Indoor navigation with machine learning

Hide & Seek

Article from Issue 255/2022

Author(s): Roland Pleger

We explore some machine learning techniques with a simple missing person app.

GPS lets you determine the position of an object down to single-centimeter accuracy – as long as the object is outside. If the object is inside, the task is a bit more complicated. Satellite navigation doesn't work well through doors and rooftops, and even if you could replace the satellite signal with equivalent transmissions from locally placed beacons or WLAN access points, the presence of interior walls and furniture muddles up the results of classical analytical techniques such as those used with GPS. What is more, when someone is inside a building, the question is not so much about "What are his coordinates." What you really want to know is "What room is he in?" Such a problem is better addressed through the tools of machine learning.

Of course, creating a complete machine learning solution to find someone in a small house might seem like overkill, but this article is intended as an exercise to show these machine learning tools and techniques in a simple situation – a kind of machine learning "Hello, World" application. One could imagine scenarios where these techniques could find broader utility, such as tracking down an executive in a large office complex or even finding a lost set of car keys.

In this example, Tom, the protagonist, has lost his way. Fortunately, his smartphone shows the signal strength of seven hotspots in his vicinity (Figure 1). Because Tom often gets lost, I have mapped the four rooms as a precaution (the blue crosses in Figure 2), and I have a machine learning dataset I can use to train a program to find Tom.

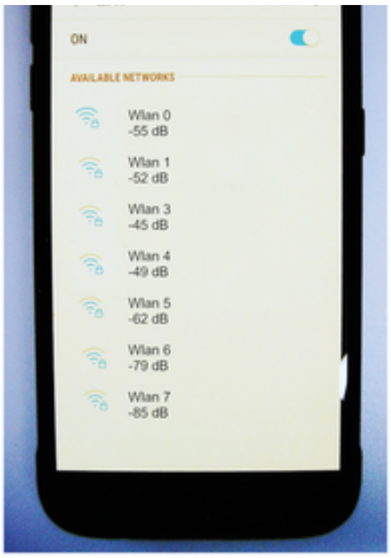


Figure 1: Tom's smartphone shows the signal strength from the WLANs in the individual rooms at his location.

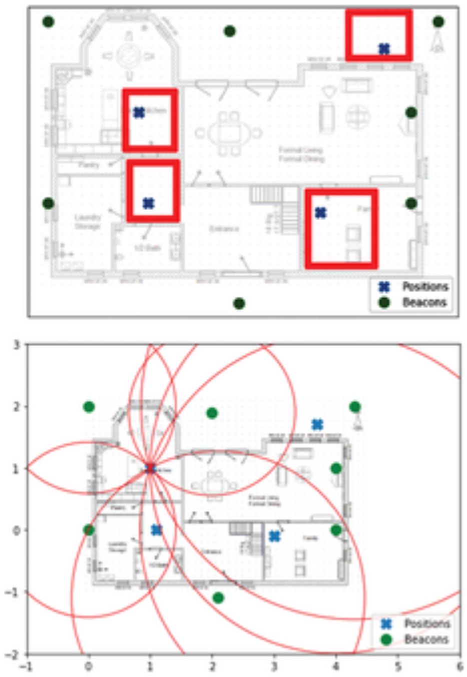


Figure 2: The possible arrangement of beacons (green dots) and positions (blue crosses) in the building where Tom is lost.

With this data, I can use artificial intelligence and supervised learning to locate Tom. Supervised learning techniques are effective, but they require some advance knowledge of the house design. What if I know that Tom is in the house but do not know the number of rooms? In that case, unsupervised learning can help clarify the situation. I will explore unsupervised learning later in this article. Finally, a method known as semi-supervised learning can help me find any errors I made when assigning the rooms.

This article shows how to navigate indoors using supervised and unsupervised machine learning methods based on Python tools. My focus will be on the machine learning technique, with less detail about Python and the libraries used in the solution.

For demonstration purposes, I will be using a dataset provided free of charge from the University California Irvine (UCI) website [1]. The UCI Wireless Indoor Localization dataset consists of eight columns: The signal strengths of the seven WLAN hotspots, measured using a smartphone, and the location of the measurements. After 500 measurements at four fixed locations, 2,000 entries are available.

In my scenario, all coordinates are initially unknown, both those of the possible positions and those of the transmitting beacons. The signal attenuation is measured. Outdoors, you could infer the distance using this measurement. However, indoors, every obstacle falsifies the estimation. Keep in mind, though, that I don't expect to find Tom's exact position in geographical longitude and latitude – I only want to find the right room.

Explorative Data Analysis

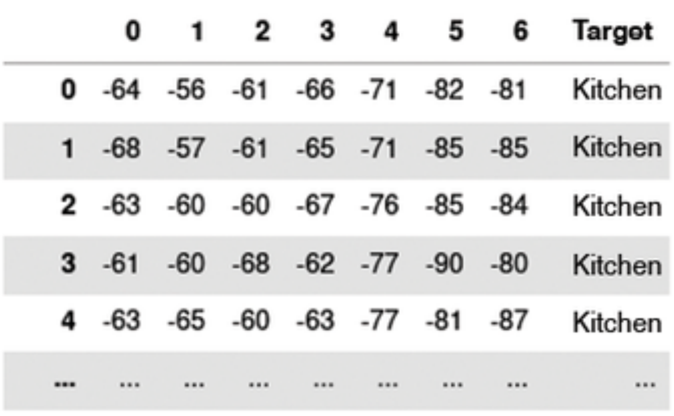
To get started, I need to prep the data (Listing 1). Fortunately, the UCI dataset has been preprocessed: It is balanced and contains no invalid values and no outliers.

Listing 1 Prepping the Data

```
# read data
import numpy as np
import pandas as pd
fn = "https://archive.ics.uci.edu/ml/machine-learning-databases/00422/wifi_localization.
txt"
colnames = [0, 1, 2, 3, 4, 5, 6, 'Target']
df = pd.read_csv(fn, header = None, names=colnames, comment = "#", sep = '\t')
rooms = {1:'Kitchen, 2: 'Hallway', 3:'Livingroom, 4:'Patio}
df['Target'] = df['Target'].map(rooms)
print(df[:2])
df.describe()
```

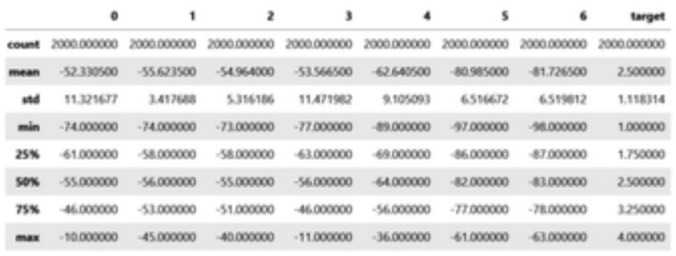
Listing 1 first imports the Python *pandas* data analytics library [2], which supports data handling. The `read_csv()` function reads local files or, as in Listing 1, resources from the Internet. `colnames` adds the column names to `6` to identify the beacons and `Target` to identify the rooms. Using `sep = '\t'` makes the input values tab-delimited.

If you are working with *pandas* for the first time, you may trip up over the index column (the far left column in Figure 3). The index column is generated automatically and does not come from the data, which is why it lacks its own column header in the printed output (Figure 3). The `describe()` function generates the data shown in Figure 4: All columns are fully populated with 2,000 values each. The signal strengths range from -10dB to -98dB. The Target column takes four discrete values, which the dictionary `rooms` replaces with the room names `Kitchen`, `Hallway`, `Livingroom`, and `Patio`.



	0	1	2	3	4	5	6	Target
0	-64	-56	-61	-66	-71	-82	-81	Kitchen
1	-68	-57	-61	-65	-71	-85	-85	Kitchen
2	-63	-60	-60	-67	-76	-85	-84	Kitchen
3	-61	-60	-68	-62	-77	-90	-80	Kitchen
4	-63	-65	-60	-63	-77	-81	-87	Kitchen
...	...	...	...	...	...	...	...	...

Figure 3: The first four lines in the pandas DataFrame.



	0	1	2	3	4	5	6	target
count	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000
mean	-52.330500	-55.623500	-54.964000	-53.566500	-62.640500	-80.985000	-81.726500	2.500000
std	11.321677	3.417688	5.316186	11.471982	9.105093	6.516672	6.519812	1.118314
min	-74.000000	-74.000000	-73.000000	-77.000000	-89.000000	-97.000000	-98.000000	1.000000
25%	-61.000000	-58.000000	-58.000000	-63.000000	-69.000000	-86.000000	-87.000000	1.750000
50%	-55.000000	-56.000000	-55.000000	-56.000000	-64.000000	-82.000000	-83.000000	2.500000
75%	-46.000000	-53.000000	-51.000000	-46.000000	-56.000000	-77.000000	-78.000000	3.250000
max	-10.000000	-45.000000	-40.000000	-11.000000	-36.000000	-61.000000	-63.000000	4.000000

Figure 4: The statistical description of the data.

The details of the quantiles 25 to 75 percent say too little about the data distribution. Instead, I want a graphical representation. Listing 2 initially restricts the dataset to the first room with the query

Listing 2 Plotting Signal Strength Distribution

```
dh = df[df['Target'] == 'Kitchen']
dh = dh.drop('Target' ,axis = 1)
dh.plot.hist(bins=12, alpha=0.5)
dh.plot.kde()
```

```
['Target'] == Kitchen
```

The `drop()` command then deletes the Target column to remove it from the evaluation. The histogram (Figure 5) is subdivided into 12 bins. Even with a transparency of `0.5`, the values overlap.

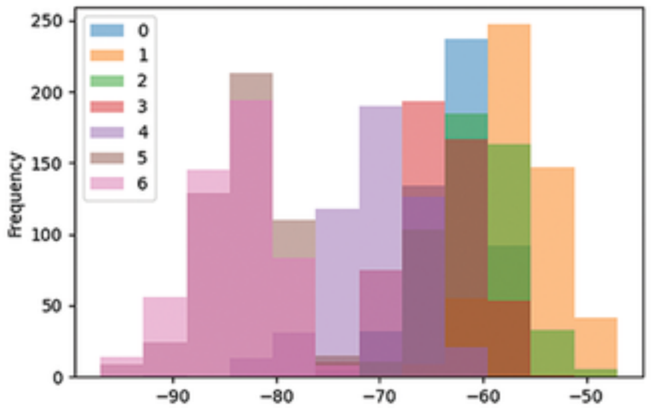


Figure 5: Scatter of the signal strengths of the seven WLAN hotspots in the kitchen.

The representation becomes clearer when the histograms are approximated by a continuous function using kernel density estimation (KDE) [3], a statistical technique used for smoothing probability density functions. The bandwidth parameter controls the smoothing, but adjustment is rarely required. The results in Figure 6 are easier to interpret than the histogram plot in Figure 5.

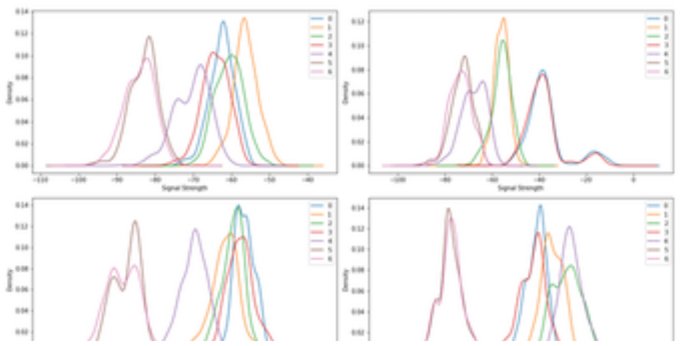


Figure 6: The kernel density estimation for all four rooms.

With the data in Figure 6, it now becomes clear that perhaps not the greatest care was taken with the measurements. Some beacons give partially identical signals and therefore no new information about the location. Some curves deviate from a simple distribution. It is possible that the data was recorded at more than four different positions. The high overlap is also one of the reasons why an analytical approach will not work here.

Supervised Training

In each of the four rooms, the signal strength of the WLAN hotspots has been measured 500 times. Averaging obscures too many details to clearly characterize the rooms. For example, one of the measuring smartphones could basically record weaker levels, and it would always miss the mean value. Machine learning methods come in useful here by putting the entirety of the data into context.

From the wide range of methods for classifying, I will limit myself to one method known as Random Forest.

A Random Forest pits many shallow decision trees against each other and optimizes internal hyperparameters. A decision tree starts at the attributes that have the highest discriminatory power. For example, to separate cherries, plums, and apples, querying by size largely divides the fruit, even if there are large cherries and small plums. Next, querying by roundness of the stone refines the subdivision. As the depth increases, the prediction improves. In the end, there is exactly one query nest for each fruit. The problem: The decision tree learns by rote. This kind of overfitting is a general problem in machine learning.

Boosting algorithms are some of the best classifiers, but an explanation would be beyond the scope of this article. Neural networks do not perform much better on this problem and are difficult to interpret due to the confusing number of parameters.

Machine learning does not perform miracles. If the initial data is inconsistent, the prediction probability is also limited, regardless of the choice of machine learning method. The Python libraries *scikit-learn* and *sklearn*, respectively, support all of these procedures. In many cases, it is sufficient to replace one line of code to classify the data according to a different method. To detect overfitting of a dataset, the data is split. The greater part is used to train the algorithm. Testing is done at the very end with the remaining data. The prediction's deviation from the test values is a measure of the quality of the estimator.

After reading the data as per Listing 1, Listing 3 splits the DataFrame `df` into properties `X` and target size `y`. Conversion to a `numpy` array is optional and otherwise done later by the classifier. The `train_test_split()` function splits the data into training and test data. In itself, this is a simple task, but the routine makes sure that the target variables appear equally in both datasets. Otherwise, the system might learn `Hallway` and `Kitchen` but not be confronted with `Livingroom` until the testing phase.

Listing 3 Preparing the Classifier

```
# split data into features and target
# (change column number)

X = df.iloc[:, 0:-1].to_numpy()
y = df.iloc[:, -1].to_numpy()

# split into training and test data

```

```
09 from sklearn.model_selection import train_test_split
10
11 X_train, X_test, y_train, y_test = train_test_split(X, y)
12
13 # select model and fit
14 # (change row number)
15
16 from sklearn.ensemble import RandomForestClassifier
17
18 classifier = RandomForestClassifier()
19 classifier.fit(X_train,y_train)
```

The `RandomForestClassifier()` command selects the Random Forest classifier. In line 19 of Listing 3, the algorithm silently learns its internal parameters. The computation time increases with the volume of data and parameters. Especially for neural networks, it is faster to train only once and then store the internal parameters (e.g., face recognition in OpenCV or cameras works according to this method). The parameters learned earlier are loaded into memory and represent a fully trained system.

Depending on the method, the internal parameters can be several megabytes. However, small datasets are processed quickly, so I will not elaborate on swapping.

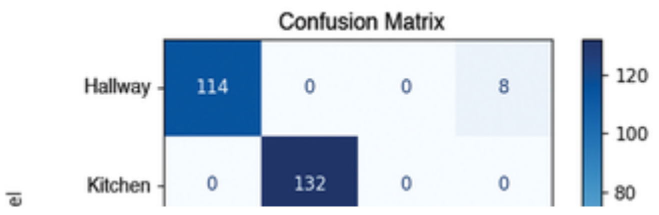
Evaluating the Performance

If no error message appears after calling `fit` in Listing 3, training was successful. However, that alone is not enough. Listing 4 evaluates how well the algorithm classified the data. The `classifier` object contains all the data that has been adjusted during training. The `classifier.predict()` method calculates target values for input sequences. I will use this tool to find Tom later. To determine the quality of the method, I compare the test values `y_test` with `y_pred`, which are the values predicted by `predict()`. The deviations are a measure of the quality of the classifier.

Listing 4 Evaluating the Classifier's Performance

```
# predict and evaluate
from sklearn.metrics import confusion_matrix
y_pred = classifier.predict(X_test)
labels = classifier.classes_
cm = confusion_matrix(y_test, y_pred, labels=labels)
print(np.trace(cm)/y_test.shape[0])
pd.DataFrame(cm, index=labels, columns=labels)
```

The confusion matrix (Figure 7) compares the values. It takes the name of the target values and their orders from the `classes_` attribute. The underscore at the end of the classes attribute follows a common convention of the *scikit-learn* library to mark all values derived from the data this way. In Figure 7, the hallway is correctly located 114 times but incorrectly identified as the living room eight times. Conversely, the living room is mistaken for the hallway twice. I will revisit this problem later.



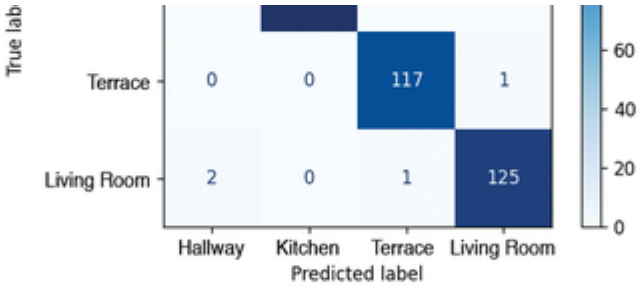


Figure 7: The confusion matrix clarifies the prediction quality.

The confusion matrix values are often aggregated to create a number: Of 500 values, 12 were wrongly assigned, corresponding to an accuracy of *1-12/500 = 0.976*. Would the result be `0.974` if there had been 13 values? Small numbers have one big flaw. The specification of the third decimal place may be mathematically correct in the concrete case, but statistically it is wrong. At best, you could limit the error to *0.97 ± 0.02*.

I have almost found Tom. His smartphone shows the field strengths of the seven WiFi networks in his environment (Figure 1). After training, the decision tree classifier derives the position from this: Tom is in the living room (Listing 5).

Listing 5 Where's Tom?

```
pTom = [-55, -52, -45, -49, -62, -79, -85]
print('Tom is here: ', classifier.predict([pTom]))
print('Error: ', classifier.predict_proba([pTom]))
# output:
# Tom is here:  ['Livingroom']
# Error: array([[0.  , 0.05, 0.18, 0.77]])
```

Not only does the `predict()` function find the room, `predict_proba()` also reveals how assured the algorithm is of its decision: 77 percent in this case. In `RandomForestClassifier`'s default setting, 100 decision trees compete against each other: 77 decide on the living room, and 18 decide on the patio. If you were to increase the number to 1,000 with `n_estimators=1000`, you would get values of 789/1000 and 163/1000. Although this technique still leaves some uncertainty, it is far better than the analytical alternative (see the sidebar entitled "Attempting an Analytical Solution").

Attempting an Analytical Solution

Satellite-based systems such as GPS or Galileo determine the position via multilateration, deriving the distance in a linear way from the signal propagation time.

If you know the position of a satellite (right black dot in Figure 8) and its distance (the gray sphere surrounding the black dot), you can guess your approximate location. The intersection of the gray sphere with the blue sphere (which represents the surface of the Earth) restricts your location to the green circle.

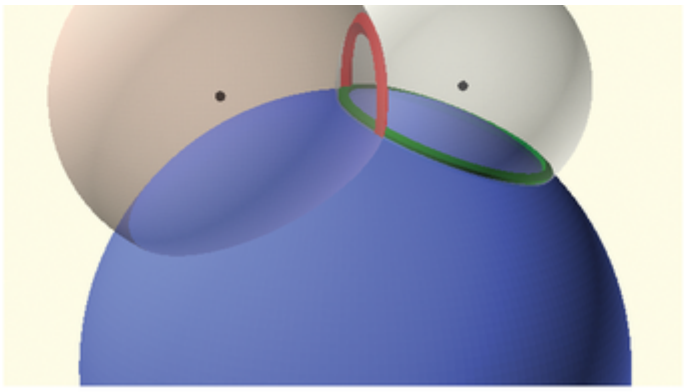


Figure 8: How multilateration works in principle.

A second satellite reduces the possible positions to the intersections between the green and red circles. If a third satellite is added, you are no longer limited to the surface of the blue sphere. You can measure altitude as an intersection with the cap of a third satellite (not shown in Figure 8). A fourth satellite is necessary to synchronize the signal propagation times. Receiving signals from additional satellites improves positioning accuracy. Modern GNSS receivers can process 30 channels and more.

In the scenario involving Tom, I am not measuring transit times but rather the WLAN signal strength, which decreases parallel to the square of the distance (see Listing 7). The transmission power is unknown, but attenuation is a more serious issue. Any obstacle weakens the signal and simulates a – location-dependent – varying distance. In the conversion, instead of a quadratic attenuation, a number larger than *2* is assumed, in this case *4*. Because of these uncertainties, the following considerations are theoretical.

This example relies on signal strengths from seven transmitter beacons. Unlike GPS satellites, these beacon positions are unknown. Instead, their signal strengths are available at four different locations, and these coordinates are also unknown. The altitude is not considered in this example, which means that each point is determined by its *x* and *y* coordinates. Listing 7 calculates the Euclidean distance to a beacon.

There are four unknown positions for the site and seven for the beacons. In addition, I am also trying to estimate the signal strengths of the seven beacons for a total of 29 unknowns. At the same time, I know the signal strengths and – in this abstract consideration – the distances to the seven beacons, for a total of 28 equations. To solve the system uniquely, I need at least one equation for each unknown. An overdetermined system of equations would be even better to compensate for the errors by means of a fit.

I fix a location by placing it at the origin. I further assume that a beacon is located in the *y* direction and that the *x* coordinate takes a value of zero accordingly. Finally, I set the signal strength of the transmitter for hotspot 0 such that the distances between the positions are on the order of meters.

That leaves 25 unknowns and 27 equations. While I will get a result, the result will not be robust because of the small amount of information and the large error in the distance estimate. For comparison, GPS requires signals from only four satellites, with a linear dependence of distance on time and nearly unobstructed views to the satellites.

Figure 9 shows one example of the nonlinear optimization solution, which locates all rooms and beacons. For clarity, the signal strengths converted to distance are plotted as circles for the living room only. The other 21 distance circles for the remaining three rooms are not shown in Figure 9.

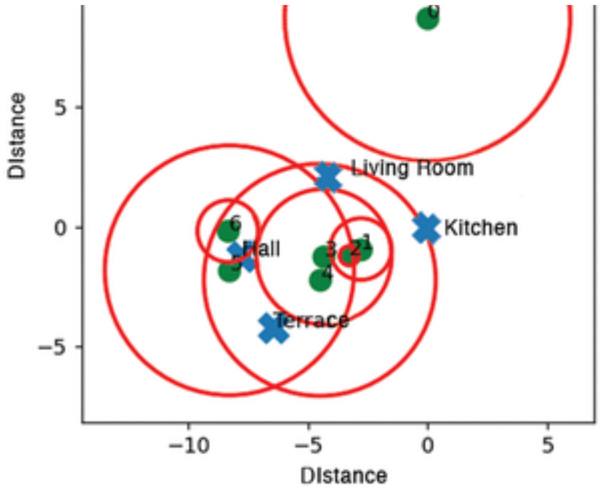


Figure 9: Example of an analytical solution.

The transmission power of hotspot 0 is fixed, corresponding to a radius of five meters in this case. If the values were robust, you would have expected a solution like the one shown in Figure 2, where all radii intersect in one point. In Figure 9, it takes good will to see where the living room should be in an optimal case.

So much for attempting to determine Tom's whereabouts analytically with the WLAN data.

1 2 3 Next »