Supervised Learning

(ML Assignment 1)

Silviu Pitis
GTID: spitis3
silviu.pitis@gmail.com

0 *Preliminaries*

- Sections marked with * are for my interest, and maybe yours, but not part of requirements
- Duplicative / non-essential plots have been omitted due to space constraints. If you are interested, they are available in the IPython notebooks along with my code.
- Note that the y-axis on each plot has been adjusted to make it frame the data better, so side-by-side comparisons of figures may be misleading.

1 Basic Datasets

A MNIST: Handwritten digits

Not original, but interesting. It is baby version of the computer vision problem. MNIST is a dataset that I already know and love---I've gone so far as to train a neural network on MNIST by minimizing only the principal eigenvector of its Hessian (while useless, this surprisingly does train, achieving an accuracy over 50%). I believe connecting knowledge is the best way to learn, and view MNIST as the most efficient way to tie the new methods I will be exploring (basically everything except neural networks) to my past knowledge.

One of the questions I am interested in is whether the success of neural networks on image tasks is owed to general superiority of neural networks to other methods, or whether it is solely a result of the convolution (i.e., it's ability to enforce a prior of spatial invariance on the input data), and whether we can improve upon the performance of convolutional neural networks using other methods (more on this Section 2).

The base dataset consists of 70,000 28x28 pixel images of handwritten digits, which I treat as 784 dimensional vectors. Each image is labeled 0 through 9 (inclusive). The task is to predict the labels given the image. So this assignment does not take forever, and so we don't get into the trouble of fitting the data too well (scoring 99.5% vs 99% on MNIST doesn't tell me much about a model's effectiveness), I use a subset of the full 70,000 images, with the following training, validation, test splits:

- **Training:** First 5000 samples from the base test set
- **Validation:** Last 5000 samples from the base test set
- **Test:** All 5000 samples from the base validation set (using the typical 55000-5000 training-validation split of the base data)

B MV: Movie reviews

The Pang and Lee sentence polarity dataset,[1] introduced by Pang & Lee (2005), is a classic in the NLP literature and the predecessor of the sentiment treebank introduced by Socher et al. (2013), which is a dataset I am currently interested in outside of this assignment. I chose to work with the original Pang and Lee dataset as opposed to the sentiment treebank because it is simpler, yet still offers the challenges associated with natural language processing (NLP) data---in particular, we have a large (high-dimensional) vocabulary and need to classify variable length sequences.

1 https://www.cs.cornell.edu/people/pabo/movie-review-data/

The base dataset consists of a total of 10662 sentences, split evenly between positive and negative labels. The labels indicate the sentiment of the sentence. An example of a sentence labeled as positive is: "offers a breath of the fresh air of true sophistication ." I use the entire dataset, and following Kim (2014), I split it into 10-folds. So this assignment does not take forever, I do not perform cross-validation, and instead use the first 8 folds for training, the 9th fold for validation, and the 10th fold for testing.

To form feature vectors for the basic dataset, I use a bag of words model with a vocabulary size of 2000 (1 unknown token + 1999 most common words in the dataset). This results in 2000-dimensional feature vectors that are mostly sparse, both within a sentence, and across sentences (e.g., this is very different from MNIST, which is not sparse across samples – the non-white pixels tend to appear in the same areas of each figure).

2 Transformed Datasets*

Our models (DTs, KNN, SVM, etc.), as applied to both basic datasets all have the same flaw: they don't take advantage of known priors. For MNIST, our models ignore spatial invariance: we know the same types of visual features appear in all parts of the picture. For MR, we have the opposite problem in that the bag of words approach ignores known sequential dependence: words depend on each other---e.g., the "not" in "not cool" modifies "cool" and makes it negative sentiment.

For this reason, we will generally see convolutional neural networks (CNNs) outperform naive models on image classification, and sequential models (e.g., recurrent neural networks (RNNs)) outperform naive models on sequence classification. But what if we evened the playing field by creating hybrids? Would the pure convolutional or recurrent neural network still beat the hybrids?

A MNIST-CONV: MNIST with convolutional filter

I pass all MNIST images through the same convolutional neural network to obtain transformed 784 pixel vectors. The CNN is very basic, consisting of a single convolutional layer with 4 5x5 filters, followed by a single max pooling layer. I keep it this simple because I want the CNN filter to do as little work as possible, while still enforcing some form of spatial invariance. So that the convolutional filter does not contribute to the digit classification, I train it in an unsupervised manner as part of a simple convolutional auto-encoder consisting of the filter, followed by a single fully connected layer to invert the filter (see Zeiler et al. (2011) for a more intricate autoencoder design). The same train-validation-test split is used as for the basic data.

B MV-EMBED: MV with embedded recurrent filter

Instead of forming the feature vectors using a bag of words approach, I instead use the 300 dimensional pre-trained GLoVE[2] vectors (Pennington et al. 2014) to transform the original sentences into dense feature matrices of shape *length* x 300. We might zero-pad these sequences so that they are all the same length and apply a naive model, but this would still not take advantage of temporally invariant term dependence (e.g., "not cool"). We could use a convolutional filter again, a la Kim (2014), but I opt for a recurrent filter instead.

I pass the embedded sequences through a GRU network (Cho et al. (2014)) with a hidden state size of 500, using the final state as the feature vector (with 500-dimensions). This RNN is trained as part of a very simple sequence autoencoder consisting of the filter itself followed by a fully-connected layer projecting the final state back into 300 dimensions, which is then compared against the average of the sequence's word vectors (see, e.g., Dai and Le (2015) for a more intricate model); because this is an unsupervised process, it is not doing any sentiment classification itself, and so the bulk of the task is left to the learning algorithms being tested. Taking only the final RNN state automatically normalizes the variable-length sequences, so that no zero-padding is necessary. Once again, the dataset is split into the same 10 folds, and the same 8-1-1 train-validation-test split is used.

2 http://nlp.stanford.edu/projects/glove/

3 Experiments on Basic Datasets

A General Process

After preparing the datasets, I ran a similar process for each:

- For each of the five training algorithms:
    1. **Hyper-parameter search using validation set**

       I evaluated the effect of key hyper-parameters on the validation set.

       Note: I used the fixed training-validation splits indicated in Section 1 and did not perform cross-validation (e.g., 10-fold cv), as both my datasets were too large to do so conveniently. Their size also allowed for sizable validation sets that provided fair estimates of the out-of-sample error. This is common practice when training neural networks due to the large hyper parameter space and significant training time. The downside of this was that I noticed some (not very significant) variance when re-running experiments, which may have been ameliorated by using cross-validation.
    2. **Learning curves**

       I plotted the accuracy of the algorithm as a function of the amount of training data used, using the best hyper-parameters as found in step 1 (on all data).

       Note: Although learning curve plots may influence the hyper-parameter search in practice (see my post on piazza @164), I kept the two steps distinct and used the learning curves solely to develop intuitions about the performance of the different learning algorithms. I believe this is justified for the non-neural network architectures, simply because the hyper-parameter space is relatively limited, and was exhausted by my initial search. This choice reflects my view of this project as a learning exercise, rather than an attempt to score state-of-the-art results on my datasets.
    3. **Evaluation**

       I evaluated the algorithm on the test set using the best hyper-parameters from step 1.

B Hyper parameter Search

I chose one or two critical hyper parameters for each learning algorithm. I trained each algorithm using several different values of those parameters, and measured the results on the validation set in order to find the best hyper parameter setting. Below is a discussion of each learning algorithm, followed by a table showing the final selected hyper parameters.

Decision Trees

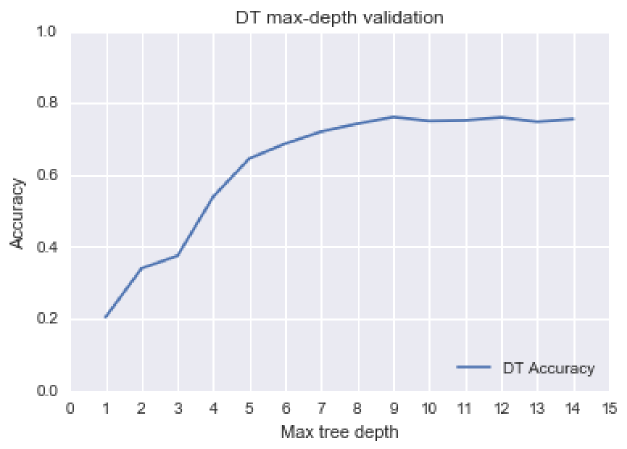
I used sklearn's DecisionTreeClassifier to fit both datasets. The sklearn decision tree algorithm is similar to the one used in class, but by default uses the "gini" criterion for splitting. Although, I was not able to develop a good intuition for what gini means (by contrast, I have good intuitions about entropy), my validation experiments were revealing. On MNIST, gini performs similarly to entropy on deeper trees (> 4 levels), and on MV, it does slightly better than entropy across the board. It also appears to run 1.5-3x faster than entropy, as it does not require a logarithm computation. I therefore used gini on both datasets.

I also ran validation against "max depth". Max depth pre-prunes the tree by capping its depth. The validation plots for max depth on MNIST and MV are shown in **Figures 1(A)** and **(B)**, respectively. There is a dip in the plot for MV, which I found to be quite interesting: I had initially chosen to stop at max depth = 8, and only decided to expand my search in order to show a nice upside down U-shape for purposes of this report. Since each decision tree split in the MV dataset is on a word, and we require several splits to bounce back up from the dip, this suggests that, as noted in

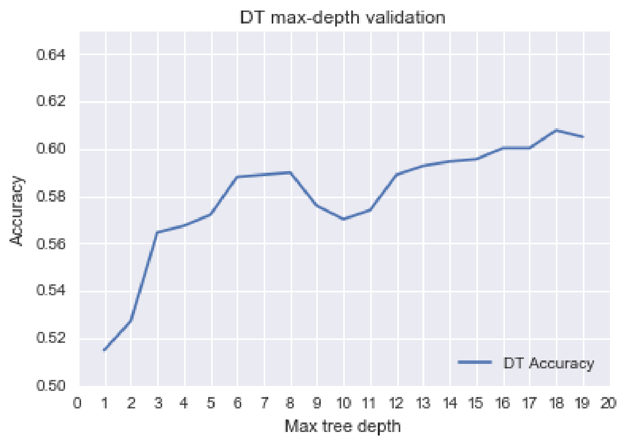
Section 2, there may be important dependencies between words (e.g., “not cool” expresses negative sentiment due to the modifier word “not”).

In addition to pre-pruning, I also evaluated post-pruning using an adapted version of Jonathan Tay’s code in piazza post @129. Unsurprisingly (since this is how post-pruning works...), this improved the results on validation. To be discussed later how this transferred to the test set. I tried post-pruning with both the best pre-pruning depth, as well as with a significantly larger pre-pruning depth, but did not notice a significant difference in results.

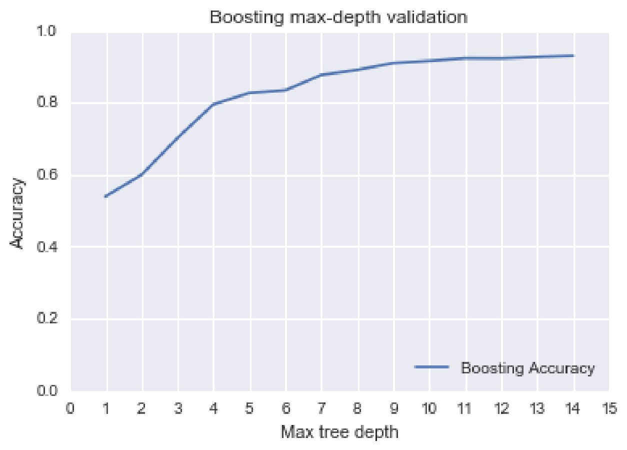
Figure 1. Selected hyper parameter validation plots



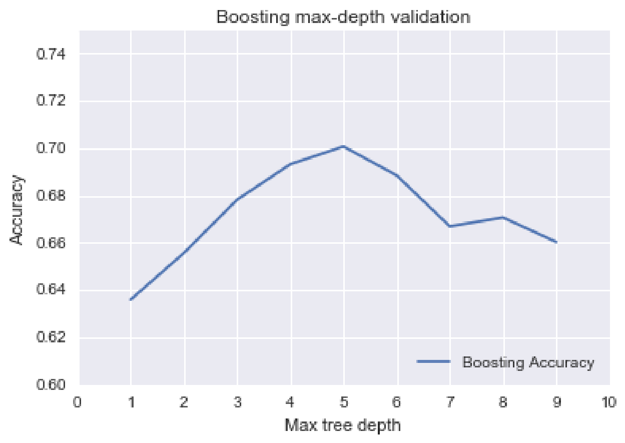
(A) Decision tree validation plot (**MNIST**)



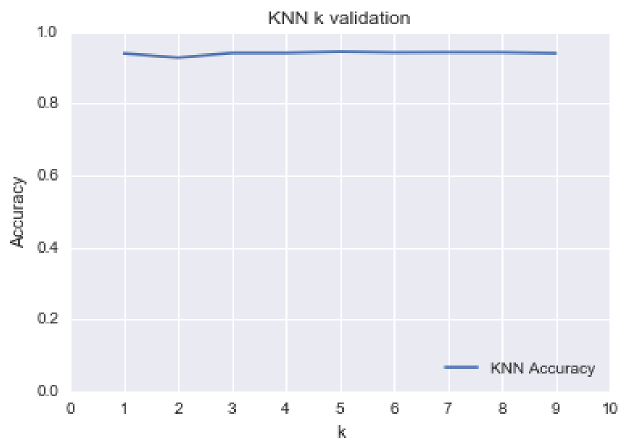
(B) Decision tree validation plot (**MV**)



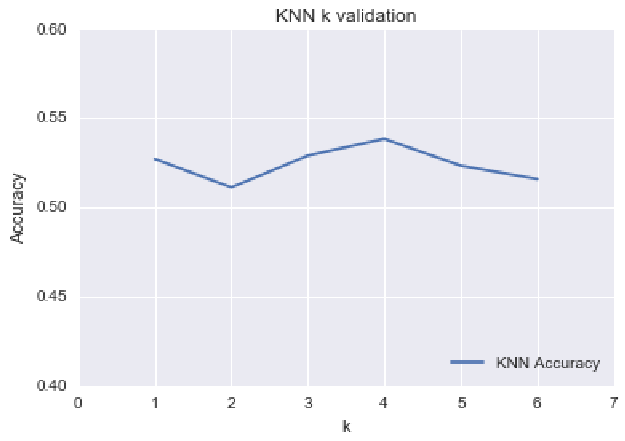
(C) Boosting validation plot (**MNIST**)



(D) Boosting validation plot (**MV**)



(E) KNN validation plot (**MNIST**)



(F) KNN validation plot (**MV**)

Boosting

I used sklearn's AdaBoostClassifier with a DecisionTreeClassifier as the underlying classifier. AdaBoostClassifier implements the AdaBoost algorithm (as presented in lectures) for binary classification (on MV), and an extension called SAMME (Zhu et al. (2009)) for multiclass classification. For validation, I once again used max depth, this time of the underlying decision tree learner. The validation plots for MNIST and MV are presented in **Figures 1(C)** and **(D)**, respectively. Interestingly, while the MV validation plot supports the suggestion in the assignment description that we might prune the underlying trees used in boosting more aggressively than an unboosted tree, the MNIST validation plot shows that this is not a hard and fast rule: the deeper our tree gets, the better the boosting performance. I conjecture that this difference is due to the relative sparsity of useful "prototypes" or "clusters" in the MV dataset as compared to MNIST:

- For MNIST no matter how we sub-sample our data, we will end up with useful prototypes (e.g., most 3s look like a 3), each of which may take several DT levels to flesh out.
- For MV, important features are sparser and sub-sampling will have a big impact on which features are relevant. It seems the most relevant ones can be captured in just a few splits.

The AdaBoostClassifier defaults to using a maximum of 50 estimators (iterations), which I did not initially perform validation over. However, when it came time to plot learning curves (next section), I realized that this was an important parameter. I was therefore able to further improve boosting performance on both datasets by doubling the number of iterations to 100 (you may view the learning curves in the next section as a validation curve for this purpose).

KNN

For KNN, I used sklearn's KNeighborsClassifier, performing validation on the "k" hyper-parameter. In both cases, I use the 'ball_tree' algorithm for indexing the data.[3] The validation plots for MNIST and MV are in **Figures 1(E)** and **(F)**, respectively. Although results are discussed in detail later, it's worth noting now just how well KNN does on MNIST for all values of k tested, and how poorly it does on sentiment classification: this supports my conjecture about the relative density of useful "prototypes" in MNIST vs MV discussed in "Boosting" above.

SVM

For SVM, I used sklearn's SVC module. MNIST is a multi-class classification problem, but basic SVMs only separate two classes. According to sklean's documentation, multi-class problems are handled via a one-vs-one scheme. According to Manning et al. (2008), one-vs-one schemes for n classes involving building n(n-1)/2 SVM classifiers (one for each pair), and then choosing the class that receives the greatest margin above the others. Here, that means sklearn builds 45 SVM classifiers for MNIST. By contrast, MV is a binary decision, and requires only a single SVM classifier. Despite this, it took significantly longer to train the SVC model on MV than on MNIST, possibly because although there is only one classifier, it is much larger than any individual classifier on MNIST.

SVM validation was first done over the type of kernel with the default parameters, obtaining the following results (measured in accuracy on the validation set):

	MNIST	MV
Linear	**0.9148**	**0.7298**
Polynomial (degree=3)	0.1136	0.5037
RBF (c=1, gamma=1/no_features)	**0.905**	**0.6060**
Sigmoid	0.1128	0.5037

3 This is just an indexing algorithm and has no impact on classification performance. Here is an excellent benchmarking the different choices for speed: https://jakevdp.github.io/blog/2013/04/29/benchmarking-nearest-neighbor-searches-in-python/.

Polynomial and sigmoid kernels failed. I haven't developed good intuitions for those ones, so I can't give a good reason why. In any case, I proceed to do validation on both the linear and rbf (gaussian) kernels, using the "C" parameter, which controls the amount of regularization (e.g., effect of outliers), and, for the rbf kernel, the gamma parameter. A higher C decreases the amount of regularization and allows the SVM to better fit the model. Gamma functions as the inverse of the variance of the gaussian kernels defined around each point, which impacts the amount of complexity the SVM can capture, but I find this very hard to visualize once you consider that MNIST inputs are not 2-dimensional, but rather 784-dimensional. Although the linear kernel outperformed the rbf kernel when using the default parameters, it showed similar results for different values of C. The performance of the rbf kernel, on the other hand, varied significantly when the C and gamma parameters were changed, and I was able to surpass the linear kernel on the both datasets. To find good parameters, I searched a 5x5 grid of (C, gamma) pairs---the results are reported below.

Dataset	**Best Kernel**	**Best (C, gamma) pair**	**Validation Accuracy**
MNIST	rbf	(50, 0.05)	0.9568
MV	rbf	(100, 0.001)	0.7439

Neural Networks

I used Keras to build a simple fully connected neural network with two hidden layers, ReLU activation functions, no regularization features, and a cross-entropy loss function. This same architecture was used for both datasets, along with early stopping based on validation loss. Optimization was done with Adam (Kingma & Ba (2014)) using default parameters (learning rate = 0.002), which tends to perform well enough in my experience so as to not require heavy validation / complex annealing schedules. While in practice I might do validation with various architectures, optimizers, regularizers, learning rates, etc.,[4] I decided to keep it simple and only validate the hidden layer size, setting both hidden layers to have the same size. The results are shown below (measured in accuracy on the validation set).

	Hidden layer sizes					
	100	200	350	500	750	1000
MNIST	0.9204	0.9356	**0.9512**	0.9432	0.9432	0.9452
MV	0.7298	0.7195	0.7298	**0.7401**	0.7354	0.7373

Final chosen hyper parameters

	MNIST	**MV**
Decision trees	max_depth = 11	max_depth = 18
Boosting	max_depth (underlying) = 11 n_estimators = 200	max_depth (underlying) = 5 n_estimators = 100
KNN	k = 5	K = 4

SVM	kernel = 'rbf'
	C = 50
	gamma = 0.05
		kernel = 'rbf'
		C = 100
		gamma = 0.001

2-layer neural network	hidden_layer_size = 350	hidden_layer_size = 500

4 I'm a bit of a neural network aficionado—if you're interested in RNNs or in Tensorflow, maybe you will find something interested on my blog @ r2rt.com.

C   Learning Curves

Using the best validation parameters found in my hyper parameter search, I plotted the learning curves for each learning algorithm. As certain algorithms are iterative (boosting, SVM, and neural networks), I have also plotted the learning curve using the number of iterations on the x-axis. The plots for each of the MNIST and MV datasets are shown in **Figure 2**.

Decision Trees – Decision tree curves are plotted in Figure 2 (A)-(D) both with and without post-pruning. We can see that post-pruning brings the lines much closer together. In my view, however, the post-pruned plot can no longer be used to judge the bias and variance of the model, and is not comparable to the other learning curves plotted. The reason is that the validation line is not longer out-of-sample due to post-pruning directly relying on the validation set (as opposed to indirectly using the validation set during validation). From the pre-pruning plots, we can see that on MNIST, the decision tree classifier suffers from high variance and overfits the training data. Nevertheless, we know from our validation plot (Figure 1(A)) that decreasing the decision tree's expressiveness by imposing a smaller max_depth hurts validation performance. Therefore, I think the best we can do here is use post-pruning and gather more data.

I note that pre-pruned learning curve for MV (Figure 1(B)) has an odd inversion in the training accuracy. This could be due to either (1) non-uniformity of the data (e.g., the latter half of the training set is easier than the first half), or (2) incorrect early splitting due to the greedy algorithm that is corrected when more data arrives. I would wager the latter, because the former should also be reflected in learning curves for other algorithms, but is not.

Boosting – Boosting learning curves are plotted in Figure 2 (E)-(H) both over the sample size used and the maximum number of iterations. The latter are very similar to validation plots, and resulted in me increasing the maximum number of iterations to use at test (to 100 on both datasets, from the default of 50).

The learning curves plotted against sample size indicate that variance is still decreasing as the amount of data increases, and so more data would be beneficial. With respect to the bias, we once again see that MNIST is a much easier dataset than MV (boosting can memorize MNIST at all data sizes, but gets progressively worse on MV as datasize increases).

KNN – KNN learning curves are shown in Figure 2(I) and (J). Both are fairly flat, but for what seems to be two opposite reasons. The MNIST curve is flat because the effective dimensionality the data is low enough that even a few samples provide closely clustered neighbors for each class, whereas the MV curve is flat because the data is so sparse that very few of the points are close together, so that the neighbors are not very informative at all. Although KNN seems to do much better on the training set for MV, this is because 1 out of 4 of the nearest neighbors are the datapoint itself, which has the correct label, and its inclusion in the 4 nearest neighbors heavily biases the prediction towards the correct one.

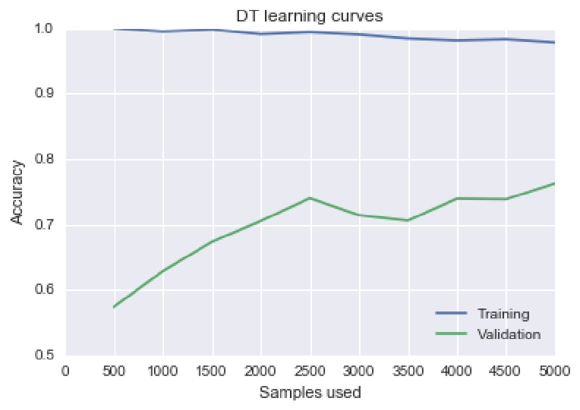
We notice that although flat, both of the training curves slope upwards, which is different from the other learning algorithms. This is because more data provides more potential neighbors, which should, in general, improve performance.

SVM – SVM learning curves are in Figure 2(K)-(N). Interestingly, on MNIST, the curves plotted by iterations flattens out completely before convergence, which suggests that we might even stop the SVM algorithm early for the same results. On MV, the curves continue to rise as the algorithm converges, which suggests that not setting a maximum on the number of iterations (default, and what was used during validation) is best.

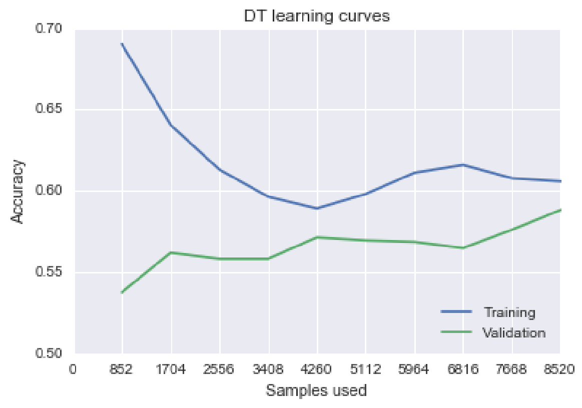
The curves plotted by samples are similar to the boosting curves, and the same comments apply. I would note, however, that the curve for MV seems to flatten out entirely, and so it's not clear that extra MV data would beneficial here.

Neural Network – The neural network learning curves are plotted in Figure 2(O)-(R), the curves with iterations on the x-axis (measured in epochs of the training data) are early stopped (to the best epoch, after two epochs of decline), and don't show the decline on the validation set that would occur thereafter due to overfitting. The curves with data size on the x-axis indicate that more data would be helpful for MNIST, but not necessarily for MV.

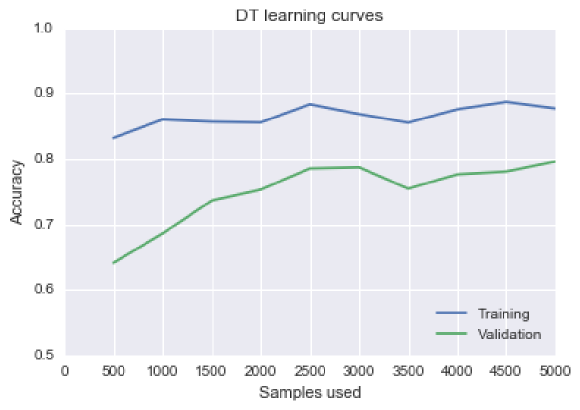
Figure 2. Learning curve plots



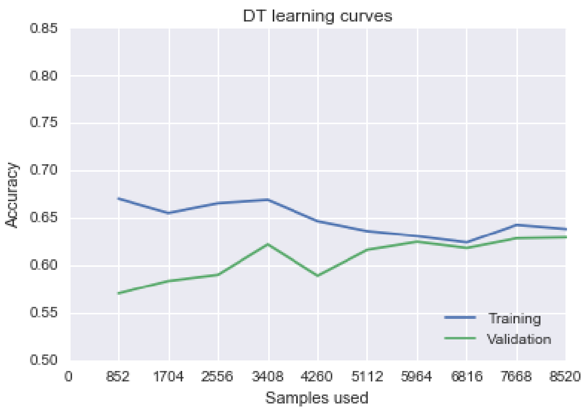
(A) DT learning curve (**MNIST**): *no post-pruning*



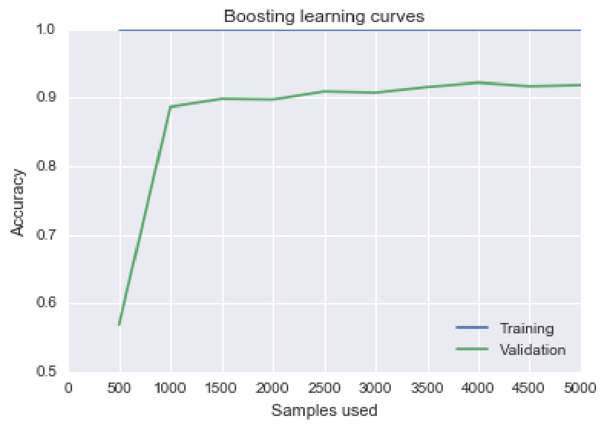
(B) DT learning curve (**MV**): *no post-pruning*



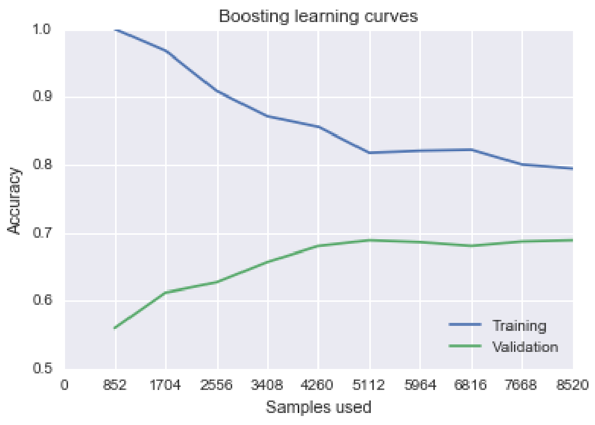
(C) DT learning curve (**MNIST**): *post-pruned*



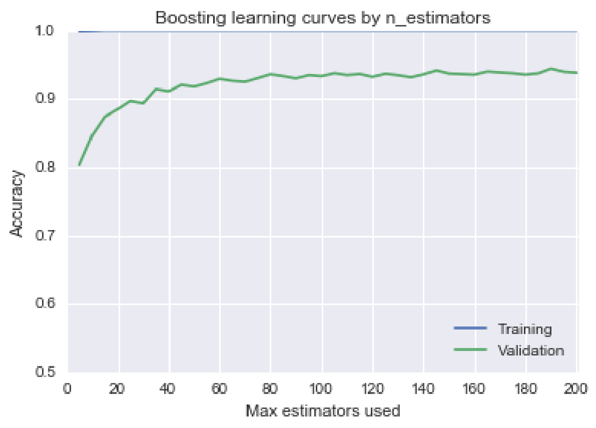
(D) DT learning curve (**MV**): *post-pruned*



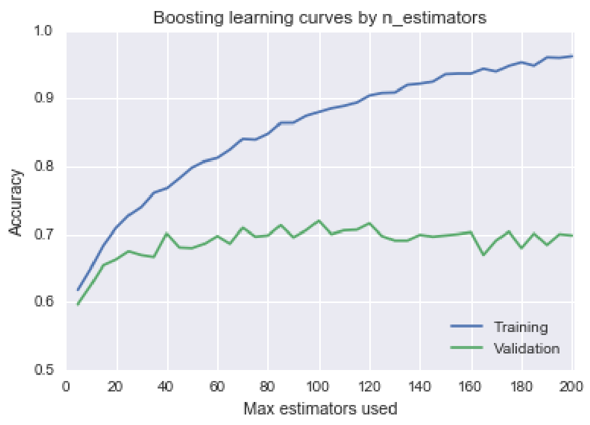
(E) Boosting learning curve (**MNIST**): *samples*



(F) Boosting learning curve (**MV**): *samples*

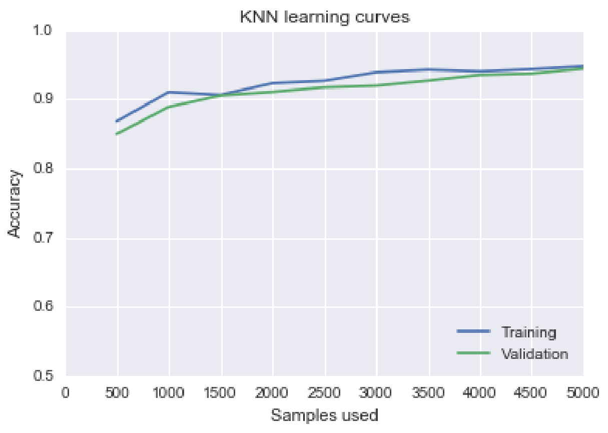


(G) Boosting learning curve (**MNIST**): *iterations*

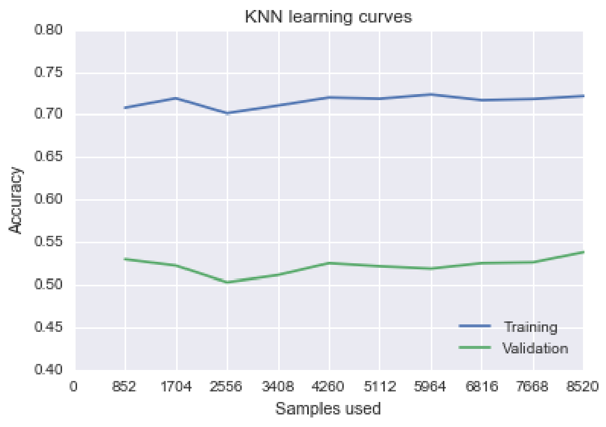


(H) Boosting learning curve (**MV**): *iterations*

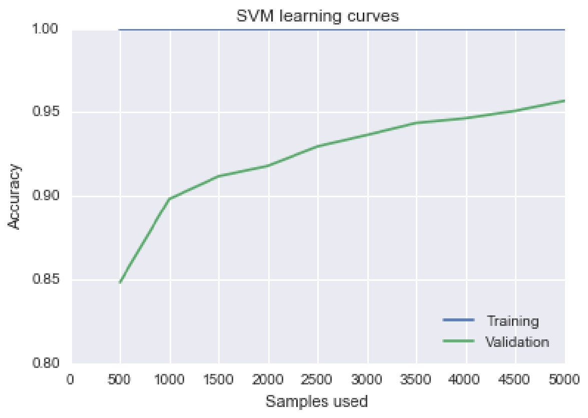
Figure 2 (continued). Learning curve plots



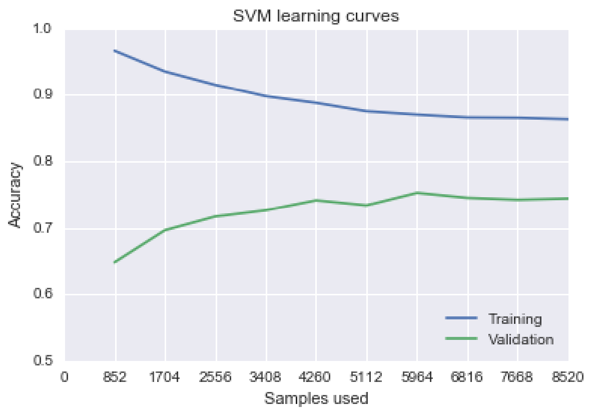
(I) KNN learning curve (**MNIST**)



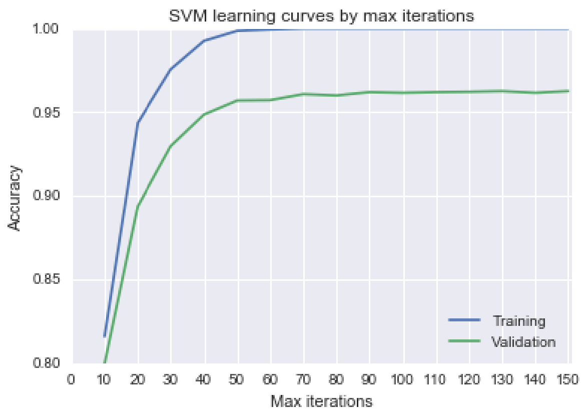
(J) KNN learning curve (**MV**)



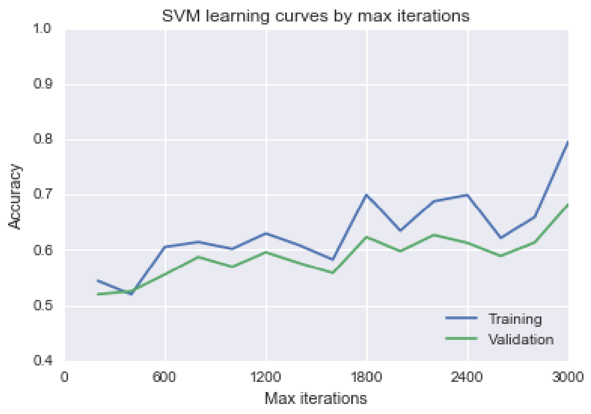
(K) SVM learning curve (**MNIST**): *samples*



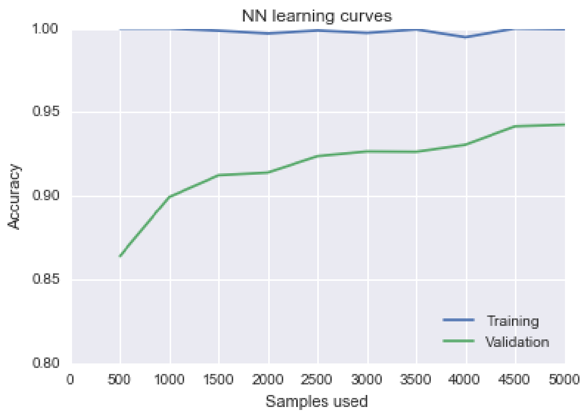
(L) SVM learning curve (**MV**): *samples*



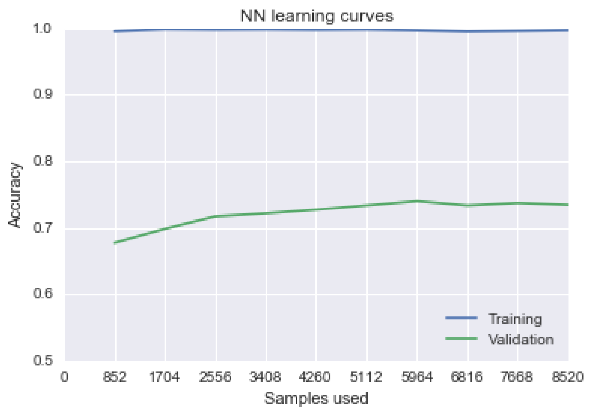
(M) SVM learning curve (**MNIST**): *iterations*



(N) SVM learning curve (**MV**): *iterations*

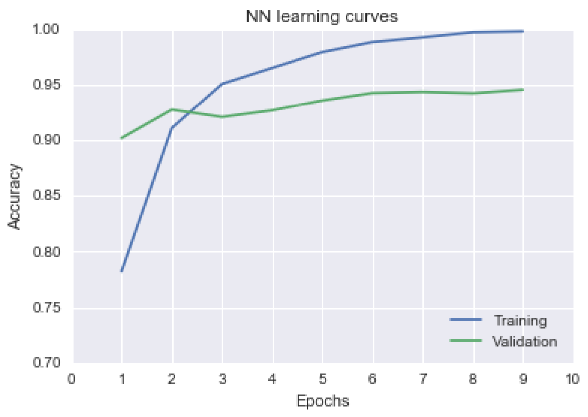


(O) NN learning curve (**MNIST**): *samples*

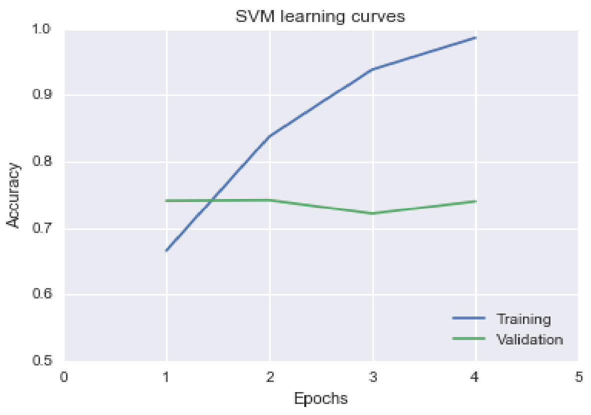


(P) NN learning curve (**MV**): *samples*

Figure 2 (continued). Learning curve plots



(Q) NN learning curve (**MNIST**): *iterations*



(R) NN learning curve (**MV**): *iterations*

Note on regularized models – Of the five models used, boosting and SVM can be considered "regularized" models. Boosting is regularized insofar as subsampling resembles bagging (this is like dropout for neural networks) and insofar as it focuses on the worst examples and therefore does not overfit the best examples (this can be related to weight decay in neural networks, which also prevents overfitting correct examples). SVMs are regularized due to the regularization factor in their cost function, which resembles weight decay in neural networks and is controlled by the "C" parameter. Interestingly, if we compare the learning curves for boosting and SVM, the shapes are almost identical on both MNIST and MV. NB how the (unregularized) neural network is able to (over)fit the MV training data perfectly. Although I do not test this, I think we can be fairly certain that a regularized neural network (e.g., with one of weight decay, dropout, stochastic activations, weight noise, etc.) would display a learning curve much like that of the SVM and boosting—due to the regularization it would no longer be able to fit the training set perfectly, and in exchange, it may see an improvement in generalization performance.

D *Training Times*

In terms of wall clock time, KNN, boosting, and SVMs took the most amount of time to train and validate. The table below shows the rough combined time for training and inference at the best hyper parameters. In addition to the times shown below, I note that boosting and SVMs had the greatest range of hyper parameters over which to perform validation, which made them take even longer. Historically, neural networks were probably among the harder models to train, but you can see that on such simple datasets, neural network training is basically free on a modern graphics card. It's unclear to me if it would even be possible to scale boosted decision trees or SVMs to a dataset the size of, say, ImageNet.

Hardware used, for reference: i5-6500 @ 3.2GHz, GeForce GTX 980 Ti (for NN only)

	MNIST	**MV**
Decision Tree	`<5 s`	`<5 s`
Boosted DT	`115 s (200 est.)`	`65 s (100 est.)`
KNN (includes indexing time)	`50 s`	`210 s`
SVM	`70 s`	`270 s`
Neural Network	`<5 s`	`<5 s`

The assignment also asks about the number of iterations, but I'm not sure why this is relevant, since number of iterations is not comparable across algorithms. Please see the x-axis of the learning curves for the # of iterations if you are curious. The most interesting observation I have on this is that SVM takes an order of magnitude more iterations to converge on the MV dataset, and this is likely due to the higher effective dimensionality (discussed throughout this report).

4 Results and Further Analysis

Using the best validation parameters found in my hyper parameter search, I ran the models on the hold-out test sets. The results (measured by accuracy = number_correct / size_of_test_set) for MNIST and MV are shown below, alongside the MNIST-CONV and MV-EMBED results.[5]

	MNIST	MNIST-CONV*	MV	MV-EMBED*
Decision Tree	0.7558	0.8054	0.6285	0.6180
Boosted DT	0.9300	0.9370	0.6704	0.6976
KNN	0.9244	0.9418	0.5590	0.6255
SVM	**0.9512**	**0.9618**	**0.7360**	**0.7584**
Neural Network	0.9360	0.9492	0.7238	0.7416

A Analysis of Basic Results

On MNIST and MV, the ranking of algorithms is almost identical: SVMs achieved the highest generalization performance in both cases, with the 2-layer neural network as a close second and the boosted decision tree in third place. Decision trees seem to be dominated by boosting, which makes sense, since boosting uses a decision tree as the underlying learner (if we use the same decision tree, it would very unlikely (perhaps impossible?) for boosting to do worse).

Although I cannot compare my MNIST results to published results (because I am using different splits), the results on MV are somewhat comparable. With the caveat that the literature generally uses a 10-fold cross validation (and reports average validation loss)---which means that they take advantage of 10% more training data than I do, and also have less variance in the reported results---my results with MV-EMBED are not too far off from state-of-the-art results of approximately 81% (see Kim (2014)). With a better recurrent filter, some parameter tinkering, and use of 10-fold CV, I might be able to match that result.

In my mind, despite the SVM's better accuracy, the neural network takes the cake—if a simple fully-connected networked with no regularization can almost match the SVM (and in ~1/100th of the training time), I'm willing to bet that a properly regularized neural network can surpass the SVM results. In fact, having written this, I went back to validate the assertion (yes I know, I'm cheating because I'm training something after running the test set, but it's for science), and I was able to edge out the SVM results (barely) on all datasets without too much validation (using batch normalization on MNIST and dropout on MV).

The only really surprising result for me was how powerful KNN was on MNIST. This hits on the discussion in Section 3(B) – Boosting, where I note that the effective dimensionality of MNIST is very small, even it is represented in 784 dimensions. Thus, the curse of dimensionality does not kick in, and KNN does very well.

B Areas for Improvement

I have detailed a number of areas where things can be improved throughout. The biggest point was in Section 2, where I commented on the data representations, and how we might better choose features that take into account known spatial invariance (for MNIST) and known sequential dependence (for MV). The results of my experiments on this are discussed in the next section.

Another point was that, due to laziness, I only did validation, not cross-validation. This was explained in Section 3(A). For the reasons stated there, I do not think CV would have improved results significantly, but it may have helped.

5 I performed a similar validation process on MNIST-CONV and MV-EMBED, but more minimalistic. Most hyper parameters that worked on MNIST also worked on MNIST-CONV. MV saw a few differences when shifting to MV-EMBED, which is expected, given that they are two very different data representations. You can check out my work on those datasets in the Jupyter notebooks, if you are interested.

Another point was on my very narrow hyper parameter search on the neural network front. I somewhat corrected this in Section 4(A), but I think the NN results could be improved by better use of regularizers and advanced architectures.

Finally, in Section 3(C) I made several remarks about algorithms which more data would or would not be beneficial to collect. On MNIST in particular, it seems that the more data we have, the better results we can get. This is confirmed by the MNIST leaderboards,[6] which show that advanced neural networks, SVMs, boosting, and KNN models can all score above 99% on the hold out test set (not the same as used for my experiments).

*C Analysis of Transformed Results**

Check those out! With the exception of of the MV decision tree (meh), using unsupervised filters on the data improved performance on both datasets for all methods, and by respectable margins! And this was accomplished with dead simple autoencoders of my own design. I think it is very likely that the results could be further improved with better filters (in fact, I know this to be true because I had initially started with a supervised convolutional filter, before I decided that this felt too much like cheating).

The most interesting improvement to me is the nearly 7 point improvement of KNN on the MV dataset. The recurrent embedding filter compressed the original sparse data into a dense 500 feature vector that brought the classes closer together in space than before.

5 Bibliography

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Dai, A. M., & Le, Q. V. (2015). Semi-supervised sequence learning. In *Advances in Neural Information Processing Systems* (pp. 3079-3087).

Freund, Y., & Schapire, R. E. (1995, March). A desicion-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory* (pp. 23-37). Springer Berlin Heidelberg.

Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.

Kingma, D., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval* (Vol. 1, No. 1, p. 496). Cambridge: Cambridge university press.

Pang, B., & Lee, L. (2005, June). Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *Proceedings of the 43rd annual meeting on association for computational linguistics* (pp. 115-124). Association for Computational Linguistics.

Pennington, J., Socher, R., & Manning, C. D. (2014, October). Glove: Global Vectors for Word Representation. In *EMNLP* (Vol. 14, pp. 1532-43).

Socher, R., Perelygin, A., Wu, J. Y., Chuang, J., Manning, C. D., Ng, A. Y., & Potts, C. (2013, October). Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the conference on empirical methods in natural language processing (EMNLP)* (Vol. 1631, p. 1642).

Zeiler, M. D., Taylor, G. W., & Fergus, R. (2011, November). Adaptive deconvolutional networks for mid and high level feature learning. In *Computer Vision (ICCV), 2011 IEEE International Conference on* (pp. 2018-2025). IEEE.

Zhu, J., Zou, H., Rosset, S., & Hastie, T. (2009). Multi-class adaboost. *Statistics and its Interface*, *2*(3), 349-360.

6 http://yann.lecun.com/exdb/mnist/