

NYCU_Artificial_Intelligence_Capstone_HW1

- This dataset is available at: [here](https://github.com/hsuyee/NYCU_Artificial_Intelligence_Capstone_Labs/tree/main/Lab1) (https://github.com/hsuyee/NYCU_Artificial_Intelligence_Capstone_Labs/tree/main/Lab1), in collaboration with teammate 111652040.
- This pdf is available at: [here](https://hackmd.io/@Origamyee/B1pfNatqJl) (<https://hackmd.io/@Origamyee/B1pfNatqJl>).
- This code is available at: [here](https://github.com/hsuyee/NYCU_Artificial_Intelligence_Capstone/tree/main/Lab1) (https://github.com/hsuyee/NYCU_Artificial_Intelligence_Capstone/tree/main/Lab1).

Part 0. Overview

In this homework, our goal is to predict the BTC/USDT cryptocurrency close price change five hours ahead and use this prediction to design our trading strategy. Below, we outline the process step by step:

1. Create Dataset
2. Clean Up Dataset
3. Algorithm and Analysis
4. Experiments
5. Discussion
6. References

Part 1. Create Dataset

To make predictions, we need OHLC (open, high, low, close) price data. Therefore, we use the Binance API to retrieve such data (details available at: [Binance Public Data](https://github.com/binance/binance-public-data) (<https://github.com/binance/binance-public-data>), and [Kline/Candlestick Data](https://developers.binance.com/docs/derivatives/usds-margined-futures/market-data/rest-api/Kline-Candlestick-Data) (<https://developers.binance.com/docs/derivatives/usds-margined-futures/market-data/rest-api/Kline-Candlestick-Data>)). The implementation is handled by `preprocess.py`. We set the K-line parameters as follows:

- a. `symbol = BTCUSDT`
- b. `interval = '1h'`
- c. `start_date = '2024-11-01'`
- d. `end_date = '2025-02-01'`

If some column values are missing, we simply use `dropna` to remove them.

After running `preprocess.py`, we obtain a CSV file named `klines_BTC.csv`, which follows the format below:

klines_BTC									
open_time	open	high	low	close	volume	taker_buy_base_asset_volume	taker_buy_quote_asset_volume	time	price
2024-11-01 00:00:00	70321.9	70160.9	70188.6	69235.539	3122.97			219650270.5021	
2024-11-01 01:00:00	70188.7	70384.8	69312.2	69424.9	230949.876			637407952.6586	
2024-11-01 02:00:00	69424.9	69621.2	68870.4	69592.7	25645.369			849623340.7967	
2024-11-01 03:00:00	69592.7	69738.0	69357.1	69397.7	6709.285			3069.884	213563233.8988
2024-11-01 04:00:00	69397.7	69650.0	69363.9	69618.5	3543.899			1949.728	135516392.2082

Part 2. Clean Up Dataset

2.1 Feature Engineering

We add the target label **target_pct_change** to the dataset, defined by $(\text{Close}_{i+1} - \text{Close}_i)/\text{Close}_i$. Since OHLC, Volume, Taker buy base asset volume, and Taker buy quote asset volume alone may not fully represent the price trend, we incorporate additional common factors to enhance the model's ability to fit price changes.

Specifically, we add volatility, price_range, ma_7, and other similar features. The implementation is handled by `add_factors.py`. After running `add_factors.py`, we obtain a CSV file named `klines_BTC_with_factors.csv`, which follows the format below:

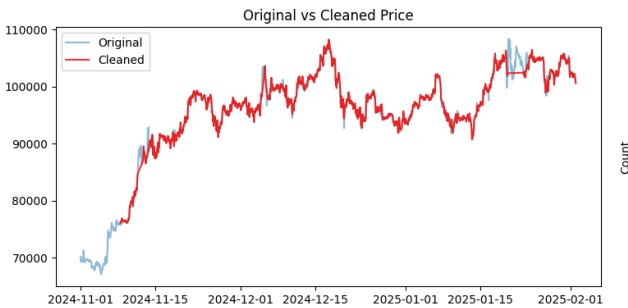
open_time	open	high	low	close	volume	taker_buy_base_asset_volume	taker_buy_quote_asset_volume	time	price	volatility	price_range	ma_7	ma_20	ma_60
2024-11-01 00:00:00	70321.9	70160.9	70188.6	69235.539	3122.97			219650270.5021						
2024-11-01 01:00:00	70188.7	70384.8	69312.2	69424.9	230949.876			637407952.6586						
2024-11-01 02:00:00	69424.9	69621.2	68870.4	69592.7	25645.369			849623340.7967						
2024-11-01 03:00:00	69592.7	69738.0	69357.1	69397.7	6709.285			3069.884						
2024-11-01 04:00:00	69397.7	69650.0	69363.9	69618.5	3543.899			1949.728						

open_time	open	high	low	close	volume	taker_buy_base_asset_volume	taker_buy_quote_asset_volume	time	price	volatility	price_range	ma_7	ma_20	ma_60
2024-11-01 00:00:00	70321.9	70160.9	70188.6	69235.539	3122.97			219650270.5021						
2024-11-01 01:00:00	70188.7	70384.8	69312.2	69424.9	230949.876			637407952.6586						
2024-11-01 02:00:00	69424.9	69621.2	68870.4	69592.7	25645.369			849623340.7967						
2024-11-01 03:00:00	69592.7	69738.0	69357.1	69397.7	6709.285			3069.884						
2024-11-01 04:00:00	69397.7	69650.0	69363.9	69618.5	3543.899			1949.728						

open_time	open	high	low	close	volume	taker_buy_base_asset_volume	taker_buy_quote_asset_volume	time	price	volatility	price_range	ma_7	ma_20	ma_60
2024-11-01 00:00:00	70321.9	70160.9	70188.6	69235.539	3122.97			219650270.5021						
2024-11-01 01:00:00	70188.7	70384.8	69312.2	69424.9	230949.876			637407952.6586						
2024-11-01 02:00:00	69424.9	69621.2	68870.4	69592.7	25645.369			849623340.7967						
2024-11-01 03:00:00	69592.7	69738.0	69357.1	69397.7	6709.285			3069.884						
2024-11-01 04:00:00	69397.7	69650.0	69363.9	69618.5	3543.899			1949.728						

2.2 Data Cleaning with Autoencoders and Z-score

Due to the high level of noise in the market, I use autoencoders and Z-score to filter out noise while preserving useful data. The refined result is shown below:



Part 3. Algorithm and Analysis

Models Selection

We use the following models from the `sklearn` library to predict `target_pct_change` or `target_multiclass`:

- Supervised learning: XGBoost regression and Transformer
- Unsupervised learning: K-Means (Class 1: lower than -2%, Class 2: -2% to -0.5%, Class 3: -0.5% to 0.5%, Class 4: 0.5% to 2%, Class 5: greater than 2%)

And the default hyperparameters and description are below:

Transformer	XGBoost	Kmeans
<pre># Define configuration settings class Config: ... Configuration class to centralize all hyperparameters and settings. # File paths and names FOLDER_PATH = "" INPUT_FILE = "klines_BTC_with_factors.csv" OUTPUT_FILE = "klines_BTC_factors_with_direction.csv" VISUALIZATION_FILE = "learning_curve_transformer.png" PREDICTION_PLOT_FILE = "predictions_vs_actual_transformer.png" MODEL_FILE = "transformer_model.h5" # Train-test split parameters TEST_SIZE = 0.6 FORECAST_HORIZON = 1 # Training parameters EPOCHS = 20 BATCH_SIZE = 32 TIME_STEPS = 60</pre>	<pre># Define configuration settings class Config: ... Configuration class to centralize all hyperparameters and settings. # File paths and names FOLDER_PATH = "" INPUT_FILE = "klines_BTC_RCA.csv" OUTPUT_FILE = "klines_BTC_factors_with_direction.csv" VISUALIZATION_FILE = "learning_curve_xgboost.png" PREDICTION_PLOT_FILE = "predictions_vs_actual_xgboost.png" MODEL_FILE = "xgboost_model.json" # Train-test split parameters TEST_SIZE = 0.6 # Training parameters N_ESTIMATORS = 1000 LEARNING_RATE = 0.05 MAX_DEPTH = 3 SUBSAMPLE = 0.8 COLSAMPLE_BYTREE = 0.8</pre>	<pre># Define configuration settings class Config: ... Configuration class to centralize all hyperparameters and settings. # File paths and names FOLDER_PATH = "" INPUT_FILE = "klines_BTC_with_factors.csv" OUTPUT_FILE = "klines_BTC_clusters.csv" CLUSTER_PLOT_FILE = "kmeans_clusters.png" # Clustering parameters TIME_STEPS = 60 N_CLUSTERS = 5 # 分成 4 群 RANDOM_STATE = 42 # 保證每次結果一樣</pre>

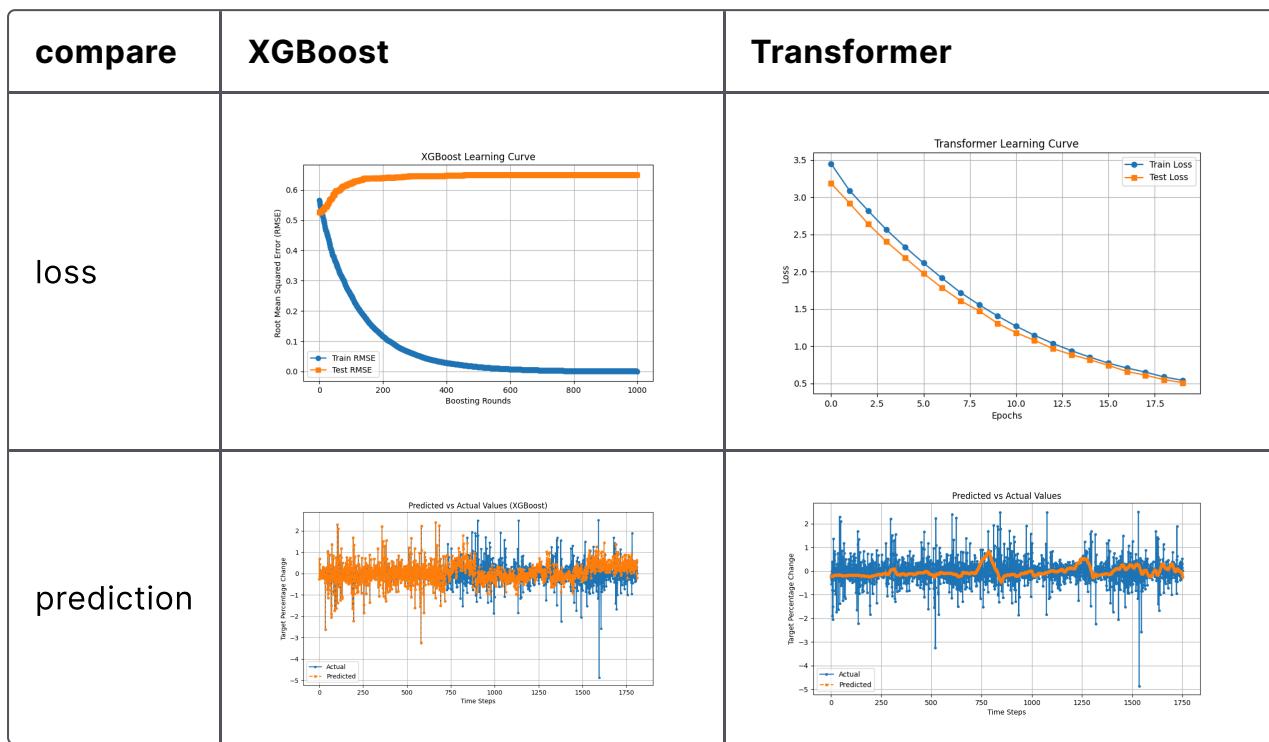
- XGBoost: A gradient boosting algorithm that builds decision trees sequentially, optimizing for efficiency and performance.
- Transformer: A deep learning model using self-attention to capture long-range dependencies in sequential data.
- K-Means: An unsupervised clustering algorithm that partitions data into K groups by minimizing intra-cluster variance.

Reference Public Libraries

- [scikit-learn \(<https://scikit-learn.org/stable/>\)](https://scikit-learn.org/stable/)

- [XGBoost](https://xgboost.readthedocs.io/en/stable/) (<https://xgboost.readthedocs.io/en/stable/>).
- [TensorFlow](https://www.tensorflow.org/?hl=zh-tw) (<https://www.tensorflow.org/?hl=zh-tw>).

Evaluate the performance (supervised learning)



Evaluate the performance (unsupervised learning)

Kmeans

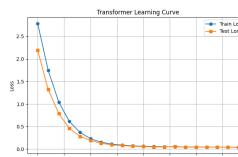
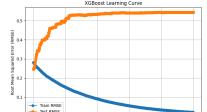
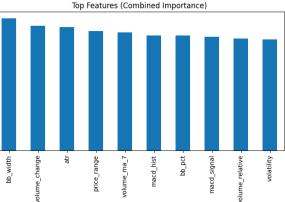
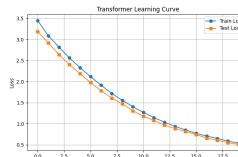
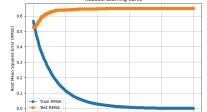
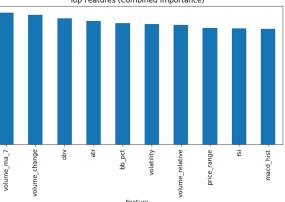
1. Silhouette Score: 0.0959 (higher is better)
2. Inertia (SSE): 2260203.5053 (lower is better)
3. Adjusted Rand Index (ARI): 0.0064 (higher is better)
4. Normalized Mutual Information (NMI): 0.0063 (higher is better)
5. Accuracy: 0.3333 (higher is better)

Part 4. Experiments

Problem 1

What if we use a different time interval?

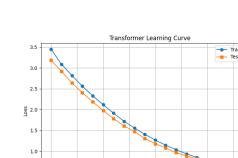
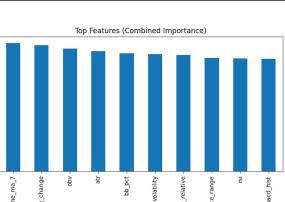
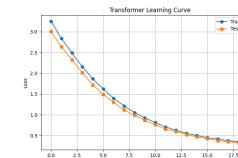
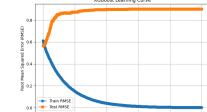
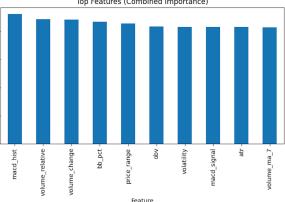
We modified the dataset's time interval setting from 1h to 15m , and the comparison results are shown below.

time_interval	Transformer	XGBoost	important features
15m			
1h			

As we can see, both models achieve better performance with a smaller time interval, as indicated by their lower testing loss. Intuitively, a smaller time interval functions similarly to data augmentation. Additionally, the Transformer converges more efficiently, while XGBoost overfits at a slower rate. Moreover, the important features remain unchanged.

Problem 2

What if we do not perform data cleaning with Autoencoders and Z-score?

data_cleaning	Transformer	XGBoost	Important features
YES			
NO			

Intuitively, this would lead to poor training and testing results.

We compare the effect of data cleaning using Autoencoders and Z-score above. As we can see, if we do not perform data cleaning, the training and testing performance of XGBoost deteriorates significantly, while the Transformer model remains nearly the same. Additionally, the important features differ slightly; for example, `rsi` and `macd_signal` do not appear in the same diagram. Why do we obtain such a result? Since we did not perform data cleaning, XGBoost may learn from noise, whereas the Transformer is more resistant to noise as it can capture temporal behavior.

Problem 3

What if we delete some important features?

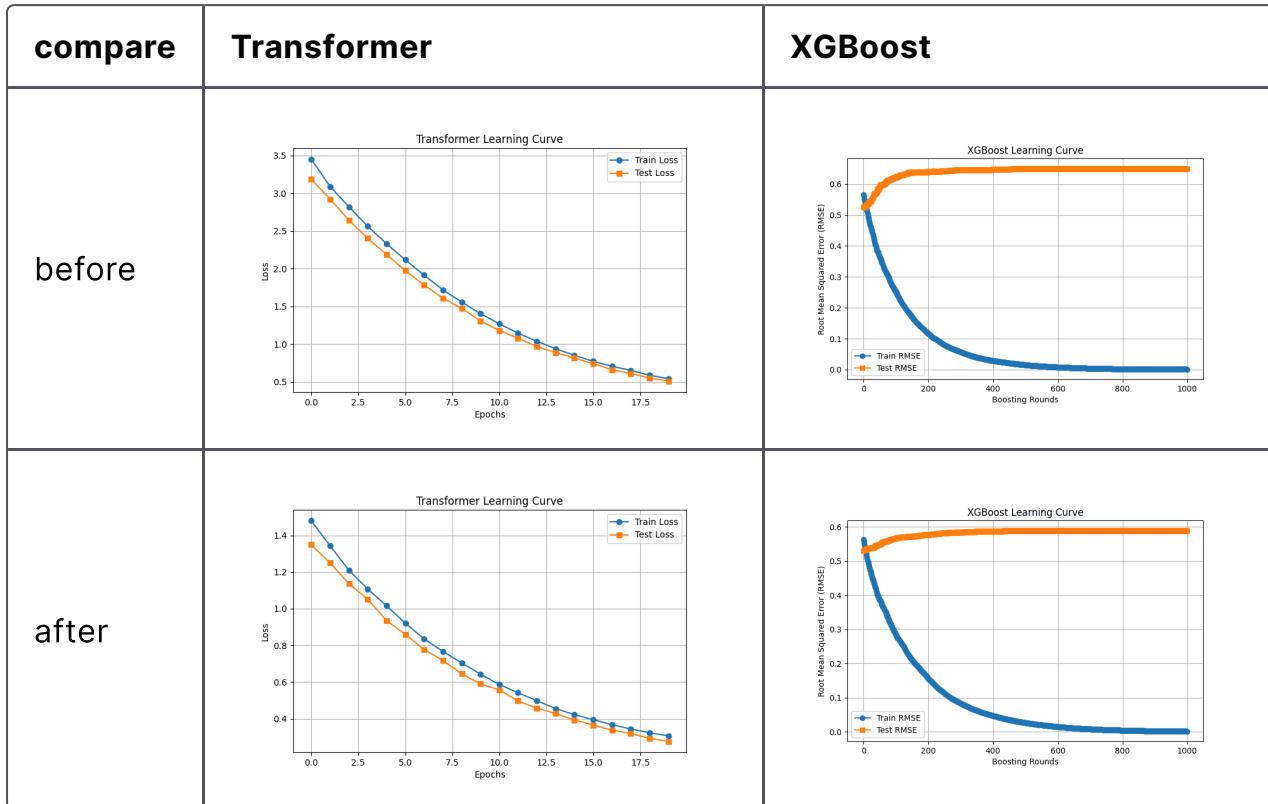
First, we define "importance" as the total decrease in impurity when a feature is used for splitting across all trees. Second, we remove the top 10 most important features and analyze their effect. The experiment results are shown below.

compare	Transformer	XGBoost	Important features
before			
after			

As we can see, when we change only the feature vectors, the Transformer model becomes more cautious after removing important features. Without key features, the model has less confidence and predicts price changes more conservatively. Additionally, this leads to an increase in XGBoost's testing loss.

Problem 4

What if we apply the PCA method to reduce noise, remove unimportant features, and decrease training time?



```

✓ Initial feature count (excluding targets): 34
✓ PCA reduced dimensions to 12, preserving 95.0% variance.
✗ Dropped 22 features (from 34 → 12)
✓ Retained feature names: ['open', 'high', 'low', 'close', 'volume', 'taker_buy_base_asset_volume',
, 'taker_buy_quote_asset_volume', 'return', 'log_return', 'volatility', 'price_range', 'ma_7']
✓ Explained variance plot saved to pca_explained_variance.png
✓ PCA-transformed data saved to klines_BTC_PCA.csv
origamyee@Hsiu-IdleMacBook-Pro:~/Desktop/NYCU_Artificial_Intelligence_Capstone/Lab1$ python3.12 predict_transformer.py
✓ Selected 12 features for training.

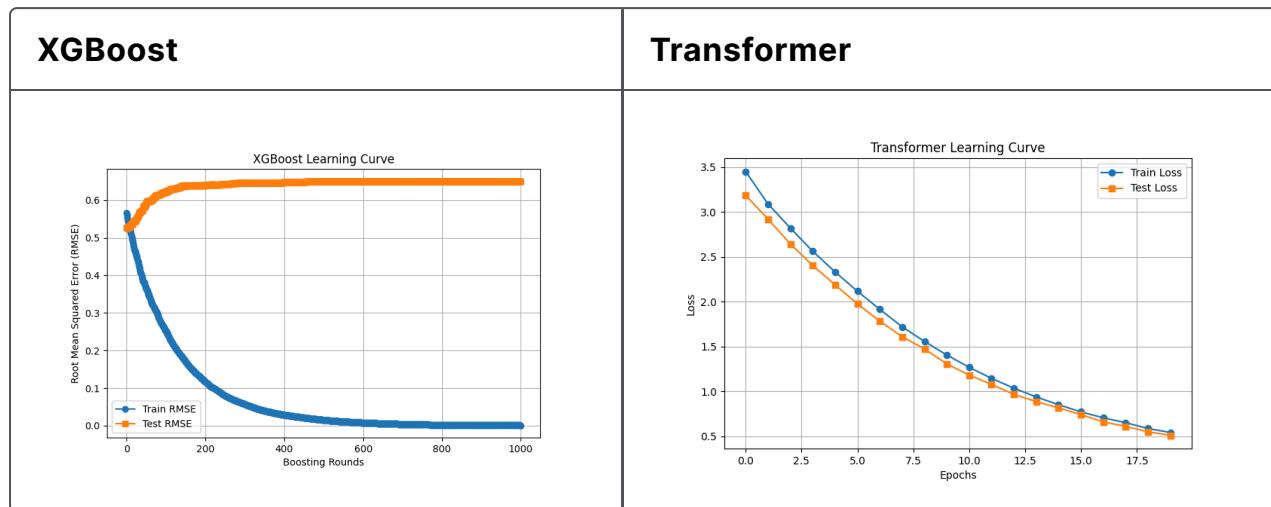
```

As we can see, when we apply the PCA method to the dataset, the training and testing loss significantly decrease after a few epochs in both models. This is because the PCA model reduces correlation among feature vectors and simplifies the complexity of the problem.

Problem 5

What if we use a non-temporal model to predict temporal behavior compared to using a temporal model?

The XGBoost model tends to overfit compared to the Transformer model. Below is an example where we use similar loss functions: Pseudo-Huber error in XGBoost and Huber loss in the Transformer model.



As we can see, the testing loss increases as the number of training iterations increases. This is a typical sign of overfitting. Intuitively, since the data depends on time, the model may fail to capture temporal effects properly.

Part 5. Discussion

Problem 1

Describe experiments that you would do if there were more time available.

1. Add more statistical indices and analyze their effects.
2. Expand our project—since we have the next price change, we can develop a long/short strategy and evaluate its performance using PnL as a metric.
3. Predict more hyperparameters, such as the stop-loss threshold, to improve balance.
4. Try more models and analyze their effects.

Problem 2

What I have learned from the experiments as well as your remaining questions?

Parts 1 and 2

In Parts 1 and 2, I learned how to clean up data. Initially, I thought we could simply add statistical indices to enhance our fitting power. However, after reading a LinkedIn post, I realized that the data might contain noise. This naturally led to an important question: "How can we filter out noise while keeping useful data?"

Marc Sniukas · 3rd+ · Following ...
For over 20 years, I've helped leaders make their comp...
1d · 1k likes

Your strategy should be making you money.
If it's not, you don't have a strategy.

Most companies treat strategy like an abstract exercise—long meetings, bloated decks, and vague goals that never see the light of day.

Then they wonder why growth is flat, profits are shrinking, and employees are disengaged.

Here's what the winners do differently:

- ✓ They move beyond traditional strategy execution and truly live their strategy.
- ✓ They cut out the noise and prioritize what actually moves the needle.**
- ✓ They adapt in real time instead of sticking to rigid long-term plans.

Companies that do this see 2x growth, 50% higher profits, and 40% better customer retention.

Not because they work harder.
Not because they have bigger budgets.

But because they actually use strategy as a performance multiplier—not a corporate buzzword.

If your strategy isn't delivering measurable ROI, you're doing it wrong.

I break down what winning companies do differently in my latest article. Read it here. ↗

What's the biggest impact a well-executed strategy has had on your business?
Drop it in the comments.

Repost to help someone in your network.

I tried to find the answer on the Internet, and the top three posts that inspired me are:

1. Towards Denoised Market Indices Pt. 1 — Static Indices

(<https://medium.com/@lavaronic/towards-denoised-market-indices-pt-1-static-indices-ac9004e930b1>).

2. Autoencoders Simplified: Real-World Data Cleaning Application

Satejraste (<https://medium.com/@satejraste/autoencoders-simplified-real-world-data-cleaning-application-5dda1b3c686d>).

3. AutoEncoder (一) - 認識與理解 (<https://medium.com/ml-note/autoencoder-%E4%B8%80-%E8%AA%8D%E8%AD%98%E8%88%87%E7%90%86%E8%A7%A3-725854ab25e8>).

Therefore, I used an autoencoder and Z-score for denoising.

Part 3

In Part 3, I learned how to use PCA for denoising. Thanks to the following tutorial, I gained an understanding of its mathematical intuition and why it works intuitively.

1. [Principal Component Analysis\(PCA\)](https://www.geeksforgeeks.org/principal-component-analysis-pca/) (<https://www.geeksforgeeks.org/principal-component-analysis-pca/>).
2. [Learning Model : Unsupervised Machine Learning_主成分分析 \(PCA\) 原理詳解](https://medium.com/ai%E5%8F%8D%E6%96%97%E5%9F%8E/preprocessing-data-%E4%B8%BB%E6%88%90%E5%88%86%E5%88%86%E6%9E%90-pca-%E5%8E%9F%E7%90%86%E8%A9%B3%E8%A7%A3-afe1fd044d4f) (<https://medium.com/ai%E5%8F%8D%E6%96%97%E5%9F%8E/preprocessing-data-%E4%B8%BB%E6%88%90%E5%88%86%E5%88%86%E6%9E%90-pca-%E5%8E%9F%E7%90%86%E8%A9%B3%E8%A7%A3-afe1fd044d4f>).
3. [【机器学习】降维——PCA \(非常详细\)](https://zhuanlan.zhihu.com/p/77151308) (<https://zhuanlan.zhihu.com/p/77151308>).

Part 6. References

1. [Binance Public Data](https://github.com/binance/binance-public-data) (<https://github.com/binance/binance-public-data>).
2. [Kline/Candlestick Data](https://developers.binance.com/docs/derivatives/usds-margined-futures/market-data/rest-api/Kline-Candlestick-Data) (<https://developers.binance.com/docs/derivatives/usds-margined-futures/market-data/rest-api/Kline-Candlestick-Data>).
3. [scikit-learn](https://scikit-learn.org/stable/) (<https://scikit-learn.org/stable/>).
4. [XGBoost](https://xgboost.readthedocs.io/en/stable/) (<https://xgboost.readthedocs.io/en/stable/>).
5. [TensorFlow](https://www.tensorflow.org/?hl=zh-tw) (<https://www.tensorflow.org/?hl=zh-tw>).
6. [Towards Denoised Market Indices Pt. 1 — Static Indices](https://medium.com/@lavaronic/towards-denoised-market-indices-pt-1-static-indices-ac9004e930b1) (<https://medium.com/@lavaronic/towards-denoised-market-indices-pt-1-static-indices-ac9004e930b1>).
7. [Autoencoders Simplified: Real-World Data Cleaning Application Satejraste](https://medium.com/@satejraste/autoencoders-simplified-real-world-data-cleaning-application-5dda1b3c686d) (<https://medium.com/@satejraste/autoencoders-simplified-real-world-data-cleaning-application-5dda1b3c686d>).
8. [AutoEncoder \(一\)-認識與理解](https://medium.com/ml-note/autoencoder-%E4%B8%80-%E8%AA%8D%E8%AD%98%E8%88%87%E7%90%86%E8%A7%A3-725854ab25e8) (<https://medium.com/ml-note/autoencoder-%E4%B8%80-%E8%AA%8D%E8%AD%98%E8%88%87%E7%90%86%E8%A7%A3-725854ab25e8>).
9. [Principal Component Analysis\(PCA\)](https://www.geeksforgeeks.org/principal-component-analysis-pca/) (<https://www.geeksforgeeks.org/principal-component-analysis-pca/>).
10. [Learning Model : Unsupervised Machine Learning_主成分分析 \(PCA\) 原理詳解](https://medium.com/ai%E5%8F%8D%E6%96%97%E5%9F%8E/preprocessing-data-%E4%B8%BB%E6%88%90%E5%88%86%E5%88%86%E6%9E%90-pca-%E5%8E%9F%E7%90%86%E8%A9%B3%E8%A7%A3-afe1fd044d4f) (<https://medium.com/ai%E5%8F%8D%E6%96%97%E5%9F%8E/preprocessing-data-%E4%B8%BB%E6%88%90%E5%88%86%E5%88%86%E6%9E%90-pca-%E5%8E%9F%E7%90%86%E8%A9%B3%E8%A7%A3-afe1fd044d4f>).
11. [【机器学习】降维——PCA \(非常详细\)](https://zhuanlan.zhihu.com/p/77151308) (<https://zhuanlan.zhihu.com/p/77151308>).
12. [ChatGPT](https://chatgpt.com/) (<https://chatgpt.com/>).

Part 1. Create Dataset

preprocess.py

```
# preprocess.py
import requests
import pandas as pd
from datetime import datetime, timedelta
import os

class Config:
    """
    Configuration class to centralize all hyperparameters and settings.
    """

    # API configuration
    BINANCE_BASE_URL = "https://fapi.binance.com/fapi"
    API_VERSION = "v1"
    API_ENDPOINT = "klines"
    API_LIMIT = 1500

    # Pandas display options
    DISPLAY_MAX_ROWS = None
    DISPLAY_MAX_COLUMNS = None
    DISPLAY_WIDTH = None

    # Default data parameters
    DEFAULT_SYMBOLS = ["BTCUSDT"]
    DEFAULT_INTERVAL = "1h"
    DEFAULT_START_DATE = "2024-11-01"
    DEFAULT_END_DATE = "2025-02-01"
    DEFAULT_FILENAME = "klines_BTC.csv"

    # Date format
    DATE_FORMAT = "%Y-%m-%d"

    # Configure pandas display options
    pd.set_option("display.max_rows", Config.DISPLAY_MAX_ROWS)
    pd.set_option("display.max_columns", Config.DISPLAY_MAX_COLUMNS)
    pd.set_option("display.width", Config.DISPLAY_WIDTH)

def fetch_kline_price_data(symbol: str, interval: str, start_date: str, end_date: str) -> pd.DataFrame:
    """
    Fetches K-line (candlestick) data from Binance API for a given trading pair.

    :param symbol: Trading pair (e.g., 'BTCUSDT')
    :param interval: Time interval (e.g., '1h')
    :param start_date: Start date (YYYY-MM-DD)
    :param end_date: End date (YYYY-MM-DD)
    :return: Processed DataFrame containing OHLC and taker buy volume data
    """

    api_endpoint = f"{Config.API_VERSION}/{Config.API_ENDPOINT}"
    start_time = datetime.strptime(start_date, Config.DATE_FORMAT)
    end_time = datetime.strptime(end_date, Config.DATE_FORMAT)

    price_data = []

    while start_time < end_time:
        start_time_ms = int(start_time.timestamp() * 1000)
        url = f"{Config.BINANCE_BASE_URL}/{api_endpoint}?symbol={symbol}&interval={interval}&limit={Config.API_LIMIT}&startT={start_time_ms}&endT={end_time.timestamp() * 1000}"

        response = requests.get(url)
        data = response.json()

        if not data:
            break

        price_data.extend(data)

        # Update start_time to the close time of the last retrieved data entry
        last_entry = data[-1]
        last_close_time = last_entry[6]  # Close time (7th element in response data)

        start_time = datetime.fromtimestamp(last_close_time / 1000.0)
```

```

return process_price_data(price_data, start_date, end_date)

def process_price_data(price_data: list, start_date: str, end_date: str) -> pd.DataFrame:
    """
    Processes raw price data from Binance API into a structured DataFrame.

    :param price_data: Raw data retrieved from API
    :param start_date: Start date for filtering
    :param end_date: End date for filtering
    :return: Processed DataFrame with selected OHLC and taker buy volume columns
    """

    df = pd.DataFrame(price_data, columns=[
        "open_time", "open", "high", "low", "close", "volume", "close_time",
        "quote_asset_volume", "num_trades", "taker_buy_base_asset_volume",
        "taker_buy_quote_asset_volume", "ignore"
    ])

    # Convert data types and remove invalid entries
    df = df.apply(pd.to_numeric, errors="coerce").dropna()

    # Convert timestamp columns to human-readable format
    df["open_time"] = pd.to_datetime(df["open_time"], unit="ms")
    df["close_time"] = pd.to_datetime(df["close_time"], unit="ms")

    # Extract relevant columns
    df = df[["open_time", "open", "high", "low", "close", "volume",
              "taker_buy_base_asset_volume", "taker_buy_quote_asset_volume"]]

    # Filter data within the specified date range
    df = df[(df["open_time"] >= pd.to_datetime(start_date)) &
            (df["open_time"] < pd.to_datetime(end_date) + timedelta(days=1))]

    return df

def save_to_drive(df: pd.DataFrame, filename: str):
    """
    Saves the given DataFrame to a CSV file.

    :param df: DataFrame to be saved
    :param filename: Name of the CSV file
    """

    # Define save path
    drive_path = f"{filename}"

    # Save DataFrame
    df.to_csv(drive_path, index=False)
    print(f"File saved to {drive_path}")

if __name__ == "__main__":
    # Load parameters from configuration
    symbols = Config.DEFAULT_SYMBOLS
    interval = Config.DEFAULT_INTERVAL
    start_date = Config.DEFAULT_START_DATE
    end_date = Config.DEFAULT_END_DATE
    filename = Config.DEFAULT_FILENAME

    # Fetch and merge data
    all_data = [fetch_kline_price_data(symbol, interval, start_date, end_date)
                for symbol in symbols]

    # Combine all symbol data
    final_df = pd.concat(all_data, ignore_index=True)

    # Save to file
    save_to_drive(final_df, filename)

```

Part 2. Clean Up Dataset

add_factors.py

```

# add_factors.py
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

```

```

from tensorflow import keras
from sklearn.preprocessing import MinMaxScaler
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier

class Config:
    """
    Configuration class to centralize all hyperparameters and settings.
    """

    # File paths and names
    FOLDER_PATH = ""
    INPUT_FILE = "klines_BTC.csv"
    OUTPUT_FILE = "klines_BTC_with_factors.csv"
    VISUALIZATION_FILE = "btc_targets_analysis.png"

    # Technical indicator parameters
    WINDOW_SIZE = 75
    VOLATILITY_WINDOW = 24
    MA_WINDOWS = [7, 25, 99]
    EMA_SHORT = 12
    EMA_LONG = 26
    MACD_SIGNAL = 9
    RSI_PERIOD = 14
    BOLLINGER_WINDOW = 20
    BOLLINGER_STD = 2

    # Outlier detection parameters
    ZSCORE_THRESHOLD = 3
    ANOMALY_PERCENTILE = 95

    # Autoencoder parameters
    AUTOENCODER_LAYERS = [64, 32, 16, 32, 64]
    REGULARIZATION_FACTOR = 0.001
    DROPOUT_RATE = 0.2
    EPOCHS = 50
    BATCH_SIZE = 32
    VALIDATION_SPLIT = 0.4
    EARLY_STOPPING_PATIENCE = 5

    # Random forest parameters
    RF_ESTIMATORS = 100
    RF_RANDOM_STATE = 42

    # Target classification thresholds
    PRICE_MOVEMENT_THRESHOLDS = [-2, -0.5, 0.5, 2]
    PRICE_MOVEMENT_LABELS = ['Significant Drop', 'Small Drop', 'Sideways', 'Small Rise', 'Significant Rise']

    # Train-test split parameters
    TEST_SIZE = 0.2
    FORECAST_HORIZON = 1

def compute_atr(df: pd.DataFrame, window_size: int) -> pd.Series:
    """
    Compute Average True Range (ATR) indicator.

    Parameters:
    -----
    df : pandas.DataFrame
        DataFrame containing OHLC price data
    window_size : int
        Rolling window size for ATR calculation

    Returns:
    -----
    pandas.Series: ATR values
    """
    high_low = df["high"] - df["low"]
    high_prev_close = abs(df["high"] - df["close"].shift(1))
    low_prev_close = abs(df["low"] - df["close"].shift(1))

    true_range = pd.concat([high_low, high_prev_close, low_prev_close], axis=1).max(axis=1)
    return true_range.rolling(window=window_size).mean()

def load_and_clean_data(file_path: str) -> pd.DataFrame:
    """
    Load, clean, and preprocess raw price data.

    Parameters:
    -----
    file_path : str
    """

```

Path to the CSV file containing raw price data

```
Returns:
-----
pandas.DataFrame: Cleaned dataframe
"""
# Load the data
df_original = pd.read_csv(file_path)

# Convert timestamp column to datetime
df_original["open_time"] = pd.to_datetime(df_original["open_time"])
df_original.sort_values(by="open_time", inplace=True)

# Make a copy before modification
df_cleaned = df_original.copy()

# Handle Missing Values
df_cleaned.dropna(inplace=True)

# Ensure Correct Data Types
numeric_cols = ["open", "high", "low", "close", "volume",
                 "taker_buy_base_asset_volume", "taker_buy_quote_asset_volume"]
df_cleaned[numeric_cols] = df_cleaned[numeric_cols].astype(float)

# Remove Duplicates
df_cleaned.drop_duplicates(subset=["open_time"], keep="first", inplace=True)

return df_cleaned

def add_technical_indicators(df: pd.DataFrame) -> pd.DataFrame:
"""
Add various technical indicators as features to the dataframe.

Parameters:
-----
df : pandas.DataFrame
    DataFrame containing OHLC price data

Returns:
-----
pandas.DataFrame: DataFrame with added technical indicators
"""
# Price-based features
df['return'] = df['close'].pct_change() # Price returns
df['log_return'] = np.log(df['close']/df['close'].shift(1)) # Log returns
df['volatility'] = df['log_return'].rolling(window=Config.VOLATILITY_WINDOW).std() # Volatility
df['price_range'] = (df['high'] - df['low']) / df['open'] # Normalized price range

# Moving averages
for window in Config.MA_WINDOWS:
    df[f'ma_{window}'] = df['close'].rolling(window=window).mean()

# ATR (Average True Range)
df["atr"] = compute_atr(df, Config.WINDOW_SIZE)

# MACD
df['ema_12'] = df['close'].ewm(span=Config.EMA_SHORT).mean()
df['ema_26'] = df['close'].ewm(span=Config.EMA_LONG).mean()
df['macd'] = df['ema_12'] - df['ema_26']
df['macd_signal'] = df['macd'].ewm(span=Config.MACD_SIGNAL).mean()
df['macd_hist'] = df['macd'] - df['macd_signal']

# RSI (Relative Strength Index)
delta = df['close'].diff()
gain = delta.where(delta > 0, 0)
loss = -delta.where(delta < 0, 0)
avg_gain = gain.rolling(window=Config.RSI_PERIOD).mean()
avg_loss = loss.rolling(window=Config.RSI_PERIOD).mean()
rs = avg_gain / avg_loss
df['rsi'] = 100 - (100 / (1 + rs))

# Bollinger Bands
bb_window = Config.BOLLINGER_WINDOW
bb_std_dev = Config.BOLLINGER_STD
df['bb_middle'] = df['close'].rolling(window=bb_window).mean()
bb_std = df['close'].rolling(window=bb_window).std()
df['bb_upper'] = df['bb_middle'] + (bb_std * bb_std_dev)
df['bb_lower'] = df['bb_middle'] - (bb_std * bb_std_dev)
df['bb_width'] = (df['bb_upper'] - df['bb_lower']) / df['bb_middle']
df['bb_pct'] = (df['close'] - df['bb_lower']) / (df['bb_upper'] - df['bb_lower'])
```

```

# Volume features
df['volume_change'] = df['volume'].pct_change()
df['volume_ma_7'] = df['volume'].rolling(window=7).mean()
df['volume_relative'] = df['volume'] / df['volume_ma_7']

# OBV (On-Balance Volume)
df['obv'] = 0
df.loc[0, 'obv'] = df.loc[0, 'volume']
for i in range(1, len(df)):
    if df.loc[i, 'close'] > df.loc[i-1, 'close']:
        df.loc[i, 'obv'] = df.loc[i-1, 'obv'] + df.loc[i, 'volume']
    elif df.loc[i, 'close'] < df.loc[i-1, 'close']:
        df.loc[i, 'obv'] = df.loc[i-1, 'obv'] - df.loc[i, 'volume']
    else:
        df.loc[i, 'obv'] = df.loc[i-1, 'obv']

# Time-based features
df['hour'] = df['open_time'].dt.hour
df['day_of_week'] = df['open_time'].dt.dayofweek
df['is_weekend'] = df['day_of_week'].apply(lambda x: 1 if x >= 5 else 0)
df['month'] = df['open_time'].dt.month

# Remove rows with NaN values that resulted from calculating indicators
df.dropna(inplace=True)

return df

```

def add_target_variables(df: pd.DataFrame) -> pd.DataFrame:

Add multiple target variables for both regression and classification tasks.

Parameters:

df : pandas.DataFrame
DataFrame with price data and technical indicators

Returns:

pandas.DataFrame: DataFrame with added target variables

```

# Target 1: Binary classification – will price go up in next period?
df['target_binary'] = (df['close'].shift(-1) > df['close']).astype(int)

# Target 2: Regression – percentage price change in next period
df['target_pct_change'] = df['close'].pct_change(periods=-1) * 100

# Target 3: Regression – absolute price change in next period
df['target_abs_change'] = df['close'].shift(-1) - df['close']

# Target 4: Multi-class classification – categorize price movement
thresholds = Config.PRICE_MOVEMENT_THRESHOLDS

def categorize_movement(pct_change):
    for i, threshold in enumerate(thresholds):
        if pct_change < threshold:
            return i
    return len(thresholds) # Last category

df['target_multiclass'] = df['target_pct_change'].apply(categorize_movement)

# Target 5: Regression – log return (often better for financial modeling)
df['target_log_return'] = np.log(df['close'].shift(-1) / df['close'])

# Target 6: Regression – normalized price change (compared to recent volatility)
volatility = df['close'].rolling(window=Config.BOLLINGER_WINDOW).std()
df['target_normalized_change'] = df['target_abs_change'] / volatility

# Remove NaNs from target creation
df.dropna(inplace=True)

return df

```

def detect_outliers(df: pd.DataFrame) -> np.ndarray:

Detect outliers using statistical methods and autoencoder.

Parameters:

df : pandas.DataFrame
DataFrame with features

```

>Returns:
-----
numpy.ndarray: Boolean mask where True indicates an outlier
"""
# Get feature columns (exclude timestamps and targets)
feature_cols = [col for col in df.columns if col not in [
    'open_time', 'hour', 'day_of_week', 'is_weekend', 'month',
    'target_binary', 'target_pct_change', 'target_abs_change',
    'target_multiclass', 'target_log_return', 'target_normalized_change'
]]
"""

# Statistical outlier detection (Z-score method)
def detect_outliers_zscore(df, columns, threshold=Config.ZSCORE_THRESHOLD):
    outliers_mask = np.zeros(len(df), dtype=bool)
    for col in columns:
        z_scores = np.abs((df[col] - df[col].mean()) / df[col].std())
        col_outliers = z_scores > threshold
        outliers_mask = outliers_mask | col_outliers
    return outliers_mask

outliers_mask_zscore = detect_outliers_zscore(df, feature_cols)
print(f"Z-score method identified {outliers_mask_zscore.sum()} outliers")

# Autoencoder-based outlier detection
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(df[feature_cols])

# Define autoencoder model with regularization
input_dim = scaled_data.shape[1]
layers = Config.AUTOENCODER_LAYERS

# Build the autoencoder model
autoencoder = keras.Sequential()
# Encoder
autoencoder.add(keras.layers.Dense(
    layers[0], activation='relu', input_shape=(input_dim,),
    kernel_regularizer=keras.regularizers.l2(Config.REGULARIZATION_FACTOR)))
autoencoder.add(keras.layers.Dropout(Config.DROPOUT_RATE))

for units in layers[1:len(layers)//2 + 1]:
    autoencoder.add(keras.layers.Dense(units, activation='relu'))

# Decoder
for units in layers[len(layers)//2 + 1:]:
    autoencoder.add(keras.layers.Dense(units, activation='relu'))

autoencoder.add(keras.layers.Dense(input_dim, activation='linear'))

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

# Train the autoencoder with early stopping
early_stopping = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=Config.EARLY_STOPPING_PATIENCE,
    restore_best_weights=True
)

autoencoder.fit(
    scaled_data, scaled_data,
    epochs=Config.EPOCHS,
    batch_size=Config.BATCH_SIZE,
    shuffle=True,
    validation_split=Config.VALIDATION_SPLIT,
    callbacks=[early_stopping],
    verbose=1
)

# Compute reconstruction error
reconstructed = autoencoder.predict(scaled_data)
reconstruction_error = np.mean(np.abs(scaled_data - reconstructed), axis=1)

# Define anomaly threshold
threshold = np.percentile(reconstruction_error, Config.ANOMALY_PERCENTILE)

# Combine outlier detection methods
outliers_mask_combined = (reconstruction_error > threshold) | outliers_mask_zscore
print(f"Combined methods identified {outliers_mask_combined.sum()} outliers")

return outliers_mask_combined

```

```

def analyze_feature_importance(df: pd.DataFrame, feature_cols: list) -> tuple:
    """
    Analyze feature importance for different target variables.

    Parameters:
    -----
    df : pandas.DataFrame
        DataFrame with features and targets
    feature_cols : list
        List of feature column names

    Returns:
    -----
    tuple: Dataframes with feature importance for binary and regression targets
    """
    X = df[feature_cols]

    # For binary classification
    y_binary = df['target_binary']
    clf = RandomForestClassifier(
        n_estimators=Config.RF_ESTIMATORS,
        random_state=Config.RF_RANDOM_STATE
    )
    clf.fit(X, y_binary)

    binary_importances = pd.DataFrame({
        'feature': feature_cols,
        'importance': clf.feature_importances_
    }).sort_values('importance', ascending=False)

    print("\nTop 10 important features for binary classification:")
    print(binary_importances.head(10))

    # For regression (percentage change)
    y_pct = df['target_pct_change']
    reg = RandomForestRegressor(
        n_estimators=Config.RF_ESTIMATORS,
        random_state=Config.RF_RANDOM_STATE
    )
    reg.fit(X, y_pct)

    reg_importances = pd.DataFrame({
        'feature': feature_cols,
        'importance': reg.feature_importances_
    }).sort_values('importance', ascending=False)

    print("\nTop 10 important features for price change regression:")
    print(reg_importances.head(10))

    return binary_importances, reg_importances

def visualize_data_and_targets(df_original: pd.DataFrame, df_cleaned: pd.DataFrame,
                               binary_importances: pd.DataFrame, reg_importances: pd.DataFrame):
    """
    Visualize various aspects of the dataset and targets.

    Parameters:
    -----
    df_original : pandas.DataFrame
        Original unprocessed DataFrame
    df_cleaned : pandas.DataFrame
        Processed DataFrame with features and targets
    binary_importances : pandas.DataFrame
        Feature importances for binary classification
    reg_importances : pandas.DataFrame
        Feature importances for regression
    """
    plt.figure(figsize=(15, 15))

    # Plot 1: Original vs Cleaned Price
    plt.subplot(3, 2, 1)
    plt.plot(df_original["open_time"], df_original["close"], label="Original", alpha=0.5)
    plt.plot(df_cleaned["open_time"], df_cleaned["close"], label="Cleaned", alpha=0.8, color='red')
    plt.title("Original vs Cleaned Price")
    plt.legend()

    # Plot 2: Binary Target Distribution
    plt.subplot(3, 2, 2)
    df_cleaned['target_binary'].value_counts().plot(kind='bar')
    plt.title("Binary Target Distribution (Up/Down)")


```

```

plt.xlabel("Price Direction (1=Up, 0=Down)")
plt.ylabel("Count")

# Plot 3: Percentage Change Distribution
plt.subplot(3, 2, 3)
plt.hist(df_cleaned['target_pct_change'], bins=50)
plt.title("Price Percentage Change Distribution")
plt.xlabel("Percentage Change (%)")
plt.ylabel("Frequency")

# Plot 4: Multi-class Target Distribution
plt.subplot(3, 2, 4)
df_cleaned['target_multiclass'].value_counts().sort_index().plot(kind='bar')
plt.title("Multi-class Target Distribution")
plt.xlabel("Price Movement Category")
plt.ylabel("Count")
plt.xticks(range(len(Config.PRICE_MOVEMENT_LABELS)), Config.PRICE_MOVEMENT_LABELS, rotation=45)

# Plot 5: Correlation Between Different Target Variables
plt.subplot(3, 2, 5)
target_corr = df_cleaned[['target_binary', 'target_pct_change', 'target_abs_change',
                         'target_multiclass', 'target_log_return', 'target_normalized_change']].corr()
plt.imshow(target_corr, cmap='coolwarm')
plt.colorbar()
plt.title("Correlation Between Target Variables")
plt.xticks(range(6), target_corr.columns, rotation=90)
plt.yticks(range(6), target_corr.columns)

# Plot 6: Top features importance (combined importance)
plt.subplot(3, 2, 6)
combined_importance = binary_importances.set_index('feature')['importance'] + reg_importances.set_index('feature')['impc'
combined_importance.sort_values(ascending=False).head(10).plot(kind='bar')
plt.title("Top Features (Combined Importance)")
plt.tight_layout()
plt.xlabel("Feature")
plt.ylabel("Combined Importance Score")

plt.tight_layout()
plt.savefig(Config.VISUALIZATION_FILE)
plt.show()

```

def prepare_train_test_splits(df: pd.DataFrame, test_size=None, forecast_horizon=None):

"""

Prepare chronological train/test splits for time series data.

Parameters:

df : pandas.DataFrame
The cleaned dataframe with features and targets
test_size : float, optional
Proportion of data to use for testing
forecast_horizon : int, optional
How many periods ahead to forecast

Returns:

dict containing train/test splits for different target variables

"""

if test_size is None:
 test_size = Config.TEST_SIZE

if forecast_horizon is None:
 forecast_horizon = Config.FORECAST_HORIZON

Feature columns (everything except targets and datetime)
feature_cols = [col for col in df.columns if col not in [
 'open_time', 'target_binary', 'target_pct_change', 'target_abs_change',
 'target_multiclass', 'target_log_return', 'target_normalized_change'
]]

Split chronologically
split_idx = int(len(df) * (1 - test_size))
train_df = df.iloc[:split_idx].copy()
test_df = df.iloc[split_idx:].copy()

print(f"Training data from {train_df['open_time'].min()} to {train_df['open_time'].max()}")
print(f"Testing data from {test_df['open_time'].min()} to {test_df['open_time'].max()}"

Prepare scalers
feature_scaler = MinMaxScaler()
X_train = feature_scaler.fit_transform(train_df[feature_cols])

```

X_test = feature_scaler.transform(test_df[feature_cols])

# Prepare regression target scalers
regression_scalers = {}
regression_targets = ['target_pct_change', 'target_abs_change',
                      'target_log_return', 'target_normalized_change']

y_train_dict = {}
y_test_dict = {}

# Classification targets
y_train_dict['binary'] = train_df['target_binary'].values
y_test_dict['binary'] = test_df['target_binary'].values

y_train_dict['multiclass'] = train_df['target_multiclass'].values
y_test_dict['multiclass'] = test_df['target_multiclass'].values

# Regression targets (scaled)
for target in regression_targets:
    scaler = MinMaxScaler()
    y_train_dict[target] = scaler.fit_transform(train_df[[target]])
    y_test_dict[target] = scaler.transform(test_df[[target]])
    regression_scalers[target] = scaler

return {
    'X_train': X_train,
    'X_test': X_test,
    'y_train': y_train_dict,
    'y_test': y_test_dict,
    'feature_scaler': feature_scaler,
    'regression_scalers': regression_scalers,
    'feature_columns': feature_cols,
    'train_dates': train_df['open_time'],
    'test_dates': test_df['open_time']
}

def main():
    """
    Main function to execute the entire data processing pipeline.
    """

    # Step 1: Load and clean data
    df_cleaned = load_and_clean_data(Config.INPUT_FILE)
    df_original = df_cleaned.copy() # Save original for visualization comparison

    # Step 2: Add technical indicators
    df_cleaned = add_technical_indicators(df_cleaned)

    # Step 3: Add target variables
    df_cleaned = add_target_variables(df_cleaned)

    # Step 4: Detect and remove outliers
    outliers_mask = detect_outliers(df_cleaned)
    df_cleaned = df_cleaned[~outliers_mask]

    # Get feature columns for analysis
    feature_cols = [col for col in df_cleaned.columns if col not in [
        'open_time', 'hour', 'day_of_week', 'is_weekend', 'month',
        'target_binary', 'target_pct_change', 'target_abs_change',
        'target_multiclass', 'target_log_return', 'target_normalized_change'
    ]]

    # Step 5: Analyze feature importance
    binary_importances, reg_importances = analyze_feature_importance(df_cleaned, feature_cols)

    # Step 6: Visualize data and targets
    visualize_data_and_targets(df_original, df_cleaned, binary_importances, reg_importances)

    # Step 7: Save cleaned data with multiple targets
    df_cleaned.to_csv(Config.OUTPUT_FILE, index=False)

    # Step 8: Prepare train/test splits
    splits = prepare_train_test_splits(df_cleaned)
    print("\nSplits created for all target types.")

    # Print summary statistics
    print("\nDataset Summary:")
    print(f"Original dataset shape: {df_original.shape}")
    print(f"Cleaned dataset shape with multiple targets: {df_cleaned.shape}")
    print(f"Number of features: {len(feature_cols)}")
    print(f"Number of target variables: 6")

```

```
if __name__ == "__main__":
    main()
```

✓ add_factors_without_data_cleaning.py

```
# add_factors_without_data_cleaning.py
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from sklearn.preprocessing import MinMaxScaler
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier

class Config:
    """
    Configuration class to centralize all hyperparameters and settings.
    """

    # File paths and names
    FOLDER_PATH = ""
    INPUT_FILE = "klines_BTC.csv"
    OUTPUT_FILE = "klines_BTC_with_factors.csv"
    VISUALIZATION_FILE = "btc_targets_analysis.png"

    # Technical indicator parameters
    WINDOW_SIZE = 75
    VOLATILITY_WINDOW = 24
    MA_WINDOWS = [7, 25, 99]
    EMA_SHORT = 12
    EMA_LONG = 26
    MACD_SIGNAL = 9
    RSI_PERIOD = 14
    BOLLINGER_WINDOW = 20
    BOLLINGER_STD = 2

    # Outlier detection parameters
    ZSCORE_THRESHOLD = 3
    ANOMALY_PERCENTILE = 95

    # Autoencoder parameters
    AUTOENCODER_LAYERS = [64, 32, 16, 32, 64]
    REGULARIZATION_FACTOR = 0.001
    DROPOUT_RATE = 0.2
    EPOCHS = 50
    BATCH_SIZE = 32
    VALIDATION_SPLIT = 0.4
    EARLY_STOPPING_PATIENCE = 5

    # Random forest parameters
    RF_ESTIMATORS = 100
    RF_RANDOM_STATE = 42

    # Target classification thresholds
    PRICE_MOVEMENT_THRESHOLDS = [-2, -0.5, 0.5, 2]
    PRICE_MOVEMENT_LABELS = ['Significant Drop', 'Small Drop', 'Sideways', 'Small Rise', 'Significant Rise']

    # Train-test split parameters
    TEST_SIZE = 0.2
    FORECAST_HORIZON = 1

def compute_atr(df: pd.DataFrame, window_size: int) -> pd.Series:
    """
    Compute Average True Range (ATR) indicator.

    Parameters:
    -----
    df : pandas.DataFrame
        DataFrame containing OHLC price data
    window_size : int
        Rolling window size for ATR calculation

    Returns:
    -----
    pandas.Series: ATR values
    """
    high_low = df["high"] - df["low"]
    high_prev_close = abs(df["high"] - df["close"].shift(1))
```

```

low_prev_close = abs(df["low"] - df["close"].shift(1))

true_range = pd.concat([high_low, high_prev_close, low_prev_close], axis=1).max(axis=1)
return true_range.rolling(window=window_size).mean()

def load_and_clean_data(file_path: str) -> pd.DataFrame:
    """
    Load, clean, and preprocess raw price data.

    Parameters:
    -----
    file_path : str
        Path to the CSV file containing raw price data

    Returns:
    -----
    pandas.DataFrame: Cleaned dataframe
    """
    # Load the data
    df_original = pd.read_csv(file_path)

    # Convert timestamp column to datetime
    df_original["open_time"] = pd.to_datetime(df_original["open_time"])
    df_original.sort_values(by="open_time", inplace=True)

    # Make a copy before modification
    df_cleaned = df_original.copy()

    # Handle Missing Values
    df_cleaned.dropna(inplace=True)

    # Ensure Correct Data Types
    numeric_cols = ["open", "high", "low", "close", "volume",
                    "taker_buy_base_asset_volume", "taker_buy_quote_asset_volume"]
    df_cleaned[numeric_cols] = df_cleaned[numeric_cols].astype(float)

    # Remove Duplicates
    df_cleaned.drop_duplicates(subset=["open_time"], keep="first", inplace=True)

    return df_cleaned

def add_technical_indicators(df: pd.DataFrame) -> pd.DataFrame:
    """
    Add various technical indicators as features to the dataframe.

    Parameters:
    -----
    df : pandas.DataFrame
        DataFrame containing OHLC price data

    Returns:
    -----
    pandas.DataFrame: DataFrame with added technical indicators
    """
    # Price-based features
    df['return'] = df['close'].pct_change() # Price returns
    df['log_return'] = np.log(df['close']/df['close'].shift(1)) # Log returns
    df['volatility'] = df['log_return'].rolling(window=Config.VOLATILITY_WINDOW).std() # Volatility
    df['price_range'] = (df['high'] - df['low']) / df['open'] # Normalized price range

    # Moving averages
    for window in Config.MA_WINDOWS:
        df[f'ma_{window}'] = df['close'].rolling(window=window).mean()

    # ATR (Average True Range)
    df["atr"] = compute_atr(df, Config.WINDOW_SIZE)

    # MACD
    df['ema_12'] = df['close'].ewm(span=Config.EMA_SHORT).mean()
    df['ema_26'] = df['close'].ewm(span=Config.EMA_LONG).mean()
    df['macd'] = df['ema_12'] - df['ema_26']
    df['macd_signal'] = df['macd'].ewm(span=Config.MACD_SIGNAL).mean()
    df['macd_hist'] = df['macd'] - df['macd_signal']

    # RSI (Relative Strength Index)
    delta = df['close'].diff()
    gain = delta.where(delta > 0, 0)
    loss = -delta.where(delta < 0, 0)
    avg_gain = gain.rolling(window=Config.RSI_PERIOD).mean()
    avg_loss = loss.rolling(window=Config.RSI_PERIOD).mean()

```

```

rs = avg_gain / avg_loss
df['rsi'] = 100 - (100 / (1 + rs))

# Bollinger Bands
bb_window = Config.BOLLINGER_WINDOW
bb_std_dev = Config.BOLLINGER_STD
df['bb_middle'] = df['close'].rolling(window=bb_window).mean()
bb_std = df['close'].rolling(window=bb_window).std()
df['bb_upper'] = df['bb_middle'] + (bb_std * bb_std_dev)
df['bb_lower'] = df['bb_middle'] - (bb_std * bb_std_dev)
df['bb_width'] = (df['bb_upper'] - df['bb_lower']) / df['bb_middle']
df['bb_pct'] = (df['close'] - df['bb_lower']) / (df['bb_upper'] - df['bb_lower'])

# Volume features
df['volume_change'] = df['volume'].pct_change()
df['volume_ma_7'] = df['volume'].rolling(window=7).mean()
df['volume_relative'] = df['volume'] / df['volume_ma_7']

# OBV (On-Balance Volume)
df['obv'] = 0
df.loc[0, 'obv'] = df.loc[0, 'volume']
for i in range(1, len(df)):
    if df.loc[i, 'close'] > df.loc[i-1, 'close']:
        df.loc[i, 'obv'] = df.loc[i-1, 'obv'] + df.loc[i, 'volume']
    elif df.loc[i, 'close'] < df.loc[i-1, 'close']:
        df.loc[i, 'obv'] = df.loc[i-1, 'obv'] - df.loc[i, 'volume']
    else:
        df.loc[i, 'obv'] = df.loc[i-1, 'obv']

# Time-based features
df['hour'] = df['open_time'].dt.hour
df['day_of_week'] = df['open_time'].dt.dayofweek
df['is_weekend'] = df['day_of_week'].apply(lambda x: 1 if x >= 5 else 0)
df['month'] = df['open_time'].dt.month

# Remove rows with NaN values that resulted from calculating indicators
df.dropna(inplace=True)

return df

```

```

def add_target_variables(df: pd.DataFrame) -> pd.DataFrame:
    """
    Add multiple target variables for both regression and classification tasks.
    """

```

Parameters:

```

df : pandas.DataFrame
    DataFrame with price data and technical indicators

```

Returns:

```

pandas.DataFrame: DataFrame with added target variables
"""

```

```

# Target 1: Binary classification - will price go up in next period?
df['target_binary'] = (df['close'].shift(-1) > df['close']).astype(int)

```

```

# Target 2: Regression - percentage price change in next period
df['target_pct_change'] = df['close'].pct_change(periods=-1) * 100

```

```

# Target 3: Regression - absolute price change in next period
df['target_abs_change'] = df['close'].shift(-1) - df['close']

```

```

# Target 4: Multi-class classification - categorize price movement
thresholds = Config.PRICE_MOVEMENT_THRESHOLDS

```

```

def categorize_movement(pct_change):
    for i, threshold in enumerate(thresholds):
        if pct_change < threshold:
            return i
    return len(thresholds) # Last category

```

```

df['target_multiclass'] = df['target_pct_change'].apply(categorize_movement)

```

```

# Target 5: Regression - log return (often better for financial modeling)
df['target_log_return'] = np.log(df['close'].shift(-1) / df['close'])

```

```

# Target 6: Regression - normalized price change (compared to recent volatility)
volatility = df['close'].rolling(window=Config.BOLLINGER_WINDOW).std()
df['target_normalized_change'] = df['target_abs_change'] / volatility

```

```

# Remove NaNs from target creation

```

```

df.dropna(inplace=True)

return df

def detect_outliers(df: pd.DataFrame) -> np.ndarray:
    """
    Detect outliers using statistical methods and autoencoder.

    Parameters:
    -----
    df : pandas.DataFrame
        DataFrame with features

    Returns:
    -----
    numpy.ndarray: Boolean mask where True indicates an outlier
    """
    # Get feature columns (exclude timestamps and targets)
    feature_cols = [col for col in df.columns if col not in [
        'open_time', 'hour', 'day_of_week', 'is_weekend', 'month',
        'target_binary', 'target_pct_change', 'target_abs_change',
        'target_multiclass', 'target_log_return', 'target_normalized_change'
    ]]

    # Statistical outlier detection (Z-score method)
    def detect_outliers_zscore(df, columns, threshold=Config.ZSCORE_THRESHOLD):
        outliers_mask = np.zeros(len(df), dtype=bool)
        for col in columns:
            z_scores = np.abs((df[col] - df[col].mean()) / df[col].std())
            col_outliers = z_scores > threshold
            outliers_mask |= col_outliers
        return outliers_mask

    outliers_mask_zscore = detect_outliers_zscore(df, feature_cols)
    print(f"Z-score method identified {outliers_mask_zscore.sum()} outliers")

    # Autoencoder-based outlier detection
    scaler = MinMaxScaler()
    scaled_data = scaler.fit_transform(df[feature_cols])

    # Define autoencoder model with regularization
    input_dim = scaled_data.shape[1]
    layers = Config.AUTOENCODER_LAYERS

    # Build the autoencoder model
    autoencoder = keras.Sequential()
    # Encoder
    autoencoder.add(keras.layers.Dense(
        layers[0], activation='relu', input_shape=(input_dim,),
        kernel_regularizer=keras.regularizers.l2(Config.REGULARIZATION_FACTOR)))
    autoencoder.add(keras.layers.Dropout(Config.DROPOUT_RATE))

    for units in layers[1:len(layers)//2 + 1]:
        autoencoder.add(keras.layers.Dense(units, activation='relu'))

    # Decoder
    for units in layers[len(layers)//2 + 1:]:
        autoencoder.add(keras.layers.Dense(units, activation='relu'))

    autoencoder.add(keras.layers.Dense(input_dim, activation='linear'))

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

    # Train the autoencoder with early stopping
    early_stopping = keras.callbacks.EarlyStopping(
        monitor='val_loss',
        patience=Config.EARLY_STOPPING_PATIENCE,
        restore_best_weights=True
    )

    autoencoder.fit(
        scaled_data, scaled_data,
        epochs=Config.EPOCHS,
        batch_size=Config.BATCH_SIZE,
        shuffle=True,
        validation_split=Config.VALIDATION_SPLIT,
        callbacks=[early_stopping],
        verbose=1
    )

```

```

# Compute reconstruction error
reconstructed = autoencoder.predict(scaled_data)
reconstruction_error = np.mean(np.abs(scaled_data - reconstructed), axis=1)

# Define anomaly threshold
threshold = np.percentile(reconstruction_error, Config.ANOMALY_PERCENTILE)

# Combine outlier detection methods
outliers_mask_combined = (reconstruction_error > threshold) | outliers_mask_zscore
print(f"Combined methods identified {outliers_mask_combined.sum()} outliers")

return outliers_mask_combined

```

def analyze_feature_importance(df: pd.DataFrame, feature_cols: list) -> tuple:

Analyze feature importance for different target variables.

Parameters:

df : pandas.DataFrame
 DataFrame with features and targets

feature_cols : list
 List of feature column names

Returns:

tuple: Dataframes with feature importance for binary and regression targets

```

X = df[feature_cols]

# For binary classification
y_binary = df['target_binary']
clf = RandomForestClassifier(
    n_estimators=Config.RF_ESTIMATORS,
    random_state=Config.RF_RANDOM_STATE
)
clf.fit(X, y_binary)

binary_importances = pd.DataFrame({
    'feature': feature_cols,
    'importance': clf.feature_importances_
}).sort_values('importance', ascending=False)

print("\nTop 10 important features for binary classification:")
print(binary_importances.head(10))

# For regression (percentage change)
y_pct = df['target_pct_change']
reg = RandomForestRegressor(
    n_estimators=Config.RF_ESTIMATORS,
    random_state=Config.RF_RANDOM_STATE
)
reg.fit(X, y_pct)

reg_importances = pd.DataFrame({
    'feature': feature_cols,
    'importance': reg.feature_importances_
}).sort_values('importance', ascending=False)

print("\nTop 10 important features for price change regression:")
print(reg_importances.head(10))

```

return binary_importances, reg_importances

def visualize_data_and_targets(df_original: pd.DataFrame, df_cleaned: pd.DataFrame,

binary_importances: pd.DataFrame, reg_importances: pd.DataFrame):

Visualize various aspects of the dataset and targets.

Parameters:

df_original : pandas.DataFrame
 Original unprocessed DataFrame

df_cleaned : pandas.DataFrame
 Processed DataFrame with features and targets

binary_importances : pandas.DataFrame
 Feature importances for binary classification

reg_importances : pandas.DataFrame
 Feature importances for regression

```

plt.figure(figsize=(15, 15))

# Plot 1: Original vs Cleaned Price
plt.subplot(3, 2, 1)
plt.plot(df_original["open_time"], df_original["close"], label="Original", alpha=0.5)
plt.plot(df_cleaned["open_time"], df_cleaned["close"], label="Cleaned", alpha=0.8, color='red')
plt.title("Original vs Cleaned Price")
plt.legend()

# Plot 2: Binary Target Distribution
plt.subplot(3, 2, 2)
df_cleaned['target_binary'].value_counts().plot(kind='bar')
plt.title("Binary Target Distribution (Up/Down)")
plt.xlabel("Price Direction (1=Up, 0=Down)")
plt.ylabel("Count")

# Plot 3: Percentage Change Distribution
plt.subplot(3, 2, 3)
plt.hist(df_cleaned['target_pct_change'], bins=50)
plt.title("Price Percentage Change Distribution")
plt.xlabel("Percentage Change (%)")
plt.ylabel("Frequency")

# Plot 4: Multi-class Target Distribution
plt.subplot(3, 2, 4)
df_cleaned['target_multiclass'].value_counts().sort_index().plot(kind='bar')
plt.title("Multi-class Target Distribution")
plt.xlabel("Price Movement Category")
plt.ylabel("Count")
plt.xticks(range(len(Config.PRICE_MOVEMENT_LABELS)), Config.PRICE_MOVEMENT_LABELS, rotation=45)

# Plot 5: Correlation Between Different Target Variables
plt.subplot(3, 2, 5)
target_corr = df_cleaned[['target_binary', 'target_pct_change', 'target_abs_change',
                         'target_multiclass', 'target_log_return', 'target_normalized_change']].corr()
plt.imshow(target_corr, cmap='coolwarm')
plt.colorbar()
plt.title("Correlation Between Target Variables")
plt.xticks(range(6), target_corr.columns, rotation=90)
plt.yticks(range(6), target_corr.columns)

# Plot 6: Top features importance (combined importance)
plt.subplot(3, 2, 6)
combined_importance = binary_importances.set_index('feature')['importance'] + reg_importances.set_index('feature')['impc']
combined_importance.sort_values(ascending=False).head(10).plot(kind='bar')
plt.title("Top Features (Combined Importance)")
plt.tight_layout()
plt.xlabel("Feature")
plt.ylabel("Combined Importance Score")

plt.tight_layout()
plt.savefig(Config.VISUALIZATION_FILE)
plt.show()

```

```
def prepare_train_test_splits(df: pd.DataFrame, test_size=None, forecast_horizon=None):
    """
    Prepare chronological train/test splits for time series data.
    
```

Parameters:

```
-----
df : pandas.DataFrame
    The cleaned dataframe with features and targets
test_size : float, optional
    Proportion of data to use for testing
forecast_horizon : int, optional
    How many periods ahead to forecast

```

Returns:

```
-----
dict containing train/test splits for different target variables
"""

```

```
if test_size is None:
    test_size = Config.TEST_SIZE

```

```
if forecast_horizon is None:
    forecast_horizon = Config.FORECAST_HORIZON

```

```
# Feature columns (everything except targets and datetime)
feature_cols = [col for col in df.columns if col not in [
    'open_time', 'target_binary', 'target_pct_change', 'target_abs_change',
    'target_multiclass', 'target_log_return', 'target_normalized_change'
]]
```

```

]]]

# Split chronologically
split_idx = int(len(df) * (1 - test_size))
train_df = df.iloc[:split_idx].copy()
test_df = df.iloc[split_idx:].copy()

print(f"Training data from {train_df['open_time'].min()} to {train_df['open_time'].max()}")
print(f"Testing data from {test_df['open_time'].min()} to {test_df['open_time'].max()}")


# Prepare scalers
feature_scaler = MinMaxScaler()
X_train = feature_scaler.fit_transform(train_df[feature_cols])
X_test = feature_scaler.transform(test_df[feature_cols])


# Prepare regression target scalers
regression_scalers = {}
regression_targets = ['target_pct_change', 'target_abs_change',
                      'target_log_return', 'target_normalized_change']


y_train_dict = {}
y_test_dict = {}


# Classification targets
y_train_dict['binary'] = train_df['target_binary'].values
y_test_dict['binary'] = test_df['target_binary'].values


y_train_dict['multiclass'] = train_df['target_multiclass'].values
y_test_dict['multiclass'] = test_df['target_multiclass'].values


# Regression targets (scaled)
for target in regression_targets:
    scaler = MinMaxScaler()
    y_train_dict[target] = scaler.fit_transform(train_df[[target]])
    y_test_dict[target] = scaler.transform(test_df[[target]])
    regression_scalers[target] = scaler


return {
    'X_train': X_train,
    'X_test': X_test,
    'y_train': y_train_dict,
    'y_test': y_test_dict,
    'feature_scaler': feature_scaler,
    'regression_scalers': regression_scalers,
    'feature_columns': feature_cols,
    'train_dates': train_df['open_time'],
    'test_dates': test_df['open_time']
}
}

def main():
    """
    Main function to execute the entire data processing pipeline.
    """

    # Step 1: Load and clean data
    df_cleaned = load_and_clean_data(Config.INPUT_FILE)
    df_original = df_cleaned.copy() # Save original for visualization comparison

    # Step 2: Add technical indicators
    df_cleaned = add_technical_indicators(df_cleaned)

    # Step 3: Add target variables
    df_cleaned = add_target_variables(df_cleaned)

    # Step 4: Detect and remove outliers
    # outliers_mask = detect_outliers(df_cleaned)
    # df_cleaned = df_cleaned[~outliers_mask]

    # Get feature columns for analysis
    feature_cols = [col for col in df_cleaned.columns if col not in [
        'open_time', 'hour', 'day_of_week', 'is_weekend', 'month',
        'target_binary', 'target_pct_change', 'target_abs_change',
        'target_multiclass', 'target_log_return', 'target_normalized_change'
    ]]

    # Step 5: Analyze feature importance
    binary_importances, reg_importances = analyze_feature_importance(df_cleaned, feature_cols)

    # Step 6: Visualize data and targets
    visualize_data_and_targets(df_original, df_cleaned, binary_importances, reg_importances)

    # Step 7: Save cleaned data with multiple targets

```

```

df_cleaned.to_csv(Config.OUTPUT_FILE, index=False)

# Step 8: Prepare train/test splits
splits = prepare_train_test_splits(df_cleaned)
print("\nSplits created for all target types.")

# Print summary statistics
print("\nDataset Summary:")
print(f"Original dataset shape: {df_original.shape}")
print(f"Cleaned dataset shape with multiple targets: {df_cleaned.shape}")
print(f"Number of features: {len(feature_cols)}")
print(f"Number of target variables: 6")

if __name__ == "__main__":
    main()

```

▼ add_factors_drop.py

```

# add_factors_drop.py
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from sklearn.preprocessing import MinMaxScaler
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier

class Config:
    """
    Configuration class to centralize all hyperparameters and settings.
    """

    # File paths and names
    FOLDER_PATH = ""
    INPUT_FILE = "klines_BTC.csv"
    OUTPUT_FILE = "klines_BTC_with_factors.csv"
    VISUALIZATION_FILE = "btc_targets_analysis.png"

    # Technical indicator parameters
    WINDOW_SIZE = 75
    VOLATILITY_WINDOW = 24
    MA_WINDOWS = [7, 25, 99]
    EMA_SHORT = 12
    EMA_LONG = 26
    MACD_SIGNAL = 9
    RSI_PERIOD = 14
    BOLLINGER_WINDOW = 20
    BOLLINGER_STD = 2

    # Outlier detection parameters
    ZSCORE_THRESHOLD = 3
    ANOMALY_PERCENTILE = 95

    # Autoencoder parameters
    AUTOENCODER_LAYERS = [64, 32, 16, 32, 64]
    REGULARIZATION_FACTOR = 0.001
    DROPOUT_RATE = 0.2
    EPOCHS = 50
    BATCH_SIZE = 32
    VALIDATION_SPLIT = 0.4
    EARLY_STOPPING_PATIENCE = 5

    # Random forest parameters
    RF_ESTIMATORS = 100
    RF_RANDOM_STATE = 42

    # Target classification thresholds
    PRICE_MOVEMENT_THRESHOLDS = [-2, -0.5, 0.5, 2]
    PRICE_MOVEMENT_LABELS = ['Significant Drop', 'Small Drop', 'Sideways', 'Small Rise', 'Significant Rise']

    # Train-test split parameters
    TEST_SIZE = 0.2
    FORECAST_HORIZON = 1

def compute_atr(df: pd.DataFrame, window_size: int) -> pd.Series:
    """
    Compute Average True Range (ATR) indicator.
    """

```

Parameters:

```
df : pandas.DataFrame
    DataFrame containing OHLC price data
window_size : int
    Rolling window size for ATR calculation

Returns:
-----
pandas.Series: ATR values
"""
high_low = df["high"] - df["low"]
high_prev_close = abs(df["high"] - df["close"].shift(1))
low_prev_close = abs(df["low"] - df["close"].shift(1))

true_range = pd.concat([high_low, high_prev_close, low_prev_close], axis=1).max(axis=1)
return true_range.rolling(window=window_size).mean()
```

def load_and_clean_data(file_path: str) -> pd.DataFrame:

```
"""
Load, clean, and preprocess raw price data.
```

Parameters:

```
file_path : str
    Path to the CSV file containing raw price data
```

Returns:

```
pandas.DataFrame: Cleaned dataframe
"""
# Load the data
df_original = pd.read_csv(file_path)
```

```
# Convert timestamp column to datetime
df_original["open_time"] = pd.to_datetime(df_original["open_time"])
df_original.sort_values(by="open_time", inplace=True)
```

```
# Make a copy before modification
df_cleaned = df_original.copy()
```

```
# Handle Missing Values
df_cleaned.dropna(inplace=True)
```

```
# Ensure Correct Data Types
numeric_cols = ["open", "high", "low", "close", "volume",
                 "taker_buy_base_asset_volume", "taker_buy_quote_asset_volume"]
df_cleaned[numeric_cols] = df_cleaned[numeric_cols].astype(float)
```

```
# Remove Duplicates
df_cleaned.drop_duplicates(subset=["open_time"], keep="first", inplace=True)
```

```
return df_cleaned
```

def add_technical_indicators(df: pd.DataFrame) -> pd.DataFrame:

```
"""
Add various technical indicators as features to the dataframe.
```

Parameters:

```
df : pandas.DataFrame
    DataFrame containing OHLC price data
```

Returns:

```
pandas.DataFrame: DataFrame with added technical indicators
"""
# Price-based features
df['return'] = df['close'].pct_change() # Price returns
df['log_return'] = np.log(df['close']/df['close'].shift(1)) # Log returns
# df['volatility'] = df['log_return'].rolling(window=Config.VOLATILITY_WINDOW).std() # Volatility
# df['price_range'] = (df['high'] - df['low']) / df['open'] # Normalized price range
```

```
# Moving averages
for window in Config.MA_WINDOWS:
    df[f'ma_{window}'] = df['close'].rolling(window=window).mean()
```

```
# ATR (Average True Range)
# df["atr"] = compute_atr(df, Config.WINDOW_SIZE)
```

```

# MACD
df['ema_12'] = df['close'].ewm(span=Config.EMA_SHORT).mean()
df['ema_26'] = df['close'].ewm(span=Config.EMA_LONG).mean()
df['macd'] = df['ema_12'] - df['ema_26']
df['macd_signal'] = df['macd'].ewm(span=Config.MACD_SIGNAL).mean()
# df['macd_hist'] = df['macd'] - df['macd_signal']

# RSI (Relative Strength Index)
delta = df['close'].diff()
gain = delta.where(delta > 0, 0)
loss = -delta.where(delta < 0, 0)
avg_gain = gain.rolling(window=Config.RSI_PERIOD).mean()
avg_loss = loss.rolling(window=Config.RSI_PERIOD).mean()
rs = avg_gain / avg_loss
# df['rsi'] = 100 - (100 / (1 + rs))

# Bollinger Bands
bb_window = Config.BOLLINGER_WINDOW
bb_std_dev = Config.BOLLINGER_STD
df['bb_middle'] = df['close'].rolling(window=bb_window).mean()
bb_std = df['close'].rolling(window=bb_window).std()
df['bb_upper'] = df['bb_middle'] + (bb_std * bb_std_dev)
df['bb_lower'] = df['bb_middle'] - (bb_std * bb_std_dev)
df['bb_width'] = (df['bb_upper'] - df['bb_lower']) / df['bb_middle']
# df['bb_pct'] = (df['close'] - df['bb_lower']) / (df['bb_upper'] - df['bb_lower'])

# Volume features
# df['volume_change'] = df['volume'].pct_change()
# df['volume_ma_7'] = df['volume'].rolling(window=7).mean()
# df['volume_relative'] = df['volume'] / df['volume_ma_7']

# OBV (On-Balance Volume)
# df['obv'] = 0
# df.loc[0, 'obv'] = df.loc[0, 'volume']
# for i in range(1, len(df)):
#     if df.loc[i, 'close'] > df.loc[i-1, 'close']:
#         df.loc[i, 'obv'] = df.loc[i-1, 'obv'] + df.loc[i, 'volume']
#     elif df.loc[i, 'close'] < df.loc[i-1, 'close']:
#         df.loc[i, 'obv'] = df.loc[i-1, 'obv'] - df.loc[i, 'volume']
#     else:
#         df.loc[i, 'obv'] = df.loc[i-1, 'obv']

# Time-based features
df['hour'] = df['open_time'].dt.hour
df['day_of_week'] = df['open_time'].dt.dayofweek
df['is_weekend'] = df['day_of_week'].apply(lambda x: 1 if x >= 5 else 0)
df['month'] = df['open_time'].dt.month

# Remove rows with NaN values that resulted from calculating indicators
df.dropna(inplace=True)

return df

```

```

def add_target_variables(df: pd.DataFrame) -> pd.DataFrame:
    """
    Add multiple target variables for both regression and classification tasks.

    Parameters:
    -----
    df : pandas.DataFrame
        DataFrame with price data and technical indicators

    Returns:
    -----
    pandas.DataFrame: DataFrame with added target variables
    """
    # Target 1: Binary classification - will price go up in next period?
    df['target_binary'] = (df['close'].shift(-1) > df['close']).astype(int)

    # Target 2: Regression - percentage price change in next period
    df['target_pct_change'] = df['close'].pct_change(periods=-1) * 100

    # Target 3: Regression - absolute price change in next period
    df['target_abs_change'] = df['close'].shift(-1) - df['close']

    # Target 4: Multi-class classification - categorize price movement
    thresholds = Config.PRICE_MOVEMENT_THRESHOLDS

    def categorize_movement(pct_change):
        for i, threshold in enumerate(thresholds):
            if pct_change < threshold:

```

```

        return i
    return len(thresholds) # Last category

df['target_multiclass'] = df['target_pct_change'].apply(categorize_movement)

# Target 5: Regression - log return (often better for financial modeling)
df['target_log_return'] = np.log(df['close'].shift(-1) / df['close'])

# Target 6: Regression - normalized price change (compared to recent volatility)
volatility = df['close'].rolling(window=Config.BOLLINGER_WINDOW).std()
df['target_normalized_change'] = df['target_abs_change'] / volatility

# Remove NaNs from target creation
df.dropna(inplace=True)

return df

```

def detect_outliers(df: pd.DataFrame) -> np.ndarray:

Detect outliers using statistical methods and autoencoder.

Parameters:

df : pandas.DataFrame
DataFrame with features

Returns:

numpy.ndarray: Boolean mask where True indicates an outlier

```

# Get feature columns (exclude timestamps and targets)
feature_cols = [col for col in df.columns if col not in [
    'open_time', 'hour', 'day_of_week', 'is_weekend', 'month',
    'target_binary', 'target_pct_change', 'target_abs_change',
    'target_multiclass', 'target_log_return', 'target_normalized_change'
]]

```

```

# Statistical outlier detection (Z-score method)
def detect_outliers_zscore(df, columns, threshold=Config.ZSCORE_THRESHOLD):
    outliers_mask = np.zeros(len(df), dtype=bool)
    for col in columns:
        z_scores = np.abs((df[col] - df[col].mean()) / df[col].std())
        col_outliers = z_scores > threshold
        outliers_mask |= col_outliers
    return outliers_mask

```

```

outliers_mask_zscore = detect_outliers_zscore(df, feature_cols)
print(f"Z-score method identified {outliers_mask_zscore.sum()} outliers")

```

```

# Autoencoder-based outlier detection
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(df[feature_cols])

```

```

# Define autoencoder model with regularization
input_dim = scaled_data.shape[1]
layers = Config.AUTOENCODER_LAYERS

# Build the autoencoder model
autoencoder = keras.Sequential()
# Encoder
autoencoder.add(keras.layers.Dense(
    layers[0], activation='relu', input_shape=(input_dim,),
    kernel_regularizer=keras.regularizers.l2(Config.REGULARIZATION_FACTOR)))
autoencoder.add(keras.layers.Dropout(Config.DROPOUT_RATE))

for units in layers[1:len(layers)//2 + 1]:
    autoencoder.add(keras.layers.Dense(units, activation='relu'))

# Decoder
for units in layers[len(layers)//2 + 1:]:
    autoencoder.add(keras.layers.Dense(units, activation='relu'))

autoencoder.add(keras.layers.Dense(input_dim, activation='linear'))

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

# Train the autoencoder with early stopping
early_stopping = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=Config.EARLY_STOPPING_PATIENCE,

```

```

        restore_best_weights=True
    )

autoencoder.fit(
    scaled_data, scaled_data,
    epochs=Config.EPOCHS,
    batch_size=Config.BATCH_SIZE,
    shuffle=True,
    validation_split=Config.VALIDATION_SPLIT,
    callbacks=[early_stopping],
    verbose=1
)

# Compute reconstruction error
reconstructed = autoencoder.predict(scaled_data)
reconstruction_error = np.mean(np.abs(scaled_data - reconstructed), axis=1)

# Define anomaly threshold
threshold = np.percentile(reconstruction_error, Config.ANOMALY_PERCENTILE)

# Combine outlier detection methods
outliers_mask_combined = (reconstruction_error > threshold) | outliers_mask_zscore
print(f"Combined methods identified {outliers_mask_combined.sum()} outliers")

return outliers_mask_combined

```

def analyze_feature_importance(df: pd.DataFrame, feature_cols: list) -> tuple:

Analyze feature importance for different target variables.

Parameters:

df : pandas.DataFrame
DataFrame with features and targets

feature_cols : list
List of feature column names

Returns:

tuple: Dataframes with feature importance for binary and regression targets

X = df[feature_cols]

For binary classification
y_binary = df['target_binary']
clf = RandomForestClassifier(
 n_estimators=Config.RF_ESTIMATORS,
 random_state=Config.RF_RANDOM_STATE
)
clf.fit(X, y_binary)

binary_importances = pd.DataFrame({
 'feature': feature_cols,
 'importance': clf.feature_importances_
}).sort_values('importance', ascending=False)

print("\nTop 10 important features for binary classification:")
print(binary_importances.head(10))

For regression (percentage change)
y_pct = df['target_pct_change']
reg = RandomForestRegressor(
 n_estimators=Config.RF_ESTIMATORS,
 random_state=Config.RF_RANDOM_STATE
)
reg.fit(X, y_pct)

reg_importances = pd.DataFrame({
 'feature': feature_cols,
 'importance': reg.feature_importances_
}).sort_values('importance', ascending=False)

print("\nTop 10 important features for price change regression:")
print(reg_importances.head(10))

return binary_importances, reg_importances

def visualize_data_and_targets(df_original: pd.DataFrame, df_cleaned: pd.DataFrame,
 binary_importances: pd.DataFrame, reg_importances: pd.DataFrame):

Visualize various aspects of the dataset and targets.

Parameters:

```
-----
df_original : pandas.DataFrame
    Original unprocessed DataFrame
df_cleaned : pandas.DataFrame
    Processed DataFrame with features and targets
binary_importances : pandas.DataFrame
    Feature importances for binary classification
reg_importances : pandas.DataFrame
    Feature importances for regression
-----
plt.figure(figsize=(15, 15))

# Plot 1: Original vs Cleaned Price
plt.subplot(3, 2, 1)
plt.plot(df_original["open_time"], df_original["close"], label="Original", alpha=0.5)
plt.plot(df_cleaned["open_time"], df_cleaned["close"], label="Cleaned", alpha=0.8, color='red')
plt.title("Original vs Cleaned Price")
plt.legend()

# Plot 2: Binary Target Distribution
plt.subplot(3, 2, 2)
df_cleaned['target_binary'].value_counts().plot(kind='bar')
plt.title("Binary Target Distribution (Up/Down)")
plt.xlabel("Price Direction (1=Up, 0=Down)")
plt.ylabel("Count")

# Plot 3: Percentage Change Distribution
plt.subplot(3, 2, 3)
plt.hist(df_cleaned['target_pct_change'], bins=50)
plt.title("Price Percentage Change Distribution")
plt.xlabel("Percentage Change (%)")
plt.ylabel("Frequency")

# Plot 4: Multi-class Target Distribution
plt.subplot(3, 2, 4)
df_cleaned['target_multiclass'].value_counts().sort_index().plot(kind='bar')
plt.title("Multi-class Target Distribution")
plt.xlabel("Price Movement Category")
plt.ylabel("Count")
plt.xticks(range(len(Config.PRICE_MOVEMENT_LABELS)), Config.PRICE_MOVEMENT_LABELS, rotation=45)

# Plot 5: Correlation Between Different Target Variables
plt.subplot(3, 2, 5)
target_corr = df_cleaned[['target_binary', 'target_pct_change', 'target_abs_change',
                         'target_multiclass', 'target_log_return', 'target_normalized_change']].corr()
plt.imshow(target_corr, cmap='coolwarm')
plt.colorbar()
plt.title("Correlation Between Target Variables")
plt.xticks(range(6), target_corr.columns, rotation=90)
plt.yticks(range(6), target_corr.columns)

# Plot 6: Top features importance (combined importance)
plt.subplot(3, 2, 6)
combined_importance = binary_importances.set_index('feature')['importance'] + reg_importances.set_index('feature')['importance']
combined_importance.sort_values(ascending=False).head(10).plot(kind='bar')
plt.title("Top Features (Combined Importance)")
plt.tight_layout()
plt.xlabel("Feature")
plt.ylabel("Combined Importance Score")

plt.tight_layout()
plt.savefig(Config.VISUALIZATION_FILE)
plt.show()
```

```
def prepare_train_test_splits(df: pd.DataFrame, test_size=None, forecast_horizon=None):
    """
    Prepare chronological train/test splits for time series data.

```

Parameters:

```
-----
df : pandas.DataFrame
    The cleaned dataframe with features and targets
test_size : float, optional
    Proportion of data to use for testing
forecast_horizon : int, optional
    How many periods ahead to forecast

```

Returns:

```

-----  

dict containing train/test splits for different target variables  

"""  

if test_size is None:  

    test_size = Config.TEST_SIZE  

if forecast_horizon is None:  

    forecast_horizon = Config.FORECAST_HORIZON  

# Feature columns (everything except targets and datetime)  

feature_cols = [col for col in df.columns if col not in [  

    'open_time', 'target_binary', 'target_pct_change', 'target_abs_change',  

    'target_multiclass', 'target_log_return', 'target_normalized_change'  

]]]  

# Split chronologically  

split_idx = int(len(df) * (1 - test_size))  

train_df = df.iloc[:split_idx].copy()  

test_df = df.iloc[split_idx:].copy()  

print(f"Training data from {train_df['open_time'].min()} to {train_df['open_time'].max()}")
print(f"Testing data from {test_df['open_time'].min()} to {test_df['open_time'].max()}")
  

# Prepare scalers  

feature_scaler = MinMaxScaler()  

X_train = feature_scaler.fit_transform(train_df[feature_cols])  

X_test = feature_scaler.transform(test_df[feature_cols])  

# Prepare regression target scalers  

regression_scalers = {}  

regression_targets = ['target_pct_change', 'target_abs_change',  

                     'target_log_return', 'target_normalized_change']  

y_train_dict = {}
y_test_dict = {}  

# Classification targets  

y_train_dict['binary'] = train_df['target_binary'].values  

y_test_dict['binary'] = test_df['target_binary'].values  

y_train_dict['multiclass'] = train_df['target_multiclass'].values  

y_test_dict['multiclass'] = test_df['target_multiclass'].values  

# Regression targets (scaled)  

for target in regression_targets:  

    scaler = MinMaxScaler()  

    y_train_dict[target] = scaler.fit_transform(train_df[[target]])  

    y_test_dict[target] = scaler.transform(test_df[[target]])  

    regression_scalers[target] = scaler  

return {
    'X_train': X_train,
    'X_test': X_test,
    'y_train': y_train_dict,
    'y_test': y_test_dict,
    'feature_scaler': feature_scaler,
    'regression_scalers': regression_scalers,
    'feature_columns': feature_cols,
    'train_dates': train_df['open_time'],
    'test_dates': test_df['open_time']
}
}

def main():
"""
Main function to execute the entire data processing pipeline.
"""

# Step 1: Load and clean data
df_cleaned = load_and_clean_data(Config.INPUT_FILE)
df_original = df_cleaned.copy() # Save original for visualization comparison

# Step 2: Add technical indicators
df_cleaned = add_technical_indicators(df_cleaned)

# Step 3: Add target variables
df_cleaned = add_target_variables(df_cleaned)

# Step 4: Detect and remove outliers
outliers_mask = detect_outliers(df_cleaned)
df_cleaned = df_cleaned[~outliers_mask]

# Get feature columns for analysis

```

```

feature_cols = [col for col in df_cleaned.columns if col not in [
    'open_time', 'hour', 'day_of_week', 'is_weekend', 'month',
    'target_binary', 'target_pct_change', 'target_abs_change',
    'target_multiclass', 'target_log_return', 'target_normalized_change'
]]
# Step 5: Analyze feature importance
binary_importances, reg_importances = analyze_feature_importance(df_cleaned, feature_cols)

# Step 6: Visualize data and targets
visualize_data_and_targets(df_original, df_cleaned, binary_importances, reg_importances)

# Step 7: Save cleaned data with multiple targets
df_cleaned.to_csv(Config.OUTPUT_FILE, index=False)

# Step 8: Prepare train/test splits
splits = prepare_train_test_splits(df_cleaned)
print("\nSplits created for all target types.")

# Print summary statistics
print("\nDataset Summary:")
print(f"Original dataset shape: {df_original.shape}")
print(f"Cleaned dataset shape with multiple targets: {df_cleaned.shape}")
print(f"Number of features: {len(feature_cols)}")
print(f"Number of target variables: 6")

if __name__ == "__main__":
    main()

```

▼ PCA.py

```

# PCA.py
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Configuration
class Config:
    INPUT_FILE = "klines_BTC_with_factors.csv"
    OUTPUT_FILE = "klines_BTC_PCA.csv"
    EXPLAINED_VARIANCE_PLOT = "pca_explained_variance.png"
    N_COMPONENTS = 0.95 # Keep 95% variance

def load_data(file_path: str):
    """Load the dataset and remove non-numeric and target columns."""
    df = pd.read_csv(file_path)

    # Exclude columns that contain "target" (since they are labels)
    feature_cols = [col for col in df.columns if "target" not in col and df[col].dtype in [np.float64, np.int64]]

    print(f"✓ Initial feature count (excluding targets): {len(feature_cols)}")

    return df, df[feature_cols], feature_cols

def apply_pca(df_features, feature_names):
    """Apply PCA to reduce dimensionality while preserving variance."""
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(df_features)

    # Apply PCA
    pca = PCA(n_components=Config.N_COMPONENTS)
    X_pca = pca.fit_transform(X_scaled)

    # Number of selected components
    n_selected = pca.n_components_
    n_dropped = len(feature_names) - n_selected

    # Get retained feature names (sorted by explained variance)
    retained_features = np.array(feature_names)[np.argsort(-pca.explained_variance_ratio_)[:n_selected]]

    print(f"✓ PCA reduced dimensions to {n_selected}, preserving {Config.N_COMPONENTS * 100}% variance.")
    print(f"✗ Dropped {n_dropped} features (from {len(feature_names)} → {n_selected})")
    print(f"✓ Retained feature names: {list(retained_features)}")

    return X_pca, pca, n_selected, n_dropped, retained_features

def plot_explained_variance(pca):

```

```

"""Plot the cumulative explained variance ratio."""
plt.figure(figsize=(8, 5))
plt.plot(np.cumsum(pca.explained_variance_ratio_), marker="o", linestyle="--")
plt.xlabel("Number of Components")
plt.ylabel("Cumulative Explained Variance")
plt.title("PCA Explained Variance Ratio")
plt.grid()
plt.savefig(Config.EXPLAINED_VARIANCE_PLOT)
print(f"✅ Explained variance plot saved to {Config.EXPLAINED_VARIANCE_PLOT}")

def save_pca_data(df, X_pca, retained_features, output_path):
    """Save the PCA-transformed data along with original target labels."""
    df_pca = pd.DataFrame(X_pca, columns=[f"PC{i+1}" for i in range(X_pca.shape[1])])

    # Add back the target columns (which were excluded from PCA)
    target_cols = [col for col in df.columns if "target" in col]
    df_pca[target_cols] = df[target_cols].iloc[len(df) - len(df_pca):].reset_index(drop=True)

    df_pca.to_csv(output_path, index=False)
    print(f"✅ PCA-transformed data saved to {output_path}")

def main():
    """Main function to execute PCA processing."""
    df, df_features, feature_names = load_data(Config.INPUT_FILE)
    X_pca, pca, n_selected, n_dropped, retained_features = apply_pca(df_features, feature_names)
    plot_explained_variance(pca)
    save_pca_data(df, X_pca, retained_features, Config.OUTPUT_FILE)

if __name__ == "__main__":
    main()

```

▼ Part 3. Algorithm and Analysis

▼ predict_XGBoost.py (supervised learning)

```

# predict_XGBoost.py
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
import xgboost as xgb
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error

# Define configuration settings
class Config:
    """
    Configuration class to centralize all hyperparameters and settings.
    """

    # File paths and names
    FOLDER_PATH = ""
    INPUT_FILE = "klines_BTC_PCA.csv"
    OUTPUT_FILE = "klines_BTC_factors_with_direction.csv"
    VISUALIZATION_FILE = "learning_curve_xgboost.png"
    PREDICTION_PLOT_FILE = "predictions_vs_actual_xgboost.png"
    MODEL_FILE = "xgboost_model.json"

    # Train-test split parameters
    TEST_SIZE = 0.6

    # Training parameters
    N_ESTIMATORS = 1000
    LEARNING_RATE = 0.05
    MAX_DEPTH = 6
    SUBSAMPLE = 0.8
    COLSAMPLE_BYTREE = 0.8

def load_data(file_path: str) -> pd.DataFrame:
    """Load the CSV file and return a DataFrame."""
    return pd.read_csv(file_path).dropna()

def prepare_features(df: pd.DataFrame):
    """Prepare features for XGBoost, excluding attributes containing 'target'."""

    if 'target_pct_change' not in df.columns:
        raise ValueError('❌ "target_pct_change" column is missing from the dataset.')

```

```

# Select numerical features except those containing "target" and timestamp
feature_cols = [col for col in df.columns if "target" not in col and col != "open_time"]

if not feature_cols:
    raise ValueError("❌ No valid features selected! Check the dataset structure.")

print(f"✅ Selected {len(feature_cols)} features for training.")

X = df[feature_cols].values
y = df["target_pct_change"].values

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

return X_scaled, y, scaler, df

def train_xgboost(X, y, test_size=0.3):
    """Train an XGBoost model with Huber loss and return the trained model and evaluation results."""
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, shuffle=False)

    model = xgb.XGBRegressor(
        objective="reg:pseudohubererror", # ✅ Correct objective for Huber-like loss
        n_estimators=Config.N_ESTIMATORS,
        learning_rate=Config.LEARNING_RATE,
        max_depth=Config.MAX_DEPTH,
        subsample=Config.SUBSAMPLE,
        colsample_bytree=Config.COLSAMPLE_BYTREE,
        eval_metric="rmse" # Keep RMSE for evaluation tracking
    )

    model.fit(
        X_train, y_train,
        eval_set=[(X_train, y_train), (X_test, y_test)],
        verbose=True
    )

    evals_result = model.evals_result()

    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    print(f"✅ XGBoost Model MSE (Huber Loss): {mse:.4f}")

    return model, evals_result

def save_model(model, output_path: str):
    """Save the trained XGBoost model to a file."""
    model.save_model(output_path)
    print(f"✅ Model saved to {output_path}")

def save_predictions(df: pd.DataFrame, y_pred, output_path: str):
    """Save the predicted values to CSV file."""
    df['predicted_return'] = y_pred.flatten()
    df.to_csv(output_path, index=False)
    print(f"✅ Predictions saved to {output_path}")

def plot_predictions(y_true, y_pred, output_path: str):
    """Plot actual vs. predicted values and save the figure."""
    plt.figure(figsize=(10, 5))
    plt.plot(y_true, label="Actual", linestyle="solid", marker="o", markersize=3)
    plt.plot(y_pred, label="Predicted", linestyle="dashed", marker="s", markersize=3)
    plt.xlabel("Time Steps")
    plt.ylabel("Target Percentage Change")
    plt.title("Predicted vs Actual Values (XGBoost)")
    plt.legend()
    plt.grid()
    plt.savefig(output_path)
    print(f"✅ Prediction plot saved to {output_path}")

def plot_learning_curve(evals_result, output_path: str):
    """Plot the learning curve for XGBoost and save the image."""
    train_rmse = evals_result["validation_0"]["rmse"]
    test_rmse = evals_result["validation_1"]["rmse"]

    plt.figure(figsize=(8, 5))
    plt.plot(train_rmse, label="Train RMSE", marker="o")
    plt.plot(test_rmse, label="Test RMSE", marker="s")
    plt.xlabel("Boosting Rounds")

```

```

plt.ylabel("Root Mean Squared Error (RMSE)")
plt.title("XGBoost Learning Curve")
plt.legend()
plt.grid()
plt.savefig(output_path)
print(f"✅ Learning curve saved to {output_path}")

def main():
    """Main execution function."""
    file_path = Config.INPUT_FILE
    output_path = Config.OUTPUT_FILE
    learning_curve_path = Config.VISUALIZATION_FILE
    prediction_plot_path = Config.PREDICTION_PLOT_FILE
    model_path = Config.MODEL_FILE

    df = load_data(file_path)
    X, y, scaler, df_filtered = prepare_features(df)

    model, evals_result = train_xgboost(X, y, test_size=Config.TEST_SIZE)

    save_model(model, model_path)

    y_pred = model.predict(X)
    save_predictions(df_filtered, y_pred, output_path)

    plot_predictions(y, y_pred, prediction_plot_path)
    plot_learning_curve(evals_result, learning_curve_path)

if __name__ == "__main__":
    main()

```

✓ predict_transformer.py (supervised learning)

```

# predict_transformer.py
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.layers import Input, Dense, Dropout, LayerNormalization, MultiHeadAttention, GlobalAveragePooling1D, A
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Define configuration settings
class Config:
    """
    Configuration class to centralize all hyperparameters and settings.
    """

    # File paths and names
    FOLDER_PATH = ""
    INPUT_FILE = "klines_BTC_with_factors.csv"
    OUTPUT_FILE = "klines_BTC_factors_with_direction.csv"
    VISUALIZATION_FILE = "learning_curve_transformer.png"
    PREDICTION_PLOT_FILE = "predictions_vs_actual_transformer.png"
    MODEL_FILE = "transformer_model.h5"

    # Train-test split parameters
    TEST_SIZE = 0.6
    FORECAST_HORIZON = 1

    # Training parameters
    EPOCHS = 20
    BATCH_SIZE = 32
    TIME_STEPS = 60

def load_data(file_path: str) -> pd.DataFrame:
    """Load the CSV file and return a DataFrame."""
    return pd.read_csv(file_path).dropna()

def prepare_features(df: pd.DataFrame):
    """Prepare all available features for Transformer model, excluding attributes containing 'target'."""

    # Ensure target column exists
    if 'target_pct_change' not in df.columns:
        raise ValueError("❌ 'target_pct_change' column is missing from the dataset.")

    # Select all numerical features except those containing "target" and timestamp
    feature_cols = [col for col in df.columns if "target" not in col and col != "open_time"]

```

```

# Validate feature selection (ensure we have at least one feature)
if not feature_cols:
    raise ValueError('✖ No valid features selected! Check the dataset structure.')

print(f"✓ Selected {len(feature_cols)} features for training.")

# Extract feature matrix (X) and target variable (y)
X = df[feature_cols].values
y = df['target_pct_change'].values # Prediction target

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Define time step length for Transformer input
time_steps = Config.TIME_STEPS

# Create sequences for Transformer input
X_transformer = np.array([X_scaled[i-time_steps:i] for i in range(time_steps, len(X_scaled))])
y_transformer = y[time_steps:]

# Retain proper indexing from the original DataFrame
df_filtered = df.iloc[time_steps:].copy()

return X_transformer, y_transformer, scaler, df_filtered

def transformer_encoder(inputs, head_size=64, num_heads=4, ff_dim=128, dropout=0.2):
    """Transformer encoder block with dropout and L2 regularization."""
    x = MultiHeadAttention(num_heads=num_heads, key_dim=head_size)(inputs, inputs)
    x = Dropout(dropout)(x)
    x = Add()([x, inputs])
    x = LayerNormalization(epsilon=1e-6)(x)

    x_ff = Dense(ff_dim, activation="relu", kernel_regularizer=tf.keras.regularizers.l2(0.01))(x)
    x_ff = Dropout(dropout)(x_ff)
    x_ff = Dense(inputs.shape[-1])(x_ff)
    x = Add()([x, x_ff])
    x = LayerNormalization(epsilon=1e-6)(x)

    return x

def build_transformer_model(input_shape, num_layers=6):
    """Build Transformer model with multiple encoder layers."""
    inputs = Input(shape=input_shape)
    x = inputs
    for _ in range(num_layers):
        x = transformer_encoder(x)
    x = GlobalAveragePooling1D()(x)
    x = Dropout(0.02)(x)
    outputs = Dense(1)(x)

    model = Model(inputs, outputs)
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0005), loss='huber')
    return model

def train_transformer(X, y, test_size=0.3, epochs=20, batch_size=32):
    """Train a Transformer model and return the trained model and loss history."""
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, shuffle=False)

    model = build_transformer_model(input_shape=(X_train.shape[1], X_train.shape[2]))

    history = model.fit(
        X_train, y_train,
        epochs=epochs,
        batch_size=batch_size,
        validation_data=(X_test, y_test),
        verbose=1
    )
    return model, history

def plot_predictions(y_true, y_pred, output_path: str):
    """Plot actual vs. predicted values and save the figure."""
    plt.figure(figsize=(10, 5))
    plt.plot(y_true, label="Actual", linestyle="solid", marker="o", markersize=3)
    plt.plot(y_pred, label="Predicted", linestyle="dashed", marker="s", markersize=3)
    plt.xlabel("Time Steps")
    plt.ylabel("Target Percentage Change")
    plt.title("Predicted vs Actual Values")
    plt.legend()
    plt.grid()
    plt.savefig(output_path)

```

```

print(f"✅ Prediction plot saved to {output_path}")

def plot_learning_curve(history, output_path: str):
    """Plot the learning curve for Transformer and save the image."""
    plt.figure(figsize=(8, 5))
    plt.plot(history.history['loss'], label='Train Loss', marker='o')
    plt.plot(history.history['val_loss'], label='Test Loss', marker='s')
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.title("Transformer Learning Curve")
    plt.legend()
    plt.grid()
    plt.savefig(output_path)
    print(f"✅ Learning curve saved to {output_path}")

def save_model(model, output_path: str):
    """Save the trained model to a file."""
    model.save(output_path)
    print(f"✅ Model saved to {output_path}")

def save_predictions(df: pd.DataFrame, y_pred, output_path: str):
    """Save the predicted values to CSV file."""
    df['predicted_return'] = y_pred.flatten()
    df.to_csv(output_path, index=False)
    print(f"✅ Predictions saved to {output_path}")

def main():
    """Main execution function."""
    file_path = Config.INPUT_FILE
    output_path = Config.OUTPUT_FILE
    learning_curve_path = Config.VISUALIZATION_FILE
    prediction_plot_path = Config.PREDICTION_PLOT_FILE
    model_path = Config.MODEL_FILE

    df = load_data(file_path)
    X, y, scaler, df_filtered = prepare_features(df)

    model, history = train_transformer(X, y, test_size=Config.TEST_SIZE, epochs=Config.EPOCHS, batch_size=Config.BATCH_SIZE)

    save_model(model, model_path)

    y_pred = model.predict(X)
    save_predictions(df_filtered, y_pred, output_path)

    plot_predictions(y, y_pred, prediction_plot_path)
    plot_learning_curve(history, learning_curve_path)

if __name__ == "__main__":
    main()

```

✓ predict_kmeans.py (unsupervised learning)

```

# predict_kmeans.py
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, adjusted_rand_score, normalized_mutual_info_score, accuracy_score
from scipy.optimize import linear_sum_assignment
from sklearn.metrics import accuracy_score

# Define configuration settings
class Config:
    """
    Configuration class to centralize all hyperparameters and settings.
    """

    # File paths and names
    INPUT_FILE = "klines_BTC_with_factors.csv"
    OUTPUT_FILE = "klines_BTC_clusters.csv"
    CLUSTER_PLOT_FILE = "kmeans_clusters.png"

    # Clustering parameters
    TIME_STEPS = 60
    N_CLUSTERS = 5  # 分成 4 群
    RANDOM_STATE = 42  # 保證每次結果一樣

def load_data(file_path: str) -> pd.DataFrame:
    """Load the CSV file and return a DataFrame."""
    df = pd.read_csv(file_path).dropna()

```

```

return df

def prepare_features(df: pd.DataFrame):
    """Prepare time-series features for unsupervised learning."""

    # 過濾掉 target 和時間欄位，避免影響特徵選擇
    feature_cols = [col for col in df.columns if "target" not in col and col != "open_time"]

    if not feature_cols:
        raise ValueError("✖ No valid features selected! Check the dataset structure.")

    print(f"✓ Selected {len(feature_cols)} features for clustering.")

    # 轉換為矩陣
    X = df[feature_cols].values

    # 標準化
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    # 創建時間序列窗口
    time_steps = Config.TIME_STEPS
    if len(X_scaled) <= time_steps:
        raise ValueError("✖ Not enough data to generate sequences. Increase dataset size.")

    X_sequences = np.array([X_scaled[i-time_steps:i] for i in range(time_steps, len(X_scaled))])
    X_flat = X_sequences.reshape(X_sequences.shape[0], -1)

    print(f"✓ Prepared sequences for clustering: {X_flat.shape}")

    df_filtered = df.iloc[time_steps:].copy()

    return X_flat, scaler, df_filtered

def run_kmeans(X):
    """Run K-Means clustering on time-series sequences."""
    print(f"◆ Running K-Means on shape: {X.shape}")

    # Apply K-Means clustering
    kmeans = KMeans(n_clusters=Config.N_CLUSTERS, random_state=Config.RANDOM_STATE, n_init=10)
    clusters = kmeans.fit_predict(X)

    unique, counts = np.unique(clusters, return_counts=True)
    print("◆ Cluster distribution:", dict(zip(unique, counts)))

    return clusters, kmeans

from scipy.optimize import linear_sum_assignment
from sklearn.metrics import accuracy_score

def compute_accuracy(true_labels, predicted_clusters):
    """Compute accuracy by mapping predicted clusters to true labels using the Hungarian algorithm."""
    if len(true_labels) != len(predicted_clusters):
        print("⚠ Warning: Length mismatch between `target_multiclass` and predicted clusters. Adjusting.")
        min_len = min(len(true_labels), len(predicted_clusters))
        true_labels, predicted_clusters = true_labels[:min_len], predicted_clusters[:min_len]

    # Create a contingency matrix
    contingency_matrix = np.zeros((Config.N_CLUSTERS, Config.N_CLUSTERS), dtype=int)
    for i in range(len(true_labels)):
        contingency_matrix[true_labels[i], predicted_clusters[i]] += 1

    # Hungarian Algorithm for optimal label assignment
    row_ind, col_ind = linear_sum_assignment(contingency_matrix.max() - contingency_matrix)
    mapping = {col: row for row, col in zip(row_ind, col_ind)}

    # Map predicted clusters to the best-matching true labels
    predicted_mapped = np.array([mapping[c] for c in predicted_clusters])
    accuracy = accuracy_score(true_labels, predicted_mapped)

    return accuracy

def evaluate_clustering(X, clusters, kmeans, true_labels=None):
    """Evaluate clustering performance using internal and external metrics."""

    print("\n◆ Clustering Evaluation Metrics:")

    # Internal metrics
    silhouette_avg = silhouette_score(X, clusters)
    inertia = kmeans.inertia_

```

```

print(f"✓ Silhouette Score: {silhouette_avg:.4f} (higher is better)")
print(f"✓ Inertia (SSE): {inertia:.4f} (lower is better)")

if true_labels is not None:
    if len(true_labels) != len(clusters):
        print("⚠ Warning: `target_multiclass` and `clusters` have different lengths!")
    true_labels = true_labels[:len(clusters)]

    ari = adjusted_rand_score(true_labels, clusters)
    nmi = normalized_mutual_info_score(true_labels, clusters)
    accuracy = compute_accuracy(true_labels, clusters)

    print(f"✓ Adjusted Rand Index (ARI): {ari:.4f} (higher is better)")
    print(f"✓ Normalized Mutual Information (NMI): {nmi:.4f} (higher is better)")
    print(f"✓ Accuracy: {accuracy:.4f} (higher is better)")

return silhouette_avg, inertia

def plot_clusters(clusters, output_path: str):
    """Visualize the distribution of clusters."""

    unique_clusters, counts = np.unique(clusters, return_counts=True)

    plt.figure(figsize=(10, 5))
    plt.bar(unique_clusters, counts, alpha=0.7, color='blue', edgecolor='black')
    plt.xlabel("Cluster Label")
    plt.ylabel("Number of Samples")
    plt.title("K-Means Cluster Distribution")
    plt.grid()
    plt.savefig(output_path)
    print(f"✓ Cluster plot saved to {output_path}")

def save_clusters(df: pd.DataFrame, clusters, output_path: str):
    """Save the clustering results to a CSV file."""
    df = df.iloc[:len(clusters)].copy() # 確保長度匹配
    df['cluster'] = clusters
    df.to_csv(output_path, index=False)
    print(f"✓ Cluster labels saved to {output_path}")

def main():
    """Main execution function."""
    file_path = Config.INPUT_FILE
    output_path = Config.OUTPUT_FILE
    cluster_plot_path = Config.CLUSTER_PLOT_FILE

    df = load_data(file_path)
    X, scaler, df_filtered = prepare_features(df)

    clusters, kmeans = run_kmeans(X)

    # 確保 `target_multiclass` 存在
    if 'target_multiclass' in df_filtered.columns:
        true_labels = df_filtered['target_multiclass'].values
    else:
        true_labels = None
        print("⚠ Warning: `target_multiclass` column not found, external evaluation skipped.")

    # Evaluate clustering
    evaluate_clustering(X, clusters, kmeans, true_labels)

    save_clusters(df_filtered, clusters, output_path)
    plot_clusters(clusters, cluster_plot_path)

if __name__ == "__main__":
    main()

```

predict_GMM.py (unsupervised learning)

```

# predict_GMM.py
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score, adjusted_rand_score, normalized_mutual_info_score, accuracy_score
from hmmlearn.hmm import GaussianHMM

# Define configuration settings
class Config:
    .....

```

```

Configuration class to centralize all hyperparameters and settings.
"""

# File paths and names
INPUT_FILE = "klines_BTC_with_factors.csv"
OUTPUT_FILE = "klines_BTC_clusters_hmm.csv"
CLUSTER_PLOT_FILE = "hmm_clusters.png"

# HMM-GMM clustering parameters
N_HIDDEN_STATES = 5 # Hidden states (市場狀態數量)
MAX_ITER = 500 # HMM 訓練迭代次數
RANDOM_STATE = 42 # 保證每次結果相同
PCA_COMPONENTS = 0.87 # 保留 95% 資訊量

def load_data(file_path: str) -> pd.DataFrame:
    """Load the CSV file and return a DataFrame."""
    df = pd.read_csv(file_path).dropna()
    return df

def prepare_features(df: pd.DataFrame):
    """Prepare time-series features for HMM-GMM clustering."""

    # 過濾掉 target 和時間欄位，避免影響特徵選擇
    feature_cols = [col for col in df.columns if "target" not in col and col != "open_time"]

    if not feature_cols:
        raise ValueError("✖ No valid features selected! Check the dataset structure.")

    print(f"✓ Selected {len(feature_cols)} features for clustering.")

    # 轉換為矩陣
    X = df[feature_cols].values

    # 標準化
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    # 降維 (保留 95% 變異數)
    pca = PCA(n_components=Config.PCA_COMPONENTS)
    X_pca = pca.fit_transform(X_scaled)

    print(f"✓ 原始維度: {X_scaled.shape[1]}, PCA 降維後: {X_pca.shape[1]}")

    return X_pca, scaler, df

def run_hmm_gmm(X_pca):
    """Run HMM-GMM clustering on time-series data."""
    print(f"◆ Running HMM-GMM on shape: {X_pca.shape}")

    # 訓練 HMM-GMM
    hmm = GaussianHMM(n_components=Config.N_HIDDEN_STATES, covariance_type="full",
                       random_state=Config.RANDOM_STATE, n_iter=Config.MAX_ITER)

    hmm.fit(X_pca)
    hidden_states = hmm.predict(X_pca)

    # 印出隱藏狀態分佈
    unique, counts = np.unique(hidden_states, return_counts=True)
    print("◆ Hidden State Distribution:", dict(zip(unique, counts)))

    return hidden_states, hmm

def evaluate_clustering(X_pca, hidden_states, true_labels=None):
    """Evaluate the clustering performance with internal and external metrics."""

    print("\n◆ Clustering Evaluation Metrics:")

    # Internal metrics
    silhouette_avg = silhouette_score(X_pca, hidden_states)
    print(f"✓ Silhouette Score: {silhouette_avg:.4f} (higher is better)")

    if true_labels is not None:
        # 確保 `true_labels` 和 `hidden_states` 具有相同長度
        if len(true_labels) != len(hidden_states):
            print("⚠ Warning: `target_multiclass` and `hidden_states` have different lengths!")
            true_labels = true_labels[:len(hidden_states)]

        # External metrics
        ari = adjusted_rand_score(true_labels, hidden_states)
        nmi = normalized_mutual_info_score(true_labels, hidden_states)
        accuracy = accuracy_score(true_labels, hidden_states)

        print(f"✓ Adjusted Rand Index (ARI): {ari:.4f} (higher is better)")

```

```
print(f"✅ Normalized Mutual Information (NMI): {nmi:.4f} (higher is better)")
print(f"✅ Accuracy: {accuracy:.4f} (higher is better)")

return silhouette_avg

def plot_clusters(hidden_states, output_path: str):
    """Visualize the distribution of hidden states."""

    unique_states, counts = np.unique(hidden_states, return_counts=True)

    plt.figure(figsize=(10, 5))
    plt.bar(unique_states, counts, alpha=0.7, color='blue', edgecolor='black')
    plt.xlabel("Hidden State")
    plt.ylabel("Number of Samples")
    plt.title("HMM-GMM Hidden State Distribution")
    plt.grid()
    plt.savefig(output_path)
    print(f"✅ Cluster plot saved to {output_path}")

def save_clusters(df: pd.DataFrame, hidden_states, output_path: str):
    """Save the clustering results to a CSV file."""
    df = df.iloc[:len(hidden_states)].copy() # 確保長度匹配
    df['hidden_state'] = hidden_states
    df.to_csv(output_path, index=False)
    print(f"✅ Hidden states saved to {output_path}")

def main():
    """Main execution function."""
    file_path = Config.INPUT_FILE
    output_path = Config.OUTPUT_FILE
    cluster_plot_path = Config.CLUSTER_PLOT_FILE

    df = load_data(file_path)
    X_pca, scaler, df_filtered = prepare_features(df)

    hidden_states, hmm = run_hmm_gmm(X_pca)

    # 確保 `target_multiclass` 存在
    if 'target_multiclass' in df_filtered.columns:
        true_labels = df_filtered['target_multiclass'].values
    else:
        true_labels = None
        print("⚠ Warning: `target_multiclass` column not found, external evaluation skipped.")

    # Evaluate clustering
    evaluate_clustering(X_pca, hidden_states, true_labels)

    save_clusters(df_filtered, hidden_states, output_path)
    plot_clusters(hidden_states, cluster_plot_path)

if __name__ == "__main__":
    main()
```