The George Washington University

Laboratory Four:

Comparing Simple Time Series Forecast Methods

Fernando Zambrano

DATS 6450: Multivariate Modeling

Dr. Reza Jafari

21 February 2020

Abstract:

One-step-ahead forecasts of an arbitrary time series are calculated using four simple forecasting methods. These methods are average, naïve, drift, and simple exponential smoothing (SES). The forecasts for each method will be subject to cross-validation to estimate the forecast error. The forecast errors are used to develop several metrics that evaluate each method. Calculations are done first by hand and then implemented in Python using NumPy and Pandas libraries**.**

Introduction:

One of the toughest decisions in forecasting is choosing a model. Several models can be applied to different types of data and contexts. This may lead to a paradox of choice – with more available options, the more difficult it becomes to choose which option. In such a case, it is best to start with simple methods that have already been developed. Do not associate "simple" with inaccurate or useless. These methods are simple due to their relatively simple implementation and easy mathematical understanding. The level of complexity of a model does not necessarily imply better accuracy. On the contrary, embracing Occam's razor, simple methods may outperform complex models *because* of their simplicity. Simple forecaster methods offer a baseline to compare more complex models. If a complex model can outperform a simple one then it may be worth using, but not if it cannot outperform a simple one. The way to evaluate and compare models is to measure and understand their resulting error. There are several ways to calculate the error, with each method capturing more information about the errors.

Methods & Theory:

There are four main simple methods, average, naïve, drift, and SES.

The average method uses the mean of the historical data as the forecast for all future values. In other words, the sum of all historical values divided by T, the total number of points. This method assigns equal weights to each data point. This means that the points that are old or recent are equally important in determining the new prediction point. This produces what is called a "flat" forecast. Since all future values are forecasted with the same value, they produce a horizontal line. The formula below calculates the h-step forecast. This method is often used for predicting short term trends

$$\hat{y}_{T+h|T} = \frac{y_1 + y_2 + \ldots + y_T}{T}$$

.

The naïve method is quite different than the average method, by assigning all of the importance of predicting future values on the most recent data point. The naïve method also produces a flat forecast. This method is often used for economic and financial data or for data that may follow a "random walk" behavior. The formula for the naïve method is below.

$$\hat{y}_{T+h|T} = y_T$$

The drift method is almost similar to regression and with a combination of the naïve method. Instead of calculating the equation of a line that will best fit all the data points, the drift line only passes through the first and last historical data points. This line then uses the last data point as its y-intercept. The drift line is then using extrapolation to predict all future values. Unlike previous methods, drift is not a flat prediction, since all future values fall on a line with a non-zero slope. The equation for drift is below. Yt is the last historical data point. T is the total number of data points, and h is the number of time steps ahead to predict.

$$\hat{y}_{T+h|T} = y_T + \frac{h}{T-1}\sum_{t=2}^{T}(y_t - y_{t-1}) = y_T + h(\frac{y_T - y_1}{T-1})$$

The final method is SES. This method is a compromise between the extremes of using the average and naïve methods. Forecast values are predicted assigning weighted averages to the historical data depending on how old they are. The oldest points have the lowest weights, and the most recent points will have the highest weights. The weights decrease exponentially. The parameter α (alpha), controls the rate at which the weights decrease. Alpha can take a value between 0 and 1. Alpha values closer to 1 give a higher weight to more recent data points. This means more recent data has a stronger influence on forecasting values. If alpha is equal to 1, then the SES method becomes the naïve method, since all the weight belongs to the last data point. In addition to alpha, ℓ0, which is the initial condition or starting point to begin calculating the weights for each data point. But how does one choose ℓ0 and alpha? These parameters are estimated through optimization that minimizes the sum of squared errors (SSE), but this process goes beyond the scope of this study. Instead, alpha and ℓ0 will be given. SES will result in a flat prediction similar to average and naïve methods. There are two ways of calculating SES, one involves the full equation, while the other involves the component form. Below are both formulas starting with the full equations.

$$\boxed{\hat{y}_{T+1|T} = \sum_{j=0}^{T-1}\alpha(1-\alpha)^j y_{T-j} + (1-\alpha)^T \ell_0}$$

Component Form

$$\begin{aligned} &\mathit{ForecastingEquation} && \hat{y}_{T+h|T} = \ell_T \\ &\mathit{SmoothingEquation} && \ell_T = \alpha y_T + (1-\alpha)\ell_{T-1} \end{aligned}$$

Correlation is a dimensionless measurement that quantifies the linear relationship between two independent features.

$$r = \frac{\sum (x_t - \bar{x})(y_t - \bar{y})}{\sqrt{\sum (x_t - \bar{x})^2} \sqrt{\sum (y_t - \bar{y})^2}}.$$

$$\hat{\tau}_k = \frac{\sum_{t=k+1}^{T} (y_t - \overline{y})(y_{t-k} - \overline{y})}{\sum_{t=1}^{T} (y_t - \overline{y})^2}$$

The numerator of this equation is the covariance between two variables. Covariance determines how linearly related two features are associated. The issue with why covariance is not used as the principle statistic to demonstrate the strength of a linear relationship is that it maintains the units of its features. This makes it difficult to draw comparisons between two features that have large differences in their spread. To compare two features without the drawback of dimensionality, the features need to be normalized. The denominator is the product of the square root of the squared variance of each feature. This process cancels out the dimensions of each feature and forces the resulting value of *r* to fall between -1 and +1. The absolute value of r demonstrates the strength of the correlation between two variables. If the correlation is positive, then both features increase and decrease correspondingly. If the correlation is negative, then both features associate inversely. As one feature increases, then the other feature decreases and vice averse.

Autocorrelation, *T*, is the correlation of a series with itself with some amount of lag *k*, instead of another feature. Lag is the number of time steps between two points in a series. Autocorrelation ranges in values from -1 to + 1. The formula for calculating the autocorrelation with k lags is the autocorrelation function (ACF) which is below.

An example of calculating autocorrelation is to create a simple series, x(t) = [2,3,4] compared to itself with lag k = 0. It might be helpful to think of comparing two different series, x(t), and the lagged version of x(t), y(t). Since k = 0, there is no lag and the autocorrelation would be estimating the correlation of x(t) = [2,3,4] with y(t) = [2,3,4]. The resulting *T* is 1.0 since there is a perfect linear relationship between two series with the same value at each respective time step. If k =1, then there would be a lag of 1 timestep and *T* would be estimating the correlation between x(t) = [2,3] and y(t) = [3,4]. The resulting *T* is no longer 1.0 since the exact points are no longer being compared. Instead, the first value in x(t), 2, is compared to the value that immediately succeeds it, which is the first value in y(t), 3. This demonstrates how to implement lag by one step. If k = 2, then the autocorrelation between x(t) and y(t) is calculated by estimating *T* between x(t) = [2] and y(t) = [4] because 4 is two time-steps behind 2. A correlogram shows the correlation of a series at multiple lag points and is also known as an ACF plot. The visual interpretation of the series' autocorrelation can provide insight into underlying ways in which the data behaves over time that may be missed in simply plotting the raw data.

Visualizing an ACF plot is not only limited to the dataset itself, but also the residuals of a regression model. If the residuals resemble the steps of a person going on a random walk, then the residuals are considered to be "white noise." When the residuals are white noise, then all of the underlying patterns in the time series have been captured by the model. The residuals of white noise should have an autocorrelation of 1 at lag = 0, and autocorrelation near 0 at all other lag points. However, when models do not fully capture the underlying patterns in the data, the residuals will show obvious signs of high autocorrelation. These kinds of residuals are considered as "colored" or "pink" noise. Hence, ACF plots illustrate a method to run diagnostics on models to make sure there is no unincorporated information in the model.

Forecast evaluation is determined by looking at the forecast errors.

SSE is the measure of how far the prediction is from the actual value.

$$SSE = \sum(e_t^2)$$

If you take the average of SSE you get MSE.

$$MSE = mean(e_t^2)$$

Mean Error is the sum of all the error divided by the number of errors.

Looking at an ACF plot may show if the model is a good fit, but with most visual interpretations it is subjective and requires a more objective analysis. This can be done through a Portmanteau test, which tests a group of autocorrelations. An effective Portmanteau test is the Box Pierce test which sums up the autocorrelation at each lag point except at zero. If there is little autocorrelation in the residuals, then Q will be small and is a good indication that the model is a good fit for the data. The formula for Q is below.

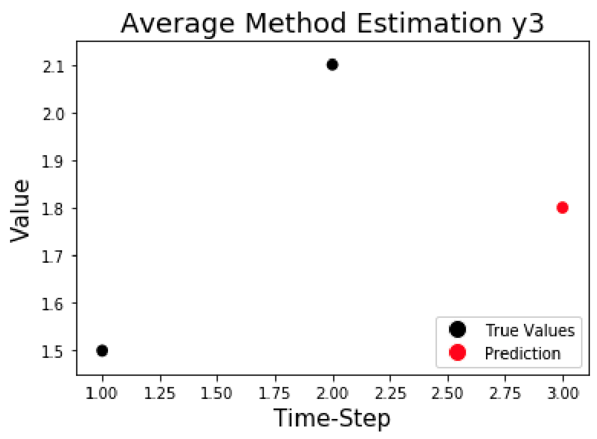
$$Q = T\sum_{k=1}^{h} r_k^2$$

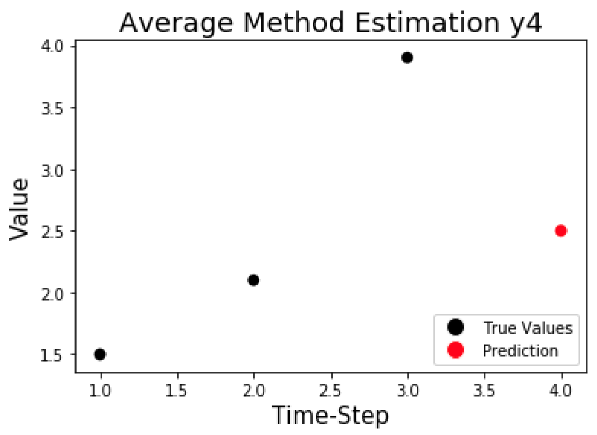
Cross-validation is a series of test sets, each consisting of a single observation where the corresponding training set consists only of observations that occurred prior to the observation that forms the test set. For this study, y1 and y2 will always be the initial training sets, and then add y3, and y4.

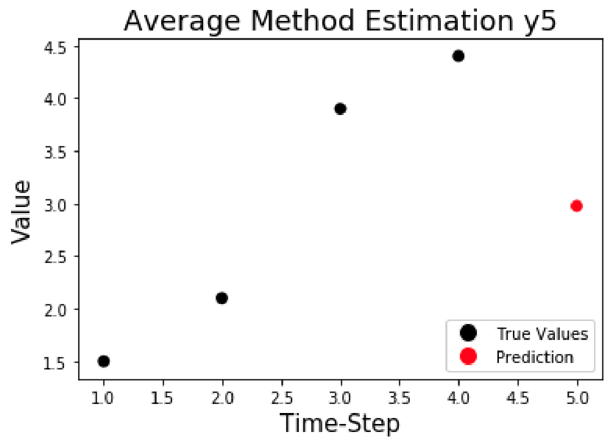
Implementation and Results:

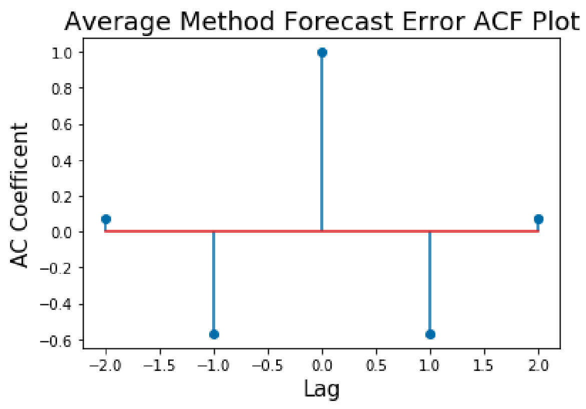
The test set for this study is a small arbitrary data set **y** = [1.5, 2.1, 3.9, 4.4, 5.2]. The implementation of each method on **y** by hand will be in the appendix before the code.

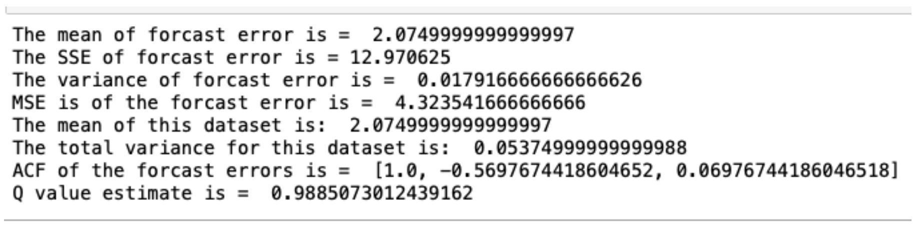
The implementation in Python resulted in created three graphs showing the predicted value at each time step after t =2. The graphs are followed an ACF plot and then finally by an error summary report. The order of methods will start with average, followed by naïve, then drift, and finally SES.



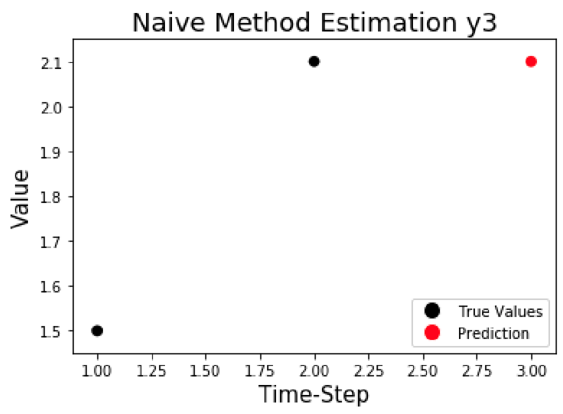


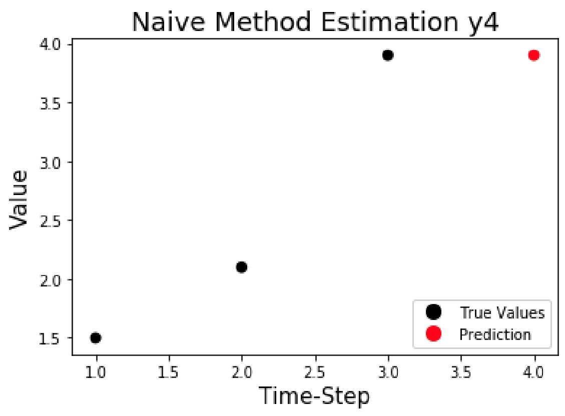


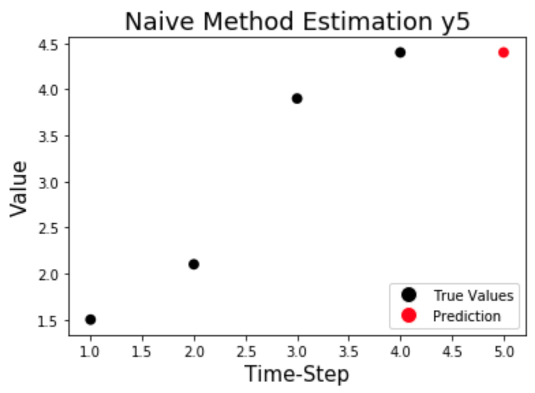


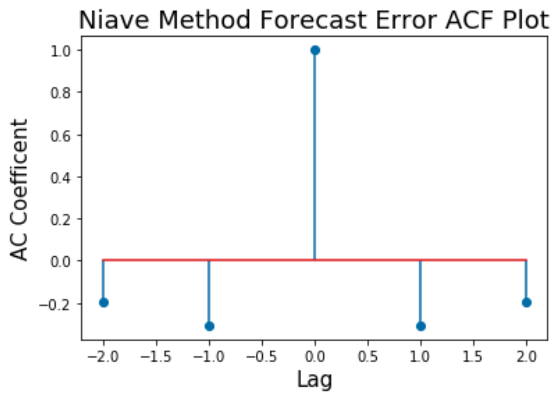


```
The mean of forcast error is =  2.0749999999999997
The SSE of forcast error is = 12.970625
The variance of forcast error is =  0.017916666666666626
MSE is of the forcast error is =  4.323541666666666
The mean of this dataset is:  2.0749999999999997
The total variance for this dataset is:  0.053749999999999988
ACF of the forcast errors is =  [1.0, -0.5697674418604652, 0.06976744186046518]
Q value estimate is =  0.9885073012439162
```

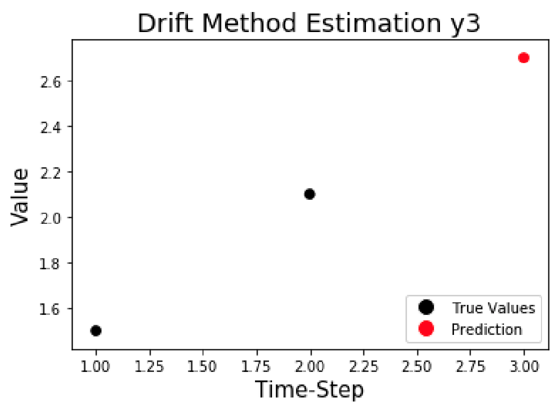


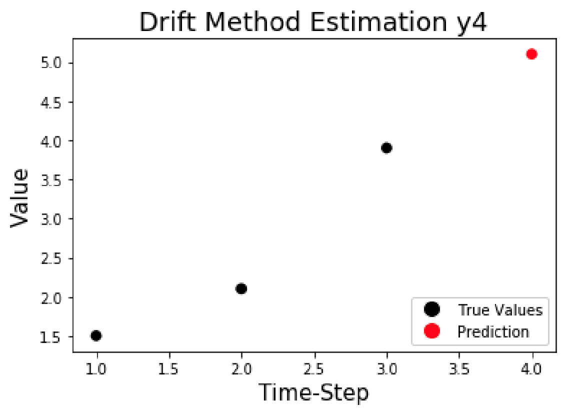


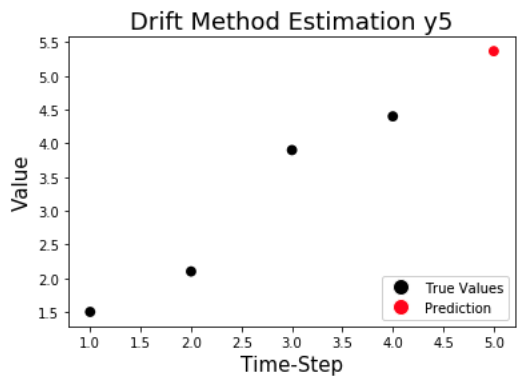


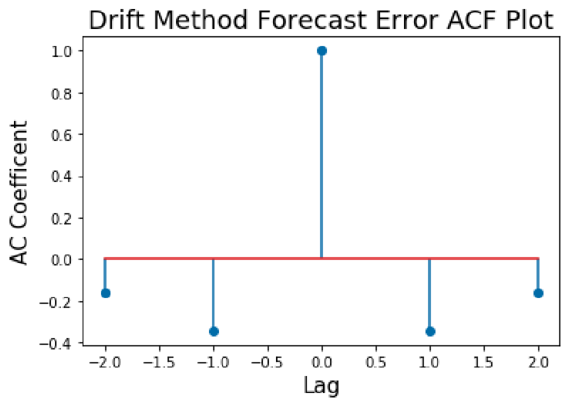


```
The mean of forcast error is =  1.0333333333333334
The SSE of forcast error is = 4.129999999999999
The variance of forcast error is =  0.3088888888888887
MSE is of the forcast error is =  1.3766666666666663
The mean of this dataset is:  1.0333333333333334
The total variance for this dataset is:  0.9266666666666661
ACF of the forcast errors is =  [1.0, -0.306954436450839, -0.19304556354916097]
Q value estimate is =  0.3944628469885959
```

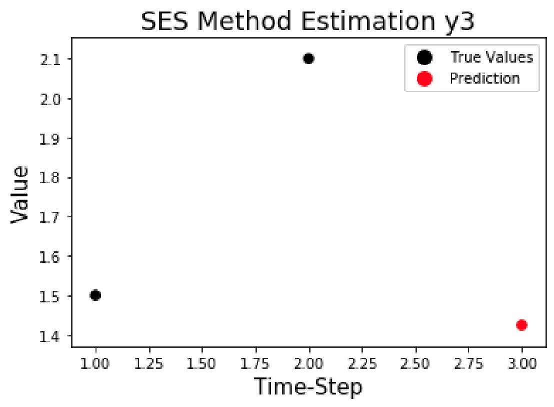


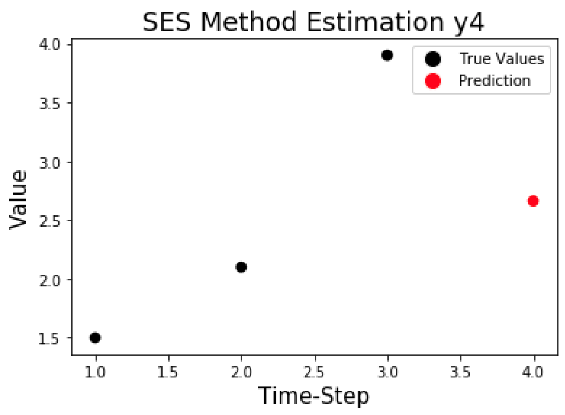


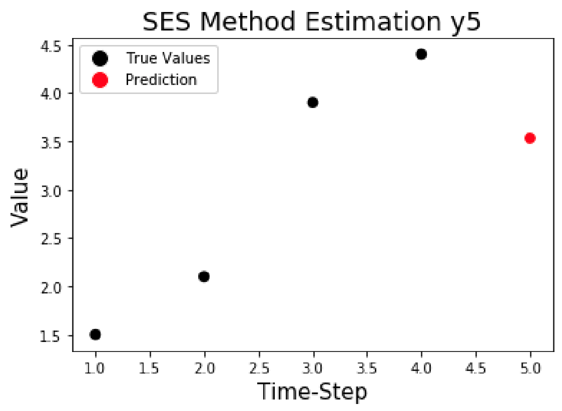


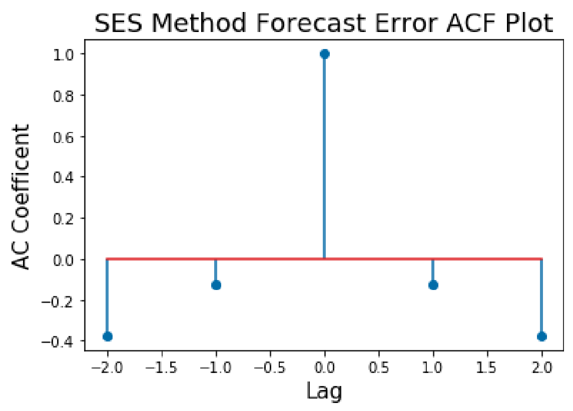


```
The mean of forcast error is =  0.11111111111111116
The SSE of forcast error is = 1.9577777777777763
The variance of forcast error is =  0.6402469135802464
MSE is of the forcast error is =  0.6525925925925921
The mean of this dataset is:  0.11111111111111116
The total variance for this dataset is:  1.9207407407407393
ACF of the forcast errors is =  [1.0, -0.34252474611132505, -0.1574752538886749]
Q value estimate is =  0.4263649718577909
```









```
The mean of forcast error is =  1.960416666666667
The SSE of forcast error is = 11.929257812500001
The variance of forcast error is =  0.1331857638888887
MSE is of the forcast error is =  3.976419270833334
The mean of this dataset is:  1.960416666666667
The total variance for this dataset is:  0.3995572916666661
ACF of the forcast errors is =  [1.0, -0.12436724673575333, -0.37563275326424667]
Q value estimate is =  0.46970153215653054
```

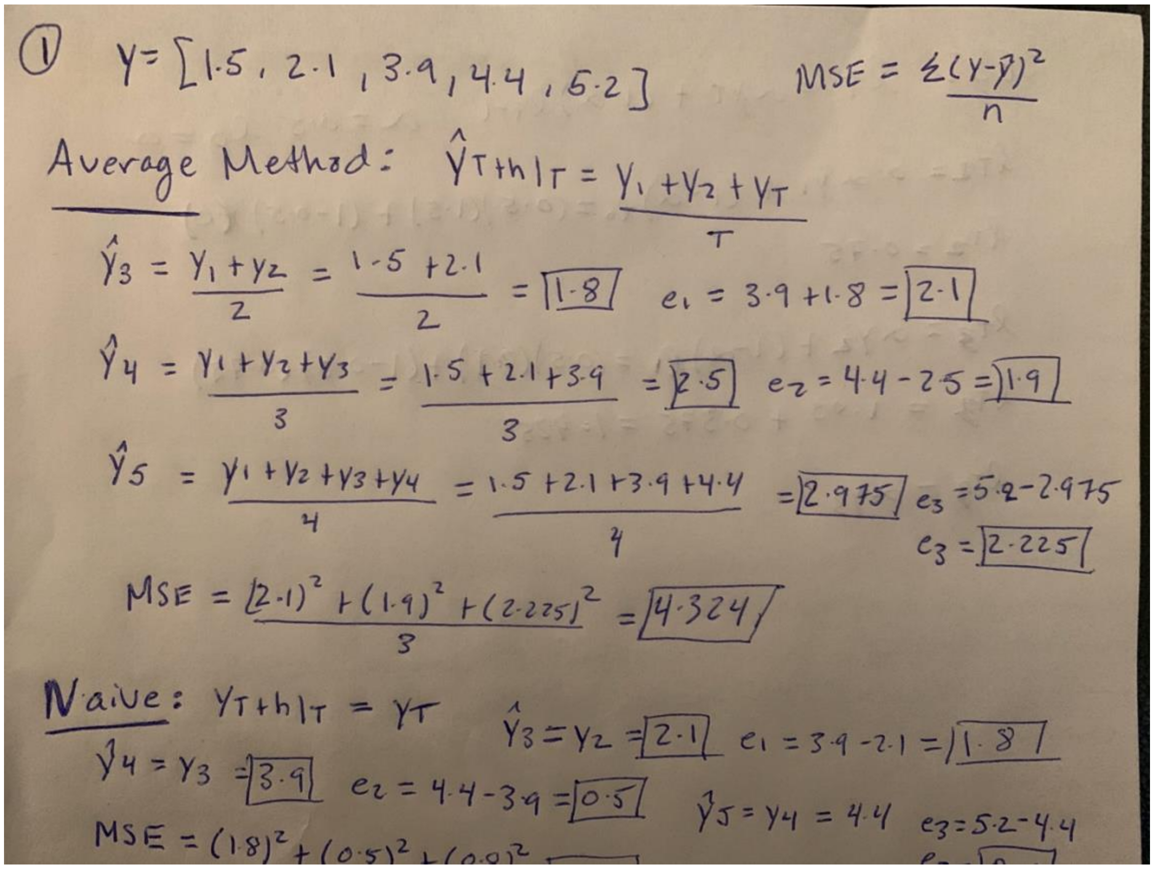
Evaluation Table

Mehtod	Mean Error	SSE	Variance	MSE	ACF	Q
Average	2.075	12.9706	0.0179167	4.32354	[1.0, -0.5697674418604652, 0.06976744186046518]	0.988507
Niave	1.03333	4.13	0.308889	1.37667	[1.0, -0.306954436450839, -0.19304556354916097]	0.394463
Drift	0.111111	1.95778	0.640247	0.652593	[1.0, -0.34252474611132505, -0.1574752538886749]	0.426365
SES	1.96042	11.9293	0.133186	3.97642	[1.0, -0.12436724673575333, -0.37563275326424667]	0.469702

Conclusions:

Overall, comparing the four methods across different error metrics using cross-validation, drift is the best estimator. It has the lowest Mean Error, MSE, and SSE. It also had the second lowest variance and Q. The error metrics demonstrate that across time, the drift method had the best predictions. This is most likely because the data is small and has a positive trend which drift method is great for capturing. The other methods, on the other hand, are flat predictions which means they do not capture the trend characteristic as well as drift thus under performing at each time step. The naïve method was surprisingly not too far behind drift in error terms and had the lowest Q. The average and SES methods performed almost identical, with SES having a slightly lower error metrics than average and much lower Q.

Appendix



① $y = [1.5, 2.1, 3.9, 4.4, 5.2]$ $\qquad MSE = \dfrac{\Sigma(y-\hat{y})^2}{n}$

Average Method: $\hat{y}_{T+h|T} = \dfrac{y_1 + y_2 + y_T}{T}$

$\hat{y}_3 = \dfrac{y_1 + y_2}{2} = \dfrac{1.5 + 2.1}{2} = \boxed{1.8}$ $\quad e_1 = 3.9 + 1.8 = \boxed{2.1}$

$\hat{y}_4 = \dfrac{y_1 + y_2 + y_3}{3} = \dfrac{1.5 + 2.1 + 3.9}{3} = \boxed{2.5}$ $\quad e_2 = 4.4 - 2.5 = \boxed{1.9}$

$\hat{y}_5 = \dfrac{y_1 + y_2 + y_3 + y_4}{4} = \dfrac{1.5 + 2.1 + 3.9 + 4.4}{4} = \boxed{2.975}$ $\quad e_3 = 5.2 - 2.975$

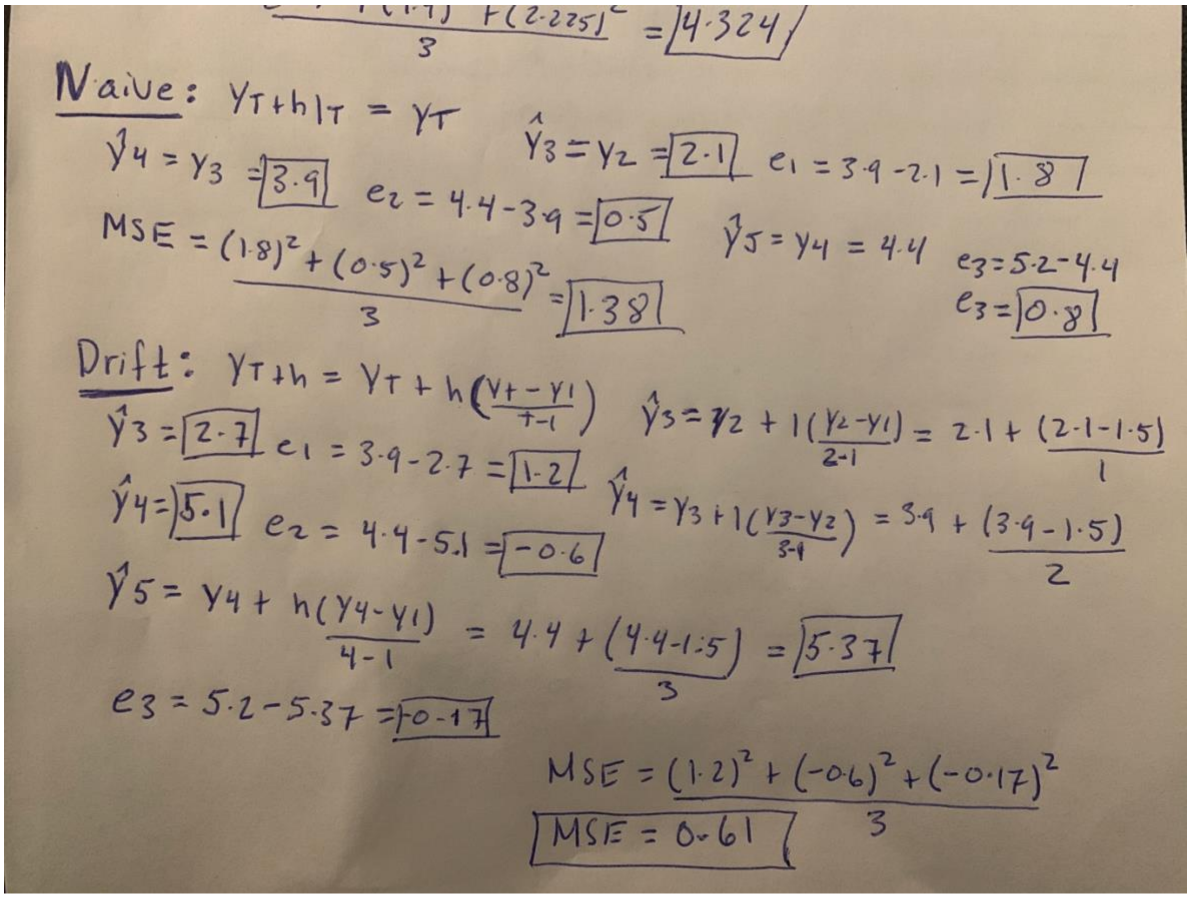
$e_3 = \boxed{2.225}$

$MSE = \dfrac{(2.1)^2 + (1.9)^2 + (2.225)^2}{3} = \boxed{4.324}$

Naive: $y_{T+h|T} = y_T$ $\qquad \hat{y}_3 = y_2 = \boxed{2.1}$ $\quad e_1 = 3.9 - 2.1 = \boxed{1.8}$

$\hat{y}_4 = y_3 = \boxed{3.9}$ $\quad e_2 = 4.4 - 3.9 = \boxed{0.5}$ $\qquad \hat{y}_5 = y_4 = 4.4$ $\quad e_3 = 5.2 - 4.4$

$MSE = (1.8)^2 + (0.5)^2 + (0.8)^2$



$$\frac{\ldots + (2.225)^2}{3} = \boxed{4.324}$$

Naive: $y_{T+h|T} = y_T$

$\hat{y}_3 = y_2 = \boxed{2.1}$ $e_1 = 3.9 - 2.1 = \boxed{1.8}$

$\hat{y}_4 = y_3 = \boxed{3.9}$ $e_2 = 4.4 - 3.9 = \boxed{0.5}$

$\hat{y}_5 = y_4 = 4.4$ $e_3 = 5.2 - 4.4$

$e_3 = \boxed{0.8}$

$$MSE = \frac{(1.8)^2 + (0.5)^2 + (0.8)^2}{3} = \boxed{1.38}$$

Drift: $y_{T+h} = y_T + h\left(\frac{y_T - y_1}{T-1}\right)$

$\hat{y}_3 = y_2 + 1\left(\frac{y_2 - y_1}{2-1}\right) = 2.1 + \frac{(2.1 - 1.5)}{1}$

$\hat{y}_3 = \boxed{2.7}$ $e_1 = 3.9 - 2.7 = \boxed{1.2}$

$\hat{y}_4 = y_3 + 1\left(\frac{y_3 - y_2}{3-1}\right) = 3.9 + \frac{(3.9 - 1.5)}{2}$

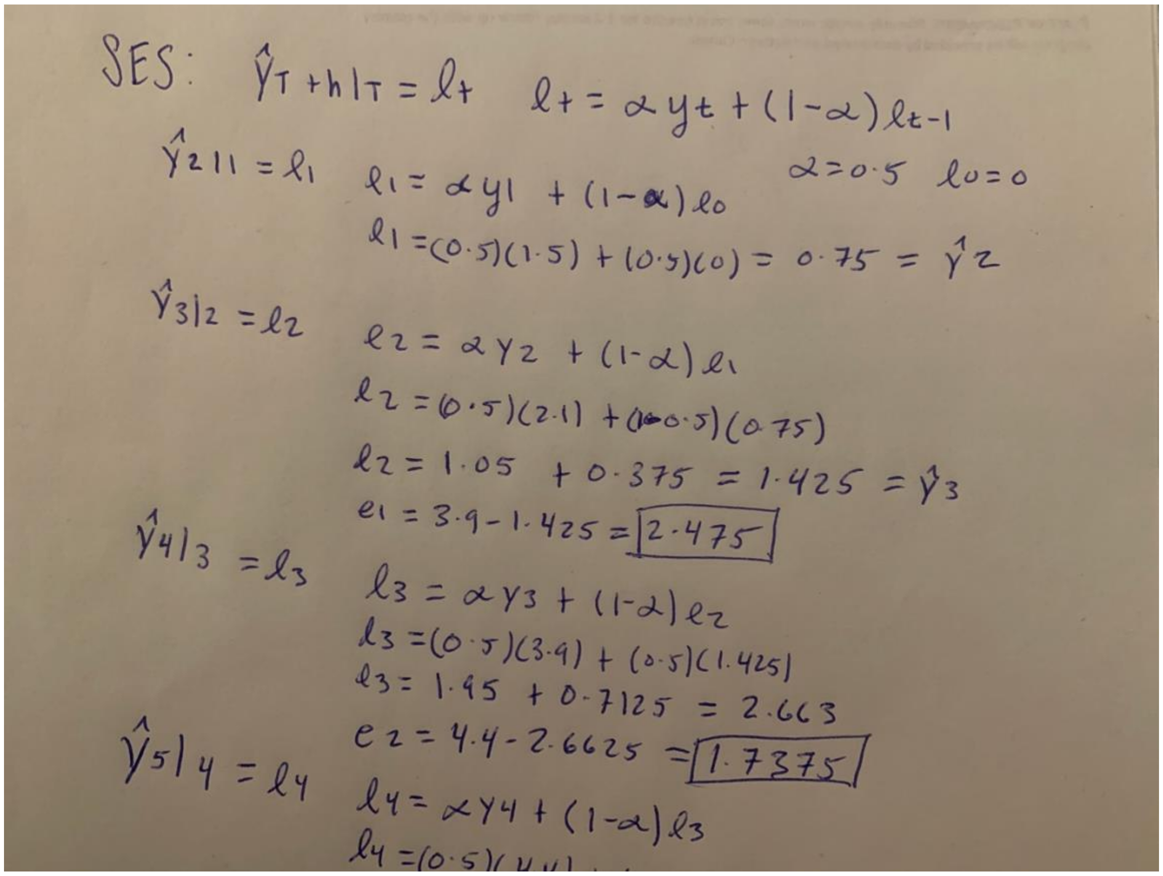
$\hat{y}_4 = \boxed{5.1}$ $e_2 = 4.4 - 5.1 = \boxed{-0.6}$

$\hat{y}_5 = y_4 + h\frac{(y_4 - y_1)}{4-1} = 4.4 + \frac{(4.4 - 1.5)}{3} = \boxed{5.37}$

$e_3 = 5.2 - 5.37 = \boxed{-0.17}$

$$MSE = \frac{(1.2)^2 + (-0.6)^2 + (-0.17)^2}{3}$$

$\boxed{MSE = 0.61}$



SES: $\hat{y}_{T+h|T} = l_t$ $\quad l_t = \alpha y_t + (1-\alpha) l_{t-1}$

$\hat{y}_{2|1} = l_1$ $\quad l_1 = \alpha y_1 + (1-\alpha) l_0$ $\quad \alpha = 0.5$ $\quad l_0 = 0$

$l_1 = (0.5)(1.5) + (0.5)(0) = 0.75 = \hat{y}_2$

$\hat{y}_{3|2} = l_2$ $\quad l_2 = \alpha y_2 + (1-\alpha) l_1$

$l_2 = (0.5)(2.1) + (0.5)(0.75)$

$l_2 = 1.05 + 0.375 = 1.425 = \hat{y}_3$

$e_1 = 3.9 - 1.425 = \boxed{2.475}$

$\hat{y}_{4|3} = l_3$ $\quad l_3 = \alpha y_3 + (1-\alpha) l_2$

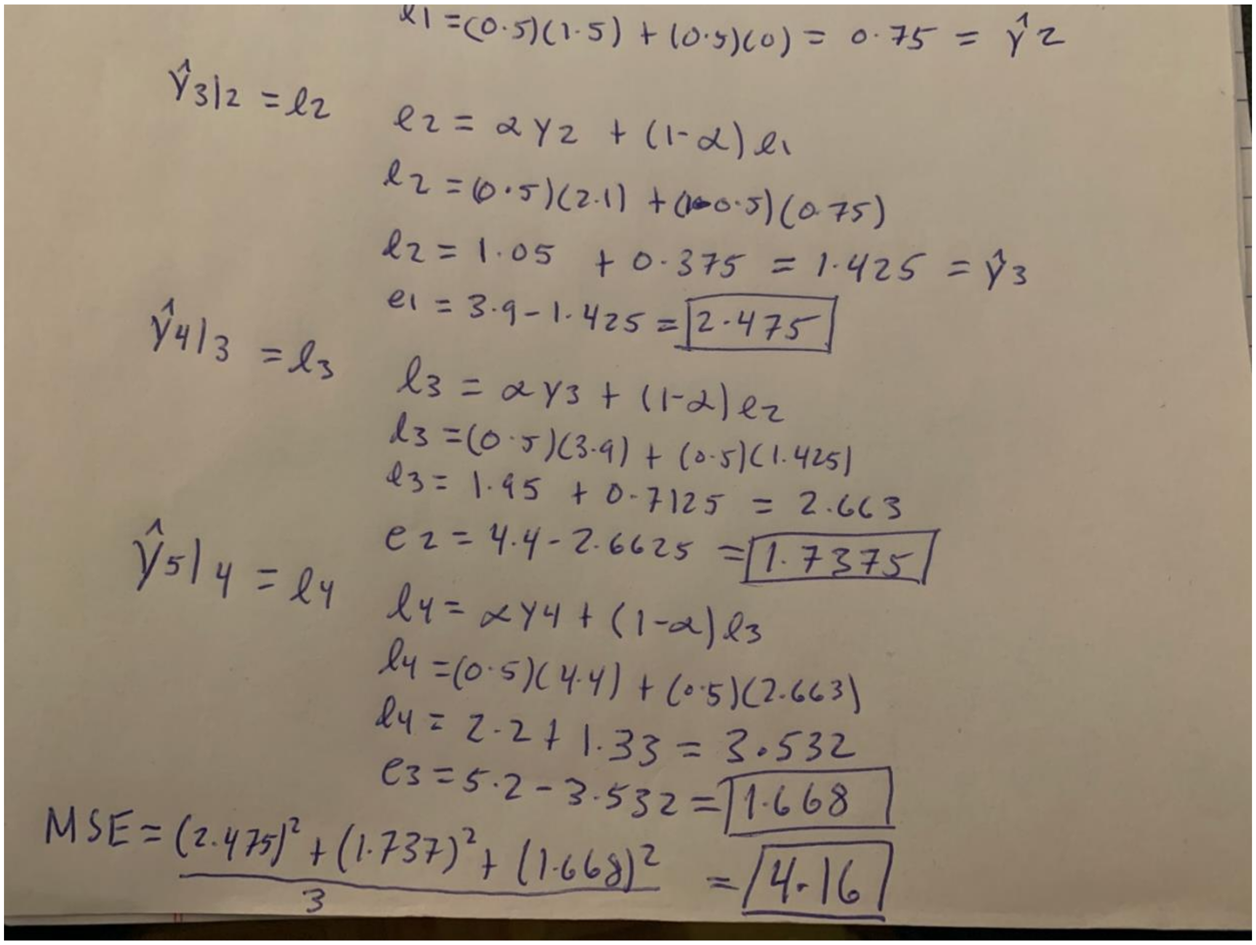
$l_3 = (0.5)(3.9) + (0.5)(1.425)$

$l_3 = 1.95 + 0.7125 = 2.663$

$e_2 = 4.4 - 2.6625 = \boxed{1.7375}$

$\hat{y}_{5|4} = l_4$ $\quad l_4 = \alpha y_4 + (1-\alpha) l_3$

$l_4 = (0.5)(4.4)$



$\ell_1 = (0.5)(1.5) + (0.5)(0) = 0.75 = \hat{y}_2$

$\hat{y}_{3|2} = \ell_2$

$\ell_2 = \alpha y_2 + (1-\alpha)\ell_1$

$\ell_2 = (0.5)(2.1) + (1-0.5)(0.75)$

$\ell_2 = 1.05 + 0.375 = 1.425 = \hat{y}_3$

$e_1 = 3.9 - 1.425 = \boxed{2.475}$

$\hat{y}_{4|3} = \ell_3$

$\ell_3 = \alpha y_3 + (1-\alpha)\ell_2$

$\ell_3 = (0.5)(3.9) + (0.5)(1.425)$

$\ell_3 = 1.95 + 0.7125 = 2.663$

$e_2 = 4.4 - 2.6625 = \boxed{1.7375}$

$\hat{y}_{5|4} = \ell_4$

$\ell_4 = \alpha y_4 + (1-\alpha)\ell_3$

$\ell_4 = (0.5)(4.4) + (0.5)(2.663)$

$\ell_4 = 2.2 + 1.33 = 3.532$

$e_3 = 5.2 - 3.532 = \boxed{1.668}$

$$MSE = \frac{(2.475)^2 + (1.737)^2 + (1.668)^2}{3} = \boxed{4.16}$$

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
import os
from statsmodels.tsa.seasonal import seasonal_decompose
from matplotlib.patches import Patch
from matplotlib.lines import Line2D

def auto_corr(data,lags):
    # convert input data into a numpy array
    data = np.array(data)
    # acf will store the autocorreltion coefficent at each lag interval
    # the first datapoint is always 1.0 since anything correlated with itsself is = 1
    acf = [1.0]
    # calculate the mean for the entire dataset
    y_bar = data.mean()
    print("The mean of this dataset is: ",y_bar)
    # subtract the mean from each observation
    yy_bar = data - y_bar
    # clacualte the total variance for the data set
    total_variance = sum(np.square(yy_bar))
    print("The total variance for this dataset is: ", total_variance)
    # perform a forloop over the dataset with the desired number of lags
    # range is 1,lags b/c the first iteration calcualtes T1
    for i in range(1,lags):
        # first nparray is removing the last element each iteration
        yy_bar_bottom = yy_bar[:-i]
        # second nparray removes the first element each interation
        yy_bar_top = yy_bar[i:]
        # take the sum of of the product of each nparray each iteration
        yy = sum(yy_bar_top * yy_bar_bottom)
        # divide the sum by total variance and append to resulting acf list
        acf.append(yy/total_variance)
    return acf

def acf_plot(y):
    #y = y.tolist()
    y_rev = y[::-1]
    y_rev.extend(y[1:])
    print(len(y_rev))
    return y_rev


def average(y, start, end):
    # data is a list with only the first two values
    data = y[:start]
    data1 = data
    print("True values: ", data)
    # Single iteration to grab the average of current values in data1
    for i in range(0, 1):
        pred = sum(data1) / len(data1)
        # append the sum of data1 back to data1
        data1.append(pred)
        print("True values with Predicted Value: ", data1)
    # calculate error
    print("True Value: ", y[start])
    print("Predicted Value: ", data1[-1])
    forcast_error = y[start] - data1[-1]
    print("Forcast Error: ", forcast_error)
    # convert list into numpy array for plotting
    dt_plt = np.array(data1)
    # create a numpy array for plotting the time steps at 1
    x = np.array(list(range(1, end)))
    # set conditions so that the predicted value is red
    col = np.where(x <= start, 'k', np.where(dt_plt > start, 'r', 'r'))
    # Plot the true values and predicted value
    plt.scatter(x, dt_plt, c=col, s=60, linewidth=0)
    plt.xlabel("Time-Step", fontsize=15)
    plt.ylabel('Value', fontsize=15)
    plt.title("Average Method Estimation y{}".format(start + 1), fontsize=18)

    legend_elements = [Line2D([0], [0], marker='o', color='w', markerfacecolor='k',
                              label='True Values', markersize=12),
                       Line2D([0], [0], marker='o', color='w', label='Prediction',
                              markerfacecolor='r', markersize=12)]
    plt.legend(handles=legend_elements, loc='lower right')
    plt.show()
    return forcast_error


# Naive Method

def niave(y, start, end):
    # data is a list with only the first two values
    data = y[:start]
    data1 = data
    print("True values: ", data)
    # Single iteration to grab the average of current values in data1
    for i in range(0, 1):
        pred = data1[-1]
        # append the sum of data1 back to data1
        data1.append(pred)
        print("True values with Predicted Value: ", data1)
    # calculate error
    print("True Value: ", y[start])
    print("Predicted Value: ", data1[-1])
    forcast_error = y[start] - data1[-1]
    print("Forcast Error: ", forcast_error)
    # convert list into numpy array for plotting
    dt_plt = np.array(data1)
    # create a numpy array for plotting the time steps at 1
    x = np.array(list(range(1, end)))
    # set conditions so that the predicted value is red
    col = np.where(x <= start, 'k', np.where(dt_plt > start, 'r', 'r'))
    # Plot the true values and predicted value
    plt.scatter(x, dt_plt, c=col, s=60, linewidth=0)
    plt.xlabel("Time-Step", fontsize=15)
    plt.ylabel('Value', fontsize=15)
    plt.title("Naive Method Estimation y{}".format(start + 1), fontsize=18)

    legend_elements = [Line2D([0], [0], marker='o', color='w', markerfacecolor='k',
                              label='True Values', markersize=12),
                       Line2D([0], [0], marker='o', color='w', label='Prediction',
                              markerfacecolor='r', markersize=12)]
    plt.legend(handles=legend_elements, loc='lower right')
    plt.show()
    return forcast_error


def drift(y, end, h):
    T = y[:end]
    y_t = T[-1]
    y0 = y[0]
    data1 = T
    Tt = len(T)
    Tt = Tt - 1
    yhat = y_t + (h * ((y_t - y0) / Tt))
    print("Y hat = ", yhat)
    data1.append(yhat)
    print("True Value:", y[h])
    error = y[h] - yhat
    print("Error = ", error)
    # convert list into numpy array for plotting
    dt_plt = np.array(data1)
    # create a numpy array for plotting the time steps at 1
    x = np.array(list(range(1, end + 2)))
    # set conditions so that the predicted value is red
    col = np.where(x <= end, 'k', np.where(dt_plt > end, 'r', 'r'))
    # Plot the true values and predicted value
    plt.scatter(x, dt_plt, c=col, s=60, linewidth=0)
    plt.xlabel("Time-Step", fontsize=15)
    plt.ylabel('Value', fontsize=15)
    plt.title("Drift Method Estimation y{}".format(h + 1), fontsize=18)

    legend_elements = [Line2D([0], [0], marker='o', color='w', markerfacecolor='k',
                              label='True Values', markersize=12),
                       Line2D([0], [0], marker='o', color='w', label='Prediction',
                              markerfacecolor='r', markersize=12)]
    plt.legend(handles=legend_elements, loc='lower right')
    plt.show()
    return error


# Make the function for SES
def ses(y, end, l0, alpha):
    # calculate y_hat
    li_1 = [l0]
    for i in range(0, len(y)):
        li = alpha * y[i] + ((1 - alpha) * li_1[i])
        li_1.append(li)
    y_hat = li_1[1:]
    # calculate errors
    error = []
```

```python
    for i in range(len(y)):
        ei = y[i] - y_hat[i]
        error.append(ei)
    print("Forcaste: ", y_hat[end])
    dt_plt = y[:end]
    y_hat_plt = y_hat[end]
    dt_plt.append(y_hat_plt)
    # convert lists into numpy arrays for plotting
    dt_plt = np.array(dt_plt)
    print("Plotting Points", dt_plt)
    # create a numpy array for plotting the time steps at 1
    x = np.array(list(range(1, end + 2)))
    # set conditions so that the predicted value is red
    col = np.where(x <= end, 'k', np.where(dt_plt > end, 'r', 'r'))
    # Plot the true values and predicted value
    plt.scatter(x, dt_plt, c=col, s=60, linewidth=0)
    plt.xlabel("Time-Step", fontsize=15)
    plt.ylabel('Value', fontsize=15)
    plt.title("SES Method Estimation y{}".format(end + 1), fontsize=18)

    legend_elements = [Line2D([0], [0], marker='o', color='w', markerfacecolor='k',
                              label='True Values', markersize=12),
                       Line2D([0], [0], marker='o', color='w', label='Prediction',
                              markerfacecolor='r', markersize=12)]
    plt.legend(handles=legend_elements, loc='lower right')
    plt.show()
    return y_hat, error

def error_stats(mt,x,n,h):
    mean_error = x.mean()
    print("The mean of forcast error is = ", mean_error)
    sse = sum((x) ** 2)
    print("The SSE of forcast error is =", sse)
    variance_error = sum((x - mean_error) ** 2)/n
    print("The variance of forcast error is = ",variance_error)
    mse = sse/n
    print("MSE is of the forcast error is = ", mse)
    acf = auto_corr(x,h)
    print("ACF of the forcast errors is = ", acf)
    Q = -1
    for i in acf:
        Q += i**2
    Q = n*Q
    print("Q value estimate is = ", Q)
    method = mt
    return method,mean_error,sse,variance_error,mse,acf,Q

# Test data
y = [1.5,2.1,3.9,4.4,5.2]

# Average Method implementation
avg_e1 = average(y,2,4)
avg_e2 = average(y,3,5)
avg_e3 = average(y,4,6)
avg_errors = np.array([avg_e1,avg_e2,avg_e3])
avg_es = error_stats("Average",avg_errors,3,3)

# Naive
n_e1 = niave(y,2,4)
n_e2 = niave(y,3,5)
n_e3 = niave(y,4,6)
niave_errors = np.array([n_e1,n_e2,n_e3])
nv_es = error_stats("Niave",niave_errors,3,3)

# Drift Method
d_e1 = drift(y,2,1)
d_e2 = drift(y,3,1)
d_e3 = drift(y,4,1)
drift_errors = np.array([d_e1,d_e2,d_e3])
dft_es = error_stats("Drift",drift_errors,3,3)

# SES
ses(y,2,0,0.5)
ses(y,3,0,0.5)
```

```
error = ses(y,4,0,0.5)

# Only need to look at the last three errors
error = error[1]
error = error[2:]
error = np.array(error)
ses_es = error_stats("SES",error,3,3)

# Plot ACF of errors
avg_eafc = avg_es[5]
avg_acf_plt = acf_plot(avg_eafc)

x = np.array(list(range(-2,3)))
figure = plt.stem(x,avg_acf_plt,use_line_collection=True)
#plt.figure(figsize=(20,20))
plt.xlabel('Lag', fontsize=15)
plt.ylabel('AC Coefficent', fontsize=15)
plt.title('Average Method Forecast Error ACF Plot',fontsize=18)
plt.show()

nv_eafc = nv_es[5]
nv_acf_plt = acf_plot(nv_eafc)

x = np.array(list(range(-2,3)))
figure = plt.stem(x,nv_acf_plt,use_line_collection=True)
#plt.figure(figsize=(20,20))
plt.xlabel('Lag', fontsize=15)
plt.ylabel('AC Coefficent', fontsize=15)
plt.title('Niave Method Forecast Error ACF Plot',fontsize=18)
plt.show()

dft_eafc = dft_es[5]
dft_acf_plt = acf_plot(dft_eafc)

x = np.array(list(range(-2,3)))
figure = plt.stem(x,dft_acf_plt,use_line_collection=True)
#plt.figure(figsize=(20,20))
```

```
plt.xlabel('Lag', fontsize=15)

plt.ylabel('AC Coefficent', fontsize=15)

plt.title('Drift Method Forecast Error ACF Plot',fontsize=18)

plt.show()


ses_eafc = ses_es[5]

ses_acf_plt = acf_plot(ses_eafc)


x = np.array(list(range(-2,3)))

figure = plt.stem(x,ses_acf_plt,use_line_collection=True)

#plt.figure(figsize=(20,20))

plt.xlabel('Lag', fontsize=15)

plt.ylabel('AC Coefficent', fontsize=15)

plt.title('SES Method Forecast Error ACF Plot',fontsize=18)

plt.show()


# Create table to evaluate models

fm_eval = np.array([avg_es,nv_es,dft_es,

                    ses_es])


df_f = pd.DataFrame(data=fm_eval)


df_f = df_f.rename(columns={0: "Mehtod", 1: "Mean Error",2: "SSE", 3: "Variance", 4: "MSE",5:"ACF", 6: "Q"})

print(df_f)
```

References: