Rec09

April 10, 2022

1 Recitation 9: Forecasting

This assignment will be handed in differently from past assignments.

To get credit for this assignment as a recitation, you only need to do a part of it. Aim to finish parts 1-3 and if doing this takes you past the end of the in-person recitation then talk to your TA and you can hand in the recitation to get full credit as long as you put in a reasonable effort.

The complete assignment will be due as part of the next homework, and full points will require completion of the entire assignmen with correct answers.

In this assignment, you will be making predictions based on time series data from a sales of a collection of consumer products from the company that sells Stovetop Stuffing. These products exhibit strong seasonal sales patterns. You'll be exposed to many different ways of making predictions and you'll get to see some of the drawbacks of each. You should be largely self-sufficient with python and pandas for data science by this point.

[ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

[ ]:
```
from datetime import datetime
```

1.1 Part 1: Data

1.1. Read in the SalesAndForecastsByLine.csv into a dataframe. Then print some of the data to see what it looks like.

[ ]:
```
# YOUR CODE HERE
```

1.2 Create a new column in the table called "Date".

Fill this column with the dates as strings in the form month-year. For example May 2020 would be 5-2020. Print the data again to see if the column was created properly.

[ ]:
```
# YOUR CODE HERE
```

1.3 Now create a column in your dataframe that contains each date in the Date field in your dataframe (which is a string) converted to the datetime type. `datetime` is the datatype that Pandas uses to store dates and it is very helpful when plotting quantities versus date or time.

One way to do this is with `iterrows()`, as in the following example code:

```
for index, row in df.iterrows():
    df.loc[index,'t'] = # calculate value from row['Date']
```

There are other ways as well. Once you are done, output some rows from the dataframe again to make sure that you are computing your datetime field correctly.

Hint: Use the strptime from the datetime module. Documentation can be found here: https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior. the strptime function will parse string arguments into datetime objects, based on the format that you provide. If you have a string stored in a variable called `date` in a format where 5-2020 represents May 2020, then calling `datetime.strptime(date,'%m-%Y')` will return a datetime object representing May 2020.

[ ]:
```
# YOUR CODE HERE
```

1.4 Now create 5 different dataframes, one for each of the product line numbers, by filtering on the `LineNumber` field.

Then plot `TotalProductSales` vs. date (using your new datetime field) for each of the 5 product lines. Put them all on one plot. This can be accomplished by calling plt.plot for each of the 5 dataframes all in 1 cell. matplotlib should figure out on its own to use different colors for each.

[ ]:
```
# YOUR CODE HERE
```

1.5 What times of year are particularly busy for these products? You may find that you need to expand the size of your graph to see the dates. You can do this with the code `plt.figure(figsize=(10,5))` run in the same cell and before any `plt.plot` commands. You can also zoom in to particular regions of the figure using `plt.xlim`. For example, `plt.xlim('2000-04','2000-12')` zooms in to April to December 2000.

Answer Here

1.2 Part 2: Linear Fit

We use statsmodels.api to fit the data using ordinary least squares (OLS).

[ ]:
```
import statsmodels.api as sm
```

2.1. Fit OLS on the data for product line 102. Use the sales data as your outcome y. Include an intercept. We will use an integer that increments by 1 each month as our single predictor.

To quickly get this integer, first use the code `df2_linear = df2.reset_index()`, where this assumes that `df2` is the name of the dataframe containing data for product line 102. This creates a dataframe that contains an additional column, `index`, that contains a sequence of integers. This relies on the fact that our dataframe had data for product line 102 in increasing time order.

Then call `summary()` to see the results of your fit.

[ ]:
```
# YOUR CODE HERE
```

2.2: Plot the predictions from linear regression and your data (TotalProductSales) for product line 102.

```
[ ]: # YOUR CODE HERE
```

2.3 What are some drawbacks of this method?

Answer here.

2.4 Calculate the sum of squared errors resulting from this method. To do this, calculate a series containing the difference between the actual and predicted values. Then calculate the square of each component of this series. Then sum the values. You should be able to do this in a single line of code.

Given a series `x`, the series `x.pow(2)` contains the square of each element of `x`.

```
[ ]: # YOUR CODE HERE
```

1.3 Part 3: Exponential Smoothing

We now use simple exponential smoothing to predict, now using product line 102. This assumes that the data has no consistent trend and consists only of a mean that slowly varies up and down and has noise added to it.

```
[ ]: from statsmodels.tsa.api import SimpleExpSmoothing
```

3.1: To use simple exponential smoothing, we need to create a Series containing TotalProductSales for product line 102 that has our datetime object as the index. The easiest way to do this is to create a new dataframe using set_index() and then extract the `TotalProductSales` field from this dataframe. Name the resulting Series `sales2`.

```
[ ]: # YOUR CODE HERE
```

Then, the following line of code creates a simple exponential fit for the data.

The code will say that "No frequency information was provided, so inferred frequency MS will be used." This is ok – it just means that we didn't tell SimpleExpSmoothing about a period over which we might expect to see repeating patterns.

```
[ ]: fit1 = SimpleExpSmoothing(sales2).fit()
```

Then, the following code gets a forecast for the next 4 upcoming periods.

```
[ ]: fore1 = fit1.forecast(4)
```

3.2: Plot the values fitted to our original timeseries, the original sales data for line 102, and the forecasted values. The fitted values are in `fit1.fittedvalues`. The forecasted values can be plotted using `fore1.plot()`.

```
[ ]: # YOUR CODE HERE
```

3.3 Calculate the sum of squared errors between this method's predictions and the actual value.

[ ]:
```
# YOUR CODE HERE
```

3.4: Based on the sum of squared errors, is this method better or worse than linear regression?

Answer here.

3.5: Where does this method do particularly badly? How does this relate to your answer to question 1.5?

Answer here.

1.4 Part 4: Holt Linear

Now we will fit this same data using the `Holt` method. We import the package `Holt` for this part and `ExponentialSmoothing` (also called the Holt-Winters method) for the next part.

The Holt method assumes that the data has a trend and that the slope of the trend is slowly varying over time. It also assumes that there is an additional change to the overall mean that is slowly shifting everything up and down, and that the observations have noise.

This method is able to make predictions that have the same form as simple exponential smoothing, and also the same form as linear regression, as two special cases.

[ ]:
```
## Holt Double Exponential Linear
from statsmodels.tsa.holtwinters import Holt, ExponentialSmoothing
```

4.1: Fit the Holt method on the data series created for the previous part and generate a forecast for the next 5 values. Above we used the code

```
fit1 = SimpleExpSmoothing(sales2).fit()
fore1 = fit1.forecast(4)
```

to fit simple exponential smoothing and forecast for 4 values. The code for the Holt method is similar, except we replace `SimpleExpSmoothing` by `Holt`, and we forecast for 5 values instead of 4.

[ ]:
```
# YOUR CODE HERE
```

4.2: Now plot the forecast, fitted values, and orignal sales data.

[ ]:
```
# YOUR CODE HERE
```

4.3 Calculate the sum of square errors between this method's predictions and the actual value.

[ ]:
```
# YOUR CODE HERE
```

4.4 How does the sum of squared errors for this method compare to simple exponential smoothing? To linear regression? Why do you think this is?

Answer here.

1.5 Part 5: Holt-Winters Method

5.1: Now we will use the Holt-Winters method, also called ExponentialSmoothing, on the same series of data. This allows for a trend (either additive or multiplicative), and seasonality (also, either additive or multiplicative), which we specify when we create the ExponentialSmoothing object. We also need to specify the number of periods to use for our seasonality.

We'll create this object using this code:

```
ExponentialSmoothing(sales2, trend = "add", seasonal = "add", seasonal_periods = ?)
```

where you need to fill in the value of seasonal_periods. This is the number of periods (each datapoint is 1 period) over which we see the strongest repeating period. Keep in mind that each datapoint is a month, and keep in mind what you learned in question 1.4 and 1.5 about patterns in the data.

Fit the Holt-Winters method and make a forecast for 12 periods.

[ ]:
```
# YOUR CODE HERE
```

5.2: Plot the fit, forecasts, and original sales data.

[ ]:
```
## YOUR CODE HERE
```

This is a good opportunity to changing the trend and seasonality type to see what happens. You can set this to `"mul"` for either `trend` or `seasonal` to use a multiplicative trend or seasonality.

If you chose seasonal_periods well, you should see that this method works really well — much better than the other methods.

5.3 Calculate the sum of squared errors between this method's predictions and the actual value. How does it compare to the sum of squared errors for the other methods?

[ ]:
```
# YOUR CODE HERE
```

5.4: What is it that this method does that allows it to do so well?

Answer here.