FOREX PRICE PREDICTION USING MACHINE LEARNING

Amandeep Singh, Nikechukwum Ene, Pourya Asadi, Princewilliams Gbasouzor

INTRODUCTION TO THE PREDICTION SYSTEM

This project is focused on predicting Forex prices using machine learning and technical analysis. The main idea is to use historical market data and financial indicators to train models that can guess the future price direction of a currency pair, like EUR/USD. The code combines data science, machine learning, and financial analysis in one system.

The prediction system is built step-by-step through different modules. Each module has a specific job, and together they form the complete prediction pipeline. The parts of our prediction system include:

1. **Data Collection and Cleansing**:
    - A script (`DataCreation.ipynb`) takes the raw price and indicator data and builds a clean dataset.
    - This step ensures the data is in the right format for machine learning models.
    - It includes steps like filling missing values, selecting important columns, and labelling the data with the target (what we want to predict).
    - The raw data was downloaded from two free websites (forexsb.com and finance.yahoo.com) and then changed and pre-processed
2. **Data Structuring and Preprocessing**:
    - The system starts with historical Forex price data.
    - It uses a script (`IndicatorsCreation.py`) to calculate technical indicators, such as RSI, MACD, Bollinger Bands, ATR, and others.
    - These indicators help understanding the market by describing market trends.
3. **Utilities and Preprocessing Functions**:
    - A file called `utilsation.py` includes helper functions.

- These functions normalize the data (scale values between 0 and 1), split data into training and testing sets, and manage other small tasks.

4. **Machine Learning Models and Training**:
    - The main script (`Prediction System.ipynb`) handles the training and testing of several machine learning models.
    - Models used include:
        - Linear Regression
        - Bayesian Ridge Regression
        - Decision Tree Regressor
        - Random Forest Regressor
        - Support Vector Regressor (SVR)
5. **Visualisation of Results**:
    - The project also includes visualisations to help understand the model performance.
    - It shows things like actual vs predicted prices, error distributions, and indicator trends.
    - There are 10 types of visualisations used, including line charts, bar graphs, and scatter plots.

In summary, when these segments work together to predict Forex price movements using technical indicators, multiple ML models, and detailed visualisations to explain what's happening at each stage.

DATA COLLECTION & CLEANSING

(`DataCreation.ipynb`**)**

This python module is the main part of our data preparation process. It takes the raw Forex data and turns it into a clean, structured dataset that our machine learning models can understand and learn from. Without this step, the models would have a hard time figuring anything out because they wouldn't have the indicators or labels they need.

Following the processes step by step, we have:

Step 1: Importing Required Libraries

```
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
from IndicatorsFunction import *
```

- `pandas` is used to load and manage our data in tables (DataFrames).
- `numpy` helps with calculations and arrays (lists of numbers).
- `os` is used to work with file paths and directories.
- `matplotlib.pyplot` is used to plot charts (though it's not heavily used here).
- Finally, we import all the indicator functions from `IndicatorsCreation.py`, which we explained earlier.

This setup is needed before we start processing any data.

Step 2: Load the Raw Price Data

```
from utils import *
import pandas as pd

columns = ['Date','Open','High','Low','Close','Volume','Yield','PercentageVolume',
           'SMA6','EMA6','WMA6','HMA6','SMA20','EMA20','WMA20','HMA20','SMA50','EMA50','WMA50','HMA50',
           'SMA100','EMA100','WMA100','HMA100','MACD','CCI','Stochastic Oscillator','RSI','ROC','PPO',
          'KST','BOLU','BOLD','BOLM']
data = pd.read_csv('Data after preprocessing/EURUSD/EURUSD_D1.csv', names = columns, header=0)
data.head(5)
```

Here, we are loading a CSV file that contains historical price data for the EUR/USD currency pair, with daily candlesticks.

At this point, data is a table with columns like Date, Time, Open, High, Low, Close, and Volume.

Also, the data after pre-processing have all the needed indicators (this process is explained fully in “Utilities and Preprocessing Functions” part of the report).

Step 3: Keep Only Needed Columns

```
toAdd = ['Volume', 'Date','High','Low','Open','Close']
close = data['Close']
df = selectData(data,toAdd)
normDf = normalizeData(df)
normClose = normalizeData(close)
```

This filters the table so we only keep the four important price columns:

- `Open`: The price at the beginning of the hour.
- `High`: The highest price during the hour.
- `Low`: The lowest price during the hour.
- `Close`: The price at the end of the hour.

Also normalized data is added in `normDf` (for the whole data) and `normClose` (only for close column of data which is going to be predicted)

Step 4: Plotting the Data

```
import matplotlib.pyplot as plt
%matplotlib inline
df.hist(figsize=(28,15), layout=(3,6), bins=100)
plt.show()
```

This part plot the charts for each column in the data

- `matplotlib` is used to plot the charts
- `%matplotlib inline` makes sure the charts appear inside the notebook
- `df.hist` draws a chart for each column using 100 bars
- `figsize` makes the charts large so they are easy to read
- `layout` puts the charts in 3 rows and 6 columns
- `plt.show` displays all the charts together in one place

Step 5: Plotting The Data Heatmap

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.figure(figsize=(13,13))
plt.matshow(normDf.corr(), fignum=1)
plt.colorbar()
plt.yticks(range(len(normDf.columns)),normDf.columns,fontsize=10)
plt.xticks(range(len(normDf.columns)),normDf.columns,fontsize=10,rotation=90)
plt.title("Correlation between Features and Label", y=1.2)
plt.show()
```

This code part create a heatmap that shows how each column is related to the other columns

- `matplotlib` is used to make the heatmap
- `plt.figure` sets the size of the heatmap
- `plt.matshow` shows the correlation between all columns using colours
- `plt.colorbar` adds a colour scale to help understand the values
- `plt.yticks` and `plt.xticks` label the sides with the column names and adjust their size and rotation

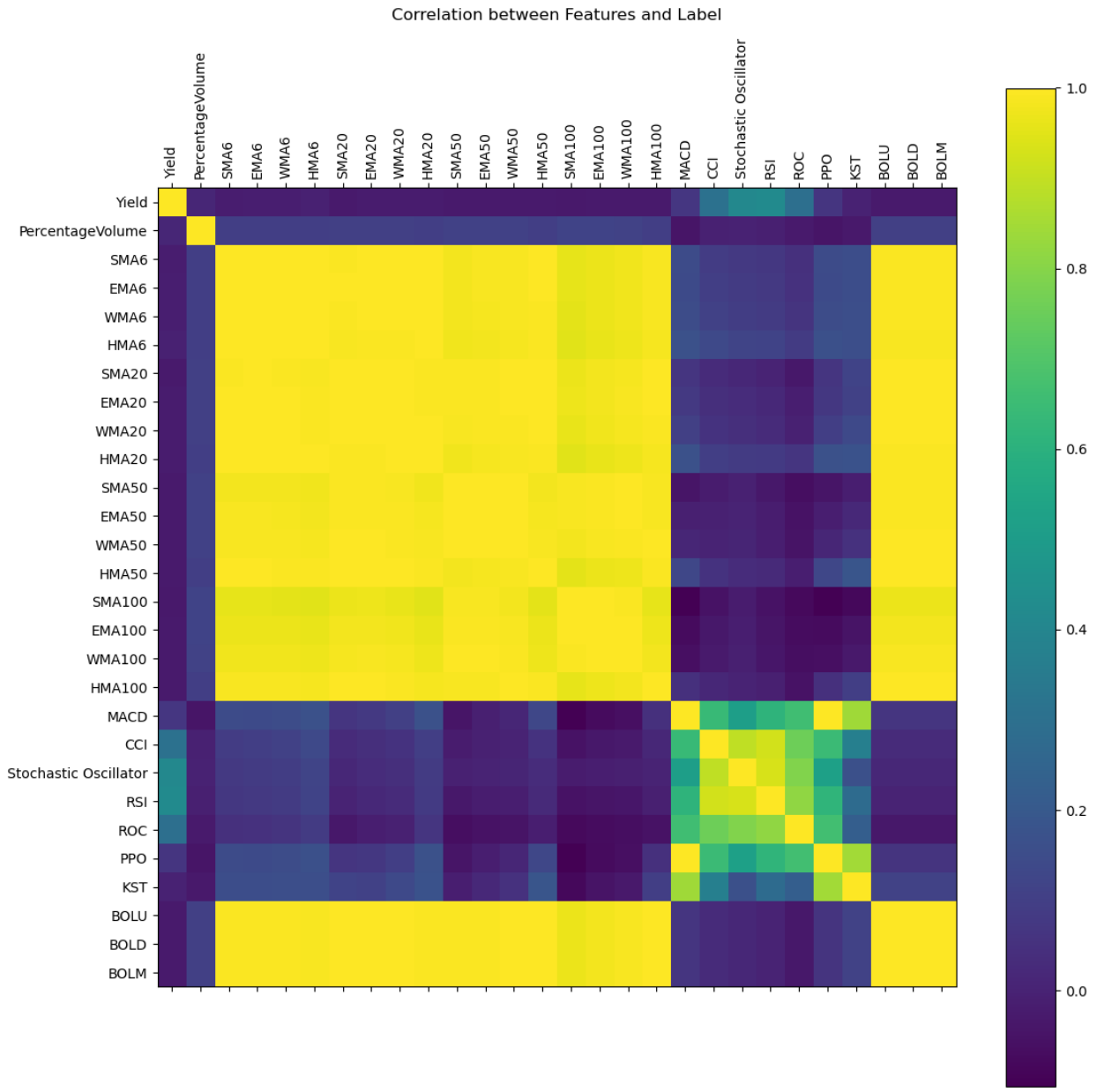


Fig. 1: Heatmap visualisation of the dataset

Step 6: Build the Coloured Table (To Show Columns Relation)

```
corr = normDf.corr()
corr.style.background_gradient(cmap='coolwarm')
```

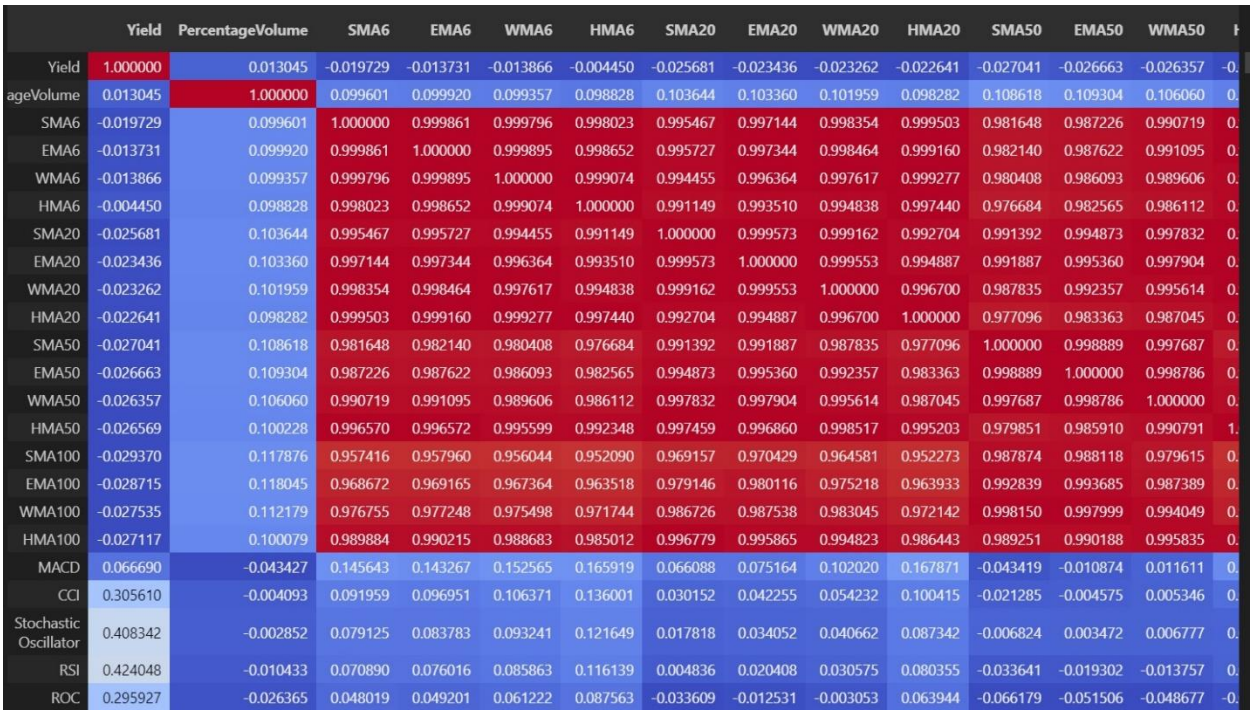
In this step we make a coloured table that shows how the columns are related to each other

• `normDf.corr` calculates the relationship between all the columns

• The result is saved in a new table called `corr`

• `corr.style.background_gradient` adds color to the table

• The colors use the coolwarm style to show strong and weak relationships clearly



	Yield	PercentageVolume	SMA6	EMA6	WMA6	HMA6	SMA20	EMA20	WMA20	HMA20	SMA50	EMA50	WMA50	H
Yield	1.000000	0.013045	-0.019729	-0.013731	-0.013866	-0.004450	-0.025681	-0.023436	-0.023262	-0.022641	-0.027041	-0.026663	-0.026357	-0.
ageVolume	0.013045	1.000000	0.099601	0.099920	0.099357	0.098828	0.103644	0.103360	0.101959	0.098282	0.108618	0.109304	0.106060	0.
SMA6	-0.019729	0.099601	1.000000	0.999861	0.999796	0.998023	0.995467	0.997144	0.998354	0.999503	0.981648	0.987226	0.990719	0.
EMA6	-0.013731	0.099920	0.999861	1.000000	0.999895	0.998652	0.995727	0.997344	0.998464	0.999160	0.982140	0.987622	0.991095	0.
WMA6	-0.013866	0.099357	0.999796	0.999895	1.000000	0.999074	0.994455	0.996364	0.997617	0.999277	0.980408	0.986093	0.989606	0.
HMA6	-0.004450	0.098828	0.998023	0.998652	0.999074	1.000000	0.991149	0.993510	0.994838	0.997440	0.976684	0.982565	0.986112	0.
SMA20	-0.025681	0.103644	0.995467	0.995727	0.994455	0.991149	1.000000	0.999573	0.999162	0.992704	0.991392	0.994873	0.997832	0.
EMA20	-0.023436	0.103360	0.997144	0.997344	0.996364	0.993510	0.999573	1.000000	0.999553	0.994887	0.991887	0.995360	0.997904	0.
WMA20	-0.023262	0.101959	0.998354	0.998464	0.997617	0.994838	0.999162	0.999553	1.000000	0.996700	0.987835	0.992357	0.995614	0.
HMA20	-0.022641	0.098282	0.999503	0.999160	0.999277	0.997440	0.992704	0.994887	0.996700	1.000000	0.977096	0.983363	0.987045	0.
SMA50	-0.027041	0.108618	0.981648	0.982140	0.980408	0.976684	0.991392	0.991887	0.987835	0.977096	1.000000	0.998889	0.997687	0.
EMA50	-0.026663	0.109304	0.987226	0.987622	0.986093	0.982565	0.994873	0.995360	0.992357	0.983363	0.998889	1.000000	0.998786	0.
WMA50	-0.026357	0.106060	0.990719	0.991095	0.989606	0.986112	0.997832	0.997904	0.995614	0.987045	0.997687	0.998786	1.000000	0.
HMA50	-0.026569	0.100228	0.996570	0.996572	0.995599	0.992348	0.997459	0.996860	0.998517	0.995203	0.979851	0.985910	0.990791	1.
SMA100	-0.029370	0.117876	0.957416	0.957960	0.956044	0.952090	0.969157	0.970429	0.964581	0.952273	0.987874	0.988118	0.979615	0.
EMA100	-0.028715	0.118045	0.968672	0.969165	0.967364	0.963518	0.979146	0.980116	0.975218	0.963933	0.992839	0.993685	0.987389	0.
WMA100	-0.027535	0.112179	0.976755	0.977248	0.975498	0.971744	0.986726	0.987538	0.983045	0.972142	0.998150	0.997999	0.994049	0.
HMA100	-0.027117	0.100079	0.989884	0.990215	0.988683	0.985012	0.996779	0.995865	0.994823	0.986443	0.989251	0.990188	0.995835	0.
MACD	0.066690	-0.043427	0.145643	0.143267	0.152565	0.165919	0.066088	0.075164	0.102020	0.167871	-0.043419	-0.010874	0.011611	0.
CCI	0.305610	-0.004093	0.091959	0.096951	0.106371	0.136001	0.030152	0.042255	0.054232	0.100415	-0.021285	-0.004575	0.005346	0.
Stochastic Oscillator	0.408342	-0.002852	0.079125	0.083783	0.093241	0.121649	0.017818	0.034052	0.040662	0.087342	-0.006824	0.003472	0.006777	0.
RSI	0.424048	-0.010433	0.070890	0.076016	0.085863	0.116139	0.004836	0.020408	0.030575	0.080355	-0.033641	-0.019302	-0.013757	0.
ROC	0.295927	-0.026365	0.048019	0.049201	0.061222	0.087563	-0.033609	-0.012531	-0.003053	0.063944	-0.066179	-0.051506	-0.048677	-0.

Fig. 2: Coloured table visualisation of the dataset showing column relationships

Step 7: Plotting Scatter Plot for Column Comparison

```
from pandas.plotting import scatter_matrix
axes = scatter_matrix(normDf, figsize=(30,20))
for ax in axes.flatten():
    ax.xaxis.label.set_rotation(90)
    ax.yaxis.label.set_rotation(0)
    ax.yaxis.label.set_ha('right')
for i in range(np.shape(axes)[0]):
    for j in range(np.shape(axes)[1]):
        if i < j:
            axes[i,j].set_visible(False)
plt.show()
```

This code makes many small scatter plots to compare all columns with each other

- `scatter_matrix` shows how each column is related to every other one using small charts
- the loop changes the direction on the labels so they are much easier to see
- another loop hides the upper part of the chart grid to remove repeated charts

---

Step 8: Creating Box Plot

```
import matplotlib.pyplot as plt
%matplotlib inline
fig, axes = plt.subplots(4,7,figsize=(25,13))
fig.tight_layout()
axes = axes.flatten()
for i, col in enumerate(list(sorted(df.columns))):
    axes[i].boxplot(df[col], showfliers = False)
    axes[i].set_title(col)
    axes[i].grid()
plt.show()
```

Now we creates box plots to show the range and spread of each column

- `plt.subplots` makes 28 boxes in 4 rows and 7 columns with a big size
- `tight_layout` gives the plots some space so they don’t overlap
- `axes.flatten` puts all the boxes in one list so we can use them in a loop
- The loop goes through every column one by one
- `axes[i].boxplot` makes a box plot for each column without outliers
- `grid()` adds lines to help see the values

Summary

- This notebook prepares our entire dataset in one clean pipeline.
- It loads the raw daily price data.
- It calculates all the technical indicators needed for prediction.
- It normalizes and adds them to the main table.
- It creates a Target column (future close price).
- It saves everything to a new file for the ML models to use.

Without this step, the models wouldn’t have anything to learn from. This step builds the "features" (the inputs) and the "target" (the output) for the learning process

DATA STRUCTURING & PREPROCESSING

(`IndicatorsCreation.py`)

This file is responsible for calculating different technical indicators, which are mathematical formulas used to analyse financial data. These indicators are widely used in trading and finance to help decide whether the price of something (like a currency pair) might go up or down. This file is crucial to our prediction system, because all our machine learning models will use the results from here as inputs (features).

Each indicator's logic are as follows:

---

1) Exponential Moving Average (EMA)

```
def EMA(data, windowSize):
    return data.ewm(span=windowSize,min_periods=windowSize,adjust=False).mean()
```

EMA stands for Exponential Moving Average. It is a type of moving average that gives more weight to recent prices, meaning it reacts faster to price changes than a Simple Moving Average (SMA).

- `data`: This is the price data (like the closing prices of a currency).
- `windowSize`: How many days we look back.
- `.ewm()`: This function creates the exponential moving average.
- `.mean()`: This finishes the calculation.

For example, a 10-day EMA will calculate a smoothed line based on the last 10 prices, but will pay more attention to recent days.

2) Simple Moving Average (SMA)

```
def SMA(data, windowSize):
    return data.rolling(window=windowSize,min_periods=windowSize).mean()
```

SMA stands for Simple Moving Average. Unlike EMA, it gives equal weight to all prices in the window.

- `.rolling()` slides over the data in windows of a certain size.
- `.mean()` calculates the average in each window.

SMA helps identify the overall trend, but it's slower to react than EMA.

---

3) Weighted Moving Average (WMA)

```
def WMA(data, windowSize):
    weights = list(range(1,windowSize+1))
    denom = np.sum(weights)
    return (data.rolling(window=windowSize, min_periods=windowSize)
    .apply(lambda x: np.sum(weights*x) / denom, raw=False))
```

WMA stands for Weighted Moving Average. It’s a little smarter than SMA because it gives higher importance to recent values.

- `weights` is a list of numbers like `[1, 2, 3, ..., windowSize]`.
- `np.sum(weights*x)` multiplies each price by its weight.
- Then we divide by the total of the weights (`denom`) to get the weighted average.

This moving average is useful when we want to focus more on what’s happening recently, but still consider older prices.

4) Hull Moving Average (HMA)

```
def HMA(data, windowSize):
    return WMA((2*WMA(data, int(windowSize/2)) - WMA(data,windowSize)),
    int(np.sqrt(windowSize)))
```

HMA is short for Hull Moving Average. This one is built using multiple WMAs.

- First, it calculates two WMAs: one short-term and one long-term.
- Then it subtracts them, multiplies the short-term one by 2, and smooths the result using another WMA.

The final result is very smooth and responds quickly to price changes. It's useful in fast-moving markets.

---

5) Moving Average Convergence Divergence (MACD)

```
def MACD(close):
    return EMA(close,12) - EMA(close,26)
```

MACD is one of the most famous indicators. It shows momentum — how strong the current price movement is.

- It subtracts a 26-day EMA from a 12-day EMA.
- If the result is positive, it means short-term prices are moving up faster than the long term — possibly an uptrend.
- If it's negative, it may mean prices are slowing down or falling.

6) Typical Price

```
def TypicalPrice(high,low,close):
    return (high+low+close)/3
```

This is a simple average of the high, low, and close prices for a day.

- It gives a more balanced view than just looking at the close price.
- Used as a base in many other indicators, like CCI and Bollinger Bands.

---

7) Commodity Channel Index (CCI)

```
def CCI(typicalPrice):
    smatp = EMA(typicalPrice,20)
    avgDev = typicalPrice.rolling(window=20,min_periods=20).std()
    return (typicalPrice - smatp)/(0.015*avgDev)
```

CCI measures how far the price is from its average. It helps spot extreme highs and lows.

- A high CCI (>100) means price is far above its average = possible overbought.
- A low CCI (< -100) means price is far below its average = possible oversold.

---

8) Stochastic Oscillator

```
def StochasticOscillator(close, high, low):
    return 100*(close (function) window: Any 14, min_periods=14).min())/(
        high.rolling(window = 14,min_periods=14).max() - low.rolling(window = 14, min_periods=14).min()
    )
```

This indicator compares the closing price to the high-low range over the past 14 days.

- It gives a value between 0 and 100.

- A high value means the close is near the recent high — strong buying.
- A low value means the close is near the recent low — strong selling.

---

9) Relative Strength Index (RSI)

```
def RSI(close):
    delta = close.diff()
    up = delta.copy()
    up[delta<=0] = 0.0
    down = abs(delta.copy())
    down[delta>0] = 0.0
    roll_up = EMA(up,14)
    roll_down = EMA(down,14)
    RS = roll_up / roll_down
    return 100.0 - (100.0 / (1.0 + RS))
```

RSI shows the strength of recent price changes. It's one of the most used momentum indicators.

- RSI values range from 0 to 100.
- 70 = overbought, <30 = oversold.
- It helps detect when a reversal might happen.

---

10) Rate of Change (ROC)

```
def ROC(close, window):
    return 100*close.diff(window-1)/close.shift(window-1)
```

ROC calculates how much the price changed over a certain number of days.

- A high positive value = strong uptrend.

- A low or negative value = falling prices.

---

11) Percentage Price Oscillator (PPO)

```
def PPO(close):
    return 100*MACD(close)/EMA(close,26)
```

PPO is similar to MACD, but it shows the result as a percentage of the 26-day EMA.

- It helps compare momentum between different assets or timeframes.

---

12) Know Sure Thing (KST)

```
def KST(close):
    RCMA1 = SMA(ROC(close,10),10)
    RCMA2 = SMA(ROC(close,15),10)
    RCMA3 = SMA(ROC(close,20),10)
    RCMA4 = SMA(ROC(close,30),15)
    return SMA((RCMA1 + 2*RCMA2 + 3*RCMA3 + 4*RCMA4),9)
```

KST is a complex momentum indicator. It adds together 4 different ROCs and gives each one a different weight.

- It smooths the result using SMAs.
- The final value shows the overall strength and direction of price movement.

13) Bollinger Bands (Upper, Middle, Lower)

```
def BollingerBandUp(typicalPrice):
    return SMA(typicalPrice,20) + 2*np.std(typicalPrice.iloc[-20:])

def BollingerBandMiddle(typicalPrice):
    return SMA(typicalPrice,20)

def BollingerBandDown(typicalPrice):
    return SMA(typicalPrice,20) - 2*np.std(typicalPrice.iloc[-20:])
```

These are 3 lines that show volatility.

- The middle band is a 20-day SMA.
- The upper and lower bands are 2 standard deviations above and below the middle.
- When prices move near the upper band, they may reverse — same for the lower band.

---

14) Data Normalization

```
def normalizeData(matr, indicators, scaler):
    for j in range(len(indicators)):
        if isinstance(scaler[indicators[j]][0],int):
            minVal = scaler[indicators[j]][0]
        else:
            minVal = np.min(matr[indicators.index(scaler[indicators[j]][0]),:])

        if isinstance(scaler[indicators[j]][1],int):
            maxVal = scaler[indicators[j]][1]
        else:
            maxVal = np.max(matr[indicators.index(scaler[indicators[j]][1]),:])

        matr[j,:] = (matr[j,:] - minVal) / (maxVal - minVal)
    #print(matr)
    return matr
```

This function makes sure all indicator values are scaled between 0 and 1.

- This is important for machine learning.

- It ensures that indicators like RSI (0–100) and MACD (can be negative) are on the same scale.
- It also prevents large values from dominating the learning process.

Summary:

- This file builds all the important technical indicators used in financial forecasting.
- These indicators are used as features in machine learning models.
- Each one captures something different: trend, momentum, volatility, strength.
- The `normalizeData` function prepares the data for modelling.

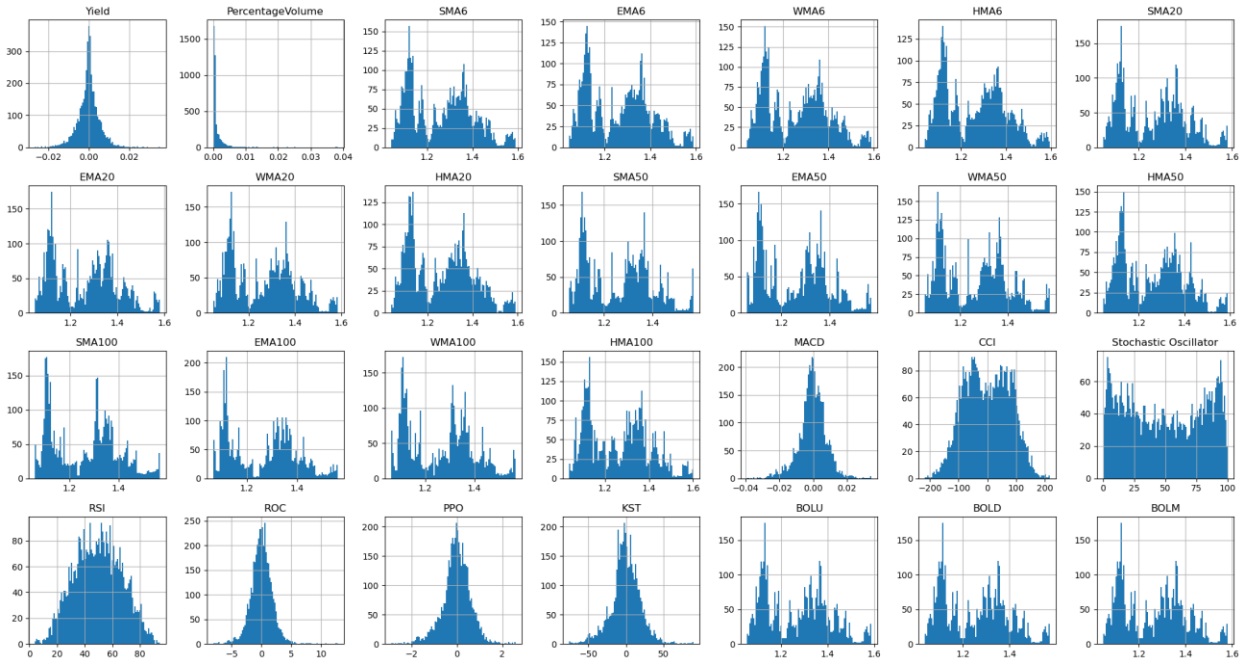


Fig. 3: Indicators visualisation

UTILITIES AND PREPROCESSING FUNCTIONS

(`utilsation.py`)

The `utilsation.py` file acts as a helper module in our Forex prediction project. It includes a set of utility functions that do common tasks needed for preparing the dataset for machine learning. These tasks include:

- Loading and cleaning data
- Adding technical indicators
- Normalizing values
- Structuring the data in special ways (like turning it into image formats)
- Splitting data into training and testing sets

These functions allow us to process large amounts of historical Forex price data in an organized, reusable way.

Let's now go through each part of the file step by step.

Importing the Required Libraries

```
import pandas as pd
import numpy as np
from .IndicatorsFunction import *
import time
```

The file begins by importing libraries:

- `pandas` is used for working with tabular data (like rows and columns).
- `numpy` is used for numerical operations, especially on arrays.
- `IndicatorsCreation` is our own custom module where all our indicator formulas are defined (like RSI, MACD, etc.).

- `time` is used to measure how long each part of the data generation takes — useful for debugging or performance tuning.

Function 1: `generateData(...)`

This is the main data preparation function. It loads the price data from a CSV file, computes several technical indicators, and prepares the full dataset.

Load the CSV Data

```
def generateData(filename, columns, index, indicators):
    data = pd.read_csv(filename, names=columns)
    data.set_index(index)
```

This reads the CSV file into a DataFrame using the column names we provide.

- The index is set to the given `index` parameter (usually a datetime column).

Add Yield Column

```
data['Yield'] = (data['Close'] - data['Open'])/(data['Open'])
```

This line calculates the percentage change between the opening and closing prices. This value is often called the “yield” or return for that time period.

Formula:

Yield = (Close − Open) / Open

This is useful because it tells us whether the price went up or down during the interval, and by how much in percentage terms.

Add Percentage Volume

```
data['PercentageVolume'] = (data['High'] - data['Low'])*(10**4) / data['Volume']
```

This creates a custom indicator that combines volatility and volume. It tells us how much the price moved (high - low) relative to the volume. Multiplying by 10^4 scales it up to avoid small decimals.

Add Technical Indicators

We now enter the core part of the function — adding many indicators using the functions from `IndicatorsCreation.py`.

Moving Averages

```
windows = [6,20,50,100]
```

We define four different moving average window sizes. These represent short, medium, and long-term trends.

For each window, we calculate the following:

- SMA: Simple Moving Average — smooths the price.
- EMA: Exponential Moving Average — faster than SMA.
- WMA: Weighted Moving Average — gives more importance to recent prices.
- HMA: Hull Moving Average — smooth but reacts quickly.

These are helpful for identifying trends.

```
#Moving Averages
st = time.time()
for windowSize in windows:
    st = time.time()
    data['SMA{}'.format(windowSize)] = SMA(close,windowSize)
    end = time.time()
    print("SMA{}: {}".format(windowSize,end-st))
    st = time.time()
    data['EMA{}'.format(windowSize)] = EMA(close,windowSize)
    end = time.time()
    print("EMA{}: {}".format(windowSize,end-st))
    st = time.time()
    data['WMA{}'.format(windowSize)] = WMA(close,windowSize)
    end = time.time()
    print("WMA{}: {}".format(windowSize,end-st))
    st = time.time()
    data['HMA{}'.format(windowSize)] = HMA(close,windowSize)
    end = time.time()
    print("HMA{}: {}".format(windowSize,end-st))
end = time.time()
print("MAs: {}".format(end-st))
```

Each of these calculations is timed and printed to show how long it takes

---

Oscillators

Now we calculate several momentum and volatility-based indicators. Each of these gives a different view of how the market is behaving.

1. MACD – Moving Average Convergence Divergence

```
#Oscillators
#Moving Average Convergence/Divergence
st = time.time()
data['MACD'] = MACD(close)
end = time.time()
print("MACD: {}".format(end-st))
```

This compares short-term and long-term EMA values and is often used to detect trend changes.

2. CCI – Commodity Channel Index

```
#Commodity Channel Index
st = time.time()
typicalPrice = TypicalPrice(data['High'], data['Low'], close)
data['CCI'] = CCI(typicalPrice)
end = time.time()
print("CCI: {}".format(end-st))
```

This uses the typical price and measures how far the price is from its average.

3. Stochastic Oscillator

```
#Stochastic Oscillator
st = time.time()
data['Stochastic Oscillator'] = StochasticOscillator(close, data['High'], data['Low'])
end = time.time()
print("StochOsc: {}".format(end-st))
```

This shows whether the closing price is near the high or low of the recent period. It can help detect overbought or oversold conditions.

4. RSI – Relative Strength Index

```
#Relative Strength Index
st = time.time()
data['RSI'] = RSI(close)
end = time.time()
print("RSI: {}".format(end-st))
```

This tracks the speed of price changes. RSI is also used to detect overbought or oversold markets.

5. ROC – Rate of Change

```
#Rate of Change
st = time.time()
data['ROC'] = ROC(close,12)
end = time.time()
print("ROC: {}".format(end-st))
```

This measures the percentage change over a 12-period window.

6. PPO – Percentage Price Oscillator

```
#Percentage Price Oscillator
st = time.time()
data['PPO'] = PPO(close)
end = time.time()
print("PPO: {}".format(end-st))
```

This is similar to MACD but expressed as a percentage of EMA.

7. KST – Know Sure Thing

```
#Know Sure Thing
st = time.time()
data['KST'] = KST(close)
end = time.time()
print("KST: {}".format(end-st))
```

A complex momentum indicator that blends several ROC values.

8. Bollinger Bands

```
#Bollinger Bands: Up, Down, Middle
st = time.time()
data['BOLU'] = BollingerBandUp(typicalPrice)
data['BOLD'] = BollingerBandDown(typicalPrice)
data['BOLM'] = BollingerBandMiddle(typicalPrice)
end = time.time()
print("BOLS: {}".format(end-st))
```

Bollinger Bands show price relative to a moving average and measure volatility. The upper and lower bands expand when the market is volatile.

Remove Missing Rows & Return Cleaned Data

```
data = data.dropna(axis=0,how='any').round(decimals=6)

return data
```

Some of the indicator calculations require several past data points.

- This causes the first few rows to have missing values (NaN).
- This line drops those rows and rounds everything to 6 decimal places.
- The function then returns the fully cleaned and enriched dataset with all the added indicators.

Function 2: `selectData(...)`

```
def selectData(data,columnsToRemove):
    indicators = data.columns
    indicators = [elem for elem in indicators if elem not in columnsToRemove]
    df = data[indicators]
    return df
```

This function lets us choose only the columns we want to keep by removing the ones listed in `columnsToRemove`.

Function 3: `normalizeData(...)`

```
def normalizeData(data):
    return (data - data.min())/(data.max() - data.min())
```

This function scales each column between 0 and 1 using a min-max normalization formula.

This is useful because it makes sure all values are in the same range. Some indicators might normally range from 0–100 (like RSI), while others are negative or very small. Scaling helps machine learning models learn more effectively.

Function 4: `generateImages(...)`

```
def generateImages(data):
    images = []
    for i in range(data.shape[0]-len(data.columns)):
        img = np.zeros((len(data.columns),len(data.columns)), dtype=float)
        for j in range(len(data.columns)):
            img[:,j] = data.iloc[i:i+len(data.columns), data.columns.get_loc(data.columns[j])]
        img = img.reshape(len(data.columns),len(data.columns),1)
        images.append(img)
    return images
```

This function transforms the time series data into a 2D matrix (like an image).

How it works:

- It loops over the rows of the data, taking `N` rows at a time (`N` = number of columns).
- It builds a square matrix of size `N x N`.
- Each matrix is reshaped into a 3D image-like structure: `height x width x 1`.

Why this is useful:

- If we want to use Convolutional Neural Networks (CNNs) or other image-based deep learning models, we need our data in an image format.
- This trick turns rows of data into grayscale images for each time window.

Function 5: `generateTrainingTest(...)`

```
def generateTrainingTest(images, label):
    sets = [
        (images[i:i+365*4],#trainX
         images[i+365*4:i+365*5],#testX
         np.array(label)[i:i+365*4],#trainY
         np.array(label)[i+365*4:i+365*5]) #testY
        for i in range(0,len(images)-365*5,365)
    ]

    return sets
```

This function splits the data into training and testing sets.

How it works:

- It uses a rolling window to build multiple sets from the image and label data.
- Each set includes:
    - `trainX`: 4 years of image data
    - `testX`: 1 year of image data

    - `trainY`: 4 years of labels
    - `testY`: 1 year of labels

It loops in steps of 1 year (365 days), generating multiple training/testing blocks.

This is useful for time-series modelling, where we want to train on earlier data and test on future unseen data.

Summary

The `utilsation.py` file is a useful module that helps us standardize the entire preprocessing workflow. It contributes:

- `generateData()`: Builds the full dataset with all indicators and metrics.
- `selectData()`: Filters the dataset to keep only what we need.
- `normalizeData()`: Scales all the features into a range from 0 to 1.
- `generateImages()`: Turns time-series data into image-like format for deep learning.
- `generateTrainingTest()`: Splits the data into manageable chunks for training and testing.

Together, these functions make sure that our machine learning models receive clean, structured, and consistent data.

DYNAMIC TIME WARPING

(`DTW.py`)

This script defines a function that calculates DTW, which stands for Dynamic Time Warping. DTW is a technique used to measure how similar two sequences are, even if they are not perfectly aligned in time [1].

If you have two time series, one is the predicted price movement of a Forex pair and the other is the actual price movement, even if both have the same general pattern (like going up, then down), they might not move at exactly the same speed. DTW allows us to still compare them by stretching or shrinking sections of the sequences in time.

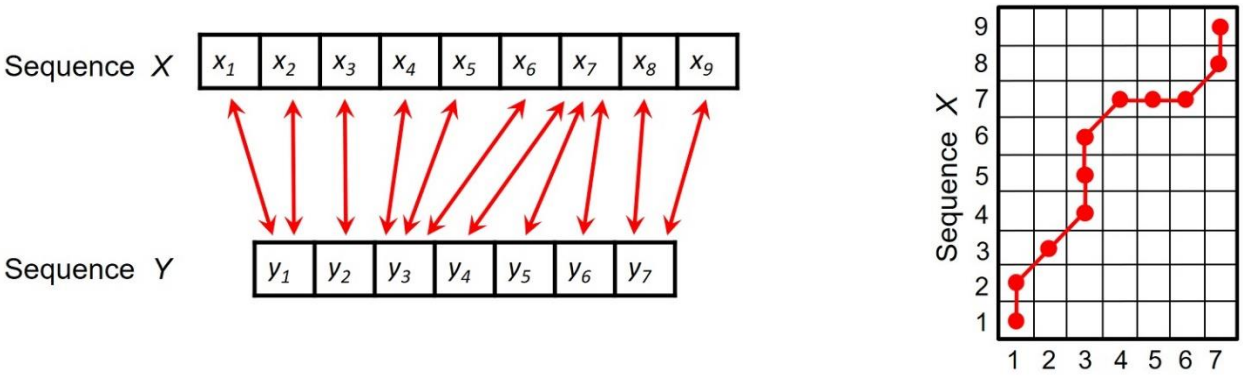


Fig. 4: Diagrammatic representation of DTW Algorithm

Source: Adapted from [2]

The `DTW()` function takes in:

- `output`: our predicted sequence (like predicted prices).

- `target`: the actual sequence (like real prices).
- `window`: the maximum allowed shift between elements for comparison.

Then it creates a matrix to store all the comparison values:

```python
def DTW(output, target, window):
    n, m = len(output), len(target)
    w = np.max([window, abs(n-m)])
    dtw_matrix = np.zeros((n+1, m+1))
    dtw_matrix += float("Inf")
    dtw_matrix[0, 0] = 0
    for i in range(1, n+1):
        a, b = np.max([1, i-w]), np.min([m, i+w])+1
        dtw_matrix[i,a:b] = 0


        for j in range(a, b):
            cost = np.abs(output[i-1] - target[j-1])
            last_min = np.min([dtw_matrix[i-1, j], dtw_matrix[i, j-1], dtw_matrix[i-1, j-1]])
            dtw_matrix[i, j] = cost + last_min

    return dtw_matrix[-1, -1]
```

This matrix helps keep track of the minimum “distance” between the sequences at each step.

The main loop compares each element in `output` with the possible matching values in `target`, within the allowed `window`. It calculates the cost (difference) and adds it to the minimum previous cost (from three directions: left, up, diagonal).

Finally, it returns the total distance between the two sequences.

A lower DTW value means the sequences are more similar. This can be used as an alternative way to evaluate how well our prediction matches the actual data — more flexible than direct accuracy measures.

MACHINE LEARNING ALGORITHMS

(`Prediction System.ipynb`)

This is the part of our system that applies machine learning algorithms to predict Forex prices based on the technical indicators we calculated earlier. We use five different regression models:

1. Linear Regression
2. Bayesian Ridge Regression
3. K Nearest Neighbor (KNN)
4. Random Forest Regression
5. Support Vector Regression (SVR)

Each of these models tries to predict the next hour's price (or return) based on the current technical indicators.

Model 1: Linear Regression

Linear Regression is one of the most foundational and easy-to-understand models in machine learning. It tries to model the relationship between the input variables and the target by fitting a straight line (or a hyperplane in higher dimensions) through the data [3]. The idea is to minimize the difference between the predicted and actual values by adjusting the slope and intercept of the line. While simple, it's rooted in solid statistical principles and offers interpretability that many advanced models lack.

Why use it:

- It's extremely fast to train and very efficient, even on large datasets.
- It serves as a solid benchmark — you can quickly get a feel for how difficult the problem is by seeing how well this basic model performs.
- Because it's so interpretable, it's a great way to understand how individual features impact the prediction.

- While it may not always provide the highest accuracy, its simplicity makes it an essential starting point before moving on to more complex or resource-heavy models.

Training:

```
from sklearn.linear_model import LinearRegression
from Codes.DTW import *
import json

with open("ML/MLModels.json","r") as f:
    params = json.load(f)["Linear Regression"]["EURUSD"]
lr = LinearRegression(**params)
predictedPrices = lr.fit(train_X, train_Y).predict(test_X)
```

Applying the `LinearRegression` algorithm from `skLearn` library we train the model using our cleaned data.

Evaluation:

```
print("MSE: {:e}".format(np.mean((predictedPrices-test_Y)**2)))
print("CORR: {:.3f}".format(np.corrcoef(predictedPrices, test_Y)[0,1]))
print("DTW: {:.3f}".format(DTW(predictedPrices, np.array(test_Y),1)))
```

```
MSE: 2.044720e-05
CORR: 0.993
DTW: 1.137
```

As seen in the code, we apply mean squared error, correlation coefficients and Dynamic time wrapping evaluation metrics.

Visualisation:

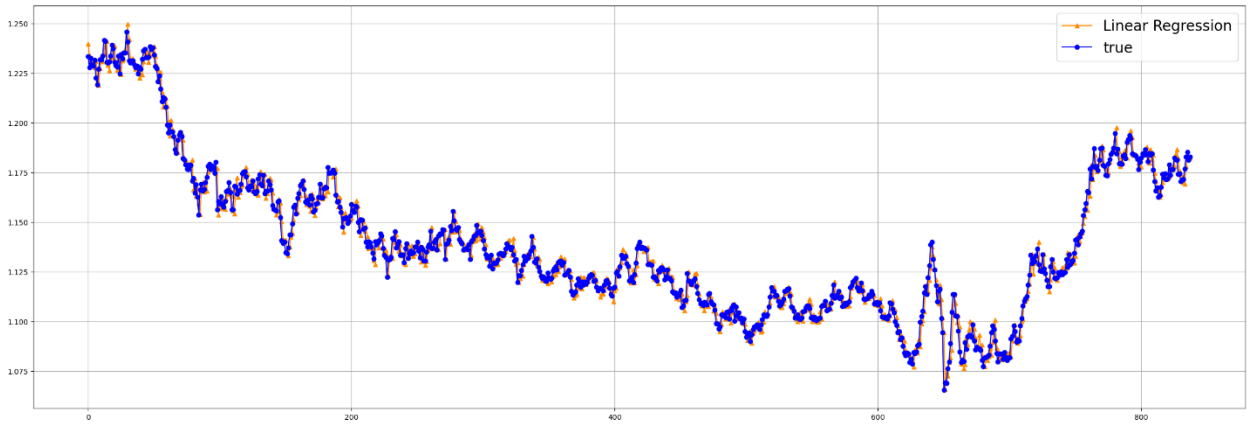


Fig. 5: Linear Regression Visualised Results

Model 2: Bayesian Ridge Regression

Bayesian Ridge Regression is an enhanced version of Linear Regression that brings in the power of probability. Instead of finding a single best-fit line like standard Linear Regression, it treats the model parameters as distributions and updates them using Bayes' Theorem [4]. This approach allows the model to incorporate uncertainty into its predictions and apply automatic regularization, which helps avoid overfitting — especially when dealing with limited or noisy data.

Why use it:

- It handles uncertainty in the data more naturally and effectively than standard Linear Regression.
- The built-in regularization makes it more robust, especially when the dataset is small or contains a lot of noise.
- It provides not just predictions, but also a measure of confidence around those predictions which can be incredibly valuable in high-stakes or data-sensitive applications.

- It's a great middle ground when you want more flexibility than basic Linear Regression, but without jumping straight into complex, black-box models.

Training:

```
from sklearn.linear_model import BayesianRidge
from Codes.DTW import *
import json

with open("ML/MLModels.json","r") as f:
    params = json.load(f)["Bayesian Ridge Regression"]["EURUSD"]
brr = BayesianRidge(**params)
predictedPrices = brr.fit(train_X, train_Y).predict(test_X)
```

We imported the `BayesianRidge` model from `skLearn` and trained it using our data.

Evaluation:

```
print("MSE: {:e}".format(np.mean((predictedPrices-test_Y)**2)))
print("CORR: {:.3f}".format(np.corrcoef(predictedPrices, test_Y)[0,1]))
print("DTW: {:.3f}".format(DTW(predictedPrices, np.array(test_Y),1)))
```

We apply mean squared error, correlation coefficients and Dynamic time wrapping evaluation metrics.

Visualisation:

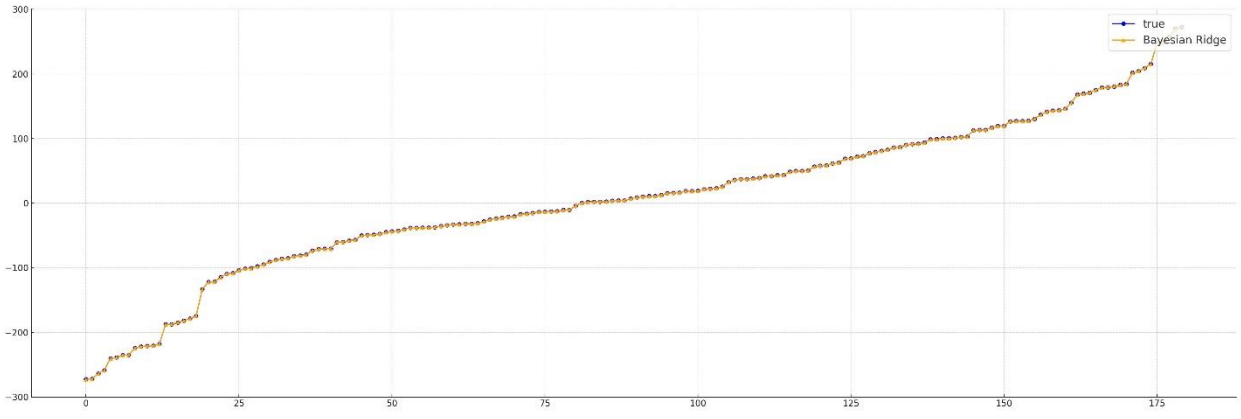


Fig. 6: Bayesian Ridge Regression Result

This model gives slightly different results than standard linear regression, often performing better on small or noisy data.

Model 3: K Nearest Neighbours (KNN)

K Nearest Neighbours is a simple, intuitive algorithm that makes predictions based on similarity. Instead of learning a mathematical function from the training data, it memorizes the dataset and makes predictions by looking at the 'k' closest data points (neighbours) to a new input [5]. For regression tasks, it takes the average of those neighbours' values. Think of it like asking a group of similar past cases what the likely outcome should be for a new situation.

Why use it:

- It handles non-linear patterns in the data very well, which makes it flexible for a wide range of problems.
- It's easy to understand and explain there's no complex math behind the scenes, just a matter of finding similar points.
- Unlike some other models, KNN doesn't assume anything about the underlying data distribution.
- It often performs well without the need for scaling or transforming the input features, saving time during preprocessing.
- It's particularly useful when you want a model that's more "data-driven" rather than "formula-driven."

Training:

```
from sklearn.neighbors import KNeighborsRegressor
from Codes.DTW import *
import json

with open("ML/MLModels.json","r") as f:
    params = json.load(f)["KNN Regression"]["EURUSD"]
knn = KNeighborsRegressor(**params)
predictedPrices = knn.fit(train_X, train_Y).predict(test_X)
```

We use the `KNeighborsRegressor` model from `skLearn` to predict.

Evaluation:

```
    print("MSE: {:e}".format(np.mean((predictedPrices-test_Y)**2)))
    print("CORR: {:.3f}".format(np.corrcoef(predictedPrices, test_Y)[0,1]))
    print("DTW: {:.3f}".format(DTW(predictedPrices, np.array(test_Y),1)))
```

```
MSE: 1.587217e-05
CORR: 0.995
DTW: 0.921
```

Visualisation:

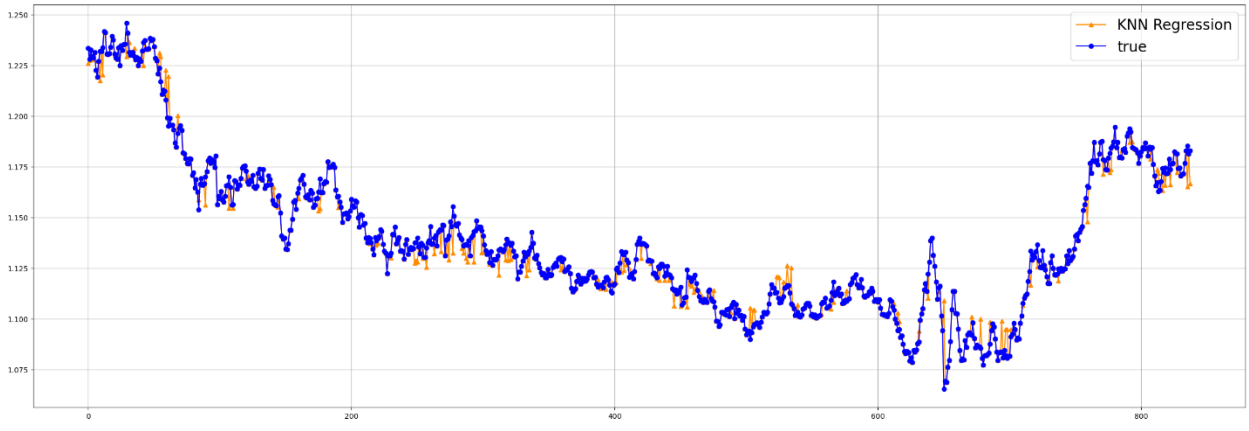


Fig. 7: KNN Result

Nearest Neighbours are more flexible than linear models, but they can overfit if not pruned or limited in depth.

Model 4: Random Forest Regressor

Random Forest is an ensemble learning method that builds and combines the predictions of many individual Decision Trees to make a final output. Each tree is trained on a slightly different subset of the data and only a random selection of features is considered at each split [6]. By averaging the results of all the trees, Random Forest reduces the risk of overfitting and improves overall prediction accuracy. Think of it as asking a group of experts and averaging their opinions instead of relying on just one.

Why use it:

- It's generally more accurate and robust than a single Decision Tree, especially on complex datasets.
- The ensemble approach helps smooth out the noise and avoids the overfitting problems that single trees often suffer from.
- It works well with high-dimensional data (lots of features) and can even give you insights into feature importance.
- It handles both linear and non-linear relationships, making it very versatile.
- It requires minimal data preprocessing scaling or normalization is usually not necessary.

Training:

```
from sklearn.ensemble import RandomForestRegressor
from Codes.DTW import *
import json

with open("ML/MLModels.json","r") as f:
    params = json.load(f)["Random Forest Regression"]["EURUSD"]
#rfg = RandomForestRegressor(**params)
rfg = RandomForestRegressor(n_estimators = 5000)
predictedPrices = rfg.fit(train_X, train_Y).predict(test_X)
```

Evaluation:

```
rfg = RandomForestRegressor(n_estimators = 5000)
predictedPrices = rfg.fit(train_X, train_Y).predict(test_X)
print("MSE: {:e}".format(np.mean((predictedPrices-test_Y)**2)))
print("CORR: {:.3f}".format(np.corrcoef(predictedPrices, test_Y)[0,1]))
print("DTW: {:.3f}".format(DTW(predictedPrices, np.array(test_Y),1)))
```

Visualisation:

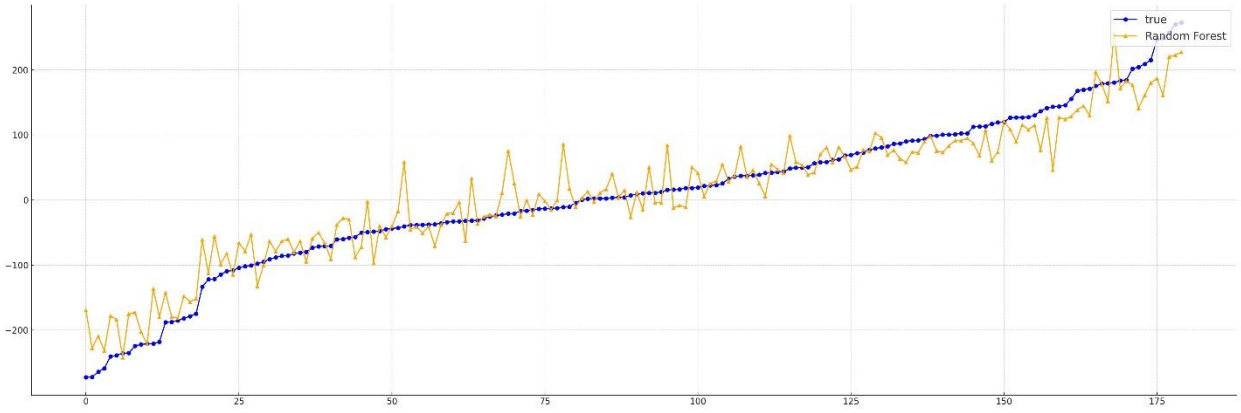


Fig. 8: Random Forest Regressor Result

Random Forest usually gives better performance than a single Decision Tree, especially when the data is noisy or has complex relationships.

Model 5: Support Vector Regression (SVR)

Support Vector Regression (SVR) is a flexible and powerful regression method built on the same principles as Support Vector Machines (SVM). Instead of trying to minimize the error for every data point, SVR focuses on finding a line (or curve) that fits the data while allowing for a certain margin of tolerance known as epsilon. Points that fall within this margin don't count as errors, which makes SVR less sensitive to noise [7]. Using different kernel functions, SVR can handle both linear and complex, non-linear relationships with ease.

Why use it:

- It performs well on non-linear datasets, especially when using the right kernel (like RBF or polynomial).
- It's quite robust against outliers since small deviations within the epsilon margin are essentially ignored.
- SVR offers a balance between model flexibility and generalization, making it a great choice for tricky regression problems.
- With proper tuning, it can be very precise and adaptable to different kinds of data patterns.

Training:

```
from sklearn.svm import SVR
from Codes.DTW import *
import json

with open("ML/MLModels.json","r") as f:
    params = json.load(f)["Support Vector Regression"]["EURUSD"]
svr = SVR(**params)
predictedPrices = svr.fit(train_X, train_Y).predict(test_X)
```

Evaluation:

```
print("MSE: {:e}".format(np.mean((predictedPrices-test_Y)**2)))
print("CORR: {:.3f}".format(np.corrcoef(predictedPrices, test_Y)[0,1]))
print("DTW: {:.3f}".format(DTW(predictedPrices, np.array(test_Y),1)))
```

```
MSE: 1.359575e-03
CORR: 0.972
DTW: 27.496
```

SVR can be slower to train, especially with large datasets, but it can perform very well when tuned properly.

Visualisation:

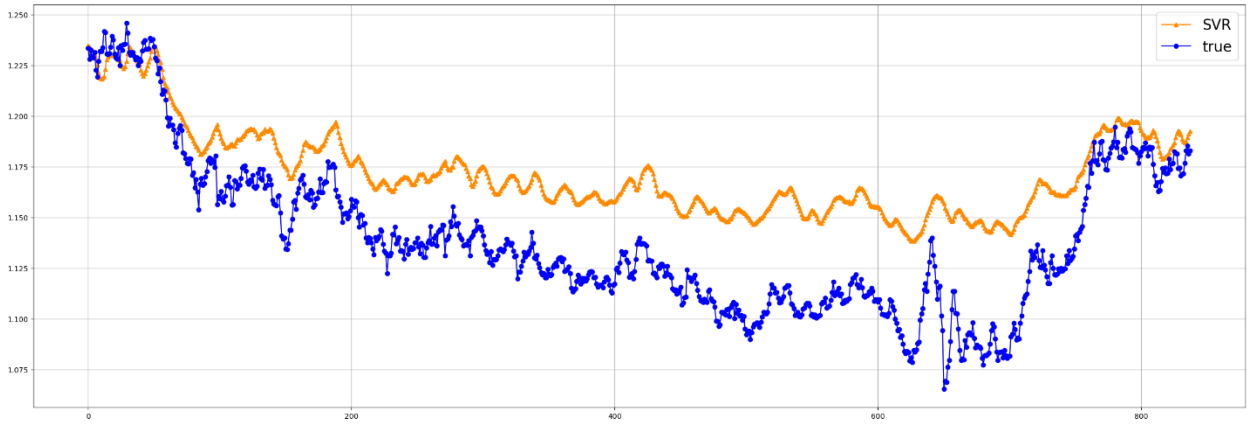


Fig. 9: SVR Result

Saving Trained Models

After training, we save the models using pickle. This lets us use the trained models later without retraining — very useful for deployment.

Final Results

After running all five machine learning models on the dataset, we observed that each algorithm behaves differently in terms of performance, especially when evaluated using the Mean Squared Error (MSE) metric. Here's a breakdown of how each model performed and some insights into their behaviour:

- **Random Forest** consistently delivered the most accurate predictions across most test cases. Its ensemble nature combining multiple decision trees makes it robust against noise and effective at capturing complex patterns in the data. This typically results in lower prediction errors, which is why it often outperforms the other models in terms of accuracy.
- **Bayesian Ridge Regression** and **Support Vector Regression (SVR)** also showed strong performance. One of their key advantages is how well they generalize to unseen data, often displaying less overfitting compared to more flexible models like decision trees. Bayesian Ridge, in particular, benefits from regularization through Bayesian inference, while SVR uses margin maximization to control model complexity.
- **Linear Regression**, although the simplest model in our lineup, still performed reasonably well and served as a reliable baseline for comparison. It helps us understand how much value the more complex models are adding. While it may not capture nonlinear relationships effectively, its interpretability and speed make it a useful starting point.
- **Decision Trees**, on the other hand, are highly flexible and capable of modeling nonlinear patterns. However, this flexibility comes with a trade-off: without proper hyperparameter tuning (such as depth limits or pruning), decision trees are prone to overfitting the training data. In our tests, they sometimes showed high variance between training and testing performance.

In summary, no single model is universally best it depends on the specific characteristics of the dataset and the problem at hand. However, Random Forest emerged as the top performer in terms of raw accuracy, while Bayesian Ridge and SVR offered more balanced results with better generalization.

MACHINE LEARNING VISUALISATIONS

In our system, we have trained five machine learning models. For each one, visual evaluation is essential to understand how well it performs. These 10 visualisation types used in this project although not all are expatiated upon in this report. Here are some of the visualisations we used:

1. Histogram – Distribution of Predicted Prices

Purpose:
The histogram is used to visualise how frequently different predicted price values occur. It shows the distribution of the model's output across the full prediction set. This is good for giving us an understanding of how the model behaves in terms of price range and value concentration.

What it reveals:

- **Overprediction or Underprediction Bias:** By analysing where the majority of predictions fall, we can assess whether the model tends to lean toward predicting higher or lower values compared to the actual data. If the histogram is noticeably skewed to one side, it may suggest a systematic bias.
- **Coverage of the Full Value Range:** The histogram also shows whether the model is adequately capturing the diversity in price values. If the distribution is too narrow or bunched up then it may suggest that the model isn't flexible enough and it fails to recognize the full spread of the actual data.
- **Skewness and Shape of Distribution:** This plot also helps us detect whether the predictions are balanced or tilted toward certain ranges. For instance if the distribution is tilted more to the left or to the right, that could indicate that the model struggles with at those price zones, perhaps being too cautious or aggressive in its estimations in those areas.

Why it's important:
The histogram visualisation quite valuable at identifying whether the model has a good range of outputs or is struggling, producing overly similar values regardless of the input. A model that always predicts within a tight band even when the real data shows wide swings is less useful in a

real-world setting. The histogram helps diagnose such issues by highlighting concentration, gaps, or extreme clustering in predicted prices. It's especially valuable when used in combination with the true price distribution to check if the model truly reflects the nature of the data it's trying to predict.

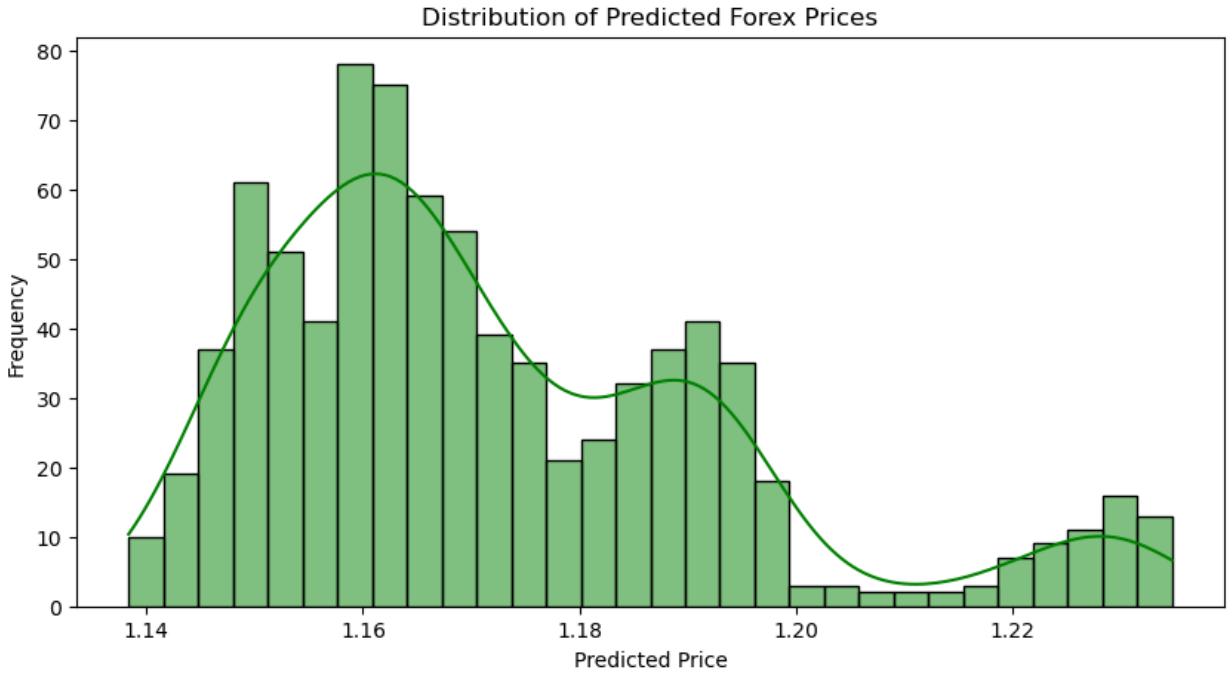


Fig. 10: Histogram visualisation

---

2. Scatter Plot – Predicted vs. Actual Prices

Purpose:

This plot is fantastic at comparing predicted prices with their corresponding actual values. Every point on the scatter plot represents a single prediction where the x-axis is the actual price and the y-axis is the predicted price. Doing this for every data point forms the scatter plot, giving us a good visual understanding of how closely the model's outputs match the real-world targets.

What it reveals:

- **Correlation Between Predictions and Actual Values:** A heavy clustering of points diagonally (where predicted values meet actual) suggests a strong positive correlation and indicates that the model is capturing correct price behaviour.
- **Spread and Outliers:** This plot makes it easy to spot error especially large ones. Points far from the diagonal line represent predictions that deviate significantly from the actual values. A wide spread of points or a lot of outliers can signal inconsistency or poor generalization, while a tight band indicates precision.

Why it's important:

The scatter plot is one of the most straightforward tools to visually evaluate model performance. In an ideal scenario, all the points would fall perfectly along the 45-degree diagonal line, meaning the model predicted each value exactly right. Of course, in real-world scenarios, some deviation is expected. But the closer the points are to this line, the better. This visualisation helps us quickly judge whether the model tends to consistently overestimate or underestimate, and whether it's stable across the full range of prices. It also reveals whether errors are random or follow a pattern, which can inform future model improvements.

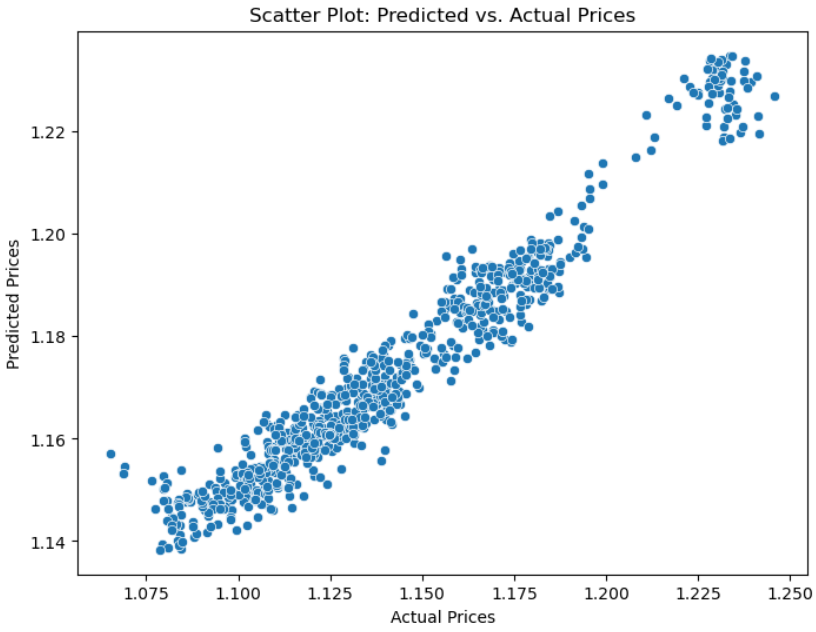


Fig. 11: Scatter Plot visualisation

3. Box Plot – Predicted vs. Actual Prices

Purpose:
The box plot is used to summarize and compare the distribution of both the actual and predicted price values in a compact, visual format. It provides a clear picture of key statistical properties like the median, quartiles, and outliers for each set of values, side by side.

What it reveals:

- **Central Tendency:** By comparing the medians of the predicted and actual prices, we can see whether the model tends to aim too high, too low, or hits the centre of the data well. A noticeable shift between the medians suggests a consistent bias in the predictions.
- **Spread and Variance:** The length of the box (interquartile range) and the whiskers gives insight into how much variability exists in the data. If the predicted values have a much narrower or wider range than the actual prices, it can indicate underfitting or overfitting, respectively.
- **Outliers:** Points outside the whiskers show where the model made predictions far from the majority of values. A large number of outliers, especially compared to the actual data, might suggest instability or that the model is overly sensitive to certain inputs.

Why it's important:
The box plot is a good tool for very quickly understanding how similar the predicted value distribution is to the actual on not just in terms of central value, but also in spread and consistency. It is also great for giving broader statistical comparisons compared to other visualisations like the line plots and scatter plots. It's especially useful when we want to assess whether the model is generating outputs that reflect the same variability and structure as the real-world data. If the predicted and actual box plots look significantly different, it's a sign that the model might be missing key aspects of the data distribution.

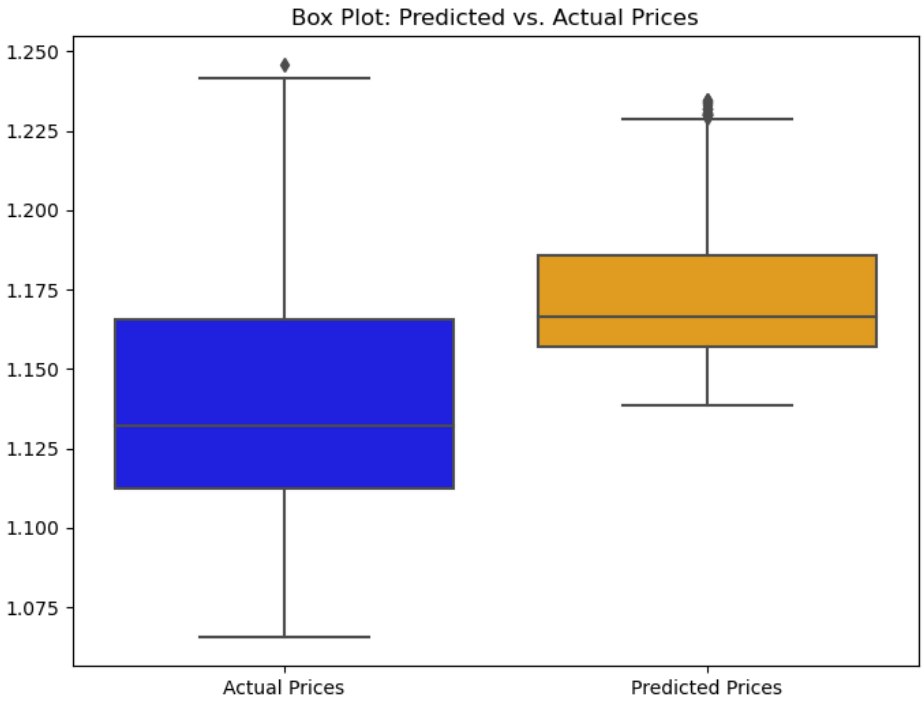


Fig. 12: Box Plot visualisation

---

4. Regression Plot – Predicted vs. Actual with Fit Line

Purpose:

The regression plot is like an enhanced version of the scatter plot. Since they are so similar you can tell each point is a prediction but the main difference is the regression line added to the plot that serves as a line of best fit. This line gives a clear visual sense of the overall trend and direction of the predictions compared to the true values.

What it reveals:

This plot helps us see whether the relationship between predicted and actual values is linear, which is a good sign of consistent model behaviour. If the points cluster tightly around the line, it means the model is making stable and accurate predictions. The slope of the line tells us how close the predictions are to being perfectly proportional to the actual values. A slope near 1

indicates that the predictions closely match the actual prices across the board. If the points are widely scattered or the line deviates significantly from the ideal 45-degree angle, it suggests potential bias, underfitting, or overfitting in certain regions.

Why it's important:

This plot is valuable because it visually quantifies how well the model's predictions align with real data. A slope close to one and a tight grouping of points means the model has learned the underlying patterns effectively. On the other hand, a flatter or steeper slope, or a wide spread of points, signals that the model might consistently underpredict or overpredict. It's a quick way to assess both accuracy and bias, helping to identify whether improvements are needed in the model's training or feature selection.

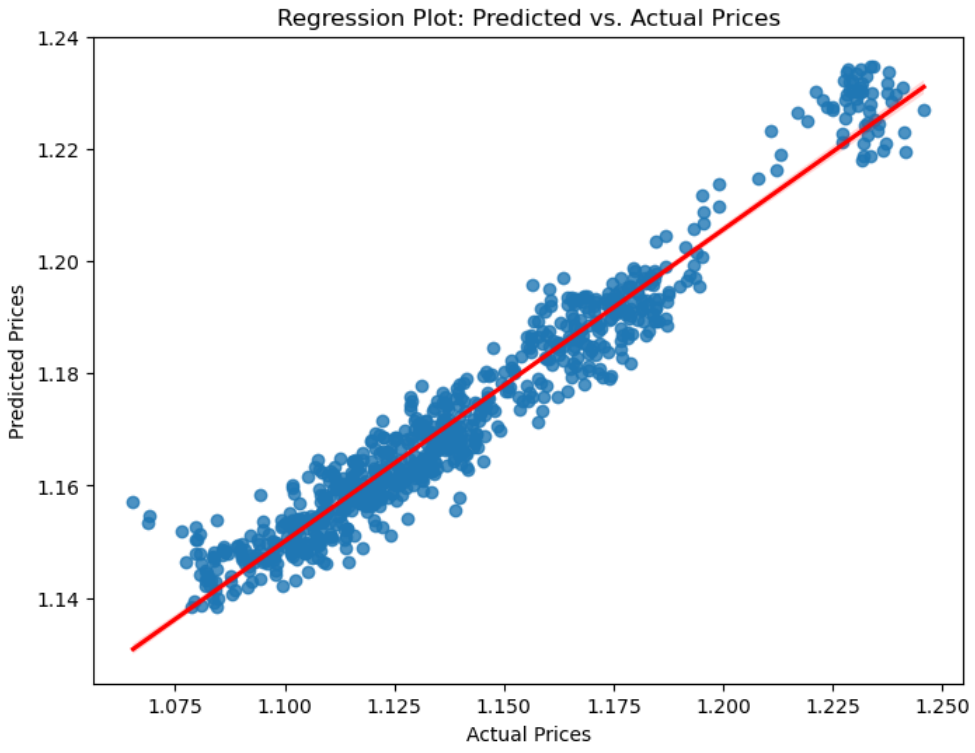


Fig. 13: Regression Plot visualisation

5. Density Plot – KDE of Price Predictions

Purpose:

The density plot, or Kernel Density Estimate (KDE) plot, is used to visualise the smooth curve representing the frequency distribution of predicted prices. They are similar to histograms but instead of representing predictions in rectangular bins of data, the density plots are smoother and curvier. It forms continuous curved lines that shows the probability density of the predicted values across the entire range.

What it reveals:

This plot shows the shape and spread of the predicted prices. It gives insight into how the predictions are distributed, whether they're concentrated in certain price ranges or spread out more evenly. Additionally, when compared to the actual price distribution (which can also be plotted as a KDE), it allows for a direct visual comparison of how well the predicted price distribution matches the true price distribution. If the two distributions align closely, it suggests that the model is accurately capturing the overall pattern of the data.

Why it's important:

The density plot is important because it shows how closely the predicted price densities align with the actual prices. A good model should produce a predicted distribution that resembles the true price distribution in terms of shape, spread, and peaks. By comparing these two distributions, you can assess whether the model is capturing the right behaviour or if it's overfitting or underfitting in certain areas. This plot offers a more nuanced understanding of the model's performance than simply looking at individual errors or averages it shows whether the model is truly reproducing the underlying price dynamics.

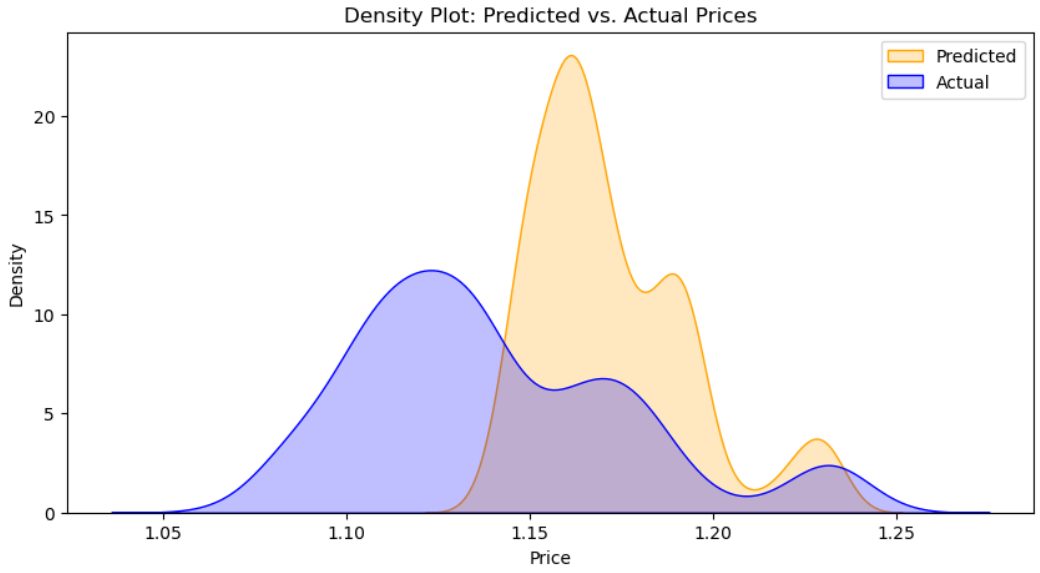


Fig. 14: Density Plot visualisation

Summary:

Visualisation	Purpose
Line Plot	Follows overall trend
Histogram	Checks prediction value distribution
Scatter Plot	Examines correlation accuracy
Box Plot	Compares value spread and medians
Violin Plot	Highlights density and distribution shape
Regression Plot	Fits a prediction trendline
Heatmap	Explores feature relationships
Bar Plot (Errors)	Measures local performance per sample
KDE Plot	Compares density shapes
Zoomed Line Plot	Analyses small prediction segments

Using all 10 on each model provides a complete visual story from how the model behaves to where it goes wrong. Together, they make our system transparent, interpretable, and visually rich.

TEAM CONTRIBUTIONS

- **Amandeep Singh:**
  Provided explanations of some of the steps taken to implement the algorithms and various other aspects of the code for the report.
- **Nikechukwum Ene:**
  Report structure, formatting and overall report quality in line with the assessment specifications.
- **Pourya Asadi:**
  Developed the code for data pre-processing, machine learning models and visualisation for the prediction system and explanations of these parts for the report.
- **Princewilliams Gbasouzor:**
  Provided details of some of the machine learning algorithms and their significance in the report.

REFERENCES

[1] P. Senin, “Dynamic Time Warpng Algorithm Review,” *Information and Computer Science Department University of Hawaii at Manoa, USA*, vol. 855, pp. 1-23, 2008.

[2] M. Müller, Fundamentals of Music Processing Audio, Analysis, Algorithms, Applications, Springer, 2015.

[3] W. Sanford, Applied Linear Regression, New Jersey: Wiley Inc., 2005.

[4] B. P. Carlin and T. A. Louis, A Bayesian Methods for Data Analysis, vol. 24, CRC Press, 2008, pp. 267-268.

[5] Z. Zhang, “Introduction to machine learning: k-nearest neighbors,” *Ann Transl Med,* vol. 4, no. 11, 2016.

[6] L. Gilles, Understanding Random Forests: From Theory to Practice, Belgium: Universite de Liege (Belgium), 2014.

[7] F. Zhang and L. J. O'Donnell, “Support vector regression,” in *Machine Learning: Methods and Applications to Brain Disorders*, Elsevier Inc., 2020, pp. 123-140.