National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Gokhan Fidan
Student ID: 22148795

School of Computing
National College of Ireland

Supervisor: Hicham Rifai

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Gokhan Fidan		
Student ID:	22148795		
Programme:	MSc in Data Analytics	**Year:**	2023
Module:	MSc Research Project		
Supervisor:	Hicham Rifai		
Submission Due Date:	14/12/2023		
Project Title:	Configuration Manual		
Word Count:	987	**Page Count:**	6

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Gokhan Fidan

Date: 14/12/2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Gokhan Fidan
Student ID: 22148795

1 Introduction

This manual provides guidance on setting up and running a Python project aimed at predicting Bitcoin prices using Long Short-Term Memory (LSTM) models combined with technical indicators such as Super Trend, Kaufman's Adaptive Moving Average, Fibonacci's Weighted Moving Average, and Average True Range Trailing Stop Loss.

2 System Requirements

2.1 Software Configuration

Python: Version 3.7 or higher
Libraries: pandas, numpy, matplotlib, pandas_ta, talib, sklearn, tensorflow
Development Environment: Jupyter Notebook or any Python IDE

2.2 Data Source

Dataset: BTC-USD.csv (Bitcoin price data)
Source: **https://finance.yahoo.com/quote/BTC-USD/history/.** Enter this website and in time period section choose 'Max'. After that apply and directly download historical data. In this study data used for the period of September 17, 2014, through December 5, 2023.

3 Environment Setup

Install Python: Ensure Python 3.7 or higher is installed on your system.

Install Required Libraries: Use pip to install the necessary libraries. Run the following commands in your terminal or command prompt:

4 Implementation

4.1 Data Preparation

Load Data: Read the Bitcoin price data from the CSV file by using 'data = pd.read_csv("BTC-USD.csv")' comment

Calculate Technical Indicators: Use the pandas_ta and talib libraries to calculate Super Trend, KAMA, and FWMA. SuperTrend, KAMA, and FWMA are calculated using the pandas_ta library. ATR (Average True Range) and ATR Trailing Stop are computed using the talib library.

In [2]:

```
# Calculate SuperTrend
datam = ta.supertrend(data.High, data.Low, data.Close, length=10, multiplier=3)
data['SuperTrend'] = datam['SUPERT_10_3.0']

# Calculate KAMA
datam1 = ta.kama(data.Close)
data['KAMA'] = datam1

# Calculate FWMA
datam2 = ta.fwma(data.Close)
data['FWMA'] = datam2
```

In [3]:

```
#Calculate ATR Trailing Stop Loss
def calculate_atr(data, period=14):
    high = data['High']
    low = data['Low']
    close = data['Close']
    atr = talib.ATR(high, low, close, timeperiod=period)
    return atr

def calculate_atr_trailing_stop(data, atr_multiplier=3.0, atr_period=14):
    data['ATR'] = calculate_atr(data, atr_period)
    data['Trailing_Stop_Long'] = data['Close'] - (atr_multiplier * data['ATR'])
    data['Trailing_Stop_Short'] = data['Close'] + (atr_multiplier * data['ATR'])

    data['Trailing_Stop_Long_Prev'] = data['Trailing_Stop_Long'].shift(1)
    data['Trailing_Stop_Short_Prev'] = data['Trailing_Stop_Short'].shift(1)

    for i in range(1, len(data)):
        data.loc[i, 'Trailing_Stop_Long'] = max(data.loc[i, 'Trailing_Stop_Long'], data.loc[i - 1, 'Trailing_Stop_Long_Prev'
        data.loc[i, 'Trailing_Stop_Short'] = min(data.loc[i, 'Trailing_Stop_Short'], data.loc[i - 1, 'Trailing_Stop_Short_Pr

    return data

calculate_atr_trailing_stop(data)
```

Data Cleaning and Preparation: Convert date columns to datetime objects, handle missing values, and drop unnecessary columns.

In [8]:

```
# Data cleaning and preparation
df = data.copy()
df['Date'] = pd.to_datetime(df['Date'])
df.set_index('Date', inplace=True)
df.fillna(method='bfill',inplace=True)
df.drop(['Open', 'High', 'Low', 'Adj Close', 'Volume', 'ATR', 'Trailing_Stop_Long_Prev', 'Trailing_Stop_Short_Prev'], axis=1
# Select features
features = ['Close', 'SuperTrend', 'KAMA', 'FWMA', 'Trailing_Stop_Long', 'Trailing_Stop_Short']

# Normalize the features
scaler = MinMaxScaler()
df[features] = scaler.fit_transform(df[features])
```

4.2 Data Transformation

Normalize Features: Use ‘MinMaxScaler’ from ‘sklearn.preprocessing’ module to scale the features. Following feature engineering, the dataset was ready for the LSTM model. This required using the MinMaxScaler from the sklearn.preprocessing module to normalize the features. Deep learning models require normalization because it uniformizes the input feature range and promotes faster training convergence. The normalization process involved to the Trailing_Stop_Long, Trailing_Stop_Short, KAMA, SuperTrend, Close and FWMA features.

Create Sequences: Transform the data into sequences for LSTM input. Sequences are the type of input data that the LSTM model needs. As a result, the prepared dataset was split up into sequences, each of which stood for a distinct time period 60 days in this case. For the data to show the temporal dependencies, this transformation was essential. From the normalized data, these sequences were produced using a custom function called create_sequences.

In [9]:
```
# Normalize the features
scaler = MinMaxScaler()
df[features] = scaler.fit_transform(df[features])

def create_sequences(data, sequence_length):
    xs, ys = [], []
    for i in range(len(data) - sequence_length):
        x = data.iloc[i:(i + sequence_length)].values
        y = data.iloc[i + sequence_length, 0]  # Assuming the first column is the target (Close price)
        xs.append(x)
        ys.append(y)
    return np.array(xs), np.array(ys)

# Define the sequence length (e.g., 60 days)
sequence_length = 60

# Create sequences
X, y = create_sequences(df[features], sequence_length)

```

4.3 Model Building

Model Architecture: The Sequential model from the tensorflow.keras.models module was used to construct the LSTM model. As previously mentioned, the model architecture consisted of Dense, Dropout, and LSTM layers. The LSTM layers in the model are made to process sequential data and retain patterns over extended periods of time. The model's 100 units per LSTM layer strikes a balance between computational efficiency and model complexity. Dropout layers, which randomly remove a percentage of the neurons (set at 20% in this study) during the training phase, are used to prevent overfitting. By doing this, it is made sure that the model is not unduly dependent on any one feature or pattern found in the training set. Dense layers are employed after the LSTM layers to interpret the features that the LSTM has learned. The model consists of a 35- unit Dense layer and a final Dense layer with an output of one unit, which represents the anticipated price.

In [10]:
```
def build_model(input_shape):
    model = Sequential()
    model.add(LSTM(100, return_sequences=True, input_shape=input_shape))
    model.add(Dropout(0.2))
    model.add(LSTM(75, return_sequences=False))
    model.add(Dropout(0.2))
    model.add(Dense(35, activation="relu"))
    model.add(Dense(1, activation="linear"))
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model

```

Optimizer: Adam optimizer is used for compiling the model. The Adam optimizer, which is well-liked for deep learning applications because of its effectiveness in managing sparse gradients and adaptable learning rates, is used to compile the model.

Loss Function: Mean Squared Error (MSE) is set as the loss function. Mean squared error (MSE) is the loss function that is employed, and it is suitable for regression tasks such as price prediction.

4.4 Cross Validation Setup

K-Fold Cross Validation: Use 'KFold' from 'sklearn.model_selection' for model validation. The model is then trained on 'k-1' folds, and its validity is checked on the remaining fold. Every fold serves as the validation set once during the 'k' iterations of this process. Five-fold cross-validation is employed in this work. With a batch size of 32, the model is trained on each fold for 15 epochs. The LSTM model is trained by feeding it data sequences, including technical indicator data, and modifying the model weights to minimize the loss function. 'KFold' is a model cross-validator that divides the dataset into k consecutive folds (in this case, 5 folds). Each fold is then used once as a validation while the k - 1 remaining folds form the training set. 'shuffle'=True ensures that the data is shuffled before splitting into batches. 'random_state'=42 sets a seed for the random number generator that shuffles the data, ensuring reproducible splits.

Training and Validation: Train the model on training folds and validate on the remaining fold.

In [11]:

```
# K-Fold Cross-Validation setup
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Lists to keep track of metrics and predictions
histories = []
metrics_df = pd.DataFrame(columns=['Fold', 'Set', 'MSE', 'RMSE', 'MAE', 'R2'])
fold_predictions = []
fold_no = 1

# Function to calculate metrics
def calculate_metrics(y_true, y_pred):
    epsilon = 1e-10  # A small number to avoid division by zero
    mse = mean_squared_error(y_true, y_pred)
    rmse = math.sqrt(mse)
    mae = mean_absolute_error(y_true, y_pred)
    r2 = r2_score(y_true, y_pred)
    return mse, rmse, mae, r2

for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Build and fit the model
    model = build_model(X_train.shape[1:])
    print(f'Training for fold {fold_no} ...')
    history = model.fit(X_train, y_train, batch_size=32, epochs=15, validation_data=(X_test, y_test))

    # Store the history
    histories.append(history)

```

```
    # Calculate metrics for training set
    train_predictions = model.predict(X_train)
    train_mse, train_rmse, train_mae, train_r2 = calculate_metrics(y_train, train_predictions)

    # Calculate metrics for validation set
    val_predictions = model.predict(X_test)
    val_mse, val_rmse, val_mae, val_r2 = calculate_metrics(y_test, val_predictions)

    # Append metrics to the DataFrame
    metrics_df = metrics_df.append({'Fold': fold_no, 'Set': 'Train', 'MSE': train_mse, 'RMSE': train_rmse, 'MAE': train_mae,
    metrics_df = metrics_df.append({'Fold': fold_no, 'Set': 'Validation', 'MSE': val_mse, 'RMSE': val_rmse, 'MAE': val_mae,

    # Store predictions and actual values with dates
    val_dates = df.index[test_index + sequence_length]  # Adjusting for the sequence length
    val_predictions_rescaled = scaler.inverse_transform(np.concatenate((val_predictions, np.zeros((len(val_predictions), len

    y_test_reshaped = y_test.reshape(-1, 1)  # Reshape y_test to be a 2D array
    y_test_rescaled = scaler.inverse_transform(np.concatenate((y_test_reshaped, np.zeros((len(y_test), len(features) - 1))),

    fold_predictions.append((val_dates, y_test_rescaled, val_predictions_rescaled))

    fold_no += 1
```

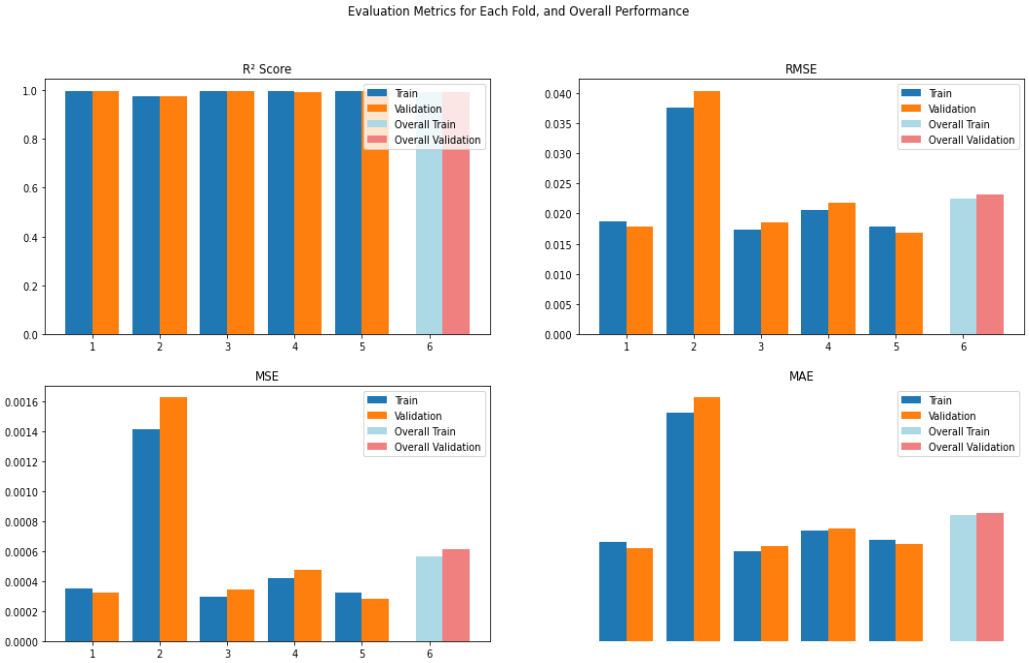
```
Training for fold 1 ...
Epoch 1/15
83/83 [==============================] - 9s 73ms/step - loss: 0.0035 - val_loss: 9.7495e-04
Epoch 2/15
```

4.5 Evaluation

Calculate Metrics: The model's performance was evaluated using metrics like MAE, MSE, RMSE and R2 Score. These metrics provided a comprehensive understanding of the model's accuracy and predictive power. The evaluation included visualizing the performance metrics and learning curves to gain an intuitive understanding of the model's learning process and predictive accuracy.

Visualization: Plot training and validation loss, and other metrics for each fold.

Fold	Set	MSE	RMSE	MAE	R2
1	Train	0.000352	0.018766	0.010465	0.993800
1	Validation	0.000320	0.017877	0.009788	0.994454
2	Train	0.001415	0.037617	0.024075	0.974639
2	Validation	0.001623	0.040281	0.025692	0.973666
3	Train	0.000299	0.017278	0.009514	0.994827
3	Validation	0.000346	0.018604	0.010065	0.993604
4	Train	0.000420	0.020505	0.011713	0.992668
4	Validation	0.000476	0.021827	0.011928	0.991427
5	Train	0.000320	0.017902	0.010685	0.994407
5	Validation	0.000283	0.016835	0.010248	0.994916



5 Running the Code

Execute the Script: Run the Python script in your development environment.

Monitor Outputs: Check the console or output cells for model performance metrics and visualizations.

6 Running the Code

This manual provides a comprehensive guide to setting up and running the Bitcoin price prediction project. Ensure all dependencies are correctly installed and follow the steps outlined for data processing, model building, training, and evaluation.

References