Capstone Project

August 2021

1 Definition

1.1 Project Overview

My project for the Capstone Project was to predict stock market data, using Recurrent Neural Networks and this data set from kaggle. The data is itself taken from *Yahoo!* finance.

The data set contains the daily prices of multiple of indexes that track stock exchanges from different countries. I focused mostly on the New York Stock Exchange, NYA. However, my work could easily be adapted to function with any of the stocks present in the data set.

1.2 Problem Statement

The problem I want to solve is to have as inputs a time series, that represents the value of the NYA index over a certain period (with each point representing a certain day) and have an output value that predicts the future prices of this index.

I explored two methods for this prediction, the first one was to try to predict multiple future values for this index at the same time, and the second approach only tried to predict the next value of the index. Only the second method provided exploitable results, however I think that the first approach would be more useful in a real-world application, which is why I decided to present my results for it in this report.

1.3 Metrics

The main metric that I will be using for this project is the root mean squared error (RMSE) between the predicted values outputted by the model and the actual values. I have access to a reference for the RMSE in this kaggle publication.

Since I am working with time series, having a graphical representation of the predicted result, overlayed with the real data, can also be quite useful in

predicting the performances of the model (however it cannot be reduced to a single real number).

2 Analysis

2.1 Data Exploration

The data set contains three csv files:

The first one is *indexData.csv*, it contains all the values for the index prices. There are 13 indexes: HSI, NYA, IXIC, SS, N225, N100, SZ, GSPTSE, NSEI, GDAXI, SSMI, TWII, J203.JO. My analysis will be mostly focused on the NYA index. Each row of the *indexData.csv* file contains 9 columns:

- Index: name of the index.
- Date: the day for this entry.
- Open: the value of the index when the stock market opened.
- High: the highest value of the index for the day.
- Low: the lowest value of the index for the day.
- Close: the closing value.
- Adj Close: the closing value adjusted for dividends. This means that if the stock pays dividends, the dividend is subtracted form the closing price. The adjusted value is a better representation of the value of the stock, since the dividend will be paid out to the investors.
- Volume: the number of shares that have been traded during the day.

The second csv file is *indexInfo.csv*, it is a short file that contains information on each index. Its region, full name, abbreviated name and currency.

The last file called *indexProcessed.csv* is the same as *indexData.csv* but with the rows that are incomplete that are removed and an additional column, CloseUSD, which is the Adj Close price converted in US dollars. As a result of this first operation, not every consecutive day is present in the data set.

Date	Index	Open	High	Low	Close	Adj Close	Volume	CloseUSD
1986-12-31	HSI	2568.300049	2568.300049	2568.300049	2568.300049	2568.300049	0.0	333.879006
1987-01-02	HSI	2540.100098	2540.100098	2540.100098	2540.100098	2540.100098	0.0	330.213013
1987-01-05	HSI	2552.399902	2552.399902	2552.399902	2552.399902	2552.399902	0.0	331.811987
1987-01-06	HSI	2583.899902	2583.899902	2583.899902	2583.899902	2583.899902	0.0	335.906987
1987-01-07	HSI	2607.100098	2607.100098	2607.100098	2607.100098	2607.100098	0.0	338.923013

Figure 1: *indexProcessed.csv* with the Date set as index

2.2 Exploratory Visualization

Here is a graphical representation of 3 of the 13 indexes. There is the New York Stock Exchange, NYA, the Hang Seng Index, HSI, the stock market index of Hong Kong, and the Euronext 100, N100, that represents the most traded stocks of the Euronext, the main stock market in the euro area.

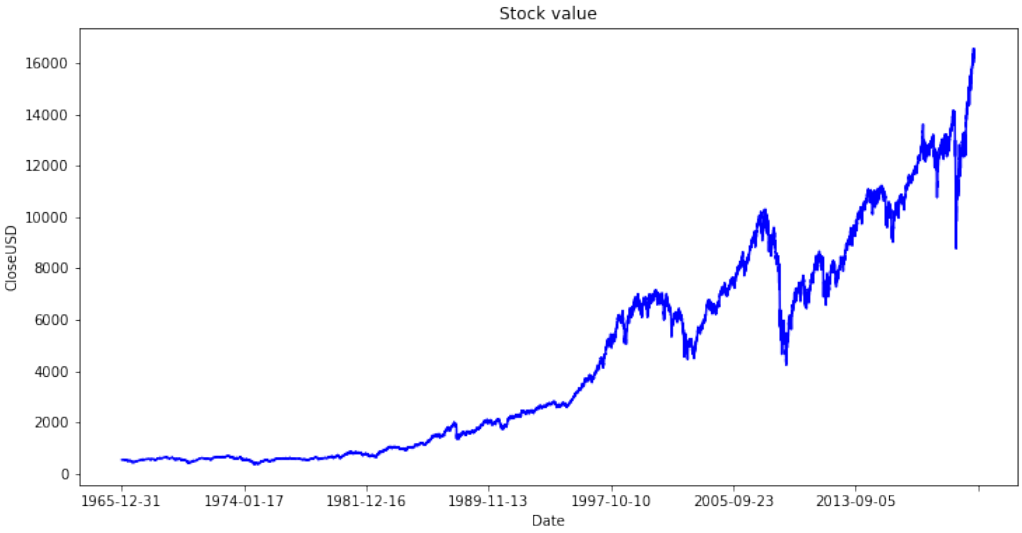

Figure 2: Data points for the NYA

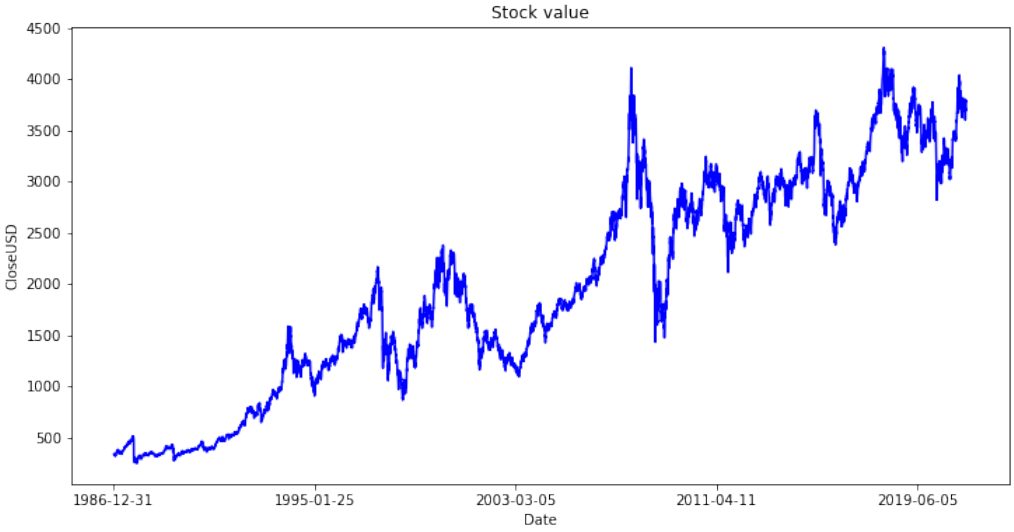

Figure 3: Data points for the HSI

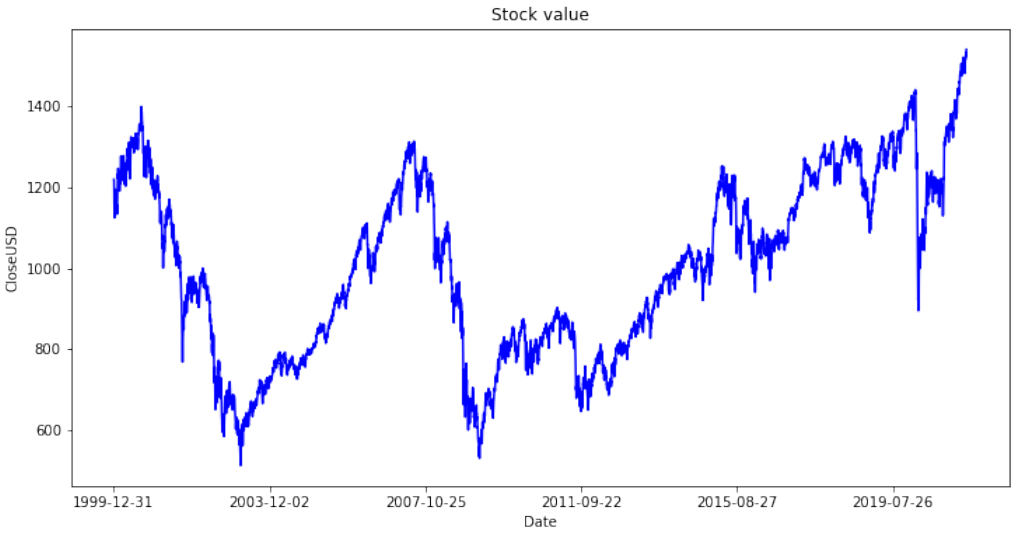

Figure 4: Data points for the N100

From these, we can gather that all the indexes have different starting dates. In the notebook I computed that the start dates range from 1965-12-31 for N225,

the Tokyo Stock Exchange, to 2012-02-08, for J203.JO, the Johannesburg Stock Exchange.

On figure 5, I have put two indexes, with the time window reduced so we can see the correlation between different indexes. The variations are similar on the two indexes, however the HSI index varies quicker than the NYA. This means that it could make sense to use all the indexes to train a single neural network.

In the first approaches that I will detail in the next section, I used all of the indexes to train the first model. For the second mode, I singled out an index for the second model. The second model yielded better results, but the first approach could probably be improved and have similar or better performances.

2.3 Algorithms and techniques

As I mentioned in the last section, I tried to solve this problem with two different approaches. The first one was inspired from the Time Forecasting machine learning case study, that uses the DeepAR forecasting algorithm. The second solution is inspired from this kaggle publication. I adapted the RNN in this notebook to work with PyTorch and SageMaker.

2.3.1 DeepAR

The first model I used was the DeepAR model, it is a forecasting algorithm provided by Amazon SageMaker that predicts a scalar value and works with a recurrent neural network (RNN). It takes as input a time series, and outputs another time series that attempts to predict the future values of the input. One of the hyperparameters of DeepAR is *prediction_length* which predicts the length of the output.

The characteristics of this model make it a perfect candidate for this problem, however I found that the extra features of the DeepAR model were not required, and that I would prefer to have more control over the model and the data structure. This is why I decided to have a second approach with a RNN build from the ground up with PyTorch.

2.3.2 PyTorch

The second model I used was a RNN I built using PyTorch. It is composed a single long short-term memory (LSTM) layer, and a dense layer after that. The LSTM layer makes this network an RNN, because the output of previous steps is fed as input to the next steps. This feature is important for my problem because the stock market values are not solely based on the last value, they also depend on previous values.

I used the model from a blog post at medium.com, the original use case of this model was to predict Amazon stock price. One thing that is important to note on this model is that since I only want to predict one value with my model, I select the last value outputted by the LSTM layer, like so:

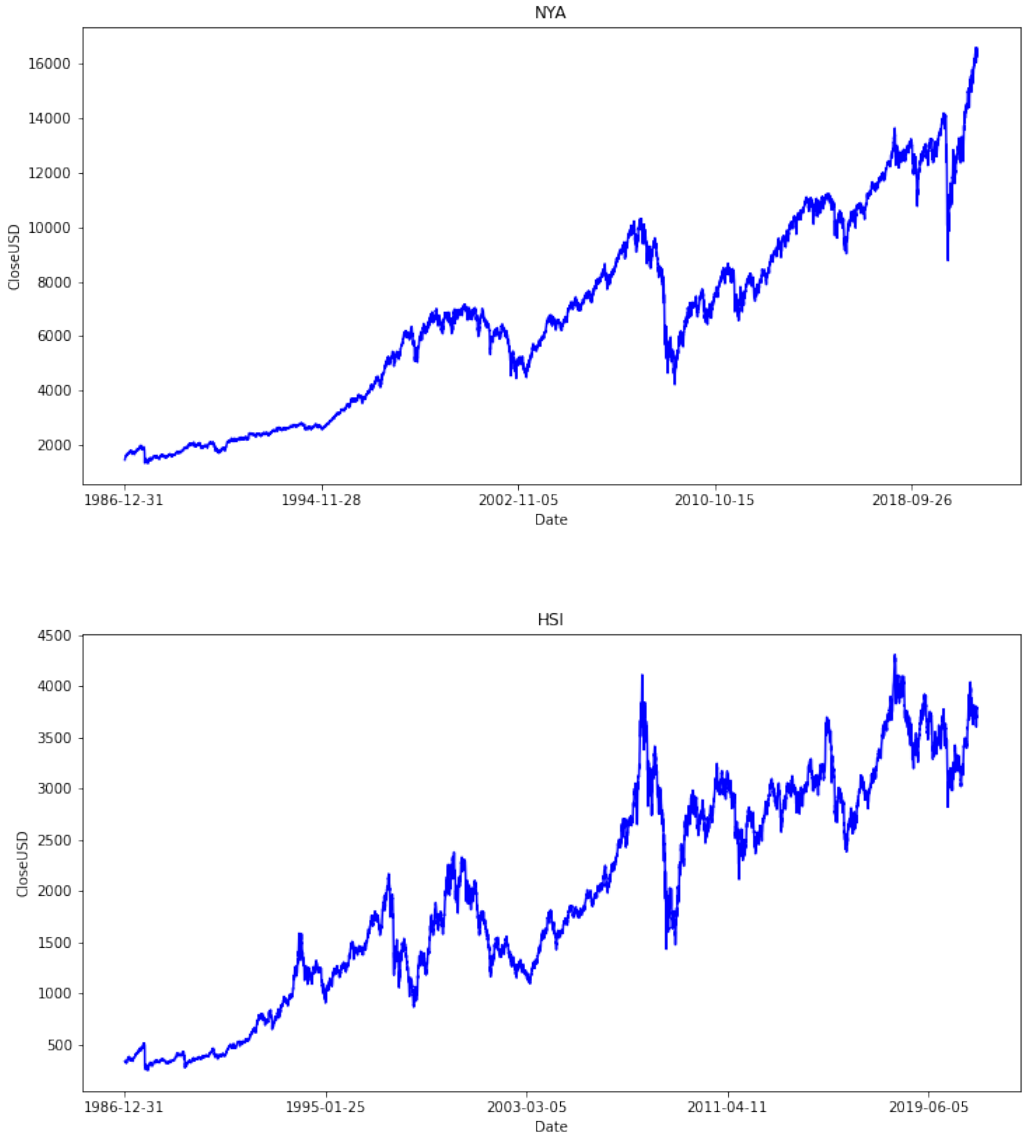

Figure 5: NYA and HSI with same time window

```
out = out[:, -1, :]
```

The drawback of this model is that it only predicts a single value at a time. This limitation is not prohibitive since the time step is a day, predictions over a day can still be useful in a real-world setting.

2.4 Benchmark

2.4.1 DeepAR

One of the reasons I decided to switch from the DeepAR model to the custom PyTorch model is that the metrics were not clearly defined for the DeepAR model. For the predictions, the output was composed of a median and an 80% confidence interval. In the kaggle publication, the metric used was RMSE, and it would not completely make sense to compute it using a median predicted value.

Another metric that could be used for the DeepAR model would simply to check the graphical output of the model compared to the actual values. This method is easy to implement, however, it lacks a simple metric in the form of a real number. I have provided an example of this in figure 6.

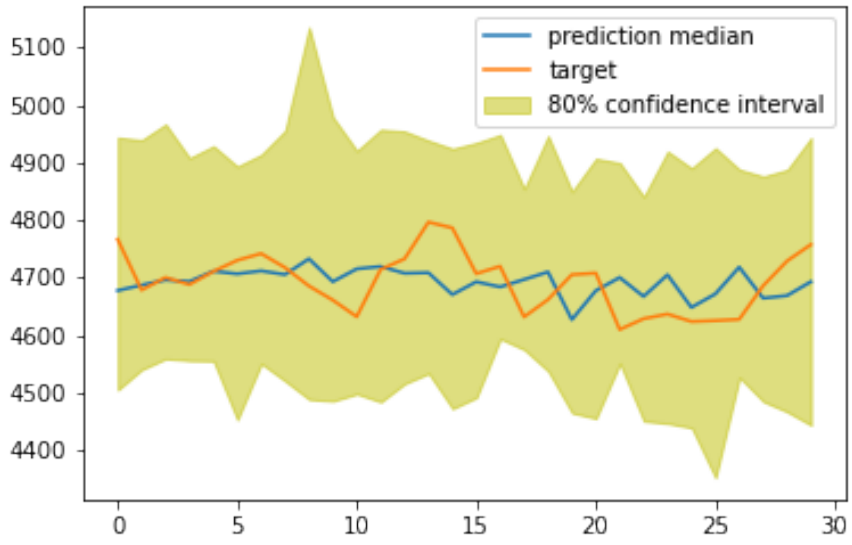

Figure 6: Ouput of the DeepAR model

2.4.2 PyTorch

For the PyTorch model, the benchmark is more straightforward, I can simply compute the root mean squared error between the predicted values and the actual values, and compare it to the RMSE presented in the kaggle publication. The graphical method is still valid and provides additional information on what might be causing the inaccuracies of the model.

3 Methodology

3.1 Data Preprocessing

The only preprocessing that was common to the two models is that I set the index of the Pandas DataFrame to be equal to the *Date* column (this step could cause some issues since the *Date* column is not unique, but each row is kept). Some of the prepossessing has already been done in the *indexProcessed.csv* file, rows that were incomplete have been removed and a column *CloseUSD* has been added, which helps if we want to train the model with data from different indexes.

3.1.1 DeepAR

For the DeepAR model, the first step was to create time series, for this I had to split the data into sequences of length *sample_lenght*, and to be careful not to mix different indexes in the same time series. I then created the time series for training, which were the previously created time series, but with the tail removed, with the number of points removed equal to *prediction_length*. On figure 7, you can see what was the result of creating these time series.

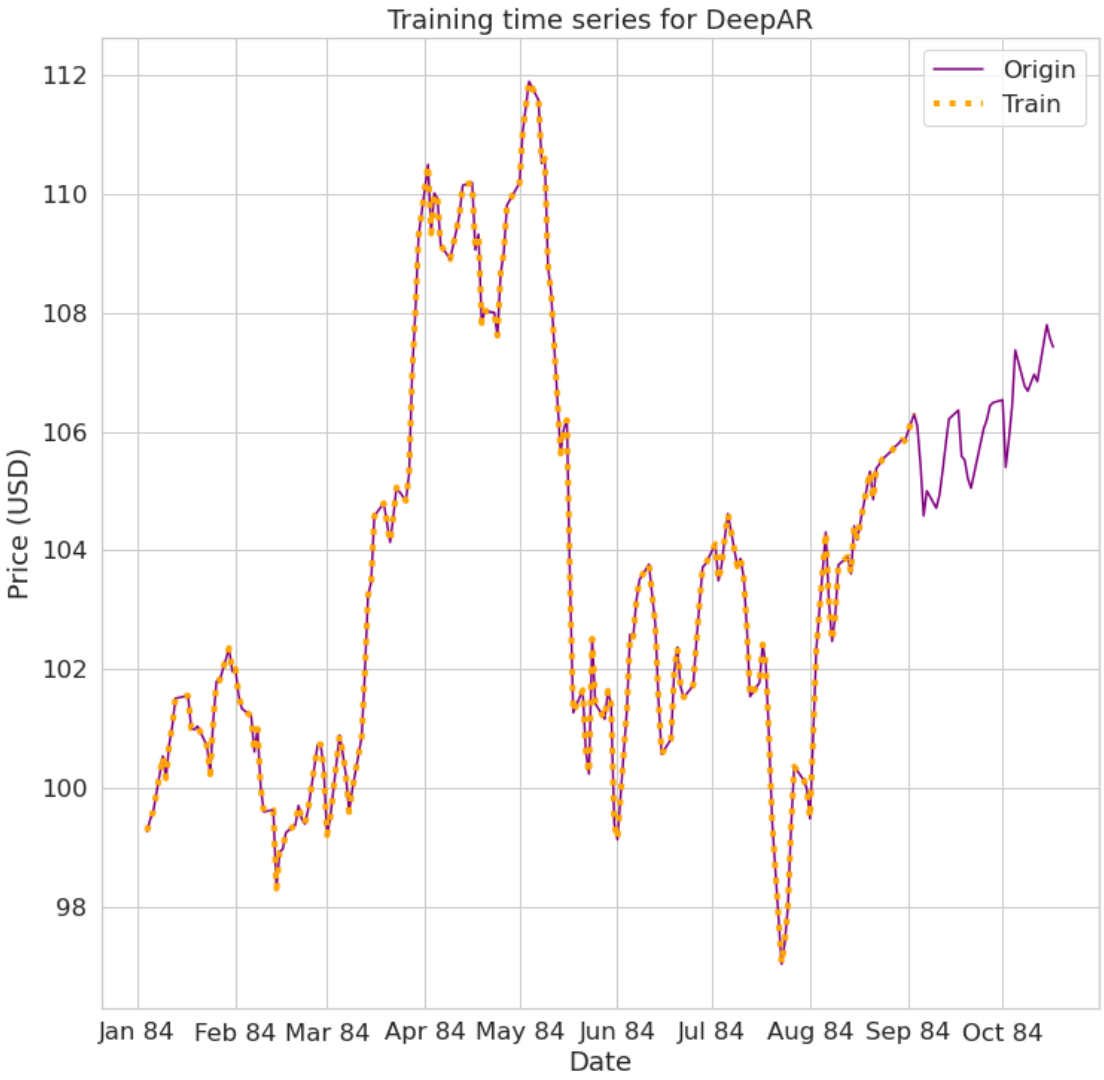

Figure 7: Time series for training and testing

After these time series are created, the *DeepAR* model requires to have a json as an input, with the shape:

$$\{'start' :< start\_date >,' target' :< time\_series >\}$$

The last step after creating these jsons, is to save them locally, and then upload them to S3.

3.1.2 PyTorch

For the PyTorch model, the first step is to select a single index. For my notebook, I used the NYA index, but everything that I did could be adapted to any index. I then use a *MinMax* scaler to have a standardized input, which could help my model train faster and produce better results. The output of the *MinMax* scaler is shown on figure 8.

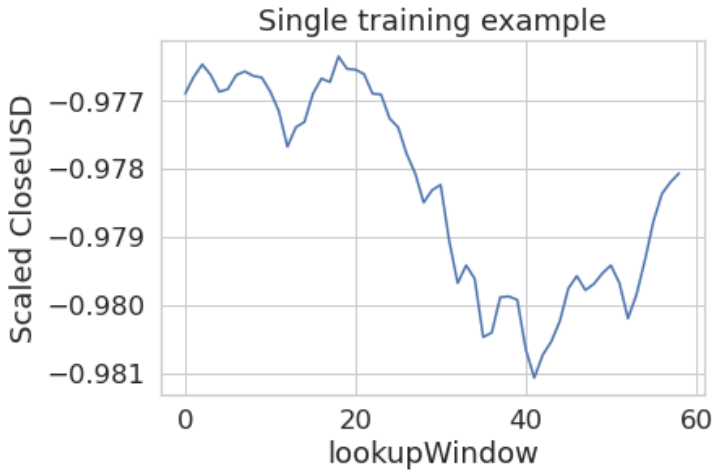

Figure 8: Output of the *MinMax* scaler

The next step is to format the data for my model. For this, I defined a *lookupWindow*, the model will take as input time series of length $lookupWindow - 1$ and will try to predict the next value. The function *split_data* does this operation of creating *x_train*, *y_train*, *x_test* and *y_test*. In addition, it creates a corresponding time index in order to have a nicer way to display the data.

Lastly, I create a csv file where the training data is stored and upload it to S3.

3.2 Implementation

3.2.1 DeepAR

For the implementation of DeepAR, I simply re-used the code from the Time Series Forecasting case study. Setting up the DeepAR model is quite straiforward as it simply consists of getting the DeepAR image uri from AWS, setting some hyperparameters, then training the model.

Since I realized quickly that the DeepAR model was not what I wanted to use for this project, I did not fine tune the hyperparameters too much.

3.2.2 PyTorch

After failing with the DeepAR model, I decided to make a model on which I had more control, using PyTorch. My first attempt was to recreate the model that was shown in the kaggle publication, which consists of two LSTM layers followed by two dense layers. As I tried to reuse the training code from the Plagiarism Detection project, I had some issues dealing with batches in my model.

My second choice was to find a model that was already written in PyTorch, preferably a model that worked with stock market data. I found this blog post that provided me with what I was looking for. From there, the main difficulty was to adapt this simple PyTorch model to the SageMaker architecture, I had to redo some parts of the train.py and predict.py. This model is simply composed of a single LSTM layer, followed by a single densely connected layer. The training does not use batches. The optimizer used for the gradient descent was the Adam optimizer, and I used the MSE loss for the criterion.

During this process, I was faced with numerous issues. The first one was that my predictor only outputted null values. The fix was simple, I had left the *np.round* of the Plagiarism Detector at the end of the predict.py file. After solving this issue, the output of my model was off by a linear transformation from the real output. To solve this one, I had to completely change the model I was using.

Overall, most of the challenges that I was faced with during this project was to understand how the LSTM layer functioned in PyTorch, especialy with batches (which I ended up not using), and adapting code that was designed for PyTorch to a SageMaker environment.

3.3 Refinement

I had 3 main development steps in this project, the first one was using the DeepAR algorithm, but it was not a perfect fit for this problem. The second was trying to adapt a Keras model to Pytorch, the issue there was that the result was off by a linear transformation, and I did not know how to correct it. The third and last one was to use a PyTorch model specifically made to study stock prices, first with batches, then without.

I did not fine tune the hyperparameters too much, I modified the learning rate, number of batches and number of epochs during the training process, according to the variations of the RMSE loss.

4 Results

4.1 Model Evaluation and Validation

4.1.1 DeepAR

The results I got from the DeepAR model were not consistent, which is yet another reason I decided to change model. In figure 9 the results look quite promising, however in figure 10 the actual result is not even in the 80% confidence intervals. The two graphs were done witht the exact same model, but with two different training jobs and testing samples. As I mentioned before, computing the RMSE for this model did not completely make sense so the graphical analysis is the only evaluation I did for this model

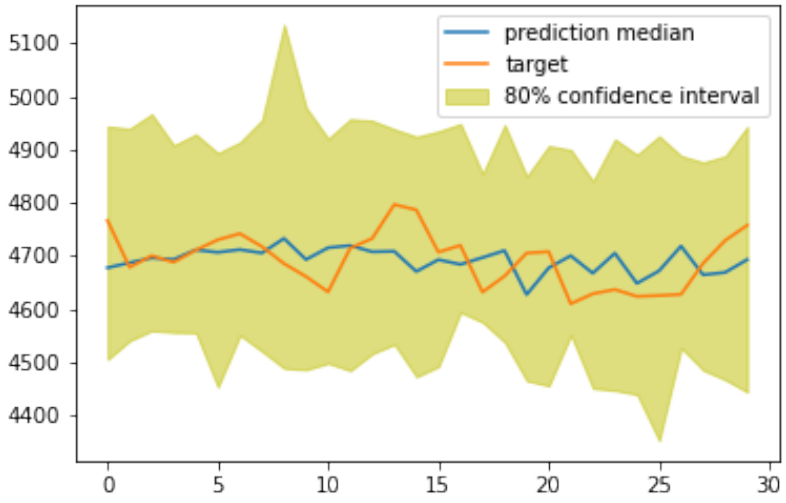

Figure 9: Accurate Predictions from the trained DeepAR model

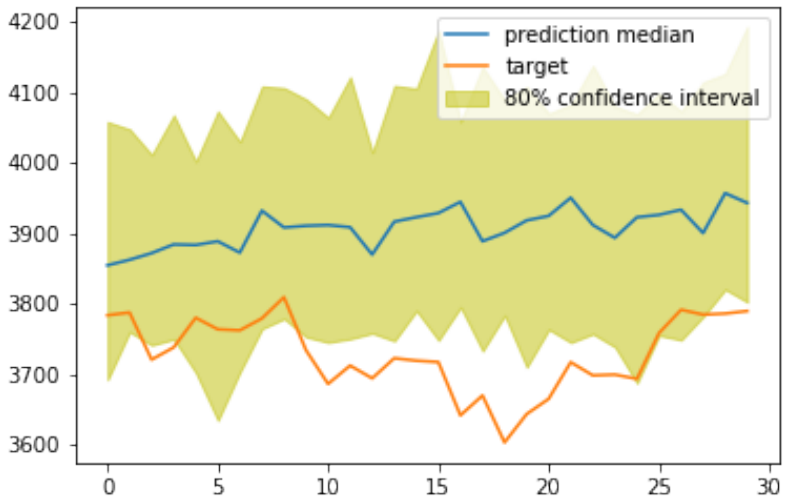

Figure 10: Inaccurate Predictions from the trained DeepAR model

4.1.2 PyTorch

For the PyTorch model, I was able to compute the RMSE with the trained data. The result was equal to 1140.40, which is more than the 129.75 provided in the kaggle notebook. However, if we plot the result as shown in figure 11, we can see that the prediction is really accurate for the first five years, which corresponds exactly to the time period studied in the kaggle notebook. If I restrict the RMSE to only take into consideration the first five years, the RMSE becomes 294.305, which is much closer to 129.75.

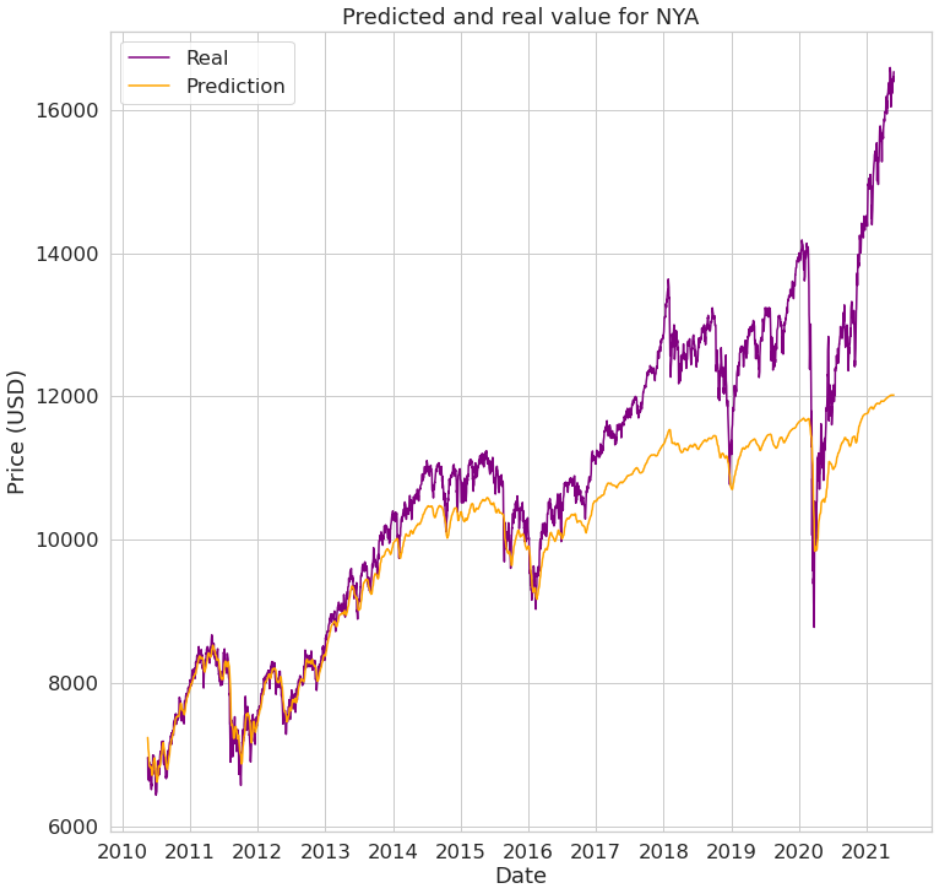

Figure 11: Predictions from the trained PyTorch model

I also tried to predict results for more than a single time period for the PyTorch model, by taking a training example, predicting the next value, and using this value to compute the next values of the stock price, and doing so recursively until I reached a certain date into the future. The result of this approach can be seen at figure 12, the results were inconclusive.

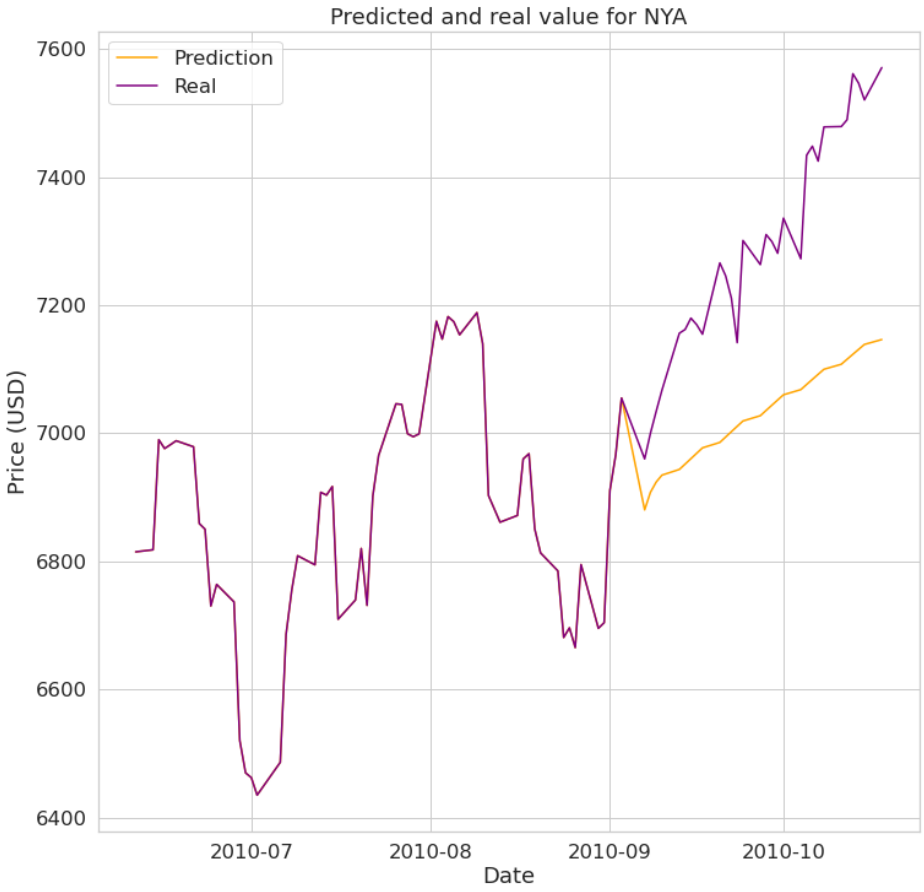

Figure 12: Predictions for multiple time intervals

4.2 Justification

Overall, the results from the PyTorch model are close to the reference for the RMSE and make sense from a graphical point of view. It is reasonable to say that the model could be used in a real-world scenario. What is quite significant is that the model manages to predict the variations in stock values. However, this model is not able to accurately predict values of the stock price multiple days in advance, and it is definitely something that could be improved. To achieve this goal, I think that the solution would be to create an entirely different model that is focused on predicting multiple future values.