The George Washington University

Total United States Vehicle Sales Forecast Analysis
Comparing Multiple Forecast Methods

CODE APPENDIX

Fernando A. Zambrano

Dr. Reza Jafari

22 April 2020

APPENDIX ONE: Python Functions

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import seasonal_decompose
from pandas.plotting import register_matplotlib_converters
from sklearn.model_selection import train_test_split
import statsmodels.api as sm
from scipy.stats import chi2
import math
import seaborn as sns
from matplotlib.patches import Patch
from matplotlib.lines import Line2D
from scipy import signal

from statsmodels.tsa.api import ExponentialSmoothing, SimpleExpSmoothing, Holt
# In[28]:

def drift(data,ytrain, ytest, h):
    # T is the length of observations
    T = len(ytrain)
    T = T - 1
    # Last observation
    y_t = ytrain.iloc[-1]
    # First observation
    y0 = ytrain.iloc[0]
    slope = ((y_t - y0) / T)
    print(slope)
    fit = []
    for i in range(len(ytrain)):
        yf = y0 + (i * (slope))
        fit.append(yf)
    yhat = []
```

```
for i in range(1, h + 1):
    yh = y_t + (i * (slope))
    yhat.append(yh)
print(len(yhat))
print(len(ytest))
ytest_l = list(ytest)
error = []
for i in range(0, len(yhat)):
    ei = ytest_l[i] - yhat[i]
    error.append(ei)

fit = np.array(fit)
residual = ytrain - fit
error = np.array(error)

print("Residual Error:")
print("SSE:")
sser = np.sum(np.square(residual))
print(sser)
print("MSE:")
mser = np.mean(np.square(residual))
print(mser)
print("RMSE:")
print(np.sqrt(mser))
print("VAR:")
print(residual.var())
print("Forecast Error:")
print("SSE:")
ssef = np.sum(np.square(error))
print(ssef)
print("MSE:")
msef = np.mean(np.square(error))
print(msef)
print("RMSE:")
print(np.sqrt(msef))
print('VAR:')
print(error.var())
```

```
    plt.plot(data.index[len(ytrain):], yhat, color = "blue", label = "Forecast")
    plt.plot(data.index[len(ytrain):], ytest, color = "orange", label = "Test")
    plt.plot(data.index[:len(ytrain)], ytrain, color = "steelblue", label = "Train")
    plt.plot(data.index[:len(ytrain)], fit, color = "red",label = "Fit")
    plt.title(" US Vehicle Sales Naive Drift Model")
    plt.xlabel("Date")
    plt.ylabel("Unit Sales in Millions")
    plt.legend(loc = "best")
    plt.show();

    return yhat, fit

def SES_df(ytrain,ytest, h, alpha, l0):
    # Calculate initialization on training data to get lT-1 for test
    li1 = ytrain[0] * alpha
    li_train = [l0, li1]
    for i in range(1, len(ytrain)):
        li = alpha * ytrain.iloc[i] + ((1 - alpha) * li_train[i])
        li_train.append(li)
    # repete the same process to get the ses prediciotns for testing
    li2 = li_train[-1]
    li_test = [li2]
    y_hat = []
    for i in range(0, h):
        li = alpha * (len(ytrain) + i)+ ((1 - alpha) * li_test[i])
        li_test.append(li)
        y_hat.append(li)
    # calculate errors
    error = []
    ytest_l = list(ytest)
    for i in range(0, len(y_hat)):
        ei = ytest_l[i] - y_hat[i]
        error.append(ei)

    return y_hat, li_train, np.array(error)
```

```python
def SES(train,test,target):
    fit1 = SimpleExpSmoothing(train).fit()
    fcast1 = fit1.forecast(len(test)).rename(r'$\alpha=%s$'%fit1.model.params['smoothing_level'])

    # Error Stats
    print("Train Mean")
    print(train.mean())
    print("SES Mean")
    print(fcast1.mean())
    SSE_residuals = fit1.sse
    print("SSE of Residuals:")
    print(SSE_residuals)
    fe = test[str(target)] - fcast1
    SSEfe = np.sum(np.square(fe))
    print("SSE of Forecast Errors:")
    print(SSEfe)
    ME = fe.mean()
    print("Mean Forecast Error:")
    print(ME)
    print("MSE:")
    MSE = np.mean(np.square(fe))
    print(MSE)
    print("Variance of Forecast Error:")
    var_e = fe.var()
    print(var_e)

    # plot
    plt.plot(train, label = "Train")
    plt.plot(test,label = "Test")
    plt.title("{} SES Model".format(target))
    plt.ylabel("Units sold in Millions")
    fcast1.plot( color='blue', legend=True)
    fit1.fittedvalues.plot(color='red', label = "Fit")
    plt.legend(loc = "best")
    plt.show();
```

```python
# Non-Stationary
def holtwinters_modelselect(train, test, period):
    import warnings
    warnings.filterwarnings("ignore")

    fit1 = ExponentialSmoothing(train, seasonal_periods=period, trend='add', seasonal='add', damped=True).fit(
        use_boxcox=True)
    fit2 = ExponentialSmoothing(train, seasonal_periods=period, trend='add', seasonal='mul', damped=True).fit(
        use_boxcox=True)
    fit3 = ExponentialSmoothing(train, seasonal_periods=period, trend='mul', seasonal='add', damped=True).fit(
        use_boxcox=True)
    fit4 = ExponentialSmoothing(train, seasonal_periods=period, trend='mul', seasonal='mul', damped=True).fit(
        use_boxcox=True)

    print("Residual SSE:")
    print("1", fit1.sse)
    print("2", fit2.sse)
    print("3", fit3.sse)
    print("4", fit4.sse)

    # fitted values
    ff1 = fit1.fittedvalues
    ff2 = fit2.fittedvalues
    ff3 = fit3.fittedvalues
    ff4 = fit4.fittedvalues

    # forecast errors
    fc1 = fit1.forecast(len(test))
    e1 = test.Sales - fc1
    fc2 = fit2.forecast(len(test))
    e2 = test.Sales - fc2
    fc3 = fit3.forecast(len(test))
    e3 = test.Sales - fc3
    fc4 = fit4.forecast(len(test))
    e4 = test.Sales - fc4

    print("Forecast SSE:")
```

```python
    fitval = [ff1, ff2, ff3, ff4]
    forecasts = [fc1, fc2, fc3, fc4]
    errors = [e1, e2, e3, e4]
    SSE = []

    for i in range(len(errors)):
        sse = np.sum(np.square(errors[i]))
        print(i + 1, sse)
        SSE.append(sse)

    results = pd.DataFrame(index=[r"$\alpha$", r"$\beta$", r"$\phi$", r"$\gamma$", r"$l_0$", "$b_0$", "SSE"])
    params = ['smoothing_level', 'smoothing_slope', 'damping_slope', 'smoothing_seasonal', 'initial_level',
              'initial_slope']
    results["M1:Add_Add"] = [fit1.params[p] for p in params] + [fit1.sse]
    results["M2:Add_Mul"] = [fit2.params[p] for p in params] + [fit2.sse]
    results["M3:Mul_Add"] = [fit3.params[p] for p in params] + [fit3.sse]
    results["M4:Mul_Mul"] = [fit4.params[p] for p in params] + [fit4.sse]

    print(results)

    #plt.plot(train, color="steelblue", label="Train")
    #plt.plot(test, color="black", label="True")
    #plt.ylabel("Units Sold in Millions")
    #fit1.forecast(len(test)).plot(color='blue', legend=True, label="Mod1")
    #fit2.forecast(len(test)).plot(color='green', legend=True, label="Mod2")
    #fit3.forecast(len(test)).plot(color='red', legend=True, label="Mod3")
    #fit4.forecast(len(test)).plot(color='orange', legend=True, label="Mod4")
    #plt.show();


def holtwinters(train, test, target, period, trnd, season):
    import warnings
    warnings.filterwarnings("ignore")

    fit1 = ExponentialSmoothing(train, seasonal_periods=period, trend=trnd, seasonal=season, damped=True).fit(
        use_boxcox=True)
    ffit = fit1.fittedvalues
```

```
residuals = train[target] - ffit
print("Residual SSE:")
sser = fit1.sse
print(sser)
print("MSE:")
print(np.mean(np.square(residuals)))
print("ME:")
print(np.mean(residuals))
print("RMSE:")
print(np.sqrt(np.mean(np.square(residuals))))
print("VAR")
print(residuals.var())

print("Forecast SSE:")
fc1 = fit1.forecast(len(test))
e1 = test[target] - fc1
SSE = np.sum(np.square(e1))
print(SSE)
print("MSE:")
mse = np.mean(np.square(e1))
print(mse)
print("ME:")
print(np.mean(e1))
print("Variance:")
print(e1.var())

plt.plot(train, color="steelblue", label="Train")
plt.title("US Car Sales \nHolt-Winters Forecast")
plt.ylabel("Units Sold in Millions")
plt.plot(test, color="orange", label="True")
f1 = fit1.fittedvalues.plot(color="red", legend=True, label="Fit")
fit1.forecast(len(test)).plot(color='blue', legend=True, label="Forecast")
plt.legend(loc="best")
plt.show();

return fc1, ffit
```

```python
def ADF_Cal(x):
    result = adfuller(x)
    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))

def corrmat(df):
    corr_matrix= df.corr()
    corr_matrix.style.background_gradient().set_precision(2)
    mask = np.array(corr_matrix)
    mask[np.tril_indices_from(mask)] = False
    sns.heatmap(corr_matrix, mask=mask,vmax=1, square=True,annot=True,cmap="Blues")
    b, t = plt.ylim() # discover the values for bottom and top
    b += 0.5 # Add 0.5 to the bottom
    t -= 0.5 # Subtract 0.5 from the top
    plt.ylim(b, t) # update the ylim(bottom, top) values
    plt.title("Correlation Matrix")
    plt.show()

# take either first or second order of a dataset
def diff(data,order):
    if order == 1:
        d1 = np.diff(data)
        return d1
    if order == 2:
        d2 = np.diff(np.diff(data))
        return d2

# Get the cummalitive mean and variance at each time step
def SubSeq(data):
    total_summation = []
    total_sum = 0
    for i in data:
        total_sum += i
```

```
        total_summation.append(total_sum)

    means = []
    for i in range(len(data)):
        ssm = (total_summation[i] / (i + 1))
        means.append(ssm)

    variance = []
    for i in range(2, len(data)):
        ssv = data[:i].var()
        variance.append(ssv)

    return means, variance
# Calculate ACF
def ACF(data, lags):
    # convert input data into a numpy array
    data = np.array(data)
    acf = [1.0]
    yy_bar = data - data.mean()
    # clacualte the total variance for the data set
    total_variance = sum(np.square(yy_bar))
    # print("The total variance for this dataset is: ", total_variance)
    # perform a forloop over the dataset with the desired number of lags
    # range is 1,lags b/c the first iteration calcualtes T1
    for i in range(1, lags):
        # first nparray is removing the last element each iteration
        yy_bar_bottom = yy_bar[:-i]
        # second nparray removes the first element each interation
        yy_bar_top = yy_bar[i:]
        # take the sum of of the product of each nparray each iteration
        yy = sum(yy_bar_top * yy_bar_bottom)
        # divide the sum by total variance and append to resulting acf list
        acf.append(yy / total_variance)
    # print("ACF:",acf)
    return acf
```

```python
# In[29]:



# Plot symetrical ACF
def acf_plot(y, var):
    # y = y.tolist()
    y_rev = y[::-1]
    y_rev.extend(y[1:])
    lb = -(math.floor(len(y_rev) / 2))
    hb = -(lb - 1)
    x = np.array(list(range(lb, hb)))
    figure = plt.stem(x, y_rev, use_line_collection=True)
    plt.xlabel('Lag', fontsize=15)
    plt.ylabel('AC Coefficent', fontsize=15)
    plt.title('ACF of {}'.format(var), fontsize=18)
    plt.show()



# In[30]:



# Construct Matrix X and Matrix Y using the training datasets
def matrix(x, y):
    x_ones = np.ones(len(x))
    X = np.concatenate((x_ones[:, None], x), axis=1)
    print(X.shape)
    Y = np.vstack(y)
    print(Y.shape)
    return X, Y



def B_hat(x, y):
    # print("XT shape:",x.T.shape)
    # print("X shape:",x.shape)
    x_xt = np.mat(x.T) * np.mat(x)
    # print("\nXT * X :\n",x_xt)
    # print("\nXT * X shape:",x_xt.shape)
```

```python
    det_x_xt = np.linalg.det(x_xt)
    # print("\nDeterminent:\n",det_x_xt)
    x_xtt = np.linalg.inv(x_xt)
    # print("\nInverse XT * X:\n",x_xtt)
    # print("\nShape Inverse XT: \n", x_xtt.shape)
    xt_x_xt = np.mat(x_xtt) * np.mat(x.T)
    # print("\nInverse XT * XT:\n", xt_x_xt)
    B = np.mat(xt_x_xt) * np.mat(y)
    print("\nBeta Coefficents: \n")
    for i in range(len(B)):
        print("B{}: ".format(i), float(B[i]))
    return B



def linreg_fit(x, b):
    y_fit = x @ b
    return np.array(y_fit)



def linreg_predict(x, b):
    y_hat = x @ b
    return np.array(y_hat)



# In[31]:



def r2(y, yprime, k):
    yt = y
    yt_bar = yt.mean()
    yh = yprime
    yh_bar = yh.mean()
    yt_var = np.sum((np.square(yt - yt_bar)))
    yh_var = np.sum(np.square((yh - yh_bar)))
    r2 = yh_var / yt_var
    # Calculate adjusted R2
    T = len(yt)
```

```
    ar2 = 1 - ((1 - r2) * (T - 1 / T - k - 1))
    print("R2 is: ", r2)
    print("Adjusted R2: ", ar2)
    return r2, ar2


# In[32]:


# Pearson Correlation
def corr(x, y):
    return np.mean((x - x.mean()) * (y - y.mean())) / (x.std() * y.std())


# In[85]:


def diagnostics(yprime, error, k):
    print("Diagnostics:")

    #re = corr(error, yprime)
    #print("Correlation of Residuals with Fitted Tartget:",re)
    #plt.scatter(yprime, error)
    #plt.title("Residuals vs Fitted Values")
    #plt.ylabel("Residuals")
    #plt.xlabel("Fitted Values")
    #plt.show()

    acf = ACF(error, len(error))
    Q = len(error) * np.sum(np.square(acf[1:]))
    print("Q:", Q)
    DOF = len(error) - (k)
    chi_c = chi2.ppf(1 - 0.01, DOF)
    if Q < chi_c:
        print("Q:", Q, " < ", "CHI Critcal:", chi_c)
        print("Residuals are white")
        print("\nEND")
```

```
    else:
        print(Q, " > ", chi_c)
        print("Residuals are not white")
    acf_plot(acf, "Residuals")
    plt.show()


# In[86]:


# takes in error and k = number of prediction variables
def error_stats(error, k):
    meaner = error.mean()
    print("Mean Error:", meaner)
    mae = np.mean(abs(error))
    print("MAE:", mae)
    mse = np.mean(error ** 2)
    print("MSE:", mse)
    rmse = np.sqrt(np.mean((error ** 2)))
    print("RMSE:", rmse)
    N = len(error)
    den = N - k - 1
    SSE = np.sum((error ** 2))
    print("SSE:", SSE)
    se = np.sqrt(SSE / den)
    print("Standard Error:", se)
    varer = error.var()
    print("Variance Error:", varer)

    return se


# In[93]:


def fit_results(y, yprime, k):
    print("\nFitted Model Resuults:")
```

```python
    print("Number of Obs:", len(y))
    error = y - yprime
    se = error_stats(error, k)
    diagnostics(yprime, error, k)
    return se


# In[94]:


def predict_results(y, yprime, k):
    print("\nPrediction Model Resuults:")
    print("Number of Obs:", len(y))
    error = y - yprime
    acf = ACF(error, len(error))
    se = error_stats(error, k)
    acf_plot(acf, "Forecast Errors")

    return se


# In[95]:


def conint_95(y_hat, se, x):
    x_xt = np.mat(x.T) * np.mat(x)
    det_x_xt = np.linalg.det(x_xt)
    x_xtt = np.linalg.inv(x_xt)

    # Select x*
    cil = []
    for i in range(0, len(x)):
        xstar1 = x[i].reshape(1, len(x[i]))
        xstar2 = xstar1.T
        xx = np.dot(xstar1, xstar2)
        ci = 1.96 * se * np.sqrt(1 + np.mat(xstar1) @ np.mat(x_xtt) @ np.mat(xstar2))
        cil.append(float(ci))
```

```python
    return np.array(cil)



# In[96]:



def linreg(data, target):
    print("\nOLS Linear Regression:")
    train, test = train_test_split(data, test_size=0.2, shuffle=False)

    xtrain_raw = train.drop([str(target)], axis=1)
    xtest_raw = test.drop([str(target)], axis=1)
    ytrain_raw = train[str(target)]
    ytest_raw = test[(str(target))]

    print("Data Shapes:")
    X_train, Y_train = matrix(xtrain_raw, ytrain_raw)
    X_Test, Y_Test = matrix(xtest_raw, ytest_raw)

    b = B_hat(X_train, Y_train)

    yfit = linreg_fit(X_train, b)
    yhat = linreg_predict(X_Test, b)
    # k = # of independent vars, so remove b0
    k = len(b)-1
    print("\nK = # Independent Vars:", k)

    ytrain = (np.array(ytrain_raw).reshape(len(ytrain_raw), 1))
    ytest = (np.array(ytest_raw).reshape(len(ytest_raw), 1))

    ser = fit_results(ytrain, yfit, k)
    model_fit = sm.OLS(Y_train, X_train).fit()
    print(model_fit.summary())
    sef = predict_results(ytest, yhat, k)
    # ci = conint_95(yhat,sef,X_Test)
    smyhat = model_fit.predict()
```

```python
    return yhat, yfit, ytrain, ytest, sef, # ,ci


def GPAC(y,a):
    acf = ACF(y, 30)
    #acf_plot(ACF(y, 15),a)
    # construct den matrix
    den = np.zeros([15, 8])
    for j in range(0, 15):
        for k in range(1, 9):
            den[j][k - 1] = acf[abs(j - k + 1)]

    # GPAC matrix
    phikk = np.zeros([8, 8])
    for j in range(0, 8):
        for k in range(0, 8):
            if k == 0:
                d = den[j][k]
                n = den[j + 1][k]
                phi = n / d
                if abs(d) < 0.00001:
                    phi = 0
                phikk[j][k] = phi
            else:
                d = den[j:j + k + 1, :k + 1]
                # capture the den info for num
                n1 = den[j:j + k + 1, :k]
                # create j+k column
                n2 = np.array(acf[j + 1:j + k + 2])
                num = np.concatenate([n1, n2], axis=1)
                phi = (np.linalg.det(num)) / (np.linalg.det(d))
                dt = (np.linalg.det(d))
                if abs(dt) < 0.00001:
                    phi = 0
                phikk[j][k] = phi
    headers = np.array(list(range(1,9)))
    df = pd.DataFrame(phikk,columns=[headers])
```

```
    print(df)

    # Plot table
    sns.heatmap(phikk, annot=True, vmax=.1, vmin=-.1)
    b, t = plt.ylim()  # discover the values for bottom and top
    b += 0.5  # Add 0.5 to the bottom
    t -= 0.5  # Subtract 0.5 from the top
    plt.ylim(b, t)  # update the ylim(bottom, top) values
    plt.title("GPAC with {} samples".format(a))
    plt.xticks(np.arange(0.5, len(phikk), 1), np.arange(1, 9, 1))
    plt.show()

    return df


def estARMA(data, target, test, na, nb, start, end, alpha):
    print("\nPredictions Based on STATS MODELS Results")

    model = sm.tsa.ARMA(data, (na, nb)).fit(disp=False)
    print(model.summary())
    yfit = model.predict(start=start, end=end)

    yfit = np.array(yfit)
    # yhat = yhat.reshape(len(yhat), 1)
    train = np.array(data[target])
    # train = train.reshape(len(train),1)

    e = train - yfit

    print("Residual Error Stats:")
    me = np.mean(e)
    print("ME:")
    print(me)
    sse = np.sum(np.square(e))
    print("SSE:")
    print(sse)
```

```
mse = np.mean(np.square(e))
print("MSE:")
print(mse)
rmse = np.sqrt(np.mean(e))
print("RMSE:")
print(rmse)
evar = e.var()
print("Variance of Error:")
print(evar)

# ACF Residuals
acf_reside = ACF(e, len(e))
# Plot ACF of Residuals
acf_plot(acf_reside, "Residuals")

Q = len(e) * np.sum(np.square(acf_reside[1:]))

DOF = len(data) - (na + nb)
chi_c = chi2.ppf(1 - alpha, DOF)
if Q < chi_c:
    print("Q:", Q, " < ", "CHI Critcal:", chi_c)
    print("Residuals are white")
else:
    print(Q, " > ", chi_c)
    print("Residuals are not white")
    print("\nTry a Different Order")

# forecast
start_index = 0
end_index = len(test) - 1
forecast = model.predict(start=start_index, end=end_index)
print("\nForecast Errors:")
validation = np.array(test[target])
forecast = np.array(forecast)

ef = validation - forecast
mef = np.mean(ef)
```

```
    print("ME:")
    print(mef)
    ssef = np.sum(np.square(ef))
    print("SSE:")
    print(ssef)
    msef = np.mean(np.square(ef))
    print("MSE:")
    print(msef)
    rmsef = np.sqrt(np.mean(ef))
    print("RMSE:")
    print(rmsef)
    evarf = ef.var()
    print("Variance of Error:")
    print(evarf)


    plt.plot(data.index, data, color="steelblue", label="Train")
    plt.plot(data.index, yfit, color="red", label="Fit")
    plt.plot(test.index, validation, color="steelblue", label="Test")
    plt.plot(test.index, forecast, color="orange", label="Forecast")
    plt.title("\nTest vs Forecast \nARMA({0},{1})".format(na, nb))
    plt.ylabel("Value")
    plt.xlabel("Date")
    plt.legend(loc='best')
    plt.show()

    print("\nEND")

    return forecast, yfit, na, nb


def backtrans(data, yfit, yhat, ytrain, ytest, na, nb):
    yfitt = np.zeros(len(ytrain))
    for i in range(len(ytrain)):
        if i == 0:
            yfitt[i] = ytrain[i]
        else:
```

```
        yfitt[i] = yfitt[i - 1] + -1 * yfit[i]

ypred = np.zeros(len(ytest))
for i in range(len(ytest)):
    if i == 0:
        ypred[i] = ytrain[-1] + yhat[i]
    else:
        ypred[i] = ypred[i - 1] + -1 * yhat[i]

ytrain = ytrain.flatten()

e = ytrain - yfitt

print("Residual Error Stats:")
me = np.mean(e)
print("ME:")
print(me)
sse = np.sum(np.square(e))
print("SSE:")
print(sse)
mse = np.mean(np.square(e))
print("MSE:")
print(mse)
rmse = np.sqrt(np.mean(e))
print("RMSE:")
print(np.sqrt(mse))
evar = e.var()
print("Variance of Error:")
print(evar)

# ACF Residuals
acf_reside = ACF(e, len(e))
# Plot ACF of Residuals
acf_plot(acf_reside, "Reverse Transformation Residuals")

Q = len(e) * np.sum(np.square(acf_reside[1:]))
```

```
DOF = len(data) - (na + nb)
chi_c = chi2.ppf(1 - 0.01, DOF)
if Q < chi_c:
    print("Q:", Q, " < ", "CHI Critcal:", chi_c)
    print("Residuals are white")
else:
    print(Q, " > ", chi_c)
    print("Residuals are not white")
    print("\nTry a Different Order")

ytest = ytest.flatten()

ef = ytest - ypred
mef = np.mean(ef)
print("ME:")
print(mef)
ssef = np.sum(np.square(ef))
print("SSE:")
print(ssef)
msef = np.mean(np.square(ef))
print("MSE:")
print(msef)
rmsef = np.sqrt(np.mean(ef))
print("RMSE:")
print(rmsef)
evarf = ef.var()
print("Variance of Error:")
print(evarf)

# Fit
plt.plot(data.index[:len(ytrain)], ytrain, color="steelblue", label="Test")
plt.plot(data.index[:len(ytrain)], yfitt, color="red", label="Forecast")
plt.plot(data.index[len(ytrain):], ytest, color="steelblue", label="Train")
plt.plot(data.index[len(ytrain):], ypred, color="orange", label="Fit")
plt.title("Backtransforamtion \nARMA({0},{1})".format(na, nb))
plt.ylabel("Value")
plt.xlabel("Date")
```

```
    plt.legend(loc="best")
    plt.show()

    # Forecast
    #plt.plot(data.index[len(ytrain):], ytest, color="steelblue", label="Train")
    #plt.plot(data.index[len(ytrain):], ypred, color="orange", label="Fit")
    #plt.title("Test vs Forecast Backtransforamtion \nARMA({0},{1})".format(na, nb))
    #plt.ylabel("Value")
    #plt.xlabel("Date")
    #plt.legend(loc="best")
    #plt.show()

    return ypred,yfit

def step0(num_true, den_true):
    # print("\nStep 0:")
    # create the true theta parameters vector based on na,nb parameters
    if len(den_true) == 1:
        theta_true = np.vstack((0, np.vstack(num_true[1:])))
    elif len(num_true) == 1:
        theta_true = np.vstack((np.vstack(den_true[1:]), 0))
    else:
        theta_true = np.vstack((np.vstack(den_true[1:]), np.vstack(num_true[1:])))
    theta_true = [aa for bb in theta_true for aa in bb]
    # Initialize theta vector to zero based on the # of paremeters
    # Revmove the 1 at the begining of num and den since they are for dlsim package and NOT ture parameters
    nb = len(num_true) - 1
    na = len(den_true) - 1
    # number of parameters
    NP = na + nb
    # intialize parameters to 0 for the length of na + nb
    theta = np.zeros(NP)
    theta = theta.reshape(len(theta), 1)
    # print("Theta true:", theta_true)
    # print("# na parm:",na)
    # print("# nb parm:",nb)
    # print("Theta shape:",theta.shape)
```

```python
    return theta_true, theta, na, nb


def calsim(y, theta, na, nb):
    # make sure num and den have same size arrays for dlsim calculations

    num = [1] + list(theta[:na].flatten())
    den = [1] + list(theta[na:].flatten())

    if len(num) < len(den):
        num = num + [0 for i in range(len(den) - len(num))]
    elif len(den) < len(num):
        den = den + [0 for i in range(len(num) - len(den))]

    system = (den, num, 1)

    _, e = signal.dlsim(system, y)
    # Change e from tuple to an array
    enew = np.zeros(len(e))
    for i in range(len(e)):
        enew[i] = e[i][0]

    enew = enew.reshape(len(e), 1)
    # print("Shape of e: ",enew.shape)
    # print("Length of e: ",len(enew))

    return enew


# Creates A and g paramters
def step1(y, theta, delta, na, nb):
    # print("Step 1:")

    e = calsim(y, theta, na, nb)
    SSEo = float(np.dot(e.T, e))
    X = np.zeros([na + nb, len(e)])
```

```python
    for i in range(na + nb):
        # print("\n theta {} + delta:".format(i+1))
        theta2 = theta.copy()
        theta2[i] = theta[i] + delta
        e2 = calsim(y, theta2, na, nb)
        X[i] = ((e - e2) / delta).flatten()
        # print("Shape of X:",X.shape)

    # Need to transpose the X matrix for the proper shapes
    X = X.T
    # print("\nShape of X:",X.shape)
    A = X.T @ X
    # print("\nMatrix A:",A)
    # print("Shape of A:", A.shape)
    g = X.T @ e
    # print("\ng:",g)
    # print("Shape of g:", g.shape)

    return A, g, SSEo, e



# Step 2 update theta
def step2(y, theta, A, g, mu, na, nb):
    # print("Step 2:")
    # Create Identity matrix with dimensions NP x NP
    I = np.identity(na + nb)
    # print("\nmu:",mu)
    muI = np.dot(mu, I)
    # Calculate change in theta
    dtheta = np.dot(np.linalg.inv(A + muI), g)
    # print("\nDelta Theta:")
    # print(dtheta)
    # print(dtheta.shape)
    # print("\nTheta old:")
    # print(theta)
    # Add the change in theta to old theta parameters
    thetaNew = np.add(theta, dtheta)
```

```python
    # print("\nTheta New:")
    # print(thetaNew)
    # Calcualte new error with updated theta parameters
    e3 = calsim(y, thetaNew, na, nb)
    # Calculate new SSE term
    SSEn = float(np.dot(e3.T, e3))
    # make sure there are no Nan values, make it really big in case
    if math.isnan(SSEn) == True:
        print("SSE = NAN", SSEn)
        SSEn = 9999
    if np.isinf(SSEn) == True:
        print("SSE = INF", SSEn)
        SSEn = 9999
    # print("\nSSE New:",SSEn)

    return SSEn, dtheta, thetaNew



# step 3 Convergence
def step3(y, thetaNew, dtheta, SSEn, SSEo, A, g, mu, muf, muMax, delta, na, nb, epsilon, theta_true, theta, e):
    MAX_ITER = 100
    iterations = 0
    SSEs = [SSEo]

    while iterations < MAX_ITER:
        # print("\nITERATION:",iterations)
        # print("\nSSE New",SSEn,"vs","SSE Old",SSEo)
        SSEs.append(SSEn)
        # print("Norm 2:",np.linalg.norm(dtheta,2)," vs Epsilon",epsilon)
        if SSEn < SSEo:
            if np.linalg.norm(dtheta, 2) < epsilon:
                # print("Reached Convergance")

                theta = thetaNew
                variance_e = SSEn / (len(y) - (na + nb))
                cov_theta = np.multiply(variance_e, np.linalg.inv(A))
                mean_e = e.mean()
```

```
        print("\nEstimated Results:")
        print("\nTotal # of Iterations:", iterations + 1)
        print("Mean Error:", mean_e)

        THETA, CI, = results(theta, na, nb, cov_theta, variance_e, theta_true)

        iterations = np.arange(0, iterations + 2, 1).tolist()

        plotSSE(SSEs, iterations)

        num, den = pselect(THETA, CI)

        return theta, cov_theta, CI, num, den, mean_e, variance_e

    else:
        # print("Not Converged")
        theta = thetaNew
        # print("mu old:",mu)
        mu /= muf
        # print("mu new",mu)

while SSEn >= SSEo:
    # print("mu old:",mu)
    mu *= muf
    # print("mu new:",mu)
    if mu > muMax:
        print("ERROR")
        return None, None, None

    SSEn, dtheta, thetaNew = step2(y, theta, A, g, mu, na, nb)

iterations += 1

if iterations > MAX_ITER:
    print("ERROR")
```

```
            return None, None, None

        # print("Theta:")
        # print(thetaNew)
        theta = thetaNew

        A, g, SSEo, e = step1(y, theta, delta, na, nb)

        SSEn, dtheta, thetaNew = step2(y, theta, A, g, mu, na, nb)



def results(theta, na, nb, cov_theta, variance_e, theta_true):
    # find the std for theta from covariance matrix
    std_theta = np.zeros(len(theta))
    for i in range(len(std_theta)):
        std_theta[i] = np.sqrt(cov_theta[i][i])

    # create confidence interval matrix
    confi = np.zeros([len(theta), 2])
    for i in range(len(theta)):
        for j in range(len(theta)):
            if j == 0:
                confi[i][j] = theta[i] - (2 * std_theta[i])
            if j == 1:
                confi[i][j] = theta[i] + (2 * std_theta[i])

    # Map CI to theta
    NP = na + nb
    VARS = list(np.zeros(NP))
    for i in range(na):
        VARS[i] = "a" + str(i + 1)
    for i in range(na, NP):
        VARS[i] = "b" + str(i - na + 1)

    VARS = np.array(VARS)
    VARS = VARS.reshape(NP, 1)
```

```python
headers = np.array(["Theta", " Theta - 2xSTD ", " Theta  + 2xSTD  "])
headers = headers.reshape(1, 3)

CI = np.concatenate((VARS, confi), axis=1)
CI = np.concatenate((headers, CI,), axis=0)

THETA = np.concatenate((VARS, theta), axis=1)
COV = np.concatenate((VARS, cov_theta), axis=1)

std_theta = std_theta.reshape(len(std_theta), 1)
STD = np.concatenate((VARS, std_theta), axis=1)

theta_true = np.array(theta_true)
theta_true = theta_true.reshape(len(theta_true), 1)
TRUE = np.concatenate((VARS, theta_true), axis=1)

print("True Theta:")
print(TRUE)
print("Estimated Theta:")
print(THETA)
print("95% Confidence Interval:")
#print(CI)
df = pd.DataFrame(data=CI)
print(df)
print("Covariance Matrix of Theta:")
print(COV)
print("Theta STD:")
print(STD)
print("Root Checks:")
num = [1] + list(theta[na:].flatten())
den = [1] + list(theta[:na].flatten())
print("Num(bn):", num)
print("Den(an):", den)
print("Roots Num:", np.roots(num))
print("Roots Den:", np.roots(den))
print("Variance of Error:")
print(variance_e)
```

```
    print("\nEND")

    df = pd.DataFrame(data=CI)

    return THETA, CI


def plotSSE(SSEs, iterations):
    plt.plot(iterations, SSEs)
    plt.title("Iterations vs SSE Decay")
    plt.xlabel("Iterations")
    plt.ylabel("SSE")
    plt.xticks(list(range(0, len(iterations), 1)));
    plt.show()


def LM(y, num_true, den_true):
    delta = 1e-6
    mu = 0.1
    muf = 10
    muMax = 10e10
    epsilon = 0.001
    np.random.seed(42)

    # Step 0 intialize theta
    theta_true, theta, na, nb = step0(num_true, den_true)
    # step1
    A, g, SSEo, e = step1(y, theta, delta, na, nb)
    # step2
    SSEn, dtheta, thetaNew = step2(y, theta, A, g, mu, na, nb)
    # step3
    theta, cov_theta, CI, num, den, mean_e, var_e = step3(y, thetaNew, dtheta, SSEn, SSEo, A, g, mu, muf, muMax,
delta,
                                                          na, nb, epsilon, theta_true, theta, e)

    return num, den, e, var_e, cov_theta
```

```python
def pselect(theta, ci):
    # find whhich parameters do not include 0 in their CI
    x = []
    for i in range(len(theta)):
        if float(ci[i + 1][1]) < 0 > float(ci[i + 1][2]):
            x.append(ci[i + 1][0])
        if float(ci[i + 1][1]) > 0 < float(ci[i + 1][2]):
            x.append(ci[i + 1][0])
        # collect the relevant theta values
    y = []
    for i in range(len(theta)):
        for j in range(len(x)):
            if theta[i][0] == x[j]:
                y.append(float(theta[i][1]))

    theta1 = np.array(x)
    theta1 = theta1.reshape(len(theta1), 1)
    values = np.array(y)
    values = values.reshape(len(values), 1)
    prediction_vars = np.concatenate((theta1, values), axis=1)
    print("Relevant Parameters:")
    print(prediction_vars)

    # formate parameters for dlsim
    num = [1]
    den = [1]

    for i in range(len(prediction_vars)):
        if (str(prediction_vars[i][0]).startswith("a")) == True:
            den.append(float(prediction_vars[i][1]))
        else:
            num.append(float(prediction_vars[i][1]))

    na = len(den)
    nb = len(num)
```

```
    if na < nb:
        zeros = np.zeros(nb - na)
        den = den + list(zeros)

    print("MA/Num parameters:", num)
    print("AR/Den parameters:", den)

    return num, den
```

APPENDIX TWO: Project Implementation

```
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose
#from pandas.plotting import register_matplotlib_converters
from sklearn.model_selection import train_test_split
```

```
import MMV_Tools as mmv
plt.rcParams.update(plt.rcParams)

#Now set some default parameters.
plt.rcParams["figure.figsize"] = (15,8) #in inches
plt.rcParams['axes.titlesize'] = 16
plt.rcParams['axes.labelsize'] = 10
plt.rcParams['xtick.labelsize']=10
plt.rcParams['ytick.labelsize']=10
plt.rcParams['legend.fontsize']=10
plt.rcParams['lines.linewidth']=2

# In[2]:


os.listdir()


# In[3]:
print("######################################")
print("\nIMPORT DATA")

data = pd.read_csv('forecast_carsales.csv')
df = data.copy()
df.reset_index(level = 0, inplace=True)
df.set_index('DATE', inplace=True)
df= df.drop(["index"],axis = 1)


# In[4]:


df.index = pd.to_datetime(df.index)


# In[5]:
```

```
df.info()


# In[6]:


df.head(529)


# In[7]:
print("####################################")
print("\nData Subsequenceing")

# 3
# Describe the dataset
# a.
# Plot Dependent(Sales) with time
plt.plot(df.index,df.Sales)
plt.xlabel('Time Step (Monthly)', fontsize=15)
plt.ylabel('Unit Sales in Millions', fontsize=15)
plt.title('Total US Vehicle Sales 1976-2020',fontsize=18)
plt.show()


submean, subvar = mmv.SubSeq(df.Sales)
plt.plot(df.index,submean, label = "Mean")
plt.title(" US Vehicle Sales Mean Sub-Sequence")
plt.ylabel("Unit Sales in Millions")
plt.xlabel("Date")
plt.show()

plt.plot(df.index[2:],subvar, label = "Mean")
plt.title(" US Vehicle Sales Variance Sub-Sequence")
plt.ylabel("Unit Sales in Millions")
plt.xlabel("Date")
plt.show()
```

```
submeand, subvard = mmv.SubSeq(np.diff(df.Sales))
plt.plot(df.index[1:],submeand, label = "Mean")
plt.title(" US Vehicle Sales Differenced Mean Sub-Sequence")
plt.ylabel("Unit Sales in Millions")
plt.xlabel("Date")
plt.show()

plt.plot(df.index[3:],subvard, label = "Mean")
plt.title(" US Vehicle Sales Differenced Variance Sub-Sequence")
plt.ylabel("Unit Sales in Millions")
plt.xlabel("Date")
plt.show()

#plt.plot(df.index[3:],subvard, label = "Mean")
#plt.title(" US Vehicle Sales Second - Differenced Variance Sub-Sequence")
#plt.ylabel("Unit Sales in Millions")
#plt.xlabel("Date")
#plt.show()

# In[8]:
print("####################################")
print("\nACF AND STATIONARITY")


mmv.acf_plot(mmv.ACF(df.Sales,len(df)),"Total US Vehicle Sales: 529 Lags")



# In[9]:



mmv.corrmat(df)



# In[10]:
```

```
# e
# Data is clean with no missing data values
df.isnull().sum()


# In[11]:


# f
# Split data into training and testing
# Create training and testing sets

train, test = train_test_split(df, test_size = 0.2, shuffle=False)


# In[12]:


# 4
# Stationarity
# Check inital stationarity
mmv.ADF_Cal(df.Sales)


# In[13]:

# Create first order difference of Sales
mmv.ADF_Cal(np.diff(df.Sales))

# In[15]:

mmv.acf_plot(mmv.ACF(np.diff(df.Sales),60),"First Order Difference US Car Sales")
plt.show

# In[16]:

ytrain = train[["Sales"]]
```

```
ytest = test[["Sales"]]


# In[17]:


print("####################################")
print("\nDECOMPOSTION")

# Review the differences between additive and multiplicative models
result = seasonal_decompose(ytrain, model ='additive',period = 181)
result.plot()
plt.title("Additive Model")
plt.show()



# In[18]:



result = seasonal_decompose(ytrain, model ='multiplicative',period = 181)
result.plot()
plt.show()



# In[19]:




# In[21]:


print("####################################")
print("\nNAIVE MODEL")

sls = df.Sales
trn = sls[:423]
tst = sls[423:]
drift_yhat, drift_fit = mmv.drift(df.Sales,trn,tst,len(tst))

dresults  = pd.DataFrame(index= ["SEE","MSE"])
```

```python
dresults["Residual"] = np.array([3398.41, 8.03 ])
dresults["Forecast"] = np.array([1423.20,13.42])

print(dresults)

# In[22]:
print("####################################")
print("\nHOLT WINTERS MODEL SELECT")

ytrain = train[["Sales"]]
ytest = test[["Sales"]]

mmv.holtwinters_modelselect(ytrain,ytest,181)


# In[23]:

print("####################################")
print("\nHOLT WINTERS")

hw_yhat, hw_fit = mmv.holtwinters(ytrain,ytest,"Sales",181,"add","mul")

hwresults  = pd.DataFrame(index= ["SEE","MSE"])
hwresults["Residual"] = np.array([191.95, 0.45 ])
hwresults["Forecast"] = np.array([216.18,2.04])

print(hwresults)





# In[26]:

print("####################################")
print("\nFEATURE SELECTION")
```

```python
# Multicolinearity
fs = df[["Sales","CPI_UsedV","CapUtil","PayrollNF","UnempRT"]]
fs = fs.copy()
#mmv.corrmat(fs)


# In[27]:


# The only PC to not pass the t-test was Sales-CPI with Payroll controlled
# This makes sense since there is multicolinearity betweent the two
# Will see what OLS regression results show


# In[28]:
print("#####################################")
print("\nOLS 1")

yhat, yfit, ytrain, ytest, sef= mmv.linreg(fs,"Sales")


# In[30]:


plt.plot(df.index[:423],yfit, color = "red", label = "Fit")
plt.plot(df.index[:423],ytrain, color = "steelblue", label = "Train")
plt.plot(df.index[423:],ytest, color = "steelblue", label = "Test")
plt.plot(df.index[423:],yhat, color = "orange", label = "Forecast")
plt.title("US Car Sales \n OLS Regression B = 4")
plt.xlabel("Date")
plt.ylabel("Sales")
plt.legend(loc="best")
plt.show();

# In[32]:
print("#####################################")
print("\nOLS 2")
```

```
fs1 = fs[["CapUtil","PayrollNF","UnempRT","Sales"]]


# In[33]:


lr_yhat, yfit, ytrain, ytest, sef= mmv.linreg(fs1,"Sales")


# In[34]:


plt.plot(df.index[:423],yfit, color = "red", label = "Fit")
plt.plot(df.index[:423],ytrain, color = "steelblue", label = "Train")
plt.plot(df.index[423:],ytest, color = "steelblue", label = "Test")
plt.plot(df.index[423:],yhat, color = "orange", label = "Forecast")
plt.title("US Car Sales \n OLS Regression B = 3")
plt.xlabel("Date")
plt.ylabel("Sales")
plt.legend(loc="best")
plt.show();


# In[38]:

print("######################################")
print("\n# LM Parameter Estimation")

# perform first orde difference on data to feed into GPAC
sales = df[["Sales"]]


# In[39]:
sales["Diff"] = np.array(list(np.diff(df.Sales))+ [0])


# In[40]:
```

```
sdiff = sales[["Diff"]]
sdm = sdiff.Diff.mean()

sdtrain = sdiff[:423]
sdtest = sdiff[423:]


# In[41]:
print("##########################################")
print("\nGPAC")

mmv.GPAC(sdtrain,'Sales Differenced with 423')


# In[ ]:


# LOOKS like an ARMA(4,3)


# In[43]:
print("##########################################")
print("\nLM")

num = [1,0,0]
den = [1,0,0,0]
nume, dene, e, var_e, cov_theta = mmv.LM(sdtrain,num,den)


# In[44]:


# only 2 parameters passed CI
# Try another combination (4,2)
num = [1,0,0]
```

```python
den = [1,0,0,0,0]
nume, dene, e, var_e, cov_theta = mmv.LM(sdtrain,num,den)



# In[ ]:




# In[48]:
print("######################################")
print("\nARMA")

yhat11, yfit11,na11,nb11 = mmv.estARMA(sdtrain,"Diff",sdtest,1,1,0,len(sdtrain)-1,0.01)



# In[49]:

mmv.backtrans(sales,yfit11,yhat11,ytrain,ytest,na11,nb11)



# In[56]:



yhat40,yfit40,na40,nb40 = mmv.estARMA(sdtrain,"Diff",sdtest,4,0,0,len(sdtrain)-1,0.01)



# In[57]:



arma_yhat, arma_yfit = mmv.backtrans(sales,yfit40,yhat40,ytrain,ytest,na40,nb40)



# In[ ]:

print("######################################")
print("\nPREDICTION")

#plt.plot(sales.index[:423], ytrain, color = "steelblue", label = "Train")
```

```
plt.plot(sales.index[423:], ytest, color = "black", label = "Test")
plt.plot(sales.index[423:], drift_yhat, color = "blue", label = "Naive Drift")
plt.plot(sales.index[423:], hw_yhat, color = "orange", label = "Holt-Winters")
plt.plot(sales.index[423:], lr_yhat, color = "red", label = "OLS")
plt.plot(sales.index[423:], arma_yhat, color = "green", label = "ARMA(4,0)")
plt.title("All Model Predictions for US Vehicle Sales")
plt.ylabel("Unit Sales in Millions")
plt.xlabel("Date")
plt.legend(loc = "upper left")
plt.show()
```