ELECTRICITY

PRICE

PREDICTION PHASE 3 : Document



TEAM MEMBERS:

S. NISHA (612721104064)

S. NIVETHA (612721104067)

P. PAVITHRA (612721104069)

S. PARAMESHWARI (612721104068)

INTRODUCTION:

Electricity price forecasting (EPF) is a branch of energy forecasting which focuses on predicting the spot and forward prices in wholesale electricity markets. Over the last 15 years electricity price forecasting have become a fundamental input to energy companies’ decision-making mechanisms at the corporate level.



Price forecasting is predicting a commodity/ product /service price by evaluating various factors like its characteristics, demand, seasonal ,trends ,other commodities prices etc.,

Analyzing Electricity Price Time Series Data using Python: Time Series Decomposition and Price Forecasting using a Vector Autoregression (VAR) Model

def retrieve \_time \_series(api, series \_ID):

"""

Return the time series data frame, based on API and unique Series ID

api: API that we're connected to series \_ID:

string. Name of the series that we want to pull from the EIA API

"""

#Retrieve Data By Series ID

Series \_search = api .data \_by \_series(series=series \_ID)

##Create a pandas

data frame from the retrieved time series df = pd .Data Frame(series \_search) return df

###Execute in the main block

#Create EIA API using your specific

API key aPi \_key = "YOR API KEY HERE"

api = eia .API(api \_key)

#Pull the electricity price data

series \_ID='ELEC.PRICE.TX-ALL.M'

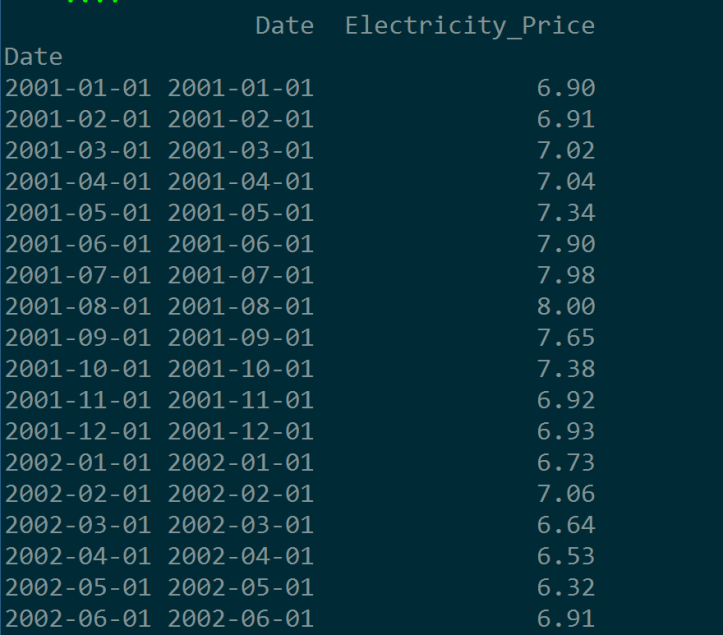
electricity \_df= retrieve \_time \_series(api, series \_ID)

electricity \_df .reset \_index(level=0, in place=True)

#Rename the columns for easer analysis

electricity \_df .rename(columns={'index' :'Date',

electricity \_df .columns[1]:'Electricity \_Price'}, in place=True)



Snapshot of the time series data for electricity prices, pulled via the EIA API

First, let’s look at whether or not the monthly electricity data displays seasonality and a trend. To do this, we use the seasonal\_decompose() function in the statsmodels.tsa.seasonal package. This function breaks down a time series into its core components: trend, seasonality, and random noise. The code and its outputs are displayed below:

def decompose\_time\_series(series):

"""

Decompose a time series and plot it in the console

Arguments:

series: series. Time series that we want to decompose

Outputs:

Decomposition plot in the console

"""

result = seasonal\_decompose(series, model='additive')

result.plot()

pyplot.show()

#Execute in the main block

#Convert the Date column into a date object

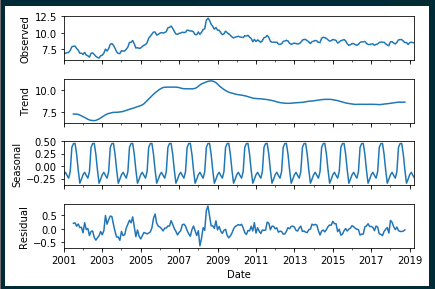
electricity\_df['Date']=pd.to\_datetime(electricity\_df['Date'])

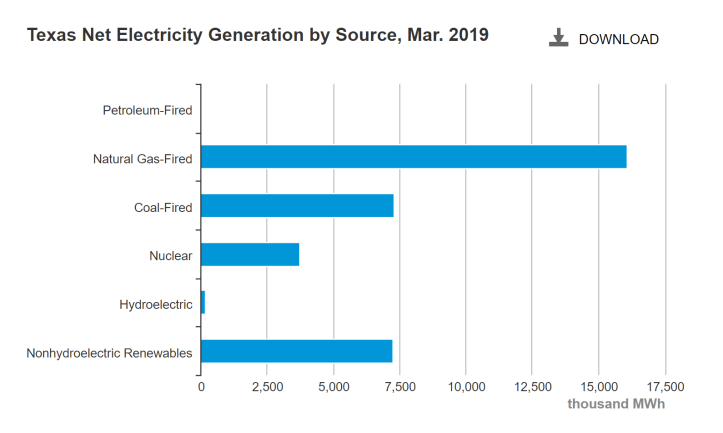
#Set Date as a Pandas DatetimeIndex

electricity\_df.index=pd.DatetimeIndex(electricity\_df['Date'])

#Decompose the time series into parts

decompose\_time\_series(electricity\_df['Electricity\_Price'])

Time Series Decomposition: Monthly Electricity Prices in TX



Although we’re analyzing electricity data going back to 2001 and the breakdown above is from March 2019, for the sake of simplicity we’ll assume that natural gas has been one of the main sources of electricity generation in Texas for the past 15-20 years. Consequently, we’re going to pull the time series of natural gas prices via the EIA API and compare it side-by-side to the TX electricity price time series:

#Pull in natural gas time series data

series\_ID='NG.N3035TX3.M'

nat\_gas\_df=retrieve\_time\_series(api, series\_ID)

nat\_gas\_df.reset\_index(level=0, inplace=True)

#Rename the columns

nat\_gas\_df.rename(columns={'index':'Date',

nat\_gas\_df.columns[1]:'Nat\_Gas\_Price\_MCF'},

inplace=True)

#Convert the Date column into a date object

nat\_gas\_df['Date']=pd.to\_datetime(nat\_gas\_df['Date'])

#Set Date as a Pandas DatetimeIndex

nat\_gas\_df.index=pd.DatetimeIndex(nat\_gas\_df['Date'])

#Decompose the time series into parts

decompose\_time\_series(nat\_gas\_df['Nat\_Gas\_Price\_MCF'])

#Merge the two time series together based on Date Index

master\_df=pd.merge(electricity\_df['Electricity\_Price'], nat\_gas\_df['Nat\_Gas\_Price\_MCF'],

left\_index=True, right\_index=True)

master\_df.reset\_index(level=0, inplace=True)

#Plot the two variables in the same plot

plt.plot(master\_df['Date'],

master\_df['Electricity\_Price'], label="Electricity\_Price")

plt.plot(master\_df['Date'],

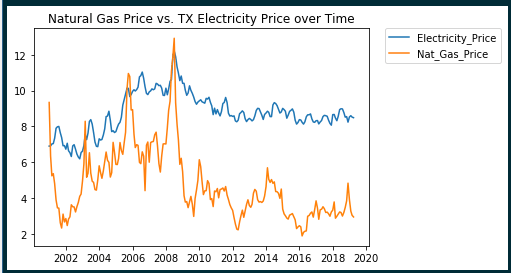
master\_df['Nat\_Gas\_Price\_MCF'], label="Nat\_Gas\_Price")

# Place a legend to the right of this smaller subplot.

plt.legend(bbox\_to\_anchor=(1.05, 1), loc=2, borderaxespad=0.)

plt.title('Natural Gas Price vs. TX Electricity Price over Time')

plt.show()

Plotted Natural Gas Prices and Electricity Prices over Time

There appears to be more variation in the natural gas price time series (the highs are especially high and the lows are especially low), but for the most part, the trends in both time series appear very similar. In fact, it’s highly likely we could estimate TX electricity price using natural gas price as a proxy.

There are a couple approaches that we can take to make a model stationary. They are as follows:

1. Difference the time series. Data differencing is, more or less, calculating the instantaneous velocity at each data point, or the amount the time series changes from one value to the next.
2. Transform the time series. This can be done by applying a log or power transformation to the data.

Next, in order for a VAR model to work, the sampling frequency (i.e. daily, monthly, yearly data) needs to be the same. If it isn’t, the data needs to be converted to the same frequency using imputation/linear interpolation, etc.

In the code below, each time series is transformed using the numpy natural log function, and then differenced by one interval:

#Transform the columns using natural log

master\_df['Electricity\_Price\_Transformed']=np.log(master\_df['Electricity\_Price'])

master\_df['Nat\_Gas\_Price\_MCF\_Transformed']=np.log(master\_df['Nat\_Gas\_Price\_MCF'])

#Difference the data by 1 month

n=1

master\_df['Electricity\_Price\_Transformed\_Differenced'] = master\_df['Electricity\_Price\_Transformed'] - master\_df['Electricity\_Price\_Transformed'].shift(n)

master\_df['Nat\_Gas\_Price\_MCF\_Transformed\_Differenced'] = master\_df['Nat\_Gas\_Price\_MCF\_Transformed'] - master\_df['Nat\_Gas\_Price\_MCF\_Transformed'].shift(n)

determine that a time series is stationary, the test must return a p-value of less than .05. In the Python code below, we run the Augmented Dickey-Fuller test on the transformed, differenced time series, determining that they are both stationary (the p-values returned are .000299 and 0 for the electricity and natural gas time series, respectively):

def augmented\_dickey\_fuller\_statistics(time\_series):

"""

Run the augmented Dickey-Fuller test on a time series

to determine if it's stationary.

Arguments:

time\_series: series. Time series that we want to test

Outputs:

Test statistics for the Augmented Dickey Fuller test in

the console

"""

result = adfuller(time\_series.values)

print('ADF Statistic: %f' % result[0])

print('p-value: %f' % result[1])

print('Critical Values:')

for key, value in result[4].items():

print('\t%s: %.3f' % (key, value))

#Execute in the main block

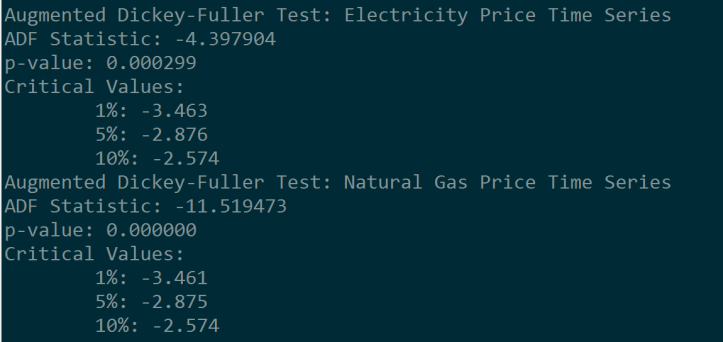
#Run each transformed, differenced time series thru the Augmented Dickey Fuller test

print('Augmented Dickey-Fuller Test: Electricity Price Time Series')

augmented\_dickey\_fuller\_statistics(master\_df['Electricity\_Price\_Transformed\_Differenced'].dropna())

print('Augmented Dickey-Fuller Test: Natural Gas Price Time Series')

augmented\_dickey\_fuller\_statistics(master\_df['Nat\_Gas\_Price\_MCF\_Transformed\_Differenced'].dropna())



Outputs for the Augmented Dickey-Fuller Test for the electricity price time series and the natural gas price time series, respectively

Now that we’ve made our two time series stationary, it’s time to fit the data to a VAR model. The Python code below successfully builds the model and returns a summary of the results, where we use a 95/5 percent split for the training/validation sets:

#Conver the dataframe to a numpy array

master\_array=np.array(master\_df[['Electricity\_Price\_Transformed\_Differenced',

'Nat\_Gas\_Price\_MCF\_Transformed\_Differenced']].dropna())

#Generate a training and test set for building the model: 95/5 split

training\_set = master\_array[:int(0.95\*(len(master\_array)))]

test\_set = master\_array[int(0.95\*(len(master\_array))):]

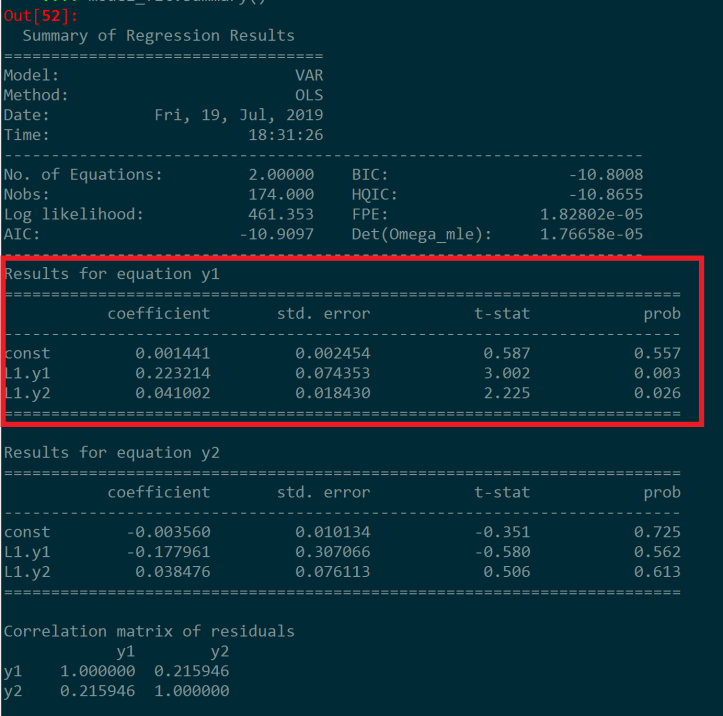
#Fit to a VAR model

model = VAR(endog=training\_set)

model\_fit = model.fit()

#Print a summary of the model results

model\_fit.summary()



VAR Model Summary, with y1=Electricity Price Time Series, and y2=Natural Gas Price Time Series

What we really want to focus on in the model summary above is the equation for y1, where y1 estimates the electricity prices in the state of Texas based on lagged values of itself and lagged natural gas prices.

def calculate\_model\_accuracy\_metrics(actual, predicted):

"""

Output model accuracy metrics, comparing predicted values

to actual values.

Arguments:

actual: list. Time series of actual values.

predicted: list. Time series of predicted values

Outputs:

Forecast bias metrics, mean absolute error, mean squared error,

and root mean squared error in the console

"""

#Calculate forecast bias

forecast\_errors = [actual[i]-predicted[i] for i in range(len(actual))]

bias = sum(forecast\_errors) \* 1.0/len(actual)

print('Bias: %f' % bias)

#Calculate mean absolute error

mae = mean\_absolute\_error(actual, predicted)

print('MAE: %f' % mae)

#Calculate mean squared error and root mean squared error

mse = mean\_squared\_error(actual, predicted)

print('MSE: %f' % mse)

rmse = sqrt(mse)

print('RMSE: %f' % rmse)

#Execute in the main block

#Un-difference the data

for i in range(1,len(master\_df.index)-1):

master\_df.at[i,'Electricity\_Price\_Transformed']= master\_df.at[i-1,'Electricity\_Price\_Transformed']+master\_df.at[i,'Electricity\_Price\_Transformed\_Differenced\_PostProcess']

#Back-transform the data

master\_df.loc[:,'Predicted\_Electricity\_Price']=np.exp(master\_df['Electricity\_Price\_Transformed'])

#Compare the forecasted data to the real data

print(master\_df[master\_df['Predicted']==1][['Date','Electricity\_Price', 'Predicted\_Electricity\_Price']])

#Evaluate the accuracy of the results

calculate\_model\_accuracy\_metrics(list(master\_df[master\_df['Predicted']==1]['Electricity\_Price']),

list(master\_df[master\_df['Predicted']==1 ['Predicted\_Electricity\_Price']))

ELECTRICITY CONSUMPTION:

In recent years, forecasting usage python learning approaches has gained popularity as a study topic. Accurately projecting future power consumption isessential for effective energy management ,cost savings, and environmental sustainability given the rising demand for energy. To make accurate forecasts,it is essential to choose the right characterize and models. Additionally , prediction energy consumption is a continual process that has to be update and monitored often to account for change in consumer behavior, environmental conditions , and other pertinent variables.