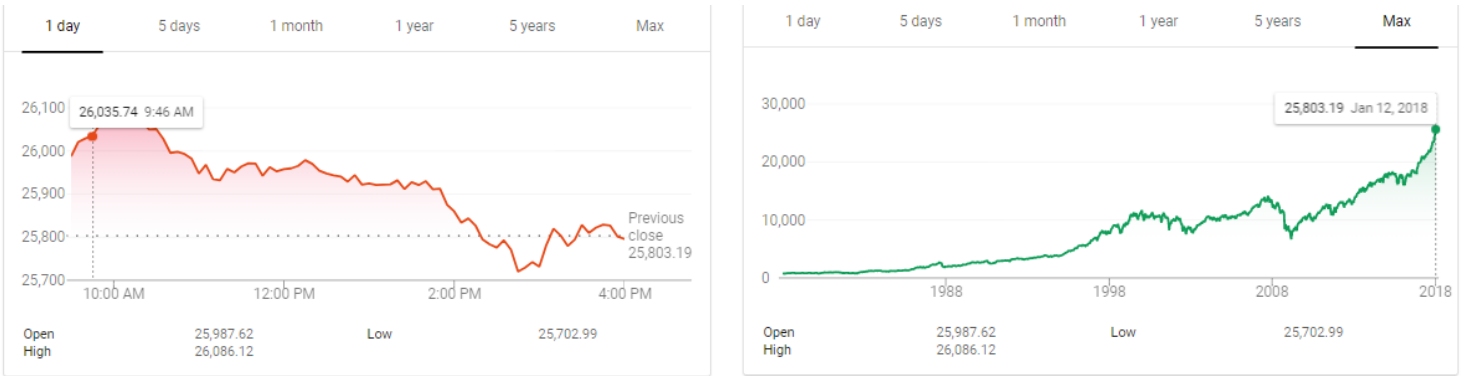
William Koehrsen Follow
Applied Data Science Researcher, Ultramarathon Runner, Cleveland Optimist
Jan 19 · 12 min read

Stock Prediction in Python

Make (and lose) fake fortunes while learning real Python

Trying to predict the stock market is an enticing prospect to data scientists motivated not so much as a desire for material gain, but for the challenge.We see the daily up and downs of the market and imagine there must be patterns we, or our models, can learn in order to beat all those day traders with business degrees. Naturally, when I started using additive models for time series prediction, I had to test the method in the proving ground of the stock market with simulated funds. Inevitably, I joined the many others who have tried to beat the market on a day-to-day basis and failed. However, in the process, I learned a ton of Python including object-oriented programming, data manipulation, modeling, and visualization. I also found out why we should avoid playing the daily stock market without losing a single dollar (all I can say is play the long game)!



One day vs 30 years: which would you rather put your money on?

When we don't experience immediate success—in any task, not just data science—we have three options:

1. Tweak the results to make it look like we were successful
2. Hide the results so no one ever notices
3. Show all our results and methods so that others (and ourselves) can learn how to do things better

While option three is the best choice on an individual and community level, it takes the most courage to implement. I can selectively choose ranges when my model delivers a handsome profit, or I can throw it away and pretend I never spent hours working on it. That seems pretty naive! We advance by repeatedly failing and learning rather than by only promoting our success. Moreover, Python code written for a difficult task is not Python code written in vain!

This post documents the prediction capabilities of Stocker, the "stock explorer" tool I developed in Python. In a previous article, I showed how to use Stocker for analysis, and the complete code is available on GitHub for anyone wanting to use it themselves or contribute to the project.

Stocker for Prediction

Stocker is a Python tool for stock exploration. Once we have the required libraries installed (check out the documentation) we can start a Jupyter Notebook in the same folder as the script and import the Stocker class:

```
from stocker import Stocker
```

The class is now accessible in our session. We construct an object of the Stocker class by passing it any valid stock ticker (**bold** is output):

```
amazon = Stocker('AMZN')
```

```
AMZN Stocker Initialized. Data covers 1997-05-16 to 2018-01-18.
```

Just like that we have 20 years of daily Amazon stock data to explore! Stocker is built on the Quandl financial library and with over 3000 stocks to use. We can make a simple plot of the stock history using the `plot_stock` method:

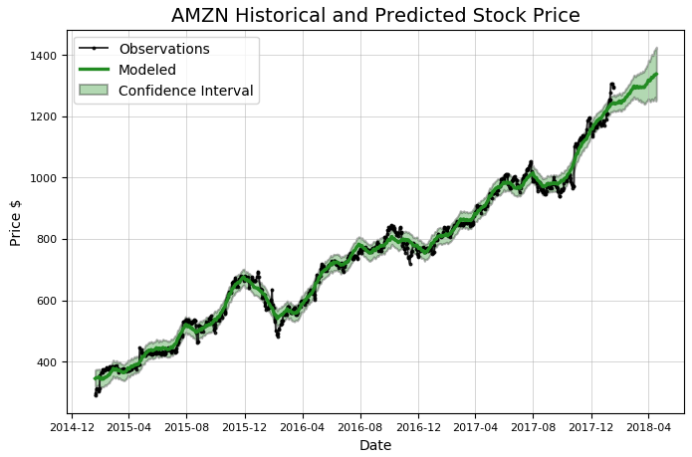
```
amazon.plot_stock()
```

```
Maximum Adj. Close = 1305.20 on 2018-01-12.
Minimum Adj. Close = 1.40 on 1997-05-22.
Current Adj. Close = 1293.32.
```

The analysis capabilities of Stocker can be used to find the overall trends and patterns within the data, but we will focus on predicting the future price. Predictions in Stocker are made using an additive model which considers a time series as a combination of an overall trend along with seasonalities on different time scales such as daily, weekly, and monthly. Stocker uses the prophet package developed by Facebook for additive modeling. Creating a model and making a prediction can be done with Stocker in a single line:

```
# predict days into the future
model, model_data = amazon.create_prophet_model(days=90)
```

```
Predicted Price on 2018-04-18 = $1336.98
```



Notice that the prediction, the green line, contains a confidence interval. This represents the model's uncertainty in the forecast. In this case, the confidence interval width is set at 80%, meaning we expect that this range will contain the actual value 80% of the time. The confidence interval grows wide further out in time because the estimate has more uncertainty as it gets further away from the data. Any time we make a prediction we must include a confidence interval. Although most people tend to want a simple answer about the future, our forecast must reflect that we live in an uncertain world!

Anyone can make stock predictions: simply pick a number and that's your estimate (I might be wrong, but I'm pretty sure this is all people on Wall Street do). For us to trust our model we need to evaluate it for accuracy.There are a number of methods in Stocker for assessing model accuracy.

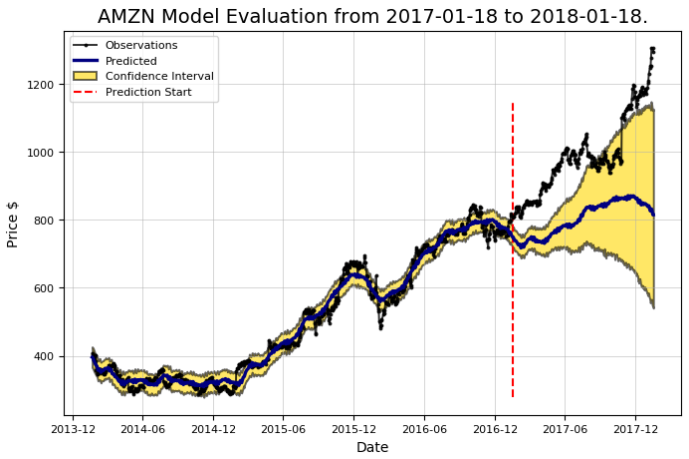
Evaluate Predictions

To calculate accuracy, we need a test set and a training set. We need to know the answers—the actual stock price—for the test set, so we will use the past one year of historical data (2017 in our case). When training, we do not let our model see the answers to the test set, so we use three years of data previous to the testing time frame (2014–2016). The basic idea of supervised learning is the model learns the patterns and relationships in the data from the training set and then is able to correctly reproduce them for the test data.

We need to quantify our accuracy, so we using the predictions for the test set and the actual values, we calculate metrics including average dollar error on the testing and training set, the percentage of the time we correctly predicted the direction of a price change, and the percentage of the time the actual price fell within the predicted 80% confidence interval. All of these calculations are automatically done by Stocker with a nice visual:

```
amazon.evaluate_prediction()
```

```
Prediction Range: 2017-01-18 to 2018-01-18.

Predicted price on 2018-01-17 = $814.77.
Actual price on    2018-01-17 = $1295.00.

Average Absolute Error on Training Data = $18.21.
Average Absolute Error on Testing  Data = $183.86.

When the model predicted an increase, the price increased
57.66% of the time.
When the model predicted a  decrease, the price decreased
44.64% of the time.

The actual value was within the 80% confidence interval
20.00% of the time.
```



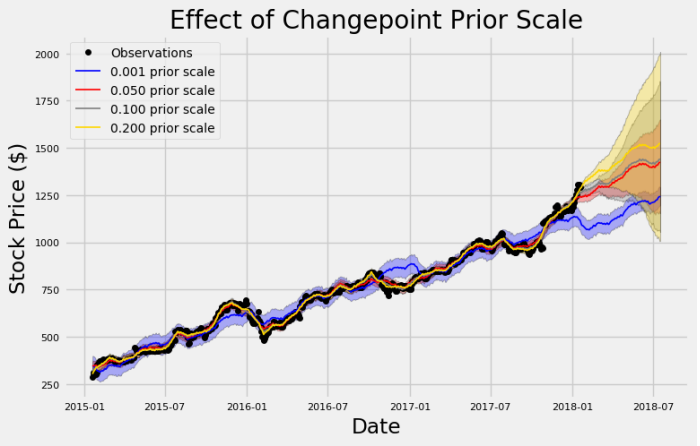
Those are abysmal stats! We might as well have flipped a coin. If we were using this to invest, we would probably be better off buying something sensible like lottery tickets. However, don't give up on the model just yet. We usually expect a first model to be rather bad because we are using the default settings (called hyperparameters). If our initial attempts are not successful, we can turn these knobs to make a better model. There are a number of different settings to adjust in a Prophet model, with the most important the changepoint prior scale which controls the amount of weight the model places on shifts in the trend of the data.

Changepoint Prior Selection

Changepoints represent where a time series goes from increasing to decreasing or from increasing slowly to increasingly rapidly (or vice versa). They occur at the places with the greatest change in the rate of the time series. The changepoint prior scale represents the amount of emphasis given to the changepoints in the model. This is used to control overfitting vs. underfitting (also known as the bias vs. variance tradeoff).

A higher prior creates a model with more weight on the changepoints and a more flexible fit. This may lead to overfitting because the model will closely stick to the training data and not be able to generalize to new test data. Lowering the prior decreases the model flexibility which can cause the opposite problem: underfitting. This occurs when our model does not follow the training data closely enough and fails to learn the underlying patterns. Figuring out the proper settings to achieve the right balance is more a matter of engineering than of theory, and here we must rely on empirical results. The Stocker class contains two different ways to choose an appropriate prior: visually and quantitatively. We can start off with the graphical method:

```
# changepoint priors is the list of changepoints to evaluate
amazon.changepoint_prior_analysis(changepoint_priors=[0.001,
0.05, 0.1, 0.2])
```



Here, we are training on three years of data and then showing predictions for six months. We do not quantify the predictions here because we are just trying to understand the role of the changepoint prior. This graph does a great job of illustrating under- vs overfitting! The lowest prior, the blue line, does not follow the training data, the black observations , very closely. It kind of does its own thing and picks a route through the general vicinity of the data. In contrast, the highest prior, the yellow line, sticks to the training observations as closely as possible. The default value for the changepoint prior is 0.05 which falls somewhere in between the two extremes.

Notice also the difference in uncertainty (shaded intervals) for the priors. The lowest prior has the largest uncertainty on the *training* data, but the smallest uncertainty on the *test* data. In contrast, the highest prior has the smallest uncertainty on the *training* data but the greatest uncertainty on the *test* data. The higher the prior, the more confident it is on the training data because it closely follows each observation. When it comes to the test data however, an overfit model is lost without any data points to anchor it. As stocks have quite a bit of variability, we probably want a more flexible model than the default so the model can capture as many patterns as possible.

Now that we have an idea of the effect of the prior, we can numerically evaluate different values using a training and validation set:

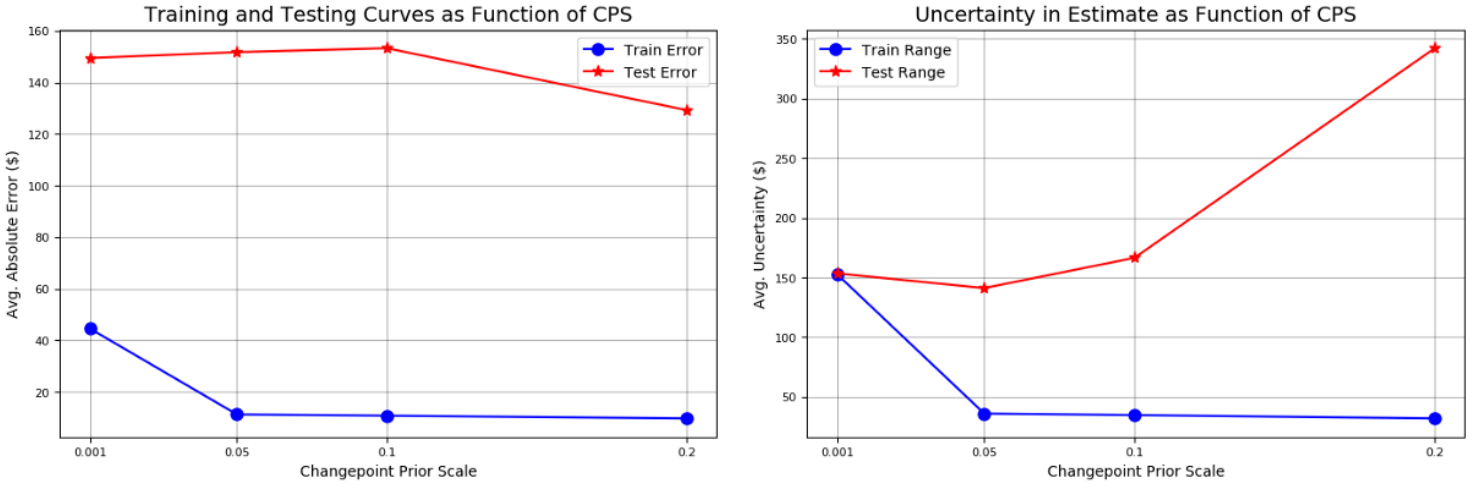
```
amazon.changepoint_prior_validation(start_date='2016-01-04',
end_date='2017-01-03', changepoint_priors=[0.001, 0.05, 0.1,
0.2])
```

```
Validation Range 2016-01-04 to 2017-01-03.

  cps  train_err  train_range    test_err  test_range
0.001  44.507495   152.673436  149.443609  153.341861
0.050  11.207666    35.840138  151.735924  141.033870
0.100  10.717128    34.537544  153.260198  166.390896
0.200   9.653979    31.735506  129.227310  342.205583
```

Here, we have to be careful that our validation data is not the same as our testing data. If this was the case, we would create the best model for the test data, but then we would just be overfitting the test data and our model could not translate to real world data. In total, as is commonly done in data science, we are using three different sets of data: a training set (2013–2015), a validation set (2016), and a testing set (2017).

We evaluated four priors with four metrics: training error, training range (confidence interval), testing error, and testing range (confidence interval) with all values in dollars. As we saw in the graph, the higher the prior, the lower the training error and the lower the uncertainty on the training data. We also see that a higher prior decreases our testing error, backing up our intuition that closely fitting to the data is a good idea with stocks. In exchange for greater accuracy on the test set, we get a greater range of uncertainty on the test data with the increased prior.

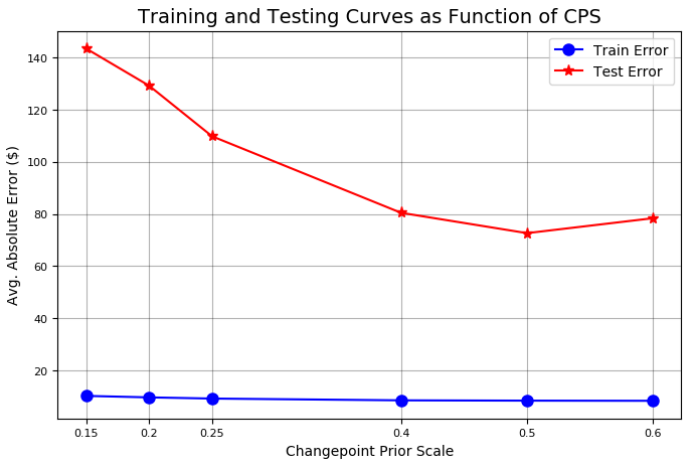
The Stocker prior validation also displays two plots illustrating these points:



Training and Testing Accuracy Curves and Uncertainty for Different Changepoint Prior Scales

Since the highest prior produced the lowest testing error, we should try to increase the prior even higher to see if we get better performance. We can refine our search by passing in additional values to the validation method:

```
# test more changepoint priors on same validation range
amazon.changepoint_prior_validation(start_date='2016-01-04',
end_date='2017-01-03', changepoint_priors=[0.15, 0.2,
0.25,0.4, 0.5, 0.6])
```



Refined Training and Testing Curves

The test set error is minimized at a prior of 0.5. We will set the changepoint prior attribute of the Stocker object appropriately.

```
amazon.changepoint_prior_scale = 0.5
```

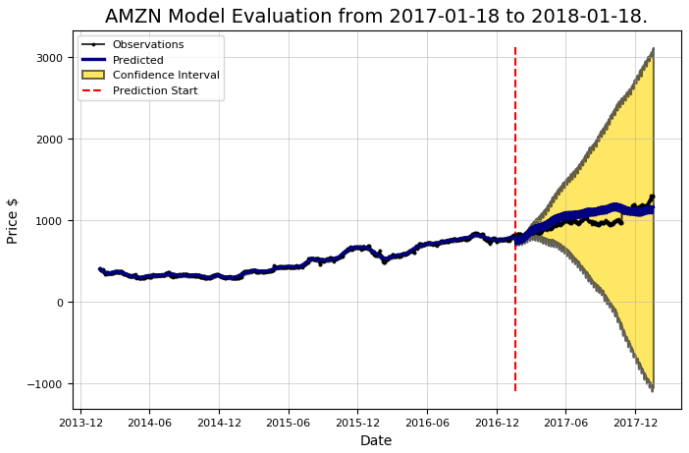
There are other settings of the model we can adjust, such as the patterns we expect to see, or the number of training years of data the model uses. Finding the best combination simply requires repeating the above procedure with a number of different values. Feel free to try out any settings!

Evaluating Refined Model

Now that our model is optimized, we can again evaluate it:

```
amazon.evaluate_prediction()
```

```
Prediction Range: 2017-01-18 to 2018-01-18.

Predicted price on 2018-01-17 = $1164.10.
Actual price on    2018-01-17 = $1295.00.

Average Absolute Error on Training Data = $10.22.
Average Absolute Error on Testing  Data = $101.19.

When the model predicted an increase, the price increased
57.99% of the time.
When the model predicted a  decrease, the price decreased
46.25% of the time.

The actual value was within the 80% confidence interval
95.20% of the time.
```



That looks better! This shows the importance of model optimization. Using default values provides a reasonable first guess, but we need to be sure we are using the correct model "settings," just like we try to optimize how a stereo sounds by adjusting balance and fade (sorry for the outdated reference).

Playing the Stock Market

Making predictions is an interesting exercise, but the real fun is looking at how well these forecasts would play out in the actual market. Using the `evaluate_prediction` method, we can "play" the stock market using our model over the evaluation period. We will use a strategy informed by our model which we can then compare to the simple strategy of buying and holding the stock over the entire period.

The rules of our strategy are straightforward:

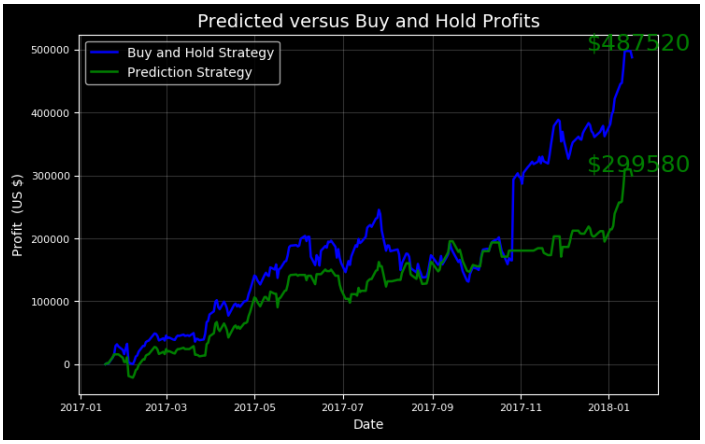
1. On each day the model predicts the stock to increase, we purchase the stock at the beginning of the day and sell at the end of the day. When the model predicts a decrease in price, we do not buy any stock.
2. If we buy stock and the price increases over the day, we make the increase times the number of shares we bought.
3. If we buy stock and the price decreases, we lose the decrease times the number of shares.

We play this each day for the entire evaluation period which in our case is 2017. To play, add the number of shares to the method call. Stocker will inform us how the strategy played out in numbers and graphs:

```
# Going big
amazon.evaluate_prediction(nshares=1000)
```

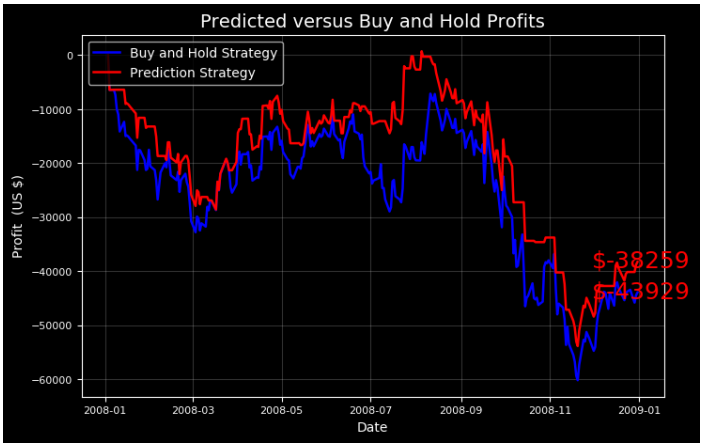
```
You played the stock market in AMZN from 2017-01-18 to 2018-01-18 with 1000 shares.
```

```
When the model predicted an increase, the price increased
57.99% of the time.
When the model predicted a  decrease, the price decreased
46.25% of the time.

The total profit using the Prophet model = $299580.00.
The Buy and Hold strategy profit =         $487520.00.

Thanks for playing the stock market!
```



This shows us a valuable lesson: buy and hold! While we would have made a considerable sum playing our strategy, the better bet would simply have been to invest for the long term.

We can try other test periods to see if there are times when our model strategy beats the buy and hold method. Our strategy is rather conservative because we do not play when we predict a market decrease, so we might expect to do better than a holding strategy when the stock takes a downturn.



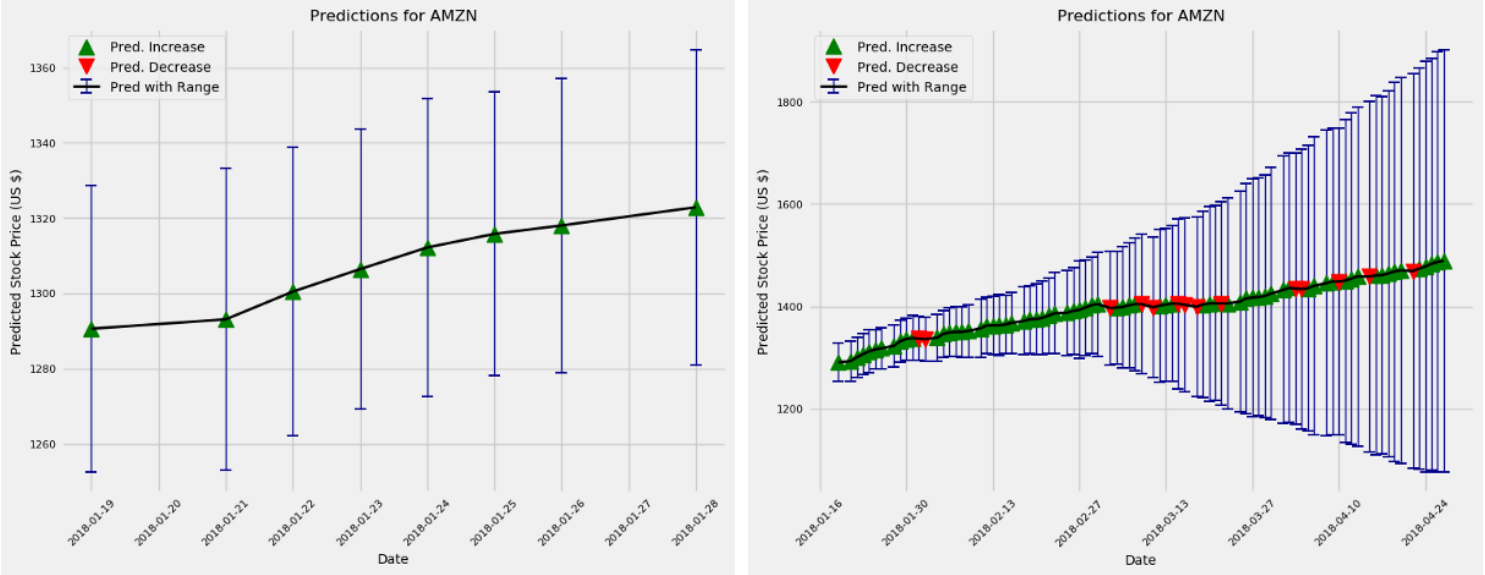
Always play with fake money!

I knew our model could do it! However, our model only beat the market when we were had the benefit of hindsight to choose the test period.

Future Predictions

Now that we are satisfied we have a decent model, we can make future predictions using the `predict_future()` method.

```
amazon.predict_future(days=10)
amazon.predict_future(days=100)
```



Predictions for the next 10 and 100 days

The model is overall bullish on Amazon as are most "professionals." Additionally, the uncertainty increases the further out in time we make estimates as expected. In reality, if we were using this model to actively trade, we would train a new model every day and would make predictions for a maximum of one day in the future.

While we might not get rich from the Stocker tool, the benefit is in the development rather than the end results! We can't actually know if we can solve a problem until we try but it's better to have tried and failed than to have never tried at all! For anyone interested in checking out the code or using Stocker themselves, it is available on GitHub.

As always, I enjoy feedback and constructive criticism. I can be reached on Twitter @koehrsen_will.