

```
# 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}&startTime={start_time_ms}"

    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)

# 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
```

Analyze feature importance for different target variables.

```
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()
```

```
# 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 = 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 (pd.DataFrame): The input DataFrame containing the data.  
    feature_cols (list): A list of feature columns to analyze.  
    """  
    # Implement feature importance analysis logic here.  
    pass
```

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()

# 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()
```

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:
```

11

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

11

```

# 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')['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:
```

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()

# 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("X '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("X 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}")
```



```
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

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, Add

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,
```



```
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

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

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()

# 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()
```