A Survey of Supervised Learning Algorithms

An Analysis of Performance and Complexity

Manikanta Reddy Dornala
CS 7641 - Machine Learning, Georgia Institute of Technology

1 CLASSIFICATION PROBLEMS

If we have a set of points ($\mathcal{X}$), in any dimension ($\mathcal{D}$), and each sample has some label given from a label set ($\mathcal{L}$), then the classification problem is to assign a label from $\mathcal{L}$ to an unlabeled point when presented with some labelled points.

$$\mathcal{X} = \{x_1, x_2, x_3, \ldots\}$$
$$\mathcal{L} = \{l_1, l_2, \ldots\}$$
$$\mathcal{F} : \mathcal{X} \to \mathcal{L}$$

The goal of any classification algorithm is to learn(*fit*) the function $\mathcal{F}$, the labeling function.

If $\mathcal{L}$ is a set of only two labels then it becomes a binary classification problem. In this survey we only consider binary classification problems. Multi-label classification problems, with L labels can be converted into L binary problems, by defining a new problem for each label that classifies them as having a label l and not having label l for each $l \in \mathcal{L}$.

Some algorithms implicitly do well with Multi-label classification while some algorithms only solve binary classification (and consequently multilabel ones). This does not mean they cannot perform good multi-label classification, just that the way we implement them is different. For example, if we use an SVM for multilabel dataset (with $|\mathcal{L}| \neq 2$ we will have to build many SVM models instead of one to do a *one-vs-one* or *one-vs-many* labeling. That didn't seem fair. Hence we compare the algorithms on equal footing with only binary classification.

2 DATASETS

We generate two different kinds of datasets to bring into light the ability of the algorithms. There are two kinds of datasets in our experiments. Linearly separable and linearly in-separable. Some algorithms are inherently linear and try to find a hyper-plane that separates the two classes, while some can create more complex decision boundaries.

The datasets generated contain points that can be classified into one class or the other based on some function, which we want the algorithms to learn. The important thing to note is that the data is separable and not randomly classified into two sets, although the algorithms will try to find the function corresponding to the random classification anyways. The datasets contain points in a D dimensional space.

2.1 Linearly Separable Data

If $\mathcal{F}$ is such that it defines a hyper-plane in D dimensions that divides the data in X into two classes($\mathcal{L} = \{l_1, l_2\}$) then we say the data is linearly separable, otherwise there will be a complex function that can classify the points making the problem linearly inseparable.

Mathematically if there exists a hyper-plane $f : w^T x + b = 0$ such that we can define F as

$$\mathcal{F} = \begin{cases} f(x) = w^T x + b > 0 & \implies l_1 \\ f(x) = w^T x + b < 0 & \implies l_2 \end{cases}$$

then the set X is linearly separable in D dimensional space.

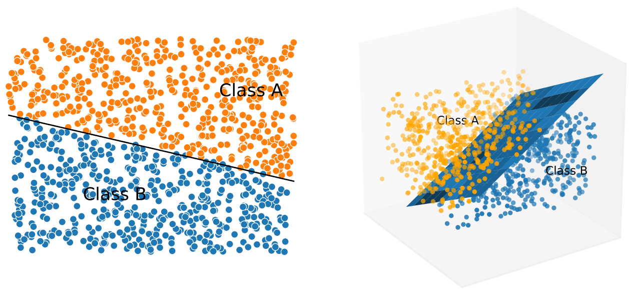


Figure 1: Visualizing Linearly Separable data in 2d and 3d

Here we see how the points are distributed in space. The two colors represent different *classes*. The separating hyper-plane f in the 2d space is the line in black and in 3d space its a plane.

2.2 Linearly Inseparable Data

In this second set we generate data such that there is no linear hyper-plane that can separate the two classes. The simplest way to do this is two imagine two concentric circles. The separating boundary in such a case would be a *hyper shell*. The points are classified based on their distance from the center (C). Such a function would look like

$$\mathcal{F} = \begin{cases} f(x) = dist(x, C) - R > 0 & \implies l_1 \\ f(x) = dist(x, C) - R < 0 & \implies l_2 \end{cases}$$

Note that this is specifically for the data being generated in our experiments. There are uncountable number of ways to generate linearly inseparable data with more complex decision boundaries.

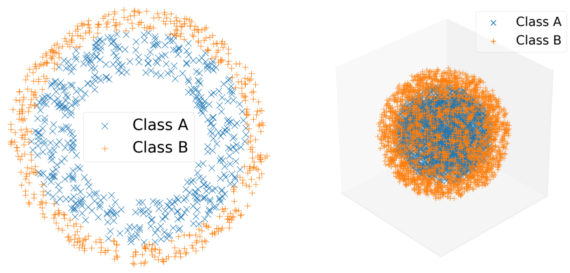


Figure 2: Visualizing Linearly In-Separable data in 2d and 3d

3 EXPERIMENTS

We'll implement various algorithms under various hyper parameters to understand how these hyper parameters affect the algorithm's performance. Performance metric for us is the accuracy of the algorithm in correctly predicting a blinded set, its stability across various data points (i,e. its ability to generalize and not be specific to the set we picked) and the time it takes to learn.

The implementation of the entire pipeline is in python and associated libraries including sklearn, numpy, pandas, seaborn and matplotlib. The plots shown in the next sections are for Accuracy, Training Time, Learning Curve and ROC. In the plots there are some areas in transparent gray. It captures the standard deviation of the metric in 5-fold cv.

3.1 Data Generation

We decided to generate a 32 Dimensional data with 10,000 unique points. They are generated using the above mentioned methods in section 2. Of these 10000 points in each dataset 20% are sampled and set aside and will not be used for training purposes. This will be our testing set and the accuracy of a model is assessed on this set when comparing different algorithms.

3.2 Hyper-Parameter Tuning

Each algorithm has a ton of hyper parameters that can be tuned. To find the best set for each algorithm and thereby generate its best model that can generalize we perform a cross validated hyper parameter space search.

The search of this space can be performed in various ways. Grid Search being a very straight forward one. If there are 3 hyper-params that can be chosen from $A = \{a_1, a_2, ..., a_{k_a}\}, B = \{b_1, b_2, ..., b_{k_b}\}, C = \{c_1, c_2, ..., c_{k_c}\}$ we do a ordered search of params among the $k_a * k_b * k_c$ possible parameter groups. This is a very time consuming process but will give us a great insight into how the variation in a particular hyper-param affects the algorithm's performance (both time and accuracy). Another way to do a hyper-param search is to perform a Random search in the hyper-param space. Usually the hyper-params are correlated with the performance (as you'll see later) and this enables us to directly sample from a distribution of hyper-params and *climb the hill*. But in our experiments we will use a grid search to understand this behavioral variance disregarding any correlation.

In order to have faith in the chosen hyper-parameter set, we should see whether the algorithm generalizes its learnings and is not conforming to the data set fed into it. In order to achieve this we perform a 5 fold cross validation. In every fold, 20% of the training set (containing 8000 points) is randomly sampled and kept aside, the model is trained on the remaining 6400 points on all hyper-parameter sets in the search space and its accuracy is tested for on the separated 20% set. Once this is done for all the 5 folds, the accuracy for each hyper-parameter set is averaged across folds and the best performing set is picked.

3.3 ROC Curve

For a classification problem it is not just enough to look at how the testing accuracy we need to look at its sensitivity and precision as well. Both of these are well captured in a ROC curve, the area under that curve being a good measure of trust.

3.4 Learning Curves

In order to understand how well an algorithm learns its good to see how much data it consumes in process. Learning curves are generated over the entire dataset of 10000 points by cross validating on 20% validation set and increasing the number of training points from 1 to 8000 in 5 folds.

3.5 Effect of Dimensions

Why 32 features is good question, hence I also threw in an experiment that compares the algorithms accuracy when changing the number of features.

3.6 SVC

Support vector machines are inherently linear algorithms that try to find an optimal hyper-plane that *discriminates* the two classes. This algorithm accepts various hyper-parameters that govern its learning. Among them we chose the Kernel function and the Regularization parameter.

3.6.1 Kernel Function

If the data is not linearly separable it helps to jump into higher dimensions and look at it again. For example if there was a third dimension in the 2d linearly in-seperable data that captured how far the point is from center, ie, $(x, y, x^2 + y^2)$ then the data would

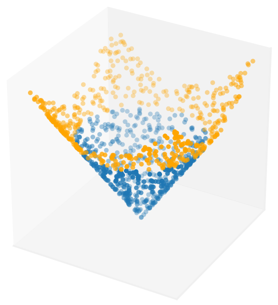


Figure 3: Kernel Function $(x, y, x^2 + y^2)$ **on some 2d inseparable data**

look something like a cone with upper half being one class and the lower half the other (Figure 3). Then we can have a hyperplane seperating the two classes. This was a function that computed the radius.

Instead SVM can use a kernel function that projects the data in a different space. We experiment on two of them. Linear and Radial Basis Function. The linear one doesn't change anything. RBF on the other hand *simulates* the projection in infinite dimensions.

3.6.2 Regularization Parameter

Termed C in sklearn library is a cost associated with the tolerance for misclassification. A smaller C would mean the machine looks for a plane that has a large margin, meaning a plane that can seperate the classes very distinctly conversely a larger C looks for a smaller margin.

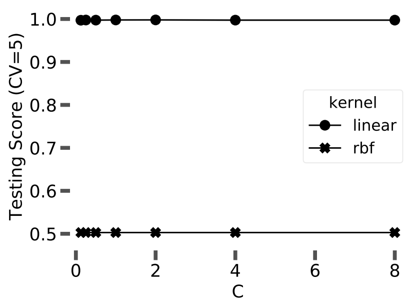
3.6.3 Analysis

SVM performs well on the linearly separable data. On dataset1 svm finds the best model with linear kernel and has no regard for C (as the data already had a clear seperation with good margin). On the other hand linear kernel on dataset 2 can never do better than 0.5 accuracy as any hyper plane would cut the hyper-sphere at best in half and its learning time keeps increasing with C. On dataset 2 it might have worked well with a polynomial kernel of degree two, however rbf is an infinite projection and may introduce some information that's hard to interpret, this is also evident from rbf performing bad in dataset 1.

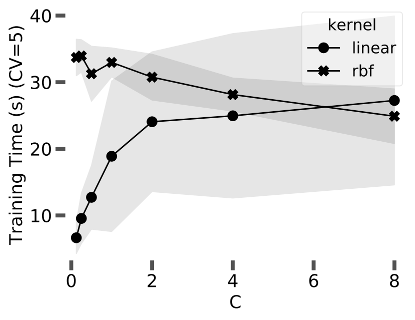
Both take relatively same amount of time for their best models for training. Notice how increasing C increases training time for linear kernel while decreasing for rbf on both datasets. This is because they are trying to find even smaller margins and taking more iterations for it but due to the actual linear hyperplane being only of a certain margin and they cannot do any better than it, which is also why the accuracy plateaus.

With dataset 1 the algorithm is able to very well generalize the seperating plane as seen in learning curves. For dataset 2, it needs relatively more points to reach better accuracies. Dataset 1 being very seperable in its own space, a small sample data is enough to generate the hyperplane.

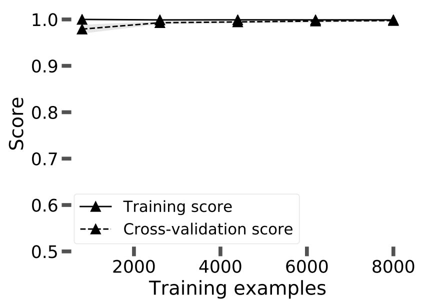
On the blind set AUC is very high for both suggesting class imbalance in test wouldn't have affected the accuracy. Notice how the testing accuracy for dataset 2 is same as the training one although the AUC is 1.



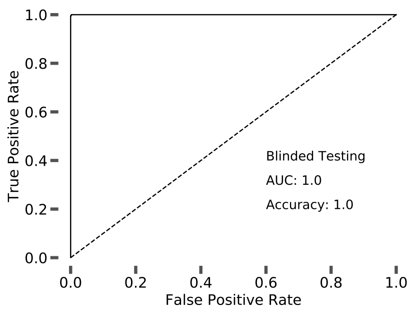
(a) Hyper Parameter Search



(b) Tranining Times for Params

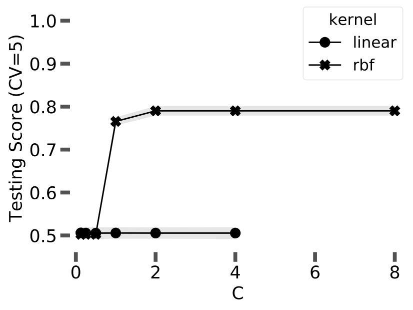


(c) Learning Curve

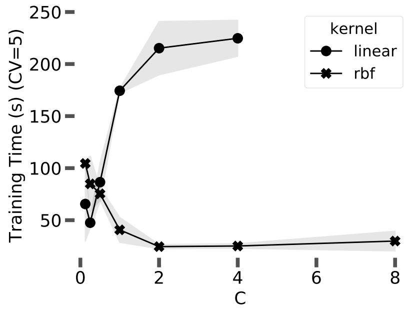


(d) ROC Curve

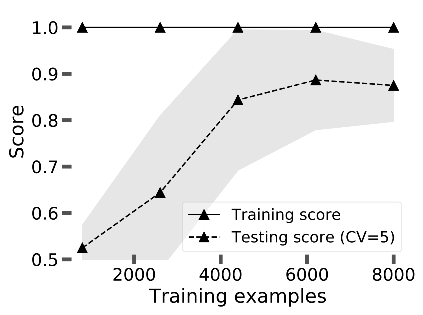
Figure 4: SVC on Linearly Seperable Data



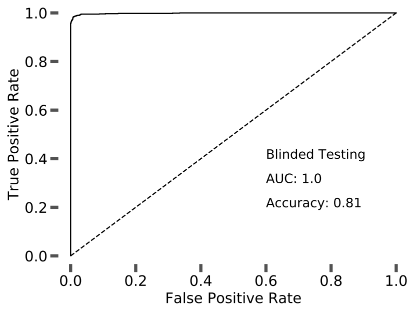
(a) Hyper Parameter Search



(b) Tranining Times for Params



(c) Learning Curve



(d) ROC Curve

Figure 5: SVC on Linearly Inseperable Data

3.7 DTC

Decision Trees work by making threshold cuts on various features at different places repeatedly. As a result they end up making high resolution grids as separation boundaries. The cuts are made until a split results in one set having some *purity* or a desired criteria is met. The basis of these cuts is decided by an impurity function. The algorithm will accept when a cut with a certain tolerance of impurity is made and the two halves are compared. The hyperparameters tested are the maximum depth of the tree and the criterion of splitting.

3.7.1 *Impurity Function*

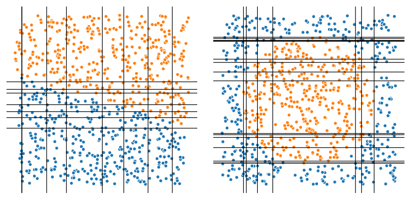
An impurity occurs when a set has instances of a different class than intended for. The algorithm may stop splitting because we run out of features to split on, or we can *tolerate* a certain percentage of impurity. This is is also called gini index of impurity.

Another way to decided upon is to look for the randomness(aka impurity) in the set as Entropy. Based on this impurity functions a DT would choose a feature, threshold to split on that causes it to have maximum information gain.

3.7.2 *Maximum Depth*

Fixing the maximum depth of the tree is a way of pruning. This allows for faster training but generates weaker learners for lower depths.

Here is a sample of how the DT algorithm would define its boundaries in 2d. This is with gini criterion and a max depth of 8. Notice how the splitting is more pronounced at the boundaries of two classes. Gini is forcing the algorithm is to make a cut around the boundary to decrease the impurity count. Essentially DT tried to approximate the circle as an edged polygon in the inseperable case and jaggered line in the seperable one.



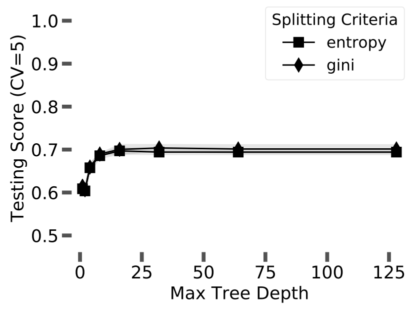
3.7.3 *Analysis*

Notice how increasing the tree depth after a certain threshold doesn't affect the accuracy as well as the training time. This is because the algorithm exhausted the available features. When does that occur? $depth > D = 32$.

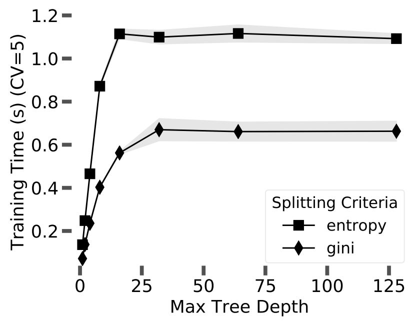
Entropy computation is costlier than simple impurity count. Hence entropy takes significantly more time in both the cases. There seems to be little effect due to the choice of splitting criteria. Maybe because both were able to provide same gain.

Surprisingly approximating the non-linear boundary is done in a better way than the linear one. Although it takes more time to do so, dataset 2 has higher accuracy in both validation and blinded testing. It is also evident from the generalization in AUC.

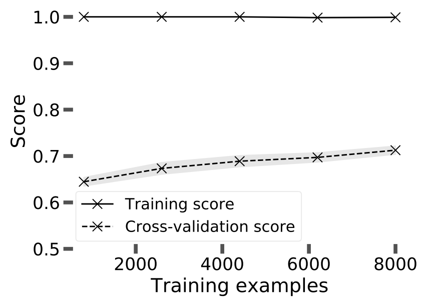
The learning curve suggests that the algorithm will definitely benefit from being trained on more samples. More samples translates to finer boundaries and better accuracy.



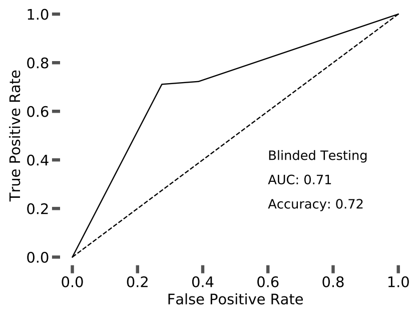
(a) Hyper Parameter Search



(b) Tranining Times for Params

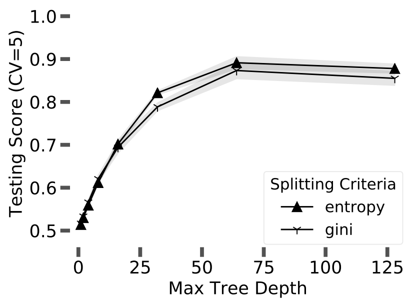


(c) Learning Curve

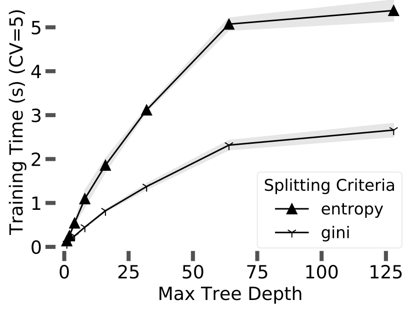


(d) ROC Curve

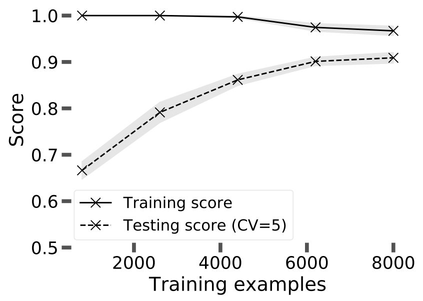
Figure 7: DTC on Linearly Seperable Data



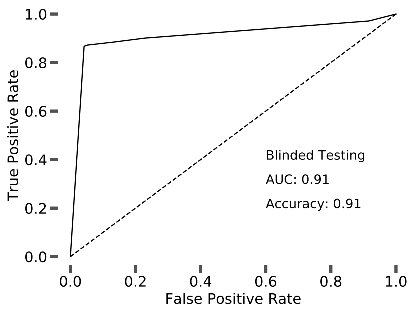
(a) Hyper Parameter Search



(b) Tranining Times for Params



(c) Learning Curve



(d) ROC Curve

Figure 8: DTC on Linearly Inseperable Data

3.8 ABC

The Ada Boost classifier is an ensemble approach that combines weak classifiers to *emulate* a better one. A single classifier may perform poorly but the combination of them, with correct voting strategies can create a better classifier with higher accuracies.

This algorithm trains new weak classifiers by selectively choosing data points based on the accuracy of previous classifiers. All such classifiers are assigned appropriate weights based on their accuracy and voting is performed.

The choosing of the data points is done in a way that mis-classifications from previous classifiers are better understood in the subsequent classifiers. The individual classifiers have to be better than random, so anything less than 0.5 is discouraged by assigning a negative weight and so on. A typical weight (Alpha) vs error rate would look like Figure 9.

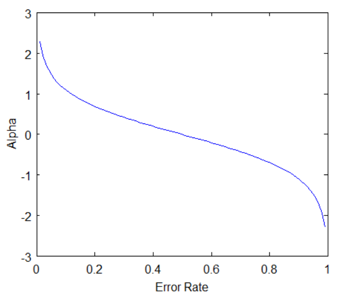


Figure 9: Typical Alpha-Error Curve [1]

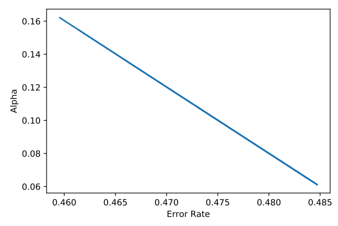


Figure 10: Current Alpha-Error Curve

The current classifiers were all better than random and lied in the small linear region (Figure 10)

Decision tress were used as the weak classifiers for the algorithm and their depth and number are varied. The decision tree splitting criteria was set to 'gini' for faster training times as seen previous section.

3.8.1 Max Depth

Varying the Max Depth of the decision tree will change the accuracies of individual learners. Hence an important parameter to see how well the ensembling works.

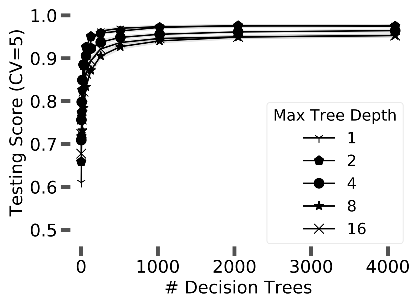
3.8.2 Number of Estimators

The number of weak learners that will be used in the algorithm.

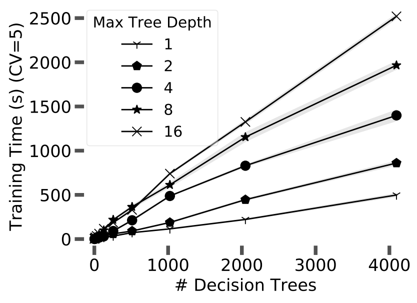
3.8.3 Analysis

Adaboost is incredibly good on both data sets and isn't affected by the boundary. It also generalizes very quickly and needs fewer points to achieve greater accuracies.

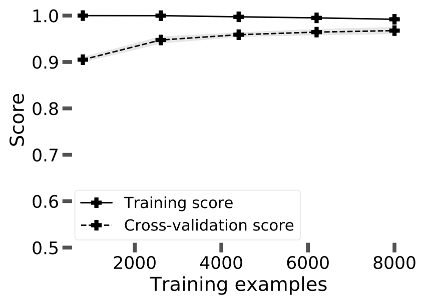
As can be seen increasing the depth of decision tree is a bad idea. It not only increases the training time very much but the lower depth trees outperform deeper trees. This is because the higher depth ones will be overfitting the sample they are showed, decreasing the overall accuracy. In lieu of both time and accuracy keeping lower depth(consequently weaker learners) helps the ensembling. The striking point being that it is unaffected by the kind of data it is acting on. Both data sets perform very well with depth 1 trees.



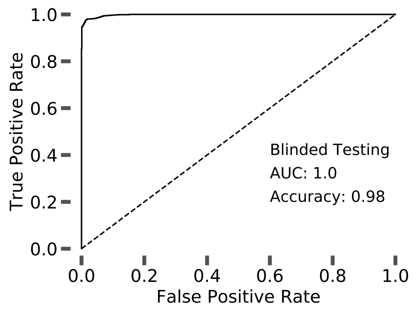
(a) Hyper Parameter Search



(b) Tranining Times for Params

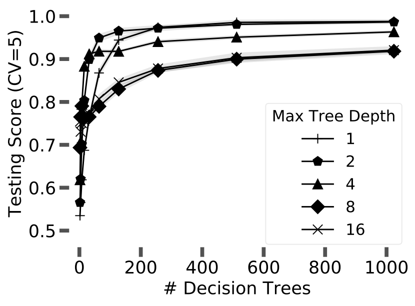


(c) Learning Curve

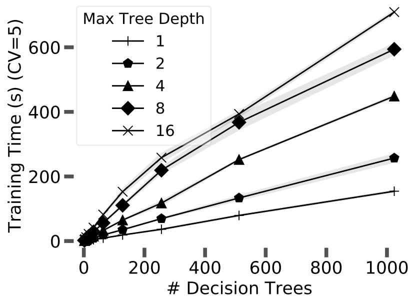


(d) ROC Curve

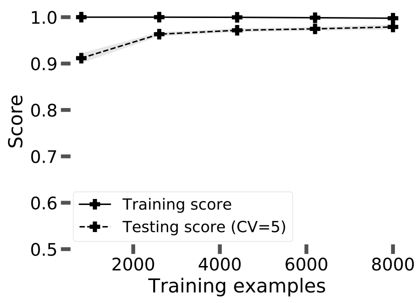
Figure 11: ABC on Linearly Seperable Data



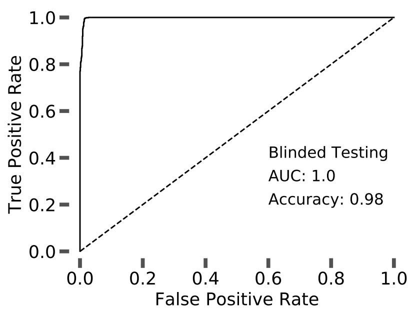
(a) Hyper Parameter Search



(b) Tranining Times for Params



(c) Learning Curve



(d) ROC Curve

Figure 12: ABC on Linearly Inseperable Data

Increasing the number of trees has an exponential effect in the beginning and then plateaus as they cannot do any better than the training set, while also being able to generalize it for the testing set.

So the best way to use Adaboost is to use weak classifiers and many of them.

3.9 KNNC

The K-nearest neighbor algorithm will try to justify that a given point behaves as its neighbors do. It considers a local neighborhood and looks at majority class in the neighborhood to predict the outcome of the current point.

3.9.1 Distance Metric

In considering a local neighborhood it is important to define a metric for proximity. There are various ways to measure distance and the Minkowski metric is one of them.

$$distance(A, B) = (\sum_{i=1}^{D} |A_i - B_i|^p)^{\frac{1}{p}}$$

By varying p you get different distances, for p=1 it is taxi cab distance, p=2 it is eucledian, etc

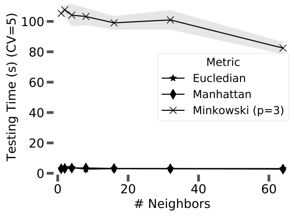
3.9.2 Number of Neighbors

Based on the distance metric, during testing the algorithm computes the closest N neighbors and puts them up for a vote. This value of N can be varied.

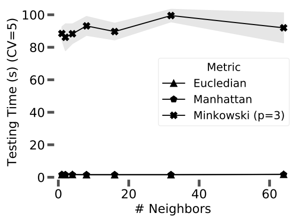
3.9.3 Analysis

KNN performs poorly on both the data sets, merely better than random for the second one. This might be due to the points on the boundaries. For the points near the boundary, any neighborhood chosen at whatever distance, it is always very likely that both classes are present in equal proportions making it hard to guess the correct class of the point. Hence the poor performance. This is also clear with the plateauing accuracy with increase in the number of neighbors. As the neighborhood size increases the equal proportion mishap is even more plausible.

There seems to be no effect of the metric chosen. Maybe because of the density and uniform distribution of data.
Note that there is no *training* for KNN. Everything happens during testing hence it is better to look at the testing times.



(a) Linearly Seperable



(b) Linearly Inseperable

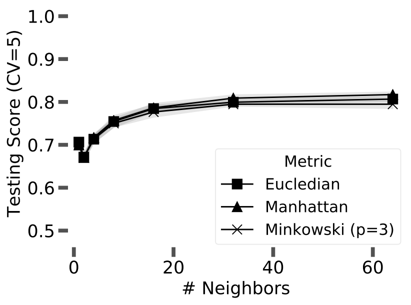
Figure 13: Testing Times for Params

Computing the polynomial metric seems to take a lot of time as opposed to the others. Maybe because Manhattan and eucledian are highly optimized metrics in the library.

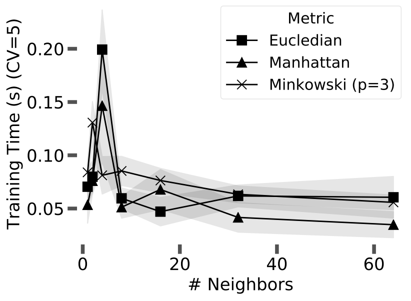
When computing the nearest neighbors we can do some preprocessing during training to store the datapoints in a efficient manner. I left it to default in the sklearn library which automatically chooses the preprocessing algorithm based on other hyper params and data from *'ball tree', 'kd tree', 'brute'*. These optimizations changing might be the reason for the peak in training time at one spot.

The learning curve suggests that knn generalizes quickly about what it can and any new data cannot make it better.

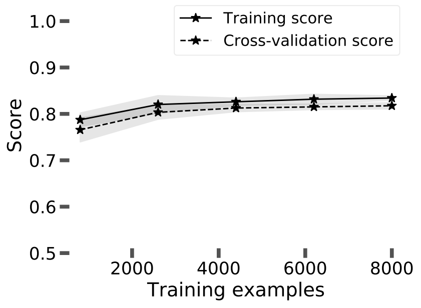
Such a low AUC would mean the hypothesis of KNN is void.



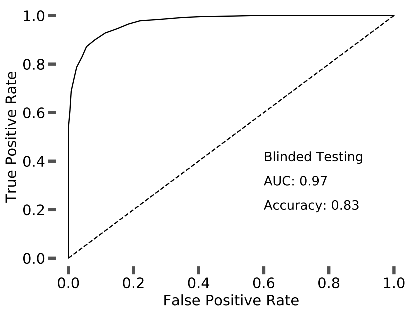
(a) Hyper Parameter Search



(b) Tranining Times for Params

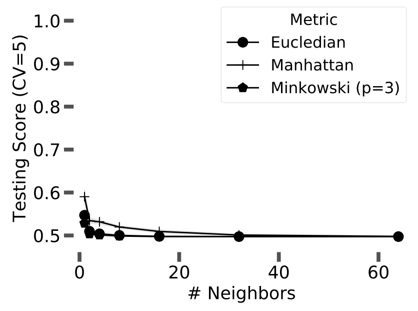


(c) Learning Curve

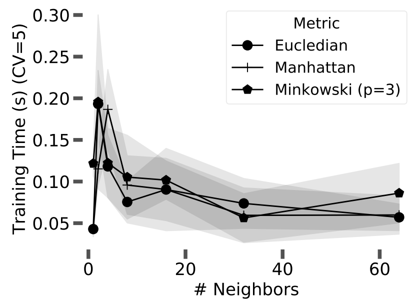


(d) ROC Curve

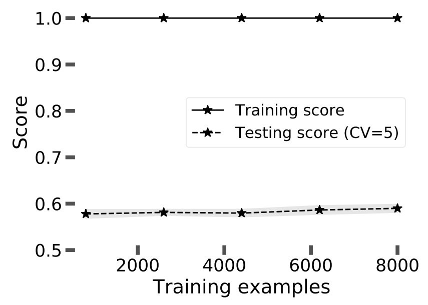
Figure 14: KNNC on Linearly Seperable Data



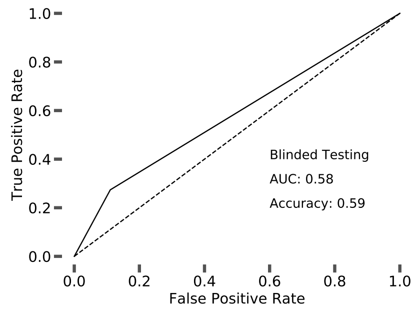
(a) Hyper Parameter Search



(b) Tranining Times for Params



(c) Learning Curve



(d) ROC Curve

Figure 15: KNNC on Linearly Inseperable Data

3.10 ANNC

For artificial neural networks I decided to stick to simple 3 layered network. I decided to cap the number of iteration on a very high end and left it to the algorithms to stop when the improvement over iterations is very small. The affect of number of iterations is captured via proxy of the accuracy plots and training time plot combined.

3.10.1 Solver

In order to optimize the weights of the network various optimizers can be used. Among them due to direct library support we here choose among "SGD", "LBFGS" and "ADAM". LBFGS method is faster than the other two for small datasets. [2]

3.10.2 Hidden Layer Size

By introducing more nodes into the hidden layer we can estimate more complex boundaries than a simple linear perceptron. In theory a single hidden layer, with enough number of nodes, should be able to approximate any function. Hence I thought it would be interesting to see how increasing the number of nodes in the single hidden layer increases the model complexity and its ability to generalize.

3.10.3 Analysis

First thoughts are that neural networks are slow. Looking at the time taken for training neural networks when compared to the other algorithms we notice they are on a different scale. But then again the choice of optimizer plays an important role.

On dataset 2: LBFGS and Adam are very fast compared to SGD even when the number of nodes in hidden layer is very large. They tend to find the minima quicker in fewer iterations. SGD maybe stuck in a local minima leading to a lower testing accuracy. However the variance in the accuracy suggests sometimes it crawls down the hill.

In terms of hidden layer size, the peaking begins to occur when the number of nodes is greater than 32 suggesting that at this model complexity the network is able to capture the non linearity and after that beyond 64 nodes any addition only leads to minor improvements (and extra time). Increasing the model complexity didn't affect the accuracy much later on.

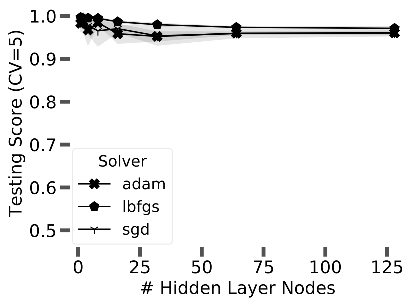
This tells us that when building models it might be helpful to have the hidden layer twice as long as many features.

On dataset 1 however increasing the model complexity decreases the accuracy. This is because for the linearly seperable data a simple perceptron would have been enough. By adding more nodes we are trying to fit higher degree functions leading to overfitting.

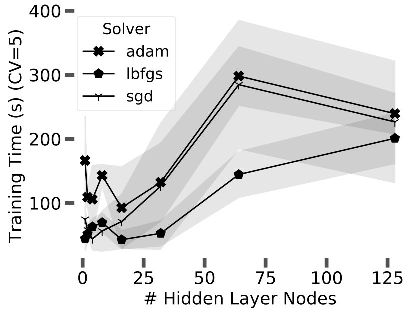
In fact with so many nodes the network can almost *remember* every training point it has seen.

On both datasets LBFGS performed the best considering both time and accuracy.

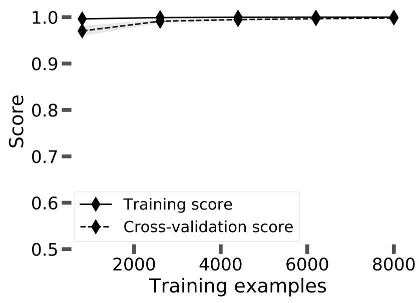
The learning curves suggest that while the simpler model in dataset 1 could easily generalize the hyperplane, more inseperable data could benefit the algorithm's accuracy on dataset 2.



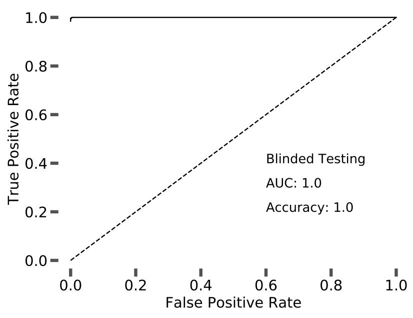
(a) Hyper Parameter Search



(b) Tranining Times for Params

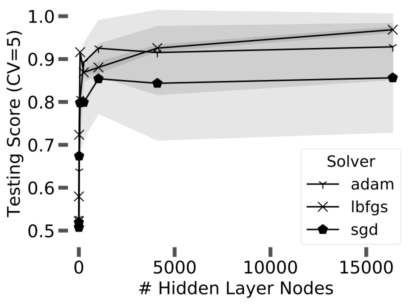


(c) Learning Curve

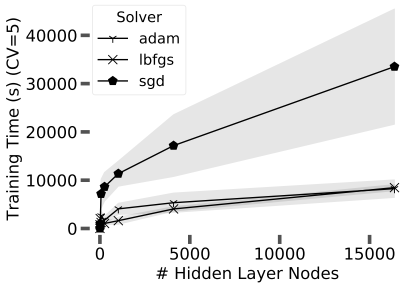


(d) ROC Curve

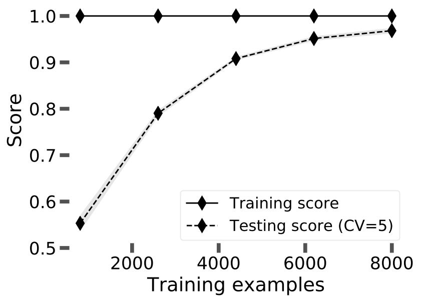
Figure 16: ANNC on Linearly Seperable Data



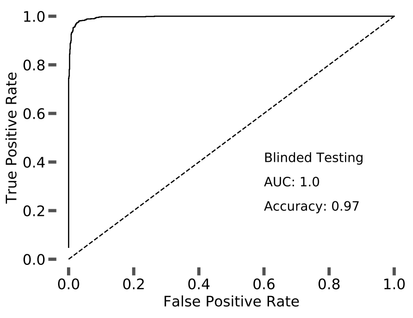
(a) Hyper Parameter Search



(b) Tranining Times for Params



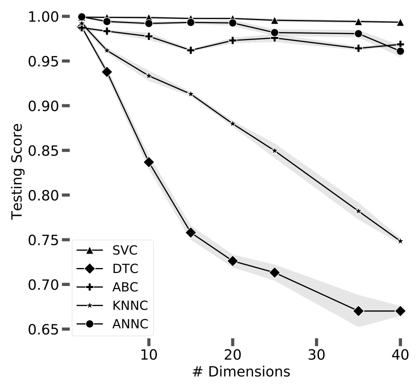
(c) Learning Curve



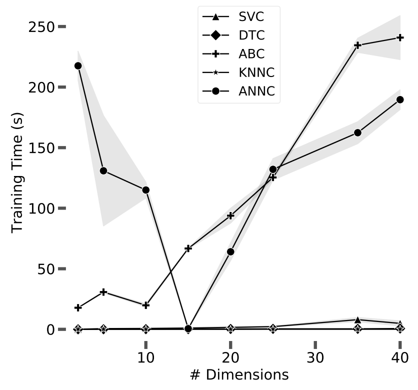
(d) ROC Curve

Figure 17: ANNC on Linearly Inseperable Data

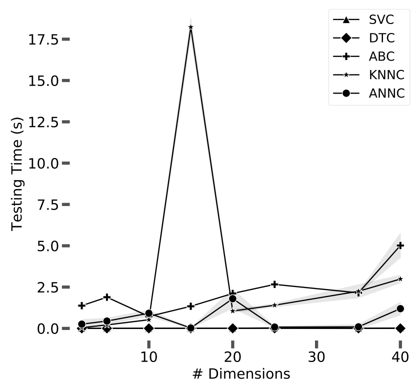
4 EFFECT OF DIMENSIONS



(a) Testing Accuracy

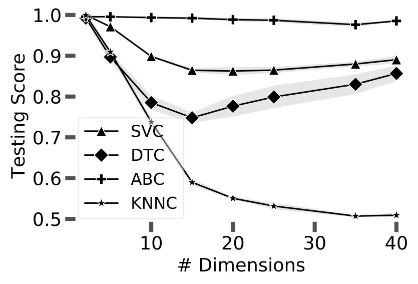


(b) Tranining Time

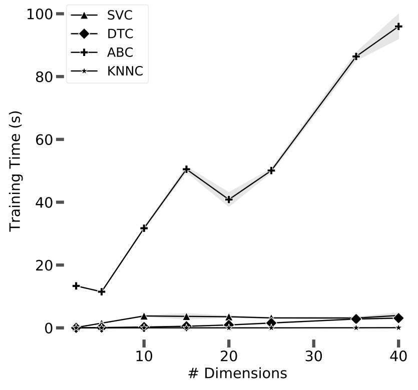


(c) Testing Time

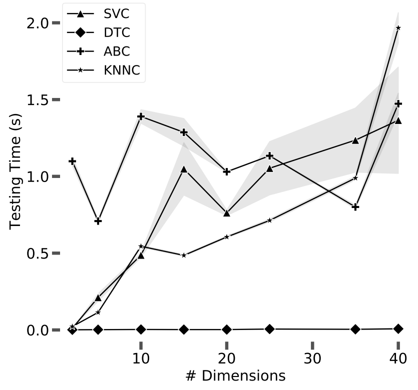
Figure 18: Effect of number of dimensions on Linearly Seperable Data



(a) Testing Accuracy



(b) Tranining Time



(c) Testing Time

Figure 19: Effect of number of dimensions on Linearly Inseperable Data

Since we generating our data it is easy to change the problem and visualize how it affects the performance. To get these metrics since we are not interested in how the algorithms hyper parameters are affecting them, we ran a Randomized Search in the same hyper parameter space as the previous sections on a 5 fold cv. The search took too long for neural nets and I decided to cut it out off discussion. Note that for every dimension there will be different model with different hyperparameters and we are comparing the best the algorithm can do with the data on any given number of features.

Support vector machines, Decision trees and Adaboost classifier(due to its base estimator being a DT) seem to be unaffected by dimensionality in terms of training time required (Although it does increases slightly). On the other hand KNN and Artificial neural networks keep increasing in time. For ANN initially there seems to be a drop, maybe because of a complex model being chosen in cross validation. But the trend is that training times increase.

For accuracy, lower dimensions are of course better, but Boosting is unaffected in both data sets. Probably because ensembling relies on low depth decision trees that diregard dimensionality.

SVM also remains stable owing to the kernel functions which emulate infinite dimensions irrespective of the feature space of the data. But then again for other algorithms it tends to decrease.

Decision tree climbs up in accuracy after certain dimenions on linearly inseperable suggesting in higher dimensions it may be easier to approximate a hyper sphere boundary as hyper cube sides.

Testing time on the other hand is almost 0 for Decision trees as a tree traversal is fast irrespective of the dimensions. But the overall times for others are very small although they show an increasing trend as well.

The amount of training data required for an algorithm is roughly 2^D, which in our case of 10000 points translates to $D \approx 13$. These algorithms do have elbow points between 10-20 for the linearly inseperable data.

5 CONCLUSION

Table 1: On Linearly Seperable Data

Algorithm	Testing Score	Training Time (s)	Testing Time (s)
SVC	**0.998**	8.554	0.020
DTC	**0.7185**	0.839	0.004
ABC	0.98	**500.2**	0.217
KNNC	0.827	**0.029**	**2.969**
ANNC	**0.997**	0.857	**0.003**

Table 2: On Linearly Inseperable Data

Algorithm	Testing Score	Training Time (s)	Testing Time (s)
SVC	0.805	42.070	2.394
DTC	0.914	6.243	**0.003**
ABC	**0.985**	85.630	0.219
KNNC	**0.588**	**0.029**	**2.8**
ANNC	**0.987**	**1351.90**	0.026

Here we are looking at the best models performance on the with the accuracy on blinded test set, total training time for the model and total testing time.

The reason Adaboost classifier on first dataset took 500s is just because the model chosen has large number of trees. From our previous discussion there was only minimal improvement beyond a point, and we could have stopped there. However the grid search picked the absolute best scorer.

K-nearest neighbors performs good on linear set while fails on dataset 2, due to seperation boundary issues discussed earlier.

Combining all of these learnings we infer that neural networks are great however we might have to spend a lot of time to find the best model. Boosting according to me the best algorithm so far owing to accuracy and training time. We can achieve very high accuracies irrespective of the dimensions and data size in relatively low times.

REFERENCES

[1] "Adaboost tutorial." [Online]. Available: http://mccormickml.com/2013/12/13/adaboost-tutorial/
[2] S. Ruder, "An overview of gradient descent optimization algorithms," Nov 2018. [Online]. Available: http://ruder.io/optimizing-gradient-descent/