Locating Sound with Machine Learning

Brady Zhou, Raymond Zhao

Introduction

With the growing ubiquity of microphones (due in large part to Alexa and Google Home), localizing sound sources is becoming a more relevant issue. The paper [1] discusses two strategies to solve this issue: affine mapping (or linear transformation) and principal component analysis (PCA). PCAs and affine mappings were conducted to determine and map the sound source. In addition to the previously mentioned methods, supervised machine learning was also mentioned (though not executed) as another possible method.

This group proposes to develop a supervised machine learning method mentioned in the paper [1] and also address some of the issues with the previous methods. This is namely the low robustness to missingness in PCAs and the need for good anchor points in affine mapping. What we found was that the SVM and Neural Network models were able to replicate the performance of the previous methods whereas Random Forest could not. And between SVM and Neural Net, the latter proved more generalizable to unseen data.

Problem

A playing speaker is placed on a desk in a room. With a set of microphone arrays placed around the room, its location can be roughly triangulated given the resulting data collected by the arrays, processed using beamforming [1]. In our experiment, our 5 arrays each produce a 3D "vector" of arrival that tells the direction from which the array perceived the signal. From here, the goal becomes to transform this data from $\mathbb{R}^{15}$ to $\mathbb{R}^3$

There are several naive solutions here, but two stand out as particularly robust: PCA and linear mapping. PCA seems a natural fit; downsizing from $15 \rightarrow 3$ with no real interpretative significance on any particular variable. While surprisingly precise, PCA suffers from 2 large issues: PCA can't handle missing values and the arrays' data output can be inconsistent, and the final result is couched in the fitted PCA space, which is extremely difficult to map back to the real world.

Linear (or affine) mappings solve the issues of missing values and abstract spaces but still leave a little to be desired, since accuracy is then limited by the fact that it still only maps linear functions when clearly the underlying dynamics of a room are likely nonlinear. This in addition to the fact that they need to be very well calibrating on anchor points to work some things to be desired, so the aim then is to find models to address the problems above while minimizing tradeoffs in accuracy or practicality.

Setup

In [1]:
```
import warnings
warnings.filterwarnings('ignore')

import sys
sys.path.append('../src')
from Mapping import *
sys.path.append('../src/prediction')
from PCA import *
from nn import *

import itertools
import pickle
import matplotlib
import matplotlib.pyplot as plt
```

In [2]:
```
V5 = pickle.load(open('../data/V5.p','rb'))
cp_list = V5["cp_list"]
active_L_table_slide_DOA = V5["active_L_table_slide_DOA"]
active_L_table_slide_matrix = V5["active_L_table_slide_matrix"]
active_long_table_slide_DOA = V5["active_long_table_slide_DOA"]
active_long_table_slide_matrix = V5["active_long_table_slide_matrix"]
```

In [3]:
```
DOA_LIST = cp_list
ROOM_COORDINATES = ROOM_COORDINATES
TABLE_CP_IND = [0,1,2,3,4,5]
CHAIR_CP_IND = [6,7,8,9,10]
ALL_CP_IND   = [0,1,2,3,4,5,6,7,8,9,10]
L_TABLE_CP_IND = [0,1,2,3]
LONG_TABLE_CP_IND = [4,5]
DATA_IND = [TABLE_CP_IND,CHAIR_CP_IND,ALL_CP_IND]

R_1 = ROOM_COORDINATES[0,:2].T.reshape(-1,1)
D_1 = np.median(DOA_LIST[0], axis=0).reshape(-1,1)

# use cp6 to calculate displacement for long table slide
R_6 = ROOM_COORDINATES[5,:2].T.reshape(-1,1)
D_6 = np.median(DOA_LIST[5], axis=0).reshape(-1,1)

R_LIST = [R_1, R_6]
D_LIST = [D_1, D_6]
```

Plots

Here are example plots of all three networks in action. The following plots show the path that the sound source travels. The path makes a semi-rectangular path as it moves from each corner of the table on the lower portion of the L-shaped table. On the long table, the sound source travels in a relatively straight line. What paths we are expected is a rectangle on the right and a line on the left.

Affine Mapping Plot (the baseline)

The baseline method for mapping sound source is an affine mapping [1]. Explained more deeply in the paper "Audio scene monitoring using redundant ad-hoc microphone array networks", affine mapping is essentially a linear transformation of the values in the DoA matrix into real-world coordinates. The paths generated by these calculations are what we want to represent in our machine learning models.

In [4]:
```
EVENT_DOA = [active_L_table_slide_DOA, active_long_table_slide_DOA]
EVENT_LABEL = ['L Table Slide', 'Long Table slide']
B_MATRIX_NAME = ['Table','Chair','All']
COLOR_LIST = ['r','b','g']
MARKER_LIST = ["$1$","$2$","$3$","$4$","$5$","$6$","$7$","$8$","$9$","$10$","$11$"]

fig = plt.figure(figsize = [16,12])
plt.rcParams['font.size'] = '16'
ax = fig.add_subplot(1,1,1)
rect_side_table = matplotlib.patches.Rectangle((0,1.71), 0.92, (3.54-1.71), alpha = 0.3, color = '0.7')
rect_main_table_1 = matplotlib.patches.Rectangle((2.08,1.81), (4.4-0.2-2.08), (2.57-1.81), alpha = 0.3, color = '0.7')
rect_main_table_2 = matplotlib.patches.Rectangle((3.45,2.58), (4.4-0.2-3.45), (3.54-2.595+0.2), alpha = 0.3, color = '0.7')

for ii in range(len(EVENT_DOA)):
    for jj in range(len(DATA_IND)):
        DOA_points = [DOA_LIST[IND] for IND in DATA_IND[jj]]
        room_coordinates = ROOM_COORDINATES[DATA_IND[jj],:]
        B,R_mean,D_mean,D = generate_linear_transform_matrix(DOA_points, room_coordinates, 2)
        R_0 = R_LIST[ii]-B @ D_LIST[ii]
        r = R_0 +B @ EVENT_DOA[ii].T
        # only plot with label once
        if ii==0:
            ax.scatter(r[0,:], r[1,:], c=COLOR_LIST[jj], s=2)
        else:
            ax.scatter(r[0,:], r[1,:], c=COLOR_LIST[jj], s=2, label=B_MATRIX_NAME[jj])
ax.add_patch(rect_side_table)
ax.add_patch(rect_main_table_1)
ax.add_patch(rect_main_table_2)
ax.set_xlabel("X (m)", fontsize = 21)
ax.set_ylabel("Y (m)", fontsize = 21)
ax.set_aspect('equal')
ax.set(xlim=(0,4.385), ylim=(0,3.918))
ax.set(xlim=(0,4.385), ylim=(1.4,3.65))#ylim=(1.4,3.918))
plt.xticks([0, 1, 2, 3, 4])
plt.yticks([1.5, 2,2.5, 3, 3.5])
# ax.scatter(ROOM_COORDINATES[:,0],ROOM_COORDINATES[:,1], c='k', s=30)
# ax.tick_params(axis='y', labelsize = 21, width = 2, length = 8)
# ax.tick_params(axis='x',labelsize = 21, width = 2, length = 8)

# for kk in range(ROOM_COORDINATES.shape[0]):
#     ax.scatter(ROOM_COORDINATES[kk,0]+0.2, ROOM_COORDINATES[kk,1], marker=MARKER_LIST[kk], s=200, c='k')
# ax.legend(markerscale=5,fontsize=15)
# plt.show()
# fig.savefig('Mappingtables.pdf', bbox_inches='tight', pad_inches=0)
```

Out[4]:
```
([<matplotlib.axis.YTick at 0x2d0bb8f4688>,
  <matplotlib.axis.YTick at 0x2d0bb818848>,
  <matplotlib.axis.YTick at 0x2d0bb8a7dc8>,
  <matplotlib.axis.YTick at 0x2d0bcec7e08>,
  <matplotlib.axis.YTick at 0x2d0bceccb88>],
 [Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, '')])
```

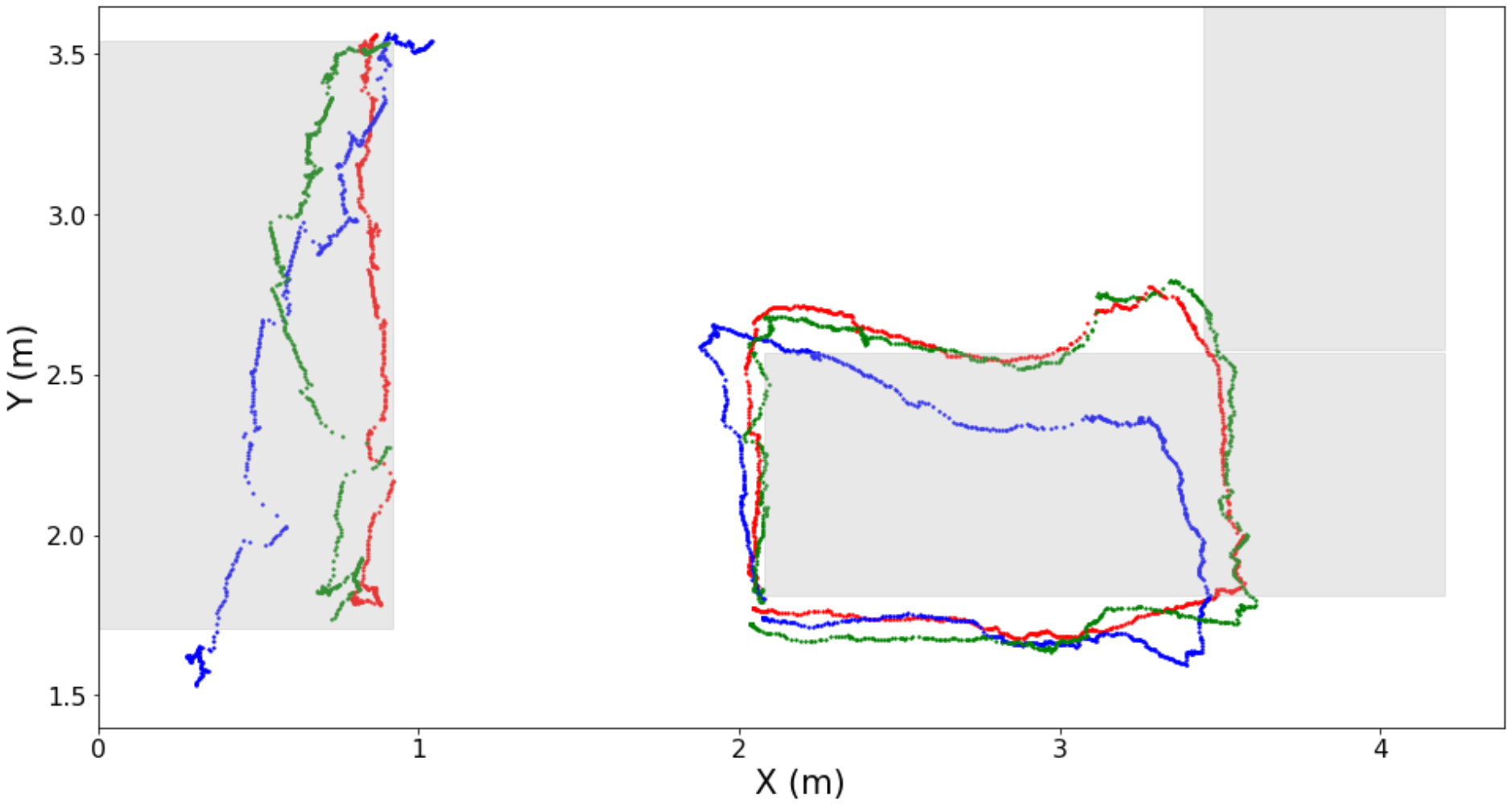


Figure 1: Blue is with the chair training points, red is the table, and green is with all

Feedforward Neural Network

Neural Networks work best by processing large datasets in ways similar to a human mind, but in ways the brain does not work. In the case of localizing sound source, this comes with processing data inputs of 15x3 shaped sound arrays. The gist of the neural network structure is a list of inputs is multiplied by a list of weights (which are determined after training) and then goes through a net input function to aggregate the points. Finally, the remaining data goes through an activation function which finally returns the output. This feedforward neural network is the classic neural network architecture. The input shape is 15x3 while the output (in this use-case) is a 2-D coordinate. The hidden size is 20 and the number of epochs is 2.

In [5]:
```
cp_torch = [torch.from_numpy(cp) for cp in cp_list[:4]]
room_coords = [torch.from_numpy(np.array([i[0], i[1]])) for i in ROOM_COORDINATES[:4]]
X = cp_torch
y = room_coords
```

In [6]:
```
model = NeuralNet(input_size, hidden_size, output_size)
model = model.float()

model.train(X, y)
predictions = model.predict(active_L_table_slide_DOA)

l_predictions = model.predict(active_long_table_slide_DOA)
maps_train = model.predict(itertools.chain(cp_list[0], cp_list[1], cp_list[2], cp_list[3]))
```

In [7]:
```
fig = plt.figure(figsize = [16,12])
plt.rcParams['font.size'] = '16'
ax = fig.add_subplot(1,1,1)

rect_side_table = matplotlib.patches.Rectangle((0,1.71), 0.92, (3.54-1.71), alpha = 0.3, color = '0.7')
rect_side_table = matplotlib.patches.Rectangle((0,1.71), 0.92, (3.54-1.71), alpha = 0.3, color = '0.7')
rect_main_table_1 = matplotlib.patches.Rectangle((2.08,1.81), (4.4-0.2-2.08), (2.57-1.81), alpha = 0.3, color = '0.7')
rect_main_table_2 = matplotlib.patches.Rectangle((3.45,2.58), (4.4-0.2-3.45), (3.54-2.595+0.2), alpha = 0.3, color = '0.7')

# plot the path from the model
mapX = [x[0] for x in predictions]
mapy = [x[1] for x in predictions]
mapX_L = [x[0] for x in l_predictions]
mapy_L = [x[1] for x in l_predictions]
mapX_train = [x[0] for x in maps_train]
mapy_train = [x[1] for x in maps_train]

# Scatter the paths
ax.scatter(mapX_L, mapy_L, s=2)
ax.scatter(mapX, mapy, s=2)
ax.scatter(mapX_train, mapy_train, s=2)

ax.add_patch(rect_side_table)
ax.add_patch(rect_main_table_1)
ax.add_patch(rect_main_table_2)
ax.set_xlabel("X (m)", fontsize = 21)
ax.set_ylabel("Y (m)", fontsize = 21)
ax.set_aspect('equal')
ax.set(xlim=(0,4.385), ylim=(0,3.918))
ax.set(xlim=(0,4.385), ylim=(1.4,3.65))#ylim=(1.4,3.918))
```

```
plt.xticks([0, 1, 2, 3, 4])
plt.yticks([1.5, 2,2.5, 3, 3.5])
```

Out[7]:
```
([<matplotlib.axis.YTick at 0x2d0bcfa9dc8>,
  <matplotlib.axis.YTick at 0x2d0bcf69e08>,
  <matplotlib.axis.YTick at 0x2d0bcf68e08>,
  <matplotlib.axis.YTick at 0x2d0bcff1208>,
  <matplotlib.axis.YTick at 0x2d0bcff4788>],
 [Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, '')])
```

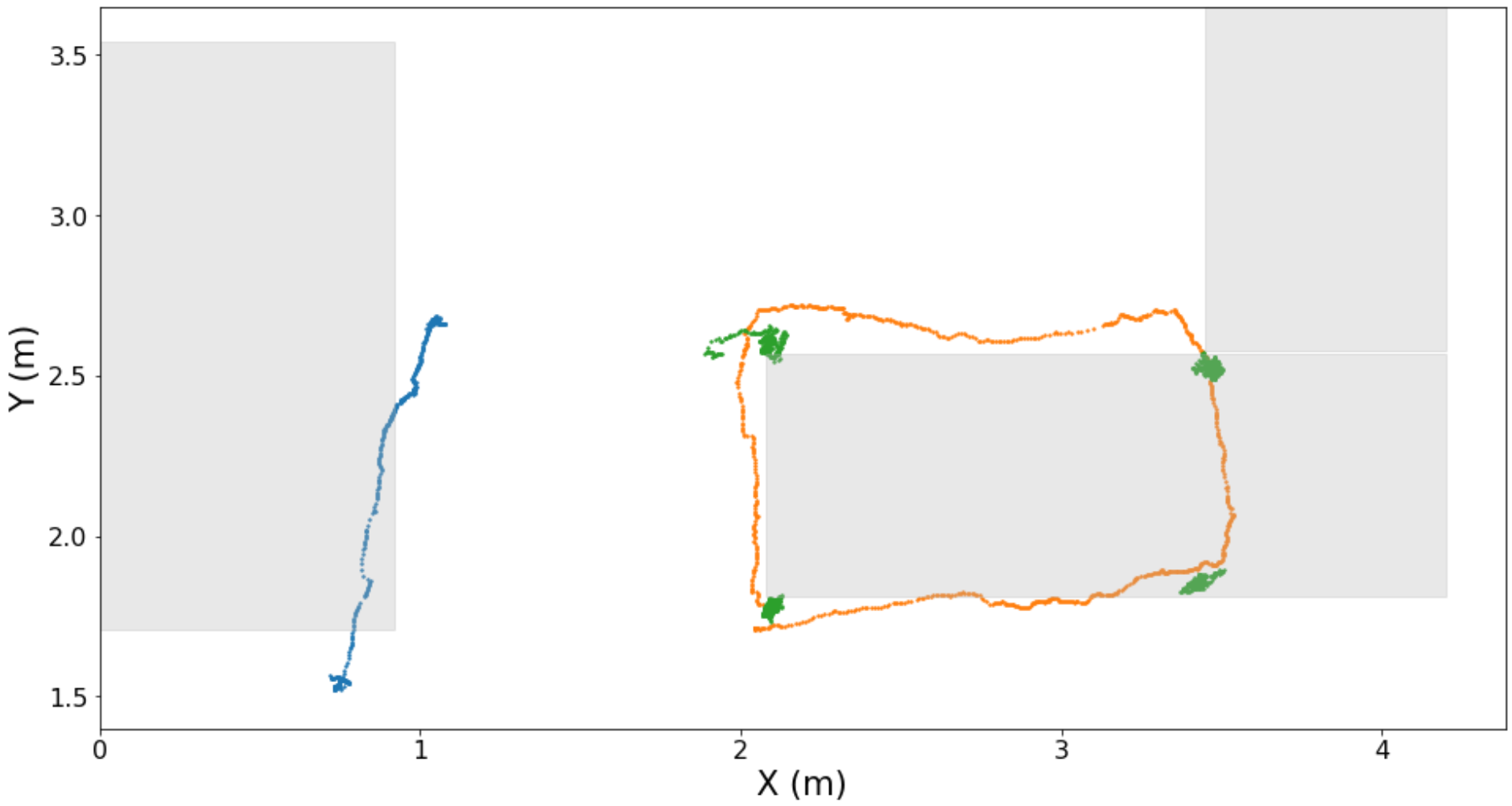


Figure 2: Blue is the long table, and yellow is the yellow table points. This is trained on just the table training data

Support Vector Machine

SVM is a supervised learning algorithm that creates "decision boundaries" with regard to the values of the input data. While an SVM is known for being a classification algorithm, it is also capable of regression and can work with this sort of problem. We can see SVM works well at finding the general path of the sound; this can be attributed to SVM's resilience with small training datasets. However, the model does not perform as well at finding sound that is far from the training points: the long-table sound data is very far off where it should be.

In [8]:
```
control_points = cp_list[:4]
coordinates = [np.array([i[0], i[1]]) for i in ROOM_COORDINATES[:4]]
X = np.vstack([c for c in cp_list[:4]])
y = np.vstack([np.full([p.shape[0], len(c)], c) for p, c in zip(control_points, coordinates)])

from sklearn.multioutput import MultiOutputRegressor
from sklearn.svm import SVR

regr = MultiOutputRegressor(SVR(kernel='rbf', C=1e3, gamma=0.1))
regr.fit(X, y)
```

Out[8]:
```
MultiOutputRegressor(estimator=SVR(C=1000.0, gamma=0.1))
```

In [9]:
```
# plot the svm
fig = plt.figure(figsize = [16,12])
plt.rcParams['font.size'] = '16'
ax = fig.add_subplot(1,1,1)

rect_side_table = matplotlib.patches.Rectangle((0,1.71), 0.92, (3.54-1.71), alpha = 0.3, color = '0.7')
rect_side_table = matplotlib.patches.Rectangle((0,1.71), 0.92, (3.54-1.71), alpha = 0.3, color = '0.7')
rect_main_table_1 = matplotlib.patches.Rectangle((2.08,1.81), (4.4-0.2-2.08), (2.57-1.81), alpha = 0.3, color = '0.7')
rect_main_table_2 = matplotlib.patches.Rectangle((3.45,2.58), (4.4-0.2-3.45), (3.54-2.595+0.2), alpha = 0.3, color = '0.7')


ax.scatter(*regr.predict(active_long_table_slide_DOA).T)
ax.scatter(*regr.predict(active_L_table_slide_DOA).T)
ax.scatter(*regr.predict(X).T)

ax.add_patch(rect_side_table)
ax.add_patch(rect_main_table_1)
ax.add_patch(rect_main_table_2)
ax.set_xlabel("X (m)", fontsize = 21)
ax.set_ylabel("Y (m)", fontsize = 21)
ax.set_aspect('equal')
ax.set(xlim=(0,4.385), ylim=(0,3.918))
```

```
ax.set(xlim=(0,4.385), ylim=(1.4,3.65))#ylim=(1.4,3.918))
plt.xticks([0, 1, 2, 3, 4])
plt.yticks([1.5, 2,2.5, 3, 3.5])
```

Out[9]:
```
([<matplotlib.axis.YTick at 0x2d0c58ff448>,
  <matplotlib.axis.YTick at 0x2d0c58fb208>,
  <matplotlib.axis.YTick at 0x2d0c58e2408>,
  <matplotlib.axis.YTick at 0x2d0c5947288>,
  <matplotlib.axis.YTick at 0x2d0c594a388>],
 [Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, '')])
```

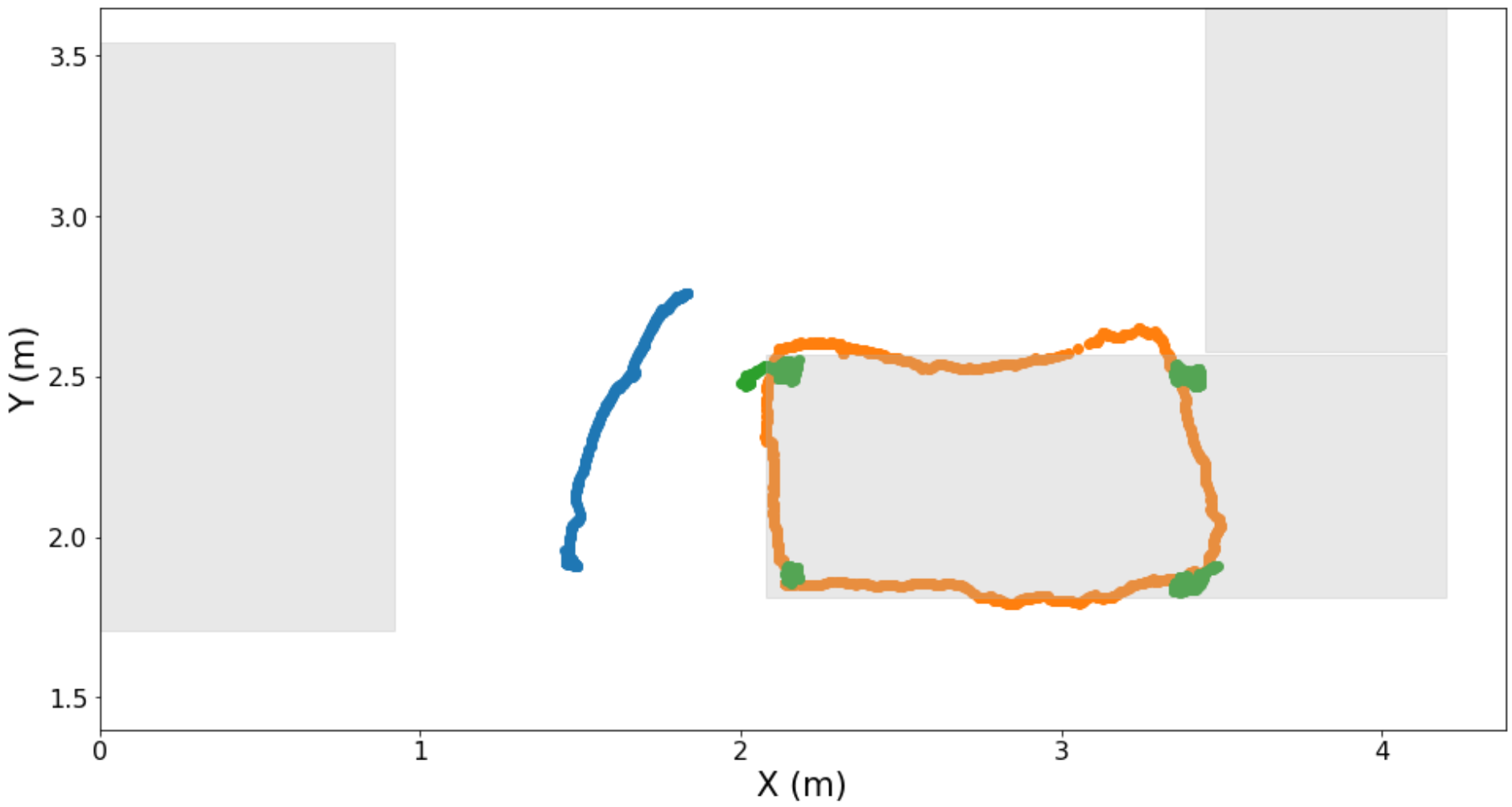


Figure 3: Blue is the long table, and yellow is the yellow table points. We can see SVM performs worse on data it is not trained on.

Linear Regression compared with Random Forest and Decision Tree

Random Forest is another supervised learning algorithm that determines output values through a decision tree. Like decision trees, random forests are an ensemble method through a number of decisions. The difference between a random forest and a decision tree, however, is that random forest allows for significantly more granular decisions. Trees in general, though, have a tendency to overfit to their training datasets and - unlike the previous methods - produce irregular patterns, as we will see later.

In [10]:
```
control_points = cp_list[:4]
coordinates = [np.array([i[0], i[1]]) for i in ROOM_COORDINATES[:4]]
X = np.vstack([c for c in cp_list[:4]])
y = np.vstack([np.full([p.shape[0], len(c)], c) for p, c in zip(control_points, coordinates)])

from sklearn.multioutput import MultiOutputRegressor
from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor

fig, axes = plt.subplots(2, 2)
axes = axes.flatten()
models = {
    'linear regression': LinearRegression,
    'svr': SVR,
    'decision tree': DecisionTreeRegressor,
    'random forest': RandomForestRegressor
}
for m, ax in zip(models.items(), axes):
    print(m)
    model_name, model = m
    regr = MultiOutputRegressor(model())
    regr.fit(X, y)
    ax.scatter(*regr.predict(X).T)
    ax.scatter(*regr.predict(active_L_table_slide_DOA).T, label=model_name)
# fig
```

```
('linear regression', <class 'sklearn.linear_model._base.LinearRegression'>)
('svr', <class 'sklearn.svm._classes.SVR'>)
('decision tree', <class 'sklearn.tree._classes.DecisionTreeRegressor'>)
('random forest', <class 'sklearn.ensemble._forest.RandomForestRegressor'>)
```

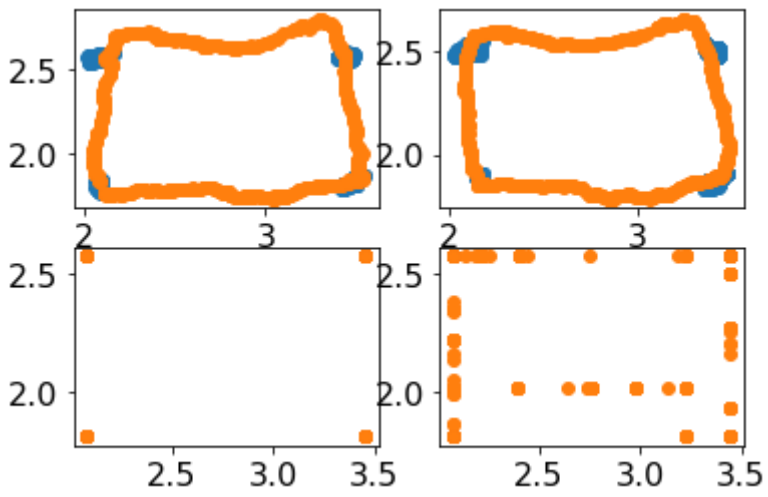


Figure 4: The top plot is the decision tree comparison with linear regression and the bottom plot is the random forest compared with linear regression. Here we can see that both random forest and decision tree - as ensemble methods - share a weakness when the training data is as scant as we have here. Because of this, the model seems unable to find more than 18 points of locations for the sound.

Random Forest Plot on its own

Here we can see both the Random Forest's limitations by the data it is fed. As with our previous models, this random forest regressor was trained on the training points from the toy dataset. Because the training data really only consists of 4 points on the L-shaped table, the model has trouble with sound data originating from elsewhere in the room. Here the model is also plotting data from the long table which is labeled with green, and we can clearly see it incorrectly places the sound on the L-shaped table.

In [11]:
```
fig = plt.figure(figsize = [16,12])
plt.rcParams['font.size'] = '16'
ax = fig.add_subplot(1,1,1)

rect_side_table = matplotlib.patches.Rectangle((0,1.71), 0.92, (3.54-1.71), alpha = 0.3, color = '0.7')
rect_side_table = matplotlib.patches.Rectangle((0,1.71), 0.92, (3.54-1.71), alpha = 0.3, color = '0.7')
rect_main_table_1 = matplotlib.patches.Rectangle((2.08,1.81), (4.4-0.2-2.08), (2.57-1.81), alpha = 0.3, color = '0.7')
rect_main_table_2 = matplotlib.patches.Rectangle((3.45,2.58), (4.4-0.2-3.45), (3.54-2.595+0.2), alpha = 0.3, color = '0.7')

plt.scatter(*regr.predict(active_long_table_slide_DOA).T)
plt.scatter(*regr.predict(active_L_table_slide_DOA).T)
plt.scatter(*regr.predict(X).T)

ax.add_patch(rect_side_table)
ax.add_patch(rect_main_table_1)
ax.add_patch(rect_main_table_2)
ax.set_xlabel("X (m)", fontsize = 21)
ax.set_ylabel("Y (m)", fontsize = 21)
ax.set_aspect('equal')
ax.set(xlim=(0,4.385), ylim=(0,3.918))
ax.set(xlim=(0,4.385), ylim=(1.4,3.65))#ylim=(1.4,3.918))
plt.xticks([0, 1, 2, 3, 4])
plt.yticks([1.5, 2,2.5, 3, 3.5])
```

Out[11]:
```
([<matplotlib.axis.YTick at 0x2d0c5fe6688>,
  <matplotlib.axis.YTick at 0x2d0c5fd78c8>,
  <matplotlib.axis.YTick at 0x2d0c5fd0b48>,
  <matplotlib.axis.YTick at 0x2d0c6026888>,
  <matplotlib.axis.YTick at 0x2d0c6030a08>],
 [Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, '')])
```

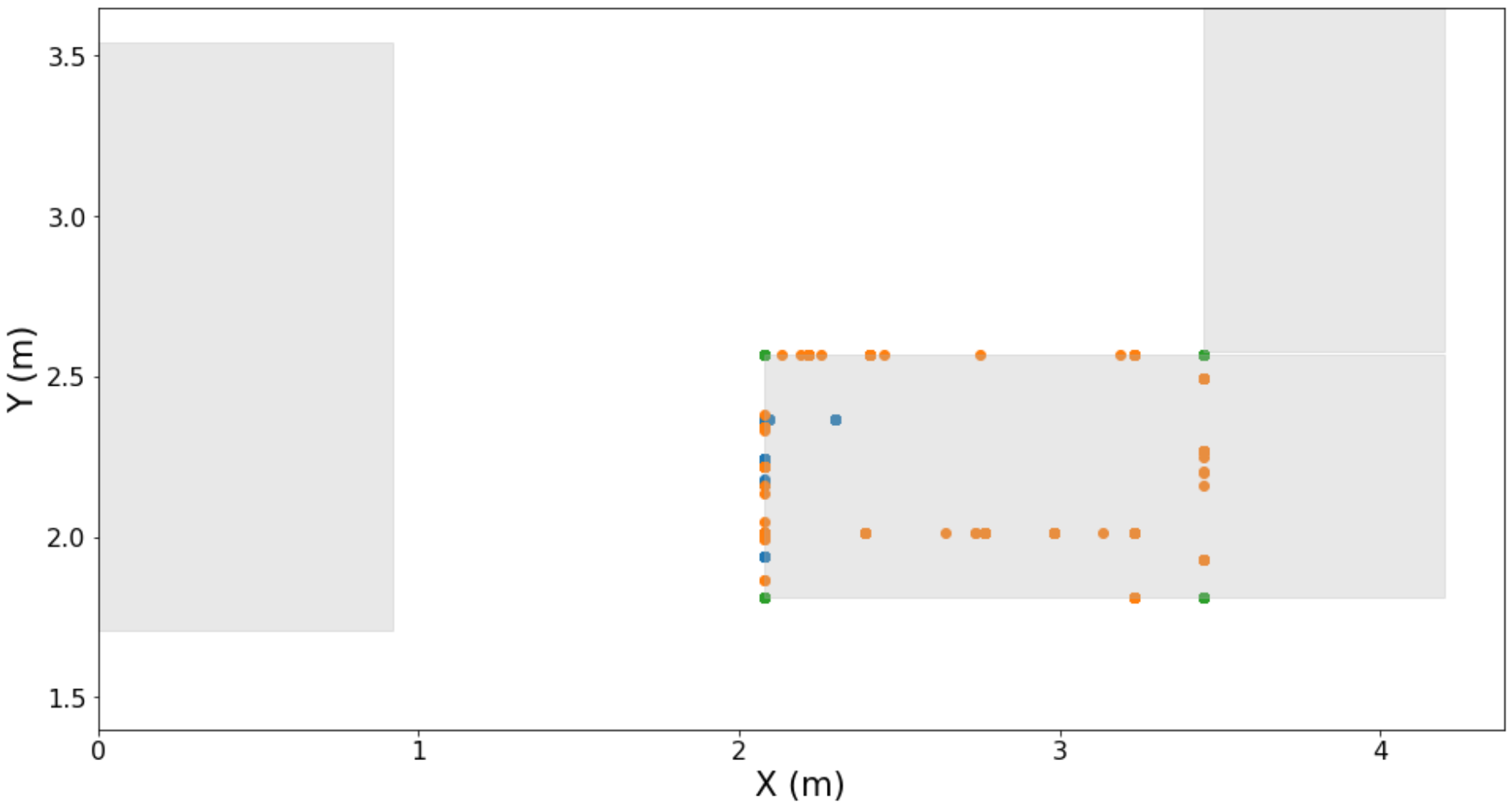


Figure 5: Random Forest is only able to map out some specifc points. And unable to predict long table data.

Data Collection

As shown, many of our predictions struggled with drawing straight lines with the continuous data. We decided we wanted to see how the models would perform on a different dataset. In our experiment we recorded from a different room and used 12 points on a singular table instead of 8 across table and chair points. We also used 3 microphone arrays as two of the arrays failed during recording. We opted to look into how using fewer arrays would affect the models in not just the new data, but also the previously collected data.

Model with shape of 9

Because the format of our data was slightly different, we decided to re-run the model with a shape of `9x3` instead of `15x3` previously used. This reflect the 3 pis used to record instead of the 5 in the office room. Each pi record three values x, y, and z which are the direction values to figure out the location of sound. This is why our model from here on out have an input shape of `9` .

In [12]:
```
import pandas as pd
```

In [13]:
```
cp_office_9 = [pd.DataFrame(cp_list[:4][i]).iloc[:,6:].to_numpy() for i in range(len(cp_list[:4]))]
office_L_9 = [active_L_table_slide_DOA[i][6:] for i in range(len(active_L_table_slide_DOA))]
office_long_9 = [active_long_table_slide_DOA[i][6:] for i in range(len(active_long_table_slide_DOA))]

cp_office_9_torch = [torch.from_numpy(cp) for cp in cp_office_9]
```

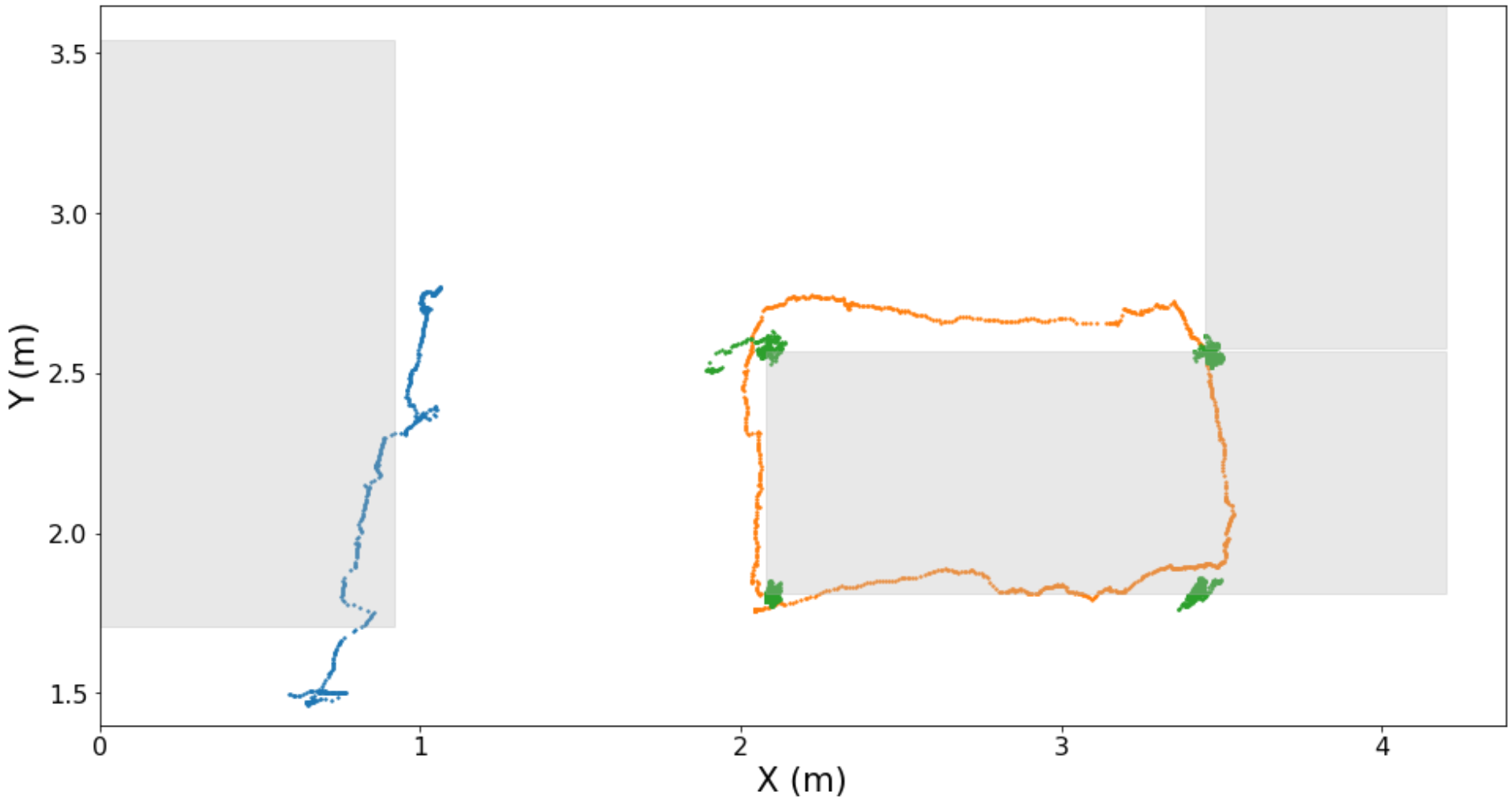
In [14]:
```
model_9 = NeuralNet(input_size=9, hidden_size=20, output_size=output_size)
model_9 = model_9.float()

model_9.train(cp_office_9_torch, room_coords)

predictions = model_9.predict(office_L_9)
l_predictions = model_9.predict(office_long_9)
maps_train = model_9.predict(itertools.chain(cp_office_9[0], cp_office_9[1], cp_office_9[2], cp_office_9[3]))
# for i in list(itertools.chain(cp_office_9[0], cp_office_9[1], cp_office_9[2], cp_office_9[3])):
#     testI = torch.from_numpy(i)
#     prediction = model_9(testI.float()).tolist()
#     maps_train.append(prediction)
```

In [15]:
```
fig = plt.figure(figsize = [16,12])
plt.rcParams['font.size'] = '16'
ax = fig.add_subplot(1,1,1)

rect_side_table = matplotlib.patches.Rectangle((0,1.71), 0.92, (3.54-1.71), alpha = 0.3, color = '0.7')
rect_side_table = matplotlib.patches.Rectangle((0,1.71), 0.92, (3.54-1.71), alpha = 0.3, color = '0.7')
rect_main_table_1 = matplotlib.patches.Rectangle((2.08,1.81), (4.4-0.2-2.08), (2.57-1.81), alpha = 0.3, color = '0.7')
rect_main_table_2 = matplotlib.patches.Rectangle((3.45,2.58), (4.4-0.2-3.45), (3.54-2.595+0.2), alpha = 0.3, color = '0.7')

# plot the path from the model
mapX = [x[0] for x in predictions]
mapy = [x[1] for x in predictions]
mapX_L = [x[0] for x in l_predictions]
mapy_L = [x[1] for x in l_predictions]
mapX_train = [x[0] for x in maps_train]
mapy_train = [x[1] for x in maps_train]
```

```
# Scatter the paths
ax.scatter(mapX_L, mapy_L, s=2)
ax.scatter(mapX, mapy, s=2)
ax.scatter(mapX_train, mapy_train, s=2)

ax.add_patch(rect_side_table)
ax.add_patch(rect_main_table_1)
ax.add_patch(rect_main_table_2)
ax.set_xlabel("X (m)", fontsize = 21)
ax.set_ylabel("Y (m)", fontsize = 21)
ax.set_aspect('equal')
ax.set(xlim=(0,4.385), ylim=(0,3.918))
ax.set(xlim=(0,4.385), ylim=(1.4,3.65))#ylim=(1.4,3.918))
plt.xticks([0, 1, 2, 3, 4])
plt.yticks([1.5, 2,2.5, 3, 3.5])
```

Out[15]:

```
([<matplotlib.axis.YTick at 0x2d0b93fc208>,
  <matplotlib.axis.YTick at 0x2d0b93f4608>,
  <matplotlib.axis.YTick at 0x2d0b93e8888>,
  <matplotlib.axis.YTick at 0x2d0bcf12b88>,
  <matplotlib.axis.YTick at 0x2d0bcef5ac8>],
 [Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, '')])
```



Model Evaluation on New Data

In [16]:

```
def inch_to_meter(inch):
    meter = inch / 39.37
    return meter
```

In [17]:

```
sys.path.append('../src/data_processing')
from make_data import *
```

In [18]:

```
new_X, new_y = load_data()
```

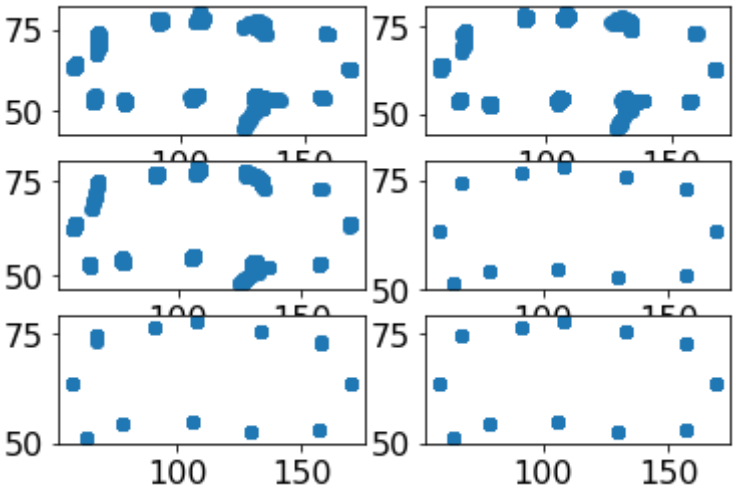
In [19]:

```
new_new_y = pd.DataFrame(new_y).applymap(inch_to_meter).to_numpy()
```

In [20]:

```
from sklearn.multioutput import MultiOutputRegressor
from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import Ridge
from xgboost import XGBRegressor
# %%
V5 = pickle.load(open('../data/V5.p','rb'))
cp_list = V5["cp_list"]
active_L_table_slide_DOA = V5["active_L_table_slide_DOA"][:,6:]
active_L_table_slide_matrix = V5["active_L_table_slide_matrix"]
active_long_table_slide_DOA = V5["active_long_table_slide_DOA"][:,6:]
active_long_table_slide_matrix = V5["active_long_table_slide_matrix"]
```

```
X, y = load_data(path='../data/doa_proj2_allData.p')
# %%
fig, axes = plt.subplots(3, 2)
axes = axes.flatten()
# %%
models = {'linear regression': LinearRegression, 'ridge': Ridge, 'svr': SVR, 'decision tree': DecisionTreeRegressor, 'random fores
for m, ax1 in zip(models.items(), axes):
        regr = MultiOutputRegressor(m[1]())
        regr.fit(X, y)
        ax1.scatter(*regr.predict(X).T)
#       ax1.scatter(*regr.predict(active_L_table_slide_DOA).T, label=m[0])
# fig
# %%
```



Neural Network on New Data

In [21]:
```
doa2 = pickle.load(open('../data/doa_proj2_allData.p','rb'))

cp4 = doa2['cp4']['source0'].dropna(axis=1, how='all').iloc[600:1200, 1:].to_numpy()
cp6 = doa2['cp6']['source0'].dropna(axis=1, how='all').iloc[600:1200, 1:].to_numpy()
cp10 = doa2['cp10']['source0'].dropna(axis=1, how='all').iloc[600:1200, 1:].to_numpy()
cp12 = doa2['cp12']['source0'].dropna(axis=1, how='all').iloc[600:1200, 1:].to_numpy()


y4 = np.array([inch_to_meter(coord) for coord in load_coordinates()['cp4']])
y6 = np.array([inch_to_meter(coord) for coord in load_coordinates()['cp6']])
y10 = np.array([inch_to_meter(coord) for coord in load_coordinates()['cp10']])
y12 = np.array([inch_to_meter(coord) for coord in load_coordinates()['cp12']])
```

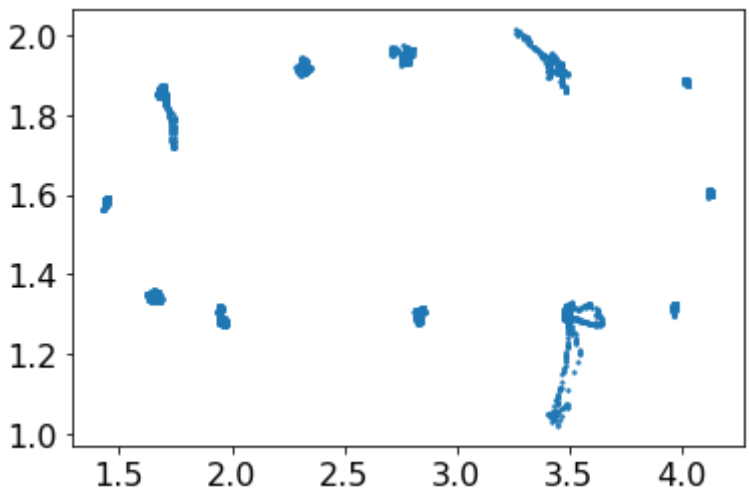
In [22]:
```
newmodel_9 = NeuralNet(input_size=9, hidden_size=20, output_size=output_size)
newmodel_9 = newmodel_9.float()
newcriterion_9 = nn.MSELoss()
newoptimizer_9 = torch.optim.SGD(newmodel_9.parameters(), lr=learning_rate)

newX_9 = [torch.from_numpy(cp4), torch.from_numpy(cp6), torch.from_numpy(cp10), torch.from_numpy(cp12)]
newy_9 = [torch.from_numpy(y4), torch.from_numpy(y6), torch.from_numpy(y10), torch.from_numpy(y12)]
```

In [23]:
```
for i in range(10000):
    for x_i, y_i in zip(newX_9, newy_9):

        outputs = newmodel_9(x_i.float())
        loss = newcriterion_9(outputs, y_i.float())
        newoptimizer_9.zero_grad()
        loss.backward()
        newoptimizer_9.step()
```

In [24]:
```
newmaps9 = []
old_newmaps9 = []
for i in new_X:
    testI = torch.from_numpy(i)
    prediction = newmodel_9(testI.float()).tolist()
    newmaps9.append(prediction)
for i in office_L_9:
    testI = torch.from_numpy(i)
    prediction = newmodel_9(testI.float()).tolist()
    old_newmaps9.append(prediction)
l_old_newmaps9 = newmodel_9.predict(office_long_9)
```

In [25]:
```
newmapX9 = [x[0] for x in newmaps9]
newmapy9 = [x[1] for x in newmaps9]
plt.scatter(newmapX9, newmapy9, s=2)
```

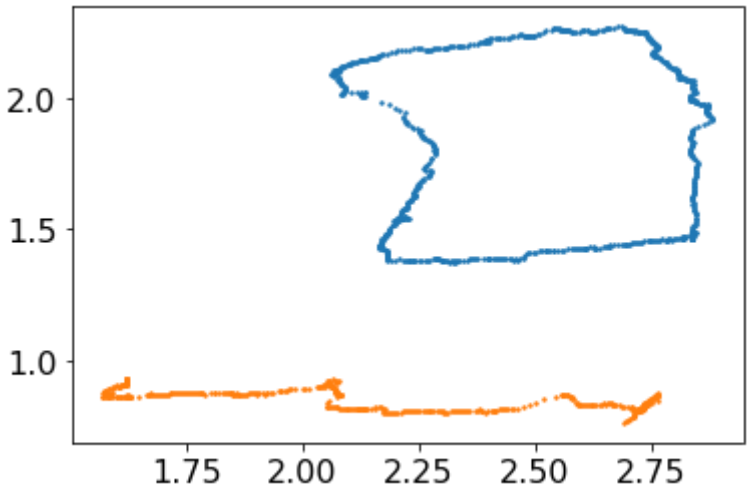
Out[25]:
```
<matplotlib.collections.PathCollection at 0x2d0c9cb7d88>
```



The models run on different rooms' data

In [26]:
```
old_newmapX9 = [x[0] for x in old_newmaps9]
old_newmapy9 = [x[1] for x in old_newmaps9]
old_newmapX9_L = [x[0] for x in l_old_newmaps9]
old_newmapy9_L = [x[1] for x in l_old_newmaps9]
plt.scatter(old_newmapX9, old_newmapy9, s=2)
plt.scatter(old_newmapX9_L, old_newmapy9_L, s=2)
```

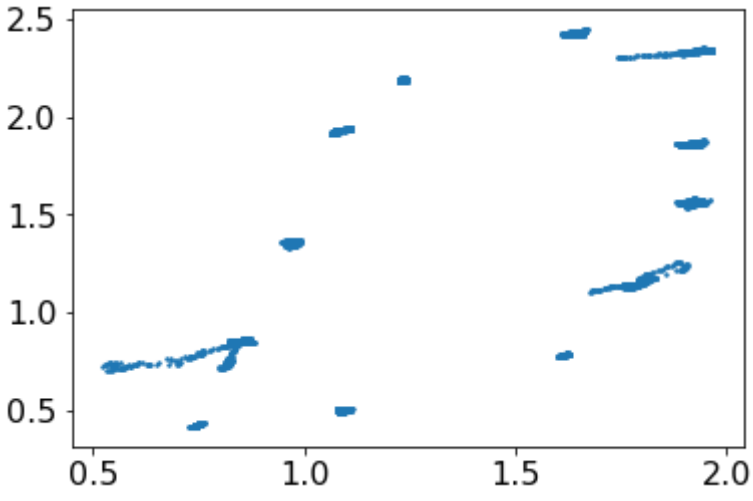
Out[26]:
```
<matplotlib.collections.PathCollection at 0x2d0c9dd3208>
```



In [27]:
```
maps9 = []
for i in new_X:
    testI = torch.from_numpy(i)
    prediction = model_9(testI.float()).tolist()
    maps9.append(prediction)
```

In [28]:
```
mapX9 = [x[0] for x in maps9]
mapy9 = [x[1] for x in maps9]
plt.scatter(mapX9, mapy9, s=2)
```

Out[28]:
```
<matplotlib.collections.PathCollection at 0x2d0c95d4648>
```



Discussion and Conclusion

We conclude that - to varying degrees - machine learning methods work well not only at localizing sound but also are resilient to array dropping. This is joined by the fact that the models can translate directly into real world space unlike the PCA models from the original paper [1] and our proposal [3]. Furthermore, this depends on what kind of sound we are mapping; random forest works well at mapping single source points, while the other two methods are better are mapping out sound paths.

In our proposal we talked about the limitations of the methods presented in the paper [1], namely PCA's dimensional reduction into PCA space and affine mapping's need for well-anchored points. With SVM and a basic neural network we were able to replicate the mappings of both methods while also outputting into real-world space. The two methods differed on performing without well-anchored points: SVM performed very well in well-defined space but not otherwise, meanwhile a basic neural network performed well in both these areas.

Using the new data, we found considerably more noise with the support vector machine and neural network methods compared to the random forest model. For the former two models, three points observed larger amounts of noise (the points nearest the microphone arrays). In the case of needing to map fewer points, having decision points actually works well. The XGBoost model also works well in this case. We believe that in this case, because the output points are few, a trees do not need to find too many unseen points.

However, in terms of generalizable models, this may not be a desirable result. This is why we found – in most cases – that our feedforward neural network was the best performing model.

In [ ]:

References

[1] P. Gerstoft, Y. Hu, M. J. Bianco, C. Patil, A. Alegre, Y. Freund, F. Grondin “Audio scene monitoring using redundant ad-hoc microphone array networks”

[2] M. Hahmann, E. Fernandez-Grande, H. Gunawan, P. Gerstoft “Sound Source Localization in 3D Using Ad-Hoc Distributed Microphone Arrays”

[3] B. Zhou, R. Zhao, L. Meng "Interpreting Microphone Arrays with Machine Learning Methods"

In [ ]: