HW3 0416246 王彥茹

Task1.

I choose **historical North Usage** to predict **future North Usage**.
Also, I decided to let the window slide for **1 hour** each time.

I create dataFrame to read the csv file from the datas created before.
Split the data into two parts since there are some missing datas from 1/25~4/19 which may cause some issue when doing the slide window

```
import numpy as np
import pandas as pd

file = "q2_north.csv"

dataset = pd.read_csv(file,header=0, delimiter=',')
dataset['north_power_usage'] = dataset['north_power_usage'].apply(pd.to_numeric)
dataset = dataset.drop(['north_temp'], axis=1)

dataset_p1 = dataset[:2637]
dataset_p2 = dataset[2637:]
dataset_p2 = dataset_p2.reset_index()
dataset_p2
```

Using pd.shift() to do slide window on two parts of the data separately.
Use pd.insert() to combine the datas during each shift. After inserting datas truncated them into 102 (96+6)hours.

```
dataset_p1 = dataset_p1.drop(['w_Time'], axis=1)
dataset_p2 = dataset_p2.drop(['w_Time', 'index'], axis=1)

for i in range(1, 1500):
    dataset_p1.insert(i, i, dataset_p1['north_power_usage'].shift(-i))
for i in range(0, 1500):
    #print(dataset_p2['north_power_usage'].shift(-i))
    dataset_p1.insert(i+1500, i+1500, dataset_p2['north_power_usage'].shift(-i))
dataset_p1
```

```
dataset = dataset_p1[:102]
dataset
```

	north_power_usage	1	2	3	4	5	6	7	8	9	...	2990	
0	875.6	834.2	804.7	786.5	776.0	768.8	800.4	852.0	942.5	1008.1	...	1235.8	1
1	834.2	804.7	786.5	776.0	768.8	800.4	852.0	942.5	1008.1	1062.0	...	1237.8	1
2	804.7	786.5	776.0	768.8	800.4	852.0	942.5	1008.1	1062.0	1088.3	...	1211.0	1
3	786.5	776.0	768.8	800.4	852.0	942.5	1008.1	1062.0	1088.3	1080.8	...	1181.6	1
4	776.0	768.8	800.4	852.0	942.5	1008.1	1062.0	1088.3	1080.8	1099.4	...	1118.4	1
5	768.8	800.4	852.0	942.5	1008.1	1062.0	1088.3	1080.8	1099.4	1084.5	...	1083.6	
6	800.4	852.0	942.5	1008.1	1062.0	1088.3	1080.8	1099.4	1084.5	1067.9	...	959.3	
7	852.0	942.5	1008.1	1062.0	1088.3	1080.8	1099.4	1084.5	1067.9	1047.7	...	902.4	
8	942.5	1008.1	1062.0	1088.3	1080.8	1099.4	1084.5	1067.9	1047.7	1044.5	...	862.9	
9	1008.1	1062.0	1088.3	1080.8	1099.4	1084.5	1067.9	1047.7	1044.5	1055.5	...	827.3	
10	1062.0	1088.3	1080.8	1099.4	1084.5	1067.9	1047.7	1044.5	1055.5	1049.9	...	801.4	
11	1088.3	1080.8	1099.4	1084.5	1067.9	1047.7	1044.5	1055.5	1049.9	1033.3	...	805.4	

this show part of the data

still need to transpose the dataFrame to get the final dataset

```
data = data.transpose()
data
```

0	1	2	3	4	5	6	7	8	9	...	92	93	94	95	96	97	98	99	100	101
875.6	834.2	804.7	786.5	776.0	768.8	800.4	852.0	942.5	1008.1	...	1266.2	1244.0	1223.3	1206.2	1196.6	1154.6	1112.7	1058.7	1015.7	879.0
834.2	804.7	786.5	776.0	768.8	800.4	852.0	942.5	1008.1	1062.0	...	1244.0	1223.3	1206.2	1196.6	1154.6	1112.7	1058.7	1015.7	879.0	839.0
804.7	786.5	776.0	768.8	800.4	852.0	942.5	1008.1	1062.0	1088.3	...	1223.3	1206.2	1196.6	1154.6	1112.7	1058.7	1015.7	879.0	839.0	811.6
786.5	776.0	768.8	800.4	852.0	942.5	1008.1	1062.0	1088.3	1080.8	...	1206.2	1196.6	1154.6	1112.7	1058.7	1015.7	879.0	839.0	811.6	799.8
776.0	768.8	800.4	852.0	942.5	1008.1	1062.0	1088.3	1080.8	1099.4	...	1196.6	1154.6	1112.7	1058.7	1015.7	879.0	839.0	811.6	799.8	802.3

then I've got 3000 rows of datas

2993	1181.6	1118.4	1083.6	959.3	902.4	862.9	827.3	801.4	805.4
2994	1118.4	1083.6	959.3	902.4	862.9	827.3	801.4	805.4	845.8
2995	1083.6	959.3	902.4	862.9	827.3	801.4	805.4	845.8	904.2
2996	959.3	902.4	862.9	827.3	801.4	805.4	845.8	904.2	980.0
2997	902.4	862.9	827.3	801.4	805.4	845.8	904.2	980.0	1044.4
2998	862.9	827.3	801.4	805.4	845.8	904.2	980.0	1044.4	1104.2
2999	827.3	801.4	805.4	845.8	904.2	980.0	1044.4	1104.2	1117.2

3000 rows × 102 columns

Task2.

Split the dataset into training dataset and testing dataset (70%, 30%) by using **numpy.random.rand()**

```
def get_train_test(df, y_col, ratio):
    mask = np.random.rand(len(df)) < ratio
    df_train = df[mask]
    df_test = df[~mask]

    Y_train = df_train[y_col].values
    Y_test = df_test[y_col].values
    df_train = df_train.drop(df_train.columns[y_col],axis=1)
    df_test = df_test.drop(df_test.columns[y_col],axis=1)
    X_train = df_train.values
    X_test = df_test.values
    return X_train, Y_train, X_test, Y_test

y_col = [96, 97, 98, 99, 100, 101]
train_test_ratio = 0.7
X_train, Y_train, X_test, Y_test = get_train_test(data, y_col, train_test_ratio)

print(X_train.shape)
print(Y_train.shape)
print(X_test.shape)
print(Y_test.shape)
```

```
(2081, 96)
(2081, 6)
(919, 96)
(919, 6)
```

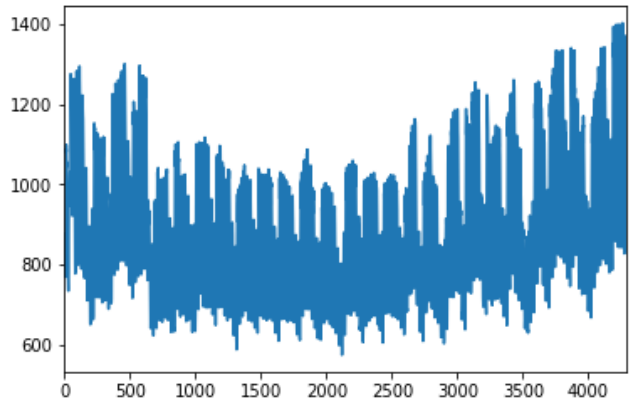
Task3.

First, do the labeling.
Draw the figure of the data to decide how to do the labels

I choose the value 600, 800, 1000, 1200 to make 5 labels, which are **>1200**, **1000~1200**, **800~1000**, **600~800**, **<600**.

```
import matplotlib.pyplot as plt

raw_data['north_power_usage'].plot()
plt.show()
```



```
Y_train_label = pd.DataFrame(data=Y_train)
Y_test_label = pd.DataFrame(data=Y_test)

def discretization(value):
    result = ''
    if value > 1200 :
        result = '>1200'
    elif value > 1000:
        result = '1000~1200'
    elif value > 800:
        result = '800~1000'
    elif value > 600:
        result = '600~800'
    else :
        result = '<600'
    return result

for i in range(0,6):
    Y_train_label[i] = Y_train_label[i].apply(lambda x: discretization(x))
    Y_test_label[i] = Y_test_label[i].apply(lambda x: discretization(x))
```

what do the partial Y_train data looks like after being labeled

	0	1	2	3	4	5
0	1000~1200	1000~1200	1000~1200	1000~1200	800~1000	800~1000
1	1000~1200	1000~1200	1000~1200	800~1000	800~1000	800~1000
2	1000~1200	1000~1200	800~1000	800~1000	800~1000	600~800
3	1000~1200	800~1000	800~1000	800~1000	600~800	800~1000
4	800~1000	600~800	800~1000	800~1000	800~1000	800~1000
5	600~800	800~1000	800~1000	800~1000	800~1000	1000~1200
6	800~1000	800~1000	800~1000	800~1000	1000~1200	>1200
7	800~1000	800~1000	1000~1200	>1200	>1200	>1200
8	800~1000	1000~1200	>1200	>1200	>1200	>1200
9	>1200	>1200	>1200	>1200	>1200	>1200
10	>1200	>1200	>1200	>1200	>1200	>1200
11	>1200	>1200	>1200	>1200	>1200	>1200
12	>1200	>1200	>1200	>1200	>1200	1000~1200
13	>1200	>1200	>1200	>1200	1000~1200	1000~1200

Classification

K-Nearest-Neighbor

The accuracy of the prediction depends on the n_neighbors's value. It should be as big that noises won't affect the prediction highly. And as low that one factor won't dominate another.

I've tried plenty of the n_neighbors values and found that when the values is about **8** the accuracy is highest(see the picture).
After setting the values higher than 15 , the accuracy starts to drop.
The accuracy is around 75% when the n_neighbors is 20; the accuracy is around 70% when it is 80; and it is around 65% accuracy when setting the value to 200.
(six rows of accuracy represent the accuracy of each predicted hour)

```
# K-Nearest-Neighbor
import time
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier
start = time.time()
neigh = KNeighborsClassifier(n_neighbors=8)
neigh.fit(X_train,Y_train_label)
Y_pred = neigh.predict(X_test)
end = time.time()

for i in range(0,6):
    test_tmp = Y_test_label.iloc[:, i]
    pred_tmp = Y_pred[:, i]
    acc = accuracy_score(test_tmp, pred_tmp)
    print(acc)

print('execution time {:.3f} s'.format(end-start))
```

```
0.802150537634
0.786021505376
0.788172043011
0.767741935484
0.77311827957
0.754838709677
execution time 0.725 s
```

Naïve Bayes

Since sklearn.naive_bayes itself doesn't support multioutput classifier, I use sklearn.multioutput. MultiOutputClassifier to help finishing the classification.

There are three types of models for Naïve Bayes classifiers which are Gaussian Naive Bayes, multivariate Bernoulli models and multinomial models.

```
from sklearn.multioutput import MultiOutputClassifier
from sklearn.naive_bayes import GaussianNB
start = time.time()
classifier = MultiOutputClassifier(GaussianNB())
classifier.fit(X_train, Y_train_label)
Y_pred = classifier.predict(X_test)
end = time.time()
#predictions = classifier.predict(X_test)
#accuracy_score(Y_test_label,predictions)
for i in range(0,6):
    test_tmp = Y_test_label.iloc[:, i]
    pred_tmp = Y_pred[:, i]
    acc = accuracy_score(test_tmp, pred_tmp)
    print(acc)
print('execution time {:.3f} s\n'.format(end-start))
```

```
0.652667423383
0.640181611805
0.619750283768
0.61180476731
0.618615209989
0.614074914869
execution time 0.191 s
```

```
from sklearn.multioutput import MultiOutputClassifier
from sklearn.naive_bayes import BernoulliNB
start = time.time()
BNB = MultiOutputClassifier(BernoulliNB())
BNB.fit(X_train, Y_train_label)
Y_pred = BNB.predict(X_test)
end = time.time()

for i in range(0,6):
    test_tmp = Y_test_label.iloc[:, i]
    pred_tmp = Y_pred[:, i]
    acc = accuracy_score(test_tmp, pred_tmp)
    print(acc)
print('execution time {:.3f} s\n'.format(end-start))
```

```
0.360953461975
0.368898978434
0.367763904654
0.371169125993
0.377979568672
0.362088535755
execution time 0.145 s
```

As we can see from the pictures, the **GaussianNB** gives the **highest accuracy**, the **BernoulliNB** gives the **lowest accuracy** and the **MultinomialNB** gives the **shortest execution time**.
Hence, in this case using **GaussianNB** can get the best performance in my opinion.

```
from sklearn.multioutput import MultiOutputClassifier
from sklearn.naive_bayes import MultinomialNB
start = time.time()
MNB = MultiOutputClassifier(MultinomialNB())
MNB.fit(X_train, Y_train_label)
Y_pred = MNB.predict(X_test)
end = time.time()
for i in range(0,6):
    test_tmp = Y_test_label.iloc[:, i]
    pred_tmp = Y_pred[:, i]
    acc = accuracy_score(test_tmp, pred_tmp)
    print(acc)
print('execution time {:.3f} s\n'.format(end-start))
```

```
0.44494892168
0.50056753689
0.451759364359
0.49829738933
0.518728717367
0.44608399546
execution time 0.083 s
```

Random Forest

Since the sklearn.ensemble. RandomForestClassifier need the labels to be float type, so I revised the original string type label into integer type.

```
def discretization(value):
    result = ''
    if value > 1200 :
        result = 5
    elif value > 1000:
        result = 4
    elif value > 800:
        result = 3
    elif value > 600:
        result = 2
    else :
        result = 1
    return result
```

	0	1	2	3	4	5
0	4	4	4	4	4	3
1	3	3	3	2	3	3
2	3	3	2	3	3	3
3	3	2	3	3	3	3
4	2	3	3	3	3	4

```python
import time
from sklearn.metrics import accuracy_score
from sklearn.ensemble import RandomForestClassifier
param_test1 = range(1,11,1)
for param in param_test1:
    print('max_depth = ',param)
    start = time.time()
    rf = RandomForestClassifier(max_depth=param)
    rf.fit(X_train, Y_train_label)
    Y_pred = rf.predict(X_test)
    end = time.time()
    for i in range(0,6):
        test_tmp = Y_test_label.iloc[:, i]
        pred_tmp = Y_pred[:, i]
        acc = accuracy_score(test_tmp, pred_tmp)
        print(acc)
    print('execution time {:.3f} s\n'.format(end-start))
```

I test the model with different max_depth value and see the accuracy.
It's obvious the accuracy is low, when the max_depth is small.

I've found that setting the max_depth value **around 20** gives a good result of classification

Also, setting the max_depth higher cause longer execution time. But it won't go too high even though setting a rather bigger value.(see the picture below)

```
max_depth =  100
0.86562150056
0.852183650616
0.8320268757
0.814109742441
0.812989921613
0.796192609183
execution time 1.614 s
```

```
max_depth =  1
0.58790593505
0.603583426652
0.63829787234
0.643896976484
0.4960806271
0.477043673012
execution time 0.396 s

max_depth =  2
0.674132138858
0.662933930571
0.650615901456
0.642777155655
0.652855543113
0.668533034714
execution time 0.455 s

max_depth =  3
0.701007838746
0.709966405375
0.683090705487
0.670772676372
0.65509518477
0.632698768197
execution time 0.378 s
```

```
max_depth =  20
0.868980963046
0.833146696529
0.844344904815
0.810750279955
0.826427771557
0.802911534155
execution time 1.702 s

max_depth =  21
0.881298992161
0.867861142217
0.849944008959
0.819708846585
0.795072788354
0.810750279955
execution time 1.962 s

max_depth =  22
0.858902575588
0.857782754759
0.847704367301
0.814109742441
0.789473684211
0.795072788354
execution time 1.682 s
```

Support vector machine(SVC)

Same as naïve bayes, the SVC itself doesn't support multioutput classifier, so I use MultiOutputClassifier to deal with this problem.

```
from sklearn.svm import SVC
start = time.time()
svc = MultiOutputClassifier(SVC())
svc.fit(X_train, Y_train_label)
Y_pred = svc.predict(X_test)
end = time.time()

for i in range(0,6):
    test_tmp = Y_test_label.iloc[:, i]
    pred_tmp = Y_pred[:, i]
    acc = accuracy_score(test_tmp, pred_tmp)
    print(acc)
print('execution time {:.3f} s\n'.format(end-start))
```

```
0.360953461975
0.368898978434
0.367763904654
0.371169125993
0.377979568672
0.362088535755
execution time 21.207 s
```

```
from sklearn.svm import SVC
start = time.time()
svc = MultiOutputClassifier(SVC(kernel='sigmoid'))
svc.fit(X_train, Y_train_label)
Y_pred = svc.predict(X_test)
end = time.time()

for i in range(0,6):
    test_tmp = Y_test_label.iloc[:, i]
    pred_tmp = Y_pred[:, i]
    acc = accuracy_score(test_tmp, pred_tmp)
    print(acc)
print('execution time {:.3f} s\n'.format(end-start))
```

```
0.360953461975
0.368898978434
0.367763904654
0.371169125993
0.377979568672
0.362088535755
execution time 7.837 s
```

There is several kernel type to choose from when using SVC, including **linear**, **sigmoid**, **poly**, and **rbf**.
After testing each kind of kernel, I think that setting **kernel='poly'** gives the best result. It has the highest accuracy but quite a long execution time.

```
svc = MultiOutputClassifier(SVC(kernel='poly'))
svc.fit(X_train, Y_train_label)
Y_pred = svc.predict(X_test)
end = time.time()
for i in range(0,6):
    test_tmp = Y_test_label.iloc[:, i]
    pred_tmp = Y_pred[:, i]
    acc = accuracy_score(test_tmp, pred_tmp)
    print(acc)
print('execution time {:.3f} s\n'.format(end-start))
```

```
0.878547105562
0.849035187287
0.811577752554
0.791146424518
0.803632236095
0.784335981839
execution time 131.368 s
```

Regression

Bayesian regression

BayesianRidge cannot perform multioutput regression so I used MultiOutputRegressor to handle it.
I've tried to change the n_iter value, but the accuracy doesn't change a lot. I'm not sure how to set the other parameters could give a better result since most of them is defaulted exponential.
Also, I think that the default settings is just fine in this case.

```
import time
from sklearn.multioutput import MultiOutputRegressor
from sklearn.linear_model import BayesianRidge
start = time.time()
regr = MultiOutputRegressor(BayesianRidge())
regr.fit(X_train, Y_train)
acc = regr.score(X_test,Y_test)
end = time.time()

print('accuracy : %f' % acc)
print('execution time {:.3f} s'.format(end-start))
```

```
accuracy : 0.914526
execution time 0.203 s
```

Decision tree regression

I've tried different value of max_depth. The accuracy isn't too high when the max_depth value is small. After a few tests, I found that set the max_depth to 10 gives the best performance. It's accuracy is high and the execution time isn't too long. Setting the value bigger than 10 doesn't really gets higher accuracy but definitely longer execution time.

```
#Decision Tree Regression
from sklearn.tree import DecisionTreeRegressor
start = time.time()
dtr = DecisionTreeRegressor(max_depth = 10)
dtr.fit(X_train, Y_train)
acc = dtr.score(X_test,Y_test)
end = time.time()

print('accuracy : %f' % acc)
print('execution time {:.3f} s'.format(en
```

```
accuracy : 0.895479
execution time 1.011 s
```

```
dtr = DecisionTreeRegressor(max_depth = 25)
dtr.fit(X_train, Y_train)
acc = dtr.score(X_test,Y_test)
end = time.time()

print('accuracy : %f' % acc)
print('execution time {:.3f} s'.format(end-start))
```

```
accuracy : 0.889871
execution time 1.240 s
```

Support vector machine(SVR)

Use the MultiOutputRegressor to solve the problem that SVR doesn't support multioutput regression.
I've tried different C values, epsilon values and every type of kernel, but the accuracy is still negative.
I thought that maybe set the kernel to 'poly' and gives the right degree can get a better result. Unfortunately, it couldn't finish executing although I gave it 10 minutes.
Hence, I don't think it's a good idea to use SVR in this case.

```
from sklearn.multioutput import MultiOutputRegressor
from sklearn.svm import SVR
start = time.time()
svr = MultiOutputRegressor(SVR(C=0.8, epsilon=0.5))
svr.fit(X_train, Y_train)
acc = svr.score(X_test,Y_test)
end = time.time()

print('accuracy : %f' % acc)
print('execution time {:.3f} s'.format(end-start))
```

```
accuracy : -0.014670
execution time 11.260 s
```