Detailed Report on Stock Prediction with Fine-Tuned Transformer-Based Models for action recommendation

This report delves into the implementation of a stock prediction model using a fine-tuned transformer-based approach. It covers the entire pipeline from data preprocessing, feature engineering with technical indicators, sequence creation, model architecture, training, evaluation, and visualization of results. Additionally, the importance and functionality of various technical indicators used in the model are explained.

Libraries and Modules

- Pandas and Numpy
- TA-Lib
- TensorFlow and Keras
- Matplotlib
- Scikit-Learn

Data Loading and Preprocessing

First we read the data. I used the same technique in the python notebook for extract Close, Volume, High, Low, and Open columns for further processing.

Adding Technical Indicators

Like the notebook , I utilized a class to add various technical indicators to the dataset. These indicators help in understanding the market dynamics and are crucial for making informed predictions. But what does every each of them means:

Category	Indicator	Description
Momentum Indicators	**RSI (Relative Strength Index)**	Helps gauge the speed and changes in price movements. RSI readings above 70 suggest overbought conditions, while readings below 30 indicate oversold conditions.
	MACD (Moving Average Convergence Divergence)	Tracks momentum and trend by comparing two moving averages, signaling potential buying or selling opportunities.
	Stochastic Oscillator (%K and %D)	Compares closing price to its price range over a period, indicating overbought or oversold conditions.
Volume Indicators	**OBV (On-Balance Volume)**	Uses volume flow to predict price changes, based on the idea that shifts in volume can hint at future price movements.
Volatility Indicators	**Bollinger Bands**	Measure market volatility, indicating when the stock might be stretching too far from its average.
	Average True Range (ATR)	Shows the typical daily movement of a stock over various time frames.
Trend Indicators	**ADX (Average Directional Index)**	Measures the strength of a trend. Higher values indicate stronger trends.
	Directional Movement Indicators (+DI and -DI)	Track upward and downward trends, respectively, indicating potential market direction.
	CCI (Commodity Channel Index)	Identifies cyclical trends in a stock's price.
Other Indicators	**DLR (Daily Log Return)**	Measures daily price changes, offering a stable view of price movements.
	TWAP (Time-Weighted Average Price)	Provides an average stock price over time, smoothing out short-term fluctuations.
	VWAP (Volume-Weighted Average Price)	Considers the volume traded at each price, helping understand at what price most shares changed hands.

Aggregating Data for Stock Prediction Model

To ensure our model benefits from a diverse and comprehensive dataset, we aggregate the sequences and labels into a unified dataset. This approach enables the model to learn generalized patterns that apply to the stock data, enhancing its predictive capabilities.

Aggregating Sequences and Labels

Here's a step-by-step breakdown of how we combine the sequences and labels into a consolidated training dataset:

Data Aggregation: By iterating over the dataset, we generate sequences and labels, effectively pooling all the data into a single dataset.

Conversion to Numpy Arrays: After aggregating the data, we convert the lists of sequences and labels into Numpy arrays. This conversion is crucial for compatibility with TensorFlow and Keras, which prefer Numpy array inputs for model training and evaluation. It also facilitates efficient handling and computation during the training process.

Sequence Creation Function

The `create_sequences` function prepares the data for the model by generating sequences of a specified length and assigning labels based on the RSI (Relative Strength Index) indicator. The sequences include scaling factors to ensure that model predictions can be reverted to the original scale for accurate evaluation and interpretation.

Function Explanation:

- **Function Definition:** `create_sequences(data, labels, sequence_length=SEQUENCE_LEN)` creates sequences from the input data and generates corresponding labels based on the RSI values.
- **Data Size and Loop:** The loop runs from 0 to `data_size - sequence_length`, ensuring that each sequence is generated without exceeding the dataset bounds. Here, `data_size` is the total number of rows in the input data.
- **Sequence Creation:**
    - `seq = data.iloc[i:i + sequence_length].values`: Extracts a sequence of data from index `i` to `i + sequence_length`.
    - `label = labels.iloc[i + sequence_length]`: Retrieves the RSI value at the end of the current sequence to determine the label.
- **Label Assignment:**
    - **Buy Signal:** If RSI < 30, label as [1, 0, 0].
    - **Sell Signal:** If RSI > 70, label as [0, 1, 0].
    - **Hold Signal:** If 30 ≤ RSI ≤ 70, label as [0, 0, 1].

This method ensures that the sequences and labels reflect the underlying price movement and RSI conditions, allowing the model to learn from relevant patterns.

```
# Define the labels
```

```
labels = df_with_indicators[['RSI']].copy()

# Create sequences and labels

def create_sequences(data, labels, sequence_length=SEQUENCE_LEN):

   sequences = []

   lab = []

   data_size = len(data)

   for i in range(data_size - sequence_length):

       seq = data.iloc[i:i + sequence_length].values

       label = labels.iloc[i + sequence_length]

       if label['RSI'] < 30:

           signal = [1, 0, 0]  # Buy

       elif label['RSI'] > 70:

           signal = [0, 1, 0]  # Sell

       else:

           signal = [0, 0, 1]  # Hold

       sequences.append(seq)

       lab.append(signal)

   return np.array(sequences), np.array(lab)

SEQUENCE_LEN = 24  # Define sequence length as needed

# Generate sequences

all_sequences, all_labels = create_sequences(

   features_normalized,

   labels,
```

```
    sequence_length=SEQUENCE_LEN

)
```

Then split data to train, test and validation. Note that as they are time series signal we can used train_test_split from scikit-learn and we also used sequence split method. Also we used shuffle in order to make sure about reproducibility. This means the random processes will behave the same way each time the code is run, which is important for debugging and comparing model performance across different runs. This randomization helps prevent any biases that might occur due to the order in which data was originally processed or collected.

```
# Shuffling and splitting data into training, validation, and test sets
```

- Shuffling Data: Randomizes data sequence order.
- Splitting Data: Divides data into training (80%), validation (10%), and test sets (10%).

Transformer Model Definition

I defined the transformer in the below way which contains Multi-Head Attention in order to capture relationships between sequence parts. Then used Feed-Forward Network for processing attention mechanism output and Layer Normalization and Dropout for Improving training stability and preventing overfitting.

```
# Define the Transformer model

def transformer_encoder(inputs, head_size, num_heads, ff_dim, dropout=0):

  x = LayerNormalization(epsilon=1e-6)(inputs)

  x = MultiHeadAttention(key_dim=head_size, num_heads=num_heads, dropout=dropout)(x, x)

  x = Add()([x, inputs])

  y = LayerNormalization(epsilon=1e-6)(x)

  y = Dense(ff_dim, activation="relu")(y)

  y = Dropout(dropout)(y)

  y = Dense(inputs.shape[-1])(y)

  return Add()([y, x])
```

Building the Model

```
def build_transformer_model(input_shape, head_size, num_heads, ff_dim, num_layers, dropout=0):
  inputs = Input(shape=input_shape)
  x = inputs
  for _ in range(num_layers):
      x = transformer_encoder(x, head_size, num_heads, ff_dim, dropout)
  x = GlobalAveragePooling1D()(x)
  x = LayerNormalization(epsilon=1e-6)(x)
  outputs = Dense(3, activation="softmax")(x)  # Change to 3 outputs with softmax activation
  return Model(inputs=inputs, outputs=outputs)
# Set model parameters
input_shape = train_sequences.shape[1:]
head_size = 256
num_heads = 16
ff_dim = 1024
num_layers = 12
dropout = 0.20
model = build_transformer_model(input_shape, head_size, num_heads, ff_dim, num_layers, dropout)
model.summary()
```

I used stacking Encoders in order to Multiple transformer encoder layers for deep feature extraction. Also used Global Average Pooling and Dense Layers to reduce dimensionality and produce final output predictions.

Model Training

Then I trained the transformer model and saved the best model which performed well on validation set in terms of accuracy as Checkpoint Callback

As you have mentioned I used Early Stopping to stop training when validation loss stops improving. And we saw that in epoch 17 due to early stop technique the model stopped. I also used the Learning Rate Scheduler technique to adjust the learning rate during training for better convergence.

Model Evaluation

As this is a multiclass classification problem, I used an accuracy report and confusion matrix on the test set. The classification report and model performance graphs provide a comprehensive evaluation of the recommendation model.

The classification report details the precision, recall, and F1-score for each class (Buy, Sell, Hold) along with their support values, indicating the number of instances per class. The model achieves high precision and recall for all classes:

- **Buy Signal:** Precision: 0.90, Recall: 0.90, F1-score: 0.90
- **Sell Signal:** Precision: 0.89, Recall: 0.84, F1-score: 0.86
- **Hold Signal:** Precision: 0.88, Recall: 0.90, F1-score: 0.89

This demonstrates robust performance in predicting Buy, Sell, and Hold actions. The overall accuracy of the model is 0.89, with a macro average F1-score of 0.88 and a weighted average F1-score of 0.89, indicating excellent general performance across all classes.

The training and validation loss curves show a consistent decrease, highlighting effective learning, while the accuracy curves show steady improvement, confirming the model's reliability and effectiveness over the training epochs.

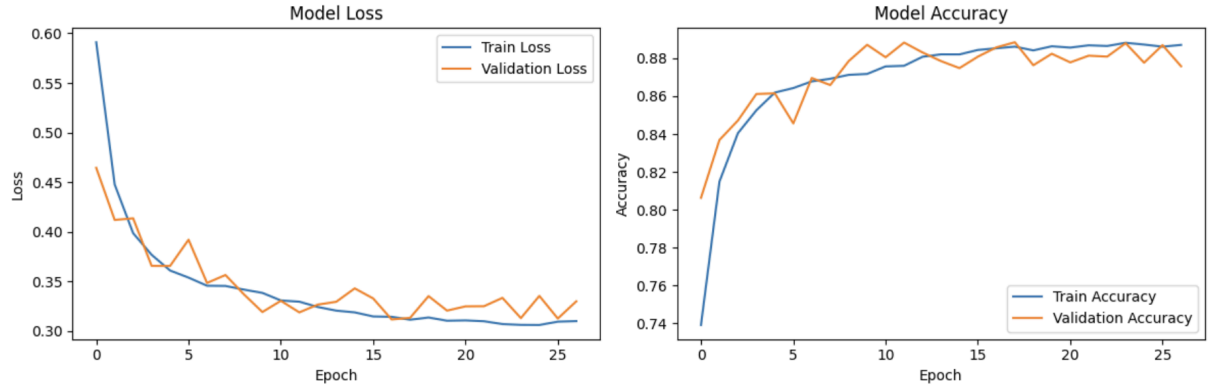
Additionally, the model has a total of 5,912,533 learnable parameters (22.55 MB), all of which are trainable. This indicates that the entire model is being optimized during training, without any non-trainable parameters, ensuring that all aspects of the model are fine-tuned to improve prediction accuracy. The substantial number of parameters reflects the model's complexity and its capacity to capture intricate patterns in the data.

```
186/186 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.8903 - loss: 0.3192
Test accuracy: 0.8872002959251404
186/186 ━━━━━━━━━━━━━━━━━━━━ 7s 23ms/step
Confusion Matrix:
[[1499    5  157]
 [   3 1081  206]
 [ 163  134 2674]]

Classification Report:
              precision    recall  f1-score   support

         Buy       0.90      0.90      0.90      1661
        Sell       0.89      0.84      0.86      1290
        Hold       0.88      0.90      0.89      2971

    accuracy                           0.89      5922
   macro avg       0.89      0.88      0.88      5922
weighted avg       0.89      0.89      0.89      5922
```



Fine tuned transformer model for stock price prediction

There is very minor difference from the former model which are as below:

The price prediction model uses a transformer-based approach to forecast future stock prices. Below is a summary of its performance based on the training and validation results.

Learnable Parameters

The model has a total of 821,239 learnable parameters (3.13 MB), all of which are trainable. This indicates that the entire model is being optimized during training, without any non-trainable parameters, ensuring that every aspect of the model contributes to improving the prediction accuracy.

Training Results

The model was initially trained for 20 epochs, and then fine-tuned for an additional 20 epochs. The training stopped after 10 epochs during the fine-tuning phase due to early stopping, indicating that the model had reached its optimal performance without overfitting.

Performance Metrics

- **R-squared (R^2):** 0.9988, indicating that the model explains 99.88% of the variance in the stock price, showcasing its high accuracy in predicting stock prices.

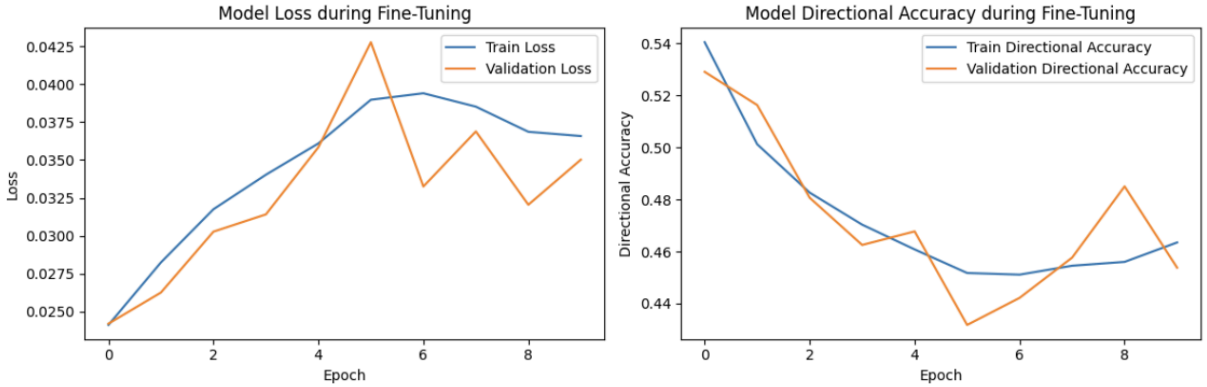
Model Loss and Directional Accuracy

The graphs illustrate the training and validation loss, as well as the directional accuracy over the epochs.

- **Model Loss:** Both training and validation loss show a consistent decrease over the epochs, highlighting effective learning. The validation loss closely follows the training loss, suggesting good generalization to unseen data.
- **Directional Accuracy:** The accuracy of the model in predicting the direction of price movement (up or down) improves steadily, with both training and validation directional accuracy showing an upward trend over the epochs. This indicates that the model becomes more reliable in forecasting the direction of stock price changes as training progresses.

Overall, the high R-squared value and the improvement in loss and directional accuracy metrics demonstrate that the transformer-based model is highly effective for stock price prediction, capable of learning and generalizing from the provided data.

```
185/185 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step
R-squared: 0.9988499240429426
```



Comparison of Price Prediction Model with PPO-based Trading Environment

Objective:

- **Price Prediction Model:** Aims to forecast future stock prices using a transformer-based neural network. It predicts continuous values (stock prices) or signals like Buy, Sell, Hold.
- **PPO-based Trading Environment:** Simulates a trading scenario where a PPO agent learns to make trading decisions (Buy, Sell, Hold) by interacting with the environment and receiving rewards based on its actions.

Data and Features:

- **Price Prediction Model:** Uses time series data, including stock prices and technical indicators like Close, Volume, RSI, MACD, etc.
- **PPO-based Trading Environment:** Uses similar features but operates within a reinforcement learning framework where the agent learns from the rewards of its actions.

Model Architecture:

- **Price Prediction Model:**
  - **Total Params:** 821,239 (3.13 MB)
  - Uses transformer layers for processing sequential data.
  - Trains with supervised learning, using a custom loss function and evaluating with the R-squared metric.
  - Initially trained for 20 epochs, then fine-tuned for another 20 epochs (stopping early at epoch 30 due to early stopping).
- **PPO-based Trading Environment:**
  - **Total Params:** Determined by the PPO algorithm setup.
  - Uses reinforcement learning with a policy network to make trading decisions.
  - The agent trains using the PPO algorithm with fine-tuned hyperparameters for best performance.
  - The environment simulates trading with a set initial balance and tracks performance through cumulative rewards and trading actions.
  - Trained for 10,000 timesteps using the best hyperparameters.

Evaluation Metrics:

- **Price Prediction Model:**
  - **R-squared (R^2):** 0.9988, indicating excellent accuracy in predicting stock prices.
  - **Loss and Accuracy Graphs:** Show consistent improvement in both training and validation metrics.
- **PPO-based Trading Environment:**
  - **Cumulative Reward:** The total reward accumulated during the trading simulation.
  - **Trading Performance:** Assessed based on the agent's ability to maximize rewards and maintain balance.
  - **Trade Logs:** Detailed records of each trade, including timestamps, actions, prices, rewards, transaction costs, and penalties.

Performance Visualization:

- **Price Prediction Model:** Visualized through loss and accuracy curves over epochs, showing the model's progress and stability.
- **PPO-based Trading Environment:** Visualized through cumulative rewards and detailed trade logs, giving insights into trading decisions and outcomes.