Carter Falkenberg
Project 1, ECE 553
Note: All run-times are on 2021 M1 MacBook Air, 8GB ram.

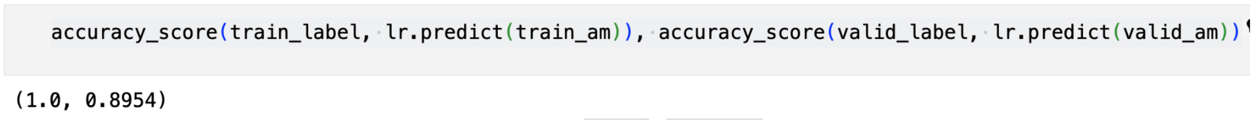
LoS/NLoS Classification

I considered a few methods for data preprocessing: modulus, angles, real values, imaginary values, (modulus, angles), and (real, imaginary). After trying all of these methods on smaller bootstrapped datasets, I found that modulus alone and (modulus, angles) seemed to perform the best. Since modulus alone was the method proposed in the example, I decided to use (modulus, angles) for a novel method. I did this in 3 steps:

1. Separate angle and modulus into 2 different datasets, each CSI matrix of shape (4, 1632) after squeezing the data, the whole dataset being shape (N, 4, 1632)
2. Stack the 2 separated datasets together into one dataset on the third dimension, creating a dataset of size (N, 4, 1632, 2). The final dimension being (modulus, angle) for a given CSI entry.
3. Finally, reshape the dataset to be of matrix form, simply by reshaping to shape (N, 4*1632*2).

After this, I normalized the data to have mean = 0 and variance = 1. I tried min-max scaling as well, but it was noticeably less effective compared to typical normalization. It seemed that when using min-max scaling, the data had a very low variance and it was therefore more difficult for the machine learning algorithms to learn.

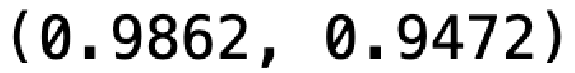
For NLoS/LoS classification, I tried Logistic Regression, SVM (svc), and RandomForestClassifier, all using scikit-learn. I was going to also use KNN, but due to the high dimensionality of the data (and the success of other methods), I decided not to use it. I first tried Logistic Regression with L2 regularization (C=1). This did quite well, having an accuracy score of 100% on the training dataset, even with regularization, and 90% on the test dataset. It was quick to run compared to most, finishing in less than 10 minutes.



```
accuracy_score(train_label, lr.predict(train_am)), accuracy_score(valid_label, lr.predict(valid_am))
```

```
(1.0, 0.8954)
```

Before trying to adjust the regularization parameter, I decided to try SVM and RandomForest. SVM did slightly better than Logistic Regression, having an accuracy score of 98.62% on the training dataset and 94.72% on the testing dataset. I used a squared L2 penalty for standardization (C=1). The process of svm.fit and sv.predict took around 2 hours.



```
(0.9862, 0.9472)
```

Finally, I used RandomForest. I used 100 estimators, with no adjustments to max_depth or min_samples_split. I figured I could tune the parameters after it was overfitting. Fortunately, this was not needed, as this method achieved 100% accuracy on the training dataset, and 99% accuracy in the validation dataset, while running in only 20 minutes.

```
accuracy_score(train_label, rf.predict(train_am)), accuracy_score(valid_label, rf.predict(valid_am))
```

```
(1.0, 0.9906)
```

For my ANN, I used a feed forward neural network with Linear layers, and ReLU on all layers besides the output, which I used Sigmoid, to squeeze the result between 0 and 1. I also used dropout (p=0.5) on all layers. The model that ended up performing the best was the following:

```
Model(
  (fcs): ModuleList(
    (0): Linear(in_features=13056, out_features=1000, bias=True)
    (1): Linear(in_features=1000, out_features=100, bias=True)
  )
  (output): Linear(in_features=100, out_features=1, bias=True)
  (relu): ReLU()
  (dropout): Dropout(p=0.5, inplace=False)
  (sigmoid): Sigmoid()
)
```

(using SGD with a learning rate of 0.01, and a batch size of 128).
I wanted to start with the simplest possible ANN I could to see if it could fit the data and slowly add to it if needed. Thus, I started with 1 input layer, 1 hidden layer, and 1 output layer. This worked extremely well on the training data, but was giving poor results on the testing data, doing worse than all machine learning techniques. I realized that the model was too complex, as it was definitely overfitting. Since the model architecture worked so quickly (i.e. minimal number of epochs and quick training time per epoch), I figured that a dropout technique would help the overfitting issue, and it should learn the training data just as well, albeit over more epochs. I started with p=0.1, and began to slowly add to the dropout, maxing at p=.7. I trained all until the validation loss plateaued, and found that p=0.5 performed the best on the validation data (they all eventually learned the training data similarly well, very low losses and 100% accuracy). With this technique, I was able to get a binary-cross entropy (BCE) loss of .002 and 0.06 on the training and validation data, respectively, and accuracy scores of 100% and 98%.
After 30 minutes of training and 200 epochs:

```
Epoch [198/200], Training Loss: 0.0027006082236766815, Test Loss: 0.0630071684718132
Epoch [199/200], Training Loss: 0.0023100057151168585, Test Loss: 0.07044312357902527
Epoch [200/200], Training Loss: 0.0022847491782158613, Test Loss: 0.06520023941993713
```

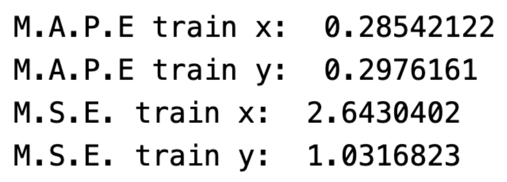
```
(1.0, 0.98)
```

What is surprising is that RandomForestClassifier actually performed better than the neural network! I suspect that this is because the data is not too complex (considering the task being binary classification) and RandomForest is very good at avoiding overfitting due to its separate 100 classifiers.

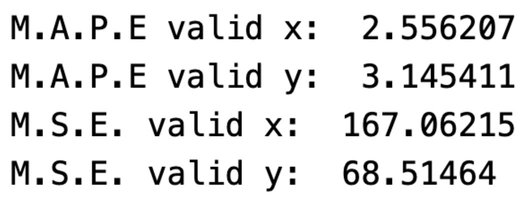
Comparison of performance for LoS/NLoS classification:

Model	Accuracy score % (train)	Accuracy score % (valid)
Logistic Regression	**100%**	89.54%
Random Forest	**100%**	**99.06%**
Support Vector Classifier	98.62%	94.72%
Neural Network	**100%**	98%

Coordinate prediction
I used the same data preprocessing method for coordinate prediction that I used for classification. It also seemed to perform the best out of all options, similar to the modulus values alone. I tried 3 machine learning algorithms: LinearRegression, SVM, and DecisionTree. I decided to use both mean absolute percentage error (MAPE) and mean squared error (MSE) to evaluate the models for coordinate prediction, as they seemed relevant given the continuous nature of the target variables. I trained 2 models for each machine learning algorithm, one to predict the X coordinate, and one to predict the Y coordinate. For linear regression, the model did not perform very well, it seemed to overfit to the training data:

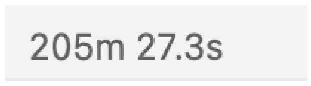


```
M.A.P.E train x:  0.28542122
M.A.P.E train y:  0.2976161
M.S.E. train x:  2.6430402
M.S.E. train y:  1.0316823
```

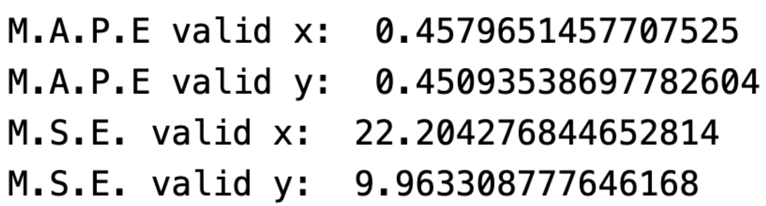


```
M.A.P.E valid x:  2.556207
M.A.P.E valid y:  3.145411
M.S.E. valid x:  167.06215
M.S.E. valid y:  68.51464
```

SVM fit the data in 65 minutes, but then I had to stop running after 3+ hours of attempting to run svm.predict(), as it was slowing down my computer considerably and showed no signs of completion. At this point, I decided that I would try DecisionTree, and if it did not do much better than logistic regression, I would use some sort of feature selection or PCA to simplify the dataset and then try SVM again. However, DecisionTree ended up being a vast improvement. In order to train the decision tree, I used a GraphSearch style algorithm to pick the best parameters (max_depth and min_samples_split). When I trained the DecisionTree without any parameters, its max depth was 29. So, I set the possible values for max_depth to [10, 15, 20, 25, 30]. Since the default of min_samples_split is 2, I set the possible values for min_samples_split to [2, 4, 6, 8, 10]. Testing all combinations took 77 minutes. The best result



```
205m 27.3s
```



```
M.A.P.E valid x:  0.4579651457707525
M.A.P.E valid y:  0.45093538697782604
M.S.E. valid x:  22.204276844652814
M.S.E. valid y:  9.963308777646168
```

ended up being max_depth = 25, min_samples_split = 4. This performed considerably better than linear regression:

```
M.A.P.E train x:  0.006232623054380457
M.A.P.E train y:  0.006992053435375219
M.S.E. train x:  0.08776657553184125
M.S.E. train y:  0.03687887105764928
```

The neural network was much trickier for coordinate prediction than it was for classification. I ran into many difficulties, including exploding ReLU issue, incorrect activation function for output layer, learning rate optimization issues, and massive amounts of overfitting. After much tampering, however, I was able to find a working combination of parameters.
For all training, I used a batch size of 64. This seemed to be a good balance between speed and performance. Initially, I tried using one hidden layer, but this was not enough for even the training data to learn (trying lr=0.1,.01,.001). This was also the case with 2 hidden layers. Once I had 3 hidden layers, I ran into an issue where I kept getting repetitive predictions. I did some research and determined it was an exploding ReLU issue, so I changed the activation function to ELU and there were no more issues (though I needed to use a low learning rate to avoid NaN results with ELU). Now, however, the model was overfitting very quickly. I knew I couldn't remove a layer, so I decided to try dropout. Any dropout higher than around p = 0.3 caused the model to never learn the training data, so I tested p = 0.1, 0.2, and 0.3 and found p=0.1 allowed for minimal overfitting with still great success on the training dataset. To try to improve, I attempted to use Adam as my optimizer, but this proved to be inferior to SGD with lr = 0.001. After around 2 hours of training (~200 epochs), the results were as follows:

MAPE for (training, validation):
```
(0.21980761, 0.49818236)
```

MSE for (training, validation):
```
(1.9301880598068237, 5.239727973937988)
```

For coordinate prediction, the neural network ended up with the best performance when taking into account MSE and MAPE, with DecisionTree closely behind. Linear regression struggled to predict the coordinates well at all.

Comparison of performance for coordinate prediction:

Model	MSE (train)	MAPE (train)	MSE (valid)	MAPE (valid)
Linear Regression	1.843	.2865	123.533	2.856
DecisionTree	**0.0555**	**0.0065**	16.722	**.4579**
Neural Network (3 hidden layers)	1.9302	.2198	**5.2397**	.4982