P5: Capstone Project: Digit Recognizer (kaggle)

Problem Definition

Define Problem:

This project will focus on image recognition, in particular on recognition of handwritten digits (0-9) from the MNIST (Modified National Institute of Standards and Technology) data set. The author will enter into the kaggle Digit Recognizer competition in order to validate project results independently as well as comparing the results with other kaggle competition participants.

Handwritten character recognition is an important part of machine learning, it is one of the pivotal bridges between human and computer interfaces, character recognition can be used in many applications including USPS automatic scanning of zip codes on mail, digitisation of handwritten documents, handwritten input of data from tablets and other devices.

Handwritten digit recognition is a simpler subset of this problem that is a good place to start our exposure to real world problems in machine learning.

In this project we will be using Python 2.7 (64 bit), sklearn, numpy, scipy and matplotlib.

We will analyze the dataset to see what type of preprocessing is needed before we continue. Next we will select a number of sklearn classification models and run them in default modes to identify models that can fit to data best. As the next step we will select one or more of the best models and optimally tune the parameters for the given model to improve the model performance even more. We will also fine tune preprocessing from step one and/or introduce other preprocessing steps as we learn still more about the dataset.

Describe a Solution & Identify Metrics:

In a perfect world the solution will be to output the exact corresponding digit for each of the handwritten input digits; however, in the real world 100% accuracy is not guaranteed

a percentage of correctly identified digits will serve as a metric to gage performance. Kaggle's leaderboard for digit recognizer contains 870 (at the moment of this writing) participating entities (individuals/teams) with a total of 3328 entries, with scores ranging from 0.00000 to 1.00000 (1.0 constitutes 100% success rate). The author will try to get into the top 100 results or better (100th result at this writing was 0.98986). One note, Kaggle is using classification accuracy for their score (basicly percentage of example predicted correctly), and this is what we will use to evaluate our final results; however, internally, while evaluating algorithms before Kaggle submissions we will be using F1 score to gage our progress. While for Kaggle using classification accuracy score is fine, internally F1 score will better meet our needs. F1 score looks at both precision and recall as well as takes a weighted average for each class. While classification accuracy score will just look at total accuracy. For example if we had an unbalanced data where one class (class B) was only 1% of the total data (10 total classes) and we can predict all classes with 100% accuracy, except on class B where we fail completely anca can't predict a single instance. Classification accuracy score for this example will be 99%, where F1 score for the same example must be something less than 99%, therefore F1 score is paying more attention to each class vs. classification accuracy score does.

Exploratory Analysis

Analyze the Problem:

The kaggle MNIST dataset contains train dataset consisting of 42,000 images (train.csv) and test dataset consisting of 28,000 images (test.csv). Each image is 28 by 28 pixels, translating into 784 pixels of features. Each pixel feature value is an integer from 0 to 255, indicating lightness or darkness of the pixel. The train dataset also contains the image label as the 1st column, the test dataset does not.

Above is a sample of 100 images from the training dataset.

Above is a sample of an instance of a digit “1” from training data set. This very same digit is represented below in a 2d array of 28 by 28 pixels. Each integer in the array represents a pixel, each pixel is a feature of the image, there are a total of 784 features or pixels for each image. The features/pixels with “0” value represent blank space (no

writing/markings in this space), the features/pixels with “1-254” values represent markings, where value of “1” is the lightest and value of “255” is the darkest pixel. The darkest pixels for each image represent more relevant pixels (the features that represent the core of a given image).

```
[[ 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   3 141 139   3   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   9 254 254   8   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   9 254 254   8   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   9 254 254 106   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   9 254 254 184   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   9 254 254 184   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   9 254 254 184   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   6 185 254 184   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   0  89 254 184   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   4 146 254 184   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   9 254 254 184   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   9 254 254 184   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   9 254 254 184   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   9 254 254 184   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   9 254 254 184   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0 156 254 254 184   0   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0 185 255 255 184   0   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0 185 254 254 184   0   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0 185 254 254 184   0   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0  63 254 254  62   0   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [ 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]]
```

There are a total of 10 unique labels/digits (0-9) refer to below table.

Labels with number of instances of each in the training dataset									
0	1	2	3	4	5	6	7	8	9
4,132	4,684	4,177	4,351	4,072	3,795	4,137	4,401	4,063	4,188

Notice how the classes are not balanced, label 5 class has only 3,795 (min) instances while label 1 class has 4,684 (max) instances that is a difference of 889 instances or ~19%. And all the other classes fall somewhere in between. Class instance count mean is 4,200 and class instance count standard deviation is 224.92. This is not optimal as some models are particularly sensitive to unbalanced class datasets. And might

increase classification error for classes that are underrepresented. We will need to balance the data or use class_weight parameter for models that allow for it.

Identify Suitable Machine Learning Algorithm(s):

The digit recognizer is a discrete (the output is 10 integer states) classification problem. We are dealing with a relatively large data set (42,000 training and 28,000 testing) with 784 features or dimensions. Given this information we first need to look at classifier models and second at models that can handle large dimensionality, models that will not suffer too much from poor performance given large number of features. The classification is overlapping (most of the classes will share pixels to a large extent, for example 5 looks very much like 6 or 8 can look like 9).

In this project we will consider the following machine learning models:

Naive Bayes - GaussianNB
Advantages: works well with noise and overlapping data (classification overlapping), empirically successful, gold standard, inference is cheap (fast)
Disadvantages: assumption of independence of features
Why selected: the algorithm should be fast, and should do relatively decent job modeling the problem, a good first model to run without tweaking parameters

Decision Tree - DecisionTreeClassifier
Advantages: simple to understand and interpret, able to handle both numerical and categorical data, O(logn) so relatively fast
Disadvantages: can create over complex trees, easily prone to overfitting, no guarantee of optimal results, if some classes dominate, tree will create biased results
Why selected: a straightforward model with a number of parameters to tweak to see how well the model does and see how difficult the problem might be, the bottom line is it will be interesting to see how well this model can be optimized.

Support Vector Machines (SVM) - SVC
Advantages: effective with large number of dimensions, versatile (kernel functions), works very well in complicated domains
Disadvantages: slow O(n^2), does not work well with lots of noise, prone to overfitting
Why selected: should work very well for this type of problem (many dimensions), but will be slow, out of all the model that will be tested this one seems the most likely to provide the best results, but will need lots of parameter tweaking and pre-processing.

Nearest Neighbors (K-NN) - KNeighborsClassifier
Advantages: fast train time (learning process cost is zero), complex concepts can be learned by local approximation
Disadvantages: slow prediction time, frequent class tends to dominate prediction, slow on large datasets, curse of dimensionality
Why selected: even though it might be slow with large number of dimensions in our problem we can use PCA to overcome the dimensionality issue, we will used K-NN to have a broader bundle of models to try.

Logistic Regression - LogisticRegression
Advantages: works well as long as features are roughly linear, robust to noise and overfitting can be avoided, output can be interpreted as probability
Disadvantages: fundamentally a binary classifier
Why selected: a good base algorithm to try, also will be using RBM with logistic regression to try out RBM to hopefully increase logistic regression performance.

Solution Implementation

Pre-processing Data:
Dimensionality of the data is large and most likely will hinder performance and add noise, for this reason we will use Principal Component Analysis (PCA) to transform the data into fewer dimensions. PCA dimensions that will show up first are the ones that explain the maximal variance most. PCA will focus more on global picture of the dataset, features that define the dataset the most. One challenge is finding the optimal number of components for PCA to derive (we will start with 50 components and later find the optimal number of components, refer to PCA Table #1 & 2 in the later section), we want to decrease the number of dimensions/features, but also keep most of the variance between data.

Additionally Restricted Boltzmann Machine (RBM) will be used instead of PCA on logistic regression model to see how well it does (again to reduce dimensionality), on top of that RBM will be used with the winning model to determine if it might be better suited than PCA (refer to RBM with SVM Table #1 & 2 in the later section). RBM is based on probabilistic model that learns nonlinear features where PCA is calculating maximally variant orthogonal components, extracted RBM features can be feed into other classifiers much like PCA.

Scaling features, MinMaxScaler will be used to decrease standard deviation between features and preserving zero entries (feature range of -1,1 will be used as it seems to return better results when used with given models). MinMaxScaler is particularly important to use with SVM as running SVM without MinMaxScaler drops F1 score to almost zero vs >0.98 with MinMaxScaler. Referring back to the 2d array of 28 by 28 pixels (from exploratory analysis section), one can see that pixel values are 0 to 255. What happens is that SVM (without MinMaxScaler) is paying too much attention to different values in between “0” and “255”, where one can argue that the important pixels are the ones with pixel value “0” and anything else, whether the pixel has a value of 255 or 245 is not that important. MinMaxScaler reduces that range significantly range of -1 to 1 in this case.

Additionally pixel nudging will be used to increase the training dataset (image pixels will be moved up, down, right, left by 1 or 2 pixels), original training data will also be preserved and combined with nudged data. The way image recognition works is that the model looks at image pattern (pixel pattern) in relation to the image canvas NOT in relation to other pixels directly, this means that offsetting the pixels even one pixel in any direction will mess with the image recognition model. This means that a handwritten digit written in exactly the same way, but slightly to the right or the left will look different to the classification model, and therefore might not be assigned the same class. For this reason a simple transformation of an image, like nudging the pixels, does constitute additional data for training, as it will prepare the model for slight offsets of a new image otherwise almost identical to sample already in the dataset.

Implementing and Measuring Performance:

Tables for initial runs are below each model has been run for dataset size of 1,000, 5,000, 10,000 and 42,000 (full train set). Train test split has been used each time of 20% of the dataset run. All runs have used preprocessing (MinMaxScaler). Runs annotated appropriately in addition to MinMaxScaler used PCA with n_components = 50.

We will be looking for train & test f1-scores to evaluate our accuracy (test scores) and evaluate potential overfitting issues (train scores). Even though running time is not our primary concern (as we are ranked on Kaggle by accuracy), running time is still something we should keep track of, particularly when dealing with larger datasets (for example when we do pixel nudging, that will expand our dataset significantly, we want the running time to be reasonable, if for no reason, but to finish the project in a timely manner. Additionally in the real world situation at least the prediction running time needs

to be relatively quick (in near real time is probably advisable), imagine if we end up using this algorithm for handwritten digit recognition on a handheld device, in that situation the recognition of digits one writes needs to be as fast as the digits being written (as far as human is concerned) to be useful.

GaussianNB

Model 1:	GaussianNB - default			
Preprocessing:	MinMaxScaler			
Dataset Size	1,000	5,000	10,000	42,000 (full)
Train f1-score	0.7188	0.5643	0.5762	0.5249
Test f1-score	0.6020	0.5549	0.5264	0.5267
Train time (sec)	0.015	0.387	0.180	0.780
Pred. test time (sec)	0.048	0.110	0.197	0.769

Model 2:	GaussianNB - default			
Preprocessing:	MinMaxScaler, PCA(50)			
Dataset Size	1,000	5,000	10,000	42,000 (full)
Train f1-score	0.8970	0.8792	0.8752	0.8732
Test f1-score	0.8018	0.8596	0.8652	0.8732
Train time (sec)	0.000	0.000	0.007	0.036
Pred. test time (sec)	0.000	0.020	0.007	0.047

From running the GaussianNB model one can see that model 1 actually decreased in accuracy from 1,000 to 42,000 dataset and that max f1-score was only 0.6020 on test dataset. Model 2 using PCA had substantially better results with a max f1-score of 0.8732 or about 0.27 improvement over non-pca model. Additionally model 2 training time was about 10 to 20 times faster than model 1, still both models as expected are very fast. One should note that PCA fit & transform time for the 42,000 size dataset takes 3.753 sec, wiping out all of the gain in processing time as relative to model 1.

Note: see log for details: logGaussianNB.txt

DecisionTreeClassifier

Model 3:	DecisionTreeClassifier - default			
Preprocessing:	MinMaxScaler			
Dataset Size	1,000	5,000	10,000	42,000 (full)
Train f1-score	1.0000	1.0000	1.0000	1.0000
Test f1-score	0.5600	0.7500	0.8077	0.8539
Train time (sec)	0.084	0.578	1.542	10.553
Pred. test time (sec)	0.001	0.016	0.000	0.015

Model 4:	DecisionTreeClassifier - default			
Preprocessing:	MinMaxScaler, PCA(50)			
Dataset Size	1,000	5,000	10,000	42,000 (full)
Train f1-score	1.0000	1.0000	1.0000	1.0000
Test f1-score	0.6116	0.7357	0.7646	0.8276
Train time (sec)	0.038	0.246	0.634	4.157
Pred. test time (sec)	0.000	0.000	0.000	0.000

After running the decision tree model the first thing that stands out is that the model is overfitting from the start the training f1-score is 1.0 even with dataset size of 1,000 additionally one can see substantial improvement in scores from 1,000 to 42,000 dataset size from 0.5600 to 0.8539 respectively. Additionally one can see that PCA transformation is actually hurting the scores on the larger datasets. Even though so far the highest score was from GaussianNB (0.8732) a score of 0.8539 from decision tree classifier is not very far off particularly considering that decision tree model is suffering from overfitting (that can be mediated as we have not used any iterative parameterization). We will take note of the score and optimize decision tree model at the later stages.

Note: see log for details: logDecisionTree.txt

KNeighborsClassifier

Model 5:	KNeighborsClassifier - default			
Preprocessing:	MinMaxScaler(-1,1)			
Dataset Size	1,000	5,000	10,000	42,000 (full)
Train f1-score	0.9034	0.9516	0.9630	0.9771
Test f1-score	0.8239	0.9187	0.9521	0.9648
Train time (sec)	0.015	0.226	0.670	8.538
Pred. test time (sec)	0.170	5.900	23.559	398.376

Model 6:	KNeighborsClassifier - default			
Preprocessing:	MinMaxScaler(-1,1), PCA(50)			
Dataset Size	1,000	5,000	10,000	42,000 (full)
Train f1-score	0.9146	0.9601	0.9694	0.9826
Test f1-score	0.8331	0.9312	0.9606	0.9726
Train time (sec)	0.002	0.000	0.024	0.157
Pred. test time (sec)	0.014	0.364	1.333	26.689

The first thing we see is that the best score is substantially better for KNN (0.9726) vs GaussianNB (0.8732) and Decision Tree (0.8539) algorithms, an improvement of ~0.1. model 6 the one using PCA is almost 0.01 better than model 5 not using PCA. Also note that as the dataset increases both training and testing scores increase, and testing score is following training test score quite closely, we are not seeing much overfitting.

Note: see log for details: logKNeighborsClassifier.txt

SVC (SVM)

Model 7:	SVC - default			
Preprocessing:	MinMaxScaler(-1,1)			
Dataset Size	1,000	5,000	10,000	42,000 (full)
Train f1-score	0.9448	0.9559	0.9588	0.9699
Test f1-score	0.8648	0.9236	0.9405	0.9604
Train time (sec)	0.429	4.801	16.283	155.162
Pred. test time (sec)	0.084	1.678	5.464	61.279

Model 8:	SVC - default			
Preprocessing:	MinMaxScaler(-1,1), PCA(50)			
Dataset Size	1,000	5,000	10,000	42,000 (full)
Train f1-score	1.0000	0.9995	0.9990	0.9984
Test f1-score	0.8397	0.9477	0.9715	0.9808
Train time (sec)	0.114	2.053	6.801	58.298
Pred. test time (sec)	0.011	0.217	0.720	8.811

The SVM with the score of 0.9808 is ~0.01 better than the last best score of 0.9726 from KNN. A nice improvement considering we are only 2% away from 1.0. Note that the algorithm at 1,000 datasize starts out overfitted, but with more data slightly falls below 1.0 (train score), while the test score keeps getting better with more data. One can also see an improvement of about 0.02 in model 8 the one with PCA vs model 7 with no PCA.

Note: see log for details: logSVC.txt

LogisticRegression

Model 9:	LogisticRegression - default			
Preprocessing:	MinMaxScaler(0,1)			
Dataset Size	1,000	5,000	10,000	42,000 (full)
Train f1-score	0.99875	0.97071	0.95417	0.93215
Test f1-score	0.83606	0.89141	0.90153	0.91417
Train time (sec)	n/a	n/a	n/a	n/a
Pred. test time (sec)	0.000	0.016	0.000	0.054

Model 10:	LogisticRegression - default			
Preprocessing:	MinMaxScaler(0,1), BernoulliRBM(256)			
Dataset Size	1,000	5,000	10,000	42,000 (full)
Train f1-score	0.97244	0.96849	0.95959	0.95947
Test f1-score	0.87535	0.93102	0.94061	0.95505
Train time (sec)	4.190	21.503	44.568	191.872
Pred. test time (sec)	0.000	0.016	0.016	0.097

Top logistic regression score is 0.91417 vs 0.95505 for logistic regression with RBM. Both model 9 and model 10 test scores increased incrementally as the dataset increased; however, it looks like the training score decreased as the dataset increased one can surmise that even if we had more data to train on Model 10 test score is not going to be higher than 0.95947 given current parameters.

Note: see log for details: logLogisticRegressionBernoulliRBM.txt

Results So Far

Best Score for Each Model (default parameters) (dataset size 42,000)					
Model	PCA-NB	Tree	PCA-KNN	PCA-SVM	LR-RBM
Train f1-score	0.8732	1.0000	0.9826	0.9984	0.9595
Test f1-score	0.8732	0.8539	0.9726	0.9808	0.9551
Train time (sec)	0.036	10.553	0.157	58.298	191.872
Pred. test time (sec)	0.047	0.015	26.689	8.811	0.097

SVM with PCA is clearly in the lead with a test score of 0.9808 followed by KNN with PCA with a test score of 0.9726 followed by Logistic Regression and RBM with a score of 0.9551. We should concentrate our efforts on SVM and KNN, but we are also going to try to optimize both Tree and Logistic Regression with RBM just to see how far we can go.

One should note that Logistic Regression and SVM have the longest training time ~191 sec and ~58 sec respectively, while KNN and Naive Bayes have the shortest. While prediction time is the longest for KNN (~27 sec) followed by SVM (~9 sec). However in this project we are trying to get the highest prediction score possible to rank high on the kaggle competition, running times will not be a concern as long as they are reasonable, so as long as we can get the results running time is not an issue.

Iterating and Reflection

Parameter Grid Search

In this section we will use sklearn grid search to find the most optimal parameters for each of the models discussed above with special emphasis on SVM, the only model we will not pursue further is GaussianNB as there is not much we can do to tweak this model. Below are models with grid search parameters, best model parameters and best F1 scores for thouse parameters.

Note: see log for details: logSVCGridAndFinalScore.txt, logRBMGridAndFinalScore.txt, logBestScoreForOtherModels.txt

SVM MinMaxScaler(-1,1), PCA(50), (GridSearchCV cv=5) (dataset size 5,000)

parameters = {'C': (5,10,20,40,80), 'gamma': (0.1,0.01,0.001,0.0001), 'kernel': ['rbf']}
Best model parameters: {'kernel': 'rbf', 'C': 5, 'gamma': 0.01}
Final F1 score for test: 0.963925193244

parameters = {'C': (2,3,4,5,6,7), 'gamma': (0.015,0.01,0.007,0.005), 'kernel': ['rbf']}
Best model parameters: {'kernel': 'rbf', 'C': 4, 'gamma': 0.007}
Final F1 score for test: 0.96390982875

parameters = {'C': (3,4,5), 'gamma': (0.008,0.007,0.006), 'kernel': ['rbf']}
Best model parameters: {'kernel': 'rbf', 'C': 3, 'gamma': 0.008}
Final F1 score for test: 0.965925616825

parameters = {'C': (2,3,4), 'gamma': (0.009,0.008,0.007), 'kernel': ['rbf']}
Best model parameters: {'kernel': 'rbf', 'C': 3, 'gamma': 0.008}
Final F1 score for test: 0.965925616825

parameters = {'C': (5,10,20,40,80), 'gamma': (0.1,0.01,0.001), 'kernel': ['poly'], 'degree': [1,5,9]}
Best model parameters: {'kernel': 'poly', 'C': 5, 'gamma': 0.1, 'degree': 5}
Final F1 score for test: 0.935932063248

parameters = {'C': (3,4,5,6), 'gamma': (0.015,0.01,0.005), 'kernel': ['poly'], 'degree': [4,5,6]}
Best model parameters: {'kernel': 'poly', 'C': 6, 'gamma': 0.005, 'degree': 4}
Final F1 score for test: 0.939160485761

parameters = {'C': (6,7,8), 'gamma': (0.006,0.005,0.004), 'kernel': ['poly'], 'degree': [3,4,5]}
Best model parameters: {'kernel': 'poly', 'C': 6, 'gamma': 0.005, 'degree': 3}
Final F1 score for test: 0.956997550843

parameters = {'C': (5,6), 'gamma': (0.007,0.005), 'kernel': ['poly'], 'degree': [2,3,4]}
Best model parameters: {'kernel': 'poly', 'C': 5, 'gamma': 0.005, 'degree': 3}
Final F1 score for test: 0.958020207645

Overall best model parameters: {'kernel': 'rbf', 'C': 3, 'gamma': 0.008}
Overall best score with full dataset: 0.9819

KNN - MinMaxScaler(-1,1), PCA(50)
Similar exhaustive grid search was done for KNN as was done for SVC
Best parameters: n_neighbors=4, algorithm="kd_tree", p=2, weights='distance', n_jobs=4

Overall best score with full dataset: 0.9745

Decision Tree - MinMaxScaler(-1,1), PCA(50)
Similar exhaustive grid search was done for decision tree as was done for SVC
Best parameters: max_depth=14, splitter='best', min_samples_split=7, min_samples_leaf=5
Overall best score with full dataset: 0.8585

Logistic Regression - MinMaxScaler(0,1), RBM
Similar exhaustive grid search was done for RBM as was done for SVC
Best parameters: learning_rate=0.01, n_iter=20, n_components=200, random_state=42
Overall best score with full dataset: 0.9513

Best parameters: learning_rate=0.01, n_iter=80, n_components=200, random_state=42
Overall best score with full dataset: 0.9545

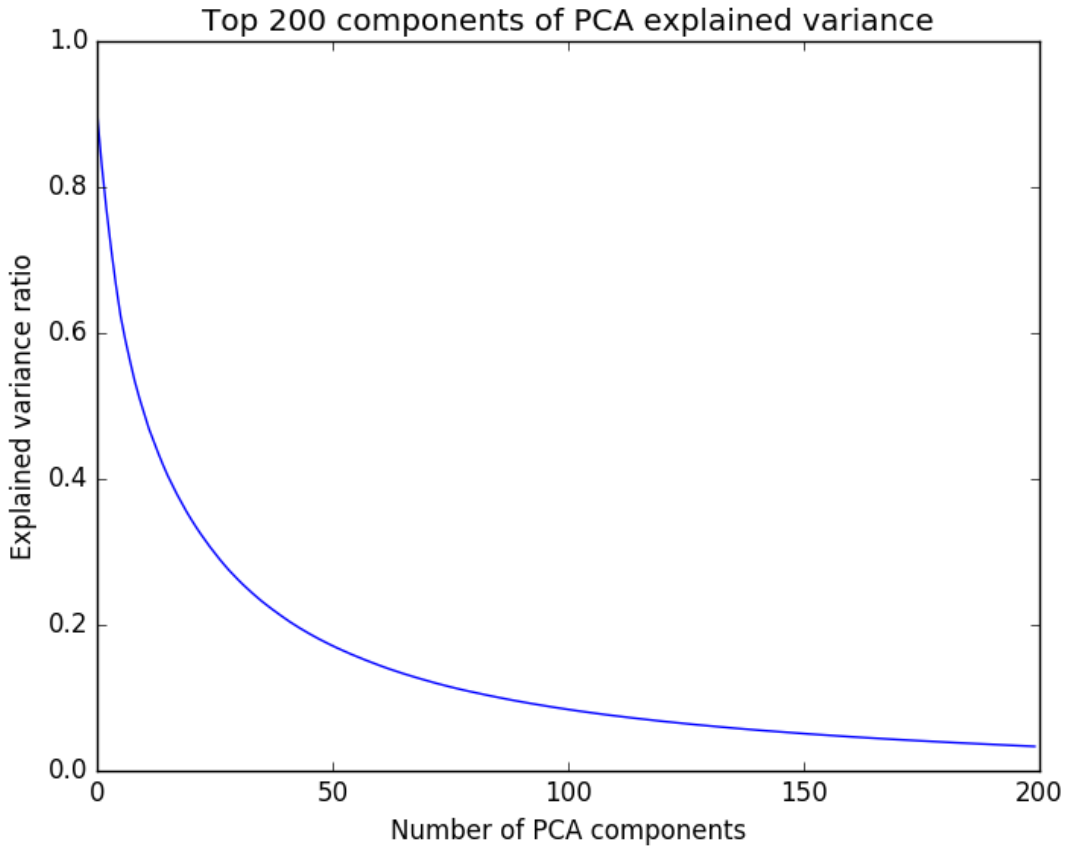
Best Score for Each Model (dataset size 42,000) - With optimal parameters				
Model	Tree	PCA-KNN	PCA-SVM	LR-RBM
Train f1-score	0.9365	1.0000	0.9984	0.9635
Test f1-score	0.8585	0.9745	0.9819	0.9545
Train time (sec)	6.930	0.164	15.949	293.937
Pred. test time (sec)	0.019	7.035	4.633	0.100

Interesting to note that none of the models improved all that much after parameter optimization for example, SVM default model (0.9808) setting to optimal model (0.9819) settings. Overall it looks like SVM did the best with a score of 0.9819 however KNN is not far behind with a score of 0.9745 (not much lower). It also looks like KNN got overfitted, so we can probably tinker with parameters a bit more to increase the test score, but our main concentration will be to improve SVM as it looks to be the best contender out of the models we tested.

Finding SVM With Best Preprocessing Set of Functions

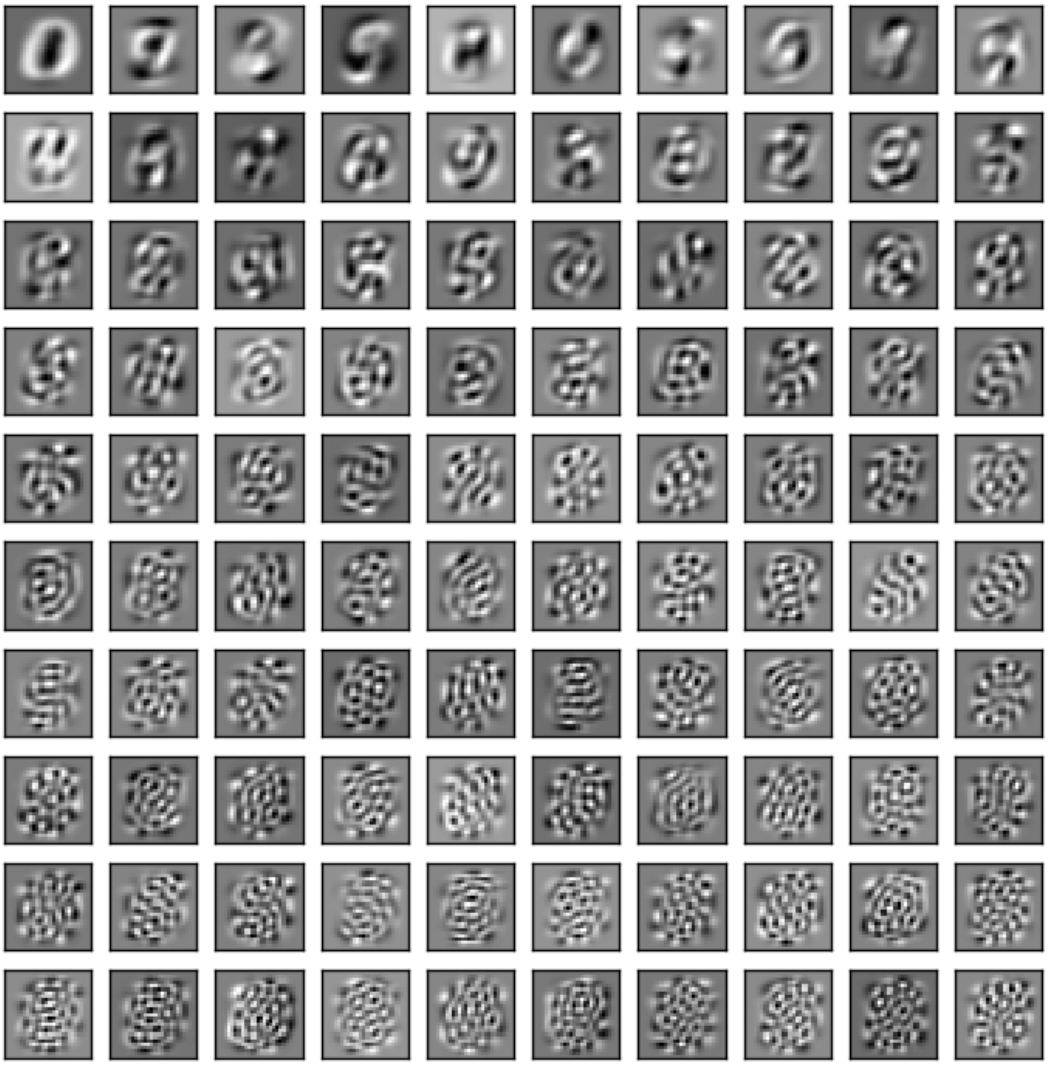
PCA

Lets try to find the best PCA components number for our SVM with the best parameters from above. We can use some visualizations to help us identify the number of PCA components as well as picture the data better. Below are top 200 components of PCA that explain the variance best. One can see that most of the variance is packed into the first 50 components; however it looks like component 50 to 100 are still relatively significant, after component 100 there is much less variance.



We can also visualize top 100 PCA components to see how they look if converted back to images.

100 components extracted by PCA



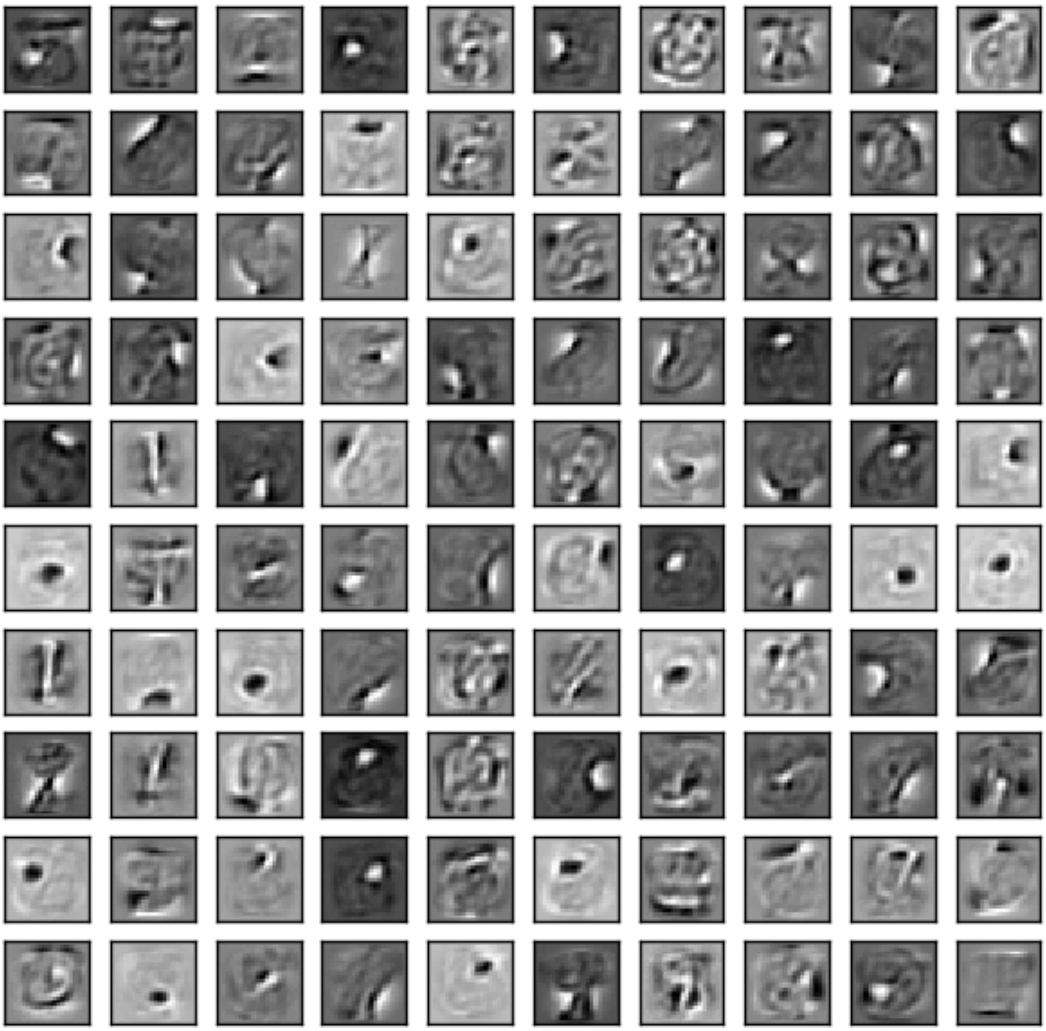
PCA Table #1 Score for SVM given PCA components (dataset size 42,000)										
PCA components	15	20	30	50	75	100	105	110	115	120
PCA-SVM Test f1-score	0.9621	0.9749	0.9813	0.9819	0.9823	0.9824	0.9823	0.9829	0.9825	0.9827

PCA Table #2 Score for SVM given PCA components (dataset size 42,000)										
PCA components	125	130	140	150	160	200	250	300		
PCA-SVM Test f1-score	0.9825	0.9824	0.9823	0.9823	0.9824	0.9819				

It looks like the best score (0.9829) is for 110 PCA components, the last best score for SVM (50 PCA components) was 0.9819 an increase of 0.01.

RBM

Next let's look at finding the best RBM to run on SVM, we can visualize RBM components the same way we did with PCA.



RBM Table #1 Score for SVM given RBM components (dataset size 42,000)										
RBM components	15	20	30	50	75	100	110	150	200	256
RBM-SVM Test f1-score	0.3322	0.7132	0.8923	0.9303	0.9461	0.9562	0.9587	0.9663	0.9708	0.9738

RBM Table #2 Score for SVM given RBM components (dataset size 42,000)										
RBM components	300	350	400							
RBM-SVM Test f1-score	0.9750	0.9755	0.9757							

It looks like the best score (0.9757) is for 400 RBM components, we might even get higher scores for larger RBM component number; however, the score is not improving much from 256 to 300, or 300 to 350, or 350 to 400. In addition the best SVM RBM score is lower by ~.007 than the best SVM PCA score (0.9829).

Nudging Data

Now let try to increase our training dataset, our current training data set is 42,000 handwritten images of digits. We can nudge each image to the left, right, top and bottom by one pixel and two pixels, leaving the original dataset we get an additional 8 datasets of 42,000 each, we can also nudge the pixels diagonally NW, NE, SW and SE getting an additional 4 datasets of 42,000 each, this will give us a total of 13 datasets (including the original) or a total of 546,000 images to work with.

Putting it All Together

Now we can run MinMaxScaler(-1,1), PCA with the best number of components (110) and our best SVM on the training dataset of 546,000 images.

parameters: {'kernel': 'rbf', 'C': 3, 'gamma': 0.008}
dataset size: 546,000
test size: 0.20
train f1-score: 0.9998
test f1-score: 0.9978
Kaggle score: 0.99229

Notice that we are not using 20% of our data for training (test size: 0.20), let's run all the same parameters again but this time let's set test size to only: 0.001, note that we don't actually need a test set, as our kaggle submission can be our test set, we only used an internal test set to evaluate our models internally.

parameters: {'kernel': 'rbf', 'C': 3, 'gamma': 0.008}

dataset size: 546,000
test size: 0.001
train f1-score: 0.99977
test f1-score: 0.99817
Kaggle score: 0.99343

By using almost all of our training set we increased our Kaggle score by 0.00114, while that does not seem like much, each fraction of a percent on the Kaggle leaderboard will raise ranking significantly. As of this writing (03/02/2016) a score of 0.99343 gives a position of 42nd on the Kaggle Digit Recognizer Leaderboard. The original goal was to make it into the top 100, with the 42nd place this goal was achieved. One thing to note is that by using data nudging our train and test scores end up a bit higher than kaggle scores, as data was skewed by generation of relatively similar data for the training set as was to be expected, however the kaggle score ends up substantially higher with data nudging than without it. Also with a test set of only 0.001 or 546 images (that's 0.1% of 546,000), there is not enough test data to give an accurate test F1 score, but again we are not relying on that test set, as we are relying on the kaggle test set. We already know that the model produces good results from a 0.2 test set, so we are just trying to do a bit better with our final run of 0.001 test set.

Note:
https://www.kaggle.com/philipgura/results
See log for details: logSVCTopScore.txt

Conclusion

This was an interesting project that exposed us to number of machine learning models, as well as near real world data processing and image recognition. We found that SVM was the best scoring model out of the ones we tried, additionally we found that data preprocessing is just as important as the main model itself, for example without MinMaxScaler, SVM returns very poor results, and PCA can reduce dimensionality making the overall model run better in addition to significantly improving overall SVM results. It is also interesting that one can expand the training data set by running functions to modify the original dataset.

One should also note that this project did not go as smoothly as this report might convey, for a while KNN models produced the best results as data scaling was not used originally, also RBM did not work correctly as again scaling was off (the same scaling for SVM did not work for RBM).

While doing research on this project I've found some other models that can produce even better results, two such models are Convolutional Nets and Deep Neural Nets. Along with some additional preprocessing techniques like width normalization and deskewing. This might be something I implement as an expansion to this project perhaps using TensorFlow.

References:

https://www.kaggle.com/c/digit-recognizer

http://scikit-learn.org/stable/auto_examples/neural_networks/plot_rbm_logistic_classification.html#example-neural-networks-plot-rbm-logistic-classification-py

https://www.kaggle.com/kakauandme/digit-recognizer/tensorflow-deep-nn/notebook

http://www.pyimagesearch.com/2014/06/23/applying-deep-learning-rbm-mnist-using-python/

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