ECE 285 Final Project

Michael Threet
mthreet@ucsd.edu

Chenyin Liu
ch1586@ucsd.edu

Rui Guo
rug009@eng.ucsd.edu

Abstract

Source localization allows for range finding in underwater acoustics. Traditionally, source localization was done using Matched Field Processing, but this method has proven to be complicated to model and computationally expensive. This paper examines the used of three machine learning methods (Random Forests, Support Vector Machines, and Neural Networks) in the source localization problem, and does some fine-tuning to achieve acceptable results. Instead of treating source localization as a regression problem, this paper creates range classes based on "cutting up" the observed ranges into uniform chunks of distances. The results when using classification were largely successful. All three machine learning methods produced accurate results, with the Support Vector Machine performing the best.

1. Introduction

Source localization is an important problem in underwater acoustics. Using an array of underwater pressure sensors, the range of a passing ship may be estimated. This is normally done using Matched Field Processing (MFP), but this technique is not always straightforward. MFP requires the local ocean environment to be accurately modeled, but this is a very complicated task that produces unpredictable results. In addition, MFP can be computationally expensive when predicting a ship's range.

This paper uses machine learning techniques to perform source localization, namely Random Forests (RFs), Support Vector Machines (SVMs), and Neural Networks (NNs). These three techniques attempt to solve the issues that arise when doing MFP. All of the techniques do not need to model an underwater environment; instead, they require a (large) set of data to be trained and evaluated on. Additionally, while the machine learning techniques require a relatively long training time, their prediction times are quick and computationally inexpensive compared to MFP.

For more information, see [6].

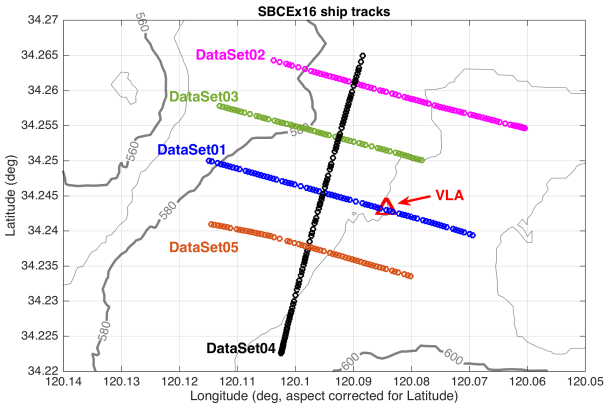


Figure 1: The paths of the ship that were tracked to obtain the training data

1.1. The Data

The data used in this paper was obtained from an underwater array of pressure sensors. To obtain the "ground truth" data, a ship was sailed on five different courses with its GPS position recorded, which provided the true ranges used for training the machine learning methods (see Figure 1). For this paper, only DataSet01 and DataSet02 were used.

2. Background

The three machine learning techniques used in this paper required an input that was a vector of observations or samples. To meet this requirement, some preprocessing was required. In addition, source localization was treated as a classification problem in this paper by discretizing the the ranges into a set number of classes. This led to much higher prediction accuracy, albeit at the cost of some range knowledge due to discretization. The three machine learning methods described in this section are therefore assumed to perform classification instead of regression.

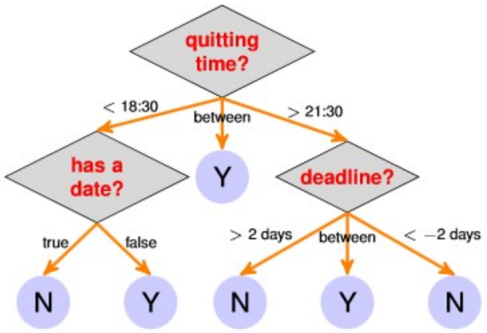


Figure 2: An example structure of a decision tree for determining whether it is time to quit a process

2.1. Preprocessing the Data

The data is initially a time series of pressure values received at L sensors. The DFT of the time series response at each sensor is taken to form a vector $\mathbf{p}(f) = [p_1(f), \cdots, p_L(f)]^T$. This vector is then normalized to

$$\tilde{\mathbf{p}}(f) = \frac{\mathbf{p}(f)}{\sqrt{\sum_{l=1}^{L} |p_l(f)|^2}} = \frac{\mathbf{p}(f)}{\|\mathbf{p}(f)\|_2}. \quad (1)$$

The sample covariance matrices are then averaged over N_s snapshots to form

$$\mathbf{C}(f) = \frac{1}{N_s} \sum_{s=1}^{N_s} \tilde{\mathbf{p}}_s(f) \tilde{\mathbf{p}}_s^H(f), \quad (2)$$

Only the real and imaginary parts of the complex valued entries of diagonal and upper triangular matrix in $\mathbf{C}(f)$ are used as input to save memory and improve calculation speed. These entries are vectorized to form the real-valued vector $\mathbf{x}$ of length $L \times (L+1)$, which is used as the input vector to all three machine learning techniques used in this paper.

For more information, see [6].

2.2. Random Forest

A RF is a well-known machine learning method based on decision trees. A RF is composed of many decision trees, each of which can provide a class prediction [5]. Since a RF is composed of many decision trees, it can form more complex features and relationships from the data than a single decision tree. At prediction time, the RF selects the class that is predicted by the most decision trees as the true class. The subsections below describe the major components of the RF algorithm.

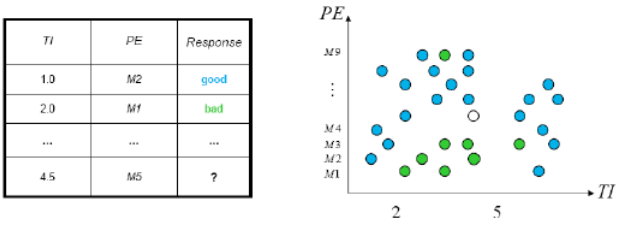


Figure 3: The raw data before decision tree classification

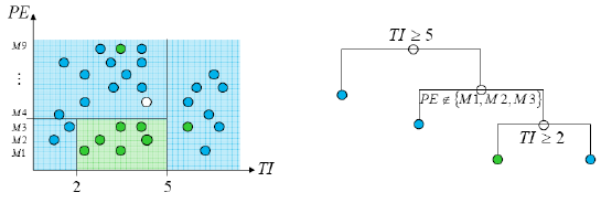


Figure 4: The data after decision tree classification

2.2.1 Decision Tree

The decision tree grows from the root, or topmost node. For this paper, the root would be something of the form: "What is the range of the observed ship given the sample covariance input?". When the root tries to grow it will judge the growth condition based on the value of Gini impurity (see Equation 3). As in Figure 2, the decision tree grows from its root and forms nodes based on input values and information from previous decisions [1]. Moreover, any feature can just be a part of decision tree and there will not be any exception.

According to the accommodation of feature adjustment, any feature of the data can be a part of a decision tree [5]. In a RF, the objective is to put features into decision trees. The number of decision trees depends on the architecture of the RF and the complexity of the input data. The decision tree is a classifier that divides the input data into different classes.

Figure 3 shows a distribution of raw data. The data consists of good, bad, and unsure samples. The data has no clear groupings or shared features between each class type. However, a decision tree can be used to classify the raw data. Figure 4 shows the result of applying a decision tree to the raw data. The groupings are not perfect, but they do manage to capture most of the classes correctly.

2.2.2 Gini Impurity

The Gini impurity is used to find the optimal partition. The Gini impurity is described as

$$I_G(f) = \sum_{i=1}^{J} f_i(1 - f_i) \quad (3)$$

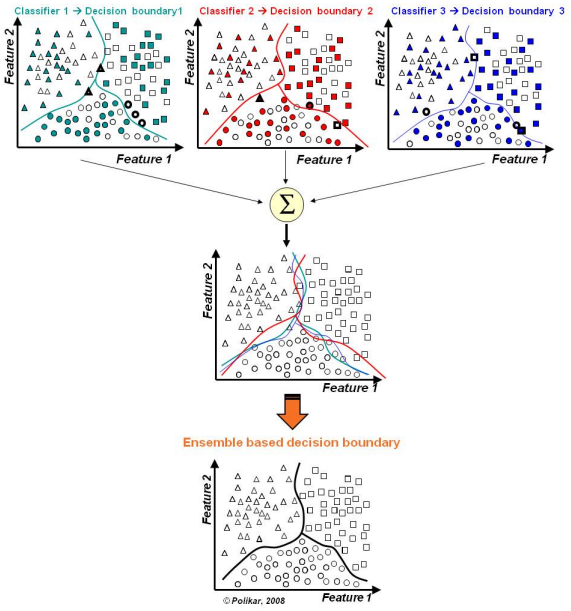


Figure 5: An example of the bagging algorithm

where $I_G(f)$ is the Gini impurity, J is the total number of classes, and f_i is the fraction of items labeled as class i in the dataset. The Gini impurity measures how often a sample would be mislabeled if its label was randomly chosen [1]. This allows the decision tree to "split" the dataset in the best manner possible, as it can measure the likelihood of a mislabel based on the current input class.

2.2.3 Bagging

Bagging is a method used in a RF to avoid overfitting. Bagging can be used for classification to improve test accuracy and lower the variance of the model [5]. Bagging involves randomly selecting (with replacement) a subset of the training data, and training a single decision tree on this subset. This is done over many training iterations, and allows for different decision trees within the RF to form independent features of the training data. At prediction time, the response of each decision tree within the RF is observed, and the class that was predicted the most often is chosen as the predicted class. See Figure 5 for a visual example of bagging.

2.3. SVM

In machine learning, Support Vector Machines are supervised learning models that analyze data and are used for classification and regression. The basic idea is, given a set of training examples each marked as belonging to one or the other of two categories, a SVM builds a model that assigns new examples to one class, making it a binary linear classifier [2]. The SVM attempts to form a hyperplane that best separates the two classes, first by maximizing the number of correctly labeled examples, and then by maximizing each correctly labeled example's distance from the hyperplane.

In addition to performing linear classification, SVMs can perform a non-linear classification by using the kernel trick, implicitly mapping the inputs into high-dimensional spaces [3]. In this paper, 7200 data inputs are mapped to 1 of 150 total classes.

2.4. Principal Component Analysis

Introduction: The main goal of PCA is to reduce the dimension of data space and fasten the model built time. PCA is a procedure that uses an orthogonal transformation to convert a set of observations into a set of values of linearly uncorrelated variables called principal components. The number of principal components is less than or equal to the number of original variables.

Take a data matrix X, with observations in its columns. The column-wise mean is then subtracted from X to center the observations around 0. PCA transforms a set of p-dimensional vectors using weights $w_k = (w_1, ..., w_p)_k$ that map each row vector x_i of X to a new vector of principal component scores $t_i = (t_1, .., t_m)_i$, given by $t_k i = x_i * w_k$ for $i = 1, ..., n$ and $k = 1, ..., m$. PCA attempts to ensure that each variable in t inherits the maximum possible variance from X, so that the transformed data retains as much of its original shape as possible. See [7] for more information.

Using PCA with a SVM: PCA is used to reduce the dimensionality of the input vectors to the SVM. In this paper, the original dimensionality is 7200, which is very large and leads to long and computationally expensive training times. With the benefit of PCA, the dimensionality of the input vectors can be reduced while still maintaining enough information to accurately predict labels.

2.5. Neural Network

A neural network contains a number of hidden layers, each with neurons that take inputs from the previous layer and connect their outputs to the next layer (see Figure 6). The number of hidden layers and the number of neurons in each hidden layer can be varied to create networks that are very deep and complex, or networks that are shallow and less computationally expensive to train [4].

Each neuron takes a linear combination of the outputs of the previous layer as its inputs. The output of each neuron is a non-linear function applied on this linear combination (usually a sigmoid, hyperbolic tangent, or ReLU function). By applying non-linear functions, the neural network is able to learn more complex features, as it is not limited to just linear combinations of the inputs.

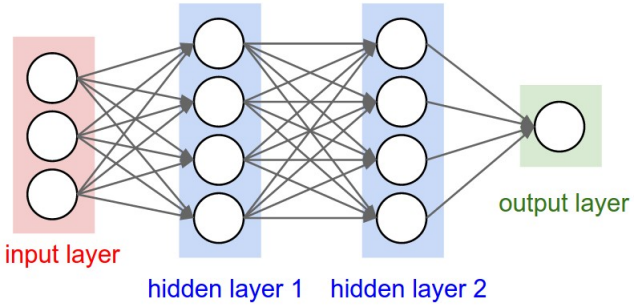


Figure 6: An example architecture of a neural network with two hidden layers

Mathematically, this means that the input to the j^{th} neuron in the k^{th} layer is

$$i_k = w_{k-1,j}^T o_{k-1} \tag{4}$$

and the output of each neuron is

$$o_k = f(i_k) \tag{5}$$

where $w_{k-1,j}$ is the vector of learned weights for the j^{th} neuron, o_{k-1} is a vector of the output of each neuron in the $(k-1)^{th}$ layer, and $f(x)$ is the activation function of the neuron.

A neural network is "trained" by using error back-propagation. A training example, accompanied with its correct label or output, is given to the network. After forward-propagating the input using the current weights, the error is calculated. Working backwards, the network can use this error to adjust the weights for every neuron in every hidden layer [4]. With enough training examples, the error should converge to a small value, and the weights should stabilize to their"ideal" values.

The neural networks used in this paper are implemented using the scikit-learn MLPClassifier. While this is not the most flexible model, it allowed for the easiest implementation and training. The networks use two hidden layers with ReLU activations. The number of neurons in each layer varied as an experimental parameter. Additionally, the solver used to train the network was varied, with the Stochastic Gradient Descent (SGD), Limited-memory Broyden-FletcherGoldfarbShanno (LBFGS), and Adam solvers being used.

3. Experiments and Results

3.1. Random Forest

The RF has many tunable parameters and implementations. A good starting point is with the default setup of the scikit-learn RandomForestClassifier.

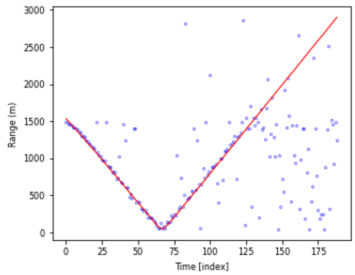


Figure 7: The RF test results before parameter tuning

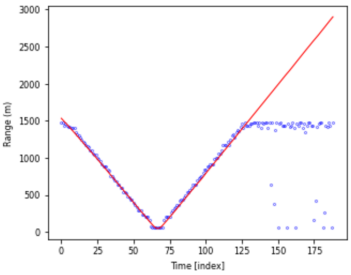


Figure 8: The RF test results for the first dataset after parameter tuning with a MAPE 17%

3.1.1 Parameters change

After some testing, it became clear that certain parameters have a larger effect on the error rate than others. The important features were found to be the number of trees, the number of features, and the depth of the trees. Figure 7 shows the test results for the first dataset with the default RF configuration. The results are not good, as there are a lot of samples scattered around without any structure.

Figure 8 shows the test results for the second dataset with the tuned RF parameters. This result looks much better, as there is a lot more structure to the predictions and a respectable MAPE of 17%. The tuned parameters were 800 trees, 100 features, and a depth of 13.

Next, the tuned RF was trained and tested on the second dataset to determine its robustness. Figure 9 shows the results of this test. While the RF produced a lower MAPE of 13% for the second dataset, the visual results do not look as promising. The RF appears to have minimized its error by guessing a nearly constant value, which is not a promising result.

3.2. SVM

When implementing models using SVMs, a variety of kernels should be considered. In this paper, three kernels were evaluated.

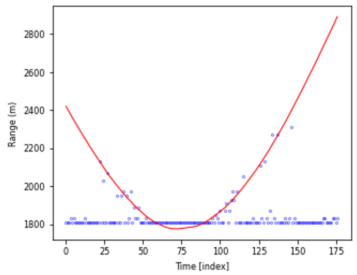


Figure 9: The RF test results for the second dataset after parameter tuning with a MAPE 13%

```
Time taken to build model: 52.8 seconds

=== Evaluation on training set ===

Time taken to test model on training data: 26.12 seconds

=== Summary ===

Correctly Classified Instances         178               87      %
Incorrectly Classified Instances         0                0      %
Kappa statistic                          1
Mean absolute error                      0.1137
Root mean squared error                  0.1826
Relative absolute error                 86.3811 %
Root relative squared error             86.4629 %
Total Number of Instances              178
```

Figure 10: The test results for the first dataset using a polynomial kernel, C = 1,degree = 1

3.2.1 Linear Kernel

A linear kernel is the simplest option and the easiest to implement. While linear kernels typically do not perform as well as non-linear ones, they are often used as a baseline due to their simplicity. Linear kernels are less computationally expensive, which allows them to achieve much faster training times, at the cost of some accuracy.

Linear kernels depend mainly on a penalty parameter. This parameter, usually specified as a float between 0 and 1, determines if the SVM should focus on increasing classification accuracy or maximizing the distance of datapoints from the separating hyperplane. For a penalty parameter that is close to 0, the SVM will try maximize the distance from the separating hyperplane, even if it leads to misclassification. For a penalty parameter that is close to 1, the SVM will focus on increasing classification accuracy, even if it leads to very small margins for hyperplane separation. For this paper, the penalty parameter was set to 1.

3.2.2 Polynomial Kernel

A polynomial kernel maps the input data into a higher dimensional space, allowing for more complex features to be formed, as the separating hyperplane can now appear as non-linear in the original sample space. A polynomial kernel will usually obtain a higher classification accuracy than a linear kernel, albeit with a longer training time and a risk of overfitting.

Polynomial kernels depend on the same penalty parameter as the linear kernel. Polynomial kernels also depend on a parameter that changes the order of their higher dimensional mapping. A higher order parameter can lead to more complex features being formed, but it may also result in longer training times and overfitting.

3.2.3 Radial Basis Function Kernel

A RBF kernel determines an input vector's distance from an arbitrary point, and uses that as a feature for learning. The RBF kernel can be thought of as a similarity measure, since it just compares a distance between two points. The RBF kernel is usually more flexible, since it can model complex functions much easier than a linear or polynomial kernel [3].

RBF kernels depend on the same penalty parameter as the linear and polynomial kernels. RBF kernels also depend on a parameter gamma that affects the "reach" or radius of a single training example. A large gamma limits the range of a training example's "reach", and leads to overfitting. A small gamma increases the "reach" of each training example, and leads to each support vector spanning the entire dataset. A "correct" value for gamma will allow for locally similar examples to be grouped together and for distant examples to be classified as a separate class.

3.2.4 SVMs in Weka

For this paper, weka was used to implement a multiclass SVM. See Section 5 for more details.

3.2.5 Test Models and Results

With the benefit of PCA, the input dimensionality can be reduced by 60% while still maintaining 90% of the original variance. This greatly reduced the training and evaluation time. When tuning the parameters, such as gamma for the RBF kernel, it is beneficial to adjust the parameters in an exponential order. For example, for gamma, generate a SVM model with a RBF kernel and try $\gamma = 2^{-16}, ..., 2^{-8}, ..., 2^{-2}, 1, 2, 4, ...$ to find a decent range of gammas to fine-tune the model with.

The best results for the first dataset came from a polynomial kernel with a degree of 1. Figure 10 shows these results.

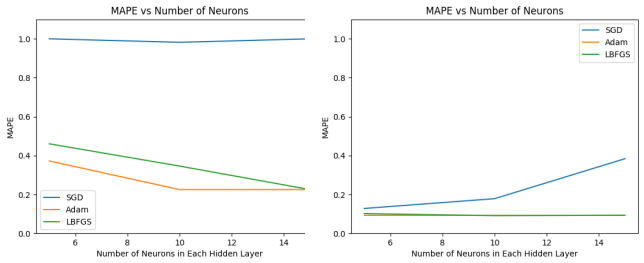
The best results for the second dataset came from a RBF kernel with a gamma of $\frac{1}{128}$. Figure 11 shows these results.

3.3. Neural Network

The NNs were first evaluated based on the number of neurons in each hidden layer. In this case there were two

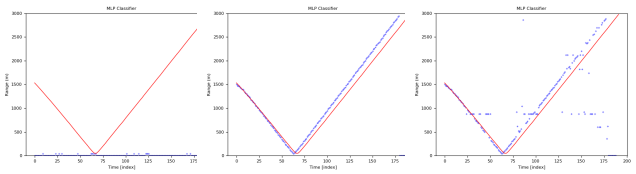
```
Time taken to build model: 55.3 seconds

=== Evaluation on training set ===

Time taken to test model on training data: 17.79 seconds

=== Summary ===

Correctly Classified Instances         178               89.3      %
Incorrectly Classified Instances        0                 0        %
Kappa statistic                         1
Mean absolute error                     0.0955
Root mean squared error                 0.1526
Relative absolute error                89.7913 %
Root relative squared error            89.5567 %
Total Number of Instances              178
```

Figure 11: The test results for the first set using a RBF kernel, C = 1, gamma = $\frac{1}{128}$



(a) (b)

Figure 12: The MAPE results for each dataset



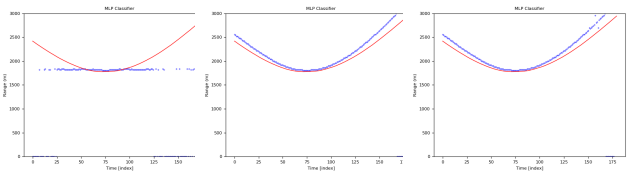
(a) (b) (c)

Figure 13: The test results for the first test case with the a) SGD, b) Adam, and c) LBFGS solvers and 10 neurons per hidden layer

hidden layers, both with 5, 10, or 15 neurons. In addition, a different solver was used for each number of neurons. Figure 12 shows the Mean Absolute Percentage Error for each dataset using the three solvers and a variable number of neurons in the hidden layers.

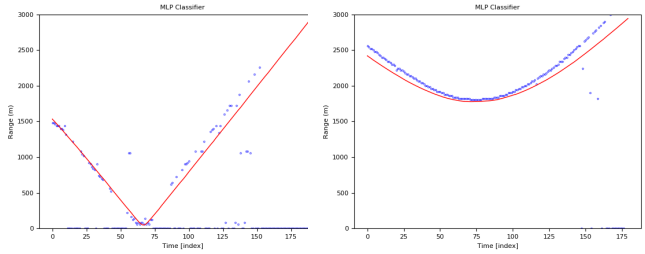
For the first dataset, the Adam solver performs best on average, but both the Adam and LBFGS solvers reach the same solution at 15 neurons per hidden layer. The SGD solver does not perform well at all, with a MAPE of almost one (the worst score possible). Figure 13 shows the test results for the case with 10 neurons in each hidden layer for the dataset.

For the second dataset both the Adam and LBFGS solvers reach essentially the same result for each number of neurons per hidden layer. The SGD solver performs much better on the second dataset, but, interestingly enough, the



(a) (b) (c)

Figure 14: The test results for the second test case with the a) SGD, b) Adam, and c) LBFGS solvers and 15 neurons per hidden layer



(a) (b)

Figure 15: The test results for the updated SGD architecture. The MAPE for the first datset was 67%, while the MAPE for the second dataset was 13%

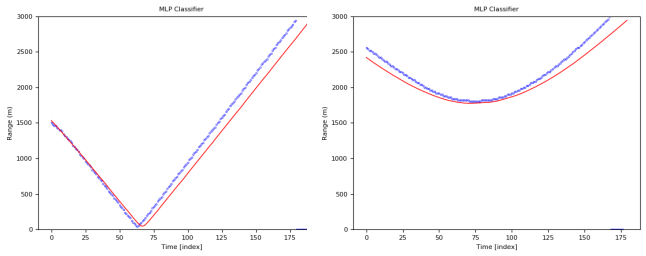
MAPE actually increases as the number of neurons increases from 10 to 15. Figure 14 shows the test results for the case with 15 neurons in each hidden layer for the second dataset.

A closer look at the results reveals that the SGD solver always guesses the minimum value of the test data. This leads to larger MAPE and a useless result, since the NN only guesses one value.

Based on these results, the NN was tested on what parameters to adjust to make sure that the SGD solver reached a better solution. One reasonable assumption was to increases the number of neurons in the hidden layer, since a larger number of neurons can help to form more complex features in the dataset. The other option was to change the activation function of each neuron. The activation was changed from a ReLU to an identity function (i.e. there was no non-linearity applied to the linear combination of inputs).

Figure 15 shows the test output for both dataset using a NN with two hidden layers, 50 neurons per hidden layer, and an identity function used at each neuron. This new architecture worked well for the second dataset, but not for the first dataset.

To make sure that the SGD solver produced an adequate result, two more changes were made. The first was to increase the number of neurons in each hidden layer to 100.



(a) (b)

Figure 16: The test results for the second updated SGD architecture. The MAPE for the first datset was 19%, while the MAPE for the second dataset was 9%

The second was to increases the number of training iterations from 200 to 1000. With more training iterations, the SGD solver should reach a better solution. Figure 16 shows the new results.

With these changes, the SGD solver was able to achieve results similar to the Adam and LBFGS solvers.

4. Conclusion

Method	Dataset 1	Dataset 2
RF	17%	13%
SVM	13%	11%
NN	19%	9%

Table 1: The best MAPE scores for each method on each dataset

All three of the machine learning methods performed well, with a MAPE of 19% or lower for all test cases. The NN achieved the lowest MAPE, but it also achieved the highest. This variance (albeit in a small sample size) could mean that a NN is not the best option for source localization. The RF performed the next best on average, but it did not achieve the best MAPE score for either of the test cases.

The SVM performed best on average, and it also performed best on the first dataset. Based on these results, it seems that the SVM is the best machine learning method to use for source localization. These results also align with the results in [6], although the MAPE scores in this paper are higher due to a lack of fine-tuning.

While some of the results were not promising (i.e. the RF on the second dataset and the NN with the SGD solver), most of the results showed great promise. With more time and computational power, a machine learning approach could reach much greater accuracy and robustness. Overall, however, the results demonstrate that machine learning approaches are a viable option for source localization.

References

[1] G. Biau and E. Scornet. A random forest guided tour, 2015.

[2] I. Carmichael and J. Marron. Geometric insights into support vector machine behavior using the kkt conditions, 2017.

[3] W. M. Czarnecki and J. Tabor. Cluster based rbf kernel for support vector machines, 2014.

[4] F. Giannini, V. Laveglia, A. Rossi, D. Zanca, and A. Zugarini. Neural networks for beginners. a fast implementation in matlab, torch, tensorflow, 2017.

[5] G. Louppe. Understanding random forests: From theory to practice, 2014.

[6] H. Niu, E. Reeves, and P. Gerstoft. Source localization in an ocean waveguide using supervised machine learning, 2017.

[7] J. Shlens. A tutorial on principal component analysis, 2014.

5. Appendix

5.1. Setting Up Weka

Installation: The lastest weka tool can be found through this link: `http://www.cs.waikato.ac.nz/ml/weka/downloading.html` It has stable and testing versions for Windows, OSX, and Linux.

Convert the Data: Weka has its own data input format, with the most common being arff and csv. In this paper we are using a csv format file. The training and testing data must be converted to a csv format with the data and labels in the same file. Data will be denoted as a numeric value while label is a nominal value. Run the python file to_csv.py provided. Be sure to specify the directory of data.

Data Preprocessing: Click "open" and navigate through the folder and open train.csv Once opened, click the choose button under the filter, select NumericToNominal under filter/unsupervised/ folder, and then click on function parameter. Change the column 7201 to nominal which is the feature. Click apply. Next, click on the choose button once again under the filter, select Principal components under the same folder, and click apply. By default it will cover the varaince under 0.95 and reduce the dimension to 5. Option: If it still gives the out of memory error, choose the resample function, change the percentage to 50, and click apply.

SVM Classification: After data preprocessing, click on classify and choose SMO filter under classifer/functions. Set the appropriate values to be the same as the parameters in python file. For the kernels discussed in this paper, weka has all the available functions. For specific setup, check Figure 17. For the test options, use training set and select (nom)class as the feature. Click start. The weka tool comsumes a lot of memory even after PCA. Be sure when starting weka to specify the -Xmx which is the memory buffer size for running this JDK.

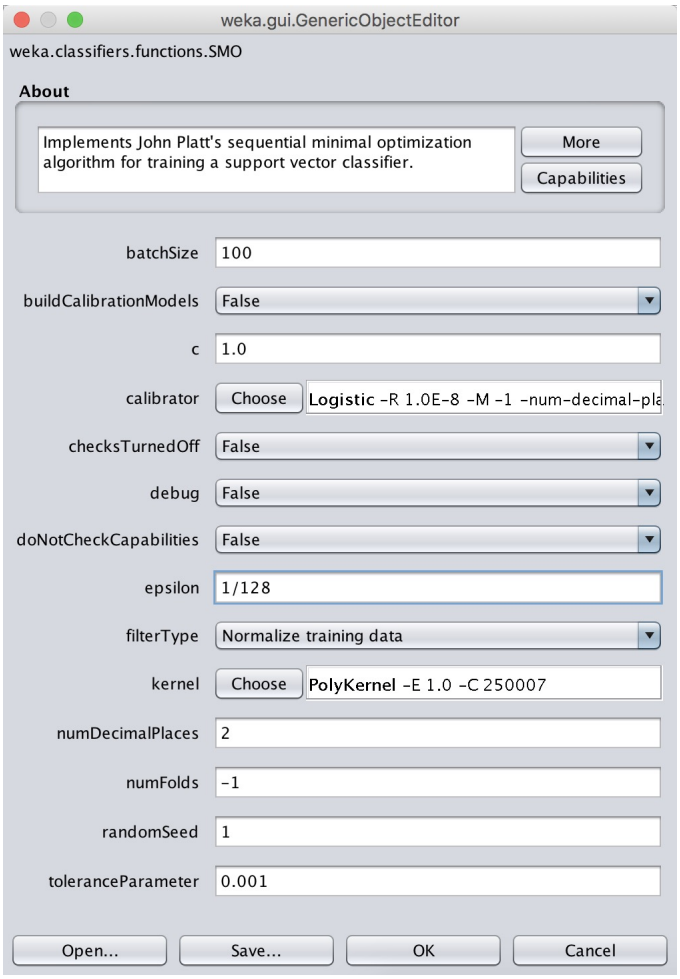


Figure 17: SMO setup for polynomial kernel