Deep Learning Model Implementation and Evaluation for Adjusted Returns Prediction

Your Name

December 1, 2024

Assignment Overview

The goal of this assignment is to implement and evaluate various deep learning models for predicting **adjusted returns** using market features such as **Signed Volume**, **Bid Fill**, **Ask Fill**, **Best Bid**, **Best Ask**, and **Price**. In particular, we are tasked with building linear and nonlinear models for price impact prediction, implementing a linear optimal strategy and its Sharpe ratio, and training a deep learning model to predict adjusted returns.

—

1 Task 1: Linear and Nonlinear Models for Price Impact Prediction

In financial markets, **price impact** refers to how a trade affects the price of an asset. The size of the trade (or **signed volume**) plays a crucial role in determining how much the market price moves in response to the trade. In this task, we aim to construct two models to predict price impact:

- **Linear OW Model**: Assumes that the price impact is linearly related to the signed volume.
- **Nonlinear AFS Model**: Assumes a nonlinear relationship where the price impact increases at a decreasing rate as the signed volume increases.

These models are inspired by the findings in the paper *Efficient Trading with Price Impact* by J. D. Farmer et al., which highlights the limitations of linear models and introduces more flexible nonlinear models. The research suggests that while linear models are simple and interpretable, they may not capture the complexities of real-world trading, where market frictions (such as liquidity) play a critical role.

1.1 Linear OW Model

The **Linear OW Model** assumes a direct proportionality between price impact and signed volume, which can be mathematically expressed as:

$$\text{Price Impact} = \lambda \times \text{Signed Volume}$$

where λ is a constant that determines the price impact factor. This model simplifies the trading dynamics but is effective in certain market conditions where liquidity is high and trades have a proportional impact on price.

```
# Linear OW Model: Price Impact = lambda * Signed volume
lambda_ow = 0.01  # Example value for lambda (price impact factor)
signed_volume = data['Signed Volume'].values

# Calculate price impact for the Linear OW model
price_impact_ow = lambda_ow * signed_volume
```

1.2 Nonlinear AFS Model

The **Nonlinear AFS Model** introduces a more realistic approach to modeling price impact, where the relationship between price impact and signed volume follows a power law. This nonlinear model is expressed as:

$$\text{Price Impact} = \lambda \times |\text{Signed Volume}|^{\beta}$$

where λ is a scaling constant, and β is an elasticity parameter that controls how rapidly the price impact increases with signed volume.

The research paper suggests that this nonlinear model better captures the decreasing effect of larger trades in more liquid markets. It reflects the market's diminishing returns as trade size increases, which aligns with empirical observations in financial markets.

```
# Nonlinear AFS Model: Price Impact = lambda * |Signed volume|^beta
lambda_afs = 0.01  # Scaling constant
beta = 0.5  # Elasticity parameter (nonlinearity factor)

# Calculate price impact for the Nonlinear AFS Model
price_impact_afs = lambda_afs * np.abs(signed_volume) ** beta
```

1.3 Handling Outliers

Both models require effective handling of outliers, as extreme values can disproportionately influence the model's predictions. Outliers can arise from various factors, including large market events or data errors. In this task, we handle outliers by removing values beyond the 99th percentile or capping them (Winsorization).

```
# Option 1: Remove extreme outliers (beyond the 99th percentile)
upper_limit = np.percentile(price_impact_ow, 99)
filtered_price_impact_ow = price_impact_ow[price_impact_ow <= upper_limit]
```

1.4 Visualizing Price Impact Distribution

We visualize the distribution of price impact for both the linear and nonlinear models using histograms. This step helps to understand the spread of price impact values and detect any potential anomalies in the data.

```
# Visualize the filtered price impact distribution for Linear OW Model
plt.hist(filtered_price_impact_ow, bins=50, alpha=0.75, color='blue')
plt.title('Linear OW Model - Price Impact Distribution (Filtered)')
plt.xlabel('Price Impact')
plt.ylabel('Frequency')
plt.show()
```

—

2 Task 2: Optimal Strategy with Linear Impact and Sharpe Ratio Visualization

In this task, we aim to implement an **optimal trading strategy** using the **Linear OW Model** and compute the **Sharpe ratio**. The Sharpe ratio is

a critical measure for evaluating the risk-adjusted performance of a trading strategy. A higher Sharpe ratio indicates that the strategy is generating higher returns for the level of risk taken.

2.1 Linear OW Model for Optimal Strategy

We use the **Linear OW Model** to predict price impact and adjust raw returns by subtracting the transaction costs (price impact). The adjusted returns reflect the actual returns after considering the cost of trading.

The strategy's performance is then evaluated by calculating the **expected return** and **risk** (standard deviation), which are used to compute the Sharpe ratio.

```
# Linear OW Model: Price Impact = lambda * Signed volume
lambda_ow = 0.01
signed_volume = np.random.uniform(1, 1000, 1000)  # Simulated trade size

# Price impact for each trade using Linear OW model
price_impact = lambda_ow * signed_volume
```

We then simulate the returns, adjust them for price impact, and calculate the expected return and risk:

$$\text{Expected Return} = \frac{1}{n}\sum_{i=1}^{n} R_i$$

where R_i represents the individual returns for each trade. The risk (or standard deviation) is calculated as:

$$\text{Risk} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(R_i - \text{Expected Return})^2}$$

2.2 Expected Return, Risk, and Sharpe Ratio

The **Sharpe ratio** is computed as the ratio of the expected return to the risk (standard deviation):

$$\text{Sharpe Ratio} = \frac{\text{Expected Return}}{\text{Risk}}$$

A higher Sharpe ratio indicates that the strategy is providing better returns for the level of risk.

```
# Simulate returns and calculate adjusted returns
returns = np.random.normal(0.001, 0.01, 1000)  # Simulated daily returns
signed_volume = np.random.uniform(1, 1000, 1000)  # Simulated signed volume
adjusted_returns = returns - (0.01 * signed_volume)  # Adjusted returns

# Calculate expected return and risk (standard deviation)
expected_return = np.mean(adjusted_returns)
risk = np.std(adjusted_returns)

# Calculate Sharpe ratio
sharpe_ratio = expected_return / risk
```

2.3 Visualization of Sharpe Ratio

We visualize the Sharpe ratio over time to track how the strategy performs. The Sharpe ratio helps identify periods where the strategy is underperforming or taking on excessive risk.

```
# Visualize the Sharpe Ratio over time
plt.plot(sharpe_ratio)
plt.title('Sharpe Ratio Over Time')
plt.xlabel('Time')
plt.ylabel('Sharpe Ratio')
plt.show()
```

—

3 Task 3: Deep Learning Model Implementation

In this task, we implement a **deep learning model** to predict **adjusted returns** based on market data such as **signed volume**, **bid and ask fills**, and other market features. Deep learning models are especially useful in capturing complex, non-linear relationships between features and target variables.

3.1 Neural Network Model

We use a simple feed-forward neural network architecture with two hidden layers. The model uses **ReLU activation** for the hidden layers and a single output neuron for predicting the adjusted returns.

```
# Define the neural network model
def create_model(input_dim):
    model
    model = Sequential()
    model.add(Dense(64, input_dim=input_dim, activation='relu'))  # First hidden l
    model.add(Dense(32, activation='relu'))  # Second hidden layer
    model.add(Dense(1))  # Output layer

    model.compile(optimizer='adam', loss='mean_squared_error')  # Adam optimizer a
    return model
```

The network architecture we used in this task is simple but effective for capturing the relationships between the market features and the adjusted returns. The use of **ReLU activation** functions ensures that the network can model complex nonlinear relationships between input features.

3.2 Training the Model

The model is trained using a set of market data features as input and adjusted returns as the target output. We use a batch size of 32 and train the model for 100 epochs to ensure convergence. The training process is monitored using the validation data to assess the model's generalization.

```
# Train the model
history = model.fit(X_train, y_train, epochs=100, batch_size=32, validation_data=(
```

Training the model involves adjusting the weights of the neural network in such a way that the loss function, **mean squared error (MSE)**, is minimized. The model is updated using the **Adam optimizer**, which adapts the learning rate for each parameter.

3.3 Model Evaluation

After training the model, we evaluate its performance using three key metrics: **Mean Squared Error (MSE)**, **R-squared**, and **Mean Absolute Error (MAE)**. These metrics provide insights into how well the model is predicting the adjusted returns.

- **MSE** measures the average squared difference between predicted and actual values. A lower MSE indicates a better model. - **R-squared** measures the proportion of variance in the target variable explained by the model. A higher R-squared value indicates better model fit. - **MAE** provides the average magnitude of the errors in the predictions, without considering their direction.

```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error

# Evaluate model performance
y_pred = model.predict(X_val)
mse = mean_squared_error(y_val, y_pred)
r2 = r2_score(y_val, y_pred)
mae = mean_absolute_error(y_val, y_pred)

print(f"MSE: {mse:.4f}, R-squared: {r2:.4f}, MAE: {mae:.4f}")
```

3.4 Model Comparison

To compare the performance of different models, we tested various configurations of neural network architectures, including models with and without dropout, models with L2 regularization, and models with a lower learning rate. Regularization techniques such as **dropout** and **L2 regularization** help prevent overfitting by limiting the complexity of the model.

```
# Model with Dropout Regularization
def create_model_with_dropout(input_dim):
    model = Sequential()
    model.add(Dense(64, input_dim=input_dim, activation='relu'))
    model.add(Dropout(0.5))  # Dropout to prevent overfitting
    model.add(Dense(32, activation='relu'))
    model.add(Dropout(0.5))  # Dropout to prevent overfitting
    model.add(Dense(1))
```

```
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model

# Model with L2 Regularization
from tensorflow.keras.regularizers import l2

def create_model_with_l2(input_dim):
    model = Sequential()
    model.add(Dense(64, input_dim=input_dim, activation='relu', kernel_regularizer
    model.add(Dense(32, activation='relu', kernel_regularizer=l2(0.01)))
    model.add(Dense(1))
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model
```

We compared the models based on the performance metrics and selected the best-performing model for deployment.

3.5 Loss Curves Visualization

To visualize the training process and check for overfitting, we plot the loss curves for each model. Overfitting is indicated by a large gap between the training and validation loss, suggesting that the model is fitting too closely to the training data.

```
# Plot training & validation loss for each model
plt.plot(history_basic.history['loss'], label='Basic Model Training Loss')
plt.plot(history_basic.history['val_loss'], label='Basic Model Validation Loss')
plt.title('Training and Validation Loss for Basic Model')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

—

4 Conclusion

This assignment involved implementing and evaluating multiple machine learning models for predicting **adjusted returns** in financial markets. Each

task presented unique challenges, and we explored several approaches to tackle them.

4.1 Task 1: Linear and Nonlinear Models for Price Impact

In Task 1, we successfully implemented two models for predicting price impact based on the signed volume:

- The **Linear OW Model** assumes a direct proportional relationship between price impact and signed volume. This model is simple but effective in scenarios where liquidity is high and price impact is expected to be linear.
- The **Nonlinear AFS Model** assumes a more complex relationship, where price impact increases at a diminishing rate as signed volume increases. This model is more representative of real-world market behavior, particularly in less liquid markets.

Through this task, we demonstrated how different models can be used to capture market dynamics and how the nonlinear model better aligns with observed market phenomena.

4.2 Task 2: Optimal Strategy with Linear Impact and Sharpe Ratio Visualization

In Task 2, we developed a **linear optimal strategy** and computed the **Sharpe ratio** to evaluate the risk-adjusted returns of the strategy. By adjusting raw returns for price impact, we obtained a more realistic estimate of the strategy's performance. - The results reinforced the importance of considering transaction costs when developing trading strategies. The Sharpe ratio served as a key metric for assessing how well the strategy balances return and risk.

4.3 Task 3: Deep Learning Model Implementation

In Task 3, we implemented a **deep learning model** to predict adjusted returns based on market features. The model was evaluated using **MSE**,

R-squared, and **MAE**. - We experimented with different architectures, including models with dropout and L2 regularization, and observed that regularization techniques helped improve the model's generalization capabilities. - The deep learning model demonstrated strong predictive performance, but further tuning and experimentation could lead to even better results.

4.4 Model Comparison and Final Evaluation

After training and evaluating each model, we compared them based on key performance metrics. The results showed that models incorporating **dropout** and **L2 regularization** performed better in terms of **MSE**, **R-squared**, and **MAE**, indicating better generalization to unseen data.

4.5 Future Work

There are several directions for future work:

- **Hyperparameter Tuning**: Further optimization of the model's hyperparameters (e.g., learning rate, batch size, number of epochs) could lead to better performance.
- **Advanced Models**: Exploring more advanced architectures, such as **LSTM** or **Transformer models**, could enhance the model's ability to capture sequential dependencies in the data.
- **Cross-validation**: Implementing **cross-validation** would help assess the model's performance on different subsets of the data and improve generalization.
- **Risk-adjusted Return Optimization**: Future work could focus on directly optimizing the trading strategy to maximize the Sharpe ratio, combining both the market prediction model and portfolio optimization techniques.

—