Lab 3: Detecting Bad Sensors in Power System Monitoring

In this lab, our goal is to detect bad sensor data measured on the IEEE 14 bus test system shown below. The power flow equations that couple the voltages and power flows are nonlinear in nature, as discussed in class. We will load the sensor data from the file 'sensorData14Bus.csv', and utilize SVM to perform the bad data detection. We aim to understand how various parameters such as the nature of the corrupt data, the number of corrupt data, etc., affect our abilities to classify the data.

First, we need to call the needed libraries

In [50]:
```python
import numpy as np
import os
from sklearn import preprocessing, svm
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from IPython.display import Image
import pandas as pd
```

Loading the data

Load the sensor data from the IEEE 14 bus test system, that has 14 buses and 20 branches. The data has been generated by adding a small noise to feasible voltages and power flows.

```
Columns 1-14 contain bus voltage magnitudes.

Columns 15-28 contain bus voltage phase angles.

Columns 29-48 contain real power flow on all branches.

Columns 49-68 contain reactive power flow on all branches.
```

In [51]:
```python
nBuses = 14
nBranches = 20

# Select the bus numbers you monitor. For convenience, we have selected it for you.
# The '-1' makes them columns as per Python's convention of starting to number
# from 0.
busesToSample = np.array([1, 2, 5, 10, 13]) - 1
columnsForBuses = np.concatenate((busesToSample, busesToSample + 14))

# Select the branches that you monitor.
branchesToSample = np.array([1, 3, 5, 10, 11, 15, 17, 20]) - 1
columnsForBranches = np.concatenate((branchesToSample + 28,
                                     branchesToSample + 48))
```

```
# Load the sensor data from the file 'sensorData14Bus.csv' in 'X' from the columns
# specified in 'columnsForBuses' and 'columnsForBranches'. The csv file is comma
# separated. Read a maximum of 5000 lines. Make sure your data is a numpy array
# with each column typecast as 'np.float32'.
X = np.genfromtxt('sensorData14Bus.csv', dtype=np.float32, delimiter=',',
                  usecols=np.concatenate((columnsForBuses, columnsForBranches)),
                  max_rows=5000)

nDataPoints = np.shape(X)[0]
nFeatures = np.shape(X)[1]

print("Loaded sensor data on IEEE 14 bus system.")
print("Number of data points = %d, number of features = %d"
      % (nDataPoints, nFeatures))
```

```
Loaded sensor data on IEEE 14 bus system.
Number of data points = 5000, number of features = 26
```

Curroption Models

Intentionally corrupt the first 'nCorrupt' rows of the data by adding a quantity to one or two sensor measurements that is not representative of our error model. We aim to study what nature of corruption is easier or difficult to detect. Specifically, we shall study 3 different models:

```
 1. 'corruptionModel' = 1 : Add a random number with a bias to one of
the measurements.

 2. 'corruptionModel' = 2 : Add a random number without bias to one of
the measurements.

 3. 'corruptionModel' = 3 : Add a random number with a bias to both the
measurements.
```

In all these cases, we will multiply the sensor data by either a uniform or a normal random number multiplied by 'multiplicationFactor'.

In [52]:

```
# Choose a corruption model.
nCorrupt = int(nDataPoints/3)
corruptionModel = 1
multiplicationFactor = 0.5

# Choose which data to tamper with, that can be a voltage magnitude,
# voltage phase angle, real power flow on a branch, reactive power flow
# on a branch. We create functions to extract the relevant column to
# corrupt the corresponding data in the 'ii'-th bus or branch.
voltageMagnitudeColumn = lambda ii: ii

voltageAngleColumn = lambda ii: ii + np.shape(busesToSample)[0]

realPowerColumn = lambda ii: ii + 2*np.shape(busesToSample)[0]
reactivePowerColumn = lambda ii: ii + 2*np.shape(busesToSample)[0] + np.shape(branche

# Encode two different kinds of columns to corrupt.
# Option 1: Corrupt real power columns only.
# Option 2: Corrupt real power and voltage magnitude.
columnsToCorruptOption = 2

if columnsToCorruptOption == 1:
    columnsToCorrupt = [realPowerColumn(1),
                        realPowerColumn(2)]
else:
    columnsToCorrupt = [voltageMagnitudeColumn(0),
                        realPowerColumn(1)]

# Corrupt the data appropriately, given the options.
for index in range(nCorrupt):

    if corruptionModel == 1:
        X[index, columnsToCorrupt[0]] \
            *= (1 + multiplicationFactor * np.random.rand())
    elif corruptionModel == 2:
        X[index, columnsToCorrupt[0]] \
            *= (1 + multiplicationFactor * np.random.randn())
    else:
        X[index, columnsToCorrupt[0]] \
            *= (1 + multiplicationFactor * np.random.rand())
        X[index, columnsToCorrupt[1]] \
            *= (1 + multiplicationFactor * np.random.rand())
```

It is always a good practice to scale your data to run SVM. Notice that we are cheating a little when we scale the entire data set 'X', because our training and test sets are derived from 'X'. Ideally, one would have to scale the training and test sets separately. Create the appropriate labels and shuffle the lists 'X' and 'Y' together.

In [53]:

```
X = preprocessing.StandardScaler().fit_transform(X)

# Create the labels as a column of 1's for the first 'nCorrupt' rows, and
# 0's for the rest.
Y = np.concatenate((np.ones(nCorrupt), np.zeros(nDataPoints-nCorrupt)))


# Shuffle the features and the labels together.
XY = list(zip(X, Y))
np.random.shuffle(XY)
X, Y = zip(*XY)
```

Task 1 (10 points)

Split the dataset into two parts: training and testing. Store the training set in the variables 'trainX' and 'trainY'. Store the testing set in the variables 'testX' and 'testY. Reserve 20% of the data for testing. The function 'train_test_split' may prove useful.

In [54]:

```
# Enter your code here
trainX, testX, trainY, testY = train_test_split(X, Y, test_size=0.2)
```

Task 2 (10 points)

Define the support vector machine classifier and train on the variables 'trainX' and 'trainY'. Use the SVC library from sklearn.svm. Only specify three hyper-parameters: 'kernel', 'degree', and 'max_iter'. Limit the maximum number of iterations to 100000 at the most. Set the kernel to be a linear classifier first. You may have to change it to report the results with other kernels. The parameter 'degree' specifies the degree for polynomial kernels. This parameter is not used for other kernels. The functions 'svm.SVC' and 'fit' will prove useful.

In [55]:

```python
# Enter your code here
svm_class = svm.SVC(kernel='linear', degree=3, max_iter=100000)
svm_class.fit(trainX, trainY)
```

Out[55]:

```
▾                    SVC

SVC(kernel='linear', max_iter=100000)
```

Task 3 (10 points)

Predict the labels on the 'testX' dataset and store them in 'predictY'.

In [56]:

```python
# Enter your code here
predictY = svm_class.predict(testX)
```

Task 4 (10 points)

Print the 'classification_report' to see how well 'predictY' matches with 'testY'.

In [57]:

```python
# Enter your code here
print(classification_report(testY, predictY))
```

```
              precision    recall  f1-score   support

         0.0       0.97      1.00      0.98       681
         1.0       0.99      0.93      0.96       319

    accuracy                           0.98      1000
   macro avg       0.98      0.97      0.97      1000
weighted avg       0.98      0.98      0.98      1000
```

Print svm's internal accuracy score as a percentage.

In [58]:

```python
# Enter your code here
print(f'svm\'s internal accuracy score = {svm_class.score(testX, testY)*100.0}%')
```

```
svm's internal accuracy score = 97.7%
```

Task 5

We would like to compare 'classification_report' with this score for various runs. Let us consider the following cases:

Case 1:

Only have sensor measurements from the first 5 branches. Choose option 1 in the 'columnsToCorruptOption'. Examine how well linear kernels perform when 'corruptionModel' = 1, 'corruptionModel' = 2, and 'corruptionModel'= 3. In case linear kernels do not perform well, you may try 'rbf' or polynomial kernels with degree 2.

Case 2:

Choose 'corruptionModel = 1' with 'linear' kernel. Does it pay to monitor voltage magnitudes than power flows? In other words, do you consistently get better results when you choose 'columnsToCorruptOption' as 2? Make these judgements using the average score of at least 5 runs.

Your task is to investigate the above two cases. You may add a few 'Markdown' and 'Code' cells below with your comments, code, and results. You can also report your results as a pandas DataFrame. You are free to report your results in your own way.

In [59]:

```
# Function copied and pasted from above in code with modifications
def corrupt(X, nCorrupt, corruptionModel, busesToSample, branchesToSample):
    multiplicationFactor = 0.5
    voltageMagnitudeColumn = lambda ii: ii
    voltageAngleColumn = lambda ii: ii + np.shape(busesToSample)[0]
    realPowerColumn = lambda ii: ii + 2*np.shape(busesToSample)[0]
    reactivePowerColumn = lambda ii: ii + 2*np.shape(busesToSample)[0] + np.shape(bra
    columnsToCorruptOption = 1
    if columnsToCorruptOption == 1:
        columnsToCorrupt = [realPowerColumn(1),
                            realPowerColumn(2)]
    else:
        columnsToCorrupt = [voltageMagnitudeColumn(0),
                            realPowerColumn(1)]
    for index in range(nCorrupt):
        if corruptionModel == 1:
            X[index, columnsToCorrupt[0]] \
                *= (1 + multiplicationFactor * np.random.rand())
        elif corruptionModel == 2:
            X[index, columnsToCorrupt[0]] \
                *= (1 + multiplicationFactor * np.random.randn())
        else:
            X[index, columnsToCorrupt[0]] \
                *= (1 + multiplicationFactor * np.random.rand())
            X[index, columnsToCorrupt[1]] \
                *= (1 + multiplicationFactor * np.random.rand())

# Function reused from before
def func(X, Y, nCorrupt):
    X = preprocessing.StandardScaler().fit_transform(X)
    Y = np.concatenate((np.ones(nCorrupt), np.zeros(nDataPoints-nCorrupt)))
    XY = list(zip(X, Y))
    np.random.shuffle(XY)
    X, Y = zip(*XY)
    return X, Y

for i in range(1,4):
    # From also copied and pasted from above
    busesToSample = np.array([1, 2, 5, 10, 13]) - 1
    columnsForBuses = np.concatenate((busesToSample, busesToSample + 14))
    branchesToSample = np.array([1, 2, 3, 4, 5]) - 1
    columnsForBranches = np.concatenate((branchesToSample + 28,
                                         branchesToSample + 48))
    X = np.genfromtxt('sensorData14Bus.csv', dtype=np.float32, delimiter=',',
                       usecols=np.concatenate((columnsForBuses, columnsForBranches))
                       max_rows=5000)
    nDataPoints = np.shape(X)[0]
    nFeatures = np.shape(X)[1]
    nCorrupt = int(nDataPoints/3)
    corrupt(X, nCorrupt, i, busesToSample, branchesToSample)

    X, Y = func(X, Y, nCorrupt)
    trainX, testX, trainY, testY = train_test_split(X, Y, test_size=0.2)
    svm_class = svm.SVC(kernel='rbf', max_iter=100000) # Tried rbf instead of linear
    svm_class.fit(trainX, trainY)

    predictY = svm_class.predict(testX)
    print('\ncorruptionModel = ', i)
    print(classification_report(testY, predictY))
    print(f'svm\'s internal accuracy score = {svm_class.score(testX, testY)*100.0}%')
```

```
corruptionModel =  1
              precision    recall  f1-score   support

         0.0       0.91      1.00      0.96       672
         1.0       1.00      0.81      0.89       328

    accuracy                           0.94      1000
   macro avg       0.96      0.90      0.92      1000
weighted avg       0.94      0.94      0.94      1000

svm's internal accuracy score = 93.7%

corruptionModel =  2
              precision    recall  f1-score   support

         0.0       0.80      1.00      0.89       652
         1.0       1.00      0.53      0.70       348

    accuracy                           0.84      1000
   macro avg       0.90      0.77      0.79      1000
weighted avg       0.87      0.84      0.82      1000

svm's internal accuracy score = 83.8%

corruptionModel =  3
              precision    recall  f1-score   support

         0.0       0.94      1.00      0.97       673
         1.0       1.00      0.86      0.92       327

    accuracy                           0.95      1000
   macro avg       0.97      0.93      0.95      1000
weighted avg       0.96      0.95      0.95      1000

svm's internal accuracy score = 95.39999999999999%
```

Answer:

The linear kernels vary in accuracy with model 3 being the most accurate and model 2 being the least accurate. Model 2 being the least accurate makes sense because it adds corruption without bias making it most inaccurate. Model 3 being the best also makes sense because more errors were available to train the classifier.

In [60]:

```python
# Function copied and pasted from above in code with modifications with edits
def corrupt(X, nCorrupt, corruptionModel, busesToSample, branchesToSample, columnsToC
    multiplicationFactor = 0.5
    voltageMagnitudeColumn = lambda ii: ii
    voltageAngleColumn = lambda ii: ii + np.shape(busesToSample)[0]
    realPowerColumn = lambda ii: ii + 2*np.shape(busesToSample)[0]
    reactivePowerColumn = lambda ii: ii + 2*np.shape(busesToSample)[0] + np.shape(bra
    if columnsToCorruptOption == 1:
        columnsToCorrupt = [realPowerColumn(1),
                            realPowerColumn(2)]
    else:
        columnsToCorrupt = [voltageMagnitudeColumn(0),
                            realPowerColumn(1)]
    for index in range(nCorrupt):
        if corruptionModel == 1:
            X[index, columnsToCorrupt[0]] \
                *= (1 + multiplicationFactor * np.random.rand())
        elif corruptionModel == 2:
            X[index, columnsToCorrupt[0]] \
                *= (1 + multiplicationFactor * np.random.randn())
        else:
            X[index, columnsToCorrupt[0]] \
                *= (1 + multiplicationFactor * np.random.rand())
            X[index, columnsToCorrupt[1]] \
                *= (1 + multiplicationFactor * np.random.rand())

# Function mostly copied and pasted with the add of an extra loop and the average cou
for i in range(1,3):
    average = 0
    for j in range(5):
        busesToSample = np.array([1, 2, 5, 10, 13]) - 1
        columnsForBuses = np.concatenate((busesToSample, busesToSample + 14))
        branchesToSample = np.array([1, 3, 5, 10, 11, 15, 17, 20]) - 1
        columnsForBranches = np.concatenate((branchesToSample + 28, branchesToSample
        X = np.genfromtxt('sensorData14Bus.csv', dtype=np.float32, delimiter=',',
                          usecols=np.concatenate((columnsForBuses, columnsForBranches
                          max_rows=5000)
        nDataPoints = np.shape(X)[0]
        nFeatures = np.shape(X)[1]
        nCorrupt = int(nDataPoints/3)
        corrupt(X, nCorrupt, 1, busesToSample, branchesToSample, i)

        X, Y = func(X, Y, nCorrupt)
        trainX, testX, trainY, testY = train_test_split(X, Y, test_size=0.2)
        svm_class = svm.SVC(kernel='linear', max_iter=100000)
        svm_class.fit(trainX, trainY)
        predictY = svm_class.predict(testX)
        average += svm_class.score(testX, testY)
    print('columnsToCorruptOption = ', i)
    print(f'svm\'s average score = {average/nRuns*100:.2f}%\n')
```

```
C:\Users\matt3\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\svm
\_base.py:299: ConvergenceWarning: Solver terminated early (max_iter=100000).  Consid
er pre-processing your data with StandardScaler or MinMaxScaler.
  warnings.warn(
C:\Users\matt3\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\svm
\_base.py:299: ConvergenceWarning: Solver terminated early (max_iter=100000).  Consid
er pre-processing your data with StandardScaler or MinMaxScaler.
  warnings.warn(
C:\Users\matt3\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\svm
\_base.py:299: ConvergenceWarning: Solver terminated early (max_iter=100000).  Consid
er pre-processing your data with StandardScaler or MinMaxScaler.
  warnings.warn(
C:\Users\matt3\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\svm
\_base.py:299: ConvergenceWarning: Solver terminated early (max_iter=100000).  Consid
er pre-processing your data with StandardScaler or MinMaxScaler.
  warnings.warn(
columnsToCorruptOption =  1
svm's average score = 75.44%

columnsToCorruptOption =  2
svm's average score = 97.50%
```

Answer:

columnsToCorruptOption 2 has a higher average score than option 1. Since option 2 corrupts more columns (voltage and real values), this means more corruption leads to a better classifier and estimation. This is also shown in case 1 where corription option 3 performed better than the other 2.