SVM and MLP For Different Dimensional Data Sets

Brandon Begaj

Department of Electrical and Computer Engineering, Temple University
tug50246@temple.edu

Introduction: Students of the machine learning class were given two different types of data sets to work with and pitted against each other to see who could produce the lowest error rate through whatever means necessary. The only real requirement was that students must produce both a Neural Network approach as well as a Traditional machine learning method. The two data sets that were provided were one 2-dimensional data set and another 5-dimensional data set. The real challenge was more focused on the 5-D data set however the 2-D set was meant to provide some practice and supposedly the sets were created in a similar fashion.

Upon observing the 2-D data it was noticed that the data could be almost distinctly separated into 4 clusters per class. This led me to believe that when it came to a traditional approach, I would require an algorithm that could produce highly nonlinear decision regions. For this reason, the first algorithms to come to mind were kNN and SVM. As there was a decent amount of overlap, I believed SVM would outperform kNN so that was the intuition behind choosing SVM for the traditional machine learning method.

Regarding the Neural Network approach, a lot of parameter tweaking had been made with the base script as well as a lot of compute time when becoming familiar with the given multi-layer perceptron. For this reason, the MLP was kept and used to classify the given evaluation data.

Support Vector Machine (Python - sklearn.svm v0.0): Support Vector Machine is an algorithm which attempts to separate the given data it is fit to with a hyperplane. The key is that most times there are an infinite number of hyperplanes that separate the data but SVM finds an optimal generalized hyperplane. This is done through maximizing the margin between the decision plane and the two classes. What this does for us is that it generalizes the decision region by making sure it isn't too close to either of the classes we are trying to differentiate. This is very intuitive when it comes to data that is linearly separable but when this is not the case such as with the data we have in this problem, the way to accomplish the separation is by projecting the data to a higher dimensionality. The goal is to project the data into a higher dimension to accomplish linear separation and then the rest is the same as with linearly separable data.

When it came to adjusting parameters of the model from sklearn, I initially decided to use a fourth-degree polynomial kernel type as I though that a polynomial function would wrap around the data clusters more easily, producing a decent decision region. The results were slightly better than the baseline but I also used a Radial basis function kernel to see what the results were like just because the data was circularly shaped in the 2-D data. This blew the performance of the polynomial kernel out of the water so I decided to keep this as my kernel function. Another parameter that was altered was the gamma value which changes the influence of a single training sample. The only difference was that I decided to not use the variance of the data when it came to adjusting gamma as the system preformed worse. The system was trained on the combination of the training data set as well as the development set. Computational complexity was not too bad for the 2-D data set however; the 5-D set would make you wait for about 25 minutes to obtain results.

Multi-Layer Perceptron (Python - PyTorch v1.3.1): The idea behind a Multi-Layer Perceptron (MLP) is that the system will preform stochastic gradient descent in order to update the wights of our neurons. The changes made to the weights are meant to minimize an error function in our system. In doing this the idea is that the system with optimal weights will give a correct output for a given input sample. The way that training occurs is through backpropagation of the error in the system. Some parameters we can tweak is how much we allow the weights to change on a given iteration as well as how to control their magnitude, how many data points we train on in a given iteration, as well as how many times we cycle through the training and how many nodes the system has. Another very interesting thing that effects the system is the order that the training data is fed into the system. The first change made to the system was a drastic lowering of the learning rate as I found that everything was changing too fast in the system to learn anything significant. After lowering the learning rate, we can run the system with more epochs and give it more time to settle on an optimal solution. While changing all of the values regarding how the weights change during a single iteration, I also adjusted the weight decay which is the value that the weights are multiplied by after updating which just restricts their magnitude. The weight decay was lowered by an order of magnitude to get the system to where it was performing alright. When it came o the number of nodes, I knew there was an optimal number below 100 but also above 26. After trial an error and training on a relatively low number of epochs to find this, the number of 75 was settled upon due to the performance increase.

Upon settling on the chosen parameters, a model was trained with 250 epochs. The performance was not bad however the loss still produced a significant value of around 0.6 which means the model can be trained for much longer. Before touching the number of epochs however, the training data was adjusted to include the development data as well. On top of the addition, the development data was sorted and added at the end of the training data and then a model was trained on this. I also tried to sort all class 0's first and then all class 1's in the training data but this actually performed worse so the training data was just the sorted development data appended to the end of the training data. A model with 10,000 epochs was trained and this performed even better but the loss still had a way to go. The final model was trained with 20,000 epochs and the performance still increased with some loss still to go. At this point however, there was not much more time to train another model so the final model was kept.

Results: Below we will see Table 1 with the results on all data sets for both systems. What is immediately noticed is that the traditional approach seems to outperform the Neural Network approach but this should be expected with the amount of data we have.

	2D			**5D**		
Algorithm	**Train**	**Dev**	**Eval**	**Train**	**Dev**	**Eval**
Scikit SVM	8.11%	7.90%	8.35%	36.56%	36.35%	36.68%
PyTorch MLP	9.02%	9.70%	9.05%	36.80%	36.79%	36.85%

Table 1: These are the results generated from the traditional machine learning approach (SVM) as well as the Neural Network approach (MLP). As can be seen, SVM outperformed the MLP in all cases. This however is likely true due to not only the structure but also the limited amount of training data as we know Neural Networks thrive with "big data"

Conclusions: After running the experiment, I was very pleased with the results on the 5-D evaluation data by the SVM. I feel that the only thing that could have been done better was to maybe filter the training data for outliers. When it came to the MLP I felt that performance could have been improved by doing two things. First of all, the training data could have been filtered but also if a way was found to model the training data and generate more of it, then I feel the model would perform statistically better than its current state. Time is a constraint however so this was not able to be implemented in time let alone have the time to train another model for the 5-D dataset. In the end however I feel that this problem provided a great deal of practice when it came to solving a machine learning problem as any method could have been performed. I also learned a great deal about what algorithms to pick, what it takes to train a neural network and how to use SVM as there were not homework's on these methods.