Required reading for this homework:

On\_Scaling.ipynb

rolling\_regression\_centering.xlsx

on\_regularization\_lasso\_ridge.ipynb

Start from CocacolaStart.ipynb:

Cocacola sales seem to follow a cycle of four quarters (using quarterly data).

Model the sales of CocaCola using the multiplicative model of seasonality, after taking logs.

by using pandas.read\_csv('CocaCola.csv', parse\_dates=['Date'], infer\_datetime\_format=True) function

read the data in "CoCaCola.csv" and save it into a pandas dataframe called df

Use pandas.to\_datetime(df['Date']) function

to convert the original Date column in df to a datetime data type

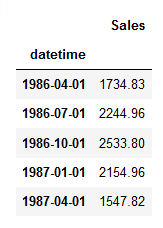
save the converted datetime as df["datetime"]

use df.set\_index('datetime', inplace=True) to set the datetime column as index of the df

get rid of the original Date column in df

by using df.drop(["Date"], axis=1, inplace=True)

print df.head(), it should look like this:



use the tsplot() function to plot the Sales column of df with lags=29

try to determine the number of quarters in a seasonal cycle

in the autocorrelation plot

using df.dropna(inplace=True) to get rid of any Nans

Using the detrendPrice() function

detrend the df and save it in a new df\_detrended dataframe

use the tsplot() function to plot the sales column of df\_detrended with lags=29

try to determine the number of quarters in a seasonal cycle

in the autocorrelation plot

Prepare the data for the multiplicative seasonal model

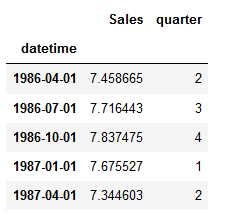
using np.log(df["Sales"]), transform the Sales data in the df into its natural log

save the result back in df["Sales"]

Using dft.index.quarter generate a column of quarters

save the resulting column into a new df["quarter"] column

print the df.head() to get:

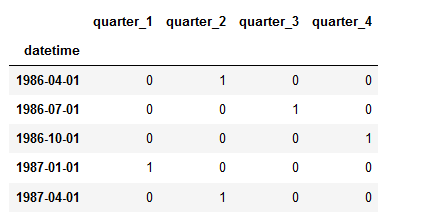


Using the pd.get\_dummies(df["quarter"], prefix="quarter") function

“one-hot-encode” the df["quarter"] column and

save the resulting dummy variables in a new df\_dummies dataframe

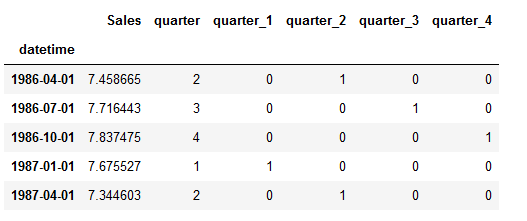
print the df\_dummies.head(), it should look like this:



using df.join(df\_dummies), join the dummy variables to the original data

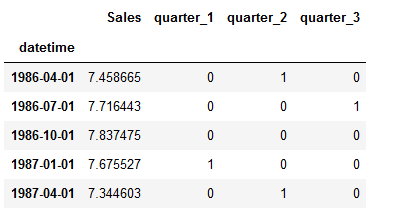
and save the result back in df

print the df.head(), it should look like:



using df.drop(["quarter","quarter\_4"], inplace=True, axis=1) drop these two columns

print the df.head(), it should look like:



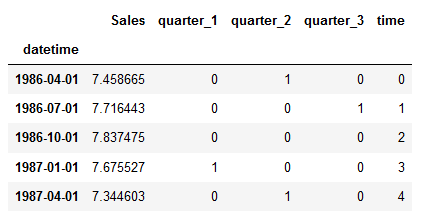
declare an empty list = []

using the list(range(0, dft.shape[0])) function and the fact that df.shape[0] contains the number of rows in the df

generate a list going from 0,1,2,....,60 (the number of rows in the df)

and save the result in df["time"] = list

print the df.head(), it should look like:



Save the df.["Sales"] column into the variable y

use df.drop(["Sales"], inplace=True, axis=1) to drop the Sales column

Save the df into the X variable

Now instatiate a linear regression, fit it to the train set and use it to predict

using the train\_test\_split() utility with random\_state=42 and shuffling=False, divide the data into 80% train and 20% test data

instantiate a linear regression model, save it in lr

fit the linear regression model using lr.fit() function, save the resulting fitted model into lr\_fitted (you can save it back into lr, but you might want to remember that you fitted lr already)

use lr\_fitted.predict() function to interpolate, giving it as input X\_train, saving the output as predictions\_train

use lr\_fitted.predict() function to forecast, giving it as input X\_test, saving the output as predictions\_test

evaluate the model

use the MAD\_mean\_ratio() function to calculate the MAD/MEAN ratio error of predictions\_train

print the error

use the MAD\_mean\_ratio() function to calculate the MAD/MEAN ratio error of predictions\_test

print th error

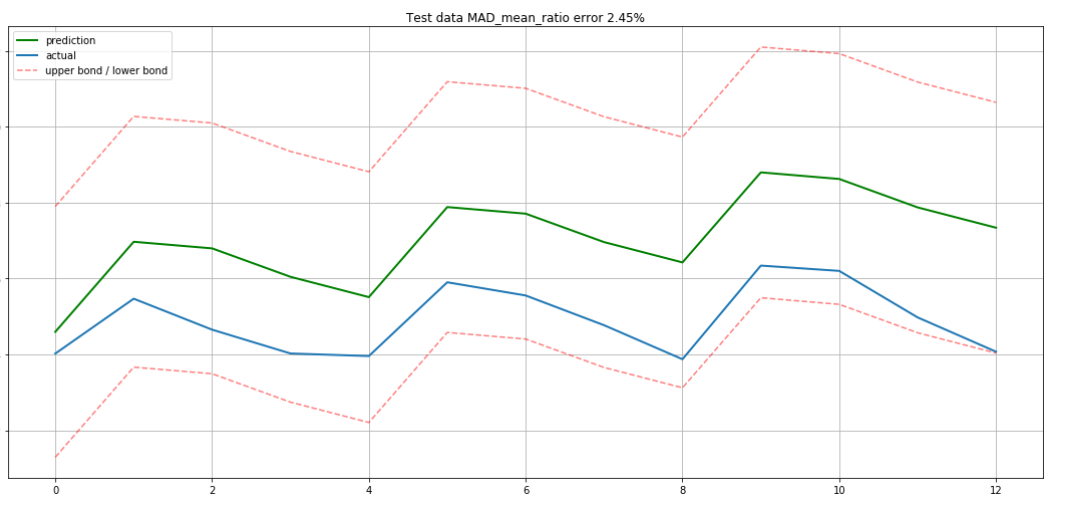
The errors should be near this (these are percentages, .75% and 2.51%):

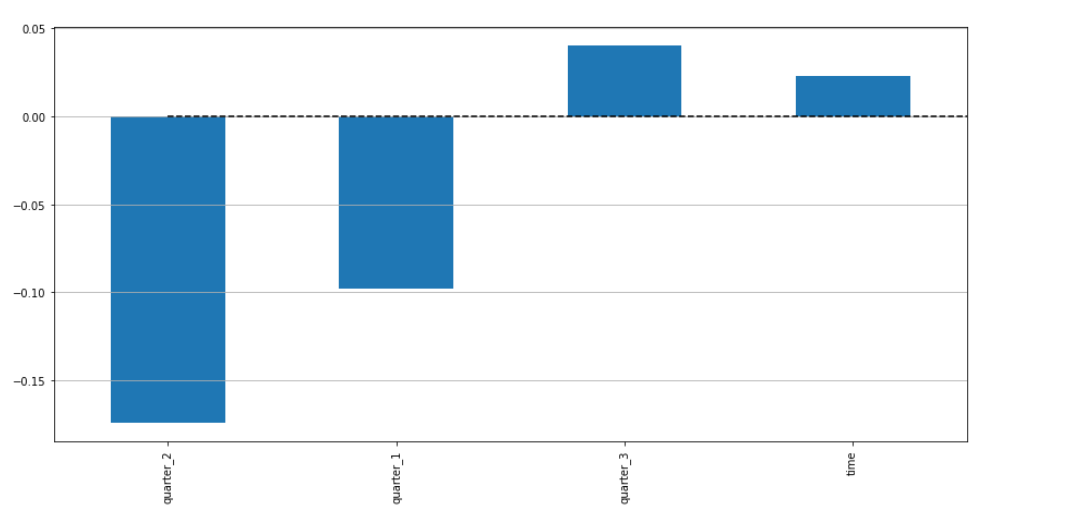


use the plotModelResults() function to plot the predictions and the true test values

use the plotCoefficients() function to plot the coefficients of the regression

The outputs should be near this:





Now let us try to bring down the error in the test predictions by introducing error (bias) into the train predictions, that is, by making the regression fit the train data not so well as before, so that it can predict the test data better. We do this by using a Ridge regression and looking for the regularization value of the alpha parameter. These are the steps:

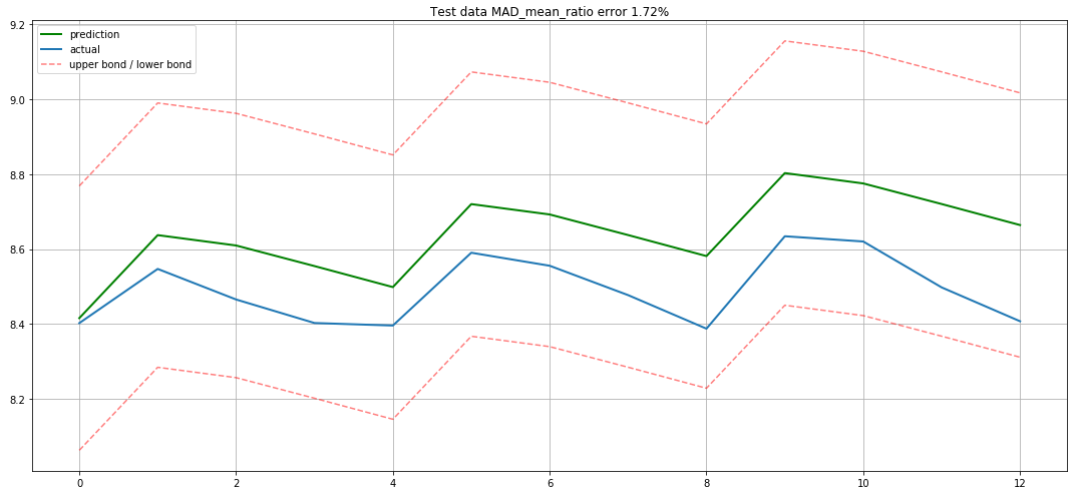
1. Load and visualize the unscaled data
2. Divide the data into test and train set: 80% train 20% test: train\_test\_split(X, y, test\_size=.2, shuffle=False, random\_state=42)
3. Set up a pipeline to scale the data appropriately during grid search: Pipeline([("scaler", StandardScaler()), ("ridge", Ridge())])
4. Separate out a percent of the train set as validation data: 20% (5 folds cv=5): TimeSeriesSplit(n\_splits=5)
5. Use the validation data to select the model with best parameters: GridSearchCV(pipe, param\_grid, cv=tscv, scoring="neg\_mean\_absolute\_error")
6. Instatiate and train (=fit) a new model with the best parameters on the training data
7. Test (=predict) the model on the testing data
8. Plot the predictions vs true values
9. Evaluate the model using a metric (=MAD/MEAN RATIO score)

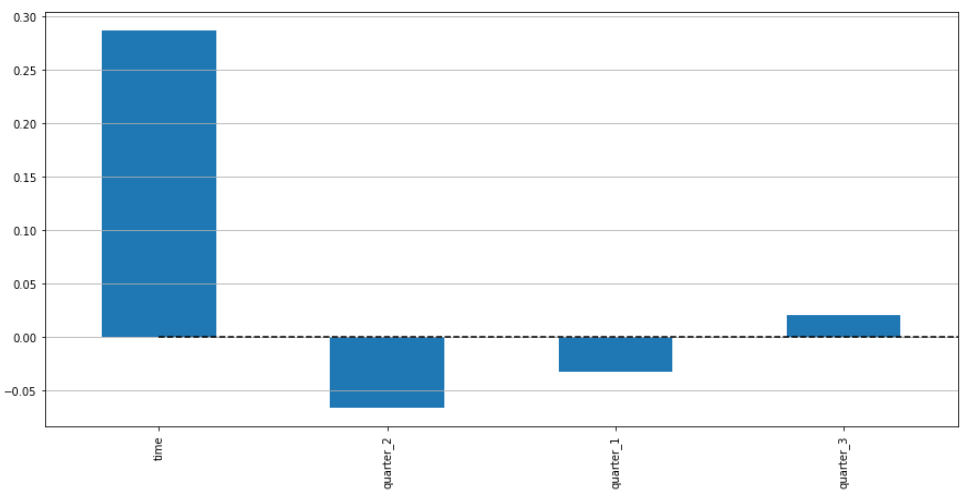
The MAD/MEAN ratios should be near:



That is, the Train set error should go up and the Test set error should go down.

The visual output of Ridge should be near:





So the ridge beta (slope) coefficients should have shrunk in size when compared to the corresponding plain regression beta (slope) coefficients.

NOTE:

To get the model directly out of the grid search object, use the property: grid\_search\_object.best\_estimator\_[1]

(grid\_search\_object.best\_estimator\_[0] is the scaler).

Regarding steps 6, 7, 8 and 9 above:

If you do (A):

A: you instantiate a whole new regression object giving it the parameters obtained from the grid search object’s best\_params\_ property and you predict the test data using that newly instantiated regression object, the results are going to be different than if you do (B):

B: you use the grid search object to predict the test data directly using the grid search object’s predict method\*\*.

Why?

Because the length of the training data is going to be different in the two cases (actually, this used to be the case but this is no longer true. In the latest implementation of GridSearchCV the refit argument defaults to True, so the two cases A and B above should result in the same output). In A, all the training data is being used to train the model, in B, part of the training data is separated out for validation. Try it both ways and see. But in the end, for this assignment select the method that gives a graph that is closest to the one suggested.

\*\*You can do (B) two ways:

You can predict with the grid search object’s predict method or

You can predict by first obtaining the grid search object’s best\_estimator\_[1] and using it to predict.

Note that the grid search object best estimator is actually a pipeline that includes scaling as the first step.

So the input of the grid search object need not be scaled.

Last step:

You may be wondering how one can determine the search space for the regularization parameter of a model. In particular, how do we know that the following search space is enough to find the best model:

param\_grid = [{'ridge\_\_alpha':[1000,500,100,50,10,5,1,.5,.1]}]

In fact we do not know. But there is a way to find out what the leftmost end point of this ridge\_\_alpha interval should be. It is not 1000 as stated above: it is LambdaMax. But what is LambdaMax? LambdaMax is the value of the regularization parameter that causes the model’s coefficients to approach zero. LambdaMax can be determined for a ridge, lasso or logistic regression by running the following function:

#This function calculates the smallest value of lambda that

#forces coefficients to go down to zero in a model (ridge, lasso, or logistic)

#This smallest value of lambda is called LambdaMax.

#This function is following the equations in:

#https://archive.is/RVysJ#selection-803.0-803.6

#https://archive.is/vhQbn

#and in glmnet.pdf here: https://jerryfriedman.su.domains/ftp/glmnet.pdf

#X should be scaled

def LambdaMax(X, Y, model\_type):

from sklearn.linear\_model import Ridge

from sklearn.linear\_model import Lasso

from sklearn.linear\_model import LogisticRegression

if not isinstance(Y, np.ndarray):

Y = Y.values #for logistic Y values should be categorical

if not isinstance(X, np.ndarray):

X = X.values #X values should be scaled

n = X.shape[0]

if model\_type=="lasso":

# Calculate LambdaMax for lasso-like regularization

LambdaMax\_lasso = np.max(np.abs(X.T @ Y))

print("LambdaMax (for Lasso):", LambdaMax\_lasso)

model = Lasso(alpha=LambdaMax\_lasso)

model.fit(X,Y)

print("lasso coefs", model.coef\_ )#coefs should be close to zero

if model\_type == "ridge":

# Calculate LambdaMax for ridge-like regularization

LambdaMax\_ridge = 2 \* np.max(np.abs(X.T @ Y)) / n

print("LambdaMax (for Ridge):", LambdaMax\_ridge)

model = Ridge(alpha=LambdaMax\_ridge)

model.fit(X,Y)

print("ridge coefs", model.coef\_ )#coefs should be close to zero

if model\_type == "logistic":

# Fit logistic regression model without regularization (high C value means less regularization)

model = LogisticRegression(penalty='l2', C=1e10, solver='liblinear')

model.fit(X, Y)

# Calculate the gradient of the loss function at the fitted coefficients

# Logistic loss gradient w.r.t. coefficients: X.T \* (Y - p)

p = model.predict\_proba(X)[:, 1] # model predictions as probabilities

gradients = np.dot(X.T, (Y - p)) # gradients of the loss function

# Calculate LambdaMax\_logistic for L2 Logistic Regression

LambdaMax\_logistic = np.max(np.abs(gradients))

print("LambdaMax (for Logistic):", LambdaMax\_logistic)

print("C (for Logistic):", 1/LambdaMax\_logistic)

model = LogisticRegression(C=1/LambdaMax\_logistic) #C = 1/LambdaMax\_logistic

model.fit(X,Y)

print("logistic coefs",model.coef\_ )#coefs should be close to zero

You can call this function as follows:

LambdaMax(X\_train, y\_train, "ridge")

And it will print the value of LambdaMax. Then you can define the 'ridge\_\_alpha' parameter grid as follows:

# Start value

start = PUT\_VALUE\_OF\_LAMBDA\_MAX\_HERE #Obtained from LambdaMax

# End value

stop = 0.1

# Number of points

num\_points = 50

# Generate evenly spaced values

evenly\_spaced\_values = np.linspace(start, stop, num\_points)

param\_grid = [{'ridge\_\_alpha':evenly\_spaced\_values}]

Run the ridge regression with this new regularization parameter grid to see if the model test set became more accurate.