**Evaluate Techniques for Wi-Fi Locationing**

by

Litan Li

IOT Analytics

August 17th, 2018

**Evaluate Techniques for Wi-Fi Locationing**

**Introduction**

The purpose of this work was the evaluate the feasibility of using Wi-Fi fingerprinting for indoor positioning. A random forest, k-nearest neighbor, and artificial neural network were trained, tuned, and optimized for performance using the UJIIndoorLoc indoor locationing benchmark downloaded from the University of California Irvine (UCI) Machine Learning Repository (Torres-Sospedra et al., 2014). A full machine learning pipeline was followed, and a sensible technique was used to deal with multi-label data. The results obtained were comparable, if not better than a baseline study reported by the same authors who created the dataset (Torres-Sospedra et al., 2015; Torres-Sospedra et al., 2017).

The problem of outdoor positioning has long been solved by global positioning systems (GPS), which uses satellite signals to triangulate the locations of users down to a level of accuracy that allows even for, for example, precise navigation of vehicles moving at high speeds. In many large, indoor, multi-floor complexes such as shopping malls, university buildings, public libraries, and airports, the abundance of overhead concrete and other building materials absorb GPS signals to a point that makes GPS ineffective. In the modern age where smartphones are pervasive, many smartphone applications could benefit tremendously from knowledge of the precise location of users in context-aware systems, even while users are indoors. These applications require a solution for accurate indoor locationing.

Much research has been conducted towards solving this problem. Currently, as of mid-2018, no single solution has achieved pervasive adoption. Proposed solutions vary in terms of the types of signals (radio-frequency identification, Bluetooth, Wi-Fi, ultrasound, etc.) used, and in the types of algorithms used. The use of Wi-Fi signal is a good choice due to the ubiquity of multiple wireless access points (WAPs) in modern-day buildings, and the ubiquity of smartphones. State-of-the-art solutions to this problem are often quite elaborate, involving various data processing choices and creative algorithms involving multiple steps. Since our goal is to demonstrate the feasibility of using Wi-Fi signals, we will be demonstrating simpler models which still give decent accuracy. To achieve this, we have built three models which give decent indoor locationing performance. Note that we use “locationing” and “positioning” interchangeably.

**Methods**

Original Data Description

The UJIIndoorLoc dataset is a set of indoor positioning datapoints collected at the Universitat Jaume I using over 25 Android devices varying in model and OS versions. Each example consists of a Wi-Fi “fingerprint” – a set of signal strengths received by the given device from 520 different WAPs at the device’s location. The signal strengths are in RSSI format, in units of decible-milliwatts (dBm), and take on integer values ranging from -104 (weak) to 0 (strong), with a value of 100 representing no signal detected.

The dataset covers roughly 1.2 million ft2 across three buildings, numbered 0, 1, and 2. Buildings 0 and 1 each contain 3 floors, numbered 0, 1, and 2. Building 2 contains 5 floors, numbered 0 through 4. The location in each example is represented by several variables: longitude, latitude, floor number, building number, space id (office, lab, etc.), and relative position (inside or outside the space’s entrance door). In addition, the dataset contains metadata for each example – the user id, phone id, and a UNIX timestamp for when the example was recorded. Table 1 summarizes key information about each variable.

|  |  |  |  |
| --- | --- | --- | --- |
| Table 1. Variables in the Dataset | |  |  |
| Column | Description | Units | Values |
| WAP001 - WAP520 | RSSI received by device from given WAP | dBm | Integer values from -104 to 0 (weak to strong), 100 (no signal) |
| LONGITUDE | Longitude of position | meters | -7695.9387549299299000 to -7299.786516730871000 |
| LATITUDE | Latitude of position | meters | 4864745.7450159714 to 4865017.3646842018 |
| FLOOR | Floor number | --- | Integer values from 0 to 4 |
| BUILDINGID | Building number | --- | Integer values from 0 to 2 |
| SPACEID | Integer identifying the space (lab, classroom, etc.) | --- | Various integer values |
| RELATIVEPOSITION | Relative position with respect to the space | --- | 1 - inside, 2 - outside in front of the door |
| USERID | User identifier | --- | Integer values from 0 to 18 |
| PHONEID | Android device identifier | --- | Integer values from 0 to 24 |
| TIMESTAMP | UNIX time when example was recorded | --- | Integer values |

The dataset comes in two csv files: “UJIIndoorLoc\_trainingData.csv” and “UJIIndoorLoc\_validationData.csv”. The former consists of 19937 examples from 933 distinct locations. The latter consist of 1111 examples from 1074 distinct locations and includes examples generated by users and smartphones that did not participate/were not used in generating “UJIIndoorLoc\_trainingData.csv”.

Dataset Terminology

The terminology we will adopt going forward for describing our datasets will be as follows. The “training set” is defined as the dataset used to train our models. The “validation set” is defined as the dataset used to check the performance of our models during tuning. The “test set” is a separate dataset, upon which predictions were made using the final model chosen after tuning. The performance of the final model on the test set is chosen as the indicator of how well the model generalizes. It is the “final” indicator of how good our chosen model is. In other words, a model is trained on the training set and tuned based on its performance on the validation set. The tuned model’s performance on the test set is then taken as how well the model does. We realize that the terminology may be slightly confusing at first, since we are calling “UJIIndoorLoc\_validaitonData.csv” as our “test set,” but we believe that this definition is necessary once we begin to explain our general machine learning approach in the following sections. Please also note that when using cross-validation, the validation set is the held-out fold in each cross-validation trial, and the average of the model’s performance on each held-out fold is reported as the cross-validation accuracy or kappa.

How Datasets are Used

In the random forest and k-nearest neighbor pipelines, “UJIIndoorLoc\_trainingData.csv” was used as the training and validation set. This dataset was used to train models with cross-validation, where in each cross-validation trial the held-out fold was used as the validation set and the remaining folds were used as the training set. This splitting into folds was done internally by the package with which we used to perform cross-validation. By “models”, we mean random forests or k-NNs with different sets of hyperparameters.

We did not use cross-validation when tuning the neural network. This choice was made because neural networks took much more time to train than conventional machine learning algorithms, even with GPU-enabled parallel processing. Therefore, in the neural network pipeline, “UJIIndoorLoc\_trainingData.csv” was split into a distinct training set and validation set, while “UJIIndoorLoc\_validationData.csv” was used as the test set.

Table 2 shows the purposes of each type of dataset used in our machine learning pipelines. Table 3 summarizes what the training set, validation set, and test sets are in each of the three models’ pipelines. We hope these tables will make exactly how we are using our data in this work clear.

|  |  |  |  |
| --- | --- | --- | --- |
| Table 2. Types of Datasets Used to in Machine Learning Pipeline | |  |  |
| Training Set | Used to train the models. |  |  |
| Validation Set | Used to check the performance of different sets of hyperparameters during tuning. |  |  |
| Is the held-out fold during each trial inside cross-validation when cross-validation was used. |  |  |
| The cross-validation accuracy and kappa is the average across the trials. |  |  |
| Is a separate dataset when cross-validation was not used. |  |  |
| Test Set | Evaluated by the tuned model as the final indication of how well the model performs. |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Table 3. | | | |
| Model | Training Set | Validation Set | Test Set |
| random forest | UJIIndoorLoc\_trainingData.csv | The held-out folds from UJIIndoorLoc\_trainingData.csv during each trial inside cross-validation, | UJIIndoorLoc\_validationData.csv |
| k-NN | UJIIndoorLoc\_trainingData.csv | The held-out folds from UJIIndoorLoc\_trainingData.csv during each trial inside cross-validation, | UJIIndoorLoc\_validationData.csv |
| neural network | A majority split from UJIIndoorLoc\_trainingData.csv | A minority split from UJIIndoorLoc\_trainingData.csv | UJIIndoorLoc\_validationData.csv |

One may wonder why we did not simply use the tuned models’ cross-validation score, or, in the case for the neural network, the tuned neural net’s performance on the validation set as the final indicator of how well the model performs. The general reason for this is that when searching across many sets of hyperparameters it is possible, by random chance alone, that a set of hyperparameters gives a good cross-validation score/good performance on the cross-validation set. This is a valid concern, especially since most models have more than 2 or 3 hyperparameters to tune, and the number of combinations of hyperparameters we can come up with is multiplicative, so we have many trials. If we reported the cross-validation score/validation set score as the final indicator of model performance, we could be falling into this trap and overestimating our tuned model. By choosing to report the tuned model’s performance on a separate test set as the final indicator, we essentially avoid this pitfall of overestimating model performance due to random chance. It is highly unlikely that a “lucky” model will get lucky both on cross-validation/on the validation set *and* on the test set.

Our test set is the “UJIIndoorLoc\_validationData- .csv” dataset, which includes fingerprints taken by devices that did not participate in generating “UJIIndoorLoc\_trainingData.csv”. This makes model performances evaluated on the test set quite indicative of real-world performance, as it contains not only examples that the model had not seen during training, but also examples generated by *devices* that the model had not seen during training. That being said, an alternative approach that we could have used it is to first mix “UJIIndoorLoc\_trainingData.csv” and “UJIIndoorLoc\_validation- Data.csv” together (randomly combining the rows from each), then 1) if using cross-validation, split the combined dataset into a training set and a test set, where the training set will be further split into folds during cross-validation, 2) if not using cross-validation, split the combined dataset into a training, validation, and test set. We think that the difference in the two approaches really depend on what philosophy we adopt when approaching this problem. In the design we have chosen, we are validating the model on examples that the model had not seen during training. Then, we are showing the model examples that it had not seen, but also examples that were generated by devices it had not seen. If the optimized model we tune will perform well on the test set, it must be more general to handle new devices, which is good. In the alternative design where we mix the two datasets, we are ensuring that the validation set and the test set come from the same distribution, so we are showing the model the devices that are only found in “UJIIndoorLoc\_validationData.csv” as well during training. We think that either design choice can be justified, and either one accomplishes the goal of this study, which is to evaluate the feasibility of Wi-Fi signals for indoor positioning.

Data Preprocessing

No missing values were found. This study used the Wi-Fi fingerprints (columns WAP001 through WAP520) as the features. Each received signal strength value was converted to a positive representation, with 0 representing no signal and 1 to 105 representing weak to strong signals. In any given example only a few WAPs were detected. Thus, a sparse matrix would more likely represent the data better, which required us to change the no signals representation from 100s to 0s.

Since the longitude, latitude, floor, and building number are enough to define a precise location, the space id and relative position were not used. Note that unlike in a typical regression or classification problem where there is a single label with two or more values or classes (e.g. “what’s the sales volume of this product”, “what is the object in this image?” or “what is the brand preference of this user, Sony or Acer?”), this problem consists of multiple labels (with each label containing multiple values/classes). To handle this, a single categorical label was created called UNIQUELOCATION, which take on integer values. As the name implies, the UNIQUELOCATION label takes on different values for each unique location, as defined by the longitude, latitude, floor number, and building id.

Features were not mean-centered, since that would destroy the sparse structure of the data. However, since gradient descent algorithms converge faster with normalized values, the features in the training set for the neural network were normalized to a 0 to 1 range by dividing by 105.

The package used to train the neural network required that categorical variables be one-hot-encoded into the dummy variable form. This was done for the UNIQUELOCATION label prior to neural network training. All features contained numerical values.

Our General Machine Learning Pipeline Approach

For each model type (random forest, k-NN, and neural network), we used the following approach to more systematically perform our data analysis and model building.

1. Import data
2. Visualize and explore the data
3. Process the data and feature engineering
4. Save processed datasets
5. Train and tune models
6. Report test set performance of model that gives best cross-validation score/validation set performance
7. Save best model
8. Predict new values (not done in this work)

Random Forest Classifier

*Algorithm*

The random forest classifier is an ensemble method, i.e. a method that combines the outputs from multiple models. A single decision tree tends to overfit the data. The random forest classifier takes the majority vote of the predicted class from *B* different trees, reducing overfitting. Each tree is constructed using a random sample of size *n*, chosen *with replacement* from the training set.

*Hyperparameters Tuned*

Here we show the names and descriptions of the hyperparameters that were tuned.

n\_estimators – the number of trees, integer.

criterion – the function used to measure the quality of a split, “gini” or “entropy”.

max\_features – the number of features to consider when looking for the best split, integer.

max\_depth – the maximum depth of each tree, integer.

All other hyperparameters were their default values as defined by the package.

*Model Training and Evaluation*

A grid search over the hyperparameters was used with 10-fold cross-validation. The set of hyperparameter values that gave the highest cross-validation accuracy was chosen as the best model. Cross-validation kappa was also calculated. The differences between the cross-validation scores and the average of the scores on the training folds were calculated to give us an idea of the degree of overfitting. The best model was then used to predict UNIQUELOCATION values on the test set. The predicted UNIQUELOCATION was then converted back to their longitude, latitude, floor number, and building id using a reference table. E.g. for UNIQUELOCATION = 1151, longitude = -7541.26 m, latitude = 4.86492e+06 m, floor = 2, and building id = 1. Finally, we report the following for the predicted test set locations:

Mean positional error – the Euclidean distance between the actual position and the predicted position, averaged over all test set examples. A position is defined by the longitude and latitude. Meters.

25, 50, 75, 95, and 100th percentiles of the positional error – also based on the Euclidean distance. Gives an indication of how close the most accurate predictions were, and how far away the most inaccurate prediction (100th percentile) were. Meters.

Building hit rate – the percentage of examples where the predicted building id was correct.

Floor hit rate – the percentage of examples where the predicted floor was correct.

K-Nearest Neighbors Classifier

*Algorithm*

The k-nearest neighbors classifier does not learn parameters as, say, a neural network does when it is trained. Given an example to be predicted, it first calculates the distances between the example and each example in the training set. Then, it assigns the majority voted class amongst the *k* closest examples, or neighbors, as the predicted class. The usual method to break ties is to simply assign the class of the closest neighbor.

*Hyperparameters Tuned*

n-neighbors – the number of neighbors to use, integer.

metric – the distance function to use; “euclidean”, “manhattan”, “chebychev”

All other hyperparameters were their default values as defined by the package.

*Model Tuning and Evaluation*

Used the same method as for random forest.

Artificial Neural Network

*Algorithm*

Neural network classification uses layers made of mathematical constructs that loosely mimic biological neurons. We will not go into much detail here, but give a sufficient overview. In a neural network, the features are inputted as a vector in the input layer, the first layer of the network. In the standard architecture, the input layer “neurons”, or units, are connected to a series of layers, each also containing multiple units, called the hidden layers. The final layer is the output layer. Since UNIQUELOCATION had more than two possible classes, we used a type of output layer that can represent the predicted class. In this case, we used a softmax output layer, which contains the same number of units as there are number of classes in UNIQUELOCATION. The value of each unit is interpreted as a predicted probability that the input example belongs to the class that the output unit represents. The predicted class for the input example is then chosen as the class with the highest predicted probability. The softmax layer uses a softmax activation function for each of its units. The hidden layer units use other activation functions. At each hidden layer unit, an activation function maps the (linear) combination of outputs from the units in the previous layer to a new scalar value. Non-linear activations functions, that is, activation functions that performs a non-linear mapping of the input to the output, is what enables neural networks to learn complicated and highly non-linear hypotheses. Finally, the weights and biases that are used to linearly combine layer outputs for feeding into units in the next layer are the parameters that the neural network learns during training. These values are sometimes learned using batch gradient descent, but more often by a faster variant called stochastic gradient descent, or other variants of stochastic gradient descent. Essentially, stochastic gradient descent finds the optimal weight and bias values by considering batches of training examples at each iteration instead of the entire training set. It then repeats this process after each training example has been used once, called an epoch.

*Hyperparameters Tuned*

Epochs – the number of epochs used during training, integer.

Batch\_size – the size of each batch used during one iteration of Adam (a variant of stochastic gradient descent that was used to train all neural networks tested), integer.

Hidden\_layers – the number of hidden layers, integer.

Neurons\_per\_hidden\_layer – the number of “neurons”, or units in each hidden layer, integer. Note that we did not try different numbers of units in each hidden layer.

L2\_reg\_lambda – L2 regularization parameter, float.

Dropout – dropout regularization parameter, the probability for a unit to be killed off during an iteration of optimization, float.

All other hyperparameters were their default values as defined by the package.

*Model Tuning and Evaluation*

A grid search was conducted manually. Cross-validation was not used due to the higher computational cost and training time required for each neural network. The set of hyperparameter values that gave the highest accuracy on the validation set was chosen as the best model. The differences between the validation set accuracies and accuracies on the training set were calculated to give us an idea of the degree of overfitting. The best model was then evaluated on the test set using the same method as for the random forest and k-nearest neighbors classifiers.

**Results**

Random Forest Classifier

*Hyperparameter Tuning Results*

Refer to “tuning\_rf.csv” to see the results for all hyperparameter combinations tried during tuning. The column names should be self-explanatory. The “mean\_val\_accuracy” and “mean\_val\_kappa” columns give the cross-validation accuracies and kappas, respectively. The “mean\_train\_accuracy” and “mean\_train\_kappa” columns give the accuracy and kappa on the training folds averaged across the 10 cross-validation trials. It’s like the cross-validation score, but for the training set.

*Hyperparameter Tuning Insights*

Before we talk about hyperparameter insights, a caveat: all insights gained should be treated with caution when extending to other datasets. For this dataset, the Gini splitting criterion generally performed better than the entropy criterion. The maximum depths we tried were too small, making the trees shallow and causing our random forest models to have high biases. We simply chose to not limit the maximum depth in subsequent trials, and allow the trees to grow to depths that give better training set and cross-validation scores. Using the square root of the number of features (~22) as the number of features to consider at each split was better than log2 (log2(520) ~ 9) and any percentage between 30 to 90% of the total number of features (156, 260, 364, 468). More trees generally reduced overfitting, as was expected. However, using more trees significantly increased memory usage during training. Due to limited PC memory, the number of trees was limited to 60 trees in the best model.

*Training Times*

Increasing the number of trees and the number of features considered at each split increased training times.

*Final Random Forest Model*

"RandomForestClassifier\_model.sav"

criterion: gini

max\_depth: not limited

max\_features: sqrt

n\_estimators: 60

Reason: gave good cross-validation scores. There's slight overfitting, suggested by the 11% gap between the cross-validation scores and the average training fold scores.

Training set performance (average of k-folds): accuracy 0.968 kappa 0.968

Cross validation performance: accuracy 0.856 kappa 0.8554

Test set performance:

mean positional error 8.579 m

25th percentile 1.466 m

50th percentile 5.551 m

75th percentile 11.218 m

95th percentile 28.303 m

100th percentile 94.208 m

Building hitrate 100%

Floor hitrate 90.4%

K-Nearest Neighbors Classifier

*Hyperparameter Tuning Results*

Refer to “tuning\_knn.csv” for the results from all hyperparameter combinations tried during tuning.

*Hyperparameter Tuning Insights*

Again, all insights gained should be treated with caution when extending to other datasets. Between the Euclidean, Manhattan, and Chebychev distance metrics, the Manhattan performed the best while the Chebychev performed considerably worse. A 1-nearest neighbor performed better than using 3, 4, and 5 neighbors.

*Training Times*

Increasing the number of neighbors to consider did not increase the training times by much, nor did the distance metrics tried seem to involve significantly more complex computations.

*Final k-NN Model*

"KNeighborsClassifier\_model.sav"

metric: manhattan

n\_neighbors: 1

Reason: gave good cross-validation scores. There’s some overfitting, suggested by the 14% gap between the cross-validation score and the average training set score across the folds.

Training set performance (average of k-folds): accuracy 0.966 kappa 0.966

Cross validation performance: accuracy 0.818 kappa 0.818

Test set performance:

mean positional error 12.394 m

25th percentile 1.875 m

50th percentile 6.017 m

75th percentile 12.342 m

95th percentile 37.355 m

100th percentile 369.158 m

Building hitrate 98.6%

Floor hitrate 88.0%

Artificial Neural Network

*Hyperparameter Tuning Results*

Refer to “tuning\_ann.csv” to see the results for all hyperparameter combinations tried during tuning.

*Hyperparameter Tuning Insights*

All insights gained should be treated with caution when extending to other datasets. Although increasing the number of epochs increased the gap between the training and validation set accuracy, both the training and validation accuracy increased. The training set accuracy just increased slightly more, causing the bigger gap. But, a higher number of epochs was favorable since the validation accuracy improved. Increasing the number of hidden layers from 1 to 2 only improved training set accuracy slightly and surprisingly, when increased to 5, decreased the training set accuracy. It seems like a shallow neural net was sufficient to represent the data. The validation set accuracy decreased for all cases with more than 1 hidden layer. The number of units in the hidden layer was set to a number between the number of input and output units. Increasing the number of hidden layer units made only a slight improvement to the validation accuracy, and using 3 hidden layers with more units still showed that using more layers did not help for this dataset. Surprisingly, neither L2 nor dropout regularization helped to reduce overfitting. That being said, we saw that dropout performed significantly better than L2 regularization for this dataset.

*Training Times*

GPU usage was only at around 18% usage during training. This was likely due to the GPU processing data faster than it can be fed data. This theory suggests read speed to the GPU was bottlenecked, which makes sense considering that increasing the batch size improved training times (without affecting the training and validation scores by much). Training times scaled linearly with the number of epochs. Not surprisingly, increasing the number of layers or the number of units increased training time, since increasing the number of units increases the number of parameters that gradient descent must find an optimal value for.

*Final ANN Model*

"ann\_model.h5"

epochs: 100

batch\_size: 800

hidden\_layers: 1

neurons\_per\_hidden\_layer: 1600

Reason: gave good validation accuracy. There's some overfitting, suggested by the 14% gap between the training and validation accuracy.

Training set performance: accuracy 0.945

Validation set performance: accuracy 0.800

Test set performance:

mean positional error 8.619 m

25th percentile 1.586 m

50th percentile 5.559 m

75th percentile 11.767 m

95th percentile 26.210 m

100th percentile 88.881 m

Building hitrate 100%

Floor hitrate 90.1%

Benchmark from Torres-Sospedra et al., 2017

Test set performance:

mean positional error 8.46 m

25th percentile 3.39 m

50th percentile 6.50 m

75th percentile 11.72 m

95th percentile 21.41 m

100th percentile 73.30 m

Building hitrate 100%

Floor hitrate 85.34%

**Conclusions**

Three solutions were created for the UJIIndoorLoc indoor positioning dataset. The random forest classifier gave the lowest positional errors, beating the benchmark in all except the 95th and 100th percentiles. The building hit rate was 100%, and the floor hit rate, at 90.4%, was around 5% higher than the benchmark. The neural network had a slightly higher overall positional error than the benchmark. It also outperformed the benchmark floor hit rate by close to 5%. Finally, the k-NN model gave higher positional errors and a slightly lower building hit rate at 98.6%. Its floor hit rate, at 88%, still outperformed the benchmark. Indoor locationing using Wi-Fi fingerprinting is certainly feasible, as we have demonstrated using two conventional machine learning algorithms and a neural network. However, there is much difference still between the performance of these models and a solution that can be successfully adopted for commercial use. We expect such a system to, for example, give a mean positional error on at least a centimeter scale.

**References**

Torres-Sospedra, J., Montoliu, R., Martínez-Usó, A., Avariento, J. P., Arnau, T.   
     J., Benedito-Bordonau, M., & Huerta, J. (2014). UJIIndoorLoc: A new   
     multi-building and multi-floor database for WLAN fingerprint-based indoor   
     localization problems. 2014 International Conference on Indoor Positioning   
     and Indoor Navigation (IPIN).

Torres-Sospedra, J., Montoliu, R., Trilles, S., Belmonte, &., & Huerta, J.   
     (2015). Comprehensive analysis of distance and similarity measures for   
     Wi-Fi fingerprinting indoor positioning systems. Expert Systems with   
     Applications, 42(23), 9263-9278.

Torres-Sospedra, J., Moreira, A., Knauth, S., Berkvens, R., Montoliu, R.,   
     Belmonte, O., . . . Peremans, H. (2017). A realistic evaluation of indoor   
     positioning systems based on Wi-Fi fingerprinting: the 2015 EvAAL-ETRI   
     competition. Journal of Ambient Intelligence and Smart Environments,   
     9(2), 263-279.