**Appendix**

**Code**

**Python Code**

import numpy as np # linear algebra

import pandas as pd # data processing, CSV file I/O (e.g. pd.read\_csv)

# visualization

import seaborn as sns

import matplotlib.pyplot as plt

%matplotlib inline

# machine learning

import seaborn as sns

from sklearn.linear\_model import LinearRegression, Ridge, Lasso

from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor

from sklearn.svm import SVR

from sklearn.neighbors import KNeighborsRegressor

from sklearn.impute import SimpleImputer

imputer=SimpleImputer(missing\_values=np.nan,strategy='median')

from sklearn.ensemble import BaggingRegressor

import xgboost

from numpy import nan

from numpy import isnan

from sklearn.metrics import mean\_squared\_error

# Load data from a CSV file

data = pd.read\_excel("Transactions\_residential\_flats 1BR\_final.xlsx")

# Perform data cleaning steps

df = data.dropna() # Remove missing values

df = data.drop\_duplicates() # Remove duplicate entries

pd.set\_option('float\_format', '{:f}'.format) #to see the real numbers

df.describe()

#Find if there are Null values in the data frame

df.isnull().sum()

#drop the null values from the data frame and call it upd\_df

# null values are dropped as it will not impact the results

upd\_df = df.dropna()

#Find if there are Null values in the data frame

upd\_df.isnull().sum()

#### Convert the categories in to dummies

upd\_df['trans\_group'] = upd\_df['trans\_group'].astype('category')

upd\_df.dtypes

upd\_df['trans\_group'] = upd\_df['trans\_group'].cat.codes

upd\_df['procedure'] = upd\_df['procedure'].astype('category')

upd\_df['property\_type'] = upd\_df['property\_type'].astype('category')

upd\_df['reg\_type'] = upd\_df['reg\_type'].astype('category')

upd\_df['area\_name'] = upd\_df['area\_name'].astype('category')

upd\_df['procedure'] = upd\_df['procedure'].cat.codes

upd\_df['property\_type'] = upd\_df['property\_type'].cat.codes

upd\_df['reg\_type'] = upd\_df['reg\_type'].cat.codes

upd\_df['area\_name'] = upd\_df['area\_name'].cat.codes

## Exploratory Data Analysis (EDA)

EDA is like peering into a treasure chest before diving in.

It helps us understand the characteristics of our data and uncover interesting insights.

In this section, we’ll leverage Python’s powerful libraries, such as matplotlib and seaborn, to create stunning visualizations that bring the data to life.

# Visualize the distribution of housing Size (sq.m)

import matplotlib.pyplot as plt

import seaborn as sns

sns.histplot(upd\_df['price'])

plt.xlabel('Price')

plt.ylabel('Frequency')

plt.title('Distribution of Housing Price')

plt.show()

import pandas as pd

import matplotlib.pyplot as plt

# Convert date column to datetime type

upd\_df['date'] = pd.to\_datetime(upd\_df['date'])

# Set date column as the index

upd\_df.set\_index('date', inplace=True)

# Visualize the average housing price over time

upd\_df.resample('M')['price'].mean().plot()

plt.xlabel('Year')

plt.ylabel('Average Price in AED')

plt.title('Average Housing Price Over Time AED')

plt.show()

## Identifying Outliers

- using box plots to visualize each feature to identify if there are outliers

- using statistical method to identify the outliers

- count the number of outliers for each feature

- reveiw the statistical summary of the outliers

- drop outliers

### Box Plots

Using box plots to visualize each float64 feature in the dataframe. Created a 2 x 4 figure containing all 7 box plots. Lower and higher limit outliers will be identified.

#create box plots to visualize data and identify any outliers

#Create a figure with 2 rows with 4 boxplots each

fig, axs = plt.subplots(2, 2)

# create Price boxplot

axs[0, 0].boxplot(upd\_df['price'])

axs[0, 0].set\_title('Price')

# create Size boxplot

axs[0, 1].boxplot(upd\_df['area size'])

axs[0, 1].set\_title('Area size per square meters')

#adjust spacing for readability

fig.subplots\_adjust(left=0.08, right=1, bottom=0.05, top=1,

 hspace=0.5, wspace=0.5)

plt.show()

### Identifying outliers

- defined a function <code>find\_outliers\_IQR</code> which takes in a dataframe as an input and returns a dataframe as an output. The returned data frame contains the outliers as numerical values and others as NaN

- identified the outliers: lower limit < q1 - 1.5 \* IQR and higher limit > q3 + 1.5 \* IQR

def find\_outliers\_IQR(upd\_df):

 Q1 = upd\_df.quantile(0.25)

 Q3 = upd\_df.quantile(0.75)

 IQR = Q3 - Q1

 outliers = upd\_df[((upd\_df<(Q1 - 1.5\*IQR)) | (upd\_df>(Q3 + 1.5\*IQR)))]

 return outliers

#### Outliers

Print the outliers for the feature 'Global\_active\_power' to identify the number of outliers, the minimum and maximum values for the outliers and to give the top and bottom five outliers for the feature. Compare the values given with the box plot.

outliers = find\_outliers\_IQR(upd\_df['price'])

print('number of outliers: ' + str(outliers.count()))

print('min outlier value: ' + str(outliers.min()))

print('max outlier value: ' + str(outliers.max()))

outliers

#### Removing Outliers

# Define function to remove outliers using IQR

def remove\_outliers\_iqr(upd\_df, columns):

 for column in columns:

 Q1 = upd\_df[column].quantile(0.25)

 Q3 = upd\_df[column].quantile(0.75)

 IQR = Q3 - Q1

 lower\_bound = Q1 - 1.5 \* IQR

 upper\_bound = Q3 + 1.5 \* IQR

 upd\_df = upd\_df[(upd\_df[column] >= lower\_bound) & (upd\_df[column] <= upper\_bound)]

 return upd\_df

# Specify columns to remove outliers from

float\_cols = upd\_df.select\_dtypes(include=[np.float64]).columns

# Remove outliers from float64 columns using IQR

upd\_df\_cleaned = remove\_outliers\_iqr(upd\_df, float\_cols)

# Print cleaned DataFrame

#print(numerical\_subset\_cleaned).reset\_index()

upd\_df\_cleaned.reset\_index()

##'Distribution of Housing Price after removing outliers

upd\_df\_cleaned['price'].hist(bins = 50, grid = False, figsize = (10,5))

plt.title('Distribution of Housing Price')

plt.show()

#create box plots to visualize data and identify any outliers

#Create a figure with 2 rows with 4 boxplots each

fig, axs = plt.subplots(2, 2)

# create Price boxplot

axs[0, 0].boxplot(upd\_df\_cleaned['price'])

axs[0, 0].set\_title('Price')

# create Size boxplot

axs[0, 1].boxplot(upd\_df\_cleaned['area size'])

axs[0, 1].set\_title('Area size per square meters')

#adjust spacing for readability

fig.subplots\_adjust(left=0.08, right=1, bottom=0.05, top=1,

 hspace=0.5, wspace=0.5)

plt.show()

##log transformation

##create a list of column names

log\_df=upd\_df\_cleaned

col\_name = list(log\_df.columns)

col\_name

temp\_lst = ['price','area size']

for col\_name in temp\_lst:

 log\_df[col\_name] = np.log(log\_df[col\_name])

log\_df.hist(['price','area size'],figsize = (8,3), grid = False)

plt.show()

## Normalizing data

To normalize the data will use the Min - Max method: (x - Min) / (Max - Min)

## normalize the data using (x - Min) / (Max - Min)

##create a list of column names

col\_name = list(log\_df.columns)

col\_name

## create a temporary list of the columns we want to use

##create a loop to normalize the data for each feature in temporary list

temp\_lst = ['price','area size']

for col\_name in temp\_lst:

 log\_df[col\_name] = (log\_df[col\_name] - log\_df[col\_name].mean())/(log\_df[col\_name].std())

### Grouping Data by Date

Aggregate the data by day to find the average values per month for each feature.

# average feature per month

log\_df\_month = log\_df.groupby(['date'], as\_index = True).mean()

log\_df\_month

# Write DataFrame to an Excel file

log\_df\_month.to\_excel('D:/University/UTM/PhD Files/CH5/Data/dubai property\_grouped.xlsx')

### Correlations

Create a correlation heat map to see if there are relationships between any of the features in the dataframe. normalized data grouped by the day.

# Correlation matrix (grouping)

plt.figure(figsize=(8,5))

sns.heatmap(log\_df\_month.corr().round(2),annot=True)

plt.title('Correlation matrix of log\_df\_month',fontsize = 10)

plt.show()

#create line plots to visualize data

#Create a figure with 2 rows with 2 boxplots each

fig, axs = plt.subplots(2, 2)

# create Price plot

axs[0, 0].plot(log\_df\_month['price'])

axs[0, 0].set\_title('Housing Price by Date')

axs[0, 0].set\_ylabel('Housing Price')

axs[0, 0].set\_xlabel('Date')

# create Area Size plot

axs[1, 0].plot(log\_df\_month['area size'])

axs[1, 0].set\_title('Area Size by Date')

axs[1, 0].set\_ylabel('Area Size')

axs[1, 0].set\_xlabel('Date')

#adjust spacing for readability

fig.subplots\_adjust(left=0.08, right=2, bottom=0.05, top=1,

 hspace=0.5, wspace=0.5)

plt.show()

# prepare date for modelling by split into training and test sets

from sklearn.model\_selection import train\_test\_split

# Prepare the data

X = log\_df\_month.drop(['price'], axis = 1)

y = log\_df\_month['price']

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2)

# Scale the features using standardization

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

#import libraries for modeling and metrics calculations

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.linear\_model import LinearRegression

from sklearn.tree import DecisionTreeRegressor

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import r2\_score, mean\_squared\_error

from sklearn.pipeline import Pipeline

from sklearn.svm import SVR

from sklearn.neighbors import KNeighborsRegressor

from sklearn.ensemble import GradientBoostingRegressor

from sklearn.neural\_network import MLPRegressor

from sklearn.metrics import mean\_squared\_error

from sklearn.metrics import mean\_absolute\_error

# Train and evaluate linear regression

lr = LinearRegression()

lr.fit(X\_train, y\_train)

y\_pred\_lr = lr.predict(X\_test)

r2\_lr = r2\_score(y\_test, y\_pred\_lr)

mse\_lr = mean\_squared\_error(y\_test, y\_pred\_lr)

RMSE\_LR = math.sqrt(mse\_lr)

MAPE\_LR=mean\_absolute\_error(y\_test, y\_pred\_lr)\*100

print('Linear Regression R2 Score: ', round(r2\_lr,4))

print('Linear Regression Mean Squared Error: ', round(mse\_lr,4))

print('Linear Regression Root Mean Squared Error: ', round(RMSE\_LR,4))

print('Linear Regression Mean Absolute Percentage Error: ', round(MAPE\_LR,4))

# Train and evaluate decision tree regression

dt = DecisionTreeRegressor(random\_state=42)

dt.fit(X\_train, y\_train)

y\_pred\_dt = dt.predict(X\_test)

r2\_dt = r2\_score(y\_test, y\_pred\_dt)

mse\_dt = mean\_squared\_error(y\_test, y\_pred\_dt)

RMSE\_DT = math.sqrt(mse\_dt)

MAPE\_DT=mean\_absolute\_error(y\_test, y\_pred\_dt)\*100

print('Decision Tree Regression R2 Score: ', round(r2\_dt,4))

print('Decision Tree Regression Mean Squared Error: ', round(mse\_dt,4))

print('Decision Tree Regression Root Mean Squared Error: ', round(RMSE\_DT,4))

print('Decision Tree Regression Mean Absolute Percentage Error: ', round(MAPE\_DT,4))

# Train and evaluate random forest regression

rf = RandomForestRegressor(n\_estimators=100, random\_state=42)

rf.fit(X\_train, y\_train)

y\_pred\_rf = rf.predict(X\_test)

r2\_rf = r2\_score(y\_test, y\_pred\_rf)

mse\_rf = mean\_squared\_error(y\_test, y\_pred\_rf)

RMSE\_RF = math.sqrt(mse\_rf)

MAPE\_RF=mean\_absolute\_error(y\_test, y\_pred\_rf)\*100

print('Random Forest Regression R2 Score: ', round(r2\_rf,4))

print('Random Forest Regression Mean Squared Error: ', round(mse\_rf,4))

print('Random Forest Regression Root Mean Squared Error: ', round(RMSE\_RF,4))

print('Random Forest Regression Mean Absolute Percentage Error: ', round(MAPE\_RF,4))

# Train and evaluate SVR

svr = SVR()

svr.fit(X\_train, y\_train)

y\_pred\_svr = svr.predict(X\_test)

r2\_svr = r2\_score(y\_test, y\_pred\_svr)

mse\_svr = mean\_squared\_error(y\_test, y\_pred\_svr)

RMSE\_SVR = math.sqrt(mse\_svr)

MAPE\_SVR=mean\_absolute\_error(y\_test, y\_pred\_svr)\*100

print('SVR R2 Score: ', round(r2\_svr,4))

print('SVR Mean Squared Error: ', round(mse\_svr,4))

print('SVR Root Mean Squared Error: ', round(RMSE\_SVR,4))

print('SVR Mean Absolute Percentage Error: ', round(MAPE\_SVR,4))

#train and evaluate KNN

knn = KNeighborsRegressor()

knn.fit(X\_train, y\_train)

y\_pred\_knn = knn.predict(X\_test)

r2\_knn = r2\_score(y\_test, y\_pred\_knn)

mse\_knn = mean\_squared\_error(y\_test, y\_pred\_knn)

RMSE\_KNN = math.sqrt(mse\_knn)

MAPE\_KNN=mean\_absolute\_error(y\_test, y\_pred\_knn)\*100

print('KNN R2 Score: ', round(r2\_knn,4))

print('KNN Mean Squared Error: ', round(mse\_knn,4))

print('KNN Root Mean Squared Error: ', round(RMSE\_KNN,4))

print('KNN Root Mean Absolute Percentage Error: ', round(MAPE\_KNN,4))

#train and evaluate GradientBoostingRegressor

gbr = GradientBoostingRegressor()

gbr.fit(X\_train, y\_train)

y\_pred\_gbr = gbr.predict(X\_test)

r2\_gbr = r2\_score(y\_test, y\_pred\_gbr)

mse\_gbr = mean\_squared\_error(y\_test, y\_pred\_gbr)

RMSE\_GBR = math.sqrt(mse\_gbr)

MAPE\_GBR=mean\_absolute\_error(y\_test, y\_pred\_gbr)\*100

print('GBR R2 Score: ', round(r2\_gbr,4))

print('GBR Mean Squared Error: ', round(mse\_gbr,4))

print('GBR Root Mean Squared Error: ', round(RMSE\_GBR,4))

print('GBR Root Mean Absolute Percentage Error: ', round(MAPE\_GBR,4))

# train and evaluate ANN

ann = MLPRegressor(hidden\_layer\_sizes=(100, 100), max\_iter=500)

ann.fit(X\_train, y\_train)

y\_pred\_ann = ann.predict(X\_test)

r2\_ann = r2\_score(y\_test, y\_pred\_ann)

mse\_ann = mean\_squared\_error(y\_test, y\_pred\_ann)

RMSE\_ANN = math.sqrt(mse\_ann)

MAPE\_ANN=mean\_absolute\_error(y\_test, y\_pred\_ann)\*100

print('ANN R2 Score: ', round(r2\_ann,4))

print('ANN Mean Squared Error: ', round(mse\_ann,4))

print('ANN Root Mean Squared Error: ', round(RMSE\_ANN,4))

print('ANN Root Mean Absolute Percentage Error: ', round(MAPE\_ANN,4))

#Plot the RMSE across models

model\_comparison = pd.DataFrame({'model': [ 'Support Vector Machine',

 'Gradient Boosted','Random Forest',

 'KNN', 'ANN','Decision Tree','Linear Regression','EEMD-SD-SVM'],

 'RMSE': [RMSE\_SVR, RMSE\_GBR,RMSE\_RF, RMSE\_KNN,RMSE\_ANN,RMSE\_DT,RMSE\_LR,0.301]})

model\_comparison.sort\_values('RMSE', ascending = False).plot(x = 'model',

 y = 'RMSE', kind = 'barh',

 edgecolor = 'black', figsize = (6,3))

plt.xlabel('Root Mean Squared Error')

plt.title('Model Comparison on Test RMSE')

plt.show()

#features important

# Create a bar chart to visualize feature importance

feature\_importances.sort()

plt.figure(figsize=(5, 3))

plt.barh(X.columns, feature\_importances)

plt.title('Feature Importance')

plt.xlabel('Features Score')

plt.ylabel('Importance')

#plt.xticks(rotation=45)

plt.show()

#perform Friedman test for model comparison

import pandas as pd

from scipy.stats import friedmanchisquare

# Assuming the performance metrics (pm) is stored in a DataFrame where each column represents a model

# and each row represents a dataset/scenario

pm = {'SVM': [0.3440, 0.1232,0.351,25.1032],

 'Gradient Boost': [0.3402, 0.1239, 0.352, 25.3897],

 'Random Forest': [0.2952, 0.1324, 0.3638, 26.0751],

 'KNN': [0.2569, 0.1396, 0.3736, 27.4998],

 'Linear Regression': [0.3039, 0.1297, 0.3602, 26.3541],

 'ANN': [0.2559, 0.1397, 0.3738, 27.2651],

 'Decision Tree': [-0.2841, 0.2412, 0.4911, 35.9393],

 'EEMD\_SVM': [0.541, 0.09, 0.301, 3.3310]}

# Create a DataFrame from the data

pm\_df = pd.DataFrame(pm)

# Perform the Friedman test

friedman\_result = friedmanchisquare(\*[pm\_df[model] for model in pm\_df.columns])

# Display the Friedman test result

print("Friedman Test Statistic:", friedman\_result.statistic)

print("p-value:", friedman\_result.pvalue)

**R Code for EEMD-SD-SVM**

# Transactions\_residential\_flats DUBA DATA

DF <- read\_xlsx("D:/University/UTM/PhD Files/CH5/Data/dubai property\_grouped.xlsx")

DF\_new <- DF[c('date','price')]

DF$date <- as.Date(DF$date)

DF\_TS <- ts(DF$price, start = c(2005,6), frequency = 30)

df\_ts\_eemd <- eemd(DF\_TS)

#auto correlation of all IMFs

# UK data

k = length(df\_ts\_eemd)/length(DF\_TS); # Total number of IMFs

for (i in 1:k)

{v<-df\_ts\_eemd[ ,i]

 print(cor(v[-1], v[-length(v)]))}

EEMD\_R <- rowSums(df\_ts\_eemd[,c(1,2,3)])

EEMD\_D <- rowSums(df\_ts\_eemd[,c(4,5,5,6,7,8,9,10,11)])

############

## Random DATA

#model\_r

# linear kernel

features <- DF %>% select(-price)

model\_r <- svm(EEMD\_R ~ ., data = features, kernel = "radial")

#model\_d

# linear kernel

model\_d <- svm(EEMD\_D ~ ., data = features, kernel = "radial")

eemd\_rd\_svm = model\_r$fitted+model\_d$fitted

# Accuracy

mse\_eemd\_rd\_svm=round(mse(DF\_TS,eemd\_rd\_svm),3)

mad\_eemd\_rd\_svm=round(mad(DF\_TS,eemd\_rd\_svm),3)

rmse\_eemd\_rd\_svm=round(rmse(DF\_TS,eemd\_rd\_svm),3)

mape\_eemd\_rd\_svm=round(mape(DF\_TS,eemd\_rd\_svm),3)

mae\_eemd\_rd\_svm=round(mae(DF\_TS,eemd\_rd\_svm),3)

mase\_eemd\_rd\_svm=round(mase(DF\_TS,eemd\_rd\_svm),3)

#rsquared <- summary(model)$r.squared

### R2

residuals <- DF\_TS-eemd\_rd\_svm

# Calculate the sum of squared residuals

sum\_sq\_residuals <- sum(residuals^2)

# Calculate the total sum of squares

mean\_y <- mean(DF\_TS)

total\_sum\_sq <- sum((DF\_TS - mean(DF\_TS))^2)

# Calculate R-squared using the formula

rsquared <- 1 - (sum\_sq\_residuals / total\_sum\_sq)

# Print R-squared value

print(rsquared)