ADVANCES IN DATA SCIENCES
INFO 7390
FINAL PROJECT:

FED ECONOMIC DATA ANALYSIS

Under the guidance of Professor Srikanth Krishnamurthy

Team 6:

Puneeth Kumar (https://www.linkedin.com/in/puneethkumarreddy),
Yamini Sehrawat (linkedin.com/in/yamini-sehrawat-1a26a197),
Vaidehi Deshpande (linkedin.com/in/vaidehid)

Table of Contents

FED Economic Indicators Dataset

Dataset Description

Dataset Link: https://www.richmondfed.org/research/regional_economy/reports/regional_profiles#tab-2

The general overview of the economic performance includes indicators relating to output, demography, labor markets, household conditions, and real estate activity. Collected by the Regional Economics section of the Federal Reserve Bank of Richmond, the data is reported at the national, state, metropolitan statistical area, and county level where available, and is updated annually each summer.
This dataset consists of annual economic data at the national level and at state level for following states and their respective counties and metropolitan areas:

- District of Columbia
- Maryland
- North Carolina
- South Carolina
- Virginia
- West Virginia

The original sources of individual economic indicator dataset have also been listed in case the data presented here is incomplete, the original sources can be looked up for correct data. Below is the list of some of these data sources:

- Bureau of Economic Analysis, Haver Analytics
- Bureau of Labor Statistics, Haver Analytics
- US Census Bureau, Haver Analytics
- US Census Bureau
- Office of Federal Housing Enterprise Oversight, Haver Analytics
- Federal Deposit Insurance Corporation

The data presented in this dataset is in the form of Time Series. The values for each economic indicator for each state/Metropolitan Area/County are available from 1977 (in some cases from 1990) to 2015. Due to availability of large number of indicators, we can identify the relationships between different indicators, how they impact values of other indicators and also the dependency of future values on the previous values of same indicator. Based on these definitions, there are two types of Time Series Data Analysis possible:

- **Univariate: The future values of variable depends on its previous values**
- **Multivariate: The values of dependent variable depend on its previous values as well as values of other variables in the dataset**

Problem Statement

Our aim here is to build machine learning models which will synthesize the available data, predict the existing values and forecast the future values. The prediction of existing values will show how robust our model is and determine the accuracy of forecasted values. We are considering following indicators for our economic analysis:

- **Gross State Product (Real)**
- **Unemployment Rate**
- **Total Personal Income**
- **House Price Index**

The forecast generated using this model can help us in knowing following details:

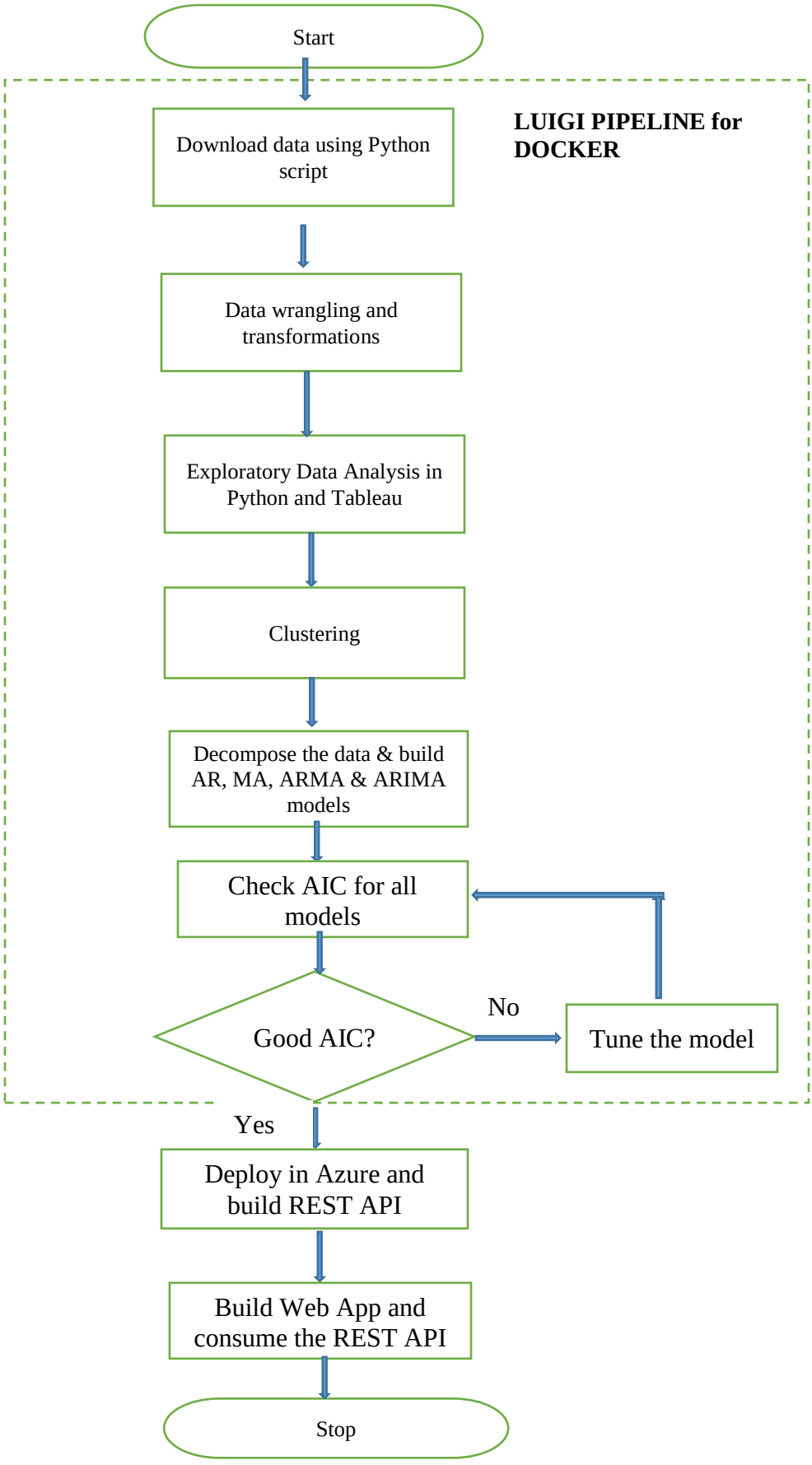
- **Economic development of a particular region: based on GDP trends**
- **Job market trends: based on Unemployment Rate**
- **Income and Earnings trends: based on Total Personal Income**
- **Real Estate Market: House Price Index trends**

We are restricting our analysis here to Univariate Time Series Analysis but at the same time building a base for Multivariate Time Series Analysis. Following points summarize our approach in handling this project:

- **Step 1: Data Download and Data Wrangling**
- **Step 2: Exploratory Data Analysis: Summarizations and Analysis**
- **Step 3: Clustering the data**
- Step 4: Decomposing Time series'
- **Step 5: Making Time Series Stationary**
- **Step 6: Building AR, MA, ARMA and ARIMA untuned models**
- **Step 7: Choosing the best tuned model**
- **Step 8: Pipeline all the activities using Luigi and Dockerize the projects**
- **Step 9: Upload the Docker image and Web app to cloud (S3)**
- **Step 10: Deploy the best model in Azure and create REST API**
- **Step 11: Consume the REST API in the web app**

The web application developed in this process will allow the user to select a state, year and economic indicator of his/her choice and get the requested values. The above mentioned steps are represented using a Flow chart and explained in further detail in following sections.

Flow Chart


Start
LUIGI PIPELINE for DOCKER
Download data using Python script
Data wrangling and transformations
Exploratory Data Analysis in Python and Tableau
Clustering
Decompose the data & build AR, MA, ARMA & ARIMA models
Check AIC for all models
Good AIC?
No
Tune the model
Yes
Deploy in Azure and build REST API
Build Web App and consume the REST API
Stop

Step 1: Data Download and Wrangling

- **The first step here is to programmatically download the data from FED Data website: https://www.richmondfed.org/research/regional_economy/reports/regional_profiles#tab-2**
- **The data for each indicator is present in separate csv files. Also data for individual states is present in separate tabs of the csv file. So first we have to clean the data, transform it, and separate the data in tabs into different files.**
- **Below is the sample screenshot of original csv files**

District of Columbia - Real GDP											
	2015	2014	2013	2012	2011	2010	2009	2008	2007	2006	2005
National (Millions of Chained 2009 $)	16023115	15653000	15317174	15126279	14833679	14628165	14320114	14718301	14798367	14593536	14211385
	2015	2014	2013	2012	2011	2010	2009	2008	2007	2006	2005
Statewide (Millions of Chained 2009 $)	107955	105312	103430	103804	103820	101904	98938	100104	97228	94943	92994
Metropolitan Statistical Areas (Millions of 2005 Chained $)	2015	2014	2013	2012	2011	2010	2009	2008	2007	2006	2005
Washington-Arlington-Alexandria, DC-VA-MD-WV	---	435583	434421	434480	433500	427539	414340	414619	405721	398768	390987

- **Following is the screenshot of transformed files created for each indicator for each state:**

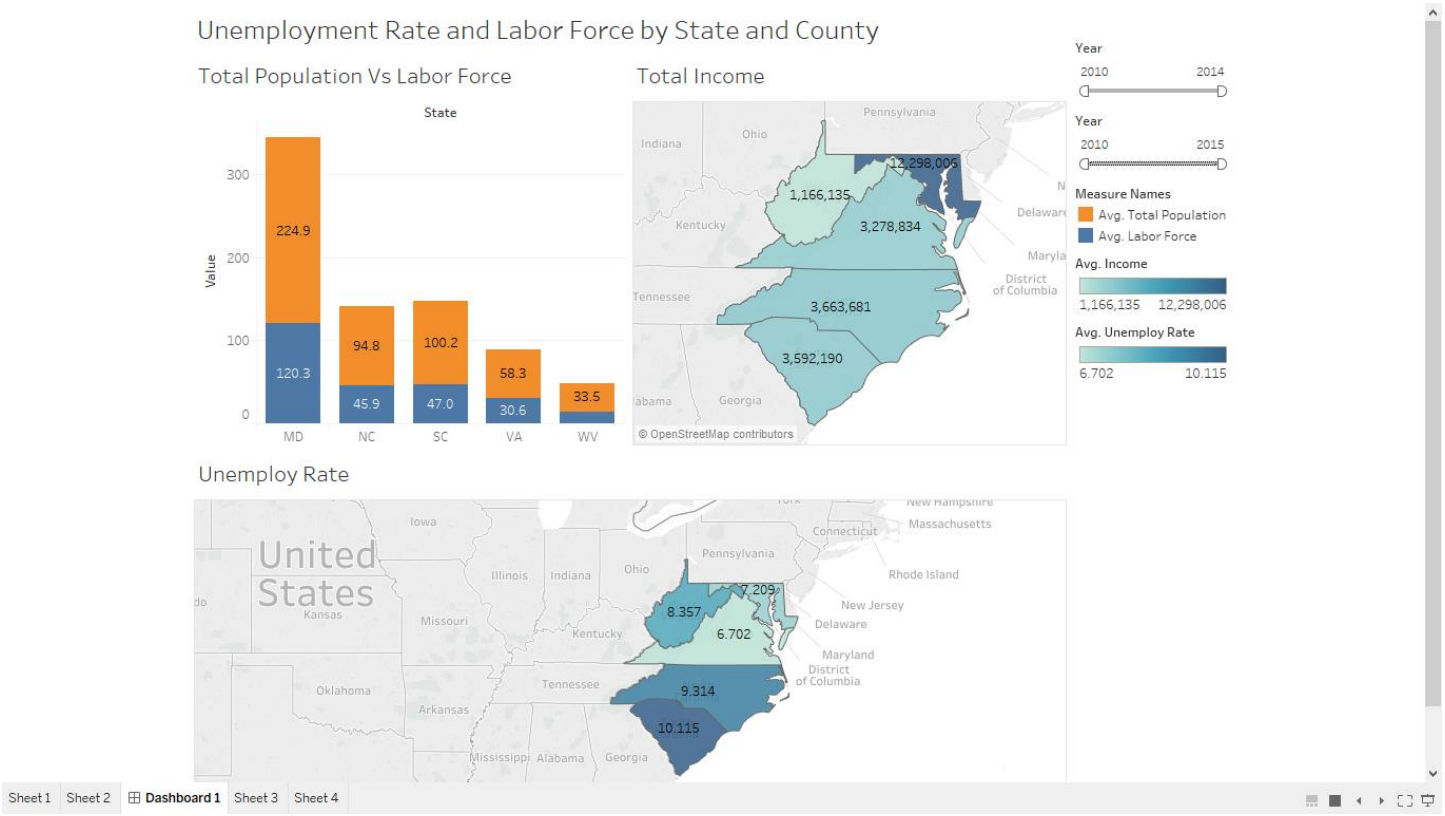
Year	DC_Gross State Product (Real)
2015	107955
2014	105312
2013	103430
2012	103804
2011	103820
2010	101904
2009	98938
2008	100104
2007	97228
2006	94943
2005	92994
2004	91569

- **The data for each state and respective counties and metro areas was present in row format, hence we had to clean it and transform it in column format to further save it as a dataframe.**
- **After creation of dataset, check for missing data and impute correct values**
- **Missing Data Handling:**
  In this case, we had very little missing data. Only some value for West Virginia for Total Personal Income was missing. Here we have imputed the value using bfill() method.
- **Build a method for data download and wrangling to be included in the Luigi pipeline for automating the task**

Step 2: Exploratory Data Analysis

Here we have summarized some important variables and tried to figure out the correlations between them. This helped us to gain some key insights on the trends and impacts of variables on other variables. Based on the conclusions it was further decided which variables to consider for creating machine learning models.

1. **Compare different states based on Total Population, Labor Force, Unemployment Rate and Total Personal Income:**

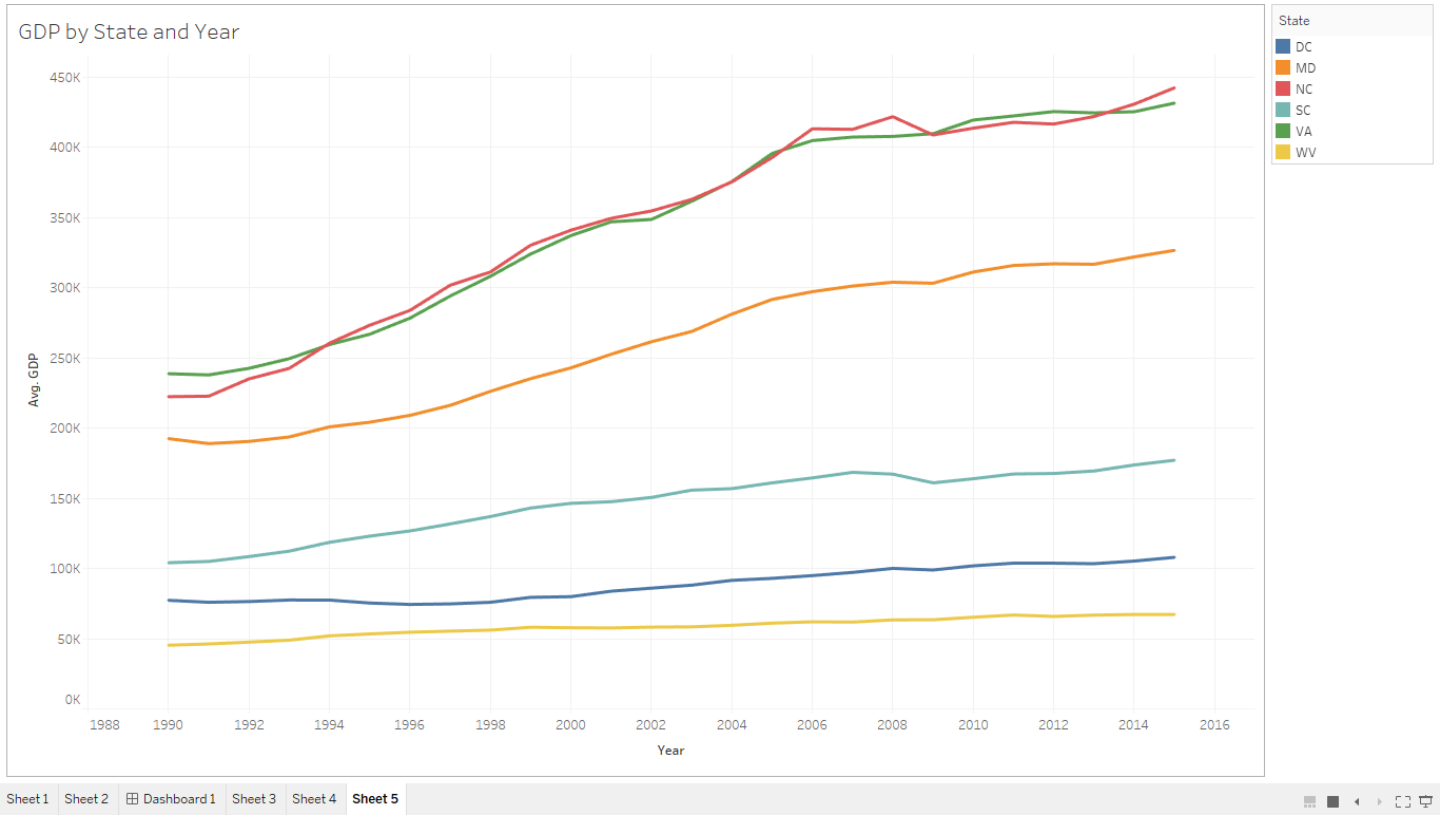


This analysis has been performed in Tableau.

Conclusions:

- **The comparison shows that Maryland has largest population and almost 50% of it is a labor force. Because of which Unemployment rate is the lowest in Maryland while Total personal income is the highest.**
- **On the other hand, for West Virginia, the Unemployment Rate is relatively higher and Total personal income is the lowest.**

2. Compare trends in GDP for all states:



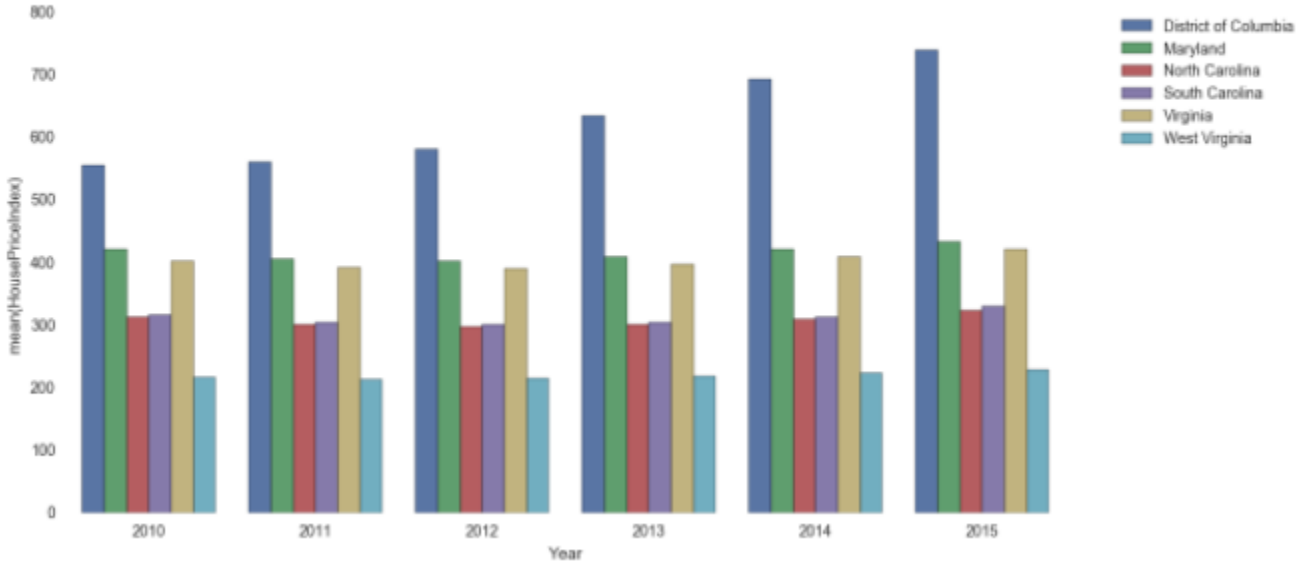
Conclusions:

- **As seen in above graph North Carolina and Virginia have comparable GDPs and high among the remaining states**
- **There is an increasing trend in GDPs for all states from 1990 till 2015 except for a small dip during economic crisis time period.**

3. Compare states and metro areas for trends in House Price Index

a) HPI for all states from 2010-2015:

```
sns.barplot(x=df_State['Year'],y=df_State['HousePriceIndex'], hue=df_State['State']);
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.show()
```

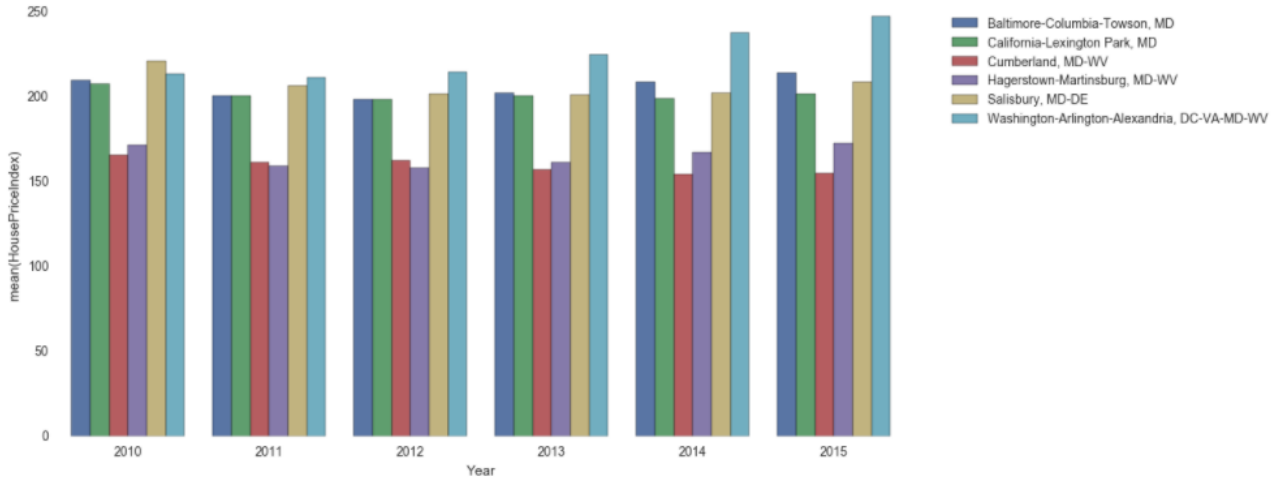


Conclusion:

- **District of Columbia seems to be a major contributor in HPI among all the other states.**
- **It shows an upward trend over 6 year period that we have considered here.**

b) HPI for Maryland and its Metro Areas:

```
In [59]: sns.barplot(x=df_Maryland['Year'],y=df_Maryland['HousePriceIndex'], hue=df_Maryland['Metropolitan Statistical Areas']);
         plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
         plt.show()
```



This analysis has been performed in Python. As inferred from the earlier analysis Maryland had highest Total Personal Income and lowest unemployment Rate hence, here we have compared the mean House

Price Index for six years from 2010-2015 for Maryland state further drilling it down to analyze the Metropolitan area contribution for each year.

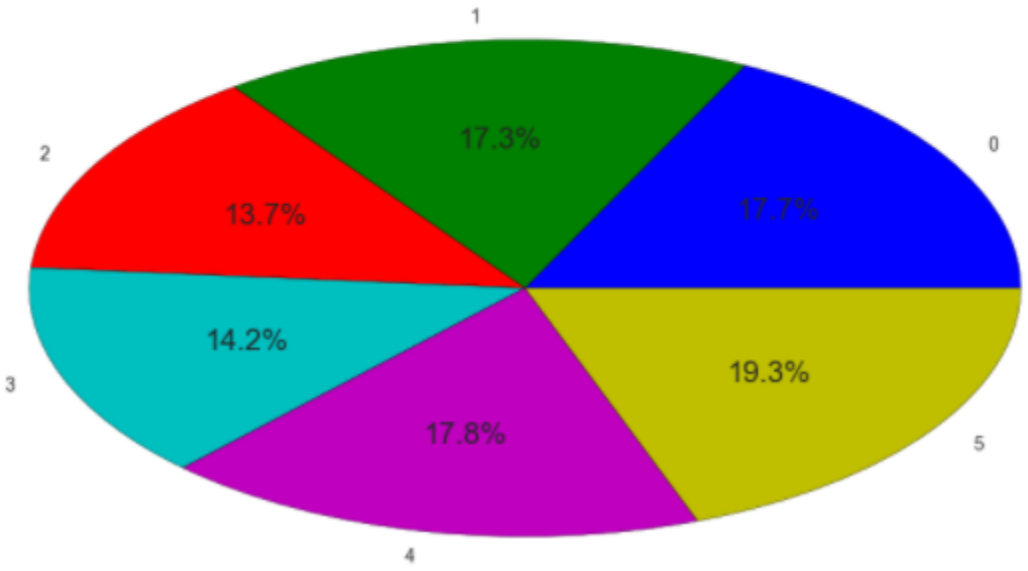
Conclusions:

- **The metro area comprising of Washington-Arlington-Alexandria shows an upward trend over a period of time in the House Price Index. This shows the growth in demand for houses in this area,**
- **The Baltimore-Columbia-Towson area and California-Lexington Park always seem comparable over a period of six years. Same applies for Cumberland and Haerstown area whose HPI remains comparable for all the years.**
- **We can also infer that HPI for other areas apart from Washington-Arlington-Alexandria has not changed for the period of 6 years 2010-2015.**
- **Going ahead we have plotted the individual contribution of each metro area in Maryland towards the House Price Index. As we can see though Washington area shows growth in the HPI, contribution of each metro area is almost comparable.**

```
HPICount
```

	Metropolitan Statistical Areas	HousePriceIndex
0	Baltimore-Columbia-Towson, MD	205.708333
1	California-Lexington Park, MD	201.430417
2	Cumberland, MD-WV	159.355000
3	Hagerstown-Martinsburg, MD-WV	165.198750
4	Salisbury, MD-DE	206.929167
5	Washington-Arlington-Alexandria, DC-VA-MD-WV	225.005000

```
plt.pie(HPICount['HousePriceIndex'],labels=HPICount.index,autopct="%1.1f%%")
plt.show()
```



Step 3: Clustering

Manual Clustering:

We have programmatically clustered the dataset based on different states and type of indicator. On these clusters we have performed further analysis of predicting the existing values and forecasting future values using AR, MA, ARMA and ARIMA models.

Clustering Algorithm

Since we have time series data of varying lengths, we cannot apply the normal techniques of calculating Euclidean distance between data points and clustering based on k-means algorithm. There is a special technique called Dynamic Time Warping (DTW) available for calculating the similarity between different time series. Below is the screenshot of the code used by DTW algorithm:

```
def DTWDistance(s1, s2):
    DTW={}

    for i in range(len(s1)):
        DTW[(i, -1)] = float('inf')
    for i in range(len(s2)):
        DTW[(-1, i)] = float('inf')
    DTW[(-1, -1)] = 0

    for i in range(len(s1)):
        for j in range(len(s2)):
            dist= (s1[i]-s2[j])**2
            DTW[(i, j)] = dist + min(DTW[(i-1, j)],DTW[(i, j-1)], DTW[(i-1, j-1)])

    return np.sqrt(DTW[len(s1)-1, len(s2)-1])
```

We have implemented this function further to calculate distance measures between time series data we have here.

```
df1=pd.read_csv("DC_UnemploymentRate.csv", header=0, parse_dates=[0],index_col=0,date_parser=dateparse)
```

```
df2=pd.read_csv("DC_HousePriceIndex.csv", header=0, parse_dates=[0],index_col=0,date_parser=dateparse)
```

```
df3=pd.read_csv("MD_UnemploymentRate.csv", header=0, parse_dates=[0],index_col=0,date_parser=dateparse)
```

```
s1=df1.ix[:,0]
```

```
s2=df2.ix[:,0]
```

```
DTWDistance(s1,s2)
```

```
2239.2517211939853
```

```
s3=df3.ix[:,0]
```

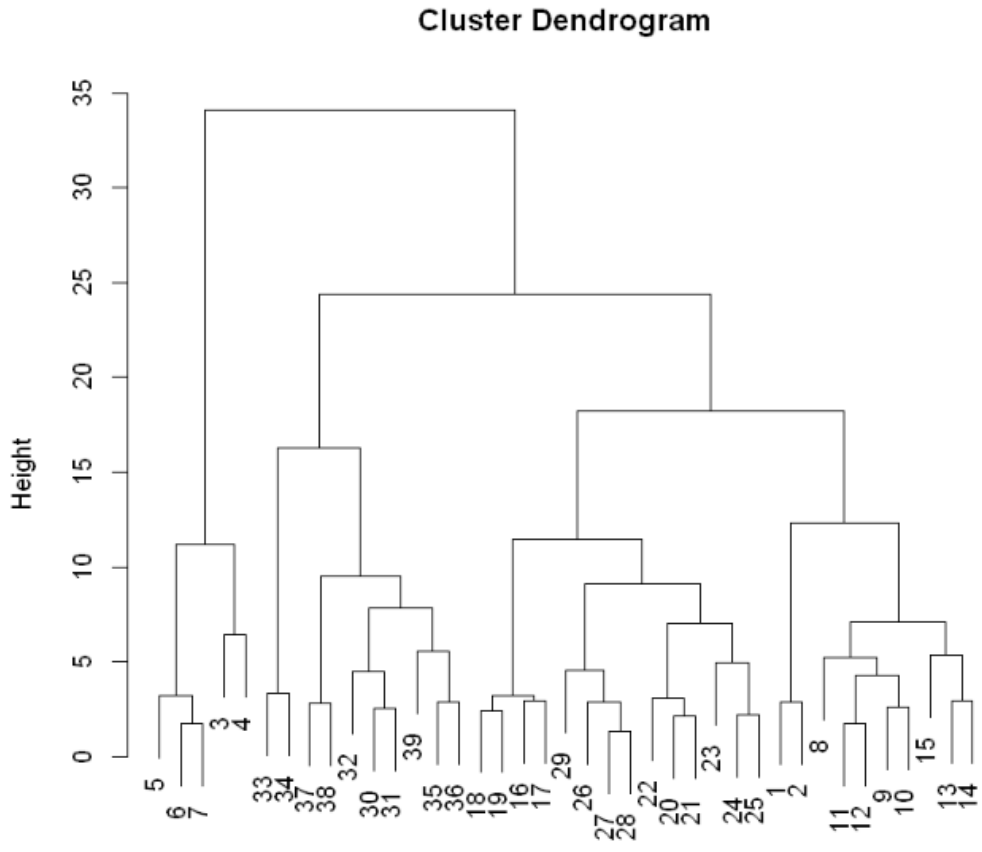
```
DTWDistance(s1,s3)
```

```
7.7351111449075001
```

So we can see that the distance between DC_UnemploymentRate time series and DC_HousePriceIndex time series is very large as compared to distance between DC_UnemploymentRate and MD_UnemploymentRate time series'.

Going ahead we have used the Hierarchical Clustering technique for grouping together similar time series data. Below is the screenshot of the Dendrogram that we got after performing Hierarchical clustering:

```
In [7]: distMatrix <- dist(tsData, method="DTW")

In [8]: hc <- hclust(distMatrix, method="average")

In [9]: plot(hc)
```



This clustering is not much useful for Univariate Time Analysis but is definitely used when doing Multivariate analysis in which we have to identify the similar features which impact the values of dependent variable.

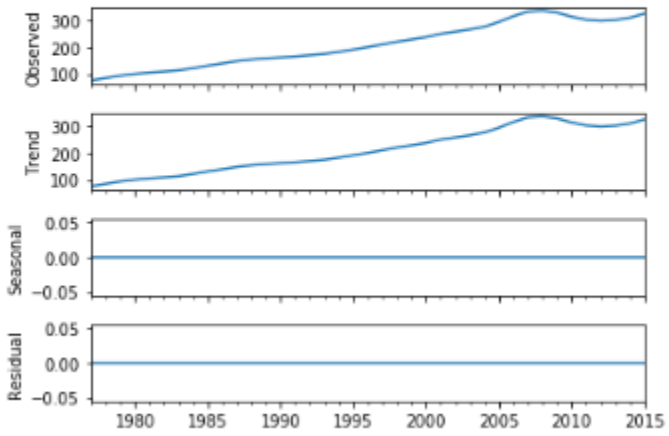
Step 4: Decomposing Time Series

The decomposition of time series is a statistical method that deconstructs a time series into several components, each representing one of the underlying categories of patterns. Here we have decomposed the time series' based on rates of change. It seeks to construct, from an observed time series, a number **of component series (that could be used to reconstruct the original by additions or multiplications) where each of these has a certain characteristic or type of behavior. This is an important technique for seasonal data adjustment. Following are the components in which time series is decomposed:**

- Trend**: at time t reflects the long-term progression of the series. A trend exists when there is an increasing or decreasing direction in the data**
- Cyclical**: at time t describes repeated but non-periodic fluctuations. The duration of these fluctuations is usually of at least two years.**
- Seasonal**: at time** t**, reflects seasonality (seasonal variation). A seasonal pattern exists when a time series is influenced by seasonal factors. Seasonality is always of a fixed and known period (e.g., the quarter of the year, the month, or day of the week)**
- Noise**: at time t, that describes random, irregular influences. It represents the residuals or remainder of the time series after the other components have been removed.**

In our dataset we had yearly data hence, the seasonality component was always zero. We could see a clear trend for all the indicators selected in this process. Below is the sample seasonal decomposition graphs of all the indicators:

```
In [13]: #Decomposing into  trend, seasonal and residual using additive model.
         decomposition = sm.tsa.seasonal_decompose(NC_HPI, model='additive')
         fig = decomposition.plot()
         plt.show()
         #fig.savefig('Seasonal_Decompose.png', dpi=150)
```



Step 5: Making Time Series Stationary

Once we decompose the available time series data, we can see that the values of indicators vary a period of time, which is expected. But in order to do further detailed analysis we need to make the time series stationary. A TS is said to be stationary if its statistical properties such as mean, variance remain constant over time. Stationary time series is defined using very strict criterion. However, for practical purposes we can assume the series to be stationary if it has constant statistical properties over time:

- **constant mean**
- **constant variance**
- **an auto covariance that does not depend on time**

As per the decomposition done above it is clear that all our time series have a clear trend. We go ahead and check the stationarity using following two methods:

- **Plotting rolling statistics:**
  Here we can plot the moving average or moving variance and see if it varies with time.

- **Dickey-Fuller Test:**
  This is one of the statistical tests for checking stationarity. Here the null hypothesis is that the TS is non-stationary. The test results comprise of a Test Statistic and some Critical Values for difference confidence levels. If the 'Test Statistic' is less than the 'Critical Value', we can reject **the null hypothesis and say that the series is stationary.**

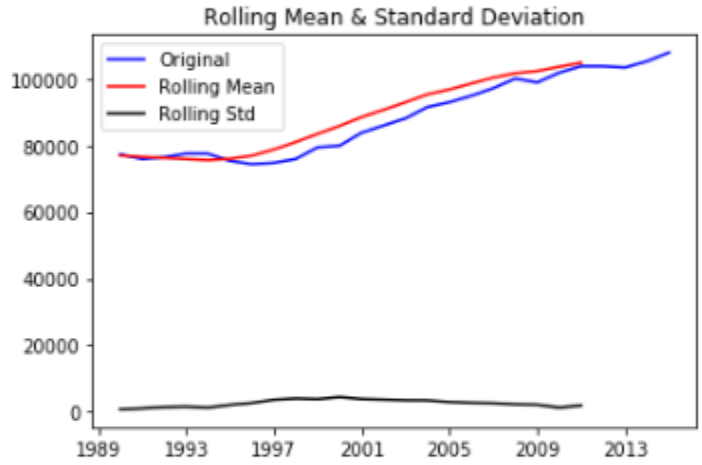
Below is the screenshot of the function that we have created to determine the stationarity of the time series.

In [14]:
```
def test_stationarity(timeseries):

    #Determing rolling statistics
    rolmean = pd.rolling_mean(timeseries, window=5)
    rolstd = pd.rolling_std(timeseries, window=5)

    #Plot rolling statistics:
    orig = plt.plot(timeseries, color='blue',label='Original')
    mean = plt.plot(rolmean, color='red', label='Rolling Mean')
    std = plt.plot(rolstd, color='black', label = 'Rolling Std')
    plt.legend(loc='best')
    plt.title('Rolling Mean & Standard Deviation')
    plt.show(block=False)

    #Perform Dickey-Fuller test:
    print('Results of Dickey-Fuller Test:')
    dftest = adfuller(timeseries, autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags Used','Number of Observations Used'])
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print(dfoutput)
```

We first check the stationarity of our original time series data frame:

```
In [12]: test_stationarity(df)
```

```
C:\Users\Vaidehi Deshpande\Miniconda3\lib\site-packages\
        DataFrame.rolling(center=False,window=5).mean()
C:\Users\Vaidehi Deshpande\Miniconda3\lib\site-packages\
        DataFrame.rolling(center=False,window=5).std()
```




```
Results of Dickey-Fuller Test:
Test Statistic                  -1.743628
p-value                          0.408786
#Lags Used                       0.000000
Number of Observations Used     25.000000
Critical Value (1%)             -3.723863
Critical Value (10%)            -2.632800
Critical Value (5%)             -2.986489
dtype: float64
```

As seen above, this time series is not stationary because of following reasons:

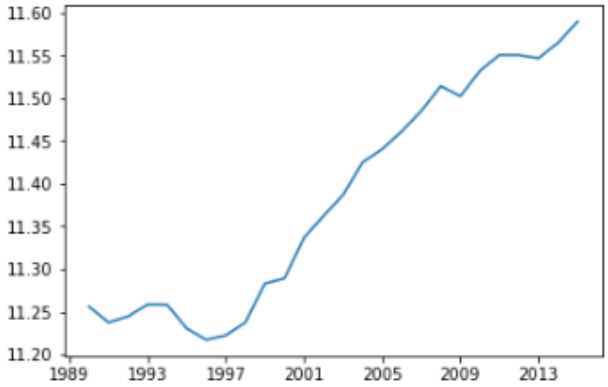
- **Mean and variance are not constant over a period of time**
- The "Test Statistic" value is more than all **three Critical Values**

In order to make the time series stationary we take logarithmic transformations and then again perform stationarity check. Below screenshots explain the same:

Perform Log Transformations to stationarize the Time Series

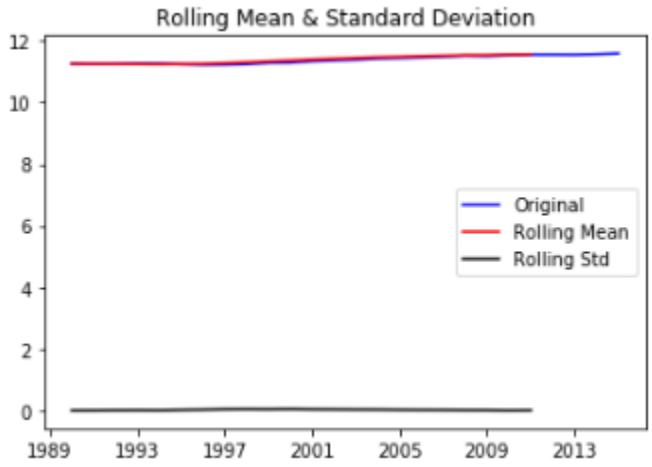
In [13]:
```
df_log=np.log(df)
```

In [14]:
```
plt.plot(df_log)
plt.show()
```



In [15]:
```
#df_log
```

In [16]:
```
test_stationarity(df_log)
```



As seen in above graphs after taking log transform the mean and standard deviation are almost constant.
Hence, we can now consider that this time series is now stationary and go ahead with building regression models.

Step 6: Building Models

In Time Series Data Analysis, we use special type of regression models known as Auto Regression (AR) models. An autoregressive model is when a value from a time series is regressed on previous values from that same time series. In addition to AR, we also have Moving Average (MA) models and a combination of both ARMA models. The predictors depend on the parameters (p,d,q) of the ARIMA model:

- **Number of AR (Auto-Regressive) terms (p): AR terms are just lags of dependent variable. For instance if p is 5, the predictors for x(t) will be x(t-**1)….x(t-**5).**
- **Number of MA (Moving Average) terms (q): MA terms are lagged forecast errors in prediction equation. For instance if q is 5, the predictors for x(t) will be e(t-**1)….e(t-**5) where e(i) is the difference between the moving average at ith instant and actual value.**
- **Number of Differences (d): These are the number of nonseasonal differences, i.e. in this case we took the first order difference. So either we can pass that variable and put d=0 or pass the original variable and put d=1. Both will generate same results.**

Since, we have used Python for our project we imported the statsmodels package which contains all these models.

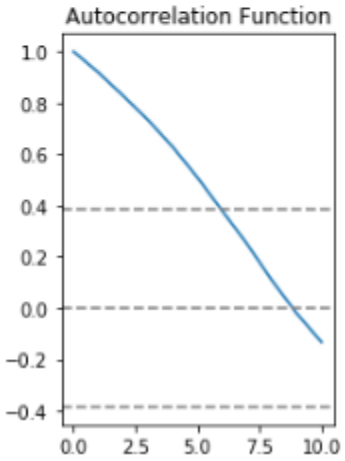
Below are the steps taken to build tuned models:

- **Plot the Autocorrelation (ACF) and Partial Autocorrelation plots.**
  Following plots help us to determine the ideal p and q values for our model.
  - **Autocorrelation Function (ACF): It is a measure of the correlation between the TS with a lagged version of itself. For instance at lag 5, ACF would compare series at time instant** 't1'…'t2' with series at instant 't1-5'…'t2-5' (t1**-5 and t2 being end points).**
    Below is the sample ACF plot from our analysis:

Determine p,q from ACF and PACF plots

In [17]:
```
lag_acf = acf(df_log, nlags=10)
lag_pacf = pacf(df_log, nlags=10, method='ols')
```

In [18]:
```
plt.subplot(121)
plt.plot(lag_acf)
plt.axhline(y=0,linestyle='--',color='gray')
plt.axhline(y=-1.96/np.sqrt(len(df_log)),linestyle='--',color='gray')
plt.axhline(y=1.96/np.sqrt(len(df_log)),linestyle='--',color='gray')
plt.title('Autocorrelation Function')
plt.show()
```



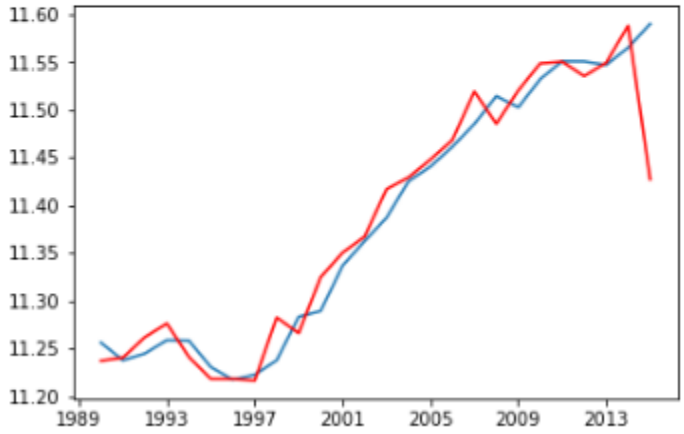
- **Partial Autocorrelation Function (PACF): This measures the correlation between the TS with a lagged version of itself but after eliminating the variations already explained by the intervening comparisons. Eg at lag 5, it will check the correlation but remove the effects already explained by lags 1 to 4.**
  Below is the sample PACF plot from our analysis:

In [19]:
```
plt.subplot(122)
plt.plot(lag_pacf)
plt.axhline(y=0,linestyle='--',color='gray')
plt.axhline(y=-1.96/np.sqrt(len(df_log)),linestyle='--',color='gray')
plt.axhline(y=1.96/np.sqrt(len(df_log)),linestyle='--',color='gray')
plt.title('Partial Autocorrelation Function')
plt.tight_layout()
plt.show()
```

- **Build Models:**
  Once the values of p and q are determined, we go ahead and build the AR, MA and ARMA models and then determine the best model based on value of AIC. The lower the value of AIC the better is the model.

  1. **AR Model:**

```
In [30]: model = ARIMA(df_log, order=(2, 0, 0))
         results_AR = model.fit(disp=0)
         ARAic=results_AR.aic

         plt.plot(df_log)
         plt.plot(results_AR.fittedvalues, color='red')
         plt.show()
         print('AIC: %.4f'% results_AR.aic)
```

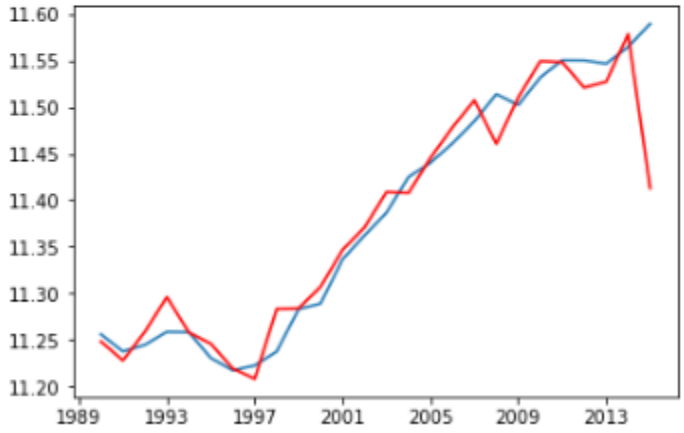



```
AIC: -118.6238
```

2. MA Model

```
In [21]: model = ARIMA(df_log, order=(0, 0, 7))
         results_MA = model.fit(disp=-1)
         MAAic=results_MA.aic
         plt.plot(df_log)
         plt.plot(results_MA.fittedvalues, color='red')
         plt.show()
         print('AIC: %.4f'% results_MA.aic)
```

```
C:\Users\Vaidehi Deshpande\Miniconda3\lib\site-packag
  'available', HessianInversionWarning)
```



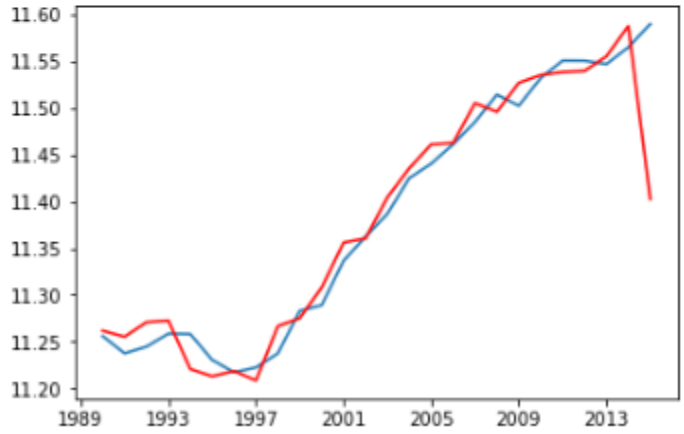

```
AIC: -103.7235
```

3. Combined Model

Combined Model

In [22]:
```
model = ARIMA(df_log, order=(1, 0, 7))
results_ARIMA = model.fit(disp=-1)
ARIMAAic=results_ARIMA.aic

plt.plot(df_log)
plt.plot(results_ARIMA.fittedvalues, color='red')
plt.show()
print('AIC: %.4f'% results_ARIMA.aic)
print('RSS: %.4f'% sum((results_ARIMA.fittedvalues-df_log.ix[:,0])**2))
```

```
C:\Users\Vaidehi Deshpande\Miniconda3\lib\site-packages\statsmodels\base'
  'available', HessianInversionWarning)
```




```
AIC: -109.9131
```

As it is visible in above model screenshots, using the ARMA function from statsmodels we build AR, MA and ARMA models respectively. Here we have taken d=0 since differencing was not needed in this case. Alternate way to build model is using ARMA function as shown below:

In [27]:
```
result=sm.tsa.ARMA(df_log, (2,0))
result=result.fit()
print(result.aic)
np.exp(result.predict(4, 4))
```

```
-118.62383060180046
```

ARMA uses only p,q values in the order. Since, we are not differencing our time series, we do not need d from order (p,d,q) one way is to make it 0 in ARIMA or use ARMA function. By both ways we get the same results (AIC is same in above shown screenshots for AR models).

Step 7: Choose the best model

We have calculated AIC (Akaike information criterion) to measure the strength of the model.

Definition of AIC:
The Akaike information criterion (AIC) is a measure of the relative quality of statistical models for a given set of data. Given a collection of models for the data, AIC estimates the quality of each model, relative to each of the other models. Hence, AIC provides a means for model selection.

Hence, using following piece of code for all indicators and states we find out the best model based on AIC value:

```
if (ARAic < MAAic) & (ARAic < ARIMAAic):
    print('AR selected')
    forecast(model_AR, 5)
    print(FittedValues(model_AR))

elif (MAAic < ARAic) & (MAAic < ARIMAAic):
    print('MA selected')
    forecast(model_MA, 5)
    print(FittedValues(model_MA))

elif (ARIMAAic < ARAic) & (ARIMAAic < MAAic):
    print('ARIMA selected')
    forecast(model_Combined, 5)
    print(FittedValues(model_Combined))
```

- **Predict Existing values and Forecast future values**
  Once the best model is selected using the above mentioned code, we can go ahead and predict the existing values in time series and also forecast the future values. Below screenshots help to explain the functions created for the same:

  - **Predict Values:**

```
def FittedValues(model):
    fittedVal=model.fittedvalues
    PredictedVal=np.exp(fittedVal)
    print('Predicted existing values are:')
    return PredictedVal
```

  - **Forecast Values:**

```
def forecast(model,numSteps):
    #model.forecast(steps=numSteps)
    output = model.forecast(steps=numSteps)[0]
    output.tolist()
    output = np.exp(output)
    print('Forecasted values are:')
    print(output)
    return output
```

  Here numSteps define the number of years in future for which we have to forecast.

- **Sample output for forecasting House Price Index:**

Here we have taken number of steps=5

```
forecast(model_Combined, 5)
print(FittedValues(model_Combined))
```

Output:

```
ARIMA selected
Forecasted values are:
[ 341.95288727  357.60566949  371.63764233  383.95644265  394.50708126]
Predicted existing values are:
1977-01-01    143.462394
1978-01-01     76.410766
1979-01-01     94.392271
1980-01-01    102.523485
1981-01-01    106.042939
1982-01-01    109.891805
1983-01-01    112.488387
1984-01-01    118.010355
1985-01-01    131.806580
1986-01-01    140.480603
1987-01-01    146.643847
1988-01-01    156.925497
1989-01-01    158.280872
1990-01-01    162.875169
1991-01-01    164.323330
1992-01-01    168.982592
1993-01-01    175.268443
1994-01-01    178.279675
1995-01-01    191.904086
1996-01-01    199.477032
1997-01-01    209.248413
1998-01-01    219.254070
1999-01-01    230.906345
2000-01-01    235.165616
2001-01-01    246.280602
2002-01-01    263.851221
2003-01-01    261.223964
2004-01-01    275.604471
2005-01-01    285.894382
2006-01-01    315.741075
2007-01-01    335.746978
2008-01-01    346.111035
2009-01-01    333.265378
2010-01-01    318.077393
2011-01-01    297.644085
2012-01-01    294.395137
2013-01-01    295.651187
2014-01-01    307.030280
2015-01-01    318.819488
dtype: float64
```

AIC and model summary:

```
aic_metric
```

	AIC	Modelname
0	-200.528162	ARIMA(ts, order=(3, 0, 0))
0	4.097298	ARIMA(ts, order=(0, 0, 1))
0	-199.626263	ARIMA(ts, order=(3, 0, 1))

This outputs are later saved in dataframes and then to csv files which are then uploaded to S3 bucket.

- **Summary of model statistics:**

  By using method model.summary() we get a detailed summary of the model values like AIC, BIC, Lag, coeff etc. Below is the screenshot showing the summary statistics for ARIMA model selected above:

```
                             ARMA Model Results
==============================================================================
Dep. Variable:     NC_HousePriceIndex   No. Observations:                   39
Model:                     ARMA(2, 1)   Log Likelihood                 104.420
Method:                       css-mle   S.D. of innovations              0.014
Date:                Thu, 27 Apr 2017   AIC                           -198.840
Time:                        14:56:22   BIC                           -190.522
Sample:                    01-01-1977   HQIC                          -195.856
                         - 01-01-2015
================================================================================================
                                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------------------------
const                          4.9661      0.811      6.125      0.000       3.377       6.555
ar.L1.NC_HousePriceIndex       1.9167      0.058     33.170      0.000       1.803       2.030
ar.L2.NC_HousePriceIndex      -0.9197      0.058    -15.785      0.000      -1.034      -0.805
ma.L1.NC_HousePriceIndex       0.5639      0.122      4.617      0.000       0.325       0.803
                                    Roots
=============================================================================
                  Real          Imaginary           Modulus         Frequency
-----------------------------------------------------------------------------
AR.1            1.0421           -0.0376j            1.0428           -0.0057
AR.2            1.0421           +0.0376j            1.0428            0.0057
MA.1           -1.7733           +0.0000j            1.7733            0.5000
-----------------------------------------------------------------------------
```

Summary of Metrics

Following are summaries of metrics selected for each state for each indicator

DC-GDP			DC-UnemploymentRate			VA-GDP			VA-UnemploymentRate			NC-HousePriceIndex			NC-TotalPersonalIncome		
AIC	Modelname	(p,q)	AIC	Modelname	(p,q)	AIC	Modelname	(p,q)	AIC	Modelname	(p,q)	AIC	Modelname	(p,q)	AIC	Modelname	(p,q)
-56.576788	MA	(0,1)	-30.471366	MA	(0,1)	-135.3014	AR	(2,0)	-34.632064	AR	(2,0)	-200.52816	AR	(2,0)	-83.388293	AR	(1,0)
-118.623831	AR	(2,0)	-30.898318	AR	(2,0)	-137.3342	MA	(0,1)	-19.46169	MA	(0,1)	-94.097298	MA	(0,3)	61.713245	MA	(0,5)
-124.913404	ARMA	(2,1)	-32.86261	ARMA	(2,1)	-134.1009	ARMA	(2,1)	-33.808703	ARMA	(2,1)	-199.62626	ARMA	(3,3)	-25.869383	ARMA	(2,1)

MD-GDP			MD-UnemploymentRate			WV-GDP			WV-UnemploymentRate			SC-HousePriceIndex			SC-TotalPersonalIncome		
AIC	Modelname	(p,q)	AIC	Modelname	(p,q)	AIC	Modelname	(p,q)	AIC	Modelname	(p,q)	AIC	Modelname	(p,q)	AIC	Modelname	(p,q)
-139.4359	AR	(2,0)	-51.034133	AR	(2,0)	-128.477	AR	(2,0)	-33.314713	AR	(2,0)	-179.68617	AR	(2,0)	-163.83511	AR	(2,0)
-36.9158	MA	(0,1)	-36.207496	MA	(0,1)	-92.6419	MA	(0,1)	-7.671177	MA	(0,1)	-39.947358	MA	(0,4)	-103.227879	MA	(0,5)
-127.776	ARMA	(2,1)	-55.062157	ARMA	(2,1)	-120.2764	ARMA	(2,1)	-32.565627	ARMA	(2,1)	-167.87506	ARMA	(2,4)	-22.659083	ARMA	(2,5)

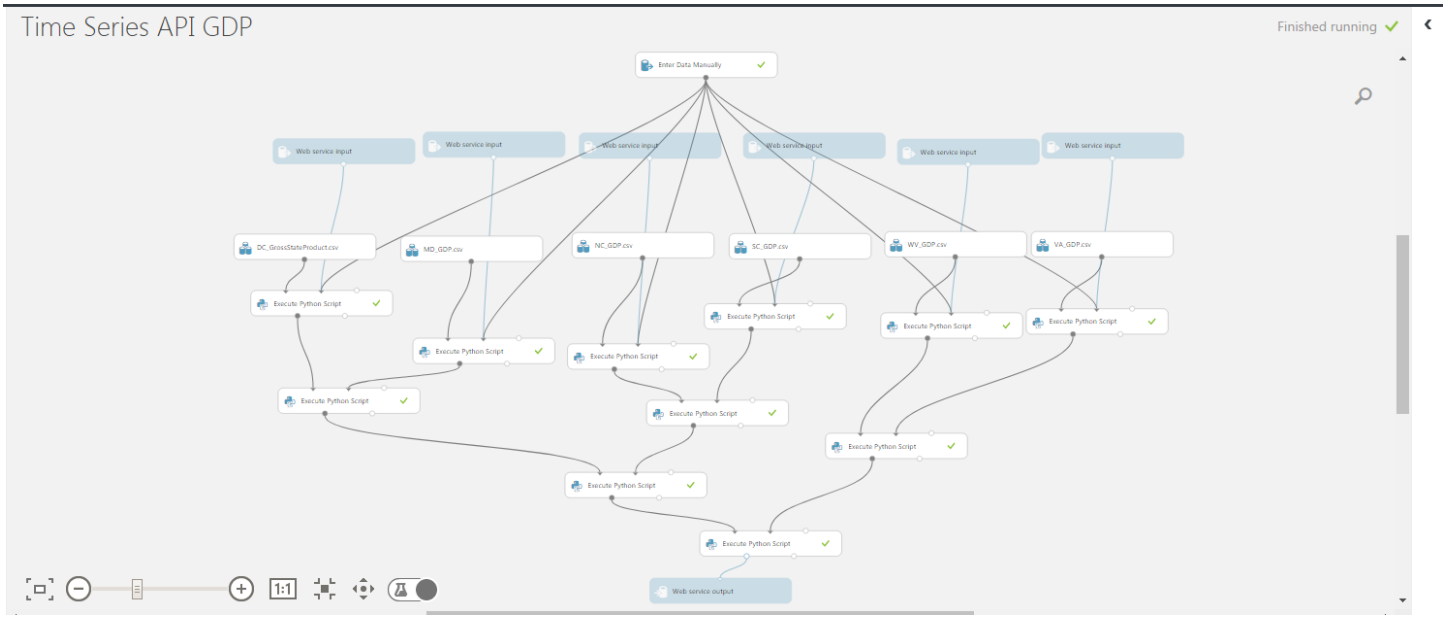
NC-GDP			NC-UnemploymentRate			DC_HousePriceIndex			DC-TotalPersonalIncome			VA-HousePriceIndex			VA-TotalPersonalIncome		
AIC	Modelname	(p,q)	AIC	Modelname	(p,q)	AIC	Modelname	(p,q)	AIC	Modelname	(p,q)	AIC	Modelname	(p,q)	AIC	Modelname	(p,q)
-114.3805	AR	(2,0)	-26.864359	AR	(2,0)	-84.04	MA	(0,1)	25.720695	MA	(0,1)	-139.92139	AR	(2,0)	-169.439467	AR	(2,0)
-124.5891	MA	(0,1)	-9.027029	MA	(0,1)	-115.287726	AR	(2,0)	-141.666221	AR	(2,0)	-19.678847	MA	(0,1)	-84.147116	MA	(0,5)
-122.9501	ARMA	(2,1)	-25.869383	ARMA	(2,1)	-120.44	ARMA	(2,1)	-145.587293	ARMA	(2,1)	-145.59676	ARMA	(2,1)	-120.525084	ARMA	(1,1)

SC-GDP			SC-UnempoymentRate			MD-HousePriceIndex			MD-TotalPersonalIncome			WV-HousePriceIndex			WV-TotalPersonalIncome		
AIC	Modelname	(p,q)	AIC	Modelname	(p,q)	AIC	Modelname	(p,q)	AIC	Modelname	(p,q)	AIC	Modelname	(p,q)	AIC	Modelname	(p,q)
-126.5582	AR	(2,0)	-24.310526	AR	(2,0)	-133.336	AR	(2,0)	-176.540712	AR	(2,0)	-135.95162	AR	(2,0)	-156.58837	AR	(2,0)
-124.8131	MA	(0,1)	-12.266087	MA	(0,1)	22.788	MA	(0,1)	-115.252752	MA	(0,3)	-92.6419	MA	(0,1)	-73.012519	MA	(0,5)
-125.3857	ARMA	(2,1)	-22.659083	ARMA	(2,1)	-146.193	ARMA	(2,1)	-55.062157	ARMA	(2,1)	612.30781	ARMA	(2,1)	-32.565627	ARMA	(2,1)

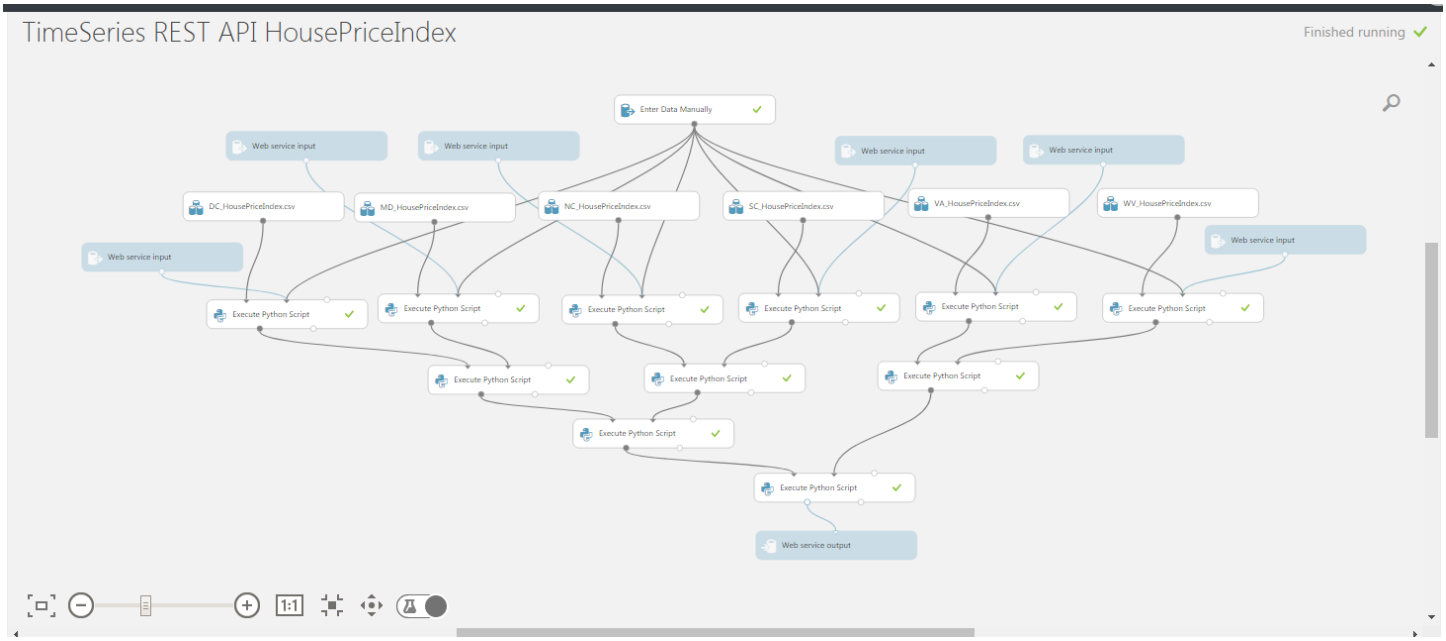
Step 8: Deploy the best model in Azure

After finding the best model using python code through the value of AIC, we have further deployed it in Azure Machine Learning Studio to generate the REST API. Since the AR/ARMA models were not available in the studio, we have used blocks of python code to implement them. Below are the models created for each indicator:

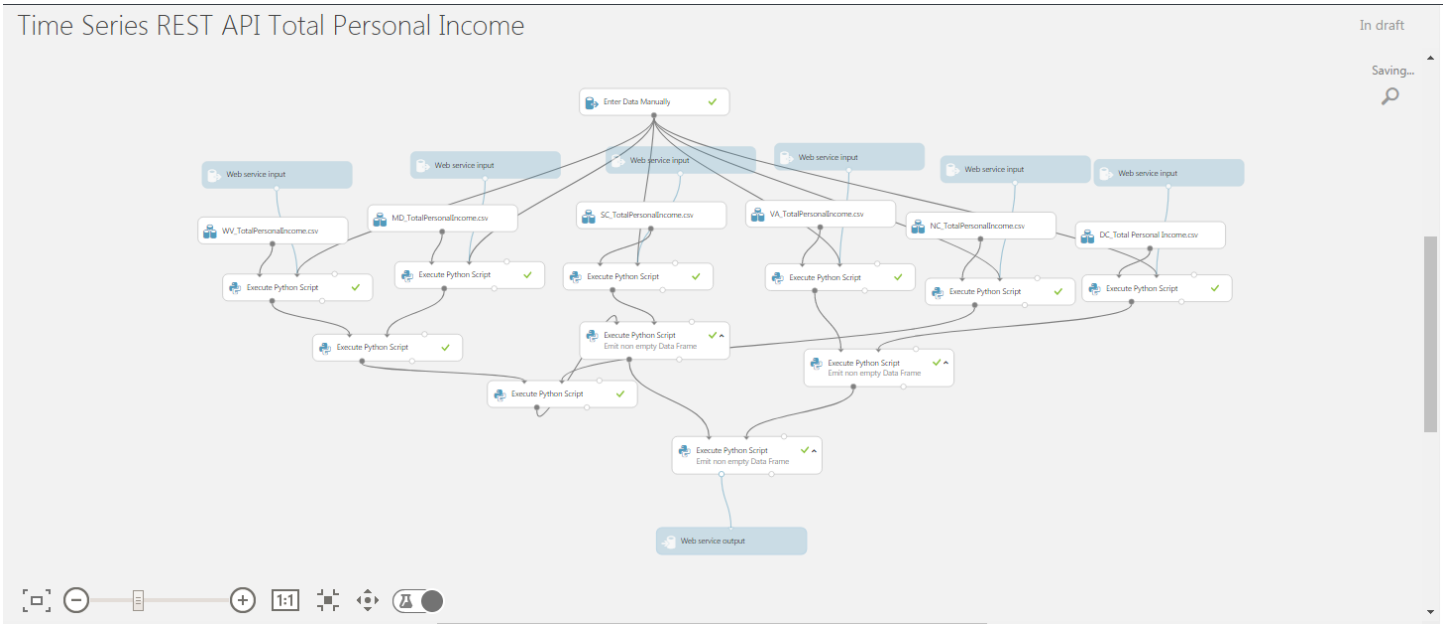
- **Gross State Product (Real):**



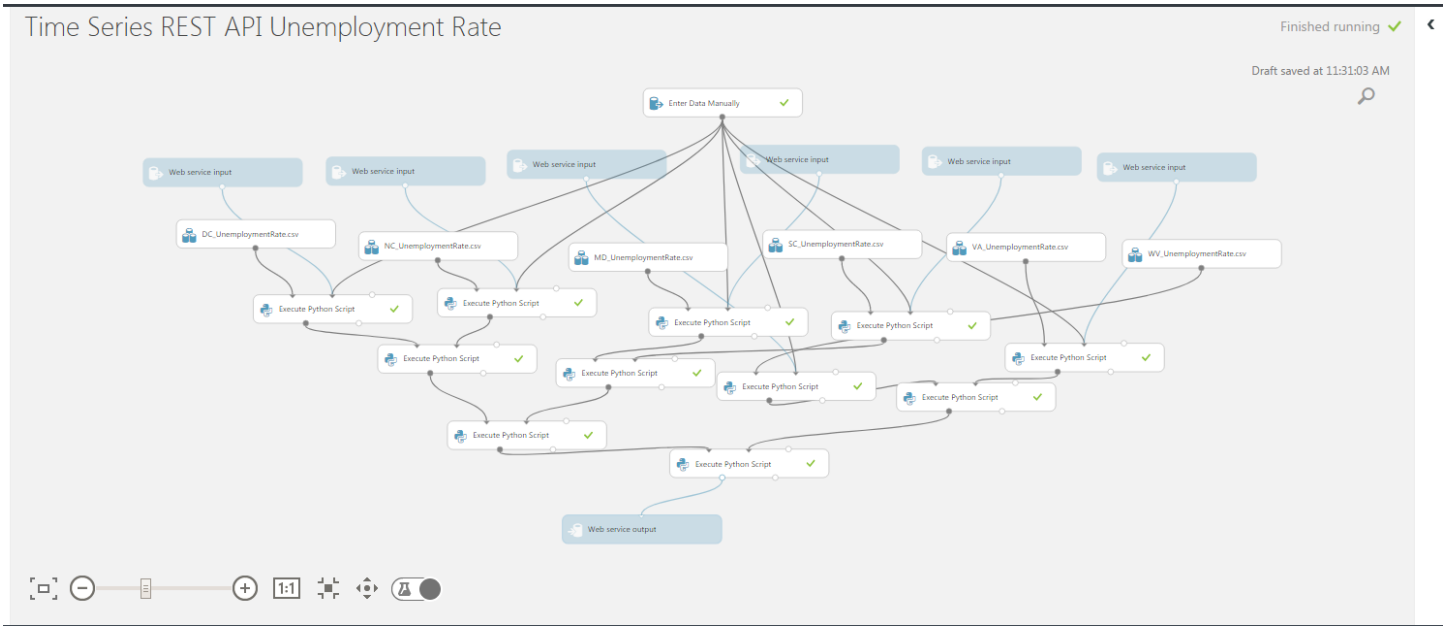
- **House Price Index:**



- **Total Personal Income**



- **Unemployment Rate**



As we can see in above screenshots, we have clubbed all the six states in one azure model for one indicator. Hence, we have attached one Web Service Input for each state available. The user can enter any state name and get the values for the particular indicator. In this process we have generated 4 REST APIs for each indicator.

Below is the sample output of the REST API for Unemployment Rate:

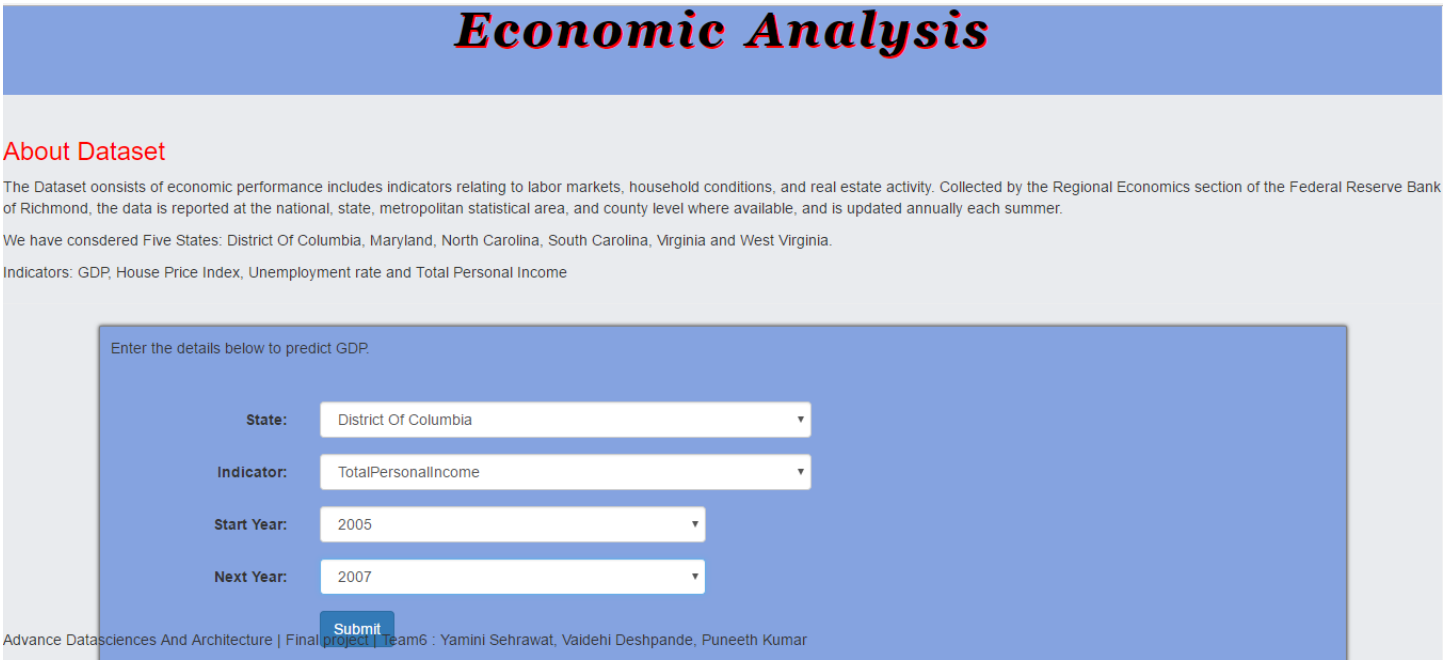
2012	2016
581.75	511.810481255817
561.590678559674	2017
2013	492.87314144779
634.6425	2018
555.380476917136	473.182842307429
2014	2019
693.605	453.333761998784
544.137020169676	2020
2015	433.766850415127
740.7425	
529.222047446614	

Step 10: Web application

Link

http://ec2-35-167-175-188.us-west-2.compute.amazonaws.com/

We have designed a web application which will consume all the 4 REST APIs generated in azure. Following is the first screen that the user will get:



Here the user will select the state, indicator and range of years for which they need the forecast or predictions. Depending on the selection the respective API will be triggered generating the output on next page as shown below:

Below are the forecasted results for TotalPersonalIncome in DC

Year	Actual Value	Forecasted Value
2005	30569.6	30611.8126785936
2006	32945.2	32266.2084876076
2007	35099.9	34010.0150584612

<< Back to Home Page

Step 11: Create Luigi Pipeline and Docker Image

All the steps in the Jupyter notebook python code till selecting the best model are created as individual methods and form tasks in Luigi pipeline.

The main task triggers the underlying tasks and finally following csv files are uploaded to S3 bucket:

- **Original datasets downloaded from FED website**
- **CSV file with Predicted and forecasted values for the selected indicator**

Below are the screenshots which display the tasks created in Luigi:

```
class S3TargetExists(luigi.Target):
    def __init__(self, aws_access_key_id, aws_secret_access_key, bucketName, fileName):
        self.aws_secret_access_key = aws_secret_access_key
        self.aws_access_key_id = aws_access_key_id
        self.bucketName = bucketName
        self.fileName = fileName

    def exists(self):
        client = CreateBotoClient(self.aws_access_key_id,self.aws_secret_access_key).createClient()
        if client:
            try:
                print('============================')
                print('Checking if bucket exists')
                print('============================')
                client.head_bucket(Bucket=self.bucketName)
                print('============================')
                print('Bucket found!')
                print('============================')
            except ClientError as e:
                if e.response['Error']['Code'] == '403':
                    print('====================')
                    print('Failed to make connection to S3')
                    print('====================')
                    return True
                elif e.response['Error']['Code'] == '404':
                    print('======================')
                    print('Bucket Not Found On S3. Exiting program')
                    print('======================')
                    return True
            try:
                client.head_object(Bucket=self.bucketName, Key=self.fileName)
                print('======================')
                print('File Already Exists On S3 Bucket')
                print('======================')
                return True
            except ClientError as e:
                print("Received error:", e)
                print(e.response['Error']['Code'])
                if e.response['Error']['Code'] == '403':
                    print('====================')
                    print('Failed to make connection to S3')
                    print('====================')
                    return True
                elif e.response['Error']['Code'] == '404':
                    print('======================')
                    print('File Not Found On S3 Bucket')
                    print('======================')
                    return False
        else:
            print('======================')
            print('Unable to make a connection to S3')
            print('======================')
```

```
class DownLoadEconomicIndicatorsTask(luigi.Task):
    awsaccesskeyid = luigi.Parameter()
    awssecretaccesskey = luigi.Parameter()
    bucketName = luigi.Parameter()


    def run(self):
        # Logic For Download Data goes here
        print('=========================')
        print('Downloading Loan Data Set to Local')
        print('=========================')
        df = FinalProject_DataDownload.driver()
        self.fileHandle = open('output.txt', 'w')
        self.fileHandle.close()
        print('=========================')
        print('Completed Downloading Loan Data Set to Local')
        print('=========================')
        for each_row in df:
            indicator_name, state_name, new_df = each_row
            self.fileToUpload = state_name + '_' + indicator_name + '.csv'
            self.uploadFileToS3()
        self.fileToUpload = 'output.txt'
        self.uploadFileToS3()

    def output(self):
        return S3TargetExists(self.awsaccesskeyid, self.awssecretaccesskey, self.bucketName, 'output.txt')

    def uploadFileToS3(self):
        client = boto3.client(
            's3',
            aws_access_key_id=self.awsaccesskeyid,
            aws_secret_access_key=self.awssecretaccesskey,
            config=Config(signature_version='s3v4')
        )
        if client:
            print('=========================')
            print('Uploading Data Files to S3 bucket')
            print('=========================')
            client.upload_file(self.fileToUpload, self.bucketName, self.fileToUpload)
```

```
class CreateBotoClient():
    def __init__(self,aws_access_key_id, aws_secret_access_key):
        self.aws_secret_access_key = aws_secret_access_key
        self.aws_access_key_id = aws_access_key_id
    def createClient(self):

        try:
            client = boto3.client(
                's3',
                aws_access_key_id=self.aws_access_key_id,
                aws_secret_access_key=self.aws_secret_access_key,
                config=Config(signature_version='s3v4')
            )
            return client
        except ClientError as e:
            print('=====================')
            print('Failed to make connection to S3')
            print('=====================')
            return
```

The entire code with Luigi pipeline is then ensembled into a docker image. Please refer to following section for docker commands.

Step 12: Run Docker Image

Commands:

1. **Docker hub link:**
   https://hub.docker.com/r/yaminis/finalproject_team6/

   Tag: finalproject

2. **Pull the docker image from docker hub:**

   docker pull yaminis/finalproject_team6:finalproject

3. **Run the Image: Please give your access key, secret key and bucket name to upload the files to Amazon S3.**

   Copy the command below and make necessary changes as instructed:

```
docker run -it -e "accesskey=<enter your accesskey>" -e "secretkey=<enter your secretkey>" -e "bucket=<enter your bucket>" yaminis/finalproject_team6:finalproject
```

Replace accesskey, secretkey and bucket with your credentials.

For ex:

```
docker run -it -e "accesskey=<accesskey>" -e  "secretkey=<secretKey>" -e "bucket= " yaminis/
```

Here, use your credentials in the command, your access key, secret key and bucket name followed by our docker image name.

For environmental variable: accesskey, secretkey and bucket. These are case sensitive, just use as given in the command.
Note: There should be no space in your credentials, otherwise the code will give error message to enter correct credentials. Please do not remove double quotes("accesskey=<enter your accesskey>") also.

4. **After successful execution files are uploaded to S3 bucket as shown in below screenshots:**

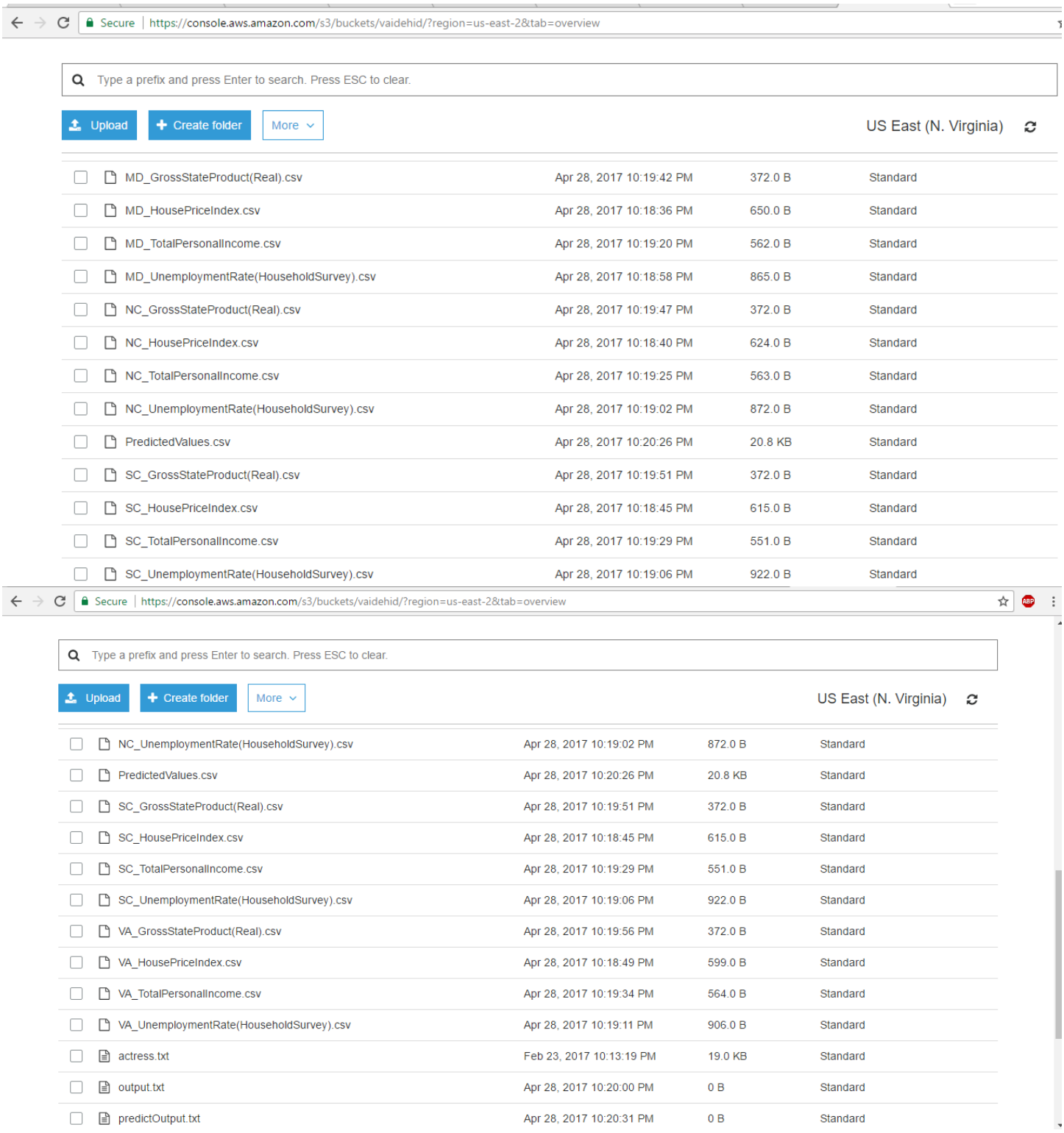


Tableau Link

The Tableau dashboards have been published and the link for the same is:

https://public.tableau.com/shared/4QFCMBDBT?:display_count=yes

Github Repo

https://github.com/Yamini-S/EconomicAnalysis_Team6

Individual Contribution

- **Tasks performed by Puneeth:**
  - **Download data and upload to S3 using Luigi pipeline**
  - **Building Azure models for all states and indicators**
  - **Deploy models and generate REST API**
  - **Create web application**

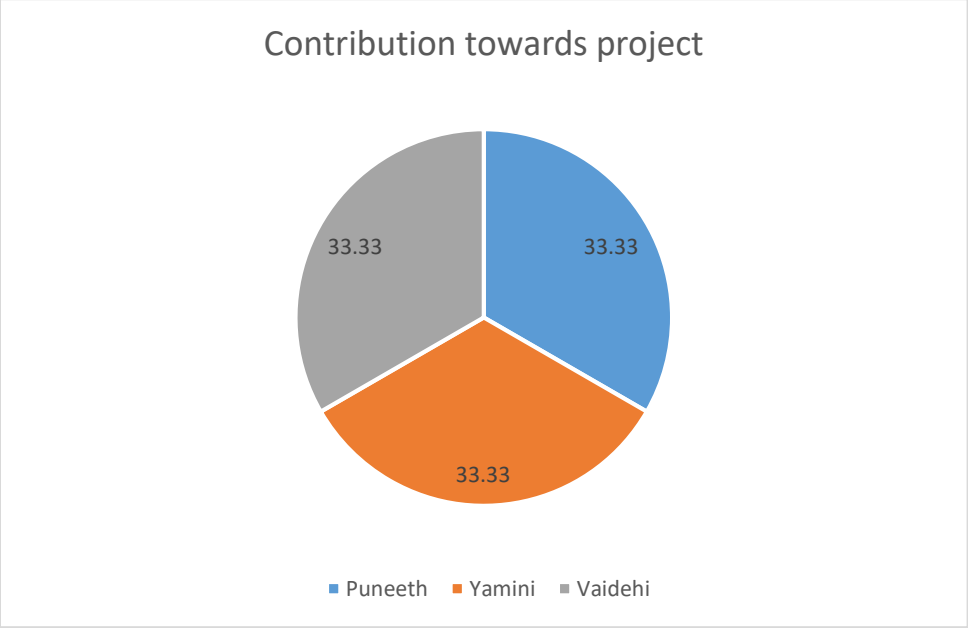
- **Tasks performed by Yamini:**
  - **Univariate Time series Models using ARMA and ARIMA for Indicators: HousePriceIndex and TotalPersonalIncome for all six states in Python Jupyter notebook. Then tuned the Models**
  - **Performed Model Tuning and determine best model**
  - **Update Files preprocessing and Build models methods in Luigi script.**
  - **Front end for Web app.**
  - **Dockerized the tasks and uploaded the output files to Amazon S3**

- **Tasks performed by Vaidehi:**
  - **Summarizations in Tableau and Python**
  - **Untuned univariate models for GDP and Unemployment Rate**
  - **Tuned models with ARIMA and ARMA for GDP and Unemployment Rate in python Jupyter notebook**
  - **Clustering**
  - **Documentation**

References

- https://www.otexts.org/fpp/9/2
- https://www.richmondfed.org/research/regional_economy/reports/regional_profiles#tab-2
- https://cran.r-project.org/web/packages/dtwclust/vignettes/dtwclust.pdf
- http://machinelearningmastery.com/arima-for-time-series-forecasting-with-python/
- https://stats.stackexchange.com/questions/191851/var-forecasting-methodology
- http://statsmodels.sourceforge.net/devel/vector_ar.html#granger-causalit

Pie Chart


Contribution towards project
33.33
33.33
33.33
Puneeth
Yamini
Vaidehi