Time series decomposition and Prediction

**TIME SERIES DECOMPOSITION & PREDICTION IN PYTHON**

[](https://www.pythonforfinance.net/wp-content/uploads/2019/07/cropped-Screenshot_6-1.png)

When considering most financial asset price time series you would be forgiven for concluding that, at various time frames (some longer, some shorter) many, many of the data sets we try to analyse can appear completely random. At least random enough that any hope of easily forecasting future value and paths is going to be a tough ask at the every least!

Although there most probably isn’t an easy fix just around the corner, there are perhaps some tools available to us that could assist.

Time series decomposition is a technique that allows us to deconstruct a time series into its individual “component parts”. These parts consist of up to 4 different components:

1) Trend component   
  
2) Seasonal component   
  
3) Cyclical component   
  
4) Noise component   
  
n.b we sometimes lump together the Cyclical and Noise components and call it the “Remainder” or some such.

A particular time series doesn’t need to contain all 4 components, it may be lacking a seasonal or trend component.

So what use is this knowledge to us? Well predicting a time series can often be really rather difficult but if we can decompose the series into components and treat each one separately we can sometimes improve overall prediction. Each component has specific properties and behaviour and with this approach we are able to use methods that are more suited to each particular component.

It is worth noting that decomposition is mainly used to help analyse and understand historical time series, but it can also prove useful when attempting a forward looking analysis.

So as mentioned previously, a series is made up of, and can be decomposed into, a number of components: Trend

Seasonality

Cyclical

and Noise

these last two are sometimes combined and labelled as the Remainder  .

It is helpful to think of the components as combining either additively or multiplicatively.

They differ as follows:

1) An additive model suggests that the components are added together as follows:

whereas a multiplicative model suggests that the components are multiplied together as follows:

2) An additive model is linear where changes over time are consistently made by the same amount, whereas a multiplicative model is nonlinear, such as quadratic or exponential and changes increase or decrease over time.  
  
3) A linear trend is a straight line, whereas a nonlinear trend is a curved line.  
  
4) A linear seasonality has the same frequency (width of cycles) and amplitude (height of cycles), whereas a non-linear seasonality has an increasing or decreasing frequency and/or amplitude over time.

Just as an point to note, the above is a general rule and in practice you may or may not be able to cleanly or perfectly break down your specific time series as an additive or multiplicative model. There may exists combinations of the two models, trends can change direction and cycles of a non-repeating nature can be mixed in with the repeating seasonality components. Basically real-world data can be messy and it doesn’t always play by the rules! They are still helpful though in defining a simple framework from which to analyse our data.

There are various methods of decomposition, with the “base” method known as “classical decomposition”, a relatively simple procedure that also forms the starting point for most other methods of time series decomposition. This method has been around for years, originating way back in the 1920s – so this isn’t exactly anything “new”.

There are two forms of classical decomposition, one for each of our two models described above (additive an multiplicative).

For additive decomposition the process (assuming a seasonal period of  ) is carried out as follows:

1) Compute the “trend-cycle” component  using a  if  is an even number, or using an  if  is an odd number.

2) Calculate the detrended series:

3) The seasonal component for each season if then calculated by simply averaging the detrended values for that season. So if we are dealing with monthly data, the seasonal component for December, for example, would just be the average of all the detrended December values in the data. Adjustments to these monthly values are made to ensure that they all sum to zero, and then they are strung together and the sequence is replicated for each year of data: the output of this is

4) The final step is to calculate the noise component by subtracting the estimated “seasonal” and “trend-cycle” components:

Classical multiplicative decomposition is similar, except that the subtractions are replaced by divisions.

While classical decomposition is still widely used, it is not recommended, as it suffers from multiple problems, such as having no trend-cycle estimates for the first few and last few observations (e.g. if  there would be no trend-cycle estimate for the first ans last 6 time periods, and therefore no remainder component either). It also suffers from a few other weakness regarding the assumptions made that the seasonal component repeats from year to year, it tends to over-smooth rapid rises and falls in the data when calculating the trend-cycle and the method is also not robust to “unusual” or “outlier” values in the data. Anyway, there are now several much better methods available such as X11 Decomposition, SEATS Decomposition or STL Decomposition.

Let’s go with STL Decomposition; the “STL is an acronym for “Seasonal and Trend decomposition using Loess”. Loess is a method for estimating non-linear relationships.

STL has several advantages over the classical, SEATS and X11 decomposition methods:

1) STL will handle any kind of seasonality, not only monthly and quarterly (unlike SEATS and X11).

2) The smoothness of the trend-cycle can be controlled by the user

3) The seasonal component is allowed to change over time, again allowing the user to control this rate.

4) It is generally more robust to “unusual”, or outlier data points. It can be carried out in a way that the estimates of the trend-cycle and seasonal components will not be affected, however the remainder component will still be affected.

Just to be fair, it does also have some disadvantages, for example it is only able to be used with additive decomposition and it doesn’t automatically account for, and handle trading day/calendar variation.

OK let’s break up the wall of text and actually implement a quick decomposition using the statsmodels package.

I will be using the EURUSD price time series in this instance and the relevant data can be downloaded below:  
PLEASE install this in a virtual environment.

With that word of caution, if you wish to avail yourself of the STL class you can follow these steps:  
  
1) Create and activate a virtual environment  
  
2) install Cython in that virtual environment  
  
pip install cython  
  
3) install statsmodels development version  
  
pip install git+https://github.com/statsmodels/statsmodels.git

Once that is successfully completed, we go straight into the main code by importing the necessary modules. We then read in the EURUSD data using Pandas, extract the relevant column and run it through both a Hodrick-Prescott Filter and an SYL decomposition separately. Just to note, the EURUSD file provided is “tab separated” rather than the more common “comma separated” hence we set sep="\t".

import pandas as pd

import statsmodels.api as sm

from statsmodels.tsa.seasonal import STL

from sklearn.metrics import mean\_squared\_error

from statsmodels.tsa.ar\_model import AR

import matplotlib.pyplot as plt

import seaborn as sns

# include below line if you are using Jupyter Notebook

%matplotlib inline

# Set figure width to 12 and height to 9

plt.rcParams['figure.figsize'] = [12, 9]

df = pd.read\_csv('EURUSD.csv',sep='\t', index\_col='Date')

df.index = pd.to\_datetime(df.index)

df.sort\_index(inplace=True)

df = df.resample('W').last()

series = df['Price']

The HP filter is a technique commonly used with macro-economic series that have a trend (long-term movements), business cycle and irregular parts (short-term fluctuations). It constructs the trend component by solving an optimisation problem. It aims to form the smoothest trend estimate that minimises the squared distances to the original series. In other words, it has to find equilibrium between the smoothness of the trend and its closeness to the original.  
  
If you want to find out more about the details, you can do that [here](https://en.wikipedia.org/wiki/Hodrick%E2%80%93Prescott_filter).

First the HP filter decomposition:

cycle, trend = sm.tsa.filters.hpfilter(series, 50)

fig, ax = plt.subplots(3,1)

ax[0].plot(series)

ax[0].set\_title('Price')

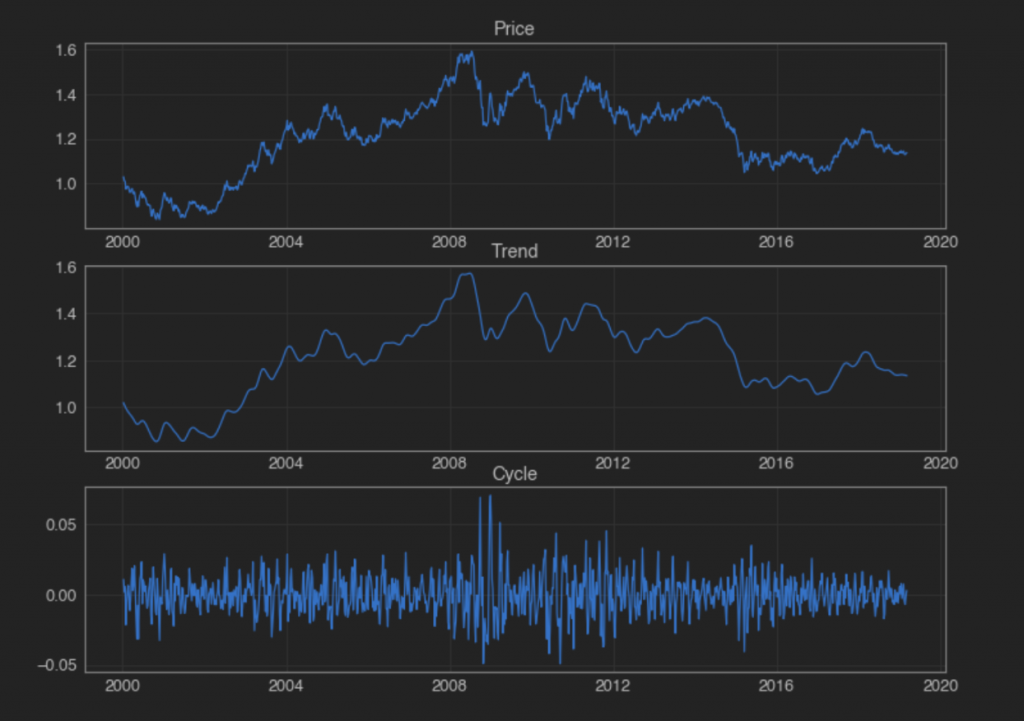
ax[1].plot(trend)

ax[1].set\_title('Trend')

ax[2].plot(cycle)

ax[2].set\_title('Cycle')

plt.show()

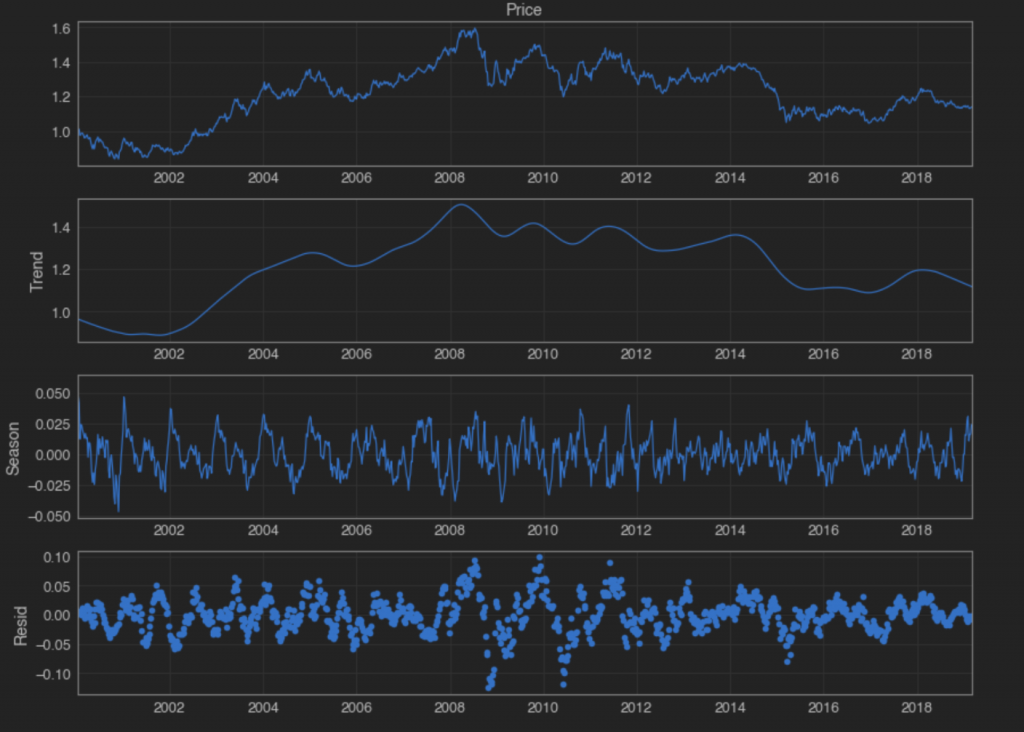


And next the SYL decomposition :

result = STL(series).fit()

chart = result.plot()

plt.show()



Right, all looks good so far – we have our decomposed series, from top down: Raw Weekly Price, Trend, Season and Residual (a.k.a Remainder). So what now? Well what if we want to try to forecast the EURUSD price movements? We could start by applying a simple “persistence model”, this is the simplest kind of model which works by assigning the last observed value as the prediction for the following value, i.e. it just “persists” the last price forward. As you can see its not exactly a very sophisticated model, however its a start and it provides a baseline of performance for the problem that we can use for comparison with an auto regression model. We will score our model performance using the Root Mean Squared Error (RMSE).

The code below begins by creating a series of “1-period forward” predictions, just shifting the last price forward one week and comparing that value with the actual price that was seen at that time. It then uses the scikit-learn “mean\_squared\_error” function to calculate the MSE, which we then simply take the square root of to produce the RMSE. We receive a value of 0.013.

You will notice that the RMSE is only calculated for the period corresponding to a “testing dataset”, which in this case is the last 30% of the data. This allows us to compare performance with more sophisticated models that require “training” over a “training” dataset, before they can be used for prediction purposes, the performance of which we judge when carried out over a “testing” dataset.

This ensures that we only judge a models performance when it is used to predict parts of a time series that it didn’t have available at the time of optimising its parameters.

The mean squared error, or MSE, is calculated as the average of the squared forecast error values. The error values are in squared units of the predicted values and a mean squared error of zero indicates perfect prediction skills, or “no error” in effect.

The MSE can be transformed back into the original units of the predictions by taking the square root as mentioned above with an RMSE of zero again indicating perfect prediction skills.

Finally, we plot just the last few data points, allowing us to see more clearly how the predicted values relate to the actual values observed.

predictions = series.shift(1).dropna()

test\_score = np.sqrt(mean\_squared\_error(series[int(len(series) \* 0.7)+1:], predictions.iloc[int(len(series) \* 0.7):]))

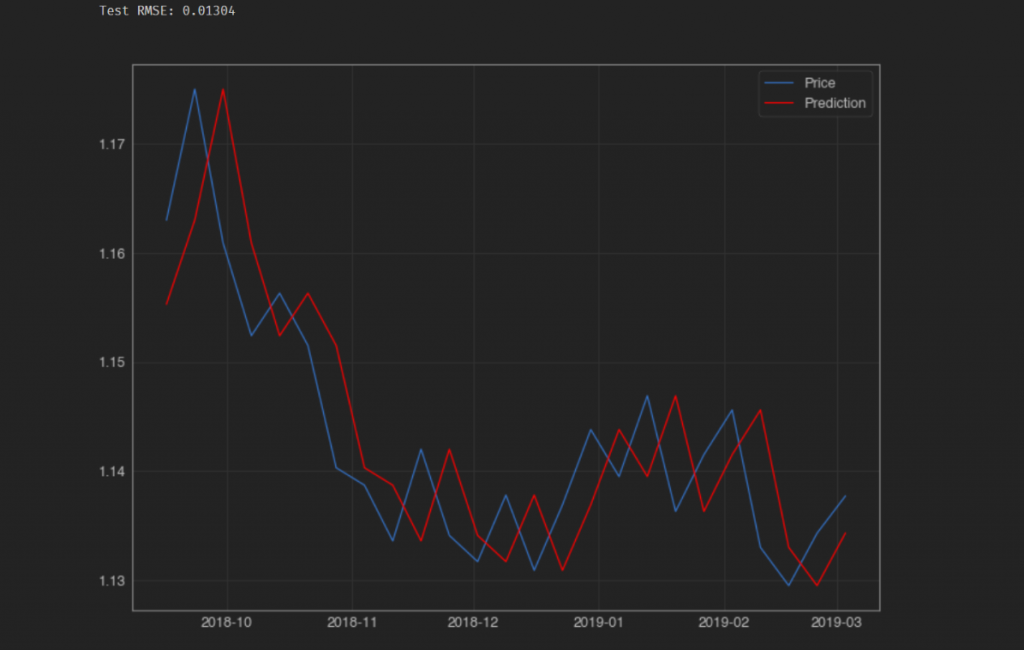
print('Test RMSE: %.5f' % test\_score)

plt.plot(series.iloc[-25:], label='Price')

plt.plot(predictions[-25:], color='red', label='Prediction')

plt.legend()

plt.show()



It might also be useful to create a scatter plot of the predicted vs the actual weekly percentage change and see if there is any noticeable relationship. We do this below, followed by calculating the Mean Absolute Error (MAE) of the predictions (i.e. by how much, on average did our prediction differ from the observed value)

fig, ax = plt.subplots()

ax = sns.regplot(series.iloc[-int(len(series) \* 0.3):].pct\_change(),

predictions.iloc[-int(len(series) \* 0.3):].pct\_change(), )

plt.xlabel('Observations')

plt.ylabel('Predictions')

plt.title('EURUSD Observed vs Predicted Values')

ax.grid(True, which='both')

ax.axhline(y=0, color='#888888')

ax.axvline(x=0, color='#888888')

sns.despine(ax=ax, offset=0)

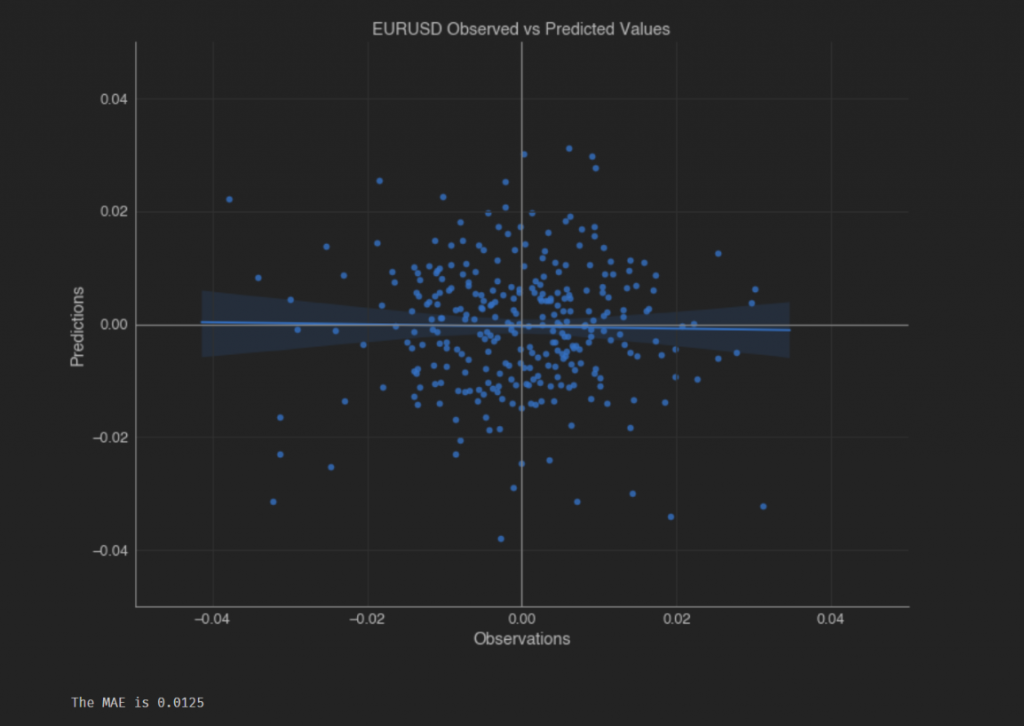
plt.xlim(-0.05, 0.05)

plt.ylim(-0.05, 0.05)

plt.show()

mae = round(abs(series.iloc[-int(len(series) \* 0.3):].pct\_change() - predictions.iloc[-int(len(series) \* 0.3):].pct\_change()).mean(),4)

print(f'The MAE is {mae}')



Looking at the above plot it seems as though there is no discernable relationship between our predictions and our observed values when considering the weekly percentage change in the EURUSD price. That’s not great at all!

If we quickly calculate the “hit rate” of how often we are able to correctly predict the direction of next week’s move in the EURUSD, we can see that it is almost exactly 50% (in fact its a little worse than that even) – no better than completely randomly guessing!

price\_pred = pd.concat([series.iloc[-int(len(series) \* 0.3):].pct\_change(), predictions.iloc[-int(len(series) \* 0.3):].pct\_change()], axis=1)

price\_pred.dropna(inplace=True)

price\_pred.columns = ['Price', 'preds']

price\_pred['hit'] = np.where(np.sign(price\_pred['Price']) == np.sign(price\_pred['preds']), 1, 0)

print(f"Hit rate: {round((price\_pred['hit'].sum() / price\_pred['hit'].count()) \* 100,2)}%")

48.49%

Now that we have our “baseline” model score, we have something to compare subsequent models to. Let’s try our predictions using an autoregressive model, which is basically a linear regression model that uses lagged values of variables as subsequent input variables.

There is a decision to make as to what lag to use for our input variable – the simplest approach for now is to use the AR class found in statsmodels which automatically selects an appropriate lag using various statistical tests and proceeds to train a linear regression model.

The code below takes our EURUSD time series and splits it into a “training” and “testing” dataset and creates an empty list which we are going to use to store our predictions. This is because this time we are going to make our weekly predictions one by one, in a “walk-forward” manner.

We are going to iterate through a “for-loop” with the number of loops dictated by the length of our test data – each time through the loop we are going to create our AR model and feed it the training data set to train and fit it on. Once fit, we generate a 1 period-forward prediction (which takes the form of a single value), we append that value to our predictions list to store it. Importantly, we also extract the value from the test data whose index corresponds to the current value of our for-loop iterator and append that value to the end of our training data.

The next run through our loops, we train and fit the model again but this time there is a new observation in the training data that we feed it, i.e. the value we just appended on the last for-loop.

This means the training data will grow as each loop sees a new observation added to the training data. This just mimics the real-life passage through time where we have an additional day’s worth of data available to us vs the day before. Nothing more magical than that!! It just ensures also that we are not exposing our model to “look-forward” bias and training it using data that wouldn’t have been available at that time.

Like before, the MSE is calculated and the last 25 data points are plotted – the observed prices vs the predictions made by the model.

historic = series.iloc[:int(len(series) \* 0.7)].to\_list()

test = series.iloc[int(len(series) \* 0.7):]

predictions = []

for i in range(len(test)):

model = AR(historic)

model\_fit = model.fit()

pred = model\_fit.predict(start=len(historic), end=len(historic), dynamic=False)

predictions.append(pred[0])

historic.append(test[i])

predictions = pd.Series(predictions, index=test.index)

test\_score = np.sqrt(mean\_squared\_error(test, predictions))

print('Test MSE: %.5f' % test\_score)

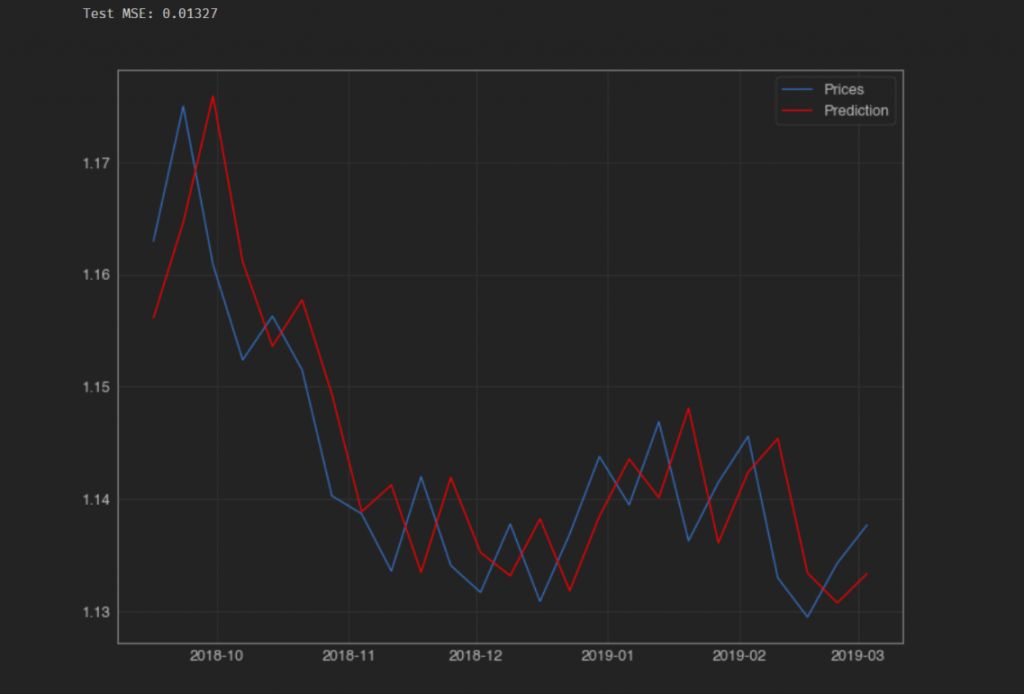
# plot results

plt.plot(test.iloc[-25:], label='Prices')

plt.plot(predictions.iloc[-25:], color='red', label='Prediction')

plt.legend()

plt.show()



Again we get an MSE very close to 0.013, this time its actually ever so slightly higher at 0.1327. The reason that the two models produce such a similar MSE is down to the fact that neither model seems to be any good at making predictions. If we were to look at the average weekly price move in the EURUSD over the period covered by our test data, I have as sneaking suspicion it would turn out to be somewhere in the region of…you guessed it – 0.013.

That’s the same as saying our model predictions are no better, on average than saying the price isn’t going to move at all, which is obviously a useless prediction.

Plotting the same scatter plot as before, but this time for the latest model confirms our suspicions – again there is no discernible relationship between our predictions of the weekly percentage price moves, and those we actually observe, and again our MAE is pretty much the same as the first model.

fig, ax = plt.subplots()

ax = sns.regplot(series.iloc[-int(len(series) \* 0.3):].pct\_change(),

predictions.iloc[-int(len(series) \* 0.3):].pct\_change(), )

plt.xlabel('Observations')

plt.ylabel('Predictions')

plt.title('EURUSD Observed vs Predicted Values')

ax.grid(True, which='both')

ax.axhline(y=0, color='#888888')

ax.axvline(x=0, color='#888888')

sns.despine(ax=ax, offset=0)

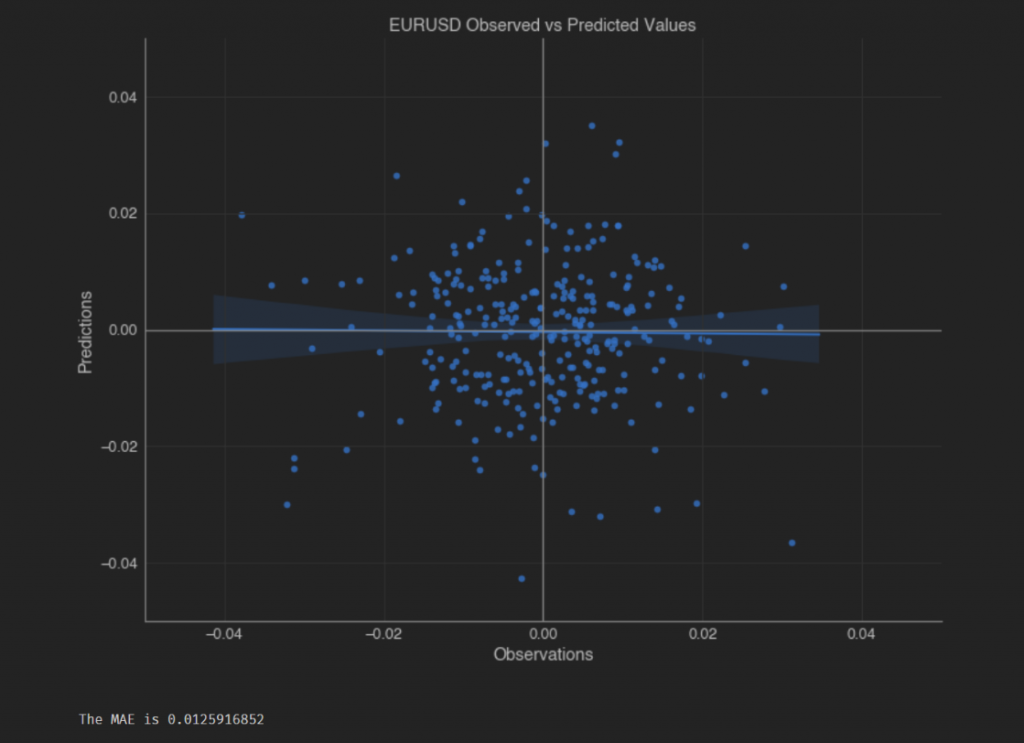
plt.xlim(-0.05, 0.05)

plt.ylim(-0.05, 0.05)

plt.show()

mae = round(abs(test.pct\_change() - predictions.pct\_change()).mean(),10)

print(f'The MAE is {mae}')



Looking at the above plot it seems as though again there is no discernible relationship between our predictions and our observed values when considering the weekly percentage change in the EURUSD price.

If we quickly calculate the “hit rate” of how often we are able to correctly predict the direction of next week’s move in the EURUSD, again we see a figure that’s slightly worse than 50%.

price\_pred = pd.concat([test.pct\_change(), predictions.pct\_change()], axis=1)

price\_pred.dropna(inplace=True)

price\_pred.columns = ['Price', 'preds']

price\_pred['hit'] = np.where(np.sign(price\_pred['Price']) == np.sign(price\_pred['preds']), 1, 0)

print(f"Hit rate: {round((price\_pred['hit'].sum() / price\_pred['hit'].count()) \* 100,2)}%")

Hit rate: 48.16%

So what can we do? How can we use the decomposition methods to help improve our predictive abilities? As mentioned before, each component has varying characteristics that suggest they should be dealt with individually in a more specialised manner. If we can improve our predictions by breaking a time series into its component, use our models to predict the components individually then in theory all we have to do is recombine the predictions back into a full time series (i.e. just add them all back together) and we should end up with a more accurate overall prediction.

So lets first decompose the EURUSD series using the Hodrick-Prescott Filter and store each component.  
We do this below and then plot the components as a quick reminder so as to what they look like.

cycle, trend = sm.tsa.filters.hpfilter(series, 50)

fig, ax = plt.subplots(3,1)

ax[0].plot(series)

ax[0].set\_title('Price')

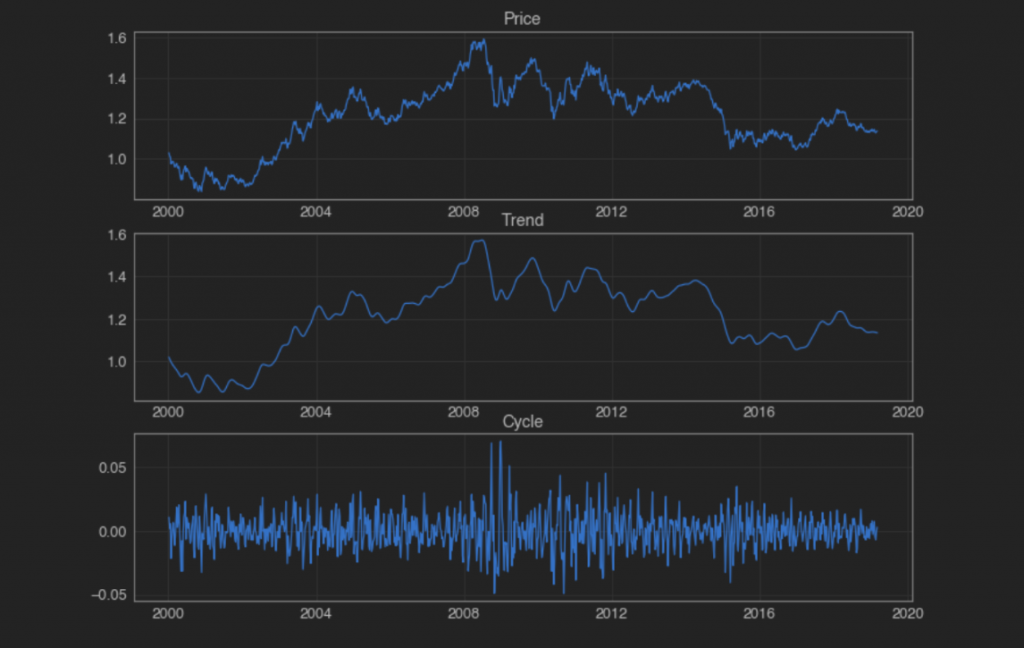
ax[1].plot(trend)

ax[1].set\_title('Trend')

ax[2].plot(cycle)

ax[2].set\_title('Cycle')

plt.show()



Now we have to write the code to run our AR model in the same fashion as before, but this time with the decomposed series as inputs.

component\_dict = {'cycle': cycle, 'trend': trend}

prediction\_results = []

for component in ['trend', 'cycle']:

historic = component\_dict[component].iloc[:int(len(series) \* 0.7)].to\_list()

test = component\_dict[component].iloc[int(len(series) \* 0.7):]

predictions = []

for i in range(len(test)):

model = AR(historic)

model\_fit = model.fit()

pred = model\_fit.predict(start=len(historic), end=len(historic), dynamic=False)

predictions.append(pred[0])

historic.append(test[i])

predictions = pd.Series(predictions, index=test.index, name=component)

prediction\_results.append(predictions)

test\_score = np.sqrt(mean\_squared\_error(test, predictions))

print(f'Test for {component} MSE: {test\_score}')

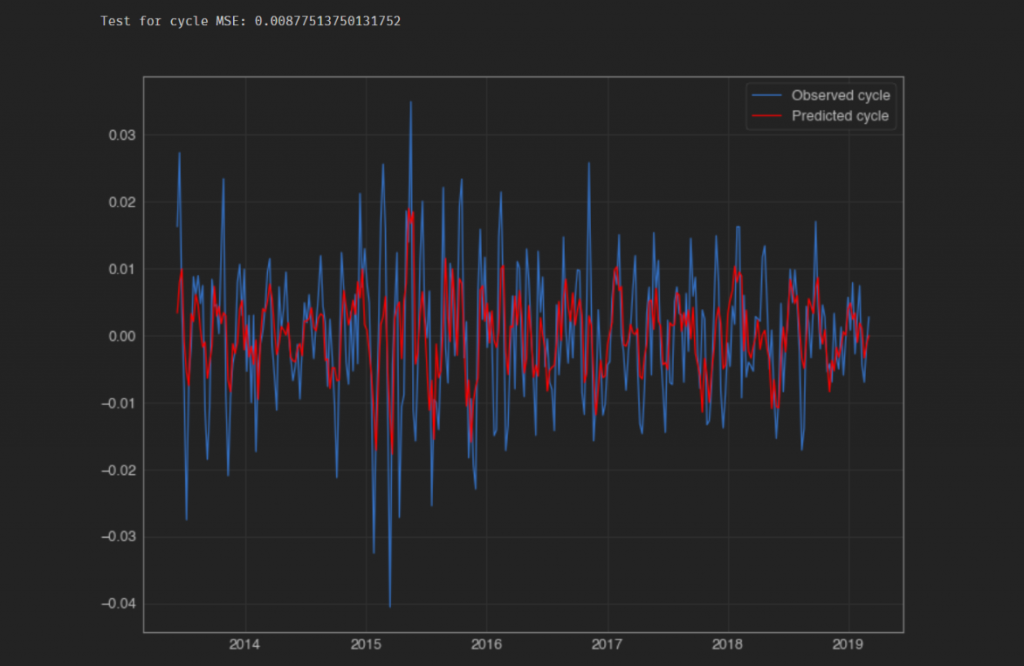
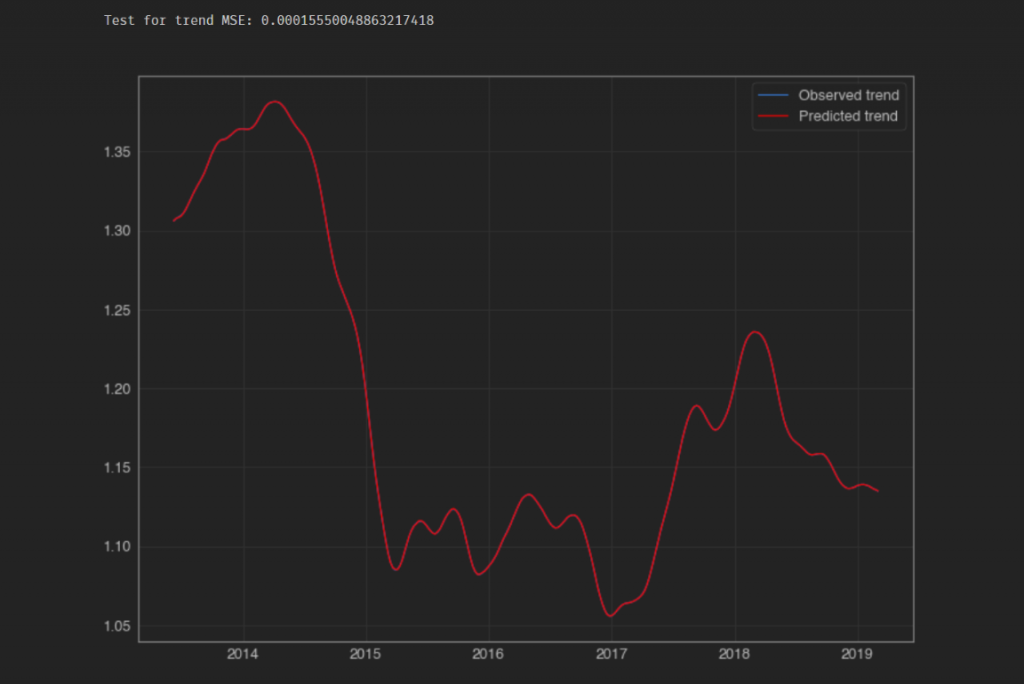
# plot results

plt.plot(test.iloc[:], label='Observed '+component)

plt.plot(predictions.iloc[:], color='red', label='Predicted '+component)

plt.legend()

plt.show()



It looks like only one line has been plotted for the trend series, but it only just looks that way as the predicted values are so close to the actual decomposed values that one line is hidden by the other.

Next we “recompose” our data in an additive manner and calculate the RMSE when comparing the resulting combined prediction performance against the observed values.

recomposed\_preds = pd.concat(prediction\_results,axis=1).sum(axis=1)

recomposed\_preds.name = 'recomposed\_preds'

plt.plot(series.iloc[int(len(series) \* 0.7):], label='Observed')

plt.plot(recomposed\_preds, color='red', label='Predicted')

plt.legend()

plt.show()

test\_score = np.sqrt(mean\_squared\_error(series.iloc[int(len(series) \* 0.7):], recomposed\_preds))

print(f'RMSE: {test\_score}')



Ok so it looks like our RMSE has fallen, now sitting around 0.0087 as opposed to the two previous attempts which were producing values closer to 0.013. That’s a fall of about 33%. Does that mean there is something to be said for this decompose/recompose approach?

Let’s plot our scatter chart and calculate the new MAE for this model.

fig, ax = plt.subplots()

ax = sns.regplot(series.iloc[-int(len(series) \* 0.3):].pct\_change(),

recomposed\_preds.iloc[-int(len(series) \* 0.3):].pct\_change())

plt.xlabel('Observations')

plt.ylabel('Predictions')

plt.title('EURUSD Observed vs Predicted Values')

ax.grid(True, which='both')

ax.axhline(y=0, color='#888888')

ax.axvline(x=0, color='#888888')

sns.despine(ax=ax, offset=0)

plt.xlim(-0.05, 0.05)

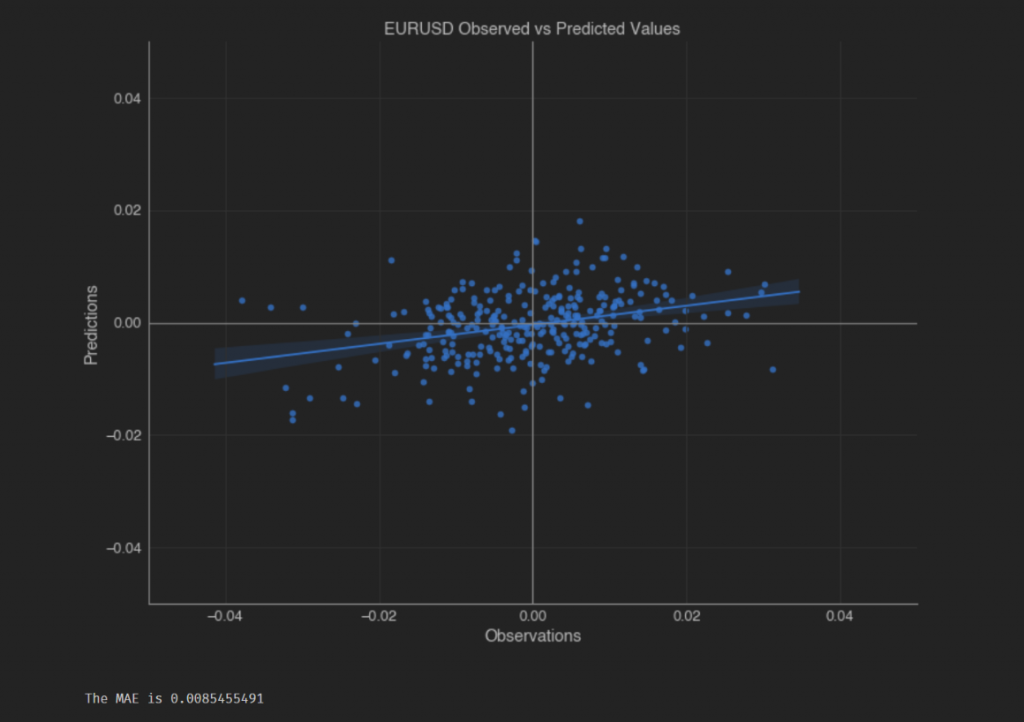
plt.ylim(-0.05, 0.05)

plt.show()

mae = round(abs(series.iloc[-int(len(series) \* 0.3):].pct\_change() -

recomposed\_preds.iloc[-int(len(series) \* 0.3):].pct\_change()).mean(),10)

print(f'The MAE is {mae}')



The MAE has fallen also, from a previous figure of around 0.013 to 0.0085 – again a difference of around 33%. Not only that but the distribution of data points on the scatter plot has taken in a slightly different shape – the regression line shows more of a positive relationship between the observations and our predictions.  
  
As shown below, we are now able to predict the direction of the next week’s price move 60% of the time, rather than a random 50% as we saw with the last two attempts.

price\_pred = pd.concat([series.iloc[-int(len(series) \* 0.3):].pct\_change(),recomposed\_preds.iloc[-int(len(series) \* 0.3):].pct\_change()], axis=1)

price\_pred.dropna(inplace=True)

price\_pred['hit'] = np.where(np.sign(price\_pred['Price']) == np.sign(price\_pred['recomposed\_preds']), 1, 0)

print(f"Hit rate: {round((price\_pred['hit'].sum() / price\_pred['hit'].count()) \* 100,2)}%")

Hit rate: 60.2%

Although the RMSE, MAE and Hit Rate have all significantly improved when we use the decompose/recompose approach, even with not a single attempt to optimise our model parameters, we would do well not to get too excited too quickly.

I chose to demonstrate the use of the HP Filter decomposition method first on purpose. One needs to be careful when applying new methods – if we don’t understand fully the way things work “behind the scenes” we can end up getting caught out quite badly.

There is a slight issue with the HP filter method. The decomposition algorithm makes use of observations that come both before and after the current estimate. This injects an element of “look-forward” bias into our analysis which is one of the most common biases that infect peoples analysis and trading/predictive models. In this case, once you know exactly how the HP Filter is calculated it seems quite obvious, but if you never took the time to familiarise yourself with the calculation method that could easily fly under the radar.

As a side note, algorithmic/systematic trading model backtesting is an area that is ABSOLUTELY ripe for and rife with various biases, with look-forward bias being near, if not top of the list.

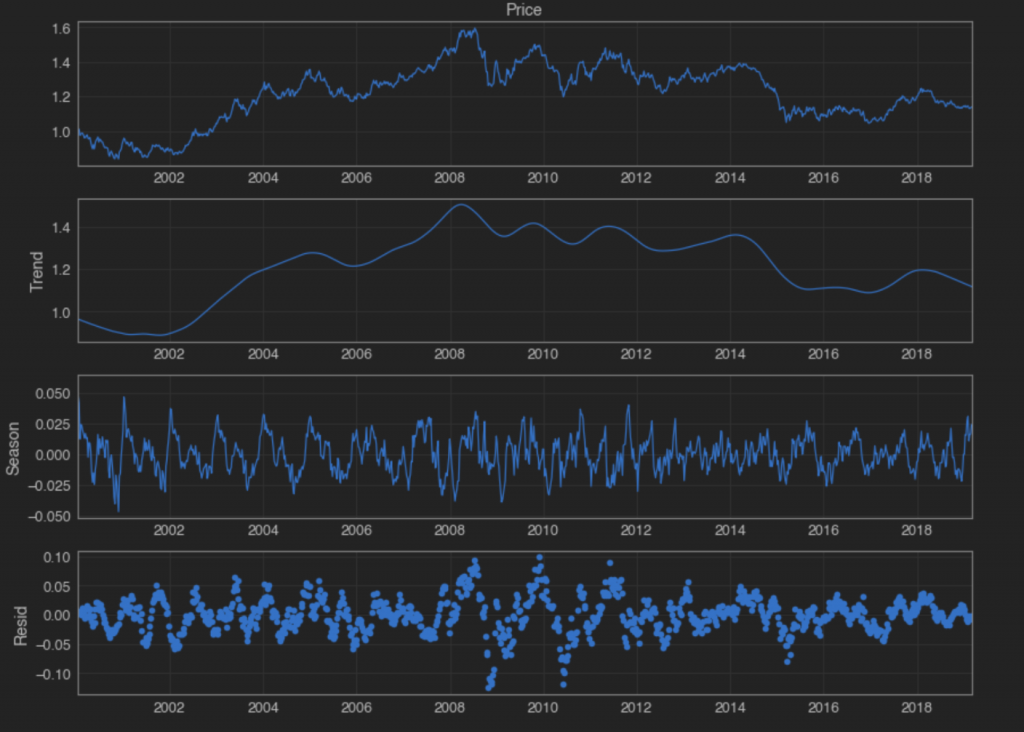
So basically the above means that the HP Filter produces decomposed series that contain information from the “future”, making our predictions more accurate in general and causing our recomposed predictions to be misleadingly accurate also.

So what happens when we revert back to using a decomposition method that doesn’t “cheat” and avail itself of “future information” when decomposing a time series.  
  
Well let’s return now to the STL Decomposition method.Below is a decomposition and visual of the results.

result = STL(series).fit()

result.plot()

plt.show()



The individual decomposed series can be accessed from the result object as follows, by using either “trend”, “seasonal” or “resid”. As an example we print out the first 5 rows of the seasonal series below.

result.seasonal.head()

Date

2000-01-09 0.060011

2000-01-16 0.047314

2000-01-23 0.041816

2000-01-30 0.012704

2000-02-06 0.024431

Freq: W-SUN, Name: season, dtype: float64

Now we have to write the code to run our AR model in the same fashion as before.

component\_dict = {'seasonal': result.seasonal, 'trend': result.trend, 'residual': result.resid}

prediction\_results = []

for component in ['seasonal', 'trend', 'residual']:

historic = component\_dict[component].iloc[:int(len(series) \* 0.7)].to\_list()

test = component\_dict[component].iloc[int(len(series) \* 0.7):]

predictions = []

for i in range(len(test)):

model = AR(historic)

model\_fit = model.fit()

pred = model\_fit.predict(start=len(historic), end=len(historic), dynamic=False)

predictions.append(pred[0])

historic.append(test[i])

predictions = pd.Series(predictions, index=test.index, name=component)

prediction\_results.append(predictions)

test\_score = np.sqrt(mean\_squared\_error(test, predictions))

print(f'Test for {component} MSE: {test\_score}')

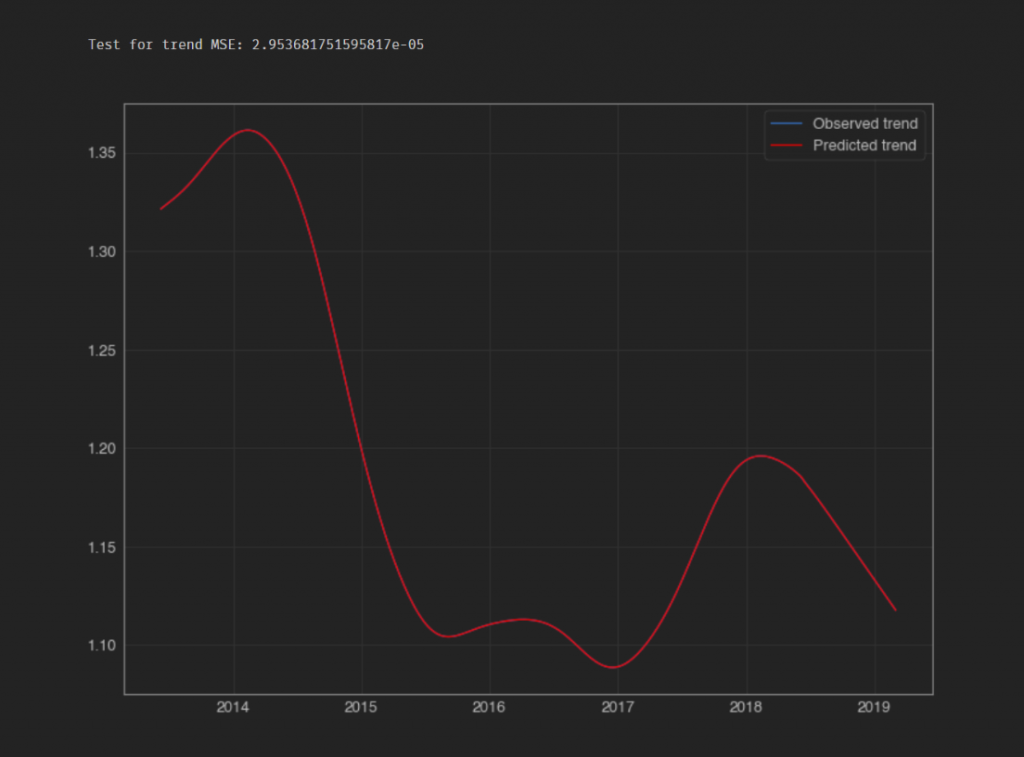
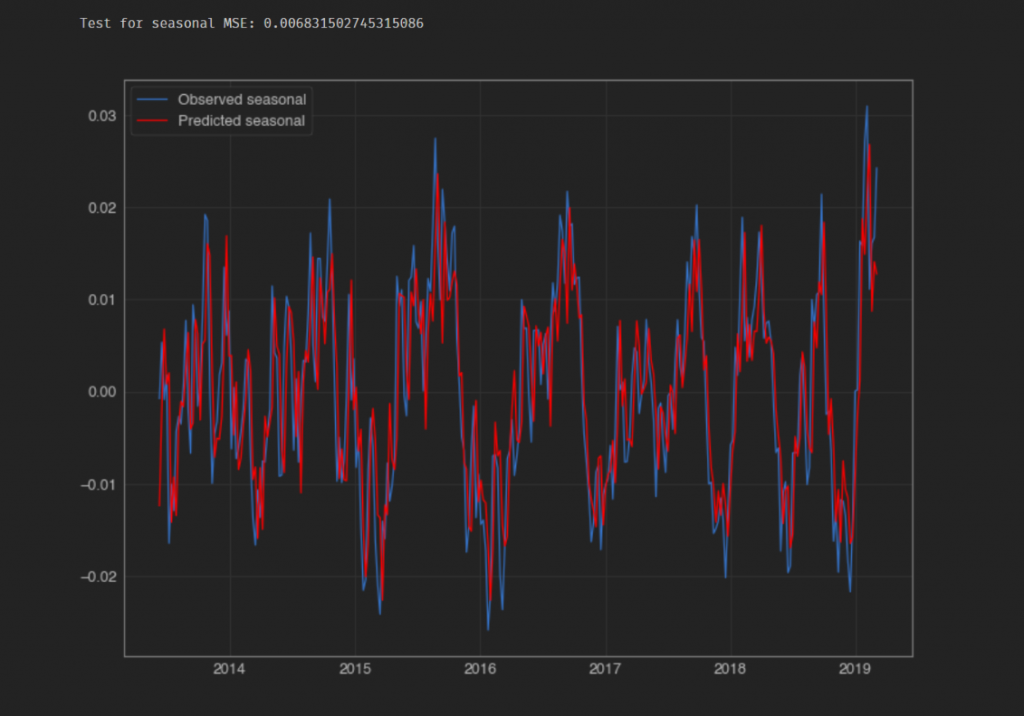
# plot results

plt.plot(test.iloc[:], label='Observed '+component)

plt.plot(predictions.iloc[:], color='red', label='Predicted '+component)

plt.legend()

plt.show()



Next we “recompose” our data in an additive manner and calculate the RMSE when comparing the resulting combined prediction performance against the observed values as we did for the HP Filter model.

recomposed\_preds = pd.concat(prediction\_results,axis=1).sum(axis=1)

plt.plot(series.iloc[int(len(series) \* 0.7):], label='Observed')

plt.plot(recomposed\_preds, color='red', label='Predicted')

plt.legend()

plt.show()

test\_score = np.sqrt(mean\_squared\_error(series.iloc[int(len(series) \* 0.7):], recomposed\_preds))

print(f'RMSE: {test\_score}')



o

That RMSE score looks like its moved in the wrong direction ( 0.0126)…its jumped right back to the kind of values we were seeing for our first two models, the “Persistence” model and the AR model that we were using with undecomposed data – i.e. using the raw price series.

That doesn’t bode well for us. How does the MAE and scatter plot of weekly percentage price changes look (observed vs predicted values).

fig, ax = plt.subplots()

ax = sns.regplot(series.iloc[-int(len(series) \* 0.3):].pct\_change(),

recomposed\_preds.iloc[-int(len(series) \* 0.3):].pct\_change(), )

plt.xlabel('Observations')

plt.ylabel('Predictions')

plt.title('EURUSD Observed vs Predicted Values')

ax.grid(True, which='both')

ax.axhline(y=0, color='#888888')

ax.axvline(x=0, color='#888888')

sns.despine(ax=ax, offset=0)

plt.xlim(-0.05, 0.05)

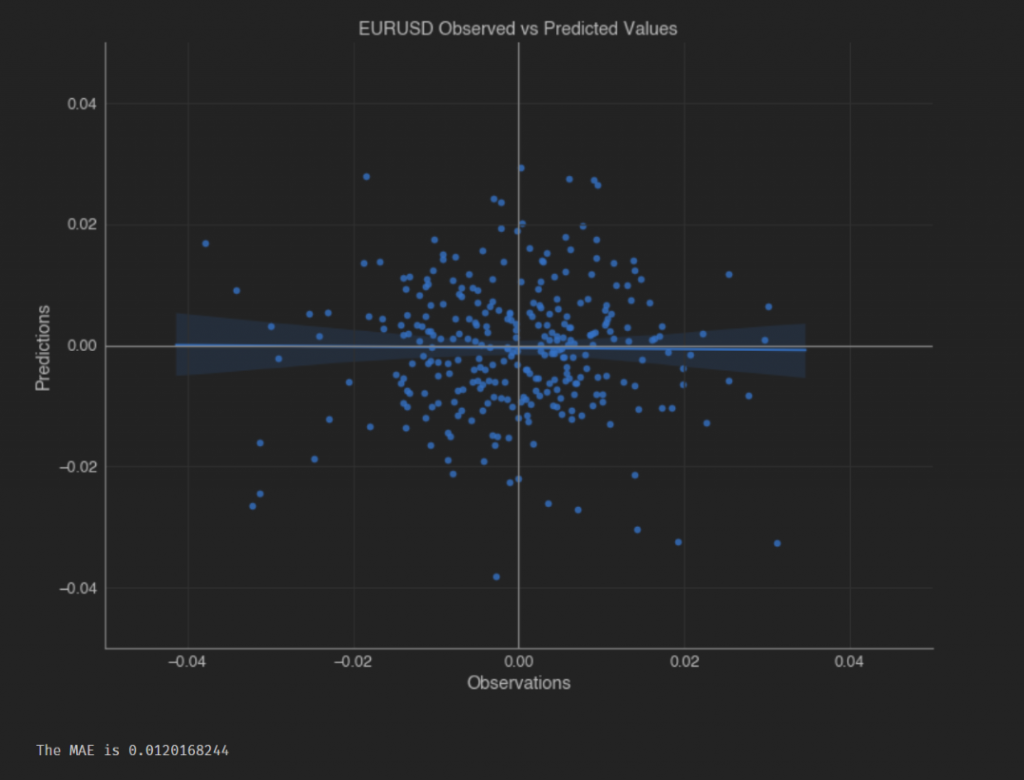
plt.ylim(-0.05, 0.05)

plt.show()

mae = round(abs(series.iloc[-int(len(series) \* 0.3):].pct\_change() -

recomposed\_preds.iloc[-int(len(series) \* 0.3):].pct\_change()).mean(),10)

print(f'The MAE is {mae}')



Our MAE is also back around the previously seen values, and just to confirm what should now be clear to us, the Hit Ratio is back at the 50% level. That translates as “your model is no better at predicting the forthcoming weeks price change direction than is the toss of a fair coin”.

price\_pred = pd.concat([series.iloc[-int(len(series) \* 0.3):].pct\_change(),

recomposed\_preds.iloc[-int(len(series) \* 0.3):].pct\_change()], axis=1)

price\_pred.dropna(inplace=True)

price\_pred.columns = ['Price', 'preds']

price\_pred['hit'] = np.where(np.sign(price\_pred['Price']) == np.sign(price\_pred['preds']), 1, 0)

print(f"Hit rate: {round((price\_pred['hit'].sum() / price\_pred['hit'].count()) \* 100,2)}%")

Hit rate: 51.17%

We have seen that the outcome of the decomposition method used hugely affects the final predictions. The AR model parameters were not estimated using future observations, nor was the STL Decomposition so its not a huge shock that in this instance it didn’t provide a magic bullet to solve our prediction problem.

What’s worth noting, as mentioned before, is that one need to take great care when using certain methods and and tools such as decomposition. Even though the AR model that was used to predict the values of the decomposed series output by the HP Filter doesn’t suffer from look-forward bias itself, the way the series are separated during the decomposition itself uses future movements unknown at the time of estimation and therefore taints the model’s predictive output.