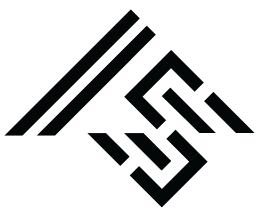
Predict stock prices

With Amazon SageMaker, create, train and deploy machine learning models, including Long Short-Term Memory (LSTM) networks, to predict the time series stock data of a publicly listed company.

Capstone Project: Machine Learning Engineer
Udacity Nanodegree | 2019

Suyash Tandon

Udacity Nanodegree | 2019



CHAPTER 1

Definition

1.1 Project Overview

Understanding market behavior, and trading and investments are complex time-dependent problems. One could rely on the historic data available in the form of tabulated stock prices and company performance metric, and use tool from statics and mathematical modeling to predict future trends[1]. Such quantitative analysis is essential for many investment firms, and hedge funds. While company performance, and related metrics affect the stock trades, other factors like the brand value, policies, news and events, all influence the stock prices. Therefore, all these factors involved in the prediction – physical factors vs. physhological, rational and irrational behaviour, etc., make share prices volatile and very difficult to predict with a high degree of accuracy. Time Series forecasting and modeling plays an important role in data analysis, and is a specialized branch of statistics used extensively in fields such as Econometrics and Operation Research. In the recent years, machine learning has shown promising results in feature extraction, pattern recognition, and parameter tuning. Therefore, machine learning algorithms are a good candidate for making stock price predictions[2].

In this project, the historical data of stock prices of a publicly listed company will be used. A mix of machine learning algorithms will be used to predict the future stock price of this company, starting with simple algorithms like linear regression, extreme gradient boosting, and then moving on to advanced techniques like long short-term memory. The core idea behind this project is to showcase the learnings from this nanodegree program by implementing these algorithms, creating estimators in Amazon SageMaker, and deploying the trained models to make future predictions on test datasets of the same company. Furthermore, learnings and insights from this project work will derive understanding that'll be useful for application in future research work.

1.2 Problem Statement

For a given time series data of stock prices of a publicly listed company, this project uses machine learning models to predict future closing price of the stock across a given period of time. This problem can be formulated as a linear regression problem, where the model learns from the historical data and predicts a value for future. The main goals of this project work are listed below.

[1] Brown, L. D. (1993). Earnings forecasting research: its implications for capital markets research. *International journal of forecasting*, 9(3), 295-320.

[2] Khan, Z. H., Alin, T. S., & Hussain, M. A. (2011). Price prediction of share market using artificial neural network (ANN). *International Journal of Computer Applications*, 22(2), 42-47.

1.2.1 Goals:

- Explore and pre-process stock price dataset,
- Implement Linear Regression & Extreme Gradient Boosting (XGBoost) as benchmark models,
- Implement a custom Long Short-Term Memory using Keras,
- Compile and deploy the models in Amazon SageMaker,
- Compare the predictions to evaluate model performances.

1.3 Evaluation Metric

Since we aim to predict future closing price of a stock across a give period of time in future by using the machine learning, the model performance for this project will be evaluated by measuring the mean squared difference between the predicted ($\widetilde{y}$) and actual values (y) of the target stock,

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \widetilde{y}_i)^2 \tag{1.1}$$

$$RMSE = (\frac{1}{n}\sum_{i=1}^{n}(y_i - \widetilde{y}_i)^2)^{1/2} = \sqrt{MSE} \tag{1.2}$$

and the delta between the performance of the benchmark models and the LSTM implementation.

The MSE and RMSE scores are highly beneficial for assessing the model performance for a given regression problem[3]. MSE works with squared error which assigns higher weights to larger errors for outliers. Therefore, if a model makes predictions with large errors it gets penalized and results in a higher MSE score. If the predictions are close to the actual values then MSE score is low and closer to 0. RMSE is similar to MSE for most cases and works with quadratic root of MSE. But its significance is more prominent for gradient-based methods.

$$\frac{\partial RMSE}{\partial \tilde{y}_i} = \frac{1}{2\sqrt{MSE}}\frac{\partial MSE}{\partial \tilde{y}_i} \tag{1.3}$$

The above equation means that traveling along MSE gradient is equivalent to traveling along RMSE gradient but with a different flowing rate and the flowing rate depends on MSE score itself. Therefore gradient-based methods some parameter like the learning rate needs to be tuned, and here RMSE is a better assessment tool.

[3] https://towardsdatascience.com/how-to-select-the-right-evaluation-metric-for-machine-learning-models-part-1-regrression-metrics-3606e25beae0

CHAPTER 2

Analysis

2.1 Datasets

Stock behavior for different companies and industries is different. Hence, this project seeks to build an end-to-end framework, that learns the trends from a given dataset and then gives prediction for some future time. To demonstrate this framework, dataset of the technology giant, Alphabet Inc. will be pulled from the online repository hosted at Yahoo Finance[1]. For ease of reproducibility and reusability the dataset from Yahoo Finance is pulled using the python package, `yfinance()`[2].

`yfinance()` takes three user inputs:

1. *symbol*: The symbol or the ticker that identifies the dataset of the company, e.g. `GOOGL` for Alphabet Inc. (Google).
2. *start*: The start date from which the data needs to be extracted. Prescribed in YYYY-MM-DD or UNIX Epoch Time formats.
3. *end*: The end date till when the data needs to be included. Prescribed in YYYY-MM-DD or UNIX Epoch Time formats.

	Date	Open	High	Low	Close	Adj_Close	Volume
0	2009-11-03	265.270264	269.019012	264.414429	268.913910	268.913910	4755600
1	2009-11-04	270.670685	273.023010	268.478485	270.435425	270.435425	4660700
2	2009-11-05	272.017029	275.160156	271.601593	274.599609	274.599609	3691700
3	2009-11-06	274.134125	276.166168	273.023010	275.825836	275.825836	3649700
4	2009-11-09	278.002991	281.571564	277.392395	281.536530	281.536530	5294500

Figure 2.1: Dataset of historical Stock Prices of Alphabet Inc. taken from Yahoo Finance.

The dataset used in this project is of the technology giant, Alphabet Inc. (*symbol*: GOOGL) and from November 1, 2009 to November 1, 2019. The dataset downloaded from Yahoo Finance using `yfinance()` looks as shown in figure 2.1. The dataset is chronologically organized in ascending order

[1] https://finance.yahoo.com
[2] https://aroussi.com/post/python-yahoo-finance

and contains information in the following columns - *Date, Open, High, Low, Close, Adj_ Close*, and *Volume*. Each of these columns can be individually treated as separate features, thus, the historical stock price dataset has a total of 7 features.

The downloaded data is saved as a `pandas`[3] dataframe. The `describe()` attribute of this dataframe gives information on the total number of entries in each column, mean, standard, deviation, and other statistics. The statistics of the dataset of Alphabet Inc. used in this project is shown below in figure 2.2.

	Open	High	Low	Close	Adj_Close	Volume
count	2516.000000	2516.000000	2516.000000	2516.000000	2516.000000	2.516000e+03
mean	635.397300	640.689145	629.668080	635.334439	635.334439	3.427869e+06
std	320.135523	322.995625	317.249482	320.265958	320.265958	2.672654e+06
min	219.374374	221.361359	217.032028	218.253250	218.253250	5.206000e+05
25%	311.961975	314.191681	310.307800	312.367371	312.367371	1.583525e+06
50%	561.025024	565.725006	556.744995	561.229980	561.229980	2.561600e+06
75%	924.400009	934.392487	919.444992	924.935013	924.935013	4.489575e+06
max	1289.119995	1299.239990	1272.310059	1296.199951	1296.199951	2.961990e+07

Figure 2.2: Statistics of the dataset of Alphabet Inc. taken from Yahoo Finance.

	Item	Open	Close	Volume
0	0	265.270264	268.913910	4755600
1	1	270.670685	270.435425	4660700
2	2	272.017029	274.599609	3691700
3	3	274.134125	275.825836	3649700
4	4	278.002991	281.536530	5294500

Figure 2.3: Modified dataset with reduced feature set.

It was observed that all the columns have 2516 entries. Further observation concluded that the dataset did not contain any abnormal entries such as empty fields, negative values or nan. Furthermore, in the tabulated data in figures 2.1 & 2.2, the *Close* and *Adj_ Close* columns have identical information. This suggests that Yahoo Finance already adjusts the closing price of the stock on the day for us, and therefore, one of these columns can be neglected. Of the remaining features, the opening price (*Open*) and the closing price (*Close*) can provide information whether on a particular day the stock saw profit or loss. Additionally, *Volume* of the share is important as a rising market should see rise in stock volume, while a decreasing stock volume depicts lack of interest. A price change in a larger volume is a stronger signal that something in the stock fundamentally changed. The *High* and *Low* features can be useful to relate the swings in stock prices to that of current affairs and news, etc. However, in this basic implementation, these features are neglected as *Volume* already contains effects of such events. Thus, in this project, the dataset will be modified to contain information on just *Date, Open, Close*, and *Volume*. Since, the date is

[3]https://pandas.pydata.org

organized in ascending order, it is replaced by *Item* number. The modified dataset with reduced feature set looks like that shown in figure 2.3.

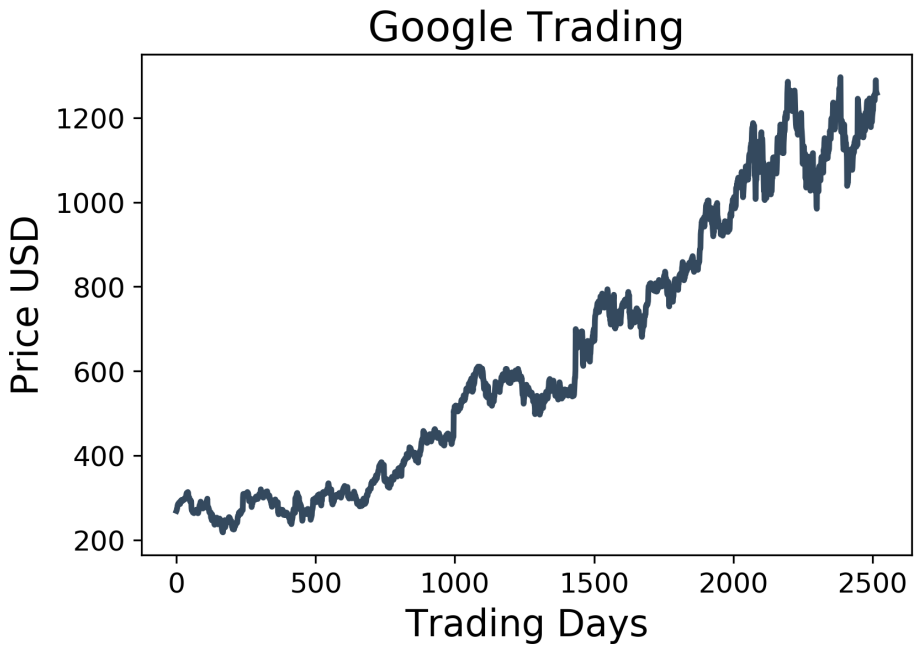


Figure 2.4: Variation of the closing (*Close*) stock price in USD with number of trading days in the dataset for Alphabet Inc.

An exploratory visualization of the dataset is achieved through the `matplotlib`[4] library. Figure 2.4 shows variation of the closing stock price (*Close*) in USD over the entire period of trading days (dates) in the dataset shown previously. The plot shows continuous rise of Alphabet Inc. after the Global Financial Crisis on 2008-2009. The large fluctuations in the last 500 trading days aligns well with the "stock shock" and high volatility seen in 2018[5].

2.2 Algorithms

From figure 2.4, the problem of stock price prediction, where the objective is to predict the closing price of a stock for a future date based on historic data, can be treated as a simple regression problem. The variation in the stock price can be captured by a function $\mathcal{F}$ that maps the behavior of the stock in time to its closing price. Thus,

$$Close = \mathcal{F}(time) \tag{2.1}$$

At a first glance, based on above principle, the stock price prediction can work with historical information on just *Date* or Item and *Close* price associated with that day. But, this can be quite misleading. As seen in figure 2.4 and 2.5, a general regression with closing price can capture the general upward trend seen in Alphabet Inc.'s growth, but it cannot explain the large fluctuations seen between $2000 - 2500$ trading days. This is because, this volatility was been driven by signs of a global economic slowdown, concerns about monetary policy, political dysfunction, inflation fears and worries about increased regulation of the technology sector[6]. It is hard to capture such

[4]https://matplotlib.org
[5]https://www.cnn.com/2018/12/31/investing/dow-stock-market-today/index.html
[6]https://www.cnn.com/2018/12/31/investing/dow-stock-market-today/index.html

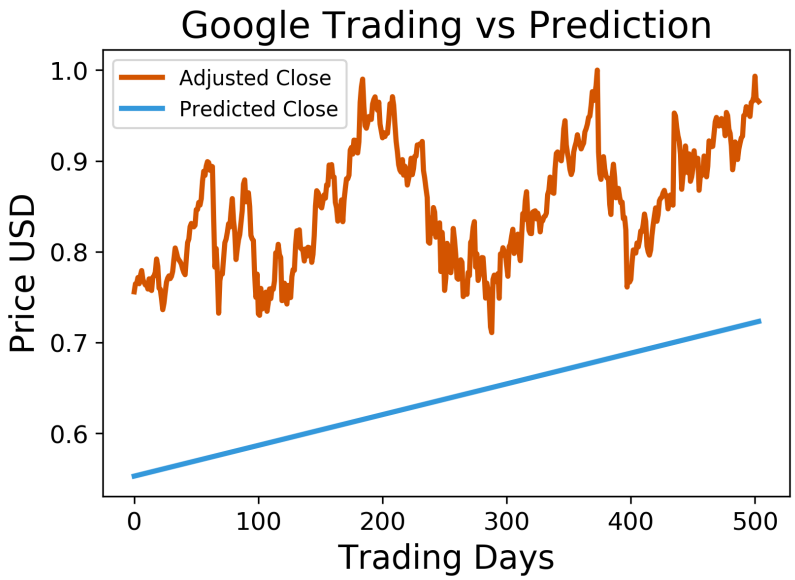


Figure 2.5: Predictions of Alphabet Inc.'s closing stock price when modeled as a simple regression problem of equation 2.1 can provide a general upward trend but cannot predict the fluctuations.

rational/irrational effects from the closing price alone. This is where the other features, *Open*, and *Volume* play an important part – as previously discussed volume suggests interest of people in a particular stock which is related to events, news, etc., and difference between open and close suggests how well a stock is performing on a daily basis.

Therefore, the regression problem of stock price prediction, turns into a multivariate functional form, where *Close* price now depends on a set of features.

$$Close = \mathcal{F}(\text{data}[\text{'Item', 'Open', 'Volume'}]) \tag{2.2}$$

The stock pice prediction using machine learning has become one of the standard, de-facto problems to test models on time-series data. There are many implementations for this problem[7]. A variant of the recurrent neural networks (RNN) is the Long Short-Term Memory (LSTM) network that specializes in learning long term dependencies [8].

The repeating modules of the standard RNN have a simple structure, such as a single tanh layer. However, the repeating modules in a LSTM have a different design with four layers interacting in a special way[9]. The key to LSTMs is the cell state $\mathcal{C}_t$. The LSTMs have the ability to remove or add information to the cell state, regulated by structures called gates – a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation. The sigmoid layer outputs numbers between 0 which means let nothing through and 1 that means let everything through!. The three important gates in a LSTM layer are:

- *Forget Gate Layer*: Decides what information to throw away from the cell state. It looks at the output from previous cell h_{t-1} and some input to current cell x_t, and outputs a number between 0 and 1. The operation can be defined as,

$$f_t = \sigma(W_f.[h_{t-1}, x_t] + b_f)$$

[7]https://www.analyticsvidhya.com/blog/2018/10/predicting-stock-price-machine-learningnd-deep-learning-techniques-python/

[8]Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8), 1735-1780.

[9]https://colah.github.io/posts/2015-08-Understanding-LSTMs/

- *Input Gate Layer*: Decides what new information to store in the cell state. First, a sigmoid layer decides which values to update. Next, a tanh layer creates a vector of new candidate values, that could be added to the state. To update the old cell state, $\mathcal{C}_{t-1}$, into the new cell state $\mathcal{C}_t$, multiply the old state by f_t forgetting the things as decided earlier. Then, add the new input $i * \widetilde{\mathcal{C}}_t$ scaled by how much to update each state value. Thus, new state $\mathcal{C}_t$ is given by,

$$\mathcal{C}_t = f_t * \mathcal{C}_{t-1} + i * \widetilde{\mathcal{C}}_t$$

- *Output Gate Layer*: Decides what to output. This output is based on the cell state, but will be a filtered version. First, a sigmoid layer decides what parts of the cell state to output. Then, the cell state is put through tanh (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

$$o_t = \sigma(W_o.[h_{t-1},\, x_t] + b_o)$$
$$h_t = o_t * tanh(\mathcal{C}_t)$$

Due to this property, the LSTMs appear to be good candidates for time-series data prediction[10]. The project will be implemented in a Jupyter Notebook using Amazon SageMaker. Using Keras implementation of the Tensor Flow library, the solution will use the LSTM neural net model. The trained model will be deployed and used to make future predictions.

2.3 Benchmark Models

This project implementation will use the Linear Regression and the Extreme Gradient Boosting (XGBoost) methods as benchmarks.

1. Linear Regression: The linear regression is a basic Machine Learning model that returns an equation which determines the relationship between the independent variables and the dependent variable. The equation for linear regression can be written as:

$$\widetilde{y} = x_1\theta_1 + x_2\theta_2 + \cdots + x_n\theta_n. \tag{2.3}$$

   Here, x_i are the independent variables and θ_i are the respective coefficients for $i = 1, 2, \ldots, n$. If we set y or the 'target' to be *Close* price of the stock that needs to be predicted, and choose the input features as *Item, Open* and *Volume*, then the linear algorithm learns a linear function and maps $\mathbf{x}$ to an approximation of the label y,

$$\widetilde{y} = \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 \tag{2.4}$$

   The Amazon SageMaker linear learner algorithm provides a solution for both classification and regression problems[11].

2. XGBoost: Gradient boosting is a process to convert weak learners to strong learners, in an iterative fashion. The name XGBoost refers to the engineering goal to push the limit of computational resources for boosted tree algorithms[12]. XGBoost has proven to be a very

[10] Bao, W., Yue, J., & Rao, Y. (2017). A deep learning framework for financial time series using stacked autoencoders and long short term memory. *PloS one*, 12(7), e0180944.

[11] (https://docs.aws.amazon.com/sagemaker/latest/dg/linear-learner.html)

[12] Chen, T., & Guestrin, C. (2016, August). Xgboost: A scalable tree boosting system. *In Proceedings of the 22nd ACM SiGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 785-794). ACM.

powerful machine learning technique and is usually the go-to algorithm in many Machine Learning competitions. It builds the model in a stage-wise fashion, and it generalizes them by allowing optimization of an arbitrary differentiable loss function. Say for instance that the goal is to "teach" a model $\mathcal{F}$ to predict values of the form $\widetilde{y} = \mathcal{F}(x)$.

At each stage m, $1 \leq m \leq M$, of gradient boosting, it may be assumed that there is some imperfect model $\mathcal{F}_m$. The gradient boosting algorithm improves on $\mathcal{F}_m$ by constructing a new model that adds an estimator h to provide a better model. To find h, the gradient boosting solution starts with the observation that a perfect h would imply:

$$\mathcal{F}_{m+1}(x) = \mathcal{F}_m(x) + h(x) = y \tag{2.5}$$

or equivalently,

$$h(x) = y - \mathcal{F}_m(x) \tag{2.6}$$

Therefore, gradient boosting will fit h to the residual $y - \mathcal{F}_m(x)$. Amazon SageMaker provides an in-built algorithm for XGBoost[13].

Both these models are extensively used for supervised learning, where given an input X, the method gives a prediction $\widetilde{y}$, and the model can be easily described as

$$\widetilde{y} = \mathcal{F}(X).$$

It'd therefore, be necessary to adapt the time-series dataset for these regression models such that the previous time step becomes the input and the next time step is the output of the supervised learning[14]. In addition, the order between the observations must be preserved, and since there is no previous value that can be used to predict the first value in the sequence, this row cannot be used.

[13](https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost.html)
[14]https://machinelearningmastery.com/time-series-forecasting-supervised-learning/

CHAPTER 3

Methodology

3.1 Helper Functions

Pseudocode of the helper functions that were used for data preparation for different machine learning models in this project are listed underneath. The source code can be found in the `processData.py` script in the project repository[1].

remove_data()

This function is used to modify the downloaded dataset to remove the unwanted features and returns the dataset with columns [*'Item', 'Open', 'Close', 'Volume'*] as shown in figure 2.3.

Algorithm 1 remove_data(data)

```
item, open, close, volume = [ ]                          ▷ Define empty lists
couter = 0
for i in range(0, len(data)) do
    item.append(counter), open.append(data['Open'][i])
    close.append(data['Close'][i]), volume.append(data['Volume'][i])
    counter += 1
stocks = pandas.DataFrame()
stocks['Item'] ← item
stocks['Open'] ← open
stocks['Close'] ← close
stocks['Volume'] ← volume
return stocks
```

normalize_data()

Often the data for different features can be scaled differently within the same dataset. When machine learning model is trained on such datasets, the weights of the trained model will be set such that the feature with large variation in values appears more or less dominant that other features. This in some cases can produce erroneous predictions. Therefore, feature scaling to standardize the dataset

[1] https://github.com/suyashtn/machine-learing-projects/tree/master/Predict_Stock_Prices

between common values in important. Here we normalize our dataset such that the values are standardized between 0 and 1. A snippet of the dataset after normalization is shown in figure 3.1.

Algorithm 2 normalize_data(data)

```
from sklearn.preprocessing import MinMaxScaler                ▷ Import library
scaler = MinMaxScaler()                                       ▷ Initialize a scaler
numerical = ['Open','Close','Volume']
data[numerical] = scaler.fit_transform(data[numerical])
return data
```

	Item	Open	Close	Volume
0	0	265.270264	268.913910	4755600
1	1	270.670685	270.435425	4660700
2	2	272.017029	274.599609	3691700
3	3	274.134125	275.825836	3649700
4	4	278.002991	281.536530	5294500

(a) Modified dataset after `remove_data()`

	Item	Open	Close	Volume
0	0	0.042904	0.046997	0.145536
1	1	0.047952	0.048409	0.142275
2	2	0.049210	0.052272	0.108975
3	3	0.051190	0.053409	0.107532
4	4	0.054806	0.058707	0.164055

(b) Normalized dataset after `normalize_data()`

Figure 3.1: Feature scaling of the dataset and standardizing the values between 0 and 1.

train_test_split()

In most machine learning techniques, the dataset is first split into training and test parts. The model is initially trained on the training dataset, and then it is tested on the test data. This ensures more reliable model development and assessment.

Algorithm 3 train_test_split()(stocks, train_frac, **kwargs)

```
val_frac ← kwargs.get('val_frac', None)                       ▷ Extract validation size if provided
train_size = int(stock.shape[0] * train_frac)                 ▷ convert stock to matrix form.
selector = [x for x in range(stock.shape[1]) if x != 2]       ▷ skips column ['Close']
if NOT val_frac is None then
    val_size = int(train_size * val_frac)
    X_train ← stock[:(train_size-val_size), selector]         ▷ train feature set
    y_train ← stock[:(train_size-val_size), 2]                ▷ train label set
    X_val ← stock[(train_size-val_size):train_size, selector] ▷ validation feature set
    y_val ← stock[(train_size-val_size):train_size, 2]        ▷ validation label set
    X_test ← stock[train_size:, selector]                     ▷ test feature set
    y_test ← stock[train_size:, 2]                            ▷ test label set
  return X_train, X_val, X_test, y_train, y_val, y_test
else
                                  ▷ Evaluate X_train, X_test, y_train, y_test similarly
   return X_train, X_test, y_train, y_test
```

3.2 Benchmark: LinearLearner

Data Preparation

In the LinearLearner case the reduced and normalized dataset as shown in figure 3.1, is split using the `train_test_split()` method described in algorithm 3. The `train_frac` is 0.8 such that 80% of the dataset is used for training while the remaining 20% is used as test data. The dataset in practice should also be split up into a validation set, that can be used to evaluate the model parameters and tune any hyperparameters if needed. This process is skipped here.

The Amazon SageMaker LinearLearner model requires the training dataset to be a Recordset object. This is done through the `linear.record_set()` that constructs a RecordSet object from *ndarray* arrays. In addition, convert the train and test datasets from 'float64' to 'float32' format for `record_set()` to work properly[2].

Implementation

To setup the in-built LinearLearner estimator in Amazon Sagemaker[3], first a SageMaker session needs to be created and associated with an execution role. Then a file system where all the output data is stored, called S3 bucket needs to be initialized. This is shown below

Algorithm 4 set-up SageMaker session, role and bucket

```
import boto3, sagemaker                          ▷ Import SageMaker and S3 libraries
from sagemaker import get_execution_role
sagemaker_session = sagemaker.Session()
role = sagemaker.get_execution_role()
bucket = sagemaker_session.default_bucket()      ▷ assign a default S3 bucket
```

Algorithm 5 set-up LinearLearner estimator

```
from sagemaker import LinearLearner                              ▷ import Linear Learner
prefix = 'stockPrices'                          ▷ specify a folder name to store data in S3
output_path = 's3://{}/{}/output'.format(bucket, prefix)
linear = LinearLearner(role=role,          ▷ instantiate LinearLearner train_instance_count=1,
            train_instance_type='ml.c4.xlarge',
            predictor_type='regressor',
            loss='squared_loss',
            output_path=output_path,
            sagemaker_session=sagemaker_session,
            epochs=15)
```

The LinearLearner estimator is set up in Amazon SageMaker that provides an easy workflow to train, and deploy the estimator. Refer the notebook for this project[4].

1. The `fit()` method is used to train the estimator,
2. After the estimator is trained, it can be deployed with the `deploy()` method
3. The deployed endpoint can be invoked to make predictions using the `predict()`

[2] https://sagemaker.readthedocs.io/en/stable/linear_learner.html#sagemaker.LinearLearner.record_set
[3] https://docs.aws.amazon.com/sagemaker/latest/dg/linear-learner.html
[4] https://github.com/suyashtn/machine-learing-projects/blob/master/Predict_Stock_Prices/Stock_Price_Prediction.ipynb

Refinement

The predictions on the test data are evaluated with a mean squared error (MSE), and root mean squared error (RMSE) metrics,

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \widetilde{y}_i)^2 \tag{3.1}$$

$$RMSE = (\frac{1}{n}\sum_{i=1}^{n}(y_i - \widetilde{y}_i)^2)^{1/2} = \sqrt{MSE} \tag{3.2}$$

The lower the MSE and RMSE, the better the performance of the trained model is. If model performance appears to be underperforming, some parameters need to be refined. Possible options for refinement include,

1. Resizing the training / testing dataset
2. Changing the loss objective function
3. Updating the number of epochs

3.3 Benchmark: XGBoost

Data Preparation

The `train_test_split()` helper function is used to split the modified and normalized dataset into train, test and validation sets with the following split:

- First the normalized dataset is split into train/test sets using `train_frac` 0.8 so that 80% of the data is for training while 20% is for testing.
- Then split the training data with `val_frac` 0.2 such that 20% of the training data is reserved for validation.

Store the train, validation and test data in `csv` format and then upload to S3.

Implementation

Algorithm 6 set-up XGBoost estimator

```
from sagemaker.amazon.amazon_estimator import get_image_uri
container = get_image_uri(sagemaker_session.boto_region_name, 'xgboost', '0.90-1')
xgb = sagemaker.estimator.Estimator(container,
            role=role,
             train_instance_count=1,
             train_instance_type='ml.m4.medium',
             output_path=output_path,
             sagemaker_session=sagemaker_session)
```

A built-in XGBoost algorithm[5] in Amazon SageMaker is used. It is necessary to provide the location of the container that contains the training code. This container is provided by Amazon.

[5] https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost.html

SageMaker provides a useful utility method called `get_image_uri` that constructs the required container image name. Before beginning the training, some model specific hyperparameters must be set. XGBoost algorithm has quite a few hyperparameters[6].

Algorithm 7 set-up XGBoost hyperparameters

```
xgb.set_hyperparameters(max_depth=5,
            eta=0.2,
            gamma=4,
            alpha=0,
            min_child_weight=6,
            subsample=0.8,
            objective='reg:linear',
            early_stopping_rounds=10,
            num_round=200)
```

SageMaker allows to launch a hyperparameter tuning[7] job that trains parallel jobs and finds the best model parameters. More details can be found in Part III of the notebook for this project[8]. Similar to the LinearLearner case,

1. The `fit()` method is used to train the tuned estimator,
2. After the estimator is tuned and trained, it needs to be evaluated. SageMaker allows a batch transformation functionality that can be used to test the predictions against actual values. If the model performance is below expectations, the hyperparameters can be tuned even further before final deployment.
3. Once the model is deployed, the endpoint can be invoked to make predictions using the `predict()`

Refinement

Algorithm 8 set-up XGBoost hyperparameters tuning jobs

```
from sagemaker.tuner import HyperparameterTuner
xgb_hyperparameter_tuner = HyperparameterTuner(estimator = xgb, ▷ The estimator object
                objective_metric_name = 'validation:rmse',        ▷ Metric to compare models
                objective_type = 'Minimize',
                max_jobs = 20,                                    ▷ number of models
                max_parallel_jobs = 3,            ▷ The number of models to train in parallel
                hyperparameter_ranges = {'max_depth': IntegerParameter(3, 12),
                                         'eta' : ContinuousParameter(0.05, 0.5),
                                         'min_child_weight': IntegerParameter(2, 8),
                                         'subsample': ContinuousParameter(0.5, 0.9),
                                         'gamma': ContinuousParameter(0, 10), })
```

XGboost relies on many hyperparameters some of which are listed in algorithm 7. To get accurate predictions, it is essential that these hyperparameters be tuned well for a given problem.

[6] https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost_hyperparameters.html
[7] https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/
[8] https://github.com/suyashtn/machine-learing-projects/blob/master/Predict_Stock_Prices/Stock_Price_Prediction.ipynb

SageMaker allows launching a tuning job (algorithm 9) for these parameters where many models can be trained in parallel, and the best model combination can be retrieved. This is analogous to doing a grid search but on a larger-scale.

The best tuned model is tested with batch transform to make predictions on test data that are compared to the actual values and MSE and RMSE scores are calculated. If the score is low and model performance is below expectations, the hyperparameters are tuned again using algorithm 9.

3.4 Long Short-Term Memory (LSTM) Networks

Data Preparation

As a preliminary step, the normalized dataset is first split into training, validation and testing fractions similar to the XGBoost case. However, these datasets need to be further modified according to the special requirements of LSTMs. The input data for the LSTM should have a 3D structure of the kind: (batch_size, time_steps, seq_length),

- batch_size: The number of samples of dataset.
- time_step: The number of time steps per sample of the data.
- seq_length: The number of inputs, or features.

A special helper function `create_dataset_lstm()` is used,

Algorithm 9 create_datasets_lstm(stocks)

1: Set history_size ▷ number of days of historical data in each sample
2: Set target_size ▷ number of days for making prediction
3: train_test_split(stocks) ▷ invoke to get preliminary train/test split
4: train ← pandas.concate(X_train, y_train) ▷ similar steps for val, test data
5: X_train, y_train = series_to_supervised(train, history_size, target_size) ▷ convert the train data to 3D structure.
6: X_val, X_test, y_val, y_test ▷ Similarly convert test and validate data

The implementation of `series_to_supervised()` can be found in the `processData.py` script in the project repository. Once the dataset in the required format is created, the LSTM model can be trained and evaluated.

Implementation

SageMaker allows custom model development and training. For building an LSTM model, the Keras library with a high-level TensorFlow API is used. The model definition is provided in the `lstm_model.py` script in our project repository. The following steps allow building, training and deployment of a custom LSTM model in SageMaker.

1. Load the model by calling the `build_lstm_model()` function that defines the custom model in `lstm_model.py`.
2. Compile the loaded model through the `model.compile()` method with `loss = mean_squared_error`, and `optimizer = 'adam'`.
3. `model.summary()` prints the structure of the custom model. The custom LSTM model defined in this project is shown in figure 3.2. It has 3 layers, with LSTM having 200 nodes and a dense layer with linear activation. All the layers have a dropout to reduce over-fitting.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, None, 14)          1008
_________________________________________________________________
dropout (Dropout)            (None, None, 14)          0
_________________________________________________________________
lstm_1 (LSTM)                (None, 200)               172000
_________________________________________________________________
dropout_1 (Dropout)          (None, 200)               0
_________________________________________________________________
dense (Dense)                (None, 1)                 201
_________________________________________________________________
activation (Activation)      (None, 1)                 0
=================================================================
Total params: 173,209
Trainable params: 173,209
Non-trainable params: 0
_________________________________________________________________
```

Figure 3.2: A custom LSTM network in Keras using TensorFlow API in Amazon SageMaker.

4. Train the compiled model locally through `model.train()`, using the training data as input.

```
1: lstm.fit(X_train,                                          ▷ features for training
2:          y_train,                                          ▷ training labels
3:          epochs=30,                                        ▷ No. of iterations of training
4:          batch_size = 128)        ▷ No. of samples to inject as input to the network at a time
```

5. The trained model can be evaluated on the validation data to return a MSE and RMSE score.

```
1: valScore = lstm.evaluate(X_val, y_val, verbose=0)
2: print('Validation Score: %.8f MSE (%.8f RMSE)' % (valScore, math.sqrt(valScore)))
```

6. If the performance of the trained model is satisfactory, then it can be deployed in SageMaker. To deploy a custom model, it needs to be exported in ProtoBuf format. For that purpose
   (a) Export the model structure to JSON file and save the weights to h5 file.
   (b) Load the model JSON file and model weights obtained from previous step.
   (c) Export the Keras model to the TensorFlow ProtoBuf format so that it can be used for deployment.
   (d) Tar the entire export directory and upload to S3
   (e) Create an LSTM model setimator by importing the exported model from S3.
   (f) The details and implementation of above steps can be found in the project notebook[9]
7. The estimator imported in the previous step can be deployed using the standard `deploy()` in SageMaker, and the predictions can be made with `predict()`.

[9]https://github.com/suyashtn/machine-learing-projects/blob/master/Predict_Stock_Prices/Stock_Price_Prediction.ipynb

Refinement

The LSTM network implemented in this project can be improved by fine tuning the following parameters:

1. Number of hidden nodes,
2. Dropout probability in each layer,
3. Modifying the number of training iterations (epochs),
4. Changing the `history_size` and `target_size`.

The model performance is evaluated by calculating the MSE and RMSE of predicted values.

3.5 Project Design

A high-level overview of the project workflow is described below.

1. Setup the project resources
    - setup Notebook instance on Amazon SageMaker,
    - import necessary libraries,
    - download the dataset,
    - create a role, setup a session, create a S3 bucket.
2. Prepare dataset:
    - load the dataset in pandas framework,
    - normalize the data,
    - split the dataset for training, validation, and testing.
3. Implement benchmark models
    - Setup LinearRegressor model,
    - Setup XGBoost model.
    - Tune hyperparameters.
4. Implement LSTM model
    - Setup LSTM model with Keras utilizing parameters from benchmark model,
    - Tune hyperparameters.
5. Evaluate and compare model performance
6. Document and visualize results

CHAPTER 4

Results

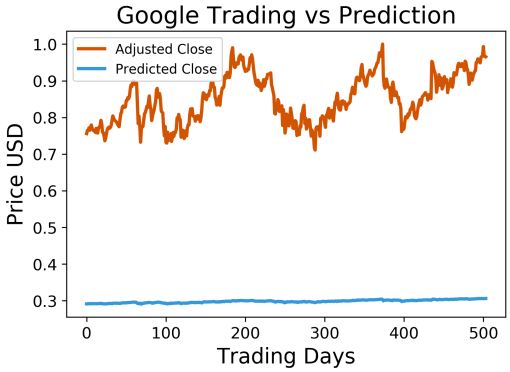
4.1 Predictions Using Benchmark Models

LinearLearner

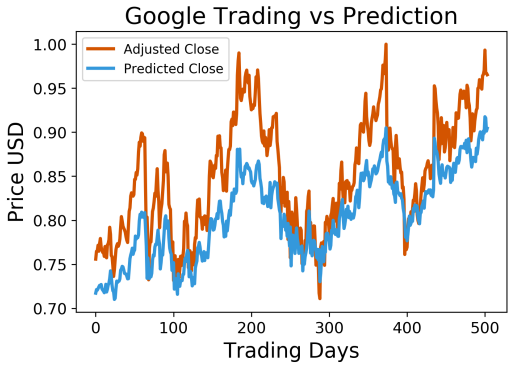
- `train_test_split()` was used to split the normalized data into 80% training and 20% testing. Features are ['Item', 'Open','Volume'] whereas the labels are ['Close'] column of the dataset.

Table 4.1: LinearLearner model parameters.

Model parameters	**Preliminary**	**Improved**
X_train; y_train	(2012, 3); (2012,)	(same)
X_test; y_test	(504, 3); (504,)	(same)
loss	squared_loss	(same)
epochs	1	15
Model performance	$MSE = 0.30790938$ (0.55489583 $RMSE$)	$MSE = 0.00276604$ (0.05259319 $RMSE$)



(a) Preliminary model with epoch = 1.



(b) Improved model with epoch = 15.

Figure 4.1: Performance of LinearLearner model on test data.

XGBoost

- `train_test_split()` was used to split the normalized data into 80% training and 20% testing. The training data was further split such that 20% of it was used for validation. Features are ['Item', 'Open','Volume'] whereas the labels are ['Close'] column of the dataset.

Table 4.2: XGBoost model parameters.

Model parameters	**Preliminary**	**Tuned Hyperparameters**
X_train; y_train	(1610, 3); (1610,)	–
X_val; y_val	(402, 3); (402,)	–
X_test; y_test	(504, 3); (504,)	–
max_depth	5	9
eta	0.2	0.0543
gamma	4	0
alpha	0	–
min_child_weight	6	2
subsample	0.8	0.8781
objective	'reg:linear'	–
early_stopping_rounds	10	–
num_round	200	–
Model performance		$MSE = 0.11284369$ (0.33592215 $RMSE$)

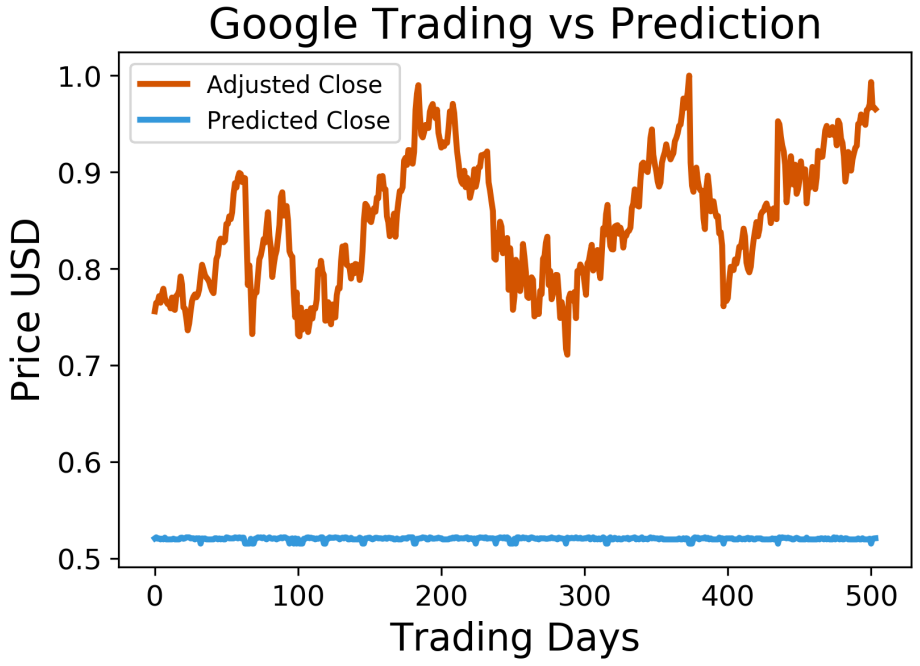


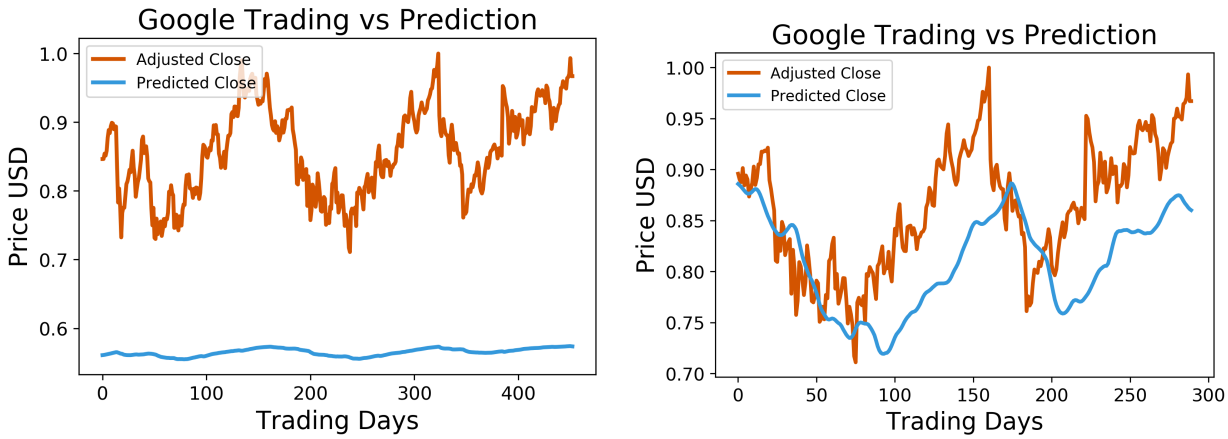
Figure 4.2: Stock price predictions with XGBoost model.

4.2 Predictions with Long Short-Term Memory (LSTM) Network

- `create_datasets_lstm()` was used which invokes `train_test_split()` to initially split the normalized data into 80% training and 20% testing. The training data was further split such that 20% of it was used for validation. The train, validation and test data were further modified into the 3D structure needed for LSTM.

Table 4.3: Custom LSTM model parameters.

Model parameters	**Preliminary**	**Improved**
history_size	50	200
target_size	1	14
X_train; y_train	(1559, 50, 3); (1559, 1)	(1396, 200, 3); (1396, 14)
X_val; y_val	(351, 50, 3); (351, 1)	(188, 200, 3); (188, 14)
X_test; y_test	(453, 50, 3); (453, 1)	(290, 200, 3); (290, 14)
dropout	0	0.2
epochs	1	30
batch_size	100	128
Model performance	$MSE = 0.00966046$ (0.09828765 $RMSE$)	$MSE = 0.00677487$ (0.08230960 $RMSE$)



(a) Preliminary model.

(b) Improved model.

Figure 4.3: Performance of custom LSTM model on test data.

4.3 Justification

The custom LSTM model is capable of modeling the historical stock price data and can make future predictions on the closing price of the stock during a desired time frame. The implementation in this project shows that with right parameter tuning, the LSTM network can perform on the same lines as the benchmark models. The improved LSTM model in figure 4.3(b) has MSE score 0.0068, while the LinearLearner, in figure 4.1(b) has MSE score 0.0028.

CHAPTER 5

Conclusions

Predicting time series stock data is a complex problem that involves many rational and irrational factors that affect the future closing price of a stock. Previous work[1,2] on this topic has shown that there is no one magic formula that can tackle this problem. Rather there are many methods which when used properly (with fine tuning of respective model parameters) can lead to a suitable way of modeling the data. In this project, three such implementations are shown – LinearLearner, XGBoost, and Long Short-Term Memory (LSTM). The metric used to evaluate the model performance is the mean squared error (MSE) and the root mean squared error (RMSE). These metric indicate how close the model predictions are compared to the actual values. The Linear Learner and XGBoost are considered benchmark models, due to their relative ease of implementation and their association to regression problems. A custom LSTM model is implemented to showcase the performance of neural networks on time series data. The implementations of these models in this project work indicates that the LinearLearner model has the least MSE score of 0.0028. The XGBoost model, after hyperparameter tuning, has a higher MSE score of 0.1128. A possible reason is that additional parameters need to be tuned as well. On the other hand, the custom LSTM model has MSE score 0.0068. This shows that LSTMs are capable of modeling the historical stock price data and can make future predictions on the closing price of the stock during a desired time frame.

In addition, this project work also showcases the use of Amazon SageMaker workflow for model development and deployment. The LinearLearner and XGBoost are in-built in SageMaker and thus facilitate ease of model training and deployment. SageMaker also facilitates use of custom models in Keras, TensorFlow, PyTorch, etc. The custom LSTM model used in this project was implemented using TensorFlow Keras wrapper, and was then exported using a high-level TensorFlow API in ProtoBuf format, so that it can be later deployed from inside SageMaker. Therefore, in all this project encompassed the three aspects of machine learning – data exploration, modeling and deployment – which is common in many real projects.

The implementations showcased in this project has much room fro improvement. The XGBoost model needs detailed investigation for parameter tuning. To show the true strength of LSTMs the dataset must be augmented with other information – such as post-drift of stocks after public news release[3]. Since LSTMs can learn long term dependencies, having such augmented datasets can enable LSTMs to outshine the conventional regression models.

[1] https://towardsdatascience.com/machine-learning-techniques-applied-to-stock-price-prediction-6c1994da8001

[2] https://www.analyticsvidhya.com/blog/2018/10/predicting-stock-price-machine-learningnd-deep-learning-techniques-python/

[3] http://www.econ.yale.edu/ shiller/behfin/2001-05-11/chan.pdf