Published in Towards Data Science

Yibin Ng Follow
Jan 28, 2019 · 11 min read ★ · Listen

Machine Learning Techniques applied to Stock Price Prediction

Image generated using Neural Style Transfer.

Machine learning has many applications, one of which is to forecast time series. One of the most interesting (or perhaps most profitable) time series to predict are, arguably, stock prices.

Recently I read a blog post applying machine learning techniques to stock price prediction. You can read it here. It is a well-written article, and various techniques were explored. However, I felt the problem could be handled with a bit more academic rigor. For example, in the article the methods "Moving Average", "Auto ARIMA" and "Prophet" had a forecast horizon of **1 year**, whereas "Linear Regression", "k-Nearest Neighbors", and "Long Short Term Memory (LSTM)" had a forecast horizon of **1 day**. Towards the end of the article, it is stated: "LSTM has easily outshone any algorithm we saw so far." But clearly, we are not comparing apples to apples here.

So, here's my take on the problem.

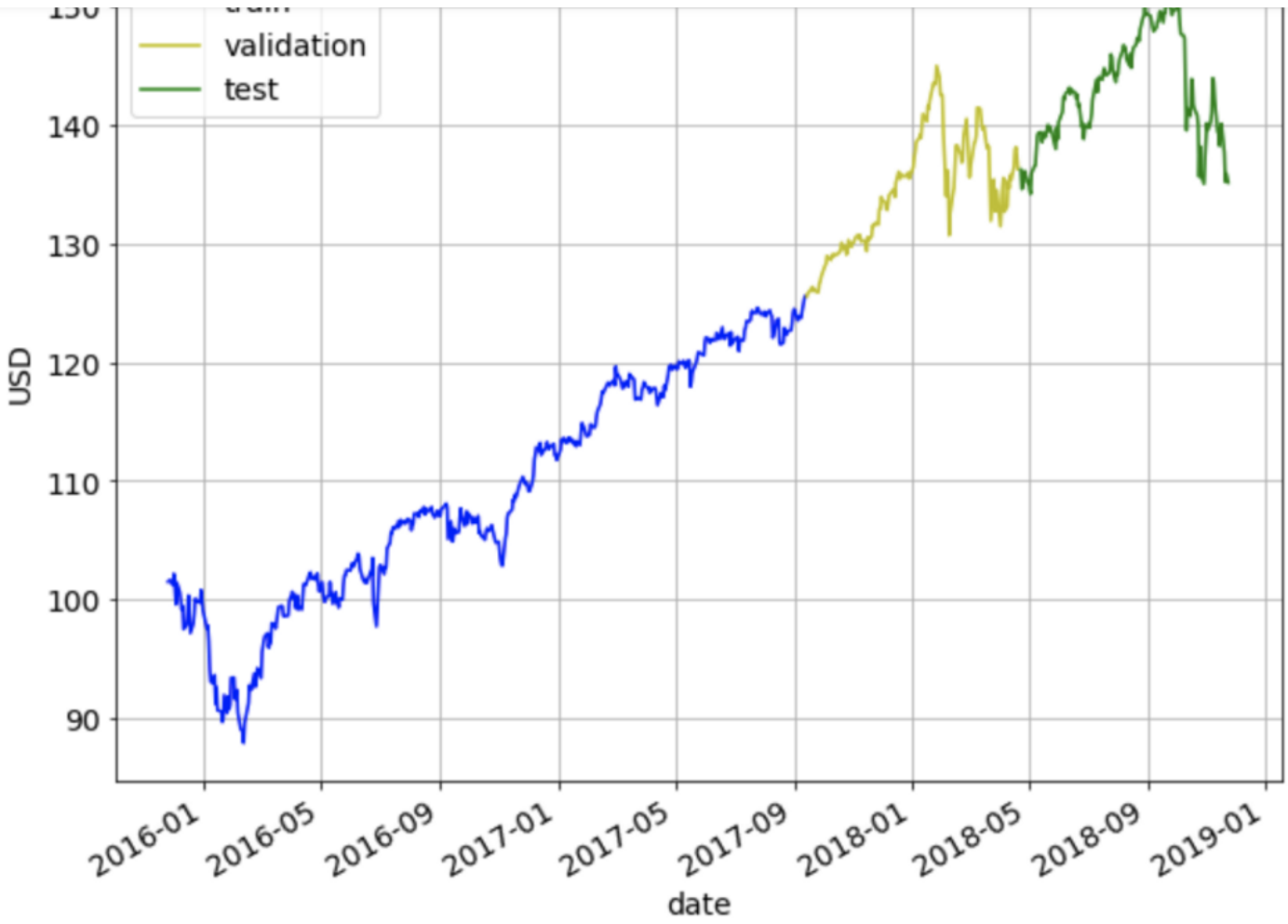
Problem Statement

We aim to predict the daily adjusted closing prices of Vanguard Total Stock Market ETF (VTI), using data from the previous N days (ie. forecast horizon=1). We will use three years of historical prices for VTI from 2015–11–25 to 2018–11–23, which can be easily downloaded from yahoo finance. After downloading, the dataset looks like this:

2015-11-25	107.510002	107.660004	107.250000	107.470001	101.497200	1820300
2015-11-27	107.589996	107.760002	107.220001	107.629997	101.648300	552400
2015-11-30	107.779999	107.849998	107.110001	107.169998	101.213867	3618100
2015-12-01	107.589996	108.209999	107.370003	108.180000	102.167740	2443600
2015-12-02	108.099998	108.269997	106.879997	107.050003	101.100533	2937200
2015-12-03	107.290001	107.480003	105.059998	105.449997	99.589470	3345600
2015-12-04	105.809998	107.540001	105.620003	107.389999	101.421646	4520000
2015-12-07	107.230003	107.269997	106.059998	106.550003	100.628342	3000500
2015-12-08	105.940002	106.400002	105.269997	105.910004	100.023895	3149600
2015-12-09	105.550003	106.750000	104.480003	105.000000	99.164467	4179800

Downloaded dataset for VTI.

We will split this dataset into 60% train, 20% validation, and 20% test. The model will be trained using the train set, model hyperparameters will be tuned using the validation set, and finally the performance of the model will be reported using the test set. Below plot shows the adjusted closing price split up into the respective train, validation and test sets.



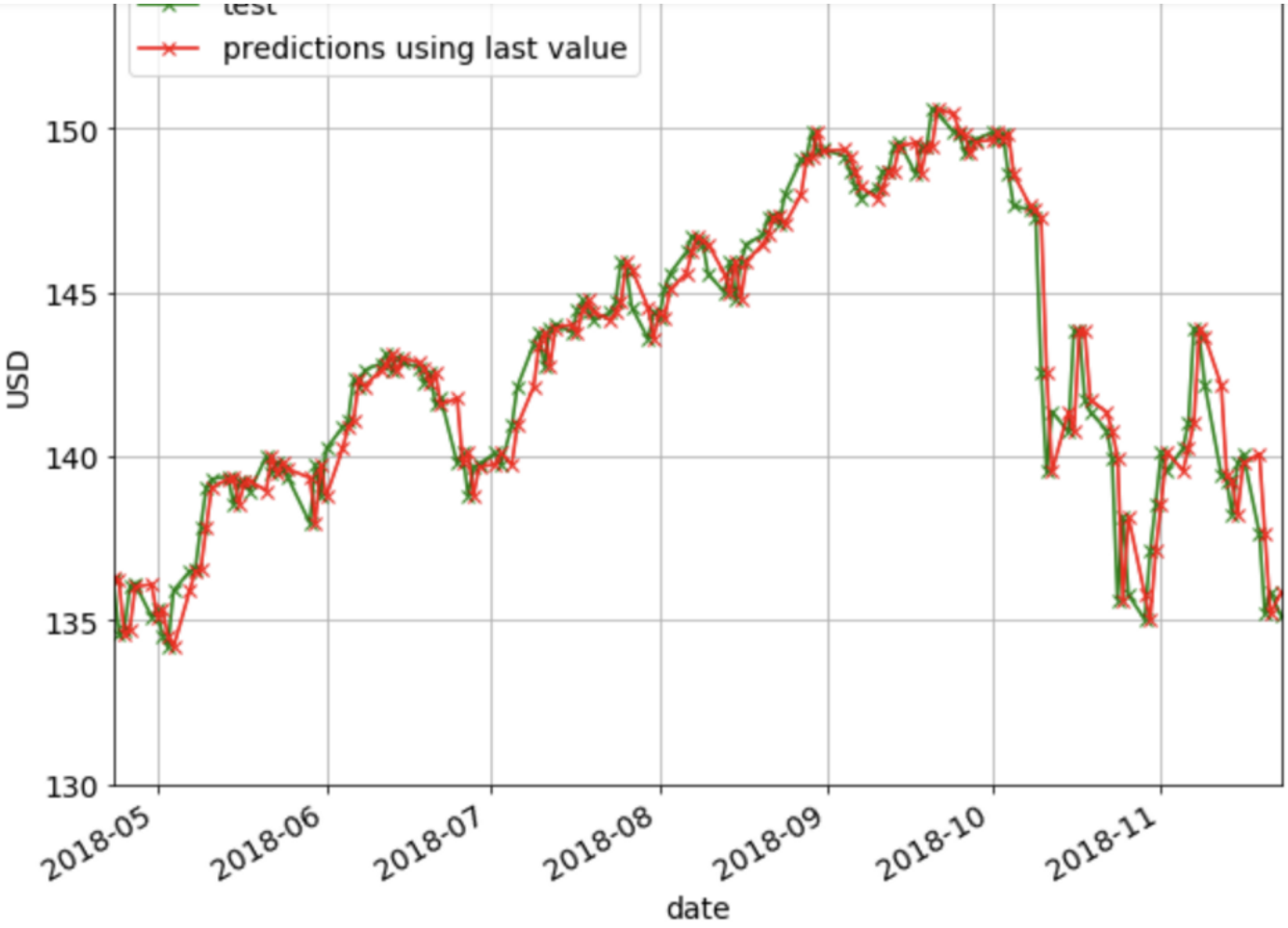
Split the dataset into 60% train, 20% validation, and 20% test.

To evaluate the effectiveness of our methods, we will use the root mean square error (RMSE) and mean absolute percentage error (MAPE) metrics. For both metrics, the lower the value, the better the prediction.

Last Value

In the Last Value method, we will simply set the prediction as the last observed value. In our context, this means we set the current adjusted closing price as the previous day's adjusted closing price. This is the most cost-effective forecasting model and is commonly used as a benchmark against which more sophisticated models can be compared. There are no hyperparameters to be tuned here.

Below plot shows the predictions using the Last Value method. If you look closely, you can see that the predictions for each day (red cross) are simply the previous day's value (green cross).

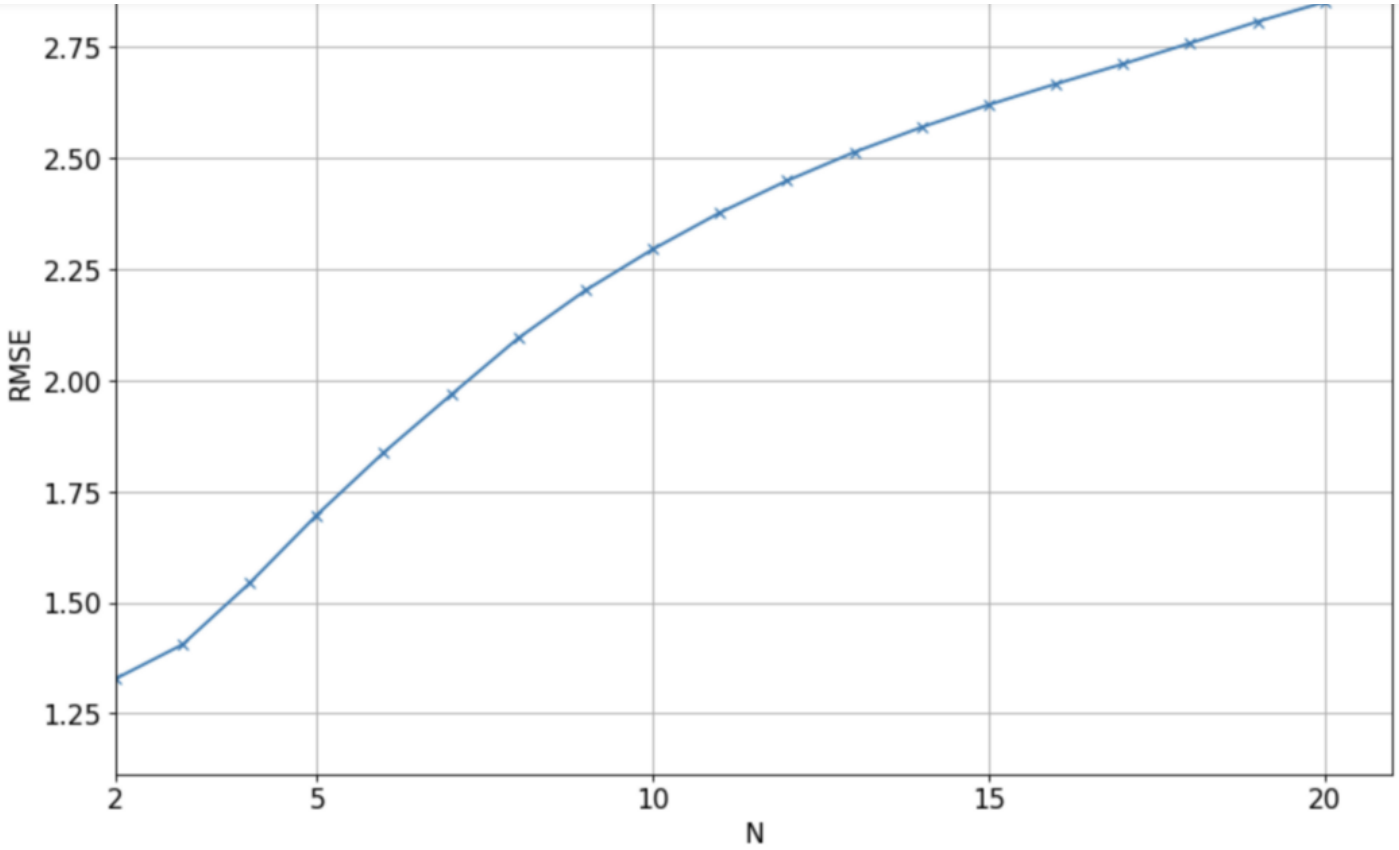


Predictions using the last value method.

Moving Average

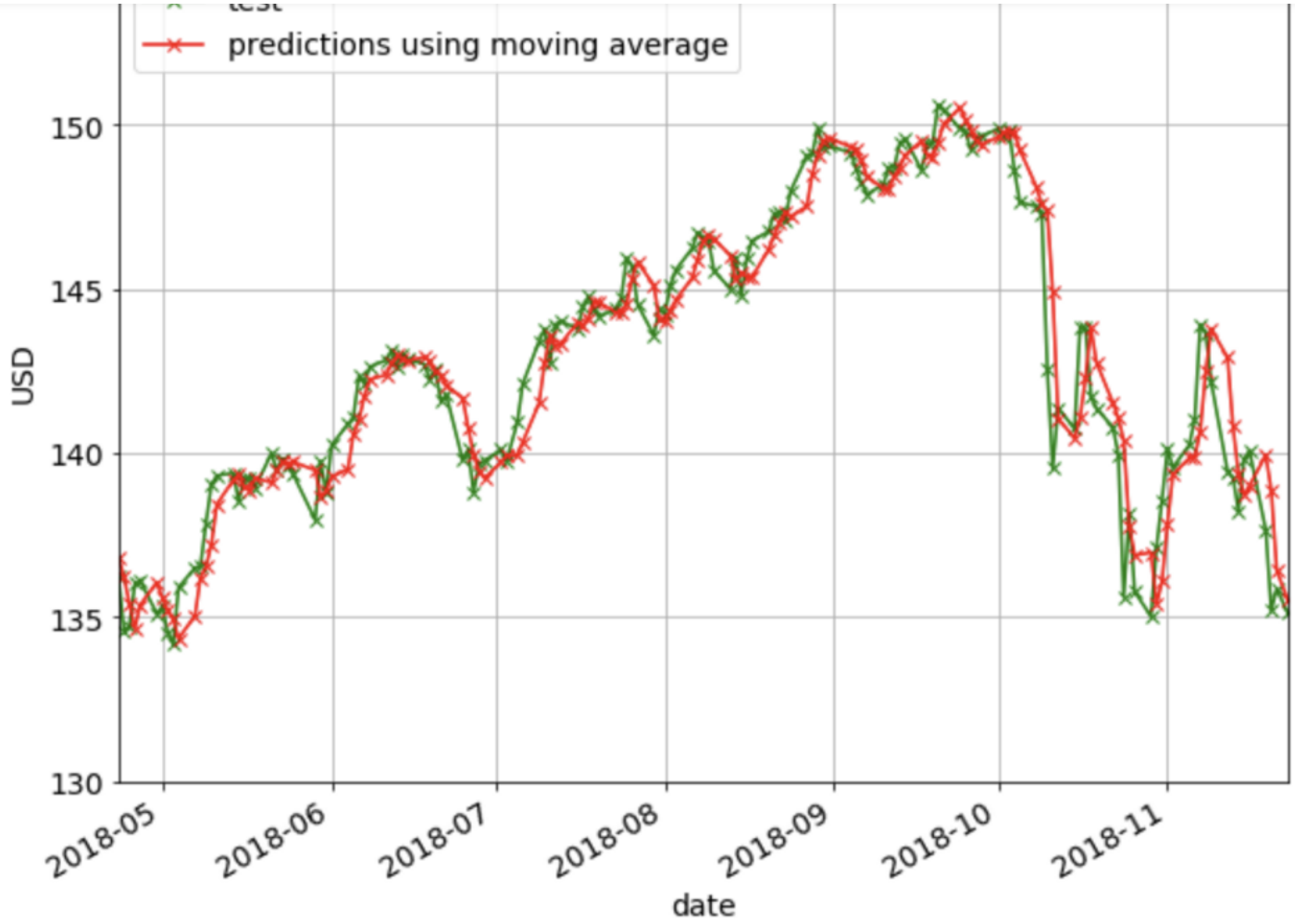
In the moving average method, the predicted value will be the mean of the previous N values. In our context, this means we set the current adjusted closing price as the mean of the adjusted closing price of the previous N days. The hyperparameter N needs to be tuned.

Below plot shows the RMSE between the actual and predicted values on the validation set, for various values of N. We will use N=2 since it gives the lowest RMSE.



RMSE between actual and predicted values on the validation set, for various N.

Below plot shows the predictions using the moving average method.

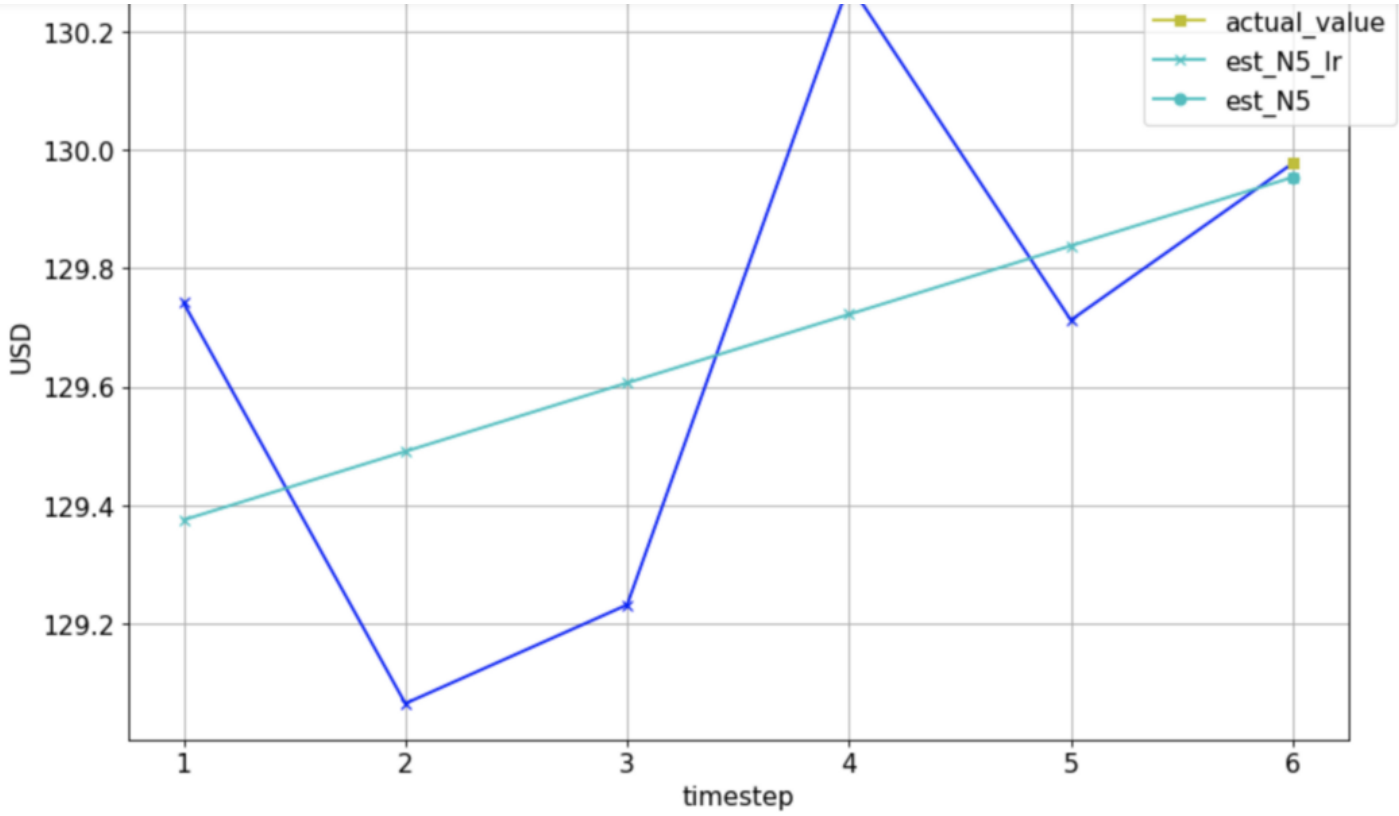


Predictions using the moving average method.

You can check out the Jupyter notebook for moving average method here.

Linear Regression

Linear regression is a linear approach to modeling the relationship between a dependent variable and one or more independent variables. The way we are going to use linear regression here is that we will fit a linear regression model to the previous N values, and use this model to predict the value on the current day. Below plot is an example for N=5. The actual adjusted closing prices are shown as dark blue cross, and we want to predict the value on day 6 (yellow square). We will fit a linear regression line (light blue line) through the first 5 actual values, and use it to do the prediction on day 6 (light blue circle).

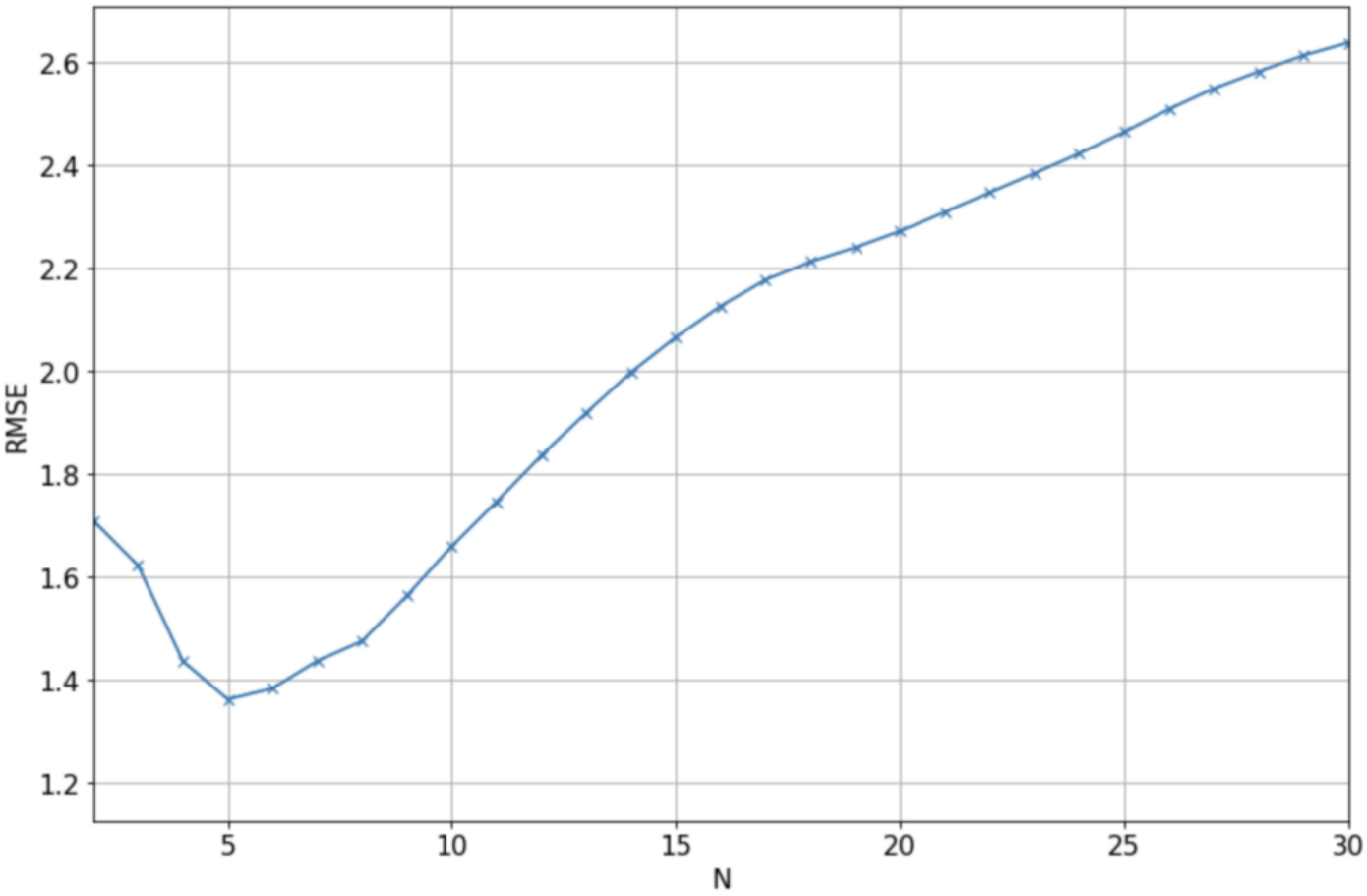


Predicting the next value using linear regression with N=5.

Below is the code we use to train the model and do predictions.

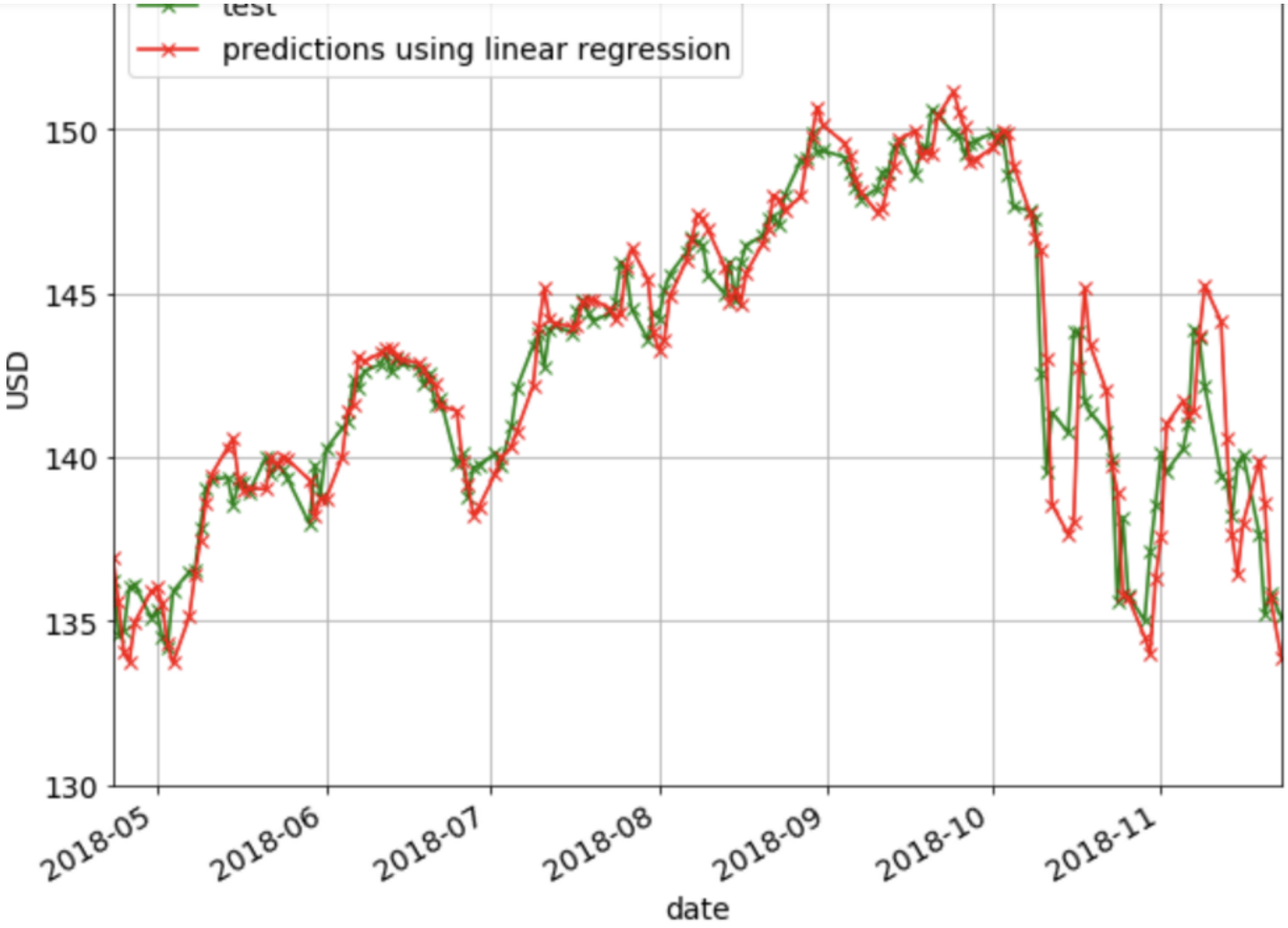
```
import numpy as np

from sklearn.linear_model import LinearRegression

def get_preds_lin_reg(df, target_col, N, pred_min, offset):
    """
    Given a dataframe, get prediction at each timestep
    Inputs
        df         : dataframe with the values you want to predict
        target_col : name of the column you want to predict
        N          : use previous N values to do prediction
        pred_min   : all predictions should be >= pred_min
        offset     : for df we only do predictions for df[offset:]
    Outputs
        pred_list  : the predictions for target_col
    """
    # Create linear regression object
    regr = LinearRegression(fit_intercept=True)

    pred_list = []

    for i in range(offset, len(df['adj_close'])):
        X_train = np.array(range(len(df['adj_close'][i-N:i])))
        y_train = np.array(df['adj_close'][i-N:i])
        X_train = X_train.reshape(-1, 1)
        y_train = y_train.reshape(-1, 1)
        regr.fit(X_train, y_train)            # Train the model
        pred = regr.predict(N)

        pred_list.append(pred[0][0])

    # If the values are < pred_min, set it to be pred_min
    pred_list = np.array(pred_list)
```

Below plot shows the RMSE between the actual and predicted values on the validation set, for various values of N. We will use N=5 since it gives the lowest RMSE.



RMSE between actual and predicted values on the validation set, for various N.

Below plot shows the predictions using linear regression method. It can be observed that this method does not capture changes in direction (ie. downtrend to uptrend and vice versa) very well.



Predictions using the linear regression method.

You can check out the Jupyter notebook for linear regression here.

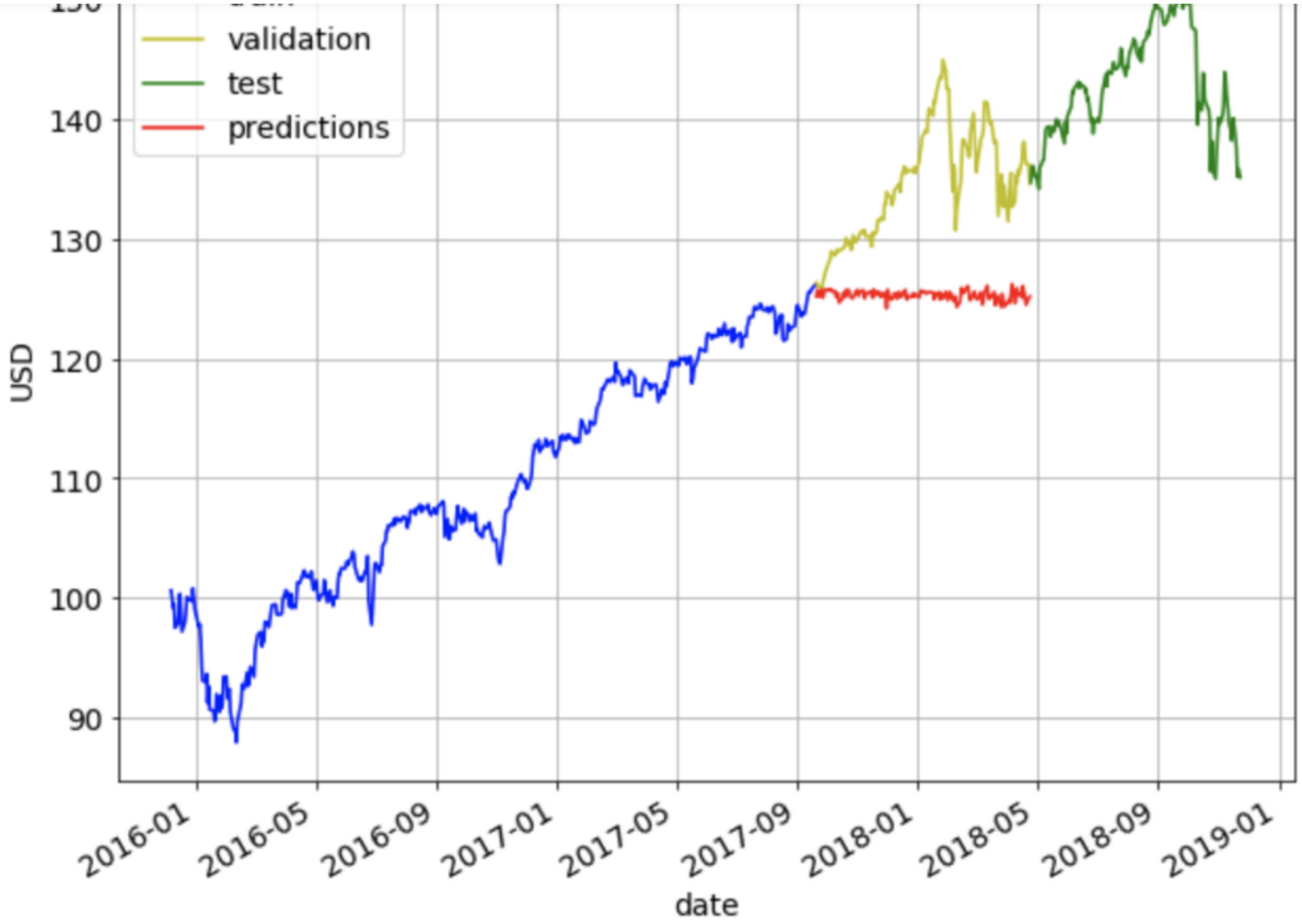
Extreme Gradient Boosting (XGBoost)

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. Ever since its introduction in 2014, XGBoost has proven to be a very powerful machine learning tech[illegible]-to algorithm in many Machine Learning competitions.

We will train the XGBoost model on the train set, tune its hyperparameters using the validation set, and finally apply the XGBoost model on the test set and report the results. Obvious features to use are the adjusted closing prices of the last N days, as well as the volume of the last N days. In addition to these features, we can do some feature engineering. The additional features we will construct are:

- difference between high and low for each day of the last N days
- difference between open and close for each day of the last N days

I learned an interesting lesson while constructing this model, which is that feature scaling is very important for the model to work properly. My first model did not implement any scaling at all, and predictions on the validation set are shown in the plot below. What happened here is that the model trained on adjusted closing price values between 89 to 125, and so the model can only output predictions within this range. When the model is trying to predict the validation set and it saw values out of this range, it is not able to generalize well.



Predictions are highly inaccurate if feature and target scaling are not done properly.

What I tried next was to scale the train set to have mean 0 and variance 1, and I applied this same transformation on the validation set. But clearly this will not work as well, because here we used the mean and variance computed from the train set to transform the validation set. Since values from the validation set are much larger than that from the train set, after scaling the values will still be larger. The result is that the predictions still look like the above, just that the values on the y-axis are now scaled.

Finally, what I did was that I scaled the train set to have mean 0 and variance 1, and use this to train the model. Subsequently, when I am doing predictions on the validation set, for each feature group of each sample, I will scale them to have mean 0 and variance 1. For example, if we are doing predictions on day T, I will take the adjusted closing prices of the last N days (days T-N to T-1) and scale them to have mean 0 and variance 1. The same is done for the volume features, where I will take the volume of the last N days and scale them to have mean 0 and variance 1. Repeat the same for the additional features we constructed above. We then use these scaled features to do prediction. The predicted values will also be scaled and we inverse transform them using their corresponding mean and variance. I found that this way of scaling gives the best performance, as we will see below.

Below is the code we use to train the model and do predictions.

```
import math
import numpy as np

from sklearn.metrics import mean_squared_error
from xgboost import XGBRegressor
```

```
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

def train_pred_eval_model(X_train_scaled, \
                          y_train_scaled, \
                          X_test_scaled, \
                          y_test, \
                          col_mean, \
                          col_std, \
                          seed=100, \
                          n_estimators=100, \
                          max_depth=3, \
                          learning_rate=0.1, \
                          min_child_weight=1, \
                          subsample=1, \
                          colsample_bytree=1, \
                          colsample_bylevel=1, \
                          gamma=0):
    '''
    Train model, do prediction, scale back to original range and do
    evaluation
    Use XGBoost here.
    Inputs
        X_train_scaled     : features for training. Scaled to have
                             mean 0 and variance 1
        y_train_scaled     : target for training. Scaled to have
                             mean 0 and variance 1
        X_test_scaled      : features for test. Each sample is
                             scaled to mean 0 and variance 1
        y_test             : target for test. Actual values, not
                             scaled
        col_mean           : means used to scale each sample of
                             X_test_scaled. Same length as
                             X_test_scaled and y_test
        col_std            : standard deviations used to scale each
                             sample of X_test_scaled. Same length as
                             X_test_scaled and y_test
        seed               : model seed
        n_estimators       : number of boosted trees to fit
        max_depth          : maximum tree depth for base learners
        learning_rate      : boosting learning rate (xgb's "eta")
        min_child_weight   : minimum sum of instance weight(hessian)
                             needed in a child
        subsample          : subsample ratio of the training
                             instance
        colsample_bytree   : subsample ratio of columns when
                             constructing each tree
        colsample_bylevel  : subsample ratio of columns for each
                             split, in each level
        gamma              : minimum loss reduction required to make
                             a further partition on a leaf node of
                             the tree
    Outputs
        rmse               : root mean square error of y_test and
                             est
        mape               : mean absolute percentage error of
                             y_test and est
        est                : predicted values. Same length as y_test
    '''

    model = XGBRegressor(seed=model_seed,
                         n_estimators=n_estimators,
                         max_depth=max_depth,
                         learning_rate=learning_rate,
                         min_child_weight=min_child_weight,
                         subsample=subsample,
                         colsample_bytree=colsample_bytree,
                         colsample_bylevel=colsample_bylevel,
                         gamma=gamma)

    # Train the model
    model.fit(X_train_scaled, y_train_scaled)

    # Get predicted labels and scale back to original range
    est_scaled = model.predict(X_test_scaled)
    est = est_scaled * col_std + col_mean
```

```
    return rmse, mape, est
```

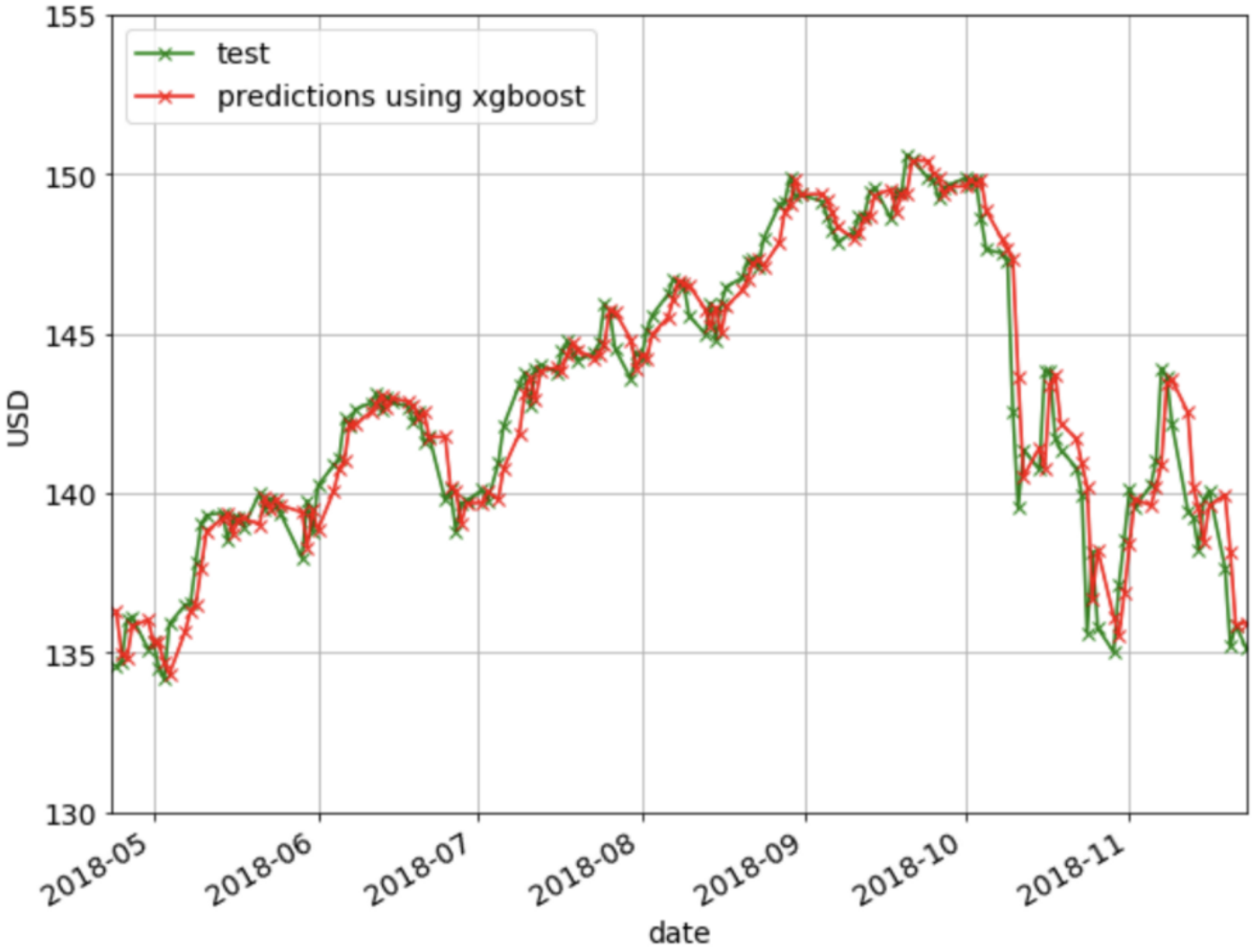
Below plot shows the RMSE between the actual and predicted values on the validation set, for various values of N. We will use N=3 since it gives the lowest RMSE.

N	rmse_dev_set	mape_pct_dev_set
2	1.225	0.585
3	1.214	0.581
4	1.231	0.590
5	1.249	0.601
6	1.254	0.609
7	1.251	0.612
14	1.498	0.763

Tuning N using RMSE and MAPE.

The hyperparameters and performance before and after tuning are shown below.

param	original	after_tuning
n_estimators	100.000	20.000
max_depth	3.000	5.000
learning_rate	0.100	0.100
min_child_weight	1.000	13.000
subsample	1.000	1.000
colsample_bytree	1.000	1.000
colsample_bylevel	1.000	1.000
gamma	0.000	0.100
rmse	1.214	1.214
mape_pct	0.581	0.578



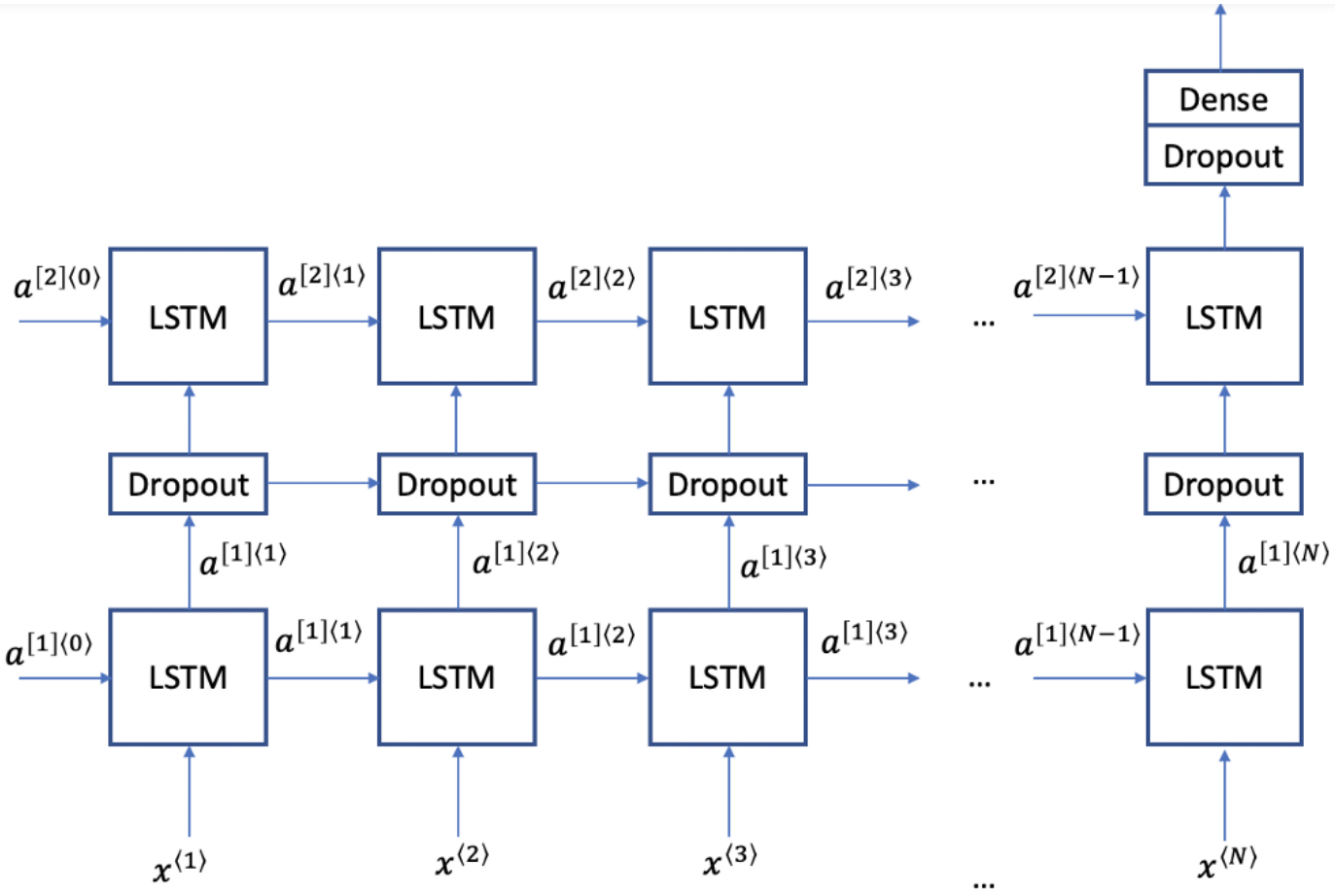
Predictions using the XGBoost method.

You can check out the Jupyter notebook for XGBoost here.

Long Short Term Memory (LSTM)

LSTM is a deep learning technique and was developed to combat the vanishing gradients problem encountered in long sequences. LSTM has three gates: the update gate, the forget gate and the output gate. The update and forget gates determine whether each element of the memory cell is updated. The output gate determines the amount of information to output as activations to the next layer.

The below shoes the LSTM architecture which we will be using. We will use two layers of LSTM modules, and a dropout layer in-between to avoid over-fitting.



LSTM network architecture.

Below is the code we use to train the model and do predictions.

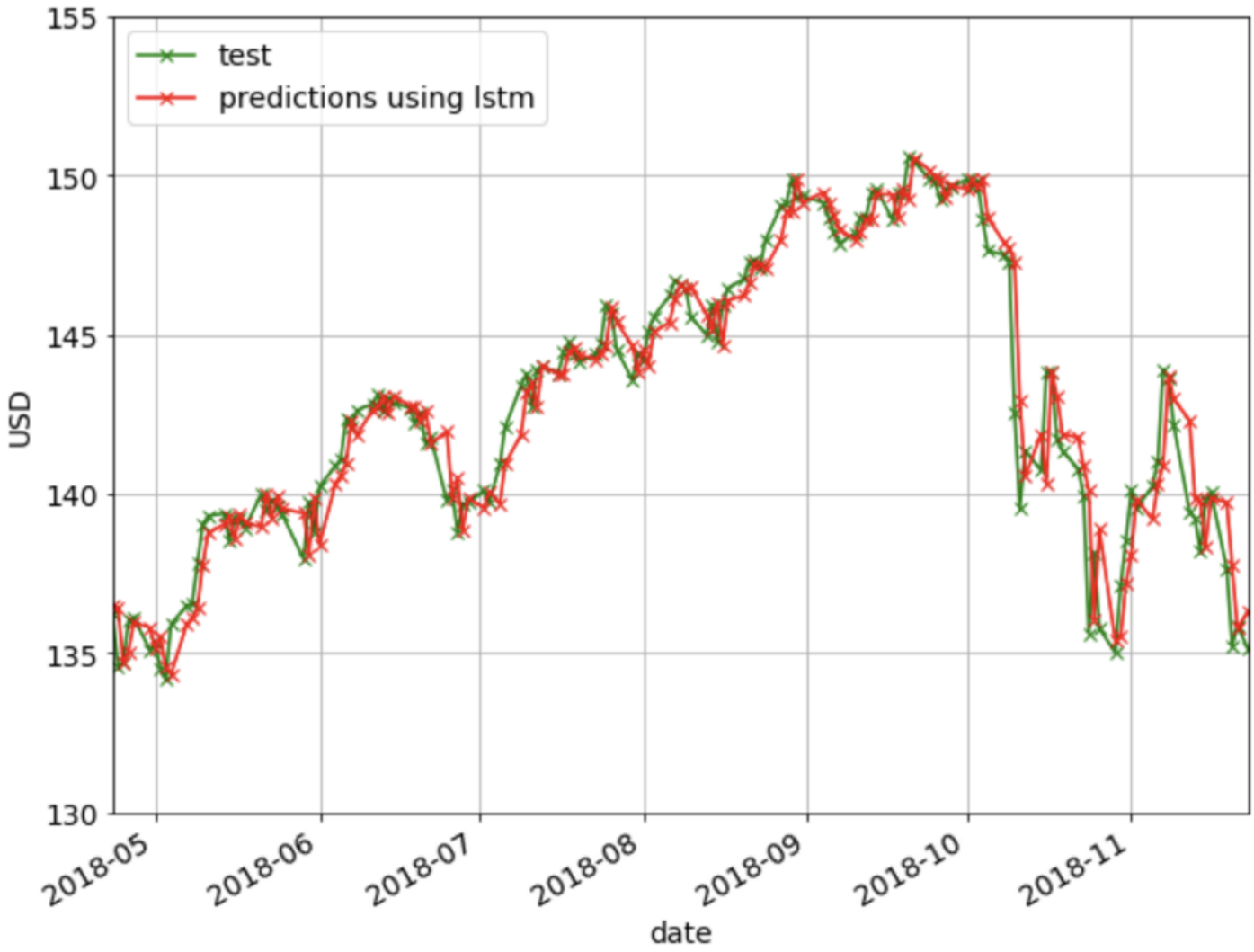
```
import math
import numpy as np

from keras.models import Sequential
from keras.layers import Dense, Dropout, LSTM

def train_pred_eval_model(x_train_scaled, \
                          y_train_scaled, \
                          x_test_scaled, \
                          y_test, \
                          mu_test_list, \
                          std_test_list, \
                          lstm_units=50, \
                          dropout_prob=0.5, \
                          optimizer='adam', \
                          epochs=1, \
                          batch_size=1):
    '''
    Train model, do prediction, scale back to original range and do
    evaluation
    Use LSTM here.
    Returns rmse, mape and predicted values
    Inputs
        x_train_scaled  : e.g. x_train_scaled.shape=(451, 9, 1).
                          Here we are using the past 9 values to
                          predict the next value
        y_train_scaled  : e.g. y_train_scaled.shape=(451, 1)
        x_test_scaled   : use this to do predictions
```

```
    dropout_prob    : fraction of the units to drop for the
                      linear transformation of the inputs
    optimizer       : optimizer for model.compile()
    epochs          : epochs for model.fit()
    batch_size      : batch size for model.fit()
Outputs
    rmse            : root mean square error
    mape            : mean absolute percentage error
    est             : predictions
'''
# Create the LSTM network
model = Sequential()
model.add(LSTM(units=lstm_units,
               return_sequences=True,
               input_shape=(x_train_scaled.shape[1],1)))

# Add dropput with a probability of 0.5
model.add(Dropout(dropout_prob))

model.add(LSTM(units=lstm_units))

# Add dropput with a probability of 0.5
model.add(Dropout(dropout_prob))

model.add(Dense(1))

# Compile and fit the LSTM network
model.compile(loss='mean_squared_error', optimizer=optimizer)
model.fit(x_train_scaled, y_train_scaled, epochs=epochs,
          batch_size=batch_size, verbose=0)

# Do prediction
est_scaled = model.predict(x_test_scaled)
est = (est_scaled * np.array(std_test_list).reshape(-1,1)) +
      np.array(mu_test_list).reshape(-1,1)

# Calculate RMSE and MAPE
rmse = math.sqrt(mean_squared_error(y_test, est))
mape = get_mape(y_test, est)

return rmse, mape, est
```

We will use the same method as in XGBoost to scale our dataset. The hyperparameters and performance of the LSTM network before and after tuning on the validation set are shown below.

param	original	after_tuning
N	9	3
lstm_units	50	128
dropout_prob	1	1
optimizer	adam	nadam
epochs	1	50
batch_size	1	8
rmse	2.22106	1.16053
mape_pct	1.13407	0.578567

Below plot shows the predictions using LSTM.

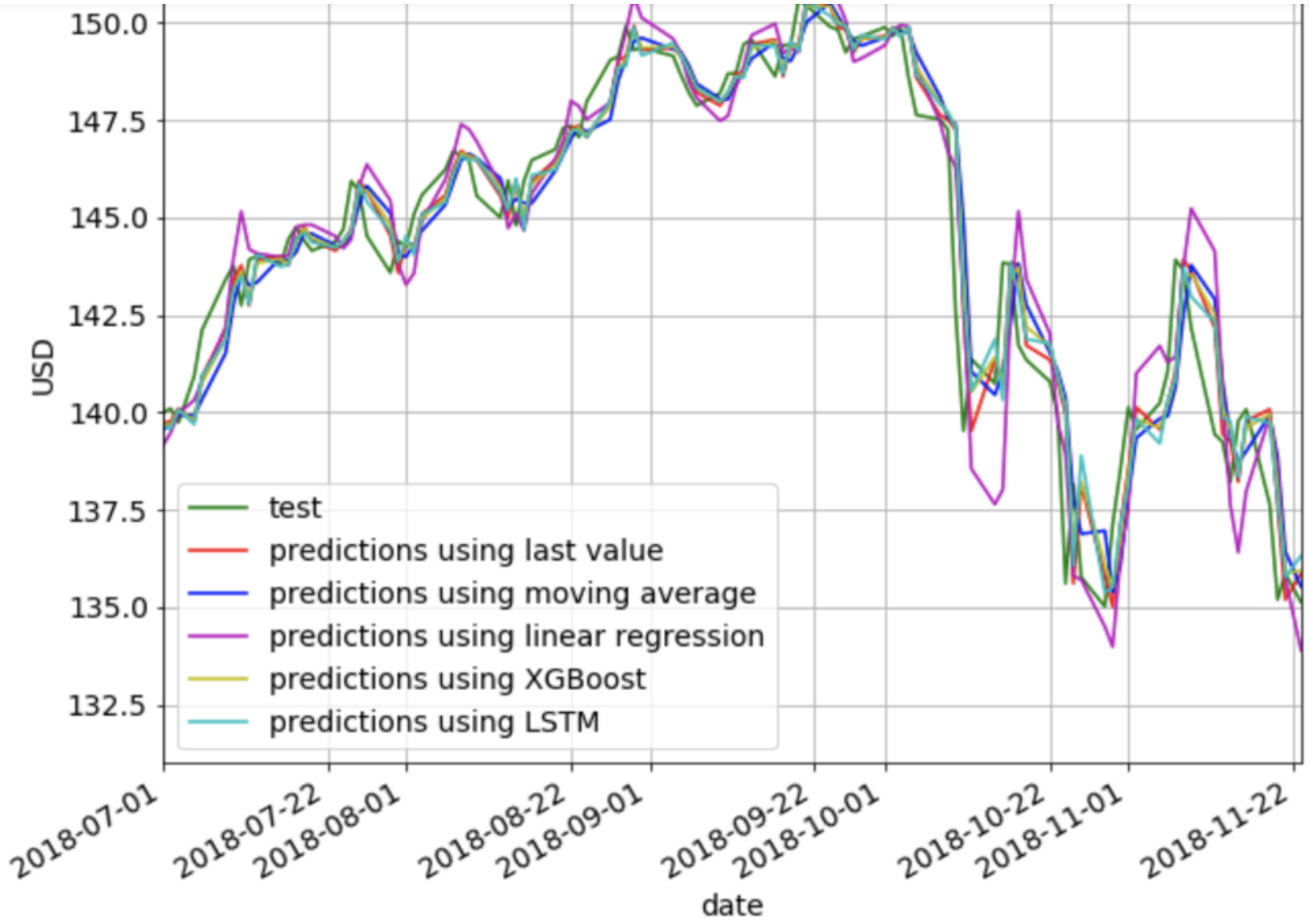


Predictions using the LSTM method.

You can check out the Jupyter notebook for LSTM here.

Findings and Future Work

Below we plot the predictions of all the methods we explored earlier in the same plot. It is clear predictions using linear regression provides the worst performance. Other than this, visually it is hard to tell which method provides the best predictions.



Below is a side-by-side comparison of the RMSE and MAPE of the various methods we explored. We see that the last value method gives the lowest RMSE and MAPE, followed by XGBoost, then LSTM. It is interesting that the simple last value method outperforms all other more sophisticated methods, but this is likely because our forecast horizon is only 1. For longer forecast horizons, I believe the other methods can capture trends and seasonality much better than the last value method.

method	RMSE	MAPE(%)
Last Value	1.127	0.565
Moving Average	1.270	0.640
Linear Regression	1.420	0.707
XGBoost	1.162	0.580
LSTM	1.164	0.583

Comparison of various methods using RMSE and MAPE.

As future work, it will be interesting to explore longer forecast horizons, for example 1 month or 1 year. It will also be interesting to explore other forecasting techniques such as autoregressive integrated moving average (ARIMA) and triple exponential smoothing (ie.