Supervised Learning

Trevor Goodell
OMSCS
Georgia Institute of Technology
Atlanta, USA
tgoodell6@gatech.edu

***Abstract*—Implemented (loosely), optimized, and explored five different learning algorithms; decision trees, boosting, k-nearest neighbors, support vector machines, and neural networks. Each model was tested on two datasets, using the same procedure for testing across all combinations.**

***Index Terms*—decision trees, boosting, k-nearest neighbors, support vector machines, neural networks**

I. Datasets

For this assignment, I looked into two datasets; the UCI Wine Quality dataset and the UCI South German Credit dataset.

A. Wine Quality Dataset

The wine quality dataset consists of 6497 instances, 1599 samples of red wine and 4898 samples of white wine, each with 12 attributes. The first 11 attributes consist of various measurements such as acidity levels, dioxide levels, pH balance, etc. Each of these attributes was some various float between 0 and their respective caps. The 12th attribute is the output variable, which is the quality of the wine on a scale of 0 to 10. This score was determined as the median score of at least 3 expert wine tasters. The features are relatively dependant on each other, which leaves room for a dimensionality reduction algorithm or feature selection algorithm to strengthen these learning algorithms.

The dataset is very unbalanced, 76% of the samples are either a quality score of 5 or 6. Therefore, not only was the problem tough to begin with as a 11-nary classification task, but by being this heavily biased there would not be enough representation of the classes for the learning algorithms to learn the remaining 8 classes. To remedy this, I changed the problem into a binary classification problem by rephrasing the problem "Would I buy this wine?", this changed the scope of problem from what quality score does this wine have to, is this wine good. Therefore, I filtered the dataset to 0's and 1's with any rating below a 6 as a 0 and everything else as a 1. Now, our two classes have well sized sample spaces, 2384 samples of 'bad' wine (a rating of 5 or below) and 4113 samples of 'good' wine. While there are still nearly double the amount of samples for good than bad, this is much more balanced than previously.

B. South German Credit Dataset

The south German credit dataset consists of 1000 samples each with 21 attributes. The first 20 attributes are features of the person current financial status, intent with the loan, job status, and other various bits of information that banks gather to determine your credit status. The 21st attribute is the output variable which is a binary value of 0 being "Bad Credit" or 1 being "Good Credit".

These attributes are on various scales of what their numbers mean. Some attributes are just regular floats or integers. Others are ordinal categories. These are difficult for learning algorithms to use because translating them to numbers might not be as linear as it suggests. For example, an ordinal category of ["Good", "Alright, "Not Good", and "Bad"] being converted to [4, 3, 2, 1] implies that "Good" is two times better than "Not Good" and "Not Good" is two times better than "Bad" and sometimes this is simply not the case. One example of this being an issues is the attribute "purpose". This has 11 options for it [0 through 10], and each number corresponds what the person's purpose with the money is. 1 is a new car, 2 is a used car, and 3 is furniture. When performing math on these values, it does not make sense to say "Furniture is three times a new car".

C. Why are these interesting?

The wine dataset is interesting in the fact that it's (almost) the dream dataset. The attributes are all naturally occurring float values which behave nicely inside of learning algorithms and there is a large amount of samples for each class, after revisions. On the other hand, the south German credit dataset is on the complete opposite end of the spectrum. There are relatively few samples, only 1000 total, and the attributes are mostly categorical which can cause issues. When comparing these two datasets, it will be interesting to see the differences in how the classifiers handle them.

II. Test Methodology

All learning algorithms and preprocessing was done with sklearn and numpy. When testing each classifier, there are 4 steps that it goes through.

First, the data is loaded in and scaled using the MinMaxScaler() method provided by sklearn. This will normalize each attribute to be between 0 and 1. Then the dataset is split into a training set and a validation set.

Second, a fresh out of the box is trained on the training set using stratified K-folds. This splits the training set into another training set and a validation set with classes proportional to the original training set. Then, it trains the model on the K number of folds, and computes the accuracy and f1_score of the model on the training set and validation set. Finally, it will average these metrics over the K folds to get the result. Also, this was done as a function of number of training instances used. So, the training set was broken into chunks to see how it the algorithm performed as amount of data increased.

Next, the model would go through a GridSearchCV to find the optimal parameters. Them, this optimal model would go through the same steps listed as the base model.

Finally, I chose an attribute or two of interest and plotted how changing that value would change accuracy and f1-score of the model.

I chose to use accuracy as one of the metrics to look at as it's the most relevant metric for any algorithm. Accuracy gives the best overall sense of how the model is performing.

f1-score is a bit more complicated in what it means. The equation is given by

$$F1\ Score = \frac{2 * recall * precision}{recall + precision}$$

where recall and precision is, respectably, given by

$$\frac{TP}{TP + FN}, \frac{TP}{TP + FP}$$

TP stands for "true positive" and this is the number of samples that were classified as 1 and their true classification is also a 1. FN stands for "False Negative" and this is the number of samples that were classified as 0 when their true classification is a 1. Similarly, FP is a "False Positive" and they were labeled as a 1, when they should have been a 0.

To understand the f1-score, we must first understand precision and recall. From the name, you can guess that precision is how precise the model is, it's the measure of how often the label of class A is correct. Precision is different than accuracy because it only looks at 1 class, whereas accuracy is the total number of properly labeled instances over all instances (TP + TN / TP + TN + FP + FN). Recall on the other hand is how often the model makes a mistake, it's the measure of how often class A is labeled correctly. These are very minute differences that are best demonstrated through an example.

A system that detects for cancer would want to have high recall. It is important that someone with cancer is not labeled as someone without cancer (False Negative), as this can lead to a patient missing treatment early in the cycle which is very important. A spam filtering model would want to have high precision. It would be bad for the system to label important emails as spam (False Positive), causing the user to miss those emails.

Recall and Precision are inversely proportional to each other. In a hyperbolic example, let's consider labeling everything as a positive. We would have 0 false negatives and tons of false positives, giving us a recall score of 1, and then a precision score of 0.5 if the dataset was perfectly balanced. For most models, the balance of precision and recall is ideal, and this leads to the f1-score, which is the harmonic mean of recall and precision [3]. For our two datasets, there is no reason to prefer either recall or precision, so f1-score was chosen.

With accuracy, it is easy to tell when it is good. All though it is problem specific, for a binary task randomly guessing on a balanced dataset would give you an accuracy of 0.5. Therefore, having a model with an accuracy below 0.5 is very bad. As your accuracy grows above 0.5, your model is better and better. With f1-score, it is a bit more difficult. With no information, the highest f1-score you can achieve is 0.66. This is done by always guessing a sample is positive. Recall would equal 1 and precision would roughly equal 0.5. So, similarly to accuracy if our models have an f1-score below 0.66 there is an issue and everything above that keeps getting better and better.

For each graph shown, the x-axis while represent the percent value of how much of the training or testing set is being used, from 0 to 1, unless otherwise specified. The y-axis represents the values indicated by the curves, blue curves represent the f1-score and the red curves represent the accuracy.

III. Results

A. Decision Trees

Decision trees are learning algorithms where the data is continuously split according to a certain parameter [1]. Decision trees mainly have two components, a leaf and a node. The node is how the decision tree splits at that level. The leaf is output of the decision tree if the input sample's path to the bottom lead there. You split at a node by determining what feature optimizes a certain criterion. Optimized criterion can mean multiple things such as less variance amongst the samples in the two (or more) groups, maximizing the information gain (entropy), or minimizing the probability of randomly guessing if the guess was drawn from the distribution of the split (gini).

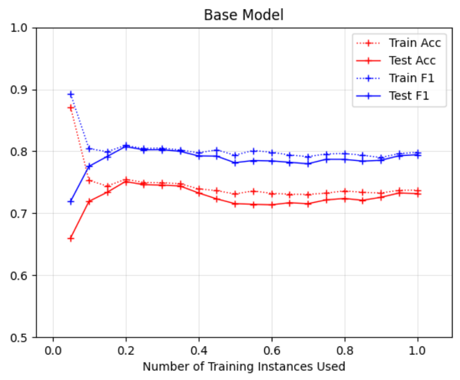
Another important aspect of decision trees is pruning, this stops the tree from overfitting to the training set by getting rid of nodes that don't provide a lot of information. Sklearn uses "Minimal Cost-Complexity Pruning" for this. Minimal Cost-Complexity Pruning is determined by:

$$R_\alpha(T) = R(T) + \alpha|\tilde{T}|$$

where T is the given decision tree, $|\tilde{T}|$ is the number of terminal nodes in T, and $R(T)$ is the total misclassification rate of the terminal nodes [5]. Ideally, you would want $R(T)$ to be as small as possible, but with decision trees you can keep splitting until each input sample maps to it's own output leaf. By using $R_\alpha(T)$ instead, you allow your tree to make more mistakes if it uses less terminal nodes. This is proportional to α which is called the 'ccp_alpha'.

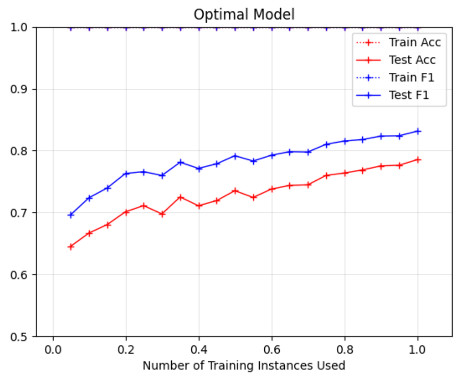
1) Wine Dataset: For the wine dataset, I started with the DecisionTreeLearner from sklearn with a ccp_alpha of 0.05. I chose to start with a higher ccp_alpha value because I wanted to show how overpruning could be harmful to the performance of the model. Since the dataset is very representative of the true sample space, this means that when splitting the training set into train and test sets, the train set will still be representative of the sample space. This allows the algorithm to "overfit" to the training set, usually this results in poor performance against the testing set, however with this dataset it is beneficial.

This resulted in an accuracy of 0.73 and an f1-score of 0.79. From the learning curve, you can see that the training and testing scores are very in-line. When the model first starts learning, you can see that the training and testing accuracy are drastically different, this is when the model hasn't seen enough data to properly learn, so it over commits to the data. However, you can see very quickly see the scores converge. Interestingly enough, you can see that the model performed really well earlier on than at the end. This shows that either there is some data that distracts the model and confuses it, or we are pruning away important data.

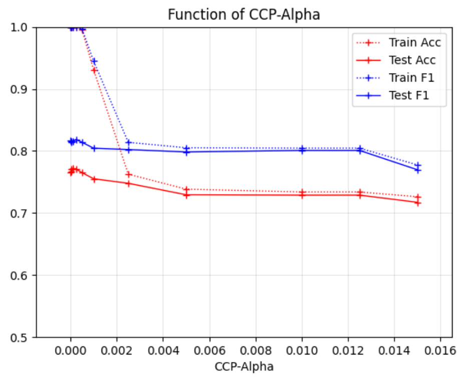


Next, I did a GridSearch over ccp_alpha and criterion. The two choices for criterion were gini and entropy. The range for ccp_alpha was a minimum of 0.00001 and a maximum of 0.0001, with a step size of 0.00001. After fitting on the training set, it chose the entropy criterion with a 0.00003.

These changes resulted in an increase in testing accuracy and f1-score, 0.79 and 0.83 respectively. This was an 8% increase in accuracy and a 5% increase in f1-score. However, you can now see that the accuracy and f1-score of the training set is 1. In this example, you can see that allowing the model to overfit actually improved performance.

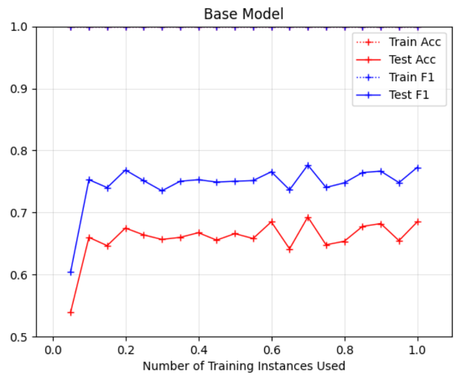


To further explore ccp_alpha, I did a wider search to see how much larger values would impact the model. This range was over [0.00001, 0.000025, 0.00005, 0.0001, 0.00025, 0.0005, 0.001, 0.0025, 0.005, 0.01, 0.0125, 0.05]. You can see on the left, there is a small bump of points which maximizes at the 0.000025 value, then slowly decreases as it increases. These results show that we were originally losing information through pruning.



2) Credit Dataset: For the credit dataset, I started with a DecisionTreeClassifier from sklearn with a ccp_alpha of 0.00001. Doing some pre-testing with this dataset, I know this dataset was much tougher than the wine dataset, if the model were to even trust the data a little too much the results would be drastic. Therefore, I chose to have a very small value of ccp_alpha to show how pruning could be beneficial as well.

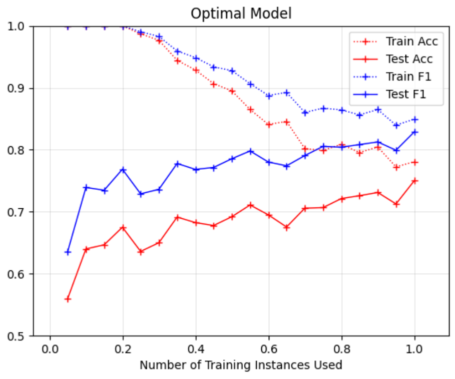
This resulted in an accuracy of 0.68 and an f1-score of 0.77. Initially, you can tell this model performed much worse than the base model of the wine dataset, showing how this dataset is much harder than the wine dataset. Similar to the wine dataset, when training first starts the training and testing scores are much different than each other. However, unlike the wine dataset, these scores never converge. You can see that the model is drastically overfitting. The training set is at an accuracy and f1-score of 1 for all of training, but the testing scores are much further behind.



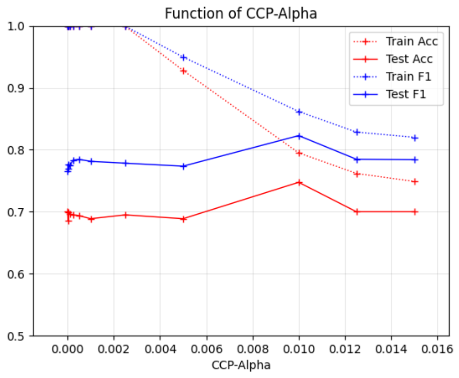
To overcome this overfitting, I used a GridSearch with the same criterion as the wine dataset, but over a much higher ccp_alpha; (0.001, 0.0025, 0.005, 0.01, 0.0125, 0.05, 0.1, 0.125, 0.15). The GridSearch settled on the 'gini' criterion with a ccp_alpha of 0.01. This is quite interesting because the gini index favors larger partitions with similar vlaues, while entropy favors smaller partitions with many unique values [4].

Every feature in the wine dataset is a float, which exists as a real number, meaning almost every sample was unique. The credit dataset, most of the features are integers on very small ranges. Therefore, it makes sense that gini would be better suited for this credit dataset, while entropy being better for the wine dataset. Also, since the dataset is not fully representative and quite small, more aggressive pruning can be beneficial.

These changes resulted in an accuracy of 0.75 and an f1-score of 0.83, an increase of 10% and 8% respectively. This dataset had much more to gain from being optimized than the wine dataset, granted I purposefully made the base model poorly perform. In this learning curve, you can see that the training scores actually decrease as training goes on while testing scores increase until they nearly converge. Here you can see how pruning actually benefits the model.



To further explore ccp_alpha with this dataset, I did a search over the same range as the wine dataset. Interestingly enough, you see the same little bump on the very left that actually decreases intially. However, the testing scores actually increase for a bit as it approaches 0.01, the same value from the GridSearch, and then decreases again.



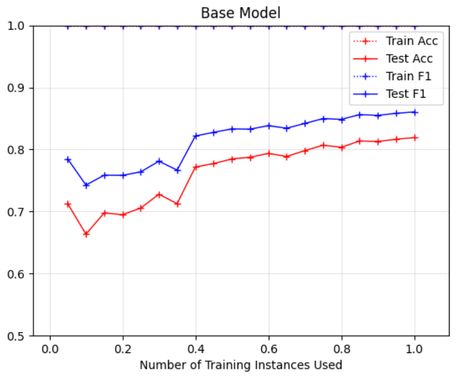
B. Boosting

Boosting is a learning algorithm that takes a series of weak learners and combines them to form strong learners, with the aim of reducing bias and variance [2]. A weak learner is said to be any sort of learning algorithm that performs better than randomly guessing. In the case of both of our problems, this would be anything better than 0.5. The idea of a ”strong” learner is a bit vague in this sense, that there really is no ”strong” learner, but the gist is that it is a model where the whole is greater than the sum of its parts. It can take a bunch of models with an accuracy of 0.55 and (theoretically) get up around 0.8 or 0.9.

Specifically with Sklearn, the boosting algorithm that will be used is the AdaBoost algorithm. The AdaBoost algorithm will select a random subset of the data and train its weak classifier against it. After it trains, it will see which samples it classified correctly and incorrectly. It will assign a higher weight, in the distribution for being picked to be trained on, so that the next learner can learn those features. It also assigns a weight proportional to the accuracy to that weak learner. This process repeats until you have reached the total number of estimators specified or there is no error. When it comes to inference for classification, it will vote amongst all the learners where their votes are weighted by the scores given to them. For regression, it would perform a weighted mean.

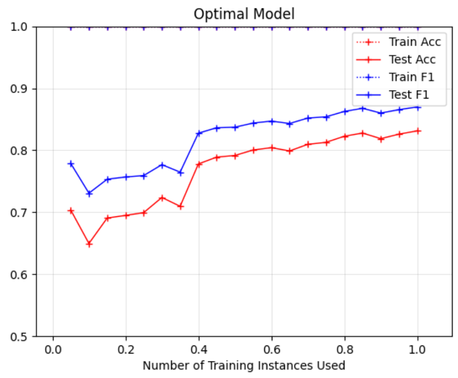
1) Wine Dataset: For the wine dataset, I used the AdaBoostClassifier with a DecisionTreeClassifier as it's base estimator, or weak learner with 50 estimators. The DecisionTreeClassifier used the entropy criterion and a ccp_alpha of 0.005. As mentioned in the Decision Tree section, the classifier there had an optimal ccp_alpha of 0.00003, roughly 166 times larger. This seems odd since we discussed that the higher values of ccp_alpha actually degraded performance for the decision tree. That is the point with boosting, you can be much more aggressive with pruning and the learner will learn a specific feature of the dataset, with the other estimators learning the things that this learner could not.

This resulted in an accuracy of 0.82 and an f1-score of 0.86. This is quite impressive, we took a model that was suboptimal, due to the higher ccp_alpha, and combined 50 of them and we had a better performance. There was an increase in 4% in accuracy and 4% f1-score for simply combining multiple decision trees. Similar to that of the singular Decision Tree, the training scores are 1 and the testing scores grow linearly similar to the decision tree, just performing slightly better at every step.

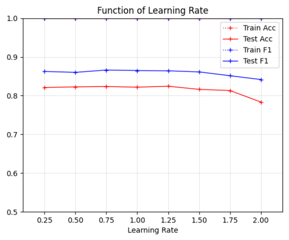


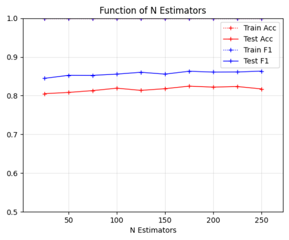
To optimize this already well performing algorithm, I chose to do a GridSearch over the number of estimators, with a minimum value of 25, a maximum value of 250, with a step size of 25, and the learning rate, with a minimum value of 0.25, a maximum value of 2, with a step size of 0.25. The learning rate contributes how much weight is applied to each learner at each iteration. After fitting on the training set, it resulted in 250 estimators and a learning rate of 1.25. Like previously mentioned, every sample is important to the model, for this reason I am not surprised that it resulted in the maximum number of estimators. This allowed the model to learn as much as possible, with something new with every learner.

These optimizations resulted in an accuracy of 0.83 and an f1-score of 0.87, an increase of 1% and 1% respectively. The results of optimization are a bit lack luster here, it appears that you slide up every point by 0.01.



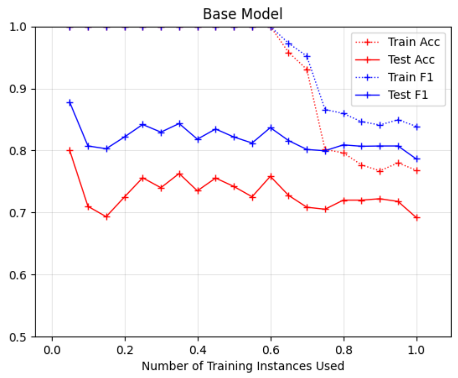
To further explore the learning rate and the number of estimators, I ran two seperate searches where I fixed the one not being tested to what was found to be optimal from the GridSearch. First, for the learning rate, I tested over the same range as the GridSearch and once again the results are a bit lack luster. The results are nearly horizontal but tails off as the learning rate increase past 1.5. If you pay close attention you can see that there is a tiny peak right at 1.25, verifying the results of the GridSearch. For the number of estimators, the results are a bit more exciting, you can tell there is a general positive linear trend as the number of estimators increases. The results appear to taper down past 175, and actually decrease a little at 250. This is most likely due to a result of randomness and drawing samples. The GridSearch uses the StratifiedKFold with 5 folds, while this used 10 folds.





2) Credit Dataset: For the credit dataset, I started with the AdaBoostClassifier with a base estimator of a DecisionTreeClassifier with a ccp_alpha of 0.01, this is the same as the DecisionTreeClassifier from the the Credit Decision Tree section. I did some pre tests and found that increasing the ccp_alpha had very negative impacts, going much higher than 0.01 resulted in the weak classifier having an accuracy below 0.5, and this would cause the learner to no longer be a weak learner and not suitable for this boosting algorithm.

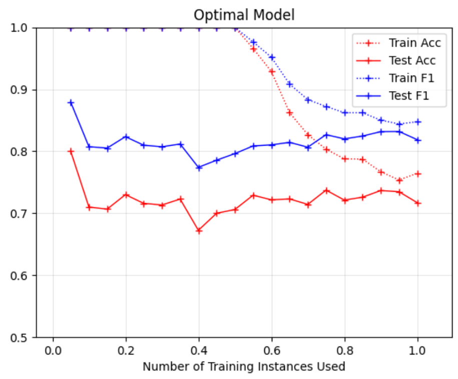
This resulted in an accuracy of 0.69 and an f1-score 0.79. This was actually quite shocking, these results are worse than the optimal decision tree, in fact it's not even a 1% increase from the base decision tree. By looking at the learning curves, you can see that it starts off pretty well, but as training goes on, training and testing scores decrease.



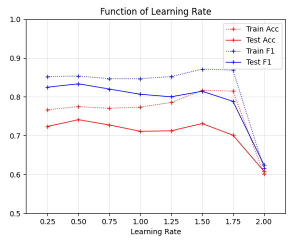
Similar to the wine dataset, I ran a GridSearch on the learning rate and number of estimators. The learning rate range was the same as for the wine dataset. The number of estimators decreased to a range of a minimum of 10, maximum of 100, and a step of 10. After fitting on the training set, the GridSearch resulted in a learning rate of 0.5 and 10 estimators. This is quite different than the wine dataset. The wine dataset landed on a high learning rate and a higher number of estimators. This makes sense when comparing the two, the data is very good in the wine dataset, the credit dataset is not like that. The longer the learner stares at this data, the worse it gets. This was noticed in the decision tree algorithm as well.

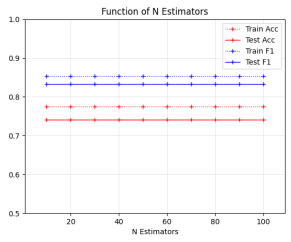
These changes resulted in very little results as well, an accuracy of 0.72 and an f1-score of 0.82. This was a 4% increase in accuracy and a 4% increase in f1-score. These results are interesting, the initial training is just as high as before, but then the training accuracy is lower than in the base model until it reaches 80% of the way through. Then, it will increase for a little bit and eventually decrease at the end. One interesting thing is the training accuracy is worse in the optimal model, further showing that beliving in this dataset can hinder it's performance.

To further explore the learning rate and the number of estimators, freezing the one that wasn't being tested. The number of estimators had absolutely no effect on the model.



The learning curve is completely horizontal, no changes as the model steps through its 10 values. The learning rate learning curve on the other hand is much more interesting. You can see it has a clear peak at 0.5, the optimal learning rate that the GridSearch found. Similar to that of the wine dataset, there was a sharp decline when the learning rate went to 2.





C. Support Vector Machines

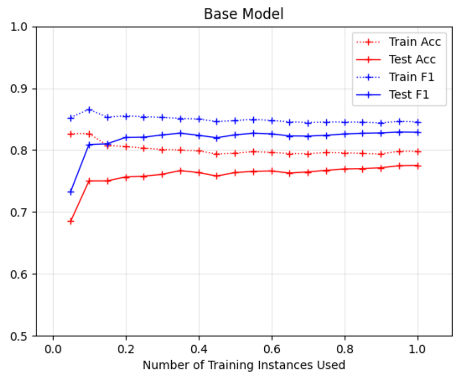
Support vector machines are a learning algorithm that aim to create an (N-1)-dimensional hyperplane that maximizes the margin between classes. As an example, in 2-dimensions, it's trying to draw a line between two clusters of data so that one cluster is on one side, and the other cluster is on the other. This works very well for linearly separable datasets, but unfortunately most datasets aren't 100% linearly separable.

There are some tricks with SVM's to circumvent this. The first being 'C'. Essentially, C determines how much you dislike misclassifying points. If given a tangled set of points, a human could go through a draw line that weaves in and out of all those points to perfectly classify everything. As number of points, features, and dependence between the classes increases, that sort of line gets more and more complex and requires a higher value of C.

Another trick is what is referred to as the kernel trick. This takes your N-dimensional feature and converts it to some higher dimension space, where it would, hopefully, then be separable by a hyperplane there. The most common example of this is two sets of clusters, where one is a disc centered at the origin, and the other is a washer centered at the origin, but the hole in the middle is large enough to fully encompass the first set. There is no line that could separate these 2 sets, however, if you casted it to a 3-rd dimension. The new feature value is the distance from the origin, the disc class would have smaller values, while the washer class would have larger values.

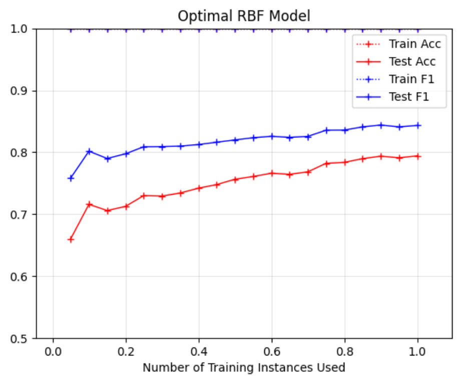
The default kernel is linear. Although it doesn't cast your data into a higher dimensional space, it is still considered a kernel method. There are several kernel's that are commonly used to increase the dimension of your sample space. The two that will be discussed in this paper are the polynomial and the radial basis function. The polynomial kernel method creates a polynomial of degree d, such as $c_d * x^d + c_{d-1} * x^{d-1} + ... + c_1 * x^1 + c_0$, where x is the sample and d is a tunable hyperparameter. The RBF kernel instead takes two samples x and x' and computes $RBF(x, x') = exp(-\gamma||x-x'||^2)$ where $||x - x'||^2$ behaves as a similarity metric and γ is a tunable hyperparameter. As $||x - x'||^2$ increases the value of the RBF kernel decreases, therefore it maps similar input samples, those with small $||x - x'||^2$ values, closer together.

1) Wine Dataset: For the wine dataset, I started with base SVC (support vector classifier) from sklearn. This uses the rbf kernel a C value of 1, and a γ set to 'scale' which is $\frac{1}{n_{features} * X.var()}$, however, the variance is 1 due to the Standard scaler applied. This resulted in an accuracy of 0.78 and an f1-score of 0.83. The learning curve is relatively flat after 40% of training samples. With this dataset, this makes sense as the data is very high quality, so it can easily learn to split the data with very few samples.

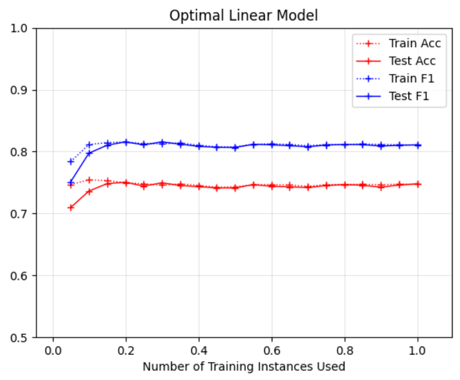


Instead of optimizing one model, I chose to optimize multiple models with different kernel functions to better see how they all depend on each other.

First, I wanted to see how much better the RBF model could do. I did a search over C and γ, where C had values of [0.1, 1, 10, 100, 1000] and γ had values of [1, 0.1, 0.01, 0.001, 0.0001]. After running the GridSearch over these ranges, it chose a C value of 100 and a gamma of 1. I was quite shocked to see such a high value of C chosen for this dataset. As mentioned a multitude of times, this dataset is quite easy to work with and behaves well. High values of C are usually intended to stop the model from overfitting, but as we saw in the decision tree and boosting sections, overfitting on the training set is actually beneficial to the testing accuracy. This model had an accuracy of 0.79 and an f1-score of 0.84, slightly better than the unoptimized version.

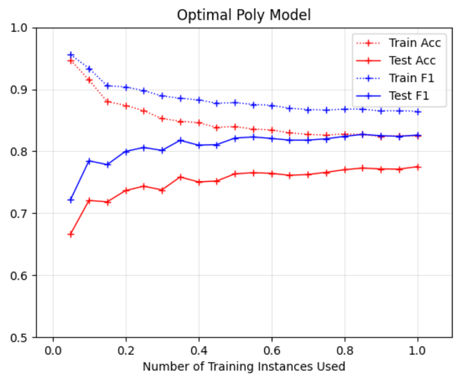


Next, I wanted to see how the linear kernel would perform. I've been bragging this whole time about how well behaved this dataset is, and if it were truly a super easy dataset, then the linear svm would have no issues classifying it. The linear kernel chose a C value of 1, which is what I would have expected, yet it's accuracy was only 0.75 and f1-score of 0.81. It still performed decently well when compared to other algorithms, but not as well as the rbf model.

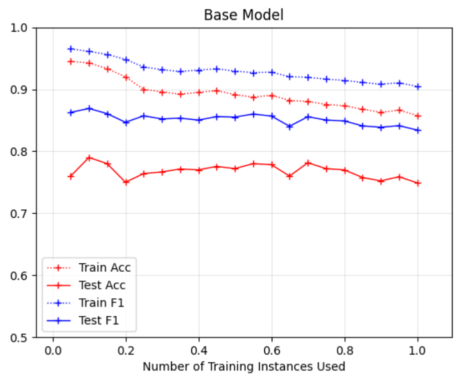


Finally, the poly kernel, this chose a C value of 1 with a degree of 4. Interestingly enough, it had an accuracy of 0.78 and an f1-score of 0.83, worse than the linear kernel! This is not surprising for this data, it has really great properties that make it behave linearly separable. This is one issue with the poly kernel, it can increase in degrees, but you can model a 3rd degree polynomial as a 4th degree polynomial but setting the coefficient associated to the 4th term near 0.

2) *Credit Dataset:* For the credit dataset, I started with the same SVC as mentioned in the wine dataset, and this time it resulted in an accuracy of 0.75 and an f1-score of 0.83. These results are quite surprising. Usually, if you used the same classifier on the wine dataset and the credit dataset, you would see wildly different results. However, this learning algorithm seemed to perform just as well on both. Also similarly to the wine dataset, the learning curve is very flat, which is also odd for this dataset. This shows how powerful the RBF kernel

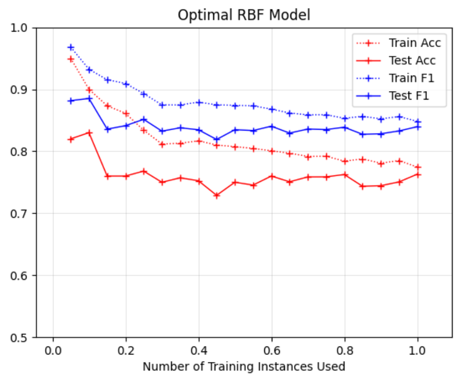


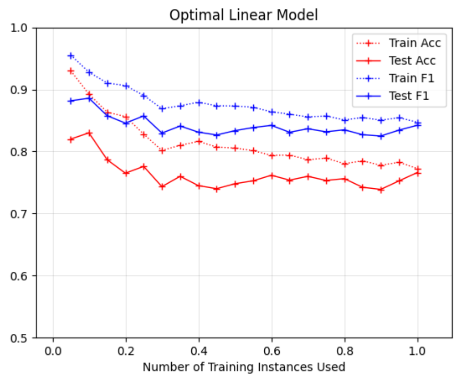
really is, trying out this kernel as the distance metric in k-NN would be very interesting.



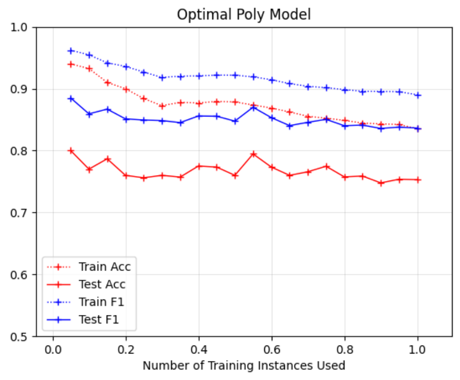
To see how optimizing this kernel would increase perform, I ran the same GridSearch over C and γ as in the wine dataset, and this resulted in a C of 1000 and a γ of 0.0001, a much larger C and smaller γ. This is not too surprising however. C is used to limit overfitting, and as we've seen in other results, believing your training set too much can really hinder performance. For γ, this controls how similar things need to be to be mapped together. Using a very small γ forces things to be very close together. With these two parameters, I think the algorithm learned to create a bunch of really small clusters and then was able to separate those. These optimizations resulted in an accuracy of 0.76 and an f1-score of 0.84. This is in-line with the increased performance seen in the wine dataset.

To see how the linear kernel handle this dataset will be interesting. As mentioned previously, the dataset does not behave well at all, it takes a lot of effort to get it to work well for the testing set. However, after searching over the same C range in the wine dataset, this chose a C of 0.1. This seems awfully low for this dataset. However, it resulted in an accuracy of 0.77 and an f1-score of 0.84. This linear kernel actually performed better on the credit dataset than the wine dataset. This is astounding! The linear kernel is the simplest of tricks in the SVM toolkit, yet it managed to perform really well here.





Finally, for the poly kernel, after performing the GridSearch, it resulted in a C of 1 and a degree of 2. This resulted in an accuracy of 0.75 and an f1-score of 0.84, just slightly better than unoptimized rbf.



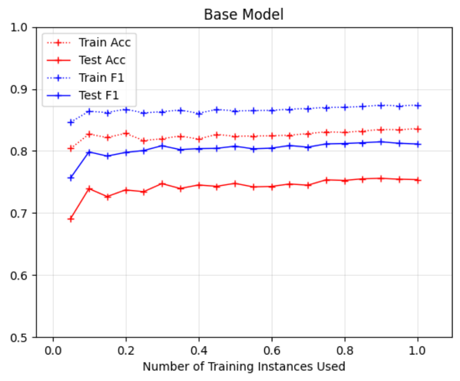
One interesting thing to note is that degree 1 was not an option for the GridSearch in the polynomial kernel. However, both datasets performed worse with the polynomial kernel instead of the linear kernel. This shows how hard it is to find the right way to change your data into a higher dimension can be.

D. k-Nearest Neighbors

k-Nearest Neighbors is a very simple learning algorithm, it is relatively self explanatory by the name. It finds the k closest samples to the given sample, and that's truly how simple the algorithm is. The training of this algorithm is to map the input given samples and do any pre-processing and that it's. At inference time, the model is given an input, it then computes the "distance" from that sample to all the training samples, grabs the k smallest values, and then those k training samples vote on the output (for classification, compute mean for regression). The hidden magic of this learning algorithm is the "distance" metric. There are tons of metrics to choose from, the most common being the "euclidean" distance. The metrics are highly dependant on the problem given.

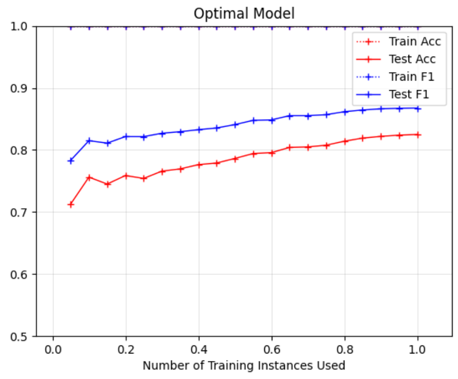
1) Wine Dataset: For the wine dataset, I used the default KNeighborsClassifer provided from sklearn. This used a k of 5, a uniform weight distribution, and the minkowski distance metric. The weight distribution determines how the neighbors will "vote" on their outputs. A uniform weight distribution says each vote is equally as important. Another option is the "distance" weight distribution, this weight distribution will scale the votes relative to the distance metric. This means that closer samples will have higher impact on the vote than something further away.

This resulted in an accuracy of 0.75 and an f1-score of 0.81. These results are slightly better than the Decision Tree but not quite as good as the Boosting learner. When looking at the learning curve, it's very flat. As the model sees more and more data, it doesn't do too much better.

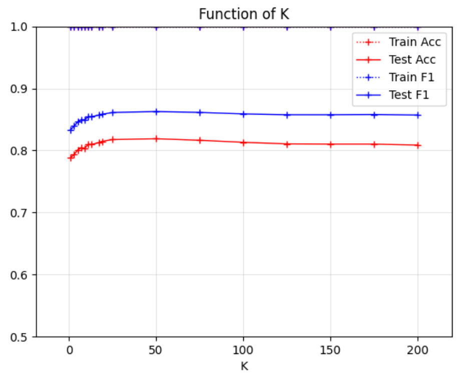


To optimize this model, I did a GridSearch over k, the weight distribution, and the distance metric. The range for k was [1, 3, 5, 7, 9, 11, 13, 17, 19, 25, 50, 75, 100, 125, 150, 175, 200], the weight distribution was choosing between ['uniform', distancee'], and the weight metrics were ['euclidean', 'manhattan', 'chebyshev']. After training on the training set, the optimal was found to be a k of 50, a distance weight distribution, and the manhattan weight metric.

These optimizations resulted in an accuracy of 0.82 and an f1-score of 0.87, an increase of 9% and 7% respectively. This is the same accuracy that the boosting algorithm reached, yet this model has a higher f1-score. By looking at the learning curve, you can see the training accuracy is at 1. With the optimal model this makes sense, using the distance weight distribution on the training set, the sample will find itself and have a significantly higher weight than any other sample. Looking at that metric is a bit misleading, it's the exact reason why there is a test set to begin with. The interesting part is the model testing accuracy increases relatively linearly as number of training instances increases. With such a high value of k, this makes sense. If you have a k of 50, but only 50 samples, you will just output the average of your training set.

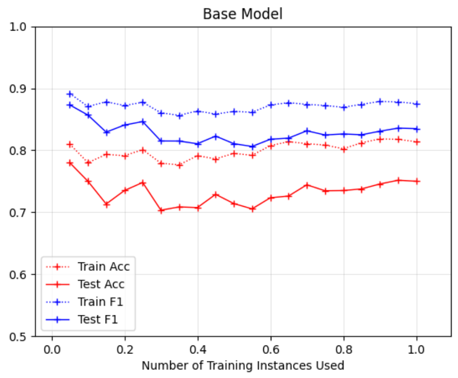


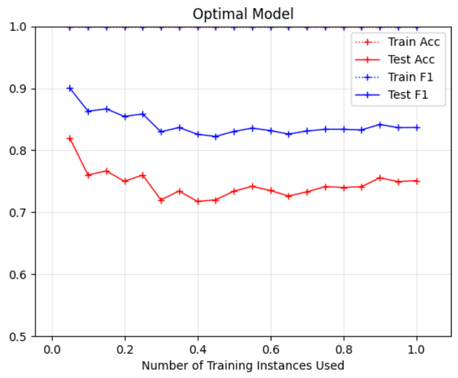
To further explore the importance of k, I searched over the same range that was done in the GridSearch. Interestingly enough, any value of K performs better than most of the other learning algorithms. However, you can see that the values increase until you reach 50, and then slowly start to decrease.



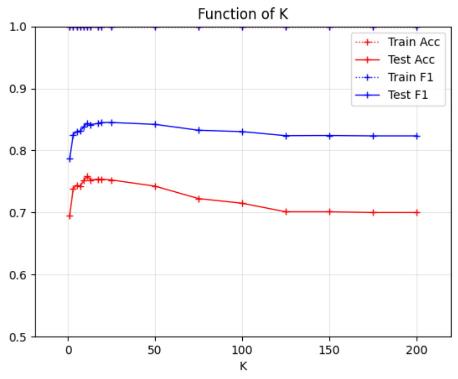
2) Credit Dataset: For the credit dataset, I started with the same base model as the wine dataset. This resulted in an accuracy of 0.75 and an f1-score of 0.83. When looking at the learning curve, it seems very wishy-washy. It oscillates up and down for awhile, and then ultimately flattens out around 0.75.

I performed the same GridSearch as done in the wine dataset, this resulted in the distance weight distribution and the manhattan weight metric, but only a k value of 11. These optimizations resulted in an accuracy of 0.75 and an f1-score of 0.84. There was no change in accuracy! As has been the theme for the other learning algorithms, this dataset cannot be trusted. Similar to the learning curve of the base model, it oscillates quite a bit, all the way up until the end.





To see how k further impacted the model, I ran the search over the same range in the GridSearch. Similar to this test on the wine dataset, there is a clear peak at the same value that the GridSearch found, hidden in the markers. But a difference here is that the other values, especially those further away, are drastically worse than the optimal value.



E. Neural Networks

Neural networks is a topic encompasses multiple textbooks on its own. In this paper, I will be discussing merely the tip of the iceberg and what could be done with more powerful toolkits like pytorch or tensorflow. In this paper, I chose to only use sklearn as it is powerful enough to gain insight on the topic, but limited in the freedom of designing your neural network.

The MLPClassifier provided by sklearn is a dense network, meaning that it is comprised of an input vector, hidden layers, and an output vector. Each node in each layer then has a connection to every node in the next layer with a weight assigned to that connection, hence the term dense. A node in the hidden layer, takes all the inputs times their respective weights from the previous layer, sums them up and passes them through an activation function to then pass onto the next layer. This is analogous to a neuron in the human mind, which is where the name "neural network" comes from. Each node determines if it will fire an input signal based on its inputs.

Neural networks are highly customizable, but in this paper we will be looking at a few of their major characteristics, the activation method each node will use, the solver they use to learn, the size of the network, and alpha.

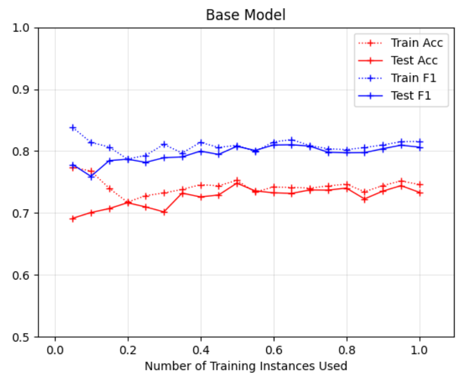
The two solvers that will be discussed are stochastic gradient descent and Adam. Gradient Descent is the process of computing the gradient of the costfunction with respect to all of the parameters for the training set. It computes this for each update and can be very slow and for large datasets, this calculation might not even fit in memory. SGD performs this operation on smaller subsets of data, which leads to a high variance updates. This then allows it to jump out of local minima and actually find the global minima much faster. Since it performs these calculations on one sample at a time, the calculations are much smaller, in terms of memory, and faster. Adam stores an exponentially decaying average of past squared gradients, RMSProp, and an exponentially decaying average of past gradients, momentum, and then combine these two metrics to update the parameters [6].

The activation method determines how a node will represent its outputs. These are usually nonlinear differentiable functions. This allows the network to learn nonlinear models while still being able to trained, since SGD and Adam require computing gradients. The three activation functions that will be used are the ReLU, Sigmoid, and Tanh functions. The general idea of each of these is they compute the weighted sum, and then output a scaled version of it to determine if the node will fire. The differences come down to the scaled output. ReLU computes $ReLU(x) = y$ where y is 0 if $ReLU(x)$ is negative otherwise, it outputs y. Sigmoid computes $Sig(x) = y$ where y is 0 if $Sig(x)$ is negative, and y is 1 if $Sig(x)$ is positive. There is a smooth transition for values close to 0. Tanh computes $Tanh(x) = y$ which behaves similarly to Sigmoid, except it outputs -1 for negative values instead.

Alpha is a parameter that aims to reduce overfitting. It does so by penalizing weights with larger magnitudes [5]. The larger alpha gets, the more it penalizes. This hyperparameter is very similar to C for SVMs, low alpha allows for more complex boundaries which will properly classify more points, while higher alphas have smoother boundaries and more errors.

The size of the network is 2 fold. There is width and depth. The width of the network is how many nodes are in each layer, and depth is how many layers there are. For this experiment, I chose to mainly look at width. Deeper networks tend to learn more and more complex patterns than shorter ones. As these datasets aren't too complicated, I chose to keep it 2 layers, and then explore wider networks.

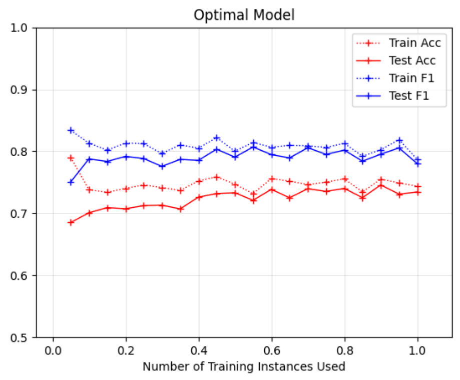
1) Wine Dataset: For the wine dataset, I started with the base MLPClassifier. This model uses 1 hidden layer with 100 nodes in it, a relu activation function, uses the Adam solver, and an alpha of 0.00001. The relu activation function with the Adam solver are standards for neural networks. This resulted in an accuracy of 0.73 and an f1-score of 0.81. The training scores and testing scores are relatively close to each other, showing that the model isn't overfitting the training set. These scores seems normal when compared to the base models of the other learning algorithms.



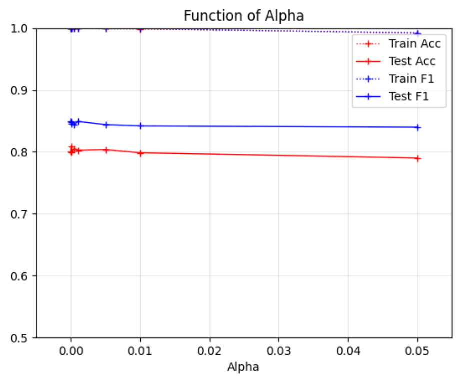
To optimize this model, I did a GridSearch over the 4 hyperparameters mentioned in the intro of this section; hidden layers, activation functions, solvers, and alpha. The hidden layers it searched over were [(22,11), (110, 55), (550, 275)]. The alpha values were [0.0001, 0.001, 0.01]. The activation function and solvers were the same ones mentioned in the introduction.

After training on the training set, the model landed on the tanh activation function, with the Adam solver, an alpha of 0.0001, and the hidden layer shape of (550, 275). With the values normalized to be between 0 and 1, I am not surprised that the tanh activation function was chosen, since they output values in the same range so it doesn't lose too much information. Also, knowing how this dataset behaves, seeing the model chose the largest hidden layer shape is not shocking. As I've learned, there is a lot to learn from this dataset.

These optimizations resulted in an accuracy of 0.73 and an f1-score of 0.78. This optimiziation actually caused the model to perform worse. Neural networks differ from the other learning algorithms because it tries to learn a function that can map an input sample to the score, whereas the other learning algorithms learned to group similar samples together. This shows that there isn't much of a mathematical backing to the quality of the wine, but a more intrinsic sense of similarity between them.

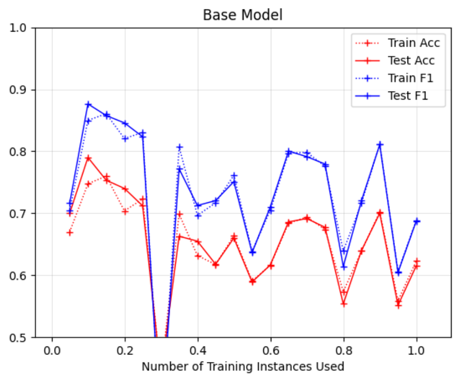


I chose to explore alpha more, and so I did a search over [0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05]. It's a relatively small effect, but you can actually see a little bump at 0.0001 if you count the number of markers. After that there is very little change. One thing to note by looking at this, is that this model was averaging a test accuracy of 0.82, which is much higher than what we saw when we optimized. This is most likely due to limited training. Each model trained for 750 iterations, a few of them converged while others did not. This shows that neural networks could still be a powerful tool for this problem it will just take slightly more optimization.

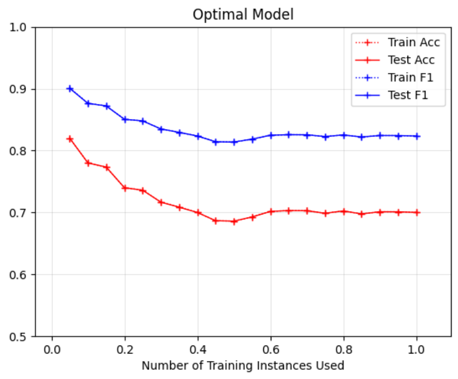


2) Credit Dataset: For the credit dataset, I started with the same base model as in the wine dataset. After training this model, it received an accuracy of 0.62 and an f1-score of 0.7. This is by far the worst base method for this problem. All though I am not too surprised. This dataset is extremely hard for unoptimized models. By looking at the learning curve it is all over the place, at one point the accuracy dips below 0.5, which means the model would've been better off flipping a coin.

I optimized over the same parameters as in the wine section, however with hidden layer sizes of [(40, 20), (200, 100), (1000, 500)]. After training, it decided on the same activation function and alpha, but it chose the middle option for hidden layer size and the sgd solver. These optimizations resulted



in an accuracy of 0.7 and 0.82, a 13% and a 17% increase respectively. This is the greatest improvements seen through optimization, yet it still hasn't been the best model for this dataset.



When looking at the learning curve, you can see that it is reasonably well behaved, it very smoothly transitions as learning goes on. This pattern hasn't been noticed in previous models. After seeing these results, I do believe that neural networks would be the best algorithm for this dataset, if it were to be explored in a more powerful tool.

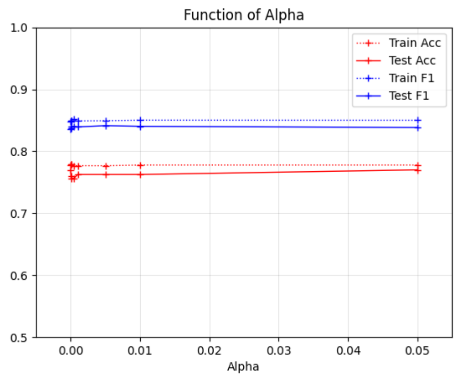
To further explore alpha on this dataset, I ran the same search over it. This time, it showed that a much smaller alpha would have been desirable, it appears that the optimal value would have been 0.00001. This was quite surprising. Since alpha serves the same purpose as C in SVMs, and this dataset chose high values of C for that model. This could mean that neural networks were able to find a relatively complex boundary that the SVM could not find.

IV. Analysis

In this section I would like to consider a few questions when comparing all of the learning algorithms we've covered in this paper and choose which method performed the best overall on our problems.

A. Training Time

For both of our datasets, k-NN was the fastest to train, this is largely due to their being no training time involved. For models



that do require training, the decision tree was by far the fastest with boosting being right behind. In terms of wall time, the decision tree was nearly instantaneous while boosting was just n_estimators times longer, still nearly instantaneous. Neural networks were much slower than boosting, but not as slow as SVMs. This is largely in part due to capping the maximum number of iterations in the training loop, most of the models did not finish training in the 750 iterations allowed. SVMs were by far the slowest, the polynomial method takes over an hour to train.

B. *Accuracy and F1-Score*

For both cases, k-NN and boosting were the top 2 performers in terms of accuracy and f1-score. However, it was boosting for the wine dataset, and k-NN for the credit dataset. The other models were significantly behind, at least 7%.

C. *Best Algorithm*

k-NN was by far the simplest algorithm, the easiest to train, the easiest to perform inference on, and a relatively well scoring algorithm. However, I think the best algorithm from this assignment was boosting. k-NN has been widely used for the reasons listed, but that is where it stops. It works great where it can, but sometimes it simply isn't powerful enough. With boosting, you could combine different weak learners to perform much more powerful ones. In our experiments, we nearly scratched the iceberg of boosting and it showed very powerful results.

This isn't to say the other algorithms are bad, they either just weren't suited well for the problems given or there were too many options to fully explore them to find the optimal method. For example, I think neural networks could be the best algorithm for the credit dataset, but it would require using a much more powerful neural network library like tensorflow.

References

[1] Decision trees for classification: A machine learning algorithm, Sep 2017.
[2] Boosting (machine learning), Nov 2021.
[3] Purva Huilgol. Precision vs recall: Precision and recall machine learning, Sep 2020.
[4] Will Johnson. Decision trees flavors: Gini and information gain, Jun 2015.
[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
[6] Sebastian Ruder. An overview of gradient descent optimization algorithms, Jan 2016.