Applications of the Kalman Filter to Timeseries Analysis

Freddie Poser & André Renom

2017

Abstract

In this experiment we investigate the effectiveness of the Kalman filter in predicting stock market data. We compare it to two other models, simple and weighted moving averages. We evaluate these models on their ability to predict long term movements in stock prices (on the scale of one day).

Contents

List of Figures

1 Introduction

The Kalman filter was first developed by Rudolf Kalman a Hungarian-born electrical engineer. The process involves, each day, making a prediction based on the previous day, and adjusting it for the measurement for that day. This adjustment is done by using the state variables and the error covariance of the data to determine the best ratio.

The Kalman filter can be applied to financial data as it allows us to estimate the "true" market price of a stock. To do this we need to first assume that there are two components to a stock's market price: a true value and a noise component. The true value reflects how the market actually values the stock whilst the noise component is simply the result of very short term market changes. The Kalman filter, if applied correctly, allows us to remove this noise and, hopefully, gain an insight into the long term trend of the data.

There are a number of different methods that also aim to do this, all with varying degrees of success. In this experiment we compare these methods to the Kalman filter and try to find the most accurate one. To do this the models were used to predict the price of a basket of stocks, simulating the profit and loss they would have made over a time period. We ignored the influence of transaction costs in this process. These charges have no bearing on the accuracy of these models as predictors.

We used the same basket of stocks and date range for each test so that we can be sure that there is a fair comparison.

All of the code is available at `https://github.com/vogon101/Trading`. This experiment was implemented in Scala[1]. For the linear algebra involved we used a mathematics library called Breeze[2]

2 Models Tested

In this experiment we tested three different smoothing filters. As they each have their own parameters that effect their operation we used trial-and-error search to find the best values.

2.1 Simple Moving Average

The simple moving average (SMA) model is the simplest smoothing filter that we tested. The simple moving average is just the mean of the last n data points where n is the period of the SMA filter [Wikipedia, 2017b]. The filter is defined as below:

Figure 1: The equations for simple moving average

(a) Simple moving average ordinal definition

$$SMA_i = \frac{U_i + U_{i-1} + U_{i-2} + \ldots + U_{i-(n-1)}}{n}$$

(b) Simple moving average iterative definition

$$SMA_i = SMA_{i-1} + \frac{U_i - U_{i-n}}{n}$$

2.2 Weighted Moving Average

A weighted moving average is much like a simple one except that the more recent values have more bearing on the output of the model [Wikipedia, 2017b]. In this experiment we only test linearly-weighted moving averages. The weighted moving average model is defined as follows:

[1] `https://www.scala-lang.org/`
[2] `https://github.com/scalanlp/breeze`

Figure 2: Weighted moving average ordinal definition

$$WMA_i = \frac{(n-1)U_i + (n-2)U_{i-1} + \ldots + 2U_{i-(n-2)} + U_{i-n}}{\sum_{r=1}^{n} r}$$

2.3 Kalman Filter

The Kalman filter, a multi-step iterative filter, is the most complex of the three models [Wikipedia, 2017a]. Instead of treating the data as scalar, it is instead treated as a vector, and as such, it is manipulated with matrices. This allows the filter to be used for modelling more complex relationships between multiple variables simultaneously. The prime example of this would be modelling stock price and sale volume alongside each other. In this experiment, however, to make this a fair test, we only used the stock price, in the form of a 1x1 vector. This was necessary because the other two models are purely scalar and can only handle one dimensional data.

The version of the Kalman filter that we are using has four matrices that define the filter's operation. These are constant within one use of the filter and are based on the data set.

Figure 3: State transition matrices used by the Kalman filter

$\mathbf{A}$ = State transition matrix
$\mathbf{H}$ = State to measurement matrix
$\mathbf{Q}$ = Covariance noise matrix
$\mathbf{R}$ = State transition noise matrix

[Esme, 2017]

Over large enough data sets, the A and H matrices are of limited importance, and as such, we took them as the identity matrix. Q and R are both diagonal matrices(multiples of the identity matrix). These form the basis of the trade-off within the Kalman filter, allowing for closer following of the curve, but with less smoothing, or vice-versa.

Figure 4: State variables used in the Kalman filter

$\hat{x}_k^-$ = State prediction on day k
$\mathbf{P}_k^-$ = Error covariance prediction on day k
$\hat{x}_k$ = State estimate for day k
$\mathbf{P}_k$ = Error covariance estimate on day k

The Kalman filter works by multiplying the last state estimate ($\hat{x}_{k-1}$) by the state transition matrix to create a prediction for the upcoming day ($\hat{x}_k^-$). It then does the same with the error co-variance, adding this time the covariance noise matrix ($\mathbf{Q}$). This gives it two predicted values, one for the state, and one for the covariance. Using these two values in conjunction with the state to measurement matrix and the state transition noise matrix, it the calculates what is known as the Kalman gain for that day ($\mathbf{K}_k$). Having found the Kalman gain for that day, the filter then introduces the measurement for that day (in this case the actual stock price that day, labelled z_k). The final estimate of the true state is calculated by adding to the prediction the difference between it and the prediction, multiplied by the Kalman gain. It then recalculates an estimate for the error covariance based on the Kalman gain calculated for that day.

Below are the equations that define the Kalman filter's operation:

Figure 5: Kalman filter definition

$$
\begin{aligned}
\hat{x}_k^- &= \mathbf{A}\hat{x}_{k-1} \\
\mathbf{P}_k^- &= \mathbf{A}P_{k-1}\mathbf{A}^T + \mathbf{Q} \\
\mathbf{K}_k &= \mathbf{P}_k^-\mathbf{H}^T(\mathbf{H}\mathbf{P}_k^-\mathbf{H}^T + \mathbf{R})^{-1} \\
\hat{x}_k &= \hat{x}_k^- + \mathbf{K}_k(z_k - \mathbf{H}\hat{x}_k^-) \\
\mathbf{P}_k &= (\mathbf{I} - \mathbf{K}_k\mathbf{H})\mathbf{P}_k^-
\end{aligned}
$$

3 Experimental Method

To test these three models we implemented them in Scala code. We used the following method for nearly 200 stocks from the London and New York stock exchanges.

1. Fetch one year of stock-openings data
2. Feed the model 30 days of data without testing it
   - This represents the previous 30 days in the run up to predicting
3. Test the model over the remaining 11 months of data, each time:
   (a) Predict the next day by using quadratic regression
   (b) Compare this prediction to the last days actual price
      - If the prediction is above the last day's price, simulate a buy at the last day's price
      - If the prediction is below the last day's price, simulate a short at the last day's price
   (c) Take the new day's actual price and simulate the profit/loss it would have made
   (d) Feed the day's actual price into the model so that it can update its estimate

This process leaves us with a final profit for each stock which we can average to find the best model.

3.1 Fetching Stock Data

To fetch stock data we used pandas datareader[3] to download a stocks history from Google finance[4]. The code then writes the data into a csv file in the format `<date>, <opening price>, <closing price>`. In the code below the list of stocks has been shortened to save space, the actual list is given in Appendix A. This gives us the data from the first of January 2016 to the 15th of May 2017.

[3] `https://pandas-datareader.readthedocs.io/en/latest/`
[4] `https://www.google.com/finance`

Figure 6: Code to fetch stock data from Google Finance

```
import pandas_datareader.data as web
import datetime

STOCKS = ["III","ADN","ADM","AAL","ANTO",...,"VOD","WBA","WDC","XLNX","YHOO"]

print("Total Stocks: " + str(len(STOCKS))

start = datetime.datetime(2016, 1, 1)
end = datetime.datetime.today()

for stock in STOCKS:
print("Fetching " + stock)
data = web.DataReader(stock, 'google', start, end)
with open(stock + ".csv", "w") as file:
file.write(data.to_csv())
```

3.2 Quadratic Regression

We used Quadratic regression to find the prediction because these models are estimates of the "true" value of the stock and do not predict on their own. Quadratic regression takes the last three points of the model and fits a quadratic curve to them. This curve is then extrapolated to the next time period: this is our predicted value.

The formula for quadratic regression is as follows. The inputs to this process are the last three estimates (S_i ,S_{i-1}, S_{i-2}, the smoothed values of the model). R is the extrapolated next point.

Figure 7: Quadratic regression over three points

$$
\begin{aligned}
G_1 &= S_{i-1} - S_{i-2} \\
G_2 &= S_i - S_{i-1} \\
d &= G_2 - G_1 \\
R &= P + G_2 + d
\end{aligned}
$$

3.3 Parameters

In this experiment we used a value of 0.08 for q and 0.96 for r in the Kalman filter. For both of the moving average filters we used a period of 15. We chose these because they give a representative view of the effectiveness of the models.

3.4 Tested Code

3.4.1 Simple Moving Average

To implement the simple moving average filter we used the ordinal definition (Figure 1a) because it was easier to implement. The moving average class itself has a time period and it keeps track of that many data points. Every time that the update function is called with the actual day's price the moving average is updated by removing the oldest point, adding the new one and re-calculating.

Figure 8: Implementation of the simple moving average filter

```
class SimpleMovingAverage (val period: Int) {

        private var _data: ListBuffer[Double] = ListBuffer()
        private var _x = 0d

        def window: List[Double] = _data.toList
        def x = _x

        def update(i: Double): Double = {
                _data.append(i)
                assert(_data.last == i)
                if(window.length > period) _data = _data.drop(1)
                _x = window.sum / window.length
                _x
        }

}
```

3.4.2 Weighted Moving Average

The weighted moving average used in this experiment had linear weights based on the period. This means that the latest data point has a weight of the period and every previous point has a weight of one less (i.e. U_{i-1} has weight $n-1$, U_{i-2} has weight $n-2$). The points that do not fall within the time period have an effective weight of 0. The weighted moving average implementation had to account for situations where an entire period of data is not available (i.e. there have only elapsed 5 "days" for a period 10 filter). This is an implementation of Figure 2.

Figure 9: Implementation of the weighted moving average filter

```
class WeightedMovingAverage (val period: Int){

        val weights: List[Double] = Range(1, period + 1).map(_.toDouble).toList

        private var _data: ListBuffer[Double] = ListBuffer()
        private var _x = 0d

        def window: List[Double] = _data.toList
        def x = _x

        def update(i: Double): Double = {
                _data.append(i)
                if (window.length > period) {
                        _data = _data.drop(1)
                        _x = window
                        .zip(weights)
                        .map(X => X._1 * X._2)
                        .sum / weights.sum
                } else {
                        _x = window
                        .zip(weights.drop(period -  window.length))
                        .map(X => X._1 * X._2)
                        .sum / weights.drop(period -  window.length).sum
                }

                _x
        }

}
```

3.4.3 Kalman Filter

In this experiment we use a slightly simplified version of the Kalman filter that ignores some of the less significant matrices (namely **A** and **H**). In this code `state` holds all of the variables that the filter needs. This code is called when the next measurement is available. It is an implementation of the equations in figure 5.

Figure 10: Implementation of the Kalman filter

```
def update(data: DenseVector[Double]): KalmanState = {

        _state = state.from_state(state.x, state.p + state.Q)
        val K = state.p * state.H.t * inv(state.H * state.p * state.H.t + state.R)
        val xn = state.x + K * (data - state.H * state.x)
        val pn = (state.eye - K * state.H) * state.p

        _smoothedValues.append(xn)

        _state = state.from_state(xn, pn)
        state
}
```

3.4.4 Quadratic Regression

The code for calculating the next point after quadratic regression is very simple. The three inputs are the three smoothed points where $d_n = S_{i-2+n}$.

Figure 11: Calculation of the next point of a quadratic regression

```
def quadraticPrediction(d0: Double, d1: Double, d2: Double): Double = {

        val g1 = d1 - d0
        val g2 = d2 - d1
        val delta = g2 - g1
        val g3 = g2 + delta

        d2 + g3

}
```

3.4.5 Profit-Loss Calculation

This code calculates the profit and loss made by a model over time. It does this by assuming that the model starts with 100. It then gets the model to predict a new price which it compares to the previous day. If this price is higher than the previous day it simulates a buy and if it is lower, a sell. These operations assume that the full balance is invested. This balance is tracked over time. This method does not take into account transaction fees.

Figure 12: Evaluation function to find profit/loss of the model

```
1  def evaluate(data: StockHistory, choiceThreshold: Double = 0.02): (List[Double],
       ↪ List[Double]) = {
2
3          trainingStage()
4          correctForDay(predictDay(), data.head)
5
6          var dayBefore = data.head
7          var cash = 100d
8          val plOverTime = ListBuffer[Double](cash)
9          val errorOverTime = ListBuffer[Double]()
10
11         for (day <- data.tail) {
12
13         val prediction = predictDay()
14         correctForDay(prediction, day)
15
16         val sign = if (prediction > dayBefore._2 * (1 + choiceThreshold)) 1
17         else if (prediction < dayBefore._2 * (1 - choiceThreshold)) -1
18         else 0
19
20         val percentageError = (day._2 - prediction) / day._2
21         errorOverTime.append(percentageError)
22
23         cash *= 1 + (sign * (day._2 - dayBefore._2) / dayBefore._2)
24
25         plOverTime.append(cash)
26
27
28         dayBefore = day
29
30         }
31
32         (plOverTime.toList, errorOverTime.toList)
33
34 }
```

3.4.6 Experimental Application

The actual experiment was performed using the following code. It opens all of the stock data for the stocks listed in Appendix A and then finds the simulated profit/loss. This is averaged over all of the stocks so that we can compare the models.

Figure 13: Evaluation function to find profit/loss of the model

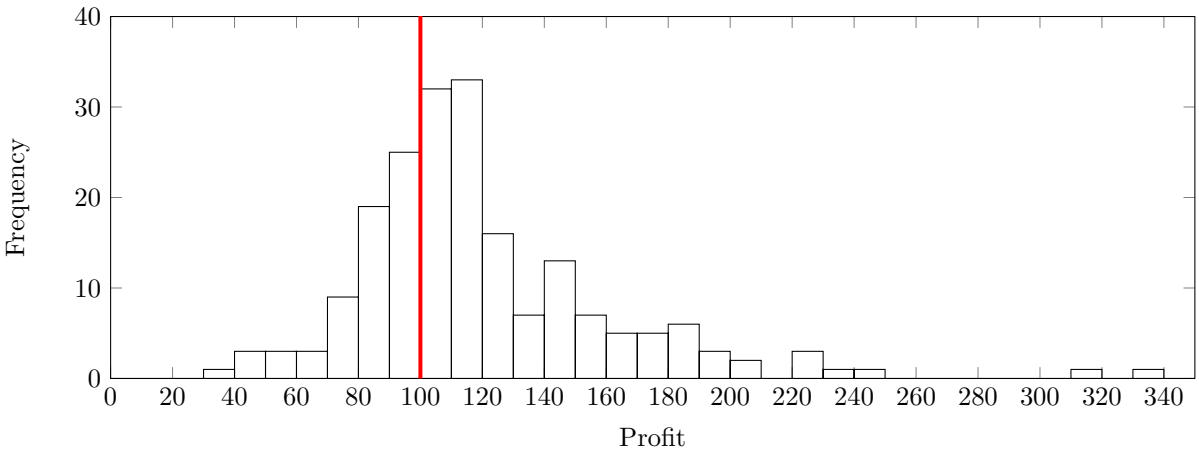
```
object KalmanTest extends App {

        val STOCKS = List("III","ADN","ADM", ..., "XLNX","YHOO")
        val stockData = STOCKS.map(X => Utils.readStock(X))
        val trainingData = stockData.map(_.take(10))
        val testData = stockData.map(_.drop(10))

        val qs = ListBuffer[Double]()
        val rs = ListBuffer[Double]()

        val kpls = ListBuffer[Double]()
        val spls = ListBuffer[Double]()
        val wpls = ListBuffer[Double]()

        for ((stock, i) <- stockData.zipWithIndex) {

                val KALPred = new KalmanPredictor(0.08,0.96,stock.take(50),false)
                val KALPL = KALPred.evaluate(
                                stock.take(50),
                                stock.drop(50),
                                0.005
                        )._1

                val period = 15

                val SMAPred = new SMAPredictor(period)
                val SMAPL = SMAPred.evaluate(
                                stock.drop(50-period).take(period),
                                stock.drop(50),
                                0d
                        )._1

                val WMAPred = new WMAPredictor(period)
                val WMAPL = WMAPred.evaluate(
                                stock.drop(50-period).take(period),
                                stock.drop(50),
                                0d
                        )._1

                spls.append(SMAPL.last)
                kpls.append(KALPL.last)
                wpls.append(WMAPL.last)
        }

        println(s"Test over ${STOCKS.length} stocks")

        println(s"Kalman Filter:")
        println(s"Average profit: ${kpls.map(_ - 100).sum / kpls.length}")
        println(s"Stocks in profit %: ${kpls.count(_ > 100) * 100d/ kpls.length}")

        println(s"SMA:")
        println(s"Average profit: ${spls.map(_ - 100).sum / spls.length}")
        println(s"Stocks in profit %: ${spls.count(_ > 100) * 100d/ spls.length}")

        println(s"WMA:")
        println(s"Average profit: ${wpls.map(_ - 100).sum / wpls.length}")
        println(s"Stocks in profit %: ${wpls.count(_ > 100) * 100d/ wpls.length}")
}
```

4 Results

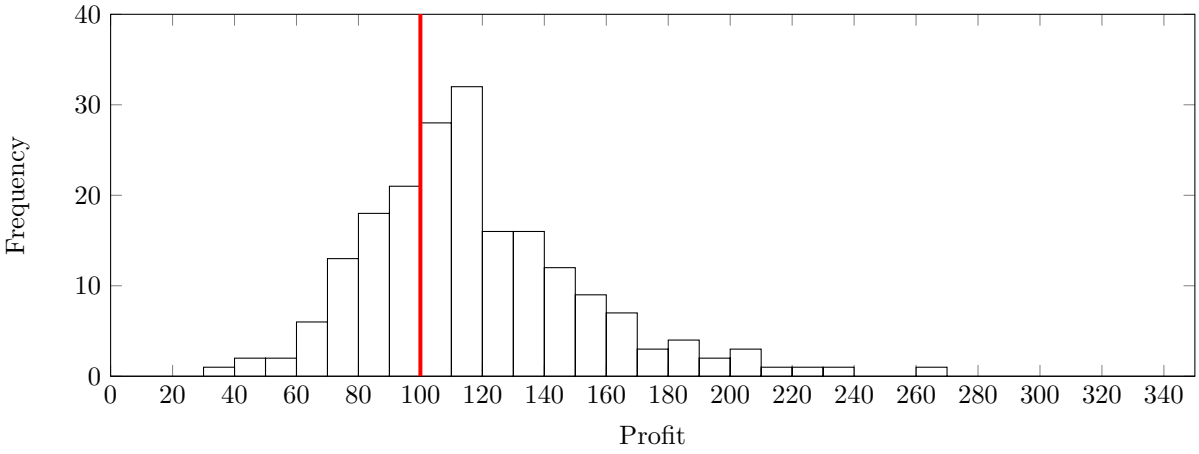
When we ran the experiment we collected the final profit that each model made on each stock. This data is listed fully in Appendix B. Bellow are histograms showing how the profits were distributed. Here 100% profit means thhat the filter broke even. There is a line at 100 to make the histograms easier to read.

Figure 14: Profit distributions for the three models

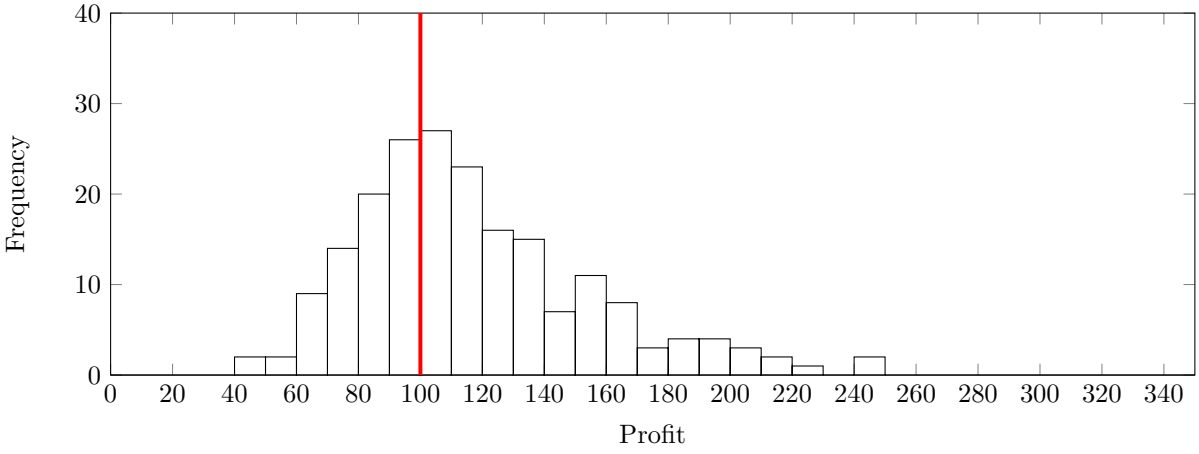
(a) Kalman filter profits distribution



(b) Weighted moving average profits distribution



(c) Simple moving average profits distribution



Bellow is a table of the average profits that the models made over all 199 stocks. The percentage average profit corresponds to an increase by that amount. The percentage stocks in profit is the number of where the final profit was above 0%.

Figure 15: Average profits for the three filters

Filter	Average Profit(%)	Stocks in Profit (%)
Kalman	19.84	68.34
SMA	17.52	63.31
WMA	17.81	68.34

5 Conclusion

In conclusion it is clear that the Kalman filter is better at predicting the movements of stock prices. The Kalman model achieved higher average profits than both the simple and weighted moving average filters. It was better than both by at least two percentage points.

It also performed better than the simple moving average in the "Stocks in Profit" metric. Interestingly it achived exactly the same result here as the weighted moving average. This suggests that perhaps they were both strong on the same stocks, ones with patterns that lend themselves to prediction with smoothing.

The Kalman filter also performed better on certain individual stocks where it was able to reach profits in the 300-350% range. These results were not common but show that the Kalman filter was able in some cases to predict the movements of stocks with great accuracy.

Overall it is clear that whilst all three of these models provide a significant improvement on chance when it comes to predicting stocks the Kalman filter is the strongest. The fact that there are stocks that made a loss on all three models shows that these filters are not enough on their own to predict stocks consistently.

6 Moving Forward

The experiment that we have presented here was the simplest test can be used to determine the effectiveness of these models. In the future it would be possible to implement enhancements to this method which might improve their average profits. This is most relevant to the Kalman filter which, because it is the most complex model, lends itself to being enhanced. These models could also be used as an input to a wider prediction system, providing one basis for the predictions.

6.1 Autocorrelation

When we filter data sets to get smoothed curves, while it is relatively simple to extrapolate the smoothed curve to predict the next state, it is much more difficult to predict the measurement, due to the fact that there is added noise. However, by taking the smoothed curve across the entire data set, we can determine the residuals for each day of the data set. These are the difference between the predicted state and the true measurement. We can use those to try and predict the next residual, which when added to the extrapolated smoothing, would give a prediction for the actual measurement. Predicting the actual measurements is important because that it the actual price that we can trade at instead of the nebulous "true" value that doesn't feature in the actual market.

However, these residuals are stochastic in nature, and as such difficult to predict conventionally. It is therefore necessary to use a process called auto-correlation to find the next data point. Partial autocorrelation functions can reveal the order of autocorrelation of the residual data (how previous data points relate to upcoming ones), and then an autoregressive model can be used, with that order, to predict the next value, providing a more accurate estimate of the real measurement.

6.2 Stock Selection

When we were testing the filters we observed that their performance varies enormously based on the type particular stock it is testing. As such, we wondered whether it would be possible to increase the performance of these by excluding stocks that met certain criteria. This would improve the average

profits if we could identify stocks that the models struggle with. The data sets are large enough that statistical analysis is significant. We tested whether excluding stocks with high standard deviation in their prices and in the filter error would improve profits. In order to judge this across all data sets, it was important to normalise this data, and as such dividing by the mean, or in the case of the residuals, the mean of the absolute values (because residuals are distributed around 0) allows us to compare them. When we plotted standard deviation against profits for the Kalman filter, we observed a light correlation, and found that not trading stocks with a high standard deviation of the error allowed for an increase in average profits of nearly 40%.

6.3 Neural Networks

Neural networks are computer programs designed to mimic, loosely, human brains. They can be trained to perform a wide range of tasks, including pattern recognition and prediction. In theory they could be applied to the world of stock trading to provide a better way of identifying stock movements in advance. This application has been tested in the past [Kar, 2014] with success. The problem that these neural networks can face is that noise makes discerning a pattern more difficult. Use of a Kalman filter to reduce this noise could allow a hybrid system that can make use of the best of both worlds.

7 References

References

[Esme, 2017] Esme, B. (2017). Kalman filter for dummies. http://bilgin.esme.org/BitsAndBytes/KalmanFilterforDummies. [Online; accessed 15-May-2017].

[Kar, 2014] Kar, A. (2014). Stock prediction using artificial neural networks. https://people.eecs.berkeley.edu/~akar/IITK_website/EE671/report_stock.pdf. [Online; accessed 27-May-2017].

[Wikipedia, 2017a] Wikipedia (2017a). Kalman filter — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Kalman_filter&oldid=778501475. [Online; accessed 27-May-2017].

[Wikipedia, 2017b] Wikipedia (2017b). Moving average — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Moving_average&oldid=782237858. [Online; accessed 27-May-2017].

Appendices

A Stocks Tested

III	GLEN	RDSA	AAL	FAST	ORLY
ADN	HMSO	RMG	AMGN	FISV	PCAR
ADM	HL.	RSA	ADI	GILD	PAYX
AAL	HIK	SGE	AAPL	HAS	PYPL
ANTO	HSBA	SBRY	AMAT	HSIC	QCOM
AHT	IMB	SDR	ADSK	HOLX	REGN
ABF	ISAT	SVT	ADP	IDXX	ROST
AZN	IHG	SHP	BIDU	ILMN	SHPG
AV.	IAG	SKY	BIIB	INCY	SIRI
BAB	ITRK	SN.	BMRN	INTC	SWKS
BA.	INTU	SMIN	AVGO	INTU	SBUX
BARC	ITV	SPD	CA	ISRG	SYMC
BDEV	JMAT	SSE	CELG	JBHT	TMUS
BLT	KGF	STAN	CERN	JD	TSLA
BP.	LAND	SL.	CHTR	KLAC	TXN
BATS	LGEN	STJ	CHKP	LRCX	KHC
BLND	LLOY	TW.	CTAS	LBTYA	PCLN
BT.A	LSE	TSCO	CSCO	LBTYK	TSCO
BNZL	MKS	TPK	CTXS	LILA	TRIP
BRBY	MERL	TUI	CTSH	LILAK	FOXA
CPI	MNDI	ULVR	CMCSA	LVNTA	ULTA
CCL	NG.	UU.	COST	QVCA	VRSK
CNA	NXT	VOD	CSX	MAT	VRTX
CCH	OML	WTB	CTRP	MXIM	VIAB
CPG	PSON	WOS	XRAY	MCHP	VOD
CRH	PSN	WPG	DISCA	MU	WBA
DCC	PFG	WPP	DISCK	MSFT	WDC
DGE	PRU	ATVI	DISH	MDLZ	XLNX
DLG	RRS	ADBE	DLTR	MNST	YHOO
EZJ	RB.	AKAM	EBAY	MYL	
EXPN	REL	ALXN	EA	NTES	
FRES	RIO	GOOG	EXPE	NFLX	
GKN	RR.	GOOGL	ESRX	NCLH	
GSK	RBS	AMZN	FB	NVDA	

B Full Data

The full experimental data is presented here for completeness. Each column shows the final profit made by each model on each stock. For example P_{kalman} is the profit made by the kalman filter on each stock.

Table 1: Full experimental data

i	Stock	P_{kalman}	P_{SMA}	P_{WMA}	i	Stock	P_{kalman}	P_{SMA}	P_{WMA}
0	III	82.14	114.31	83.36	100	GOOGL	106.72	96.47	123.51
1	ADN	100.93	162.61	141.25	101	AMZN	101.66	117.22	119.98
2	ADM	142.85	156.77	163.86	102	AAL	106.59	108.66	157.36
3	AAL	106.59	108.66	157.36	103	AMGN	70.74	77.64	79.05
4	ANTO	86.67	83.96	112.19	104	ADI	114.06	87.93	104.26
5	AHT	149.16	196.25	169.94	105	AAPL	86.44	66.21	64.31
6	ABF	89.59	98.99	110.13	106	AMAT	85	92.25	91.25
7	AZN	77.94	55.14	62.33	107	ADSK	109.13	139.73	117.75
8	AV.	222.1	248.37	197.84	108	ADP	104.83	97.73	102.04
9	BAB	116.87	136.23	119.11	109	BIDU	117.26	117.36	120.5
10	BA.	111.05	135.51	128.34	110	BIIB	85.3	65.94	95
11	BARC	157.89	138.87	114.97	111	BMRN	194.59	111.46	148.23
12	BDEV	182.33	142.48	151.53	112	AVGO	83.65	100.97	74.14
13	BLT	71.53	75.28	90.65	113	CA	128.13	132.44	147.06
14	BP.	94.07	88.74	84.47	114	CELG	93.74	73.7	85.23
15	BATS	116.91	113.98	114.4	115	CERN	66.79	85.38	83.29
16	BLND	177.88	205.77	173.23	116	CHTR	105.61	132.68	114.5
17	BT.A	189.5	113.36	117.02	117	CHKP	122.4	89.56	106.15
18	BNZL	117.34	118.84	128.09	118	CTAS	89.46	100.01	84.56
19	BRBY	116.18	80.79	91.85	119	CSCO	146.05	113.38	130.46
20	CPI	63.15	102.12	72.09	120	CTXS	198.26	185.71	205.97
21	CCL	115.99	133.25	113.08	121	CTSH	155.35	168.27	163.57
22	CNA	105.76	111.19	108.44	122	CMCSA	143.14	113.5	115.62
23	CCH	77.45	73.89	82.18	123	COST	335.78	201.71	225.91
24	CPG	116.79	85.4	94.03	124	CSX	122.09	101.15	107.24
25	CRH	111.04	117	125.63	125	CTRP	108.4	68.22	73.57
26	DCC	96.89	111.26	90.47	126	XRAY	132.05	155.49	162.84
27	DGE	106.26	106.61	100.08	127	DISCA	113.91	129.14	135.09
28	DLG	182.76	177.13	173.35	128	DISCK	95.65	108.03	109.55
29	EZJ	100.23	92.93	84.13	129	DISH	117.93	122.02	151.78
30	EXPN	99.26	89.9	107.73	130	DLTR	99.26	106.37	88.79
31	FRES	30.87	50.57	38.71	131	EBAY	110.38	123.35	117.44
32	GKN	112.59	109.2	107.69	132	EA	138.4	107.44	133.84
33	GSK	71.48	69.69	76.67	133	EXPE	104.57	115.29	94.08
34	GLEN	92.23	82.69	122.53	134	ESRX	98.44	120.19	94.55
35	HMSO	207.65	208.52	263.7	135	FB	122.94	116.49	107.03
36	HL.	126.18	138.56	114.87	136	FAST	77.01	89.28	94.22
37	HIK	129.31	86.38	91.51	137	FISV	112.21	103.26	100.09
38	HSBA	103.86	98.69	90.57	138	GILD	103.18	107.12	84.74
39	IMB	123.84	118.41	128.3	139	HAS	134.68	128.12	117.43
40	ISAT	45.73	44.61	45.49	140	HSIC	97.77	97.21	103.05
41	IHG	59.69	66.54	59.89	141	HOLX	113.08	137.54	147.64
42	IAG	239.43	145.58	187.05	142	IDXX	115.16	112.65	115.9
43	ITRK	103.15	100.05	104.1	143	ILMN	142.22	139.25	142.05
44	INTU	141.42	123.35	126.92	144	INCY	90.36	73.18	76.39
45	ITV	223.16	187.75	192.53	145	INTC	98.28	98.11	102.67
46	JMAT	91.53	98.45	98.41	146	INTU	141.42	123.35	126.92
47	KGF	143.24	108.13	105.5	147	ISRG	119.65	96.79	118.2
48	LAND	222.26	163.3	182.8	148	JBHT	99.46	115.77	107.46
49	LGEN	156.01	184.23	137.93	149	JD	147.11	228.68	202.09

Table 1: Full experimental data

i	Stock	P_{kalman}	P_{SMA}	P_{WMA}	i	Stock	P_{kalman}	P_{SMA}	P_{WMA}
50	LLOY	171.81	163.58	120.09	150	KLAC	96.79	91.09	94.5
51	LSE	120.18	145.06	153.22	151	LRCX	90.13	105.29	95.47
52	MKS	197.57	241.67	166.45	152	LBTYA	155.94	167.95	161.35
53	MERL	110.24	95.18	112.19	153	LBTYK	84.89	150.57	125.59
54	MNDI	124.78	145.26	141.07	154	LILA	57.19	71.07	57.68
55	NG.	119.55	102.83	114.74	155	LILAK	42.62	83.86	69.99
56	NXT	146.45	189.64	148.46	156	LVNTA	83.14	83.07	79.48
57	OML	115.05	84.55	98.25	157	QVCA	82.31	96.56	80.36
58	PSON	70.69	64.26	69.97	158	MAT	106.11	158.25	145.74
59	PSN	310.03	211.12	154.03	159	MXIM	117.57	121.66	113.93
60	PFG	80.45	74.43	69.12	160	MCHP	111.94	93.2	95.74
61	PRU	167.93	140.07	137.56	161	MU	74.45	76.26	83.23
62	RRS	99.68	90.67	112.26	162	MSFT	131.92	127.92	123.78
63	RB.	185.21	118.31	145.78	163	MDLZ	169.84	168.3	179.82
64	REL	80.51	75.96	80.65	164	MNST	208	150.35	154.77
65	RIO	180.08	115.65	182.89	165	MYL	112.55	104.96	98.05
66	RR.	109.07	142.41	119.44	166	NTES	122.09	66.82	79.69
67	RBS	91.75	171.43	98.79	167	NFLX	144.33	119.22	113.16
68	RDSA	155.94	121.45	135.77	168	NCLH	111.62	114.34	115.69
69	RMG	104.31	133.97	151.3	169	NVDA	48.34	43.01	41.23
70	RSA	135.79	152.71	146.07	170	ORLY	152.83	155.71	146.67
71	SGE	107.25	91.03	106.5	171	PCAR	132.75	139.17	132.36
72	SBRY	126.18	120.99	115.23	172	PAYX	81.43	80.6	76.93
73	SDR	144.84	161.57	130.04	173	PYPL	123.98	120.56	119.19
74	SVT	173.25	147.52	150.7	174	QCOM	81.53	86.8	85.98
75	SHP	57.49	86.04	75.96	175	REGN	98.2	77.86	93.93
76	SKY	95.95	88.44	105.19	176	ROST	156.13	120.62	137.25
77	SN.	95.89	89.34	94.7	177	SHPG	90.82	97.64	119.2
78	SMIN	91.77	109.21	113.7	178	SIRI	121.72	113.41	127.32
79	SPD	143.02	193.2	123.28	179	SWKS	105.38	77.8	83.79
80	SSE	168.2	150.72	146.04	180	SBUX	103.71	101.34	103.51
81	STAN	82.37	104.3	112.53	181	SYMC	115.97	93.64	108.19
82	SL.	102.21	93.08	112.43	182	TMUS	111.9	128.07	119.05
83	STJ	176.88	211.58	187.71	183	TSLA	133.04	65.28	75.82
84	TW.	247.96	199.04	200.02	184	TXN	112.44	106.39	102.68
85	TSCO	103.22	151.91	137.55	185	KHC	125.8	136.73	134.38
86	TPK	180.79	196.94	218.8	186	PCLN	122.2	117.63	129.27
87	TUI	177.98	168.1	231.24	187	TSCO	103.22	151.91	137.55
88	ULVR	88.7	93.6	88.81	188	TRIP	113.1	139.84	127.95
89	UU.	160.59	150.8	139.86	189	FOXA	111.35	94.01	109.41
90	VOD	103.77	79.72	81.86	190	ULTA	92.17	105.24	116.94
91	WTB	88.73	98.01	76.94	191	VRSK	115.26	136.83	135.89
92	WOS	126.28	108.72	104.98	192	VRTX	74.66	80.01	78.51
93	WPG	106.76	123.7	113.41	193	VIAB	113.75	175.47	165.16
94	WPP	107.89	100.54	105.74	194	VOD	103.77	79.72	81.86
95	ATVI	109.24	98.75	138.84	195	WBA	112.63	103.53	109.38
96	ADBE	94.2	92.91	92.33	196	WDC	69.44	70.04	87.07
97	AKAM	109.69	100.18	100.54	197	XLNX	94.08	97.59	105.95
98	ALXN	89.32	61.94	66.34	198	YHOO	166.68	129.25	133.43
99	GOOG	103.94	99.58	107.65					