CS 4641: Machine Learning
Final Project

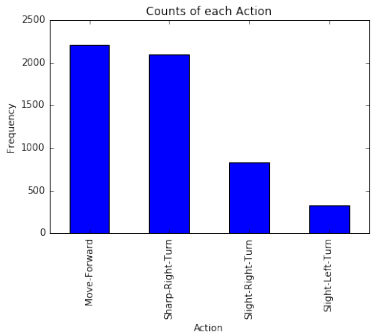
Sachin Konan

December 7, 2019

The Dataset

The dataset that will be investigated is the Wall-Following Robot Navigation set from the the UCI Public Machine Learning Repository. The set was created by **Ananda Freire, Marcus Veloso and Guilherme Barreto**, and consists of 24 radially positioned ultrasonic sensors which measure distance in a floating-point format at approximately 9 samples per second. Based on these ultrasonic measurements the robot makes 1 out of 4 of the following decisions: **Move-Forward**, **Slight-Right-Turn**, **Sharp-Right-Turn**, **Slight-Left-Turn**. The 24 radially positioned sensors are placed at 5°or 15°increments around the robot. The goal of this experiment was for the robot to constantly "follow" a wall. This problem is particularly interesting to me, because I have a passion for robotics, and in the past, I have created wall-following robots, but used simple, if-else logic. I thought it might be interesting to apply Machine Learning to this same problem, and see if I can get better results.

The underlying supervised learning problem here is determining a relation between the sensor measurements and the action of the robot, in order to maintain a "closeness" to the wall. Since all the sensors are radially oriented around the robot, the problem immediately seems to require a non-linear classifier, because while it may seem logical that if a wall were close to the robot, the closest co-linear sensor might linearly increase or decrease, but since there is a circular distribution of sensors, the data-set can be assumed to be non-linear *but this will be proved in the next section*. In order to test the performance of non-linear classifier models, firstly we will look to the cross-validation of the model, or in other words, how the model performs on data it has never seen before. The dataset consists of 5456 rows, but will split at 80, 20 for training and testing respectively.



In terms of measuring "performance," a confusion matrix will be constructed. This measures the number of true positives, false positives, false negatives, and true negatives. Ideally, if the data is balances in terms of the distribution of y-class labels, then it would suffice to just use accuracy with the test data, but as seen in the graph to the left. Since there is a significantly larger portion of **Move-Forward** and **Sharp Right-Turn** labels than the other two simply calculating the $\frac{true\ positives}{total}$ would be irrepresentative of the true performance of the model. There are two other statistics, precision, recall, and f1-score that may aid in measuring the performance of a model. $Precision = \frac{TP}{TP+FP}, Recall = \frac{TP}{TP+FN}$. Precision will measure how well the model performs on positive labels; recall is similar, but incorporates FN to penalize when the model makes a wrong positive prediction. In order to incorporate both statistics, we will use also record the F1-Score, or the harmonic mean of the precision and recall: $\frac{2*(Precision*Recall)}{Precision+Recall}$.

Algorithms

Linear Separability

I sought to use Logistic Regression to prove that this data-set can't be trivially solved by simply generating hyper-planes that separate the data, because this would indicate that the data is linearly separable. This achieves a **67.3%** accuracy rating on the validation data-set, which, because the error rate is so high, shows that the data is likely linearly inseparable. Additionally, this can be seen by looking at the extensive classification report:

	precision	recall	f1-score	support
0	0.69	0.71	0.70	543
1	0.79	0.65	0.72	628
2	0.39	0.56	0.46	55
3	0.43	0.64	0.51	138
accuracy			0.67	1364
macro avg	0.58	0.64	0.60	1364
weighted avg	0.70	0.67	0.68	1364

The model has an overall f1-score of 68%, but the non-linear aspect can be seen by examining the low f1-scores present on the 2nd and 3rd class labels. This shows that some sort of non-linear model will be needed to alleviate these issues.

Random Forests

Decision Trees are models that look to classify by "splitting" the data on different feature values in the data-set. In order to improve the accuracy of the model, one can increase the depth of a decision tree, but often times this can lead to over-fitting of the model, because the splits for certain classes, will become specific and become incapable of generalizing. An improvement on this technique is using bagging, which is the technique that will be used in this report. Bagging consists of splitting the data-set into many sub-samples and training individual decision trees on those samples. Upon prediction, one can simply take the most frequently reported class across all Decision Tree's; this method reduces variance relative to a standard Decision Tree. Random Forests improve upon the bagging concept, by also randomizing the sub-set of features a Decision Tree can split upon. This helps prevent over-fitting, because upon each iteration of training there is a different set of features to split against and there is more "exploration" of the split of features. The following are important hyper-parameters for tuning a Random Forest Algorithm:

1. **number of random split points** is generally known to use $\sqrt{p}$, where p is the number of features.
2. **the max depth of each decision tree** determines how many levels or splits that the individual tree takes.
3. **the number of estimators** is the number of decision trees that will be used in the forest. The more decision trees, the less variance in the overall prediction, but this has its costs in accuracy.

Increasing any of the above parameters, increasing the complexity of the hypothesis class.

Support Vector Machine

Support Vector Machines are a class of Machine Learning algorithms which seek to generate a hyper-plane which separates the data into discrete classes based on the maximum margin principle. The maximum margin principle essentially means the distance between the separating hyper-plane and the "support-vectors", the data-points closest to the given hyper-plane, is maximal. All together, the prediction looks like:

$h = sgn(w^T x + w_0)$. This algorithm works by minimizing the overall Lagrange error and such that all the support-vectors are correctly predicted. This can be summarized by the below equations:

$$\alpha = \arg\max_{\alpha} \sum_{i=1}^{N} \alpha^{(i)} - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha^{(i)} \alpha^{(j)} y^{(i)} y^{(j)} \mathbf{x}^{(i)\top} \mathbf{x}^{(j)} \tag{1}$$

Subject to

$$\sum_{i=1}^{N} \alpha^{(i)} y^{(i)} = 0, \quad \alpha^{(i)} \geq 0, \quad \forall \alpha^{(i)}$$

The above equations describe a linear SVM, however the addition of kernel functions into the h function or the Lagrange error summation, allow Linear SVM's to solve non-linear classification problems. For example, if we are working with data from two horizontal cross-sections of a parabaloid, then the data isn't separable in 2D, but in 3-D it is separable. SVM's can be improved by tuning the following hyper-parameters:

1. **Kernel Function** Since it has already been proven that the data is not linearly separable, non-linear kernel functions will be tested. Radial Basis, Sigmoid, and Polynomial will be tested. For the Polynomial basis function, varying degrees of polynomial ≥ 1 will be tested.

2. **Gamma** For non-linear kernel functions, there is an additional gamma term that is multiplied to the output of the kernel. Increasing gamma results in increases in bias and decreases in variance.

3. **C** is known as the cost of mis-classifying a point. Lower values of C are known to lower bias, because you harshly change the model when a point is misclassified.

Neural Networks

Neural Networks are a set of Machine Learning Models based off neurons in the brain. Each neuron has edges coming in and out, where each edge is a weight which is multiplied with the data flowing into that edge. At each neuron, all the incoming edges are summed and some sort of function is applied to generate non-linearity. Neural Networks can consists of many different parameters, which is why to test a Neural Network it is useful to tune the following hyper-parameters.

1. **Number of Hidden Layers**: The number of layers in between the input and output. Increasing the number of layers helps fine-tune the model, because there are many more weights which contribute to a NN's overall performance. This increases the complexity of the model greatly.

2. **Number of Hidden Nodes**: The number of nodes in each hidden layer. Generally increasing the number of hidden nodes has a larger immediate effect on performance of a model, and increases the complexity of the model.

3. **Activation Function**: The function that introduces non-linearity at each node. The ones that will be tested are: *sigmoid*, *tanh*, *relu*.

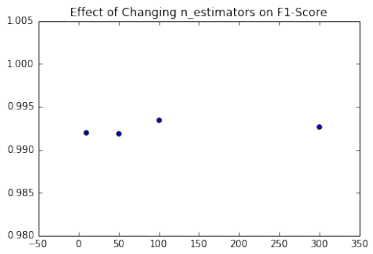
4. **Batch Size**: The number of training examples inputted into the model at once. Large batch sizes mean the model's current parameters are exposed to more data at once.

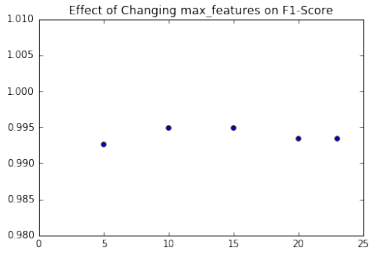
Tuning Hyper-Parameters

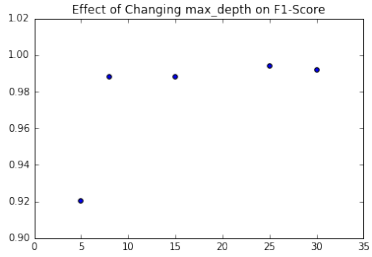
The data should be noted that the data was split into 75% and 25% for training and testing, respectively. This means that the training data consists of 4092 values and the testing data consists of 1364. Each of the tuned models were validated using the test data.

Random Forests

$$n_{estimators} = [10, 50, 100, 300]$$
$$max_{features} = [5, 10, 15, 20, 24]$$
$$max_{depth} = [5, 8, 15, 25, 30]$$







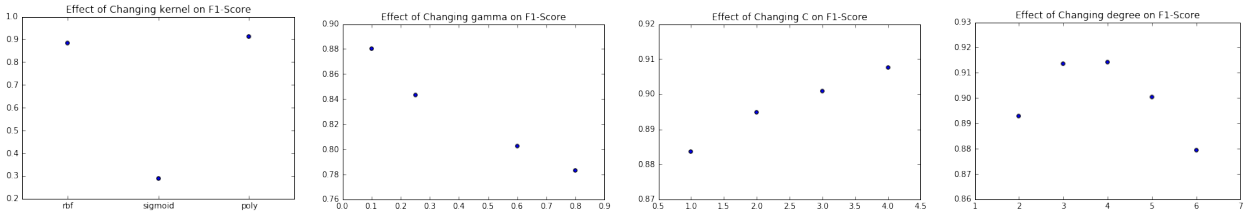
Generally speaking from these graphs, we can see that as we increase the number of estimators, there isn't a significant increase in the f1-score, but the highest f1-score is at **100**, so that will be selected as a the $n_{estimators}$ parameter. On the next graph, we find that the largest f1-score occurs at **15** max-random-features, so that will be selected as the $max_{features}$ parameter. On the next graph we see that as long as the max_{depth} exceeds 5, the max_{depth} doesn't change much. Therefore, we simply select the largest max_{depth}, which is **30**. This produces the following metrics on the test data set.

	precision	recall	f1-score	support
0	0.99	0.99	0.99	560
1	1.00	0.99	1.00	521
2	0.96	1.00	0.98	77
3	1.00	1.00	1.00	206
accuracy			0.99	1364
macro avg	0.99	1.00	0.99	1364
weighted avg	0.99	0.99	0.99	1364

Random Forests ended with an f1-score of **99%**. In order to validate the tuning, a **grid-search** was performed on the exact same ranges, and the same results were found. Additionally, each graphic and its experiment took approximately 30 seconds to generate.

Support Vector Machine

$$kernels = [rbf, sigmoid, poly]$$
$$gamma = [0.1, 0.25, 0.6, 0.8]$$
$$C = [1, 2, 3, 4]$$
$$degree = [2, 3, 4, 5]$$



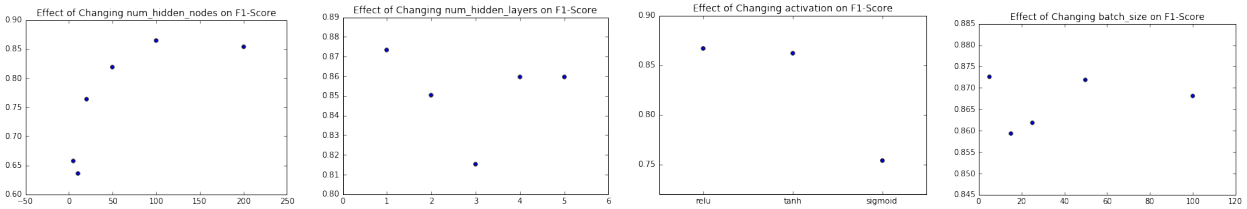
Firstly, the kernel was tested, as it has the highest on the non-linear ability of the SVM. As seen on the below graph, **rbf** and **poly** have the highest f1-scores, but **poly** has the highest f1-score, so it was selected as the **kernel**. On the next graph as we increase the **gamma** value, the f1-score decreases at an increasing rate, so $gamma = 1$. As we increase **C**, the f1-score decreases, so the lowest C value was selected, $C = 0.1$. Finally, since the **poly** kernel was tested, various degrees of the polynomial function were tested and it was found that $degree = 3$ maximized the f1-score.

	precision	recall	f1-score	support
0	0.92	0.91	0.91	569
1	0.92	0.94	0.93	508
2	0.93	0.95	0.94	78
3	0.90	0.89	0.89	209
accuracy			0.92	1364
macro avg	0.92	0.92	0.92	1364
weighted avg	0.92	0.92	0.92	1364

Overall, the SVM model achieved a **92%** f1-score. In order to validate the tuning, a **grid-search** was performed on the exact same ranges, and the same results were found. Additionally, each graphic and its experiment took approximately 43 seconds to generate.

Neural Networks

$$\text{num of hidden nodes} = [5, 10, 20, 50, 100, 200]$$
$$\text{num of hidden layers} = [1, 2, 3, 4, 5]$$
$$\text{activation} = ['relu', 'tanh', 'sigmoid']$$
$$\text{batch size} = [5, 15, 25, 50, 100]$$



Firstly, the number of hidden nodes was tuned from [5, 200]. It can be seen as the number hidden nodes increase, the f1-score seems to increase at an increasing rate until the number of hidden nodes is 100. This is the local maximum of the graph, so therefore the most optimum number of hidden nodes is **100**. The number of hidden layers doesn't follow the same pattern, the most optimal number of hidden layers seems to maximized when the number of hidden layers is **1**, so that was chosen. For the activation function, relu performed the best to the activation function chosen was **relu**. The batch size which maximized the f1-score was a batch size of **5**, so that was the chosen batch size of the model.

	precision	recall	f1-score	support
0	0.90	0.95	0.93	529
1	0.97	0.92	0.96	544
2	0.96	0.94	0.94	82
3	0.92	0.91	0.89	209
accuracy			0.94	1364
macro avg	0.94	0.93	0.93	1364
weighted avg	0.93	0.93	0.94	1364

The neural Network ended with a 94% f1-rating on the validation data. Each graphic and its experiment took much longer than SVM and RandomForest, because each neural network was trained with 100 epochs. Each graph and its experiment took around 5 minutes to generate.

Evaluation

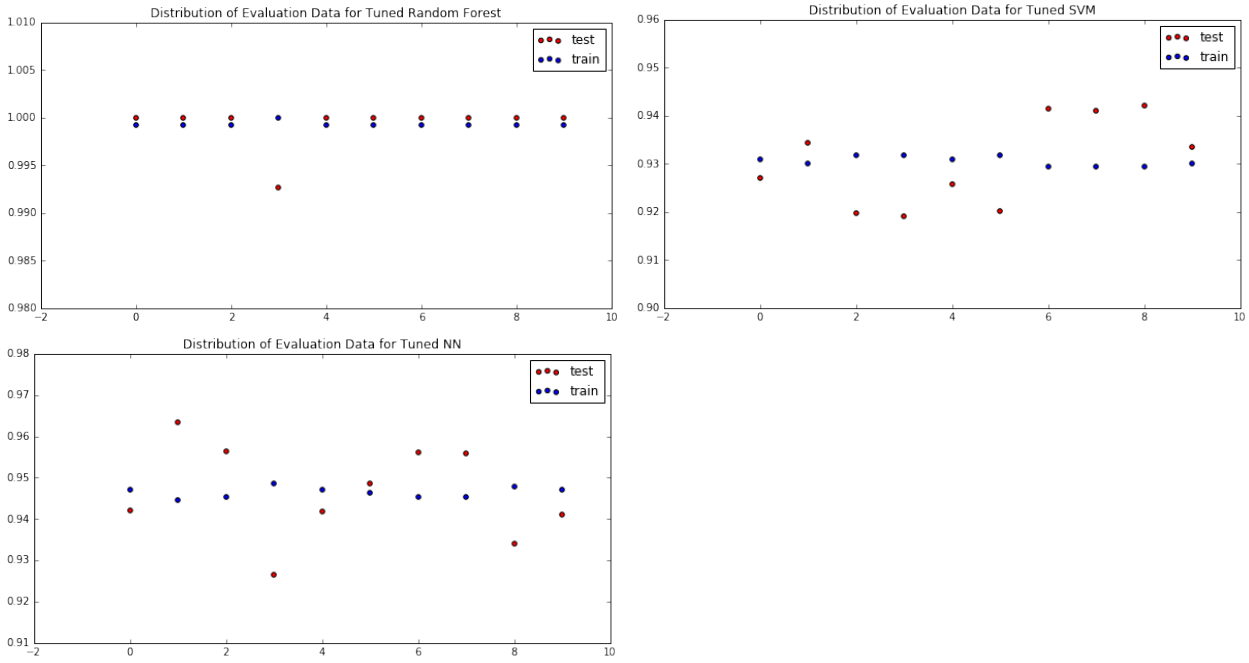
As stated earlier the data were split into a 75% and 25% for training and testing respectively. In order to test each tuned models' effectiveness, a K-fold cross validation, with $k = 10$ was run and the mean, standard deviation, and distribution of scores for each model was recorded.

For the training folds:

	random forest	svm	neural network
Mean	0.99926629	0.9305307	0.94643708
Std	0.00024457	0.00095902	0.00122223

For the testing folds:

	random forest	svm	neural network
Mean	0.99926535	0.93042176	0.94659084
Std	0.00220396	0.0088799	0.01091705



It is clear that there is a lot of statistical and graphical variability from the NN experiment. The points have a larger spread than any of the other graphs, which indicates that the model may have been over-fitting

to the training data. For the SVM graph, there is less variability, but only slightly. This comes at the cost of f1-rating, because the neural network is almost 1.6%. The model with the f1-rating and lowest variability is the Random Forest. There is very little spread in the data, and it very consistently centered around 99% accuracy, which is outstanding for the test data. Random Forest vastly performs all other models without over-fitting, but the question is why? This can be solved by going back to the introduction of the data-set. I was actually able to program a robot to follow a wall with only if-else statements, with logic such as "if the forward sensor has a high value, the backwards sensor has a high value, the left sensor has a high value, but the right sensor has a low value, then I will turn right, because the wall is closest to the right sensor". This logic can be expanded for many other-cases, but in-essence there are a series of conditionals that can more or less encapsulate this problem. This is why Random Forest works so well! Random Forests are essentially if-else statements on different features in the dataset, but with much more complexity and randomness, which help improve its bias and variance. However, NN's and SVM's, will transform the numerical data with dot products and matrix multiplications, and at the very end decision boundaries and argmax, respectively determine the class of a given input, which is where some of the loss comes about. In essence, this problem was meant for something like a Decision Tree to begin with, because trees will retain the splitting property necessary for this problem.

Conclusion

Overall, it was clear that the Random Forest performed the best on this dataset. In terms of the number of testable hyper-parameters, Random Forest actually had the least, because SVM had 4 (3 main ones and 1 for testing the degree of only the poly kernel) and the Neural Network had 4 as well. However, in terms of timing, Random Forests were the most optimal in terms of tuning hyper-parameters. The only time Random Forests were slower was during GridSearch, because there were more sub-classes for each hyper-parameter. The algorithm with the greatest complexity and greatest computational overhead was the Neural Network, which had the most number of floating point weights, and required two matrix multiplications for every forward pass. The algorithm with the least complexity was the Random Forest, because it simply had to split across various features, but that was bounded by the hyperparameter. Overall, Random Forest would easily make the most sense to be used in a real-world scenario, especially in robotics. Robotics requires real-time performance, and decision trees/random forests are really nice because they skip the floating point calculations required by other machine learning models. Because there is a series of comparable conditions for determining the class of the data, a roboticist could decompose a random forest algorithm into a series of if-else statements, or even implement them in an FPGA hardware device, which uses AND, OR, NOT, and NOR gates to simulate the same computational behavior, but faster.

Acknowledgements

1. For adding f1-metric to keras model:
   https://datascience.stackexchange.com/questions/45165/how-to-get-accuracy-f1-precision-and-recall-for-a-keras-model

2. For giving me ideas on how to run the models on my own data:
   https://www.kaggle.com/propanon/cardiotocography-under-a-microscope