Security Price Analysis using Sentiment Analysis, Polynomial Regression, Markov Chains, and CART Trees

Siddharth Mohanty[1]

[1]Affiliation not available

March 17, 2023

Abstract

Many quantitative funds rely on data collection from buying user trades from brokerages while other funds attempt to make money off of small differences in price over minuscule amounts of time. This paper attempts to architect a python[1] system by which we analyze signals from Markov Chains[2], Polynomial Regression, and Sentiment Analysis latter passed into Decision Trees[3] for a holistic evaluation based solely on publicly available information.

Security Price Analysis using Sentiment Analysis, Polynomial Regression, Markov Chains, and CART Trees

Siddharth Mohanty

April 27, 2022

Abstract

Many quantitative funds rely on data collection from buying user trades from brokerages while other funds attempt to make money off of small differences in price over minuscule amounts of time. This paper attempts to architect a python[1] system by which we analyze signals from Markov Chains[2], Polynomial Regression, and Sentiment Analysis latter passed into Decision Trees[3] for a holistic evaluation based solely on publicly available information.

Keywords: Quantitative Finance, Markov Chains, Sentiment Analysis, Decision Trees, Regression

1 Introduction

The pandemic era economic situation—fueled by record low interest rates, unemployment, and gravity defying stock growth—prompted millions of retail investors to turn to day trading to rapidly make money; The vast majority lost money. On the other end of the spectrum, Institutional investing firms have historically under performed relative to index funds. It should be possible to leverage Machine Learning and Mathematical Models to provide better returns.

Review of the Existing Literature

This idea of using Mathematical Models and Machine learning in finance to has been widely researched by a variety of papers however few attempt to use Markov Chains, Polynomial Regression, or Sentiment Analysis to analyze securities on which this paper builds on.

Soloviev, Saptsin, and Chabanenko[4] suggest a prediction construction algorithm by which we "transform the time series of increments into the series of state numbers" similar to our construction algorithm. McLeish and Kolkiewicz[5], suggest a stochastic differential equation which transforms into a polynomial which we attempt to recreate through regression.

2 Sentiment Analysis

This system uses sentiment analysis as a process of extracting and objectively representing affective information. This system relies on wordlists to accurately and efficiently obtain an average score for the news pertaining to a specific stock.

2.1 Constructing Wordlists

We first obtain historical news for a stock and the price movement, positive or negative, and pair day i's news with day $i+1$'s movement. If there is positive movement we increment by 1 if there is negative movement we decrement by 1.

Given l is the number of occurrences and M is the set containing the associated movements for a word W.

$$W : \{l, \sum_{n=0}^{l} M_n\}$$

We then divide the sum by the number of occurrences to get the value of each word.

2.2 Usage Algorithm

Psuedocode is provided below

```
def sentiment(text) -> float:
    articles_val = 0.0
    articles_ct = 0.0
    for article in stock:
        sentences_val= 0.0
        sentences_ct = 0.0
        for sentence in article:
            words_val = 0.0
            words_ct = 0.0
            for word in sentence:
                words_val += word_list_value(word)
                words_ct += 1
            sentences_val += words_val / words_ct
            sentences_val += 1
        articles_val += sentences_val / sentences_ct
        articles_ct += 1
    return articles_val/articles_ct
```

3 Polynomial Regression

This project attempts to use Polynomial Regression to create accurate predictions. However relying on a polynomial of too low of a degree relies on inaccurate data and too high of a degree results in overfitting to the training data. In an attempt to work around this we take the average of the predictions of many classifiers. Grid Search reveals that bounds of 5 and 8 create the optimal result. Instead of implementing polynomial fitting we use the numpy[6] implementation

3.1 Average Polyfit Algorithm

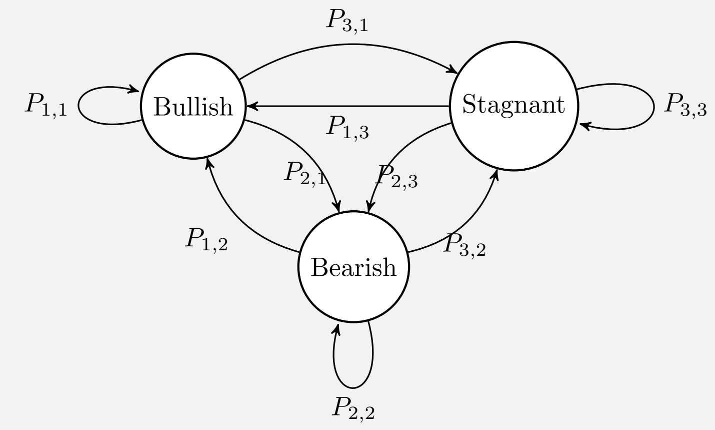
Psuedocode is provided below

```
def average_poly_fit(data, index) -> float:
    return(
        poly_fit(data, deg=5, index) +
        poly_fit(data, deg=6, index) +
        poly_fit(data, deg=7, index) +
        poly_fit(data, deg=8, index)
    )/4
```

4 Markov Chains

4.1 Implementation

This project attempts to use Markov chains which encode the following system:



We can now store this in the following stochastic matrix.

$$\mathbf{P} = \begin{array}{r} \\ \textit{Bullish} \\ \textit{Bearish} \\ \textit{Stagnant} \end{array} \begin{array}{c} \begin{array}{rrr} \textit{Bullish} & \textit{Bearish} & \textit{Stagnant} \end{array} \\ \begin{bmatrix} 0.8 & 0.15 & 0.45 \\ 0.1 & 0.5 & 0.05 \\ 0.1 & 0.35 & 0.5 \end{bmatrix} \end{array}$$

where $P_{i,j}$ denotes the probability of going from state j to state i.

4.2 Construction

Using historic data, if we find that if

$$d_{i+1} - d_i > s$$

where s is a bias variable and d is the data. Then, if

$$d_i - d_{i-1} > s$$

We increment $P_{1,1}$ by 1. Adjusting our formula for other nodes is trivial so we will not cover it. This produces a matrix which treats old data with the same importance as new data. We can adjust this by incrementing by i as opposed to 1 to proportionally weight data by recency. We can then test this data using the test covered in section 4.3 We can then divide all elements of a row by the sum of the row to obtain a matrix where all values in a row add up to 1. We then find the steady state vector of P, q and return q_1/q_2.

4.3 Usage

If we set q to the steady-state vector of $\mathbf{P}$

$$\mathbf{P}q = q$$

This is nothing more than finding an eigenvector of $\mathbf{P}$ and dividing all elements by the sum of the total. Luckily this is quite trivial to implement in python as it has already been implemented as a numpy[4] function.

Once we have acquired q we return q_1/q_2.

Psuedocode is provided below

```
def steady_state(markov):
    q = [real(s) for s in eig(markov)[1][0]]
    total = sum(q)
    q = [s/total for s in sol]
    return q[0]/q[1]
```

5 Decision Trees

Decision Trees offer a very fast way for us to determine whether a stock has a positive or negative outlook.

5.1 Training

Instead of manually implementing the CART algorithm, we rely on the scikit-learn[7] implementation. We feed the historical data in the following format:

$$\begin{bmatrix}\text{Weighed Regression Prediction}\\ \text{Markov Chain Difference}\\ \text{Sentiment Score}\end{bmatrix}$$

5.2 Pruning

After training and plotting we see the following tree:

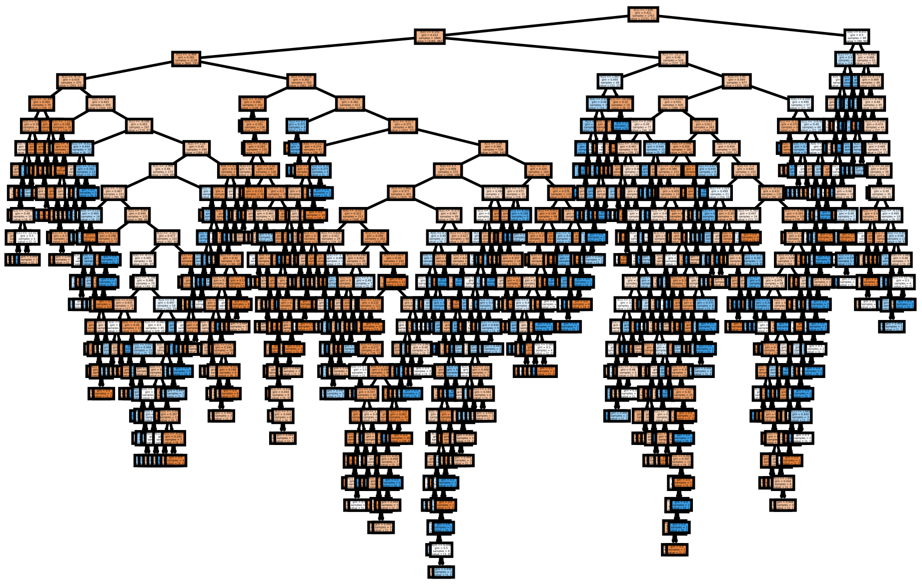


Figure 1: Decision Tree *before* pruning

Obviously this tree is much too convoluted and as a result leads to overfitting. We can remedy this by setting the ccp_alpha parameter of the tree to 0.001(A value acquired through Grid Search) which yields the following tree:

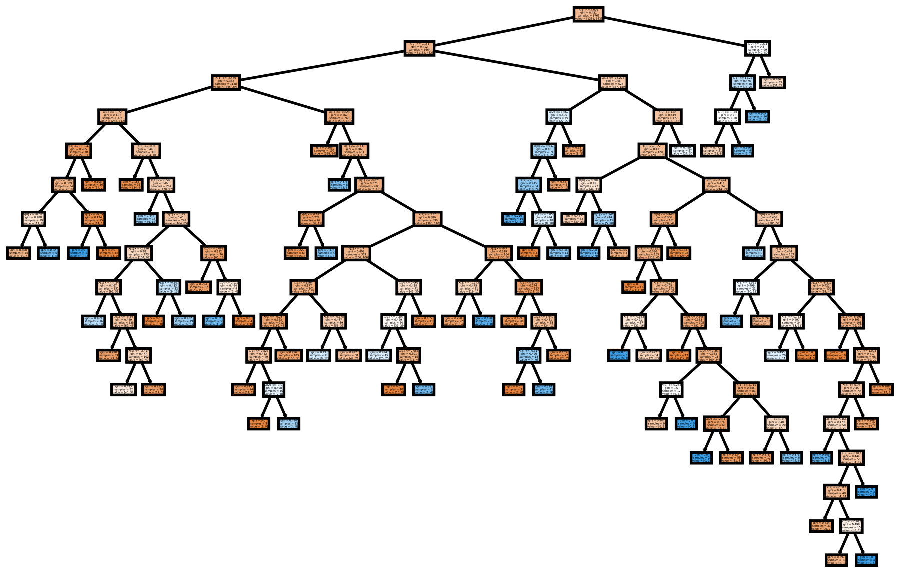


Figure 2: Decision Tree *after* pruning

6 Discussion

From initial data a few of our hypotheses are affirmed namely,

1. A stock-specific wordlist far outperforms a general wordlist.
2. Average Polynomial Regression provides a stable predictor of market conditions.
3. Markov Chains allow for us to represent the trends in historical data efficiently and accurately.

In testing we find this system has *92.06%* accuracy and takes *0.358* ms for each evaluation. This manifests *40.72%* annual returns in backtesting.

7 References

This project makes ample use of the resources listed below.

1. Van Rossum, G., & Drake, F. L. (2009). Python 3 Reference Manual. Scotts Valley, CA: CreateSpace.
2. Lay, David C., et al. "4.9 Applications to Markov Chains." Linear Algebra and Its Applications, Pearson, Boston, MA, 2016.
3. Krzywinski, M., Altman, N. Classification and regression trees. Nat Methods 14, 757–758 (2017). https://doi.org/10.1038/nmeth.4370
4. arXiv:1111.5254 [q-fin.ST]
5. Don L. McLeish, and Adam W. Kolkiewicz. "Fitting Diffusion Models in Finance." Lecture Notes-Monograph Series, vol. 32, 1997, pp. 327–50, http://www.jstor.org/stable/4356025. Accessed 27 Apr. 2022.
6. Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2.
7. Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

Disclosure

No funding was required for this paper. This paper has not been influenced by any parties other than the other. The Author claims to be the sole contributor to this project.

Source Code and Training Data

The Source Code, Saved ML Models, and wordlists are available for free on Github licensed under a Creative Commons Attribution-ShareAlike 4.0 International Public License

`https://github.com/s-spektrum-m/trader`

Authors

1. Siddharth Mohanty, Independent

Sentiment Analysis

```python
from robin_stocks import robinhood as rh
from json import load, dump

_l = lambda a: load(open(f"{a}.json", "r"))
```

Extracting Data

We extract data from the robinhood news and price api and place it in the following class:

```
"ticker": [
    article1,
    article2,
    article2,
    ...
]
```

```python
class Stock:
    def __init__(self, ticker, articles):
        self.ticker = ticker
        self.articles = articles
```

```python
stocks = _l('stocks')

def articles(ticker: str) -> Stock:
    news = rh.stocks.get_news(ticker)
    s = [n['title'] for n in news]
    return Stock(ticker, s)

today = {}
for stock in stocks:
    today[stock] = articles(stock).articles
```

Raw Data to a Wordlist

Once we acquire our raw data, we perform the following algorithm to arrive at our wordlist:

Load all words found in the news for a stock, s, on day i. If the value of s goes up on day $i + 1$ increment all words pretaining to s by 1 else decrement by 1. Add 1 to the occurences count every time the value is mutated.

$$W : \{l, \sum_{n=0}^{l} M_n\}$$

Where W is the word and M is the set of associated movements. We later refine our wordlist into the following

$$W : \frac{\sum_{n=0}^{l} M_n}{l}$$

We dump this to a json file.

```
wl = {}
prices = _l('prices')

for ticker, article_list in zip(today.keys(), today):
    for article in article_list:
        words = article.split()
        for word in words:
            if word not in wl:
                wl[word] = [0, 0]
            wl[word][0] += 1
            wl[word][1] += prices[ticker]

for word in wl:
    word = word[1] / word[0]
```

Querying the Wordlist

Since we use a json file for our wordlist, we can load it as a python dictionary where the words are the keys.

```
wl = _l('wl')

def query(word):
    return wl[word]
```

```
query("Good")
```

```
3.0
```

Polynomial Regression for Share Price Analysis

Imports and Config

```python
import os
import numpy as np
from matplotlib import pyplot as plt
from auth import USERNAME, PASSWORD
from robin_stocks import robinhood as rh
```

Historicals Function

Returns a python list of floats representing the price of `ticker` in the last month.

```python
def _historicals(ticker) -> list:
    prices = rh.stocks.get_stock_historicals(ticker, span="month")
    if prices is not None:
        return [float(price['open_price']) for price in prices]
    return [None]
```

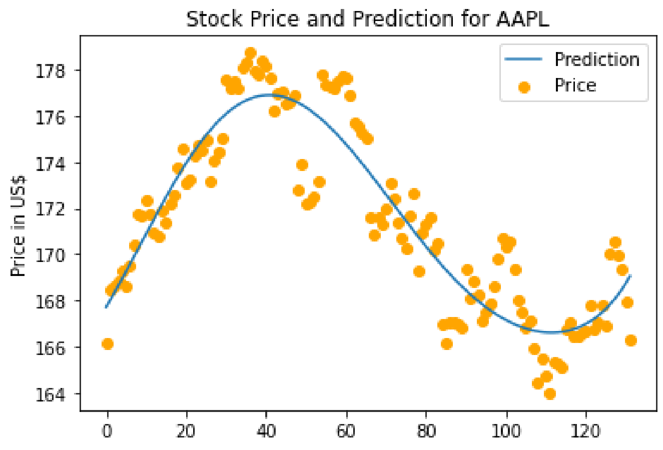
Polynomial Prediction Function

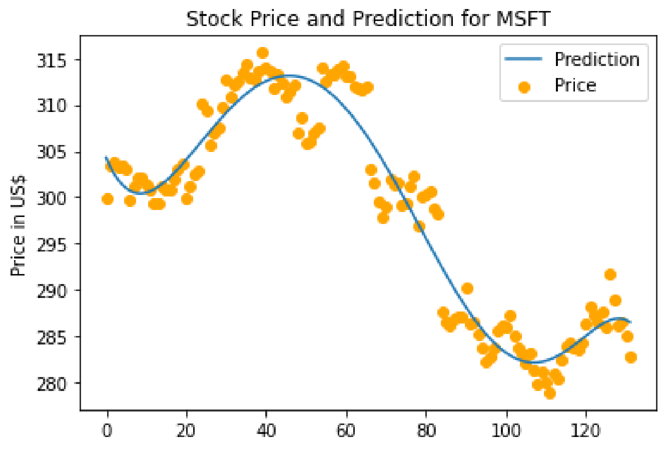
Creates polynomials that fit the data ranging from `deg=5` to `deg=8` to increase accuracy while decreasing overfitting

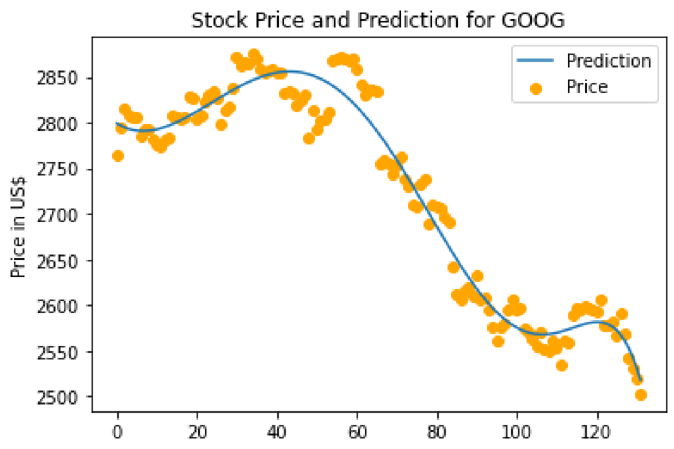
```python
def _poly_fit(args, x) -> float:
    return np.poly1d(args)(x)

def _poly_pred(hist, pred_day) -> float:
    y = hist
    x = np.arange(len(y))
    preds = []
    for i in range(5,  8):
        args = np.polyfit(x, y, i)
        preds.append(_poly_fit(args, pred_day))
    return sum(preds)/len(preds)

def pred(ticker) -> float:
    hists = _historicals(ticker)
    return round(_poly_pred(hists, len(hists) + 1) / float(rh.stocks.get_latest_price(ticker)[0])) - 1
```

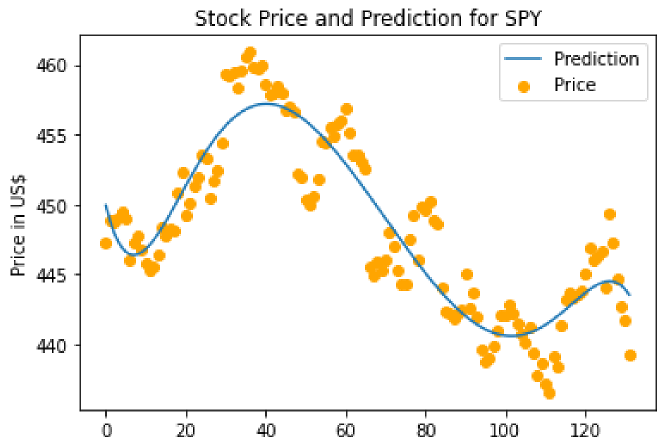
```python
def graph(stock):
    y = _historicals(stock)
    x = np.arange(len(y))
    prediction = [_poly_pred(y, i) for i in x]
    plt.plot(x, prediction)         # Prediction
    plt.scatter(x, y, c="Orange")   # Actual
    plt.title(f'Stock Price and Prediction for {stock}')
    plt.ylabel('Price in US$')
    plt.legend(['Prediction', 'Price'])
    plt.savefig(f'{stock}.png')
    plt.show()
```

```python
graph('AAPL')
graph('MSFT')
graph('GOOG')
graph('SPY')
```






Stock Price and Prediction for GOOG
Prediction
Price
Price in US$
2500
2550
2600
2650
2700
2750
2800
2850
0
20
40
60
80
100
120


Stock Price and Prediction for SPY
Prediction
Price
Price in US$
440
445
450
455
460
0
20
40
60
80
100
120

Evaluating the Steady State of a Markov Chain

q, the steady-state vector of a markov chain, $\mathbf{P}$ is defined as:

$$\mathbf{P}q = q$$

This is quite simmilar to the definition of an eigenvector

$$\mathbf{A}x = \lambda x$$

So, we can simply find an eigenvector of P, x and then find q with the following formula:

$$q = \frac{x}{\sum_{n=1}^{3} x_n}$$

This ensures the sum of all elements is 1 The function returns

$$q_1 - q_2$$

```python
import numpy as np
from numpy import linalg as la
```

```python
def eig(markov):
    for i in la.eig(markov)[1]:
        if (i[0] >= 0 and i[1] >= 0 and i[2] >= 0) or (i[0] < 0 and i[1] < 0 and i[2] < 0):
            return i

def steady_state_eval(markov):
    sol = eig(markov)
    l = sum(sol)
    new = [i/l for i in sol]
    return np.real(new[0]/new[1])

P = np.array([
    [0.8, 0.15, 0.45],
    [0.1, 0.5, 0.05],
    [0.1, 0.35, 0.5]
])

steady_state_eval(P)
```

```
1.2966159592818978
```