Alex Vaziri

Av474

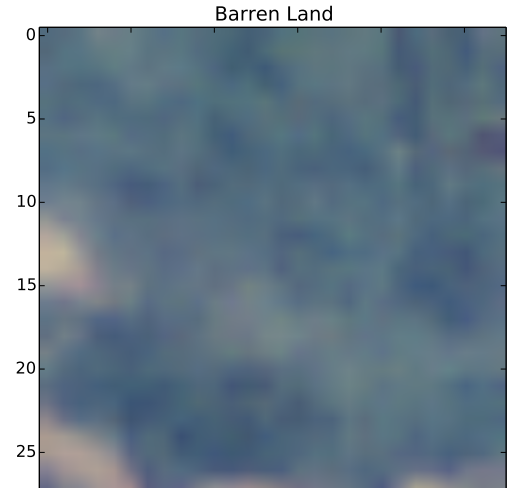
Professor Wagner

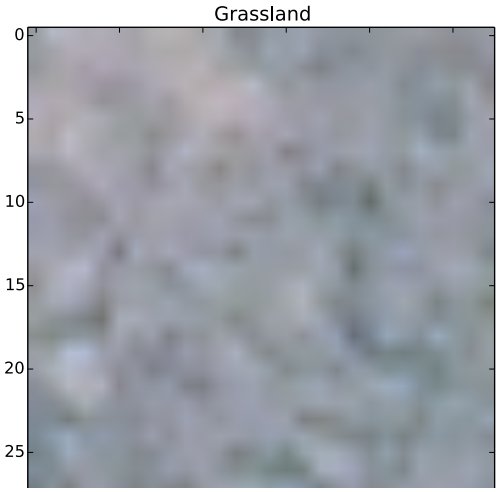
8 December 2019

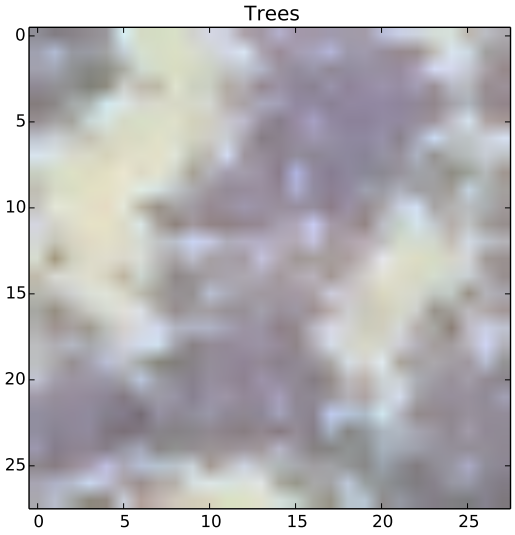
Acknowledgements: Maxim Baduk (mb2474) helped with reshaping, Sourbh Bidane

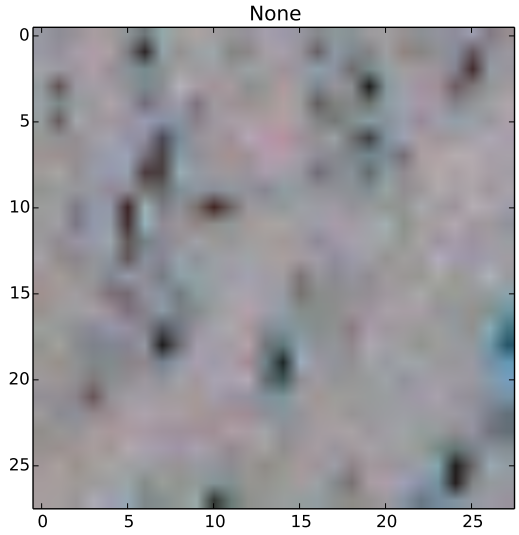
Final Programming Project Report

When examining the data in the image classification, the first thing I did was convert all of the data in the MATLAB file to dictionaries, and then numpy arrays using the loadmat functionality of scipy.io. Then, I started to look at what each of the images in each possible classification look like. Here are the 4 different images I got:







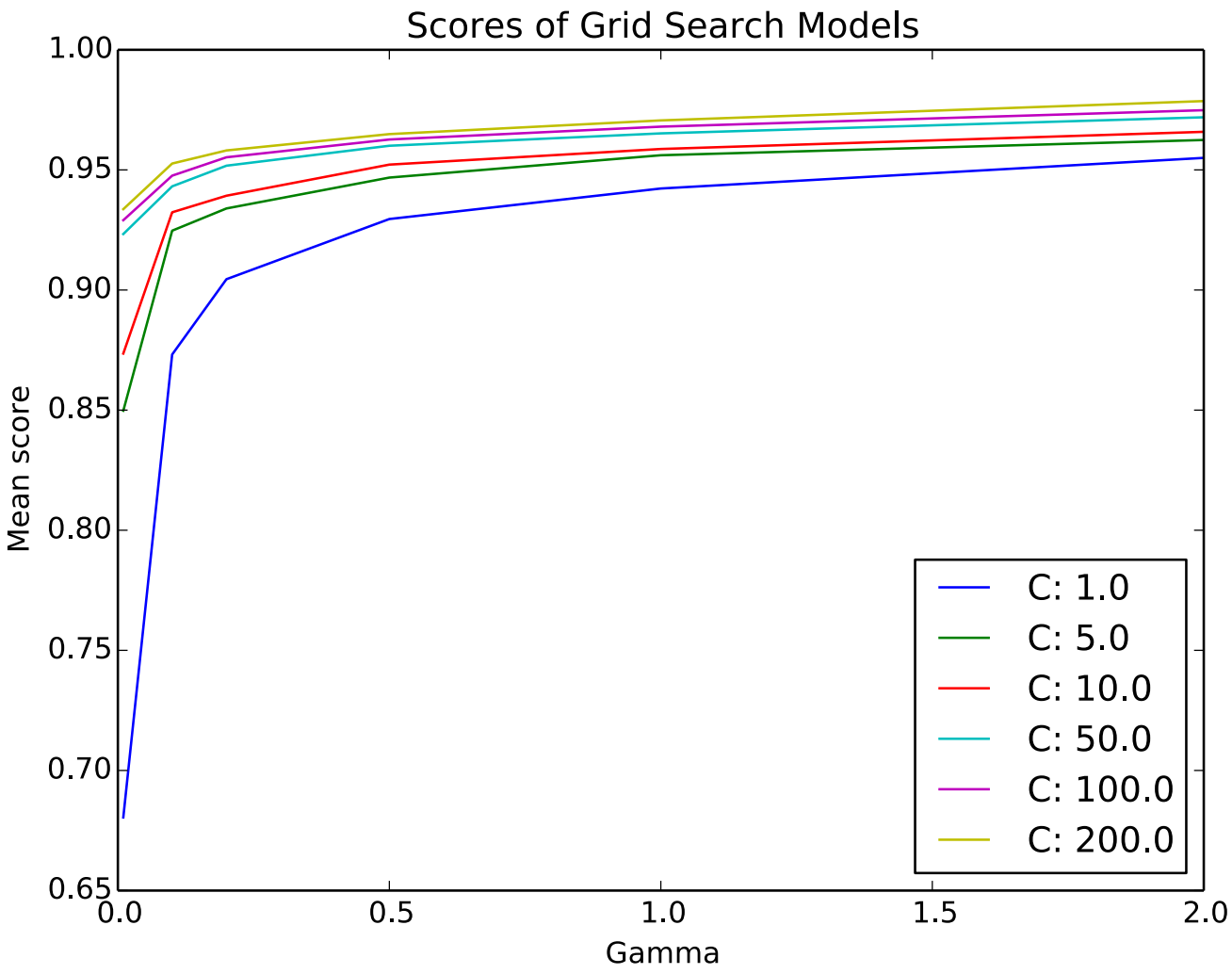


Evidently, each of the pictures have do have relatively distinct colors, which I assumed would make classifying relatively accurate. The one real questionable decision boundary seems to be None versus Grassland.

After, I tried to understand the formatting of the train_x data, and how it could be reshaped so that its 28x28x4 dimension could be scaled down. To do this, I had to really understand what 28x28x4 meant, and what each element in that space was. I thought of each image as a square matrix whose elements are vectors of length 4, containing a red, blue, green, and NIR value. When I thought of it this way, I found it easier to understand how I could preprocess the data in a way to make it both understandable to people and readable by the machine learning algorithms. After extensive computation, I reduced the dimensions to 8x400,000, 8 rows of length 400,000. Now, each image is described by a vector whose length is 8. The elements of these vectors correspond to a mean and standard deviation of red, green, blue, and NIR (in that order). After, I normalized these values by scaling them down by a factor of 255. Alongside preprocessing the train_x data was preprocessing the train_y data. For the train_y data, I converted the one hot encoding to scalar values. This reduced the data to one dimension and also made it easier to read. In the encoding, there were 4 unique unit vector that represented the different image classifications. I instead mapped these vectors to scalar integers between 0 and 3 inclusive.

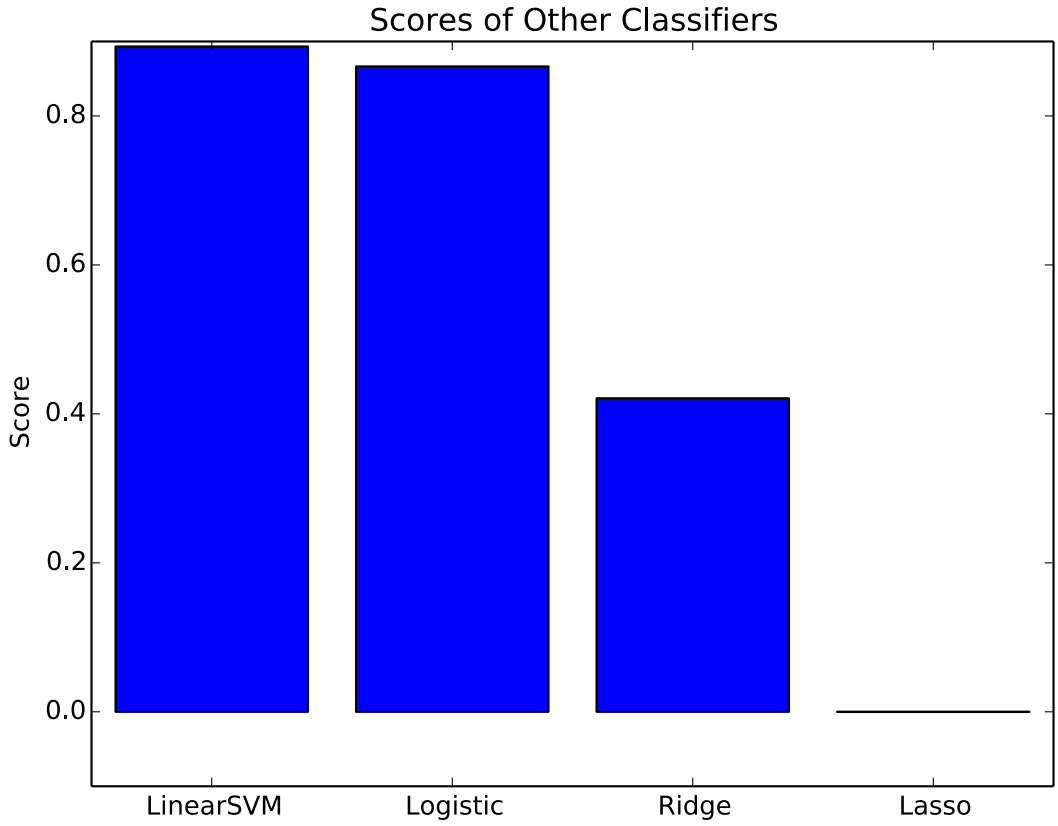
For designing the best classifier, I figured that running a bunch of algorithms and checking their performance individually would not be an efficient way to spend my time. Instead, I focused on one algorithm, and found the best hyper parameters to construct the model without overfitting. To do this, I used the GridSearchCV functionality of sklearn on an rbf-kernel

SVM. What I like about using grid search cross validation is, as the name suggests, it cross validates for you. I did 3 folds, and although I would have preferred to do 5 or 10 folds, I did 3 to save time, but even then the model should not be overfitted. I inputted 6 different parameters for both the C and γ hyper parameters. When running this algorithm, I knew in advance that it would take a long time to get results, but after it was done, the results were quite interesting. I plotted 6 lines (one for each value of the C hyper parameter) showcasing the performance of the mean scores as a function of the gamma values. The results are shown below:



It seems as though as the value of gamma increases, the performance for all values of C start to converge at around $\gamma = 2$. I find this interesting given that as C increases, making for a smaller margin in the hyperplane (and as such less chance of misclassification), the value of gamma has less effect on the overall performance. Additionally, one can see that the best hyper parameters to make predictions are when C = 200 and $\gamma = 2$. I like this result because it the model for C = 200 is not sensitive to changes in γ. Nevertheless, a large C value can sometimes lead to overfitting, as too harsh of a penalty on misclassifications can result in overfitting on the training data. However, I feel that the cross validation aspect of grid searching compensates for the large misclassification penalty. Overall, the model performs very well, having over 95% accuracy in classification, so I would conclude that grid search made hyper parameter tuning extremely easy.

I later decided to try other classifiers, those being linear SVM, logistic regression, ridge regression, and lasso regression, and on average linear SVM regression performed the best of the four, but it did not perform better than the grid search cross validation, so I ultimately stuck with the grid search model. I was curious as to how lasso regression and ridge regression would perform on classification problems, so I mainly did those out of curiosity. The results of all of the classifiers are shown below:



As you can see, linear SVM performed the best, and logistic regression came in a close second, both have accuracies of over 80%. However, a majority of the hyper parameters from the grid search performed model better on average than the logistic regression, linear SVM, ridge regression, and lasso regression models.

Overall, I think a model with over 95% accuracy is strong for this dataset, especially considering that I only trained with a sample size of 20,000. In addition, I did not randomly sample images from the dataset, I simply took the first 20,000 elements. This, however, did not affect performance that much, as I tried randomly sampling the dataset using the code: random_samples = train_x_shaped[np.random.choice(train_x_shaped.shape[0], 20000, replace= False), :], but ultimately did not end up using it. However, looking back, I believe it would have been a better data science practice to randomly sample, even if performance did not improve.