Forecasting Methods in Fintech 096292

Project Summary

Subject: Predict GBP/JPY

Shalev Hermon 208159400
Vladislav Comantany 312905789
Raz Biton 315507780
Uriah Asulin 322691819

Introduction:

We aimed to predict the GBP/JPY closing rate.

- Frequency: daily
- Forecasting horizon: a day ahead, the model is tested over 80 days.
- Data range: start date: 1.10.2018 end date: 31.12.2023

Features (will explain in detailed later on how we pre-processed them):

- Volume.
- Open, low, high, close prices.
- RSI, Weighted Moving Average, Bollinger Bands.
- Capital flows – Nissa, Honda, Barclays, Mitsubishi Motors, Sony.
- "TONA" (Japan Tokyo Overnight Average Rate).
- "SONIA" (Sterling Overnight Index Average).
- Currencies Rates that in correlated with our rate: USD/ JPY, EUR/JPY AUD/JPY
- Crude oil prices (in USD).

Data Sources:

Apart from TONA/SONIA which were taken from the official websites of the banks, all data was taken from the website: https://www.dukascopy.com/ which is a Swiss banking group (broker).

The Statistical Evaluation Measures:

We evaluated the model using several common metrics for regression tasks, including:

- **Mean Absolute Error (MAE)**: The average difference between predicted and actual values.

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

- **Root Mean Square Error (RMSE)**: A more punishing metric for larger errors.

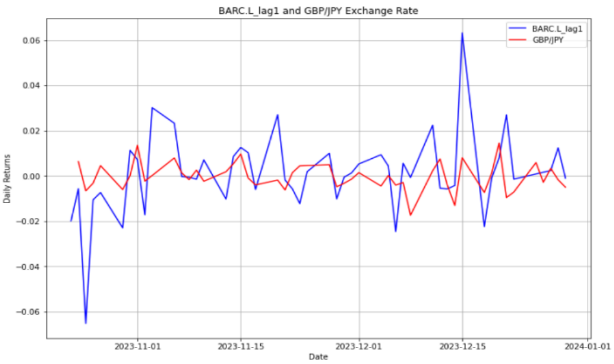
$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

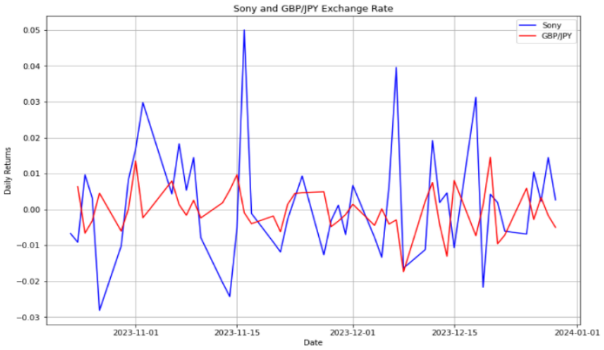
Selection of Capital Flows Based on Stock Market Correlations

***The correlation checks were performed in the notebook titled "Find Correlation between stocks from Japan and England to our rate.ipynb." that we added to the submission.**

In the process of selecting capital flows to inform our model for predicting the GBP/JPY exchange rate, we began by compiling a list of significant stocks from Japan's Nikkei and the UK market. The Nikkei 225 index is a prominent stock market index that includes major Japanese companies, while the FTSE 100 serves as the equivalent in the UK, comprising the largest companies listed on the London Stock Exchange. The initial stock selections from Japan included major corporations such as Toyota, Sony, Mitsubishi UFJ, SoftBank, Honda, Nissan, Canon, Panasonic, Hitachi, NTT, KDDI, Mitsui, Sumitomo Mitsui, Keyence, Takeda, Japan Tobacco, Shin-Etsu, Daiichi Sankyo, Tokyo Electron, NTT DOCOMO, Nomura, Mitsubishi Corp, Itochu, and Mitsubishi Electric. Similarly, we included prominent UK stocks, such as Unilever, BP, HSBC, GSK, AstraZeneca, Diageo, Barclays, Vodafone, British American Tobacco, Tesco, Lloyds, National Grid, Rio Tinto, Rolls-Royce, Prudential, Royal Dutch Shell, Reckitt Benckiser, Experian, BT Group, Aviva, Anglo American, and Compass Group.

To determine the most relevant stocks for our analysis, we conducted a correlation analysis between the selected stocks and the GBP/JPY exchange rate. This involved examining the correlations not only between current values but also considering lagged values by shifting the explanatory stock data back by several days. This step allowed us to identify potential leading indicators that could influence the exchange rate. After conducting this analysis, we ultimately selected the following stocks based on their high correlation with the GBP/JPY exchange rate: Nissan, Honda, Barclays, Mitsubishi Motors, and Sony. This selection is expected to provide valuable insights into capital flows that can impact the exchange rate dynamics in our predictive model. In the attached notebook, ***"Find Correlations.ipynb"***, we performed a comprehensive analysis focused on exchange rate differences and lagged correlations between explanatory features and the target variable (exchange rate)






```
Correlations with GBP/JPY exchange rate (lagged 1 day):
ULVR.L_lag1      -0.070519
AZN.L_lag1       -0.059297
GSK.L_lag1       -0.052748
RIO.L_lag1       -0.027668
BATS.L_lag1      -0.026050
                     ...
6501.T_lag1       0.300650
7201.T_lag1       0.306040
BARC.L_lag2       0.308880
7267.T_lag1       0.312106
Adj Close_lag1    1.000000
Name: Adj Close_lag1, Length: 90, dtype: float64

Correlations with GBP/JPY exchange rate (lagged 2 days):
ULVR.L_lag2      -0.070385
AZN.L_lag2       -0.059230
GSK.L_lag2       -0.052591
RIO.L_lag1       -0.035055
8604.T_lag1      -0.031256
                     ...
8031.T_lag2       0.288718
6501.T_lag2       0.300610
7201.T_lag2       0.306027
7267.T_lag2       0.312083
Adj Close_lag2    1.000000
```

Data Pre-processing and Model Evaluation

After obtaining all the raw data, we identified the dates where all features were available across the dataset. To handle missing data, we applied a forward-fill method to fill missing values, ensuring that no gaps existed in the feature columns. Additionally, we dropped any rows containing NaN values that couldn't be filled. Once the dataset was cleaned, all features were concatenated into a single large dataframe to create a unified dataset for model training.

The data preprocessing involved the prepare_sequences function, which structured the dataset into sequences based on the parameter sequence_length = 50. This parameter defines the number of timesteps to consider from the historical data, providing the LSTM model with relevant context for learning temporal patterns. Each input sequence consisted of the past 50 time-steps of features, with the target being the next value following this sequence. This method ensures that the model learns from past data without introducing any future information, effectively preventing data leakage.

In the prepare_sequences function, the dataset is iterated over, and for each position in the dataset (up to the length of the dataset minus the sequence length), a new input-output pair is created. The input is a sequence of feature values taken from the historical data, and the output is the subsequent target value that the model is trying to predict. This sequential structuring ensures that predictions on both the validation and test sets are made based solely on the training dataset, thereby preserving the chronological order of data and eliminating any risk of using future information during training.

Techniques We Used to Preventing Overfitting

To evaluate the model's performance, we implemented a careful split of the dataset into training, validation, and test sets while preserving the chronological order of the data. Various techniques were employed to prevent overfitting during model training:

1. **Early Stopping:** This technique involves monitoring the validation loss during training and halting the training process when the validation loss starts to increase, indicating that the model may be beginning to memorize the training data rather than learning to generalize. This helps maintain a balance between fitting the training data and retaining the ability to generalize to unseen data.
2. **Dropout Layers:** Dropout is a regularization technique used in neural networks to randomly set a fraction of the input units to zero at each update during training time, which helps to prevent overfitting. By reducing the reliance on specific neurons, dropout promotes the model's ability to learn more robust features that generalize better to new data.
3. **Restoring Best Weights:** This technique involves saving the model's weights at the epoch with the best validation loss, ensuring that the final model used for predictions is the one that performed best on the validation data. This strategy mitigates the risk of using weights from later epochs where overfitting may have begun.

Predictions Based Solely on Training Data (Prevent Data Leakage)

The model's ability to make sequential predictions on the test set was enhanced by iteratively using prior predictions to form new input sequences. The predictions for the test set were solely based on the data from the training set; after training the model on the training dataset, we initialized the prediction process by creating an input sequence from the last available historical data points from the training set.

During the prediction process, as the model generates predictions for the test set, it continuously appends the latest predicted value to the sequence, replacing the oldest value in the input sequence. This approach maintains the integrity of the data used for prediction by ensuring that only previously known information is utilized at each step.

Data Splitting for Time Series Forecasting

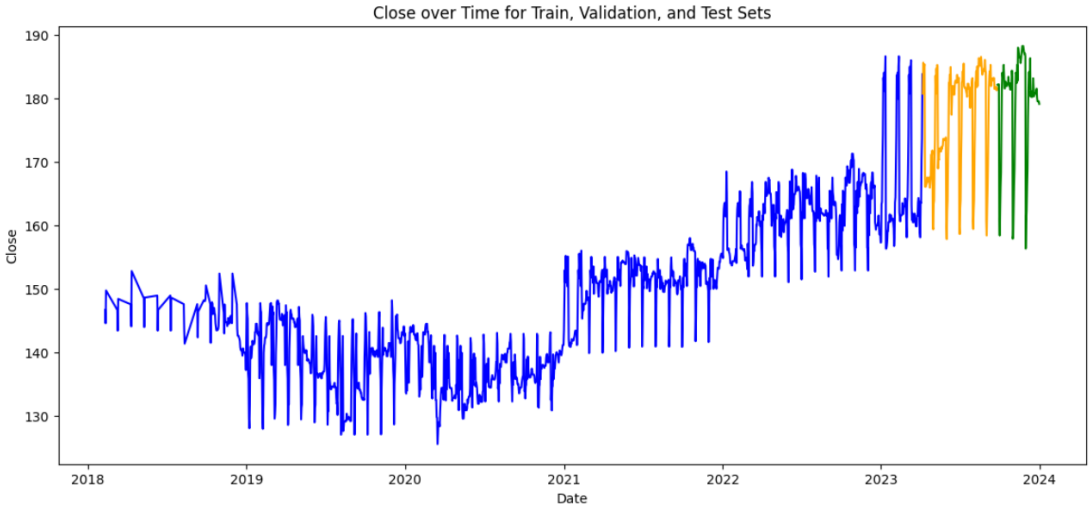
When splitting a time series dataset into training, validation, and test sets, it's essential to maintain the temporal order of the data while ensuring that the statistical properties (mean and variance) across the sets are aligned. This approach helps us create a model that generalizes well and avoids biases in evaluation. Here's a detailed explanation of how we did it:

1. Temporal Integrity

- **Sequential Split**: Unlike random splitting in other machine learning tasks, time series data must be split in a sequential manner. This means the training set will consist of the earliest observations, followed by the validation set, and finally the test set. This preserves the chronological order and prevents data leakage, ensuring that future data points do not influence the model training.

2. Statistical Property Alignment

- **Visual Inspection**: Before splitting, we visualized the time series data to understand its trends, seasonality, and fluctuations. Look for any patterns in the mean and variance over time.
  The plot_close function was used to generate this visualization, helping to illustrate how the data is segmented for model training and evaluation.



In addition, we created a custom function called get_statistics to plot and analyze the statistical information of each set. This allowed us to verify that the fluctuations and statistical characteristics, such as mean, standard deviation, and other descriptive metrics,were approximately similar across the training, validation, and test sets. By ensuring that the statistical properties of the datasets were aligned, we aimed to prevent any potential biases or imbalances during model training and evaluation:

```
Statistics for Training DataFrame:
Mean: 147.79614693118415
Variance: 123.60981547157374
Standard Deviation: 11.11799511924
Minimum: 125.601
Maximum: 186.568
25th Percentile: 138.895
Median: 146.106
75th Percentile: 155.717
```

```
Statistics for Validation DataFrame:
Mean: 172.18475555555557
Variance: 80.55426927562186
Standard Deviation: 8.975203021415274
Minimum: 157.832
Maximum: 185.944
25th Percentile: 163.7335
Median: 171.609
75th Percentile: 181.8775
```

```
Statistics for Test DataFrame:
Mean: 179.9333468208093
Variance: 56.28801312320205
Standard Deviation: 7.50253378
Minimum: 156.34
Maximum: 188.192
25th Percentile: 180.276
Median: 181.888
75th Percentile: 184.148
```

Data Normalization

In our data pre-processing, we employed standard scaling to normalize the features of our dataset.

- we used standard scaling, also known as Z-score normalization, to transform the data such that it has a mean of zero and a standard deviation of one. This is achieved by subtracting the mean of the dataset from each data point and then dividing by the standard deviation. Mathematically:

$$X_{\text{scaled}} = \frac{X - \mu}{\sigma}$$

For each column of the features, where X is the original value, μ is the mean of the column, and σ is the standard deviation.

We applied standard scaling solely to the training set to prevent overfitting. This means that the mean and standard deviation were computed only from the training data, ensuring that the model does not have access to information from the validation or test sets during training. By standardizing fitted only the training data, we avoid any potential data leakage that could bias the model's performance metrics.

Final Architectures and Results:

Architectures

We created and experimented with over 650 different architectures of the LSTM model, testing a wide variety of hyperparameter combinations to explore the best configurations. Below are the details of the hyperparameter combinations we explored:

1. **Hidden Layer Architectures**:
    - We tested a wide range of hidden layer configurations, from simple architectures to more complex, deeper networks:
        - **[128, 64]**: A balanced, two-layer architecture, serving as a strong starting point.

        - **[256, 128]**: A larger configuration, capable of capturing more complex patterns.
        - **[128]**: A simple, one-layer model used for baseline comparisons.
        - **[512, 256]**: A high-capacity network for more complex data relationships.
        - **[256, 128, 64]**: A three-layer architecture for learning intricate interactions in the data.
        - **[256, 256]**: Two large layers for deeper learning and increased capacity.
        - **[128, 64, 32, 16]**: A progressively smaller, deeper network to capture finer details.
        - **[128, 128]**: Two equal-sized layers providing a balanced capacity for feature learning.
        - **[512, 256, 128]**: A deeper model to explore more complex feature learning.
        - **[128, 64, 64]**: A wider model with an additional layer to enhance feature learning.
        - **[256, 128, 64, 32]**: A complex model with varying layer sizes to capture diverse structures in the data.
2. **Dense Layer Units (stack on the LSTM Layers)**:
    - We tested different sizes for the fully connected (dense) layers:
        - **[128, 64]**: A balanced two-layer setup, ideal for a variety of data patterns.
        - **[64, 32, 16]**: Smaller layers to explore the performance of basic configurations.
        - **[256, 128]**: Larger dense layers to explore the impact of more parameters.
        - **[64, 32]**: Smaller layers, helping us test if larger models overfit.
        - **[128, 128]**: Two equal-sized layers, offering a balanced feature learning capacity.
        - **[64, 64, 32]**: A three-layer structure for capturing interactions between different feature sets.
        - **[256, 128, 64]**: Larger dense layers to explore the impact of higher-capacity layers on model performance.
        - **[128, 32, 16]**: A simpler setup with progressively decreasing sizes.
        - **[128, 64, 32, 16]**: A deeper architecture to examine how additional layers affect performance.
        - **[128, 64, 64]**: A wider model with an additional layer for enhanced learning.

        - **[128, 32]**: A basic two-layer configuration, used as a reference point for comparison.

3. **Dropout Rate**: We experimented with different dropout rates to address overfitting and improve generalization:
    - **0.1, 0.2, 0.25**: A moderate levels of regularization, used in most experiments.
    - **0.3** and **0.5**: Higher dropout rates, particularly useful when testing large models to prevent overfitting.
4. **Learning Rates**:
    - We tested several learning rates to control the speed of model training and convergence:
        - **1e-3**: A common learning rate that worked well in most configurations.
        - **5e-4**: A slightly lower learning rate, offering a balance between training speed and accuracy.
        - **1e-5**: A very small learning rate, tested to observe its impact on model refinement.
5. **Batch Sizes**: We experimented with different batch sizes to explore how they influenced training dynamics:
    - **4 and 8**: Smaller batch sizes were tested to observe if they could capture finer details and lead to better learning.
    - **16 and 32**: Mid-sized batch sizes that were most commonly used, striking a balance between training speed and model convergence.
    - **64**: Larger batch sizes were tested to explore their impact on training speed and potential generalization improvements.
6. **Epochs**:
    - We tested models with **100** and **200** epochs to ensure adequate training time for convergence while preventing overfitting with patience and early stop.

Results

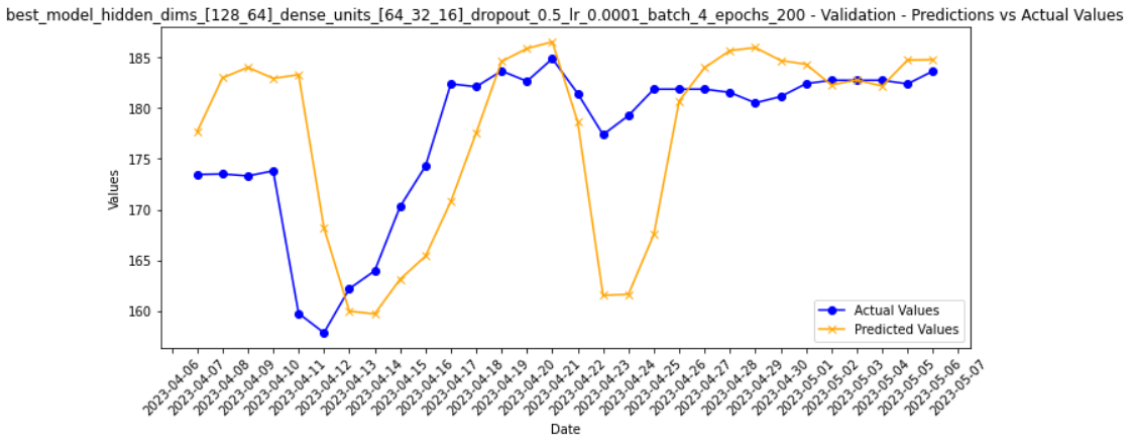
After testing various combinations of the above hyperparameters, the most successful configuration was:

- **Hidden Dimensions**: [128,64]
- **Dense Units**: [64,32,16]
- **Dropout Rate**: 0.5
- **Learning Rate**: 0.0001
- **Batch Size**: 4
- **Epochs**: 200

The final model was able to capture the trends of the actual values, though there were some challenges in fully capturing all variations in the data (as reflected in the

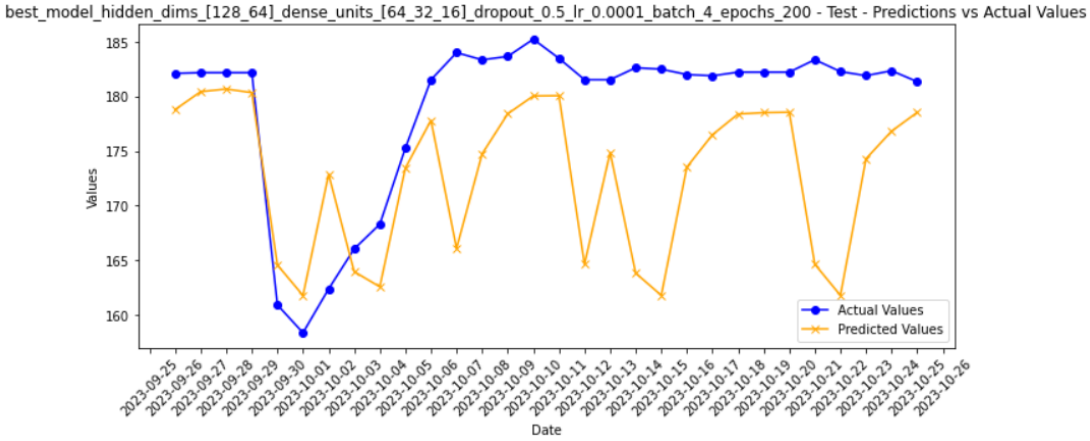
gap between actual and predicted values). However, with careful tuning, we significantly improved the model's ability to generalize.

Prediction on Validation set:



Validation MSE: 96.713, Validation RMSE: 9.834, Validation MAE: 6.678

Prediction on Test set:



Test MSE: 137.6700, Test RMSE: 11.7332, Test MAE: 9.4284

In comparison to other, the less effective architectures, the results for those models were as follows:

- Mean of MAE on the validation set: 9.78
- Mean RMSE on the validation set: 9.35
- Mean of MAE on the test set: 30.94
- Mean of the RMSE on the test set: 31.11

For the less optimal models, the error metrics were higher, reflecting the differences in performance between the various architectures. While we acknowledge that the results may not have achieved perfect accuracy or the absolute best performance, we dedicated considerable effort to fine-tuning the model and optimizing its

performance to the best of our ability.
Our goal was to strike a balance between model complexity and generalization. Though there is certainly room for further improvement, the outcomes represent a strong foundation, especially considering the constraints of computational power available to us.

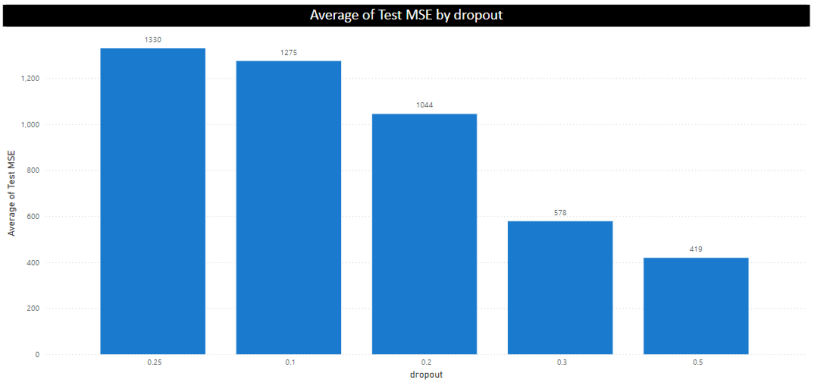
Architectures Comparison

We developed Python code to analyze and compare the different architectures. The analysis is documented in a Jupyter notebook titled **"Data Analysis - Comparison Architectures.ipynb"**. Additionally, we used Power BI to enhance the clarity of the visualizations, providing a more comprehensive view of the results. These files allow for deeper insights into the model performance comparisons and can be easily accessed for further exploration.

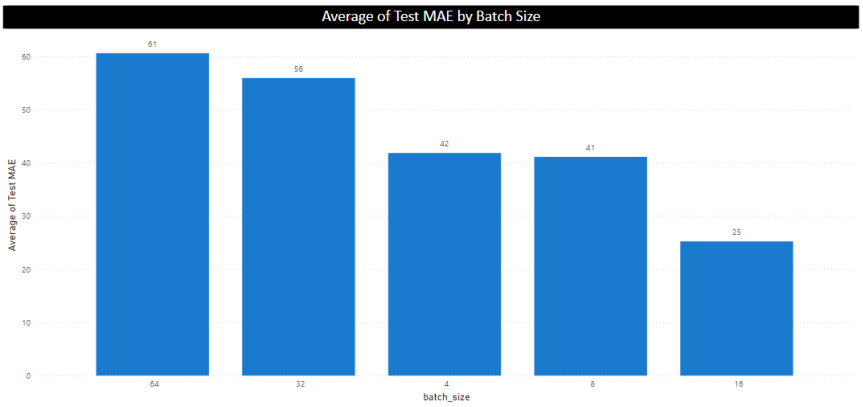
Summary of Model Architecture Comparison

In the final stage of our analysis, we conducted a thorough comparison of various architectures, focusing on key hyperparameters such as dropout rate, batch size, and learning rate. The performance of each configuration was evaluated using Mean Squared Error (MSE) and Mean Absolute Error (MAE) on the test set. Here's a summary of the findings:

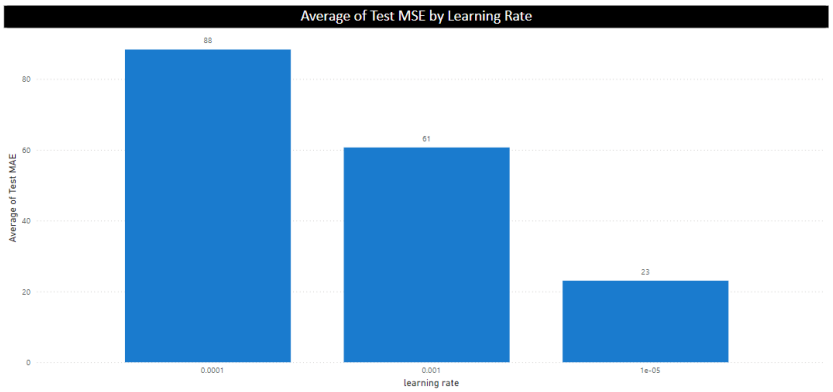
- **Best Dropout Rate**: The dropout rate of **0.5** emerged as the most effective, yielding the lowest MSE on the test set. This suggests that a moderate level of regularization helped prevent overfitting while retaining sufficient learning capacity.



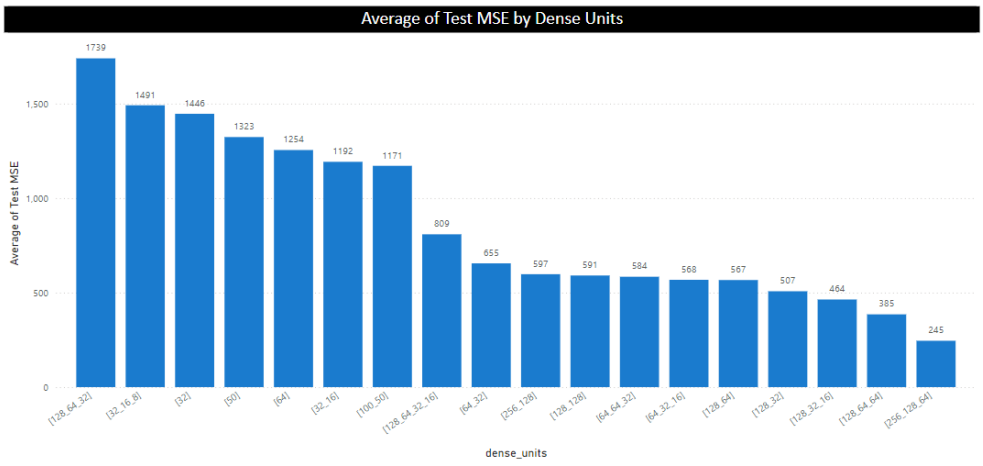
- **Best Batch Size**: A batch size of **16** produced the best results when compared by MAE on the test set. The smaller batch size likely contributed to better generalization, allowing the model to make more accurate predictions on unseen data.



- **Learning Rate**: The optimal learning rate was found to be **0.00001** when compared by MSE on the test set. Given that the dataset was relatively small (2000 samples), the lower learning rate allowed the model to converge more steadily, avoiding overshooting and helping it to fine-tune the parameters effectively.



- **Dense Units**: In addition to the hyperparameter comparison, a bar chart was plotted to show the impact of different dense unit configurations on the MSE of the test set. This analysis provided insight into how varying the number of dense units influences model performance, allowing us to identify the optimal architecture for this specific dataset.



The findings from this analysis clearly demonstrate the importance of carefully tuning hyperparameters, such as dropout rate, batch size, and learning rate, in optimizing the performance of LSTM architectures, particularly when working with smaller datasets. Each of these hyperparameters has a significant impact on how well the model generalizes and learns from the data.
These insights offer a clear takeaway: Hyperparameter tuning is not a one-size-fits-all process. The combination of dropout, batch size, and learning rate must be carefully balanced to achieve the best possible performance for a given dataset. This tuning process, while time-consuming, provides a foundation for building more effective and robust neural networks, particularly when applied to datasets with limited samples.