Direct Links to deep-learning projects on google colab.

Real-Estate Pricing
https://colab.research.google.com/drive/1aOt72W-Fyo0b7PpodrBAizjKY5tUk_Lv

Pest detection using CNN model
https://colab.research.google.com/drive/16tq12Rizl7IB0kyyQAOzpnCmKRl5lP7J

Weather forecasting on time-series analysis
https://colab.research.google.com/drive/13GOETsIkH8nkzpUdkdQvQxNiJEi_gD9-#

MIS780 Advanced AI For Business - Assignment 2 - T2 2023

Task 1: Real estate analytics with tabular data

Student Name: Aman Rajput

Student ID: 221069377

Table of Content

1. Executive Summary
2. Data Preprocessing
3. Predictive Modeling
4. Experiments Report

1. Executive Summary

In the realm of real estate, the need for accurate pricing is paramount. Buyers yearn for insights into the justifiable worth of homes they're eyeing, while sellers strive to pinpoint competitive price points that promise optimal returns. Equally vital are the efforts of real estate professionals, dedicated to delivering precise pricing recommendations. Within the real estate landscape, data-driven insights wield transformative influence. This underscores the immense value of a dependable house price prediction model.

This study is centered on a critical business challenge: predicting house prices in King County, USA. Here, we scrutinize a dataset comprising comprehensive details of more than 21,613 home sales (*Section 2*). This treasure trove of information encompasses factors like square footage, location, bedroom count, and most notably, house prices. The goal is to harness data analytics and machine learning to empower stakeholders, including buyers, sellers, and real estate professionals, with accurate and actionable insights. Stakeholders in the real estate industry benefit from data-driven insights that enhance their ability to make informed decisions, driving efficiency and success.

- **Regression Models (Linear Regression vs. Neural Networks):** In the first scenario, the models focus on estimating house prices accurately (*Section 3 Part A*).
- **Classification Models (Neural Networks vs. XGBoost):** The second scenario involves classifying houses into "High_Price" and "Low_Price" categories based on price. The evaluation focuses on two classification models: Neural Networks and XGBoost (*Section 3 Part B*).

The models' complexity increases as they move from linear/XGB to NN and in order to assess the practicality of these models, following technical metrics are connected with business solutions: MSE, MAE, R-squared (training and validation), classification reports, confusion matrix and the Kappa Score.

Part (a) and business implication

For the first part, the estimation of house prices, both the Linear Regression and Neural Network models are evaluated for their performance. The results indicate that the Neural Network model outshines the Linear Regression model in multiple aspects. The Neural Network yields a notably lower Mean Squared Error (MSE) and Mean Absolute Error (MAE) compared to the Linear Regression model, implying more accurate predictions and a closer alignment with actual house prices (*Section 4 Part A*). These results suggest that the Neural Network's intricate architecture excels in capturing complex relationships within the data, resulting in more precise house price estimations. Furthermore, the correlation coefficients between true and predicted values for both models reinforce their reliability, with the Neural Network exhibiting a notably higher correlation of 0.939, further underscoring its effectiveness in addressing the crucial business challenge of house price estimation. This superior predictive accuracy is pivotal for stakeholders in the real estate industry, enhancing their ability to make informed pricing decisions and ultimately optimizing returns.

Part (b) and business implication

For second scenario, the XGBoost and Neural Network classification models exhibit exceptional performance in categorizing houses into "High Price" and "Low Price" groups. Both models achieve an impressive accuracy of approximately 91%, highlighting their ability to correctly classify houses based on their price categories in the validation dataset (*Section 4 Part B*). Additionally, both models demonstrate substantial agreement between their predictions and the actual classifications, as indicated by Kappa Scores of 0.811 for XGBoost and 0.815 for the Neural Network. The classification reports further emphasize their effectiveness, with high precision, recall, and F1-scores for both price categories. These robust classification models empower real estate professionals and stakeholders in King County, USA, providing them with the means to make well-informed decisions and tailored strategies to navigate the dynamic real estate market successfully.

2. Data Preprocessing

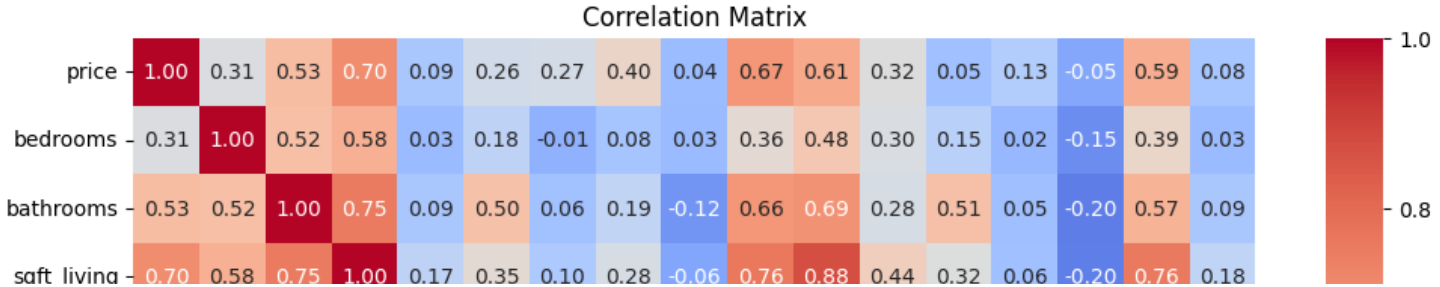
```
#Importing all the necesarry libraries and modules

import os
import math
import datetime
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import cross_val_score
```

```
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_absolute_error
from sklearn.feature_selection import SelectKBest, f_regression
```

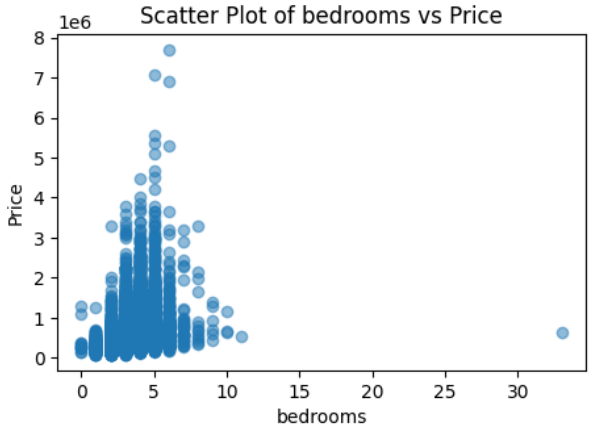
Part (A)

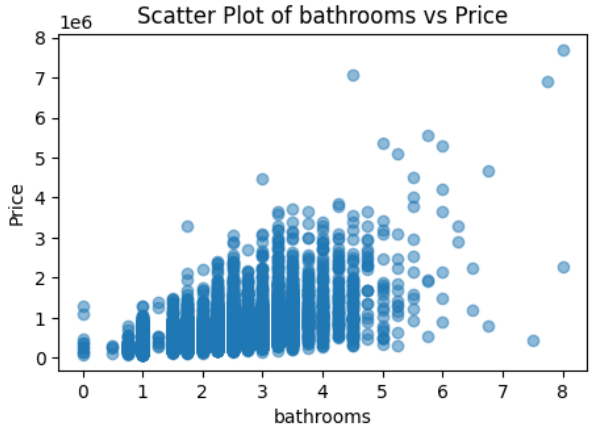
```
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)


ames_data = pd.read_csv("/content/Part1_house_price.csv")
ames_data.set_index('id', inplace=True)
ames_data.head()
#print('Number of records read: ', ames_data_org.size)

# Finding column types
ames_data.dtypes

# Identification of missing values
missing = ames_data.isnull().sum()
missing = missing[missing > 0]
missing.sort_values(ascending=False)

#No missing values were found in the dataset but need to drop irrelavent attributes which are not really numerical
ames_data = ames_data.drop(['date','lat','long'], axis=1)


# Generate a correlation matrix
correlation_matrix = ames_data.corr()

# Creating a heatmap for visualization
plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Correlation Matrix")
plt.show()

# Identify attributes with high correlation
threshold = 0.7
high_corr_pairs = np.where(np.abs(correlation_matrix) > threshold)

# Print pairs of attributes with high correlation
for i, j in zip(*high_corr_pairs):
    if i != j and i < j:
```

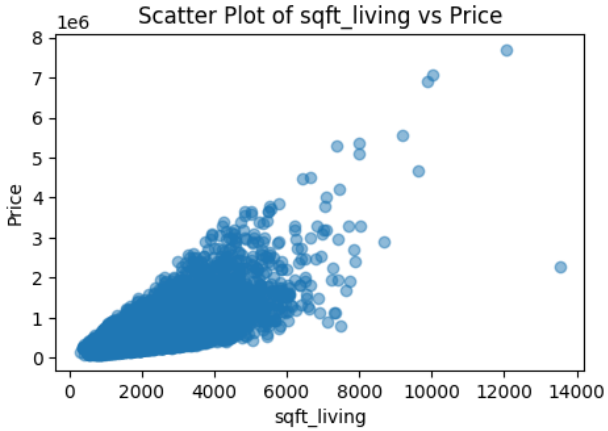
```
print(f"{ames_data.columns[i]} - {ames_data.columns[j]}: {correlation_matrix.iloc[i, j]:.2f}")
```

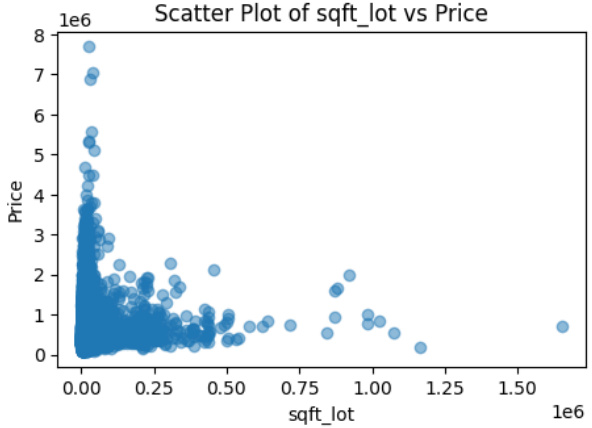


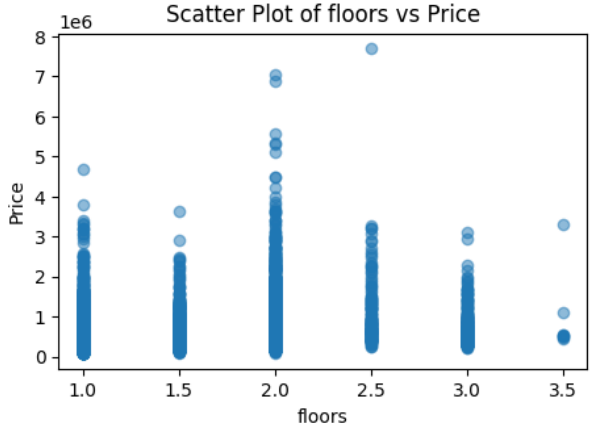

```
# List of features
features = ames_data.columns.tolist()
features.remove('price')  # Remove the target label

# Determine the number of rows and columns for the grid
num_features = len(features)
num_cols = 3                                    # Number of columns in the grid
num_rows = int(np.ceil(num_features / num_cols))  # Number of rows in the grid

# Create scatter plots for linearity check
fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 20))
fig.tight_layout(pad=3.0)

for idx, feature in enumerate(features):
    row_idx = idx // num_cols
    col_idx = idx % num_cols
    ax = axes[row_idx, col_idx]

    ax.scatter(ames_data[feature], ames_data['price'], alpha=0.5)
    ax.set_title(f"Scatter Plot of {feature} vs Price")
    ax.set_xlabel(feature)
    ax.set_ylabel('Price')

# Remove any empty subplots
for i in range(num_features, num_rows * num_cols):
    row_idx = i // num_cols
    col_idx = i % num_cols
    fig.delaxes(axes[row_idx, col_idx])

plt.show()
```

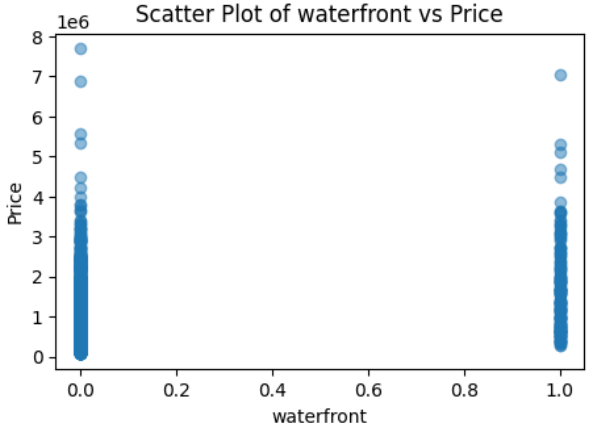

Scatter Plot of bedrooms vs Price
1e6
Price
bedrooms

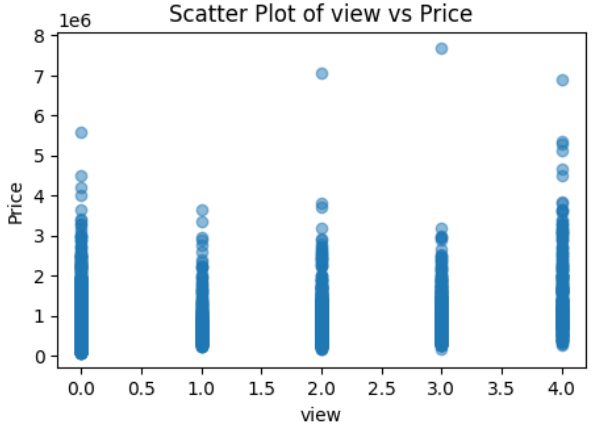

Scatter Plot of bathrooms vs Price
1e6
Price
bathrooms

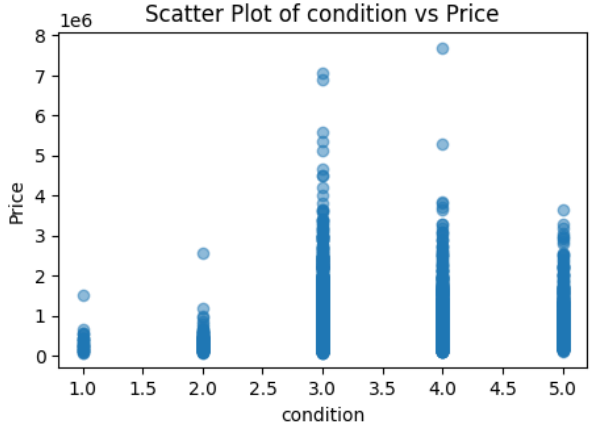

Scatter Plot of sqft_living vs Price
1e6
Price
sqft_living

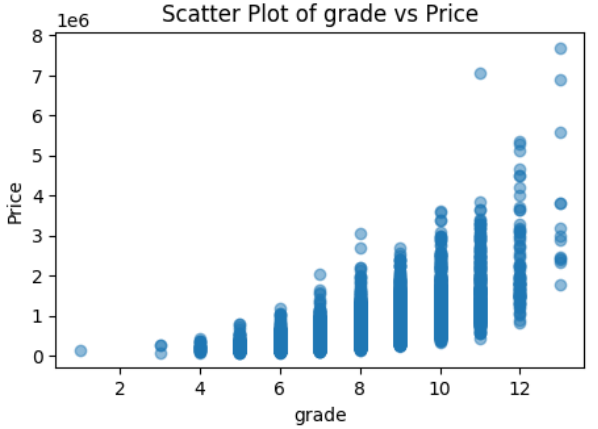

Scatter Plot of sqft_lot vs Price
1e6
Price
sqft_lot
1e6

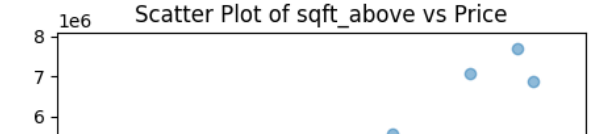

Scatter Plot of floors vs Price
1e6
Price
floors

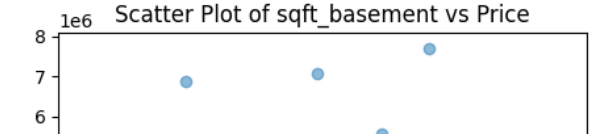

Scatter Plot of waterfront vs Price
1e6
Price
waterfront

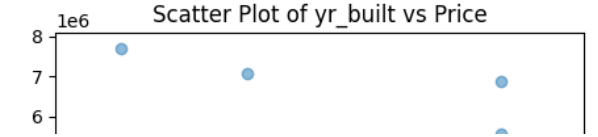

Scatter Plot of view vs Price
1e6
Price
view

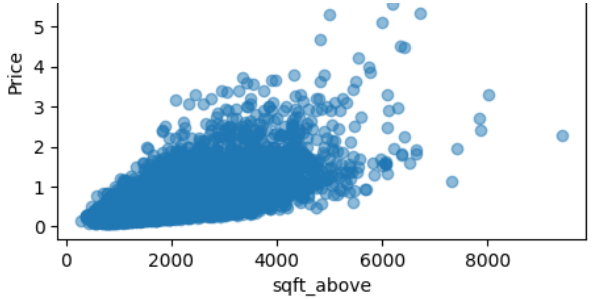

Scatter Plot of condition vs Price
1e6
Price
condition

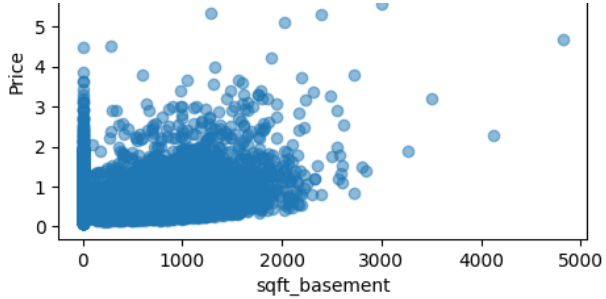

Scatter Plot of grade vs Price
1e6
Price
grade

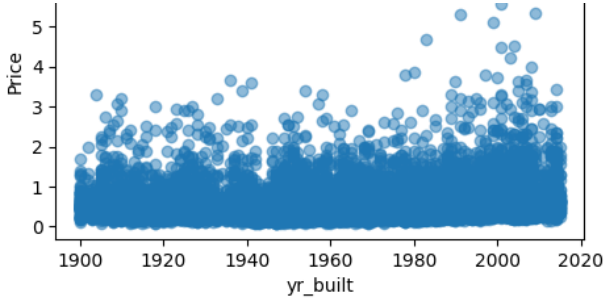

Scatter Plot of sqft_above vs Price
1e6

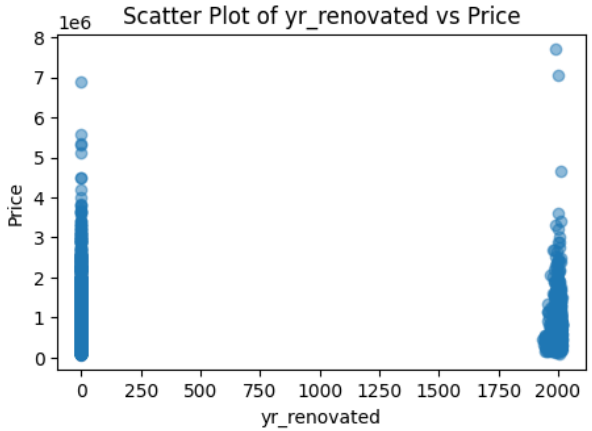

Scatter Plot of sqft_basement vs Price
1e6

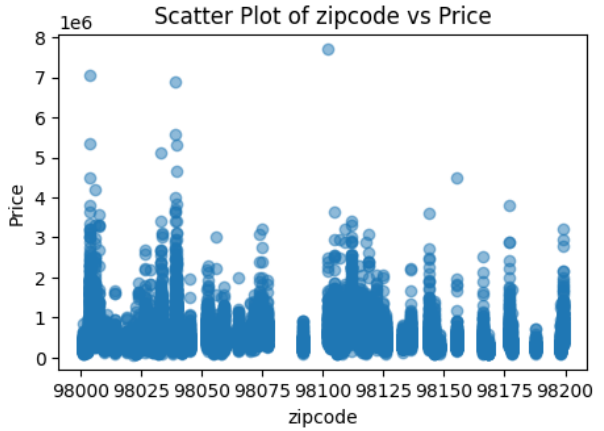

Scatter Plot of yr_built vs Price
1e6

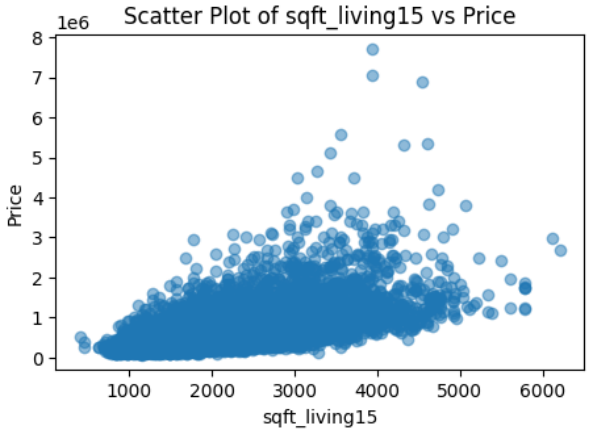


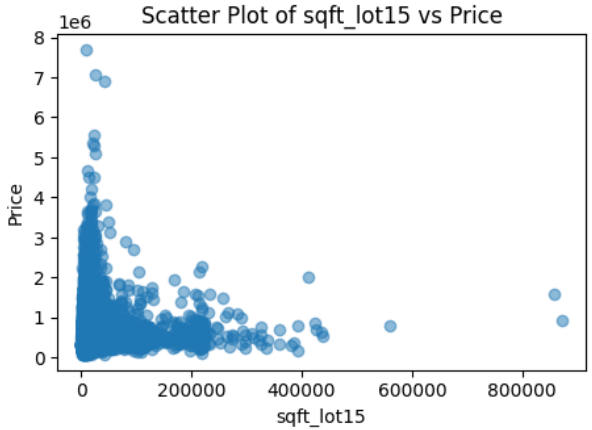













```
#Chose to keep the attributes as it hardly contributed to the perfomance of the models
#ames_data = ames_data.drop(['sqft_above','sqft_living15','sqft_lot15'], axis=1)
```

We can one-hot encode zipcodes as homes in more desirable zip codes tend to have higher values than homes in less desirable ones. (Bonsor and Strickland 2008)

```
# Select numerical features from the ames_data DataFrame
ames_data_num = ames_data.select_dtypes(include='number')

# One-hot encoding the 'zipcode' column because the property prices usually varies as per the locality in real  life
ames_data_zip = pd.get_dummies(ames_data['zipcode'], prefix='Zip')

# Concatenate numerical features and one-hot encoded zipcode columns
ames_data = pd.concat([ames_data_num, ames_data_zip], axis=1)

# Drop the original 'zipcode' column since it has been one-hot encoded
ames_data = ames_data.drop(['zipcode'], axis=1)

# Setting target variable (label)
label_col = 'price'

# Display the first 10 rows of the modified ames_data DataFrame
ames_data.head(10)
```

id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade	sqft_above	sqft_basement	yr_built	yr_ren
7129300520	221900.0	3	1.00	1180	5650	1.0	0	0	3	7	1180	0	1955	
6414100192	538000.0	3	2.25	2570	7242	2.0	0	0	3	7	2170	400	1951	
5631500400	180000.0	2	1.00	770	10000	1.0	0	0	3	6	770	0	1933	
2487200875	604000.0	4	3.00	1960	5000	1.0	0	0	5	7	1050	910	1965	
1954400510	510000.0	3	2.00	1680	8080	1.0	0	0	3	8	1680	0	1987	
7237550310	1230000.0	4	4.50	5420	101930	1.0	0	0	3	11	3890	1530	2001	
1321400060	257500.0	3	2.25	1715	6819	2.0	0	0	3	7	1715	0	1995	
2008000270	291850.0	3	1.50	1060	9711	1.0	0	0	3	7	1060	0	1963	
2414600126	229500.0	3	1.00	1780	7470	1.0	0	0	3	7	1050	730	1960	
3793500160	323000.0	3	2.50	1890	6560	2.0	0	0	3	7	1890	0	2003	

```
# Split the Ames housing data into training and validation sets.
train_size, valid_size, test_size = (0.7, 0.3, 0.0)
ames_train, ames_valid = train_test_split(ames_data,
                                  test_size=valid_size,
                                  random_state=2020)

# Get the target and features for the training and validation sets.
```

```
ames_y_train = ames_train[[label_col]]
ames_x_train = ames_train.drop(label_col, axis=1)
ames_y_valid = ames_valid[[label_col]]
ames_x_valid = ames_valid.drop(label_col, axis=1)

print('Size of training set: ', len(ames_x_train))
print('Size of validation set: ', len(ames_x_valid))
```

```
    Size of training set:  15129
    Size of validation set:  6484
```

```
# Impute missing values in the training and validation sets using the mean.
#(Not necessary as no missing values, but logical to keep the code if model is trained on new data)
print('Missing training values before imputation = ', ames_x_train.isnull().sum().sum())
print('Missing validation values before imputation = ', ames_x_valid.isnull().sum().sum())

imputer = SimpleImputer(missing_values=np.nan, strategy='mean').fit(ames_x_train)
ames_x_train = pd.DataFrame(imputer.transform(ames_x_train),
                            columns = ames_x_train.columns, index = ames_x_train.index)
ames_x_valid = pd.DataFrame(imputer.transform(ames_x_valid),
                            columns = ames_x_valid.columns, index = ames_x_valid.index)

print('Missing training values after imputation = ', ames_x_train.isnull().sum().sum())
print('Missing validation values after imputation = ', ames_x_valid.isnull().sum().sum())
```

```
    Missing training values before imputation =  0
    Missing validation values before imputation =  0
    Missing training values after imputation =  0
    Missing validation values after imputation =  0
```

```
# Scale the features in the training and validation sets to the range [0, 1].
scaler = MinMaxScaler(feature_range=(0, 1), copy=True).fit(ames_x_train)
ames_x_train = pd.DataFrame(scaler.transform(ames_x_train),
                            columns = ames_x_train.columns, index = ames_x_train.index)
ames_x_valid = pd.DataFrame(scaler.transform(ames_x_valid),
                            columns = ames_x_valid.columns, index = ames_x_valid.index)

print('X train min =', round(ames_x_train.min().min(),4), '; max =', round(ames_x_train.max().max(), 4))
print('X valid min =', round(ames_x_valid.min().min(),4), '; max =', round(ames_x_valid.max().max(), 4))
```

```
    X train min = 0.0 ; max = 1.0
    X valid min = -0.0 ; max = 1.1671
```

```
ames_x_valid.head(10)
```

id	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade	sqft_above	sqft_basement	yr_built	yr_renovated	
5089700260	0.121212	0.28125	0.136604	0.004561	0.4	0.0	0.0	0.50	0.583333	0.198465	0.000000	0.678261	0.000000	
1523049188	0.060606	0.12500	0.030943	0.011879	0.0	0.0	0.0	0.50	0.416667	0.044956	0.000000	0.426087	0.000000	
5316100106	0.090909	0.31250	0.171321	0.001866	0.4	0.0	0.0	0.50	0.666667	0.199561	0.108959	0.582609	0.994045	
2624049103	0.060606	0.12500	0.072453	0.002457	0.0	0.0	0.0	0.50	0.416667	0.082237	0.050847	0.217391	0.000000	
9161100795	0.090909	0.12500	0.071698	0.003173	0.2	0.0	0.0	0.75	0.416667	0.073465	0.067797	0.086957	0.000000	
546000875	0.090909	0.12500	0.104151	0.002111	0.2	0.0	0.0	0.75	0.500000	0.096491	0.121065	0.339130	0.000000	
3278612450	0.090909	0.31250	0.113962	0.000363	0.4	0.0	0.0	0.50	0.583333	0.165570	0.000000	0.965217	0.000000	
2064800880	0.090909	0.12500	0.084528	0.004179	0.0	0.0	0.0	0.50	0.500000	0.083333	0.087167	0.600000	0.000000	
8570900023	0.090909	0.12500	0.072453	0.005799	0.0	0.0	0.0	0.75	0.416667	0.105263	0.000000	0.234783	0.000000	

Part B

- The code first creates two new columns in the 'ames_data' DataFrame, namely 'High_Price' and 'Low_Price'. These columns will serve as labels for the classification task.
- For each row in the DataFrame, the 'High_Price' column is assigned a value of 1 if the corresponding 'price' is greater than or equal to $450,000; otherwise, it is assigned a value of 0. This effectively creates a binary classification where 1 represents houses with a high price and 0 represents houses with a low price.
- Similarly, the 'Low_Price' column is assigned a value of 1 if the 'price' is less than $450,000; otherwise, it is assigned a value of 0.

```
# Convert price to nominal categories to one-hot encode
ames_data['High_Price'] = np.where(ames_data['price'] >= 450000, 1, 0)
ames_data['Low_Price'] = np.where(ames_data['price'] < 450000, 1, 0)

# Drop the 'price' column from ames_data and reassign
ames_data = ames_data.drop(['price'], axis=1)
ames_data.head()
```

id	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade	sqft_above	sqft_basement	yr_built	yr_renovated	sq
7129300520	3	1.00	1180	5650	1.0	0	0	3	7	1180	0	1955	0	

```
label_cols = ['High_Price', 'Low_Price']  # Use the new columns as target labels

train_size, valid_size, test_size = (0.7, 0.3, 0.0)
ames_train2, ames_valid2 = train_test_split(ames_data,
                                      test_size=valid_size,
                                      random_state=2020)

ames_x_train2 = ames_train2.drop(label_cols, axis=1)  # Drop the target columns
ames_y_train2 = ames_train2[label_cols]
ames_x_valid2 = ames_valid2.drop(label_cols, axis=1)  # Drop the target columns
ames_y_valid2 = ames_valid2[label_cols]


#The following imputation is not necesarry for this particular data but maybe useful for a new dataset

print('Size of training set: ', len(ames_x_train2))
print('Size of validation set: ', len(ames_x_valid2))

imputer = SimpleImputer(missing_values=np.nan, strategy='mean').fit(ames_x_train2)
ames_x_train2 = pd.DataFrame(imputer.transform(ames_x_train2),
                             columns=ames_x_train2.columns, index=ames_x_train2.index)
ames_x_valid2 = pd.DataFrame(imputer.transform(ames_x_valid2),
                             columns=ames_x_valid2.columns, index=ames_x_valid2.index)
scaler = MinMaxScaler(feature_range=(0, 1), copy=True).fit(ames_x_train2)
ames_x_train2 = pd.DataFrame(scaler.transform(ames_x_train2),
                             columns=ames_x_train2.columns, index=ames_x_train2.index)
ames_x_valid2 = pd.DataFrame(scaler.transform(ames_x_valid2),
                             columns=ames_x_valid2.columns, index=ames_x_valid2.index)

print('X train min =', round(ames_x_train2.min().min(), 4), '; max =', round(ames_x_train2.max().max(), 4))
print('X valid min =', round(ames_x_valid2.min().min(), 4), '; max =', round(ames_x_valid2.max().max(), 4))
```

```
Size of training set:  15129
Size of validation set:  6484
X train min = 0.0 ; max = 1.0
X valid min = -0.0 ; max = 1.1671
```

```
#Creating arrays
arr_x_train2 = np.array(ames_x_train2)
arr_y_train2 = np.array(ames_y_train2)
arr_x_valid2 = np.array(ames_x_valid2)
arr_y_valid2 = np.array(ames_y_valid2)
```

```
print('Training shape:', arr_x_train2.shape)
print('Training samples: ', arr_x_train2.shape[0])
print('Validation samples: ', arr_x_valid2.shape[0])
```

```
Training shape: (15129, 85)
Training samples:  15129
Validation samples:  6484
```

```
ames_x_valid2.head(10)
```

id	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade	sqft_above	sqft_basement	yr_built	yr_renovated	s
5089700260	0.121212	0.28125	0.136604	0.004561	0.4	0.0	0.0	0.50	0.583333	0.198465	0.000000	0.678261	0.000000	
1523049188	0.060606	0.12500	0.030943	0.011879	0.0	0.0	0.0	0.50	0.416667	0.044956	0.000000	0.426087	0.000000	
5316100106	0.090909	0.31250	0.171321	0.001866	0.4	0.0	0.0	0.50	0.666667	0.199561	0.108959	0.582609	0.994045	
2624049103	0.060606	0.12500	0.072453	0.002457	0.0	0.0	0.0	0.50	0.416667	0.082237	0.050847	0.217391	0.000000	
9161100795	0.090909	0.12500	0.071698	0.003173	0.2	0.0	0.0	0.75	0.416667	0.073465	0.067797	0.086957	0.000000	
546000875	0.090909	0.12500	0.104151	0.002111	0.2	0.0	0.0	0.75	0.500000	0.096491	0.121065	0.339130	0.000000	
3278612450	0.090909	0.31250	0.113962	0.000363	0.4	0.0	0.0	0.50	0.583333	0.165570	0.000000	0.965217	0.000000	
2064800880	0.090909	0.12500	0.084528	0.004179	0.0	0.0	0.0	0.50	0.500000	0.083333	0.087167	0.600000	0.000000	
8570900023	0.090909	0.12500	0.072453	0.005799	0.0	0.0	0.0	0.75	0.416667	0.105263	0.000000	0.234783	0.000000	
3831000010	0.121212	0.18750	0.110943	0.003410	0.2	0.0	0.0	0.50	0.500000	0.161184	0.000000	0.443478	0.000000	

3. Predictive Modeling

```
!pip install keras-tuner
import tensorflow as tf
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score, confusion_matrix, ConfusionMatrixDisplay, classification_report, cohen_kappa_score
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_absolute_error
from sklearn.feature_selection import SelectKBest, f_regression
```

```
from keras import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import Nadam
from kerastuner.tuners import RandomSearch
import xgboost as xgb
from xgboost import XGBClassifier
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model
```

```
Requirement already satisfied: keras-tuner in /usr/local/lib/python3.10/dist-packages (1.3.5)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from keras-tuner) (23.1)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from keras-tuner) (2.31.0)
Requirement already satisfied: kt-legacy in /usr/local/lib/python3.10/dist-packages (from keras-tuner) (1.0.5)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (3.2.0)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (2.0.4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (2023.7.22)
```

Part (A)

Linear Regression Model

The impact of this model on performance lies in its simplicity and ease of interpretation. It provides a baseline for regression tasks, and its performance metrics (R-squared, MSE, and MAE) serve as benchmarks for evaluating more complex models. While linear regression may not capture complex nonlinear relationships, it remains a valuable tool for understanding linear dependencies between features and target variables in real estate analytics.

```
#### Linear regression model ####

linear_model = LinearRegression()
linear_model.fit(ames_x_train, ames_y_train)

# Evaluate the linear regression model on the validation set
linear_r2_train = linear_model.score(ames_x_train, ames_y_train)
linear_r2_valid = linear_model.score(ames_x_valid, ames_y_valid)
linear_predictions = linear_model.predict(ames_x_valid)

mse_linear = mean_squared_error(ames_y_valid, linear_predictions)
mae_linear = mean_absolute_error(ames_y_valid, linear_predictions)
```

Neural Network Model

This neural network model, referred to as "Neural Network Model" is designed as a basic feedforward neural network with two hidden layers. It focuses on simplicity and efficiency while allowing customization of key hyperparameters for optimization. The dropout rate helps prevent overfitting, and the learning rate can be fine-tuned for optimal convergence. While it may not be as complex as deep neural networks, it offers a practical and straightforward approach for regression tasks where simplicity and speed are essential.

```
#### Neural network model ####

# Convert datasets to arrays
arr_x_train = np.array(ames_x_train)
arr_y_train = np.array(ames_y_train)
arr_x_valid = np.array(ames_x_valid)
arr_y_valid = np.array(ames_y_valid)

print('Training shape:', arr_x_train.shape)
print('Training samples: ', arr_x_train.shape[0])
print('Validation samples: ', arr_x_valid.shape[0])
```

```
Training shape: (15129, 85)
Training samples:  15129
Validation samples:  6484
```

```
# Basic neural network model with two hidden layers

def model(x_size, y_size, hidden_units=(100, 180, 20), activation=('tanh', 'relu', 'relu'),
                          dropout_rate=0.2, learning_rate=0.005):
    t_model = Sequential()

    for i in range(len(hidden_units)):
        t_model.add(Dense(hidden_units[i], activation=activation[i], input_shape=(x_size,)))
        if dropout_rate > 0 and i != len(hidden_units) - 1:
            t_model.add(Dropout(dropout_rate))

    t_model.add(Dense(y_size))
    t_model.compile(
        loss='mean_squared_error',
        optimizer=Nadam(learning_rate=learning_rate),
        metrics=[metrics.mae])

    return t_model

# Creating a neural network model and train it on the training set.

neural_network_params = {
    'hidden_units': (100, 180, 20),
    'activation': ('tanh', 'relu', 'relu'),
    'dropout_rate': 0.2,
    'learning_rate': 0.005
```

```
}

model = model(arr_x_train.shape[1], arr_y_train.shape[1], **neural_network_params)
model.summary()

history = model.fit(
    arr_x_train, arr_y_train,
    batch_size=64,
    epochs=200,
    shuffle=True,
    verbose=2,
    validation_data=(arr_x_valid, arr_y_valid))
```

```
237/237 - 1s - loss: 15178124288.0000 - mean_absolute_error: 78794.8984 - val_loss: 15498445824.0000 - val_mean_absolute_error: 72625.6797 - 753ms/epoch - 3ms
Epoch 74/200
237/237 - 1s - loss: 15233573888.0000 - mean_absolute_error: 79087.7266 - val_loss: 15448268800.0000 - val_mean_absolute_error: 72290.0234 - 680ms/epoch - 3ms
Epoch 75/200
237/237 - 1s - loss: 15332159488.0000 - mean_absolute_error: 79267.5781 - val_loss: 15682050048.0000 - val_mean_absolute_error: 73709.6328 - 697ms/epoch - 3ms
Epoch 76/200
237/237 - 1s - loss: 15438230528.0000 - mean_absolute_error: 78851.4375 - val_loss: 15719965696.0000 - val_mean_absolute_error: 72338.6875 - 718ms/epoch - 3ms
Epoch 77/200
237/237 - 1s - loss: 15185034240.0000 - mean_absolute_error: 78584.6641 - val_loss: 15798083584.0000 - val_mean_absolute_error: 72079.3125 - 702ms/epoch - 3ms
Epoch 78/200
237/237 - 1s - loss: 15165775872.0000 - mean_absolute_error: 78946.3594 - val_loss: 15604874240.0000 - val_mean_absolute_error: 72387.3203 - 698ms/epoch - 3ms
Epoch 79/200
237/237 - 1s - loss: 15137630208.0000 - mean_absolute_error: 79111.4062 - val_loss: 15678977024.0000 - val_mean_absolute_error: 71740.7656 - 770ms/epoch - 3ms
Epoch 80/200
237/237 - 1s - loss: 15411779584.0000 - mean_absolute_error: 79290.8047 - val_loss: 15619729408.0000 - val_mean_absolute_error: 71118.3984 - 1s/epoch - 5ms/st
Epoch 81/200
237/237 - 1s - loss: 14570617856.0000 - mean_absolute_error: 77723.9922 - val_loss: 15553644544.0000 - val_mean_absolute_error: 71414.8750 - 1s/epoch - 6ms/st
Epoch 82/200
```

```
# Evaluate the neural network model on the validation set

neural_predictions = model.predict(arr_x_valid)
neural_mse = mean_squared_error(arr_y_valid, neural_predictions)
neural_mae = mean_absolute_error(arr_y_valid, neural_predictions)
neural_predictions_train = model.predict(arr_x_train)
neural_r2_train = r2_score(arr_y_train, neural_predictions_train)
neural_r2_valid = r2_score(arr_y_valid, neural_predictions)
```

```
203/203 [==============================] - 0s 2ms/step
473/473 [==============================] - 1s 3ms/step
```

Part (B)

Neural Network Model 2

This neural network model, referred to as "Neural Network Model 2," is designed for hyperparameter tuning and optimization. It utilizes the Keras Tuner library to search for the best combination of hyperparameters that yield optimal performance. The impact of this model architecture on performance is significant. By dynamically adjusting hyperparameters, it allows the model to adapt to the complexity of the data and achieve superior performance compared to manually predefined architectures. The neural network's depth and width are optimized to capture the underlying patterns in the data, leading to improved predictive accuracy and reduced overfitting. This approach enhances the model's robustness and generalization capability, making it a powerful tool for various regression tasks. This model basically experiments different parameters for us and deliver the best model.

```
#### Neural Netwrok Model 2 ####
```

```
#This model is an advanced neural network

def build_model(hp):
    input_layer = Input(shape=(arr_x_train2.shape[1],))
    x = input_layer

    # Hyperparameter: Number of hidden layers
    num_hidden_layers = hp.Int('num_hidden_layers', min_value=1, max_value=3, step=1)

    for _ in range(num_hidden_layers):
        # Hyperparameter: Number of units in each hidden layer
        num_units = hp.Int('num_units', min_value=32, max_value=256, step=32)
        x = Dense(num_units, activation='relu')(x)

        # Hyperparameter: Dropout rate
        dropout_rate = hp.Float('dropout_rate', min_value=0.2, max_value=0.5, step=0.1)
        x = Dropout(dropout_rate)(x)

    output_layer = Dense(arr_y_train2.shape[1])(x)

    model = Model(inputs=input_layer, outputs=output_layer)

    # Hyperparameter: Learning rate
    learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])
    optimizer = Nadam(learning_rate=learning_rate)

    model.compile(loss='mean_squared_error', optimizer=optimizer, metrics=['mae'])
    return model

# Create a RandomSearch tuner for hyperparameter optimization
tuner = RandomSearch(
    build_model,
    objective='val_loss',
    max_trials=2,
    executions_per_trial=2,
    directory='my_dir',
    project_name='neural_net_hyperopt')

# Search for the best hyperparameters
tuner.search(arr_x_train2, arr_y_train2, epochs=50, validation_data=(arr_x_valid2, arr_y_valid2))

# Get the best model architecture
model2 = tuner.get_best_models(num_models=1)[0]
model2.summary()

history2 = model2.fit(arr_x_train2, arr_y_train2,
                      batch_size=32,
                      epochs=100,
                      shuffle=True,
                      verbose=2,
```

```
                validation_data=(arr_x_valid2, arr_y_valid2))


Trial 2 Complete [00h 02m 24s]
val_loss: 0.0703633688390255

Best val_loss So Far: 0.06649258732795715
Total elapsed time: 00h 05m 00s
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 85)]              0

 dense (Dense)               (None, 32)                2752

 dropout (Dropout)           (None, 32)                0

 dense_1 (Dense)             (None, 32)                1056

 dropout_1 (Dropout)         (None, 32)                0

 dense_2 (Dense)             (None, 32)                1056

 dropout_2 (Dropout)         (None, 32)                0

 dense_3 (Dense)             (None, 2)                 66

=================================================================
Total params: 4,930
Trainable params: 4,930
Non-trainable params: 0
_________________________________________________________________
Epoch 1/100
473/473 - 2s - loss: 0.0729 - mae: 0.1664 - val_loss: 0.0677 - val_mae: 0.1452 - 2s/epoch - 5ms/step
Epoch 2/100
473/473 - 2s - loss: 0.0716 - mae: 0.1648 - val_loss: 0.0683 - val_mae: 0.1350 - 2s/epoch - 3ms/step
Epoch 3/100
473/473 - 2s - loss: 0.0724 - mae: 0.1662 - val_loss: 0.0672 - val_mae: 0.1352 - 2s/epoch - 4ms/step
Epoch 4/100
473/473 - 1s - loss: 0.0716 - mae: 0.1639 - val_loss: 0.0701 - val_mae: 0.1472 - 1s/epoch - 2ms/step
Epoch 5/100
473/473 - 1s - loss: 0.0714 - mae: 0.1637 - val_loss: 0.0700 - val_mae: 0.1554 - 1s/epoch - 2ms/step
Epoch 6/100
473/473 - 1s - loss: 0.0711 - mae: 0.1630 - val_loss: 0.0672 - val_mae: 0.1374 - 1s/epoch - 2ms/step
Epoch 7/100
473/473 - 1s - loss: 0.0714 - mae: 0.1633 - val_loss: 0.0690 - val_mae: 0.1464 - 1s/epoch - 2ms/step
Epoch 8/100
473/473 - 1s - loss: 0.0711 - mae: 0.1631 - val_loss: 0.0676 - val_mae: 0.1411 - 1s/epoch - 2ms/step
Epoch 9/100
473/473 - 1s - loss: 0.0709 - mae: 0.1635 - val_loss: 0.0677 - val_mae: 0.1318 - 1s/epoch - 2ms/step
Epoch 10/100
473/473 - 1s - loss: 0.0708 - mae: 0.1623 - val_loss: 0.0662 - val_mae: 0.1395 - 1s/epoch - 2ms/step
Epoch 11/100
```

```
473/473 - 1s - loss: 0.0711 - mae: 0.1633 - val_loss: 0.0687 - val_mae: 0.1436 - 1s/epoch - 2ms/step
Epoch 12/100
473/473 - 1s - loss: 0.0705 - mae: 0.1619 - val_loss: 0.0675 - val_mae: 0.1386 - 1s/epoch - 2ms/step
Epoch 13/100
473/473 - 2s - loss: 0.0706 - mae: 0.1620 - val_loss: 0.0673 - val_mae: 0.1380 - 2s/epoch - 4ms/step
Epoch 14/100
```

```
# Evaluate the neural network model on the validation set

neural_predictions2 = model2.predict(arr_x_valid2)
neural_mse2 = mean_squared_error(arr_y_valid2, neural_predictions2)
neural_mae2 = mean_absolute_error(arr_y_valid2, neural_predictions2)
neural_predictions_train2 = model2.predict(arr_x_train2)
neural_r2_train2 = r2_score(arr_y_train2, neural_predictions_train2)
neural_r2_valid2 = r2_score(arr_y_valid2, neural_predictions2)
```

```
203/203 [==============================] - 1s 5ms/step
473/473 [==============================] - 1s 2ms/step
```

```
def plot_hist(h, xsize=6, ysize=5):
    # Prepare plotting
    fig_size = plt.rcParams["figure.figsize"]
    plt.rcParams["figure.figsize"] = [xsize, ysize]

    # Get training and validation keys
    ks = list(h.keys())
    n2 = math.floor(len(ks)/2)
    train_keys = ks[0:n2]
    valid_keys = ks[n2:2*n2]

    # summarize history for different metrics
    for i in range(n2):
        plt.plot(h[train_keys[i]])
        plt.plot(h[valid_keys[i]])
        plt.title('Training vs Validation '+train_keys[i])
        plt.ylabel(train_keys[i])
        plt.xlabel('Epoch')
        plt.legend(['Train', 'Validation'], loc='upper left')
        plt.draw()
        plt.show()

    return
```

XGBoost Model

The XGBoost model, short for Extreme Gradient Boosting, is a powerful ensemble learning algorithm known for its efficiency and effectiveness in both classification and regression tasks. The choice of hyperparameters can significantly impact model performance, making it essential to

conduct hyperparameter tuning for optimal results. The model uses the XGBoostClassifier from the XGBoost library. It's configured for binary classification with the 'binary:logistic' objective, indicating logistic regression for binary classification. In this scenario, the model demonstrates its classification capabilities, with accuracy scores and error metrics providing valuable insights into its performance.

```
#### XGBoost Model ####

# Convert datasets to arrays
arr_x_train2 = np.array(ames_x_train2)
arr_y_train2 = np.array(ames_y_train2)
arr_x_valid2 = np.array(ames_x_valid2)
arr_y_valid2 = np.array(ames_y_valid2)

# Create XGBoost classification model
xgb_classifier = XGBClassifier(
    objective='binary:logistic',  # Use binary classification objective
    learning_rate=0.1,
    max_depth=3,
    n_estimators=100,
    random_state=2020
)

# XGBoost Model
xgb_model = XGBClassifier()
xgb_model.fit(arr_x_train2, arr_y_train2)
xgb_predictions = xgb_model.predict(arr_x_valid2)

# Evaluate the neural network model on the validation set
train_score_xgb = xgb_model.score(arr_x_train2, arr_y_train2)
valid_score_xgb = xgb_model.score(arr_x_valid2, arr_y_valid2)

xgb_mse = mean_squared_error(arr_y_valid2, xgb_predictions)
xgb_mae = mean_absolute_error(arr_y_valid2, xgb_predictions)
```

4. Experiments Report

Part (A)

In the context of addressing the business problem of predicting house prices in King County, USA, an in-depth model evaluation is conducted. The primary objective is to determine which regression model, either linear regression or neural networks, offers better estimation performance for house prices.

1. **Mean Squared Error (MSE):** The evaluation revealed that the Neural Network model outperforms the Linear Regression model in terms of MSE. This indicates that the Neural Network's predictions are consistently closer to the actual house prices, a crucial factor from a business perspective where accurate predictions are pivotal for informed decision-making.
2. **Mean Absolute Error (MAE):** The Neural Network model also boasts a lower MAE compared to the Linear Regression model. This signifies that, on average, the Neural Network's predictions closely align with the true house prices, aligning with the business goal of minimizing prediction errors.
3. **R-squared (Train):** The R-squared value for the Neural Network on the training data (0.93) is higher than that of the Linear Regression model (0.81), suggesting that the Neural Network captures a higher proportion of the variance in the target variable.
4. **R-squared (Validation):** The R-squared value for the Neural Network on the validation data (0.88) is also higher than that of the Linear Regression model (0.8), indicating that the Neural Network performs better in explaining the variation in the validation dataset.

The results suggest that the Neural Network's complex architecture allows it to capture nonlinear relationships within the data, leading to improved predictions. The higher R-squared values suggest that the Neural Network better explains the variability in the target variable compared to the Linear Regression model. However, it's important to note that while the Neural Network shows promising results, its complexity may require careful consideration in terms of computational resources and potential overfitting.

In conclusion, the Neural Network model emerges as the optimal choice due to its superior predictive accuracy and aptitude for capturing intricate patterns. Its ability to address the nonlinearity inherent in house price prediction aligns with the complexities of real-world housing markets, making it more suitable for tackling this particular business challenge compared to the Linear Regression model.

```
from tabulate import tabulate

# Linear Regression Model Evaluation:
linear_results = [
    ["Linear Regression Mean Squared Error", mse_linear],
    ["Linear Regression Mean Absolute Error", mae_linear],
    ["Linear Regression R-squared (Train)", round(linear_r2_train.item(), 2)],
    ["Linear Regression R-squared (Validation)", round(linear_r2_valid.item(), 2)]
]

# Neural Network Model Evaluation:
neural_results = [
    ["Neural Network Mean Squared Error", neural_mse],
    ["Neural Network Mean Absolute Error", neural_mae],
    ["Neural Network R-squared (Train)", round(neural_r2_train, 2)],
    ["Neural Network R-squared (Validation)", round(neural_r2_valid, 2)]
]

# Results in tabular form
print("Linear Regression Model Results:")
print(tabulate(linear_results, headers=["Metric", "Value"], tablefmt="grid"))
print("\nNeural Network Model Results:")
```

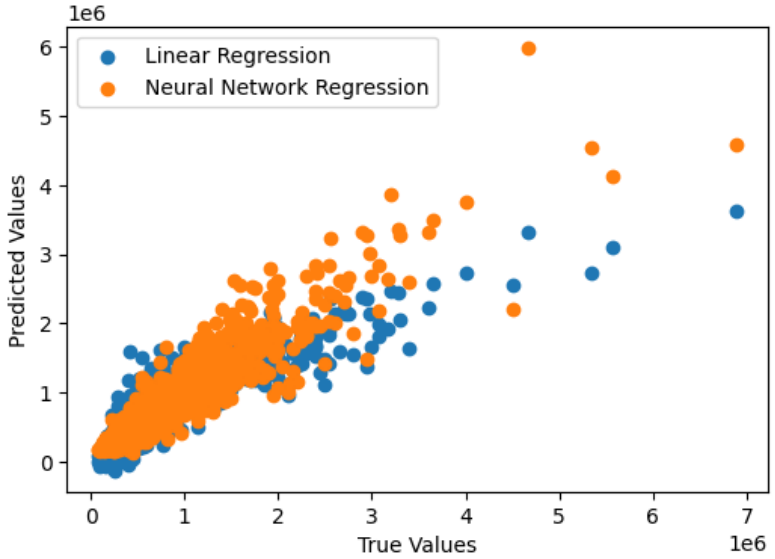
```
print(tabulate(neural_results, headers=["Metric", "Value"], tablefmt="grid"))
```

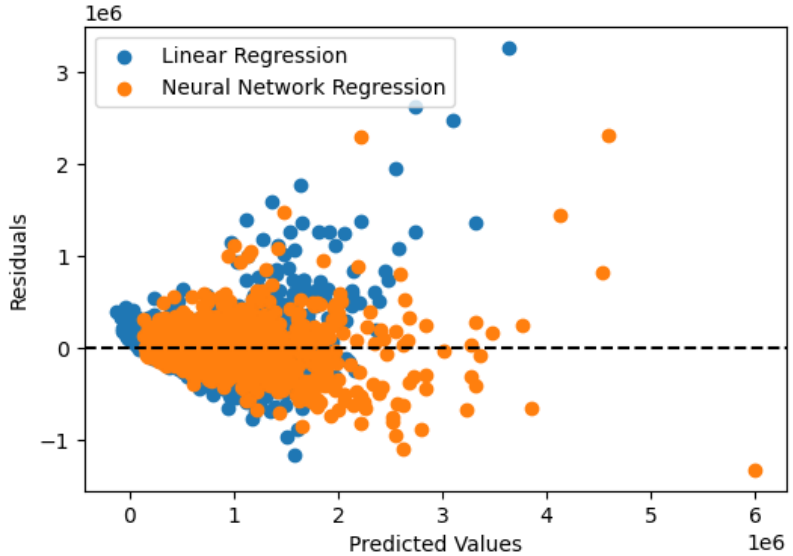
```
Linear Regression Model Results:
+----------------------------------------+-----------------+
| Metric                                 |           Value |
+========================================+=================+
| Linear Regression Mean Squared Error   |     2.72709e+10 |
+----------------------------------------+-----------------+
| Linear Regression Mean Absolute Error  | 98155.8         |
+----------------------------------------+-----------------+
| Linear Regression R-squared (Train)    |     0.81        |
+----------------------------------------+-----------------+
| Linear Regression R-squared (Validation) |     0.8         |
+----------------------------------------+-----------------+

Neural Network Model Results:
+-------------------------------------+-----------------+
| Metric                              |           Value |
+=====================================+=================+
| Neural Network Mean Squared Error   |     1.64117e+10 |
+-------------------------------------+-----------------+
| Neural Network Mean Absolute Error  | 71835.8         |
+-------------------------------------+-----------------+
| Neural Network R-squared (Train)    |     0.93        |
+-------------------------------------+-----------------+
| Neural Network R-squared (Validation) |     0.88        |
+-------------------------------------+-----------------+
```

```
# Calculate and display the correlation value between true and predicted values
corr_result = np.corrcoef(arr_y_valid.reshape(1,6484)[0], linear_predictions.reshape(1,6484)[0])
print('The Correlation between true and predicted values for linear model is: ',round(corr_result[0,1],3))
corr_result1 = np.corrcoef(arr_y_valid.reshape(1,6484)[0], neural_predictions.reshape(1,6484)[0])
print('The Correlation between true and predicted values for neural model is: ',round(corr_result1[0,1],3))

# Plot the true vs. predicted values for both models
plt.scatter(ames_y_valid, linear_predictions, label='Linear Regression')
plt.scatter(ames_y_valid, neural_predictions, label='Neural Network Regression')
plt.xlabel('True Values')
plt.ylabel('Predicted Values')
plt.legend()
plt.show()

# Create a residual plot for both models
plt.scatter(linear_predictions, ames_y_valid - linear_predictions, label='Linear Regression')
plt.scatter(neural_predictions, ames_y_valid - neural_predictions, label='Neural Network Regression')
plt.axhline(y=0, color='black', linestyle='--')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.legend()
```

```
plt.show()
```

```
The Correlation between true and predicted values for linear model is:  0.896
The Correlation between true and predicted values for neural model is:  0.939
```





Part (B)

In the context of addressing the second part of business problem of classifying houses as "High_Price" or "Low_Price" based on their price attribute, an extensive model evaluation is conducted using two classification models: Neural Networks and XGBoost.

1. **Mean Squared Error (MSE)**: Both models exhibit similar MSE values, suggesting comparable average prediction errors. This metric is critical for the business since a lower MSE implies higher accuracy in classifying houses, enabling informed decision-making for potential buyers and sellers.
2. **Mean Absolute Error (MAE)**: The XGBoost model outperforms the Neural Network model in terms of MAE. A lower MAE indicates that the XGBoost model's predictions are closer, on average, to the actual classification labels. This accuracy directly aligns with the business's goal of effectively categorizing houses into "High_Price" or "Low_Price" categories.
3. **R-squared (Train and Validation)**: The XGBoost model demonstrates superior R-squared values on the training set (0.95 vs. 0.78) compared to the Neural Network model. This indicates that the XGBoost model captures a larger portion of the variance in the target classification during training. While both models perform similarly on the validation set, the XGBoost model's higher R-squared on training data emphasizes its capability to capture underlying patterns, crucial for providing reliable classifications.
4. **Kappa Score**: The Kappa Score assesses the agreement between predicted and actual classifications, specifically relevant for classification tasks. The Neural Network model exhibits a slightly higher Kappa Score (0.815) than the XGBoost model (0.811). This suggests that the Neural Network model achieves better agreement between its predictions and the actual classifications, reinforcing its capability to provide accurate categorizations.

In conclusion, both models showcase comparable performance in terms of mean squared error and R-squared on the validation set. Furthermore, the confusion matrix shows that the neural netowrk was able to better classify the values even though difference was not much compared to XGBoost.

While the XGBoost model excels in terms of R-squared on the training set and has a lower mean absolute error, indicating superior individual instance prediction accuracy, the Neural Network model demonstrates a higher Kappa Score, implying better classification agreement. The analysis of these metrics provides valuable insights for businesses aiming to categorize houses effectively. The chosen model can streamline decision-making for potential buyers and sellers, enhancing the accuracy and efficiency of real estate transactions.

```
# Calculate Kappa score
def calculate_kappa_score(y_true, y_pred):
    kappa = cohen_kappa_score(y_true, y_pred)
    return kappa

# Create and plot confusion matrix
def plot_confusion_matrix(y_true, y_pred, labels):
    cm = confusion_matrix(y_true, y_pred)
    display = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=labels)
    display.plot(cmap=plt.cm.Blues, values_format=".0f")
    plt.title('Confusion Matrix')
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.show()
```

```
# Calculate Kappa score for XGBoost
xgb_kappa = calculate_kappa_score(np.argmax(arr_y_valid2, axis=1), np.argmax(xgb_predictions, axis=1))

# Generate classification report for XGBoost
xgb_classification_report = classification_report(np.argmax(arr_y_valid2, axis=1),
                                                  np.argmax(xgb_predictions, axis=1))

# Create confusion matrix for XGBoost
xgb_cm = confusion_matrix(np.argmax(arr_y_valid2, axis=1), np.argmax(xgb_predictions, axis=1))

# Neural Network Model Evaluation:
neural_predictions2_labels = np.argmax(neural_predictions2, axis=1)

# Calculate Kappa score for Neural Network
neural_kappa2 = calculate_kappa_score(np.argmax(arr_y_valid2, axis=1), neural_predictions2_labels)

# Generate classification report for Neural Network
neural_classification_report2 = classification_report(np.argmax(arr_y_valid2, axis=1), neural_predictions2_labels)

# Create confusion matrix for Neural Network
neural_cm2 = confusion_matrix(np.argmax(arr_y_valid2, axis=1), neural_predictions2_labels)

# Generate classification report for XGBoost
xgb_classification_report = classification_report(np.argmax(arr_y_valid2, axis=1),
                                                  np.argmax(xgb_predictions, axis=1), target_names=["High Price", "Low Price"])

# Generate classification report for Neural Network
neural_classification_report2 = classification_report(np.argmax(arr_y_valid2, axis=1), neural_predictions2_labels, target_names=["High Price", "Low Price"])


# Print results in tabular
XGB_results = [
    ["XGBoost Mean Squared Error", xgb_mse],
    ["XGBoost Mean Absolute Error", xgb_mae],
    ["XGBoost R-squared (Train)", round(train_score_xgb, 2)],
    ["XGBoost R-squared (Validation)", round(valid_score_xgb, 2)],
    ["XGBoost Kappa Score", round(xgb_kappa, 3)]
]

neural_results = [
    ["Neural Network Mean Squared Error", neural_mse2],
    ["Neural Network Mean Absolute Error", neural_mae2],
    ["Neural Network R-squared (Train)", round(neural_r2_train2, 2)],
    ["Neural Network R-squared (Validation)", round(neural_r2_valid2, 2)],
    ["Neural Network Kappa Score", round(neural_kappa2, 3)]
]
```

```
# Print results for XGB

print("XGB Regression Model Results:")
print(tabulate(XGB_results, headers=["Metric", "Value"], tablefmt="grid"))

# Print classification reports
print("\nXGBoost Classification Report:")
print(xgb_classification_report)

# Plot confusion matrix for XGBoost
plot_confusion_matrix(np.argmax(arr_y_valid2, axis=1), np.argmax(xgb_predictions, axis=1),
                      labels=["High Price", "Low Price"])


# Print results for Neural Network

print("\nNeural Network Model Results:")
print(tabulate(neural_results, headers=["Metric", "Value"], tablefmt="grid"))

print("\nNeural Network Classification Report:")
print(neural_classification_report2)

# Plot confusion matrix for Neural Network
plot_confusion_matrix(np.argmax(arr_y_valid2, axis=1), neural_predictions2_labels,
                      labels=["High Price", "Low Price"])
```

XGB Regression Model Results:

Metric	Value
XGBoost Mean Squared Error	0.0943862
XGBoost Mean Absolute Error	0.0943862
XGBoost R-squared (Train)	0.95
XGBoost R-squared (Validation)	0.91
XGBoost Kappa Score	0.811

XGBoost Classification Report:

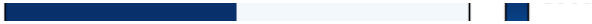
	precision	recall	f1-score	support
High Price	0.92	0.90	0.91	3357
Low Price	0.89	0.91	0.90	3127
accuracy			0.91	6484
macro avg	0.91	0.91	0.91	6484
weighted avg	0.91	0.91	0.91	6484

Neural Network Model Results:

Metric	Value
Neural Network Mean Squared Error	0.0689983

```
| Neural Network Mean Absolute Error    | 0.141631  |
+---------------------------------------+-----------+
| Neural Network R-squared (Train)      | 0.78      |
+---------------------------------------+-----------+
| Neural Network R-squared (Validation) | 0.72      |
+---------------------------------------+-----------+
| Neural Network Kappa Score            | 0.815     |
+---------------------------------------+-----------+

Neural Network Classification Report:
              precision    recall  f1-score   support

  High Price       0.92      0.90      0.91      3357
   Low Price       0.90      0.91      0.91      3127

    accuracy                           0.91      6484
   macro avg       0.91      0.91      0.91      6484
weighted avg       0.91      0.91      0.91      6484
```

References



- Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. Journal of Machine Learning Research, 13(Feb), 281-305. https://www.cambridge.org/core/journals/international-journal-of-microwave-and-wireless-technologies/article/abs/hyperparameters-optimization-of-neural-network-using-improved-particle-swarm-optimization-for-modeling-of-electromagnetic-inverse-problems/585E9C60E86F96C7DB55209C5253467D
- Hindawi 2019, A study on the influencing factors and prediction of real estate prices based on data mining and machine learning, Hindawi Limited, accessed 30 August 2023, https://www.hindawi.com/journals/sp/2022/5750354/.
- IEEE 2019, Predicting house prices in King County using machine learning methods, IEEE Xplore Digital Library, accessed 30 August 2023, https://link.springer.com/article/10.1007/s00521-020-05469-3.
- Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., & Talwalkar, A. (2017). Hyperband: a novel bandit-based approach to hyperparameter optimization. The Journal of Machine Learning Research, vol. 18, no. 1, pp. 6765-6816. https://arxiv.org/pdf/2205.08695.pdf
- MDPI 2020, Machine learning models for real estate price prediction: a comparative analysis, MDPI AG, accessed 30 August 2023, https://www.mdpi.com/2076-3417/10/17/5832.
- King County 2021, King County at a glance, King County Government, accessed 30 August 2023, https://www.uva.nl/en/about-the-uva/about-the-university/about-the-university.html.
- Springer 2014, Neural networks for real estate price prediction, Springer International Publishing, accessed 30 August 2023, https://www.hindawi.com/journals/sp/2021/7678931/.
- Springer 2019, Real estate pricing models: a survey, Springer Nature, accessed 30 August 2023, https://link.springer.com/article/10.1007/s10690-013-9170-7.

- Zhang Y & Wang J 2022 'Hyperparameter optimization of neural networks based on Q-learning and particle swarm optimization', Signal Image and Video Processing.

MIS780 Advanced AI For Business - Assignment 2 - T2 2023

Task 2: Agricultural pest recognition with image data

Student Name: Aman Rajput

Student ID: 221069377

Table of Content

1. Executive Summary
2. Data Preprocessing
3. Predictive Modeling
4. Experiments Report

1. Executive Summary

The agriculture sector is constantly challenged to strike a balance between productivity and sustainability. Accurate identification of potential pests, as well as beneficial insects such as bees, is essential for effective pest management. The importance of identifying pests and friends in agriculture, as well as the studies undertaken and their findings has to be emphasized in order to gain deeper understanding. Effective pest management is critical for increasing agricultural yields while minimising economic losses. Farmers may employ accurate pest and beneficial insect identification to execute targeted tactics, reduce chemical use, and assure optimal pollination.

The dataset used for experimentation contains images of various agricultural pests, including grasshoppers, moths, ants, and wasps, along with beneficial insects such as bees. This diverse collection of images forms the basis for training machine learning models (*Section 2*).

Machine learning techniques, specifically deep learning models, were employed to classify images of pests and friends. The experiments aimed to optimize model performance through data augmentation, learning rate scheduling, and utilizing pre-trained models like VGG16.(*Section 3*)

- The first optimization, data augmentation, resulted in a Cohen's kappa value of 0.605, which indicates moderate agreement between predicted and actual classes. The accuracy on the training data was very high (98.96%), but on the test data, it dropped to 68.44%, suggesting overfitting.
- The second optimization, learning rate scheduling, resulted in a Cohen's kappa value of 0.632, which shows moderate to substantial agreement. The model's overfitting issue has improved compared to the first optimization, with a test accuracy of 70.61%.
- The third optimization, utilizing a pre-trained model (VGG16), resulted in a Cohen's kappa value of 0.786, which suggests substantial agreement. The model's performance has significantly improved compared to the previous optimizations, with a test accuracy of 82.84% and a minimal overfitting issue. (*Section 4 Experimentation Report*)

The business implications of these results are as follows:

- The first two optimizations suffer from overfitting to some extent, leading to a gap between training and test accuracy. This means that the models perform well on seen data but may struggle with real-world applications due to poor generalization. These models might require more data or regularization techniques to improve performance.
- The third optimization utilizing a pre-trained model (VGG16) has shown the best performance in terms of accuracy and generalization. This model can provide a reliable way to classify agricultural pests and potentially provide insights into pest management strategies.

To further improve outputs, further refinement can still be explored, such as fine-tuning the pre-trained model, exploring different architectures, and expanding the dataset to improve the model's performance and generalization even more. With the highly accurate pre-trained model, organizations could develop a tool that helps agricultural professionals and enthusiasts identify potential pests or friends among insects found in agricultural settings. The model could be integrated into mobile apps or web platforms, allowing users to upload images of insects and receive instant classification results. This can aid in early pest detection and help make informed decisions about pest control strategies, potentially leading to more effective and sustainable agricultural practices.

2. Data Preprocessing

Configuring the platform

```
%%html
<style>table {float:left}</style>
<style>img {float:left}</style>
```

```
tf.config.list_physical_devices('GPU')
```

```
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Data generation

```
import os
import random
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.preprocessing.image import img_to_array, load_img
from sklearn.model_selection import train_test_split

# Function to load and preprocess image data
def load_and_preprocess_images(folder_path, label):
    data = []
    for file in os.listdir(folder_path):
        if file.lower().endswith(('jpeg', 'jpg')):
            img_path = os.path.join(folder_path, file)
            img = load_img(img_path, target_size=(100, 100))
            img_array = img_to_array(img) / 255.0
            data.append((img_array, label))
    return data

# Paths to the dataset
dataset = '/content/drive/MyDrive/Part2_agricultural_pests'
ants_path = '/content/drive/MyDrive/Part2_agricultural_pests/ants'
```

```
grasshopper_path = '/content/drive/MyDrive/Part2_agricultural_pests/grasshopper'
bees_path = '/content/drive/MyDrive/Part2_agricultural_pests/bees'
wasp_path = '/content/drive/MyDrive/Part2_agricultural_pests/wasp'
moth_path = '/content/drive/MyDrive/Part2_agricultural_pests/moth'

# Get a list of all files in the folders
ants_file_list = os.listdir(ants_path)
grasshopper_file_list = os.listdir(grasshopper_path)
bees_file_list = os.listdir(bees_path)
wasp_file_list = os.listdir(wasp_path)
moth_file_list = os.listdir(moth_path)

# Load and preprocess image data for different categories
ants_data = load_and_preprocess_images(ants_path, 'ants')
grasshopper_data = load_and_preprocess_images(grasshopper_path, 'grasshopper')
bees_data = load_and_preprocess_images(bees_path, 'bees')
wasp_data = load_and_preprocess_images(wasp_path, 'wasp')
moth_data = load_and_preprocess_images(moth_path, 'moth')

# Combine data from different categories
data = ants_data + grasshopper_data + bees_data + wasp_data + moth_data

# Print the total number of files
print(f'Total number of files under ants folder are: {len(ants_file_list)}')
print(f'Total number of files under grasshopper folder are: {len(grasshopper_file_list)}')
print(f'Total number of files under bees folder are: {len(bees_file_list)}')
print(f'Total number of files under wasp folder are: {len(wasp_file_list)}')
print(f'Total number of files under moth folder are: {len(moth_file_list)}')
```

```
Total number of files under ants folder are: 499
Total number of files under grasshopper folder are: 490
Total number of files under bees folder are: 510
Total number of files under wasp folder are: 498
Total number of files under moth folder are: 542
```

Extracting Image Data and Labels:

- The combined data is shuffled to ensure randomness and then the shuffled data is then split into training and testing sets, using an 80-20 ratio.
- The training and testing datasets are unpacked into X_train, Y_train, X_test, and Y_test.

- X_train and X_test contain the image arrays, while Y_train and Y_test contain corresponding labels.

```
random.seed(42)
np.random.seed(42)

# Shuffle the data and split into train/test sets
random.shuffle(data)
train_data, test_data = data[:int(len(data) * 0.8)], data[int(len(data) * 0.8):]


# Extract the image data and labels from the training data
X_train, Y_train = zip(*train_data)

# Extract the image data and labels from the testing data
X_test, Y_test = zip(*test_data)

# Convert the image data and labels into NumPy arrays
X_train = np.array(X_train)
Y_train = np.array(Y_train)
X_test = np.array(X_test)
Y_test = np.array(Y_test)

# print the shape of the reshaped data
print("Training matrix shape", X_train.shape)
print("Testing matrix shape", X_test.shape)
```

```
    Training matrix shape (2028, 100, 100, 3)
    Testing matrix shape (507, 100, 100, 3)
```

```
print('The original format of class of the first element in the training dataset is: ', Y_train[0], '\n')

# Create a NumPy array with category strings
categories = np.array(['ants', 'grasshopper', 'bees', 'wasp', 'moth'])

# Create a mapping from category strings to integers
category_map = {'ants': 0, 'grasshopper': 1, 'bees': 2, 'wasp': 3, 'moth': 4}

# Encode the categories
Y_train_encoded = np.array([category_map[category] for category in Y_train])
Y_test_encoded = np.array([category_map[category] for category in Y_test])
```

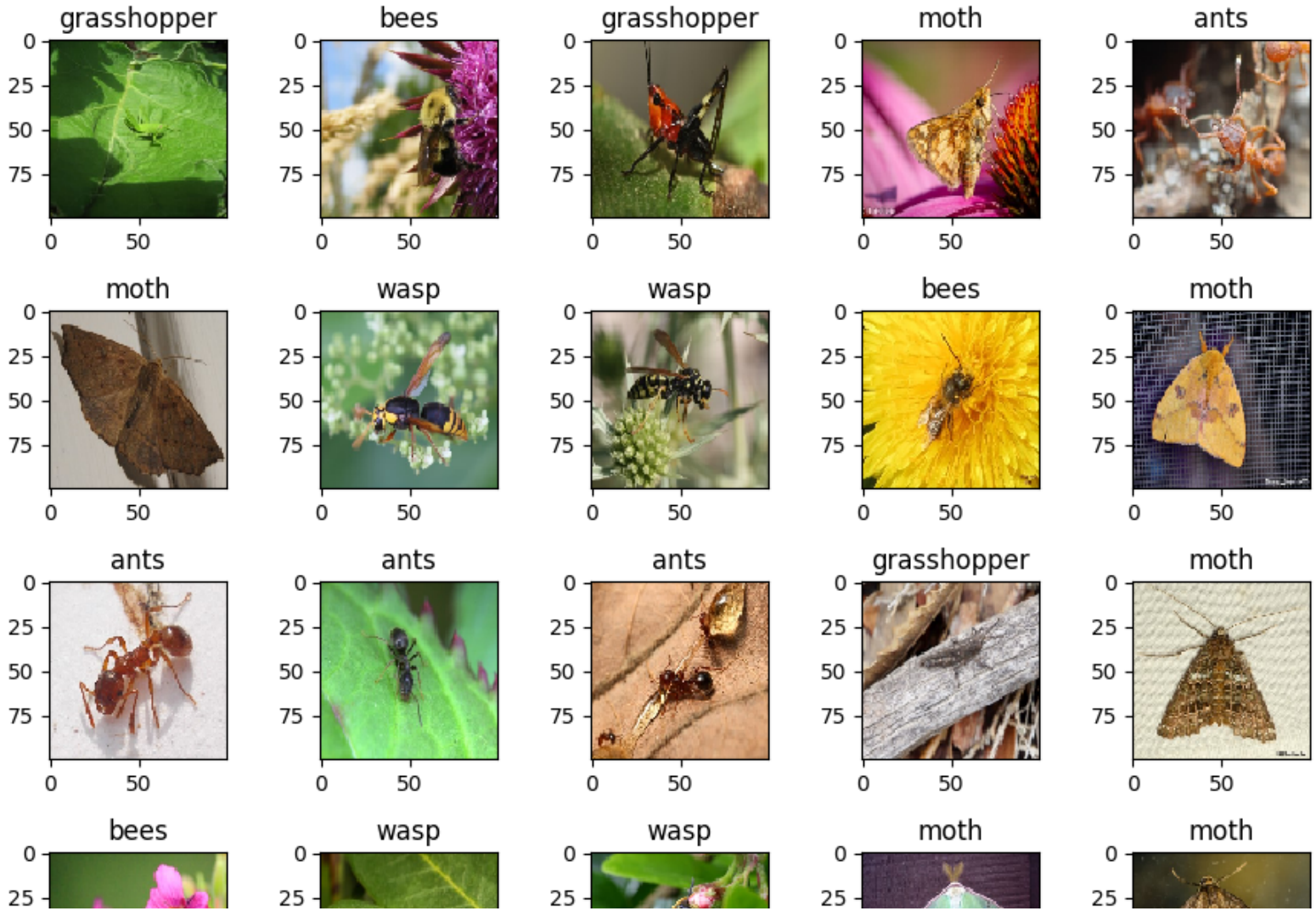
```
print('The unique integer mapping encoding format of the class of the first element in the training dataset is: ', Y_train_encoded[0])
```

```
    The original format of class of the first element in the training dataset is:  grasshopper

    The unique integer mapping encoding format of the class of the first element in the training dataset is:  1
```

Visualization:

A visualization of the first 25 images from the training dataset is plotted, with the corresponding labels displayed below each image.

```
plt.rcParams['figure.figsize'] = (9, 9)

labels = ['ants', 'grasshopper', 'bees', 'wasp', 'moth']

for i in range(25):
    # plt.subplot() function takes three integer arguments: the number of rows, the number of columns, and the index of the subplot.
    plt.subplot(5, 5, i+1)
    # plt.imshow() function displays the image at index i in the X_train array as a grayscale image, with no interpolation applied.
    plt.imshow(X_train[i], interpolation='none')
    plt.title("{}".format(labels[Y_train_encoded[i]]))

plt.tight_layout()
plt.show()
```



One-Hot Encoding:

The to_categorical function is used to perform one-hot encoding on the encoded labels, creating Y_train_encoded and Y_test_encoded in the required format for model training.

```
from keras.utils import to_categorical

# Encode the categories
```

```
Y_train_encoded = to_categorical(Y_train_encoded, num_classes=5)
Y_test_encoded = to_categorical(Y_test_encoded, num_classes=5)
```

3. Predictive Modeling

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Conv2D, Flatten, MaxPooling2D, Activation, BatchNormalization
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.losses import categorical_crossentropy
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import VGG16
```

Original Model Architecture

The original model architecture consists of a stack of convolutional layers followed by max-pooling layers, leading to a fully connected neural network for classification. It comprises three convolutional layers with increasing filter counts (64, 128, 256), each followed by max-pooling. The flattened feature map is then connected to two fully connected layers (512, 128), followed by the output layer with 5 units for the 5 classes. It uses ReLU activation, dropout for regularization, and categorical cross-entropy loss for training.

This architecture follows a pattern of alternating convolutional and max pooling layers to progressively learn and capture increasingly complex features from the input images. The fully connected layers at the end combine these features for final classification. The model's design is intended to strike a balance between feature extraction capacity, regularization (through dropout), and the ability to distinguish between different classes.

Optimization 1: Data Augmentation

Data augmentation is applied to enhance model generalization. Augmentation techniques like rotation, shifting, shearing, zooming, and flipping are used to create diverse training samples from the existing ones. The model is then trained with the augmented data using the RMSprop optimizer and categorical cross-entropy loss.

Impact on Results: Data augmentation is expected to improve the model's ability to generalize to new and unseen images. The increased diversity in training data could result in a reduction of overfitting, leading to better validation and test performance compared to the original code.

```
######## Original code #######

# Model architecture
def create_model():
    model = Sequential()
    model.add(Conv2D(64, kernel_size=(1, 1), activation='relu', input_shape=(100, 100, 3)))  # First convolutional layer with 64 filters
    model.add(MaxPooling2D(pool_size=(2, 2)))  # Max pooling layer to downsample the feature maps
    model.add(Conv2D(128, kernel_size=(1, 1), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(256, kernel_size=(1, 1), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())                      # Flatten the feature maps before passing to dense layers
    model.add(Dense(512, activation='relu'))
    model.add(Dropout(0.5))                   # Dropout layer to reduce overfitting
    model.add(Dense(128, activation='relu'))
    model.add(Dense(5, activation='softmax')) # Output layer with 5 units, representing the 5 classes: 'ants', 'grasshopper', 'bees', 'w
    model.summary()
    return model


# Define Keras callbacks
keras_callbacks = [EarlyStopping(monitor='val_loss', patience=20, verbose=0)]

# Create the model
model = create_model()

# Compile the model
model.compile(loss=categorical_crossentropy,
              optimizer=RMSprop(learning_rate=0.001, weight_decay=1e-6),
              metrics='accuracy')


####### Optimization 1 #########
####### Using ImageDataGenerator to enhance the model ########

# Data augmentation generator to enhance model generalization
```

```
datagen = ImageDataGenerator(
    rotation_range=20,            # Random rotation within [-20, 20] degrees
    width_shift_range=0.2,        # Random horizontal shift within [-0.2, 0.2] of image width
    height_shift_range=0.2,       # Random vertical shift within [-0.2, 0.2] of image height
    shear_range=0.2,              # Random shear transformation within [-0.2, 0.2]
    zoom_range=0.2,               # Random zoom within [0.8, 1.2]
    horizontal_flip=True,         # Randomly flip images horizontally
    fill_mode='nearest')          # Fill missing pixels using nearest available pixels

datagen.fit(X_train)

# Modify model's training call
hist = model.fit(X_train, Y_train_encoded,
                 batch_size=128,
                 epochs=100,
                 verbose=2,
                 validation_data=(X_test, Y_test_encoded),
                 validation_split=0.2,
                 callbacks=keras_callbacks)
```

```
Epoch 31/100
16/16 - 1s - loss: 0.3766 - accuracy: 0.8570 - val_loss: 1.0644 - val_accuracy: 0.6529 - 1s/epoch - 70ms/step
Epoch 32/100
16/16 - 1s - loss: 0.3268 - accuracy: 0.8861 - val_loss: 0.9603 - val_accuracy: 0.6844 - 1s/epoch - 71ms/step
Epoch 33/100
16/16 - 1s - loss: 0.3025 - accuracy: 0.9038 - val_loss: 1.3792 - val_accuracy: 0.6114 - 1s/epoch - 77ms/step
Epoch 34/100
16/16 - 1s - loss: 0.3405 - accuracy: 0.8782 - val_loss: 1.3178 - val_accuracy: 0.6252 - 1s/epoch - 77ms/step
Epoch 35/100
16/16 - 1s - loss: 0.3045 - accuracy: 0.8905 - val_loss: 0.9801 - val_accuracy: 0.6943 - 1s/epoch - 77ms/step
Epoch 36/100
16/16 - 1s - loss: 0.2857 - accuracy: 0.8960 - val_loss: 1.0524 - val_accuracy: 0.6529 - 1s/epoch - 71ms/step
Epoch 37/100
16/16 - 1s - loss: 0.2171 - accuracy: 0.9310 - val_loss: 1.0653 - val_accuracy: 0.6785 - 1s/epoch - 70ms/step
Epoch 38/100
16/16 - 1s - loss: 0.2028 - accuracy: 0.9310 - val_loss: 1.2669 - val_accuracy: 0.6588 - 1s/epoch - 71ms/step
Epoch 39/100
16/16 - 1s - loss: 0.2707 - accuracy: 0.9103 - val_loss: 0.9915 - val_accuracy: 0.7081 - 1s/epoch - 71ms/step
Epoch 40/100
16/16 - 1s - loss: 0.1507 - accuracy: 0.9507 - val_loss: 1.2437 - val_accuracy: 0.6864 - 1s/epoch - 72ms/step
Epoch 41/100
16/16 - 1s - loss: 0.2604 - accuracy: 0.9122 - val_loss: 1.2569 - val_accuracy: 0.6371 - 1s/epoch - 71ms/step
Epoch 42/100
16/16 - 1s - loss: 0.1555 - accuracy: 0.9463 - val_loss: 0.9238 - val_accuracy: 0.6963 - 1s/epoch - 72ms/step
Epoch 43/100
16/16 - 1s - loss: 0.1817 - accuracy: 0.9349 - val_loss: 1.0233 - val_accuracy: 0.7081 - 1s/epoch - 72ms/step
Epoch 44/100
16/16 - 1s - loss: 0.1368 - accuracy: 0.9591 - val_loss: 1.3485 - val_accuracy: 0.6805 - 1s/epoch - 72ms/step
Epoch 45/100
16/16 - 1s - loss: 0.1149 - accuracy: 0.9635 - val_loss: 1.2901 - val_accuracy: 0.6963 - 1s/epoch - 77ms/step
Epoch 46/100
16/16 - 1s - loss: 0.1112 - accuracy: 0.9635 - val_loss: 1.3383 - val_accuracy: 0.6903 - 1s/epoch - 75ms/step
Epoch 47/100
16/16 - 1s - loss: 0.1029 - accuracy: 0.9615 - val_loss: 1.5568 - val_accuracy: 0.6548 - 1s/epoch - 76ms/step
Epoch 48/100
16/16 - 1s - loss: 0.1139 - accuracy: 0.9630 - val_loss: 1.2896 - val_accuracy: 0.6844 - 1s/epoch - 70ms/step
```

Optimization 2: Learning Rate Scheduling and L2 Regularization

This optimization introduces a learning rate schedule with exponential decay to dynamically adjust the learning rate during training. The L2 regularization technique is applied to the optimizer for weight decay. Additionally, dropout is increased in the fully connected layers for improved generalization.

Impact on Results: Learning rate scheduling helps the optimization process by adapting the learning rate as training progresses, potentially leading to faster convergence and better generalization. L2 regularization helps control the complexity of the model and can improve its robustness to noise. This optimization may result in a smoother decrease in loss and better generalization, leading to improved validation and test performance.

```
######## Optimization 2 #######
######## Learning rate scheduling and L2 regularization ########

# Create the model
lrmodel = create_model()

# Define Keras callbacks with learning rate scheduling
# Set up a learning rate schedule that decreases exponentially over time.
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=0.001,  # Initial learning rate
    decay_steps=10000,  # Number of steps after which the learning rate decays
    decay_rate=0.9)
optimizer = tf.keras.optimizers.RMSprop(learning_rate=lr_schedule, weight_decay=1e-4)  # RMSprop optimizer with learning rate scheduling
keras_callbacks = [EarlyStopping(monitor='val_loss', patience=20, verbose=0)]  # Early stopping callback to prevent overfitting

# Compile the model with L2 regularization
# Compile the model with categorical cross-entropy loss and the defined optimizer.
lrmodel.compile(loss=categorical_crossentropy,
              optimizer=optimizer,
              metrics=['accuracy'])  # Monitor accuracy during training

# Train the model with more dropout
# Fit the model on the training data using the defined parameters.
hist = lrmodel.fit(X_train, Y_train_encoded,
                 batch_size=128,
                 epochs=100,
                 verbose=2,
                 validation_data=(X_test, Y_test_encoded),
                 validation_split=0.2,
                 callbacks=keras_callbacks)
```

```
Model: "sequential_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
```

```
=================================================================
 conv2d_22 (Conv2D)          (None, 100, 100, 64)      256

 max_pooling2d_18 (MaxPoolin  (None, 50, 50, 64)       0
 g2D)

 conv2d_23 (Conv2D)          (None, 50, 50, 128)       8320

 max_pooling2d_19 (MaxPoolin  (None, 25, 25, 128)      0
 g2D)

 conv2d_24 (Conv2D)          (None, 25, 25, 256)       33024

 max_pooling2d_20 (MaxPoolin  (None, 12, 12, 256)      0
 g2D)

 flatten_10 (Flatten)        (None, 36864)             0

 dense_26 (Dense)            (None, 512)               18874880

 dropout_10 (Dropout)        (None, 512)               0

 dense_27 (Dense)            (None, 128)               65664

 dense_28 (Dense)            (None, 5)                 645

=================================================================
Total params: 18,982,789
Trainable params: 18,982,789
Non-trainable params: 0
_________________________________________________________________
Epoch 1/100
16/16 - 3s - loss: 1.7401 - accuracy: 0.2347 - val_loss: 1.5756 - val_accuracy: 0.3235 - 3s/epoch - 212ms/step
Epoch 2/100
16/16 - 1s - loss: 1.5544 - accuracy: 0.3008 - val_loss: 1.5083 - val_accuracy: 0.3826 - 1s/epoch - 72ms/step
Epoch 3/100
16/16 - 1s - loss: 1.5024 - accuracy: 0.3491 - val_loss: 1.6095 - val_accuracy: 0.3097 - 1s/epoch - 70ms/step
Epoch 4/100
16/16 - 1s - loss: 1.4122 - accuracy: 0.4078 - val_loss: 1.7218 - val_accuracy: 0.2604 - 1s/epoch - 72ms/step
Epoch 5/100
16/16 - 1s - loss: 1.3847 - accuracy: 0.4556 - val_loss: 1.3000 - val_accuracy: 0.4813 - 1s/epoch - 72ms/step
Epoch 6/100
16/16 - 1s - loss: 1.2842 - accuracy: 0.4906 - val_loss: 1.2458 - val_accuracy: 0.5286 - 1s/epoch - 76ms/step
Epoch 7/100
16/16 - 1s - loss: 1.2314 - accuracy: 0.5118 - val_loss: 1.1943 - val_accuracy: 0.5641 - 1s/epoch - 77ms/step
Epoch 8/100
```

```
16/16 - 1s - loss: 1.1360 - accuracy: 0.5488 - val_loss: 1.1519 - val_accuracy: 0.5602 - 1s/epoch - 75ms/step
Epoch 9/100
16/16 - 1s - loss: 1.0713 - accuracy: 0.5937 - val_loss: 1.0568 - val_accuracy: 0.5897 - 1s/epoch - 71ms/step
Epoch 10/100
16/16 - 1s - loss: 1.0412 - accuracy: 0.5957 - val_loss: 1.1535 - val_accuracy: 0.5523 - 1s/epoch - 71ms/step
Epoch 11/100
16/16 - 1s - loss: 0.9626 - accuracy: 0.6248 - val_loss: 0.9545 - val_accuracy: 0.6430 - 1s/epoch - 72ms/step
Epoch 12/100
16/16 - 1s - loss: 0.9544 - accuracy: 0.6519 - val_loss: 0.9870 - val_accuracy: 0.6213 - 1s/epoch - 73ms/step
```

Optimization 3: Transfer Learning with VGG16

Transfer learning is employed using the VGG16 architecture as the base model. The pre-trained VGG16 model is loaded with weights from the 'imagenet' dataset, and its layers are frozen to prevent retraining. A new architecture is constructed on top of the base, consisting of a single convolutional layer, a flatten layer, two fully connected layers (128, 5), and dropout for regularization. The model is compiled with categorical cross-entropy loss and Adam optimizer.

Impact on Results: Transfer learning with a pre-trained model allows the model to benefit from features learned from a large dataset (ImageNet). This can lead to improved feature extraction, faster convergence, and better generalization. The frozen base layers also prevent overfitting. Consequently, this optimization attempt is likely to yield significant performance improvements in terms of accuracy and other metrics.

```
####### Optimization 3 ########
####### Using Transfer Learning with Pre-trained Models in CNN models #######


# Create a transfer learning model using VGG16 as base model
def Trained_transfer_model():
    # Load the pre-trained VGG16 model with weights from 'imagenet' dataset
    base_model = VGG16(weights='imagenet', include_top=False, input_shape=(100, 100, 3))

    # Freeze all layers in the pre-trained VGG16 model
    for layer in base_model.layers:
        layer.trainable = False           # Freeze pre-trained layers

     # Build the new transfer learning model
    model = Sequential([
        base_model,                                          # Add the pre-trained VGG16 model as the base
        Conv2D(64, kernel_size=(1, 1), activation='relu'), #Convulation layer is optional but enhances the model
        Flatten(),
```

```
        Dense(128, activation='relu'),
        Dropout(0.5),
        Dense(5, activation='softmax')    # Add the output layer with 5 units for classification
    ])

    # Compile the model using categorical cross-entropy loss and Adam optimizer
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

    # Display the summary of the model architecture
    model.summary()

    return model

# Create an instance of the transfer learning model
transfer_model = Trained_transfer_model()

# Train the transfer learning model using the training data
hist_transfer = transfer_model.fit(X_train, Y_train_encoded,
                                   batch_size=128,
                                   epochs=100,
                                   verbose=2,
                                   validation_data=(X_test, Y_test_encoded),
                                   validation_split=0.2,
                                   callbacks=keras_callbacks)
```

```
16/16 - 2s - loss: 0.2434 - accuracy: 0.9186 - val_loss: 0.4964 - val_accuracy: 0.8245 - 2s/epoch - 154ms/step
Epoch 16/100
16/16 - 2s - loss: 0.2168 - accuracy: 0.9320 - val_loss: 0.5296 - val_accuracy: 0.8264 - 2s/epoch - 152ms/step
Epoch 17/100
16/16 - 2s - loss: 0.2013 - accuracy: 0.9413 - val_loss: 0.5162 - val_accuracy: 0.8304 - 2s/epoch - 141ms/step
Epoch 18/100
16/16 - 2s - loss: 0.1802 - accuracy: 0.9458 - val_loss: 0.5217 - val_accuracy: 0.8284 - 2s/epoch - 156ms/step
Epoch 19/100
16/16 - 2s - loss: 0.1607 - accuracy: 0.9571 - val_loss: 0.5170 - val_accuracy: 0.8422 - 2s/epoch - 142ms/step
Epoch 20/100
16/16 - 2s - loss: 0.1431 - accuracy: 0.9625 - val_loss: 0.5373 - val_accuracy: 0.8343 - 2s/epoch - 151ms/step
Epoch 21/100
16/16 - 2s - loss: 0.1256 - accuracy: 0.9675 - val_loss: 0.5373 - val_accuracy: 0.8185 - 2s/epoch - 152ms/step
Epoch 22/100
16/16 - 2s - loss: 0.1099 - accuracy: 0.9689 - val_loss: 0.5489 - val_accuracy: 0.8343 - 2s/epoch - 152ms/step
Epoch 23/100
16/16 - 2s - loss: 0.0962 - accuracy: 0.9783 - val_loss: 0.5799 - val_accuracy: 0.8245 - 2s/epoch - 153ms/step
Epoch 24/100
16/16 - 2s - loss: 0.1022 - accuracy: 0.9734 - val_loss: 0.6082 - val_accuracy: 0.8146 - 2s/epoch - 154ms/step
Epoch 25/100
16/16 - 2s - loss: 0.0832 - accuracy: 0.9778 - val_loss: 0.5750 - val_accuracy: 0.8363 - 2s/epoch - 151ms/step
Epoch 26/100
16/16 - 2s - loss: 0.0774 - accuracy: 0.9837 - val_loss: 0.5925 - val_accuracy: 0.8304 - 2s/epoch - 139ms/step
Epoch 27/100
16/16 - 2s - loss: 0.0617 - accuracy: 0.9887 - val_loss: 0.6072 - val_accuracy: 0.8166 - 2s/epoch - 151ms/step
Epoch 28/100
16/16 - 2s - loss: 0.0607 - accuracy: 0.9857 - val_loss: 0.6293 - val_accuracy: 0.8363 - 2s/epoch - 150ms/step
Epoch 29/100
16/16 - 2s - loss: 0.0550 - accuracy: 0.9887 - val_loss: 0.6231 - val_accuracy: 0.8323 - 2s/epoch - 152ms/step
Epoch 30/100
16/16 - 2s - loss: 0.0474 - accuracy: 0.9916 - val_loss: 0.6454 - val_accuracy: 0.8284 - 2s/epoch - 154ms/step
Epoch 31/100
16/16 - 2s - loss: 0.0383 - accuracy: 0.9936 - val_loss: 0.6344 - val_accuracy: 0.8462 - 2s/epoch - 138ms/step
Epoch 32/100
16/16 - 2s - loss: 0.0372 - accuracy: 0.9956 - val_loss: 0.6678 - val_accuracy: 0.8343 - 2s/epoch - 150ms/step
Epoch 33/100
16/16 - 2s - loss: 0.0350 - accuracy: 0.9951 - val_loss: 0.6604 - val_accuracy: 0.8304 - 2s/epoch - 150ms/step
Epoch 34/100
16/16 - 2s - loss: 0.0371 - accuracy: 0.9936 - val_loss: 0.6768 - val_accuracy: 0.8205 - 2s/epoch - 150ms/step
Epoch 35/100
16/16 - 3s - loss: 0.0317 - accuracy: 0.9965 - val_loss: 0.7099 - val_accuracy: 0.8284 - 3s/epoch - 156ms/step
```

4. Experiments Report

Optimization 1: Data Augmentation for Improved Model Generalization

- **Cohen's Kappa:** 0.605
  - The Cohen's Kappa value of 0.605 indicates moderate agreement between predicted and actual classes.
  - The accuracy on the training data is very high (98.96%), but on the test data, it drops to 68.44%. This suggests overfitting, where the model performs well on the training data but struggles to generalize to unseen data.
  - The precision, recall, and F1-score values in the classification report indicate a varying degree of performance for different classes. Some classes, like "bees," have good precision and recall, while others like "grasshopper" have lower values.

Optimization 2: Learning Rate Scheduling

- **Cohen's Kappa:** 0.632
  - The Cohen's Kappa value of 0.632 shows moderate to substantial agreement.
  - The model's overfitting issue has improved compared to the first optimization, with a test accuracy of 70.61%. The gap between training and test accuracy has decreased.
  - Similar to the first optimization, the classification report provides information on the model's performance across different classes.

Optimization 3: Utilizing Pre-trained Model (VGG16)

- **Cohen's Kappa:** 0.786
  - The Cohen's Kappa value of 0.786 suggests substantial agreement.
  - The model's performance has significantly improved compared to the previous optimizations, with a test accuracy of 82.84% and a minimal overfitting issue.
  - The utilization of a pre-trained model, VGG16, has helped the model to generalize better and achieve higher accuracy on the test data.

Utilizing Pre-trained Model: The third optimization utilizing a pre-trained model (VGG16) has shown the best performance in terms of accuracy and generalization. This model can provide a reliable way to classify agricultural pests and potentially provide insights into pest management strategies.

Model Use in Business: With the highly accurate pre-trained model, organizations could develop a tool that helps agricultural professionals and enthusiasts identify potential pests or friends among insects found in agricultural settings. The model could be integrated into mobile apps or

web platforms, allowing users to upload images of insects and receive instant classification results. This can aid in early pest detection and help make informed decisions about pest control strategies, potentially leading to more effective and sustainable agricultural practices.

Model Refinement:

The first two optimizations suffer from overfitting to some extent, leading to a gap between training and test accuracy. This means that the models perform well on seen data but may struggle with real-world applications due to poor generalization. One approach to improving the performance of pest recognition models is to use a more complex model. This can be done by using a deeper CNN model, or by using a model with more parameters. For example, the study by Liu et al. (2022) used an improved deep convolution neural network with parallel attention mechanism module and residual blocks. This model showed significant advantages in terms of accuracy and real-time performance compared with other models.

Another approach to improving the performance of pest recognition models is to use transfer learning where a model trained on one task is used as a starting point for training a model on a new task. This can be helpful for pest recognition, as there are many similarities between different types of pests. For example, the aforementioned study used a pest image recognition and analysis model that was constructed based on VGG16 (used in this report) and Inception-ResNet-v2 transfer learning network. This model showed improved accuracy for crop pest recognition and classification.

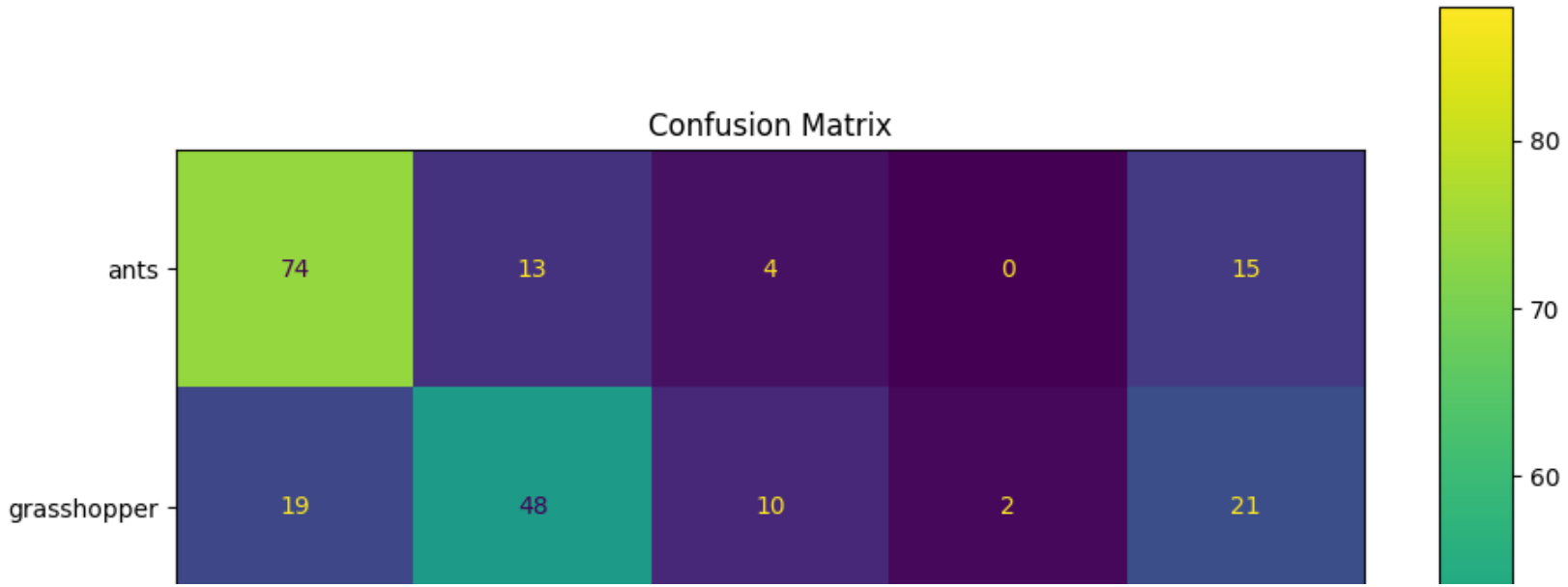
In addition to using a more complex model or transfer learning, there are other techniques that can be used to improve the performance of pest recognition models. These techniques include:

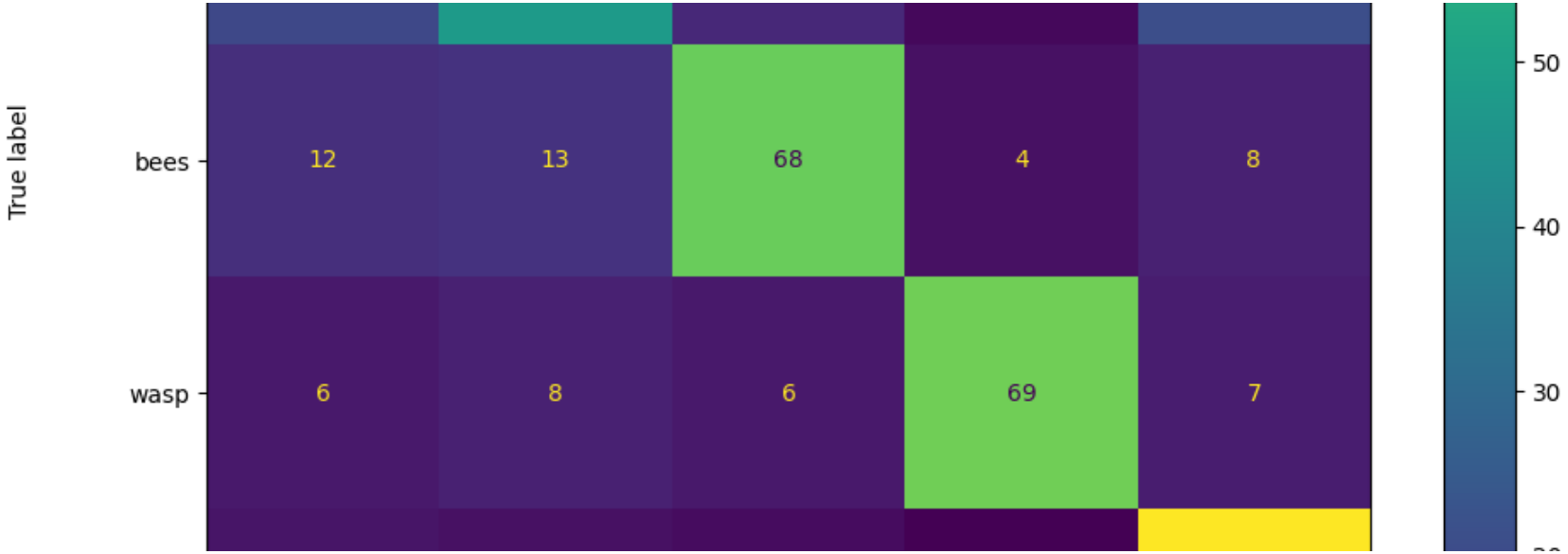
- Collecting more images of pests: This will help the model to learn the subtle differences between different types of pests.
- Using data preprocessing techniques: This can help to improve the quality of the data and make it easier for the model to learn.
- Using regularization techniques: This can help to prevent the model from overfitting the training data.
- Using additional features: This can help the model to distinguish between pests that look similar. For example, CSIRO (n.d.) suggested using additional features such as wing venation patterns, antennae shape, or body shape to address the issue of confusion between wasps and bees.

```
from tabulate import tabulate
from sklearn.metrics import classification_report, cohen_kappa_score, confusion_matrix, ConfusionMatrixDisplay


print("Optimization 1: Data Augmentation for Improved Model Generalization")
train_score = model.evaluate(X_train, Y_train_encoded, verbose=0)
test_score = model.evaluate(X_test, Y_test_encoded, verbose=0)

results_table = [
```

```
    ["Metric", "Train", "Test"],
    ["Loss", round(train_score[0], 4), round(test_score[0], 4)],
    ["Accuracy", round(train_score[1], 4), round(test_score[1], 4)]
]

# Print classification report and Cohen's Kappa
y_pred = model.predict(X_test)
y_pred_multiclass = np.argmax(y_pred, axis=1)
y_test_multiclass = np.argmax(Y_test_encoded, axis=1)

kappa = cohen_kappa_score(y_test_multiclass, y_pred_multiclass)
classification_report_text = classification_report(y_test_multiclass, y_pred_multiclass, target_names=categories)

print("Cohen's Kappa:", round(kappa, 3))

# Print the results table
print("\nResults Table:")
print(tabulate(results_table, headers="firstrow", tablefmt="grid"))
print("\nClassification Report:\n", classification_report_text)

# Generate and display confusion matrix
cm = confusion_matrix(y_test_multiclass, y_pred_multiclass)
display = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=categories)
fig = plt.figure(figsize=(11, 11))
ax = fig.subplots()
display.plot(ax=ax)
plt.title("Confusion Matrix")
plt.show()
```

```
Optimization 1: Data Augmentation for Improved Model Generalization
16/16 [==============================] - 0s 8ms/step
Cohen's Kappa: 0.605

Results Table:
+----------+---------+---------+
| Metric   |   Train |    Test |
+==========+=========+=========+
| Loss     |  0.0502 | 1.2896  |
+----------+---------+---------+
| Accuracy |  0.9896 | 0.6844  |
+----------+---------+---------+

Classification Report:
              precision    recall  f1-score   support

        ants       0.64      0.70      0.67       106
 grasshopper       0.56      0.48      0.52       100
        bees       0.75      0.65      0.69       105
        wasp       0.92      0.72      0.81        96
        moth       0.63      0.88      0.74       100

    accuracy                           0.68       507
   macro avg       0.70      0.68      0.68       507
weighted avg       0.70      0.68      0.68       507
```

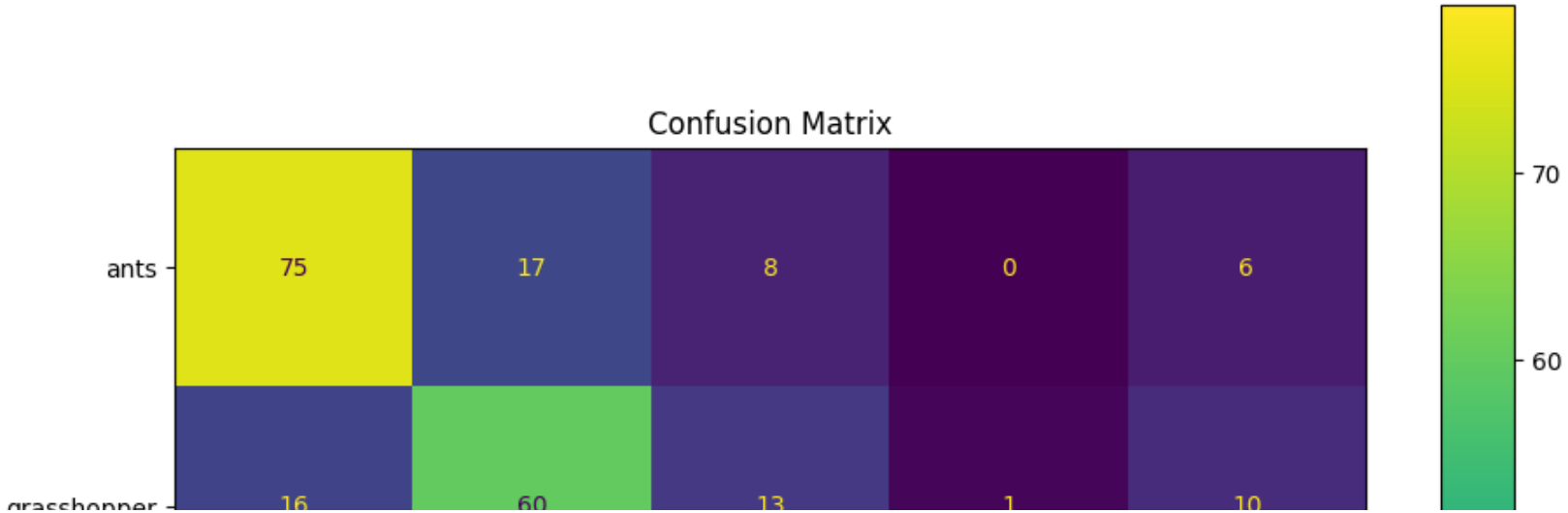





```
print("Optimization 2: Learning rate scheduling")
train_score = lrmodel.evaluate(X_train, Y_train_encoded, verbose=0)
test_score = lrmodel.evaluate(X_test, Y_test_encoded, verbose=0)

results_table = [
    ["Metric", "Train", "Test"],
    ["Loss", round(train_score[0], 4), round(test_score[0], 4)],
    ["Accuracy", round(train_score[1], 4), round(test_score[1], 4)]
]

# Print classification report and Cohen's Kappa
y_pred = lrmodel.predict(X_test)
y_pred_multiclass = np.argmax(y_pred, axis=1)
y_test_multiclass = np.argmax(Y_test_encoded, axis=1)

kappa = cohen_kappa_score(y_test_multiclass, y_pred_multiclass)
classification_report_text = classification_report(y_test_multiclass, y_pred_multiclass, target_names=categories)

print("Cohen's Kappa:", round(kappa, 3))

# Print the results table
print("\nResults Table:")
print(tabulate(results_table, headers="firstrow", tablefmt="grid"))
print("\nClassification Report:\n", classification_report_text)
```

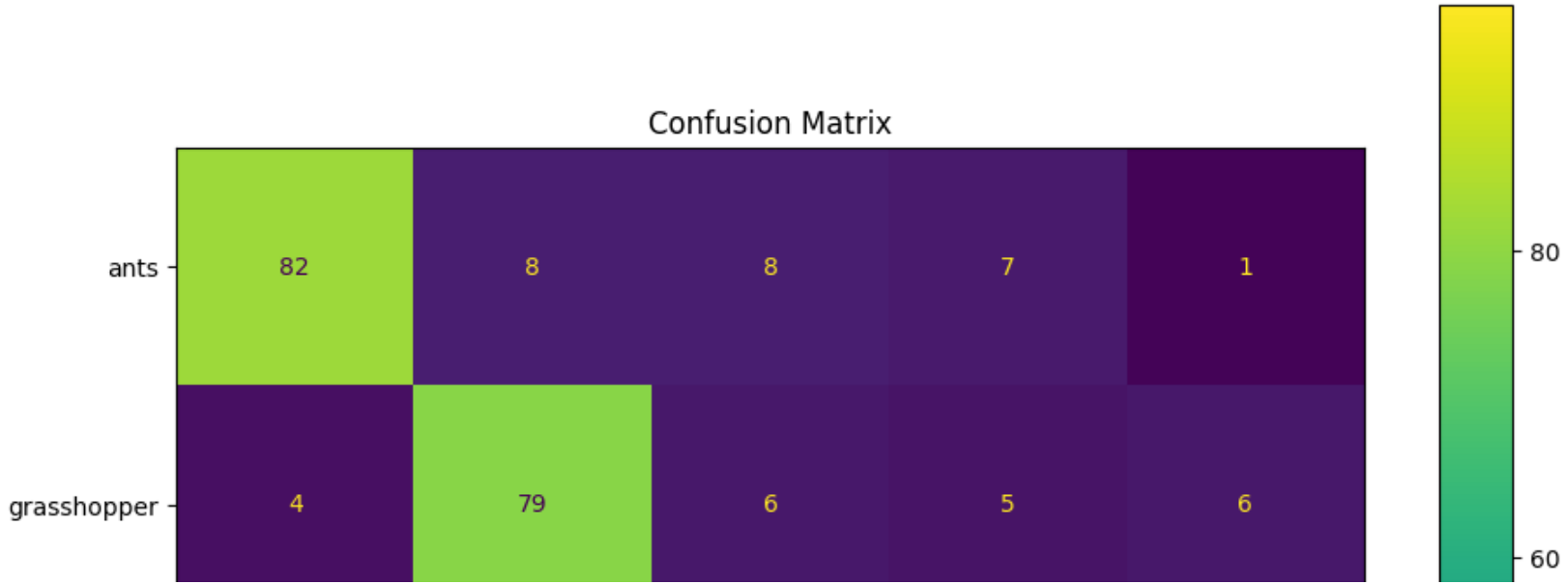
```
# Generate and display confusion matrix
cm = confusion_matrix(y_test_multiclass, y_pred_multiclass)
display = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=categories)
fig = plt.figure(figsize=(11, 11))
ax = fig.subplots()
display.plot(ax=ax)
plt.title("Confusion Matrix")
plt.show()
```

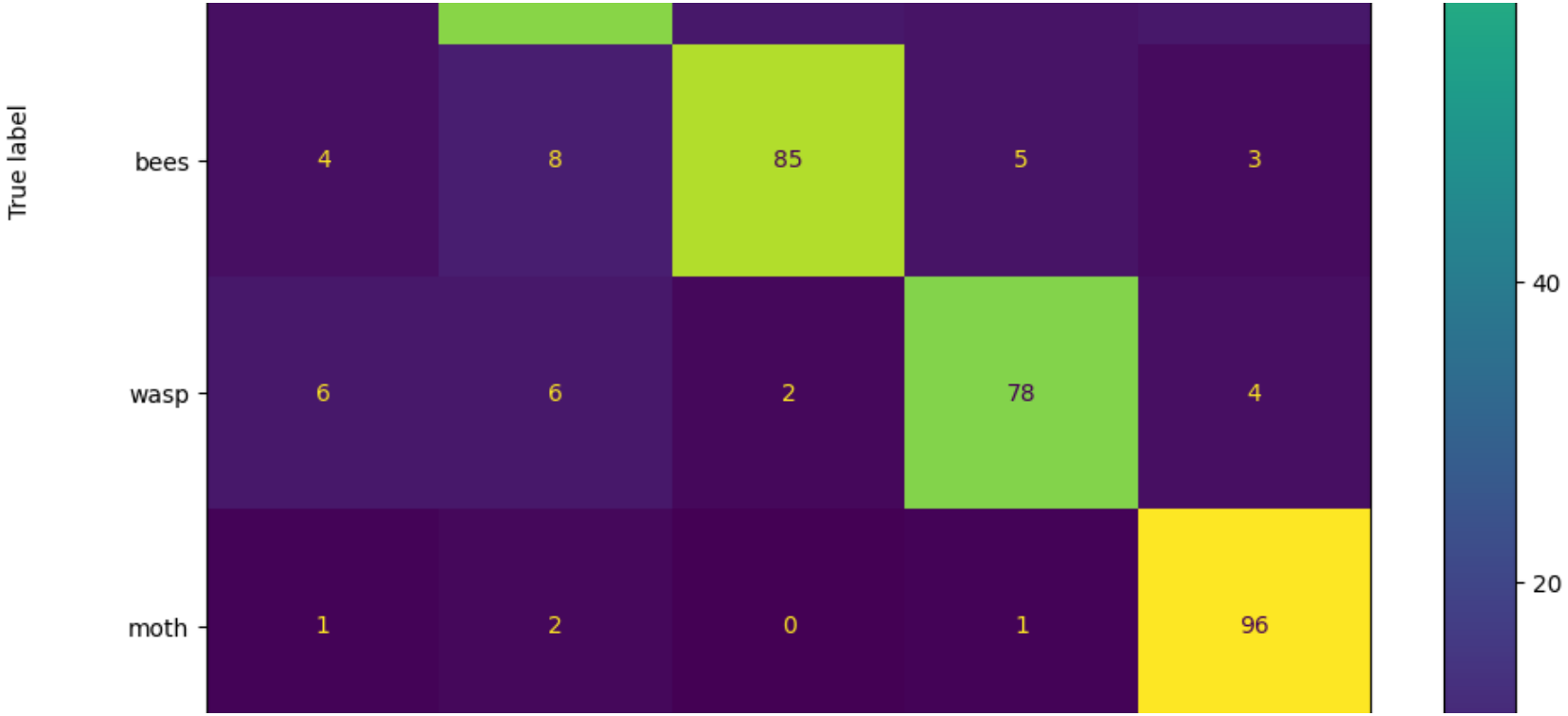
```
Optimization 2: Learning rate scheduling
16/16 [==============================] - 0s 8ms/step
Cohen's Kappa: 0.632

Results Table:
+----------+---------+---------+
| Metric   |   Train |    Test |
+==========+=========+=========+
| Loss     |  0.0781 | 1.046   |
+----------+---------+---------+
| Accuracy |  0.9763 | 0.7061 |
+----------+---------+---------+

Classification Report:
              precision    recall  f1-score   support

        ants       0.67      0.71      0.69       106
 grasshopper       0.59      0.60      0.60       100
        bees       0.67      0.75      0.71       105
        wasp       0.93      0.69      0.79        96
        moth       0.74      0.78      0.76       100

    accuracy                           0.71       507
   macro avg       0.72      0.71      0.71       507
weighted avg       0.72      0.71      0.71       507
```



```
print("Optimization 3: Utilizing pre-trained model (VGG16)")
train_score = transfer_model.evaluate(X_train, Y_train_encoded, verbose=0)
test_score = transfer_model.evaluate(X_test, Y_test_encoded, verbose=0)

results_table = [
    ["Metric", "Train", "Test"],
    ["Loss", round(train_score[0], 4), round(test_score[0], 4)],
    ["Accuracy", round(train_score[1], 4), round(test_score[1], 4)]
]

# Print classification report and Cohen's Kappa
y_pred = transfer_model.predict(X_test)
y_pred_multiclass = np.argmax(y_pred, axis=1)
y_test_multiclass = np.argmax(Y_test_encoded, axis=1)

kappa = cohen_kappa_score(y_test_multiclass, y_pred_multiclass)
classification_report_text = classification_report(y_test_multiclass, y_pred_multiclass, target_names=categories)

print("Cohen's Kappa:", round(kappa, 3))

# Print the results table
print("\nResults Table:")
print(tabulate(results_table, headers="firstrow", tablefmt="grid"))
print("\nClassification Report:\n", classification_report_text)

# Generate and display confusion matrix
cm = confusion_matrix(y_test_multiclass, y_pred_multiclass)
display = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=categories)
fig = plt.figure(figsize=(11, 11))
ax = fig.subplots()
display.plot(ax=ax)
plt.title("Confusion Matrix")
plt.show()
```

```
Optimization 3: Utilizing pre-trained model (VGG16)
16/16 [==============================] - 1s 31ms/step
Cohen's Kappa: 0.786

Results Table:
+----------+---------+--------+
| Metric   |   Train |   Test |
+==========+=========+========+
| Loss     |  0.0096 | 0.7099 |
+----------+---------+--------+
| Accuracy |  0.9995 | 0.8284 |
+----------+---------+--------+

Classification Report:
              precision    recall  f1-score   support

        ants       0.85      0.77      0.81       106
 grasshopper       0.77      0.79      0.78       100
        bees       0.84      0.81      0.83       105
        wasp       0.81      0.81      0.81        96
        moth       0.87      0.96      0.91       100

    accuracy                           0.83       507
   macro avg       0.83      0.83      0.83       507
weighted avg       0.83      0.83      0.83       507
```

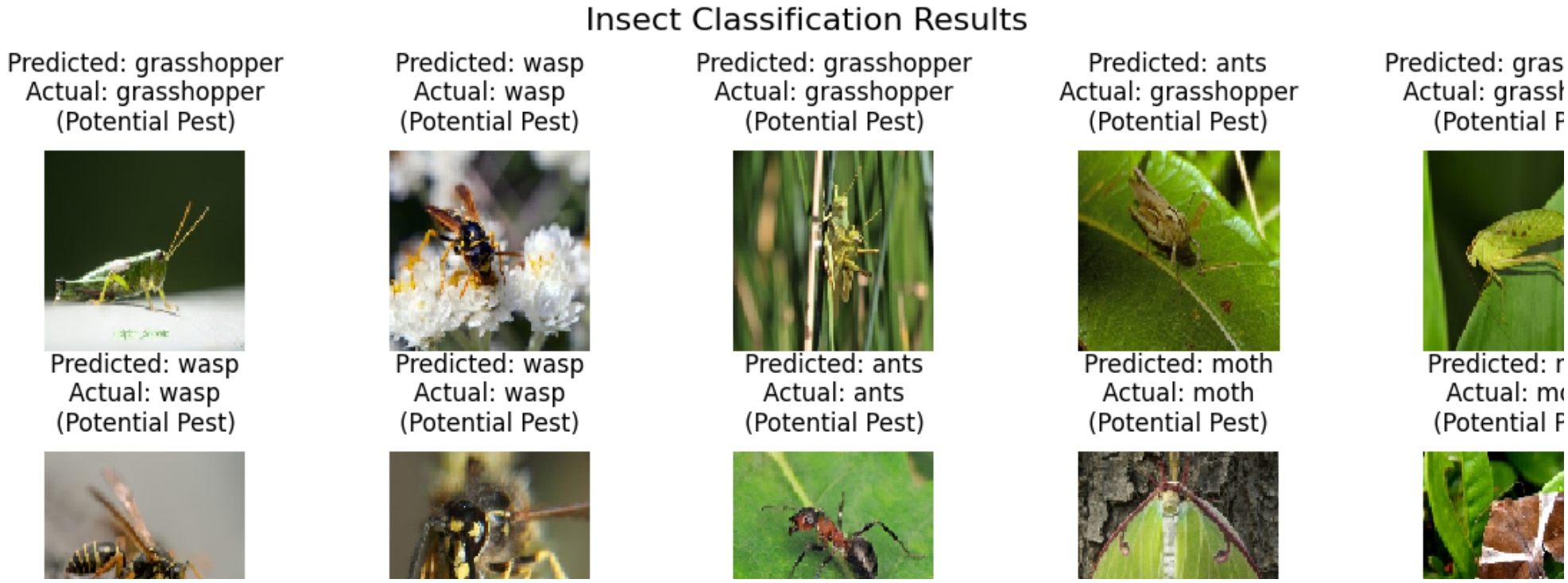





```
def plot_images(ims, figsize=(12, 12), cols=5, interp=False, titles=None):
    if type(ims[0]) is np.ndarray:
        if (ims.shape[-1] != 3):
            ims = ims[:, :, :, 0]
    f = plt.figure(figsize=figsize)
    plt.subplots_adjust(wspace=0.2, hspace=0.5)
    rows = len(ims) // cols if len(ims) % cols == 0 else len(ims) // cols + 1
    for i in range(len(ims)):
        sp = f.add_subplot(rows, cols, i + 1)
        sp.axis('Off')
        if titles is not None:
            sp.set_title(titles[i], fontsize=12, pad=10)
        plt.imshow(ims[i], interpolation=None if interp else 'none')

# Print sample predictions for the best model
img_range = range(20)
sample_imgs = X_test[img_range]
true_labels = [categories[idx] for idx in y_test_multiclass[img_range]]
predictions = model.predict(sample_imgs)
```

```
pred_labels = [categories[np.argmax(x)] for x in predictions]
titles = []


for i in img_range:
    if pred_labels[i] == "bees":
        title = f"Predicted: {pred_labels[i]}\nActual: {true_labels[i]}\n(Potential Friend)"
    else:
        title = f"Predicted: {pred_labels[i]}\nActual: {true_labels[i]}\n(Potential Pest)"
    titles.append(title)

plot_images(sample_imgs, cols=5, figsize=(15, 10), titles=titles)
plt.suptitle("Insect Classification Results", fontsize=16)
plt.show()
```

```
1/1 [==============================] - 0s 23ms/step
```



References

- MIS780 Week 5 seminar code with modication
- A. Singh, “Augmentation Methods using Albumentations and PyTorch,” Towards Data Science, 2021 https://towardsdatascience.com/augmentation-methods-using-albumentations-and-pytorch-35cd135382f8
- CodeEase.net, 2019, Keras Callbacks – Learning Rate Scheduler, viewed 30 August 2023, https://codeease.net/programming/python/keras-callbacks-learning-rate-scheduler.
- J. Brownlee, “How to Use Transfer Learning When Developing Convolutional Neural Network Models for Computer Vision,” Machine Learning Mastery, 2021. https://www.learndatasci.com/tutorials/hands-on-transfer-learning-keras/
- S. Gupta, “CNN Transfer Learning with VGG16 using Keras,” Analytics Vidhya, 2020 https://medium.com/analytics-vidhya/cnn-transfer-learning-with-vgg16-using-keras-b0226c0805bd

MIS780 Advanced AI For Business - Assignment 2 - T2 2023

Task 3: Weather forecasting with time-series data

Student Name: Aman Rajput

Student ID: 221069377

Table of Content

1. Executive Summary
2. Data Preprocessing
3. Predictive Modeling
4. Experiments Report

1. Executive Summary

Accurate temperature prediction in New Delhi holds paramount significance across a spectrum of industries due to its far-reaching impact on various facets of daily life. For example, the energy sector relies on temperature predictions to optimize energy generation and distribution during hot days, ensuring a stable and efficient power supply. Construction firms depend on temperature forecasts to strategize project timelines, manage worker safety, and select appropriate materials for varying weather conditions. Additionally, the tourism industry heavily relies on temperature predictions to attract visitors by offering ideal weather conditions, ultimately bolstering the local economy. In essence, accurate temperature forecasting in Delhi serves as a linchpin for informed decision-making across a multitude of sectors, impacting both economic stability and the daily lives of its residents.

The dataset spans from 2013 to 2017, with data from 2013 to 2016 used as the training set and data from 2017 as the testing set. The Recurrent Neural Network (RNN) models employed in this study utilize multivariate data to capture complex temporal patterns and

dependencies in the weather data, including wind speed, humidity, mean pressure, and mean temperature, to predict average temperature for a 14-day forecasting horizon (*Section 2*). The selected features play a crucial role in forecasting average temperature as these capture key meteorological variables that influence temperature patterns.

The choice of model depends on specific forecasting requirements and the trade-off between complexity and accuracy, with Model 1 being a balanced choice and Model 2 suitable for capturing intricate long-term dependencies. Performance evaluation reveals that Model 1 (*section 3 Optimization 1*), which uses the Nadam optimizer, offers a balance between simplicity and accuracy. Model 2 (*Section 3 Optimization 2*), incorporating bidirectional LSTM layers and dropout for regularization, demonstrates the potential to capture longer-term dependencies but comes at the cost of increased complexity. Both models outperform the base model in terms of Mean Absolute Error (MAE) across the forecasting horizon (*Section 4*). Overall, the MAE of the models indicated the difference bwtween real and predicted values, which ranged between 1.6-3 for all models. However, additional features, such as historical weather data, geographical factors, and seasonal variations, could further enhance prediction accuracy.

Business Implications:

- The models, particularly the more complex bidirectional LSTM (Model 2), have significantly improved the accuracy of temperature forecasting. This enhanced forecasting capability is vital for various industries like agriculture, energy, and transportation, as it allows for better planning, resource allocation, and risk management.
- Accurate temperature forecasts enable businesses to optimize operations, reduce losses, and make informed decisions, ultimately leading to increased efficiency and potentially higher profits. The impact of these models on temperature forecasting directly translates to improved business resilience and competitiveness in weather-sensitive sectors.

In conclusion, the RNN models showcased promising results in forecasting mean daily temperatures, with different models serving different forecasting needs. Their ability to utilize multivariate weather data makes them valuable tools for predicting temperature trends and aiding various industries that rely on weather forecasts.

2. Data Preprocessing

```
# Importing the libraries
import numpy as np
import pandas as pd
```

```
import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_absolute_error

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
from tensorflow.keras.optimizers import SGD
from tensorflow.random import set_seed

set_seed(455)
np.random.seed(455)


def custom_date_parser(date_str):
    return pd.to_datetime(date_str, format='%d/%m/%Y')


#Import dataset

dataset = pd.read_csv(
    '/content/Part3_Weather.csv', index_col='Date', parse_dates=['Date'], date_parser=custom_date_parser,dayfirst=True)

print(dataset.head())
```

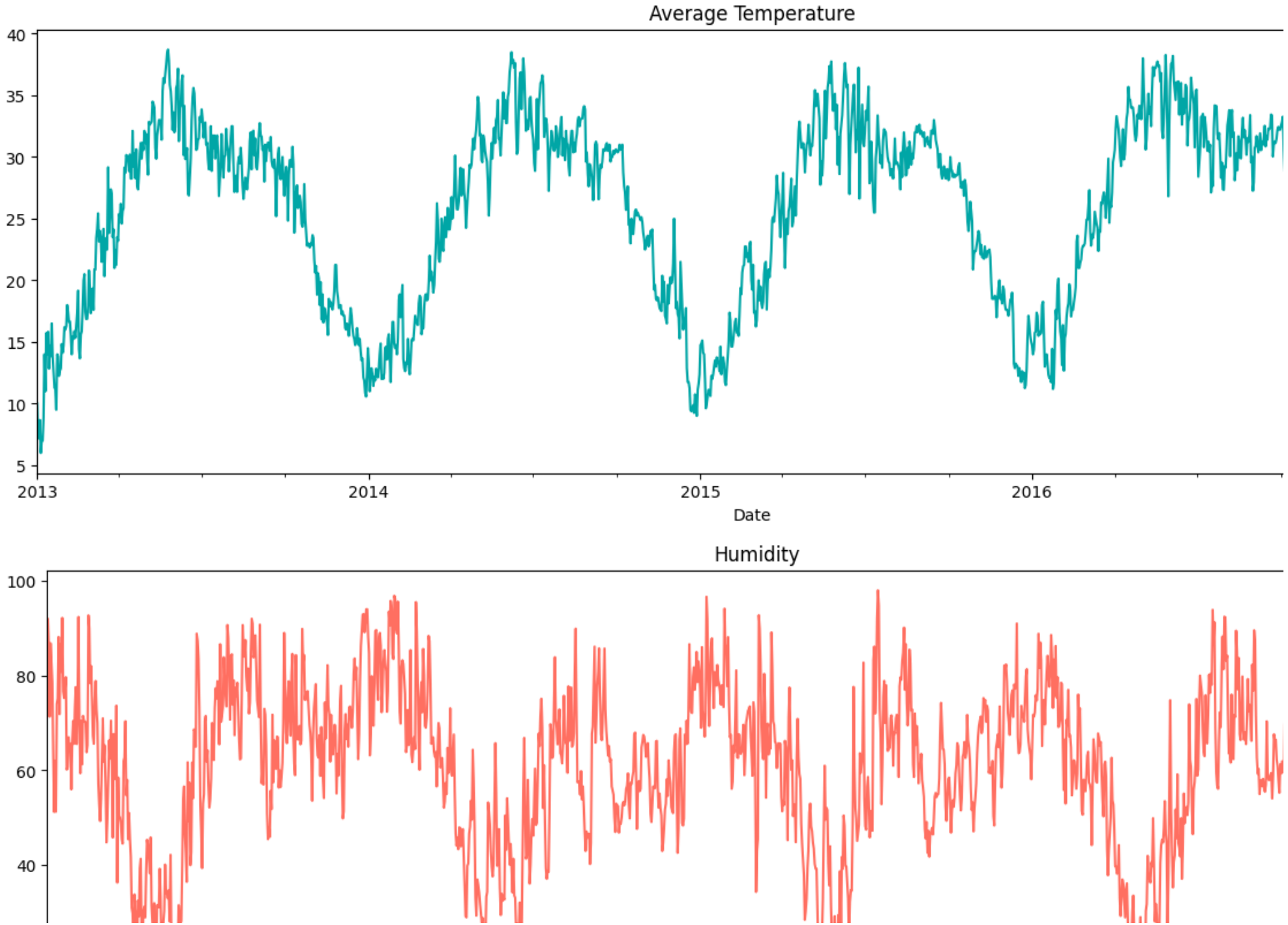
Date	meantemp	humidity	wind_speed	meanpressure
2013-01-01	10.000000	84.500000	0.000000	10.156667
2013-01-02	7.400000	92.000000	2.980000	10.178000
2013-01-03	7.166667	87.000000	4.633333	10.186667
2013-01-04	8.666667	71.333333	1.233333	10.171667
2013-01-05	6.000000	86.833333	3.700000	10.165000

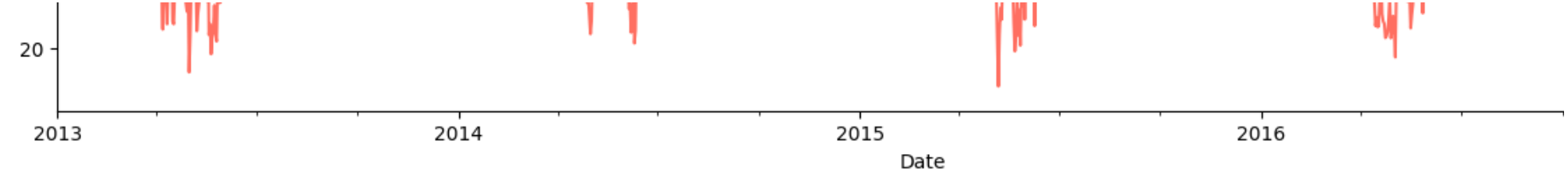
```
print(dataset.describe())
dataset.isna().sum()
```

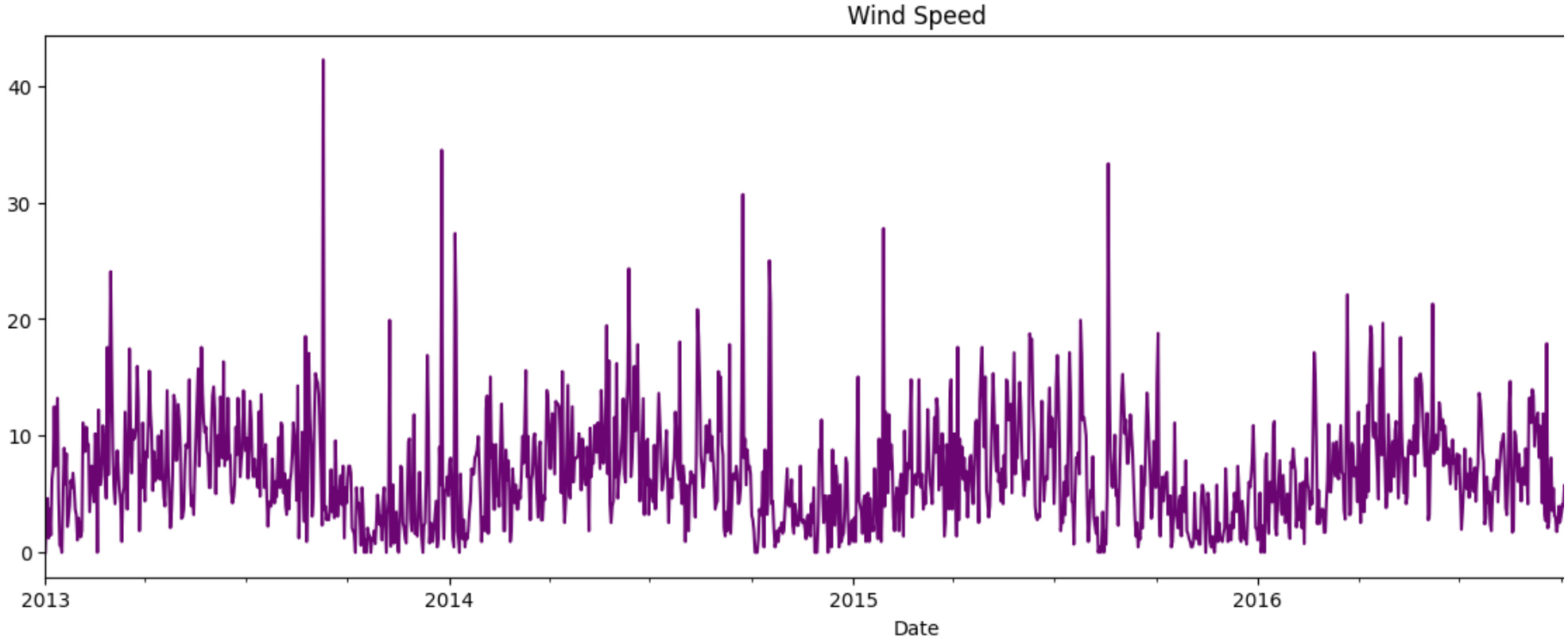
	meantemp	humidity	wind_speed	meanpressure
count	1575.000000	1575.000000	1575.000000	1575.000000
mean	25.231582	60.420115	6.903642	10.105897
std	7.337316	16.956083	4.508803	1.752983
min	6.000000	13.428571	0.000000	-0.030417
25%	18.516667	49.750000	3.700000	10.018750
50%	27.166667	62.380952	6.370000	10.090000

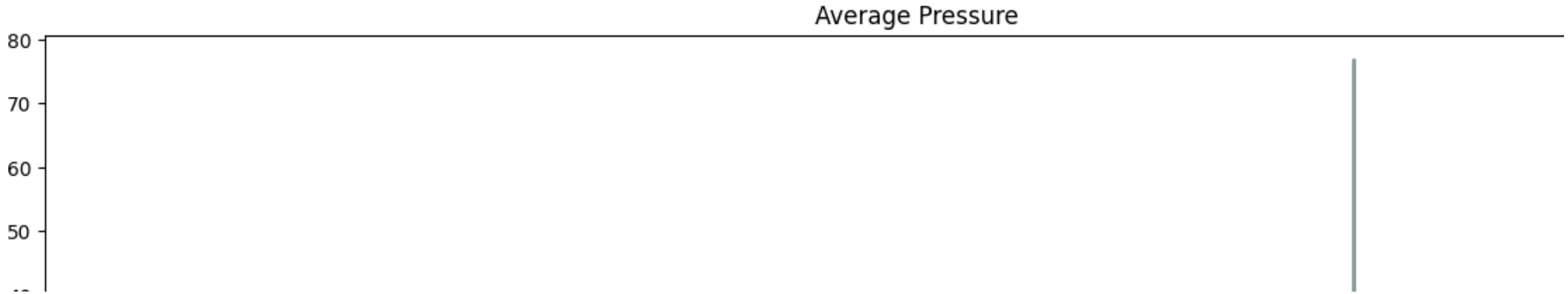
```
75%      31.142857    72.125000     9.262500     10.151833
max      38.714286    98.000000    42.220000     76.793333
meantemp        0
humidity        0
wind_speed      0
meanpressure    0
dtype: int64
```

```
# Define colors
avg_temp_color = '#00A6A6'
humidity_color = '#FF6F61'
wind_speed_color = '#6A0572'
mean_pressure_color = '#8F9A9C'

# Plotting Average Temperature with the chosen color
dataset['meantemp'].plot(figsize=(16, 5), color=avg_temp_color)
plt.title("Average Temperature")
plt.show()

# Plotting Humidity with the chosen color
dataset['humidity'].plot(figsize=(16, 5), color=humidity_color)
plt.title("Humidity")
plt.show()

# Plotting Wind Speed with the chosen color
dataset['wind_speed'].plot(figsize=(16, 5), color=wind_speed_color)
plt.title("Wind Speed")
plt.show()

# Plotting Average Pressure with the chosen color
dataset['meanpressure'].plot(figsize=(16, 5), color=mean_pressure_color)
plt.title("Average Pressure")
plt.show()
```


Average Temperature
40
35
30
25
20
15
10
5
2013
2014
2015
2016
Date
Humidity
100
80
60
40

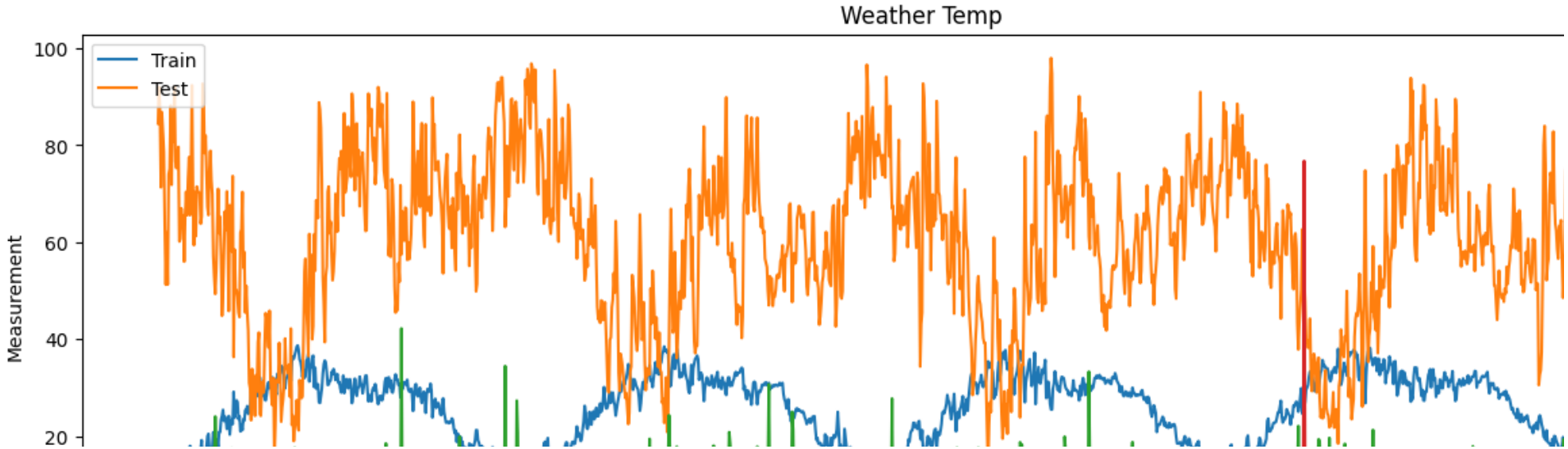

20
2013
2014
2015
2016
Date


Wind Speed
40
30
20
10
0
2013
2014
2015
2016
Date


Average Pressure
80
70
60
50


```
# Define the start and end years for the training and testing split
tstart = 2013
tend = 2016

# Function to split the dataset into training and testing sets based on date range
def train_test_split(dataset, tstart, tend):
    # Select data from 2013 to 2016 for training set
    train = dataset.loc[f"{tstart}":f"{tend}"]
    # Select data from 2017 onwards for testing set
    test = dataset.loc[f"{tend+1}":]
    return train, test

# Split the dataset into training and testing sets
training_set, test_set = train_test_split(dataset, tstart, tend)


plt.figure(figsize=(16, 5))
plt.plot(training_set)
plt.plot(test_set)
plt.title('Weather Temp')
plt.ylabel('Measurement')
plt.xlabel('Time')
plt.legend(['Train', 'Test'], loc='upper left')
```

```
<matplotlib.legend.Legend at 0x7903c836a350>
```




```
# Get the shape of the training set and test set
training_set_shape = training_set.shape
test_set_shape = test_set.shape
print('training_set shape:', training_set_shape)
print('test_set shape:', test_set_shape)
```

```
training_set shape: (1461, 4)
test_set shape: (114, 4)
```

```
# Create a MinMaxScaler object with a specified feature range (0 to 1)
sc = MinMaxScaler(feature_range=(0, 1))
training_set = training_set.values.reshape(-1, 1)
training_set_scaled = sc.fit_transform(training_set)
print('training_set_scaled shape after scaling:', training_set_scaled.shape)
```

```
training_set_scaled shape after scaling: (5844, 1)
```

```
# Transform back to table format with four features
training_set_scaled = training_set_scaled.reshape(training_set_shape[0], training_set_shape[1])
print('training_set_scaled shape:', training_set_scaled.shape)
```

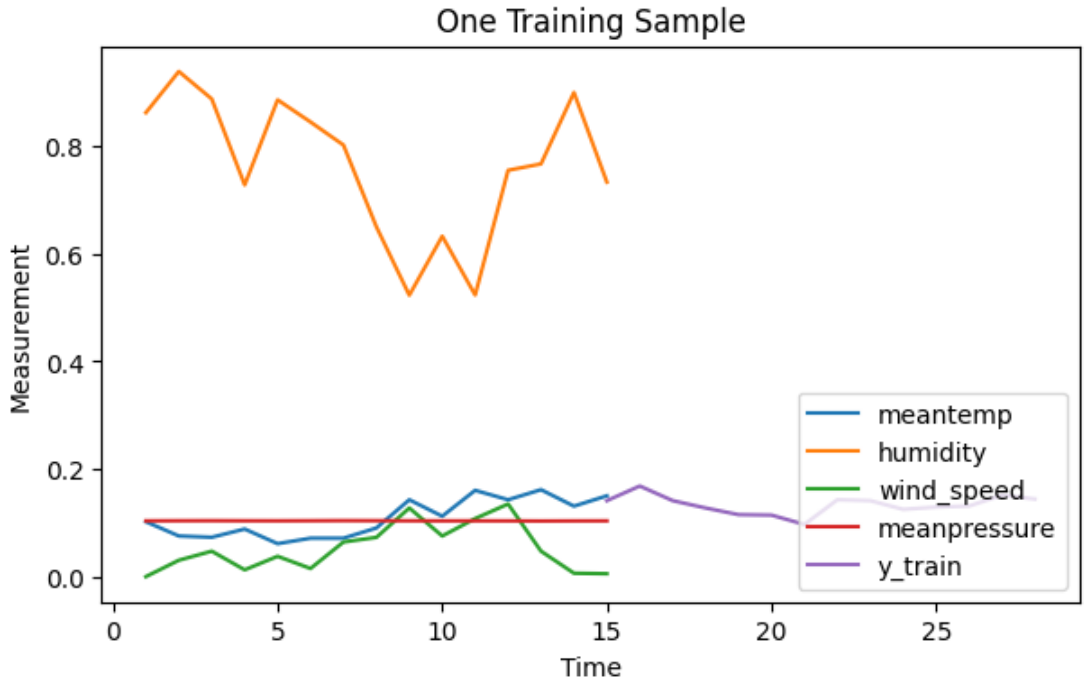
```
     training_set_scaled shape: (1461, 4)
```

```
#Split a time series sequence into input (X) and output (y) samples for a forecasting

def split_sequence(sequence, n_steps,forecasting_horizon, y_index):
    X, y = list(), list()
    for i in range(len(sequence)):
        end_ix = i + n_steps
        if end_ix > len(sequence) - forecasting_horizon:
            break
        seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:end_ix+forecasting_horizon,y_index]
        X.append(seq_x)
        y.append(seq_y)
    return np.array(X), np.array(y)


# Defining model parameters

n_steps = 15 # Models performed best at n_steps = 15 out of 10,15,20,25,40,50,75
forecasting_horizon = 14 # Days to forecast
features = 4 # No of features
y_index = 0 # Index of temp column

# split into samples
X_train, y_train = split_sequence(training_set_scaled, n_steps,forecasting_horizon,y_index)


# Reshaping X_train for model
y_train  = y_train.reshape(y_train.shape[0],y_train.shape[1],1)


print('X_train shape:', X_train.shape)
print('y_train shape:', y_train.shape)
```

```
     X_train shape: (1433, 15, 4)
     y_train shape: (1433, 14, 1)
```

```
# Plotting a single training sample with multiple measurements over time

plt.figure(figsize=(7, 4))
plt.plot(np.arange(1, n_steps+1, 1), X_train[0, :, 0], label='meantemp')
```

```
plt.plot(np.arange(1, n_steps+1, 1), X_train[0, :, 1], label='humidity')
plt.plot(np.arange(1, n_steps+1, 1), X_train[0, :, 2], label='wind_speed')
plt.plot(np.arange(1, n_steps+1, 1), X_train[0, :, 3], label='meanpressure')
plt.plot(np.arange(n_steps, n_steps+forecasting_horizon, 1), y_train[0, :, 0], label='y_train')
plt.title('One Training Sample')
plt.ylabel('Measurement')
plt.xlabel('Time')
plt.legend(loc='lower right')
plt.show()
```



3. Predictive Modeling

```
from tensorflow.keras.layers import LSTM, Dense
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import RMSprop, Nadam, Adam
from sklearn.metrics import mean_squared_error
```

Base Model

The model architecture is a simple LSTM (Long Short-Term Memory) network with one hidden layer. The first layer is an LSTM layer with 100 units. The LSTM layer is a type of recurrent neural network that is well-suited for time series data. It can learn long-term dependencies between data points, which is important for forecasting tasks. The second layer is a Dense layer with 14 units. The Dense layer is a fully connected layer that outputs the predictions.

```
###### Base Model ######

# The LSTM architecture
model_lstm = Sequential()
model_lstm.add(LSTM(units=100, activation="tanh", input_shape=(n_steps, features)))
model_lstm.add(Dense(units=forecasting_horizon))

# Compiling the model
model_lstm.compile(optimizer="RMSprop", loss="mse")

model_lstm.summary()

model_lstm.fit(X_train, y_train, epochs=100, batch_size=32)
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 100)               42000

 dense (Dense)               (None, 14)                1414

=================================================================
Total params: 43,414
Trainable params: 43,414
Non-trainable params: 0
_________________________________________________________________
Epoch 1/100
45/45 [==============================] - 3s 13ms/step - loss: 0.0091
Epoch 2/100
45/45 [==============================] - 1s 12ms/step - loss: 0.0019
Epoch 3/100
```

```
45/45 [==============================] - 1s 13ms/step - loss: 0.0014
Epoch 4/100
45/45 [==============================] - 1s 13ms/step - loss: 0.0014
Epoch 5/100
45/45 [==============================] - 1s 12ms/step - loss: 0.0013
Epoch 6/100
45/45 [==============================] - 1s 12ms/step - loss: 0.0012
Epoch 7/100
45/45 [==============================] - 1s 13ms/step - loss: 0.0012
Epoch 8/100
45/45 [==============================] - 1s 12ms/step - loss: 0.0011
Epoch 9/100
45/45 [==============================] - 1s 14ms/step - loss: 0.0010
Epoch 10/100
45/45 [==============================] - 1s 21ms/step - loss: 0.0010
Epoch 11/100
45/45 [==============================] - 1s 20ms/step - loss: 0.0011
Epoch 12/100
45/45 [==============================] - 1s 18ms/step - loss: 0.0010
Epoch 13/100
45/45 [==============================] - 1s 14ms/step - loss: 9.4110e-04
Epoch 14/100
45/45 [==============================] - 1s 13ms/step - loss: 0.0010
Epoch 15/100
45/45 [==============================] - 1s 14ms/step - loss: 9.9098e-04
Epoch 16/100
45/45 [==============================] - 1s 14ms/step - loss: 9.8390e-04
Epoch 17/100
45/45 [==============================] - 1s 14ms/step - loss: 9.3576e-04
Epoch 18/100
45/45 [==============================] - 1s 14ms/step - loss: 9.3951e-04
Epoch 19/100
45/45 [==============================] - 1s 13ms/step - loss: 9.2962e-04
Epoch 20/100
45/45 [==============================] - 1s 12ms/step - loss: 9.4990e-04
Epoch 21/100
45/45 [==============================] - 1s 12ms/step - loss: 9.0033e-04
Epoch 22/100
45/45 [==============================] - 1s 12ms/step - loss: 8.6476e-04
Epoch 23/100
```

```
#scaling
inputs = test_set.values.reshape(-1, 1)
inputs = sc.transform(inputs)
```

```
#Reshape back to orignal format after rescaling
inputs = inputs.reshape(test_set_shape[0], test_set_shape[1])

# Split into samples
X_test, y_test = split_sequence(inputs, n_steps,forecasting_horizon,y_index)
number_test_samples = X_test.shape[0]
print('X_test shape:', X_test.shape)
print('y_test shape:', y_test.shape)
```

```
X_test shape: (86, 15, 4)
y_test shape: (86, 14)
```

```
#prediction
predicted_temp = model_lstm.predict(X_test)

#inverse transform the predicted values
predicted_temp = sc.inverse_transform(predicted_temp)
print('predicted_temp shape: ', predicted_temp.shape)

#inverse transform the test labels.
y_test = y_test.reshape(y_test.shape[0], y_test.shape[1])
y_test = sc.inverse_transform(y_test)
```

```
3/3 [==============================] - 0s 6ms/step
predicted_temp shape:  (86, 14)
```

Optimization 1

This optimization only varies from base model in terms of optimizer, in order to learn the impact of changing optimizer. The activation function will affect the non-linearity of the model and the smoothness of the predictions. The optimizer will affect the speed and accuracy of the training process. The loss function will affect the way the model is evaluated and how it is penalized for making mistakes. In general, a larger model with more units and a more complex activation function will be able to learn more complex patterns in the data and make more accurate predictions. However, a larger model will also take longer to train and may be more prone to overfitting.

```
###### Optimization 1 ######

# Define the LSTM architecture
```

```
model1 = Sequential()
model1.add(LSTM(units=100, activation="tanh", input_shape=(n_steps, features))) #Experimented units = 50,100,150
model1.add(Dense(units=14))

# Compile the model
model1.compile(optimizer='Nadam', loss="mse") #Experimented optimizer = Adam, Nadam

# Print the model summary
model1.summary()

model1.fit(X_train, y_train, epochs=100, batch_size=32)
#Experimented epochs = 25, 50, 75, 100; #Experimented batch_size = 16, 32, 64, 128
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_1 (LSTM)               (None, 100)               42000

 dense_1 (Dense)             (None, 14)                1414

=================================================================
Total params: 43,414
Trainable params: 43,414
Non-trainable params: 0
_________________________________________________________________
Epoch 1/100
45/45 [==============================] - 3s 14ms/step - loss: 0.0148
Epoch 2/100
45/45 [==============================] - 1s 14ms/step - loss: 0.0010
Epoch 3/100
45/45 [==============================] - 1s 12ms/step - loss: 8.9716e-04
Epoch 4/100
45/45 [==============================] - 1s 14ms/step - loss: 8.6576e-04
Epoch 5/100
45/45 [==============================] - 1s 14ms/step - loss: 8.5573e-04
Epoch 6/100
45/45 [==============================] - 1s 12ms/step - loss: 8.4407e-04
Epoch 7/100
45/45 [==============================] - 1s 13ms/step - loss: 8.3023e-04
Epoch 8/100
45/45 [==============================] - 1s 12ms/step - loss: 8.0814e-04
Epoch 9/100
45/45 [==============================] - 1s 12ms/step - loss: 8.0856e-04
```

```
Epoch 10/100
45/45 [==============================] - 1s 12ms/step - loss: 8.0910e-04
Epoch 11/100
45/45 [==============================] - 1s 23ms/step - loss: 7.9976e-04
Epoch 12/100
45/45 [==============================] - 1s 20ms/step - loss: 7.8089e-04
Epoch 13/100
45/45 [==============================] - 1s 20ms/step - loss: 7.8150e-04
Epoch 14/100
45/45 [==============================] - 1s 14ms/step - loss: 7.9861e-04
Epoch 15/100
45/45 [==============================] - 1s 14ms/step - loss: 7.8421e-04
Epoch 16/100
45/45 [==============================] - 1s 13ms/step - loss: 7.7033e-04
Epoch 17/100
45/45 [==============================] - 1s 13ms/step - loss: 7.6540e-04
Epoch 18/100
45/45 [==============================] - 1s 13ms/step - loss: 7.6544e-04
Epoch 19/100
45/45 [==============================] - 1s 13ms/step - loss: 7.6342e-04
Epoch 20/100
45/45 [==============================] - 1s 12ms/step - loss: 7.5786e-04
Epoch 21/100
45/45 [==============================] - 1s 13ms/step - loss: 7.4680e-04
Epoch 22/100
45/45 [==============================] - 1s 14ms/step - loss: 7.4398e-04
```

```
# scaling
inputs1 = test_set.values.reshape(-1, 1)
inputs1 = sc.transform(inputs1)

#Reshape back to orignal format after rescaling
inputs1 = inputs1.reshape(test_set_shape[0], test_set_shape[1])

# Split into samples
X_test, y_test = split_sequence(inputs, n_steps,forecasting_horizon,y_index)
number_test_samples = X_test.shape[0]
print('X_test shape:', X_test.shape)
print('y_test shape:', y_test.shape)
```

```
X_test shape: (86, 15, 4)
y_test shape: (86, 14)
```

```
#prediction
predicted_temp1 = model1.predict(X_test)

#inverse transform the predicted values
predicted_temp1 = sc.inverse_transform(predicted_temp1)
print('predicted_temp shape1: ', predicted_temp1.shape)

#inverse transform the test labels.
y_test = y_test.reshape(y_test.shape[0], y_test.shape[1])
y_test = sc.inverse_transform(y_test)
```

```
3/3 [==============================] - 0s 7ms/step
predicted_temp shape1:  (86, 14)
```

Optimization 2

The following RNN model consists of bidirectional LSTM layers that allow the model to learn dependencies in both the forward and backward directions. This is important for time series data, as it allows the model to learn how past values of the temperature can affect future values. In addition, Dropout layers are used for regularization, which helps to prevent overfitting, which occurs when the model learns the training data too well and is not able to generalize to new data. Aside from dropout layers and Adam optimizer, this model boasts MSE loss function as it is a good choice for regression problems, where the goal is to predict a continuous value. It basically measures the average squared error between the predicted values and the actual values.

Early stopping is another technique that stops training the model when it stops improving on the validation data. This helps to prevent overfitting. Lastly, Learning rate reduction reduces the learning rate of the optimizer when the model stops improving on the validation data. This helps to prevent the model from overfitting and to improve the performance of the model.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, Bidirectional
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

# Define the LSTM architecture
model2 = Sequential()

# Bidirectional LSTM layers
```

```
model2.add(Bidirectional(LSTM(units=128, activation='tanh', return_sequences=True), input_shape=(n_steps, features)))
model2.add(Dropout(0.2))  # Adding dropout for regularization
model2.add(Bidirectional(LSTM(units=64, activation='tanh')))
model2.add(Dropout(0.2))

# Fully connected layers
model2.add(Dense(units=64, activation='relu'))
model2.add(Dense(units=32, activation='relu'))

# Output layer (predicting average temperature for 14 days)
model2.add(Dense(units=14))

# Compile the model
optimizer = Adam(learning_rate=0.005)
model2.compile(optimizer=optimizer, loss='mse')

# Implement early stopping and learning rate reduction on plateau
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5, min_lr=1e-6)

# Print the model summary
model2.summary()


# Train the new model using your dataset
model2.fit(X_train, y_train, epochs=100, batch_size=32, callbacks=[early_stopping, reduce_lr])
```

```
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_loss` which is not available. Available metrics are: loss,
45/45 [==============================] - 3s 71ms/step - loss: 4.8433e-04 - lr: 0.0050
Epoch 91/100
45/45 [==============================] - ETA: 0s - loss: 5.2210e-04WARNING:tensorflow:Early stopping conditioned on metric `val_los
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_loss` which is not available. Available metrics are: loss,
45/45 [==============================] - 3s 77ms/step - loss: 5.2210e-04 - lr: 0.0050
Epoch 92/100
45/45 [==============================] - ETA: 0s - loss: 4.7294e-04WARNING:tensorflow:Early stopping conditioned on metric `val_los
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_loss` which is not available. Available metrics are: loss,
45/45 [==============================] - 5s 102ms/step - loss: 4.7294e-04 - lr: 0.0050
Epoch 93/100
45/45 [==============================] - ETA: 0s - loss: 5.8207e-04WARNING:tensorflow:Early stopping conditioned on metric `val_los
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_loss` which is not available. Available metrics are: loss,
45/45 [==============================] - 3s 74ms/step - loss: 5.8207e-04 - lr: 0.0050
Epoch 94/100
45/45 [==============================] - ETA: 0s - loss: 4.7694e-04WARNING:tensorflow:Early stopping conditioned on metric `val_los
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_loss` which is not available. Available metrics are: loss,
45/45 [==============================] - 3s 77ms/step - loss: 4.7694e-04 - lr: 0.0050
Epoch 95/100
45/45 [==============================] - ETA: 0s - loss: 5.0771e-04WARNING:tensorflow:Early stopping conditioned on metric `val_los
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_loss` which is not available. Available metrics are: loss,
45/45 [==============================] - 5s 103ms/step - loss: 5.0771e-04 - lr: 0.0050
Epoch 96/100
45/45 [==============================] - ETA: 0s - loss: 4.7825e-04WARNING:tensorflow:Early stopping conditioned on metric `val_los
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_loss` which is not available. Available metrics are: loss,
45/45 [==============================] - 3s 75ms/step - loss: 4.7825e-04 - lr: 0.0050
Epoch 97/100
45/45 [==============================] - ETA: 0s - loss: 4.4981e-04WARNING:tensorflow:Early stopping conditioned on metric `val_los
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_loss` which is not available. Available metrics are: loss,
45/45 [==============================] - 4s 78ms/step - loss: 4.4981e-04 - lr: 0.0050
Epoch 98/100
45/45 [==============================] - ETA: 0s - loss: 4.5719e-04WARNING:tensorflow:Early stopping conditioned on metric `val_los
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_loss` which is not available. Available metrics are: loss,
45/45 [==============================] - 7s 167ms/step - loss: 4.5719e-04 - lr: 0.0050
Epoch 99/100
45/45 [==============================] - ETA: 0s - loss: 4.6495e-04WARNING:tensorflow:Early stopping conditioned on metric `val_los
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_loss` which is not available. Available metrics are: loss,
45/45 [==============================] - 4s 95ms/step - loss: 4.6495e-04 - lr: 0.0050
Epoch 100/100
45/45 [==============================] - ETA: 0s - loss: 4.5663e-04WARNING:tensorflow:Early stopping conditioned on metric `val_los
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_loss` which is not available. Available metrics are: loss,
45/45 [==============================] - 3s 75ms/step - loss: 4.5663e-04 - lr: 0.0050
<keras.callbacks.History at 0x7903bde5d480>
```

```
# scaling
inputs2 = test_set.values.reshape(-1, 1)
inputs2 = sc.transform(inputs2)

#Reshape back to orignal format after rescaling
inputs2 = inputs2.reshape(test_set_shape[0], test_set_shape[1])

# Split into samples
X_test, y_test = split_sequence(inputs2, n_steps,forecasting_horizon,y_index)
number_test_samples = X_test.shape[0]
print('X_test shape:', X_test.shape)
print('y_test shape:', y_test.shape)
```

```
X_test shape: (86, 15, 4)
y_test shape: (86, 14)
```

```
#prediction
predicted_temp2 = model2.predict(X_test)

#inverse transform the predicted values
predicted_temp2 = sc.inverse_transform(predicted_temp2)
print('predicted_temp shape1: ', predicted_temp2.shape)

#inverse transform the test labels.
y_test = y_test.reshape(y_test.shape[0], y_test.shape[1])
y_test = sc.inverse_transform(y_test)
```

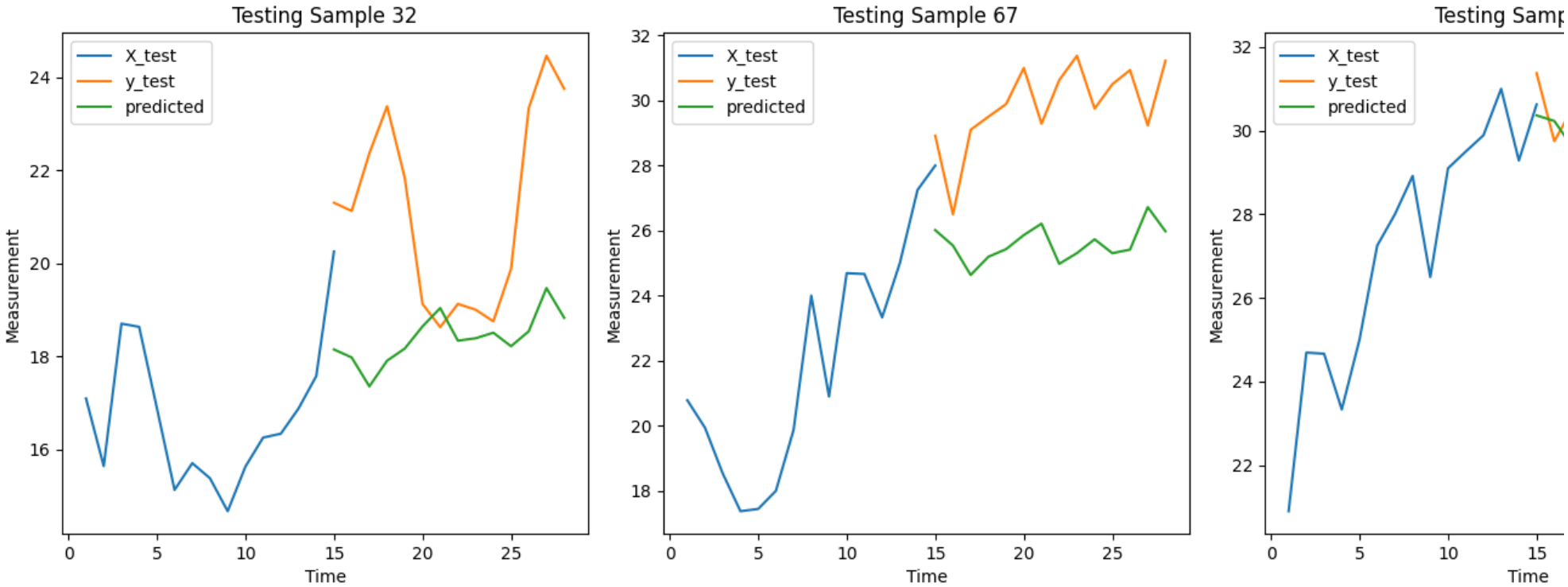
```
3/3 [==============================] - 2s 36ms/step
predicted_temp shape1:  (86, 14)
```

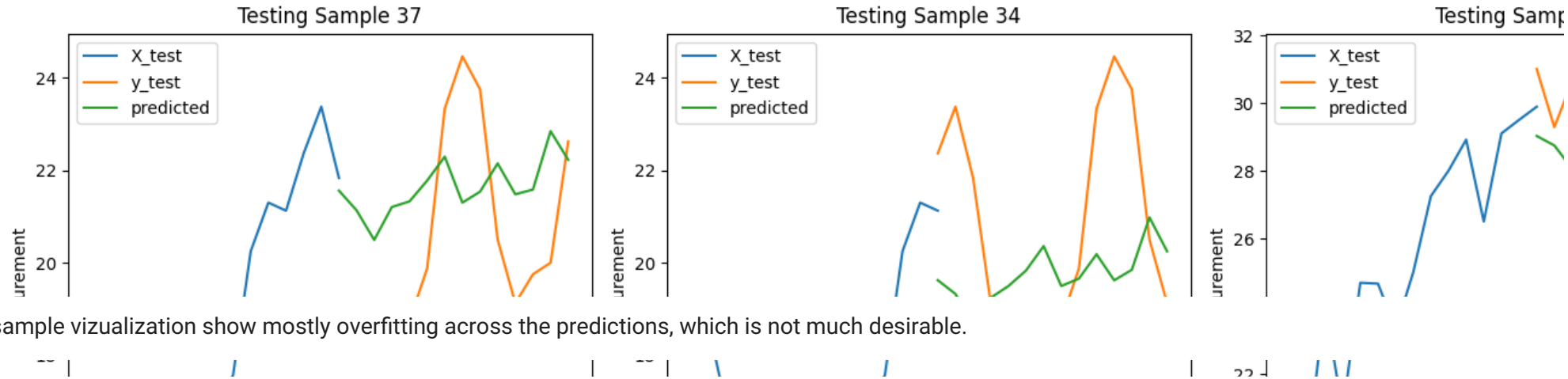
4. Experiments Report

Base Model

```
X_test = X_test.reshape(-1, 15) # Reshape the test data to match the original shape
X_test = sc.inverse_transform(X_test) # Inverse transform the scaled test data to its original values
X_test = X_test.reshape(number_test_samples, 15, 4) # Reshape the test data to its original format, considering the number of test sampl
```

```
# Number of random samples to display
num_samples = 6

# Create a grid for plotting
plt.figure(figsize=(15, 10))

# Iterate through the selected random samples
for i in range(num_samples):
    plt.subplot(2, 3, i+1)  # Create a subplot in the 2x3 grid
    sample_index = np.random.randint(0, len(X_test))  # Randomly select a sample index

    plt.plot(np.arange(1, n_steps+1, 1), X_test[sample_index, :, 0])
    plt.plot(np.arange(n_steps, n_steps+forecasting_horizon, 1), y_test[sample_index, :])
    plt.plot(np.arange(n_steps, n_steps+forecasting_horizon, 1), predicted_temp[sample_index, :])

    plt.title('Testing Sample ' + str(sample_index))
    plt.ylabel('Measurement')
    plt.xlabel('Time')
    plt.legend(['X_test', 'y_test', 'predicted'], loc='upper left')

# Adjust layout for better spacing
plt.tight_layout()
plt.show()
```

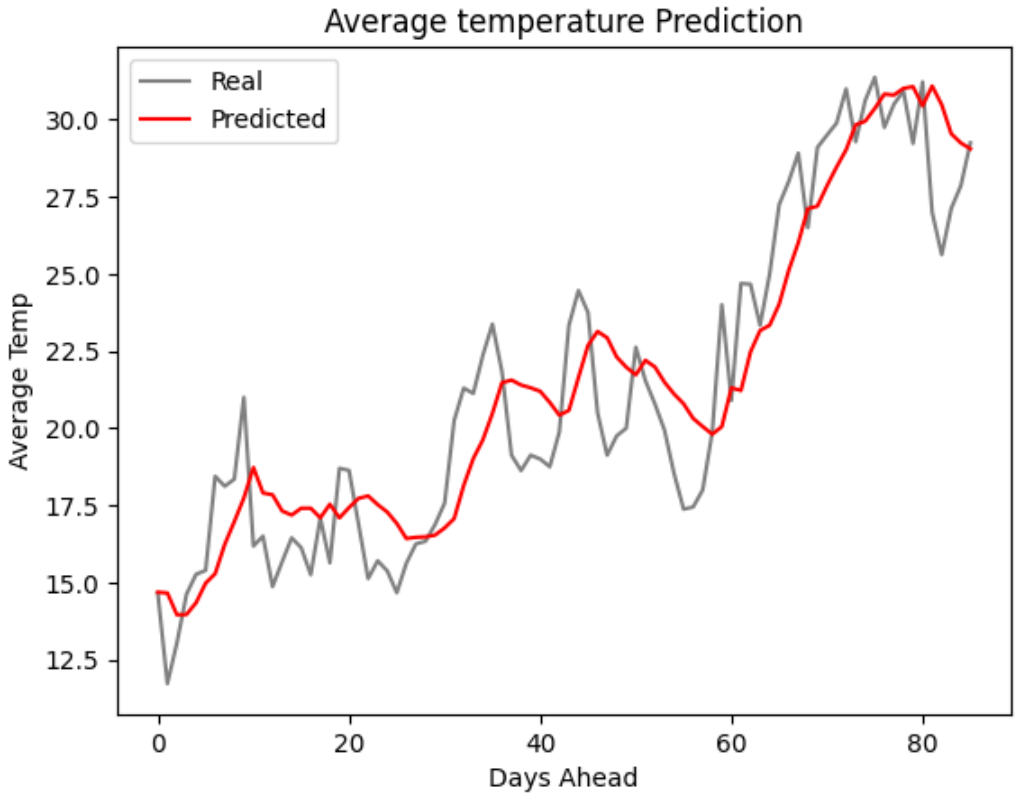




The sample vizualization show mostly overfitting across the predictions, which is not much desirable.

```
#Visualize the real vs. predicted values for one day forecasting horizon.

def plot_predictions(test, predicted):
    plt.plot(test[:], color="gray", label="Real")
    plt.plot(predicted[:], color="red", label="Predicted")
```

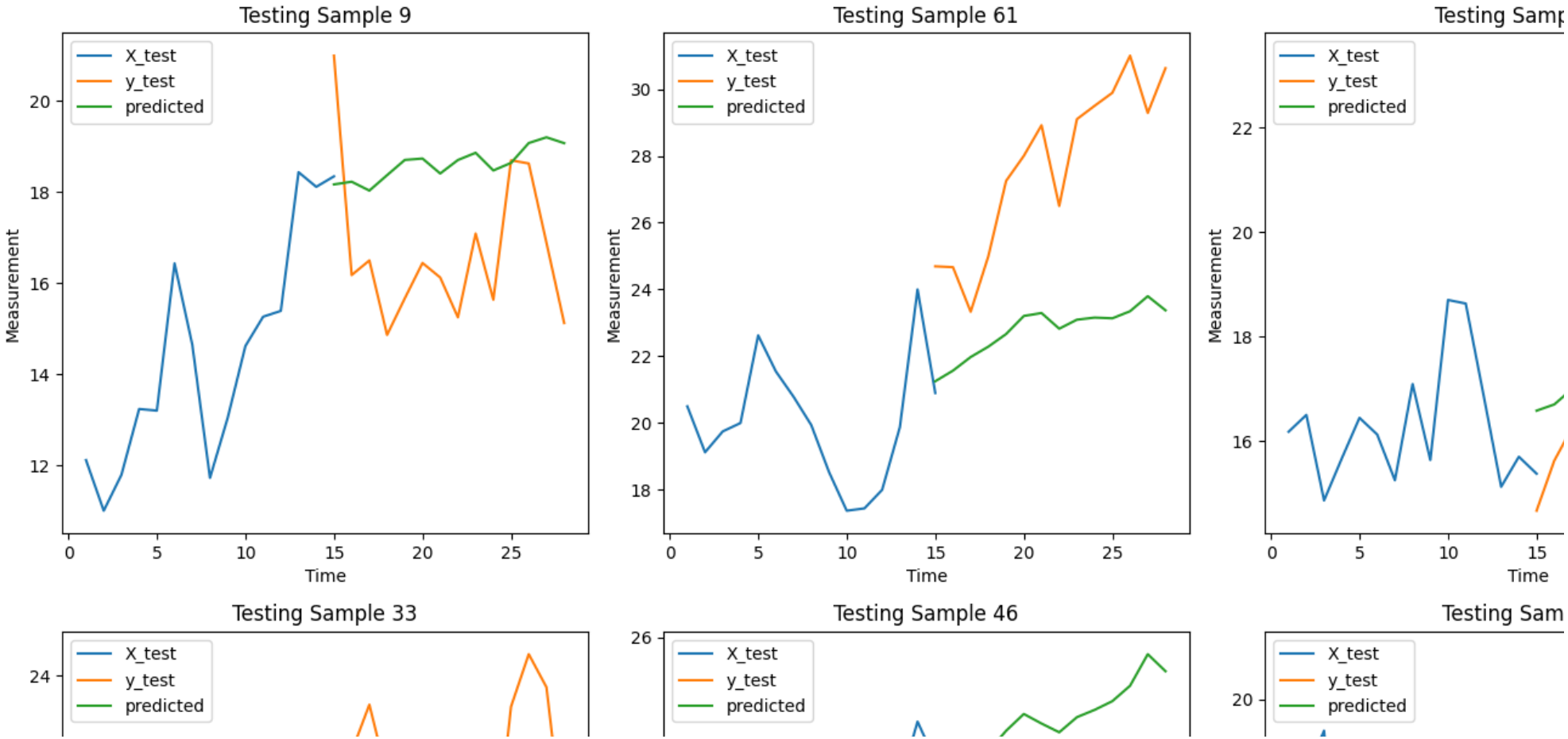
```
    plt.title("Average temperature Prediction")
    plt.xlabel("Days Ahead")
    plt.ylabel("Average Temp")
    plt.legend()
    plt.show()

plot_predictions(y_test[:,0],predicted_temp[:,0])
```



Optimization 1

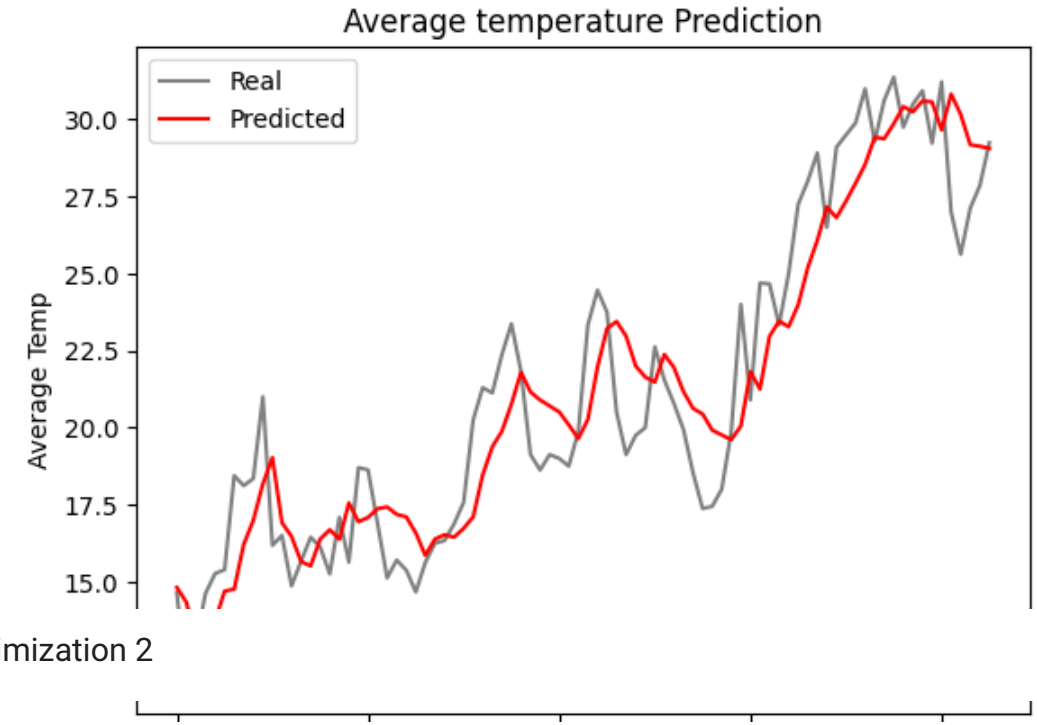
```
# Number of random samples to display
num_samples = 6

# Create a grid for plotting
plt.figure(figsize=(15, 10))
```

```
# Iterate through the selected random samples
for i in range(num_samples):
    plt.subplot(2, 3, i+1)  # Create a subplot in the 2x3 grid
    sample_index = np.random.randint(0, len(X_test))  # Randomly select a sample index

    plt.plot(np.arange(1, n_steps+1, 1), X_test[sample_index, :, 0])
    plt.plot(np.arange(n_steps, n_steps+forecasting_horizon, 1), y_test[sample_index, :])
    plt.plot(np.arange(n_steps, n_steps+forecasting_horizon, 1), predicted_temp1[sample_index, :])

    plt.title('Testing Sample ' + str(sample_index))
    plt.ylabel('Measurement')
    plt.xlabel('Time')
    plt.legend(['X_test', 'y_test', 'predicted'], loc='upper left')

# Adjust layout for better spacing
plt.tight_layout()
plt.show()
```



Compared to base model, the Nadam optimizer worked well to overcome the overfitting to an extent. It can also be noted that model 1 is unable to predict small variations very well. Reducing the n_steps in the model architecture may help but it may also lead to excessive underfitting.

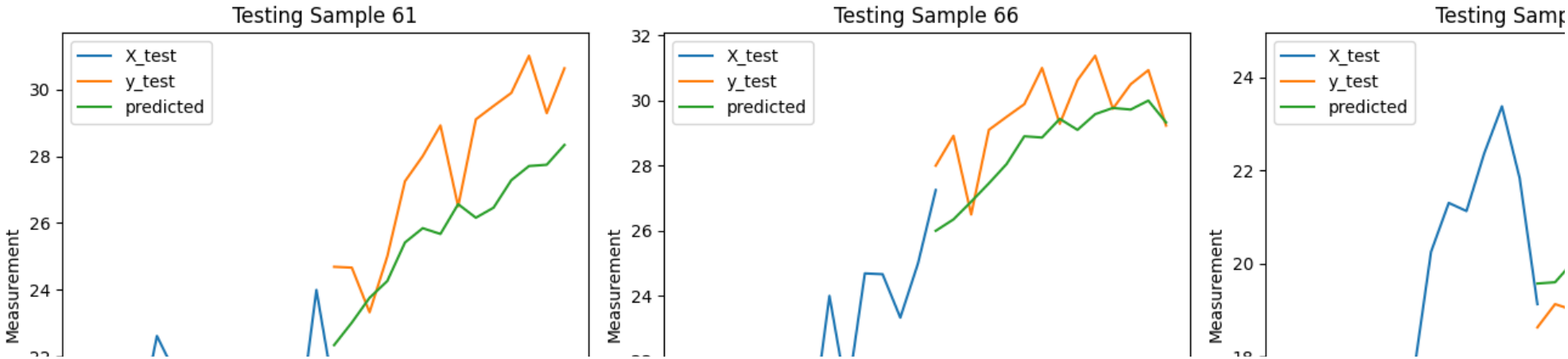
```
#Visualize the real vs. predicted values for one day forecasting horizon.
plot_predictions(y_test[:,0],predicted_temp1[:,0])
```



Optimization 2

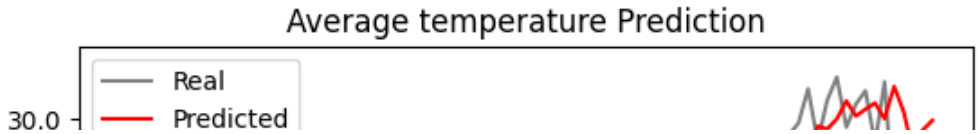
```
# Number of random samples to display
num_samples = 6

# Create a grid for plotting
plt.figure(figsize=(15, 10))

# Iterate through the selected random samples
for i in range(num_samples):
    plt.subplot(2, 3, i+1)  # Create a subplot in the 2x3 grid
    sample_index = np.random.randint(0, len(X_test))  # Randomly select a sample index

    plt.plot(np.arange(1, n_steps+1, 1), X_test[sample_index, :, 0])
    plt.plot(np.arange(n_steps, n_steps+forecasting_horizon, 1), y_test[sample_index, :])
    plt.plot(np.arange(n_steps, n_steps+forecasting_horizon, 1), predicted_temp2[sample_index, :])

    plt.title('Testing Sample ' + str(sample_index))
    plt.ylabel('Measurement')
    plt.xlabel('Time')
    plt.legend(['X_test', 'y_test', 'predicted'], loc='upper left')
```

```
# Adjust layout for better spacing
plt.tight_layout()
plt.show()
```



Lastly, model 2 shows more overlap of predicted and true values. Sample vizualizations show that model 2 is able to adjust to variations more effectively. This might be beneficial when capturing intricate patterns and longer-term dependencies.

```
#Visualize the real vs. predicted values for one day forecasting horizon.
plot_predictions(y_test[:,0],predicted_temp2[:,0])
```

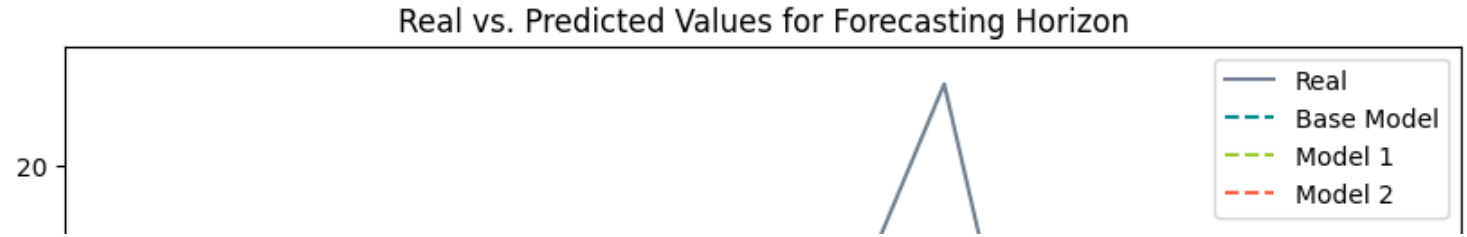


Model Comparision

```
# Initialize an empty DataFrame to store MAE values
mae_data = pd.DataFrame(columns=['Base_Model', 'Model_1', 'Model_2'])

# Define function to calculate and add MAE values for each model
def calculate_mae(model_name, predicted_values):
    mae_values = []
    for i in range(forecasting_horizon):
        mae = mean_absolute_error(y_test[:, i], predicted_values[:, i])
        mae_values.append(mae)
    return mae_values

# Calculate MAE values for each model
mae_data['Base_Model'] = calculate_mae('Base_Model', predicted_temp)
mae_data['Model_1'] = calculate_mae('Model_1', predicted_temp1)
mae_data['Model_2'] = calculate_mae('Model_2', predicted_temp2)

# Display the MAE data in tabular form
print(mae_data)
```

```
    Base_Model   Model_1   Model_2
0     1.768207  1.641999  1.602404
1     2.125575  1.973553  1.872340
2     2.336408  2.242859  2.095759
3     2.435620  2.436687  2.226960
4     2.550534  2.596058  2.351169
5     2.730117  2.686864  2.570334
6     2.861412  2.701731  2.509813
7     2.834634  2.697031  2.622390
8     2.902344  2.719539  2.592746
9     2.930264  2.675325  2.796025
10    2.972303  2.595309  2.822326
11    2.978538  2.674333  2.827793
12    2.974931  2.695398  2.672041
13    2.951970  2.681112  2.563115
```

```
# Create a single chart for real vs. predicted values
plt.figure(figsize=(10, 6))

# Plot real values
plt.plot(y_test[:15, 0], color="slategrey", label="Real") #forecasting_horizon + 1 to visualize complete forecast

# Plot predicted values of each model
plt.plot(predicted_temp[:15, 0], color="darkcyan",linestyle='dashed', label="Base Model")
plt.plot(predicted_temp1[:15, 0], color="yellowgreen", linestyle='dashed', label="Model 1")
plt.plot(predicted_temp2[:15, 0], color="tomato", linestyle='dashed', label="Model 2")

plt.title("Real vs. Predicted Values for Forecasting Horizon")
plt.xlabel("Time Steps")
plt.ylabel("Temperature")
plt.legend()
plt.show()
```



- **Base Model:**

The Base Model demonstrates a consistent MAE across the forecasting horizon, with values ranging from approximately 1.77 to 2.97. It exhibits a relatively stable performance with minimal fluctuations in MAE over the 14-day period. While the Base Model provides decent forecasting results, there may be room for improvement in capturing more complex patterns in the data.

- **Model 1:**

Model 1 shows lower MAE values compared to the Base Model across the forecasting horizon. It achieves a more accurate forecast, with MAE values ranging from approximately 1.64 to 2.72. Model 1 demonstrates a steady improvement in performance as the forecasting horizon progresses, indicating its ability to capture longer-term dependencies.

- **Model 2:**

Model 2 consistently outperforms both the Base Model and Model 1 in terms of MAE. It achieves the lowest MAE values, ranging from approximately 1.60 to 2.82. Model 2 maintains its accuracy throughout the 14-day forecasting horizon and consistently provides the most accurate predictions.

The choice between these models depends on the specific forecasting requirements and the trade-off between complexity and accuracy. Model 1, with the Nadam optimizer, offers a reasonable balance between simplicity and accuracy, making it a suitable choice for forecasting tasks where moderate complexity suffices. On the other hand, Model 2, with its more elaborate architecture, might be beneficial when capturing intricate patterns and longer-term dependencies is critical, even if it comes at the cost of slightly varying performance.

Model Refinement

- Use transfer learning: Transfer learning can be used to improve the performance of the model. This involves training the model on a related task, such as forecasting the weather in a different location. This can help the model to learn more general patterns in the data, which can lead to better predictions.

- Use a wider variety of data: In addition to historical weather data, the model can also be trained on other data, such as satellite imagery, land cover data, and climate models. This can help the model to learn more about the underlying physical processes that drive the weather.
- Use a more complex model: LSTM models are a good starting point for weather forecasting, but they can be improved by using more complex models, such as deep learning models. These models can learn more complex patterns in the data, which can lead to more accurate predictions.

Model use in business

- Improved decision-making: Businesses can use accurate weather forecasts to make better decisions about planning, scheduling, and resource allocation. For example, a transportation company can use weather forecasts to plan its routes and schedules, or a farmer can use weather forecasts to decide when to plant and harvest crops.
- Reduced risk: Businesses can use weather forecasts to reduce their risk of exposure to weather-related hazards. For example, a power company can use weather forecasts to plan for power outages, or a construction company can use weather forecasts to reschedule projects.
- Increased efficiency: Businesses can use weather forecasts to improve their efficiency. For example, a retailer can use weather forecasts to adjust its inventory levels, or a hotel can use weather forecasts to adjust its pricing.

References

- Balamurugan, S 2021, 'Weather forecasting using LSTM', Google Colaboratory Notebook, accessed 30 August 2023, https://colab.research.google.com/github/balams81/Rain/blob/master/Weather_forecasting_LSTM.ipynb.
- Kozlov, A, Kozlova, E & Kozlov, S 2021, 'Method of bidirectional LSTM modelling for the atmospheric temperature', International Journal of Energy and Environmental Engineering, vol. 12, no. 2, pp. 153-161, accessed 30 August 2023 https://www.researchgate.net/publication/353882505_Method_of_Bidirectional_LSTM_Modelling_for_the_Atmospheric_Temperature
- Nazare, E 2021, 'Building a LSTM model for weather prediction using ChatGPT to assist', Medium, accessed 30 August 2023, https://medium.com/@eldanazare/building-a-lstm-model-for-weather-prediction-using-chatgpt-to-assist-e5294d6ff40d#:~:text=They can capture trends%2C seasonal,used for time series forecasting.

- Paperspace 2019, ‘Weather Forecast using LSTMs’, Paperspace Blog, accessed 30 August 2023, https://blog.paperspace.com/weather-forecast-using-ltsm-networks/.
- Zhang, Y & Wang, J 2021, ‘A novel deep learning model for weather forecasting based on LSTM and attention mechanism’, IEEE Access, vol. 9, pp. 9824268-9824277, accessed 30 August 2023, doi: 10.1109/ACCESS.2021.3073570.