

COURSEWORK 2 - SECURITY OF CYBER PHYSICAL SYSTEMS

Student id: 32390327

Table of contents

Introduction	1
Classification	1
First attempts	2
SVM classifier	4
How does SVM work?	4
My final decision - SVM + total billing feature	4
Why SVM classifier?	6
Solution's Code explanation	6
Linear Programming (LP) - solution	7
Testing Data - Classification, Analysis & Energy scheduling	10
Analysing Abnormal Curves - Testing Data	12
Results' energy plots	15
Appendix	44

Introduction

This report provides information about how we can identify a possible abnormal guideline price curve by implementing Machine Learning Classification techniques and Linear Programming solutions. The context of the report is related to energy consumption manipulation. More precisely, in the first approach, cyber attackers try to reduce their energy cost by creating fake predictive guideline prices. These guidelines include artificial pricing peaks in periods in which the cyber attacker wants to consume more and pay less. The smart meters read an incorrect guideline price and schedule their tasks according to it. Therefore, the attacker's energy consumption costs less since the community does not consume the same time period when fake high prices exist, and the actual real price is going lower. The second approach of a cyberattack is related to overload. Here, the cyber attacker tries to create low price periods to lead the community to consume more in a specific time window. If the community's smart meters read that the price is very low, they will execute their task simultaneously that will overload the power grid, causing a cascading disruption.

Github link: <https://github.com/AnastasisDasyras/IdentiftAbnormalEnergyScheduling>

In this report, I will discuss my results, the methods I used to find my solution, and provide the required files for the implementation. The files of the project are:

- **Final.ipynb** which includes all the coding parts. The final code is on the first cell and below there are the extra code blocks from different attempts.
- **TotalBillPerCurve.txt** which includes the total billing information according to each curve of the training dataset. This was used as an extra feature to the classification model.
- **total_bill_results_per_curve.txt** which contains the total billing information according to each curve of the testing dataset. This was used as an extra feature during prediction of the results.
- **TestingData.txt** which includes the curves that need to be classified
- **TrainingData.txt** which consists of the training data for classification
- **COMP3217CW2Input.xlsx** which includes the information about the tasks' execution
- **TestingResults.txt** is the output file that includes the labels of testing data
- **environment.yml** which includes the dependencies for setting the correct environment and reproduce the results

I worked with the Anaconda framework on this report, and I used the Jupyter Notebook to run my code. All the dependencies needed can be found inside the environment.yml file. You can create the appropriate environment quickly by installing the dependencies with the command `conda env create -f environment.yml`

Classification

First attempts

My first attempts were not related to ML techniques. In the beginning, I tried to classify the curves by using line similarities and points' distance metrics. The challenge in these approaches was to define a curve that will be used as the threshold in the classification. My first simple thought to find this “magic” curve was to calculate the mean of 3000 good and bad curves accordingly. The result of this idea is shown below ([implemented code](#)).

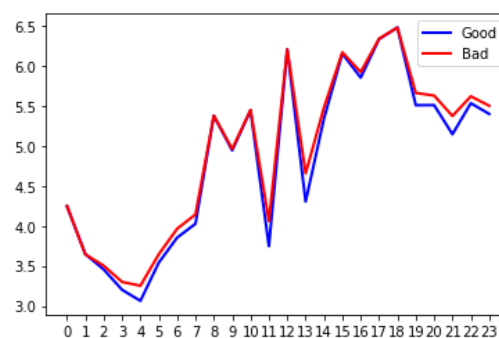


Figure 1: Mean Good and Bad curves for classification

As we can observe, the 2 curves were almost identical. However, I tried different line similarities metrics, which returns zero if 2 curves are similar such as:

- **Partial Curve Mapping (PCM)** matches the area between the subsets of 2 curves
- **Area method:** Calculates the area between 2 curves in 2D.
- **Discrete Fréchet distance:** Calculates the shortest distance between curves
- **Curve Length:** Calculates the differences between 2 curves based on the arc length of each curve
- **Dynamic Time Warping (DTW):** A non-metric distance between 2 time-series curves

[The accuracy for each metric](#) calculated based on the mean curves was:

```
PCM Accuracy: 0.5
Discrete Frechet distance Accuracy: 0.7209
Area Between 2 Curves Accuracy: 0.8686
Curve Length based similarity measure Accuracy: 0.604
Dynamic Time Warping distance Accuracy: 0.872
```

Figure 2: Accuracy from line similarities metrics

Although the DTW similarity measure returned satisfying accuracy, I continued to search to increase it. The next attempt was related to the max distance between 2 points from each curve and the mean curves. I worked with [points distances](#) such as Euclidean distance, Manhattan distance, Chebyshev distance, Canberra distance and Cosine distance. Still, the two means curves were very close to each other, and the results of these approaches were

close to 50-55% accuracy, which was not acceptable. Therefore, I decided to change the way I calculated the "magic" curve and tried to use Support Vector Regression (SVR). SVR uses the same principle as SVM but for regression problems that can predict a line. I used this technique to predict a line according to the data that we have from the dataset. The problem with this approach was that for each column (hour) of the training data, all the values need to be inputted into the model, which means that the total input data was equal to 72000 since I used only 3000 curves with 24 columns.

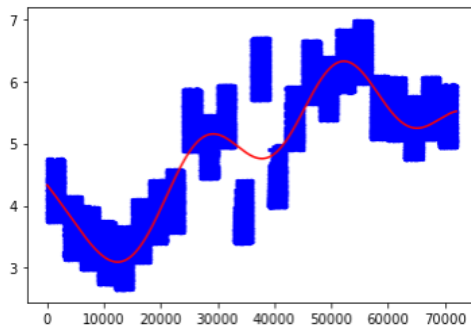


Figure 3: SVR predicted curve - 3000 normal curves

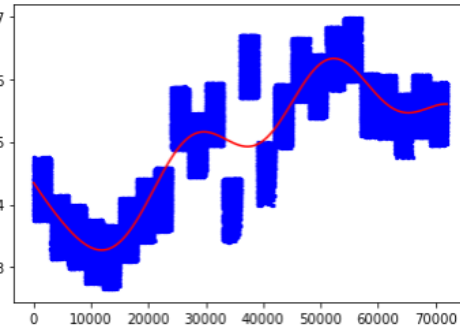


Figure 4: SVR predicted curve - 3000 abnormal curves

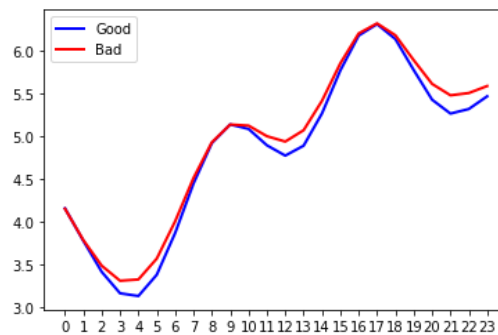


Figure 5: SVR predicted normal and abnormal curves in scale 24 hours

In figure 5, the predicted curves are again very similar. Only the average cost per hour in the normal curve is lower in specific periods during the day. However, these small changes did not increase the previous techniques' accuracy because the similarities between lines and points are very close. So, I decided to focus on Machine Learning approaches.

The first attempt from ML approaches was the Decision Tree classifier since it is a fast classifier and can handle multiple features. However, the classification process returned a low accuracy close to 77%. [The implemented code](#) can be found in the appendix.

Decision Tree Accuracy: 0.7732

Figure 6: Decision Tree accuracy

I continued my research, and I found that for multiple features, the best classifiers are the Support Vector Machine and the XGBoost. Firstly, I worked with [XGBooster](#), which returned very good accuracy in the classification. I separated 10000 data from the dataset to

7500 training and 2500 testing data. The results are pictured in figure 7. Although the results were impressive, I also wanted to check the SVM classifier to take my final decision.

```
xgboost Accuracy: 0.94
xgboost precision_recall_fscore (0.9372019077901431, 0.9432, 0.9401913875598087, None)
```

Figure 7: XGBoost classifier results

SVM classifier

How does SVM work?

SVM uses hyperplanes to separate the data. More precisely, Hyperplane is a boundary that the SVM model tries to find to classify the data correctly. The best Hyperplane, which is selected, is the one that maximizes the margin between the classified tags. Therefore, if we have 2-dimensional data there might be a linear hyperplane that will be able to classify our data correctly. However, in our example we have 24 different columns which means that the classifier needs to find a solution in a 24-dimensional problem. These problems are impossible to be plotted to visualize the results. In addition, SVM supports hyperparameters that help researchers to tune their model into their data. For instance, if the data that are used for training are not linear, it might be better to use a polynomial kernel to find the best solution in a multidimensional problem.

My final decision - SVM + total billing feature

After the aforementioned attempts, I focused on SVM. I used Python programming language and the sklearn library to build my model. I separated the dataset again in the same way as before to 7500 for training and 2500 for testing data. During my interaction with the SVM model, I tuned the hyperparameters to tailor the behaviour of the algorithm to my dataset. To define the best combination of parameters, I used the [GridSearchCV](#) function that gets as input predefined values for hyperparameters and calculates the results of their combination. The results of the GridSearchCV function are pictured in the image below, in which we can see that the classification accuracy is over 95% in many cases.

```
Best: 0.964833 using {'C': 50, 'gamma': 'scale', 'kernel': 'poly'}
0.937333 (0.009676) with: {'C': 0.01, 'gamma': 'scale', 'kernel': 'poly'}
0.512000 (0.003372) with: {'C': 0.01, 'gamma': 'scale', 'kernel': 'rbf'}
0.932333 (0.009941) with: {'C': 0.01, 'gamma': 'scale', 'kernel': 'linear'}
0.952222 (0.007218) with: {'C': 1, 'gamma': 'scale', 'kernel': 'poly'}
0.943278 (0.008740) with: {'C': 1, 'gamma': 'scale', 'kernel': 'rbf'}
0.937333 (0.009414) with: {'C': 1, 'gamma': 'scale', 'kernel': 'linear'}
0.964000 (0.006185) with: {'C': 10, 'gamma': 'scale', 'kernel': 'poly'}
0.957222 (0.007495) with: {'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}
0.938333 (0.008788) with: {'C': 10, 'gamma': 'scale', 'kernel': 'linear'}
0.964611 (0.007132) with: {'C': 20, 'gamma': 'scale', 'kernel': 'poly'}
0.961056 (0.007509) with: {'C': 20, 'gamma': 'scale', 'kernel': 'rbf'}
0.937944 (0.009125) with: {'C': 20, 'gamma': 'scale', 'kernel': 'linear'}
0.964833 (0.007750) with: {'C': 50, 'gamma': 'scale', 'kernel': 'poly'}
0.963333 (0.007328) with: {'C': 50, 'gamma': 'scale', 'kernel': 'rbf'}
0.938278 (0.008803) with: {'C': 50, 'gamma': 'scale', 'kernel': 'linear'}
```

Figure 8: GridSearchCV results for the best hyperparameters

The most significant parameters on the SVM models are the kernel, the C (penalty) parameter and the gamma parameter. I set the gamma parameter to default and combined different kernels with different C values. From the results, we can see that the best option is the polynomial kernel with C=50. The C parameter defines how much we want the SVM classifier to avoid misclassification on training. Small values of C lead the classifier to look for larger-margin hyperplanes, whereas high values of C leads to smaller-margin hyperplanes. A high C value concludes to overfitting the model in the training data, so I will not follow the best option, which might overfit my training data, but I will reduce the C parameter to 20 that also returns good results on the classification. On the other hand, the kernel also plays a vital role in our model. It is worth noting that the polynomial kernel looks at the given features and their combination, which is needed in our example, so I selected this one to be the model's kernel. The result of this model is shown below.

```
SVM Accuracy: 0.9616
SVM precision_recall_fscore (0.9623397435897436, 0.9608, 0.9615692554043235, None)
```

Figure 9: SVM model metrics (C=20, gamma='scale',kernel='poly')

Furthermore, we need also to check the precision, recall and F-score of our model that are also figured in the image below. It is clear that the accuracy returns how accurate the results are in comparison with the real labels. Precision quantifies the positives prediction that actually belongs to the positive class, whereas Recall quantifies the positives predictions compared to the total positive samples in the dataset, and F-score tries to balance the precision and recall scores by providing one single number. The function of each value can be calculated with the equations below.

$$Accuracy = \frac{True\ Positives + True\ Negatives}{True\ Positives + True\ Negatives + False\ positives + False\ Negatives}$$
$$Precision = \frac{True\ Positives}{True\ Positives + False\ positives}$$
$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$
$$Fscore = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Figure 10: Mathematical equations of main metrics

It is evident that the classifier returns good results during testing since it returned high values in each metric (e.g., Accuracy_test, precision_recall_fscore_testing)

Moreover, I wanted to find out the result if I add an extra feature that I calculated from the Linear programming solution. The feature was the total energy bill of the community according to each curve. I calculated and stored all the total bills per curve in a file since it

was very time consuming to find the results for 10000 curves. Then I created an extra column to my dataset, and I inserted the total bill of each curve.

5	6	7	8	9	...	16	17	18	19	20	21	22	23	Total_bill	24
3.330895	3.723629	3.709737	5.848922	5.252491	...	6.200070	5.842163	6.148899	5.333915	5.385360	4.851503	5.118068	5.113795	418.629398	0
3.105662	3.801500	3.796322	5.062057	4.987285	...	5.749618	6.747027	6.537529	5.186155	5.420567	4.777363	5.120788	5.136329	420.418943	0
3.133526	3.635403	3.581440	5.607975	4.847624	...	5.844224	6.073829	6.075009	5.572521	5.194047	4.771206	5.424286	5.628824	420.947460	0
3.311229	3.855523	3.877643	5.054537	4.982864	...	5.531023	5.888425	6.627419	6.000076	5.059838	4.898310	5.887313	5.005183	423.164833	0

Figure 11: Inserted the total bill for each curve

I used this dataset for training and testing, and the final result of the classification model was astonished. As we can see in figure 12, the accuracy is close to 100%, and I will explain in the analysis part how this was achieved. Therefore, [the final model](#) I used for classification was trained on the figure 11 dataset and has an accuracy of 99,92% at testing data.

Accuracy_Training: 0.9992
precision_recall_fscore_training (1.0, 0.9984, 0.9991993594875901, None)

Figure 12: SVM model (C=20, gamma='scale', kernel='poly') plus total bill feature

Why SVM classifier?

In my final approach, using the feature of total bill per curve it was possible to create the classification model without implementing machine learning techniques. However, I wanted the model to have the capability of future extension such as adding or removing some features and the SVM model simplifies the process. Moreover, SVM handles with great accuracy and efficiency multiple features which is better than any other implementation which will be created from scratch.

Solution's Code explanation

I will explain my implementation's coding solution to be more precise how I classified the testing data. In the beginning, I read and stored the training data in a dataframe. Also, I read from the file the LP solution related to the total bill and inserted it in the dataset.

```
#Read file - insert bill feature in dataset
dataset = pd.read_csv('coursework/TrainingData.txt', header = None)
total_bill = pd.read_csv('TotalBillPerCurve.txt', header = None)
dataset.insert(24, column='Total_bill', value=total_bill)
```

Figure 13: Read useful information from files (total bill and training data)

To continue, I split the training dataset to avoid overfitting during training. I used the train_test_split function of sklearn to train my model based on the 7500 data (¾). I also added the stratify variable to keep the same analogy of normal and abnormal curves in both testing and training data.

```
#split dataset
X_train, X_test, y_train, y_test = train_test_split(dataset.drop(24, axis=1), dataset.iloc[:, -1:],
                                                    test_size=0.25, stratify=dataset.iloc[:, -1:])
```

Figure 14: Split data to create the classifier and test it

Finally, I set up the classifier using the best hyperparameters such as polynomial kernel and C=20 and calculated the accuracy by using the sklearn metrics library. Finally, I figured the testing data results and wrote the labels on the file TestingResults.txt.

```
#split dataset
X_train, X_test, y_train, y_test = train_test_split(dataset.drop(24, axis=1), dataset.iloc[:, -1:],
                                                    test_size=0.20, stratify=dataset.iloc[:, -1:])
#Create a svm Classifier - polynomial kernel returns the best results
clf = svm.SVC(kernel='poly', C=20)
#Train the model using the training sets
clf.fit(X_train, y_train.values.ravel().tolist())
#Predict the response for test dataset
y_pred_svm = clf.predict(X_test)
# Model Accuracy training:
print("Accuracy Training:", metrics.accuracy_score(y_test, y_pred_svm))
PRF = precision_recall_fscore_support(y_test, y_pred_svm, average='binary')
print('precision_recall_fscore_training', PRF)
#predict the real TestingData.txt
dataTest = pd.read_csv('TestingData.txt', header = None)
total_bill_test = pd.read_csv('total_bill_results_per_curve.txt', header = None)
dataTest.insert(24, column='Total_bill_test', value=total_bill_test)
#classify testing data
predictions_Test = clf.predict(dataTest)
#print results
print (predictions_Test)
#create file TestingResults.txt
#read data
write_data = []
with open('TestingData.txt', 'r') as f:
    for i, line in enumerate (f.readlines(), start=0):
        line = line + ',' + str(predictions_Test[i])
        write_data.append(line)
#write data
with open('TestingResults.txt', 'w') as f:
    for item in write_data:
        f.write("%s\n" % item)
```

Figure 15: Model classifier and print the results.

Linear Programming (LP) - solution

In the previous chapter, I mentioned that I used the LP solution of each curve to classify the testing data. My LP solver was created using the Python programming language and the `lpsolve55` library, which offers the capability to build the LP problem and find the solution through python or pass the problem's structure to a file that is readable by LPsolver IDE. I created 2 different programmes, [the first one](#) calculates the total bill information and of the community per curve, and [the second](#) calculates the billing information and tasks' execution time per user per curve. I followed this strategy because breaking down the solution for each user was easier to analyse an abnormal behaviour and calculate the total energy consumption per hour. Therefore, in the final implementation I used the second solution to plot the results.

In both cases, the linear programming-based scheduling algorithm works by reading all the information related to the tasks from the excel file. This information includes the number of the tasks, their execution time window and the maximum energy consumption per hour.

```
def LPsol(curve):
    #READ TRAINING DATASET FROM FILE
    dataset = pd.read_csv('coursework/TrainingData.txt', header = None)

    #READ TASKS THAT WILL BE EXECUTED
    xls = pd.ExcelFile('coursework/COMP3217CW2Input.xlsx')
    df1 = pd.read_excel(xls, 'User & Task ID')

    #SET CONSTANTS OF THE PROBLEM
    users_num = 5
    hours = 24
    tasks = df1['User & Task ID'].values.ravel().tolist()
    tasks_num = len(tasks)
```

Figure 16: Read LP solution data

The next step is to create all possible variables and store them in an array. The total number of variables is 1200 since we have 5 users * 10 tasks each * 24 possible execution hours. However, we will not need all of them in the solution, so the next step is to keep only the variables included in the execution time window of the tasks and their related cost that we can find through the input curve.

```
#CREATE ALL POSSIBLE VARIABLES - EACH USER HAS 10 TASKS WITH 24 HOURS WINDOW TO BE EXECUTED -> 5 * 10 * 24
var_names = [i+'_'+hour+'_'+str(j) for i in tasks for j in range(hours)]

#KEEP ONLY THE VARIABLES THAT ARE IN THE TIME WINDOW OF EXECUTION
executed_variables = []
#KEEP THE VALUES OF THE COST PER HOUR FOR EACH VARIABLE - OBJECTIVE FUNCTION
obj_row = []
#KEEP VALUES (0,1) TO SET THE CONSTRAINTS BELOW PER TASK
constraint_energy_demand = []
for i in tasks:
    for j in range(hours):
        #FIND THE TASK'S ROW AND CHECK THE TIME WINDOW
        if (df1.loc[df1['User & Task ID'] == i]['Ready Time'].values <= j
            <= df1.loc[df1['User & Task ID'] == i]['Deadline'].values):
            executed_variables.append(i+'_'+hour+'_'+str(j))
            #KEEP PRICE OF THE EXECUTION TIME
            obj_row.append(curve[j])
            #KEEP THE POSITION OF THE VARIABLE THAT WILL BE EXECUTED
            constraint_energy_demand.append(1)
        else:
            constraint_energy_demand.append(0)
```

Figure 17: Keep variables for the objective function and the constraints

All this information is needed to set up the objective function. To continue, we need to create the lp problem and insert the constraints. We have 2 types of constraints, the first one is related to the total amount of energy needed and the energy consumption per hour. The latter is easier to be defined since we already have the variables that will be used, and we know that the acceptable max energy consumption per hour is 1. The total amount of energy constraint

needs to be defined by acquiring the proper value from the excel file and adding all the variables related to one specific task.

```
#start creating the LP problem
#define the length of rows and cols, rows will be added later now are zero
lp = lpsolve('make_lp', 0, len(executed_variables))
#only important errors should be shown
lpsolve('set_verbose', lp, IMPORTANT)
#set name on variables to be readable
for i in range(len(executed_variables)):
    lpsolve('set_col_name', lp, i+1, executed_variables[i])
#set objective function and we want to minimize it
lpsolve('set_obj_fn', lp, obj_row)
lpsolve('set_minim', lp)

#for each variable set 0 <= var <= 1 - maximum energy per hour
for i in range(len(executed_variables)):
    values = np.zeros(len(executed_variables))
    values[i] = 1
    lpsolve('add_constraint', lp, values, 'LE', 1.00)
    lpsolve('add_constraint', lp, values, 'GE', 0.00)

#for each task send energy demand in total
#reshape the constraint_energy_demand because it is a continuous list and we want to break it in 24 items lists
constraint_energy_demand = np.array(constraint_energy_demand).reshape(tasks_num, hours)
start = 0
end = sum(constraint_energy_demand[0])
for i, task in enumerate(tasks, start=0):
    temp = np.zeros(len(executed_variables))
    temp[start:end] = 1
    if(i != tasks_num-1):
        end = sum(constraint_energy_demand[i+1]) + end
    start = sum(constraint_energy_demand[i]) + start
    energy_demand = df1.loc[df1['User & Task ID'] == task]['Energy Demand'].values
    lpsolve('add_constraint', lp, temp, 'EQ', energy_demand)
```

Figure 18: Setup the constraints of our LP solution

Finally, we can solve the problem by defining that we want to minimise the objective function and call the command `lpsolve('solve', lp)` and `lpsolve('get_objective', lp)` to get the objective function.

```
#solve the LP problem
lpsolve('solve', lp)
#get optimal billing
bill = lpsolve('get_objective', lp)
community_bills.append(bill)
```

Figure 18: Solve LP problem keep the total bill

In figures 16-18, I described how the first LP coding solution works and calculates the total bill for all the community's members. [The second LP](#) coding approach follows the same idea with slight differences and can be found in the appendix. Finally, to get a visual result of the linear programming-based scheduling algorithm, we can see how the structure is shown in the LPSolver IDE, which is more comprehensive. In figures 19 and 20, we can see the output of the linear programming-based scheduling algorithm for user 5. The objective function includes only the variables related to the execution time window. Each variable should be equal or greater to zero and less than or equal to 1. The total execution time of one task should be equal to the required energy.

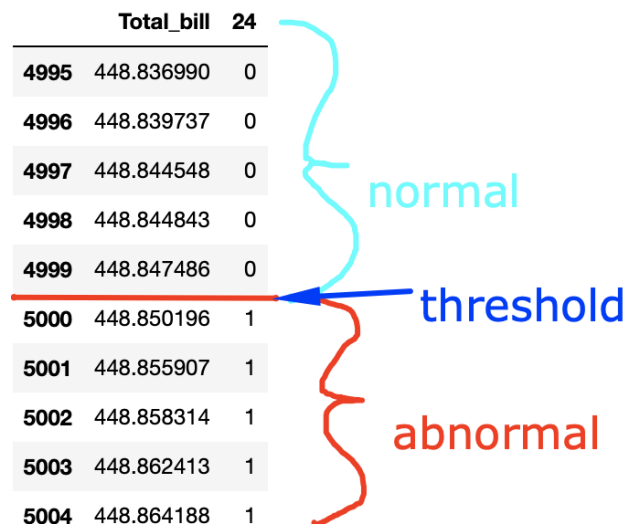
Figure 21: classification results.

As we can see, the results of the classification were split into 50 normal and 50 abnormal curves. This happens because the SVM classifier identified a support vector that classifies the data with almost 100% accuracy. More precisely, the classifier can reach 100% accuracy if we train it in 8000 testing data. This happens because the LP solution from the linear programming-based scheduling algorithm can identify if a curve is normal or abnormal according to the produced total bill.

```
Accuracy_Training: 1.0  
precision_recall_fscore_training (1.0, 1.0, 1.0, None)
```

Figure 22: Training the model on 8000 data.

It is clear from the training data that there is a threshold between normal and abnormal curves that it can be used to classify our testing data and upon which the SVM relies to take the final decision. The threshold is set approximately at 448.85 since we can see that the curves 5000 to 10000, which are all abnormal, have values greater than the threshold.



	Total_bill	24
4995	448.836990	0
4996	448.839737	0
4997	448.844548	0
4998	448.844843	0
4999	448.847486	0
5000	448.850196	1
5001	448.855907	1
5002	448.858314	1
5003	448.862413	1
5004	448.864188	1

Figure 23: Threshold between normal and abnormal curves on Training data

Furthermore, I applied the same technique in my testing data. It is obvious from the results below that the threshold which was set before at 448.85 works also for the testing data since the first 50 curves returned a value lower than the threshold and the next 50 curves returned a value bigger than the threshold with a small offset between the normal and abnormal curves. So, my model based on this evidence classified the first 50 curves as normal and the last 50 curves as abnormal (figure 24).

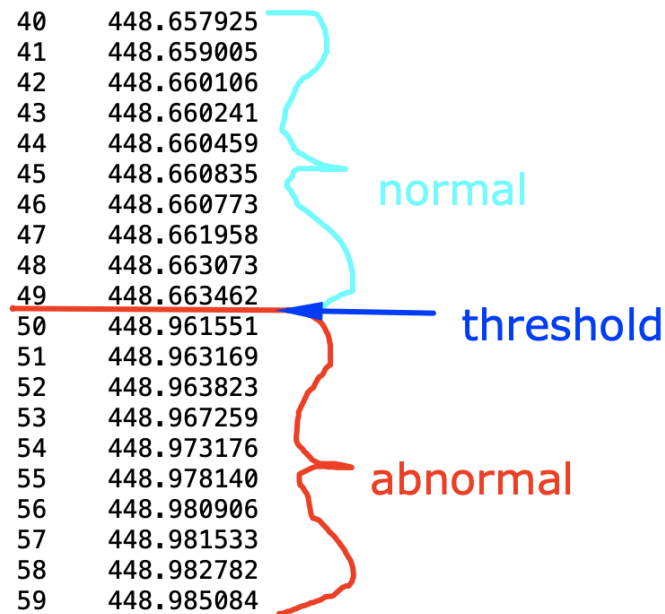


Figure 24: Threshold between normal and abnormal curves Testing data

Analysing Abnormal Curves - Testing Data

Focusing more on analysis I noticed some extra points that help identify an abnormal curve. By analysing the good curves, I calculated the mean normal bill for each user, which is displayed in table 1 below.

User ID	Mean billing of normal guidelines
User 1	99.68184369008743
User 2	78.50496606239784
User 3	94.88141828187388
User 4	90.91914588757155
User 5	78.02358277449035

Table 1: Mean billing information according to normal curves per user

By comparing the bill per user per curve with the mean bill information per user from normal curves, I noticed that most of the values are higher than the expected, which justifies and the small increase in the total billing.

Row	User1 > 100	User2 > 79	User3 > 95.5	User4 > 91.5	User5 > 79
-----	----------------	---------------	-----------------	-----------------	---------------

0	True	True	True	True	True
1	True	True	True	True	True
2	True	False	True	False	True
3	True	True	True	True	False
4	True	True	False	True	False
5	True	True	True	True	False
6	True	False	True	True	True
7	True	True	True	True	False
8	True	True	True	True	True
9	True	True	True	True	False
10	True	True	True	True	False
11	True	True	True	True	False
12	True	False	True	True	False
13	True	False	True	True	True
14	True	True	True	True	False
15	True	True	False	True	True
16	True	True	True	False	False
17	True	True	True	False	True
18	True	True	True	True	True
19	True	True	False	True	False
20	True	False	True	True	True
21	True	True	False	True	False

22	True	True	True	True	False
23	True	False	True	True	True
24	True	True	True	True	False
25	True	True	True	False	True
26	True	False	True	True	True
27	True	True	True	True	True
28	True	True	True	True	False
29	True	True	True	True	False
30	True	True	True	True	False
31	True	True	True	True	True
32	True	True	True	True	False
33	True	True	True	True	False
34	True	True	True	False	True
35	True	True	True	True	False
36	True	True	False	True	False
37	True	True	True	True	False
38	True	True	True	True	False
39	True	True	True	True	True
40	True	True	True	False	True
41	True	False	True	True	True
42	True	False	True	True	True
43	True	True	True	True	False

44	True	True	True	True	False
45	True	False	True	True	True
46	True	True	True	True	True
47	True	True	True	True	True
48	True	True	True	True	True
49	False	True	True	True	True

Table 2: Billing information per user which exceed the normal mean threshold

Results' energy plots

Moreover, I calculate the total bill per user per abnormal curve, the PAR variable and the energy scheduling. The energy consumption per hour can be seen in the diagrams below for each curve. The PAR variable after some research did not help the classification of the curves because the sample we have with 5 users in a community is not enough to see actual changes in the PAR variable. Therefore, we can see that the value 3.5643564356435644 is appeared many times and we can assume that values over 4 are an alert for suspicious curves, but the final decision cannot only rely on it.

Finally, from the plots below we can notice that the pattern of energy scheduling in all the abnormal guidelines is very similar. It is noteworthy that at 4:00am, 11:00 am and 21:00 pm the energy consumption is very high that indicates that the price on this time-window might be low often.

Abnormal Curve: 1

Bill User1:101.96954317736

Bill User2:79.314663436768

Bill User3:96.11932452407999

Bill User4:92.11861122785602

Bill User5:79.43940911361199

PAR:4.514851485148515

Energy scheduling: [0. 0. 0. 6. 8. 0. 2. 9. 0. 0. 1. 19. 0. 15. 7. 0. 5. 0.
0. 6. 1. 10. 1. 11.]

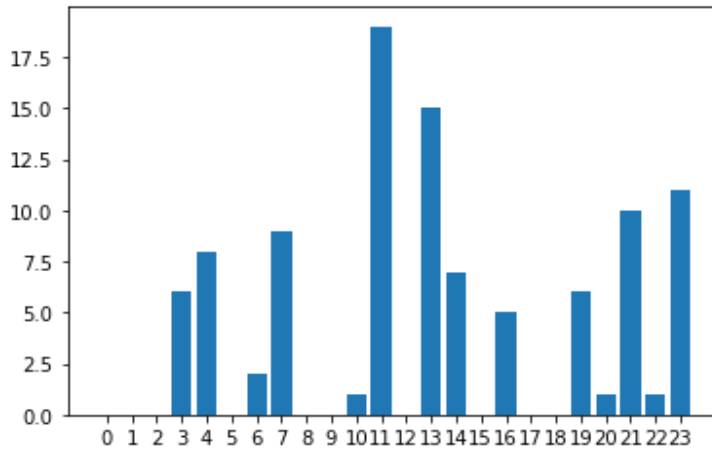


Figure 25: 1st abnormal curve energy scheduling

Abnormal Curve: 2

Bill User1:100.155242626396

Bill User2:79.882610478964

Bill User3:96.77083781080799

Bill User4:92.970874628788

Bill User5:79.18360331433999

PAR:3.5643564356435644

Energy scheduling: [0. 1. 2. 4. 10. 5. 5. 6. 0. 1. 0. 11. 0. 15. 5. 1. 0. 1.
0. 8. 14. 9. 0. 3.]

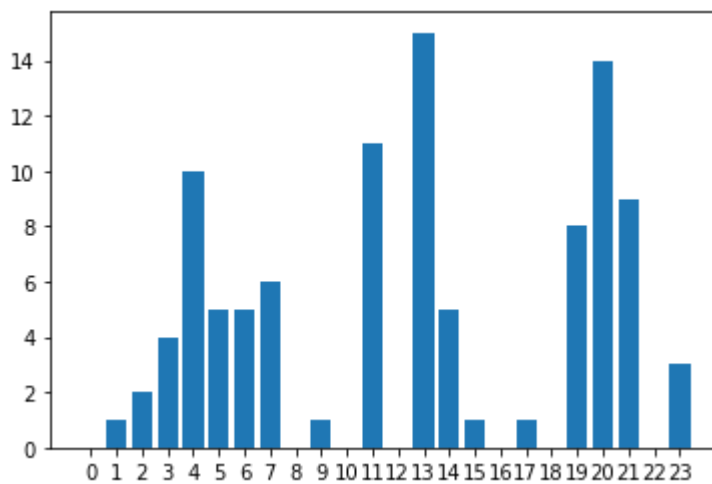


Figure 26: 2st abnormal curve energy scheduling

Abnormal Curve: 3

Bill User1:100.91659850817602

Bill User2:78.694764143484

Bill User3:96.83676768258397

Bill User4:91.18448144061601

Bill User5:81.33121119601199

PAR:4.03960396039604

Energy scheduling: [0. 0. 0. 1. 6. 12. 4. 2. 0. 1. 0. 17. 0. 17. 7. 4. 1. 0.

0. 5. 1. 10. 2. 11.]

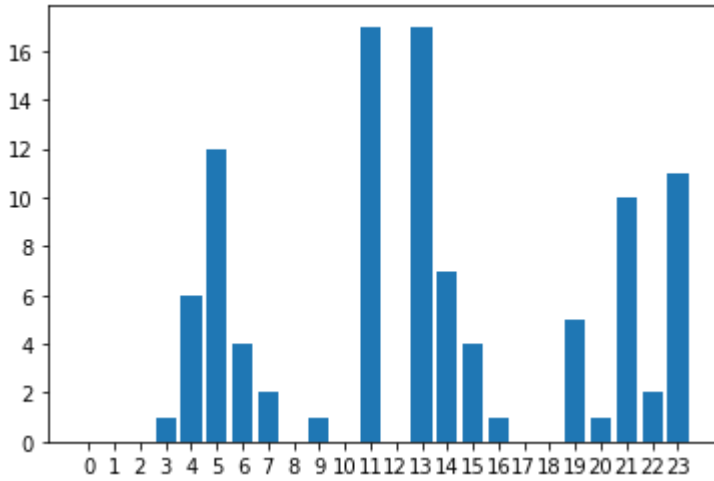


Figure 27: 3st abnormal curve energy scheduling

Abnormal Curve: 4

Bill User1:101.64219977869199

Bill User2:79.75519051186399

Bill User3:96.36516590646

Bill User4:93.06866616226802

Bill User5:78.136036919744

PAR:3.5643564356435644

Energy scheduling: [0. 0. 3. 6. 8. 5. 4. 7. 0. 1. 0. 11. 0. 15. 7. 0. 2. 0.
0. 5. 6. 14. 0. 7.]

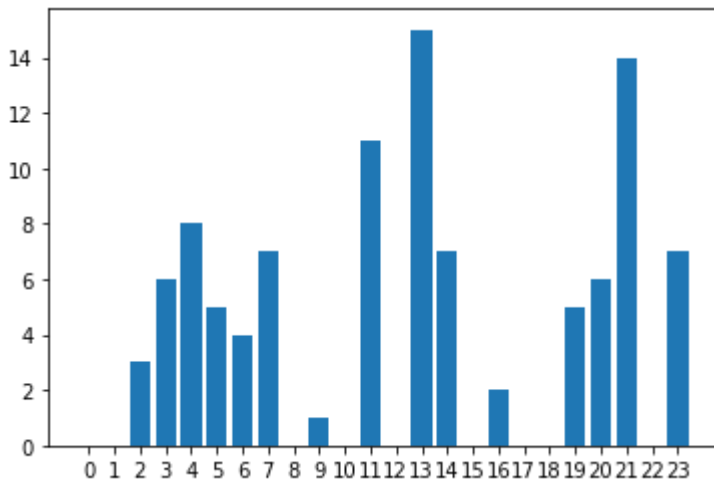


Figure 28: 4st abnormal curve energy scheduling

Abnormal Curve: 5

Bill User1:102.410063219532

Bill User2:80.087971517536

Bill User3:95.42925611419999

Bill User4:93.50976971802399

Bill User5:77.53611535313998

PAR:3.5643564356435644

Energy scheduling: [0. 0. 0. 4. 10. 8. 5. 4. 0. 0. 1. 13. 0. 15. 7. 0. 4. 0.
0. 3. 9. 14. 0. 4.]

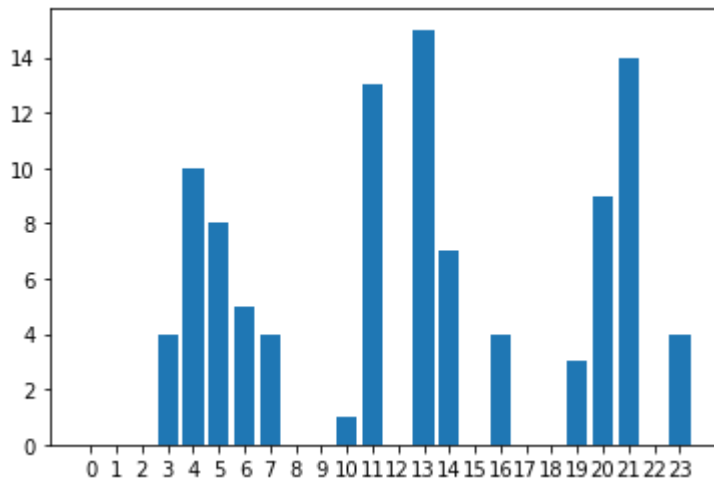


Figure 29: 5st abnormal curve energy scheduling

Abnormal Curve: 6

Bill User1:101.53929327488

Bill User2:80.10349627834401

Bill User3:96.011890706444

Bill User4:93.78033809773599

Bill User5:77.54312173306

PAR:3.5643564356435644

Energy scheduling: [0. 1. 0. 4. 10. 1. 11. 6. 0. 3. 0. 11. 0. 12. 5. 1. 0. 1.
0. 9. 4. 15. 0. 7.]

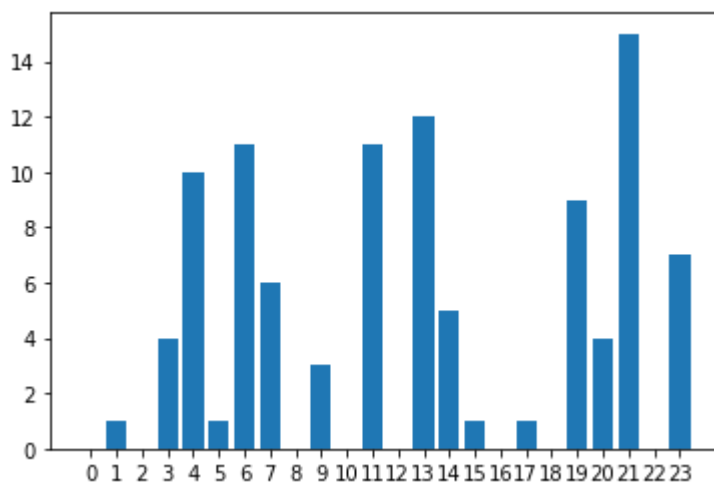


Figure 30: 6st abnormal curve energy scheduling

Abnormal Curve: 7

Bill User1:100.99165711641601

Bill User2:78.89217786431202

Bill User3:97.70602767014401

Bill User4:91.96970353650002

Bill User5:79.42133960893999

PAR:3.8019801980198022

Energy scheduling: [0. 0. 0. 4. 10. 8. 1. 7. 0. 1. 0. 13. 0. 16. 6. 0. 2. 0.
0. 14. 3. 11. 0. 5.]

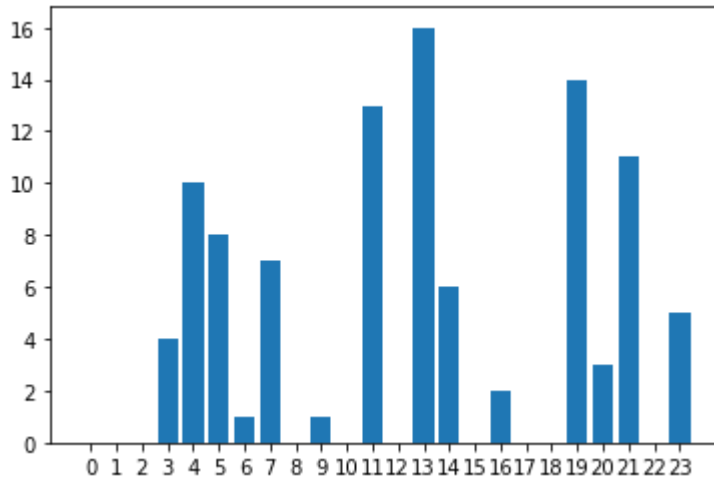


Figure 31: 7st abnormal curve energy scheduling

Abnormal Curve: 8

Bill User1:102.03108720868399

Bill User2:79.75143531522

Bill User3:95.838238055016

Bill User4:92.59899670374

Bill User5:78.76177562871601

PAR:3.5643564356435644

Energy scheduling: [0. 0. 0. 6. 6. 10. 4. 7. 0. 0. 0. 11. 0. 15. 8. 0. 2. 0.
0. 8. 3. 14. 0. 7.]

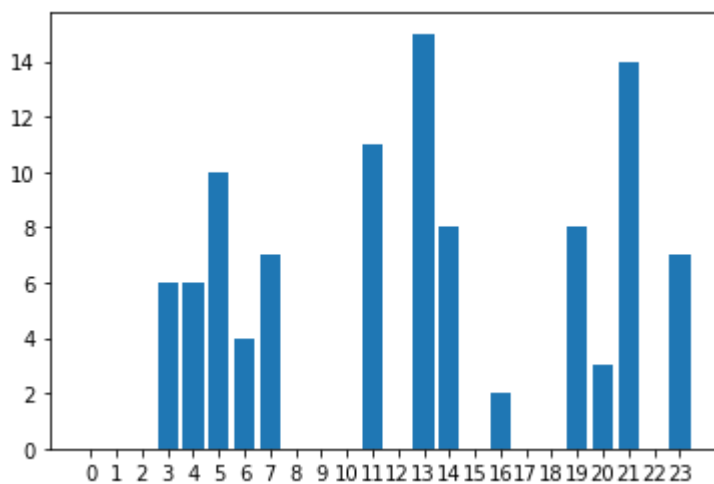


Figure 32: 8st abnormal curve energy scheduling

Abnormal Curve: 9

Bill User1:100.455906551108

Bill User2:79.090917821648

Bill User3:96.20275956571199

Bill User4:92.94417531156401

Bill User5:80.289022660532

PAR:3.5643564356435644

Energy scheduling: [0. 0. 5. 0. 8. 0. 12. 4. 0. 1. 0. 15. 0. 15. 7. 3. 1. 0.
0. 2. 6. 10. 1. 11.]

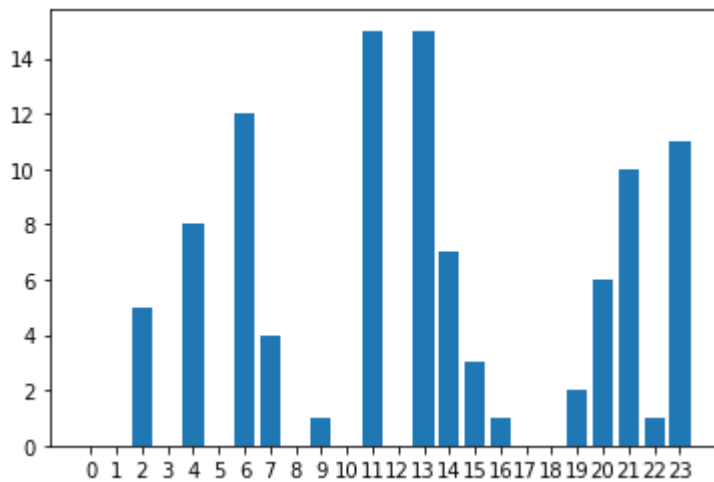


Figure 33: 9st abnormal curve energy scheduling

Abnormal Curve: 10

Bill User1:100.64412680710001

Bill User2:80.816571732376

Bill User3:96.29201478648001

Bill User4:92.53201436016401

Bill User5:78.7003566619

PAR:3.5643564356435644

Energy scheduling: [0. 0. 5. 4. 8. 5. 5. 4. 0. 1. 0. 13. 0. 15. 7. 0. 2. 0.
0. 10. 5. 14. 0. 3.]

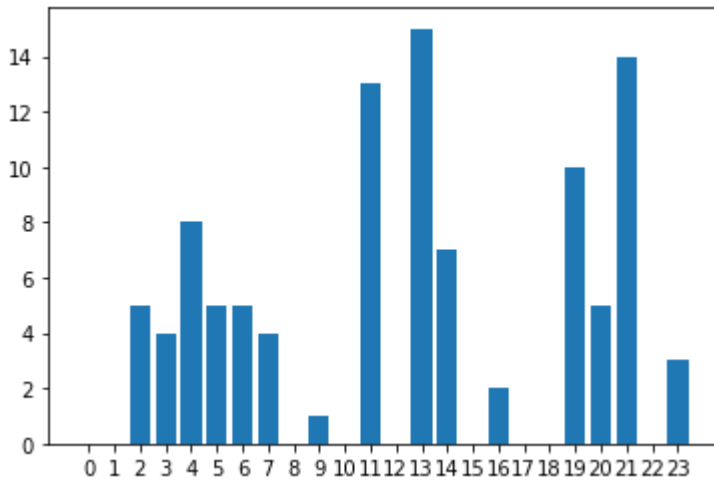


Figure 34: 10st abnormal curve energy scheduling

Abnormal Curve: 11

Bill User1:101.904178524148

Bill User2:80.70462989081999

Bill User3:95.905687807324

Bill User4:91.93942662013598

Bill User5:78.53182108172

PAR:4.03960396039604

Energy scheduling: [0. 1. 0. 6. 8. 7. 5. 6. 1. 0. 0. 9. 0. 17. 3. 0. 9. 0.
0. 5. 1. 14. 8. 1.]

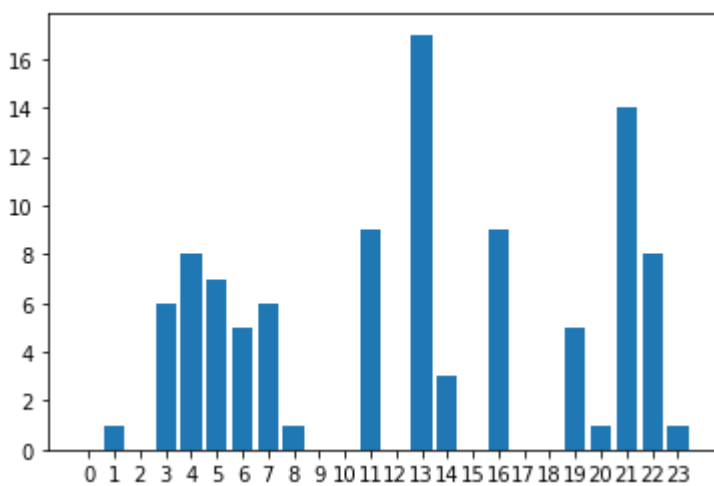


Figure 35: 11st abnormal curve energy scheduling

Abnormal Curve: 12

Bill User1:101.92484416509598

Bill User2:79.691731586532

Bill User3:95.557060593352

Bill User4:93.25347894837196

Bill User5:78.559305717524

PAR:3.5643564356435644

Energy scheduling: [0. 0. 3. 1. 10. 8. 4. 7. 0. 1. 0. 11. 0. 15. 5. 0. 2. 0.
0. 7. 6. 14. 0. 7.]

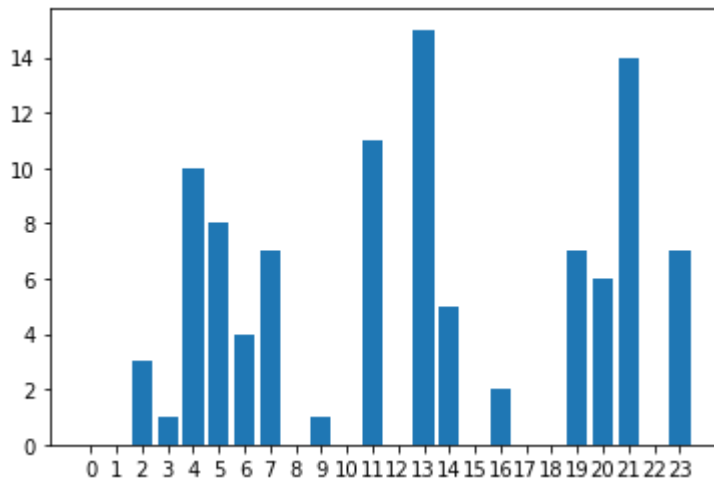


Figure 36: 12st abnormal curve energy scheduling

Abnormal Curve: 13

Bill User1:102.441406686068

Bill User2:78.837322293396

Bill User3:96.06285784729201

Bill User4:93.52986977447199

Bill User5:78.11998310597599

PAR:4.03960396039604

Energy scheduling: [0. 0. 3. 4. 10. 2. 4. 4. 3. 0. 0. 17. 0. 13. 6. 0. 2. 0.
0. 9. 3. 14. 0. 7.]

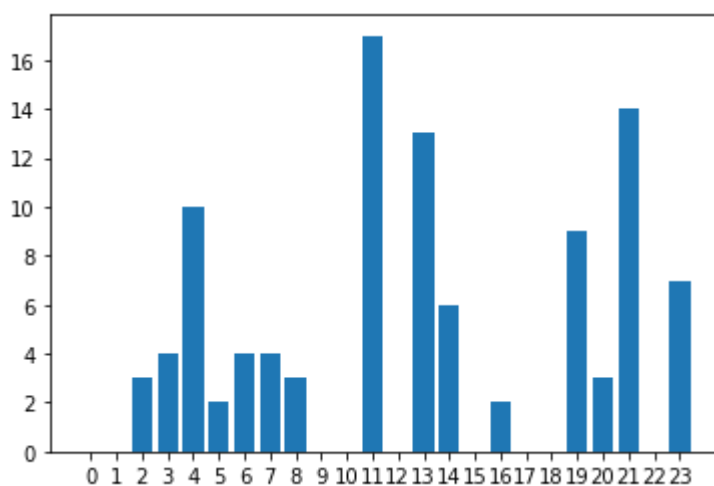


Figure 37: 13st abnormal curve energy scheduling

Abnormal Curve: 14

Bill User1:101.46887352092396

Bill User2:78.88806389063599

Bill User3:96.1880910126

Bill User4:91.76377377248

Bill User5:80.68540934423201

PAR:3.8019801980198022

Energy scheduling: [0. 1. 5. 4. 3. 9. 1. 6. 0. 1. 0. 14. 4. 16. 2. 0. 0. 3.
0. 12. 2. 7. 0. 11.]

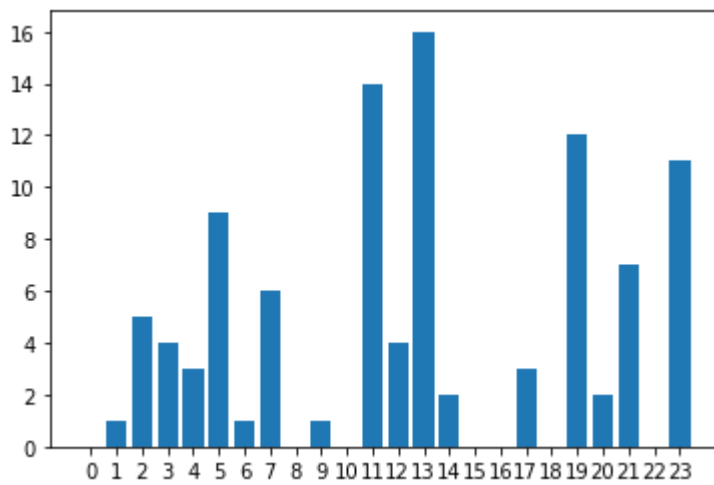


Figure 38: 14st abnormal curve energy scheduling

Abnormal Curve: 15

Bill User1:101.323436680232

Bill User2:79.616115185516

Bill User3:96.933759038256

Bill User4:92.24362675846801

Bill User5:78.87952035159199

PAR:4.03960396039604

Energy scheduling: [0. 0. 0. 4. 10. 1. 12. 2. 1. 0. 0. 13. 0. 17. 2. 0. 9. 0.
0. 2. 14. 6. 8. 0.]

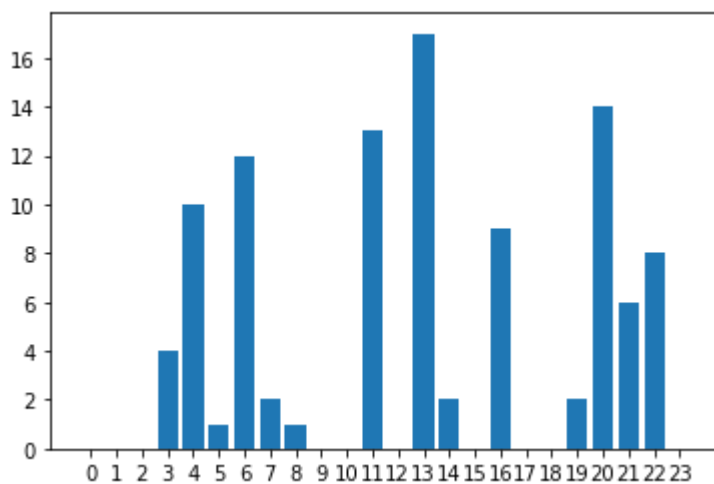


Figure 39: 15st abnormal curve energy scheduling

Abnormal Curve: 16

Bill User1:100.77599655582799

Bill User2:80.203496694448

Bill User3:95.396690531232

Bill User4:93.34989083068398

Bill User5:79.27029641841199

PAR:3.5643564356435644

Energy scheduling: [0. 1. 3. 6. 7. 1. 2. 11. 0. 0. 1. 13. 0. 15. 5. 1. 1. 0.
0. 8. 9. 14. 3. 0.]

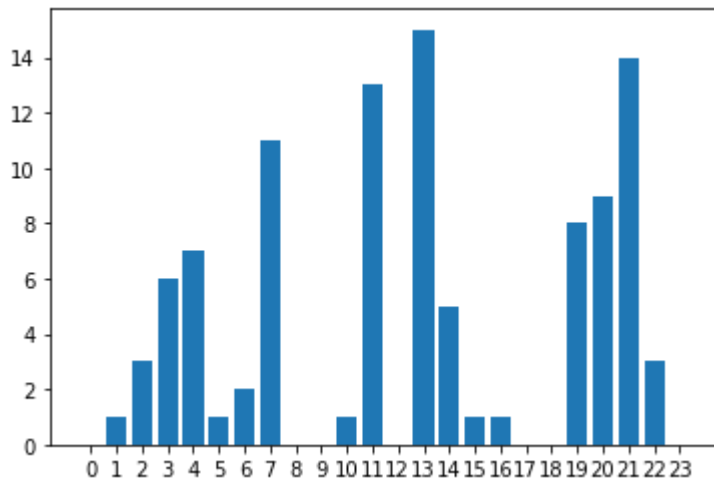


Figure 40: 16st abnormal curve energy scheduling

Abnormal Curve: 17

Bill User1:101.92877736750798

Bill User2:80.98696432811599

Bill User3:96.03244819263199

Bill User4:91.14575403158798

Bill User5:78.90449478017202

PAR:4.03960396039604

Energy scheduling: [0. 0. 3. 6. 8. 2. 4. 4. 0. 1. 0. 17. 0. 15. 6. 1. 1. 0.
0. 10. 2. 3. 7. 11.]

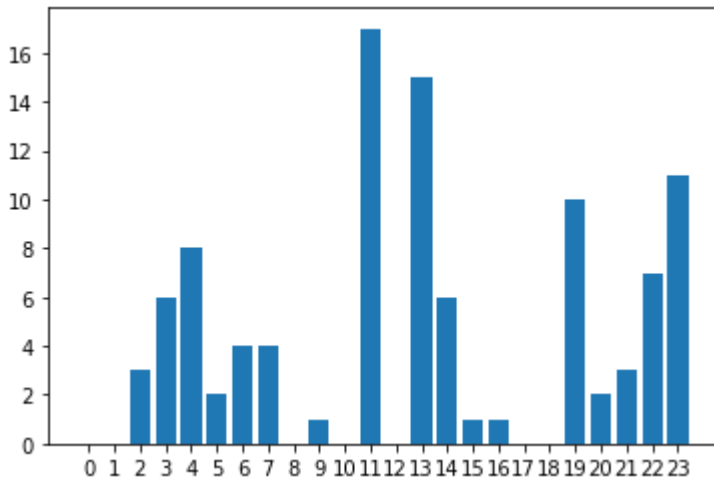


Figure 41: 17st abnormal curve energy scheduling

Abnormal Curve: 18

Bill User1:102.15241812674401

Bill User2:81.149131622484

Bill User3:96.49081274149198

Bill User4:89.57845803811199

Bill User5:79.62963200927202

PAR:4.03960396039604

Energy scheduling: [0. 1. 2. 4. 10. 2. 4. 2. 1. 0. 0. 17. 0. 17. 7. 0. 5. 0.
0. 5. 1. 5. 12. 6.]

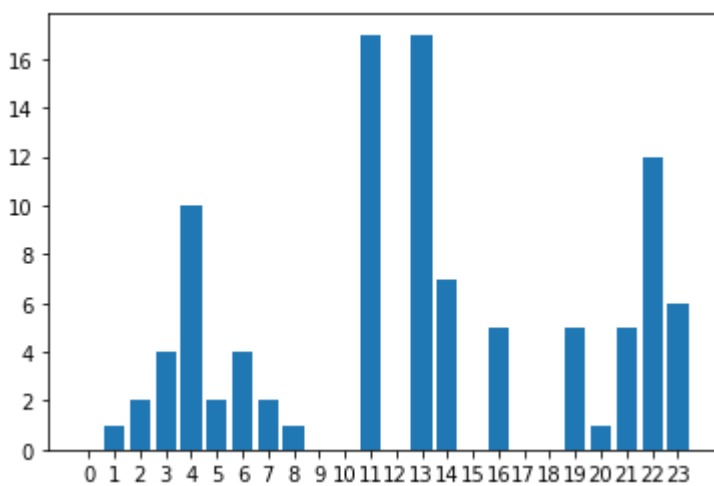


Figure 42: 18st abnormal curve energy scheduling

Abnormal Curve: 19

Bill User1:102.10702706505202

Bill User2:79.01665310736001

Bill User3:95.95554624109201

Bill User4:92.5557241638

Bill User5:79.371443654904

PAR:3.5643564356435644

Energy scheduling: [0. 1. 2. 6. 3. 10. 2. 7. 0. 0. 0. 13. 0. 15. 8. 2. 0. 1.
0. 8. 1. 10. 1. 11.]

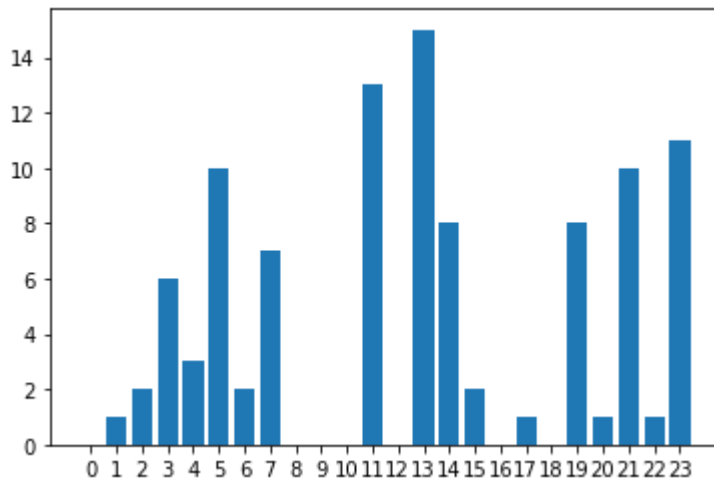


Figure 43: 19st abnormal curve energy scheduling

Abnormal Curve: 20

Bill User1:101.39723135218802

Bill User2:80.55749028413598

Bill User3:95.09022151635997

Bill User4:94.98393674730401

Bill User5:76.97892462979598

PAR:3.5643564356435644

Energy scheduling: [0. 0. 3. 6. 8. 0. 5. 11. 3. 0. 0. 11. 0. 12. 7. 0. 0. 2.
0. 6. 9. 15. 0. 3.]

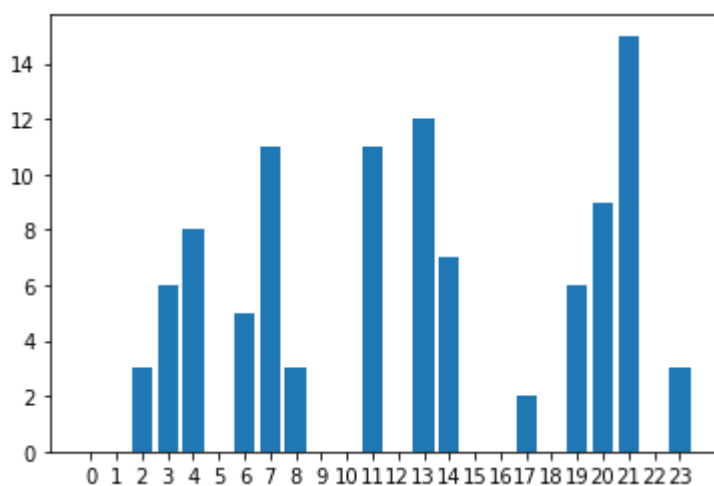


Figure 44: 20st abnormal curve energy scheduling

Abnormal Curve: 21

Bill User1:101.18440883627203

Bill User2:78.888072196208

Bill User3:96.68176070665197

Bill User4:92.24917577544001

Bill User5:80.00598002457998

PAR:3.5643564356435644

Energy scheduling: [0. 0. 0. 6. 8. 8. 5. 6. 0. 1. 0. 11. 0. 15. 7. 0. 2. 0.
0. 8. 2. 14. 1. 7.]

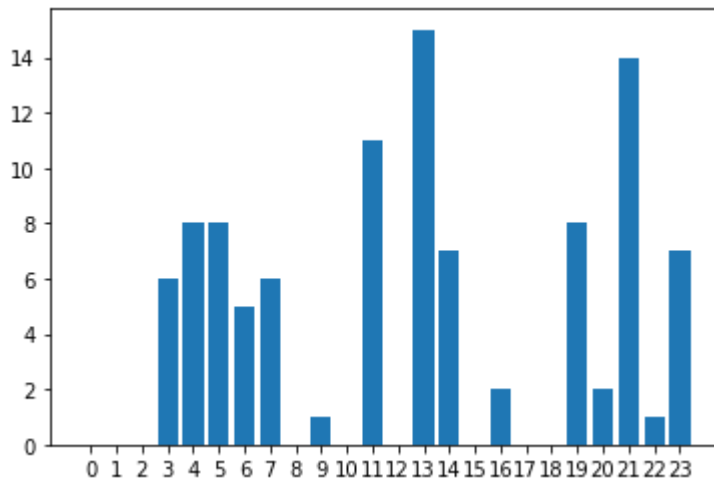


Figure 45: 21st abnormal curve energy scheduling

Abnormal Curve: 22

Bill User1:101.89728791096002

Bill User2:81.16230258710401

Bill User3:95.226262359372

Bill User4:92.388880200124

Bill User5:78.33710508092798

PAR:3.5643564356435644

Energy scheduling: [0. 0. 3. 6. 8. 5. 2. 6. 0. 1. 0. 14. 0. 15. 0. 1. 6. 0.
0. 12. 5. 1. 5. 11.]

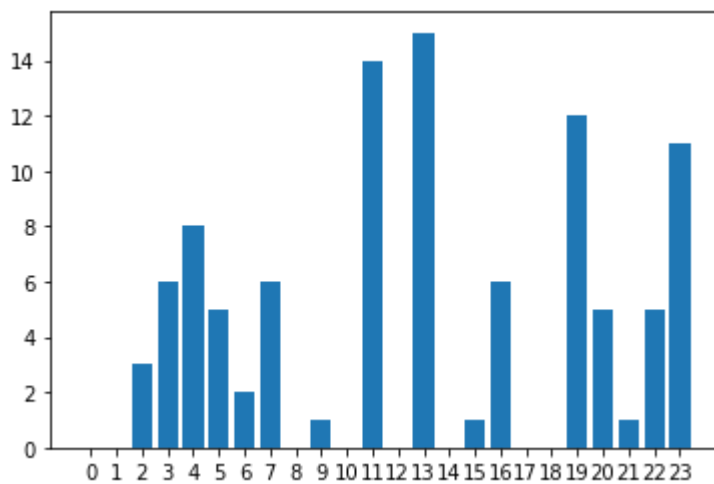


Figure 46: 22st abnormal curve energy scheduling

Abnormal Curve: 23

Bill User1:102.16168197944398

Bill User2:80.42628426773199

Bill User3:95.58201103409998

Bill User4:92.722439870048

Bill User5:78.123193367016

PAR:4.03960396039604

Energy scheduling: [0. 0. 3. 4. 10. 2. 2. 6. 0. 0. 1. 17. 0. 15. 7. 0. 2. 0.
0. 5. 10. 6. 0. 11.]

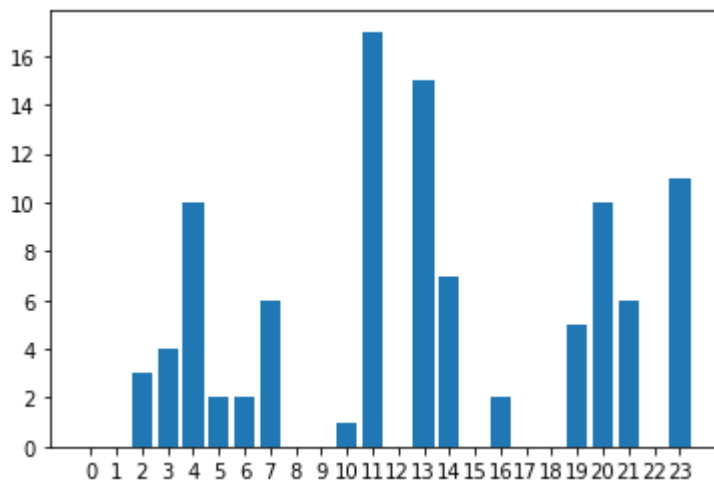


Figure 47: 23st abnormal curve energy scheduling

Abnormal Curve: 24

Bill User1:101.656352714228

Bill User2:78.51561742341602

Bill User3:96.90889030877999

Bill User4:92.44757600983598

Bill User5:79.48725614230798

PAR:4.03960396039604

Energy scheduling: [0. 1. 0. 6. 0. 10. 10. 2. 0. 0. 1. 13. 0. 17. 5. 1. 1. 0.
0. 12. 7. 1. 12. 2.]

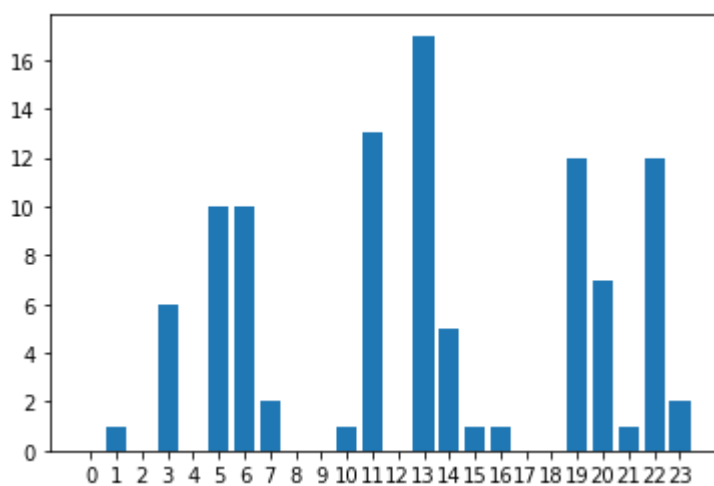


Figure 48: 24st abnormal curve energy scheduling

Abnormal Curve: 25

Bill User1:101.84606292570803

Bill User2:80.75197617913201

Bill User3:96.22361828852802

Bill User4:92.78771566546399

Bill User5:77.407682223976

PAR:3.5643564356435644

Energy scheduling: [0. 1. 0. 4. 10. 7. 4. 7. 0. 0. 0. 11. 0. 15. 8. 0. 2. 0.
0. 8. 3. 14. 0. 7.]

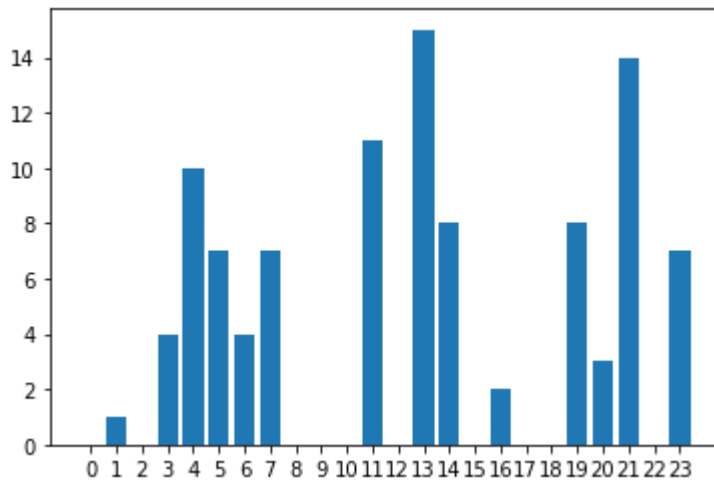


Figure 49: 25st abnormal curve energy scheduling

Abnormal Curve: 26

Bill User1:101.297759950896

Bill User2:81.671588085268

Bill User3:96.24124510155599

Bill User4:90.59010201081601

Bill User5:79.21637444403599

PAR:4.03960396039604

Energy scheduling: [0. 0. 3. 4. 10. 2. 4. 4. 0. 0. 0. 17. 0. 15. 8. 2. 1. 0.
0. 11. 1. 7. 1. 11.]

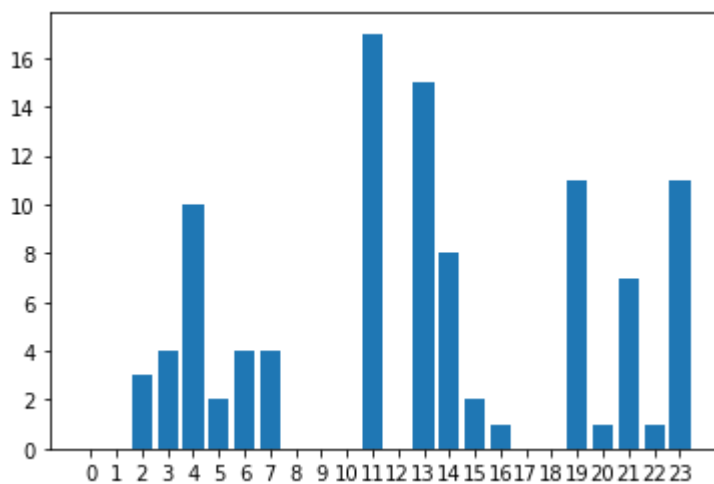


Figure 50: 26st abnormal curve energy scheduling

Abnormal Curve: 27

Bill User1:102.117242059344
 Bill User2:78.87772690721198
 Bill User3:95.94077355652
 Bill User4:93.04823861923599
 Bill User5:79.03417983124399
 PAR:3.5643564356435644
 Energy scheduling: [0. 0. 3. 6. 3. 10. 4. 4. 0. 1. 0. 14. 0. 15. 7. 4. 1. 0.
 0. 2. 4. 10. 12. 1.]

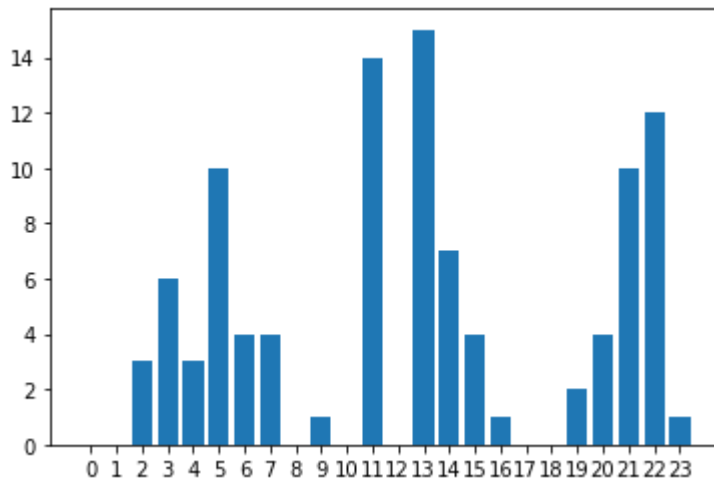


Figure 51: 27st abnormal curve energy scheduling

Abnormal Curve: 28
 Bill User1:101.67269915811198
 Bill User2:79.095577623828
 Bill User3:95.87636871359199
 Bill User4:93.04227143385599
 Bill User5:79.333257859912
 PAR:4.514851485148515
 Energy scheduling: [0. 1. 0. 4. 10. 0. 2. 8. 0. 1. 0. 19. 0. 15. 7. 0. 4. 0.
 0. 2. 10. 3. 4. 11.]

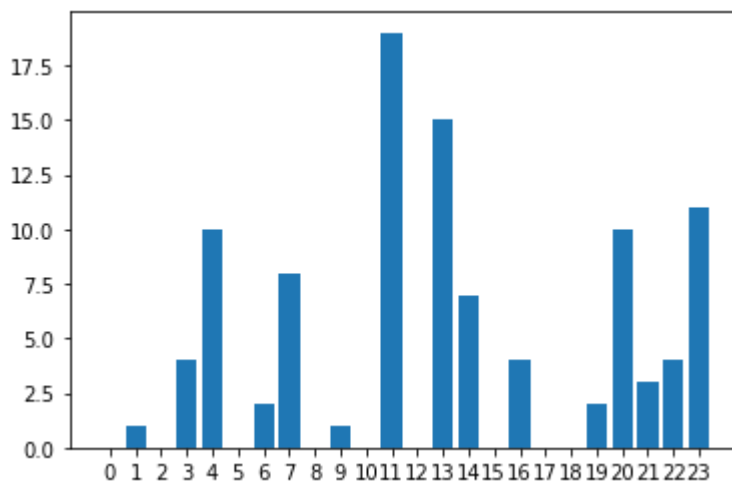


Figure 52: 28st abnormal curve energy scheduling

Abnormal Curve: 29

Bill User1:101.51438405807198

Bill User2:80.47906451737599

Bill User3:95.541779473888

Bill User4:93.13675219881601

Bill User5:78.34846539480799

PAR:3.326732673267327

Energy scheduling: [0. 0. 0. 6. 8. 8. 4. 4. 0. 3. 0. 14. 0. 13. 5. 0. 2. 0.
0. 8. 9. 14. 3. 0.]

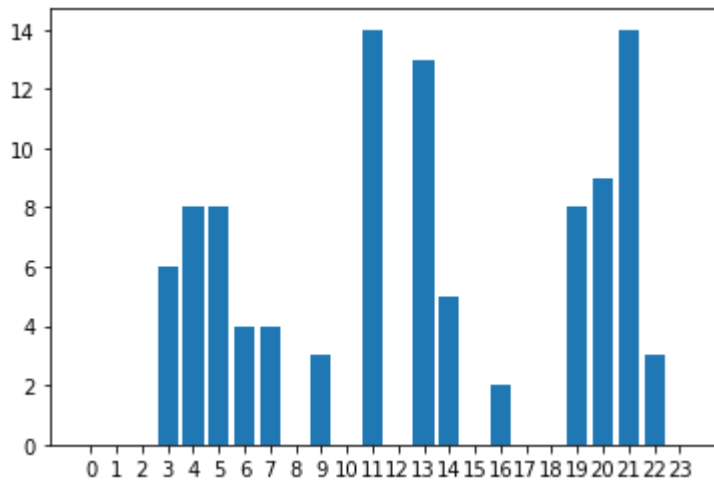


Figure 53: 29st abnormal curve energy scheduling

Abnormal Curve: 30

Bill User1:102.61359530375601

Bill User2:79.419655926736

Bill User3:96.40697330581598

Bill User4:92.076911180356

Bill User5:78.50421323376798

PAR:4.03960396039604

Energy scheduling: [0. 0. 3. 4. 10. 1. 9. 2. 0. 1. 0. 13. 0. 17. 6. 0. 3. 0.
0. 10. 1. 3. 7. 11.]

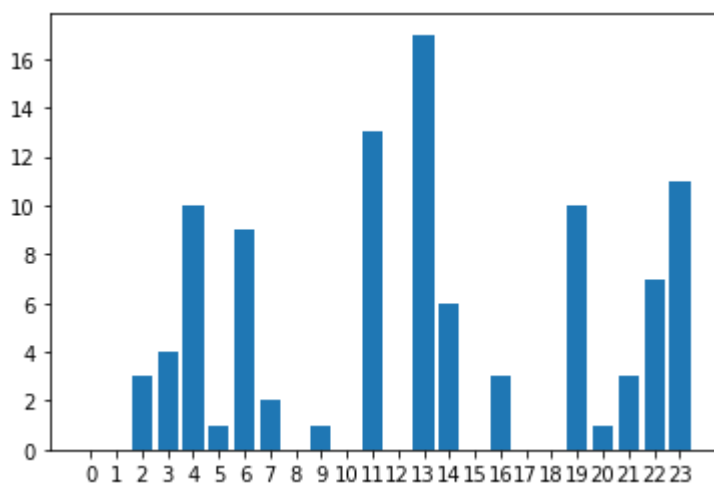


Figure 54: 30st abnormal curve energy scheduling

Abnormal Curve: 31

Bill User1:100.988404445348

Bill User2:81.716240559412

Bill User3:95.75561710500799

Bill User4:91.72848325284401

Bill User5:78.83886358199999

PAR:3.8019801980198022

Energy scheduling: [0. 0. 3. 4. 10. 5. 1. 7. 0. 1. 0. 13. 0. 16. 7. 1. 1. 0.
0. 4. 10. 6. 12. 0.]

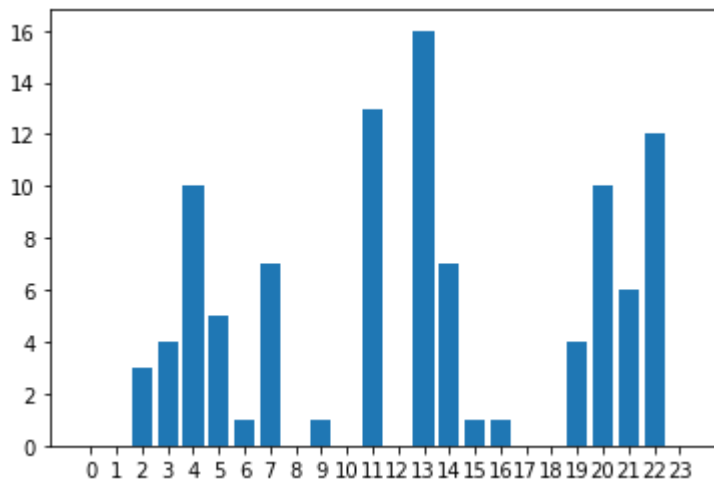


Figure 55: 31st abnormal curve energy scheduling

Abnormal Curve: 32

Bill User1:100.505067093784

Bill User2:79.762845102616

Bill User3:96.10927229294

Bill User4:93.51050067873197

Bill User5:79.14168805649199

PAR:3.5643564356435644

Energy scheduling: [0. 2. 3. 0. 9. 7. 4. 8. 0. 1. 0. 11. 0. 15. 5. 1. 0. 0.
1. 6. 6. 14. 8. 0.]

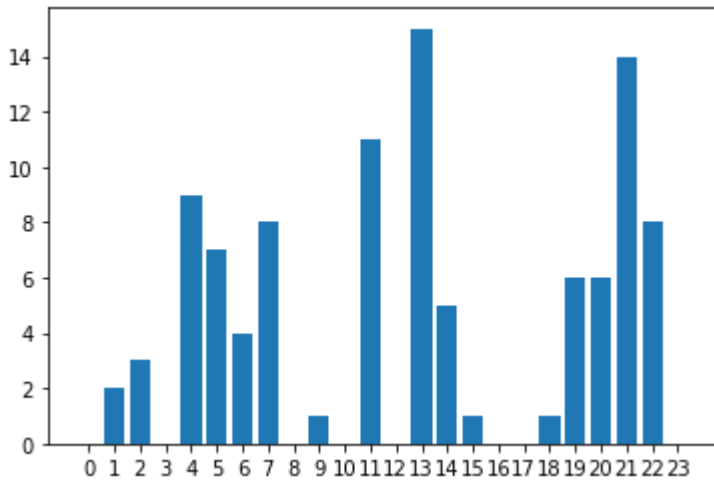


Figure 56: 32st abnormal curve energy scheduling

Abnormal Curve: 33

Bill User1:101.81623462278799

Bill User2:79.185895563896

Bill User3:96.24971674242799

Bill User4:92.850524736016

Bill User5:78.92759508784398

PAR:4.03960396039604

Energy scheduling: [0. 0. 3. 4. 10. 2. 4. 4. 0. 1. 0. 17. 0. 15. 5. 0. 2. 0.
0. 7. 10. 6. 0. 11.]

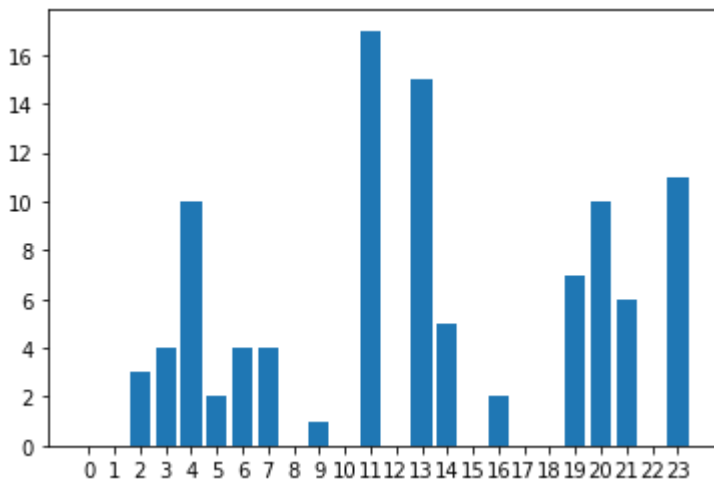


Figure 57: 33st abnormal curve energy scheduling

Abnormal Curve: 34

Bill User1:102.03069373286002

Bill User2:80.69285123620399

Bill User3:96.67386144618398

Bill User4:91.796451780796

Bill User5:77.83932564697598

PAR:3.5643564356435644

Energy scheduling: [0. 1. 0. 4. 10. 0. 11. 4. 0. 0. 0. 14. 0. 15. 8. 0. 5. 0.
0. 2. 5. 10. 12. 0.]

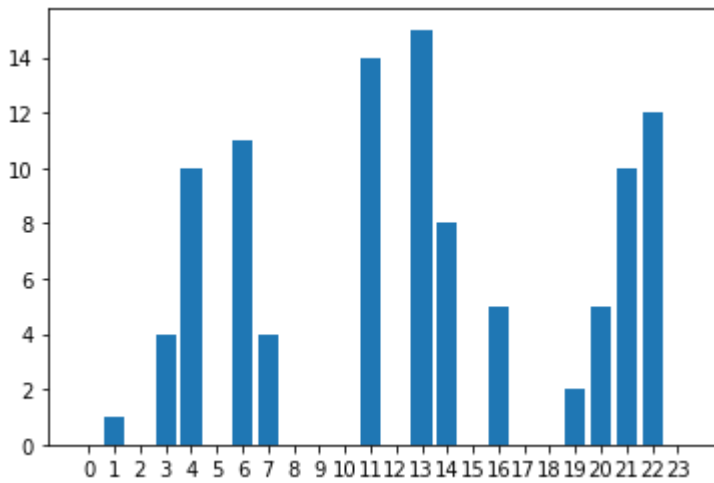


Figure 58: 34st abnormal curve energy scheduling

Abnormal Curve: 35

Bill User1:101.05990003186795

Bill User2:80.9420401644

Bill User3:95.81522183040798

Bill User4:91.31012600151998

Bill User5:79.90787389470799

PAR:4.03960396039604

Energy scheduling: [0. 1. 3. 6. 7. 5. 5. 2. 0. 1. 0. 13. 0. 17. 1. 0. 6. 0.
0. 14. 7. 3. 10. 0.]

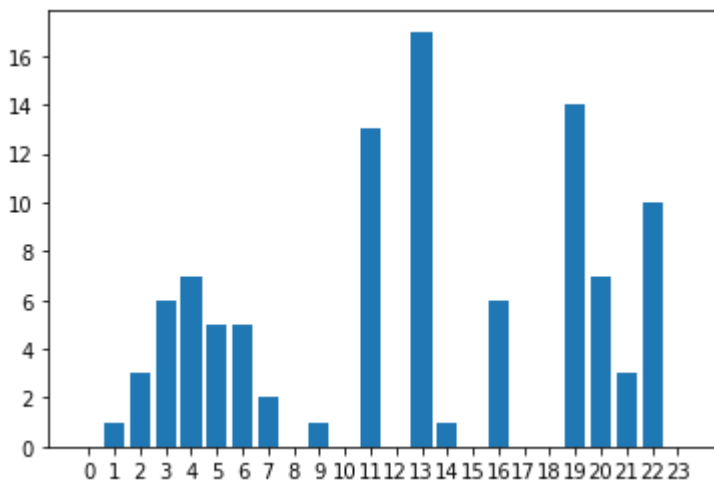


Figure 59: 35st abnormal curve energy scheduling

Abnormal Curve: 36

Bill User1:101.096565150956

Bill User2:79.218502008236

Bill User3:97.172481914676

Bill User4:93.99585321099198

Bill User5:77.55289245698401

PAR:3.326732673267327

Energy scheduling: [0. 0. 0. 4. 10. 8. 4. 7. 0. 3. 0. 11. 0. 13. 7. 0. 2. 0.

0. 6. 9. 14. 0. 3.]

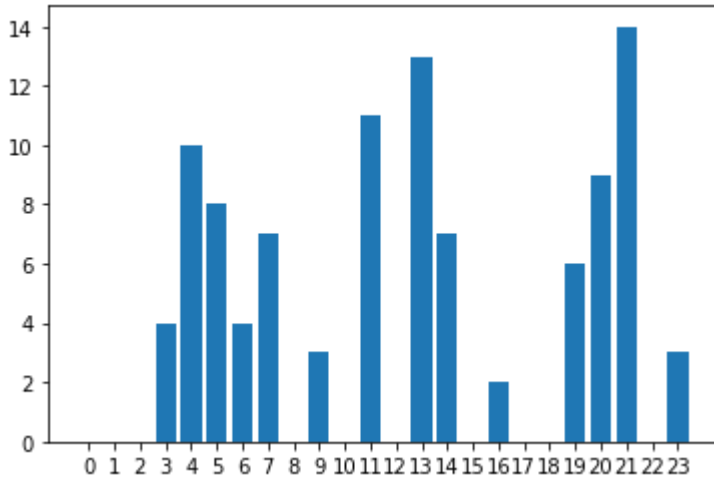


Figure 60: 36st abnormal curve energy scheduling

Abnormal Curve: 37

Bill User1:101.67758334832396

Bill User2:80.31641410517997

Bill User3:95.46857612865999

Bill User4:93.71752463943598

Bill User5:77.85616925062001

PAR:3.5643564356435644

Energy scheduling: [0. 0. 3. 4. 10. 5. 2. 7. 0. 1. 0. 13. 0. 15. 7. 0. 0. 2.

0. 4. 10. 3. 4. 11.]

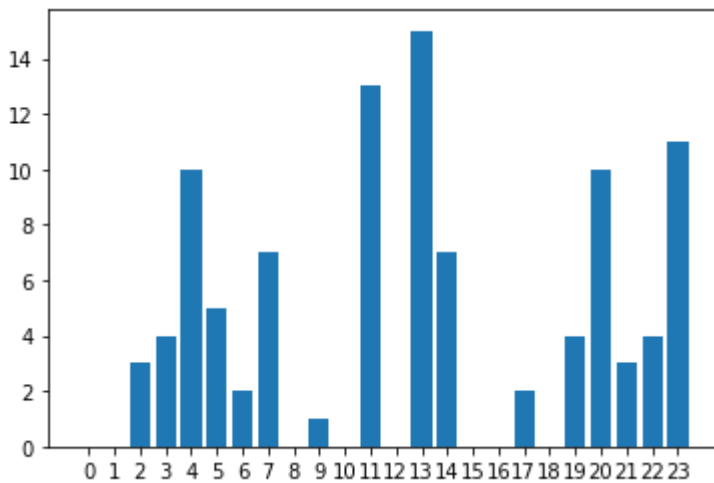


Figure 61: 37st abnormal curve energy scheduling

Abnormal Curve: 38

Bill User1:102.32861416793199

Bill User2:79.992410459416

Bill User3:96.32191218130801

Bill User4:93.136575226772

Bill User5:77.259353628712

PAR:3.5643564356435644

Energy scheduling: [0. 0. 3. 4. 10. 5. 4. 4. 1. 0. 0. 14. 0. 15. 2. 0. 9. 0.
0. 2. 14. 6. 8. 0.]

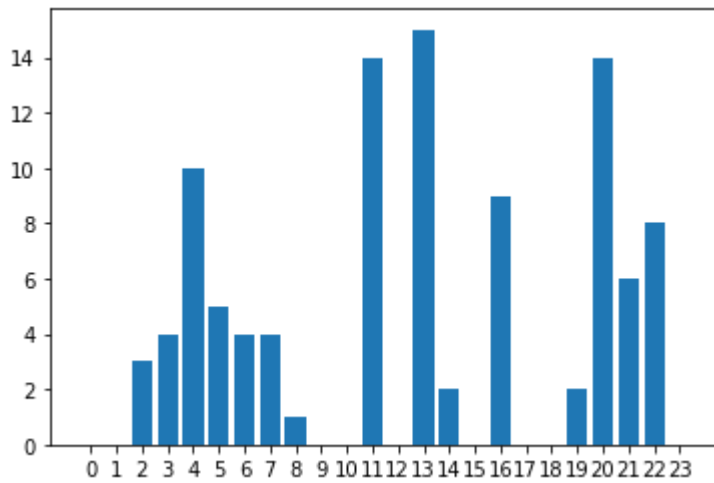


Figure 62: 38st abnormal curve energy scheduling

Abnormal Curve: 39

Bill User1:100.650664432336

Bill User2:80.9256820046

Bill User3:96.63612591286402

Bill User4:92.45768305944802

Bill User5:78.36979922293199

PAR:3.5643564356435644

Energy scheduling: [0. 0. 3. 4. 10. 5. 2. 7. 0. 1. 0. 13. 0. 15. 7. 0. 4. 0.
0. 2. 14. 6. 1. 7.]

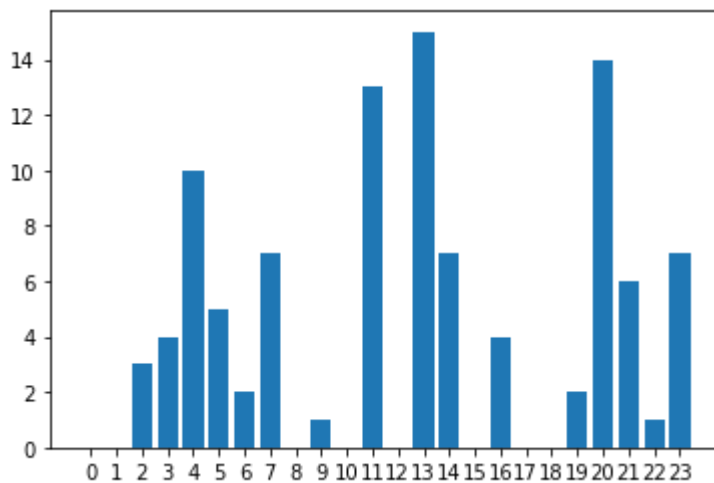


Figure 63: 39st abnormal curve energy scheduling

Abnormal Curve: 40

Bill User1:101.101263414068

Bill User2:79.45642719322

Bill User3:96.88835169775199

Bill User4:91.91296040953999

Bill User5:79.68146009324401

PAR:3.5643564356435644

Energy scheduling: [0. 0. 3. 1. 10. 8. 4. 4. 0. 0. 1. 14. 0. 15. 5. 0. 2. 0.
0. 14. 7. 4. 0. 9.]

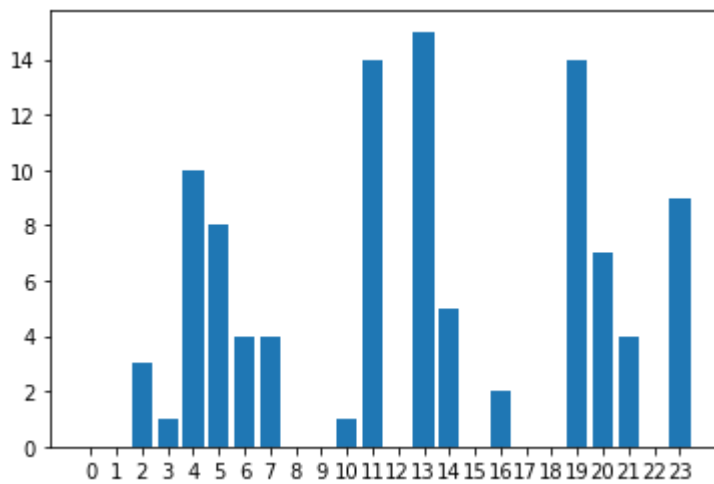


Figure 64: 40st abnormal curve energy scheduling

Abnormal Curve: 41

Bill User1:101.07078275575597

Bill User2:80.46075567070399

Bill User3:96.947593700296

Bill User4:91.04176346745598

Bill User5:79.519928964092

PAR:4.03960396039604

Energy scheduling: [0. 1. 3. 1. 10. 7. 5. 2. 0. 1. 0. 13. 1. 17. 7. 2. 1. 0.
0. 2. 6. 10. 1. 11.]

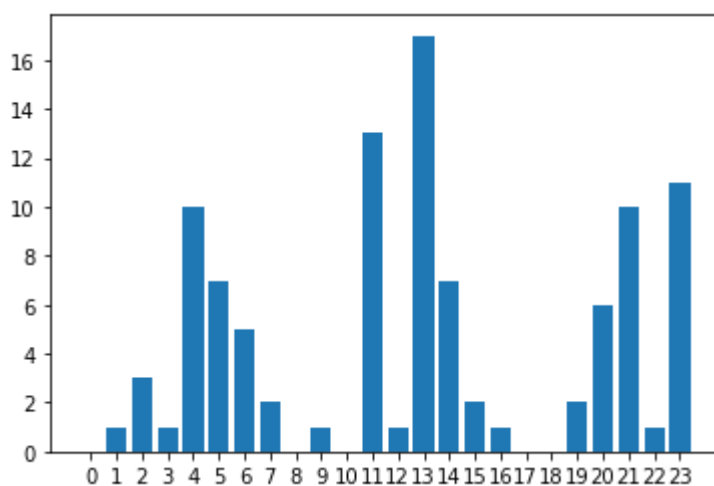


Figure 65: 41st abnormal curve energy scheduling

Abnormal Curve: 42

Bill User1:100.67738323138796

Bill User2:78.9545327005

Bill User3:97.28518030935999

Bill User4:92.49858600146398

Bill User5:79.627980961384

PAR:4.514851485148515

Energy scheduling: [0. 0. 1. 4. 10. 0. 6. 4. 0. 0. 1. 19. 0. 15. 7. 0. 2. 0.
0. 6. 14. 9. 0. 3.]

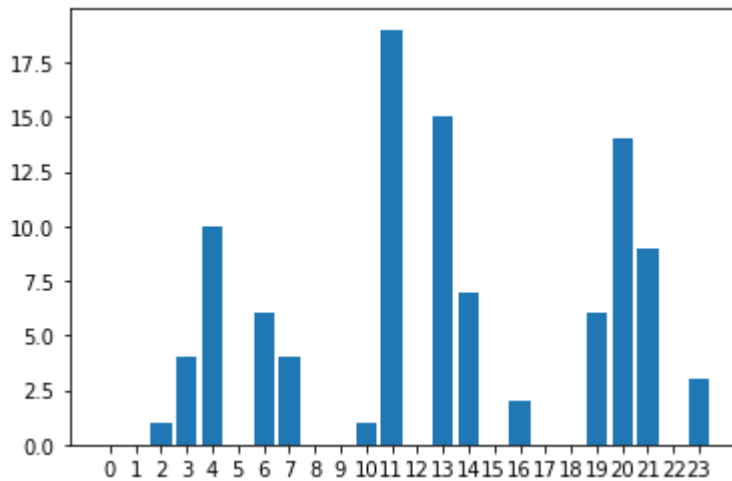


Figure 66: 42st abnormal curve energy scheduling

Abnormal Curve: 43

Bill User1:102.12214878473596

Bill User2:78.656637814344

Bill User3:97.174887345152

Bill User4:91.70951531054398

Bill User5:79.38197282711198

PAR:4.03960396039604

Energy scheduling: [0. 0. 0. 4. 10. 0. 12. 2. 1. 0. 0. 14. 0. 17. 7. 0. 0. 2.
0. 7. 2. 10. 2. 11.]

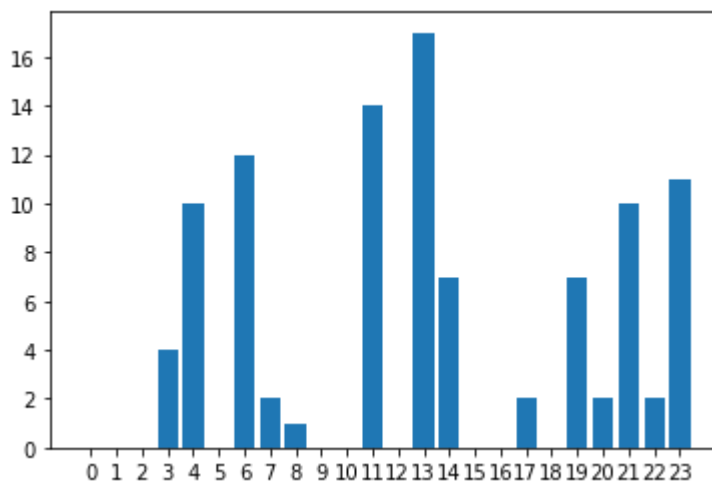


Figure 67: 43st abnormal curve energy scheduling

Abnormal Curve: 44

Bill User1:101.811086490632

Bill User2:80.14662618752001

Bill User3:95.88148319874

Bill User4:92.55630340902002

Bill User5:78.655465158064

PAR:4.03960396039604

Energy scheduling: [0. 0. 0. 6. 8. 8. 5. 2. 0. 1. 0. 13. 1. 17. 7. 0. 0. 2.
0. 3. 9. 14. 1. 4.]

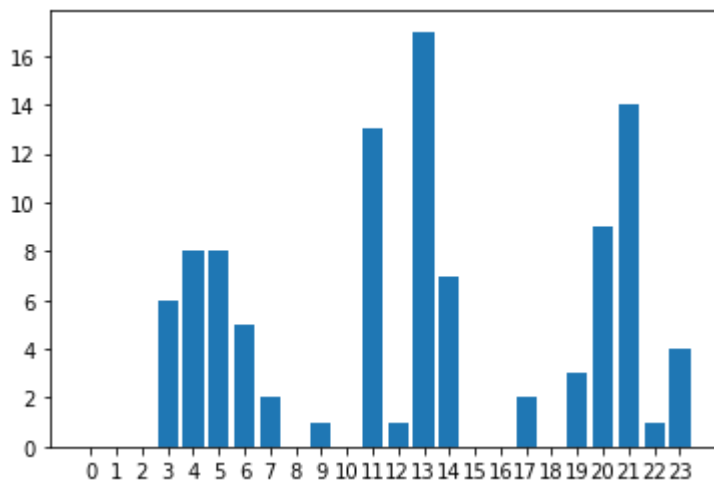


Figure 68: 44st abnormal curve energy scheduling

Abnormal Curve: 45

Bill User1:100.57339620054002

Bill User2:81.23233444298401

Bill User3:95.95319018067198

Bill User4:92.838299380336

Bill User5:78.45989672836

PAR:4.03960396039604

Energy scheduling: [0. 0. 3. 4. 10. 2. 2. 6. 0. 3. 0. 17. 0. 13. 0. 1. 6. 0.
0. 11. 6. 14. 3. 0.]

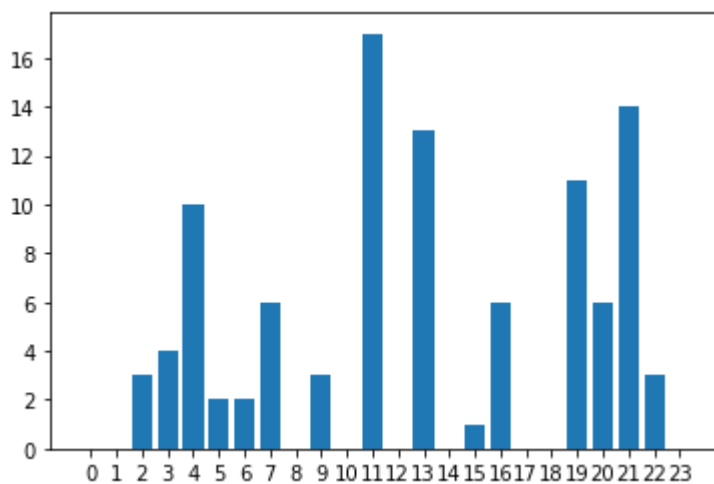


Figure 69: 45st abnormal curve energy scheduling

Abnormal Curve: 46

Bill User1:101.66818743323999
 Bill User2:78.61484059826398
 Bill User3:96.08682900726399
 Bill User4:93.54716220979199
 Bill User5:79.14115318202401
 PAR:3.8019801980198022
 Energy scheduling: [0. 0. 3. 6. 1. 10. 4. 4. 0. 3. 0. 16. 0. 13. 7. 0. 0. 2.
 0. 4. 9. 14. 5. 0.]

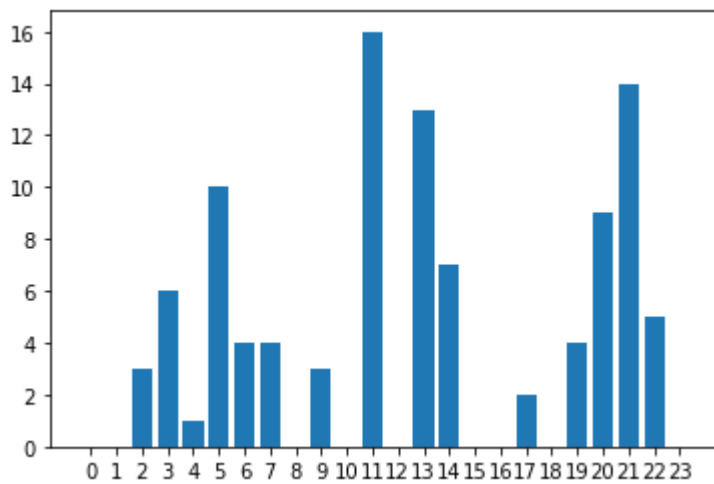


Figure 70: 46st abnormal curve energy scheduling

Abnormal Curve: 47
 Bill User1:100.28479201224401
 Bill User2:79.720144837264
 Bill User3:96.36734003248002
 Bill User4:92.509407879148
 Bill User5:80.1790614948
 PAR:3.5643564356435644
 Energy scheduling: [0. 0. 0. 4. 10. 8. 2. 6. 0. 1. 0. 14. 0. 15. 6. 0. 2. 0.
 0. 9. 3. 14. 0. 7.]

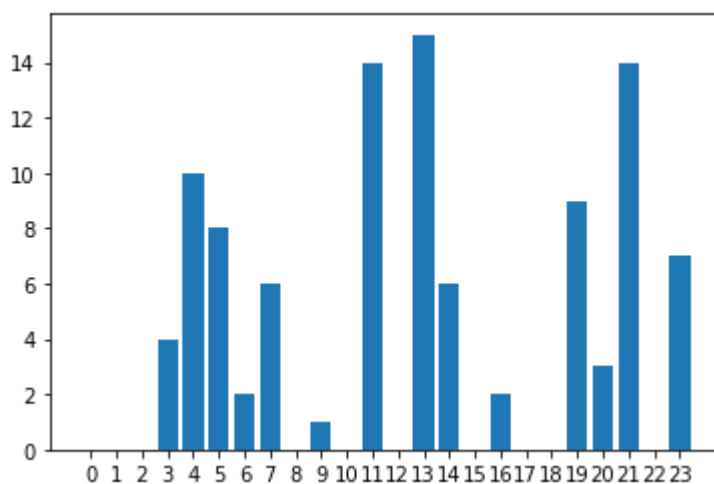


Figure 71: 47st abnormal curve energy scheduling

Abnormal Curve: 48

Bill User1:100.29965467488401

Bill User2:80.07332151780399

Bill User3:97.07669635009196

Bill User4:92.25070408283199

Bill User5:79.36042981754

PAR:4.03960396039604

Energy scheduling: [0. 1. 3. 3. 10. 0. 6. 4. 1. 0. 0. 17. 0. 15. 5. 1. 1. 0.
0. 11. 14. 6. 0. 3.]

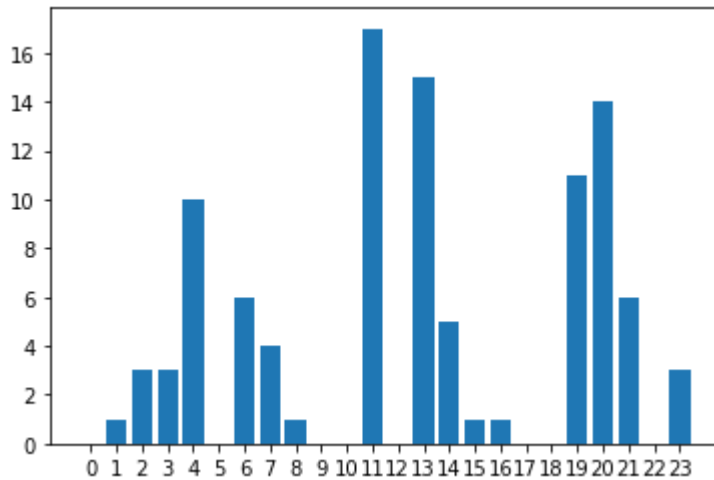


Figure 72: 48st abnormal curve energy scheduling

Abnormal Curve: 49

Bill User1:101.12999242812796

Bill User2:80.34713615122799

Bill User3:95.83458465206398

Bill User4:91.85908465233999

Bill User5:79.89017595457199

PAR:4.514851485148515

Energy scheduling: [0. 0. 1. 6. 8. 2. 2. 6. 0. 1. 0. 19. 0. 15. 3. 0. 10. 0.
0. 2. 5. 14. 7. 0.]

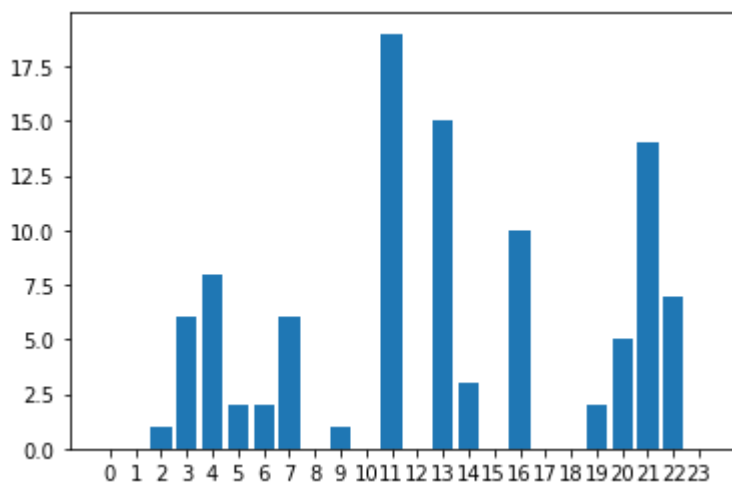


Figure 73: 49st abnormal curve energy scheduling

Abnormal Curve: 50

Bill User1:99.81179358122003

Bill User2:79.697138954636

Bill User3:96.39936101384001

Bill User4:92.21729913892402

Bill User5:80.939150074516

PAR:4.03960396039604

Energy scheduling: [0. 0. 3. 1. 10. 8. 4. 2. 1. 0. 0. 14. 0. 17. 5. 0. 2. 0.
0. 6. 9. 14. 1. 4.]

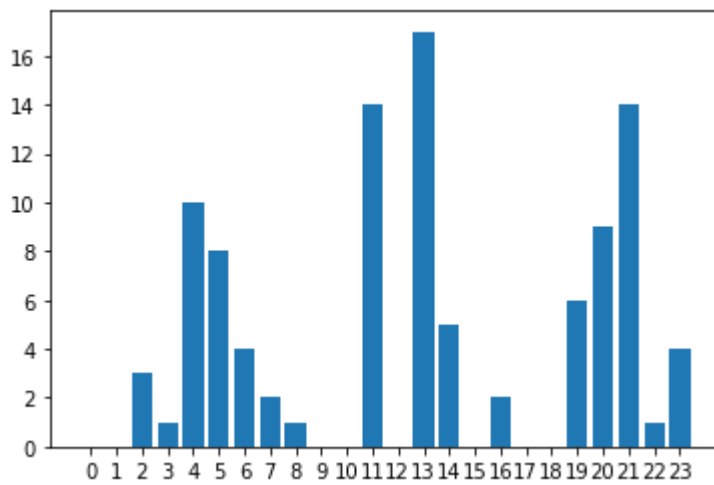


Figure 74: 50st abnormal curve energy scheduling

(this page intentionally left blank)

Appendix

```
#CREATE GOOD MEAN LINE
import numpy as np
import pandas as pd
from operator import add

dataset = pd.read_csv('coursework/TrainingData.txt', header = None)
dataset = dataset.drop(24, axis=1)
mean_curve_good = [0 for i in range (24)]

good_lines = dataset[:3000]

good_data_len = len(good_lines)

for i in range (good_data_len):
    mean_curve_good = list(map(add,
mean_curve_good,good_lines[i:i+1].values))

mean_curve_good = [x / good_data_len for x in mean_curve_good]

#CREATE BAD MEAN LINE
mean_curve_bad = [0 for i in range (24)]

bad_lines = dataset[7000:10000]

bad_data_len = len(bad_lines)

for i in range (bad_data_len):
    mean_curve_bad = list( map(add,
mean_curve_bad,bad_lines[i:i+1].values) )

mean_curve_bad = [x / good_data_len for x in mean_curve_bad]

import matplotlib.pyplot as plt
xx = np.arange(24)
plt.plot(xx, mean_curve_good[0], lw=2, color="blue", label="Good")
plt.plot(xx, mean_curve_bad[0], lw=2, color="red", label="Bad")
plt.xticks(xx)
plt.legend()
plt.show()
```

```

#USE LINE SIMILARITY TO DETECT ABNORMAL LINES
import similaritymeasures
import numpy as np
import pandas as pd
#read data
dataset = pd.read_csv('TrainingData.txt', header = None)
#remove labels
testing_data = dataset.drop(24, axis=1)
#arrays to keep if a line is normal or abnormal - 0 or 1
res_pcm = [];res_area = [];res_dtw = [];res_fr = [];res_cl = [];
y = np.array([k for k in range (24)])
good_curve = np.zeros((24, 2))
#set normal and abnormal threshold curve
x = np.array(mean_curve_good)
good_curve[:, 0] = y
good_curve[:, 1] = x
bad_curve = np.zeros((24, 2))
x = np.array(mean_curve_bad)
bad_curve[:, 0] = y
bad_curve[:, 1] = x

#PCM
for i in range (len(testing_data)):
    test_curve = np.zeros((24, 2))
    x = np.array(testing_data[i:i+1].values.ravel().tolist())
    test_curve[:, 0] = y
    test_curve[:, 1] = x
    #compare curve from dataset to see with which line is more similar
    pcm1 = similaritymeasures.pcm(test_curve, good_curve)
    pcm2 = similaritymeasures.pcm(test_curve, bad_curve)
    if (pcm1 < pcm2):
        res_pcm.append(0)
    else:
        res_pcm.append(1)

from sklearn import metrics
print("PCM Accuracy:",metrics.accuracy_score(dataset.iloc[:,-
1:].values.tolist(), res_pcm))

#Discrete Frechet distance
for i in range (len(testing_data)):
    test_curve = np.zeros((24, 2))
    x = np.array(testing_data[i:i+1].values.ravel().tolist())
    test_curve[:, 0] = y
    test_curve[:, 1] = x
    fr1 = similaritymeasures.frechet_dist(test_curve, good_curve)

```

```

fr2 = similaritymeasures.frechet_dist(test_curve, bad_curve)
if (fr1 < fr2):
    res_fr.append(0)
else:
    res_fr.append(1)
print("Discrete Frechet distance
Accuracy:",metrics.accuracy_score(dataset.iloc[:,-1:].values.tolist(),
res_fr))

#area_between_two_curves
for i in range (len(testing_data)):
    test_curve = np.zeros((24, 2))
    x = np.array(testing_data[i:i+1].values.ravel().tolist())
    test_curve[:, 0] = y
    test_curve[:, 1] = x
    area1 = similaritymeasures.area_between_two_curves(test_curve,
good_curve)
    area2 = similaritymeasures.area_between_two_curves(test_curve,
bad_curve)
    if (area1 < area2):
        res_area.append(0)
    else:
        res_area.append(1)
print("Area Between 2 Curves
Accuracy:",metrics.accuracy_score(dataset.iloc[:,-1:].values.tolist(),
res_area))

#Curve Length based similarity measure
for i in range (len(testing_data)):
    test_curve = np.zeros((24, 2))
    x = np.array(testing_data[i:i+1].values.ravel().tolist())
    test_curve[:, 0] = y
    test_curve[:, 1] = x
    cl1 = similaritymeasures.curve_length_measure(test_curve,
good_curve)
    cl2 = similaritymeasures.curve_length_measure(test_curve, bad_curve)
    if (cl1 < cl2):
        res_cl.append(0)
    else:
        res_cl.append(1)
print("Curve Length based similarity measure
Accuracy:",metrics.accuracy_score(dataset.iloc[:,-1:].values.tolist(),
res_cl))

#Dynamic Time Warping distance
for i in range (len(testing_data)):

```

```

test_curve = np.zeros((24, 2))
x = np.array(testing_data[i:i+1].values.ravel().tolist())
test_curve[:, 0] = y
test_curve[:, 1] = x
dtw1,d1 = similaritymeasures.dtw(test_curve, good_curve)
dtw2,d2 = similaritymeasures.dtw(test_curve, bad_curve)
if (dtw1 < dtw2):
    res_dtw.append(0)
else:
    res_dtw.append(1)
print("Dynamic Time Warping distance
Accuracy:",metrics.accuracy_score(dataset.iloc[:,-1:].values.tolist(),
res_dtw))

#USE DISTANCE BETWEEN POINTS TO FIND ABNORMAL LINES

# ALL metrics - if you want to change the metric below
# print '\nEuclidean distance is', dist.euclidean(A, B)
# print 'Manhattan distance is', dist.cityblock(A, B)
# print 'Chebyshev distance is', dist.chebyshev(A, B)
# print 'Canberra distance is', dist.canberra(A, B)
# print 'Cosine distance is', dist.cosine(A, B)

import scipy.spatial.distance as dist
bad_curves = dataset[5000:10000]
good_curves = dataset[:5000]

dataset = pd.read_csv('TrainingData.txt', header = None)
bad_dis = []
good_dis = []
tempg = []
tempb = []
#calcualte the distance and keep only the max distance from the magic
curve - change dist to calculate based on different metric
for j in range (len(good_curves)):
    for i in range (24):
        euclidean_good =
dist.euclidean(good_curves[j:j+1].values.ravel()[i],mean_curve_good[0][i
])
        tempg.append(euclidean_good)
        euclidean_bad =
dist.euclidean(bad_curves[j:j+1].values.ravel()[i],mean_curve_good[0][i
])
        tempb.append(euclidean_bad)
    bad_dis.append(max(tempb))

```



```

    good_dis.append(max(tempg))
    tempg = []
    tempb = []

#keep all max distances from the mean good curve together
lst = good_dis + bad_dis

#find and set a threshold
print(np.mean(good_dis),np.mean(bad_dis))

res = []
#calculate the label - threshold 0.025
for dis in lst:
    if (abs(dis-0.555)<= 0.025):
        res.append(1)
    else:
        res.append(0)

print("Euclidean distance
Accuracy:",metrics.accuracy_score(dataset.iloc[:,-1:].values.tolist(),
res))

```

```

# USE DECISION TREES FOR CLASSIFICATION
import pandas as pd
from sklearn.tree import DecisionTreeClassifier # Import Decision Tree
Classifier
from sklearn.model_selection import train_test_split # Import
train_test_split function
from sklearn import metrics #Import scikit-learn metrics module for
accuracy calculatio
from sklearn import metrics

# Read file
dataset = pd.read_csv('coursework/TrainingData.txt', header = None)

#split training - testing data
X_train, X_test, y_train, y_test = train_test_split(dataset.drop(24,
axis=1), dataset.iloc[:,24],
test_size=0.25,random_state=45,stratify=dataset.iloc[:,24])

# Create Decision Tree classifier object
clf = DecisionTreeClassifier()

```

```
# Train Decision Tree Classifier
clf = clf.fit(X_train,y_train)

#Predict the response for test dataset
y_pred = clf.predict(X_test)

# Model Accuracy:
print("Decision Tree Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
# USE XGBOOST FOR CLASSIFICATION
import xgboost as xgb
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.metrics import precision_recall_fscore_support
import pandas as pd
import numpy as np

# #Read file - split the dataset
dataset = pd.read_csv('coursework/TrainingData.txt', header = None)
# I will keep 75% of the training data for training and 25% of the
training data for testing
X_train, X_test, y_train, y_test = train_test_split(dataset.drop(24,
axis=1), dataset.iloc[:,24], test_size=0.25,stratify=dataset.iloc[:,24])

# fit model no training data
model = xgb.XGBClassifier()
model.fit(X_train, y_train)

# make predictions for test data
y_pred = model.predict(X_test)
predictions = [round(value) for value in y_pred]

print("xgboost Accuracy:",metrics.accuracy_score(y_test, predictions))
PRF = precision_recall_fscore_support(y_test, predictions,
average='binary')
print('xgboost precision_recall_fscore', PRF)
```

```
#CHECK HYPERPARAMETERS

#searching key hyperparametres for SVC
from sklearn.datasets import make_blobs
from sklearn.model_selection import RepeatedStratifiedKFold
```

```

from sklearn.model_selection import GridSearchCV
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

#Read file - split the dataset
dataset = pd.read_csv('TrainingData.txt', header = None)

#split training - testing data
X_train, X_test, y_train, y_test = train_test_split(dataset.drop(24,
axis=1), dataset.iloc[:, -1:], test_size=0.25, random_state=12)

# define model and parameters
model = SVC()
kernel = ['poly', 'rbf', 'linear']
C = [0.01, 1, 10, 20, 50]
gamma = ['scale']
# define grid search
grid = dict(kernel=kernel, C=C, gamma=gamma)
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1,
cv=cv, scoring='accuracy', error_score=0)
grid_result = grid_search.fit(X_train, y_train)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_,
grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))

```

```

#USE SVM FOR CLASSIFICATION - ADD BILL FEATURE

from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.metrics import precision_recall_fscore_support
from sklearn import svm
import pandas as pd
import numpy as np
from lpSolve55 import *
import matplotlib.pyplot as plt
%matplotlib inline

```

```

#Read file - insert bill feature in dataset
dataset = pd.read_csv('TrainingData.txt', header = None)
total_bill = pd.read_csv('TotalBillPerCurve.txt', header = None)
dataset.insert(24, column='Total_bill', value=total_bill)

#split dataset
X_train, X_test, y_train, y_test = train_test_split(dataset.drop(24,
axis=1), dataset.iloc[:, -1:],

test_size=0.20, stratify=dataset.iloc[:, -1:])

#Create a svm Classifier - polynomial kernel returns the best results
clf = svm.SVC(kernel='poly', C=20)

#Train the model using the training sets
clf.fit(X_train, y_train.values.ravel().tolist())

#Predict the response for test dataset
y_pred_svm = clf.predict(X_test)

# Model Accuracy training:
print("Accuracy_Training:", metrics.accuracy_score(y_test, y_pred_svm))
PRF = precision_recall_fscore_support(y_test, y_pred_svm,
average='binary')
print('precision_recall_fscore_training', PRF)

#predict the real TestingData.txt
dataTest = pd.read_csv('TestingData.txt', header = None)
total_bill_test = pd.read_csv('total_bill_results_per_curve.txt', header
= None)
dataTest.insert(24, column='Total_bill_test', value=total_bill_test)
#classify testing data
predictions_Test = clf.predict(dataTest)
#print results
print (predictions_Test)

#create file TestingResults.txt
#read data
write_data = []
with open('TestingData.txt', 'r') as f:
    for i, line in enumerate (f.readlines(), start=0):
        line = line + ',' + str(predictions_Test[i])
        write_data.append(line)
#write data

```

```

with open('TestingResults.txt', 'w') as f:
    for item in write_data:
        f.write("%s\n" % item)

#LP SOLVER SOLUTION

#each row is for each guideline price
total_community_bill = []
#each row has the 5 different bills per user array
total_bill_per_user = []
#each row has 240 variables where 1 means the user(i) executed the task
that time. every 5 rows the curve is changed
execution_time_per_user = []
#we want accuracy up to 14 digits
np.set_printoptions(precision=14)

def LPsol(userID, cost_curve):
    if(userID > 5 or userID < 1):
        print ('wrong user id')
        return
    if(not(isinstance(userID, int))):
        print ('wrong user id')
        return
    #parse the excel file
    xls = pd.ExcelFile('COMP3217CW2Input.xlsx')
    df1 = pd.read_excel(xls, 'User & Task ID')
    #cost_curve = pd.read_excel(xls, 'NormalGuidelinePricing')['Unit
    Cost']

    #create all possible variables - each user's task for each time
    period I created a variable such as userN_taskMhX
    col_names = ['u'+str(i)+'t'+str(j)+'h'+str(k) for i in range(1,6)
    for j in range(1,11) for k in range(0,24)]
    #keep only the variables related to the userID - each user needs 240
    (= 10 tasks * 24 hours) variables
    col_names = col_names[(userID-1)*240:(userID)*240]
    ncols = len(col_names)

    #obj_row keeps the rates for each variable that will be used -
    variables in the task's time window only
    obj_row = np.array(cost_curve)
    for i in range (9):
        obj_row = np.append(obj_row, cost_curve)

```

```

values = np.zeros(ncols)
task_counter = 0
const_names = []
for i in range(userID,userID+1):
    for j in range (10):
        for k in range(0,24):
            #keep only the variables between the timing window a
            task can be executed
            if not(df1.iloc[j+(userID-1)*10]['Ready Time'] <= k <=
df1.iloc[j+(userID-1)*10]['Deadline']):

const_names.append('u'+str(i)+'t'+str(j+1)+'h'+str(k))
    indices = [col_names.index(name) for name in const_names]
    obj_row[indices] = 0.0

#start creating the LP problem
#define the length of rows and cols, rows will be added later now
are zero
lp = lpsolve('make_lp', 0, ncols)
#only important errors should be shown
lpsolve('set_verbos', lp, IMPORTANT)
#set name on variables to be readable
for i in range(ncols):
    lpsolve('set_col_name', lp, i+1, col_names[i])
#set objective function and we want to minimize it
lpsolve('set_obj_fn', lp, obj_row)
lpsolve('set_minim', lp)

#constrains for the variables that will be used only
#for the used functions
values = np.zeros(ncols) ; values[indices]=1.0
#values array keep the non used function - reverse it
values = 1 - values
for i in range (240):
    if(values[i] == 1):
        constraint_per_hour = np.zeros(ncols)
        constraint_per_hour[i] = constraint_per_hour[i] + values[i]
        lpsolve('add_constraint',lp, constraint_per_hour, 'LE',
1.00)
        lpsolve('add_constraint',lp, constraint_per_hour, 'GE',
0.00)

#energy demand per hour per task
for i in range(10):
    constraint_energy_demand = np.zeros(ncols)

```

```

        constraint_energy_demand[i*24:(i+1)*24] =
constraint_energy_demand[i*24:(i+1)*24] + values[i*24:(i+1)*24]
        lpsolve('add_constraint',lp, constraint_energy_demand, 'EQ',
df1.iloc[i+(userID-1)*10]['Energy Demand'])

#write to lp file to be able to preview on the lpsolver
lpsolve('solve', lp)
#get optimal billing
bill = lpsolve('get_objective', lp)
total_bill_per_user.append(bill)

varlp = lpsolve('get_variables', lp)
#round number because the LP returns some 1.0000000002 and
0.99999999998 - it does not affect the result
varlp_int = []
for i in varlp[0]:
    varlp_int.append(round(i))
#keep the execution window of each user
execution_time_per_user.append(varlp_int)

#lpsolve('write_lp', lp,
'energy_curve'+str(guidelineID)+'_user'+str(userID)+'.lp')
lpsolve('delete_lp', lp)

#CALCULATE THE LP SOLUTIONS FOR THE ABNORMAL GUIDELINES
dataTest = dataTest.drop('Total_bill_test',axis=1)
for i in range((len(dataTest))):
    #for curves that were classified as abnormal calculate LP solution
    if(predictions_Test[i] == 1):
        cost_curve = dataTest[i:i+1].values.ravel().tolist()
        for j in range(1,6):
            LPSol(j,cost_curve)

#lines 0,5,10... keep the daily bill of user1 - other users accordingly
bill_user1 = [];bill_user2 = [];bill_user3=[];bill_user4=[];bill_user5=[]
count = 0
#group bills per user
for i in total_bill_per_user:
    if(count%5 == 0):
        bill_user1.append(i)
    elif(count%5 == 1):
        bill_user2.append(i)
    elif(count%5 == 2):
        bill_user3.append(i)

```

```

elif(count%5 == 3):
    bill_user4.append(i)
elif(count%5 == 4):
    bill_user5.append(i)
count += 1

#Calculate Community's bill per abnormal line
community_bill = []
for i in range(len(bill_user1)):
    temp = bill_user1[i] + bill_user2[i] + bill_user3[i] + bill_user4[i]
+ bill_user5[i]
    community_bill.append(temp)

#function to calculate the PAR per curve
def calc_PAR(arr):
    max_item = max(arr)
    avg = 0
    for i in arr:
        avg += i
    avg = avg / (len(arr))
    PAR = max_item / avg
    return PAR

#sum the energy consumption per hour to create diagrams
def calc_hourly_consumption(arr):
    counter = 0
    total_energy_hour = np.zeros(24)

    #add all the variables that will be executed in the same time to
    find the hourly consumption - vars were produced from the LP
    for i in arr:
        hour = counter % 24
        if (hour == 0):
            total_energy_hour[hour] = total_energy_hour[hour] + i
        elif (hour == 1):
            total_energy_hour[hour] = total_energy_hour[hour] + i
        elif (hour == 2):
            total_energy_hour[hour] = total_energy_hour[hour] + i
        elif (hour == 3):
            total_energy_hour[hour] = total_energy_hour[hour] + i
        elif (hour == 4):
            total_energy_hour[hour] = total_energy_hour[hour] + i
        elif (hour == 5):
            total_energy_hour[hour] = total_energy_hour[hour] + i
        elif (hour == 6):
            total_energy_hour[hour] = total_energy_hour[hour] + i

```



```

elif (hour == 7):
    total_energy_hour[hour] = total_energy_hour[hour] + i
elif (hour == 8):
    total_energy_hour[hour] = total_energy_hour[hour] + i
elif (hour == 9):
    total_energy_hour[hour] = total_energy_hour[hour] + i
elif (hour == 10):
    total_energy_hour[hour] = total_energy_hour[hour] + i
elif (hour == 11):
    total_energy_hour[hour] = total_energy_hour[hour] + i
elif (hour == 12):
    total_energy_hour[hour] = total_energy_hour[hour] + i
elif (hour == 13):
    total_energy_hour[hour] = total_energy_hour[hour] + i
elif (hour == 14):
    total_energy_hour[hour] = total_energy_hour[hour] + i
elif (hour == 15):
    total_energy_hour[hour] = total_energy_hour[hour] + i
elif (hour == 16):
    total_energy_hour[hour] = total_energy_hour[hour] + i
elif (hour == 17):
    total_energy_hour[hour] = total_energy_hour[hour] + i
elif (hour == 18):
    total_energy_hour[hour] = total_energy_hour[hour] + i
elif (hour == 19):
    total_energy_hour[hour] = total_energy_hour[hour] + i
elif (hour == 20):
    total_energy_hour[hour] = total_energy_hour[hour] + i
elif (hour == 21):
    total_energy_hour[hour] = total_energy_hour[hour] + i
elif (hour == 22):
    total_energy_hour[hour] = total_energy_hour[hour] + i
elif (hour == 23):
    total_energy_hour[hour] = total_energy_hour[hour] + i
    counter += 1
return total_energy_hour

```

```

#lines 0,5,10... keep the daily execution hours of user1 - other users
accordingly
exec_user1=[];exec_user2=[];exec_user3=[];exec_user4=[];exec_user5=[]
count = 0
for i in execution_time_per_user:
    if(count%5 == 0):
        exec_user1.append(i)
    elif(count%5 == 1):
        exec_user2.append(i)

```

```

elif(count%5 == 2):
    exec_user3.append(i)
elif(count%5 == 3):
    exec_user4.append(i)
elif(count%5 == 4):
    exec_user5.append(i)
count += 1

#CALCULATE THE TOTAL EXECUTION TIME WINDOW PER ABNORMAL LINE
community_execution_times = []
for i in range(0,len(execution_time_per_user),5):
    #keep each time 5 users
    arr = np.array(execution_time_per_user[i:i+5])
    #sum the rows to calculate the total energy consumption per hour per
    task - remember we have 240 possible slots -> 24 hours * 10 tasks
    community_execution_times.append(arr.sum(axis=0))

#sum all the tasks that will be executed the same time
hourly_consumption_per_line = []
for i in range(len(community_execution_times)):

    hourly_consumption_per_line.append(calc_hourly_consumption(community_exe
    cution_times[i]))

#Calculate PAR per line
PAR = []
for i in hourly_consumption_per_line:
    PAR.append(calc_PAR(i))

#PLOT CONSUMPTION PER HOUR
x = np.arange(24)
for i in range(len(hourly_consumption_per_line)):
    print ('Abnormal Curve: '+str(i+1))
    print ('Bill User1:' + str(bill_user1[i]))
    print ('Bill User2:' + str(bill_user2[i]))
    print ('Bill User3:' + str(bill_user3[i]))
    print ('Bill User4:' + str(bill_user4[i]))
    print ('Bill User5:' + str(bill_user5[i]))
    print ('PAR:' + str(PAR[i]))
    print ('Energy scheduling: ' + str(hourly_consumption_per_line[i]))
    plt.xticks(x)
    plt.bar(x, hourly_consumption_per_line[i])
    plt.show()
    print()

#create a tample that will be used in result analysis - sho if users'

```

```

bill in abnormal curves are higher than the mean normal cost
comparelst = []
for i in range(len(bill_user1)):
    temp = []
    if (bill_user1[i] > 100):
        temp.append(True)
    else:
        temp.append(False)
    if (bill_user2[i] > 79):
        temp.append(True)
    else:
        temp.append(False)
    if (bill_user3[i] > 95.5):
        temp.append(True)
    else:
        temp.append(False)
    if (bill_user4[i] > 91.5):
        temp.append(True)
    else:
        temp.append(False)
    if (bill_user5[i] > 79):
        temp.append(True)
    else:
        temp.append(False)
    comparelst.append(temp)
df = pd.DataFrame(comparelst, columns = ['User1', 'User2', 'User3',
'User4', 'User5'],
dtype = bool)
print(df)

```

```

#LP SOLVER SOLUTION - COMMUNITY - ANOTHER APPROACH RETURNS ONLY THE
TOTAL BILL OF THE COMMUNITY
community_bills = []

def LPsol(curve):
    #READ TRAINING DATASET FROM FILE
    dataset = pd.read_csv('coursework/TrainingData.txt', header = None)

    #READ TASKS THAT WILL BE EXECUTED
    xls = pd.ExcelFile('coursework/COMP3217CW2Input.xlsx')
    df1 = pd.read_excel(xls, 'User & Task ID')

    #SET CONSTANTS OF THE PROBLEM
    users_num = 5

```

```

hours = 24
tasks = df1['User & Task ID'].values.ravel().tolist()
tasks_num = len(tasks)

#CREATE ALL POSSIBLE VARIABLES - EACH USER HAS 10 TASKS WITH 24
HOURS WINDOW TO BE EXECUTED -> 5 * 10 * 24 = 1200 POSSIBLE VARIABLES
var_names = [i+'_hour'+str(j) for i in tasks for j in range(hours)]

#KEEP ONLY THE VARIABLES THAT ARE IN THE TIME WINDOW OF EXECUTION
executed_variables = []
#KEEP THE VALUES OF THE COST PER HOUR FOR EACH VARIABLE - OBJECTIVE
FUNCTION
obj_row = []
#KEEP VALUES (0,1) TO SET THE CONSTRAINTS BELOW PER TASK
constraint_energy_demand = []
for i in tasks:
    for j in range(hours):
        #FIND THE TASK'S ROW AND CHECK THE TIME WINDOW
        if (df1.loc[df1['User & Task ID'] == i]['Ready Time'].values
<= j
                <= df1.loc[df1['User & Task ID'] ==
i]['Deadline'].values):
            executed_variables.append(i+'_hour'+str(j))
            #KEEP PRICE OF THE EXECUTION TIME
            obj_row.append(curve[j])
            #KEEP THE POSITION OF THE VARIABLE THAT WILL BE EXECUTED
            constraint_energy_demand.append(1)
        else:
            constraint_energy_demand.append(0)

#start creating the LP problem
#define the length of rows and cols, rows will be added later now
are zero
lp = lpsolve('make_lp', 0, len(executed_variables))
#only important errors should be shown
lpsolve('set_verbos', lp, IMPORTANT)
#set name on variables to be readable
for i in range(len(executed_variables)):
    lpsolve('set_col_name', lp, i+1, executed_variables[i])
#set objective function and we want to minimize it
lpsolve('set_obj_fn', lp, obj_row)
lpsolve('set_minim', lp)

#for each variable set 0 <= var <= 1 - maximum energy per hour
for i in range(len(executed_variables)):
    values = np.zeros(len(executed_variables))

```

```

        values[i] = 1
        lpsolve('add_constraint',lp, values, 'LE', 1.00)
        lpsolve('add_constraint',lp, values, 'GE', 0.00)

    #for each task send energy demand in total
    #reshape the constraint_energy_demand because it is a continuous
    list and we want to break it in 24 items lists
    constraint_energy_demand =
np.array(constraint_energy_demand).reshape(tasks_num, hours)
    start = 0
    end = sum(constraint_energy_demand[0])
    for i, task in enumerate (tasks, start=0):
        temp = np.zeros(len(executed_variables))
        temp[start:end] = 1
        if(i!=tasks_num-1):
            end = sum(constraint_energy_demand[i+1]) + end
            start = sum(constraint_energy_demand[i]) + start
        energy_demand = df1.loc[df1['User & Task ID'] == task]['Energy
Demand'].values
        lpsolve('add_constraint',lp, temp, 'EQ', energy_demand)

    #solve the LP problem
    lpsolve('solve', lp)
    #get optimal billing
    bill = lpsolve('get_objective', lp)
    community_bills.append(bill)

#TO EXECUTE
# dataTest = dataTest.drop('Total_bill_test',axis=1)
# for i in range((len(dataTest))):
#     #for curves that were classified as abnormal calculate LP solution
#     if(predictions_Test[i] == 1):
#         cost_curve = dataTest[i:i+1].values.ravel().tolist()
#         LPsol(cost_curve)

```