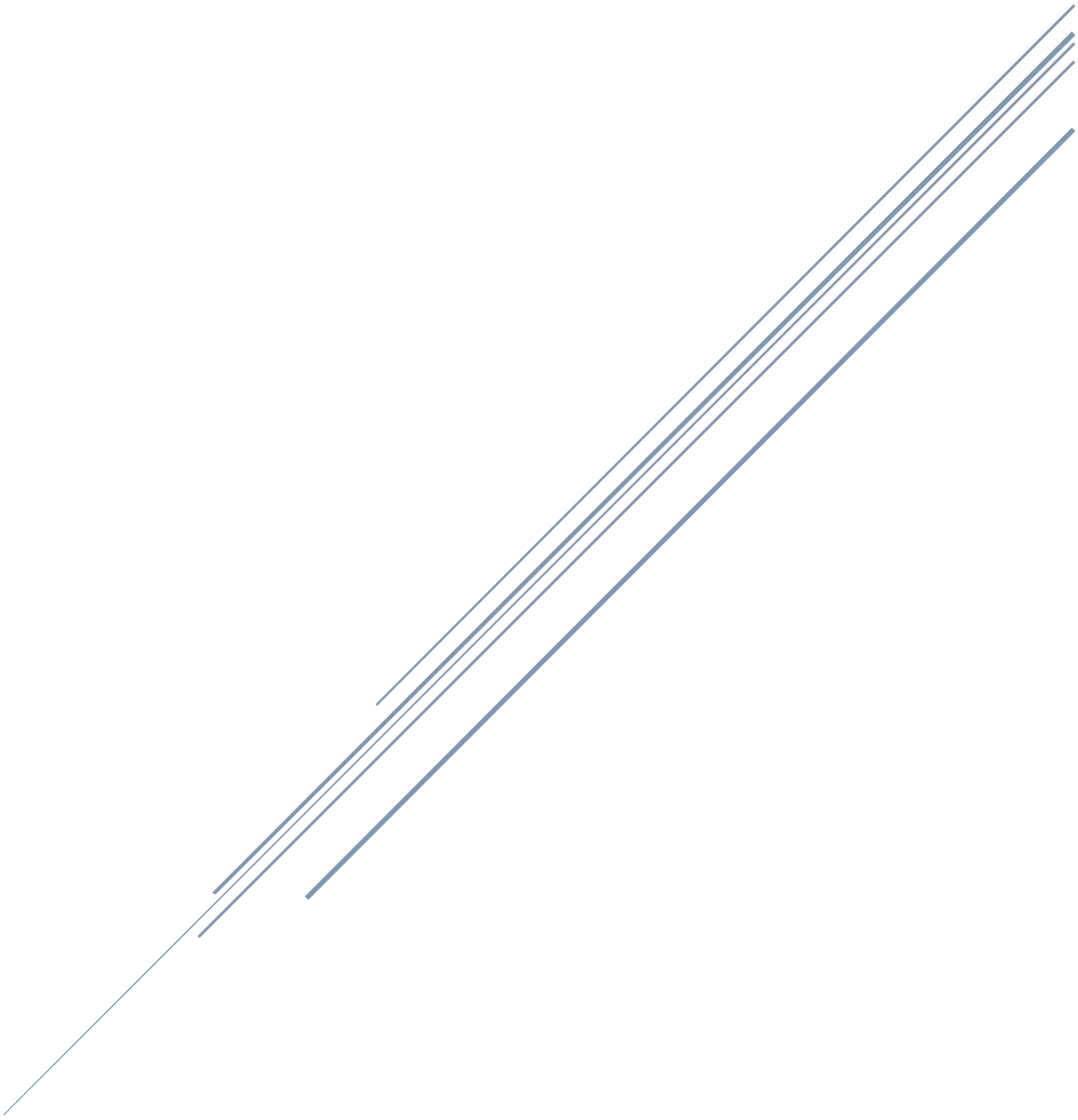
FNCE30010: Algorithmic Trading
Group Assignment

Name	Student ID Number	Tutor	Tutorial Day & Time	Tutorial Location
Joel Thomas	**915951**	**Nitin Yadav**	**Friday 2:00 pm**	**The Spot 3010**
Wenlin Zhang	**813033**	**Nitin Yadav**	**Friday 2:00 pm**	**The Spot 3010**
Liam Horrocks	**912386**	**Nitin Yadav**	**Friday 2:00 pm**	**The Spot 3010**

THE HIDDEN MARKOV MODEL

Application of Machine Learning to Algorithmic Trading



Joel Thomas, Victoria Zhang and Liam Horrocks

FNCE30010: Algorithmic Trading

Table of Contents

Abstract 3

Introduction 3

The Hidden Markov Model 3

Key Assumptions in HMM 4

Three Fundamental Problems of HMMs 4

Gaussian Mixture Models (GMMs) 4

Out of Sample Prediction with HMM 5

Original HMM Trading Strategy 5

Results and Discussion 5

Improvised HMM Trading Strategy 7

Results and Discussion 8

General Weaknesses of HMM for Asset Price Prediction 10

Alternative Ideas 11

Conclusion 12

References 12

Appendix 13

Detailed description of the three HMM algorithms 13

Mathematical overview for training Scaled Continuous HMM with Single Observation Sequence 14

Quantopian Code 15

Original Strategy 15

Improvised Strategy 23

Abstract

The purpose of this report is to analyse both the profitability and feasibility of the Hidden Markov Model (HMM) applied to the context of algorithmic trading. Initially, the theory of the HMM model is discussed briefly before explaining the Gaussian Mixture Model as an extension to model continuous data such as stock prices. Next, the prediction method based on this model is explained in detail. The original trading strategy used the HMM solely for raw price prediction, however given the weak results, improvised strategy used HMM as a secondary confirmation signal strategy combined with a momentum filter for stock selection. It was discovered that both strategies fail over the long run. Finally, general weaknesses of the model were discussed, and alternative ideas were raised for the reader to consider and investigate.

Introduction

The Hidden Markov model (HMM) is an Unsupervised Machine Learning algorithm which is an area of machine learning that learns from unlabelled, unclassified or uncategorised test data (Saslow, 2018). HMMs are an extension to Markov chains, which are described as a method to probabilistically model random processes. It is a mathematical system that governs transitions of a random process from one state to another using probability (Maltby). The difference between a Markov model and an HMM is the inclusion of 'hidden' or 'latent' variables which are defined as processes that occur besides what is observed or measured.

An HMM can be used as a trading strategy. Stock prices for example, can be treated as continuous observations in which are susceptible to several real-world factors. However, it is not possible to quantitatively enumerate all these factors perfectly. A potential remedy for such a situation is the adoption of a latent variable model because it can be used to predict the hidden states of an HMM. Therefore, after observing a given sequence of stock prices, it is possible to work backwards to find the most likely hidden state sequence, allowing one to subsequently model next periods most likely observation (e.g. tomorrow's stock price).

The Hidden Markov Model

An HMM used to model **discrete** observations is traditionally described by the following parameters:

Elements	Notation/Definitions
Length of observation data	T
Number of states	M
Number of symbols per state	K
Observation sequence	$X = \{x_1, x_2, \dots, x_T\}$
Hidden state sequence	$Z = \{z_1, z_2, \dots, z_T\}$
Possible values of each state	$z = \{1,2, \dots, M\}$
Possible symbols per state	$\{1,2, \dots, K\}$
Initial state distribution	π
State transition matrix	A
Emission probability matrix	B
Set of HMM parameters	$\lambda = \{\pi, A, B\}$

Table 1: Variables associated with discrete HMM

The set of HMM parameters have the following definitions:

$\pi_i =$ probability of starting at state i, also known as Initial State Distribution

$A_{ij} =$ probability of transitioning from state i to state j, also known as State Transition Matrix

$B_{jk} =$ probability of observing symbol k in state j, also known as Emission Probability Matrix

The parameters π and A are applicable to both Markov models and HMMs but B is only applicable to HMMs. The difference between a Markov model and an HMM becomes clearer as A_{ij} is defined for observed states in a Markov chain whereas A_{ij} is defined for hidden states in an HMM.

Key Assumptions in HMM

The three main assumptions of an HMM are (Warakagoda, 1996):

- The Markov assumption:

  Assumes that the Markov property (i.e. the memoryless property of a stochastic process) holds for the realisation process of hidden states:

  $\forall i,j \in \{1,2,\dots M\}, \forall t \in \{1,2,\dots T\}, \qquad A_{ij} = p(z_{t+1} = j | z_t = i)$

- The stationarity assumption:

  Assumes that the hidden state transition probabilities remain constant regardless of the actual time step where the transition occurs:

  $\forall i,j \in \{1,2,\dots M\}, \forall t \in \{2,3,\dots T-1\}, \qquad p(z_{n+1} = j | z_n = i) = p(z_n = j | z_{n-1} = i)$

- The output independence assumption:
  Assumes the observation generated at each time step is independent of the realised observation at the previous step:
  Given the observation sequence $X = \{x_1, x_2, \dots, x_T\}$, and the parameter set λ

  $$P(X | z_1, z_2, \dots, x_T) = \prod_{t=1}^{T} P(x_t | z_t, \lambda)$$

Three Fundamental Problems of HMMs

Traditionally there are three fundamental problems associated with HMMs (Rabiner, 1989):

1. Likelihood computation: Finding the probability of an observation sequence $p(X|\lambda)$ where X is the observation sequence. This can be solved using Forward algorithm (appendix).
2. Decoding: Finding the most likely sequence of hidden states given an observation sequence. This can be solved using Viterbi algorithm (appendix).
3. HMM training: Given an observation sequence and number of possible hidden states, find the HMM parameters λ. HMM training is usually performed using the Baum-Welch algorithm (appendix).

Gaussian Mixture Models (GMMs)

As stock prices are continuous, instead of having a finite number of symbols as in the discrete case, a continuous distribution should be applied because there are infinitely many symbols. Assuming movements in stock prices can be represented by a univariate Gaussian distribution is a strong assumption, hence it might fail to represent characteristics such as high levels of kurtosis or multimodal peaks which appear frequently in the real world. As a result, the use of a Gaussian Mixture Model is proposed. The GMM is defined as a sum of weighted multivariate gaussians:

$$p(x) = R_1 N(x_t; \mu_1, \Sigma_1) + R_2 N(x_t; \mu_2, \Sigma_2) + \cdots + R_k N(x_t; \mu_k, \Sigma_k) = \sum_{i=1}^{k} R_i N(x_t; \mu_i, \Sigma_i)$$

$R_i =$ the probability that a given x_t belongs to the ith multivariate Gaussian
$N(x_t; \mu_i, \Sigma_i) =$ the PDF obtained from a multivariate Gaussian of x_t with mean μ_i and covariance Σ_i
Given the case of continuous data, the emission probabilities ($B_j(k)$) must be replaced by a different set of parameters $\{R, \mu, \Sigma\}$.

Out of Sample Prediction with HMM

This section will discuss the methodology used to incorporate out of sample forecasts using HMMs (Hassan et al., 2005). In order to forecast at time $T+1$, the following steps are required:

1) Let the window over which the HMM is trained be denoted by D i.e. HMM's parameters $\{\pi, A, R, \mu, \Sigma\}$ are trained over the time period $T-D$ to T
2) Using the trained parameters:
    - Recalculate the newest B (since $\{R, \mu, \Sigma\}$ are updated after B at the end of the final iteration, need to re-estimate B again)
    - Then calculate the forward variable α (forward algorithm)
    - At the termination step, obtain the log-likelihood of the observation sequence $\log p(X|\lambda)$ using the trained parameters (see appendix)
3) Shift the window of data backwards by 1 unit such that the new time period is from $T-D-1$ to $T-1$
4) Using the trained parameters, perform step 2) to calculate the log-likelihood of this new observation sequence defined $\log p(X^*|\lambda)$
5) Record the new log-likelihood if it is reasonably close to the training period log-likelihood e.g. if $< \epsilon = 1e^{-1}$
6) Iterate steps 3) to 5) until window has been shifted all the way to time period $T-D-(T-D)=0$ to $T-(T-D)=D$
7) Find the highest similar log-likelihood in the recorded log-likelihoods such that $\log p(X|\lambda) \approx \log p(X^*|\lambda)$
    - Denote this time period as $T-D-H$ to $T-H$
8) The final prediction of the asset's price at time $T+1$ is given by:
    - $\hat{x}_{T+1} = x_T + (x_{T-H+1} - x_{T-H}) \times sign(\log p(X|\lambda) - \log p(X^*|\lambda))$

Original HMM Trading Strategy

In this section, the basic trading strategy is discussed. The end goal is to assess the raw prediction power of the HMM as a standalone trading strategy. The selected asset to trade is the SPY S&P 500 ETF, based on the idea that it best represents overall market performance and will likely never face survivorship bias. The back-test period runs from January 4th, 2005 to October 18th, 2019. At the start of every week, the HMM model discussed above is trained with 2 hidden states ($M=2$) and 2 Gaussians ($K=2$) based on the past 6 weeks of dividend adjusted (SPY pays out dividends) data resampled from daily to weekly prices ($D=6$ data points). The values for M and K are chosen to capture the minimum bounds for both these variables under which it is still possible to train the HMM.

Based on the out of sample prediction method discussed above, the data shifts 1 week behind in every iteration for a total of 3 years of historical data, the reason being that a history greater than might not be all that relevant (Figlewski, 1994). Conversely, the shorter the window, the fewer amount of pertinent observations HMM has for prediction.

Once the most similar past log-likelihood is obtained, the trained HMM predicts next week's closing price for the ETF. Only if the next weeks closing price prediction is above the last price on the ETF, a position size of 100% of the portfolio value is opened via buying the SPY-ETF if the algorithm currently has no open position. However, if the trading position is already open, the algorithm keeps it as such. Conversely, the opposite holds true if HMM predicts next week's closing price to drop. Furthermore, short selling is avoided as it greatly increases the risk of opening a position when the HMM predicts wrong.

Results and Discussion

As of January 1st, 2005, the 10-year treasury rate was 4.23% and as of January 1st, 2015, the 5-year treasury rate was 1.61% (Department of the Treasury, 2019). Hence the estimated risk free over the back-test period is:

$$r_f = [(1+0.0423)^{10}(1+0.0161)^5]^{\frac{1}{15}} - 1 = 0.033 \text{ or } 3.3\%\ p.a.$$

Back-test Results for Original Strategy	
Start date	04-01-2005
End date	18-10-2019
Total months	177
Annual return	6.4%
Cumulative returns	148.9%
Annual volatility	9.2%
Unadjusted Sharpe ratio of strategy (daily $r_f = \frac{0}{252}$)	0.72
Adjusted Sharpe ratio of strategy (daily $r_f = \frac{0.033}{252}$)	0.36
Unadjusted Sharpe ratio of benchmark (daily $r_f = \frac{0}{252}$)	0.54
Adjusted Sharpe ratio of benchmark (daily $r_f = \frac{0.033}{252}$)	0.36
Max drawdown	-23.5%
Alpha	0.04
Beta	0.25
$p - val$ for $t - stat$ for $H_0: \frac{r_t - r_{t-1}}{r_{t-1}} = 0$	0.006
Transaction cost per share traded (Kalyan, 2017)	$0.001

Table 2: Results of back-test

Figure 1: Cumulative returns of strategy vs. benchmark over back-test period

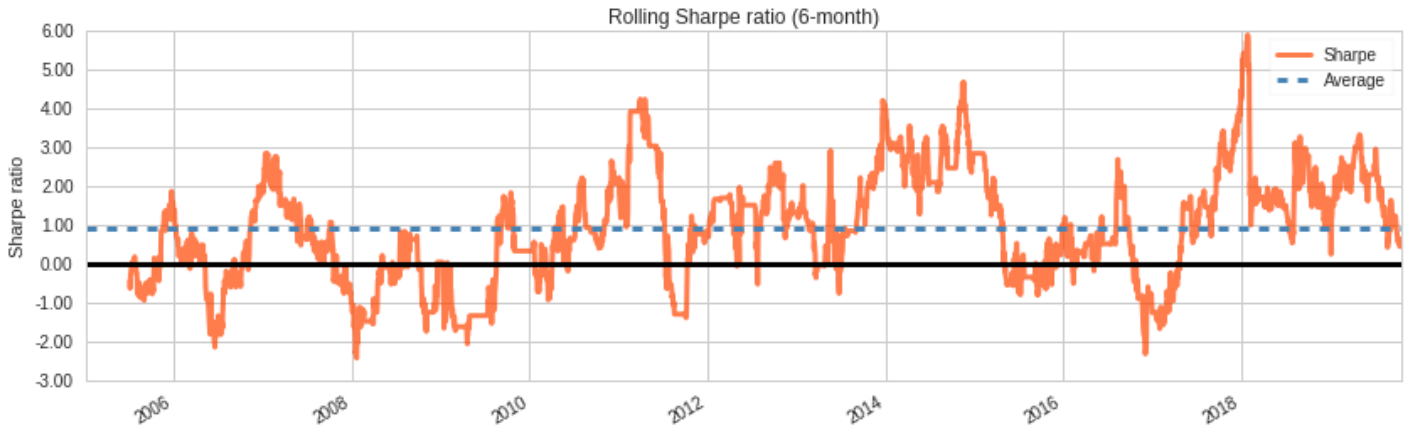


Figure 2: 6-month rolling Sharpe ratio with $r_f = 0\%$

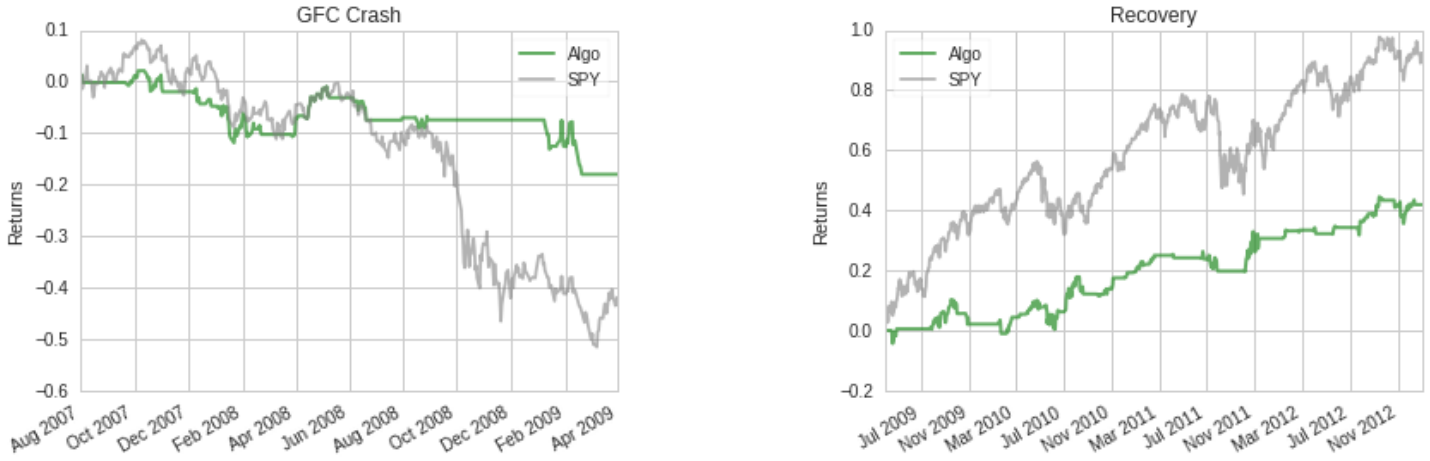


Figure 3: Stress test of back-test on GFC and Recovery results

The back-test period was chosen based on incorporating a sizeable lookback period of 3 years (Quantopian data only available from 2002) while also observing the effects of a stock market crash like the GFC in 2008 on the strategy. Based on the table and figures above, the strategy performs adequately from a raw price prediction perspective. The maximum drawdown of 23.5% from table 1 was observed during the GFC. A test for statistical significance of daily returns of the strategy reveals that the strategy has statistically significant positive daily returns at the 1% level.

Based on the above results, it is observed that the HMM alone cannot forecast accurately all of the time and hence from figure 1, it is seen that the HMM often mis-predicts a sell when the market is in an uptrend and vice versa. This may be the reason why the HMM does not beat the market in the long run. Conversely, from figure 3, the HMM predicted the GFC well and outperformed the market by not opening positions during this period.

Improvised HMM Trading Strategy

Based on the weaknesses above, improvisation is needed to beat the market, thus momentum trading is proposed. Momentum stock picking strategies are based on the notion that in the short term, the future repeats the past (Jegadeesh et al., 2001). Whilst there is much conjecture over the nature of momentum strategies and its purported violation of the efficient-market-hypothesis, out-of-period meta-analysis' have shown that such strategies can empirically generate profits in the short run (Jegadeesh et al., 2001). Consequently, the SPY is not traded here because it is a well-diversified ETF and this strategy generally requires exposure to individual stocks at each point in time.

Given that the HMM's weak predictability in the base strategy, it is reasonable to help it by using it only for price-signal confirmation. The primary method for stock picking is used via momentum indicators, designed to segregate assets believed to be on an upward trend. It is assumed the stocks displaying momentum have an upward sloping short term price history whilst conversely, the SPY market index can show any number of patterns. Ergo, the rationale for this improvisation is that an HMM model can better predict **price movements** when the underlying assets show historic upward trends, yielding a more probabilistic prediction as opposed to relatively unstable observational sequences (as seen in the base strategy). Most of the back-test setup is retained from the base strategy e.g. M, K, training period, etc.

As follows, this strategy relies on retrieving stocks on an upward trend by taking the entire universe of US stocks, applying a volume filter to discover the top 50 volume stocks based on the days before data and then sorting this list by highest returns based on the past month of trading days. The top three stocks are chosen for the back-test granted that HMM approves of their expected price movements.

Analogous to the basic strategy, at the start of every week, if the next weeks closing price prediction of a stock is above the last price, the algorithm buys, and creates an equally weighted portfolio of the stocks it chooses to buy. At the end of a week, all open positions are closed, and the filter and training processes repeat. Once again, short selling is avoided.

Results and Discussion

Back-test Results for Improvised Strategy	
Start date	04-01-2005
End date	21-10-2019
Total months	177
Annual return	2.7%
Cumulative returns	48.3%
Annual volatility	25.5%
Unadjusted Sharpe ratio of strategy (daily $r_f = \frac{0}{252}$)	0.23
Adjusted Sharpe ratio of strategy (daily $r_f = \frac{0.033}{252}$)	0.10
Unadjusted Sharpe ratio of benchmark (daily $r_f = \frac{0}{252}$)	0.54
Adjusted Sharpe ratio of benchmark (daily $r_f = \frac{0.033}{252}$)	0.36
Max drawdown	-68.9%
Alpha	0.01
Beta	0.52
$p - val$ for $t - stat$ for H_0: $\frac{r_t - r_{t-1}}{r_{t-1}} = 0$	0.373
Transaction cost per share traded (Kalyan, 2017)	$0.001

Table 3: Results of back-test

Figure 4: Cumulative returns of strategy vs. benchmark over back-test period

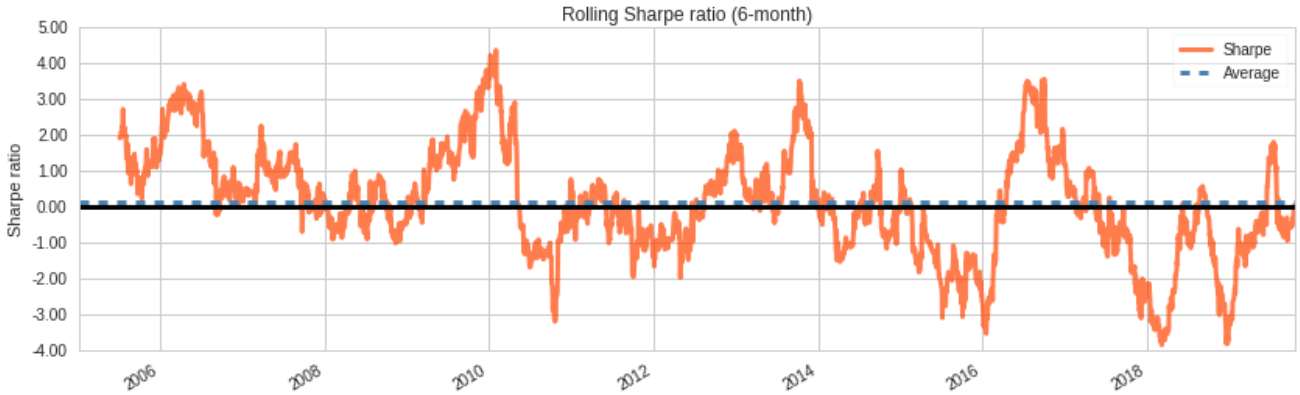


Figure 5: 6-month rolling Sharpe ratio with $r_f = 0\%$

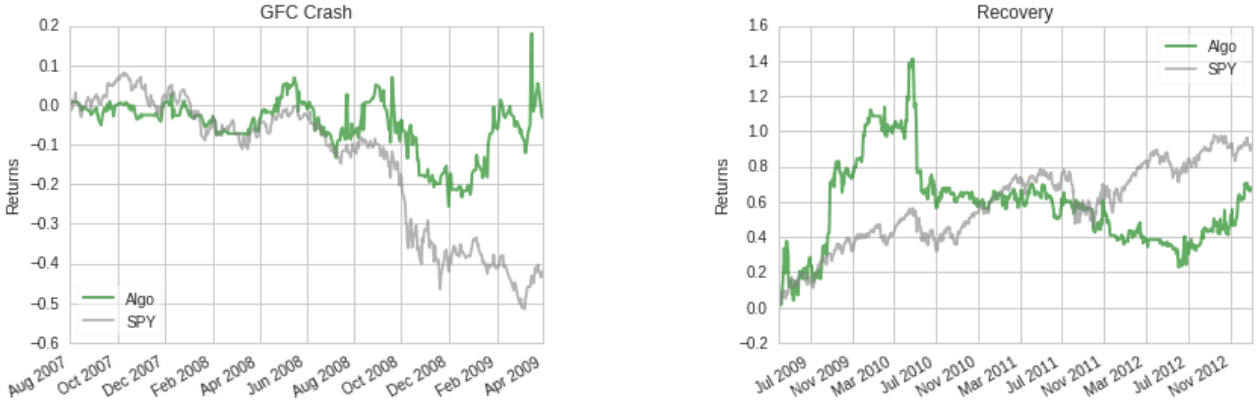


Figure 6: Stress test of back-test on GFC and Recovery results

Based on the table and figures above, there is evidence that the improvised strategy is no better than the original strategy. The maximum drawdown is during the back-test is 68.9%. A test for statistical significance of daily returns of the strategy reveals that the strategy has statistically insignificant positive daily returns at the 1% level. The Sharpe ratio suffers immensely due to the high volatility of the strategy as seen in figure 4.

Whilst the improvised strategy appears to perform extremely well up to the middle of 2010, cumulative returns are erased thereafter thus suggesting that this strategy loses in the long run. Like the original strategy, the improvised performs better than the market during the GFC in 2008, however may be purely due to luck.

General Weaknesses of HMM for Asset Price Prediction

These weaknesses apply to both strategies and the HMM in general when applied to asset price prediction:

1) Choice of hyperparameters:
   Hyperparameters are values used by a model that are set before a model begins training. Examples of these in the HMM are the number of hidden states M, the number of Gaussians K, maximum number of training iterations, the difference value for gauging similar log likelihoods ϵ. As it is impossible to know the true value for these parameters without domain expertise, it is necessary to conduct a technique such as (K-Fold) cross-validation to uncover these parameters.

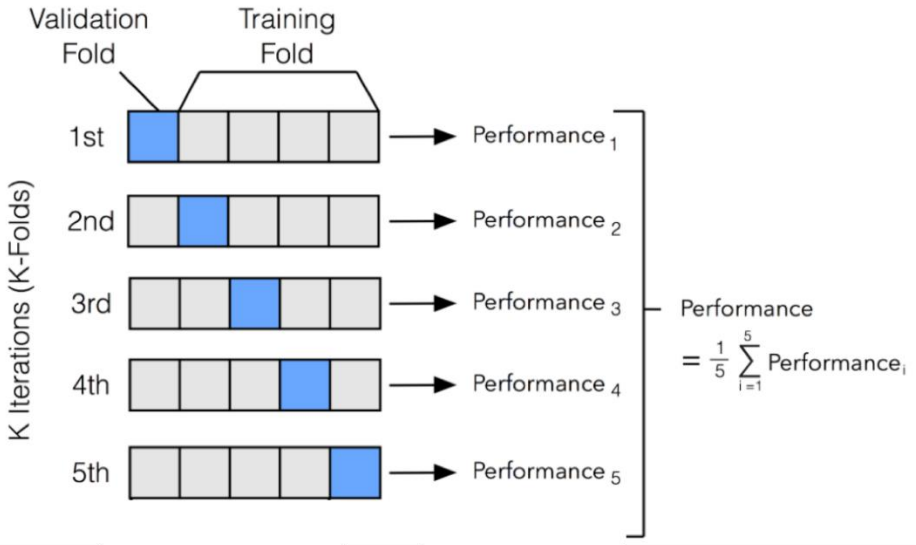


Figure 7: Diagram explaining K-Fold cross validation (Ethen, 2018)

   However, this popular technique is problematic for time series data in general because for example, patterns that appear chronologically across some periods may be ignored by the methodology of the technique. Another method of finding these parameters is required. Note that this weakness is not specific to an HMM but to time series data.
2) Computational power:
   Attempting to extend the historical window (e.g. 3 years to 5 years) to obtain the most similar log-likelihood of a past observation sequence or the number of assets (3 to 10 to improve diversification benefits) can add huge computational expense and slow the back-test immensely. Moreover, attempting to add additional hidden states M or Gaussians K to the model can also have the same effect (appendix – detailed description of HMMs). Given time constraints for the project, it was not possible to adjust these settings, so they were kept constrained to enable comparative statics across both strategies.

3) Inherent model assumptions:

The Markov property assumption of HMM assumes that the realisation of next hidden state solely depends on current realised hidden state. Intuitively, this can be interpreted as the choice of future underlying situations that affects the stock price solely depends on the choice of the current situation, which oversimplifies the process. Ideally, the mechanism should incorporate previous memories in the hidden state generating processes.

$$\forall i, j \in \{1,2, \dots M\}, \forall t \in \{1,2, \dots T\}, \qquad A_{ij} = p(z_{t+1} = j | z_t, z_{t-1}, z_{t-2}, \dots, z_{t-m})$$

Note that in changing this assumption, the model final formulas provided in the appendix may change.

4) Oversimplification of state transition matrix A:
The predicted observation (i.e. future stock price) is generated using a probability measure based on the estimated parameters, which include the hidden states transition matrix. The transition matrix can be considered as a lower-dimensional representation of the underlying factors that affect the generating process of realised observations (i.e. stock prices). Intuitively, the hidden states transition matrix should have some explanatory power of the underlying factors.
5) Initialisation of parameters:
The state transition matrix A is randomly initialised which implies that the back-test results are sensitive to a particular initialisation of this HMM parameter. Similarly, all other parameters are initialised to either vectors/matrices of 0s or 1s and hence without domain expertise on initialisation of these values, perfect back-test replicability with HMMs is difficult.

Note that these discovered weaknesses are based on the back-test results and the writers' own understanding, which may be biased due to the knowledge constraints.

Alternative Ideas

Here, alternative ideas that were thought of but not trialled due to time constraints are provided:

1) Directional change from stock pricing to stock selection
A detailed strategy proposed by Nguyen and Nguyen (2015) generates approximately 6 times more annualised return compared to the market from December 1999 to December 2014. This method uses a discrete HMM to predict a vector with binary elements (1 and 2) representing macroeconomic factors (e.g. inflation, industrial index etc.). Once HMMs predicts the next period's macroeconomic factors, the prediction method used in this project is repeated. Then, stock evaluations based on financial ratios and their corresponding dynamic weights in an evaluation matrix is conducted, thereby updating the portfolio accordingly.

2) Using other machine learning algorithms for price prediction:
Recursive Neutral Networks and other deep learning models may perform better due to more involved training but tend to overfit on the training data and may try to explain the noise as well. Some academic papers reveal an obvious lagged prediction, which perhaps does no better than using current price as the best prediction of tomorrow's price (Zhang et al., 2019, Wang et al., 1996, Kohara et al., 1998). Hence, in general, this area of machine learning may not generate high accuracy when it comes to stock pricing prediction.

3) Trade on low volatility assets:
Quite simply, another reason why HMM may does not predict so well is because the underlying assets used have been quite volatile. Low volatility assets tend to have smooth sequences of observation, which HMM should be able to better predict.

As always, it is necessary to conduct independent back-tests to verify results on any trading strategy.

Conclusion

This paper aimed to explore the feasibility of HMMs as an algorithm to profitably trade stocks. After back-testing the original strategy, it was found that the HMM model was weakly accurate in predicting raw stock prices but performed worse than the market over the long-run. Because of this weak prediction, the improvised model was designed to use HMM predictions only as reinsurance to a momentum strategy. It was found that it also failed to beat the market in the long run. However, these tested HMM strategies following the basic Markov assumptions perform moderately well in predicting market crashes such as the GFC. Historical analysis has been proven over time to not be all that reliable for future inference and thus following on, without modifications to the model it is inadvisable to use HMM in trading based on these strategies.

References

Saslow, E. (08/11/2018), Unsupervised Machine Learning, *Towards Data Science*, https://towardsdatascience.com/unsupervised-machine-learning-9329c97d6d9f

Maltby, H., Markov Chains, *Brilliant*, https://brilliant.org/wiki/markov-chains/

Soni, D. (06/03/2018), Introduction to Markov Chains, *Towards Data Science*, https://towardsdatascience.com/introduction-to-markov-chains-50da3645a50d

Gupta, T., Gaussian Mixture Model, *GeeksforGeeks*, https://www.geeksforgeeks.org/gaussian-mixture-model/

Rabiner, L. R. (1989), A tutorial on Hidden Markov Models and selected applications in speech recognition, *Proceedings of the IEEE*, 77(2), 257–286

Jurafsy, D. (02/10/2019), Chapter A: Hidden Markov Models, *Speech and Language Processing (3rd ed.)*, 5

Baum, L. E. (1972), An inequality and associated maximization technique in statistical estimation for probabilistic functions of Markov processes, *Inequalities III: Proceedings of the 3rd Symposium on Inequalities*, 1–8, Academic Press

Zhai, C. (16/03/2003), A Brief Note on the Hidden Markov Models (HMMs), *Department of CS University of Illinois*, 1

Warakagoda, N. (10/05/1996), Assumptions in the theory of HMMs, *Budapest University of Technology and Economics*, http://jedlik.phy.bme.hu/~gerjanos/HMM/node5.html

Jegadeesh, N., Titman, S. (2001). Profitability of Momentum Strategies: An Evaluation of Alternative Explanations. *The Journal of Finance,* 56(2), 699-720

Ethen (17/09/2018), K-Fold Cross Validation, *GitHub*, http://ethen8181.github.io/machine-learning/model_selection/model_selection.html

Figlewski, S. (1994). Forecasting volatility using historical data.

Nguyen, N., Nguyen, D. (2015), Hidden Markov Model for Stock Selection, Risks 2015

Department of the Treasury (23/10/2019), Daily Treasury Yield Curve Rates, *U.S. DEPARTMENT OF THE TREASURY*, https://www.treasury.gov/resource-center/data-chart-center/interest-rates/Pages/TextView.aspx?data=yield

Hassan, M. R., Nath, B. (08/09/2005), Stock Market Forecasting Using Hidden Markov Models: A New Approach, *IEEE fifth International Conference on Intelligent Systems Design and Applications*, 192–196.

Zhang, X., Li, C., Morimoto, Y. (01/01/2019), A Multi-factor Approach for Stock Price Prediction by using Recurrent Neural Networks, *Department of IE Hiroshima University*, 13

Wang, J., Leu, J. (1996), Stock Market Trend Prediction Using ARIMA-based Neural Network, *Department of EE National Taiwan Ocean University*, 2164

Kohara, K., Ishikawa, T., Fukuhara, Y., Nakamura, Y. (04/12/1998), Stock Price Prediction Using Prior Knowledge and Neural Networks

Kalyan, A. (08/11/2017), New Default Commissions and Capital, *Quantopian*, https://www.quantopian.com/posts/new-default-commissions-and-capital?fbclid=IwAR0QoBcG8c6smzE8IGqVhZHzbPHBRSoD7Wm7ad5pIcrS1Q7gFildRt2VA-o

Appendix

Detailed description of the three HMM algorithms

1) In general, as the hidden states are not known in advance, the total probability of an observation sequence can be calculated by summing the joint probability of the observations with a particular hidden state sequence over all possible hidden state sequences (Jurafsky, 2019).

$$p(X) = \sum_{i=1}^{M} p(X, z_i) = \sum_{i=1}^{M} p(X|z_i)p(z_i)$$

However, this is very computationally expensive and is considered to be an $O(TM^T)$ exponential algorithm. This problem is solved through the Forward algorithm which is a dynamic programming strategy i.e. stores intermediate values as it progresses towards the final step. The Forward algorithm does the above computation efficiently by folding each path into a single forward trellis (Jurafsky, 2019). The forward variable $\alpha_t(i)$ is defined as the probability of witnessing every observation up to time t, then arriving in state i at time t given the HMM parameters $\lambda = \{\pi, A, B\}$. This algorithm is $O(TM^2)$.

$$\alpha_t(i) = p(x_1, x_2, \dots, x_t, z_t = i|\lambda)$$

2) From above, it is restated that calculating the likelihood of an observation sequence given a hidden state sequence and then choosing the hidden state sequence with the highest likelihood is highly computationally expensive. The Viterbi algorithm uses two variables, $\delta_t(j)$ and $\psi_t(j)$, to calculate both the probability the HMM is in state j after witnessing the first t observations and passing through the most likely state sequence $z_1, \dots, z_{t-1}$ given the HMM parameters λ and the final most likely hidden state sequence given the entire observation sequence (Jurafsky, 2019).

$$\delta_t(j) = \max_{z_1, z_2, \dots, z_M} p(z_1, z_2, \dots, z_{t-1}, x_1, x_2, \dots, x_t, z_t = j|\lambda)$$

$$\psi_t(j) = \operatorname*{argmax}_{z_1, z_2, \dots, z_M} p(z_1, z_2, \dots, z_{t-1}, x_1, x_2, \dots, x_t, z_t = j|\lambda)$$

The optimal state sequence can be calculated by:

$$z_{t-1}^* = \psi_t(z_t^*)$$

3) HMM training is usually performed using the Baum-Welch algorithm. which is a special instance of the Expectation-Maximisation (E-M) algorithm (Baum, 1972). The Baum-Welch algorithm seeks to iteratively compute better estimates for the HMM parameters λ. In order to use this algorithm, the backward variable $\beta_t(i)$, defined as the probability of seeing an observed sequence at future time steps given that the hidden state at time t is i and the HMM parameters λ, must be calculated alongside the forward variable (Jurafsky, 2019).

$$\beta_t(i) = p(x_{t+1}, x_{t+2}, \dots, x_T|z_t = i, \lambda)$$

Now, $\xi_t(i, j)$ is defined as the probability of being in state i and state j at time t and time $t + 1$ respectively given the observation sequence and HMM parameters λ and $\gamma_t(i)$ is defined as the probability of being solely in state i at time t give the observation sequence X and HMM parameters λ (Jurafsky, 2019).

$$\xi_t(i, j) = p(z_t = i, z_{t+1} = j|X, \lambda)$$

$$\gamma_t(i) = p(z_t = i|X, \lambda) = \sum_{j=1}^{M} \xi_t(i,j)$$

A separate problem that commonly arises with HMMs is the underflow problem (Zhai, 2003). The underflow problem is described as the issue that arises when multiplying very long sequences of probabilities which is computationally problematic due to loss of numerical precision from very small numbers and rounding errors. This is dealt with through the concept of scaling. In the Viterbi algorithm, as only multiplication is involved, applying the natural logarithm will turn this into a summation algorithm. In the forward and backward algorithms, a scale factor c_t needs to be created to appropriately scale $\alpha_t(i)$ and $\beta_t(i)$. A careful layout of all the final updates and formulas, initialisation, induction and termination steps satisfying the above three fundamental problems are displayed below.

Mathematical overview for training Scaled Continuous HMM with Single Observation Sequence

1) Initialisation:

Randomly initialise $\{\pi, A, R, \mu, \Sigma\}$ (see code for specific individual initialisation)

2) E-M algorithm:

E-step:

Randomly initialise c_t (see code for specific individual initialisation)

$Comp_j(k,t) = R_j(k)N(x(t); \mu_j(k), \Sigma_j(k)$ ($Comp =$ individual mixture component of GMM)

$$B_t(j) = \sum_{k=1}^{K} Comp_j(k,t)$$

Forward algorithm:

$$c_t = \sum_{i=1}^{M} \alpha'_t(i)$$

Initialisation:

$\alpha'_1(i) = \pi_i B_j\big(x(1)\big)$ (Temporary variable)

$\alpha_1(i) = \frac{a'_1(i)}{c_1}$ (Scaled variable)

Induction:

$$\alpha'_t(i) = \sum_{i=1}^{M} \alpha_{t-1}(i)A_i(j)B_j\big(x(t)\big)$$

$$\alpha_t(i) = \frac{\alpha'_t(i)}{c_t}$$

Termination:

$$\log p(X) = \sum_{t=1}^{T} c_t$$

Backward algorithm:

Initialisation:

$$\hat{\beta}_T(i) = 1$$

Induction using same scale factor from before:

$$\hat{\beta}_t(i) = \frac{\sum_{j=1}^{M} A_i(j) B_j\big(x(t+1)\big)\hat{\beta}_{t+1}(j)}{c_{t+1}}$$

$$\gamma_j(k,t) = \frac{\alpha_t(j)\beta_t(j)}{\sum_{j'=1}^{M}\alpha_t(j')\beta_t(j')} \times \frac{Comp_j(k,t)}{B_j(t)}$$

M-step:

$$\pi_i = \alpha_1(i)\beta_1(i)$$

$$A_i(j) = \frac{\sum_{t=1}^{T-1}\frac{\alpha_t(i)\beta_{t+1}(j)}{c_{t+1}} A_i(j) B_j\big(x(t+1)\big)}{\sum_{t=1}^{T-1}\alpha_t(i)\beta_t(i)}$$

$$R_j(k) = \frac{\sum_{t=1}^{T}\gamma_j(k,t)}{\sum_{t=1}^{T}\sum_{k'=1}^{K}\gamma_j(k',t)}$$

$$\mu_j(k) = \frac{\sum_{t=1}^{T}\gamma_j(k,t)x_t}{\sum_{t=1}^{T}\gamma_j(k,t)}$$

$$\Sigma_j(k) = \frac{\sum_{t=1}^{T}\gamma_j(k,t)\Big(x_t - \mu_j(k)\Big)\Big(x_t - \mu_j(k)\Big)^T}{\sum_{t=1}^{T}\gamma_j(k,t)}$$

3) Repeat 2) until convergence or preset maximum number of iterations achieved

Quantopian Code

Original Strategy

```
"""
Original strategy using HMM
Authors: Joel Thomas, Liam Horrocks, Victoria Zhang
"""
import numpy as np
import quantopian.algorithm as algo
from quantopian.pipeline import Pipeline
from quantopian.pipeline.data.builtin import USEquityPricing
from quantopian.pipeline.filters import QTradableStocksUS

# Training period
ROLLING_WINDOW = 6
TRADE_DAYS_YEAR = 252
NUM_YEARS = 3


def random_normalized(d1, d2):
```

```
    """
    # Generates a random normalised matrix with dim(d1, d2)
    :param d1: Size of dimension 1 of matrix
    :param d2: Size of dimension 2 of matrix
    :return: X / X.sum(axis=1, keepdims=True): Randomly normalised matrix with given dimensions
    """
    X = np.random.random((d1, d2))
    return X / X.sum(axis=1, keepdims=True)


class MVN:
    """
    Multivariate Normal Gaussian PDF
    """
    def __init__(self, mu, sigma):
        """
        Constructor for this class
        :param mu: Mean vector of multivariate normal distribution
        :param sigma: Covariance matrix of multivariate normal distribution
        """
        self.mu = mu
        self.sigma = sigma

    def pdf(self, X):
        """
        Calculates PDF given distribution parameters
        :param X: Observation sequence
        :return: np.array(ps): array of probabilities for each x in X
        """
        ps = []
        for x in X:
            expression_1 = 1 / (((2 * np.pi)**(len(self.mu)/2)) * (np.linalg.det(self.sigma)**(1/2)))
            expression_2 = (-1/2) * ((x - self.mu).T.dot(np.linalg.inv(self.sigma))).dot((x - self.mu))
            ps.append(float(expression_1 * np.exp(expression_2)))
        return np.array(ps)


class HMM:
    """
    Continuous-observation HMM with scaling and allowing for only one observation sequence
    """
    def __init__(self, M, K):
        """
        Constructor for the class
```

```
        :param M: Total number of hidden states
        :param K: Total number of Gaussians in Gaussian Mixture Model (GMM)
        """
        self.M = M  # number of hidden states
        self.K = K  # number of Gaussians
        self.pi = None  # Initial state distribution
        self.A = None  # State transition matrix
        self.R = None  # Mixture proportions (responsibilities, probabilities of all K Gaussian)
        self.mu = None  # Means of all K Gaussians, dim = M*K*D
        self.Sigma = None  # Covariance matrices for all K Gaussians, dim = M*K*D*D


    def fit(self, X, D=1, max_iter=10, eps=1e-1):
        """
        Uses Baum-Welch together with Forward and Backward algorithms to fit HMM parameters (pi, A, B) on to
training
        data
        :param X: Data to fit model on to
        :param D: Dimension of X e.g. (open, high, low, close) --> 4
        :param max_iter: Maximum number of training iterations
        :param eps: Epsilon, used for smoothing
        """
        # Assume X (observation sequence) is organized (T, D)
        T = len(X)

        # Initialise all parameters
        self.pi = np.ones(self.M) / self.M
        self.A = random_normalized(self.M, self.M)
        self.R = np.ones((self.M, self.K)) / self.K
        self.mu = np.zeros((self.M, self.K, D))
        for i in range(self.M):
            for k in range(self.K):
                random_index = np.random.choice(T)
                self.mu[i, k] = X[random_index]
        self.Sigma = np.ones((self.M, self.K, D, D))

        costs = []
        # E-STEP FOR EM OPTIMISATION ALGORITHM, estimate B, alpha, beta, gamma
        for iteration in range(max_iter):
            # Scale to solve underflow problem
            scale = np.zeros(T)

            # Calculate B so can lookup when updating alpha and beta
            B = np.zeros((self.M, T))
            component = np.zeros((self.M, self.K, T))
            for j in range(self.M):
```

```
    for k in range(self.K):
        # Probability of observing jth observation at time t given symbol k
        p = self.R[j, k] * MVN(self.mu[j, k], self.Sigma[j, k]).pdf(X)
        component[j, k, :] = p
        # Probability = sum of individual mixture components
        B[j, :] += p

# FORWARD ALGORITHM
alpha = np.zeros((T, self.M))
# Initialisation step
alpha[0] = self.pi * B[:, 0]
scale[0] = alpha[0].sum()
alpha[0] /= scale[0]
# Induction step
for t in range(1, T):
    alpha_t_prime = alpha[t - 1].dot(self.A) * B[:, t]
    scale[t] = alpha_t_prime.sum()
    alpha[t] = alpha_t_prime / scale[t]
# Termination step
logP = np.log(scale).sum()

# BACKWARD ALGORITHM
beta = np.zeros((T, self.M))
# Initialisation step
beta[-1] = 1
# Induction step
for t in range(T - 2, -1, -1):
    for i in range(self.M):
        factor = 0
        for j in range(self.M):
            factor += self.A[i, j] * B[j, t + 1] * beta[t + 1, j]
        beta[t, i] = factor / scale[t + 1]

# Update for Gaussians
gamma = np.zeros((T, self.M, self.K))
for t in range(T):
    alpha_beta = alpha[t, :].dot(beta[t, :])
    for j in range(self.M):
        factor = alpha[t, j] * beta[t, j] / alpha_beta
        for k in range(self.K):
            gamma[t, j, k] = factor * component[j, k, t] / B[j, t]

# Cost = log-likelihood for all observation sequences
costs.append(logP)
```

```
# M-STEP FOR EM OPTIMISATION ALGORITHM, re-estimate pi, A, R, mu, Sigma
for i in range(self.M):
    self.pi[i] = alpha[0, i] * beta[0, i]

# Numerator for A
A_num = np.zeros((self.M, self.M))
# Denominator for A
A_den = np.zeros((self.M, 1))

for i in range(self.M):
    for t in range(T - 1):
        A_den[i] += alpha[t, i] * beta[t, i]
        for j in range(self.M):
            A_num[i, j] += alpha[t, i] * beta[t + 1, j] * self.A[i, j] * B[j, t + 1] / scale[t + 1]
# Update A
self.A = A_num / A_den

# Update individual mixture components
R_num_n = np.zeros((self.M, self.K))
R_den_n = np.zeros(self.M)
for j in range(self.M):
    for k in range(self.K):
        for t in range(T):
            R_num_n[j, k] += gamma[t, j, k]
            R_den_n[j] += gamma[t, j, k]
R_num = R_num_n
R_den = R_den_n

mu_num_n = np.zeros((self.M, self.K, D))
Sigma_num_n = np.zeros((self.M, self.K, D))
for j in range(self.M):
    for k in range(self.K):
        for t in range(T):
            # Update means
            mu_num_n[j, k] += gamma[t, j, k] * X[t]

            # Update covariances
            Sigma_num_n[j, k] += gamma[t, j, k] * (X[t] - self.mu[j, k]) ** 2
mu_num = mu_num_n
Sigma_num = Sigma_num_n

# Update R, mu, sigma
for j in range(self.M):
```

```
            for k in range(self.K):
                self.R[j, k] = R_num[j, k] / R_den[j]
                self.mu[j, k] = mu_num[j, k] / R_num[j, k]
                self.Sigma[j, k] = Sigma_num[j, k] / R_num[j, k] + np.ones(D) * eps

    # print(f"Final pi: {self.pi}")
    # print(f"Final A: {self.A}")
    # print(f"Final mu: {self.mu}")
    # print(f"Final Sigma: {self.Sigma}")
    # print(f"Final R: {self.R}")

    # plt.figure(figsize=(10, 6))
    # plt.plot(costs)
    # plt.tight_layout()
    # plt.show()

def log_likelihood(self, X):
    """
    Returns log(P(X|model))
    :param X: Observation sequence
    :return: np.log(scale).sum(): Probability of an observation sequence given parameters
    """
    # FORWARD ALGORITHM
    T = len(X)
    scale = np.zeros(T)
    B = np.zeros((self.M, T))
    for j in range(self.M):
        for k in range(self.K):
            p = self.R[j, k] * MVN(self.mu[j, k], self.Sigma[j, k]).pdf(X)
            B[j, :] += p

    alpha = np.zeros((T, self.M))
    # Initialisation step
    alpha[0] = self.pi * B[:, 0]
    scale[0] = alpha[0].sum()
    alpha[0] /= scale[0]
    # Induction step
    for t in range(1, T):
        alpha_t_prime = alpha[t - 1].dot(self.A) * B[:, t]
        scale[t] = alpha_t_prime.sum()
        alpha[t] = alpha_t_prime / scale[t]
    # Termination step
    return np.log(scale).sum()
```

```
    def predict_similar_likelihood(self, X, T, likelihood, epsilon=10):
        """
        Uses current log likelihood on trained window to compare and retrieve similar likelihoods based on past
windows
        :param X: Observation sequence
        :param T: Length of observation sequence
        :param likelihood: Likelihood of observation sequence given HMM parameters
        :param epsilon: Threshold for similar likelihoods
        :return: prediction: Predicted value at end of next period
        """
        # List of all similar past likelihoods
        likelihoods = []
        # List of observation sequences for which similar likelihoods
        historical_windows = []
        # List of time periods to move back to to obtain similar likelihoods on historical window
        ts = []
        for t in range(1, T-ROLLING_WINDOW):
            X_past = X[T-ROLLING_WINDOW-t:T-t]
            likelihood_past = self.log_likelihood(X_past)
            difference = likelihood_past - likelihood
            if abs(difference) < epsilon:
                likelihoods.append(likelihood_past)
                historical_windows.append(X_past)
                ts.append(t)
        # If no similar likelihoods --> no prediction
        try:
            highest_similar_likelihood_data = historical_windows[likelihoods.index(max(likelihoods))]
            highest_similar_likelihood = likelihoods[likelihoods.index(max(likelihoods))]
            highest_similar_likelihood_t = ts[likelihoods.index(max(likelihoods))]
        except ValueError:
            return None
        # Prediction formula from appendix
        prediction = X[-1] + (X[T-highest_similar_likelihood_t] - highest_similar_likelihood_data[-1]) * \
            np.sign(likelihood - highest_similar_likelihood)
        return prediction



def initialize(context):
    """
    Called once at the start of the algorithm.
    """
    # Schedule predict_and_open_trades to run every Friday 1 hour before market close
    algo.schedule_function(
        predict_and_open_trades,
        algo.date_rules.week_end(0),
```

```
        algo.time_rules.market_close(hours=1),
    )

    # SID for SPDR S&P 500 ETF
    context.spy = sid(8554)


def close_trade(context):
    """
    Close an open trade
    """
    spy = context.spy
    order_size = context.portfolio.positions[spy].amount
    if order_size != 0:
        order_target_percent(spy, 0)


def predict_and_open_trades(context, data):
    """
    Original strategy from report
    """
    spy = context.spy
    price = data.history(spy, fields="price", bar_count=TRADE_DAYS_YEAR*NUM_YEARS, frequency="1d")
    # Resample prices to weekly
    price = price.resample("1W").last()
    X = np.array(price, dtype=float)
    T = len(X)

    # Initialise HMM model with 2 hidden states and 2 Gaussians
    hmm_model = HMM(2, 2)
    X_train = X[-ROLLING_WINDOW:]
    hmm_model.fit(X_train)
    likelihood = hmm_model.log_likelihood(X_train)
    prediction = hmm_model.predict_similar_likelihood(X, T, likelihood)
    # Record predictions for later
    record("Prediction", prediction, "Price", price[-1])
    if prediction:
        # If prediction greater than last price, open long position with entire portfolio value
        if prediction > price.iloc[-1]:
            order_target_percent(spy, 1.0)
        # Keep open positions open unless next period's prediction is lower than last price
        elif prediction < price.iloc[-1]:
            close_trade(context)
```

Improvised Strategy

```
"""
Improvised strategy using HMM
Authors: Joel Thomas, Liam Horrocks, Victoria Zhang
"""
import numpy as np
import quantopian.algorithm as algo
from quantopian.pipeline import Pipeline, CustomFactor
from quantopian.pipeline.data.builtin import USEquityPricing
from quantopian.pipeline.filters import QTradableStocksUS

# Training period
ROLLING_WINDOW = 6
# Required number of days of historical data for inferring momentum of a stock
MOMENTUM_WINDOW_DAYS = 22
TRADE_DAYS_YEAR = 252
NUM_YEARS = 3
NUM_ASSETS = 3
TOP_50_ASSETS = 50


def random_normalized(d1, d2):
    """
    # Generates a random normalised matrix with dim(d1, d2)
    :param d1: Size of dimension 1 of matrix
    :param d2: Size of dimension 2 of matrix
    :return: X / X.sum(axis=1, keepdims=True): Randomly normalised matrix with given dimensions
    """
    X = np.random.random((d1, d2))
    return X / X.sum(axis=1, keepdims=True)


class MVN:
    """
    Multivariate Normal Gaussian PDF
    """
    def __init__(self, mu, sigma):
        """
        Constructor for this class
        :param mu: Mean vector of multivariate normal distribution
        :param sigma: Covariance matrix of multivariate normal distribution
        """
        self.mu = mu
        self.sigma = sigma
```

```
    def pdf(self, X):
        """
        Calculates PDF given distribution parameters
        :param X: Observation sequence
        :return: np.array(ps): array of probabilities for each x in X
        """
        ps = []
        for x in X:
            expression_1 = 1 / (((2 * np.pi)**(len(self.mu)/2)) * (np.linalg.det(self.sigma)**(1/2)))
            expression_2 = (-1/2) * ((x - self.mu).T.dot(np.linalg.inv(self.sigma))).dot((x - self.mu))
            ps.append(float(expression_1 * np.exp(expression_2)))
        return np.array(ps)



class HMM:
    """
    Continuous-observation HMM with scaling and allowing for only one observation sequence
    """
    def __init__(self, M, K):
        """
        Constructor for the class
        :param M: Total number of hidden states
        :param K: Total number of Gaussians in Gaussian Mixture Model (GMM)
        """
        self.M = M  # number of hidden states
        self.K = K  # number of Gaussians
        self.pi = None  # Initial state distribution
        self.A = None  # State transition matrix
        self.R = None  # Mixture proportions (responsibilities, probabilities of all K Gaussian)
        self.mu = None  # Means of all K Gaussians, dim = M*K*D
        self.Sigma = None  # Covariance matrices for all K Gaussians, dim = M*K*D*D


    def fit(self, X, D=1, max_iter=10, eps=1e-1):
        """
        Uses Baum-Welch together with Forward and Backward algorithms to fit HMM parameters (pi, A, B) on to
training
        data
        :param X: Data to fit model on to
        :param D: Dimension of X e.g. (open, high, low, close) --> 4
        :param max_iter: Maximum number of training iterations
        :param eps: Epsilon, used for smoothing
        """
        # Assume X (observation sequence) is organized (T, D)
        T = len(X)
```

```
# Initialise all parameters
self.pi = np.ones(self.M) / self.M
self.A = random_normalized(self.M, self.M)
self.R = np.ones((self.M, self.K)) / self.K
self.mu = np.zeros((self.M, self.K, D))
for i in range(self.M):
    for k in range(self.K):
        random_index = np.random.choice(T)
        self.mu[i, k] = X[random_index]
self.Sigma = np.ones((self.M, self.K, D, D))

costs = []
# E-STEP FOR EM OPTIMISATION ALGORITHM, estimate B, alpha, beta, gamma
for iteration in range(max_iter):
    # Scale to solve underflow problem
    scale = np.zeros(T)

    # Calculate B so can lookup when updating alpha and beta
    B = np.zeros((self.M, T))
    component = np.zeros((self.M, self.K, T))
    for j in range(self.M):
        for k in range(self.K):
            # Probability of observing jth observation at time t given symbol k
            p = self.R[j, k] * MVN(self.mu[j, k], self.Sigma[j, k]).pdf(X)
            component[j, k, :] = p
            # Probability = sum of individual mixture components
            B[j, :] += p

    # FORWARD ALGORITHM
    alpha = np.zeros((T, self.M))
    # Initialisation step
    alpha[0] = self.pi * B[:, 0]
    scale[0] = alpha[0].sum()
    alpha[0] /= scale[0]
    # Induction step
    for t in range(1, T):
        alpha_t_prime = alpha[t - 1].dot(self.A) * B[:, t]
        scale[t] = alpha_t_prime.sum()
        alpha[t] = alpha_t_prime / scale[t]
    # Termination step
    logP = np.log(scale).sum()

    # BACKWARD ALGORITHM
```

```
beta = np.zeros((T, self.M))
# Initialisation step
beta[-1] = 1
# Induction step
for t in range(T - 2, -1, -1):
    for i in range(self.M):
        factor = 0
        for j in range(self.M):
            factor += self.A[i, j] * B[j, t + 1] * beta[t + 1, j]
        beta[t, i] = factor / scale[t + 1]

# Update for Gaussians
gamma = np.zeros((T, self.M, self.K))
for t in range(T):
    alpha_beta = alpha[t, :].dot(beta[t, :])
    for j in range(self.M):
        factor = alpha[t, j] * beta[t, j] / alpha_beta
        for k in range(self.K):
            gamma[t, j, k] = factor * component[j, k, t] / B[j, t]

# Cost = log-likelihood for all observation sequences
costs.append(logP)

# M-STEP FOR EM OPTIMISATION ALGORITHM, re-estimate pi, A, R, mu, Sigma
for i in range(self.M):
    self.pi[i] = alpha[0, i] * beta[0, i]

# Numerator for A
A_num = np.zeros((self.M, self.M))
# Denominator for A
A_den = np.zeros((self.M, 1))

for i in range(self.M):
    for t in range(T - 1):
        A_den[i] += alpha[t, i] * beta[t, i]
        for j in range(self.M):
            A_num[i, j] += alpha[t, i] * beta[t + 1, j] * self.A[i, j] * B[j, t + 1] / scale[t + 1]
# Update A
self.A = A_num / A_den

# Update individual mixture components
R_num_n = np.zeros((self.M, self.K))
R_den_n = np.zeros(self.M)
for j in range(self.M):
```

```
            for k in range(self.K):
                for t in range(T):
                    R_num_n[j, k] += gamma[t, j, k]
                    R_den_n[j] += gamma[t, j, k]
        R_num = R_num_n
        R_den = R_den_n

        mu_num_n = np.zeros((self.M, self.K, D))
        Sigma_num_n = np.zeros((self.M, self.K, D))
        for j in range(self.M):
            for k in range(self.K):
                for t in range(T):
                    # Update means
                    mu_num_n[j, k] += gamma[t, j, k] * X[t]

                    # Update covariances
                    Sigma_num_n[j, k] += gamma[t, j, k] * (X[t] - self.mu[j, k]) ** 2
        mu_num = mu_num_n
        Sigma_num = Sigma_num_n

        # Update R, mu, sigma
        for j in range(self.M):
            for k in range(self.K):
                self.R[j, k] = R_num[j, k] / R_den[j]
                self.mu[j, k] = mu_num[j, k] / R_num[j, k]
                self.Sigma[j, k] = Sigma_num[j, k] / R_num[j, k] + np.ones(D) * eps

    # print(f"Final pi: {self.pi}")
    # print(f"Final A: {self.A}")
    # print(f"Final mu: {self.mu}")
    # print(f"Final Sigma: {self.Sigma}")
    # print(f"Final R: {self.R}")

    # plt.figure(figsize=(10, 6))
    # plt.plot(costs)
    # plt.tight_layout()
    # plt.show()

def log_likelihood(self, X):
    """
    Returns log(P(X|model))
    :param X: Observation sequence
    :return: np.log(scale).sum(): Probability of an observation sequence given parameters
    """
```

```
        # FORWARD ALGORITHM
        T = len(X)
        scale = np.zeros(T)
        B = np.zeros((self.M, T))
        for j in range(self.M):
            for k in range(self.K):
                p = self.R[j, k] * MVN(self.mu[j, k], self.Sigma[j, k]).pdf(X)
                B[j, :] += p

        alpha = np.zeros((T, self.M))
        # Initialisation step
        alpha[0] = self.pi * B[:, 0]
        scale[0] = alpha[0].sum()
        alpha[0] /= scale[0]
        # Induction step
        for t in range(1, T):
            alpha_t_prime = alpha[t - 1].dot(self.A) * B[:, t]
            scale[t] = alpha_t_prime.sum()
            alpha[t] = alpha_t_prime / scale[t]
        # Termination step
        return np.log(scale).sum()

    def predict_similar_likelihood(self, X, T, likelihood, epsilon=10):
        """
        Uses current log likelihood on trained window to compare and retrieve similar likelihoods based on past
windows
        :param X: Observation sequence
        :param T: Length of observation sequence
        :param likelihood: Likelihood of observation sequence given HMM parameters
        :param epsilon: Threshold for similar likelihoods
        :return: prediction: Predicted value at end of next period
        """
        # List of all similar past likelihoods
        likelihoods = []
        # List of observation sequences for which similar likelihoods
        historical_windows = []
        # List of time periods to move back to to obtain similar likelihoods on historical window
        ts = []
        for t in range(1, T-ROLLING_WINDOW):
            X_past = X[T-ROLLING_WINDOW-t:T-t]
            likelihood_past = self.log_likelihood(X_past)
            difference = likelihood_past - likelihood
            if abs(difference) < epsilon:
                likelihoods.append(likelihood_past)
                historical_windows.append(X_past)
```

```
                ts.append(t)
        # If no similar likelihoods --> no prediction
        try:
            highest_similar_likelihood_data = historical_windows[likelihoods.index(max(likelihoods))]
            highest_similar_likelihood = likelihoods[likelihoods.index(max(likelihoods))]
            highest_similar_likelihood_t = ts[likelihoods.index(max(likelihoods))]
        except ValueError:
            return None
        # Prediction formula from appendix
        prediction = X[-1] + (X[T-highest_similar_likelihood_t] - highest_similar_likelihood_data[-1]) * \
            np.sign(likelihood - highest_similar_likelihood)
        return prediction


class CumReturn(CustomFactor):
    """
    Custom factor to find cumulative returns over past preset number of days (MOMENTUM_WINDOW_DAYS)
    """
    inputs = [USEquityPricing.close]
    window_length = MOMENTUM_WINDOW_DAYS

    # Computes cumulative returns based on Pipeline
    def compute(self, today, asset_ids, out, values):
        out[:] = (values[-1]-values[0])/values[0]


def make_pipeline():
    """
    Makes pipeline of given factors on entire US stocks universe
    :return: pipe: Custom pipeline to obtain cumulative returns based on preset number of days and volume
    """
    cum_returns = CumReturn()
    pipe = Pipeline(columns={"Cumulative Return": cum_returns,
                             "Volume": USEquityPricing.volume.latest}, screen=QTradableStocksUS())
    return pipe


def initialize(context):
    """
    Called once at the start of the algorithm.
    """
    algo.attach_pipeline(make_pipeline(), 'long_momentum_stocks')

    # Schedule close_trade to run every Friday 1 hour after market open
```

```
    algo.schedule_function(
        close_trade,
        algo.date_rules.week_end(0),
        algo.time_rules.market_open(hours=1),
    )

    # Schedule before_trade to run every Friday 2 hours after market open
    algo.schedule_function(
        before_trade,
        algo.date_rules.week_end(0),
        algo.time_rules.market_open(hours=2),
    )

    # Schedule predict_and_open_trades to run every Friday 3 hours after market open
    algo.schedule_function(
        predict_and_open_trades,
        algo.date_rules.week_end(0),
        algo.time_rules.market_open(hours=3),
    )


def before_trade(context, data):
    """
    Obtain pipeline data based on given factors
    """
    context.pipeline_data = algo.pipeline_output('long_momentum_stocks')


def close_trade(context, data):
    """
    Close an open trade
    """
    for stock in context.portfolio.positions:
        order_target_percent(stock, 0)


def predict_and_open_trades(context, data):
    """
    Original strategy from report
    """
    pipeline_data = context.pipeline_data
    # Highest volume filter based on top 50 stocks
    highest_volume = pipeline_data.sort_values("Volume", ascending=False).head(TOP_50_ASSETS).index
    # Highest returns filter based on number of stocks under consideration
```

```
    highest_returns = pipeline_data.loc[highest_volume].sort_values("Cumulative Return",
ascending=False).head(NUM_ASSETS)

    stocks = list(highest_returns.index)

    prices = data.history(stocks, fields="price", bar_count=TRADE_DAYS_YEAR*NUM_YEARS, frequency="1d")

    # Resample prices to weekly

    prices = prices.resample("1W").last()

    # Train HMM on every stock after filters have been applied

    for stock in prices:

        X = np.array(prices[stock])

        T = len(X)


        hmm_model = HMM(2, 2)

        X_train = X[-ROLLING_WINDOW:]

        hmm_model.fit(X_train)

        likelihood = hmm_model.log_likelihood(X_train)

        prediction = hmm_model.predict_similar_likelihood(X, T, likelihood)

        if prediction:

            # If prediction greater than last price, open long position with weighted portfolio value (number of
assets

            # under consideration

            if prediction > prices[stock].iloc[-1]:

                order_target_percent(stock, 1.0/NUM_ASSETS)
```