**###Predictive Models###**

***1: LSTM in Python***

##Import Necessary library functions##

import os

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler

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

from math import sqrt

from keras.models import Sequential

from keras.layers import LSTM, Dense, Dropout, Bidirectional

from keras.preprocessing.sequence import TimeseriesGenerator

from keras.callbacks import EarlyStopping, ReduceLROnPlateau

##Load dataset & Drop null values if any remaining also set the data frequency as monthly##

df = pd.read\_csv('D:\Thesis\Data and Paper\Models\syl70new\python\monthly.csv', index\_col=['Year'], parse\_dates=['Year'])

df = df.asfreq('MS')

df.dropna(inplace=True)

df.head()

## Plot data to visually inspect the data set##

df['Water\_level'].plot(figsize=(12,6), title='Water Level: Syl70')

plt.show()

##Seasonal Decomposition of data set to further understand trend, seasonal component of data set to consider seasonal impact on groundwater level##

from statsmodels.tsa.seasonal import seasonal\_decompose

import matplotlib.pyplot as plt

# Perform seasonal decomposition

results = seasonal\_decompose(df['Water\_level'], model='additive')

# Plotting each component separately

plt.figure(figsize=(8, 6))

plt.subplot(4, 1, 1)

plt.plot(results.observed, color='blue')

plt.title('(a) Observed Groundwater level(m) ')

plt.ylabel('GWL (m)')

plt.xlabel('Time (month)')

plt.subplot(4, 1, 2)

plt.plot(results.trend, color='red')

plt.title('(b) Trend in GWL')

plt.ylabel('Trend(m)')

plt.xlabel('Time (month)')

plt.subplot(4, 1, 3)

plt.plot(results.seasonal, color='green')

plt.title('(c) Seasonal Component')

plt.ylabel('Seasonal Component(m)')

plt.xlabel('Time (month)')

plt.subplot(4, 1, 4)

plt.plot(results.resid, color='purple')

plt.title('(d) Residuals')

plt.ylabel('Residual(m)')

plt.xlabel('Time (month)')

plt.tight\_layout()

plt.show()

##check data length##

len(df)

##split data into training and testing set based on data length: (80% train 20% test)

train\_size = int(len(df) \* 0.8)

train, test = df.iloc[:train\_size], df.iloc[train\_size:]

## Normalize data into a common scale (0-1) from varying range##

scaler = MinMaxScaler()

scaler.fit(df[['Water\_level']])

scaled\_train = scaler.transform(train[['Water\_level']])

scaled\_test = scaler.transform(test[['Water\_level']])

##check normalized data set##

Scaled\_train[:10]

scaled\_test[:10]

## Time-series generator##

n\_input = 36

n\_features = 1

generator = TimeseriesGenerator(scaled\_train, scaled\_train, length=n\_input, batch\_size=4)

# ##Model structure: 1###

model = Sequential([

Bidirectional(LSTM(128, activation='relu', return\_sequences=True), input\_shape=(n\_input, n\_features)),

Dropout(0.2),

LSTM(128, return\_sequences=True),

Dropout(0.01),

LSTM(64),

Dropout(0.01),

Dense(1)

])

model.compile(optimizer='adam', loss='mse')

model.summary()

## Training callbacks to prevent over fitting##

early\_stopping = EarlyStopping(monitor='loss', patience=20, restore\_best\_weights=True)

reduce\_lr = ReduceLROnPlateau(monitor='loss', factor=0.5, patience=10, verbose=1)

## Fit the model in training data##

model.fit(generator, epochs=300, callbacks=[early\_stopping, reduce\_lr])

##plot loss function of the model and adjust model until loss is fully stable: low and parallel to X axis; Nb in that case do not use early stopping condition##

loss\_per\_epochs = model.history.history['loss']

plt.figure(figsize=(4, 3)) # Define the figure size (width, height)

plt.plot(range(len(loss\_per\_epochs)), loss\_per\_epochs)

#plt.title('Training Loss per Epoch')

plt.xlabel('Epochs(Nos)')

plt.ylabel('Loss(mse) ')

#plt.grid(True) # Add a grid

plt.show()

## Forecasting##

predictions = []

current\_batch = scaled\_train[-n\_input:].reshape((1, n\_input, n\_features))

for \_ in range(len(test)):

pred = model.predict(current\_batch)[0]

predictions.append(pred)

current\_batch = np.append(current\_batch[:, 1:, :], [[pred]], axis=1)

## Convert predicted data to original data scale##

true\_predictions = scaler.inverse\_transform(predictions)

test['Predictions'] = true\_predictions

## Plot prediction results##

test[['Water\_level', 'Predictions']].plot(figsize=(12,6))

plt.show()

# Evaluation

rmse = sqrt(mean\_squared\_error(test['Water\_level'], test['Predictions']))

mae = mean\_absolute\_error(test['Water\_level'], test['Predictions'])

r2 = r2\_score(test['Water\_level'], test['Predictions'])

print(f'RMSE: {rmse:.3f}, MAE: {mae:.3f}, R2 Score: {r2:.3f}')

***2: ARIMA in R-Software***

####import library functions

library(forecast)

library(tseries)

library(ggplot2)

library(caTools)

library(hydroGOF)

library(hydroTSM)

####read data

y<-read.csv("monthly.csv")

#change data into time series

yt<-ts(y$Water\_level,start = c(1990), frequency = 12)

###preliminary data check

autoplot(yt)+

ggtitle("monthly average water level")+

ylab("water level")

###check stationarity

adf.test(yt,k=12)

###p value 1%=0.01 to 10%=0.1 stationary otherwise run differencing##

#yd<-diff(yt,d=1)

###check for seasonality on time series data or differenced data if differenced##

ggseasonplot(yt)+

ggtitle("seasonal variation of water level")+

ylab("water level")

ggsubseriesplot(yt)

###Create a subseries plot with blue color###

p <- ggsubseriesplot(yt, col = "blue")

### Add a trend line with red color###

p + geom\_smooth(method = "lm", se = FALSE, color = "red")

###training and testing data set##

train\_data<-ts(y$Water\_level,start = c(1990),end = c(2005),frequency = 12)

test\_data<-ts(y$Water\_level,start = c(2006),end = c(2024),frequency = 12)

##plot the ACF and Pacf to select q term and p term of arima model##

acf(train\_data,main="ACF", lwd=2,col='green')

##plot the PACF##

pacf(train\_data, main="PACF", lwd=2,col='green')

###find how much differencing require for removing stationarity thus selecting d term##

ndiffs(train\_data)

###Fit seasonal ARIMA model manually using selected p, d and q term otherwise use auto arima function below##

#train\_arima<-arima(train\_data,order = c(0,1,0), seasonal = list(order = c(1, 0, 1), period = 12))

###use auto arima to find out arima model##

#train\_arima<-auto.arima(train\_data,max.p=8,max.q=8,start.p=0,start.q=0,stepwise = FALSE,approximation = FALSE,seasonal = TRUE,ic=c("aic"),trace = TRUE)

train\_arima <- auto.arima(train\_data, max.p=8, max.q=8, start.p=0, start.q=0, lambda="auto", stepwise = FALSE, approximation = FALSE, seasonal = TRUE, ic=c("aicc", "bic"), trace = TRUE)

###plot arima model###

plot(train\_arima)

print(summary(train\_arima))

checkresiduals(train\_arima)

###forecast##

train\_arima\_forecast<-forecast(train\_arima,level = c(95),h=96)

####plot forescasted and train\_data set##

plot(train\_data,col="green")

lines(fitted(train\_arima\_forecast))

####plot forecasted and test data set###

autoplot(train\_arima\_forecast, include = 0) +

ggtitle("Water Level Forecast") +

ylab("Water Level") +

geom\_line(data = data.frame(test\_data), aes(x = time(test\_data), y = test\_data, color = "Testing"), size = 0.5) +

scale\_color\_manual(values = c("Forecasted" = "green", "Testing" = "red")) +

theme(legend.position = "top")

summary(train\_arima\_forecast)

gof(train\_data,fitted(train\_arima))

### Create a data frame with train\_data and ARIMA model values###

df <- data.frame(train\_data = train\_data, arima\_model = fitted(train\_arima))

### Plot train\_data vs ARIMA model data using ggplot2###

ggplot(df, aes(x = train\_data, y = arima\_model)) +

geom\_point() +

geom\_smooth(method = "lm", se = FALSE, color = "red")+

labs(x = "Train Data", y = "ARIMA Model Data")

###model diagonistics###

acf(train\_arima\_forecast$residuals,main=" Residuals ACF",lwd=2,col="green")

pacf(train\_arima\_forecast$residuals,main="Residuals PACF",lwd=2,col="green")

coef(train\_arima\_forecast)

print(fitted(train\_arima))

### Extract fitted values###

#fitted\_values <- as.data.frame(fitted(train\_arima))

## Write fitted values to CSV file###

#write.csv(fitted\_values, file = "fitted\_values70.csv", row.names = FALSE)

#predicted\_values <- as.data.frame(forecast(train\_arima\_forecast))

#write.csv(predicted\_values, file = "predictedauto.csv", row.names = FALSE)

###check NSE Nash-Sutcliffe Efficiency (NSE)

###A value of 1 indicates a perfect match between the observed and modeled data,

###while values less than 0 indicate poor model performance.

gof(train\_data,fitted(train\_arima))

***2: Random Forest in Python***

## Import Required Libraries##

import pandas as pd

import numpy as np

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import mean\_squared\_error

import matplotlib.pyplot as plt

from statsmodels.tsa.tsatools import lagmat

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error, mean\_squared\_error

from sklearn.model\_selection import GridSearchCV

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error, mean\_squared\_error

## Load the data set##

df = pd.read\_csv(r'D:\Thesis\Data and Paper\Data\prepared data\prepared data monthly avg wl\data\monthly time series\syl82.csv', index\_col='Year', parse\_dates=True)

##Create lagged variables##

for i in range(1, 13):

df[f"lag\_{i}"] = df["Water\_level"].shift(i)

## Remove null values##

df = df.dropna()

##Split the data into training and testing sets##

train\_size = int(0.8 \* len(df))

train\_data = df.iloc[:train\_size, :]

test\_data = df.iloc[train\_size:, :]

##Define features 'X' and target variable 'y' for training##

X\_train = train\_data.drop("Water\_level", axis=1)

y\_train = train\_data["Water\_level"]

###Define the Random Forest Regressor model##

model = RandomForestRegressor(random\_state=42) # Setting the random state for reproducibility

### Define the parameters grid for the grid search##

param\_grid = {

'n\_estimators': [50, 100, 200, 300], # Number of trees in the forest

'max\_depth': [5, 10, 15, 20], # Maximum depth of the trees

# Other hyperparameters to be tuned can be added here

}

### Create the GridSearchCV object###

grid\_search = GridSearchCV(estimator=model, param\_grid=param\_grid, cv=3, scoring='neg\_mean\_squared\_error')

###Fit the GridSearch to find the best parameters###

grid\_search.fit(X\_train, y\_train)

### Extract grid search results##

results = grid\_search.cv\_results\_

n\_estimators = param\_grid['n\_estimators']

max\_depth = param\_grid['max\_depth']

scores = -1 \* np.array(results['mean\_test\_score']).reshape(len(n\_estimators), len(max\_depth))

### Plotting the heatmap of MSE for different hyperparameters###

plt.figure(figsize=(6, 6))

plt.imshow(scores, cmap='viridis', interpolation='nearest')

plt.title('Grid Search Mean Squared Error')

plt.xlabel('max\_depth')

plt.ylabel('n\_estimators')

plt.colorbar()

plt.xticks(np.arange(len(max\_depth)), max\_depth)

plt.yticks(np.arange(len(n\_estimators)), n\_estimators)

plt.show()

###Get the best parameters and the best score###

best\_params = grid\_search.best\_params\_

best\_score = grid\_search.best\_score\_

print("Best Parameters:", best\_params)

print("Best Score:", best\_score)

### Random Forest Regression Model###

model = RandomForestRegressor(n\_estimators=100, max\_depth=10, random\_state=42)

###Train the model using the training data###

X\_train = train\_data.drop(['Year', 'Water\_level'], axis=1)

y\_train = train\_data['Water\_level']

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

###Predict the water levels on the train data###

y\_train\_pred = model.predict(X\_train)

###Predict the water levels on the test data##

X\_test = test\_data.drop(['Year', 'Water\_level'], axis=1)

y\_test = test\_data['Water\_level']

y\_pred = model.predict(X\_test)