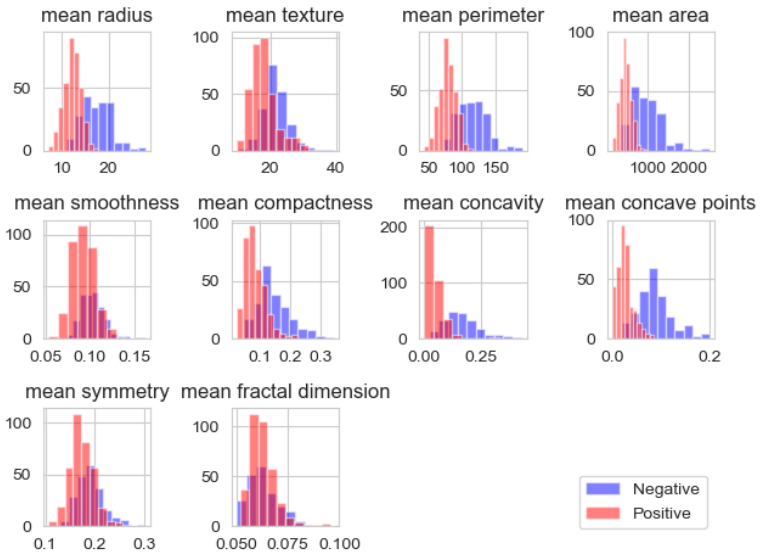
Introduction to Dataset

Interesting Dataset?

In order to compare the performance of different machine learning algorithms, two datasets, UCI's Breast Cancer Wisconsin Data Set and Caltech 101 Data Set, have been deliberately chosen. Even though both datasets deal with image classification, one of these datasets (UCI dataset) has been translated into categorical and numerical data, while the other stays in its raw pixel form. In addition, only the important features are included in the UCI dataset and has a binary classification output. This contrast with the Caltech dataset which has 101 categories output. In other words, this paper tries to compare the capacity of various supervised learning algorithm on recognizing noises on a "easy" (the feature extracted and less noisy binary UCI dataset) versus "hard" (the raw and multi-categorical Caltech dataset) problem.

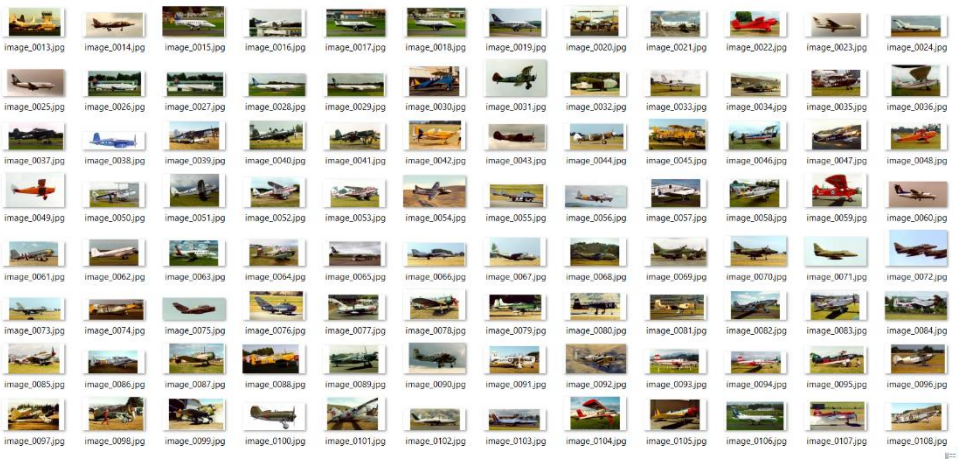
The Caltech 101 dataset has approximately 10k instances, while the UCI Breast Cancer Dataset contains 569 entries. In order to make the comparison, the Caltech 101 dataset has been reduced to 10 categories with approximately 1k data points. This is also done since the time to load and train algorithms have been too long to neglect, taking as much as 4 hours to run and train a single model.

Considering the number of categories for the Caltech dataset (around 100 images per category), this paper will assume it is almost fair to make the comparison for a multicategory versus binary data points. (It is not easy to find another dataset with the right number of instances.) On other hand, as the result unfold, if there is a disparaging difference between the performance, there would be a discussion on whether the number of training data could have affected the performance. Nonetheless, the difference in size would also enable a comparison between training and testing time for various algorithms.



The following diagram shows some of the features that was used in breast cancer classification with positive and negative labels. Note that many of them show a small p value and are statistically significant.

In contrast with the feature data from UCI, the Caltech dataset contains only images, and might in turn encode significantly more feature (information) than UCI dataset. This paper will conduct some basic preprocessing, explain in the next section, to extract the features from the Caltech data. The follow is some images from the UCI dataset without preprocessing.



A Note on Preprocessing and Grid Search (with Cross Validation)

To facilitate fair and comparable results, the same preprocessing steps are performed to the datasets. In particular, the images in the Caltech 101 dataset are resized, converted to grayscale (to make training faster), and feature extracted with histogram of oriented gradients (HOG). The hyperparameters for feature extraction are chosen by

referencing other online repositories that use the Caltech 101 dataset. In addition, the labels for the Caltech dataset are one hot encoded. On the other hand, the UCI dataset is normalized (excluding the labels.)

For each algorithm, some form of grid search is used to find the best and ensure that result is not biased from the lack of hyperparameter tuning. For most algorithms, the two most "important" hyperparameters are chosen and plotted to visualize performance. The arbitrary choice of these "important" hyperparameters will be explained for each algorithm. Also, two are chosen so it is easy to graph (performance versus grid parameter 1, and one plotted line for each grid parameter 2) and trains in a shorter amount of time as grid search runs an exhaustive search.

k-Nearest Neighbor

UCI Breast Cancer Dataset

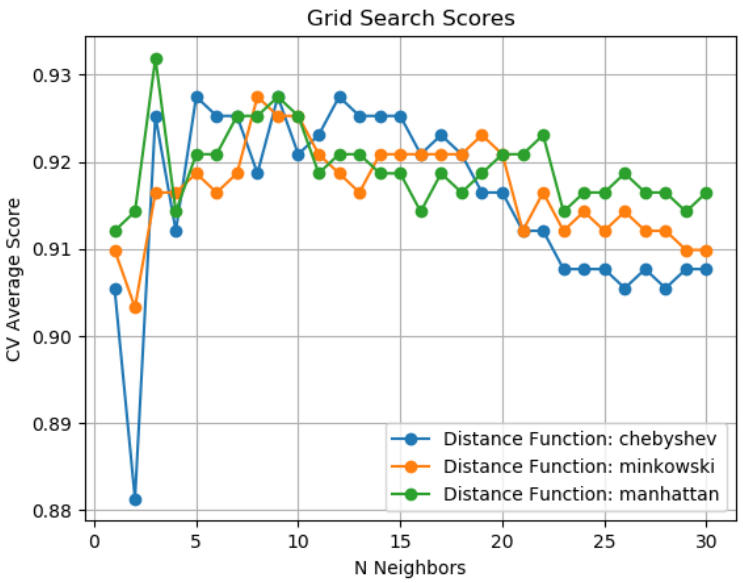
The Scikit-Learn framework supports some of the most common distance metrics, including metrics optimized for two-dimensional vector space, integer-value vector space, boolean-value vector space (where input data are in the form of boolean values, not the output), and real-value vector space which best fit the UCI dataset. These real value vector space distance metrics include: euclidean, manhattan, Chebyshev, and minkowski. Since Euclidean distance is only a special form of minkowski distance (when p = 2 for minkowski = sum(|x - y|^p)^(1/p)), and that the sklearn classifier would help tune the p value, Euclidean distance has been removed from the grid search.

The first grid search is evaluated by the cross validated average score with 10 folds and two parameters, the number of neighbors and distance function. The number of neighbors is perhaps the single most important hyperparameter, since kNN classifies the result based on the mean decisions of the closest k neighbors. If the value of k is too small, the classification result might be inaccurate due to outliners in the training set. On the other hand, if the value of k is too large, it would not be able to capture the feature difference for different labels. Such an example includes having a k value that is as large as the training set size, which would yield the same classification result and not be accurate.

The second parameter that this paper attempts to explore is the distance function, which is the metric to decide how to measure the closeness of each data points. Like mentioned above, the three different distance metrics evaluated are Chebyshev, minkowski, and manhattan, which are some of the most commonly used and best performing metrics. The grid search took 7.12 seconds which is relatively fast for over 30 k-value * 3 distance metrics = 900 different kNN models to evaluate on 10 folds cross validation. This happens since kNN uses instance base learning, also called lazy learning, meaning that it does not attempt to build a model to describe the relationship between the features and the classification result, but instead saves all the instances. This means it would take up spaces as much as the training data O(n) in space where n is number of instances, and O(n + k * log(n)) (or linear time to input data if k is constant) in terms of speed if we use a priority queue (heap) to rank the first k instances closest to the test instance. In other words, there isn't really a "training" time, and instead there is only a time needed to calculate at testing time.

The result of the grid search shows that manhattan distance has the best performance with 3 nearest neighbors. We were able to also get an accuracy of 92.11% when evaluating the best model against the test set.

- grid search took 7.12 seconds
- grid search accuracy on test set: 92.11%
- grid search best average validation score: 93.19%
- grid search best parameters: {'metric': 'manhattan', 'n_neighbors': 3}

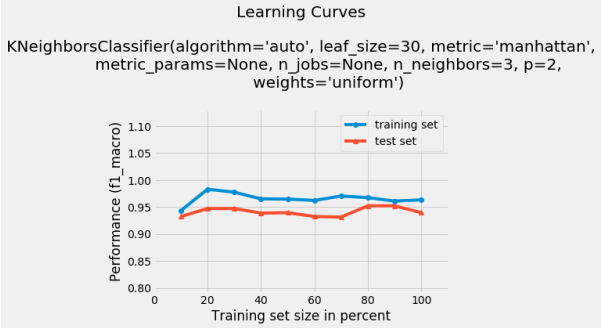


Another interesting discussion in class we had was how the distance should be weighted. The default method used in the previous grid search is uniform weight, which has no assigned weights for the distance functions. The other method is the distance weighted kNN which assigns a heavier weight to instances that are closer to the test instance. Using the best estimator's parameters from the previous result, the following result was obtained.

- grid search took 5.90 seconds
- grid search accuracy on test set: 92.11%
- grid search best parameters: {'n_neighbors': 3, 'weights': 'uniform'}
- grid search best average validation score: 93.19%

It can be quickly realized that both uniform and distance have the same cross validated mean scores at their peeks, which is, at another glance, same as the best configuration in the previous grid search. Another interesting realization obvious from the graph is that the distance weighted function consistently outperforms the uniform weight function. This could be caused by the fact that a smaller number of neighbors consistently outperforms larger k (observe the decreasing trend) so that the distance weighted function behaves like there is a smaller k.

Both grid search performed above uses 20% testing data with 80% training data. By examining the learning curve as a function of training set size and scored by f1 micro, it is obvious to see that there isn't a lot to learn from this dataset when using kNN. The performance of the testing set does not show a steady increase, but instead already has a high f1 score when the training size is only at 10%.

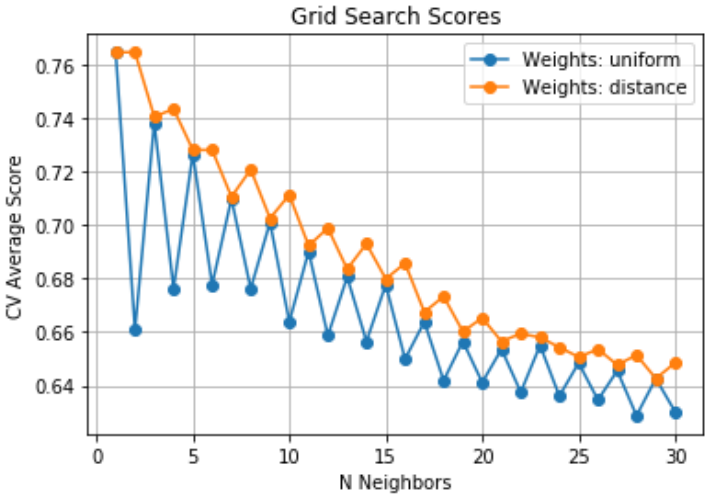


Caltech 101 Image Dataset

In hindsight, it would have been probably not a good idea to use kNN to classify image data without much preprocessing. There are simply too many noises in the data even after feature extraction. This leads to another issue with many machine learning algorithms, and a major issue when using kNN, the curse of dimensionality. The k nearest neighbor algorithm is susceptible to the number of dimensions because it treats every dimension (feature) equally, and as the number of features increase, kNN will be computing distances to features that are or contains a lot of noises (does not actually relate to the classification result) or covariate with other features (giving too much weight to a single feature after corrected for orthogonality). Since most kNN algorithms lack the ability to perform dimension reduction either for features that are not correlated with the classification result, or features that have a high covariance between each other. It would be better to use Independent Component Analysis (ICA) and especially Principal Component Analysis (PCA) before kNN. The following are the same grid search that was performed with the UCI dataset.

- grid search took 634.93 seconds
- grid search accuracy on test set: 77.26%
- grid search best parameters: {'metric': 'manhattan', 'n_neighbors': 1}
- grid search best average validation score: 76.44%

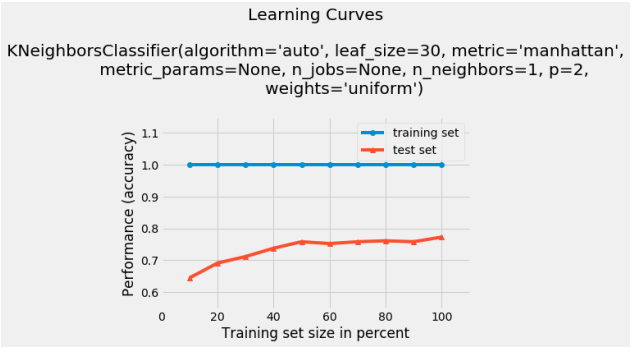
Notice the interesting pattern as the number of neighbors increase. First, the cross validated average score steadily decreases as K increases. This might be due to poor feature extraction that leads to the model to learn too much from the noise when K increases. Second, notice that at every even number of neighbors, the performance drops significantly. To understand this, the paper examines the way that the sklearn library conducts its majority vote. Since kNN relies on a majority vote of the k nearest neighbors, most k values are usually chosen to be an odd number to avoid the issue of getting equal numbers of votes. A quick examination of the source code of sklearn reveals that it uses the mode method from scipy.stats, which by default chooses the smallest value. This could have potentially screwed the result for the classification.



- grid search took 431.17 seconds
- grid search accuracy on test set: 77.26%
- grid search best parameters: {'n_neighbors': 1, 'weights': 'uniform'}
- grid search best average validation score: 76.44%

With this in mind, running the grid search for weights vs k as a function of performance reveals that this issue might be fixed when weighting the distance. (So the closer neighbors take a larger weight.)

The learning curve on the Caltech dataset shows a stark difference. The test set shows a steady growth of testing accuracy that might be due to the complex feature of image data that can be learned as the number of training example increases.



Decision Trees

UCI Breast Cancer Dataset

Decision Tree predicts the target label by learning the knowledge represented by a tree. It has the advantage of being visualized, or being a white box method, and the explanation for the prediction can be understood through Boolean logic. Decision tree can also overfit by creating an overly complex tree, so pruning is recommended. Also, since learning the optimal decision tree is known to be a NP-Complete task, most implementation uses some greedy algorithm with heuristics that finds locally optimal decisions. These algorithm fails to deliver the global optimal tree, thus multiple trees are usually trained together to mitigate this issue, also called ensemble. In addition, since decision tree learn via information gain or Gini score, it is likely to create a bias tree if some classes dominate.

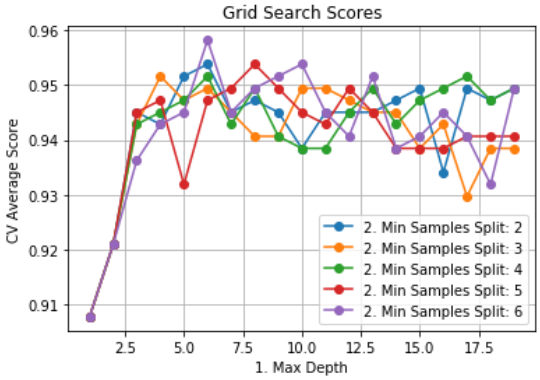
In order to prevent overfitting, this paper would pre-prune the tree with the maximum depth for each tree. On the other hand, this paper would also explore both entropy (information gain with ID3) and gini (Gini impurity with CART) as a measure of the quality of the split. When constructing the decision tree, ID3 (iterative dichotomiser 3) uses the highest information gain to determine the attribute that best classifies (and separates) the training data at every node. Information gain is determined by the entropy, which looks at the labels of the instances that is classified. For example, if the classified examples are all a single class, then the entropy is zero, and if there is an equal mix of all classes, then entropy has a value of 1. CART is similar to ID3 except it only construct binary splits (makes a binary tree) on only one attribute, which contrast ID3 which can test on multiple attribute at the same time and perform multiple splits. CART splits by searching the one attribute that produces the highest Gini score, which is the attribute that splits the most target values.

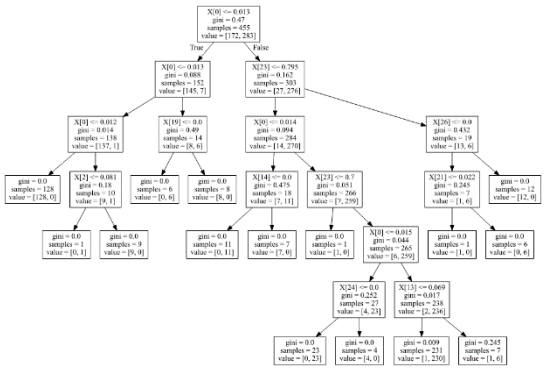
Like mentioned, decision tree can easily overfit when the size of the tree grows too large. To compensate for the bias-variance tradeoff, we use the maximum depth of the tree and minimum sample split to also observe the learning curve. The minimum sample split sets a minimum number of samples needed at each node to perform a split. Both of these prevent the tree from growing until there are only pure classes, which learns too much about the

noises of the dataset. To see the learning curve, the following is plotted to see the max depth against the cross validated score.

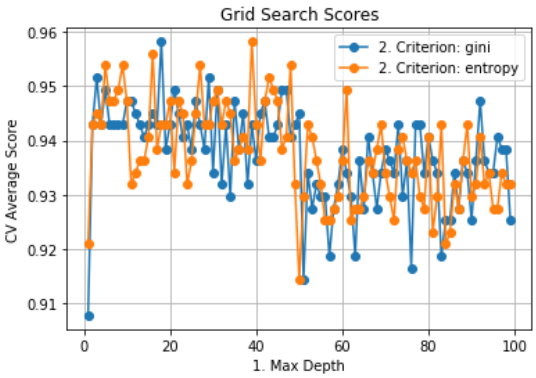
To avoid the local optimal decision tree problem, this paper uses random forest classifier which builds decision trees on a subset of the training data and using the average decision as the final prediction.

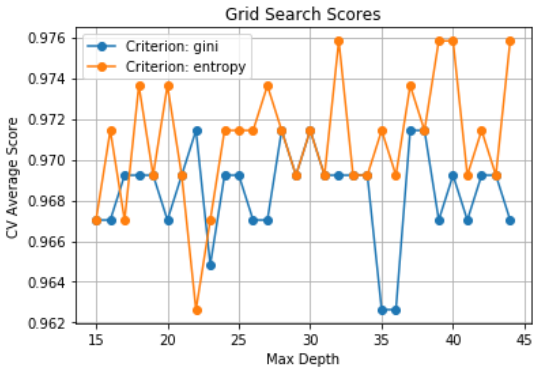
The following is a grid search of random forest classifier on gini index, and the visualized form of the best individual. It can observe that there is a general trend for the cross validated score to drop as the max depth increases. On the other hand, it is hard to see if there is a general trend for the effect of minimum samples split on the score even when increasing the range of the grid search. (Not including the graph since space is limited.)





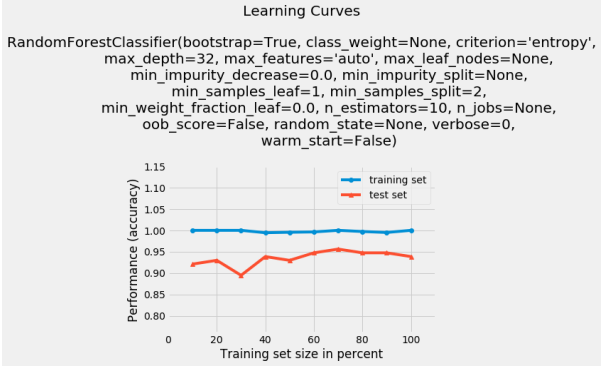
With a larger parameter search for max depth going from 1 to 100, there is a more obvious trend of lowering cross validation score. Also, instead of minimum samples split, the plot is replaced by criterion on the quality of split, either using gini or entropy score. At a close up (range from 15 to 45), there isn't a huge difference between the effects of either score, perhaps due to the ineffectiveness of numerical feature data as features instead of discrete labels.





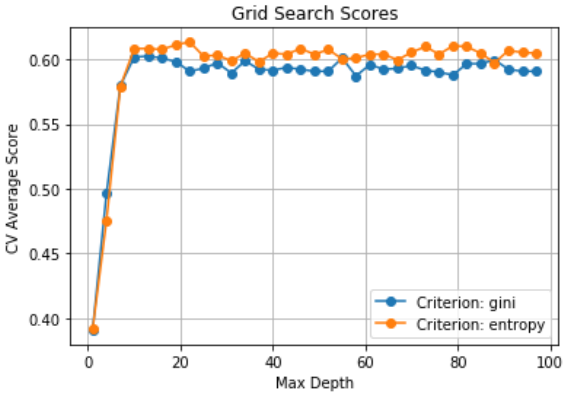
- **For the grid search with max depth from 15 to 45**
- grid search took 59.73 seconds
- grid search accuracy on test set: 93.86%
- grid search best parameters: {'criterion': 'entropy', 'max_depth': 32}
- grid search best average validation score: 97.58%

Using the best individual from the grid search, the learning curve shows that as the training size increase, the testing accuracy does not really drop and instead shows an upward trend. This means that as the best individual pre-prune the max depth to 32, it hardly over-train on training data. Nevertheless, it is also easy to tell that there is still generally a gap between the training and testing accuracy that is hard to close up. This could be the inherent bias that using decision trees can introduce.



Caltech 101 Image Dataset

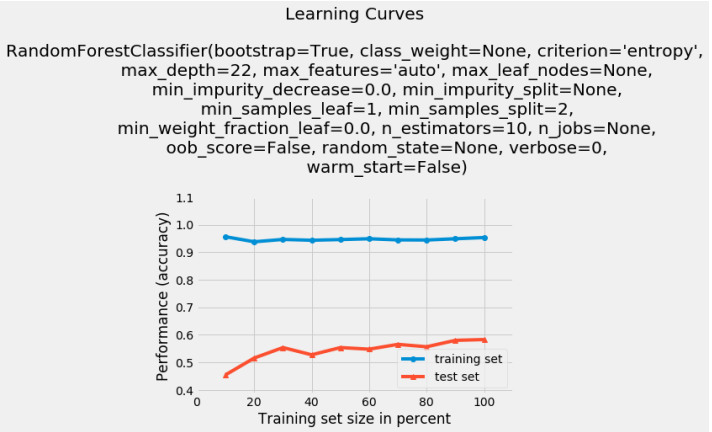
Like mentioned, the image dataset has not been heavily preprocessed, and the raw inputs can contain a lot of noises. This contrast the feature data from UCI which are preprocessed and does not contain as much noises. Since decision trees depends heavily on the specific feature of the data set and has already been showed to be vulnerable to noises, this task to classify images becomes a lot harder. Although the effect might not be obvious because classifying image is a hard task, decision trees are also inherently multiclass. This means that there will not be performance impact when predicting the binary UCI data and the multiclass Caltech dataset.



- grid search took 2438.69 seconds
- grid search accuracy on test set: 58.89%
- grid search best parameters: {'criterion': 'entropy', 'max_depth': 22}
- grid search best average validation score: 61.34%

Like shown, the average score of the random forest is bounded around 60% and 70%, which is not as high as the UCI dataset. However, considering that this task is a multi-label classification, and that the cross validated score is based on the instances classified correctly, this score shows that the decision trees are performing a lot better than just guessing.

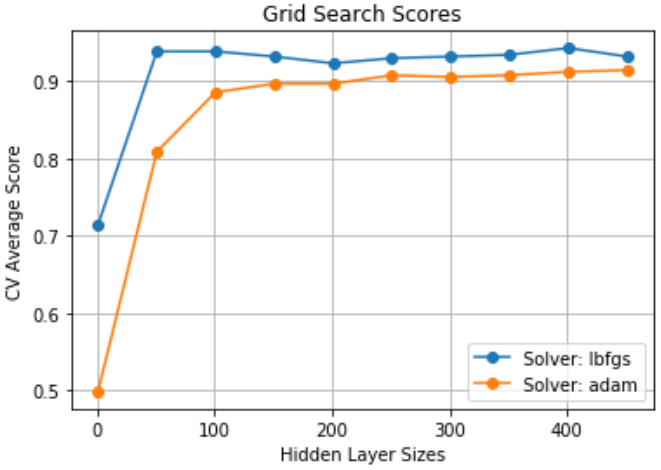
Through the learning curve, we can see the same overfit problem as the UCI dataset. This time since there are more noises in the dataset, it can be seen that the gap is a lot larger than the UCI dataset. Note that the training set accuracy here is different from the average cross validated score when conducting grid search. This training accuracy is the accuracy of the actual training data used to train the tree and not a withheld cross validation fold. It is obvious that the tree learns too much from the noises, and thus performs really well on the training data. Also, the graph demonstrates the gradual increase of testing accuracy as the training set size increases. This implies that random forest might perform better when the amount of data increases and that the Caltech dataset might be too small for a task as hard as image classification.



Neural Network

UCI Breast Cancer Dataset

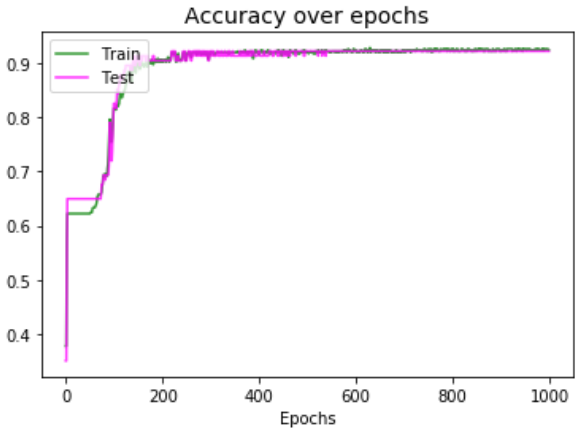
Neural Networks are showed to be able to learn any non-linear function as long as there is at least one fully connected layer (and enough nodes and a non-linearity, of course.) The Scikit-learn neural network uses backpropagation with some form of gradient descent algorithm. In terms of neural network, this means that it turns the parameters of individual connected nodes by the performance of some loss function and by finding how to get to the local optimal value. For our classification task at hand, it uses cross-entropy loss function and provides a vector of probability estimates per sample. Cross-entropy loss measures the classification model and increases as the predicted probability diverge from the actual value. The multi-layer perceptron from Scikit-learn also inherently supports multi-label classification. Our comparison will include the number of hidden layers and the optimizer used for tuning the parameters on the nodes. Even though one fully connected hidden layer with nonlinear activation function is sufficient to model any non-linear function, with more hidden layers, it can be showed that the neural network can actually learn the function “easier”. (Like CNN with representational learning.)

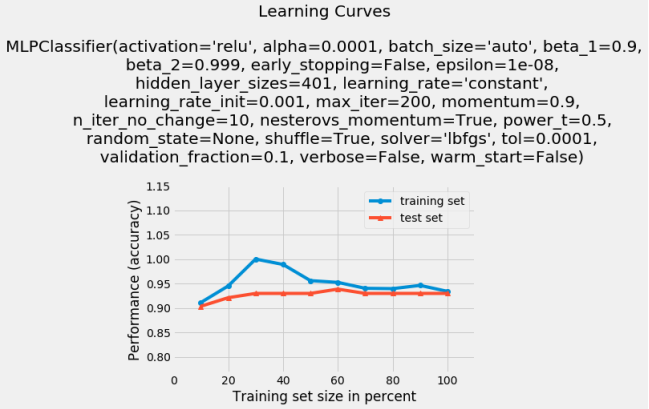


- grid search took 124.53 seconds
- grid search accuracy on test set: 92.11%
- grid search best parameters: {'hidden_layer_sizes': 401, 'solver': 'lbfgs'}
- grid search best average validation score: 94.29%

The number of neurons in the hidden layer is also a very important, since too few neurons will result in under fitting and fails to detect the signal in the data set. On the other hand, too many neurons may result in overfitting. In order to make the training time reasonable, this paper will only explore a single hidden layer while varying the number of neurons.

The grid search shows that average cross validated score stabilizes at around 100 neurons for both Adam Optimizer and the L-BFGS Optimizer. In addition, it is also interesting to see that the L-BFGS Optimizer consistently performs better than the Adam Optimizer. The L-BFGS Optimizer estimates the curvature of the parameter space through an approximation of the Hessian, which means updating the Hessian at every step. This means that as the input space grows, the cost adds up. But with a small dataset from UCI, this is relatively fast and performs well above Adam. Adam Optimizer, on the other hand, constructs a diagonal Hessian at every step using only the past gradients, which is less cost than the L-BFGS Optimizer, but only perform well with more data and larger input space.

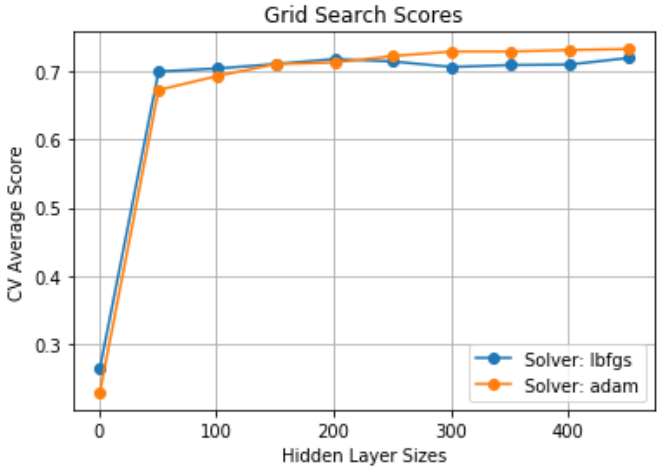




Both the learning curve from training set size and from number of epochs shows a general increase with more training. This is surprising since the author expected the accuracy of testing set to drop as the number of epochs increases. This might be the fact that there is only a single hidden layer, making it harder to over-train and overfit on the dataset.

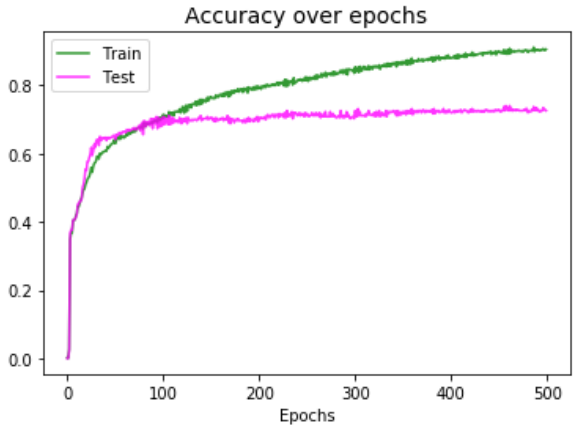
Caltech 101 Image Dataset

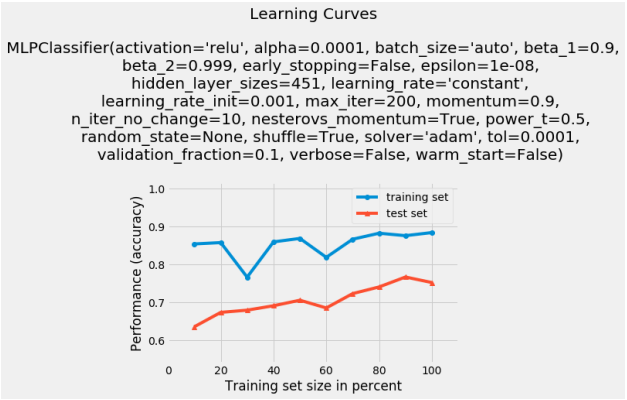
Like previously mentioned, Adam optimizer performs better when the input data space increases. This is demonstrated below when we run both optimizers on different numbers of neurons. The Adam Optimizer eventually takes over L-BFGS as the layer size increases. This is most likely since as the number of neurons increases, the more signal the Adam Optimizer can capture, making it a more suitable optimizer when the input signal is as complex as image data. We can also observe that Adam Optimizer still has a trend of gradual increase in cross validation score as the number of neurons increase.



- grid search took 863.26 seconds
- grid search accuracy on test set: 76.38%
- grid search best parameters: {'hidden_layer_sizes': 451, 'solver': 'adam'}
- grid search best average validation score: 73.16%

Just like expected, the accuracy over epochs for the testing set also stabilizes around 70% accuracy at around 100 epochs. From the 100th epochs on onward, the gap between the accuracy of training data and the testing data increases, meaning that the classifier is over-trained and overfitted. Since the image input data space is very complex, even though the learning curve as a function of training set size does show a shortening gap, but there is not enough data to show an overfitted (stabilizing or even decreasing testing set accuracy) trend. This agrees with the intuition that the input space is so complex that there is not enough data supplied to even overfit the classifier unless running more than 100 epochs on the same training data. On the other hand, this result also shows that neural network of one hidden layer is also able to capture a lot of information even though the input space is huge. (Input space refers to the complexity of the input dimensions; the author is referring to it as if it is the problem space.)

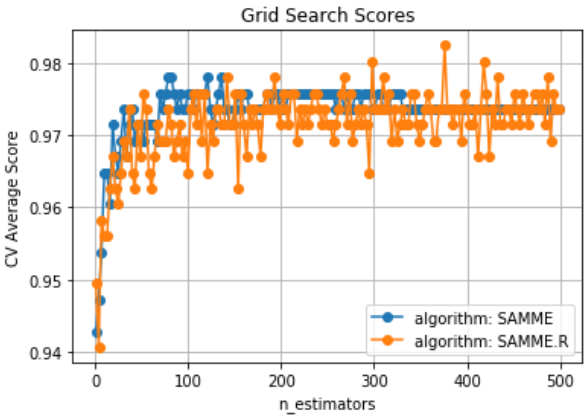




Boosting

UCI Breast Cancer Dataset

Boosting combines weak learners to form a strong rule that can together form a strong prediction. The two main boosting algorithms that are used today include AdaBoosting (Adaptive Boosting) and Gradient Tree Boosting. AdaBoost fits a sequence of weak learners on training data that are weighted differently. Initially, the weak learner is trained on training data with equal weight to each observation. If the weak learner fails to provide the correct output, AdaBoost then gives higher weight to those observations that have been predicted incorrectly. Instead of giving more weights to the wrong classification, Gradient Boosting create more models that minimizes the loss function. In essence, this builds more base learners that is correlated with the negative gradient of the loss function, finding the optimal estimate of the true underlying function.



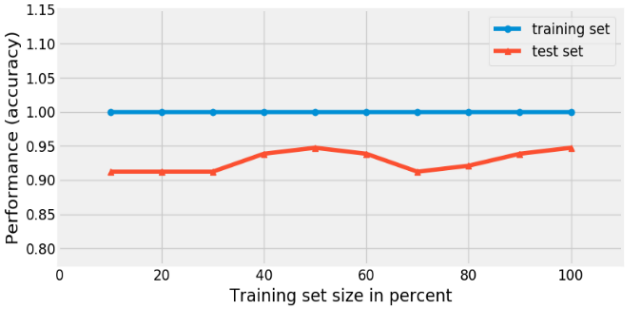
- **AdaBoost with pre-pruned decision tree of max depth = 3**
- grid search took 2550.04 sec (**30 mins!**)
- grid search accuracy on test set: 97.37%
- grid search best parameters: {'algorithm': 'SAMME.R', 'n_estimators': 376}
- grid search best average validation score: 98.24%

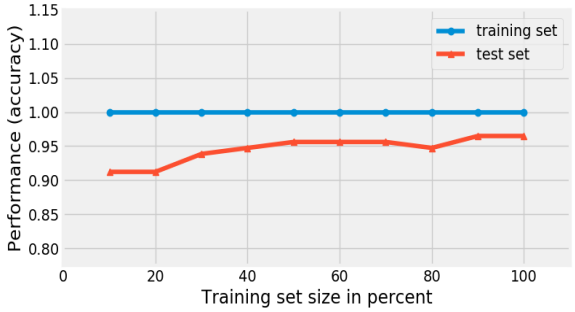
- **Gradient Boost with pre-pruned decision tree of max depth = 3**
- grid search took 20.19 seconds
- grid search accuracy on test set: 94.74%
- grid search best parameters: {'loss': 'deviance', 'n_estimators': 81}
- grid search best average validation score: 97.58%

Both algorithms show considerable improvements to just random forest alone. The algorithms used in AdaBoost, SAMME and SAMME.R are both based on the same principal outlined above. The difference lies on how specifically the models are updated, with SAMME using only the wrongful classification and SAMME.R using a probability estimate giving more weights to the wrongful classification. SAMME.R usually converge faster with this in mind. Like previously mentioned, Gradient Boost is based on optimizing via the loss function. The two different loss function that can be applied are deviance (or logistic loss function) and exponential loss function.

Tuning the hyper-parameters from AdaBoost and Gradient Boost did not change the convergence speed or accuracy

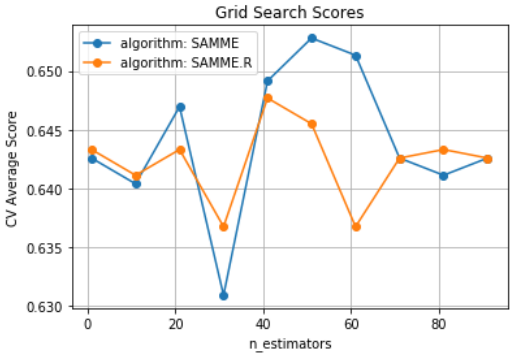
by too much (the orange and the blue lines are not far apart). However, AdaBoost does produce the best performance so far with 97.37% accuracy on the test set. A note here is that AdaBoost with decision tree of max depth of 1 was able to train significantly faster than AdaBoost with max depth of 3 as depicted above. (Max depth of 1 is not shown.) On the other hand, Gradient Boost can run relatively fast running approximately 300 seconds when also exhaustive searching for number of estimators from 1 to 500. (Also not shown.) In other words, AdaBoost took **9 times** the time it took for Gradient Boost to train when pre-pruned to have max depth of 3. The default in Scikit-Learn for AdaBoost recommends decision trees of max depth of 1, which is a weak learner, but does not perform as well as the results shown above with max depth of 3. This is another hyperparameter (perhaps a costly one) to tune; a question of how "weak" the base leaners must be. (Which could be dependent on the complexity of the input space, e.g. How hard the problem is.)

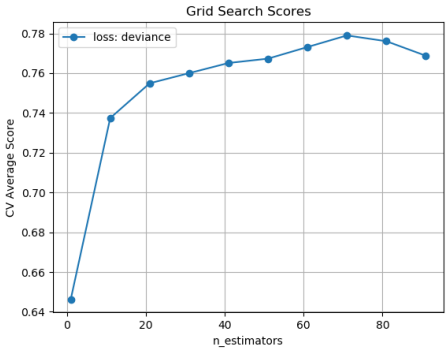




AdaBoost (left figure) and Gradient Boost (right figure) both shows a gradual increase in test set accuracy as the training size increase. This agrees with the intuition since as the classifier is exposed to more data, all the weak learners have to opportunity to keep learning from the wrongful classification. Unlike other learning curves where the test accuracy reaches a threshold, Boosting has the potential to keep learning from more data until eventually it

Caltech 101 Image Dataset



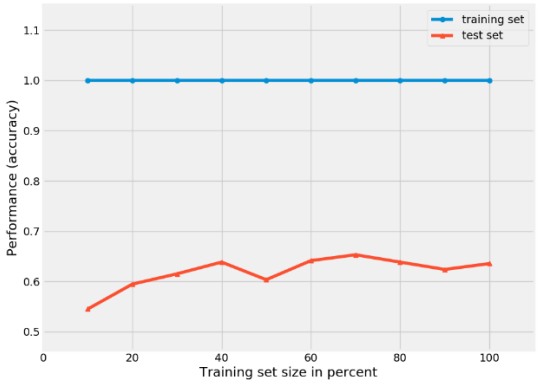


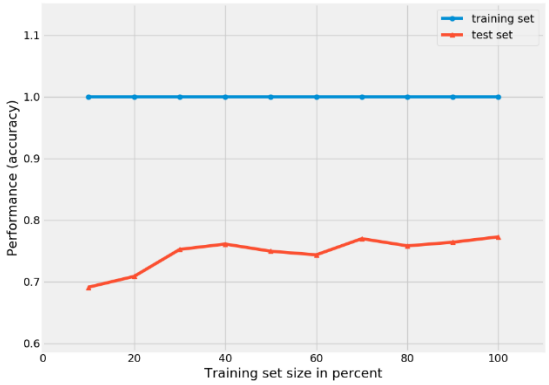
The 'exponential' loss function that was built into Gradient Boost does not innately classify multiclass data (only binary), so it was omitted as shown above.

- **AdaBoost with pre-pruned decision tree of max depth = 1**
- grid search took 63.72 seconds
- grid search accuracy on test set: 62.39%
- grid search best parameters: {'algorithm': 'SAMME', 'n_estimators': 51}
- grid search best average validation score: 65.28%

- **Gradient Boost with pre-pruned decision tree of max depth = 3**
- grid search took 1291.66 seconds
- grid search accuracy on test set: 77.26%
- grid search best parameters: {'loss': 'deviance', 'n_estimators': 71}
- grid search best average validation score: 77.90%

Like previously discussed, decision trees are prone to imbalanced classes, since it might produce trees that are bias and skewed for some predictions. To solve this issue, the paper used random forest classification outline above which builds trees on randomly selected training data. Boosting helps aid this by weighting the error term (or the wrongly classified instances) so Boosting is less prone to the bias introduced by imbalance classes. This is further backed by the high score produced for the learning curve.

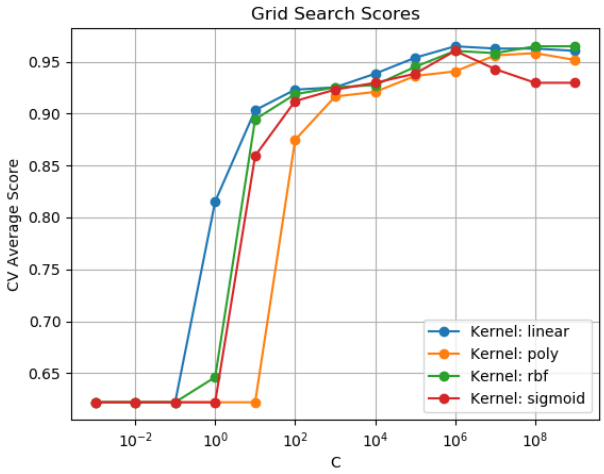




Support Vector Machine

UCI Breast Cancer Dataset

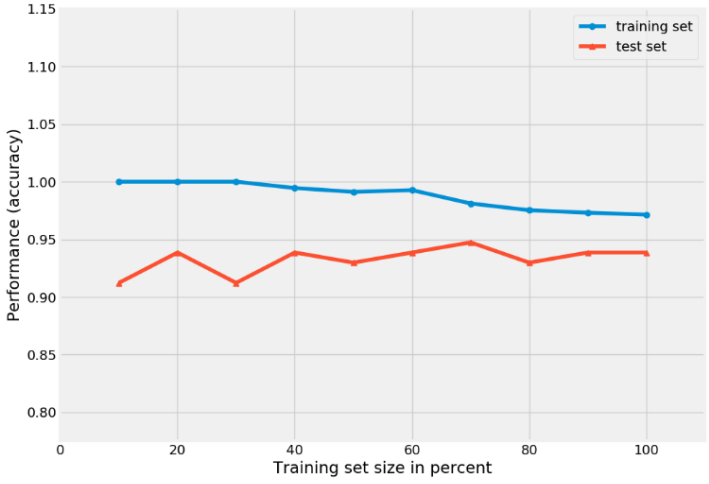
SVM classifies targets by finding a best fit line that can separate the different labels; those instances closest to the lines are the support vectors that helps bound the line. SVM does do by making the line as far away as possible from the support vectors, enlarging the margin. SVM is based on the idea of projecting the data points into a higher dimension (even an infinite feature dimension) and plot the linear hyperplane to separate the different labels, which could look nonlinear when returned to the original dimension. The way SVM project the data depends on the kernels used. Scikit-Learn supports some of the most common kernels, including linear, ploy, rbf (radial basis function), and sigmoid. All these kernels provide a shortcut for the SVM to calculate a similarity measure without actually projecting the complex decision boundary which are computationally expensive. Since SVM "pseudo" projects the features onto a higher dimension, it is obviously good at handling data with high dimensional spaces.



- grid search took 29.97 seconds
- grid search accuracy on test set: 93.86%
- grid search best parameters: {'C': 1000000, 'kernel': 'linear'}
- grid search best average validation score: 96.48%

The C term behaves like a regularization term. A smaller C will promote a larger margin for the decision function, at the cost of training accuracy. A larger C will produce decision function with smaller margin that might cause overfitting.

From the scores above produced by various kernels, linear kernel with a C value of 10^6 produces the best result. The C value is larger than what the author is usually using, meaning that there is a lot of signals that can be learned from this dataset.

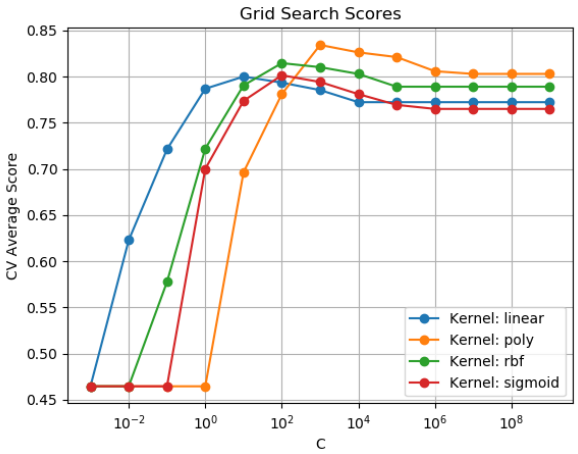


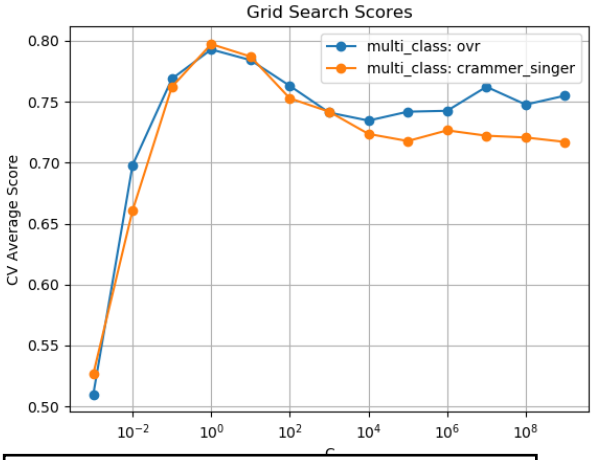
The learning curve from the best individual known in the grid search produces an interesting observation. As the training sample increases, the training set accuracy starts to decrease. This is not seen with other algorithms, although there is a steady increase of testing accuracy.

Caltech 101 Image Dataset

SVM does not innately support multiclass classification as it only produces lines separating two distinct labels. Two methods are proposed to resolve this issue: one-against-one (so a total of n*(n – 1)/2 classifiers for every pair) and one-against-the-rest (n classifiers for each class against the rest). Both of these methods produce multiple classifiers

and scores them based on the confidence of the result to obtain the final prediction.





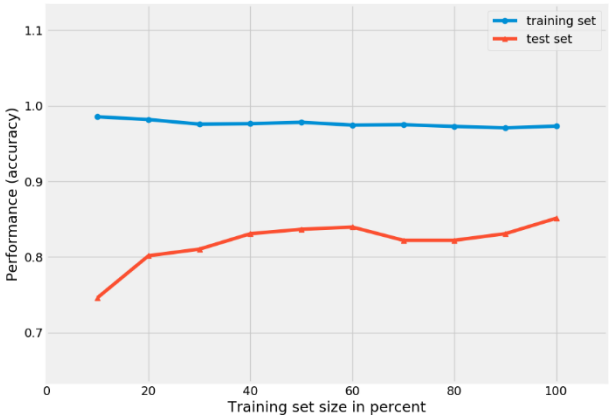
- **SVC (one vs one scheme)**
- grid search took 470.30 seconds
- grid search accuracy on test set: 85.13%
- grid search best parameters: {'C': 1000, 'kernel': 'poly'}
- grid search best average validation score: 83.44%

- **LinearSVC (one vs the rest)**
- grid search took **8012.04** seconds!
- grid search accuracy on test set: 81.05%
- grid search best parameters: {'C': 1, 'multi_class': 'crammer_singer'}
- grid search best average validation score: 79.72%

The two grid search reveals both the Support Vector Classifier with linear kernel against the Linear SVC classifier in Scikit-Learn, one of the main differences is that the LinearSVC classifier uses one versus the rest while SVC deploys the one against one scheme. Perhaps a surprising result is that the LinearSVC used significantly more time than the SVC despite, first, running only two exhaustive search (crammer singer & ovr) against SVC running four exhaustive search (linear, poly, rbf, and sigmoid,) and second, training only n classifiers for each class against the rest, against SVC's scheme to run n*(n - 1)/2 classifiers. Nevertheless, the Scikit-Learn library does provide the LinearSVC with more flexibility and more functionality than SVC's linear kernel, which could explain the significant time difference. Finally, note that both the SVC's linear kernel and the LinearSVC produces accuracy that are similar in terms of performance. In hindsight, the LinearSVC might be superior if more time is spent on tuning the hyperparameters.

Also, looking back at the results, since the image data are significantly more complex (has a complex input space dimension) than the feature data, it would make sense for signals from the image data to be better captured in polynomial space rather than linear space.

A final look at the learning curve reveals a similarity between the UCI dataset, where there is small but gradual decrease in training accuracy as the amount of training data increases, meanwhile the test data increases.



Final Analysis

This paper has been gradually providing analysis between datasets and effects of different hyperparameter tuning, this last section seeks to provide a final analysis between the various algorithms and a hopefully a satisfying conclusion. There were a lot of discussion on hyperparameters and detailed implementations of the algorithm that was delete from the previous section to focus instead on the similarities and differences of the two datasets and discussion on only the important hyperparameters. Please forgive the author for the undiscussed contents.

The algorithms that provided the best out-of-the-box results (without much hyperparameter tuning) were AdaBoost on the UCI feature dataset and Support Vector Machine with polynomial kernel on the Caltech image dataset. AdaBoost was able to achieve an astonishing 97.37% accuracy and SVC with an also impressive 85.13% accuracy. Most algorithms were able to easily achieve an above 90% accuracy on the UCI feature dataset, given that it has already been heavily feature engineered and is less noisy in comparison to image data. The last few percentages of accuracy (or f1 score) are usually the hardest to achieve. Nonetheless, AdaBoost was able to break the ceiling by

focusing on the wrongful classification and optimizing from these classifications. The image dataset was not able to perform as well on AdaBoost perhaps due to the dimensionality of the input space, which cannot be easily captured by the weak learners (decision trees) that this paper used. On the other hand, Support Vector Machines are known to learn fast on a small amount of data, which in other words mean that it is easy for these classifiers to get to the initial 90 or 95 percent accuracy, but it becomes an extremely hard task for them to learn the last few percentages without overfitting. This could explain for SVM's out-of-the-box performance on the Caltech image data. In conclusion, the author hopes there was more space and time (and computing power) to tune and discuss the hyperparameters that could have been turning points for many of these algorithms.

Reference

Cybenko, G. (1989) "Approximations by superpositions of sigmoidal functions", Mathematics of Control, Signals, and Systems, 2 (4), 303-314

Kurt Hornik (1991) "Approximation Capabilities of Multilayer Feedforward Networks", Neural Networks, 4(2), 251–257. doi:10.1016/0893-6080(91)90009-T

Hinton, G. E.; Osindero, S.; Teh, Y. W. (2006). "A Fast Learning Algorithm for Deep Belief Nets" (PDF). Neural Computation. 18 (7): 1527–1554.