	0%
Missing	0	0%
Mean	77.6	
Std. Deviation	3.11	
Quantiles	71.3	Min
	75.7	25%
	76.8	50%
	78.1	75%
	89.6	Max

Coolant temp



Valid	19.5k	100%
Mismatched	0	0%
Missing	0	0%
Mean	78.4	
Std. Deviation	6.21	
Quantiles	61.7	Min
	73.9	25%
	78.3	50%
	82.9	75%
	196	Max

Engine Condition

Valid	19.5k	100%
Mismatched	0	0%
Missing	0	0%
Mean	0.63	
Std. Deviation	0.48	
Quantiles	0	Min
	0	25%
	1	50%
	1	75%
	1	Max

Source Code

```
import torch
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler, StandardScaler, MaxAbsScaler,
RobustScaler
from torch.utils.data import TensorDataset, DataLoader
from sklearn.metrics import mean_absolute_error, mean_squared_error  # ,
mean_absolute_percentage_error

import warnings

warnings.filterwarnings('ignore')

device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"{device}" " is available.")


def plot_dataset_trendline(df, title, label='condition'):
   plt.figure(figsize=(10, 7))
   sns.lineplot(x=df.index, y=df.condition, label=label)

   # plot trendline (have to be clean from missing data)
   x = range(0, len(df))
   z = np.polyfit(x, y=df.condition, deg=1)
   p = np.poly1d(z)
   plt.plot(df.index, p(x), c="r", ls='-')

   plt.title(title)
   plt.xlabel('Engine RPM')
   plt.ylabel('Engine Condition')
   plt.show()

def onehot_encode_pd(df, cols):
   for col in cols:
       dummies = pd.get_dummies(df[col], prefix=col)
       df = pd.concat([df, dummies], axis=1)  # .drop(columns=col)
   return df

def generate_cyclical_features(df, col_name, period, start_num=0):
   kwargs = {
       f'sin_{col_name}': lambda x: np.sin(2 * np.pi * (df[col_name] -
start_num) / period),
       f'cos_{col_name}': lambda x: np.cos(2 * np.pi * (df[col_name] -
start_num) / period)
   }
   return df.assign(**kwargs).drop(columns=[col_name])

# function for splitting target and predictor variable
def feature_label_split(df, target_col):
   y = df[[target_col]]
   X = df.drop(columns=[target_col])
   return X, y

# function for splitting data to train, test, and validation data
```

```
def train_val_test_split(df, target_col, test_ratio):
   val_ratio = test_ratio / (1 - test_ratio)
   X, y = feature_label_split(df, target_col)
   X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=test_ratio, shuffle=False)
   X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=val_ratio, shuffle=False)
   return X_train, X_val, X_test, y_train, y_val, y_test

# function to pick scaler
def get_scaler(scaler):
   scalers = {
       "minmax": MinMaxScaler,
       "standard": StandardScaler,
       "maxabs": MaxAbsScaler,
       "robust": RobustScaler,
   }
   return scalers.get(scaler.lower())()

# inverse the scale from model result
def inverse_transform(scaler, df, columns):
   for col in columns:
       df[col] = scaler.inverse_transform(df[col])
   return df

def calculate_metrics(df):
   result_metrics = {'mae': mean_absolute_error(df.value, df.prediction),
                     'rmse': np.sqrt(mean_squared_error(df.value,
df.prediction)), }
   #                        'mape' : mean_absolute_percentage_error(df.value,
df.prediction)}

   print("Mean Absolute Error:       ", result_metrics["mae"])
   print("Root Mean Squared Error:   ", result_metrics["rmse"])
   #     print("MAPE Score:                 ", result_metrics["mape"])

   return result_metrics

def format_predictions(predictions, values, df_test, scaler):
   vals = np.concatenate(values, axis=0).ravel()
   preds = np.concatenate(predictions, axis=0).ravel()
   df_result = pd.DataFrame(data={"value": vals, "prediction": preds},
index=df_test.head(len(vals)).index)
   df_result = df_result.sort_index()
   df_result = inverse_transform(scaler, df_result, [["value",
"prediction"]])
   return df_result

def plot_all(df_result, df_train, df_val):
   plt.figure(figsize=(12, 5))
   observation = sns.lineplot(x=df_train.index, y=df_train.condition,
color='k', alpha=0.3, label='observation')
   validation = sns.lineplot(x=df_val.index, y=df_val.prediction, color='b',
alpha=0.8, label='validation')
   prediction = sns.lineplot(x=df_result.index, y=df_result.prediction,
color='g', alpha=0.8, label='prediction')

   plt.title("Observation, Validation, and Prediction Values of Automotive
Vehicles Engine Health")
   plt.xlabel('Engine RPM')
   plt.ylabel('Engine Condition')
```

```
    plt.show()

def plot_predictions(df_result):
    plt.figure(figsize=(12, 5))
    value = sns.lineplot(x=df_result.index, y=df_result.condition, color='k',
alpha=0.3, label='condition')
    prediction = sns.lineplot(x=df_result.index, y=df_result.prediction,
color='g', alpha=0.8, label='prediction')

    plt.title("Predictions vs Actual Values of Automotive Vehicles Engine
Health")
    plt.xlabel('Engine RPM')
    plt.ylabel('Engine Condition')

def plot_dataset_trendline_after_model(df, title):
    plt.figure(figsize=(12, 5))
    # plot dataset
    sns.lineplot(x=df.index, y=df.condition, color='k', alpha=0.3,
label='condition')
    sns.lineplot(x=df.index, y=df.prediction, color='g', alpha=0.5,
label='prediction')

    # plot trendline of observation value
    x = range(0, len(df))
    z = np.polyfit(x, y=df.condition, deg=1)
    p = np.poly1d(z)
    plt.plot(df.index, p(x), c="k", ls='-')

    # plot trendline of prediction (have to be clean from missing data)
    x = range(0, len(df))
    z = np.polyfit(x, y=df.prediction, deg=1)
    p = np.poly1d(z)
    plt.plot(df.index, p(x), c="g", ls='-')

    plt.title(title)
    plt.xlabel('Engine RPM')
    plt.ylabel('Engine Condition')
    plt.show()

df = pd.read_csv('C:/Users/HP/Downloads/engine_data.csv')

# drop data in first line because its not technically data that we need
df.drop([0], inplace=True)

# make a new dataset for plotting (because we need to drop some missing
values)
df_plot = df.dropna()
df_plot = df_plot.set_index(['Engine_rpm'])
df_plot = df_plot.rename(columns={'Engine_Condition': 'condition'})
plot_dataset_trendline(df_plot, 'Automotive Vehicles Engine Health')

df_features = (df_plot
              .assign(Lub_oil_pressure=df_plot.Lub_oil_pressure)
              .assign(Fuel_pressure=df_plot.Fuel_pressure)
              .assign(Coolant_pressure=df_plot.Coolant_pressure)
              .assign(lub_oil_temp=df_plot.lub_oil_temp)
              .assign(Coolant_temp=df_plot.Coolant_temp)
              )

# one-hot encoding for categorical value from datetime feature
```

```
df_features = onehot_encode_pd(df_features, ['Lub_oil_pressure',
'Fuel_pressure', 'Coolant_pressure',
                                            'lub_oil_temp', 'Coolant_temp'])

X_train, X_val, X_test, y_train, y_val, y_test =
train_val_test_split(df_features, 'condition', 0.2)
scaler = get_scaler('minmax')

# fit and apply scaler to predictor variable
X_train_arr = scaler.fit_transform(X_train)
X_val_arr = scaler.transform(X_val)
X_test_arr = scaler.transform(X_test)

# fit and apply scaler to target variable
y_train_arr = scaler.fit_transform(y_train)
y_val_arr = scaler.transform(y_val)
y_test_arr = scaler.transform(y_test)

batch_size = 32

# convert data shape to tensor (multi-dimensional matrix containing elements
of a single data type)
train_features = torch.Tensor(X_train_arr)
train_targets = torch.Tensor(y_train_arr)
val_features = torch.Tensor(X_val_arr)
val_targets = torch.Tensor(y_val_arr)
test_features = torch.Tensor(X_test_arr)
test_targets = torch.Tensor(y_test_arr)

# wrapping the tensors above as Dataset
train = TensorDataset(train_features, train_targets)
val = TensorDataset(val_features, val_targets)
test = TensorDataset(test_features, test_targets)

# convert data to Pytorch DataLoader (collating data samples into batches)
train_loader = DataLoader(train, batch_size=batch_size, shuffle=False,
drop_last=True)
val_loader = DataLoader(val, batch_size=batch_size, shuffle=False,
drop_last=True)
test_loader = DataLoader(test, batch_size=batch_size, shuffle=False,
drop_last=True)


class LSTMModel(nn.Module):
   """LSTMModel class extends nn.Module class and works as a constructor
for LSTMs.

      LSTMModel class initiates a LSTM module based on PyTorch's nn.Module
class.
      It has only two methods, namely init() and forward(). While the
init()
      method initiates the model with the given input parameters, the
forward()
      method defines how the forward propagation needs to be calculated.
      Since PyTorch automatically defines back propagation, there is no
need
      to define back propagation method.

      --Attributes--
          hidden_dim: int
              The number of nodes in each layer
```

```
        layer_dim: int
            The number of layers in the network
        lstm: nn.LSTM
            The LSTM model constructed with the input parameters.
        fc: nn.Linear
            The fully connected layer to convert the final state of LSTMs
to our desired output shape.

    """

    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim,
dropout_prob):
        """The __init__ method that initiates a LSTM instance.

        --Arguments--
            input_dim: int
                The number of nodes in the input layer
            hidden_dim: int
                The number of nodes in each layer
            layer_dim: int
                The number of layers in the network
            output_dim: int
                The number of nodes in the output layer
            dropout_prob: float
                The probability of nodes being dropped out

        """
        super(LSTMModel, self).__init__()

        # Defining the number of layers and the nodes in each layer
        self.hidden_dim = hidden_dim
        self.layer_dim = layer_dim

        # LSTM layers
        self.lstm = nn.LSTM(input_dim, hidden_dim, layer_dim,
batch_first=True, dropout=dropout_prob)

        # Fully connected layer
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        """The forward method takes input tensor x and does forward
propagation

        --Arguments--
            x: torch.Tensor
                The input tensor of the shape (batch size, sequence length,
input_dim)

        --Returns--
            out: torch.Tensor
                The output tensor of the shape (batch size, output_dim)

        """
        # Initializing hidden state for first input with zeros
        h0 = torch.zeros(self.layer_dim, x.size(0),
self.hidden_dim).requires_grad_()

        # Initializing cell state for first input with zeros
        c0 = torch.zeros(self.layer_dim, x.size(0),
self.hidden_dim).requires_grad_()
```

```
        # We need to detach as we are doing truncated backpropagation through
time (BPTT)
        # If we don't, we'll backprop all the way to the start even after
going through another batch
        # Forward propagation by passing in the input, hidden state, and cell
state into the model
        out, (hn, cn) = self.lstm(x, (h0.detach(), c0.detach()))

        # Reshaping the outputs in the shape of (batch_size, seq_length,
hidden_size)
        # so that it can fit into the fully connected layer
        out = out[:, -1, :]

        # Convert the final state to our desired output shape (batch_size,
output_dim)
        out = self.fc(out)

        return out


class Optimization:
    """
    Optimization is a helper class that takes model, loss function,
optimizer function
    learning scheduler (optional), early stopping (optional) as inputs. In
return, it
    provides a framework to train and validate the models, and to predict
future values
    based on the models.

    --Attributes--
        model:
            Model class created for the type of RNN
        loss_fn: torch.nn.modules.Loss
            Loss function to calculate the losses
        optimizer: torch.optim.Optimizer
            Optimizer function to optimize the loss function
        train_losses: list[float]
            The loss values from the training
        val_losses: list[float]
            The loss values from the validation
    """

    def __init__(self, model, loss_fn, optimizer):
        self.model = model
        self.loss_fn = loss_fn
        self.optimizer = optimizer
        self.train_losses = []
        self.val_losses = []

    def train_step(self, x, y):
        """
        Given the features (x) and the target values (y) tensors, the method
completes
        one step of the training. First, it activates the train mode to
enable back prop.
        After generating predicted values (yhat) by doing forward
propagation, it calculates
        the losses by using the loss function. Then, it computes the
gradients by doing
```

```
        back propagation and updates the weights by calling step() function.

        --Arguments--
            x: torch.Tensor
                Tensor for features to train one step
            y: torch.Tensor
                Tensor for target values to calculate losses

        """
        # Sets model to train mode
        self.model.train()

        # Makes predictions
        yhat = self.model(x)

        # Computes loss
        loss = self.loss_fn(y, yhat)

        # Computes gradients
        loss.backward()

        # Updates parameters and zeroes gradients
        self.optimizer.step()
        self.optimizer.zero_grad()

        # Returns the loss
        return loss.item()

    def train(self, train_loader, val_loader, batch_size=64, n_epochs=50,
n_features=1):
        """
        The method takes DataLoaders for training and validation datasets,
batch size for
        mini-batch training, number of epochs to train, and number of
features as inputs.
        Then, it carries out the training by iteratively calling the method
train_step for
        n_epochs times. Finally, it saves the model in a designated file
path.

        --Arguments--
            train_loader: torch.utils.data.DataLoader
                DataLoader that stores training data
            val_loader: torch.utils.data.DataLoader
                DataLoader that stores validation data
            batch_size: int
                Batch size for mini-batch training
            n_epochs: int
                Number of epochs, i.e., train steps, to train
            n_features: int
                Number of feature columns

        """
        model_path = f'model_lstm'

        for epoch in range(1, n_epochs + 1):
            # mini-batch training iteration of training datasets
            batch_losses = []
            for x_batch, y_batch in train_loader:
                x_batch = x_batch.view([batch_size, -1,
n_features]).to(device)
```

```
                y_batch = y_batch.to(device)
                loss = self.train_step(x_batch, y_batch)
                batch_losses.append(loss)
                # update training loss value
            training_loss = np.mean(batch_losses)
            self.train_losses.append(training_loss)

            with torch.no_grad():
                # mini-batch training iteration of validation datasets
                batch_val_losses = []
                validation = []
                validation_values = []
                for x_val, y_val in val_loader:
                    x_val = x_val.view([batch_size, -1,
n_features]).to(device)
                    y_val = y_val.to(device)
                    self.model.eval()
                    yhat = self.model(x_val)
                    val_loss = self.loss_fn(y_val, yhat).item()
                    batch_val_losses.append(val_loss)
                    #
                    validation.append(yhat.to(device).detach().numpy())

validation_values.append(y_val.to(device).detach().numpy())
                # update validation loss value
                validation_loss = np.mean(batch_val_losses)
                self.val_losses.append(validation_loss)

            # print loss value per epoch period
            if (epoch <= 10) | (epoch % 10 == 0):
                print(
                    f"[{epoch}/{n_epochs}] Training loss:
{training_loss:.4f}\t Validation loss: {validation_loss:.4f}"
                )

        torch.save(self.model.state_dict(), model_path)
        return validation, validation_values

    def evaluate(self, test_loader, batch_size=1, n_features=1):
        """
        The method takes DataLoaders for the test dataset, batch size for
mini-batch testing,
        and number of features as inputs. Similar to the model validation,
it iteratively
        predicts the target values and calculates losses. Then, it returns
two lists that
        hold the predictions and the actual values.

        Note:
            This method assumes that the prediction from the previous step
is available at
            the time of the prediction, and only does one-step prediction
into the future.

        --Arguments--
            test_loader: torch.utils.data.DataLoader
                DataLoader that stores test data
            batch_size: int
                Batch size for mini-batch training
            n_features: int
                Number of feature columns
```

```
        --Returns--
            predictions: list[float]
                The values predicted by the model
            values: list[float]
                The actual values in the test set.

        """
        # mini-batch testing to evaluate data from test dataset
        with torch.no_grad():
            predictions = []
            values = []
            for x_test, y_test in test_loader:
                x_test = x_test.view([batch_size, -1, n_features]).to(device)
                y_test = y_test.to(device)
                self.model.eval()
                yhat = self.model(x_test)
                # save model prediction result to list
                predictions.append(yhat.to(device).detach().numpy())
                values.append(y_test.to(device).detach().numpy())

        return predictions, values

    def predict(self, future_loader, batch_size=1, n_features=1):
        """
        The method takes DataLoaders for the predicting future dataset,
batch size for mini-batch testing,
        and number of features as inputs.

        --Arguments--
            test_loader: torch.utils.data.DataLoader
                DataLoader that stores test data
            batch_size: int
                Batch size for mini-batch training
            n_features: int
                Number of feature columns

        --Returns--
            predictions: list[float]
                The values predicted by the model

        """
        # mini-batch testing to predict data from future dataset
        with torch.no_grad():
            predictions = []
            for x_test in test_loader:
                x_test = x_test.view([batch_size, -1, n_features]).to(device)
                self.model.eval()
                yhat = self.model(x_test)
                # save model prediction result to list
                predictions.append(yhat.to(device).detach().numpy())

        return predictions

    def plot_losses(self):
        """
        The method plots the calculated loss values for training and
validation
        """
        plt.figure(figsize=[8, 5])
        plt.plot(self.train_losses, label="Training loss")
```

```
        plt.plot(self.val_losses, label="Validation loss")
        plt.legend()
        plt.title("Losses")
        plt.show()
        plt.close()


# LSTM config
input_dim = len(X_train.columns)
output_dim = 1
hidden_dim = 64
layer_dim = 4
batch_size = batch_size
dropout = 0.05
# training and evaluate config
n_epochs = 30
# weight optimization config
learning_rate = 1e-3
weight_decay = 1e-6

# bundle config in dictionary
model_params = {'input_dim': input_dim,
               'hidden_dim': hidden_dim,
               'layer_dim': layer_dim,
               'output_dim': output_dim,
               'dropout_prob': dropout}

model = LSTMModel(**model_params)

# set criterion to calculate loss gradient
loss_fn = nn.MSELoss(reduction="mean")
# set model optimizer (process of adjusting model parameters to reduce model
error in each training step)
optimizer = optim.AdamW(model.parameters(),
                       lr=learning_rate,
                       weight_decay=weight_decay)

# training model
opt = Optimization(model=model,
                  loss_fn=loss_fn,
                  optimizer=optimizer)
validation, validation_values = opt.train(train_loader,
                                         val_loader,
                                         batch_size=batch_size,
                                         n_epochs=n_epochs,
                                         n_features=input_dim)
opt.plot_losses()

# evaluate model based on model from training dataset
predictions, values = opt.evaluate(test_loader, batch_size=batch_size,
n_features=input_dim)
df_val = format_predictions(validation, validation_values, X_val, scaler)
df_result = format_predictions(predictions, values, X_test, scaler)
result_metrics = calculate_metrics(df_result)

plot_predictions(df_result)
plot_all(df_result, df_features, df_val)
plot_dataset_trendline_after_model(df_result, 'Automotive Vehicles Engine
Health')
```