

Implementation of a model for predicting stock prices using neural network and deep learning

Luka Abramović

Faculty of Informatics, Juraj Dobrila University of Pula

17. June, 2024

This project will predict the stock prices of the company Johnson & Johnson (JNJ). The prediction was performed by a Long Short-Term Memory (LSTM) neural network. The data that was used for the model was downloaded from Kaggle. The model was trained on scaled data with MinMaxScaler. The model's architecture consists of two LSTM layers, other regularization layers, and a Dense layer. The model has been trained with the Adam optimizer and mean squared error (MSE) loss function. In order to evaluate the model's performance the predicted prices were compared with the actual stock prices. Achieved results were satisfactory with accurate predictions. Results were visualized with a graphical representation, as well as future prices for the next 30 days outside the collected data. Overall the model demonstrated good effectiveness in predicting stock prices and achieving success in this task.

Content

1. Introduction	4
2. Existing models	5
3. Dataset	6
4. Methodology.....	9
5. Model training	10
5.1 Final model	31
6. Conclusion	35
7. Pictures	36
8. References	37

1. Introduction

In today's world, the prediction of prices has become an incredibly important area. Predicting prices, such as stock prices can provide an advantage to numerous investors which will allow them to make better decisions to maximize their income. The idea behind this project is to create a model that is capable of predicting future stock prices of Johnson & Johnson by using their historical data. Stock price prediction is a very complex issue because of the financial markets. They are impacted by various factors such as companies events, and influence on stock prices. The dataset that was used for the project was downloaded from Kaggle, which included historical stock prices of Johnson & Johnson which consisted of open, high, low, and close prices and trading volume. By implementing the LSTM model on the data, the aim was to make a model which can predict the prices compared to the real ones, and give value to financial analysts. One of my motivations for choosing this model is the potential use and development of the model which can be applied. This project's methods can be applied and adapted to predict prices of other stock prices or even cryptocurrencies. Also, the skill and knowledge gained from creating this project will help in goal of becoming a data analyst, by allowing to more thoroughly understand the control of data and its evaluation. The paper is organized as follows. In chapter 2, it is provided an overview of already existing solutions and other LSTM models that can be used for stock price prediction. Chapter 3 will describe the methodology used to solve the problem and explain the steps that were used to solve it. Chapter 4 will present all the model training iterations, the steps used to improve it, and the final and best model that will be used. For the 6th and last chapter conclusion will be made about the whole project.

2. Existing models

In the search of existing models, multiple public projects appeared that have implemented the LSTM model to predict future stock prices. In this chapter, the overview of similar projects and their methods will be presented and explained. The first project can be found on Kaggle by BryanB, (<https://www.kaggle.com/code/bryanb/stock-prices-forecasting-with-lstm>), the author in his project used historical stock price data to predict future prices with LSTM model. The model is structured with multiple layers, as well as LSTM layers, Dense layers, and Adam optimizer. The author demonstrated how the use of hyperparameters such as learning rate or batch size can help achieve significant accuracy in predictions. Another project that was also found on Kaggle (<https://www.kaggle.com/code/pablocastilla/predict-stock-prices-with-lstm>), was created by Pablo Castilla who implemented a similar approach. The author in his project used the dataset which had historical stock prices. The model was fine-tuned by hyperparameters such as epochs, batch size, and learning rate as well as using a validation set. The author also used techniques such as early stopping and learning rate reduction to prevent overfitting. Finally, the last project was created by Dpam Gautam, which can also be found on Kaggle (<https://www.kaggle.com/code/dpamgautam/stock-price-prediction-lstm-gru-rnn>). The author of this project analyzed by using LSTM, GRU, and RNN to predict stock prices. A dataset was used which provided historical stock data. The model architecture was made out of LSTM, GRU, and RNN, where LSTM was used to capture long-term dependencies, GRU was used as an alternative to LSTM with a more simple architecture, and RNN as a basic method for comparison.

All of these projects are effective and show the capability of the LSTM network to predict stock prices. These models have shown great potential to predict stock prices by implementing numerous strategies such as model architecture or changes of hyperparameters.

3. Dataset

The dataset which will be used for price prediction is stock prices from the company Johnson & Johnson. Dataset was obtained from the Kaggle website, where all time data of stock prices is displayed in excel table (<https://www.kaggle.com/datasets/dhruvshan/johnson-and-johnson-stock-data-all-time>).

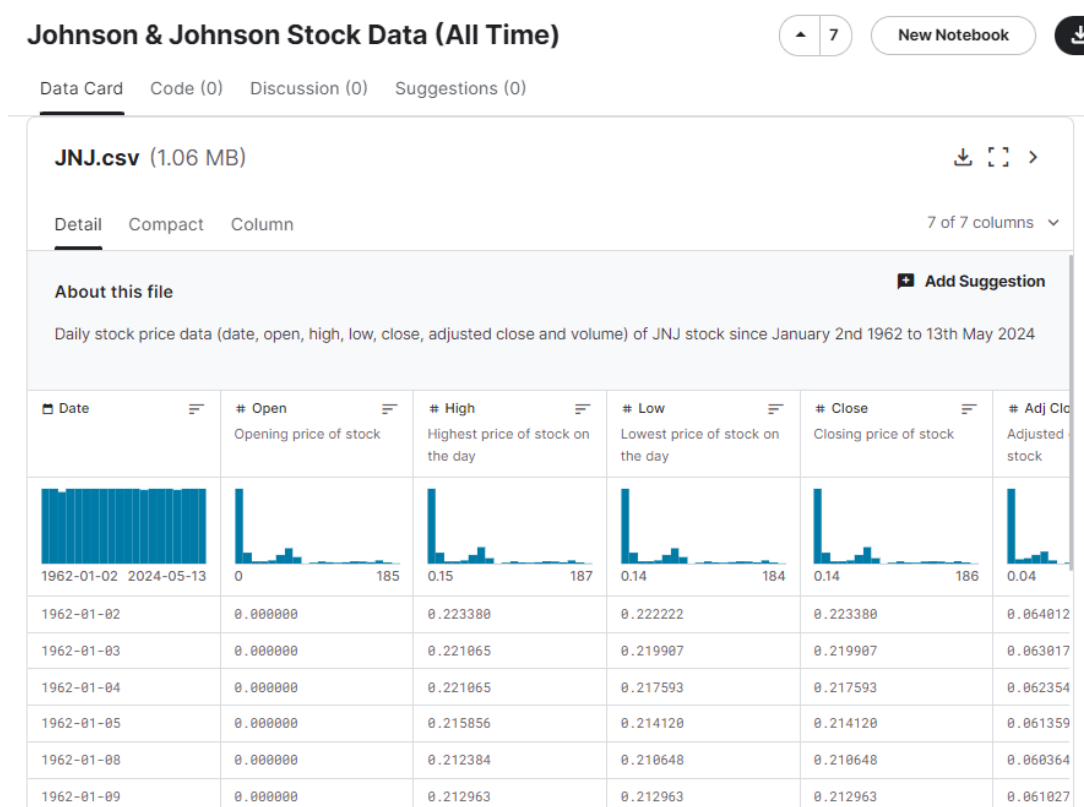


Figure 1 JNJ - Stock prices data

Dataset is composed out of features, where each feature consists out of certain price of the stock during the day.

Open – represents the price which the stock had at the start of the day

High – represents the highest price which the stock had during the day

Low – represents the lowest price which the stock had during the day

Close – represents the price which stock had at the end of the day

Volume – which represents the volume of which stock had traded during the day.

For this model, and our purpose, we will be using the “Close” feature, which represents the price of the stock at the end of the day. These are the basic features of the dataset, however in addition to these, technical indicators were added which were able to help improving the prediction.

SMA (Simple moving average) – This is an average price which is calculated over a certain period, in this case for this model we will use period of 20 days.

EMA (Exponential Moving Average) – This is similar to the SMA, which will also be calculated over the period of 20 days, however, this indicator is more capable of answering faster to the price changes.

RSI (Relative Strength Index) – This indicator shows the strength index over time, where it will be calculated over 14 days. It is very helpful because it allows us to track how fast or slow the price moves.

OBV (On-Balance Volume) – It uses the volume in trading which can help to indicate when it is the buying or selling pressure of the stocks.

After the data has been imported, it has been preprocessed. All the missing values, if there were any, were removed to ensure effectiveness, and the data column has been adjusted. Because all the features were different, MinMaxScaler was implemented to normalize all data in the range of 0 to 1. This ensured that all the features during the training had the same contribution to the model, which in the end allowed him better performance. The data has been made into sequences; each sequence is made out of 60 days. That means that for the closing price on day 61, it had to be calculated from the past 60 days, which are 1 to 60, and for the day 62 it has to be calculated from days 2 to 61, etc. That sequence allows the model to learn patterns more effectively over time. For that sequence, only the closing price will be observed because it is the price after the end of the day and gives us the most valuable information. The data has been split into a training set, a validation set inside the training set and a testing set. When validation_split was used for “fit” in the Keras library, it immediately made part of the training set into validation, which means that out of 80% which is designed for training, 10% of that will be used for validation. For the training, 80% of the data was used, while for the test the remaining 20%. This was applied to all iterations except the first one which was just a basic introduction, and the

final iteration which used the K-Fold cross-validation. K-fold cross-validation in the final iteration was used to split the set into K subsets, in this case 5 of them. That means that the model was trained K(5) times, and each time from the dataset different data was used. Each time it was trained on a different fold, the average loss was calculated from all 5 iterations to get the final validation loss. After it was trained, the model predicted the stock prices on the test set, where the predicted prices were compared with the actual ones to see the model's accuracy.

4. Methodology

The aim of this project is to create a model which can predict stock prices of company called Johnson & Johnson (JNJ) by using their historical data from stock prices. The problem will be solved with LSTM neural network which is especially effective for time-series forecast. That is because of its ability to learn and remember long-term dependencies. The first step in solving this problem is by collecting and loading the data. The data has been downloaded from Kaggle and includes historical stock prices of Johnson & Johnson. In order for data to be more easily controlled it has been loaded into Pandas DataFrame. Data has been preprocessed as well, such as converting the “date” column to datetime format, and scaling the values between 0 and 1 so it helps in the training process. Data has been split into 2 sets, which are training and test sets, while the validation set is inside the training. They are split into 80% for training, 10% for validation inside the training, and 20% for testing. The first set is for the model to be trained while the testing set will be used to test its performance. The validation set is inside the training set, which means it will use 10% out of 80% of the data for validation. In the final model, the architecture is made out of LSTM layers, Dropout layers, Batch normalization, and Dense layer. There are two LSTM layers, while dropout layers have been added to prevent overfitting, batch normalization to improve stability and the speed of training, and a dense layer to output the predicted closing price. The model is compiled with Adam optimizer, and the loss function (MSE) will measure the difference that is between the predicted and actual stock prices. Early stopping and learning rates are implemented to overfitting can be prevented and to ensure optimal training. Unlike the previous iterations, the final iteration wasn’t done on 80%, and 20% split but rather on K-fold cross-validation. All of the data has been split into 5 folds and randomized. After the model had been trained the stock prices were predicted using the test set.

The reason LSTM has been chosen for this project is because it is effective in handling time-series data perfectly. Compared to other networks, LSTM is amazing at finding patterns in stock prices. By following that approach, this project developed an accurate model to predict Johnson & Johnson stock prices.

5. Model training

In this chapter, the training process will be explained in detail of the LSTM model project designed to predict stock prices. Training iterations will be explained along with hyperparameters that were chosen and the reason behind those choices. After every iteration results will be shown that were achieved by using that iteration. Each iteration will represent an attempt to improve the model even further starting from 1st iteration until the 6th.

1st Iteration

After the dataset has been imported, the needed features were selected. Which are “Open”, “High”, “Low”, “Close”, 'Volume’. All features were scaled from 0 to 1. The next step involved creating a sequence. In the sequence function feature “Close” was taken to create it. The function sequence allows the model to use the historic data prices to calculate new ones, this sequence does that over the period of 60 days, which means it will use days 1 to 60 for day 61, then days 2 to 61 for days 62, and so on.

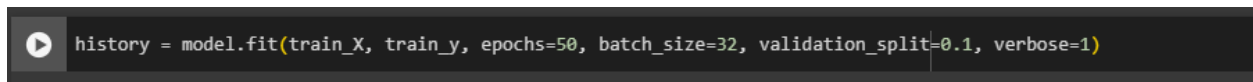
Because this is a first and simple iteration, just to see where we stand with the model, the training data wasn't split like in all other iterations (80%, 20%), but it was split into all data except the last 60 days. This means the testing was included in only the last 60 days and not before, like on the iterations that will follow.

```
[ ] model = Sequential([
    LSTM(100, return_sequences=True, input_shape=(X.shape[1], X.shape[2])),
    Dropout(0.3),
    LSTM(100, return_sequences=True),
    Dropout(0.3),
    LSTM(100),
    Dropout(0.3),
    Dense(1)
])

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

Figure 2 Model – 1

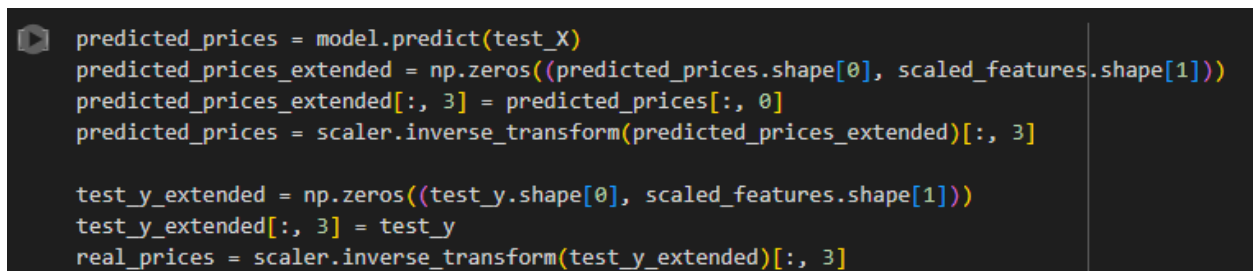
The decision to use three LSTM layers, was so temporal dependencies in stock price data could be captured because it allows them to learn different aspects of the time pattern. Units of 100 have been chosen so there is enough capacity for the model to learn from the data. Dropout could be categorized as high, but it was chosen to prevent overfitting and the MSE loss function which task is to measure the difference between the predicted and actual prices.

A code snippet in a dark-themed editor showing the training history command: `history = model.fit(train_X, train_y, epochs=50, batch_size=32, validation_split=0.1, verbose=1)`.

```
history = model.fit(train_X, train_y, epochs=50, batch_size=32, validation_split=0.1, verbose=1)
```

Figure 3 Training 1 – History

Epochs of 50 were chosen so the model can have enough opportunity to learn from the data without causing it to overfits. The batch size was 32 so it gives a good balance between memory efficiency and performance.

A code snippet in a dark-themed editor showing the testing process. It includes commands to predict prices, extend the array with zeros, and then use the scaler to inverse transform the predicted prices and test y values to get real prices.

```
predicted_prices = model.predict(test_X)
predicted_prices_extended = np.zeros((predicted_prices.shape[0], scaled_features.shape[1]))
predicted_prices_extended[:, 3] = predicted_prices[:, 0]
predicted_prices = scaler.inverse_transform(predicted_prices_extended)[:, 3]

test_y_extended = np.zeros((test_y.shape[0], scaled_features.shape[1]))
test_y_extended[:, 3] = test_y
real_prices = scaler.inverse_transform(test_y_extended)[:, 3]
```

Figure 4 Testing

This uses the trained model for predicting stock prices based on the data used for test. First there is row of 0 being created where it has the same number of rows from “predicted_prices” and the same number of columns from “scaled_features” and all elements are initialized to 0. After that, all the prices are put into a column that corresponds with the “Close” feature, a feature that is in the original dataset. Following that, the inversion transformation is made to return the values to the original scale. A similar process is also applied to “test_y”. This process is a key part with allows the testing of the model.

During all future iterations, this testing will be applied to every iteration and will not change.

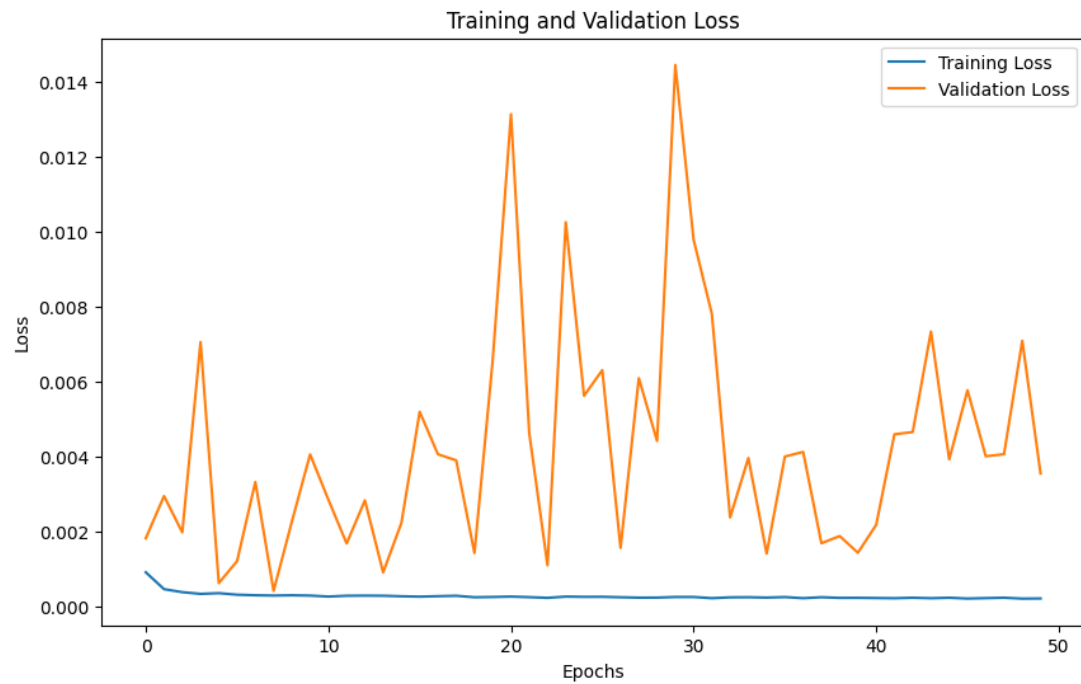


Figure 5 Training and Validation – 1

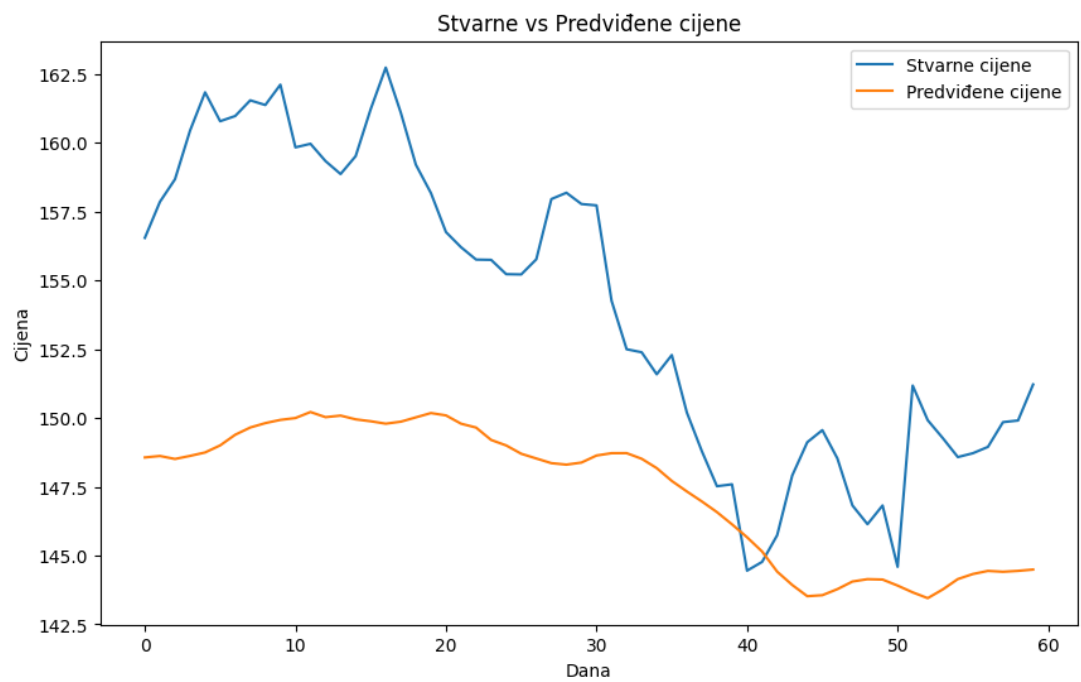


Figure 6 Real vs Predicted – 1

The initial results were terrible. Training loss significantly lower than the validation loss and not going up at all which gave a sign it was overfitting. The prediction accuracy was struggling as well, the predicted prices were not accurate at all with the actual ones.

2nd Iteration

For the second iteration, loading the dataset and preprocessing remained the same. The sequence function which calculates the price based on the history 60 days remained the same as well, however the data split has changed.

```
def create_sequences(data, sequence_length=60):  
    X, y = [], []  
    for i in range(len(data) - sequence_length):  
        X.append(data[i:i+sequence_length])  
        y.append(data[i+sequence_length, 3])  
    return np.array(X), np.array(y)  
  
sequence_length = 60  
X, y = create_sequences(scaled_features, sequence_length)  
train_size = int(len(X) * 0.8)  
train_X, train_y = X[:train_size], y[:train_size]  
test_X, test_y = X[train_size:], y[train_size:]
```

Figure 7 Sequence – 1

The dataset was split in two parts. First part is 80%, which is used for the training set. The validation set which is used to access the performance of the model during training is 10% and it did not change. However, the test set is now 20%.

The selected split set will be used on all the next iterations and will not change.

```
model = Sequential([  
    LSTM(50, return_sequences=True, input_shape=(train_X.shape[1], train_X.shape[2])),  
    Dropout(0.2),  
    BatchNormalization(),  
    LSTM(50, return_sequences=True),  
    Dropout(0.2),  
    BatchNormalization(),  
    LSTM(50),  
    Dropout(0.2),  
    BatchNormalization(),  
    Dense(1)  
])  
model.compile(optimizer='adam', loss='mean_squared_error')
```

Figure 8 Model – 2

Comparing to the 1st iteration, in the LSTM layers the units have been reduced from 100 to 50 to balance the model, and make it less complex but still being concerned about the overfitting so it can be prevented. Also dropout has been reduced to 0.2.

```
[ ] history = model.fit(train_X, train_y, epochs=50, batch_size=64, validation_split=0.1, verbose=1)
```

Figure 9 Training 2 – history

The initial 50 epochs were the same as well as the validation split, however the batch size has been increased to 64, the goal was to improve the training efficiency.

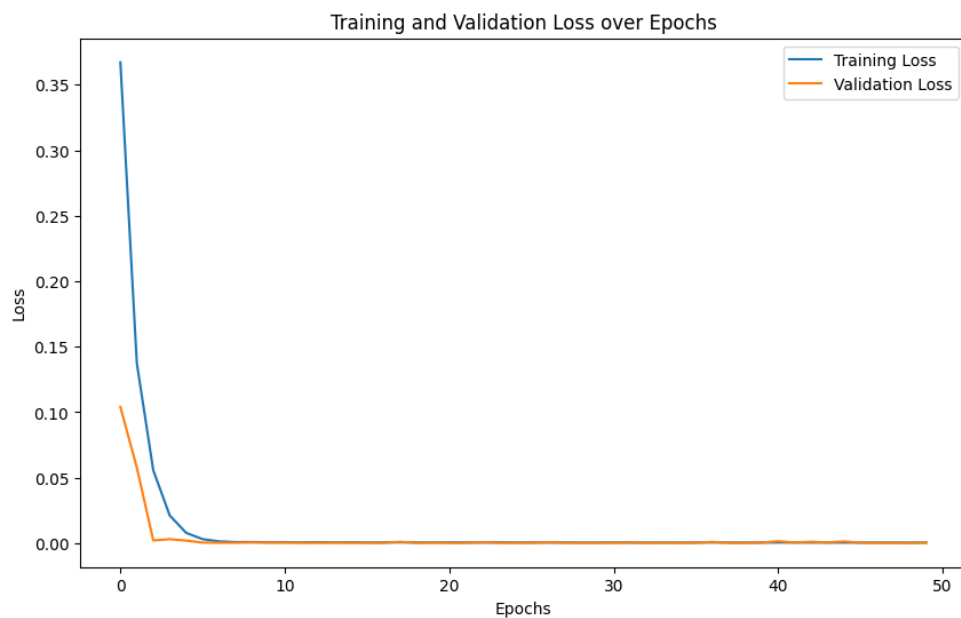


Figure 10 Training and Validation - 2

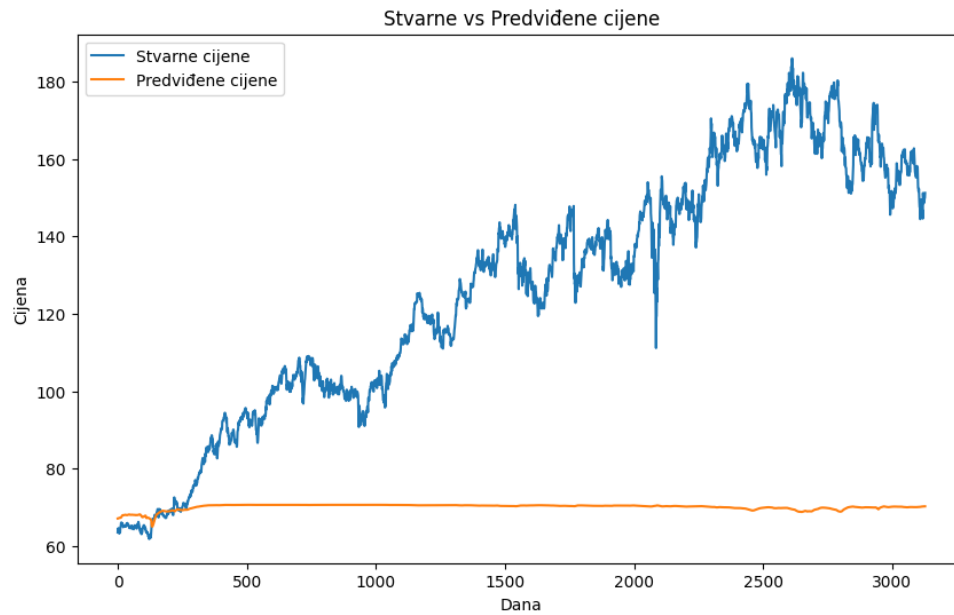


Figure 11 Real vs Predicted – 2

Training and validation plot showed that learning was very effective and overfitting was extremely minimal, which is an improvement comparing to the first iteration. However the accuracy of predicted prices was incredibly bad. The model was struggling to predict the real prices.

3RD Iteration

```
model = Sequential([
    LSTM(50, return_sequences=True, input_shape=(train_X.shape[1], train_X.shape[2]), kernel_regularizer=l2(0.001)),
    Dropout(0.2),
    BatchNormalization(),
    LSTM(50, return_sequences=True, kernel_regularizer=l2(0.001)),
    Dropout(0.2),
    BatchNormalization(),
    LSTM(50, kernel_regularizer=l2(0.001)),
    Dropout(0.2),
    BatchNormalization(),
    Dense(1)
])
model.compile(optimizer='adam', loss='mean_squared_error')
```

Figure 12 Model – 3

The initial three LSTM layers with 50 units have been kept, however L2 regularization has been added so overfitting can be prevented.

```
[ ] early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
    reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.001)
```

Figure 13 Stopping – 1

Callbacks have been added so when the models performance stops improving the overfitting will be stopped. ReduceLROnPlateau is added so it can adjust the learning rate.

```
[ ] history = model.fit(train_X, train_y, epochs=50, batch_size=64, validation_split=0.1, verbose=1, callbacks=[early_stopping, reduce_lr])
```

Figure 14 Training 3- history

Everything was kept the same except that the new callbacks that were added, which are early stopping and reduction for learning rate.

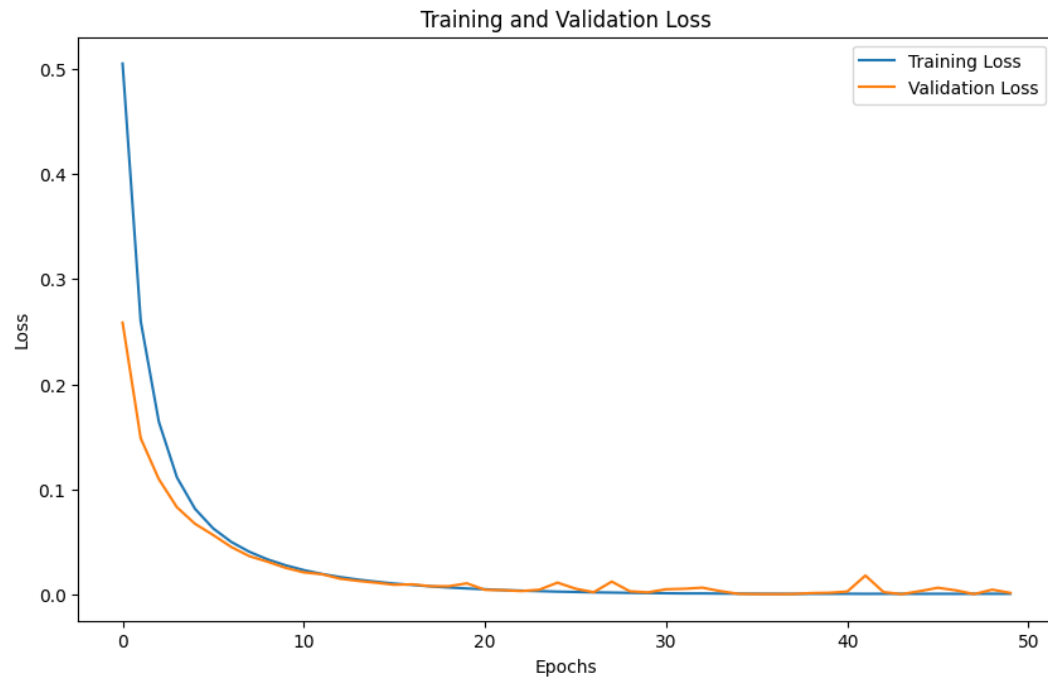


Figure 15 Training and Validation - 3

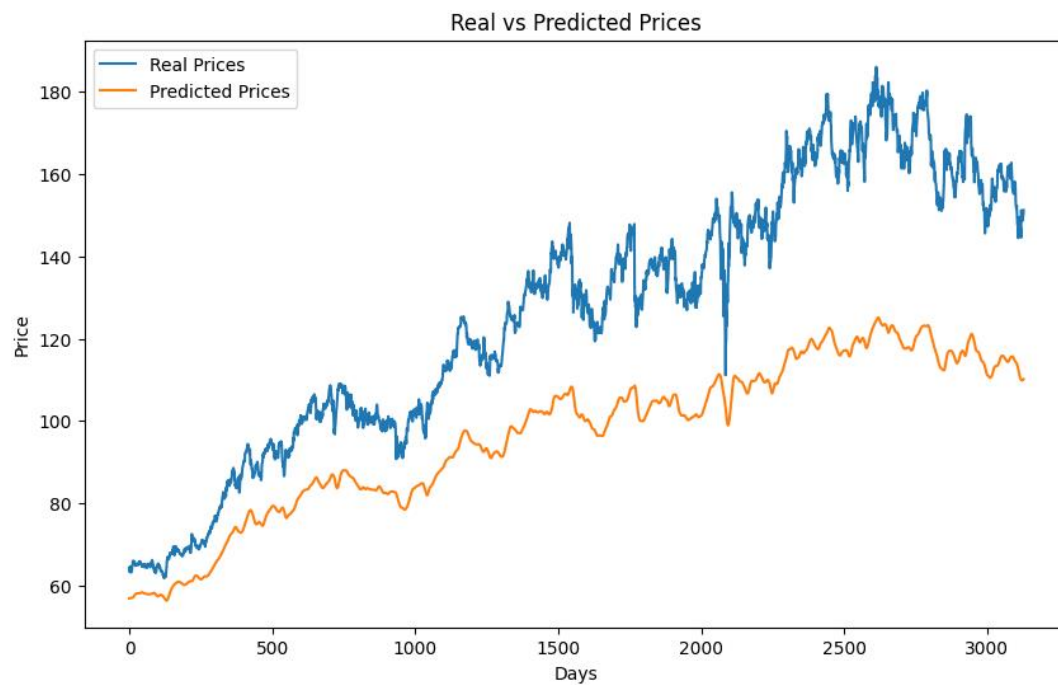


Figure 16 Real vs Predicted – 3

The training and validation plot showed that what model learned was effective, and it did good on unseen data in validation. The prediction for prices has significantly improved since the 1st

and 2nd iteration which is a very good sign, however the predicted prices were still not accurate to the real ones.

4th Iteration

```
model = Sequential([
    Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(train_X.shape[1], train_X.shape[2])),
    MaxPooling1D(pool_size=2),
    LSTM(50, return_sequences=True, kernel_regularizer=l2(0.001)),
    Dropout(0.2),
    BatchNormalization(),
    GRU(50, return_sequences=True, kernel_regularizer=l2(0.001)),
    Dropout(0.2),
    BatchNormalization(),
    GRU(50, kernel_regularizer=l2(0.001)),
    Dropout(0.2),
    BatchNormalization(),
    Dense(1)
])

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

Figure 17 Model – 4

For the 4th iteration, many additional layers have been added. Conv1D Layer was added because the aim was to catch local patterns before it passes it to LSTM and GRU layers. And to reduce the output of Conv1D was added MaxPooling1D layer. LSTM and GRU layers were used together so we can get the best out of both of them.

```
[ ] early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
    reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.001)

[ ] history = model.fit(train_X, train_y, epochs=50, batch_size=64, validation_split=0.1, verbose=1, callbacks=[early_stopping, reduce_lr])
```

Figure 18 Callbacks and training

The callbacks, as well as epochs, batch size all remained the same and there was no goal or aim to change anything in that aspect yet.

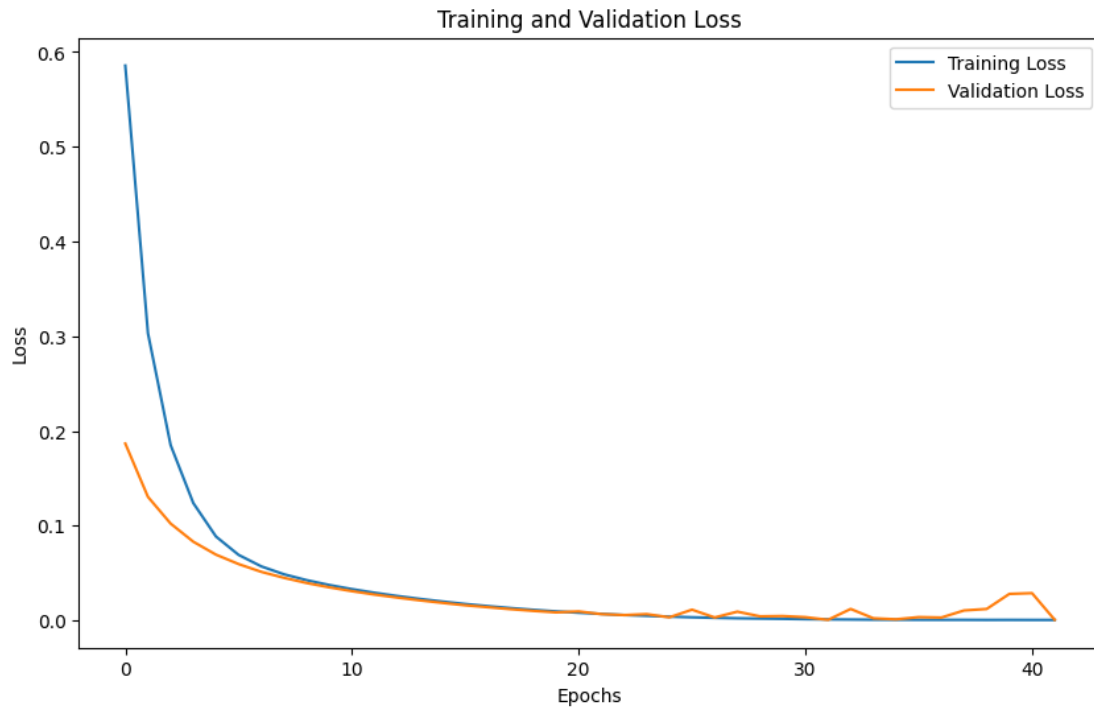


Figure 19 Training and Validation 4

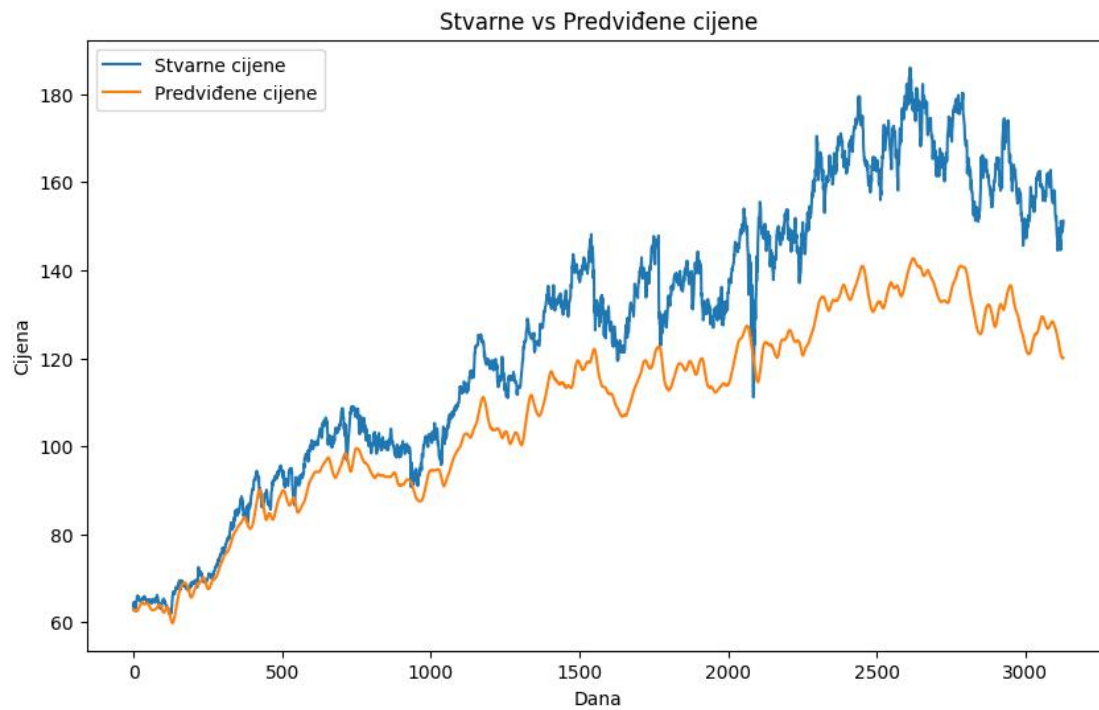


Figure 20 Real vs Predicted 4

Training and validation loss were shown in plot how the model learned effectively and both losses were near a low value, however there were some spikes in the validation loss. When it comes to price prediction accuracy, the predicted prices started off good but fell off at the start of the middle which gave us a sign that model still needs improving.

5TH Iteration

```
model = Sequential([
    Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(train_X.shape[1], train_X.shape[2])),
    MaxPooling1D(pool_size=2),
    LSTM(100, return_sequences=True, kernel_regularizer=l2(0.001)),
    Dropout(0.3),
    BatchNormalization(),
    GRU(100, return_sequences=True, kernel_regularizer=l2(0.001)),
    Dropout(0.3),
    BatchNormalization(),
    GRU(100, kernel_regularizer=l2(0.001)),
    Dropout(0.3),
    BatchNormalization(),
    Dense(1)
])

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

Figure 21 Model 5

In the 5th iteration there have not been changes in layers, however units have been increased to 100, because the aim was for the model to learn more complex patterns. Also dropout has been increased to 0.3, the aim behind that was to prevent the overfitting even better.

```
[ ] history = model.fit(train_X, train_y, epochs=100, batch_size=32, validation_split=0.1, verbose=1, callbacks=[early_stopping, reduce_lr])
```

Figure 22 Training 5 – history

There have been changes in training the model, where epochs were increased from 50 to 100, however the model in training did not go after 30th epoch. Also the batch size has been reduced from 64 to 32 because goal was to improve efficiency during training.

```
+ Code + Text
Epoch 15/100
352/352 [=====] - 32s 90ms/step - loss: 0.0029 - val_loss: 0.0018 - lr: 0.0010
Epoch 16/100
352/352 [=====] - 32s 90ms/step - loss: 0.0022 - val_loss: 0.0123 - lr: 0.0010
Epoch 17/100
352/352 [=====] - 33s 94ms/step - loss: 0.0015 - val_loss: 7.4969e-04 - lr: 0.0010
Epoch 18/100
352/352 [=====] - 32s 90ms/step - loss: 0.0012 - val_loss: 5.2351e-04 - lr: 0.0010
Epoch 19/100
352/352 [=====] - 32s 90ms/step - loss: 9.3793e-04 - val_loss: 0.0010 - lr: 0.0010
Epoch 20/100
352/352 [=====] - 32s 91ms/step - loss: 8.4882e-04 - val_loss: 2.7008e-04 - lr: 0.0010
Epoch 21/100
352/352 [=====] - 32s 91ms/step - loss: 8.1261e-04 - val_loss: 0.0166 - lr: 0.0010
Epoch 22/100
352/352 [=====] - 32s 90ms/step - loss: 8.5679e-04 - val_loss: 0.0062 - lr: 0.0010
Epoch 23/100
352/352 [=====] - 32s 90ms/step - loss: 8.7424e-04 - val_loss: 0.0102 - lr: 0.0010
Epoch 24/100
352/352 [=====] - 33s 93ms/step - loss: 9.4024e-04 - val_loss: 0.0070 - lr: 0.0010
Epoch 25/100
352/352 [=====] - 32s 90ms/step - loss: 8.6380e-04 - val_loss: 0.0024 - lr: 0.0010
Epoch 26/100
352/352 [=====] - 32s 90ms/step - loss: 9.3350e-04 - val_loss: 5.4238e-04 - lr: 0.0010
Epoch 27/100
352/352 [=====] - 32s 91ms/step - loss: 9.3561e-04 - val_loss: 0.0018 - lr: 0.0010
Epoch 28/100
352/352 [=====] - 33s 92ms/step - loss: 0.0010 - val_loss: 0.0096 - lr: 0.0010
Epoch 29/100
352/352 [=====] - 32s 91ms/step - loss: 9.3791e-04 - val_loss: 0.0018 - lr: 0.0010
Epoch 30/100
352/352 [=====] - 32s 90ms/step - loss: 9.0819e-04 - val_loss: 0.0035 - lr: 0.0010
```

Figure 23 Epoch 5

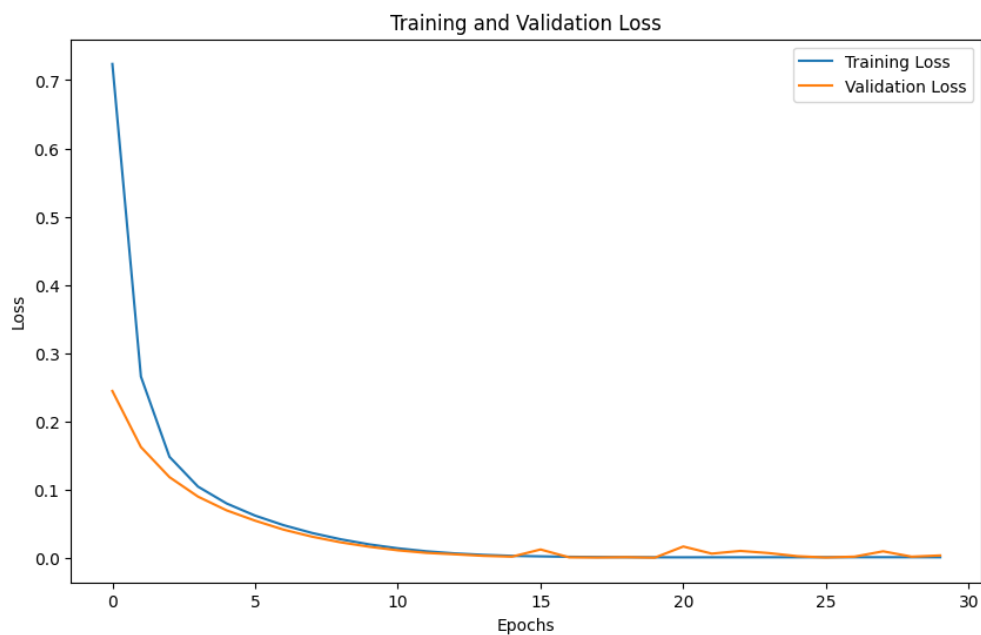


Figure 24 Training and Validation 5

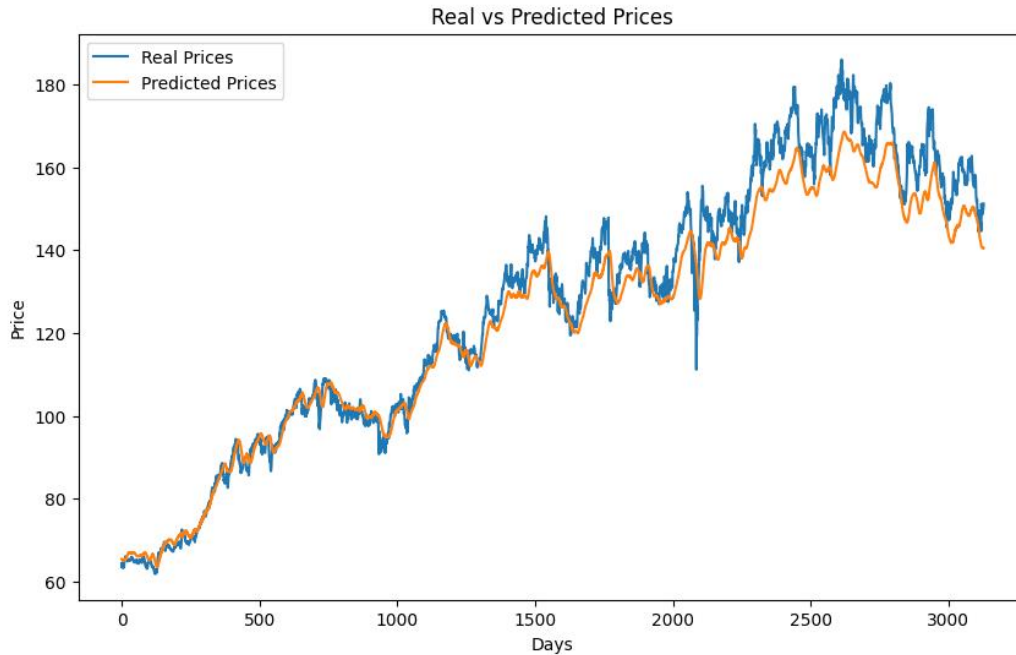


Figure 25 Real vs Predicted 5

Training and validation loss are again indicating that model learned effectively, however there were less spikes noticeable in validation loss so the model did well during unseen data in validation. When it comes to prediction accuracy the improvement is extremely significant. Conclusion is that the increased units and dropout contributed to the result however after the middle the model was still not accurate, and was not following the real prices, which gave a thought it might be too complex so in the final iteration the model was simplified and unnecessary parts removed.

6th Iteration

The first step was importing all the libraries that were needed and reading the dataset. The features that were used for the model include: “Open”, “High”, “Low”, “Close” and “Volume”. And then the data was scaled with “MinMaxScaler”.

```
sequence_length = 60
X, y = create_sequences(scaled_features, sequence_length)

train_size = int(len(X) * 0.8)
train_X, train_y = X[:train_size], y[:train_size]
test_X, test_y = X[train_size:], y[train_size:]
```

Figure 26 Sequence – Final

Sequence part did not change, there was still length of 60, and the dataset was split. First part which was training was made out of 80% and 10% of validation set inside it, while the testing dataset one was 20%.

```
model = Sequential([
    LSTM(50, return_sequences=True, input_shape=(train_X.shape[1], train_X.shape[2])),
    Dropout(0.2),
    BatchNormalization(),
    LSTM(50, kernel_regularizer=tf.keras.regularizers.l2(0.001)),
    Dropout(0.2),
    BatchNormalization(),
    Dense(1)
])

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0005), loss='mean_squared_error')
```

Figure 27 Model Final

For this model, only 2 LSTM layers were used and each layer had 50 units. Return sequences parameters was true so that the LSTM layers were a sequence and they can stack. Dropouts were reduced to 0.2 after each LSTM layer. Batch normalization was applied so the training process is more stable and faster. Adam optimizer was chosen because it is the best one and with biggest efficiency and the learning rate is of lower value (0.0005) because it gives better updates to the weights.

```
[ ] early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
    reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.00001)

[ ] history = model.fit(train_X, train_y, epochs=50, batch_size=32, validation_split=0.1, verbose=1, callbacks=[early_stopping, reduce_lr])
```

Figure 28 Callbacks, training final

The callbacks did not change and they remained the same. However the epochs were reduced from 100 to 50 and the batch size along with the validation split remained all the same.

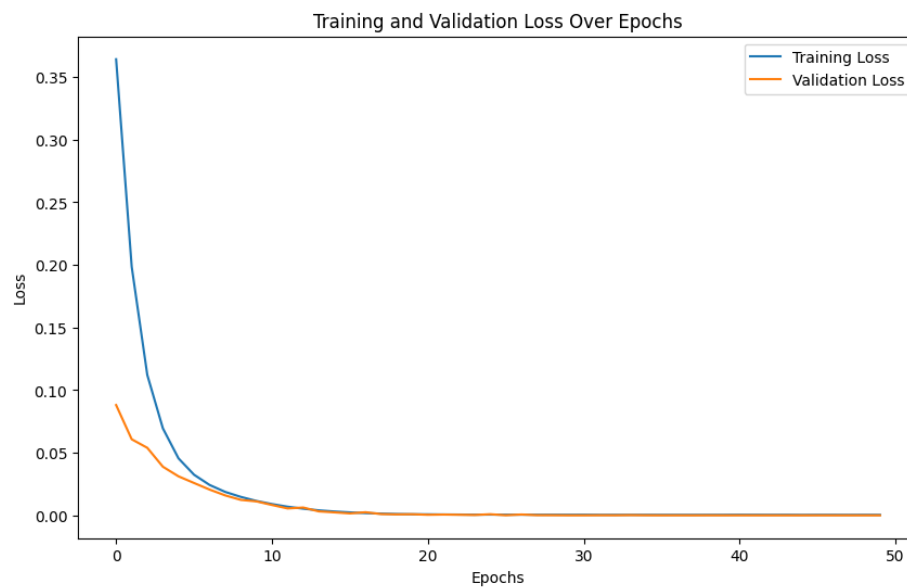


Figure 29 Training and Validation Final

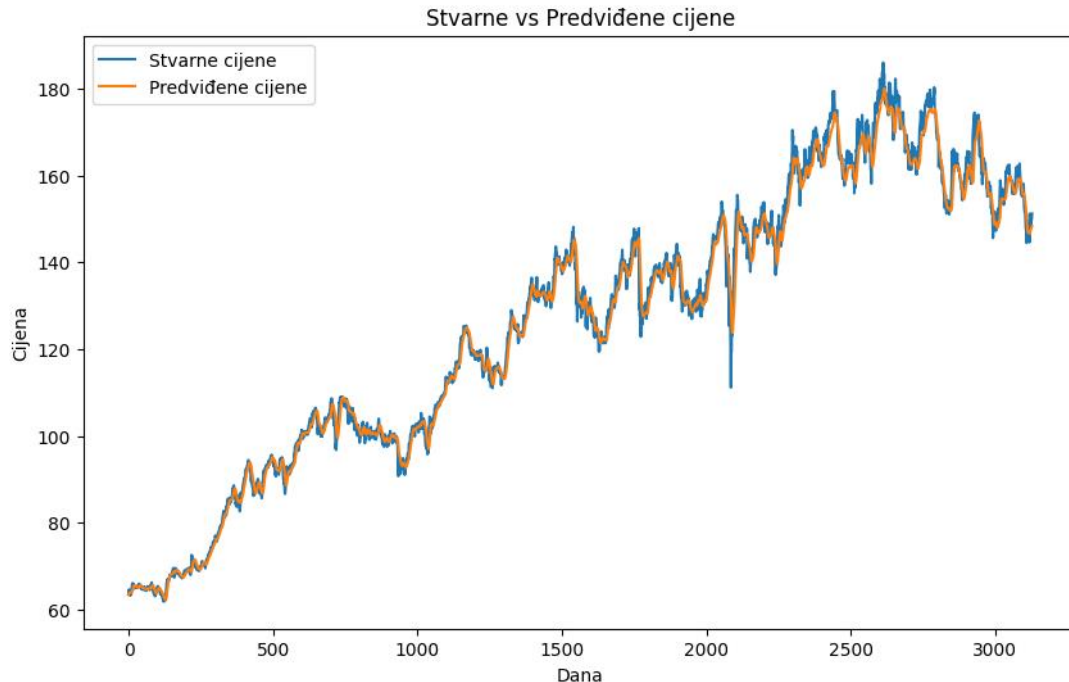


Figure 30 Real vs Predicted Final

The final plots showed incredibly significant improvement in prediction accuracy as well as training and validation loss. There are no jumps or spikes on validation loss, and the predicted prices are going well along the actual ones.

```
def predict_future_prices(model, data, days=30, sequence_length=60):
    future_predictions = []
    current_sequence = data[-sequence_length:]

    for _ in range(days):
        current_sequence_resaped = current_sequence.reshape((1, sequence_length, data.shape[1]))
        future_price = model.predict(current_sequence_resaped)[0, 0]

        future_predictions.append(future_price)

        new_sequence_element = np.append(current_sequence[-1, :-1], future_price)
        current_sequence = np.append(current_sequence[1:], new_sequence_element.reshape((1, data.shape[1])), axis=0)

    return future_predictions

[ ] future_days = 30
future_predictions = predict_future_prices(model, scaled_features, days=future_days)
future_predictions = scaler.inverse_transform(np.concatenate(
    (np.zeros((future_days, 3)), np.array(future_predictions).reshape((-1, 1), np.zeros((future_days, 1))), axis=1))[:, 3])
```

Figure 31 Future prices Final

This function was made to take the parameters of the model and create future prices that are not displayed on the graph. The prediction loop was designed to go 30 days into the future based on the model that has been created and the data.

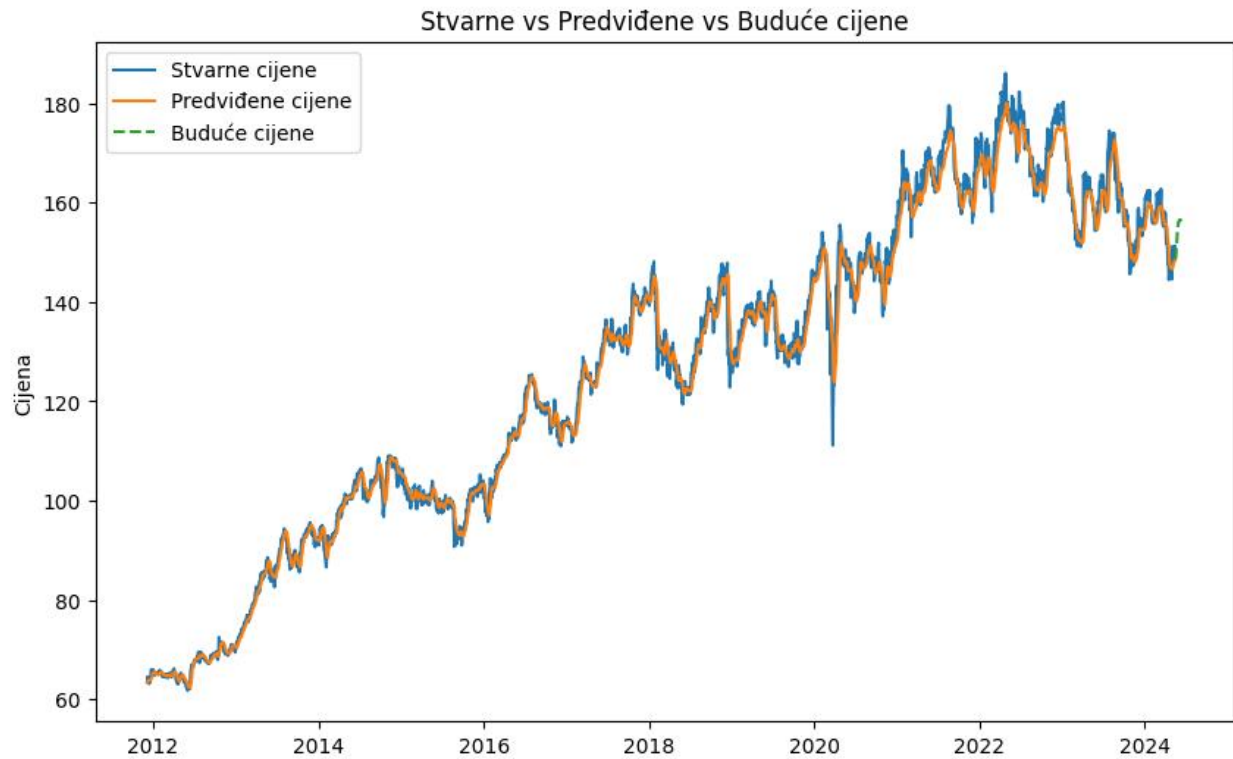


Figure 32 Future prices Final

```

predicted_test_prices = scaler.inverse_transform(np.concatenate(
    (np.zeros_like(predicted_test_prices), np.zeros_like(predicted_test_prices), np.zeros_like(predicted_test_prices)))
real_test_prices = scaler.inverse_transform(np.concatenate(
    (np.zeros_like(test_y.reshape(-1,1)), np.zeros_like(test_y.reshape(-1,1)), np.zeros_like(test_y.reshape(-1,1)))))

from sklearn.metrics import mean_squared_error, mean_absolute_error

mse = mean_squared_error(real_test_prices, predicted_test_prices)
rmse = np.sqrt(mse)
mae = mean_absolute_error(real_test_prices, predicted_test_prices)

print(f'Test MSE: {mse}')
print(f'Test RMSE: {rmse}')
print(f'Test MAE: {mae}')
Test MSE: 7.432022330539274
Test RMSE: 2.726173569408095
Test MAE: 1.9191963399079073

```

```

Test MSE: 7.432022330539274
Test RMSE: 2.726173569408095
Test MAE: 1.9191963399079073

```

Figure 33 TEST

Separately the test has been created to evaluate if the model is doing correctly or if it might have been overfitting on seen data, which turned out not to be true.

5.1 Final model

```
[ ] df['SMA_20'] = df['Close'].rolling(window=20).mean()
    df['EMA_20'] = df['Close'].ewm(span=20, adjust=False).mean()
    df['RSI'] = 100 - (100 / (1 + df['Close'].diff().apply(lambda x: x if x > 0 else 0).rolling(window=14).mean()))
    df['OBV'] = (np.sign(df['Close'].diff()) * df['Volume']).fillna(0).cumsum()
    df.fillna(method='bfill', inplace=True)
```

Figure 34 Indicators

For the final mode, additional indicators were applied. SMA which will help us track the average stock price of the period of 20 days. EMA also, which does the same however it responds much faster to new information, in comparison to SMA. RSI will measure the speed the stock prices change during the period of 14 days, and OBV which will help to follow the trend of stock prices.

```
▶ kf = KFold(n_splits=5, shuffle=True, random_state=42)
  results = []

  for train_index, val_index in kf.split(X):
      train_X, val_X = X[train_index], X[val_index]
      train_y, val_y = y[train_index], y[val_index]

      model = Sequential([
          LSTM(50, return_sequences=True, input_shape=(train_X.shape[1], train_X.shape[2])),
          Dropout(0.2),
          BatchNormalization(),
          LSTM(50, kernel_regularizer=regularizers.l2(0.001)),
          Dropout(0.2),
          BatchNormalization(),
          Dense(1)
      ])

      model.compile(optimizer=Adam(learning_rate=0.0005), loss='mean_squared_error')

      early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
      reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.00001)

      history = model.fit(train_X, train_y, epochs=50, batch_size=32, validation_data=(val_X, val_y), verbose=1, callbacks=[early_stopping, reduce_lr])

      val_loss = model.evaluate(val_X, val_y, verbose=0)
      results.append(val_loss)
```

Figure 35 Training

Unlike the previous iterations, here k-fold cross validation. The dataset has been split into 5 folds. The “shuffle” equals true which means the data will be randomized and not done in straight time line. The model, as well as the callbacks will remain the same as in 6th iteration, the only change that happened is the way data is split.

```
[ ] average_val_loss = np.mean(results)
    print(f'Average Validation Loss: {average_val_loss}')
```

➡ Average Validation Loss: 9.39228106290102e-05

Figure 36 Validation loss

Average validation loss across all the folds has been calculated. It was done to evaluate the performance of a model, to see if it is reliable. By doing this step it assures that the model does not depend on just one data split.

```
model = Sequential([
    LSTM(50, return_sequences=True, input_shape=(X.shape[1], X.shape[2])),
    Dropout(0.2),
    BatchNormalization(),
    LSTM(50, kernel_regularizer=regularizers.l2(0.001)),
    Dropout(0.2),
    BatchNormalization(),
    Dense(1)
])

model.compile(optimizer=Adam(learning_rate=0.0005), loss='mean_squared_error')

early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.00001)

history = model.fit(X, y, epochs=50, batch_size=32, validation_split=0.1, verbose=1, callbacks=[early_stopping, reduce_lr])
```

Figure 37 Training 2nd model, final

The second model has the same composition as the last one, as in the same layers and hyperparameters, however it trains on the entire dataset with a single validation split, it uses validation splits based on K-fold and makes it quickly train and evaluate the performance.

```
predicted_prices = model.predict(X)
predicted_prices_extended = np.zeros((predicted_prices.shape[0], scaled_features.shape[1]))
predicted_prices_extended[:, 3] = predicted_prices[:, 0]
predicted_prices = scaler.inverse_transform(predicted_prices_extended)[:, 3]
```

Figure 38 Test

By using the trained LSTM model it will make prediction on the input data. The 2D array of zeros will be created just like in the previous iterations. Then the predicted prices will be put into the fourth column and then the inverse transformation will scale back values from the fourth column in their original scale.



Figure 39 Training and Validation

Training and the validation loss reduce quite quickly which gives a sign that the model learns effectively the patterns in data. After the 20 epoch, they both stabilize which indicates that the model reached its optimal performance both on training and validation sets and shows no signs of overfitting

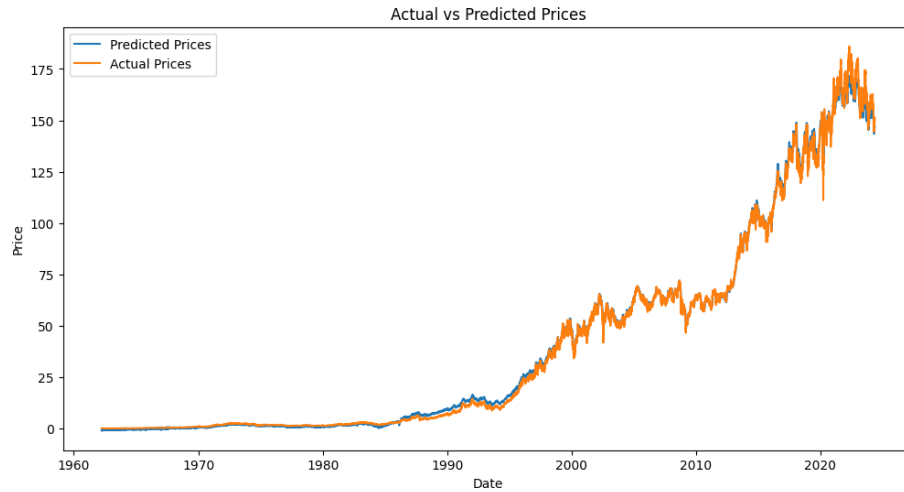


Figure 40 Predicted prices over all folds

Based on the results, we can see that the model works very well in its job to predict stock prices. Results that are obtained with K-Fold cross-validation indicate that the model has a strong predictive performance over the entire timeline.

6. Conclusion

This project that made an LSTM neural network. The model was quite successful and efficient in its task to predict stock prices of Johnson & Johnson by using their historical stock data. The model started poorly however, through the various iterations and alterations of the model the results improved each time and it provided a valuable insight. Techniques were used such as data preprocessing in order to choose what columns will be implemented for the prices, as well as constant alteration of layers in the model and hyperparameters. Projects such as this are very valuable to the future, because of its ability for stock price prediction in the future and possibly some other financial tasks. The skills gained from this project also contribute to my bigger goal of becoming a data analyst.

7. Pictures

Figure 1 JNJ - Stock prices data.....	6
Figure 2 Model – 1.....	10
Figure 3 Training 1 – History	11
Figure 4 Testing	11
Figure 5 Training and Validation – 1.....	12
Figure 6 Real vs Predicted – 1	12
Figure 7 Sequence – 1	14
Figure 8 Model – 2.....	14
Figure 9 Training 2 – history	15
Figure 10 Training and Validation - 2	15
Figure 11 Real vs Predicted – 2	16
Figure 12 Model – 3.....	17
Figure 13 Stopping – 1.....	17
Figure 14 Training 3- history	17
Figure 15 Training and Validation - 3	18
Figure 16 Real vs Predicted – 3	18
Figure 17 Model – 4.....	20
Figure 18 Callbacks and training	20
Figure 19 Training and Validation 4.....	21
Figure 20 Real vs Predicted 4	21
Figure 21 Model 5.....	23
Figure 22 Training 5 – history	23
Figure 23 Epoch 5.....	24
Figure 24 Training and Validation 5.....	24
Figure 25 Real vs Predicted 5	25
Figure 26 Sequence – Final.....	26
Figure 27 Model Final	26
Figure 28 Callbacks, training final.....	27
Figure 29 Training and Validation Final	27
Figure 30 Real vs Predicted Final	28
Figure 31 Future prices Final.....	28
Figure 32 Future prices Final.....	29
Figure 33 TEST.....	29
Figure 34 Indicators	31
Figure 35 Training	31
Figure 36 Validation loss	32
Figure 37 Training 2nd model, final	32
Figure 38 Test	32
Figure 39 Training and Validation.....	33
Figure 40 Predicted prices over all folds	34

8. References

1. Abhay Kumar Yadav, Virendra P. Vishwakarma, An Integrated Blockchain Based Real Time Stock Price Prediction Model by CNN, Bi LSTM and AM, Procedia Computer Science, Volume 235, 2024, Pages 2630-2640, ISSN 1877-0509, <https://doi.org/10.1016/j.procs.2024.04.248>.
2. Burak Gülmez, Stock price prediction with optimized deep LSTM network with artificial rabbits optimization algorithm, Expert Systems with Applications, Volume 227, 2023, 120346, ISSN 0957-4174, <https://doi.org/10.1016/j.eswa.2023.120346>.
3. Abdul Quadir Md, Sanjit Kapoor, Chris Junni A.V., Arun Kumar Sivaraman, Kong Fah Tee, Sabireen H., Janakiraman N., Novel optimization approach for stock price forecasting using multi-layered sequential LSTM, Applied Soft Computing, Volume 134, 2023, 109830, ISSN 1568-4946, <https://doi.org/10.1016/j.asoc.2022.109830>.
4. Hum Nath Bhandari, Binod Rimal, Nawa Raj Pokhrel, Ramchandra Rimal, Keshab R. Dahal, Rajendra K.C. Khatri, Predicting stock market index using LSTM, Machine Learning with Applications, Volume 9, 2022, 100320, ISSN 2666 <https://doi.org/10.1016/j.mlwa.2022.100320>.
5. Adil Moghar, Mhamed Hamiche, Stock Market Prediction Using LSTM Recurrent Neural Network, Procedia Computer Science, Volume 170, 2020, Pages 1168-1173, ISSN 1877-0509, <https://doi.org/10.1016/j.procs.2020.03.049>.
6. SATISH GUNJA Tutorial: K Fold Cross Validation (2020) <https://www.kaggle.com/code/satishgunjal/tutorial-k-fold-cross-validation>
7. Tandon, Sakshi & Tripathi, Shreya & Saraswat, Pragya & Dabas, Chetna. (2019). Bitcoin Price Forecasting using LSTM and 10-Fold Cross validation. 323-328. 10.1109/ICSC45622.2019.8938251.