Section 1: Introduction

This report details the answer to the assignment for technical assessment (Data Science). The data set is collected in an office environment where a number of Bluetooth Low Energy (BLE) receivers are installed. A BLE beacon is then used as a transmitter to broadcast beacon signals while BLE receivers are receiving the broadcasted beacon signal. Based on the received signals, a wireless fingerprint is created and the fingerprint data can be used to train a model so that location of BLE beacon can be predicted. There are two datasets, Offline and Online dataset.

Hypothesis

The main hypothesis is that there's a moderate to strong correlation between the signal strengths and the position of the BLE receivers relative to the BLE beacons. If the hypothesis is true, it would be possible to use the model to predict the location of BLE beacons using the provided data. Otherwise, using machine learning to predict the location of BLE beacons would not yield any meaningful results.

It is assumed that the ambient signals from other devices and building layout does not affect the readings of the BLE transmitters and receivers. Further limitations are discussed after the conclusion.

Initialization

To begin, the relevant libraries are imported:

In [1]:
```
from os import getcwd
import pandas as pd
import numpy as np
import plotly.express as px
from plotly.subplots import make_subplots
import plotly.graph_objects as go
from sklearn.impute import KNNImputer
from sklearn.multioutput import MultiOutputRegressor
#from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
from xgboost import XGBRegressor
```

Next, the datasets will be loaded:

In [2]:
```
wd = getcwd()
x_train_file = wd + '/offline/X_train.csv'
y_train_file = wd + '/offline/y_train.csv'
x_test_file = wd + '/online/X_test.csv'
y_test_file = wd + '/online/y_test.csv'
x_pred_file = wd + '/online/X_test_submission.csv'
output_file = wd + '/submissions.csv'

offline_pininfo_file = wd + '/offline/PinInfo.csv' #train
online_pininfo_file = wd + '/online/PinInfo.csv' #test
```

Reading CSV files into Pandas dataframes:

In [3]:
```
x_train_df = pd.read_csv(x_train_file)
y_train_df = pd.read_csv(y_train_file)
x_test_df = pd.read_csv(x_test_file)
y_test_df = pd.read_csv(y_test_file)
x_pred_df = pd.read_csv(x_pred_file)

offline_pininfo_df = pd.read_csv(offline_pininfo_file) #train
online_pininfo_df = pd.read_csv(online_pininfo_file) #test
```

Section 2:Data Exploration

Basic exploration

The dimension of the train, test and test prediction (pred):

In [4]:
```
x_train_df.shape
```

Out[4]: (1575, 17)

In [5]:
```
x_test_df.shape
```

Out[5]: (291, 17)

In [6]:
```
x_pred_df.shape
```

Out[6]: (288, 17)

Description of the data distribution for x_train and x_test. This provides the data distribution for every column, including:

1. Cell count (not including NaN)
2. Mean
3. Standard deviation
4. Minimum value
5. First quartile value (25th percentile)
6. Second quartile value (50th percentile)
7. Third quartile value (75th percentile)
8. Maximum value

In [7]:
```
x_train_df.describe()
```

Out[7]:

	CD4533FFC0E1	D2B6503554D7	D7EE034361F8	DD697EA75B68	DF231643E227	E43355CA8B96	E6D9D20DD197	E8FD0B453DC4	F1307ECB3B90	F1EDAF28E08A	F69A8(
count	1500.000000	1329.000000	1489.000000	1480.000000	1418.000000	1054.000000	836.000000	1153.000000	1387.000000	818.000000	1413
mean	-85.307795	-85.906442	-86.432911	-86.708529	-86.057089	-84.928891	-85.388299	-88.843995	-83.073399	-86.576718	-84
std	7.447779	5.079135	7.624345	6.356746	7.100093	7.749371	9.042254	6.640661	7.829528	8.676981	7
min	-98.000000	-97.000000	-98.000000	-96.800000	-98.000000	-98.000000	-98.000000	-98.000000	-98.000000	-98.000000	-98
25%	-90.333333	-89.294118	-92.350000	-91.648810	-92.534091	-91.732692	-93.584936	-93.931034	-88.900000	-93.154687	-90
50%	-86.635255	-86.833333	-87.384615	-88.045455	-86.414216	-86.130682	-87.488511	-91.153846	-84.545455	-89.298246	-85
75%	-81.480249	-83.653846	-81.526316	-83.747984	-81.166493	-79.716912	-79.417360	-84.900000	-77.288690	-80.155000	-79
max	-63.967742	-67.121212	-65.500000	-66.818182	-69.000000	-69.000000	-63.626087	-71.888889	-64.970588	-67.476923	-66

In [8]:
```
x_test_df.describe()
```

Out[8]:

	CD4533FFC0E1	D2B6503554D7	D7EE034361F8	DD697EA75B68	DF231643E227	E43355CA8B96	E6D9D20DD197	E8FD0B453DC4	F1307ECB3B90	F1EDAF28E08A	F69A8(
count	251.000000	219.000000	265.000000	206.000000	195.000000	220.000000	142.000000	134.000000	262.000000	223.000000	209
mean	-84.900196	-87.666215	-76.388799	-84.259426	-88.090399	-80.602907	-82.694950	-90.531992	-78.496438	-80.587239	-80
std	6.683748	5.756873	8.002872	5.232772	5.063967	5.466703	11.747603	4.644202	7.843029	8.682446	3
min	-96.000000	-98.000000	-98.000000	-98.000000	-97.000000	-92.500000	-98.000000	-98.000000	-98.000000	-96.000000	-89
25%	-89.375000	-92.000000	-83.600000	-87.000000	-92.000000	-84.083333	-94.000000	-94.000000	-86.000000	-85.645833	-83
50%	-87.000000	-89.000000	-72.750000	-82.708333	-88.200000	-80.000000	-89.000000	-92.000000	-76.062500	-79.000000	-81
75%	-84.000000	-83.550000	-68.857143	-80.050000	-85.200000	-75.702381	-68.538462	-86.000000	-73.375000	-71.000000	-78
max	-70.000000	-76.000000	-66.500000	-76.500000	-76.000000	-72.000000	-67.666667	-82.500000	-66.000000	-67.333333	-75

Adding a column to label the original dataset:

In [9]:
```
len_x_train_df = len(x_train_df)
len_x_test_df = len(x_test_df)
len_x_pred_df = len(x_pred_df)
dataset_label_text_df = pd.DataFrame(['train'] * len_x_train_df + ['test'] * len_x_test_df + ['pred'] * len_x_pred_df,columns=['dataset'])
```

Combining all the X-values from x_train, x_test and x_pred to visualize the combined distribution:

In [10]:
```
x_comb_df = pd.concat([x_train_df,x_test_df,x_pred_df],axis=0)
x_comb_df = x_comb_df.reset_index(drop=True) #The index of the original dataframes are dropped to merge with the dataset_label_df
x_comb_df = pd.concat([x_comb_df,dataset_label_text_df],axis=1)
```

Combining the pininfo values from the online and offline datasets for visualization:

In [11]:
```
onoff_pin_label_df = pd.DataFrame(['Off'] * len(offline_pininfo_df) + ['On'] * len(online_pininfo_df),columns=['status'])
comb_pininfo_df = pd.concat([offline_pininfo_df,online_pininfo_df],axis=0)
comb_pininfo_df = comb_pininfo_df.reset_index(drop=True)
comb_pininfo_df = pd.concat([comb_pininfo_df,onoff_pin_label_df],axis=1)
```

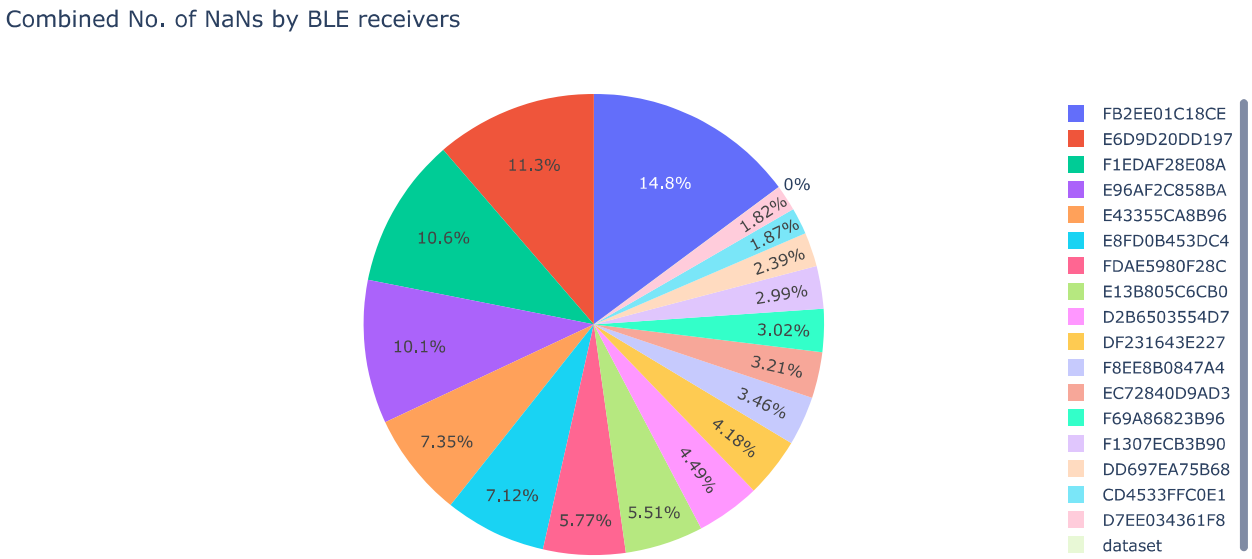
Distribution of NaNs by columns:

In [12]:
```
col_list = x_comb_df.columns
col_na_list = []
for i in col_list:
    col_na_list.append(x_comb_df[i].isna().sum())
col_na_df = pd.concat([pd.DataFrame(col_list,columns=['BLE']),pd.DataFrame(col_na_list,columns=['No. of NaNs'])],axis=1)
```

Visualizations

Visualizing the total no. of NaNs by BLE (Figure 1) :

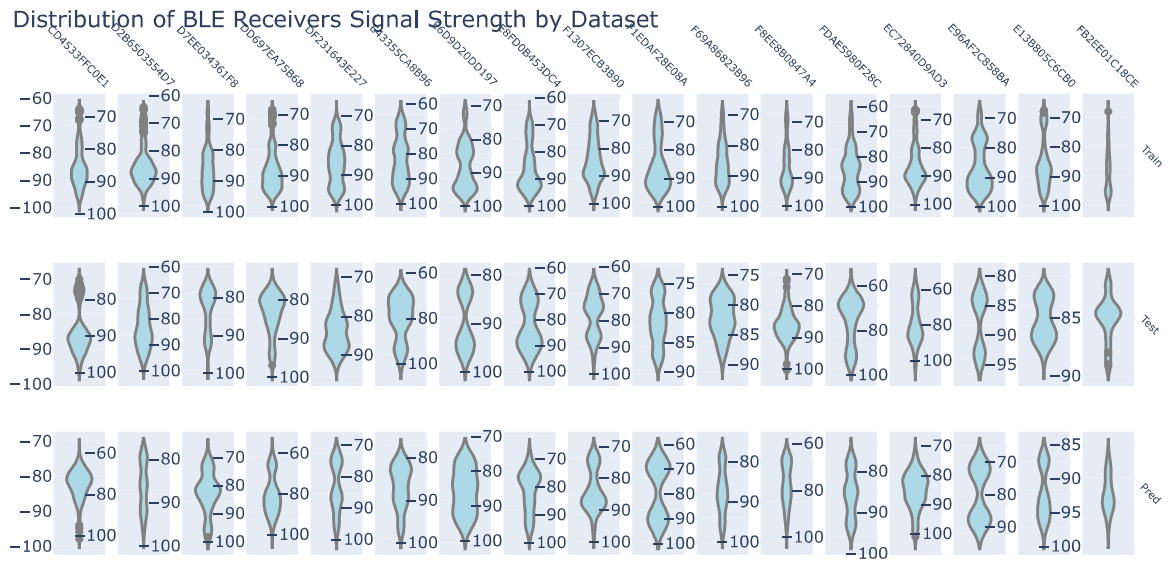
In [13]:
```
fix_na = px.pie(col_na_df,values='No. of NaNs',names='BLE',title='Combined No. of NaNs by BLE receivers')
fix_na.show()
```



The pie chart above (Figure 1) shows the combined number of missing data (NaN) by BLE receiver. It is shown that there are more missing data for some receivers compared to others.

Visualizing the data point distributions in a violin-plot (Figure 2):

In [14]:
```
row_titles = ['Train','Test','Pred']
fig_subplot = make_subplots(rows=3,cols=17,row_titles=row_titles,column_titles=list(col_list))
df_list = [x_train_df,x_test_df,x_pred_df]
for i,j in enumerate(df_list):
    for k,l in enumerate(j.columns):
        fig_subplot.append_trace(go.Violin(y=j[l].values,name=str(l),fillcolor='lightblue',line_color='grey'),row=i+1,col=k+1)
        #fig_subplot.append_trace(go.Violin(x=j.columns,y=j[j[l]],name=str(l)),row=i+1,col=1)
fig_subplot.update_xaxes(showticklabels=False)
fig_subplot.update_annotations(font=dict(size=8))
fig_subplot.update_annotations(textangle=45)
fig_subplot.layout.update(showlegend=False,title_text = 'Distribution of BLE Receivers Signal Strength by Dataset')
fig_subplot.show()
```

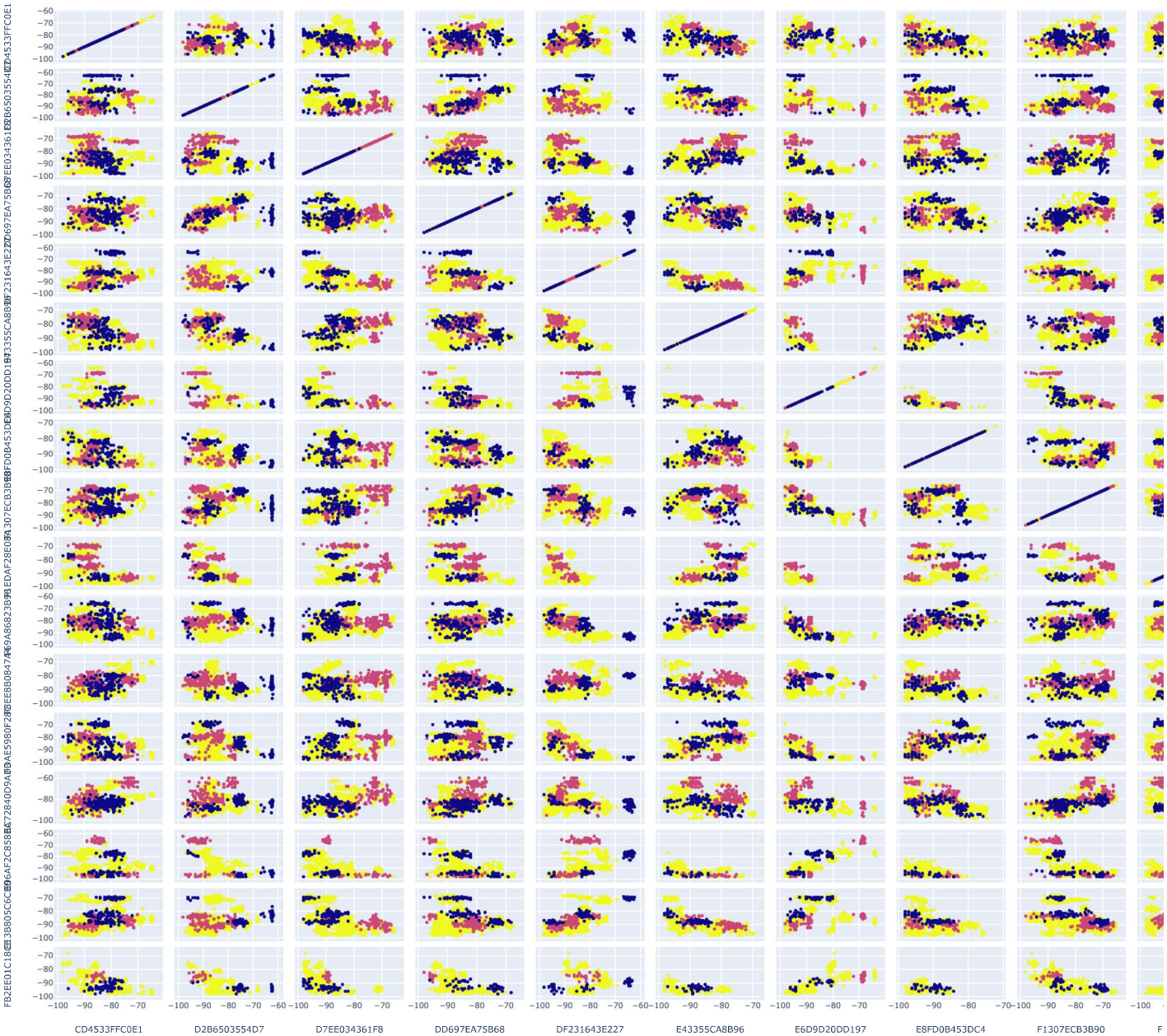


The violin-plots above (Figure 2) show the distributions of the BLE receiver signal strengths for the 3 datasets (train, test and pred). As expected, the shapes are slightly different for each BLE receiver across different datasets because they contain subsets of signals from different BLE beacons, resulting in different distributions.

Scatter matrix to visualize the correlation between BLE receiver data points (Figure 3):

In [15]:
```
index_vals = x_comb_df['dataset'].astype('category').cat.codes
dimensions = []
for i in col_list[:-1]:
    dimensions.append(dict(label=i,values=x_comb_df[i]))

fig_scm = go.Figure(data=go.Splom(
                dimensions=dimensions,
                text = x_comb_df['dataset'],
                marker=dict(color=index_vals,line_color='white', line_width=0.1,size=3),
                ))
fig_scm.update_layout(font=dict(size=7),width=2048,height=1080,showlegend=True)
fig_scm.show()
```

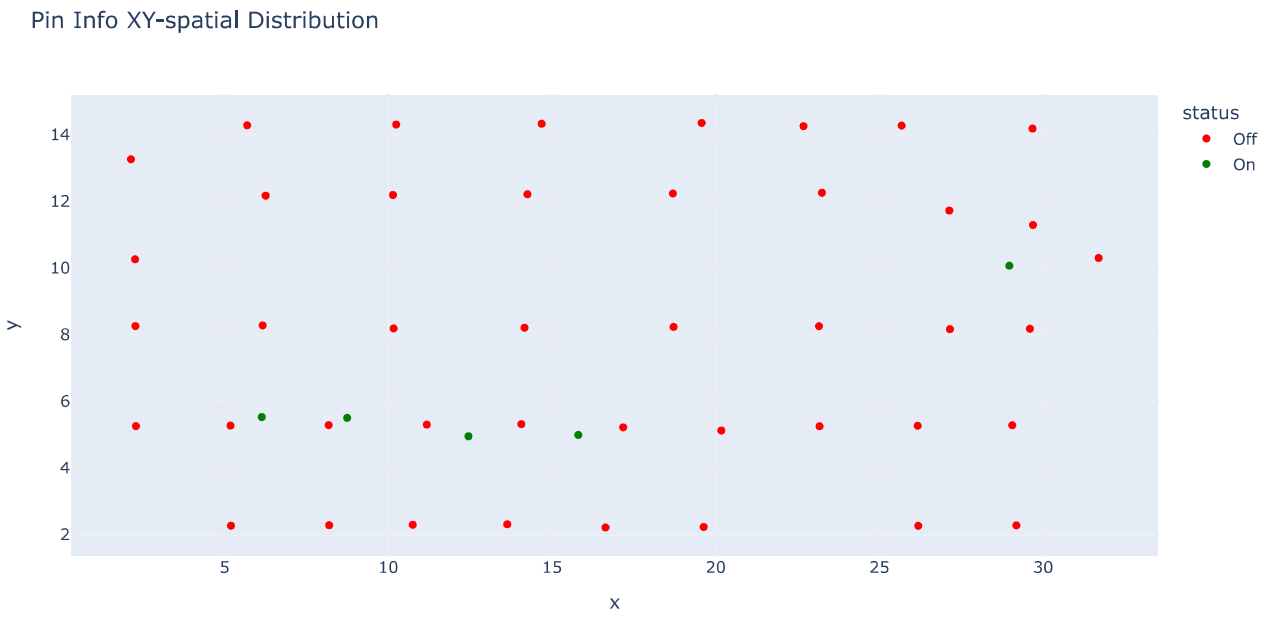


The scatter matrix (Figure 3) examines the correlation between the signals of each BLE receiver against every other. The yellow, red and blue dots correspond to train, test and pred datasets, respectively. The diagonal plots are straight lines as expected, because correlation between the signals of a BLE receiver and itself is 1.0. The other plots show varying degrees of correlation, with none in particular having very high correlation.

Visualizing the pin info xy-spatial distribution (Figure 4):

In [16]:
```
x_comb_df = x_comb_df.drop(columns=['dataset'])
dataset_label_df = pd.DataFrame(['1'] * len_x_train_df + ['2'] * len_x_test_df + ['3'] * len_x_pred_df,columns=['dataset']) #Labels: 1:Train, 2:Test, 3:Pred
x_comb_df = x_comb_df.reset_index(drop=True)
x_comb_df = pd.concat([x_comb_df,dataset_label_df],axis=1)

fig_y = px.scatter(comb_pininfo_df,x='x',y='y',color='status',color_discrete_sequence=['red','green'],title='Pin Info XY-spatial Distribution')
fig_y.show()
```



The above scatter plot (Figure 4), visualizes the pins XY-coordinates spatially. They look uniformly distributed across the room. Green dots represent online pins, and red dots represent offline pins.

Section 3: Handling Missing Data

There is a significantly large number of NaNs in the dataset, especially in x_pred. The percentage of NaNs by BLE receivers is as shown by the previous pie chart.

In [17]:
```
def count_na_pct(df):
    df_na = df.isna().sum().sum()
    df_size = df.shape[0] * df.shape[1]
    df_na_pct = 100 * df_na / df_size
    return df_na_pct

x_train_na_pct = count_na_pct(x_train_df)
x_test_na_pct = count_na_pct(x_test_df)
x_pred_na_pct = count_na_pct(x_pred_df)
print("%.2f %.2f %.2f" %(x_train_na_pct,x_test_na_pct,x_pred_na_pct))
```

```
22.69 32.42 39.03
```

The percentage of NaNs is 22.69%, 32.42% and 39.03% respectively for the train, test and pred datasets.

A data imputation technique is required. It's possible to substitute NaNs with the following options:

1. Zeros
2. Column Mean
3. Column Mode
4. k-Nearest Neighbors

k-Nearest Neighbors is arguably the technique with the highest accuracy among the four. It is explored below:

In [18]:
```
imputer = KNNImputer(n_neighbors=750)
imputer.fit(x_comb_df)
x_comb = imputer.transform(x_comb_df)
x_comb_df = pd.DataFrame(x_comb,columns=col_list)
x_comb_df
```

Out[18]:

	CD4533FFC0E1	D2B6503554D7	D7EE034361F8	DD697EA75B68	DF231643E227	E43355CA8B96	E6D9D20DD197	E8FD0B453DC4	F1307ECB3B90	F1EDAF28E08A	F69A86
0	-83.852941	-87.000000	-66.034483	-86.791667	-89.041667	-76.217391	-96.333333	-85.909091	-69.342105	-84.575000	-79.
1	-84.800000	-87.600000	-66.136364	-85.812500	-88.411765	-76.720930	-95.000000	-88.416667	-69.117647	-84.978261	-78.
2	-84.000000	-87.454545	-66.531250	-86.142857	-90.000000	-76.766667	-95.000000	-86.333333	-69.291667	-85.081081	-78.
3	-87.450000	-89.428571	-65.818182	-85.769231	-89.333333	-76.388889	-96.000000	-88.384615	-69.850000	-84.520000	-77.
4	-84.757576	-88.333333	-65.636364	-86.214286	-89.444444	-76.818182	-88.878095	-87.666667	-68.551724	-86.306122	-78.
...	...	...	...	...	...	...	...	...	...	...	
2149	-82.000000	-94.000000	-82.000000	-88.500000	-88.644336	-81.703496	-84.924271	-80.333333	-81.184973	-93.400000	-65.
2150	-91.000000	-86.000000	-80.666667	-82.000000	-89.016999	-79.000000	-85.356551	-81.500000	-78.000000	-95.000000	-65.
2151	-84.500000	-84.839656	-84.000000	-84.772194	-88.337306	-78.800000	-86.111336	-81.750000	-89.500000	-94.000000	-65.
2152	-86.817077	-88.333333	-82.428571	-86.333333	-88.867677	-85.000000	-86.343423	-82.000000	-79.000000	-92.333333	-65.
2153	-85.000000	-85.188707	-85.435825	-86.000000	-88.578886	-81.333333	-85.120449	-83.000000	-87.000000	-93.200000	-65.

2154 rows × 18 columns

It can be verified that the NaNs have been imputed and the NaNs have disappeared, as seen in the table above.

Splitting the imputed datasets back into the original train (dataset=1),test (dataset=2) and pred (dataset=3):

In [19]:
```
x_train_df = x_comb_df[x_comb_df['dataset']==1]
x_test_df = x_comb_df[x_comb_df['dataset']==2]
x_pred_df = x_comb_df[x_comb_df['dataset']==3]
x_train_test_df = x_comb_df[x_comb_df['dataset']!=3]
```

Section 4: Model training and prediction

Further Preprocessing

At this point the dataset is ready for model training. But first, some minor changes will have to be applied to the dataset. The dataset labels are dropped because it's not required for prediction:

In [20]:
```
x_train_df = x_train_df.drop(columns=['dataset'])
x_test_df = x_test_df.drop(columns=['dataset'])
x_pred_df = x_pred_df.drop(columns=['dataset'])

x_train_test_df = x_train_test_df.drop(columns=['dataset'])
```

Further, merging of the y-dataframes with PinInfo, and dropping the PinId once done:

In [21]:
```
y_train_df.columns = y_test_df.columns = ['pinId']
y_train_df = pd.merge(y_train_df,offline_pininfo_df)
y_test_df = pd.merge(y_test_df,online_pininfo_df)
y_train_df = y_train_df.drop(columns=['pinId'])
y_test_df = y_test_df.drop(columns=['pinId'])
y_train_test_df = pd.concat([y_train_df,y_test_df],axis=0)
y_train_test_df = y_train_test_df.reset_index(drop=True)
```

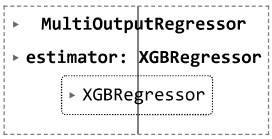
Converting the dataframes into np.array format:

In [22]:
```
x_train_arr = np.array(x_train_df)
x_test_arr = np.array(x_test_df)
x_pred_arr = np.array(x_pred_df)
x_train_test_arr = np.array(x_train_test_df)
y_train_arr = np.array(y_train_df)
y_train_test_arr = np.array(y_train_test_df)
```

Model Training and Fitting

In [23]:
```
estimator = XGBRegressor(n_estimators=150,eta=0.42,max_depth=4)
model = MultiOutputRegressor(estimator = estimator,n_jobs = -1)
model.fit(x_train_arr, y_train_arr)
```

Out[23]:



Checking Model Accuracy

Since there is a X-coordinate and Y-coordinate for every predicted value, evaluation can be performed in severals ways:

- Mean X error
- Mean Y error
- Mean XY error
- R^2 score

Mean XY error is the average of X-coordinate and Y-coordinate errors.

In [24]:
```
def error_calculation(y,yhat):
    summed_x_error = 0.0
    summed_y_error = 0.0
    summed_xy_error = 0.0
    len_y = len(y)
    for i in range(len_y):
        x_diff = abs(y['x'][i] - yhat['x'][i]) #Difference of x-coordinates between y_hat and y
        y_diff = abs(y['y'][i] - yhat['y'][i]) #Difference of y-coordinates between y_hat and y
        summed_x_error += x_diff
        summed_y_error += y_diff
        summed_xy_error += ((x_diff + y_diff) / 2.0) #Getting the average between the two
    mean_x_error = summed_x_error / len_y
    mean_y_error = summed_y_error / len_y
    mean_xy_error = summed_xy_error / len_y
    return mean_x_error,mean_y_error,mean_xy_error

yhat_test_arr = model.predict(x_test_arr)
yhat_test_df = pd.DataFrame(yhat_test_arr,columns=['x','y'])
mean_x_error,mean_y_error,mean_xy_error = error_calculation(y_test_df,yhat_test_df)
print("Error: %.3f,%.3f,%.3f" %(mean_x_error,mean_y_error,mean_xy_error))

print("R2_score: %.3f" %r2_score(y_test_df,yhat_test_df))
```

```
Error: 1.069,1.092,1.080
R2_score: 0.732
```

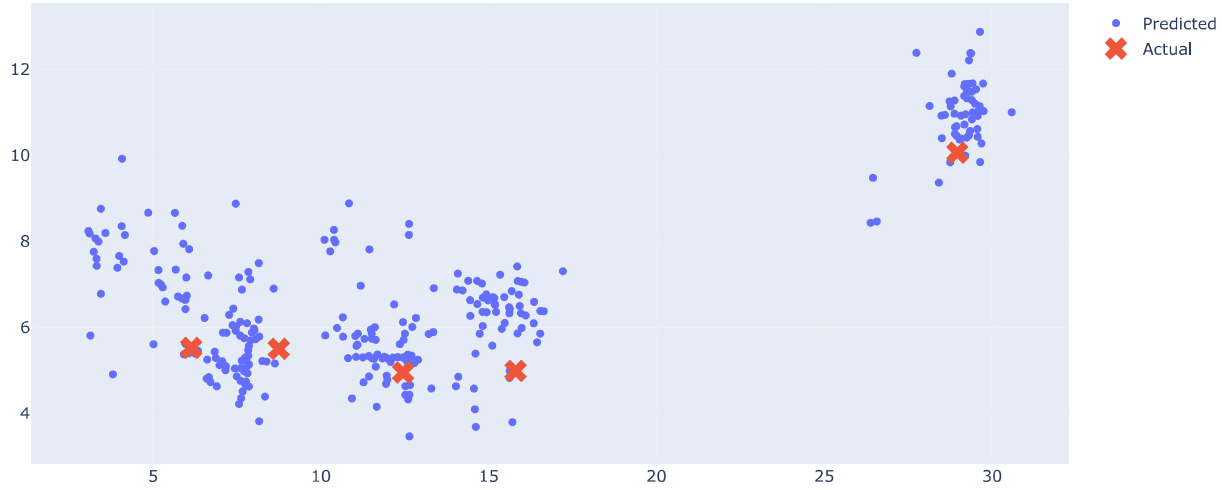
The mean errors and R^2 score for the test dataset is as follows:

- Mean X error: 1.069
- Mean Y error: 1.092
- Mean XY error: 1.080
- R^2 score: 0.732.

The R^2 score is known as the coefficient of determination. It is defined as the statistical measure that represents the goodness of fit of a regression model.

Visualizing the Predicted Pins (Figure 5)

In [25]:
```
#Visualizing the y_test against yhat_test
fig_yhat = go.Figure()
fig_yhat.add_trace(go.Scatter(x=yhat_test_df['x'],y=yhat_test_df['y'],mode='markers',name='Predicted'))
fig_yhat.add_trace(go.Scatter(x=y_test_df['x'],y=y_test_df['y'],mode='markers',marker={'size':15,'symbol':'x'},name='Actual'))
fig_yhat.show()
```



The scatter plot above (Figure 5) shows the predicted pin XY-locations in blue against the actual pin locations in red. The predicted points are clustered around the location of the actual points, showing that the model can predict the location of the pins.

Prediction

Finally, prediction is made using the model for the held out pred dataset, using the combined train and test data:

In [27]:
```
model = MultiOutputRegressor(estimator = estimator,n_jobs = -1).fit(x_train_test_arr, y_train_test_arr)
yhat_pred_arr = model.predict(x_pred_arr)
yhat_pred_df = pd.DataFrame(yhat_pred_arr,columns=['x','y'])
yhat_pred_df.to_csv(output_file,index=False)
yhat_pred_df
```

Out[27]:

	x	y
0	2.264782	7.295297
1	2.474912	7.334783
2	2.836583	8.218587
3	2.495074	8.915661
4	2.914079	5.758885
...	...	...
283	8.294208	10.309168
284	10.543693	12.243903
285	8.853438	11.832061
286	11.699753	12.007940
287	6.989229	10.956576

288 rows × 2 columns

Conclusion

Based on the average errors as observed above, it is proven that there is at least a moderate correlation between the signal strengths and the position of the BLE receivers relative to the BLE beacons. The correlation makes it possible to predict the location of the BLE beacons using a machine learning model trained using the online and offline BLE receiver data.

Limitations

1. There is a significant number of missing data points (NaNs) in the training, test and test submission datasets (22.69%, 32.42% and 39.03% respectively). It is assumed that this affects the model's ability to predict the outcome.
2. There training data set size is quite limited and having more data points would produce a more accurate result.
3. Hyperparameter tuning was performed using the test dataset as a benchmark. It is possible that some degree of overfitting has occured, given that the pred dataset data distribution is dissimilar to that of train and test datasets (as seen in Figures 2 and 3).