EXPLORATORY PROJECT REPORT

Time Series Forecasting Using Quantum Machine Learning.

Dr.Vignesh Sivaraman
Dept. of Computer Science
IIT(BHU), Varanasi
Email:vignesh.cse@iitbhu.ac.in

Kovuru Geethesh Chandra
Dept. of Computer Science
IIT(BHU), Varanasi
Email:kgeethesh.chandra.cse23@iitbhu.ac.in

Rakesh Achutha
Dept. of Computer Science
IIT(BHU), Varanasi
Email:achutha.rakesh.mat20@itbhu.ac.in

I. Introduction:

Predicting financial data, such as stock prices or commodity values, is a difficult task due to many changing factors like economic news, global events, and investor behavior. These values are often volatile and do not follow stable patterns, which makes it hard for traditional models to track and predict future trends.

Older forecasting methods assume that data stays consistent over time, but in real markets, that is rarely true. Because of this, machine learning and deep learning methods are now widely used. They can work with large amounts of data and learn patterns that are too complex for basic models.

Quantum computing gives a new way to handle this problem. Quantum systems can explore many possibilities at the same time using special features like superposition and entanglement. This can make it faster to process complex and large financial datasets.

Some quantum algorithms, like variational quantum algorithms, work well for predicting time-based data. They are more flexible with changing trends in the market. Also, quantum circuits can represent complicated data in a simpler way, helping to spot useful patterns. This can lead to better predictions, even when the market changes often.

II. Dataset Descriptions:

In this section, we present the three datasets used for our stock prediction task. All the three datasets were sourced from the Yahoo Finance. The datasets include **S&P 500 Index (SP500)**, which captures the daily performance of 500 leading U.S. companies and includes values like open, high, low, close, and volume. The **NIFTY 50 Index (NIFTY)** represents the top 50 Indian companies from various sectors, with daily records of similar financial indicators. Lastly, the **WTI Crude Oil Prices (WTI)** dataset provides daily information on the price movements of West Texas Intermediate crude oil, including open, high, low, close, and trading volume. These datasets together offer a diverse view of stock and commodity market behaviour.

The **S&P 500 Index** dataset contains daily records for 505 individual stocks, covering the period from February 2013 to February 2023. It includes key financial indicators such as opening price, closing price, high, low, and volume. For evaluation, the dataset was split into two time frames: the training set spans from February 2013 to December 2017 using all 505 stocks, while the test set contains data from January 2018 to February 2023 using the overall S&P 500 index. This temporal separation was adopted to evaluate the model on completely unseen data, mimicking real-world forecasting conditions. The **NIFTY 50 Index** dataset includes data for 2152 stocks, with daily records from September 2007 to April 2025. The training set spans from September 2007 to July 2016, and the test set covers July 2016 to April 2025. This setup allows us to evaluate model performance across a broad range of Indian market conditions. The **WTI Crude Oil** dataset consists of 2542 daily entries, with the training set ranging from January 2005 to March 2015 and the test set from March 2015 to April 2025. This dataset offers a perspective from the energy commodity market, adding diversity to the prediction task.

Each dataset was pre-processed using standard data cleaning and normalization techniques. Initially, all rows containing missing values were removed to ensure data quality. After cleaning, the values were scaled using a technique similar to Min-Max scaling. This method transforms the data so that all values fall within the range of 0 to 1. It works by subtracting the minimum value in the dataset from each data point and then dividing by the overall range of the data. A small buffer was added to avoid any division by zero. This scaling ensures that all features contribute equally during the model training process.

III. Code Explanation:

1. Project directory structure: The project is located inside the directory ‘quantum-ml-main’ under the subdirectory named ‘QEncoder_SP500_prediction’.After downloading the zip file from Github, extract it and open the folder using terminal or any code editor of your choice. Navigate to quantum-ml-main (use command **cd quantum-ml-main**).This directory consists several sub directories and files. Create a new virtual environment by using command **python –m venv {name of environment}** and install all the required dependencies. It is available in the requirements.txt file. It can be done using the command **pip install –r requirements.txt.**

2. Linking the folder to github:

Scenario

- Initially, the project was downloaded as a ZIP from GitHub and used locally.
- Meanwhile, the original repository on GitHub was updated with new changes.
- We want to **safely merge updated GitHub code** while also **retaining locally generated files** (e.g., outputs, datasets).
- If you are using the code for the first time directly follow from step b below and skip step c.

a. Backup Existing Local Folder

We **rename the original local project folder** to keep its contents safe.

This ensures no files are lost before syncing with Git.

b. Clone the Latest Repository from GitHub

We clone the **fresh and updated version** of the repository from GitHub:

Use command :

git clone https://github.com/Networking-lab-iitbhu/QML-based-Time-Series-Forecasting.git

This creates a folder named QML-based-Time-Series-Forecasting with Git tracking and latest commits.

c. Move Locally Generated Files into the New Git Folder

From the **renamed backup folder** (QML-based-Time-Series-Forecasting-old), we **manually copy only the required files** (like newly generated dataset-related files) into the cloned folder.

So whatever files that were modified on the local computer before , every file has to be copied into the cloned folder.

d. Add and Commit These New Files

After copying the necessary files, we add and commit them:

Use the following commands:

```
cd QML-based-Time-Series-Forecasting
git add .
git commit -m "Added locally generated files from old run"
```

5. Push Final Changes to GitHub

Finally, we push everything to the GitHub repository:

Use command:

git push origin main

This updated the remote repository with:

- The latest GitHub code
- The locally generated files that were manually moved
- (There wont be any such locally generated files if you are using the code for the first time)

3. Sub-directories and files:

The QEncoder_SP500_prediction directory contains several sub-directories and files each of which is described in the following section.

datasets/:

This sub-directory contains all the input datasets (e.g., S&P500, Nifty50, and WTI) in CSV format, organized for easy access during dataset splitting and model training. Any new dataset to be included should also be placed in this sub-directory. The current files include: combined_dataset.csv (S&P500 dataset) , NIFTY50_Cleaned_Data.csv(Nifty50 dataset) and WTI_Offshore_Cleaned_Data.csv

encoder_details/:

This sub-directory contains all the details of the encoder related to a particular latent-trash qubit configuration. It has files named in a structured manner. It has a readme_file.txt as well.

It contains details such as encoder losses,weights stored during training and also contains the best encoder weights along with the trained encoder too.

evaluation_results/:

This sub-directory stores all the results obtained on both training and test sets. It also has a readme_file.txt. It has the weights of the model stored during training which can be later used to resume the training from a checkpoint or can be used to load the trained model to run on test set. It also has accs sub-directory which stores r2,mse,mape and mae.

processed_data/:

This sub-directory contains all the files that are obtained by processing the training and test set. Basically, this model uses a window size of 10 days (timestamps) to predict the opening price of the 11^{th} timestamp. Hence one training sample will consist of the data of all the 5 attributes (open,close,high,low,volume) for 10 timestamps. Similar format for test set also.

X_{dataset_name}_{test_percentage}.npy = training set partitioned into windows of size 10

Y_{dataset_name}_{test_percentage}.npy = corresponding training labels

tX_{dataset_name}_{test_percenatge}.npy = test set partitioned into windows of size 10

tY_{dataset_name}_{test_percenatge}.npy = corresponding test labels

F_{dataset_name}_{test_percentage}.npy = flattened dataset basically all the attributes (refer to code in dataset.py for more info)

test_percentage: train-test split percent

File Name: classification_circuits.py

Purpose:
This file defines the **Variational Quantum Circuit (VQC)** used for predicting stock trends based on 10-day features (Open, High, Low, Close, Volume). The design incorporates an **autoencoder circuit** to compress and process features sequentially using a method known as **Encode-Inject-Process (EIP)**.

Key Components:

a. Imports:

- Adds the project root to sys.path for relative imports.
- Imports:
    - pennylane for building quantum circuits.
    - torch and numpy for tensor operations.
    - autoencoder_circuit_trained from the encoder module.

b. Function: circuit_7(weights, args)
This function builds a layered quantum circuit using rotation gates and entanglement:

- For each layer (number of layers = args.depth):
    - Applies RX and RZ rotations to all qubits.
    - Applies Controlled-RZ (CRZ) gates between adjacent qubits.
    - Repeats with different arrangements.

Each parameter (weight) controls the angle of a rotation gate.
This design is inspired by the paper "Quantum Circuit Learning" (https://arxiv.org/pdf/1905.10876.pdf).

c. Function: keep_feature_from_padding(args, feature, time_step)
Checks whether all values in a feature vector (e.g., 10 days of open prices) are equal:

- If all values are the same, it returns **False** — the feature is likely padded or constant and uninformative.
- If values vary, returns **True** — meaning it's a valid feature to inject into the circuit.

This avoids injecting meaningless data into the quantum circuit.

d. Function: construct_classification_circuit(args, weights, features, trained_encoder=None)
This is the **main function** that builds the full quantum circuit for classification.

Step-by-step Overview:

1. **Setup**:
   Initializes a quantum device with num_latent + 2*num_trash + 1 wires (qubits).
   Returns a Pennylane QNode, which is differentiable and compatible with PyTorch.
2. **Input**:
   - features: A list of 5 arrays (Open, High, Low, Close, Volume), each containing 10 time steps.
   - weights: Learnable parameters for each quantum processing layer.
   - trained_encoder: A pretrained quantum autoencoder used for compressing previous feature states.
3. **First Feature (features[0])**:
   - Injected directly using qml.AngleEmbedding (rotation-X embedding).
   - Processed using the first circuit_7 block.
4. **Subsequent Features (features[1:]):**
   For each feature:
   - If it is not padded (verified by keep_feature_from_padding):
     - The previously held state is compressed using autoencoder_circuit_trained.
     - The new feature is injected using rotation-X and rotation-Y gates.
     - It is then processed using a new circuit_7 block.
5. **Final Layer:**
   After all valid features have been injected and processed, one final circuit_7 layer is applied to entangle and finalize the qubit state.
6. **Measurement:**
   The circuit returns the probability of measuring qubit 0 in the $|0\rangle$ state.
   This output is used as the classification probability.
7. **Return:**
   Returns the QNode — a quantum function that accepts inputs and produces prediction probabilities.
   It can be directly used in training or inference inside a PyTorch model.

Core Concepts Used:

Concept	Purpose
AngleEmbedding	Converts classical features to quantum states via parameterized gates.
autoencoder_circuit_trained	Compresses quantum state to reduce noise and retain essential info.
circuit_7	Trainable variational quantum layer for processing.
keep_feature_from_padding	Skips constant or padded features to improve efficiency and accuracy.
QNode	A quantum circuit compiled for integration with PyTorch models.

File Name: classification_model.py

Purpose:
Defines the **Classifier model**, a PyTorch module that wraps the **quantum classification circuit**. This model takes in stock features (Open, High, Low, Close, Volume over 10 days), passes them through a quantum circuit, and returns predictions for each input sample.

Key Components

1. Imports:

- Adds the root directory to sys.path for proper relative imports.
- Imports:
  - torch and torch.nn for defining the model.
  - construct_classification_circuit from classification_circuits.py, which builds the quantum circuit used for inference.

2. Class: Classifier(nn.Module)
Inherits from PyTorch's nn.Module. This class defines the main model architecture used for classification.

Constructor: __init__(self, encoder, args, model_weights=None)

- **Parameters:**
  - encoder: The pretrained quantum autoencoder used to compress inputs during inference.
  - args: Command-line arguments or configuration object containing:
    - num_latent: Number of latent qubits.
    - num_trash: Number of trash qubits.
    - depth: Number of layers in the variational quantum circuit.
    - n_cells: Number of times the Encode-Inject-Process cycle is applied.
  - model_weights: Optional learnable parameters; if not provided, initialized randomly.
- **Initialization:**
  - Computes the number of trainable quantum weights:
    - n_weights = ((4 * n_qubits - 2) * depth) * 4
    - The factor 4 accounts for multiple features processed by the circuit.
  - Initializes self.model_weights as a learnable nn.Parameter.

3. Forward Method: forward(self, features)

- **Input:**
  - features: A batch of stock data. Each element is a 5×10 matrix:
    - 5 features = [Open, High, Low, Close, Volume]
    - 10 days = time steps for each feature
  - Shape of features is typically [256, 5, 10] during training.
- **Process:**
  Iterates over each "document" (i.e., each 5×10 window):
  1. Calls construct_classification_circuit:
     - Passes in the current input window (document), the learnable model weights, the autoencoder, and args.
     - Returns a **quantum probability output**: [p_0, p_1], where p_0 is the probability of measuring |0⟩.
  2. The predicted value is set to p_0 (i.e., the 0th index of the returned array).
     - This is treated as the model's confidence for a particular class.
- **Return:**
  - Returns a tensor of predictions (floating-point probabilities) for the entire batch.

Concepts Used:

Concept	Explanation
nn.Parameter	A learnable tensor in PyTorch (used for quantum circuit weights).
construct_classification_circuit	Builds and returns a quantum circuit output for a single feature window.
p_0 probability	The circuit's prediction — probability that the measured qubit collapses to 0.

Concept	Explanation
Autoencoder	Used to compress previous feature information before injecting the next.

Summary:
The Classifier class wraps the **quantum classification process** in a PyTorch-compatible module. It:

- Accepts a batch of stock windows.
- Processes each window using a quantum circuit designed for classification.
- Uses a trained quantum autoencoder and learnable quantum weights.
- Returns a probability prediction (between 0 and 1) for each input.

This model is trainable using standard PyTorch loss functions and optimizers, making it suitable for hybrid quantum-classical learning.

File Name: dataset.py

This module is responsible for preparing the dataset to be fed into the quantum classifier model. It handles raw CSV reading, preprocessing, windowing of time-series data, and caching preprocessed results for faster reuse.

Key Responsibilities:

- Load datasets from either raw .csv files (datasets/ directory) or preprocessed .npy files (processed_data/ directory).
- Normalize each of the five features – Open, High, Low, Close, Volume – using MinMax scaling.
- Create fixed-length time windows of 10 days, with 5 features each, resulting in a shape of (samples, 5, 10).
- Store all processed NumPy arrays locally to avoid re-computation on future runs.

Supported Datasets:

- **WTI** – Crude oil dataset (WTI_Offshore_Cleaned_Data.csv)
- **NIFTY50** – Indian stock index dataset (NIFTY50_Cleaned_Data.csv)
- **S&P 500** – Composite of S&P500 and its individual stock histories (combined_dataset.csv)

Core Functions:

- load_and_split_csv(filename, args)
  Loads a CSV and returns train/test splits.
- fillxy(data)
  Applies normalization and creates a series of 10-day windows from the raw data. The target is the "Open" price on the 11th day.
- load_dataset(args)
  Main function that handles:
  - Checking if preprocessed files exist
  - Calling helper functions to load and preprocess raw data if needed
  - Saving the resulting arrays for future use
    Returns: X, Y, tX, tY, flattened

  Where:

  - X, Y → training data and labels
  - tX, tY → test data and labels
  - flattened → flattened feature vectors (used for PCA or visualizations)

Output Shapes:

- X: shape (num_samples, 5, 10) – contains 10 days of normalized features.
- Y: shape (num_samples,) – normalized "Open" price for the next (11th) day.
- flattened: shape (num_samples, 10) – used in feature compression or analysis.

File Name: encoder.py

Purpose:
This file implements a quantum autoencoder designed to compress and process stock market data features using a quantum circuit simulated by PennyLane. It defines the quantum circuit, wraps it into a PyTorch-compatible model, and provides a training function to optimize the model parameters.

Key Components:

a. Imports:

- pennylane for building and simulating quantum circuits.
- torch and torch.nn for PyTorch neural network integration and optimization.
- numpy from PennyLane for tensor operations.
- copy and os for handling model saving and checkpointing.

b. Function: construct_autoencoder_circuit(args, weights, features=None)

- Constructs the quantum autoencoder circuit with the following logic:
  - Initializes a PennyLane quantum device with wires based on the number of latent and trash qubits plus an auxiliary qubit.
  - Embeds classical data into the quantum circuit using AngleEmbedding on selected qubits.
  - Applies variational layers using BasicEntanglerLayers controlled by trainable weights.
  - Performs a swap test between trash qubits and auxiliary qubit to assess information loss.
  - Returns the probability distribution of measuring the auxiliary qubit.
- The function runs the quantum circuit and returns a PyTorch tensor of probabilities used as the model output.

c. Class: AutoEncoder(nn.Module)

- A PyTorch model wrapper that encapsulates the quantum autoencoder circuit.
- Initializes trainable parameters (weights) representing the circuit parameters.
- Defines a forward method that feeds features into the quantum circuit and returns the measurement probability corresponding to the encoded information quality.

d. Function: train_encoder(flattened, args)

- Manages the training process of the quantum autoencoder.
- Loads existing checkpoints if available to resume training or skips training if a best model exists.
- In each training iteration:
  - Samples a batch of data randomly from the input dataset.
  - Runs the forward pass of the autoencoder circuit.
  - Computes the loss (sum of probabilities).
  - Performs backpropagation and optimizer step (SGD).
  - Saves model weights and training losses periodically.
  - Keeps track of the best model based on loss and saves it separately.
- Saves the final trained model, training losses, and all intermediate weights for future use.

Additional Notes:

- The quantum circuit uses num_latent and num_trash qubit registers, where num_trash qubits hold information discarded in the compression process.

- The swap test with an auxiliary qubit is crucial for measuring the quality of compression in the quantum autoencoder.
- The file handles robust checkpointing and model saving to allow long training sessions without losing progress.
- The model output is designed to integrate seamlessly with classical training workflows using PyTorch optimizers.

Summary:
This file implements a novel quantum autoencoder integrated with PyTorch for training on classical data (such as stock market features). It defines the quantum circuit, wraps it into a trainable PyTorch model, and includes a training loop with checkpointing and logging to develop an effective quantum compression model.

File Name: main.py

Purpose:
This file serves as the main entry point for the project. It parses command-line arguments, loads datasets, trains the quantum autoencoder encoder and classifier, and handles switching between training and testing modes.

Key Components:

a. Imports:

- argparse for parsing command-line arguments.
- Project modules including train_loop.train, encoder.train_encoder, classification_model.Classifier, dataset.load_dataset & split_features_labels, and test.test.

b. Argument Parsing:

- Uses argparse.ArgumentParser to define configurable parameters such as:
  - Loss function type (MSE or BCE).
  - Number of training iterations.
  - Dataset choice (sp500, nifty, wti).
  - Number of EIP cells and circuit depth.
  - Learning rate and batch size.
  - Number of latent and trash qubits in the quantum autoencoder.
  - Mode (train or test).
  - Train/validation/test split ratios.

- Parses these arguments at runtime to flexibly control the experiment setup.

c. Dataset Loading:

- Calls load_dataset(args) to load the dataset based on the selected options.
- Receives processed inputs (X, Y) and test inputs (tX, tY), along with a flattened version of the data for the encoder.

d. Encoder Training:

- Trains the quantum autoencoder by invoking train_encoder(flattened, args).
- Returns a trained encoder model used later for classification.

e. Model Initialization:

- Instantiates the Classifier model, passing the trained encoder and the arguments.
- Prints the model's state dictionary keys for debugging purposes.

f. Dataset Splitting:

- Splits the dataset into training and validation sets using split_features_labels(X, Y, args).
- Assigns test data and labels separately for evaluation.

g. Training or Testing Control Flow:

- If mode is "train":
  - Calls the train() function to train the classifier with the training and validation sets.
  - Prints the size of the test set for debugging.
- If mode is "test":
  - Defines the directory containing pre-trained model weights.
  - Calls the test() function to evaluate the model on the test set.

Summary:
This file coordinates the entire workflow of the project. It handles argument parsing, dataset preparation, model training (both encoder and classifier), and testing, enabling flexible experimentation through command-line options.

File Name: train_loop.py

Purpose:
This file contains the training loop logic for the quantum-enhanced classifier model. It manages the optimization process, evaluates model performance on validation data, handles checkpoint saving and resuming training from saved checkpoints.

Key Components:

a. Imports:

- pennylane.numpy as np for numerical operations compatible with PennyLane quantum computing.
- torch and torch.optim for building and optimizing the PyTorch model.
- os for filesystem operations.
- Metrics from sklearn.metrics for evaluating regression performance:
  - r2_score
  - mean_squared_error (MSE)
  - mean_absolute_error (MAE)
  - mean_absolute_percentage_error (MAPE)

b. Function: accuracy(y, y_hat)

- Calculates regression performance metrics between true labels (y) and predicted labels (y_hat):
  - R^2 score
  - Mean Squared Error (MSE)
  - Mean Absolute Error (MAE)
  - Mean Absolute Percentage Error (MAPE)
- Returns all four metrics.

c. Function: train(model, train_set, labels_train, validation_set, labels_val, args)
Main training function that performs the following:

Training Setup:

- Prints a message indicating training has started.

- Initializes empty lists for storing loss and metrics history (losses, r2_scores, mses, maes, mapes).
- Creates an RMSprop optimizer with learning rate specified by args.lr.
- Defines directory paths for saving experiment results and weights under ./QEncoder_SP500_prediction/evaluation_results/.

Checkpoint Loading:

- Constructs file paths for latest and best model checkpoints based on experiment parameters (dataset, loss, depth, etc.).
- If a checkpoint exists, loads model and optimizer states and resumes training from the saved iteration/epoch, restoring stored loss and metric history.
- If no checkpoint is found, training starts from scratch.

Loss Function Selection:

- Supports "MSE" (mean squared error) and "BCE" (binary cross-entropy) losses based on args.loss.
- Throws an error if an unsupported loss is specified.

Training Loop:

- Runs from the starting iteration up to args.train_iter.
- For each iteration:
    - Clears model gradients.
    - Samples a random batch of indices from the training set.
    - Prepares batch labels and input features (only up to args.n_cells timesteps per sample).
    - Runs a forward pass through the model to get predictions.
    - Computes loss and performs backpropagation.
    - Calculates training metrics (R^2, MSE, MAE, MAPE) and prints them.
    - Stores training loss.

Validation & Saving:

- Every args.eval_every iterations:
    - Samples a random batch from the validation set and evaluates the model.
    - Calculates and prints validation metrics.
    - Stores validation metrics history.
    - Saves the model checkpoint as the "best" model if validation MSE is the lowest seen so far.

Checkpointing:

- Saves the current model, optimizer states, and metrics as the "latest" checkpoint every iteration to enable resuming.
- Also saves metrics arrays to .npy files for later analysis.

Return:

- Returns the trained model after finishing all iterations.

Summary:
This module robustly manages the model training process, including batch sampling, loss computation, optimizer stepping, evaluation on validation data, and saving/loading of checkpoints to facilitate training continuation. It also tracks and saves multiple performance metrics for detailed experiment analysis.

File Name: test.py

Purpose:
This file contains the testing logic for evaluating the trained quantum-enhanced classifier model on a test dataset. It loads the saved model checkpoint, runs inference on the test data, and computes performance metrics.

Key Components:

a. Imports:

- os for filesystem operations.
- torch for PyTorch model loading and tensor operations.
- metrics function from the local .metrics module for calculating evaluation metrics.

b. Function: test(model, args, test_dir, test_set, labels_test)

This function performs model evaluation on test data, consisting of the following steps:

Step 1: Define Experiment and Load Checkpoint

- Constructs a unique experiment name string from parameters: dataset, loss function, model depth, number of EIP cells, number of latent and trash qubits.
- Loads the saved model checkpoint from the specified directory (test_dir), specifically the file named {experiment}_weights_iteration_1.
- Prints the epoch number stored in the checkpoint for reference.
- Adjusts the loaded state dictionary keys if necessary (renames p_weights to model_weights) for compatibility.
- Loads the checkpointed model parameters into the model instance with strict=False to allow partial matching.

Step 2: Prepare for Evaluation

- Switches the model into evaluation mode with model.eval() to disable training-specific behaviors such as dropout and batch normalization updates.

Step 3: Inference on Test Set

- Disables gradient computation with torch.no_grad() to speed up evaluation and reduce memory usage.
- Converts the test_set to a PyTorch tensor and feeds it through the model to obtain predictions (test_out).

Step 4: Compute and Display Metrics

- Calls the imported metrics function, passing the model's predictions and the true test labels (labels_test) for performance evaluation and output.

Summary:
This module provides a streamlined testing procedure that loads a saved model checkpoint, runs inference on the given test dataset, and reports evaluation metrics. It ensures efficient evaluation by disabling gradients and properly setting the model to evaluation mode.

File Name: metrics.py

Purpose:
This file provides functions to calculate standard regression evaluation metrics and key financial performance indicators for stock prediction results. It enables comprehensive assessment of model predictions both statistically and financially.

Key Components:

a. Imports:

- numpy as np for numerical operations.
- Several metrics from sklearn.metrics for standard regression evaluation:
  - r2_score — coefficient of determination
  - mean_squared_error — average squared error
  - mean_absolute_error — average absolute error

b. Function: mape(y_true, y_pred)

- Computes Mean Absolute Percentage Error (MAPE), expressing prediction error as a percentage.
- Formula: average of absolute relative errors multiplied by 100.
- Input: true values and predicted values (arrays).
- Output: percentage error value.

c. Function: max_drawdown(returns)

- Calculates the Maximum Drawdown, which measures the largest drop from a peak to a trough in cumulative returns.
- Uses cumulative product of returns to simulate growth.
- Returns the worst relative drop (negative value) experienced during the period.
- Important metric to quantify downside risk in finance.

d. Function: sharpe_ratio(returns, risk_free_rate=0.0)

- Computes the Sharpe Ratio, a popular risk-adjusted return metric.
- Defined as mean excess return over the standard deviation of returns.
- Higher Sharpe Ratio indicates better return per unit risk.

- Uses a small epsilon to avoid division by zero.

e. Function: sortino_ratio(returns, risk_free_rate=0.0)

- Calculates the Sortino Ratio, similar to Sharpe but penalizes only downside volatility.
- Uses standard deviation of negative returns (below risk-free rate) in denominator.
- Provides a more conservative risk assessment focused on harmful volatility.

f. Function: metrics(y_pred, y_true)

- Main evaluation function called after predictions are made.
- Flattens input arrays to 1D vectors for compatibility.
- Computes and prints:
    - R2 Score (goodness of fit)
    - MSE (mean squared error)
    - MAE (mean absolute error)
    - MAPE (percentage error)
    - Total Return % based on predicted returns
    - Sharpe Ratio (risk-adjusted return)
    - Sortino Ratio (downside risk-adjusted return)
    - Maximum Drawdown (largest loss from peak)

Summary:
This module aggregates important statistical and financial metrics into one interface for evaluating model performance on stock price prediction tasks. It is critical for understanding both accuracy and risk-adjusted returns from the predictions.

IV. Some commands:

To run the code:

python -m "QEncoder_SP500_prediction.main" --dataset {dataset_name} --mode {train/test} –train_iter {value} –encoder_train_iter{value}

any argument that is not used in the above command will take its default value defined in main.py.