Bull's Eye

An Intelligent Trading Tool
Powered by Deep Learning for Forex Markets

Presented By:

- **Ananya Sachan (22070126010)**
- **Aparna Iyer (22070126017)**
- **Hevardhan S. (22070126046)**
- **Riya Shukla (22070126090)**

Problem Statement

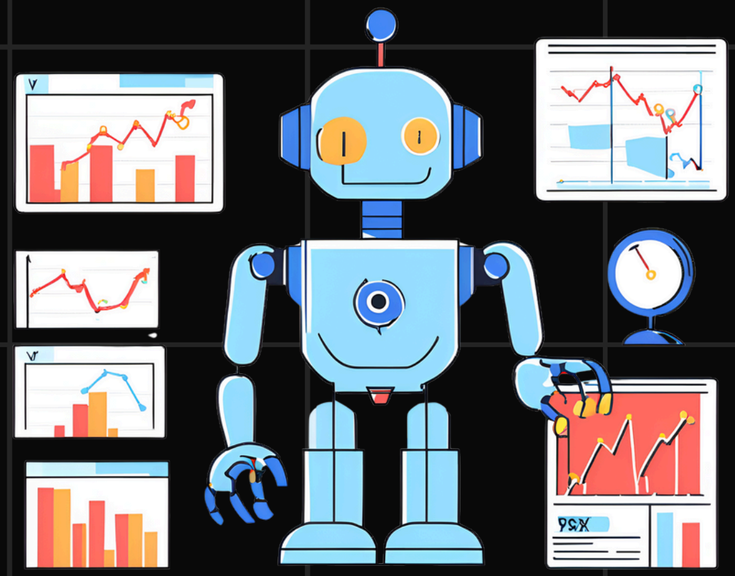
‘Bull’s Eye’ uses Deep Learning to predict optimal stock sale points, maximizing gains and minimizing losses in volatile markets.

It identifies ideal ‘buy’ and ‘sell’ points, capturing price dependencies and enhancing trend recognition.

Objectives

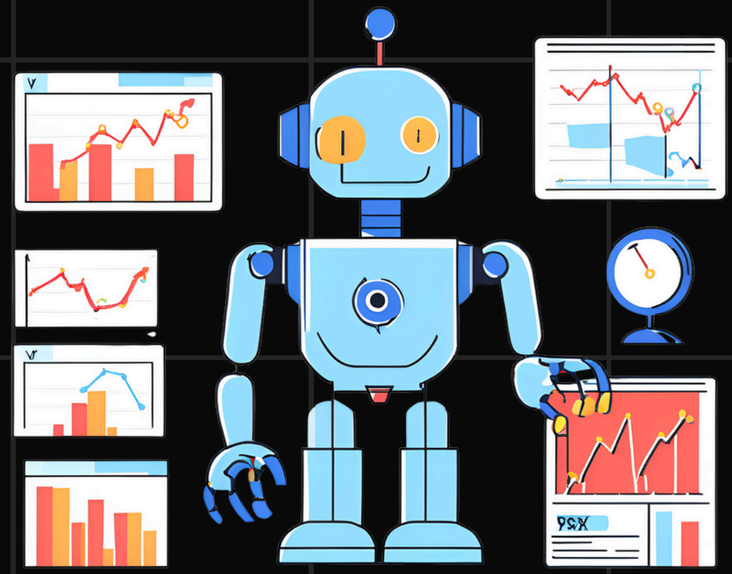
BullsEye focuses on a comprehensive set of objectives, including:

- Risk Management
- Reduce Human Error
- Automate Trading Decisions
- Optimized Trading Strategies



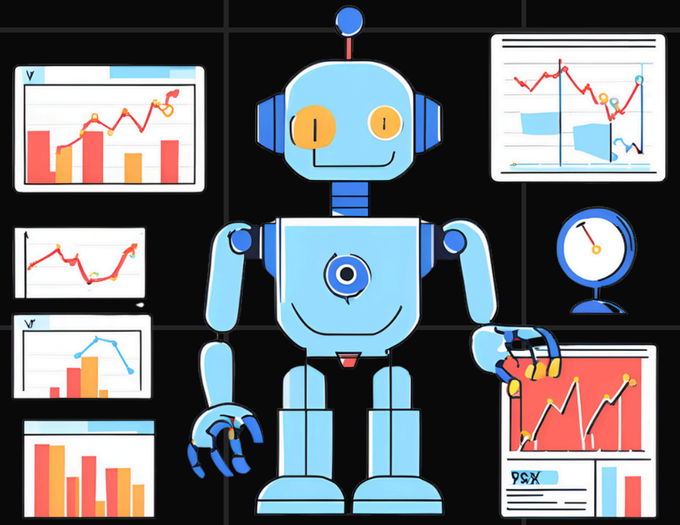
Software Requirements

- OS : Windows/Linux/Mac
- Python : 3.7 version
- MetaTrader 5
- Libraries:Numpy & Pandas Pytorch Tensorflow MetaTrader5
- IDE : VS Code.



Hardware Requirements

- Processor : i5 or higher
- Memory : 8 GB or higher
- Storage : 25 GB
- Network : Stable Internet Connection for real-time trading and data retrieval.

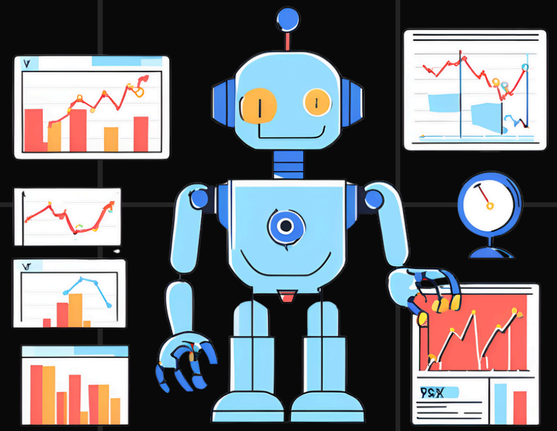


Dataset Information

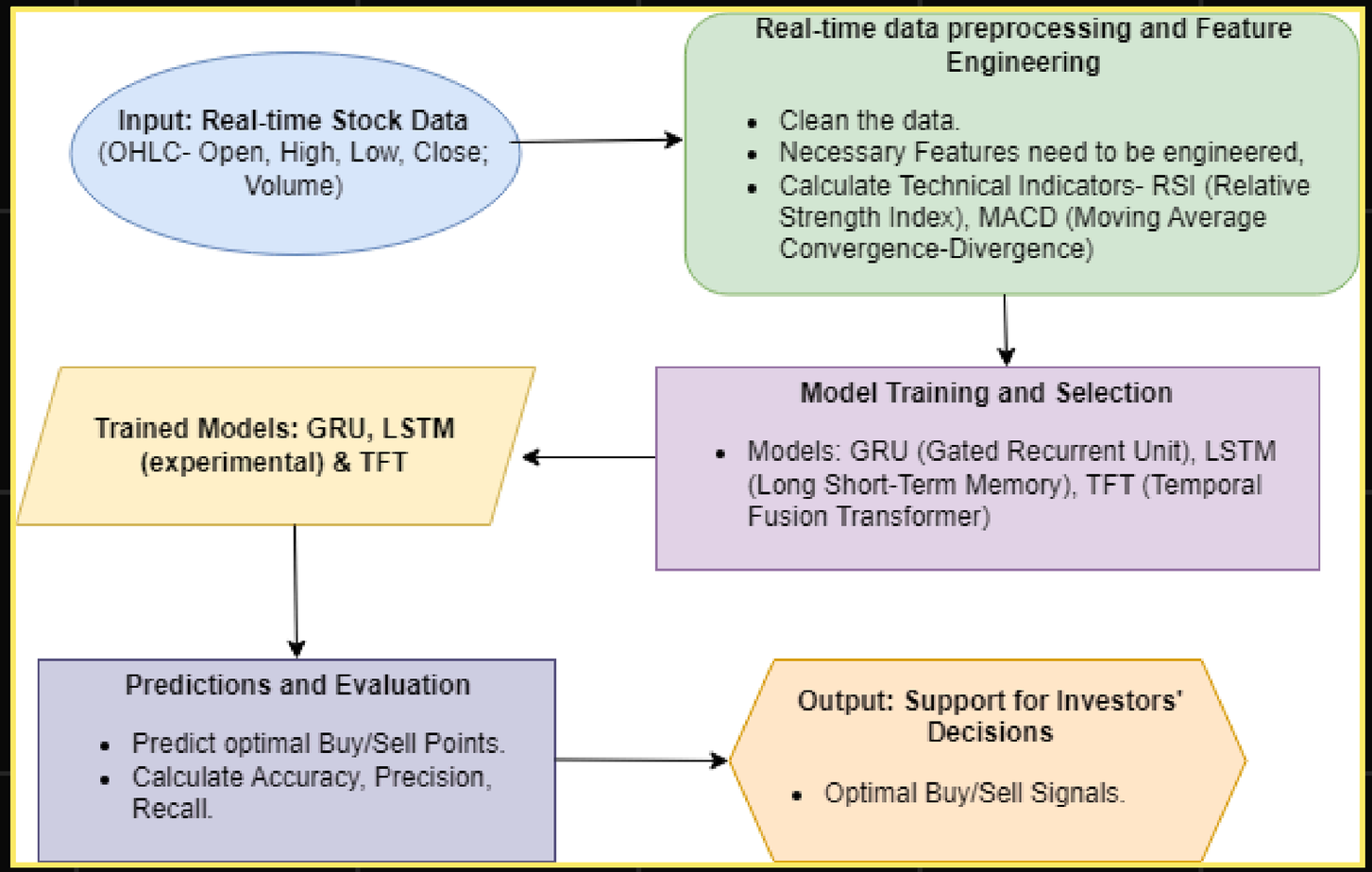
Source	Metatrader
Size	500000 Rows, 7 Columns
Modality	Time Series

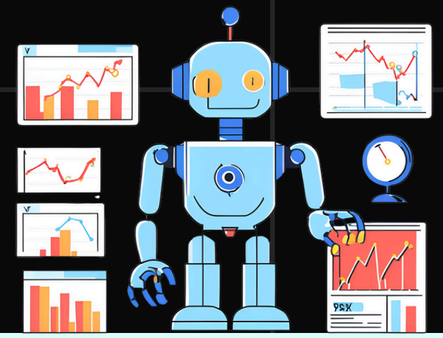
Symbol	EURUSD
Timeframe	5-Minute
Duration	2022 - Current

MetaTrader 5



Methodology



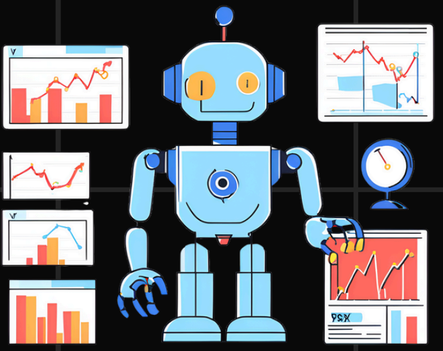


Algorithms

- **LSTM (Long Short-Term Memory):** Captures long-term dependencies and sequential patterns in time-series data

- **Temporal Fusion Transformer (TFT):** Integrates time-series with attention mechanisms, enhancing prediction accuracy by highlighting key temporal dependencies.

- **Gated Recurrent Unit (GRU) :** Use gating mechanisms to selectively update the hidden state of the network at each time step.



Financial Indicators

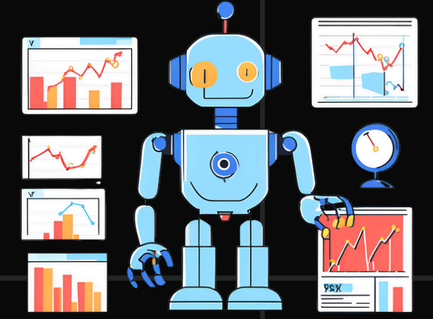
- RSI (Relative Strength Index):Quantifies momentum, indicating potential overbought/oversold conditions.
- MACD (Moving Average Convergence Divergence):Analyzes trend strength by examining short-and long-term price movements.

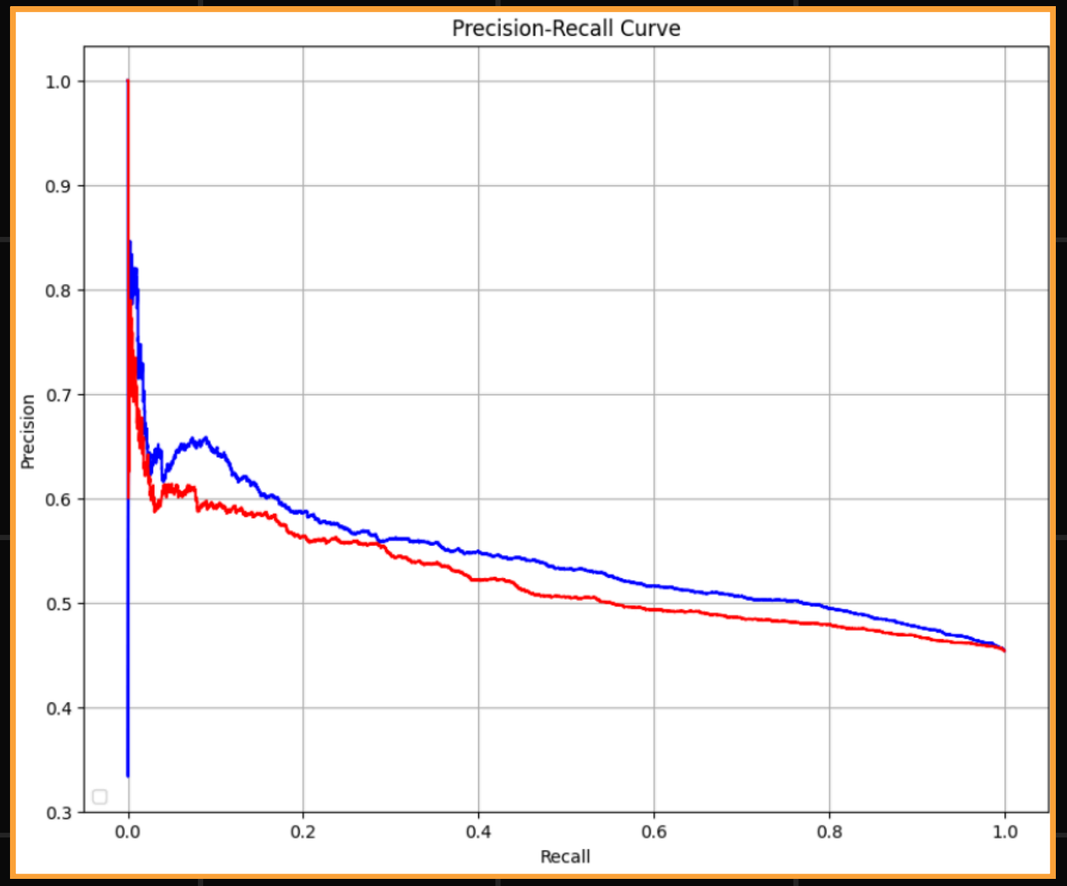
$$RSI = 100 - \left(\frac{100}{1 + \frac{\text{Average Gain}}{\text{Average Loss}}} \right)$$

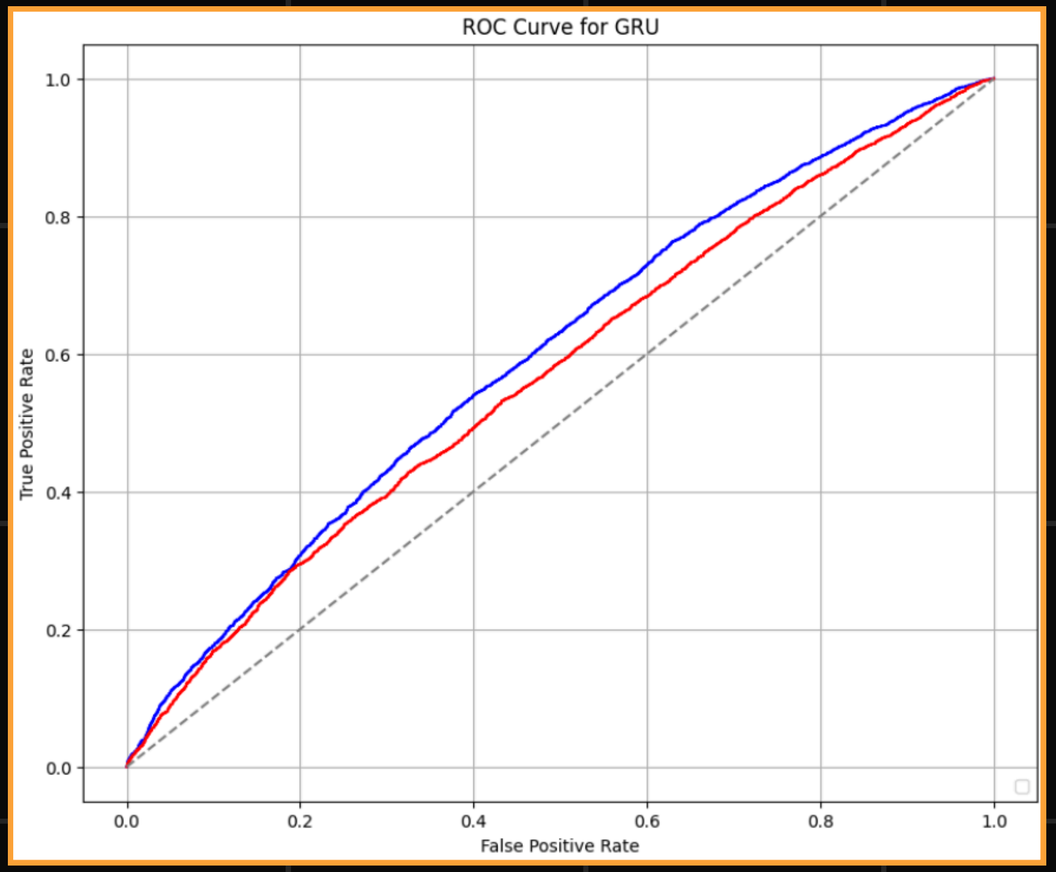
$$\text{MACD} = \text{EMA}_{\text{Fast}} - \text{EMA}_{\text{Slow}}$$

EMA: ExponentialMovingAverage to calculate the Convergence and Divergence.
EMAfast : EMA with Period 12
EMAslow: EMA with Period 26

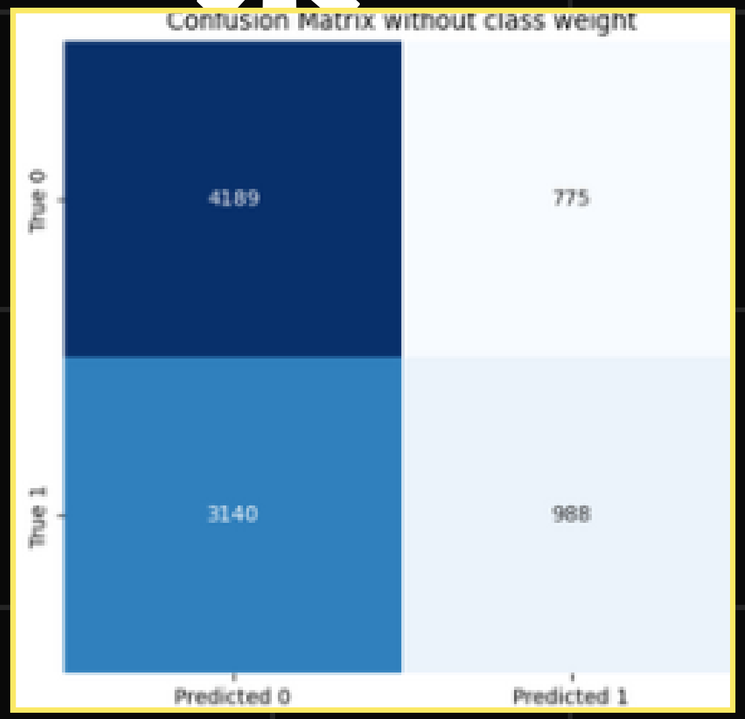
Results and Metrics

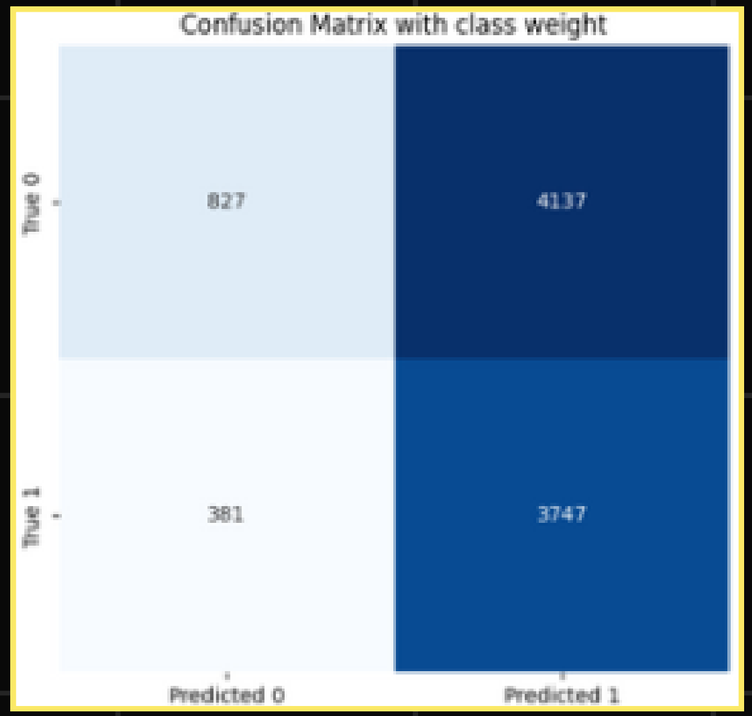




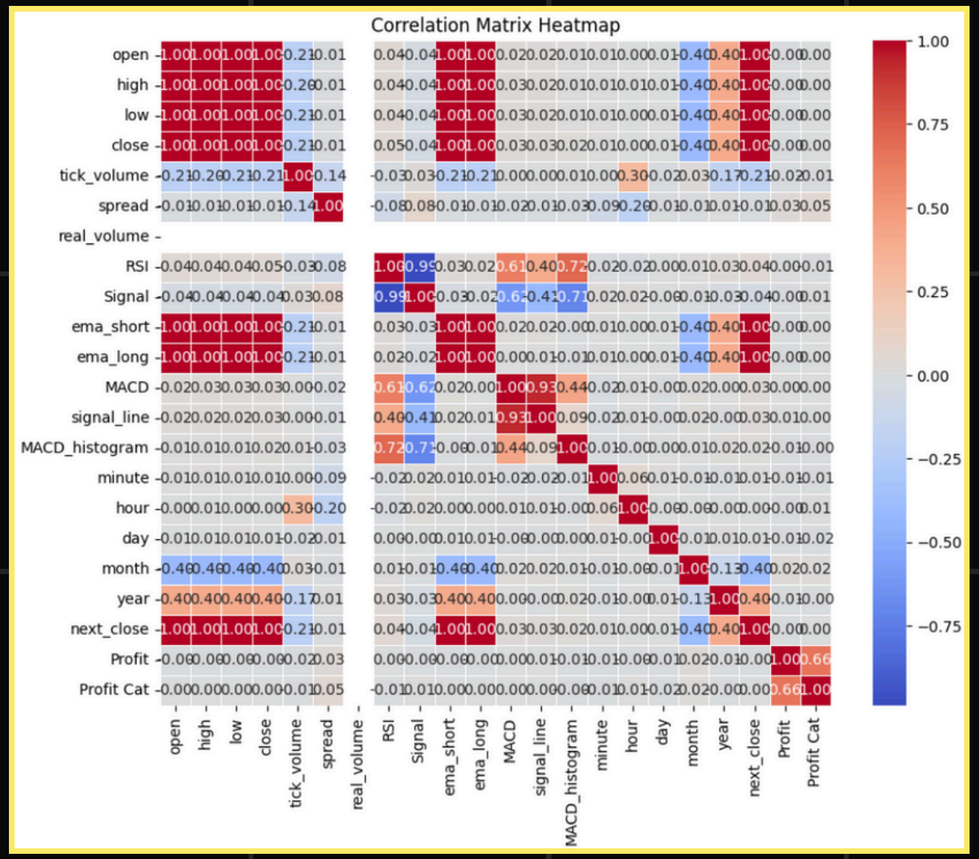


Results and Metrics





Confusion Matrix



Correlation Heatmap after Feature Engineering

Results and Metrics

Classification Report for Ensemble Model

Classification Report for Model 1 (with class weights):

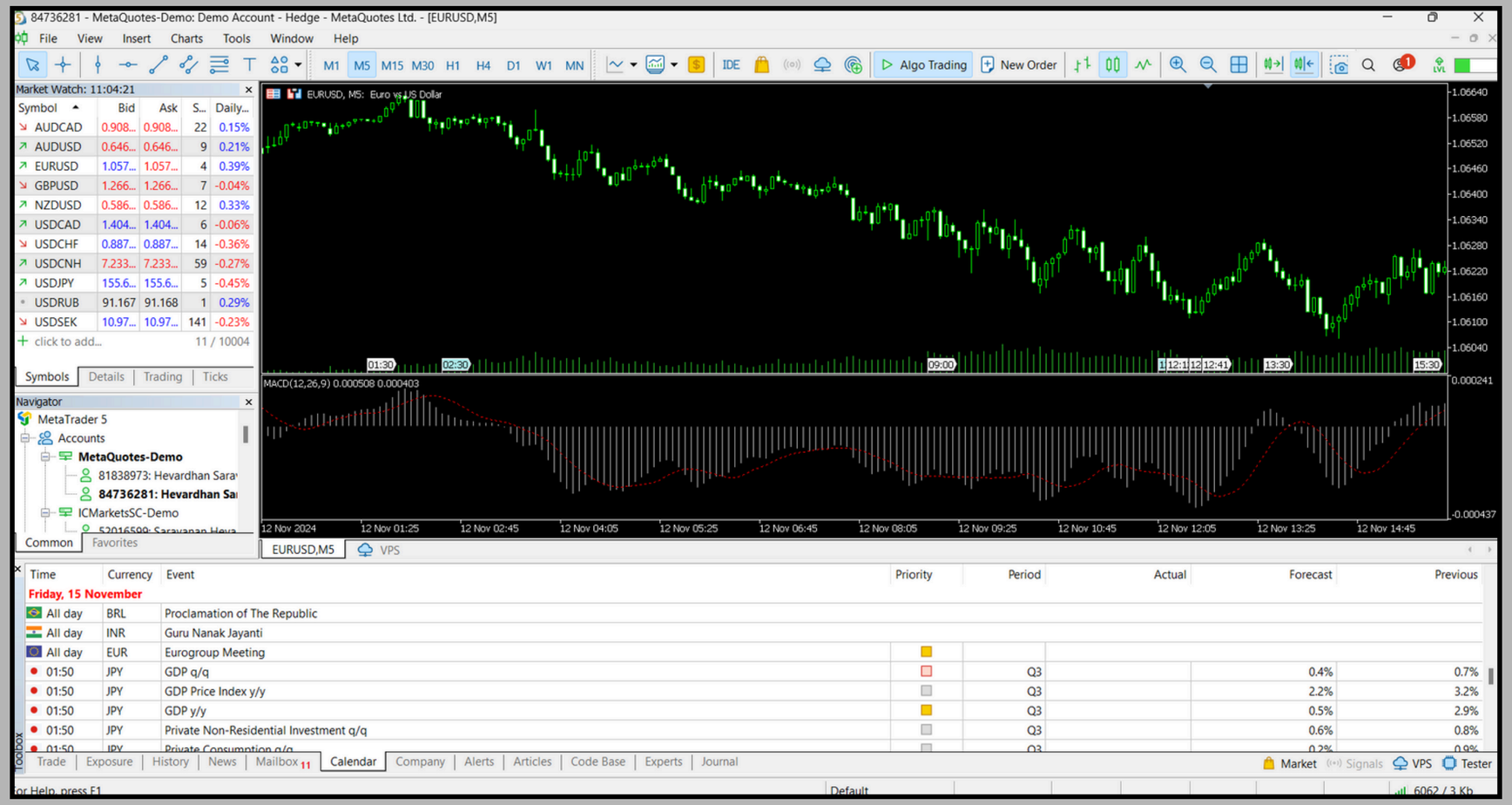
	precision	recall	f1-score	support
0.0	0.61	0.62	0.62	4964
1.0	0.53	0.52	0.52	4128
accuracy			0.57	9092
macro avg	0.57	0.57	0.57	9092
weighted avg	0.57	0.57	0.57	9092

Classification Report for Model 2 (without class weights):

	precision	recall	f1-score	support
0.0	0.60	0.42	0.49	4964
1.0	0.49	0.67	0.56	4128
accuracy			0.53	9092
macro avg	0.55	0.54	0.53	9092
weighted avg	0.55	0.53	0.53	9092

Classification Report for Weighted Ensemble Model

Screenshots of Implementation



Screenshots of Implementation

```
import numpy as np
import pandas as pd
```

✓ 1.0s Python

```
from functions import *


df = pd.read_csv("October.csv")
df.head()
```

✓ 0.2s Python

	Unnamed: 0	time	open	high	low	close	tick_volume	spread	real_volume
0	0	2022-01-03 01:00:00	1.13753	1.13784	1.13748	1.13782	192	0	0
1	1	2022-01-03 01:05:00	1.13782	1.13782	1.13742	1.13763	119	0	0
2	2	2022-01-03 01:10:00	1.13762	1.13763	1.13728	1.13730	106	0	0
3	3	2022-01-03 01:15:00	1.13729	1.13762	1.13713	1.13718	115	0	0
4	4	2022-01-03 01:20:00	1.13718	1.13721	1.13697	1.13705	44	0	0

```
del df['Unnamed: 0']
```

✓ 0.0s Python

```
rsi(df)
```

✓ 0.0s Python

	time	open	high	low	close	tick_volume	spread	real_volume	RSI
0	2022-01-03 01:00:00	1.13753	1.13784	1.13748	1.13782	192	0	0	NaN
1	2022-01-03 01:05:00	1.13782	1.13782	1.13742	1.13763	119	0	0	0.000000
2	2022-01-03 01:10:00	1.13762	1.13763	1.13728	1.13730	106	0	0	0.000000
3	2022-01-03 01:15:00	1.13729	1.13762	1.13713	1.13718	115	0	0	0.000000
4	2022-01-03 01:20:00	1.13718	1.13721	1.13697	1.13705	44	0	0	0.000000

OHLC Data loaded into a DataFrame

Screenshots of Implementation

```python
def rsi_signal(data, period=14):
    """Calculate the RSI and generate buy/sell signals based on the first crossover with thresholds."""
    # Ensure RSI is calculated
    data = rsi(data, period)

    # Initialize the 'Signal' column
    data['Signal'] = 0  # Default: No signal

    # Track the previous state of the RSI and signals
    prev_rsi = None
    buy_signal_triggered = False
    sell_signal_triggered = False

    for i in range(len(data)):
        current_rsi = data['RSI'].iloc[i]

        # Only check if the previous RSI value is available
        if prev_rsi is not None:
            # Check for buy signal (first time crossing below 30)
            if current_rsi < 30 and prev_rsi >= 30 and not buy_signal_triggered:
                data['Signal'].iloc[i] = 1  # Buy signal
                buy_signal_triggered = True  # Mark buy signal as triggered

            # Check for sell signal (first time crossing above 70)
            elif current_rsi > 70 and prev_rsi <= 70 and not sell_signal_triggered:
                data['Signal'].iloc[i] = -1  # Sell signal
                sell_signal_triggered = True  # Mark sell signal as triggered

            # Reset signals if crossing back into neutral range (30-70)
            if current_rsi >= 30 and current_rsi <= 70:
                buy_signal_triggered = False
                sell_signal_triggered = False

        # Update previous RSI
        prev_rsi = current_rsi

    return data

# Usage example:
# data = rsi_signal(data)
```

```python
import pandas as pd

def add_next_close_price_on_signal(df):
    """
    Adds a new column to the DataFrame with the close price of the next signal (1 or -1).
    Sets next_close to 0 if the current signal is 0.

    Parameters:
    df (pd.DataFrame): The input DataFrame with 'close' and 'Signal' columns.

    Returns:
    pd.DataFrame: The DataFrame with the new column 'next_close'.
    """
    # Initialize a new column
    df['next_close'] = None

    # Loop through the DataFrame to find the next signal's close price
    for i in range(len(df)):
        if df.at[i, 'Signal'] == 0:
            df.at[i, 'next_close'] = 0  # Set next_close to 0 if Signal is 0
            df.at[i, 'Profit'] = None  # Set next_close to 0 if Signal is 0
        elif df.at[i, 'Signal'] in [1, -1]:  # Check for buy or sell signal
            # Find the next row with either a 1 or -1 signal
            for j in range(i + 1, len(df)):
                if df.at[j, 'Signal'] in [1, -1]:
                    df.at[i, 'next_close'] = df.at[j, 'close']
                    break  # Exit the inner loop once the next signal is found

    return df
```

0.0s

Code for RSI Signal

Screenshots of Implementation

```python
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Sample data
# df = pd.read_csv('your_dataset.csv')

df_alt = df[['open','high','low','close']]

# Step 1: Standardize the data
scaler = StandardScaler()
scaled_data = scaler.fit_transform(df_alt)

# Step 2: Apply PCA
# Set the number of components you want to retain (e.g., 2 or enough to explain 95% variance)
pca = PCA(n_components=0.95)  # Retain 95% of the variance
principal_components = pca.fit_transform(scaled_data)

# Convert the result to a DataFrame for easier handling
principal_df = pd.DataFrame(data=principal_components, columns=[f"PC{i+1}" for i in range(principal_components.shape[1])])

# Explained variance
explained_variance = pca.explained_variance_ratio_
print("Explained variance by each principal component:", explained_variance)

# Resulting DataFrame with Principal Components
print(principal_df.head())
```

✓ 0.1s

```
Explained variance by each principal component: [0.99993198]
        PC1
0  3.645246
1  3.592314
2  3.593027
3  3.604582
4  3.536968
```

```python
principal_df
```

✓ 0.0s

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import GRU, Dense, Dropout
from tensorflow.keras.optimizers import Adam
import numpy as np

# Define the model architecture function
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import GRU, Dropout, Dense, Bidirectional, BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ReduceLROnPlateau

def create_model(input_shape):
    model = Sequential()

    # Bidirectional GRU layer
    model.add(Bidirectional(GRU(128, return_sequences=True, recurrent_dropout=0.2), input_shape=input_shape))
    model.add(Dropout(0.3))
    model.add(BatchNormalization())

    # Additional GRU layers
    model.add(GRU(64, return_sequences=True, recurrent_dropout=0.2))
    model.add(Dropout(0.3))

    model.add(GRU(64, recurrent_dropout=0.2))
    model.add(Dropout(0.3))

    # Output layer for binary classification
    model.add(Dense(1, activation='sigmoid'))

    # Compile with Adam optimizer and binary cross-entropy loss
    model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])

    return model

# Define input shape based on training data
input_shape = (X_train.shape[1], X_train.shape[2])

# Initialize two models with the same architecture
model1 = create_model(input_shape)
model2 = create_model(input_shape)
```

Dimensionality Reduction (PCA) and GRU Model Building

Screenshots of Implementation

```python
# Define input shape based on training data
input_shape = (X_train.shape[1], X_train.shape[2])

# Initialize two models with the same architecture
model1 = create_model(input_shape)
model2 = create_model(input_shape)

# Class weights for model1
from sklearn.utils import class_weight
class_weights = class_weight.compute_class_weight(
    class_weight='balanced',  # This option adjusts weights inversely proportional to class frequencies
    classes=np.unique(y_train),  # List of all possible classes (e.g., 0 and 1)
    y=y_train  # The training labels
)

reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.9, patience=2, min_lr=1e-5)

# Convert class_weights into a dictionary
class_weight_dict = {0: class_weights[0], 1: class_weights[1]}
print("Class weights:", class_weight_dict)

class_weights_dict = {0: 0.9, 1: 1.5}

# Train the first model with class weights
history1 = model1.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=20,
    batch_size=32,
    class_weight=class_weights_dict,
    callbacks=[reduce_lr],
    verbose=1
)
```

```python
arr = y_train

unique_values, counts = np.unique(arr, return_counts=True)

print(f"Unique values: {unique_values}")
print(f"Counts: {counts}")
```

✓ 0.0s

```
Unique values: [0. 1.]
Counts: [4964 4128]
```

```python
# Train the second model without class weights
history2 = model2.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=10,
    batch_size=32,
    callbacks=[reduce_lr],
    verbose=1
)
```

✓ 4m 45.6s

```
Epoch 1/10
285/285 [==============================] - 35s 100ms/step - loss: 0.7035 - accuracy: 0.5189 - val_loss: 0.6927 - val_accuracy: 0.5054 - lr: 0.0010
Epoch 2/10
285/285 [==============================] - 28s 97ms/step - loss: 0.6956 - accuracy: 0.5313 - val_loss: 0.6912 - val_accuracy: 0.5506 - lr: 0.0010
Epoch 3/10
285/285 [==============================] - 28s 98ms/step - loss: 0.6966 - accuracy: 0.5232 - val_loss: 0.6886 - val_accuracy: 0.5548 - lr: 0.0010
Epoch 4/10
285/285 [==============================] - 28s 97ms/step - loss: 0.6934 - accuracy: 0.5283 - val_loss: 0.6920 - val_accuracy: 0.5491 - lr: 0.0010
Epoch 5/10
285/285 [==============================] - 28s 98ms/step - loss: 0.6913 - accuracy: 0.5339 - val_loss: 0.6892 - val_accuracy: 0.5424 - lr: 0.0010
Epoch 6/10
285/285 [==============================] - 28s 99ms/step - loss: 0.6905 - accuracy: 0.5378 - val_loss: 0.6893 - val_accuracy: 0.5429 - lr: 9.0000e-
Epoch 7/10
285/285 [==============================] - 28s 97ms/step - loss: 0.6892 - accuracy: 0.5445 - val_loss: 0.6888 - val_accuracy: 0.5558 - lr: 9.0000e-
Epoch 8/10
285/285 [==============================] - 28s 98ms/step - loss: 0.6874 - accuracy: 0.5472 - val_loss: 0.6917 - val_accuracy: 0.5224 - lr: 8.1000e-
Epoch 9/10
285/285 [==============================] - 28s 99ms/step - loss: 0.6883 - accuracy: 0.5488 - val_loss: 0.6909 - val_accuracy: 0.5429 - lr: 8.1000e-
```

GRU: Training

Screenshots of Implementation

```
# Step 6: Define the LSTM Model
model = Sequential()
model.add(LSTM(50, return_sequences=True, input_shape=(X_train_scaled.shape[1], X_train_scaled.shape[2])))
model.add(LSTM(50))
model.add(Dense(25, activation='relu'))
model.add(Dense(3, activation='softmax'))  # Output layer for 3 classes

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/rnn/rnn.py:204: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(
  super().__init__(**kwargs)
```

```
# Step 7: Train the model
history = model.fit(X_train_scaled, y_train_categorical, epochs=50, batch_size=32, validation_data=(X_test_scaled, y_test_categorical))
```

```
Epoch 1/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 4s 4s/step - accuracy: 0.4062 - loss: 1.0679 - val_accuracy: 0.6250 - val_loss: 1.0517
Epoch 2/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 186ms/step - accuracy: 0.4062 - loss: 1.0372 - val_accuracy: 0.7500 - val_loss: 1.0345
Epoch 3/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 138ms/step - accuracy: 0.6875 - loss: 1.0130 - val_accuracy: 0.7500 - val_loss: 1.0174
Epoch 4/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 71ms/step - accuracy: 0.8125 - loss: 0.9903 - val_accuracy: 0.7500 - val_loss: 0.9999
Epoch 5/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 71ms/step - accuracy: 0.8125 - loss: 0.9668 - val_accuracy: 0.7500 - val_loss: 0.9821
Epoch 6/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 69ms/step - accuracy: 0.8125 - loss: 0.9425 - val_accuracy: 0.7500 - val_loss: 0.9650
Epoch 7/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 75ms/step - accuracy: 0.8125 - loss: 0.9169 - val_accuracy: 0.7500 - val_loss: 0.9497
Epoch 8/50
```

LSTM for MACD

Screenshots of Implementation

```
Epoch 24/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 70ms/step - accuracy: 0.8125 - loss: 0.5054 - val_accuracy: 0.7500 - val_loss: 1.0000
Epoch 25/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 69ms/step - accuracy: 0.8125 - loss: 0.4966 - val_accuracy: 0.7500 - val_loss: 1.0060
Epoch 26/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 132ms/step - accuracy: 0.8125 - loss: 0.4883 - val_accuracy: 0.7500 - val_loss: 1.0096
Epoch 27/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 79ms/step - accuracy: 0.8125 - loss: 0.4802 - val_accuracy: 0.7500 - val_loss: 1.0113
Epoch 28/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 142ms/step - accuracy: 0.8125 - loss: 0.4719 - val_accuracy: 0.7500 - val_loss: 1.0114
Epoch 29/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 69ms/step - accuracy: 0.8125 - loss: 0.4633 - val_accuracy: 0.7500 - val_loss: 1.0100
```

```
[ ] # Step 8: Evaluate the model
    test_loss, test_accuracy = model.evaluate(X_test_scaled, y_test_categorical)
    print(f"Test Accuracy: {test_accuracy * 100:.2f}%")

    # The model is now trained and ready to make predictions on new data
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - accuracy: 0.7500 - loss: 1.1813
Test Accuracy: 75.00%
```

Training LSTM Model

Video Link for Tool Demo

DL_Bull’sEye Video Demo Link:

https://drive.google.com/file/d/12S1rIj4u2GVuoj4xPAI0rlK74vk4WQJf/view?usp=drive_link

GitHub Link for Source Code

GitHub Link for Bull's Eye:

https://github.com/hevardhan/BullsEye---v5

Thank
You!