APPENDIX

term_project.py

```
import pandas as pd
import helper as helpme
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
import seaborn as sns
from sklearn.model_selection import train_test_split
from statsmodels.tsa.seasonal import seasonal_decompose
import numpy as np
from statsmodels.tsa.holtwinters import ExponentialSmoothing
import arma_estimator as my_arma
from scipy import signal
import statsmodels.api as sm
from scipy.stats import chi2



coffee_data=pd.read_csv("data/COFFEE.csv")
coffee_data.head()

corn_data=pd.read_csv("data/corn.csv")
corn_data.head()

cotton_data=pd.read_csv("data/COTTON.csv")
cotton_data.head()

gold_data=pd.read_csv("data/GOLD.csv")
gold_data.head()

lumber_data=pd.read_csv("data/LUMBER.csv")
lumber_data.head()

oil_data=pd.read_csv("data/OIL.csv")
oil_data.head()

wheat_data=pd.read_csv("data/WHEAT.csv")
wheat_data.head()

snp_data=pd.read_csv("data/S&P500.csv")
snp_data.head()
```

```
oil_data=helpme.datetime_transformer(oil_data,['DATE'])
oil_data=oil_data.rename(columns={"CLOSING PRICE": "oil"})
oil_data.drop(oil_data.columns.difference(['DATE','oil','DATE_year','DATE_month','DATE
_day']), 1, inplace=True)

gold_data=helpme.datetime_transformer(gold_data,['DATE'])
gold_data=gold_data.rename(columns={"CLOSING PRICE": "gold"})
gold_data.drop(gold_data.columns.difference(['DATE','gold','DATE_year','DATE_month','
DATE_day']), 1, inplace=True)

#join and drop unnecessary
all_data = pd.merge(oil_data, gold_data,  how='inner',
left_on=['DATE_year','DATE_month','DATE_day'], right_on =
['DATE_year','DATE_month','DATE_day'])
all_data.drop(all_data.columns.difference(['DATE','oil','gold','DATE_year','DATE_month','
DATE_day']), 1, inplace=True)

coffee_data=helpme.datetime_transformer(coffee_data,['Date'])
coffee_data=coffee_data.rename(columns={"Price": "coffee"})
coffee_data.drop(coffee_data.columns.difference(['Date','coffee','Date_year','Date_month'
,'Date_day']), 1, inplace=True)

#join and drop unnecessary
all_data = pd.merge(all_data, coffee_data,  how='inner',
left_on=['DATE_year','DATE_month','DATE_day'], right_on =
['Date_year','Date_month','Date_day'])
all_data.drop(all_data.columns.difference(['DATE','oil','gold','coffee','DATE_year','DATE_
month','DATE_day']), 1, inplace=True)

corn_data=helpme.datetime_transformer(corn_data,['Date'])
corn_data=corn_data.rename(columns={"Price": "corn"})
corn_data.drop(corn_data.columns.difference(['Date','corn','Date_year','Date_month','Dat
e_day']), 1, inplace=True)

#join and drop unnecessary
all_data = pd.merge(all_data, corn_data,  how='inner',
left_on=['DATE_year','DATE_month','DATE_day'], right_on =
['Date_year','Date_month','Date_day'])
all_data.drop(all_data.columns.difference(['DATE','oil','gold','coffee','corn','DATE_year','D
ATE_month','DATE_day']), 1, inplace=True)

cotton_data=helpme.datetime_transformer(cotton_data,['Date'])
cotton_data=cotton_data.rename(columns={"Price": "cotton"})
```

```
cotton_data.drop(cotton_data.columns.difference(['Date','cotton','Date_year','Date_month
','Date_day']), 1, inplace=True)

#join and drop unnecessary
all_data = pd.merge(all_data, cotton_data,  how='inner',
left_on=['DATE_year','DATE_month','DATE_day'], right_on =
['Date_year','Date_month','Date_day'])
all_data.drop(all_data.columns.difference(['DATE','oil','gold','coffee','corn','cotton','DATE
_year','DATE_month','DATE_day']), 1, inplace=True)

lumber_data=helpme.datetime_transformer(lumber_data,['Date'])
lumber_data=lumber_data.rename(columns={"Price": "lumber"})
lumber_data.drop(lumber_data.columns.difference(['Date','lumber','Date_year','Date_mon
th','Date_day']), 1, inplace=True)

#join and drop unnecessary
all_data = pd.merge(all_data, lumber_data,  how='inner',
left_on=['DATE_year','DATE_month','DATE_day'], right_on =
['Date_year','Date_month','Date_day'])
all_data.drop(all_data.columns.difference(['DATE','oil','gold','coffee','corn','cotton','lumbe
r','DATE_year','DATE_month','DATE_day']), 1, inplace=True)

wheat_data=helpme.datetime_transformer(wheat_data,['Date'])
wheat_data=wheat_data.rename(columns={"Price": "wheat"})
wheat_data.drop(wheat_data.columns.difference(['Date','wheat','Date_year','Date_month',
'Date_day']), 1, inplace=True)

#join and drop unnecessary
all_data = pd.merge(all_data, wheat_data,  how='inner',
left_on=['DATE_year','DATE_month','DATE_day'], right_on =
['Date_year','Date_month','Date_day'])
all_data.drop(all_data.columns.difference(['DATE','oil','gold','coffee','corn','cotton','lumbe
r','wheat','DATE_year','DATE_month','DATE_day']), 1, inplace=True)

snp_data=helpme.datetime_transformer(snp_data,['Date'])
snp_data=snp_data.rename(columns={"Close": "snp"})
snp_data.drop(snp_data.columns.difference(['Date','snp','Date_year','Date_month','Date_
day']), 1, inplace=True)

#join and drop unnecessary
all_data = pd.merge(all_data, snp_data,  how='inner',
left_on=['DATE_year','DATE_month','DATE_day'], right_on =
['Date_year','Date_month','Date_day'])
```

```
all_data.drop(all_data.columns.difference(['DATE','oil','gold','coffee','corn','cotton','lumbe
r','wheat','snp','DATE_year','DATE_month','DATE_day']), 1, inplace=True)

all_data

#prep
#all_data.to_csv(r'data/all_data.csv', index = False)

#plottiong dependent variable against time
plt.plot(all_data["snp"])
plt.title("5 Years data of SNP 500")
plt.xlabel("Days")
plt.ylabel("SNP500 Closing Index")
plt.show()

#adf of the dependent variable
dependent_variable=all_data["snp"]
result = adfuller(dependent_variable)
print('ADF for dependent variable:')
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))

# p-value is > 0.05:
# Fail to reject the null hypothesis (H0),
# the data does have a unit root and is definetely not stationary.

#plotting acf of dependent variable
helpme.acf_plotter(dependent_variable,len(dependent_variable))

corr_d={'coffee': all_data['coffee'],
        'corn': all_data['corn'],
        'cotton': all_data['cotton'],
                'gold': all_data['gold'],
                'lumber': all_data['lumber'],
                'oil': all_data['oil'],
                'wheat': all_data['wheat'],
                'snp': all_data['snp']}
df=pd.DataFrame(data=corr_d)
sns.heatmap(df.corr(),annot=True)
```

```
#checking for nans
helpme.nan_checker(all_data)



#splitting training and testing
Xd={'coffee': all_data['coffee'],
   'corn': all_data['corn'],
   'cotton': all_data['cotton'],
         'gold': all_data['gold'],
         'lumber': all_data['lumber'],
         'oil': all_data['oil'],
         'wheat': all_data['wheat']}
X_df=pd.DataFrame(data=Xd)

#X_df

Yd={'snp': all_data['snp']}
Y_df=pd.DataFrame(data=Yd)

#setting index for holt


#Y_df
# 80% train, 20% test
x_train, x_test, y_train, y_test = train_test_split(X_df, Y_df, train_size = 0.8, test_size =
0.2, shuffle = False)

#x_train
#x_test
#y_train['snp']
#y_test

# making dependent variable stationary using first differencing method
Y = y_train.values
diff = []
for i in range(1, len(Y)):
         value = Y[i] - Y[i - 1]
         diff.append(value)

#before detrending training portion of dependent variable
plt.plot(Y)
```

```
plt.title("Before detrending")
plt.xlabel("Days")
plt.ylabel("SNP500 Closing Index 80% Train")
plt.show()

#after detrending training portion of dependent variable
plt.plot(diff)
plt.title("After detrending using first differencing")
plt.xlabel("Days")
plt.ylabel("SNP500 Closing Index 80% Train")
plt.show()


#adf of the differenced traiing dependent variable
diff_train_dependent_variable=diff
result = adfuller(diff_train_dependent_variable)
print('ADF for differenced traiing dependent variable:')
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))

# p-value is less than  0.05:
# Reject the null hypothesis (H0),
# the data does not have a unit root and is definetely stationary.

# multiplicative decomposition
y_arr=np.array(Y_df)
result=seasonal_decompose(y_arr,model='multiplicative', freq=1)
result.plot()
plt.show()

# additive decomposition

result2=seasonal_decompose(y_arr,model='additive', freq=1)
result2.plot()
plt.show()

#hotwinter model: <start>

days=range(1,len(all_data)+1)
holt_Y={'day': days,
```

```
                    'snp': all_data['snp']}
holt_df=pd.DataFrame(data=holt_Y)

#holt_df['day']=pd.to_datetime(holt_df['day'])
holt_df.set_index('day',inplace=True)
holt_df.index.freq='M'

holt_train, holt_test = holt_df.iloc[:996,0], holt_df.iloc[995:,0]

holt_model = ExponentialSmoothing(holt_train, trend='add', damped=False,
seasonal='add', seasonal_periods=2).fit()
holt_predictions = holt_model.predict(start=holt_test.index[0], end=holt_test.index[-1])

plt.plot(holt_train.index, holt_train, label="train")
plt.plot(holt_test.index, holt_test, label="test")
plt.plot(holt_predictions.index, holt_predictions, label="predictions")
plt.show()

#holtwinter model: <end>

#holts accuracy <start>
forecast_error=holt_test-holt_predictions
#forecast_error

holt_forecast_error_d = {'Y': holt_test,
    'Y`': holt_predictions,
    'e=Y-Y`': forecast_error}
holt_forecast_error_df = pd.DataFrame(data=holt_forecast_error_d)
holt_forecast_error_df

RMSE = np.sqrt(np.mean(forecast_error**2))
print("The Holt Root Mean Square of Forecast Error is "+str(RMSE))

mean_forecast_error=np.mean(forecast_error)
print("The Holts Mean of Forecast Error is "+str(mean_forecast_error))

def standard_error(forecast_error,num_of_predictors):
   return np.sqrt(np.sum(forecast_error**2)/(len(forecast_error)-num_of_predictors-1))

se=standard_error(forecast_error,1)
print("The standard error using holt is "+str(se))

R_Squared=((np.corrcoef(holt_test, holt_predictions)[0, 1])**2)
```

```
print("The R^2 using holt is "+str(R_Squared))


T=len(holt_test) #size of sample
k=1 #number of predictors
Adjusted_R_Squared=(1-((1-R_Squared)*((T-1)/(T-k-1))))
print("The Adj R^2 using holt is "+str(Adjusted_R_Squared))


#holst accuracy end

# multiple linear regression start

X_M = x_train[["coffee", "corn", "cotton", "gold", "lumber", "oil", "wheat"]]
Y_M = y_train["snp"]
X_M=np.mat(X_M)
Y_M=np.mat(Y_M)

lin_reg_model = sm.OLS(Y_M,X_M)

results = lin_reg_model.fit()
print (results.params)

test_pred_ols=results.predict(x_test)
test_pred_ols

print(results.summary())

# multiple linear regression end



ry=my_arma.acf_values(diff_train_dependent_variable,100)
ax=my_arma.cal_GPAC(ry,25,25)
x_axis_labels=range(1,25)
plt.figure(figsize = (25,25))
hmap = sns.heatmap(ax, xticklabels=x_axis_labels, annot=True, linewidths=.5, vmin=-0.5,
vmax=0.5, cmap="YlGnBu")
hmap


"""
print("Estimated parameters for ARMA(15,16) :")
```

```
my_arma.levenburgMarquardt(diff_train_dependent_variable,15,16,120)

def arma_yhat(na,theta,sampleSize):
        T=sampleSize
        mu, sigma =0, 1
        e = np.random.normal(mu, sigma, T)
        num=[1]
        den=[1]
        den=np.concatenate((den,theta[0:na]))
        num=np.concatenate((num,theta[na:]))

        if len(num)<len(den):
                z=np.zeros(len(den)-len(num))
                num=np.concatenate((num,z),axis=None)
        elif len(num)>len(den):
                z=np.zeros(len(num)-len(den))
                den=np.concatenate((den,z),axis=None)
        system = (num,den,1)
        t_in=np.arange(0,T)
        t_out, y = signal.dlsim(system,e,t=t_in)
        return y

theta=[-0.0451566 , -0.00802712,  0.18580267,  0.39536419, -0.18220256,
       0.16160088, -0.02275384,  0.19785412,  0.0784035 ,  0.00378659,
       0.22446751,  0.10849447,  0.23731303,  0.03900169,  0.06361364,
      -0.04673583, -0.05854274,  0.23820085,  0.34755728, -0.22201763,
       0.15678795,  0.04174483,  0.06039233,  0.08552093, -0.01700574,
       0.25083316,  0.05473999,  0.25375103, -0.0786032 ,  0.033727  ,
       0.04586758]
my_arma_yhat=arma_yhat(15,theta,len(diff_train_dependent_variable))

print("Using parameters from my ARMA:")
plt.figure()
plt.plot(diff_train_dependent_variable,'r',label="Actual")
plt.plot(my_arma_yhat,'b',label="implemented_prediction")
plt.xlabel("Samples")
plt.ylabel("Magnitude")
plt.legend()
plt.title("ACTUAL vs Implemented ARMA prediction")
plt.show()

"""
#using stats model for ARMA
```

```
import statsmodels.api as sm

na=15
nb=16
arma_model=sm.tsa.ARMA(diff_train_dependent_variable,(na,nb)).fit(trend='nc',disp=0)
for i in range (na):
        print("The AR coeff a{}".format(i), "is:",arma_model.params[i])

for i in range (nb):
        print("The MA coeff b{}".format(i), "is:",arma_model.params[i+na])

print(arma_model.summary())

# prediction
arma_model_y_hat=arma_model.predict(start=1, end=len(diff_train_dependent_variable))
# residual testing and chisquare test

plt.figure()
plt.plot(diff_train_dependent_variable,'r',label="Actual")
plt.plot(arma_model_y_hat,'b',label="prediction")
plt.xlabel("Samples")
plt.ylabel("Magnitude")
plt.legend()
plt.title("ACTUAL vs ARMA prediction")
plt.show()

#residual testing and chi square test of ARMA(15,16)
"""lags=365
e=diff_train_dependent_variable-arma_model_y_hat
re=helpme.autocorrelation_cal(e,lags)
Q=len(diff_train_dependent_variable)*np.sum(np.square(re[lags:]))
DOF=lags-na-nb
alfa=0.01
chi_critical=chi2.ppf(1-alfa,DOF)
if Q<chi_critical:
        print("the residual is white")
else:
        print("the residual is not white")
lbvalue,pvalue=sm.stats.acorr_ljungbox(e,lags=[lags])
print(lbvalue)
print(pvalue)"""
```

```
#



na2=16
nb2=23
arma_model2=sm.tsa.ARMA(diff_train_dependent_variable,(na2,nb2)).fit(trend='nc',disp=
0)
for i in range (na2):
        print("The AR coeff a{}".format(i), "is:",arma_model2.params[i])

for i in range (nb2):
        print("The MA coeff b{}".format(i), "is:",arma_model2.params[i+na2])

print(arma_model2.summary())

# prediction
arma_model_y_hat2=arma_model2.predict(start=1,
end=len(diff_train_dependent_variable))
# residual testing and chisquare test

plt.figure()
plt.plot(diff_train_dependent_variable,'r',label="Actual")
plt.plot(arma_model_y_hat2,'b',label="prediction")
plt.xlabel("Samples")
plt.ylabel("Magnitude")
plt.legend()
plt.title("ACTUAL vs ARMA prediction")
plt.show()

arma_model_forecast=arma_model.forecast(steps=len(y_test), exog=None, alpha=0.05)

arma_model2_forecast=arma_model2.forecast(steps=len(y_test), exog=None,
alpha=0.05)

plt.plot(arma_model_forecast[0])
plt.show()

plt.plot(arma_model2_forecast[0])
```

```
plt.show()




def reverse_transform_for_differencing(original_input_list,
differenced_df_list_with_predicted_values):
    """ returns transformed values for predicted values only"""
    last_index = len(original_input_list) - 1
    prediction_range = len(differenced_df_list_with_predicted_values) -
len(original_input_list) + 1

    back_transformed = []
    predicted_sum = 0
    for i in range(prediction_range):
        predicted_sum += differenced_df_list_with_predicted_values[last_index + i]
        predicted_value = original_input_list[last_index] + predicted_sum
        back_transformed.append(predicted_value)

    return back_transformed

#arma (15,16)
diff_forecast1=diff+arma_model_forecast[0].tolist()
rev_pred=reverse_transform_for_differencing(Y,diff_forecast1)
rev_pred_arr=np.array(rev_pred)

plt.plot(Y_df.values, label="actual")
plt.plot(np.concatenate((Y,rev_pred_arr)), label="prediction")
#plt.plot(Y, label="actual")
plt.show()


#arma (16,23)
diff_forecast2=diff+arma_model2_forecast[0].tolist()
rev_pred2=reverse_transform_for_differencing(Y,diff_forecast2)
rev_pred_arr2=np.array(rev_pred2)

plt.plot(Y_df.values, label="actual")
plt.plot(np.concatenate((Y,rev_pred_arr2)), label="prediction")
#plt.plot(Y, label="actual")
plt.show()
```

```
# arma (15,16) forecast errors

arma1_forecast_error=y_test.values-rev_pred_arr
#forecast_error

"""holt_forecast_error_d = {'Y': holt_test,
    'Y`': holt_predictions,
    'e=Y-Y`': forecast_error}
holt_forecast_error_df = pd.DataFrame(data=holt_forecast_error_d)
holt_forecast_error_df"""

RMSE_arma1 = np.sqrt(np.mean(arma1_forecast_error**2))
print("The ARMA(15,16) Root Mean Square of Forecast Error is "+str(RMSE_arma1))

arma1_mean_forecast_error=np.mean(arma1_forecast_error)
print("The ARMA(15,16) Mean of Forecast Error is "+str(arma1_mean_forecast_error))


se_arma1=standard_error(arma1_forecast_error,1)
print("The standard error using ARMA(15,16) is "+str(se_arma1))

"""arma1_R_Squared=((np.corrcoef(y_test.values, rev_pred_arr)[0, 1])**2)
print("The R^2 using ARMA(15,16) is "+str(arma1_R_Squared))

T=len(y_test) #size of sample
k=1 #number of predictors
arma1_Adjusted_R_Squared=(1-((1-R_Squared)*((T-1)/(T-k-1))))
print("The Adj R^2 using ARMA(15,16) is "+str(arma1_Adjusted_R_Squared))
"""
# arma (16,23) forecast errors

arma2_forecast_error=y_test.values-rev_pred_arr2

RMSE_arma2 = np.sqrt(np.mean(arma2_forecast_error**2))
print("The ARMA(16,23) Root Mean Square of Forecast Error is "+str(RMSE_arma2))

arma2_mean_forecast_error=np.mean(arma2_forecast_error)
print("The ARMA(16,23) Mean of Forecast Error is "+str(arma2_mean_forecast_error))
```

```
se_arma2=standard_error(arma2_forecast_error,1)
print("The standard error using ARMA(16,23) is "+str(se_arma2))
```

ols.py

```python
import pandas as pd
import helper as helpme
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
import seaborn as sns
from sklearn.model_selection import train_test_split
from statsmodels.tsa.seasonal import seasonal_decompose
import numpy as np
from statsmodels.tsa.holtwinters import ExponentialSmoothing
import arma_estimator as my_arma
from scipy import signal
import statsmodels.api as sm


all_data=pd.read_csv("data/all_data_numeric.csv")
all_data.head()

train, test = train_test_split(all_data, train_size = 0.8, test_size = 0.2, shuffle = False)



X = train[["coffee", "corn", "cotton", "gold", "lumber", "oil", "wheat"]]
X = sm.add_constant(X)
Y=  train["snp"]

x_test=test[["coffee", "corn", "cotton", "gold", "lumber", "oil", "wheat"]]
x_test = sm.add_constant(x_test)

lin_reg_model = sm.OLS(Y,X.astype(float))

results = lin_reg_model.fit()
print (results.params)

test_pred_ols=results.predict(x_test)
test_pred_ols

print(results.summary())

plt.plot(np.concatenate((Y,test["snp"])), label="actual")
plt.plot(np.concatenate((Y,test_pred_ols)), label="prediction")
#plt.plot(Y, label="actual")
```

```
plt.show()


forecast_error=test["snp"]-test_pred_ols

RMSE = np.sqrt(np.mean(forecast_error**2))
print("The Root Mean Square of Forecast Error using OLS is "+str(RMSE))

mean_forecast_error=np.mean(forecast_error)
print("The Mean of Forecast Error using is "+str(mean_forecast_error))

def standard_error(forecast_error,num_of_predictors):
   return np.sqrt(np.sum(forecast_error**2)/(len(forecast_error)-num_of_predictors-1))

se=standard_error(forecast_error,1)
print("The standard error using OLS is "+str(se))
```

helper.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

def datetime_transformer(df, datetime_vars):
    """
    The datetime transformer

    Parameters
    ----------
    df : dataframe
    datetime_vars : the datetime variables

    Returns
    ----------
    The dataframe where datetime_vars are transformed into the following 3 datetime
types:
    year, month, and day
    """

    # The dictionary with key as datetime type and value as datetime type operator
    dict_ = {'year'   : lambda x : x.dt.year,
             'month'  : lambda x : x.dt.month,
             'day'    : lambda x : x.dt.day}

    # Make a copy of df
    df_datetime = df.copy(deep=True)

    # For each variable in datetime_vars
    for var in datetime_vars:
        # Cast the variable to datetime
        df_datetime[var] = pd.to_datetime(df_datetime[var])

        # For each item (datetime_type and datetime_type_operator) in dict_
        for datetime_type, datetime_type_operator in dict_.items():
            # Add a new variable to df_datetime where:
            # the variable's name is var + '_' + datetime_type
            # the variable's values are the ones obtained by datetime_type_operator
            df_datetime[var + '_' + datetime_type] = datetime_type_operator(df_datetime[var])

    # Remove datetime_vars from df_datetime
```

```
    # df_datetime = df_datetime.drop(columns=datetime_vars)

    return df_datetime

def autocorrelation_cal(y,k):
    T=len(y)
    mean_y=np.mean(y)

    numerator=0
    denominator=0
    T_k=0

    for t in range(0,T):
        denominator=denominator+(np.square(y[t]-mean_y))

    for t in range(k,T):
        numerator=numerator+((y[t]-mean_y)*(y[t-k]-mean_y))

    T_k=numerator/denominator
    return T_k

def acf_plotter(y,l):
    #acf over y with 100 samples
    lags=[]
    autoCorr=[]
    max_lag=l
    for i in range(0,max_lag):
        lags.append(i)
        autoCorr.append(autocorrelation_cal(y,i))

    #making symmetrical acf plot about y axis
    autoCorr_copy=autoCorr[1:].copy()
    autoCorr_copy.reverse()
    autoCorr_copy=np.concatenate((autoCorr_copy,autoCorr),axis=None)

    lags_rev=lags[1:].copy()
    lags_rev.reverse()
    lags_rev=np.negative(lags_rev)
    lags_rev=np.concatenate((lags_rev,lags),axis=None)

    #plotting acf over y
    plt.stem(lags_rev,autoCorr_copy,use_line_collection=True)
    plt.title("ACF Plot for Sample size {} with {} lags".format(len(y),l))
```

```
    plt.show()


def nan_checker(df):

    # Get the variables with NaN, their proportion of NaN and dtype
    df_nan = pd.DataFrame([[var, df[var].isna().sum() / df.shape[0], df[var].dtype]
                           for var in df.columns if df[var].isna().sum() > 0],
                          columns=['var', 'proportion', 'dtype'])

    # Sort df_nan in accending order of the proportion of NaN
    df_nan = df_nan.sort_values(by='proportion', ascending=False)

    return df_nan
```

arma_estimator.py

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from scipy import signal
import copy

def cal_GPAC(acf_values,j_max,k_max):
    gpac_ndarray=np.zeros((j_max,k_max-1))
    for k in range(1,k_max):
        for j in range(0,j_max):
            #form the denominator matrix (k*k)
            den_mat=np.zeros((k,k))
            for row in range(k):
                for col in range(k):
                    den_mat[row][col]=acf_values[abs(j+row-col)]
    #form the numerator matrix (same as denominator matrix except for last column)
            num_mat=copy.deepcopy(den_mat)
            for row in range(k):
                num_mat[row][k-1]=acf_values[j+row+1]

            det_num=np.linalg.det(num_mat)
            det_den=np.linalg.det(den_mat)
            gpac_ndarray[j][k-1]=det_num/det_den

    # return the GPAC ndarray
    return gpac_ndarray

def autocorrelation_cal(y,k):
    T=len(y)
    mean_y=np.mean(y)
    numerator=0
    denominator=0
    T_k=0

    for t in range(0,T):
        denominator=denominator+(np.square(y[t]-mean_y))
    for t in range(k,T):
        numerator=numerator+((y[t]-mean_y)*(y[t-k]-mean_y))
        T_k=numerator/denominator
    return T_k
```

```
def acf_values(y,ml):
    #lags=[]
    autoCorr=[]
    max_lag=ml
    for i in range(0,max_lag):
        #lags.append(i)
        autoCorr.append(autocorrelation_cal(y,i))

    return autoCorr


def calc_e(y,na,theta):
    num = [1]
    den = [1]
    den=np.concatenate((den,theta[0:na]))
    num=np.concatenate((num,theta[na:]))

    if len(num)<len(den):
        z=np.zeros(len(den)-len(num))
        num=np.concatenate((num,z),axis=None)
    elif len(num)>len(den):
        z=np.zeros(len(num)-len(den))
        den=np.concatenate((den,z),axis=None)

    system = (den,num,1)
    T=len(y)
    t_in=np.arange(0,T)
    t_out, e = signal.dlsim(system,y,t=t_in)
    return e

def levenburgMarquardtStepOne(y,na,nb,theta,delta,N,n):
    e=calc_e(y,na,theta)
    E=np.mat(e)
    SSE=E.T.dot(E)
    X=np.zeros((N,n))
    X=np.mat(X)
    for i in range (0,n):#1 ≤ i ≤ n
        theta_copy=copy.deepcopy(theta)
        theta_copy[i]=theta_copy[i]+delta
        e2=calc_e(y,na,theta_copy)
        x=e-e2
        x=x/delta
```

```
        X[:,i]=x
    A=X.T.dot(X)
    g=X.T.dot(e)

    return A,g,SSE

def levenburgMarquardtStepTwo(y,na,nb,theta,A,g,mu,n):
    I=np.identity(n)
    del_theta=np.linalg.inv(A+(mu*I)).dot(g)
    del_theta_arr=np.array(del_theta).flatten()
    theta_new=theta+del_theta_arr
    e_new=calc_e(y,na,theta_new)
    E_NEW=np.mat(e_new)
    SSE_NEW=E_NEW.T.dot(E_NEW)
    return SSE_NEW,del_theta_arr,theta_new

def levenburgMarquardt(y,na,nb,numOfIter):
    # returns the estimated parameter
    # input parameters are:
    # y (generated using arma process)
    # order of ar process in arma, na
    # order of ma process in ma, nb

    #step 1
    # defining maximum number of iteration

    # the very first theta
    N=len(y)
    n=na+nb
    theta=np.zeros((n))
    delta=0.001
    A,g,SSE=levenburgMarquardtStepOne(y,na,nb,theta,delta,N,n)

    mu=0.01
    SSE_NEW,del_theta,theta_new=levenburgMarquardtStepTwo(y,na,nb,theta,A,g,mu,n)

    iterator=0
    maxIterations=numOfIter
    mu_max=10000000000

    while iterator < maxIterations:
        if SSE_NEW < SSE:
            mag_del_theta = np.linalg.norm(del_theta)
```

```
            if mag_del_theta < 1:
                theta=theta_new
                sigma_e_sq=SSE_NEW/(N-n)
                cov=np.multiply(sigma_e_sq,np.linalg.inv(A))
                conf=np.diagonal(np.sqrt(cov))

                print("i="+str(iterator)+", SSE new less than SSE old, ||del_theta||<0.001 :")
                print("theta=")
                print(theta)
                print("confidence interval = +/-"+str(conf))
                print("Estimated variance of error:")
                print(sigma_e_sq)
                print("covariance matrix:")
                print(cov)
                print("SSE=")
                print(SSE)


                break
                #return theta

            else:
                theta=theta_new
                mu=mu/10

        while SSE_NEW > SSE:
            mu=mu*10
            if mu>mu_max:
                print("i="+str(iterator)+", mu>mu_max")
                print(theta)

                #return theta

    SSE_NEW,del_theta,theta_new=levenburgMarquardtStepTwo(y,na,nb,theta,A,g,mu,n)

        iterator=iterator+1
        if iterator>maxIterations:
            print("i="+str(iterator)+", iter > maxIter")
            print(theta)
            #return theta


        theta=theta_new
```

```
        A,g,SSE=levenburgMarquardtStepOne(y,na,nb,theta,delta,N,n)
SSE_NEW,del_theta,theta_new=levenburgMarquardtStepTwo(y,na,nb,theta,A,g,mu,n)
    return theta
```