# Cyber-physical systems coursework 2

## Option A

Miroslav Milanov

## Introduction

The first part of this challenge is to build a model and make correct predictions on the test set.
I am going to be using Python in particular for the scikit-learn library. Apart from the obvious choice of using it for the predictive models themselves, it also offers a variety of utility functions that will enable me to make an educated decision about which algorithm performs best.
I am also going to be using the pandas library for the DataFrame class.

First of all, I will focus on the TrainingData.txt dataset to gather information on the data, which model to use, and what hyperparameters work best. For that reason I will split the TrainingData into three datasets:

- Training dataset – used to train models during the internal testing.
- Validation dataset – we predict the results of the model on the validation dataset and use it to calculate accuracies. We use those accuracies to make design decisions.
- Testing dataset – once we've finalized our design decisions – algorithm choices, hyperparameters, we use predict the internal testing dataset to make sure that everything is working as expected and to ensure the quality of our design choices.
  It is important to know that predicting the accuracy on the testing set should only be done once – as doing it more than that defeats the purpose of having a testing dataset. In reality, you don't have access to the results of the accuracy on the testing dataset, so you should not make decisions based on that.

Once we've prepared a model, we can use it to predict the original testing dataset in TestingData.txt and output the results to TestingResults.txt. We can then use those results to compute the pricing information of abnormal ones using Linear Programming.

## Implementation

### dataSplit.py

dataSplit.py is responsible for preparing the data that we will use.

The fileToDf function takes a path to the file that we will read and a Boolean. If the Boolean is true, we're parsing TrainingData and if it is false we are parsing the TestingData.

The function converts the raw file into a pandas dataframe by splitting the csv styled .txt file and inputting it into a DataFrame class. All the values are converted into floats and ints. Columns are named from 0 to 23 and the TrainingData DF has a label called "Labels" with values of 0 and 1.

splitTrainingSet takes as input the training DataFrame and splits it into the three abovementioned datasets. The split is as follows:

Training set – 80%; Validation set – 10%; Training set – 10%.

The sets are then exported as .csv files for the main program. The reason for that is that we want the same training, validation and test sets each time, so we cannot randomly split them every single time we run the main program.

### main.py

main.py is where all the computation happens.

The function generateConfusionMatrix is a utility function used during internal testing to visualize the F1 score of the models. This is to ensure high precision and recall when evaluating the accuracy of different models.

The main function has two Booleans. The workingOnTraining Boolean is set to True when we are doing the internal testing. The finalTest Boolean is set to True when we are ready to make the final check on the internal testing set.

First we extract the data from the generated .csv files and we split it into input and label data.

Then we initialize a model and fit the training data.

Finally, depending on the two Booleans we either predict the internal validation or training sets, or we predict the original training set.

Using sklearn's utility functions we can analyze the accuracy using an F1 score and a confusion matrix. Utilizing the F1 score is generally better than a simple $\frac{\#Accurate\ predictions}{\#Total\ entries}$ accuracy formulae, especially in heavily biased sets. For example, in a dataset that has 99% normal entries, a simple algorithm that always outputs 0 has an accuracy of 0.99. This set is balanced, but it is still a good idea to use an F1 score.

Ultimately, we end up with a file called TestingResults.txt containing the output we expect.

## Design Choices

The problem in this coursework is a simple classification one. That being said, I usually first check results using a Linear Regression model. The reason for that is that for most simple and linear problems it performs well. It is also reasonably computationally inexpensive. It is important to remember, though, that when using a regression problem in a classification problem the results need to be parsed. In our example, the output of the Linear Regression was a float.

All we need to do is pipe the final output we have through an activation function:

Values equal to or less than 0.5 are equal to 0; Values more than 0.5 are equal to 1.

Interstingly enough, a default Linear Regression model has an accuracy of around 0.94. This is significantly more accurate than a Support Vector Regressor model, which achieves around 0.76.

From the classifier models I tried out a Decision Tree Regressor, which achieved a result of around 0.81 and a Support Vector Classifier, which had an accuracy of 0.96.

| Lin Reg | SVR | DecTR | SVC |
|---|---|---|---|
| 0.94 | 0.76 | 0.81 | 0.96 |

An accuracy increase of 0.02 is equivalent to about 10 less erroneous predictions. That meant that SVC had around 30-35% less errors than the second best algorithm – Linear Regression.

For that reason, I decided to work with a Support Vector Classifier.

There are many hyper parameters that are available for the SVC class. Some of the more significant ones are kernel, C and degree(if using poly kernel). There are a total of 15 in the documentation, however, fine tuning too much can lead to overfitting. For this, I will only look at kernel and C.

For the hyperparameter tuning I first tried out different kernels.

| linear | poly | rbf |
|---|---|---|
| 0.9308300395256918 | 0.9384920634920635 | 0.9497435897435897 |
| 0.937888198757764 | 0.930460333006856 | 0.95635305528613 |
| 0.9339999999999999 | 0.9374999999999999 | 0.9503407984420643 |
| 0.9525691699604742 | 0.944062806673209 | 0.962051282051282 |
| 0.9344262295081966 | 0.9402697495183044 | 0.9465346534653465 |

Rbf is the default kernel used by SVC and it performed the best on average.

Next, I evaluated the results of different C values. The default value of C is 1.0 and it is the one used in the above kernel comparison.

| 0.1 | 0.5 | 1.0 |
|---|---|---|
| 0.9308943089430894 | 0.9505154639175258 | 0.9497435897435897 |
| 0.950950950950951 | 0.943934760448522 | 0.95635305528613 |
| 0.9477832512315272 | 0.9496124031007752 | 0.9503407984420643 |
| 0.9427740058195926 | 0.9530761209593327 | 0.962051282051282 |
| 0.9348268839103869 | 0.9478957915831663 | 0.9465346534653465 |

| 2.0 | 5.0 |
|---|---|
| 0.9492600422832981 | 0.9592233009708738 |
| 0.9509202453987731 | 0.9573835480673935 |
| 0.9589041095890412 | 0.9496402877697842 |
| 0.9432404540763675 | 0.9622063329928499 |
| 0.9536945812807881 | 0.9643564356435644 |

As you can see, there isn’t a significant difference in the results. 5.0 is ever so slightly higher on average than the rest, but C-values of 5 and above tend to overfit, so it is not recommended to pick it.

Using the comparison we made above, we can finalize our hyperparameter tuning using the following values – kernel = “rbf” and C = “1.0”.

Running the finished model on the internal testing set yields an f1 score of: 0.95499021526418

## Results

When computing the TestingResults.txt, the labels are as follows:

| Index | Label |
|---|---|
| 0 | 1 |
| 1 | 1 |
| 2 | 0 |
| 3 | 0 |
| 4 | 1 |
| 5 | 1 |
| 6 | 0 |
| 7 | 1 |
| 8 | 1 |
| 9 | 0 |
| 10 | 0 |
| 11 | 0 |
| 12 | 1 |
| 13 | 0 |
| 14 | 0 |
| 15 | 1 |
| 16 | 1 |
| 17 | 1 |
| 18 | 1 |
| 19 | 1 |
| 20 | 0 |
| 21 | 1 |
| 22 | 0 |
| 23 | 0 |
| 24 | 0 |
| 25 | 1 |
| 26 | 0 |
| 27 | 1 |
| 28 | 0 |
| 29 | 1 |
| 30 | 0 |
| 31 | 1 |
| 32 | 0 |
| 33 | 0 |
| 34 | 1 |
| 35 | 0 |
| 36 | 0 |
| 37 | 1 |
| 38 | 1 |
| 39 | 0 |
| 40 | 0 |
| 41 | 0 |
| 42 | 0 |
| 43 | 0 |
| 44 | 0 |
| 45 | 1 |
| 46 | 0 |
| 47 | 1 |
| 48 | 1 |
| 49 | 0 |
| 50 | 1 |
| 51 | 0 |
| 52 | 1 |
| 53 | 0 |
| 54 | 1 |
| 55 | 1 |
| 56 | 1 |
| 57 | 0 |
| 58 | 1 |
| 59 | 0 |
| 60 | 0 |
| 61 | 0 |
| 62 | 0 |
| 63 | 1 |
| 64 | 1 |
| 65 | 0 |
| 66 | 0 |
| 67 | 1 |
| 68 | 1 |
| 69 | 1 |
| 70 | 0 |
| 71 | 0 |
| 72 | 0 |
| 73 | 1 |
| 74 | 0 |
| 75 | 0 |
| 76 | 1 |
| 77 | 1 |
| 78 | 1 |
| 79 | 1 |
| 80 | 1 |
| 81 | 1 |
| 82 | 0 |
| 83 | 1 |
| 84 | 1 |
| 85 | 1 |
| 86 | 1 |
| 87 | 1 |
| 88 | 1 |
| 89 | 1 |
| 90 | 0 |
| 91 | 0 |
| 92 | 0 |
| 93 | 1 |
| 94 | 1 |
| 95 | 1 |
| 96 | 0 |
| 97 | 0 |
| 98 | 1 |
| 99 | 0 |

## Linear Programming

The Linear Programming is mostly done in Python.

First, we read from the Excel file to generate the information we need for the LP programs. I've parsed the input into a DataFrame and modified the column names for easier use. Additionally, the User and Task ID column were split into two for easier access.

With the prepared DataFrame we can start parsing the data into a custom data structure.

The data structure I've opted for is as follows:

A list of tuples with each tuple being:

(userNumber, Restrictions, Variables)

The reasoning behind this is that since each user makes their electricity use separately, we do not apply game theory in their decision making – therefore we should prioritize structuring the data such that we can separate each user and the program that calculates their energy usage.

Once we've generated the common data – the user tasks from the Excel file, we can load the pricing and abnormality data. A method called fileToDf parses the input from the TestingResults.txt.

For each user and each abnormal entry, we use the data that we've collected and the pricing information to combine them into a final weighted restriction for our linear program.

Finally, we combine all restrictions into a single .lp file, which we can run using the LP IDE.

Given that we generate one file for each household for each abnormal entry, that means we generate:

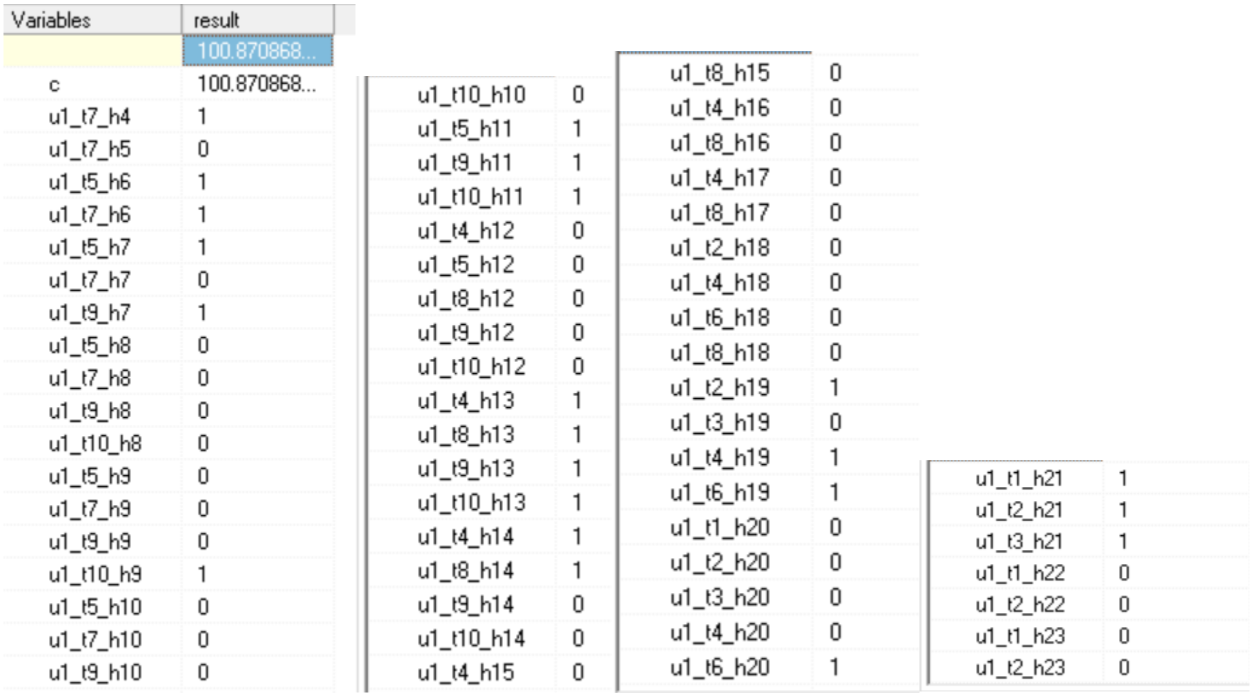
52 abnormal entries * 5 households = 260 .lp files.

For obvious reasons, I cannot display all of the computations in this report, so I will show the results of a single abnormal entry. According to the prediction algorithm, row 0 is abnormal.

Here are the values of row 0:

| | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4.512853401202281 | 3.43658107523701 | 3.68255559134804 | 3.06271757467767 | 3.45627812546856 | 4.02803853654439 | 3.53047024505285 | 4.29234009190491 | 5.01899399964376 | 4.7831896810001 | 5.39649373091393 | 3.62164525353888 | 6.52511405066837 | 4.34263631482957 | 5.85722269007359 | 6.38160238785678 | 6.11551929701169 | 6.29475554382194 | 6.5131445728803 | 5.25018950601471 | 5.913804588632 | 5.12382685690879 | 5.62943812228955 | 5.75354475562452 |

The result from the LP IDE of row 0 for household 1 is as follows:



| Variables | result |
|---|---|
| | 100.870868... |
| c | 100.870868... |
| u1_t7_h4 | 1 |
| u1_t7_h5 | 0 |
| u1_t5_h6 | 1 |
| u1_t7_h6 | 1 |
| u1_t5_h7 | 1 |
| u1_t7_h7 | 0 |
| u1_t9_h7 | 1 |
| u1_t5_h8 | 0 |
| u1_t7_h8 | 0 |
| u1_t9_h8 | 0 |
| u1_t10_h8 | 0 |
| u1_t5_h9 | 0 |
| u1_t7_h9 | 0 |
| u1_t9_h9 | 0 |
| u1_t10_h9 | 1 |
| u1_t5_h10 | 0 |
| u1_t7_h10 | 0 |
| u1_t9_h10 | 0 |
| u1_t10_h10 | 0 |
| u1_t5_h11 | 1 |
| u1_t9_h11 | 1 |
| u1_t10_h11 | 1 |
| u1_t4_h12 | 0 |
| u1_t5_h12 | 0 |
| u1_t8_h12 | 0 |
| u1_t9_h12 | 0 |
| u1_t10_h12 | 0 |
| u1_t4_h13 | 1 |
| u1_t8_h13 | 1 |
| u1_t9_h13 | 1 |
| u1_t10_h13 | 1 |
| u1_t4_h14 | 1 |
| u1_t8_h14 | 1 |
| u1_t9_h14 | 0 |
| u1_t10_h14 | 0 |
| u1_t4_h15 | 0 |
| u1_t8_h15 | 0 |
| u1_t4_h16 | 0 |
| u1_t8_h16 | 0 |
| u1_t4_h17 | 0 |
| u1_t8_h17 | 0 |
| u1_t2_h18 | 0 |
| u1_t4_h18 | 0 |
| u1_t6_h18 | 0 |
| u1_t8_h18 | 0 |
| u1_t2_h19 | 1 |
| u1_t3_h19 | 0 |
| u1_t4_h19 | 1 |
| u1_t6_h19 | 1 |
| u1_t1_h20 | 0 |
| u1_t2_h20 | 0 |
| u1_t3_h20 | 0 |
| u1_t4_h20 | 0 |
| u1_t6_h20 | 1 |
| u1_t1_h21 | 1 |
| u1_t2_h21 | 1 |
| u1_t3_h21 | 1 |
| u1_t1_h22 | 0 |
| u1_t2_h22 | 0 |
| u1_t1_h23 | 0 |
| u1_t2_h23 | 0 |

We can create a table for an easier visualization: (Cost = 100)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 2 | 2 | 0 | 1 | 0 | 3 | 0 | 4 | 2 | 0 | 0 | 0 | 0 | 3 | 1 | 3 | 0 | 0 |

The table for household 2: (Cost = 78)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 0 | 4 | 1 | 0 | 0 | 0 | 3 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 2 | 1 | 0 |

The table for household 3: (Cost = 96)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 2 | 2 | 0 | 1 | 1 | 0 | 0 | 0 | 3 | 0 | 3 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 4 | 2 | 0 |

The table for household 4: (Cost = 92)

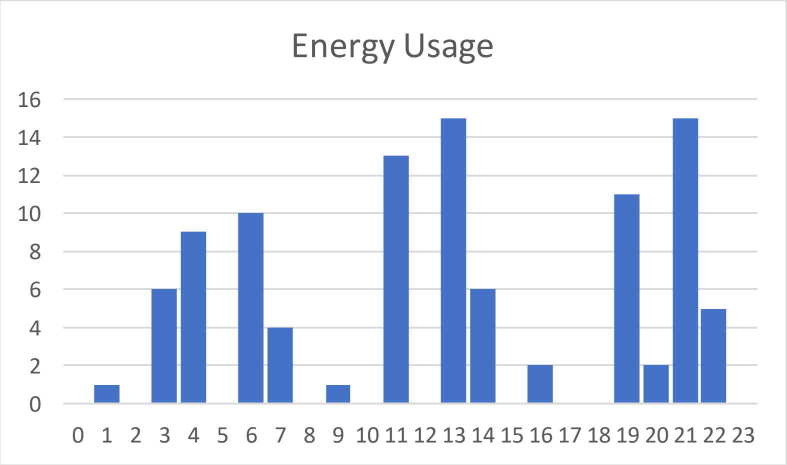
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 3 | 0 | 5 | 2 | 0 | 1 | 0 | 0 | 3 | 0 | 1 | 2 | 0 |

The table for household 5: (Cost = 80)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 2 | 3 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 2 | 0 | 5 | 0 | 0 |

With that, we can compute the total energy usage of the neighbourhood:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 6 | 9 | 0 | 10 | 4 | 0 | 1 | 0 | 13 | 0 | 15 | 6 | 0 | 2 | 0 | 0 | 11 | 2 | 15 | 5 | 0 |



The rest of the programs are in the LPs directory of the submission.

## Project Management

The project has been managed through github.

https://github.com/DichoMire/CyberPhysical