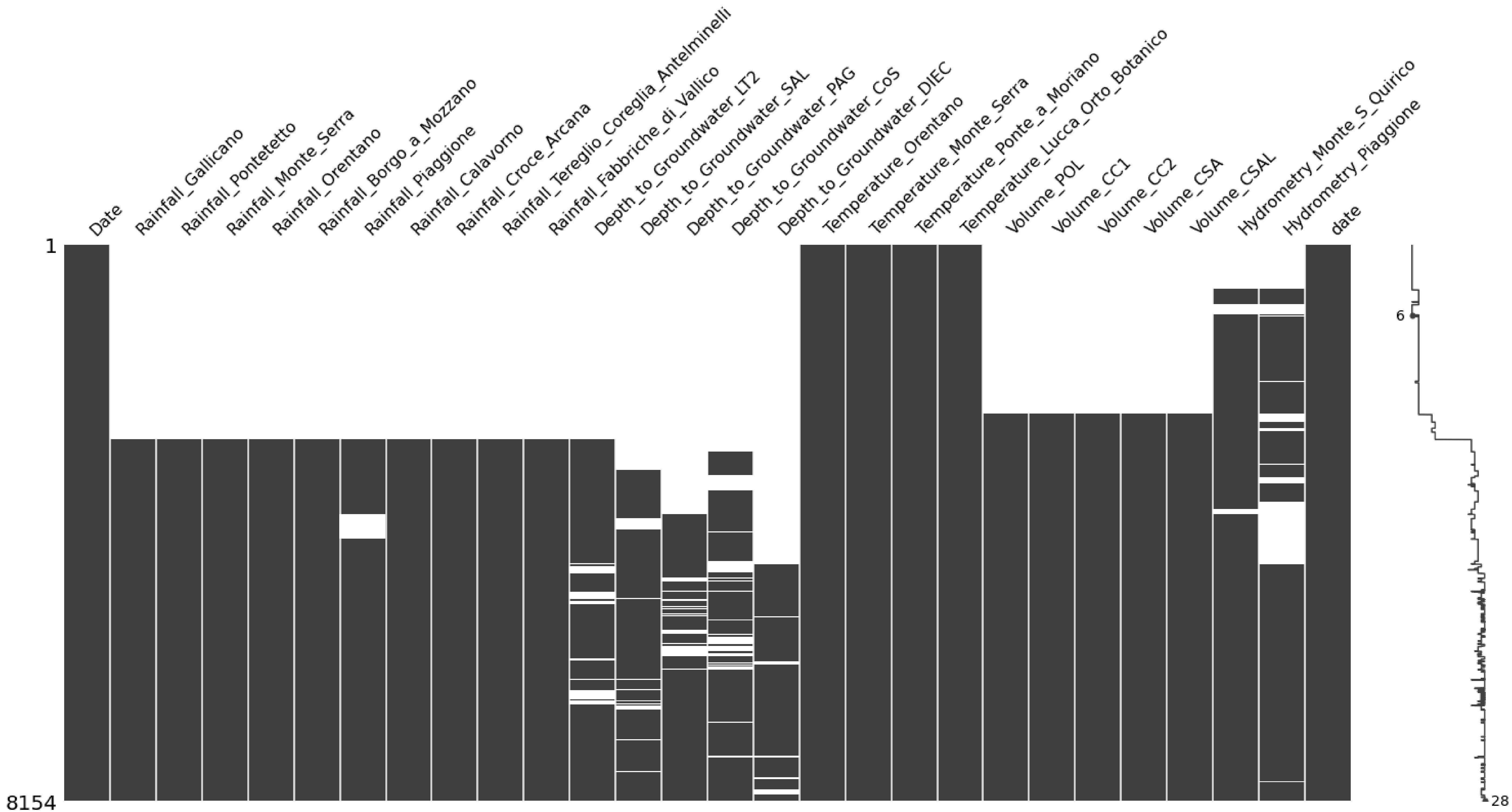
In [2]:
```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.metrics import mean_absolute_error, mean_squared_error
import numpy as np
import pandas as pd
from pandas import DataFrame
%matplotlib inline
import missingno as msno
from datetime import datetime
from numpy.random import multivariate_normal as mvnrnd
from scipy.stats import wishart
from scipy.stats import invwishart
from numpy.linalg import inv as inv
import scipy.io
import time
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
from minepy import MINE
import torch
from torch import nn
from torch.autograd import Variable
from sklearn.linear_model import RidgeCV
from math import sqrt
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
from pykalman import KalmanFilter
from sklearn.ensemble import RandomForestRegressor
import random
```

Data import

We use the data of Aquifer_Auser as an example to demonstrate our model.

In [2]:
```
df = pd.read_csv("D:/Acea_project/Aquifer_Auser.csv")
df['date'] = df['Date'].apply(lambda x: datetime.strptime(x, "%d/%m/%Y"))
target_variable =['Depth_to_Groundwater_SAL', 'Depth_to_Groundwater_CoS', 'Depth_to_Groundwater_LT2']
columns_name = df.columns.values.tolist()
rain_list = [ a for a in columns_name if a.startswith('Rain')]
Temp_list = [ a for a in columns_name if a.startswith('Temperature')]
Depth_list = [ a for a in columns_name if a.startswith('Depth')]
n_row = df.shape[0]
msno.matrix(df)
```

Out[2]: <AxesSubplot:>



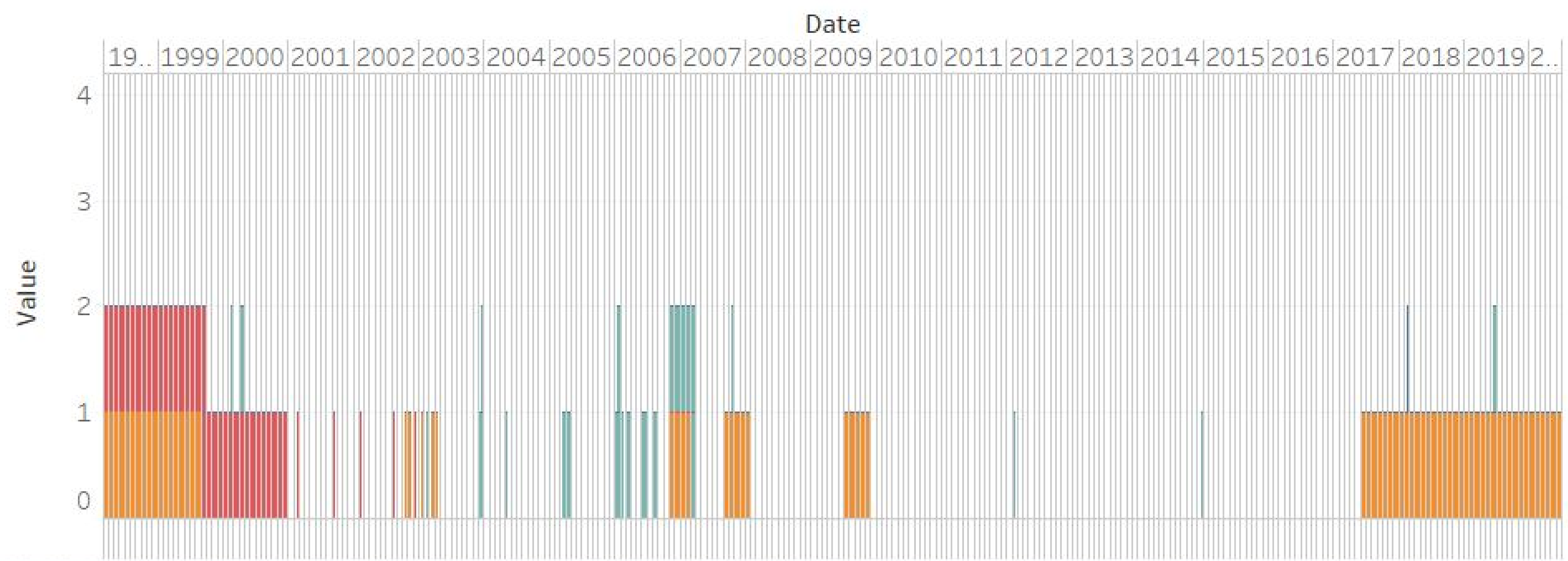
from the figure above, we can see that there are a huge number of missing values, and the density of these missing values is quite large. For the variables that will be used as predicted values, many sparse missing values are scattered in them. We decided to remove the rows with large density of missing values first, and then fill in the sparse missing values.

We found that there are some continuous zero values in the temperature variable and depth variable, which is quite abnormal. To prevent these zero values from being caused by measurement errors, we turn them into missing values.

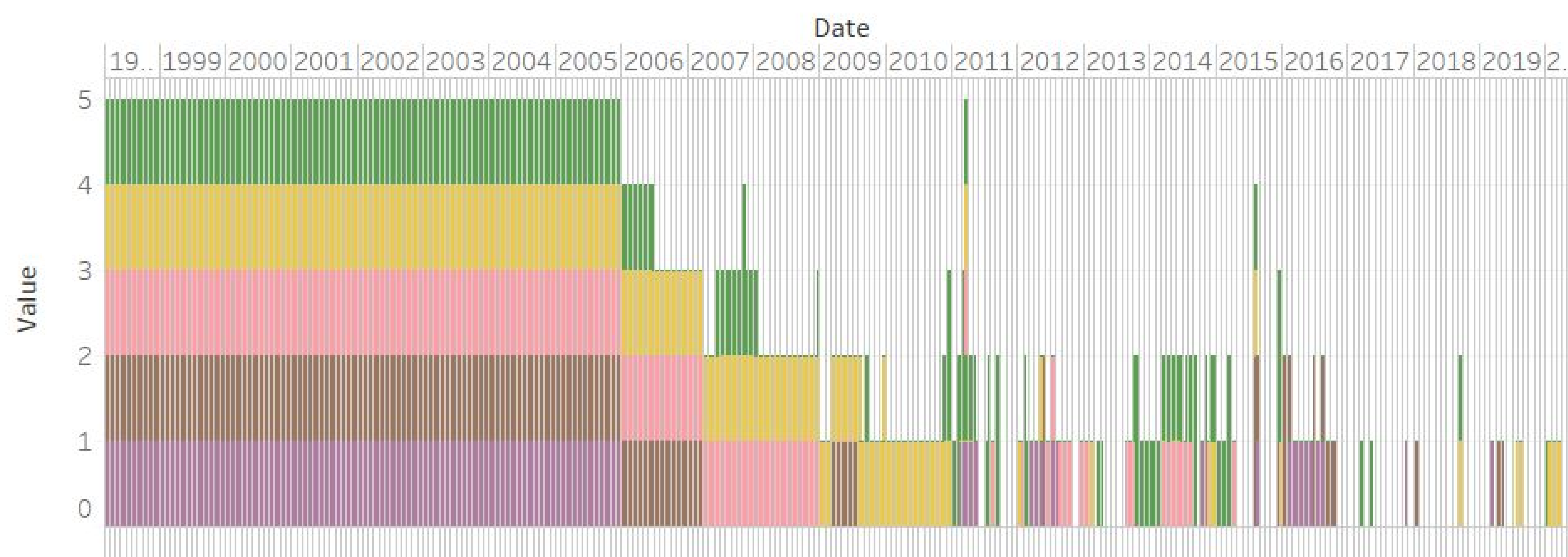
In [12]:
```
for i in range(len(Depth_list)):
        df[df[[Depth_list[i]]]==0]=np.nan

for i in range(len(Temp_list)):
        df[df[[Temp_list[i]]]==0]=np.nan
```

We used tableau to create a dashboard to observe the missing value distribution.

NullValue in Temperature



NullValue in Depth



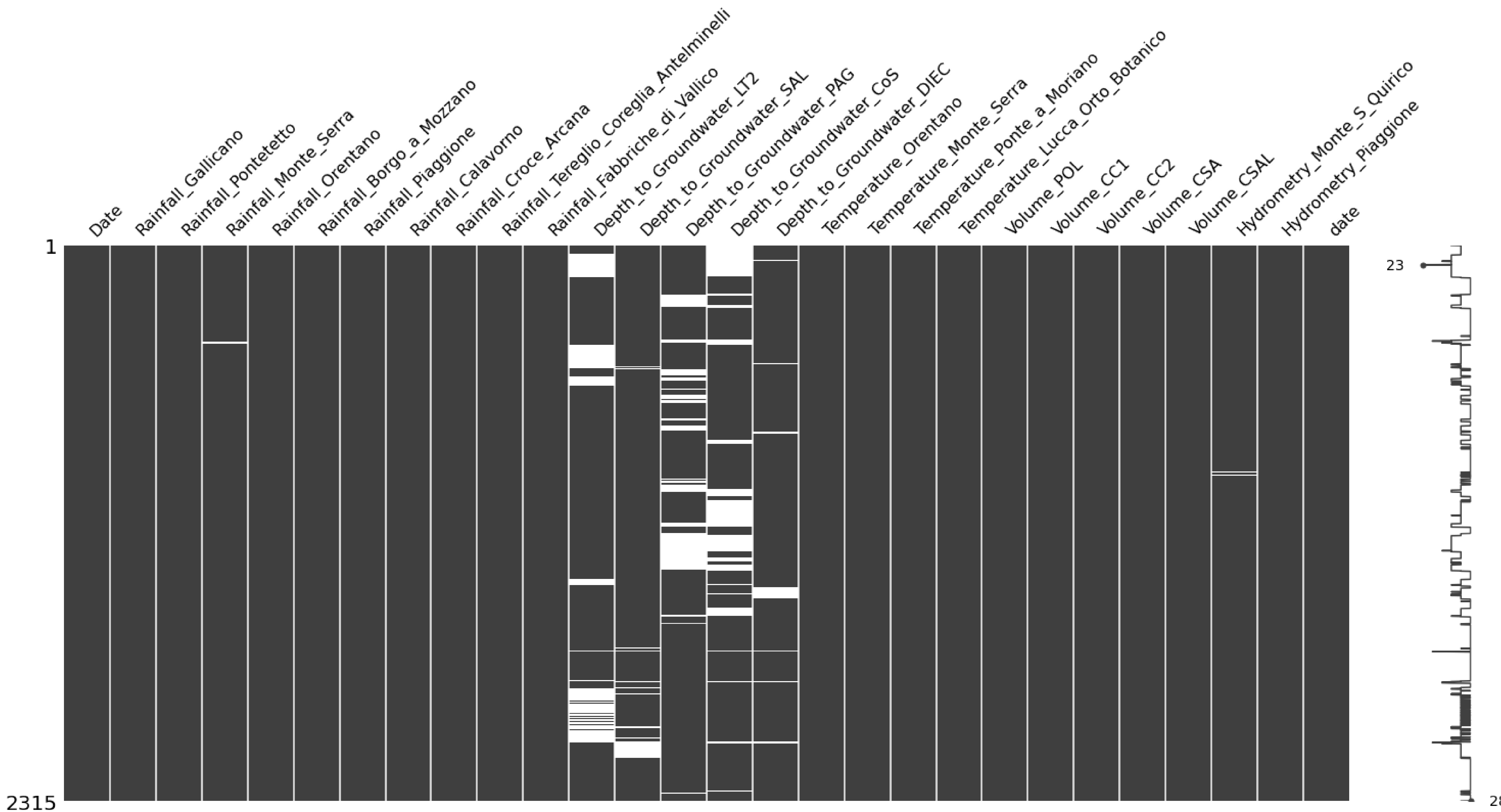
according to tableau and dashboard above, we chose all the data from 2011-01-01 to 2017-05-04 to build the model.

In [16]:
```
df[df['Date']=='31/12/2010'].index.tolist()
```

Out[16]:
```
[4684]
```

In [19]:
```
df[df['Date']=='04/05/2017'].index.tolist()
```
Out[19]:
```
[7000]
```

In [20]:
```
dfsecond = df[4685:7000]
msno.matrix(dfsecond)
```
Out[20]:
```
<AxesSubplot:>
```



Feature Engineering

Rainfall variables and temperature variables are important independent variables in this dataset, but their influence on the Depth_to_Groundwater variable may lag behind. So we need to find how many days will it take for their impact on the target variables to be reflected in the value. Take 'Depth_to_Groundwater_CoS' as excample, shift the data in this variable 31 times and create a variable after each shift.

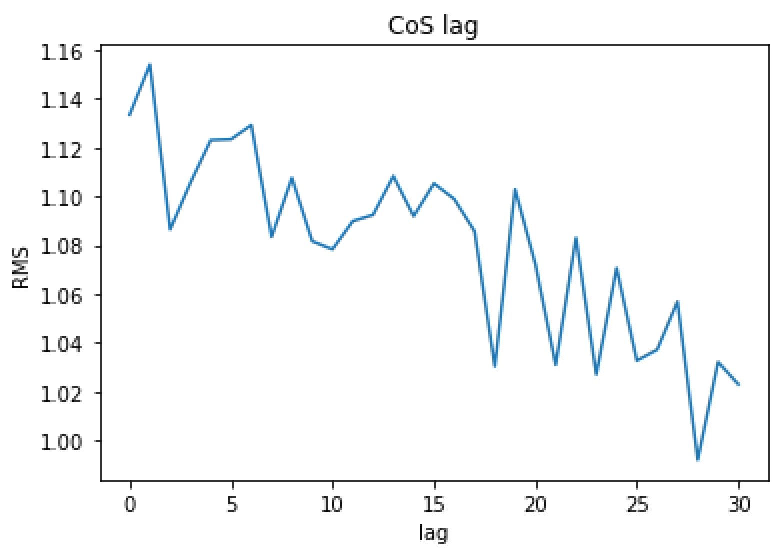
In [21]:
```
dfTestLag = dfsecond[rain_list+Temp_list]
for i in range(0,31):
    dfTestLag['CoS'+str(i)] = dfsecond['Depth_to_Groundwater_CoS'].shift(-1*i)

targetlistCoS = ['CoS0','CoS1','CoS2', 'CoS3', 'CoS4', 'CoS5', 'CoS6', 'CoS7', 'CoS8', 'CoS9', 'CoS10', 'CoS11',
                 'CoS12', 'CoS13', 'CoS14', 'CoS15', 'CoS16', 'CoS17', 'CoS18', 'CoS19', 'CoS20', 'CoS21', 'CoS22',
                 'CoS23', 'CoS24', 'CoS25', 'CoS26', 'CoS27','CoS28','CoS29','CoS30']
```

```
<ipython-input-21-705f77992d01>:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy)
  dfTestLag['CoS'+str(i)] = dfsecond['Depth_to_Groundwater_CoS'].shift(-1*i)
```

We use ridgeCV regression to fit the independent variables to 31 new variables to create 31 models, then we test the rmse of each model, draw a plot, and choose the point with the smallest rmse value to determine the lag value.

In [22]:
```
dfTestLag = dfTestLag.ffill().bfill()
CoSList =[]

for i in range(len(targetlistCoS)):
    YCoS =dfTestLag [targetlistCoS[i]]
    X = dfTestLag[rain_list+Temp_list]
    X_train1,X_test1,y_train1,y_test1 = train_test_split(X,YCoS,test_size=0.2,random_state=i)
    ridgecv = RidgeCV(alphas=[0.01, 0.1, 0.5, 1, 5, 7, 10, 30,100, 200])
    model = ridgecv.fit(X_train1, y_train1)
    y_pred1 = model.predict(X_test1)
    rms1 = sqrt(mean_squared_error(y_test1, y_pred1))
    CoSList.append(rms1)

dfLT2 = pd.DataFrame(CoSList,columns=['rms'])
plt.title("CoS lag")
plt.ylabel("RMS")
plt.xlabel('lag')
plt.plot(dfLT2['rms'])
```

Out[22]:
```
[<matplotlib.lines.Line2D at 0x1e23f2b5130>]
```



We used the same method to test the lag value of other target variables, then we chose '28' as the lag value of target variables.

Bayesian Temporal Matrix Factorization (BTMF)

We chose BTMF method to fill in the missing values of Depth_to_Groundwater variables. The BTMF method has better performance in filling the missing values of long-term time series data sets.

In [23]:
```
dfmeasure = dfsecond[['Depth_to_Groundwater_LT2',
 'Depth_to_Groundwater_SAL',
 'Depth_to_Groundwater_PAG',
 'Depth_to_Groundwater_CoS',
 'Depth_to_Groundwater_DIEC']]
dfdens = dfmeasure[['Depth_to_Groundwater_LT2',
 'Depth_to_Groundwater_SAL',
 'Depth_to_Groundwater_PAG',
 'Depth_to_Groundwater_CoS',
 'Depth_to_Groundwater_DIEC']]
dfdens['Depth_to_Groundwater_LT2'] = dfdens['Depth_to_Groundwater_LT2'].interpolate()
dfdens['Depth_to_Groundwater_SAL'] = dfdens['Depth_to_Groundwater_SAL'].interpolate()
dfdens['Depth_to_Groundwater_PAG'] = dfdens['Depth_to_Groundwater_PAG'].interpolate()
dfdens['Depth_to_Groundwater_CoS'] = dfdens['Depth_to_Groundwater_CoS'].interpolate()
dfdens['Depth_to_Groundwater_DIEC'] = dfdens['Depth_to_Groundwater_DIEC'].interpolate()
dfdens = dfdens.ffill().bfill()
dfdens = np.delete(dfdens.to_numpy().T,range(len(dfsecond)-(len(dfsecond)//28)*28),axis = 1)
dfdealMis = np.delete(dfmeasure.fillna(0).to_numpy().T,range(len(dfsecond)-(len(dfsecond)//28)*28),axis = 1)

def kr_prod(a, b):
    return np.einsum('ir, jr -> ijr', a, b).reshape(a.shape[0] * b.shape[0], -1)

def cov_mat(mat):
    dim1, dim2 = mat.shape
    new_mat = np.zeros((dim2, dim2))
    mat_bar = np.mean(mat, axis = 0)
    for i in range(dim1):
        new_mat += np.einsum('i, j -> ij', mat[i, :] - mat_bar, mat[i, :] - mat_bar)
    return new_mat

def ten2mat(tensor, mode):
    return np.reshape(np.moveaxis(tensor, mode, 0), (tensor.shape[mode], -1), order = 'F')

def mat2ten(mat, tensor_size, mode):
    index = list()
    index.append(mode)
    for i in range(tensor_size.shape[0]):
        if i != mode:
            index.append(i)
    return np.moveaxis(np.reshape(mat, list(tensor_size[index]), order = 'F'), 0, mode)

def mnrnd(M, U, V):
    """
    Generate matrix normal distributed random matrix.
    M is a m-by-n matrix, U is a m-by-m matrix, and V is a n-by-n matrix.
    """
    dim1, dim2 = M.shape
    X0 = np.random.rand(dim1, dim2)
    P = np.linalg.cholesky(U)
    Q = np.linalg.cholesky(V)
    return M + np.matmul(np.matmul(P, X0), Q.T)

def BTMF(dense_mat, sparse_mat, init, rank, time_lags, maxiter1, maxiter2):
    """Bayesian Temporal Matrix Factorization, BTMF."""
    W = init["W"]
    X = init["X"]
```

```
    d = time_lags.shape[0]
    dim1, dim2 = sparse_mat.shape
    pos = np.where((dense_mat != 0) & (sparse_mat == 0))
    position = np.where(sparse_mat != 0)
    binary_mat = np.zeros((dim1, dim2))
    binary_mat[position] = 1

    beta0 = 1
    nu0 = rank
    mu0 = np.zeros((rank))
    W0 = np.eye(rank)
    tau = 1
    alpha = 1e-6
    beta = 1e-6
    S0 = np.eye(rank)
    Psi0 = np.eye(rank * d)
    M0 = np.zeros((rank * d, rank))

    W_plus = np.zeros((dim1, rank))
    X_plus = np.zeros((dim2, rank))
    X_new_plus = np.zeros((dim2 + 1, rank))
    A_plus = np.zeros((rank, rank, d))
    mat_hat_plus = np.zeros((dim1, dim2 + 1))
    for iters in range(maxiter1):
        W_bar = np.mean(W, axis = 0)
        var_mu_hyper = (dim1 * W_bar)/(dim1 + beta0)
        var_W_hyper = inv(inv(W0) + cov_mat(W) + dim1 * beta0/(dim1 + beta0) * np.outer(W_bar, W_bar))
        var_Lambda_hyper = wishart(df = dim1 + nu0, scale = var_W_hyper, seed = None).rvs()
        var_mu_hyper = mvnrnd(var_mu_hyper, inv((dim1 + beta0) * var_Lambda_hyper))

        var1 = X.T
        var2 = kr_prod(var1, var1)
        var3 = tau * np.matmul(var2, binary_mat.T).reshape([rank, rank, dim1]) + np.dstack([var_Lambda_hyper] * dim1)
        var4 = (tau * np.matmul(var1, sparse_mat.T)
                + np.dstack([np.matmul(var_Lambda_hyper, var_mu_hyper)] * dim1)[0, :, :])
        for i in range(dim1):
            inv_var_Lambda = inv(var3[:, :, i])
            W[i, :] = mvnrnd(np.matmul(inv_var_Lambda, var4[:, i]), inv_var_Lambda)
        if iters + 1 > maxiter1 - maxiter2:
            W_plus += W

        Z_mat = X[np.max(time_lags) : dim2, :]
        Q_mat = np.zeros((dim2 - np.max(time_lags), rank * d))
        for t in range(np.max(time_lags), dim2):
            Q_mat[t - np.max(time_lags), :] = X[t - time_lags, :].reshape([rank * d])
        var_Psi = inv(inv(Psi0) + np.matmul(Q_mat.T, Q_mat))
        var_M = np.matmul(var_Psi, np.matmul(inv(Psi0), M0) + np.matmul(Q_mat.T, Z_mat))
        var_S = (S0 + np.matmul(Z_mat.T, Z_mat) + np.matmul(np.matmul(M0.T, inv(Psi0)), M0)
                 - np.matmul(np.matmul(var_M.T, inv(var_Psi)), var_M))
        Sigma = invwishart(df = nu0 + dim2 - np.max(time_lags), scale = var_S, seed = None).rvs()
        A = mat2ten(mnrnd(var_M, var_Psi, Sigma).T, np.array([rank, rank, d]), 0)
        if iters + 1 > maxiter1 - maxiter2:
            A_plus += A

        Lambda_x = inv(Sigma)
        var1 = W.T
```

```
        var2 = kr_prod(var1, var1)
        var3 = tau * np.matmul(var2, binary_mat).reshape([rank, rank, dim2]) + np.dstack([Lambda_x] * dim2)
        var4 = tau * np.matmul(var1, sparse_mat)
        for t in range(dim2):
            Mt = np.zeros((rank, rank))
            Nt = np.zeros(rank)
            if t < np.max(time_lags):
                Qt = np.zeros(rank)
            else:
                Qt = np.matmul(Lambda_x, np.matmul(ten2mat(A, 0), X[t - time_lags, :].reshape([rank * d])))
            if t < dim2 - np.min(time_lags):
                if t >= np.max(time_lags) and t < dim2 - np.max(time_lags):
                    index = list(range(0, d))
                else:
                    index = list(np.where((t + time_lags >= np.max(time_lags)) & (t + time_lags < dim2)))[0]
                for k in index:
                    Ak = A[:, :, k]
                    Mt += np.matmul(np.matmul(Ak.T, Lambda_x), Ak)
                    A0 = A.copy()
                    A0[:, :, k] = 0
                    var5 = (X[t + time_lags[k], :]
                            - np.matmul(ten2mat(A0, 0), X[t + time_lags[k] - time_lags, :].reshape([rank * d])))
                    Nt += np.matmul(np.matmul(Ak.T, Lambda_x), var5)
            var_mu = var4[:, t] + Nt + Qt
            if t < np.max(time_lags):
                inv_var_Lambda = inv(var3[:, :, t] + Mt - Lambda_x + np.eye(rank))
            else:
                inv_var_Lambda = inv(var3[:, :, t] + Mt)
            X[t, :] = mvnrnd(np.matmul(inv_var_Lambda, var_mu), inv_var_Lambda)
        mat_hat = np.matmul(W, X.T)

        X_new = np.zeros((dim2 + 1, rank))
        if iters + 1 > maxiter1 - maxiter2:
            X_new[0 : dim2, :] = X.copy()
            X_new[dim2, :] = np.matmul(ten2mat(A, 0), X_new[dim2 - time_lags, :].reshape([rank * d]))
            X_new_plus += X_new
            mat_hat_plus += np.matmul(W, X_new.T)

        tau = np.random.gamma(alpha + 0.5 * sparse_mat[position].shape[0],
                              1/(beta + 0.5 * np.sum((sparse_mat - mat_hat)[position] ** 2)))
        rmse = np.sqrt(np.sum((dense_mat[pos] - mat_hat[pos]) ** 2)/dense_mat[pos].shape[0])
        if (iters + 1) % 200 == 0 and iters < maxiter1 - maxiter2:
            print('Iter: {}'.format(iters + 1))
            print('RMSE: {:.6}'.format(rmse))
            print()

    W = W_plus/maxiter2
    X_new = X_new_plus/maxiter2
    A = A_plus/maxiter2
    mat_hat = mat_hat_plus/maxiter2
    if maxiter1 >= 100:
        final_mape = np.sum(np.abs(dense_mat[pos] - mat_hat[pos])/dense_mat[pos])/dense_mat[pos].shape[0]
        final_rmse = np.sqrt(np.sum((dense_mat[pos] - mat_hat[pos]) ** 2)/dense_mat[pos].shape[0])
        print('Imputation MAPE: {:.6}'.format(final_mape))
        print('Imputation RMSE: {:.6}'.format(final_rmse))
        print()
```

```
    return mat_hat, W, X_new, A


sparse_mat = dfdealMis
dense_mat = dfdens
import time
start = time.time()
dim1, dim2 = sparse_mat.shape
rank = 10
time_lags = np.array([1, 2, (len(dfsecond)//28)])
init = {"W": 0.1 * np.random.rand(dim1, rank), "X": 0.1 * np.random.rand(dim2, rank)}
maxiter1 = 1100
maxiter2 = 100
a,b,c,d = BTMF(dense_mat, sparse_mat, init, rank, time_lags, maxiter1, maxiter2)
end = time.time()
print('Running time: %d seconds'%(end - start))
```

```
Iter: 200
RMSE: 1.33503

Iter: 400
RMSE: 1.28233

Iter: 600
RMSE: 1.35674

Iter: 800
RMSE: 1.18284

Iter: 1000
RMSE: 1.19417

Imputation MAPE: -0.0783126
Imputation RMSE: 0.577262

Running time: 1543 seconds
```

After imputation we got a matrix called 'a', which is the array contains all 'Depth_to_Groundwater' variable values after filling in the missing values. We use this matrix to replace the original data of 'Depth_to_Groundwater' variable in the dataset.

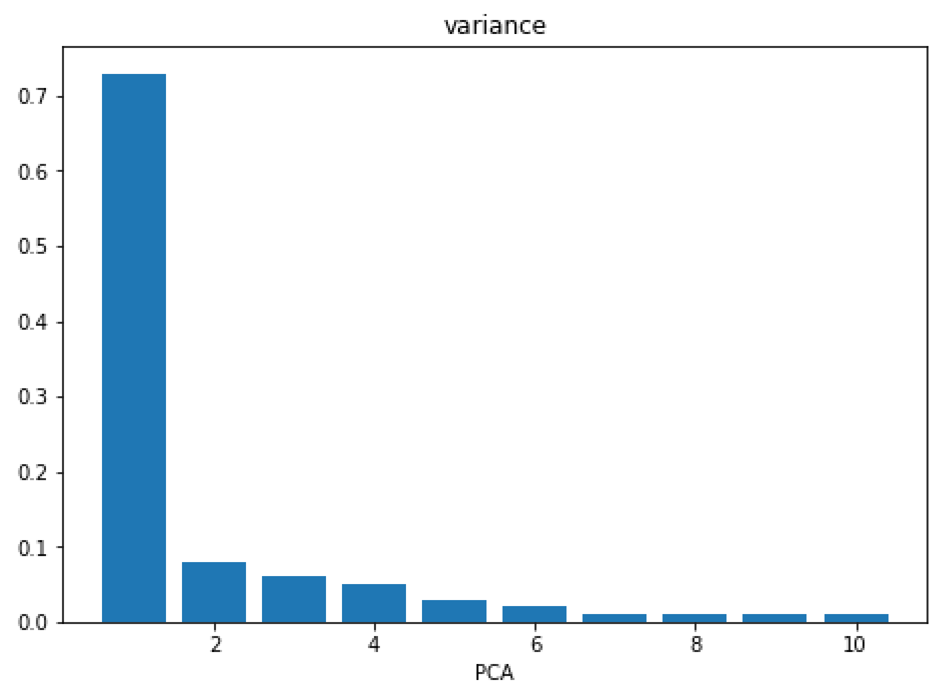
In [24]:

```
a = np.delete(a,-1,axis = 1)
dfRainfall = dfsecond[rain_list].to_numpy()
dfRainfall = np.delete(dfRainfall,range(len(dfsecond)-(len(dfsecond)//28)*28),axis = 0)
dfTemp = dfsecond[Temp_list].to_numpy()
dfTemp = np.delete(dfTemp,range(len(dfsecond)-(len(dfsecond)//28)*28),axis = 0)
pdate = pd.DataFrame(dfsecond['date'].values.astype('float32'), columns=['Date'])
dfDate = pdate['Date'].to_numpy()
dfDate = np.delete(dfDate,range(len(dfsecond)-(len(dfsecond)//28)*28),axis = 0)
dfDate = dfDate.reshape(-1,1)
a = a.T
wholedata = np.hstack((a,dfRainfall,dfTemp,dfDate))
wholelist = Depth_list+rain_list+Temp_list+['date']
newFrame = DataFrame(wholedata,index=None,columns = wholelist)
```

PCA

We found there are too many rainfall variables, so we decided to use PCA to reduce the number of these variables.

In [25]:
```
test=newFrame[rain_list]
test = test.ffill().bfill()

fig = plt.figure()
ax = fig.add_axes([0,0,1,1])

pca = PCA(n_components=10)
pca.fit(test)
v = pca.explained_variance_ratio_.round(2)

ax.bar(range(1,11),v)
plt.xlabel("PCA")
plt.title("variance")
plt.show()
```



We chose the first two PCA to replace the rainfall variables.

In [26]:
```
Xpca= PCA(n_components=2).fit_transform(test)
pf = pd.DataFrame(Xpca, columns=['PCA1','PCA2'])
newFrame['PCA1'] = pf['PCA1']
newFrame['PCA2'] = pf['PCA2']
for i in range(len(rain_list)):
    newFrame = newFrame.drop(rain_list[i],axis=1)
newFrame = newFrame.ffill().bfill()
```

We use ".shift(-28)" to creat three new variables, which are the target variables for the prediction model of this dataset.

In [27]:
```
lag_target = []
for i in range(len(target_variable)):
    newFrame[target_variable[i]+'28'] = newFrame[target_variable[i]].shift(-28)
    lag_target.append(target_variable[i]+'28')
newFrame
```

Out[27]:

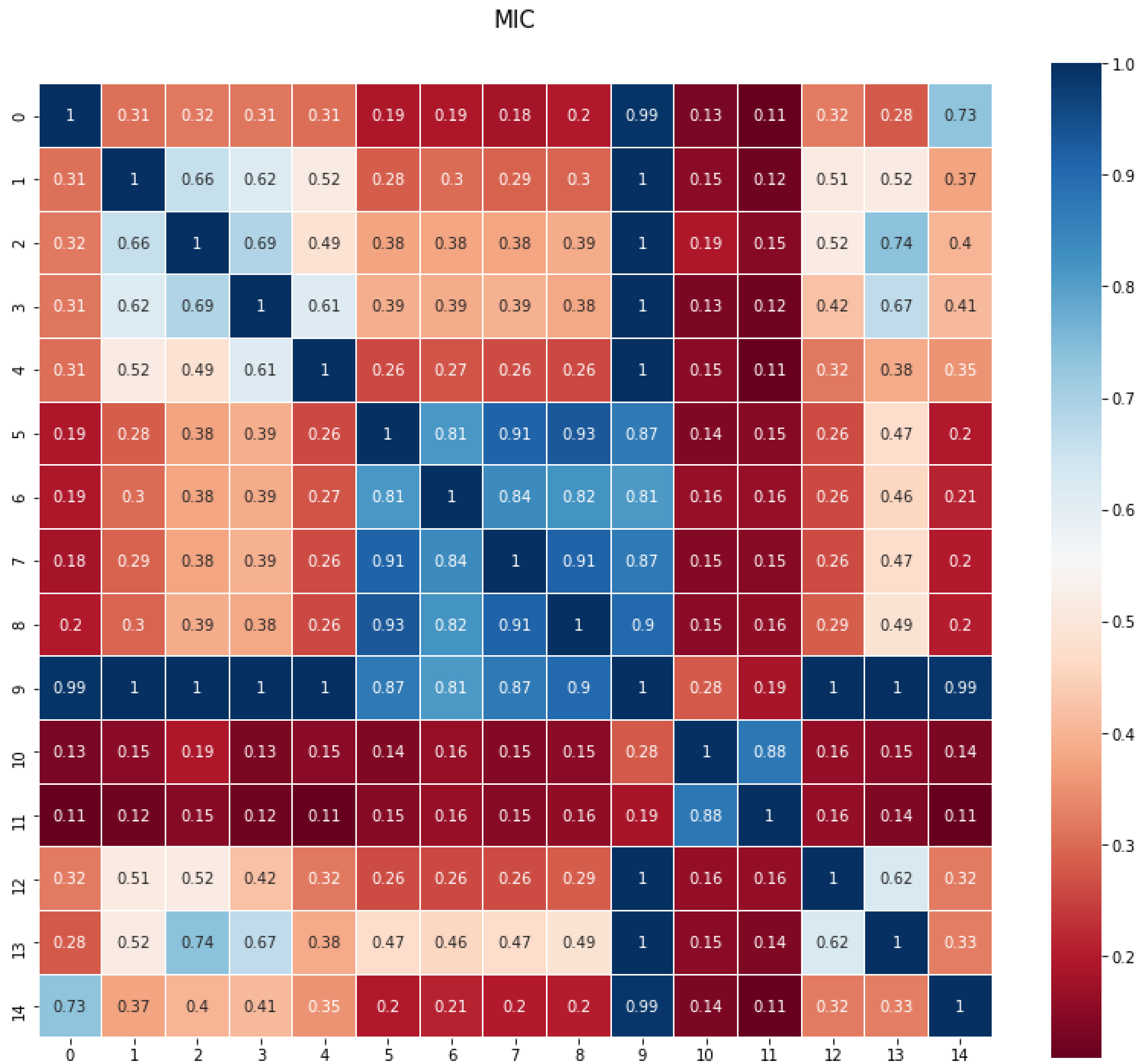
	Depth_to_Groundwater_LT2	Depth_to_Groundwater_SAL	Depth_to_Groundwater_PAG	Depth_to_Groundwater_CoS	Depth_to_Groundwater_DIEC	Temperature_Orentano	Temperature_Monte_Serra	Temperature_Ponte_a_Moriano	T
0	-13.179348	-5.199800	-1.469652	-4.794773	-2.719689	7.35	1.20	6.15	
1	-13.180207	-5.219736	-1.590671	-5.129179	-2.770137	5.95	-0.85	6.65	
2	-13.170578	-5.239350	-1.650583	-6.326457	-2.809943	2.35	-2.15	2.25	
3	-13.170842	-5.269656	-1.679675	-5.975174	-2.839602	2.30	-0.70	3.75	
4	-13.180397	-5.289587	-1.699447	-5.393212	-2.859416	1.10	0.60	3.30	
...	...	...	...	...	...	...	...	...	
2291	-12.249551	-5.700631	-2.268955	-4.740573	-3.939061	11.75	8.75	12.50	
2292	-12.250390	-5.700358	-2.240766	-4.799599	-3.949835	13.55	9.15	14.10	
2293	-12.239961	-5.699237	-2.230016	-4.730611	-3.949963	12.45	8.45	12.30	
2294	-12.279879	-5.660030	-2.209793	-4.795808	-3.970235	13.45	8.70	13.25	
2295	-12.269984	-5.641207	-2.250465	-4.797124	-3.970142	15.00	9.55	15.15	

2296 rows × 15 columns

Correlation Matrix of the new dataset

We checked MIC values between all the variables, and delete those variables had higher MIC values with some other variables.

In [28]:
```
def MIC_matirx(dataframe, mine):

    data_array = np.array(dataframe)
    n = len(data_array[0, :])
    output = np.zeros([n, n])

    for i in range(n):
        for j in range(n):
            mine.compute_score(data_array[:, i], data_array[:, j])
            output[i, j] = mine.mic()
            output[j, i] = mine.mic()
    mic_value = pd.DataFrame(output)
    return mic_value


mine = MINE(alpha=0.6, c=15)
Matrix_mic_value = MIC_matirx(newFrame, mine)

def HeatMap(DataFrame):
    %matplotlib inline
    colormap = plt.cm.RdBu
    plt.figure(figsize=(14,12))
    plt.title('MIC', y=1.05, size=15)
    sns.heatmap(DataFrame.astype(float),linewidths=0.1,vmax=1.0, square=True, cmap=colormap, linecolor='white', annot=True)
    plt.show()
HeatMap(Matrix_mic_value)
```


MIC

In [29]:
```
newFrame.columns.values.tolist()
```

Out[29]:
```
['Depth_to_Groundwater_LT2',
 'Depth_to_Groundwater_SAL',
 'Depth_to_Groundwater_PAG',
 'Depth_to_Groundwater_CoS',
 'Depth_to_Groundwater_DIEC',
 'Temperature_Orentano',
 'Temperature_Monte_Serra',
 'Temperature_Ponte_a_Moriano',
 'Temperature_Lucca_Orto_Botanico',
 'date',
 'PCA1',
 'PCA2',
 'Depth_to_Groundwater_SAL28',
 'Depth_to_Groundwater_CoS28',
 'Depth_to_Groundwater_LT228']
```

According to the MIC matrix above, we can delete columns named 'Temperature_Ponte_a_Moriano' and 'Temperature_Lucca_Orto_Botanico'.

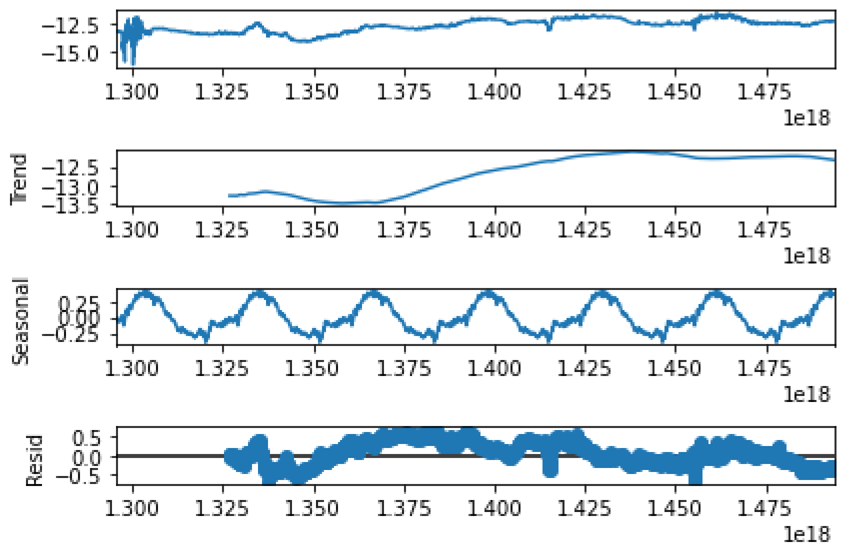
In [38]:
```
newFrame = newFrame.drop([ 'Temperature_Ponte_a_Moriano','Temperature_Lucca_Orto_Botanico'], axis=1)
```

In [42]:
```
Temp_list.remove('Temperature_Ponte_a_Moriano')
Temp_list.remove('Temperature_Lucca_Orto_Botanico')
```

Seasonal and Trend

Take "Depth_to_Groundwater_LT2" as an excample, we use "seasonal_decompose()" to decompose it's data into trend and seasonality.

In [31]:
```
dfseason = pd.Series(newFrame['Depth_to_Groundwater_LT2'].tolist(),index = newFrame['date'].tolist())


decomposition = seasonal_decompose(dfseason, model='additive',period = 365,two_sided = False)
decomposition.plot()
plt.show()
```



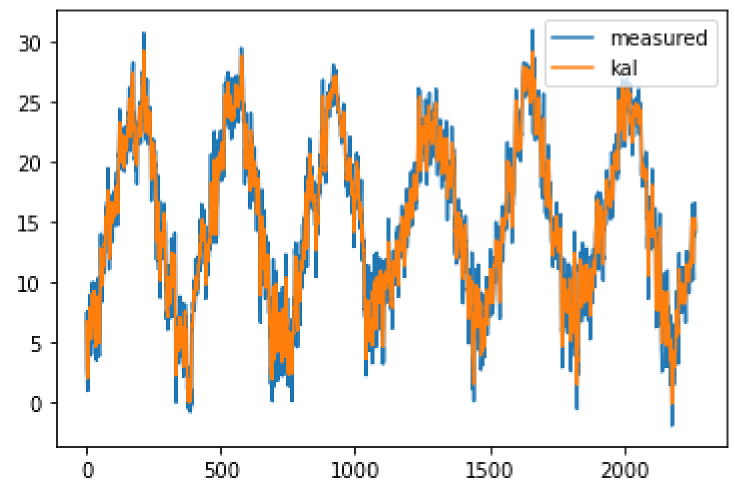
in this project, we don't need to remove or adjust and of these three components. According to the figure above, we can see that the seasonality of this variable is not obvious. In our dataset, although temperature has significant seasonality， its influence on groundwater depth is not significantly stronger than the other factors' influence.

Kalman filter

Errors, such as measurement errors, will add some noise to the data, which will affect the accuracy of prediction. Since rainfall variables have already been converted to PCA value, We used Kalman filter to remove this noise in Depth_to_Groundwater variables and temperature variables.

In [34]:
```
def Kalman1D(data,damping=1):
    observation_covariance = damping
    first_value = data[0]
    transition_matrix = 1
    transition_covariance = 0.1
    first_value
    kf = KalmanFilter(
            initial_state_mean=first_value,
            initial_state_covariance=observation_covariance,
            observation_covariance=observation_covariance,
            transition_covariance=transition_covariance,
            transition_matrices=transition_matrix
        )
    pred_state, state_cov = kf.smooth(data)
    return pred_state
```

In [39]:
```
dffull = newFrame[:len(newFrame)-28]
orenArray = dffull['Temperature_Orentano'].to_numpy()
orenkal = Kalman1D(orenArray,0.1)
plt.plot(np.array(list(range(len(newFrame)-28))),orenArray,label='measured')
plt.plot(np.array(list(range(len(newFrame)-28))),orenkal,label='kal')

plt.legend()
plt.show()
```



The orange line shows the temperature value after the noise is eliminated. We can see that it has become smoother than the blue line which shows the original temperature value.

After confirming that this method is effective, we applied it to all the Depth_to_Groundwater variables and temperature variables.

In [43]:
```
dffullkal = dffull.drop(Depth_list+Temp_list,axis =1 )
for i in range(len(Depth_list)):
    DepthArray = dffull[Depth_list[i]].to_numpy()
    Depthkal = Kalman1D(DepthArray,0.1)
    kallist = map(lambda x: x[0], Depthkal)
    Depthkalseries = pd.Series(kallist)
    dffullkal[Depth_list[i]] = Depthkalseries

for i in range(len(Temp_list)):
    TempArray = dffull[Temp_list[i]].to_numpy()
    Tempkal = Kalman1D(TempArray,0.1)
    kallist = map(lambda x: x[0], Tempkal)
    Tempkalseries = pd.Series(kallist)
    dffullkal[Temp_list[i]] = Tempkalseries
```

In [44]:
```
dffullkal
```

Out[44]:

	date	PCA1	PCA2	Depth_to_Groundwater_SAL28	Depth_to_Groundwater_CoS28	Depth_to_Groundwater_LT228	Depth_to_Groundwater_LT2	Depth_to_Groundwater_SAL	Depth_to_Groundwater_PAG	Depth_to_(
0	1.295482e+18	-6.487342	-2.421435	-5.530616	-4.668232	-15.629891	-13.178780	-5.207789	-1.505218	
1	1.295568e+18	-8.342031	15.768792	-5.430167	-3.488856	-15.927838	-13.177644	-5.223766	-1.576348	
2	1.295654e+18	-9.039918	13.559566	-5.329853	-5.617913	-14.072373	-13.173946	-5.243772	-1.633155	
3	1.295741e+18	-11.857972	5.817508	-5.260403	-4.495875	-13.839855	-13.173616	-5.268200	-1.672534	
4	1.295827e+18	-13.410054	1.424599	-5.239764	-3.758904	-14.149658	-13.176060	-5.291171	-1.704773	
...	...	...	...	...	...	...	...	...	...	
2263	1.491005e+18	-13.402127	1.281889	-5.700631	-4.740573	-12.249551	-12.272537	-5.520379	-2.012318	
2264	1.491091e+18	-12.565226	1.107666	-5.700358	-4.799599	-12.250390	-12.254853	-5.532563	-2.030649	
2265	1.491178e+18	-10.447227	-2.741653	-5.699237	-4.730611	-12.239961	-12.252024	-5.547166	-2.042454	
2266	1.491264e+18	-13.402596	1.260261	-5.660030	-4.795808	-12.279879	-12.250676	-5.558961	-2.066901	
2267	1.491350e+18	-12.097390	0.892563	-5.641207	-4.797124	-12.269984	-12.250165	-5.569725	-2.077615	

2268 rows × 13 columns

LSTM

Recurrent Neural Network (RNN) is a neural network used to process sequence data. Compared with the general neural network, it can process the data of the sequence change. Long short-term memory (LSTM) is a special RNN, mainly to solve the problem of gradient disappearance and gradient explosion in the training process of long sequences. Simply put, LSTM can perform better in longer sequences than ordinary RNNs.

In [45]:
```
class lstm_reg(nn.Module):
    def __init__(self, input_size, hidden_size, output_size=1, num_layers=2):
        super(lstm_reg, self).__init__()

        self.rnn = nn.LSTM(input_size, hidden_size, num_layers,dropout = 0.3)
        self.reg = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x, _ = self.rnn(x)
        s, b, h = x.shape
        x = x.view(s*b, h)
        x = self.reg(x)
        x = x.view(s, b, -1)
        return x
```

Take 'Depth_to_Groundwater_SAL28' as the example. if we want to predict the value of 'Depth_to_Groundwater_SAL28', the features we need is 'date','PCA1','PCA2','Depth_to_Groundwater_SAL', 'Temperature_Orentano','Temperature_Monte_Serra'.

In [46]:
```
n_test = int(((len(dffullkal)-28)/28)//5*28)
n_train = len(dffullkal)-28 - n_test

dftrain = dffullkal[:n_train]
dftest = dffullkal[n_train:len(dffullkal)-28]


dftrainX = dftrain[['date','PCA1','PCA2','Depth_to_Groundwater_SAL', 'Temperature_Orentano','Temperature_Monte_Serra']]
n_feature = len(dftrainX.columns.values.tolist())
dflistX = np.reshape(dftrainX.values.tolist(),(28,-1,n_feature))

dftrainY = dftrain['Depth_to_Groundwater_SAL28']
dflistY = np.reshape(dftrainY.values.tolist(),(28,-1,1))
dflistX = dflistX.astype('float32')
dflistY = dflistY.astype('float32')
tensorx = torch.from_numpy(dflistX)
tensory = torch.from_numpy(dflistY)

net = lstm_reg(n_feature, 100)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(net.parameters(), lr=1e-2)

for e in range(100):
    var_x = Variable(tensorx)
    var_y = Variable(tensory)

    out = net(var_x)
    loss = criterion(out, var_y)

    optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_norm_(net.parameters(), 1.1)#gradient clipping, used to avoid Exploding Gradients

    optimizer.step()
    if (e + 1) % 10 == 0:
            print('Epoch: {}, Loss: {:.5f}'.format(e + 1, loss.data))
```

```
Epoch: 10, Loss: 1.58393
Epoch: 20, Loss: 0.49902
Epoch: 30, Loss: 0.41989
Epoch: 40, Loss: 0.33652
Epoch: 50, Loss: 0.27244
Epoch: 60, Loss: 0.22826
Epoch: 70, Loss: 0.21964
Epoch: 80, Loss: 0.21842
Epoch: 90, Loss: 0.20856
Epoch: 100, Loss: 0.20272
```

In [47]:
```
dftestX = dftest[['date','PCA1','PCA2','Depth_to_Groundwater_SAL', 'Temperature_Orentano','Temperature_Monte_Serra']]
dftestlistX = np.reshape(dftestX.values.tolist(),(28,-1,n_feature))

dftestY = dftest['Depth_to_Groundwater_SAL28']
dftestlistY = np.reshape(dftestY.values.tolist(),(28,-1,1))
dftestlistX = dftestlistX.astype('float32')
dftestlistY = dftestlistY.astype('float32')
tensortestx = torch.from_numpy(dftestlistX)
tensortesty = torch.from_numpy(dftestlistY)
testvar_x = Variable(tensortestx)
testvar_y = Variable(tensortesty)

nettest = net.eval()
pred_teste = nettest(testvar_x)
loss = criterion(pred_teste, testvar_y)
print('Epoch: {}, Loss: {:.5f}'.format('mse', loss.data))

a = nn.L1Loss()
maeloss = a(pred_teste, testvar_y)
print('Epoch: {}, Loss: {:.5f}'.format('mae', maeloss.data))
```

```
Epoch: mse, Loss: 0.20286
Epoch: mae, Loss: 0.34701
```

The result looks great. Now we need to apply it to all the 9 datasets. So we need to create a class to collect all the methods of data processing we did in this project.

referance table

We used the missingno library to visualize the missing values of each table, and make dashboard to observe the distribution of missing values in tableau, then we can determine the range of data used to build the prediction model in each table. We refer to the introduction of each table in the 'datasets_description.xlsx' to determine the output of each table and the variables(except for the outputs themselves) that may be used to predict these outputs. We make all these information into a table so that they can be used when needed.

In [3]:
```
referdata = {'table':['Aquifer_Auser','Aquifer_Doganella','Aquifer_Luco',
                      'Aquifer_Petrignano','Lake_Bilancino','River_Arno',
                       'Water_Spring_Amiata','Water_Spring_Lupa','Water_Spring_Madonna_di_Canneto'],
       'start':[4685,3075,6540,1000,1000,2250,5600,600,1600],#Get data begin from which row
       'end':[7000,3950,6950,5223,6000,3450,7487,4199,2500],#Stop getting data after reaching which row
        'feature':[['Rain','Temperature','date'],['Rain','Temperature','date'],
                  ['Rain','Temperature','date'],['Rain','Temperature','date'],
                  ['Rain','Temperature','date'],['Rain','Temperature','date'],
                  ['Rain','Temperature','date'],['Rain','date'],['Rain','Temperature','date']]}
referdf = DataFrame(referdata)
referdf
```

Out[3]:

	table	start	end	feature
0	Aquifer_Auser	4685	7000	[Rain, Temperature, date]
1	Aquifer_Doganella	3075	3950	[Rain, Temperature, date]
2	Aquifer_Luco	6540	6950	[Rain, Temperature, date]
3	Aquifer_Petrignano	1000	5223	[Rain, Temperature, date]
4	Lake_Bilancino	1000	6000	[Rain, Temperature, date]
5	River_Arno	2250	3450	[Rain, Temperature, date]
6	Water_Spring_Amiata	5600	7487	[Rain, Temperature, date]
7	Water_Spring_Lupa	600	4199	[Rain, date]
8	Water_Spring_Madonna_di_Canneto	1600	2500	[Rain, Temperature, date]

In [80]:
```
class data_cook():



    def __init__(self, dataframe, start,end,target_variable):
        '''
        target_variable contains the name of all the variables which cound be used as target variable in this table.
        '''

        self.dfsecond = dataframe[start:end]# the start and end number could be check in the table 'referdf'.
        self.columns_name = dataframe.columns.values.tolist()
        self.Rain_list = [ a for a in self.columns_name if a.startswith('Rain')]
        self.Depth_list = [ a for a in self.columns_name if a.startswith('Depth')]
        self.Temp_list = [ a for a in self.columns_name if a.startswith('Temperature')]
        self.Flow_list = [ a for a in self.columns_name if a.startswith('Flow')]
        self.n_row = dataframe.shape[0]
        self.target_list = target_variable

    def BasicInformation(self):
        msno.matrix(self.dfsecond)
        print(self.columns_name)

    #BTMF
    def FillNullBTMF(self,nullValue_list):
        '''
        the nullValue_list contain the columns' name which we want to fill the null value.
        '''
        for i in range(len(self.Depth_list)):
            self.dfsecond[self.dfsecond[[self.Depth_list[i]]]==0]=np.nan

        for i in range(len(self.Temp_list)):
            self.dfsecond[self.dfsecond[[self.Temp_list[i]]]==0]=np.nan

        for i in range(len(self.Flow_list)):
            self.dfsecond[self.dfsecond[[self.Flow_list[i]]]==0]=np.nan

        dfmeasure = self.dfsecond[nullValue_list]
        dfdens = dfmeasure[nullValue_list]
        for i in range(len(nullValue_list)):
            dfdens[nullValue_list[i]] = dfdens[nullValue_list[i]].interpolate()

        dfdens = dfdens.ffill().bfill()
        dfdens = np.delete(dfdens.to_numpy().T,range(len(self.dfsecond)-(len(self.dfsecond)//28)*28),axis = 1)
        dfdealMis = np.delete(dfmeasure.fillna(0).to_numpy().T,range(len(self.dfsecond)-(len(self.dfsecond)//28)*28),axis = 1)

        def kr_prod(a, b):
            return np.einsum('ir, jr -> ijr', a, b).reshape(a.shape[0] * b.shape[0], -1)

        def cov_mat(mat):
            dim1, dim2 = mat.shape
            new_mat = np.zeros((dim2, dim2))
            mat_bar = np.mean(mat, axis = 0)
            for i in range(dim1):
                new_mat += np.einsum('i, j -> ij', mat[i, :] - mat_bar, mat[i, :] - mat_bar)
            return new_mat
```

```
def ten2mat(tensor, mode):
    return np.reshape(np.moveaxis(tensor, mode, 0), (tensor.shape[mode], -1), order = 'F')

def mat2ten(mat, tensor_size, mode):
    index = list()
    index.append(mode)
    for i in range(tensor_size.shape[0]):
        if i != mode:
            index.append(i)
    return np.moveaxis(np.reshape(mat, list(tensor_size[index]), order = 'F'), 0, mode)

def mnrnd(M, U, V):
    """
    Generate matrix normal distributed random matrix.
    M is a m-by-n matrix, U is a m-by-m matrix, and V is a n-by-n matrix.
    """
    dim1, dim2 = M.shape
    X0 = np.random.rand(dim1, dim2)
    P = np.linalg.cholesky(U)
    Q = np.linalg.cholesky(V)
    return M + np.matmul(np.matmul(P, X0), Q.T)

def BTMF(dense_mat, sparse_mat, init, rank, time_lags, maxiter1, maxiter2):
    """Bayesian Temporal Matrix Factorization, BTMF."""
    W = init["W"]
    X = init["X"]

    d = time_lags.shape[0]
    dim1, dim2 = sparse_mat.shape
    pos = np.where((dense_mat != 0) & (sparse_mat == 0))
    position = np.where(sparse_mat != 0)
    binary_mat = np.zeros((dim1, dim2))
    binary_mat[position] = 1

    beta0 = 1
    nu0 = rank
    mu0 = np.zeros((rank))
    W0 = np.eye(rank)
    tau = 1
    alpha = 1e-6
    beta = 1e-6
    S0 = np.eye(rank)
    Psi0 = np.eye(rank * d)
    M0 = np.zeros((rank * d, rank))

    W_plus = np.zeros((dim1, rank))
    X_plus = np.zeros((dim2, rank))
    X_new_plus = np.zeros((dim2 + 1, rank))
    A_plus = np.zeros((rank, rank, d))
    mat_hat_plus = np.zeros((dim1, dim2 + 1))
    for iters in range(maxiter1):
        W_bar = np.mean(W, axis = 0)
        var_mu_hyper = (dim1 * W_bar)/(dim1 + beta0)
        var_W_hyper = inv(inv(W0) + cov_mat(W) + dim1 * beta0/(dim1 + beta0) * np.outer(W_bar, W_bar))
        var_Lambda_hyper = wishart(df = dim1 + nu0, scale = var_W_hyper, seed = None).rvs()
        var_mu_hyper = mvnrnd(var_mu_hyper, inv((dim1 + beta0) * var_Lambda_hyper))
```

```
        var1 = X.T
        var2 = kr_prod(var1, var1)
        var3 = tau * np.matmul(var2, binary_mat.T).reshape([rank, rank, dim1]) + np.dstack([var_Lambda_hyper] * dim1)
        var4 = (tau * np.matmul(var1, sparse_mat.T)
                + np.dstack([np.matmul(var_Lambda_hyper, var_mu_hyper)] * dim1)[0, :, :])
        for i in range(dim1):
            inv_var_Lambda = inv(var3[:, :, i])
            W[i, :] = mvnrnd(np.matmul(inv_var_Lambda, var4[:, i]), inv_var_Lambda)
        if iters + 1 > maxiter1 - maxiter2:
            W_plus += W

        Z_mat = X[np.max(time_lags) : dim2, :]
        Q_mat = np.zeros((dim2 - np.max(time_lags), rank * d))
        for t in range(np.max(time_lags), dim2):
            Q_mat[t - np.max(time_lags), :] = X[t - time_lags, :].reshape([rank * d])
        var_Psi = inv(inv(Psi0) + np.matmul(Q_mat.T, Q_mat))
        var_M = np.matmul(var_Psi, np.matmul(inv(Psi0), M0) + np.matmul(Q_mat.T, Z_mat))
        var_S = (S0 + np.matmul(Z_mat.T, Z_mat) + np.matmul(np.matmul(M0.T, inv(Psi0)), M0)
                 - np.matmul(np.matmul(var_M.T, inv(var_Psi)), var_M))
        Sigma = invwishart(df = nu0 + dim2 - np.max(time_lags), scale = var_S, seed = None).rvs()
        A = mat2ten(mnrnd(var_M, var_Psi, Sigma).T, np.array([rank, rank, d]), 0)
        if iters + 1 > maxiter1 - maxiter2:
            A_plus += A

        Lambda_x = inv(Sigma)
        var1 = W.T
        var2 = kr_prod(var1, var1)
        var3 = tau * np.matmul(var2, binary_mat).reshape([rank, rank, dim2]) + np.dstack([Lambda_x] * dim2)
        var4 = tau * np.matmul(var1, sparse_mat)
        for t in range(dim2):
            Mt = np.zeros((rank, rank))
            Nt = np.zeros(rank)
            if t < np.max(time_lags):
                Qt = np.zeros(rank)
            else:
                Qt = np.matmul(Lambda_x, np.matmul(ten2mat(A, 0), X[t - time_lags, :].reshape([rank * d])))
            if t < dim2 - np.min(time_lags):
                if t >= np.max(time_lags) and t < dim2 - np.max(time_lags):
                    index = list(range(0, d))
                else:
                    index = list(np.where((t + time_lags >= np.max(time_lags)) & (t + time_lags < dim2)))[0]
                for k in index:
                    Ak = A[:, :, k]
                    Mt += np.matmul(np.matmul(Ak.T, Lambda_x), Ak)
                    A0 = A.copy()
                    A0[:, :, k] = 0
                    var5 = (X[t + time_lags[k], :]
                            - np.matmul(ten2mat(A0, 0), X[t + time_lags[k] - time_lags, :].reshape([rank * d])))
                    Nt += np.matmul(np.matmul(Ak.T, Lambda_x), var5)
            var_mu = var4[:, t] + Nt + Qt
            if t < np.max(time_lags):
                inv_var_Lambda = inv(var3[:, :, t] + Mt - Lambda_x + np.eye(rank))
            else:
                inv_var_Lambda = inv(var3[:, :, t] + Mt)
            X[t, :] = mvnrnd(np.matmul(inv_var_Lambda, var_mu), inv_var_Lambda)
        mat_hat = np.matmul(W, X.T)
```

```
            X_new = np.zeros((dim2 + 1, rank))
            if iters + 1 > maxiter1 - maxiter2:
                X_new[0 : dim2, :] = X.copy()
                X_new[dim2, :] = np.matmul(ten2mat(A, 0), X_new[dim2 - time_lags, :].reshape([rank * d]))
                X_new_plus += X_new
                mat_hat_plus += np.matmul(W, X_new.T)

            tau = np.random.gamma(alpha + 0.5 * sparse_mat[position].shape[0],
                                  1/(beta + 0.5 * np.sum((sparse_mat - mat_hat)[position] ** 2)))
            rmse = np.sqrt(np.sum((dense_mat[pos] - mat_hat[pos]) ** 2)/dense_mat[pos].shape[0])
            if (iters + 1) % 200 == 0 and iters < maxiter1 - maxiter2:
                print('Iter: {}'.format(iters + 1))
                print('RMSE: {:.6}'.format(rmse))
                print()

        W = W_plus/maxiter2
        X_new = X_new_plus/maxiter2
        A = A_plus/maxiter2
        mat_hat = mat_hat_plus/maxiter2
        if maxiter1 >= 100:
            final_mape = np.sum(np.abs(dense_mat[pos] - mat_hat[pos])/dense_mat[pos])/dense_mat[pos].shape[0]
            final_rmse = np.sqrt(np.sum((dense_mat[pos] - mat_hat[pos]) ** 2)/dense_mat[pos].shape[0])
            print('Imputation MAPE: {:.6}'.format(final_mape))
            print('Imputation RMSE: {:.6}'.format(final_rmse))
            print()

        return mat_hat, W, X_new, A

    sparse_mat = dfdealMis
    dense_mat = dfdens
    if (np.isnan(sparse_mat).any()==False):
        self.dfsecond = self.dfsecond.reset_index(drop = True)
        pdate = pd.DataFrame(self.dfsecond['date'].values.astype('float32'), columns=['Datefloat'])
        self.dfsecond['Datefloat'] = pdate['Datefloat']
        return self.dfsecond
    start = time.time()
    dim1, dim2 = sparse_mat.shape
    rank = 10
    time_lags = np.array([1, 2, (len(self.dfsecond)//28)])
    init = {"W": 0.1 * np.random.rand(dim1, rank), "X": 0.1 * np.random.rand(dim2, rank)}
    maxiter1 = 1100
    maxiter2 = 100
    a,b,c,d = BTMF(dense_mat, sparse_mat, init, rank, time_lags, maxiter1, maxiter2)
    end = time.time()
    print('Running time: %d seconds'%(end - start))

    a = np.delete(a,-1,axis = 1)
    dfRainfall = self.dfsecond[self.Rain_list].to_numpy()
    dfRainfall = np.delete(dfRainfall,range(len(self.dfsecond)-(len(self.dfsecond)//28)*28),axis = 0)
    dfFlow = self.dfsecond[self.Flow_list].to_numpy()
    dfFlow = np.delete(dfFlow,range(len(self.dfsecond)-(len(self.dfsecond)//28)*28),axis = 0)
    dfTemp = self.dfsecond[self.Flow_list].to_numpy()
    dfTemp = np.delete(dfTemp,range(len(self.dfsecond)-(len(self.dfsecond)//28)*28),axis = 0)
    pdate = pd.DataFrame(self.dfsecond['date'].values.astype('float32'), columns=['Datefloat'])
    dfDate = pdate['Datefloat'].to_numpy()
    dfDate = np.delete(dfDate,range(len(self.dfsecond)-(len(self.dfsecond)//28)*28),axis = 0)
    dfDate = dfDate.reshape(-1,1)
```

```
        a = a.T
        wholedata = np.hstack((a,dfRainfall,dfTemp,dfDate))
        wholelist = self.Depth_list+self.Rain_list+self.Temp_list+['Datefloat']
        newFrame = DataFrame(wholedata,index=None,columns = wholelist)
        self.dfsecond = newFrame
        self.dfsecond = self.dfsecond.reset_index(drop = True)
        return self.dfsecond

    def PCA_trans(self,feature_list):
        if len(feature_list)<=2:
            return self.dfsecond
        test=self.dfsecond[feature_list].ffill().bfill()
        Xpca= PCA(n_components=2).fit_transform(test)
        pf = pd.DataFrame(Xpca, columns=['PCA1','PCA2'])
        self.dfsecond['PCA1'] = pf['PCA1']
        self.dfsecond['PCA2'] = pf['PCA2']
        for i in range(len(feature_list)):
            self.dfsecond = self.dfsecond.drop(feature_list[i],axis=1)
        self.dfsecond = self.dfsecond.ffill().bfill()
        self.PCA_list = ['PCA1','PCA2']
        return self.dfsecond



    #use.shift(-28) made target variable
    def target_made(self,potential_list):
        self.lag_target=[]
        '''
        potential_list contains the name of all the variables which cound be seen as output in this table.
        '''
        for i in range(len(potential_list)):
            name = potential_list[i]+'28'
            self.dfsecond[name] = self.dfsecond[potential_list[i]].shift(-28)
            self.lag_target.append(name)
        return self.dfsecond


    def MICMethod(self):
        '''
        use MICMethod to delete those variables which have higher MIC values with some other variables.
        '''
        mine = MINE(alpha=0.6, c=15)
        deldep_feature =[]
        deltem_feature =[]
        delflow_feature = []

        if((len(self.Depth_list)!=0)&(self.target_list[0] not in self.Depth_list)):
            dataDepth = self.dfsecond[self.Depth_list]
            data_array = np.array(dataDepth)
            n = len(data_array[0, :])
            for i in range(n):
                for j in range(n):
                    mine.compute_score(data_array[:, i], data_array[:, j])
                    if((mine.mic()>=0.9)&(i!=j)):
                        if (self.Depth_list[j] not in deldep_feature):
                            deldep_feature.append(self.Depth_list[i])
                            break
```

```
        if(len(self.Temp_list)!=0):
            dataTem = self.dfsecond[self.Temp_list]
            data_array = np.array(dataTem)
            n = len(data_array[0, :])
            for i in range(n):
                for j in range(n):
                    mine.compute_score(data_array[:, i], data_array[:, j])
                    if((mine.mic()>=0.9)&(i!=j)):
                        if (self.Temp_list[j] not in deltem_feature):
                            deltem_feature.append(self.Temp_list[i])
                            break

        if((len(self.Flow_list)!=0)&(self.target_list[0] not in self.Flow_list)):
            dataflow = self.dfsecond[self.Flow_list]
            data_array = np.array(dataflow)
            n = len(data_array[0, :])
            for i in range(n):
                for j in range(n):
                    mine.compute_score(data_array[:, i], data_array[:, j])
                    if((mine.mic()>=0.9)&(i!=j)):
                        if (self.Flow_list[j] not in delflow_feature):
                            delflow_feature.append(self.Flow_list[i])
                            break

        if len(self.PCA_list):
            datapca = self.dfsecond[['PCA1','PCA2']]
            data_array = np.array(datapca)
            mine.compute_score(data_array[:, 0], data_array[:, 1])
            if(mine.mic()>=0.9):
                delpca_feature = ['PCA2']
        self.dfsecond = self.dfsecond.drop(deldep_feature+deltem_feature+delpca_feature+delflow_feature,axis =1)

        for i in range(len(deldep_feature)):
            self.Depth_list.remove(deldep_feature[i])
        for i in range(len(deltem_feature)):
            self.Temp_list.remove(deltem_feature[i])
        for i in range(len(delflow_feature)):
            self.Flow_list.remove(delflow_feature[i])
        self.PCA_list = ['PCA1']
        return self.dfsecond

    def KalmanCook(self):
        '''
        used Kalman filter to remove noise in Depth_to_Groundwater,flow, and temperature variables.
        '''
        def Kalman1D(data,damping=1):
            observation_covariance = damping
            first_value = data[0]
            transition_matrix = 1
            transition_covariance = 0.1
            first_value
            kf = KalmanFilter(
                    initial_state_mean=first_value,
                    initial_state_covariance=observation_covariance,
                    observation_covariance=observation_covariance,
                    transition_covariance=transition_covariance,
```

```
                transition_matrices=transition_matrix
            )
        pred_state, state_cov = kf.smooth(data)
        return pred_state

    dfreborn = self.dfsecond.drop(self.Depth_list+self.Temp_list+self.Flow_list,axis =1)
    for i in range(len(self.Depth_list)):
        tryArray = self.dfsecond[self.Depth_list[i]].to_numpy()
        trykal = Kalman1D(tryArray,0.1)
        kallist = map(lambda x: x[0], trykal)
        trykalseries = pd.Series(kallist)
        dfreborn[self.Depth_list[i]] = trykalseries

    for i in range(len(self.Temp_list)):
        tryArray = self.dfsecond[self.Temp_list[i]].to_numpy()
        trykal = Kalman1D(tryArray,0.1)
        kallist = map(lambda x: x[0], trykal)
        trykalseries = pd.Series(kallist)
        dfreborn[self.Temp_list[i]] = trykalseries

    for i in range(len(self.Flow_list)):
        tryArray = self.dfsecond[self.Flow_list[i]].to_numpy()
        trykal = Kalman1D(tryArray,0.1)
        kallist = map(lambda x: x[0], trykal)
        trykalseries = pd.Series(kallist)
        dfreborn[self.Flow_list[i]] = trykalseries
    self.dfsecond = dfreborn
    return self.dfsecond,self.lag_target


def LSTMGo(self,target_variable):
    '''
    Target_variable is the name of the variable which would be used as dependent variable in the LSTM model.
    This method will print out the results of the training phase and the test phase, and return the forcast
    results of the last 28 days.
    '''
    self.target_variable = target_variable
    self.n_test = int(((len(self.dfsecond)-28)/28)//5*28)
    self.n_train = int(((len(self.dfsecond)-28)/28)//5*4*28)

    self.dftrain = self.dfsecond[:self.n_train]
    self.dftest = self.dfsecond[self.n_train:self.n_test+self.n_train]

    if(target_variable.startswith('Depth')):
        fake_target = [a for a in self.Depth_list if target_variable.startswith(a)][0]
    else:
        fake_target = [a for a in self.Flow_list if target_variable.startswith(a)][0]

    self.feature_name = ['Datefloat']+self.PCA_list+self.Temp_list+[fake_target]
    dftrainX = self.dftrain[self.feature_name]
    n_feature = len(dftrainX.columns.values.tolist())
    dflistX = np.reshape(dftrainX.values.tolist(),(28,-1,n_feature))

    dftrainY = self.dftrain[target_variable]
    dflistY = np.reshape(dftrainY.values.tolist(),(28,-1,1))
    dflistX = dflistX.astype('float32')
    dflistY = dflistY.astype('float32')
```

```
        tensorx = torch.from_numpy(dflistX)
        tensory = torch.from_numpy(dflistY)

        net = lstm_reg(n_feature, 100)
        criterion = nn.MSELoss()
        optimizer = torch.optim.Adam(net.parameters(), lr=1e-2)

        for e in range(100):
            var_x = Variable(tensorx)
            var_y = Variable(tensory)

            out = net(var_x)
            loss = criterion(out, var_y)

            optimizer.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(net.parameters(), 1.1)#gradient clipping, used to avoid Exploding Gradients

            optimizer.step()
            if (e + 1) % 10 == 0:
                    print('Epoch: {}, Loss: {:.5f}'.format(e + 1, loss.data))


        dftestX = self.dftest[self.feature_name]
        n_feature = len(dftestX.columns.values.tolist())
        dftestlistX = np.reshape(dftestX.values.tolist(),(28,-1,n_feature))

        dftestY = self.dftest[self.target_variable]
        dftestlistY = np.reshape(dftestY.values.tolist(),(28,-1,1))
        dftestlistX = dftestlistX.astype('float32')
        dftestlistY = dftestlistY.astype('float32')
        tensortestx = torch.from_numpy(dftestlistX)
        tensortesty = torch.from_numpy(dftestlistY)
        testvar_x = Variable(tensortestx)
        testvar_y = Variable(tensortesty)

        nettest = net.eval()
        pred_teste = nettest(testvar_x)
        loss = criterion(pred_teste, testvar_y)
        print('Epoch: {}, Loss: {:.5f}'.format('mse', loss.data))

        a = nn.L1Loss()
        maeloss = a(pred_teste, testvar_y)
        print('Epoch: {}, Loss: {:.5f}'.format('mae', maeloss.data))

        dfpre = self.dfsecond.tail(28)
        dfpreX = dfpre[self.feature_name]
        n_feature = len(dfpreX.columns.values.tolist())
        dfprelistX = np.reshape(dfpreX.values.tolist(),(28,-1,n_feature))
        dfprelistX = dfprelistX.astype('float32')
        tensorprex = torch.from_numpy(dfprelistX)
        prevar_x = Variable(tensorprex)
        preY= net(prevar_x)
        return preY


class lstm_reg(nn.Module):
```

```
    def __init__(self, input_size, hidden_size, output_size=1, num_layers=2):
        super(lstm_reg, self).__init__()

        self.rnn = nn.LSTM(input_size, hidden_size, num_layers,dropout = 0.3)
        self.reg = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x, _ = self.rnn(x)
        s, b, h = x.shape
        x = x.view(s*b, h)
        x = self.reg(x)
        x = x.view(s, b, -1)
        return x

    def output_y_hc(self, x, hc):
        y, hc = self.rnn(x, hc)  # y, (h, c) = self.rnn(x)
        s, b, h = y.size()
        y = y.view(s*b, h)
        y = self.reg(y)
        y = y.view(s, b, -1)
        return y, hc
```

In [ ]: