Designing and implementing a Hidden Markov Model to predict stock prices

Brandon Sandhu

July 19, 2024

Contents

1 Abstract **1**

2 Theory **1**

2.1 Markov Chains to Hidden Markov Models 1

2.2 Optimising a Hidden Markov Model 3

3 Implementation **4**

3.1 Code Explanation 4

3.2 Backtesting Results 7

1 Abstract

Our work is primarily inspired by Renaissance Technologies, which aimed to develop systematic trading strategies using quantitative models based on mathematical and statistical analysis. Founded in 1982 by mathematician and philanthropist James Harris Simons, Renaissance Technologies had its roots in Simons' experience as a code breaker during the Cold War. Many of his initial recruits for the firm were colleagues from his code-breaking days.

The Medallion Fund, renowned for its record-setting investment returns, has achieved an average annual return of 62% from its inception in 1988 until 2021. Although the specific strategies employed by Medallion remain undisclosed due to perpetual non-disclosure agreements signed by its employees, we can still make some educated guesses about their methods.

Given that Simons was a mathematical researcher for much of his life, his approach centered around research. Consequently, many of his recruits were academics holding PhDs or other advanced qualifications. One of the first recruits was Leonard Baum, who was with the company for only two years. Baum's joint development of the Baum-Welch algorithm sparked the initial types of models used at Renaissance Technologies. The Baum-Welch algorithm is a type of expectation-maximising algorithm that finds the most probable parameters of a Hidden Markov Model. By recognising this connection between Baum and Renaissance, we begin with the theory of Hidden Markov Models and explore how they can be used to forecast stock market prices.

2 Theory

We begin by describing the general setup of a Hidden Markov Model and how it can be used to forecast future events. We follow the approach outlined in Aditya Gupta [1] but may include further details for a sounder understanding.

2.1 Markov Chains to Hidden Markov Models

A Markov chain is a stochastic model that describes a sequence of possible events, assuming that the probability of each event depends solely on the state of the preceding event. More formally,

Definition 2.1. A *Markov chain* is a sequence $\{X_0, X_1, X_2, \dots\}$ of random variables, which one thinks of as events in the above definition, that satisfies the *Markov property* which states

$$\mathbb{P}(X_{n+1} = x_{n+1} \mid X_0 = x_0, X_1 = x_1, \cdots, X_n = x_n) = \mathbb{P}(X_{n+1} = x_{n+1} \mid X_n = x_n)$$

for all $n \geq 0$, where $\{x_0, x_1, \dots, x_{n+1} \in S$ where S is the state space. It should be clear this Markov property clearly encodes the idea that the probability of an event being in a certain state depends only on the state of the previous event.

In the context of the stock market, trends generally follow historical data, with more recent data weighing more heavily in predicting immediate future movements. Thus, we can see how the Markov property could be a valuable tool for modeling stock prices.

We can visualise a Markov chain as shown in Figure 1.

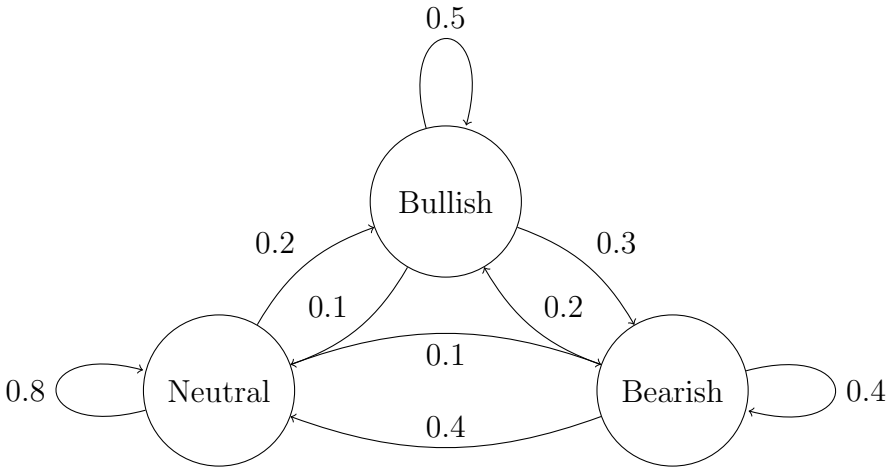

Figure 1: An example of a Markov chain

We categorize market conditions into three states: bullish, bearish, and neutral, often referred to as observation states. These states can be identified using technical indicators like the relative strength index (RSI). When leveraging a Markov chain to forecast the next likely state, assuming the Markov property holds, we simply follow the arrow leading to the highest probability from the current state.

We can organize these probabilities into a matrix format. Each row of the matrix represents the current state, and each column represents the next state. Assuming our state order as Bearish, Bullish, and Neutral, the corresponding matrix would appear as follows,

$$\begin{pmatrix} 0.4 & 0.2 & 0.4 \\ 0.3 & 0.5 & 0.1 \\ 0.1 & 0.2 & 0.8. \end{pmatrix}$$

This matrix is known as the *transition matrix*. It encapsulates the probabilities of moving from one state to another, assuming adherence to the Markov property.

In our current Markov model, all states are observable. However, we can enhance this model by introducing hidden states—variables that are not directly observed. These hidden states can represent various factors such as market sentiment or hypothesized influences on the market. The key requirement is to determine the number of these hidden states.

A hidden Markov model (HMM) aims to optimise the weighting of each hidden state based on observed data patterns, typically through machine learning techniques. Once the model is trained and optimised, similar to a regular Markov chain, we can use it to predict future states in the market. We can visualise a hidden Markov model as shown below in Figure 2.

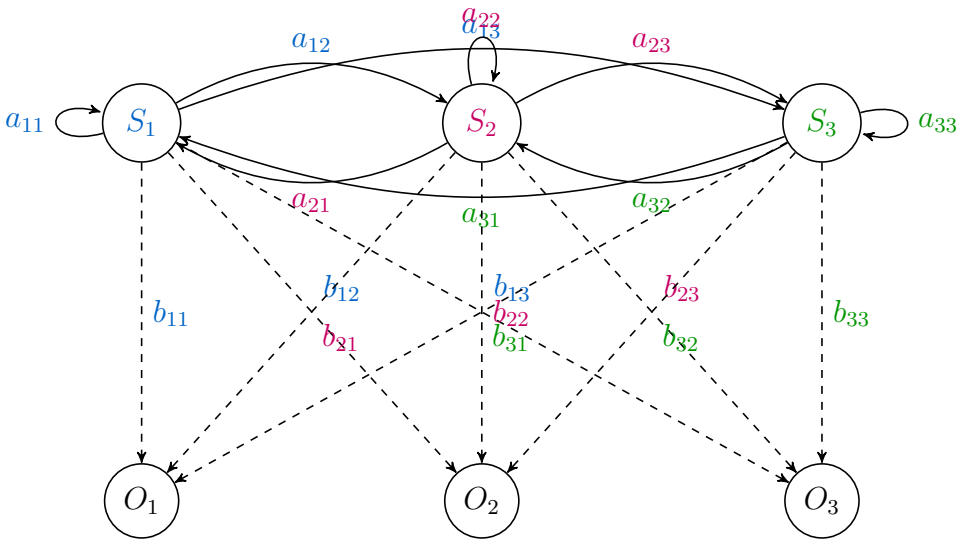

Figure 2: An example of a hidden Markov model

The observation states are denoted by O_i and the hidden states are denoted by S_i.

The transition matrix, as previously defined with an ordering of S_1, S_2, S_3, is represented as follows,

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33}. \end{pmatrix}$$

In the hidden Markov model, our objective is to optimise the transition matrix based on the observation data O_i. Additionally, the model incorporates hidden states and emission probabilities, depicted as dotted arrows. These emission probabilities signify the likelihood of observing O_i when the hidden state is S_i. We can package these into a matrix too as follows,

$$\begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}.$$

The matrix is called the *emission matrix*. In this matrix, each row corresponds to a specific hidden state S_i, while each column corresponds to an observed state O_i. In a typical hidden Markov model, we also establish initial probabilities, which denote the likelihood of each observed state occurring from a starting hidden state. However, in our specific case, we do not need to define this because we are utilising historical stock data, which inherently serves as our starting point.

Therefore, our hidden Markov model is completely determined by the pair (A, B) where A denotes the transmission matrix and B denotes the emission matrix.

2.2 Optimising a Hidden Markov Model

Currently, our hidden Markov model is represented by the pair $H = (A, B)$. However, before it can effectively predict based on observed data, the model requires training. This is where the

Baum-Welch algorithm becomes essential. Known as an expectation-maximization algorithm, its formal objective is to find a local maximum for

$$\theta^* = \text{argmax}_A \mathbb{P}(Y \mid H).$$

For detailed implementation specifics of the Baum-Welch algorithm, you can visit its Wikipedia page at `https://en.wikipedia.org/wiki/Baum-Welch_algorithm`. Essentially, the algorithm is an iterative parameter optimisation method. In practice, Python packages are available that automate the application of this algorithm, simplifying its implementation.

3 Implementation

We now use Python to implement and train a hidden Markov model to predict the next ticks price given historically observed data.

3.1 Code Explanation

As demonstrated in the accompanying Jupyter notebook, we've developed a data scraper that retrieves stock price data from TradingView. This scraper generates a CSV file containing information on open, high, low, close, and volume data. We've encapsulated this functionality in a function named `GetStockData`.

We begin by importing some key modules

```
import numpy as np
from hmmlearn.hmm import GaussianHMM
import pandas as pd
import matplotlib.pyplot as plt
from io import StringIO
```

`numpy` is a widely-used package for manipulating arrays of data.

`hmmlearn` specialises in hidden Markov models (HMMs), offering a Gaussian HMM variation that models observations (like stock prices) as Gaussian distributions.

`pandas` is a versatile package for managing data frames with multiple columns. It simplifies tasks like displaying large datasets in a user-friendly format and supports basic statistical operations and data manipulations such as differencing.

`matplotlib` is used for plotting our results.

`StringIO` is specifically employed to parse CSV data into a pandas dataframe, enhancing data handling capabilities.

```
data = pd.read_csv(StringIO(GetStockData("NASDAQ:AAPL", 1, 210)[0]))
```

The provided code retrieves stock data for Apple over a 3-year period on a daily basis. It captures the open, close, high, low and volume data. In our model we will consider all but volume data.

The exclusion of volume data immediately highlights a need for improvement, as volume plays a crucial role in understanding current stock prices.

```
    def features(dataframe):
        CloseOpen = (dataframe['Close']-dataframe['Open'])/dataframe['Open']
        HighOpen = (dataframe['High']-dataframe['Open'])/dataframe['Open']
        OpenLow = (dataframe['Open']-dataframe['Low'])/dataframe['Open']
        NewDataframe = pd.DataFrame({'OpenClose': CloseOpen,
                                     'HighOpen': HighOpen,
                                     'LowOpen': OpenLow})
        NewDataframe.set_index(dataframe.index)
        return NewDataframe

    def formatFeatures(dataframe):
        return np.column_stack((dataframe['OpenClose'], dataframe['HighOpen'
], dataframe['LowOpen']))
```

The function `features()` computes fractional changes in market prices. Specifically, it calculates the percentage change between pairs of variables: (Close, Open), (High, Open), and (Low, Open). These changes are then organised into a new dataframe, which is subsequently converted into an array format using the `formatFeatures()` function. This formatted array is compatible with `hmmlearn`, facilitating its use in our hidden Markov model.

Another potential area for improvement could involve incorporating additional features to assess whether they enhance the accuracy of our model.

```
    trainSize = int(0.8*data.shape[0])

    trainData = data.iloc[0:trainSize]
    testData = data.iloc[trainSize+1:]

    trainFeatures = features(trainData)
    trainFeaturesFormatted = formatFeatures(trainFeatures)
    model = GaussianHMM(n_components=10)
    model.fit(trainFeaturesFormatted)
```

This code snippet forms the core of our models training phase. Initially, we allocate 80% of our dataset for training the model, reserving the remaining data for future back-testing. Using previously defined functions, we extract observational features from the training data and proceed to construct a hidden Markov model with 10 hidden states. The `model.fit` function then employs the Baum-Welch expectation maximisation algorithm to optimise the transition probabilities among these states.

Choosing an appropriate number of hidden states is crucial; too many may lead to overfitting due to increased complexity and the need for more extensive training data.

An area for potential improvement involves experimenting with different numbers of hidden states to ascertain which configuration yields more accurate predictions of stock prices. Furthermore, while interpreting the meaning of each hidden state in market terms is challenging due to numerous influencing factors, exploring these associations could provide deeper insights into market dynamics.

```
    import itertools

    OpenClose = trainFeatures['OpenClose']
    HighOpen = trainFeatures['HighOpen']
    OpenLow = trainFeatures['LowOpen']

    SampleSpaceOC = np.linspace(OpenClose.min(), OpenClose.max(), 50)
    SampleSpaceHO = np.linspace(HighOpen.min(), HighOpen.max(), 10)
    SampleSpaceOL = np.linspace(OpenLow.min(), OpenLow.max(), 10)

    possibleOutcomes = np.array(list(itertools.product(SampleSpaceOC,
SampleSpaceHO, SampleSpaceOL)))
```

Since our observation variable is continuous, we encounter the issue where $\mathbb{P}(\text{Price} = x_1) = 0$ with $x_1 \in [a, b]$. To address this, we need to discretise our observation variables. The code above handles this discretisation process, giving greater priority to the (Open, Close) percentage increase over the other two observation variables, as this is the key aspect we aim to model. By predicting an accurate percentage increase, we can calculate the next tick close price using the formula `Open Price × Predicted Percentage Increase = Next Tick Close Price`. The `itertools` package is utilised to perform a Cartesian product of all the observation variables, generating a finite set of possible outcomes. In the code above, there will be 5000 such outcomes.

```
    numLatentDays = 10
    numDaysToPredict = 20
```

We define two key variables for our prediction model. The first is *latency*, which determines the number of preceding days to include in our observation sequence. This sequence is used to run the model and predict the subsequent output. The second variable is `numDaysToPredict`, which specifies the number of days to backtest within the test data.

```
    from tqdm import tqdm

    predictedClosePrices = []
    for i in tqdm(range(numDaysToPredict)):
        # Calculate start and end indices
        previousDataStartIndex = max(0, i - numLatentDays)
        previousDataEndIndex = max(0, i)
        # Acquire test data features for these days
        previousData = formatFeatures(features(testData.iloc[
previousDataStartIndex:previousDataEndIndex]))

        outcomeScores = []
        for outcome in possibleOutcomes:
            # Append each outcome one by one to see which sequence generates
 the highest score
            totalData = np.row_stack((previousData, outcome))
            outcomeScores.append(model.score(totalData))

        # Take the most probable outcome as the one with the highest score
        mostProbableOutcome = possibleOutcomes[np.argmax(outcomeScores)]
        predictedClosePrices.append(testData.iloc[i]['Open'] * (1 +
mostProbableOutcome[0]))
```

This is our backtesting code. The module `tqdm` is used for debugging purposes; it displays the progress of the for loop, as this process can be time-consuming. We iterate through

the `numDaysToPredict` variable, setting `previousData` to contain the features from the last `numLatentDays` of the test data. By applying the model, which has been trained on historical patterns, to this specific dataset, we can make accurate predictions for the subsequent days closing price. We then iterate through each possible outcome of the closing price and append the one with the highest probability, effectively selecting the maximum emission from all previously computed observation variables. Finally, the predicted close price is calculated by multiplying the current day's open price by the predicted (Close, Open) fractional change.

```
    plt.figure(figsize=(30,10), dpi=80)
    plt.rcParams.update({'font.size': 18})

    xAxis = np.array(testData.index[0:numDaysToPredict], dtype='datetime64[
    ms]')
    plt.plot(xAxis, np.array(testData.iloc[0:numDaysToPredict]['Close']), 'b
    +-', label="Actual close prices")
    plt.plot(xAxis, predictedClosePrices, 'ro-', label="Predicted close
    prices")
    plt.legend(prop={'size': 20})
    plt.show()
```

Here, we plot our predicted closing prices against the true closing prices for each day. We observe a lag between the actual and predicted prices, indicating that the model struggles to predict trend turning points before they occur. This suggests a limitation in our model. However, the model successfully follows the overall trend.

From this, we can conclude that while the model is not reliable for predicting specific stock prices, it is effective in forecasting the general trend of a stock. This trend-following capability could be valuable when used alongside other metrics to forecast future stock prices.

```
    ae = abs(testData.iloc[0:numDaysToPredict]['Close'] -
    predictedClosePrices)

    plt.figure(figsize=(30,10), dpi=80)

    plt.plot(xAxis, np.array(ae), 'go-', label="Error")
    plt.legend(prop={'size': 20})
    plt.show()
```

The above plot displays the absolute difference between the true and predicted prices, allowing us to see how far off the models predictions are. Generally, the peaks in this plot correspond to areas where the trend is changing. By minimising the errors at these peaks, we can reduce the model's inaccuracy during trend changes.

3.2 Backtesting Results

The date we collected this data is $18/07/2024 at 7:20 GMT$. We select the following parameters: a 64-day time period for our testing data with 5-minute tick intervals. We train on the first 80% of the dataset and backtest on the remaining 20%. Due to running time constraints, we backtest on the first 300 ticks of our testing dataset, which corresponds to 25 hours of trading time. We use a latency of 25 ticks, equating to approximately 2 hours. Initially using 10 hidden states, we produce the following result:

We also plot the absolute error differences

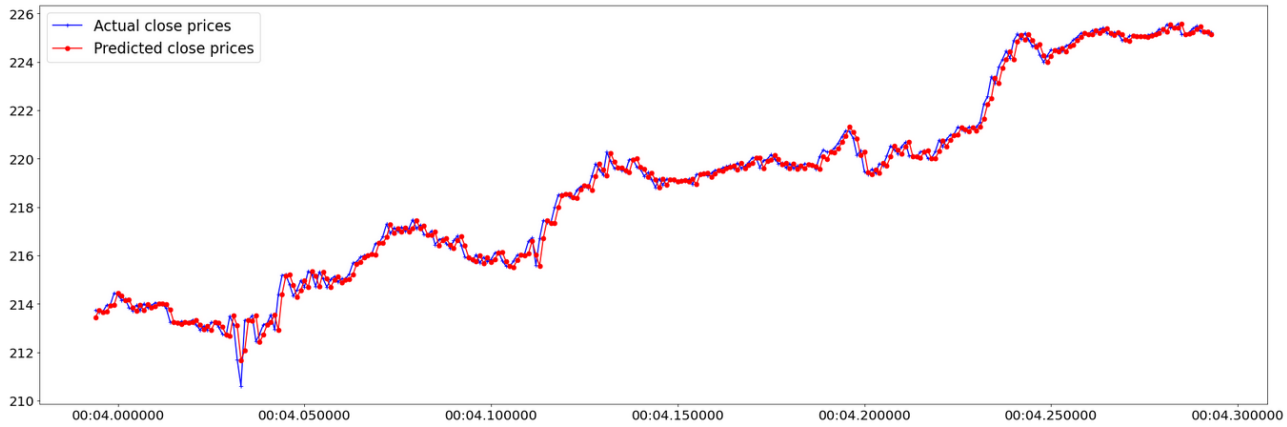

Figure 3: Output of predictive model with parameters described above

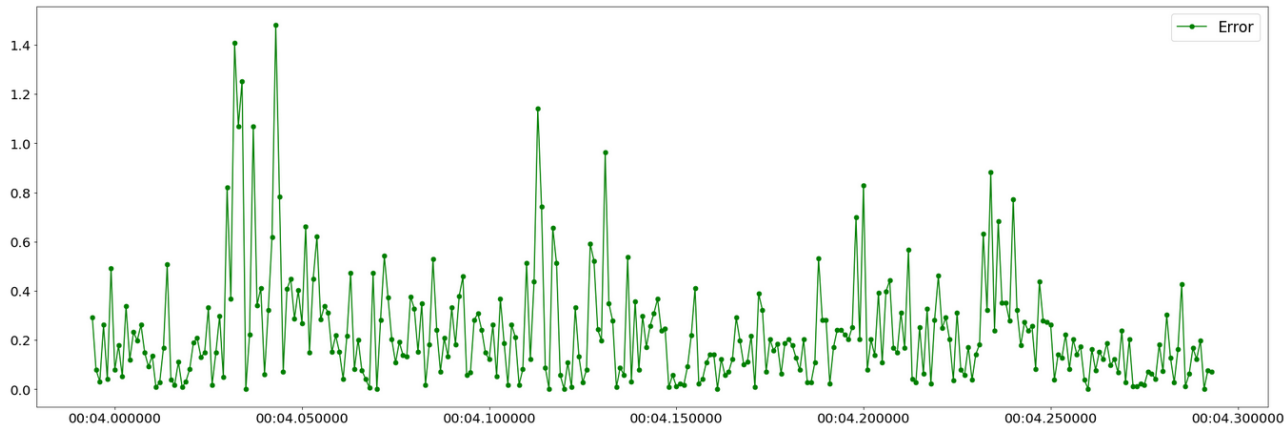

Figure 4: Error of predictive model with parameters described above

We observe three main peak regions. Given that each trading day on the NASDAQ stock exchange spans 6 hours and 30 minutes, we expect roughly three trading days to fit into the 25-hour backtesting window. These peaks correspond to the sudden price changes that occur at the start of each trading day. Observe the errors during the trading days are relatively low so we can instantly conclude that volatility of this stock is highest at the start of a trading day. We tabulate the following error metrics

Max Error	Min Error	Average Error
0.237	0.013	0.113

References

[1] Bhuwan Dhingra Aditya Gupta. Stock market prediction using Hidden Markov Models. 2012.