PRACTICAL DATA STUDY

Take Home Final
Statistical Learning

ANNA MOSENZON
ID 200320836

FEBRUARY 15, 2021

Contents

The analysis of the three-class problem of separating the digits 2,3,5 in the zip code data was performed on the following methods:

1. Linear discriminant analysis
2. Multi-class logistic regression
3. K-Nearest neighbors with Euclidean distance, with k equal to 1, 5, 10 and 20
4. Random Forest
5. Neural Network

For each model I executed the model with the default values that are defined in the relevant python package, mostly used is the SKLRN package. SKLRN is a package that provides a big variety of machine learning algorithms implementations. In addition, I executed hyper parameters tuning. For all first 4 models I used a function called Grid search, which performs an exhaustive search over specified parameter values for an estimator. For each model, I compared the default model with the tuned model. For the neural network I tuned the hyper parameters manually. The final results are presented in the Conclusions section.

Linear discriminant analysis

A classifier with a linear decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule.

The model fits a Gaussian density to each class, assuming that all classes share the same covariance matrix.

I tried both with dimensionality reduction and without, and the result was very similar. To tune the hyper parameters, I executed grid search with three solvers:

- 'svd': SVD: Singular value decomposition. Does not compute the covariance matrix, therefore this solver is recommended for data with many features.
- 'lsqr': Least squares solution
- 'eigen': Eigenvalue decompositions

And with different shrinkage parameters, which can be used only on the Least squares solution and Eigenvalue decompositions solvers:

- None: no shrinkage
- Automatic shrinkage using the Ledoit-Wolf lemma.
- Fixed shrinkage parameter: a set of values between 0 and 1 (with 0.01 step).

I tried both default values, that were given by SKLRN package in Python, and the grid search best values, and the results are described in the Summary.

Before hyper parameters tuning:

Test results:

Classification report:

	precision	recall	f1-score	support
2	0.9684	0.9293	0.9485	198.0000
3	0.8409	0.8916	0.8655	166.0000
5	0.8987	0.8875	0.8931	160.0000
accuracy	0.9046	0.9046	0.9046	0.9046
error rate	0.0954	0.0954	0.0954	0.0954

macro avg	0.9027	0.9028	0.9023	524.0000
weighted avg	0.9067	0.9046	0.9053	524.0000

Confusion matrix:

	2	**3**	**5**
2	184	10	4
3	6	148	12
5	0	18	142

Train results:

Classification report:

	precision	**recall**	**f1-score**	**support**
2.0	0.9863	0.9836	0.9849	731.0000
3.0	0.9612	0.9787	0.9699	658.0000
5.0	0.9835	0.9658	0.9746	556.0000
accuracy	0.9769	0.9769	0.9769	0.9769
error rate	0.0237	0.0237	0.0237	0.0237
macro avg	0.9770	0.9760	0.9765	1945.0000
weighted avg	0.9770	0.9769	0.9769	1945.0000

Confusion matrix:

	2	**3**	**5**
2	719	10	2
3	7	644	7
5	3	16	537

After hyper parameters tuning:

Test results:

Classification report:

	precision	**recall**	**f1-score**	**support**
2	0.9684	0.9293	0.9485	198.0000
3	0.8409	0.8916	0.8655	166.0000
5	0.8987	0.8875	0.8931	160.0000
accuracy	0.9046	0.9046	0.9046	0.9046
error rate	0.0954	0.0954	0.0954	0.0954
macro avg	0.9027	0.9028	0.9023	524.0000
weighted avg	0.9067	0.9046	0.9053	524.0000

Confusion matrix:

	2	**3**	**5**
2	184	10	4
3	6	148	12
5	0	18	142

Train results:

Classification report:

	precision	**recall**	**f1-score**	**support**
2.0	0.9849	0.9836	0.9843	731.0000
3.0	0.9598	0.9787	0.9691	658.0000

5.0	0.9835	0.9622	0.9727	556.0000
accuracy	0.9758	0.9758	0.9758	0.9758
error rate	0.0242	0.0242	0.0242	0.0242
macro avg	0.9760	0.9748	0.9754	1945.0000
weighted avg	0.9760	0.9758	0.9759	1945.0000

Confusion matrix:

	2	**3**	**5**
2	719	10	2
3	7	644	7
5	4	17	535

Summary

	solver	**shrinkage**	**Train error rate**	**Test error rate**
default	'svd'	-	0.0237	0.0954
tuned	'lsqr'	0.01	0.0242	0.0954*

The test error both for the SVD method with no dimension reduction and the least squared error with shrinkage parameter of 0.01 method, yield the same test error rate.

*Since the default values were examined in the hyper parameters tuning phase as well, these results are surprising but can be explained by technical explanation: the hyper parameter tuning grid search method performs K-fold cross-validation which means that it will split the data into train and validation. Consequently, the model never trains with the whole data set, in contrast to the default initial phase when I trained the entire data set and calculated the performance measures.

Taking that in consideration, the best model is the tuned model, with the least squared solution and the shrinkage parameter.

The main of Linear Discriminant Analysis is separate example of classes linearly moving them to a different feature space, therefore since the dataset is linear separable, only applying LDA as a classifier we will get great results. Dimensionality reduction provide very similar results. It can be explained by the attributes of the data set. Since the data set is an image data set, the examples are with different feature space, so we get a representation of the data set, but the classes are still separable at the same way, so we wouldn't expect a very different result.

Multi-class logistic regression

Logistic regression is a linear model for classification. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logistic function.

To tune the hyper parameters, I executed grid search with the following parameters:

- Solver:
  - 'newton-cg': uses curvature information (i.e. the second derivative) to take a more direct route, comparing to gradient descent.
  - 'lbfgs': an optimization algorithm that approximates the Broyden–Fletcher–Goldfarb–Shanno algorithm, which belongs to quasi-Newton methods.
  - 'liblinear': uses a coordinate descent (CD) algorithm (with one-vs-rest method)
  - 'sag': Stochastic Average Gradient descent
  - 'saga': a variant of "sag" that also supports the non-smooth penalty="l1".

- Penalty type:
  - L2
  - L1
  - Elastic net
  - No penalty
- C parameter values: C parameter is the inverse of regularization strength. Smaller values specify stronger regularization. I tried the following values: 100, 10, 1.0, 0.1, 0.01.

I tried both default values, that were given by SKLRN package in Python, and the grid search best values, and the results described in the summary.

Before hyper parameters tuning:

Test results:

Classification report:

	precision	recall	f1-score	support
2	0.9689	0.9444	0.9565	198.0000
3	0.8830	0.9096	0.8961	166.0000
5	0.9125	0.9125	0.9125	160.0000
accuracy	0.9237	0.9237	0.9237	0.9237
error rate	0.0763	0.0763	0.0763	0.0763
macro avg	0.9215	0.9222	0.9217	524.0000
weighted avg	0.9245	0.9237	0.9240	524.0000

Confusion matrix:

	2	3	5
2	187	7	4
3	5	151	10
5	1	13	146

Train results:

Classification report:

	precision	recall	f1-score	support
2.0	1.0000	1.0000	1.0000	731.0000
3.0	1.0000	0.9985	0.9992	658.0000
5.0	0.9982	1.0000	0.9991	556.0000
accuracy	0.9995	0.9995	0.9995	0.9995
error rate	0.0005	0.0005	0.0005	0.0005
macro avg	0.9994	0.9995	0.9994	1945.0000
weighted avg	0.9995	0.9995	0.9995	1945.0000

Confusion matrix:

	2	3	5
2	731	0	0
3	0	657	1
5	0	0	556

After hyper parameters tuning:

Test results:

Classification report:

	precision	recall	f1-score	support
2	0.9689	0.9444	0.9565	198.0000
3	0.8941	0.9157	0.9048	166.0000
5	0.9193	0.9250	0.9221	160.0000
accuracy	0.9294	0.9294	0.9294	0.9294
error rate	0.0706	0.0706	0.0706	0.0706
macro avg	0.9274	0.9284	0.9278	524.0000
weighted avg	0.9301	0.9294	0.9296	524.0000

Confusion matrix:

	2	**3**	**5**
2	187	7	4
3	5	152	9
5	1	11	148

Train results:

Classification report:

	precision	recall	f1-score	support
2.0	0.9905	0.9945	0.9925	731.0000
3.0	0.9863	0.9878	0.9871	658.0000
5.0	0.9855	0.9784	0.9819	556.0000
accuracy	0.9877	0.9877	0.9877	0.9877
error rate	0.0123	0.0123	0.0123	0.0123
macro avg	0.9874	0.9869	0.9872	1945.0000
weighted avg	0.9877	0.9877	0.9877	1945.0000

Confusion matrix:

	2	**3**	**5**
2	727	2	2
3	2	650	6
5	5	7	544

Summary

	solver	**penalty**	**C parameter (regularization inverse)**	**Train error rate**	**Test error rate**
default	'lbfgs'	L2	1.0	0.0005	0.0763
tuned	'newton-cg'	L2	0.1	0.0123	0.0706

The best results were performed by the tuned model, that was with high regularization (but not too high) and with 'newton-cg' solver. Both solvers of Newton's method and Broyden–Fletcher–Goldfarb–Shanno algorithm are performing very well and exploit the second derivative attribute for better optimization, but their disadvantage is the high computational time. It seems that the main factor here was the regularization. In the default settings there was no regularization (C parameter equal to 1) and the tuned setting included a relatively high regularization factor. The regularization

prevented overfitting and eventually lead to better performance on the test set. Looking at the default setting, which did not included regularization, presented an extremely low train error, which emphasizes the importance of regularization for preventing overfitting.

K-Nearest neighbor with Euclidean distance

Neighbors-based classification is computed from a majority vote of the nearest neighbors of each point: a query point is assigned the data class which has the most representatives within the nearest neighbors of the point. I tried four different k values: 1, 5, 10 and 20. In addition, I performed a hyper parameter tuning and it included also additional values to k.

To tune the hyper parameters, I executed a grid search with the following parameters:

- K equal to all (integer) values between 1-20.
- Weight to neighbors in neighborhood:
  - 'uniform': uniform weights. All points in each neighborhood are weighted equally.
  - 'distance': weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- Metrics to measure distance:
  - 'euclidean': sqrt(sum((x - y)^2))
  - 'manhattan': sum(|x - y|)
  - 'minkowski' - sum(|x - y|^p)^(1/p)
    - The default metric, and with p=2 is equivalent to the standard Euclidean metric. Practically there were only 2 types of metrics that were examined - 'euclidean' and 'manhattan', since the p value wasn't tuned.

I tried both default values, that were given by SKLRN package in Python, and the grid search best values, and the results are described in the summary.

Before hyper parameters tuning:

k = 1

Test results:

Classification report:

	precision	recall	f1-score	support
2	0.9746	0.9697	0.9722	198.0000
3	0.9394	0.9337	0.9366	166.0000
5	0.9444	0.9562	0.9503	160.0000
accuracy	0.9542	0.9542	0.9542	0.9542
error rate	0.0458	0.0458	0.0458	0.0458
macro avg	0.9528	0.9532	0.9530	524.0000
weighted avg	0.9542	0.9542	0.9542	524.0000

Confusion matrix:

	2	3	5
2	192	5	1
3	3	155	8
5	2	5	153

Train results:

Classification report:

	precision	recall	f1-score	support
2.0	1.0	1.0	1.0	731.0
3.0	1.0	1.0	1.0	658.0
5.0	1.0	1.0	1.0	556.0
accuracy	1.0	1.0	1.0	1.0
error rate	0	0	0	0
macro avg	1.0	1.0	1.0	1945.0
weighted avg	1.0	1.0	1.0	1945.0

Confusion matrix:

	2	3	5
2	731	0	0
3	0	658	0
5	0	0	556

k = 5

Test results:

Classification report:

	precision	recall	f1-score	support
2	0.9745	0.9646	0.9695	198.0000
3	0.9281	0.9337	0.9309	166.0000
5	0.9503	0.9562	0.9533	160.0000
accuracy	0.9523	0.9523	0.9523	0.9523
error rate	0.0477	0.0477	0.0477	0.0477
macro avg	0.9510	0.9515	0.9512	524.0000
weighted avg	0.9524	0.9523	0.9523	524.0000

Confusion matrix:

	2	3	5
2	191	6	1
3	4	155	7
5	1	6	153

Train results:

Classification report:

	precision	recall	f1-score	support
2.0	0.9905	0.9932	0.9918	731.0000
3.0	0.9849	0.9909	0.9879	658.0000
5.0	0.9945	0.9838	0.9892	556.0000
accuracy	0.9897	0.9897	0.9897	0.9897
error rate	0.0103	0.0103	0.0103	0.0103
macro avg	0.9900	0.9893	0.9896	1945.0000
weighted avg	0.9897	0.9897	0.9897	1945.0000

Confusion matrix:

	2	3	5
2	726	5	0
3	3	652	3
5	4	5	547

k = 10

Test results:

Classification report:

	precision	recall	f1-score	support
2	0.9792	0.9495	0.9641	198.0000
3	0.9107	0.9217	0.9162	166.0000
5	0.9329	0.9562	0.9444	160.0000
accuracy	0.9427	0.9427	0.9427	0.9427
error rate	0.0573	0.0573	0.0573	0.0573
macro avg	0.9409	0.9425	0.9416	524.0000
weighted avg	0.9434	0.9427	0.9429	524.0000

Confusion matrix:

	2	3	5
2	188	9	1
3	3	153	10
5	1	6	153

Train results:

Classification report:

	precision	recall	f1-score	support
2.0	0.9864	0.9904	0.9884	731.0000
3.0	0.9731	0.9878	0.9804	658.0000
5.0	0.9926	0.9694	0.9809	556.0000
accuracy	0.9835	0.9835	0.9835	0.9835
error rate	0.0165	0.0165	0.0165	0.0165
macro avg	0.9840	0.9826	0.9832	1945.0000
weighted avg	0.9837	0.9835	0.9835	1945.0000

Confusion matrix:

	2	3	5
2	724	7	0
3	4	650	4
5	6	11	539

k = 20

Test results:

Classification report:

	precision	recall	f1-score	support
2	0.9737	0.9343	0.9536	198.0000
3	0.8908	0.9337	0.9118	166.0000

5	0.9375	0.9375	0.9375	160.0000
accuracy	0.9351	0.9351	0.9351	0.9351
error rate	0.0649	0.0649	0.0649	0.0649
macro avg	0.9340	0.9352	0.9343	524.0000
weighted avg	0.9364	0.9351	0.9354	524.0000

Confusion matrix:

	2	**3**	**5**
2	185	11	2
3	3	155	8
5	2	8	150

Train results:

Classification report:

	precision	**recall**	**f1-score**	**support**
2.0	0.9836	0.9850	0.9843	731.0000
3.0	0.9686	0.9848	0.9766	658.0000
5.0	0.9890	0.9676	0.9782	556.0000
accuracy	0.9799	0.9799	0.9799	0.9799
error rate	0.0201	0.0201	0.0201	0.0201
macro avg	0.9804	0.9791	0.9797	1945.0000
weighted avg	0.9801	0.9799	0.9800	1945.0000

Confusion matrix:

	2	**3**	**5**
2	720	11	0
3	4	648	6
5	8	10	538

K	**1**	**5**	**10**	**20**
Train error rate	0	0.0103	0.0165	0.0201
Test error rate	0.0458	0.0477	0.0573	0.0649

We can see that the lowest test error was with the k=1 model, in the default settings. We can assume that the lowest the k model is with lowest bias, and that the bias is more significant to the model than the variance in our data set. In the hyper parameter tuning I tried different k values in addition to these four values.

After hyper parameters tuning:

Test results:

Classification report:

	precision	**recall**	**f1-score**	**support**
2	0.9746	0.9697	0.9722	198.0000
3	0.9390	0.9277	0.9333	166.0000
5	0.9387	0.9562	0.9474	160.0000
accuracy	0.9523	0.9523	0.9523	0.9523
error rate	0.0477	0.0477	0.0477	0.0477
macro avg	0.9508	0.9512	0.9510	524.0000

weighted avg	0.9524	0.9523	0.9523	524.0000

Confusion matrix:

	2	**3**	**5**
2	192	5	1
3	3	154	9
5	2	5	153

Train results:

Classification report:

	precision	**recall**	**f1-score**	**support**
2.0	1.0	1.0	1.0	731.0
3.0	1.0	1.0	1.0	658.0
5.0	1.0	1.0	1.0	556.0
accuracy	1.0	1.0	1.0	1.0
error rate	0	0	0	0
macro avg	1.0	1.0	1.0	1945.0
weighted avg	1.0	1.0	1.0	1945.0

Confusion matrix:

	2	**3**	**5**
2	731	0	0
3	0	658	0
5	0	0	556

Summary

The results were slightly better with the default settings:

	metric	**k**	**weights**	**Train error rate**	**Test error rate**
default	'minkowski'	1 (best out of the 4)	'uniform'	0	0.0458
tuned	'euclidean'	4	'distance'	0	0.0477*

*Since the default values were examined in the hyper parameters tuning phase as well, these results are surprising but can be explained by technical explanation: the hyper parameter tuning grid search method performs K-fold cross-validation which means that it will split the data into train and validation. Consequently, the model never trains with the whole data set, in contrast to the default initial phase when I trained the entire data set and calculated the performance measures.

Taking that in consideration, the best model is with k=4 and the points are weighted by the inverse of their distance. By defining k = 4, we are getting a good trade off point between the bias and variance of the model. Another parameter that seems to have influence on the model is the weight given to each point, while in the default setting each point was given the same weight whereas in the tuned model closer neighbors of a query point had have a greater influence.

Random Forest

Before hyper parameters tuning:

A random forest is an estimator that fits several decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. Since the random forest is computational costly, I used Randomized search function. In contrast to Grid Search function, which perform an exhaustive search over all possible parameter combinations, not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions. The number of parameter settings that I defined is 100. Additional advantage to the randomized search is that the chances of finding the optimal parameter are comparatively higher in random search because of the random search pattern where the model might end up being trained on the optimized parameters without any convergence. In the paper Random Search for Hyper-Parameter Optimization by Bergstra and Bengio, the authors show empirically and theoretically that random search is more efficient for parameter optimization than grid search.

The possible parameters that I defined to tune the hyper parameters are the following parameters and values:

- 'bootstrap': Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.
    - I tried with or without the bootstarp.
- 'max_depth': The maximum depth of the tree.
    - Values I tried: 80, 90, 100, 110.
- 'max_features': The number of features to consider when looking for the best split.
    - Values I tried: 2, 3
- 'min_samples_leaf': The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches.
    - Values I tried: 3, 4, 5
- 'min_samples_split': The minimum number of samples required to split an internal node.
    - Values I tried: 8, 10, 12
- 'n_estimators': The number of trees in the forest.
    - Values I tried: 100, 200, 300, 1000

I tried both default values, that were given by SKLRN package in Python, and the random search best values, and the results are described in the summary.

Test results:

Classification report:

	precision	recall	f1-score	support
2	0.9594	0.9545	0.9570	198.000
3	0.9250	0.8916	0.9080	166.000
5	0.9222	0.9625	0.9419	160.000
accuracy	0.9370	0.9370	0.9370	0.937
error rate	0.0630	0.0630	0.0630	0.0630
macro avg	0.9313	0.9324	0.9317	524.0000
weighted avg	0.9335	0.9332	0.9331	524.0000

Confusion matrix:

	2	3	5

2	189	7	2
3	7	148	11
5	1	5	154

Train results:

Classification report:

	precision	recall	f1-score	support
2.0	1.0	1.0	1.0	731.0
3.0	1.0	1.0	1.0	658.0
5.0	1.0	1.0	1.0	556.0
accuracy	1.0	1.0	1.0	1.0
error rate	0	0	0	0
macro avg	1.0	1.0	1.0	1945.0
weighted avg	1.0	1.0	1.0	1945.0

Confusion matrix:

	2	**3**	**5**
2	731	0	0
3	0	658	0
5	0	0	556

After hyper parameters tuning:

Test results:

Classification report:

	precision	recall	f1-score	support
2	0.9740	0.9444	0.9590	198.0000
3	0.9255	0.8976	0.9113	166.0000
5	0.9123	0.9750	0.9426	160.0000
accuracy	0.9389	0.9389	0.9389	0.9389
error rate	0.0611	0.0611	0.0611	0.0611
macro avg	0.9372	0.9390	0.9376	524.0000
weighted avg	0.9398	0.9389	0.9389	524.0000

Confusion matrix:

	2	**3**	**5**
2	187	9	2
3	4	149	13
5	1	3	156

Train results:

Classification report:

	precision	recall	f1-score	support
2.0	1.0	1.0	1.0	731.0
3.0	1.0	1.0	1.0	658.0
5.0	1.0	1.0	1.0	556.0
accuracy	1.0	1.0	1.0	1.0
error rate	0	0	0	0
macro avg	1.0	1.0	1.0	1945.0

weighted avg	1.0	1.0	1.0	1945.0

Confusion matrix:

	2	**3**	**5**
2	731	0	0
3	0	658	0
5	0	0	556

Summary

	bootstrap	**Max depth**	**Max features**	**Min samples leaf**	**Min samples split**	**N estimators**	**Train error rate**	**Test error rate**
default	False	None*	'auto'**	1	2	100	0	0.0630
tuned	False	90	30	1	2	300	0	0.0611

*If Max depth is defined as "None", then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

** If Max features is defined as "auto", then max_features=sqrt(n_features).

We can see that the tuned model produced better results. The tuned model has relatively short depth, which gives some regularization to the model. Also, there are almost twice features that are sampled in each tree than the default model. It can give an advantage in our context, as the data set is high dimensional and bigger sample of the feature dimension can give more information than the smaller sample. Also, as we learned in class, the higher number of trees is better for Random Forest models, so as expected, the tuned model has better performance with 300 trees than the default model which has only 100 trees.

Neural Networks

In this section I implemented Neural network inspired by the VGG neural network modern architecture, 2014 winner of the ImageNet Large Scale Visual Recognition Challenge or ILSVRC. The ILSVRC is a benchmark in object category classification and detection on hundreds of object categories and millions of images. The challenge has been run annually from 2010 to present (in a different setting), attracting participation from more than fifty institutions. Researchers from the Oxford Visual Geometry Group, or VGG, participated in the ILSVRC challenge and won the ImageNet Challenge for localization and classification in 2014.

The Neural network that I developed is much simpler than the VGG net but have a few similarities. Even though, it produced a very good result. I used Keras Python package which provide implementation of Neural network.

The Neural Network architecture is described as follows:

- Input layer with 256 neurons (as the number of the features in our data set), the input was reshaped to 16X16X1 dimensions.
- 2 convolution layers with 20 filters and size of 3X3, with ReLU activation function.
- Max pooling layer with pooling size of 2X2.
- 2 convolution layers with 64 filters and size of 3X3, with ReLU activation function.
- Max pooling layer with pooling size of 2X2.
- Fully connected layer with 128 neurons, with ReLU activation function.
- Output layer with 3 neurons, as the number of classes, with Softmax activation function.

The “VGG inspiration” is mainly in the construction of block with 2 convolution layers and Max pooling layer that separate between the blocks. The number of layers is 7, which do not include the pooling layers, which are not learned layers.

The convolution layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs.

The max pooling layer down-samples the input representation by taking the maximum value over the window defined by pooling size (I defined it to be 2X2) for each dimension along the features' axis. The max pooling help to prevent overfitting by providing an abstracted form of the representation that was inserted to it as an input. As well, it reduces the computational cost by reducing the number of parameters to learn and provides basic translation invariance to the internal representation.

ReLU activation function is rectified linear unit: max(x, 0). It provides the non-linearity of the net which is important when we deal with image data set, such as our data set.

The loss function is Cross entropy with more than 2 classes (called “categorial cross entropy” in the Keras python package).

I tuned the hyper parameters of the neural net manually and find that the following hyper parameters gave the best performance:

- The optimizer SGD with a momentum function, which is accelerates gradient descent in the relevant direction and dampens oscillations.
- Learning rate of 0.0001.
- Batch size of 8 samples. The batch size defines the number of samples that will be propagated through the network. Typically, networks train faster with mini-batches. That is because it updates the weights after each propagation
- Number of epochs = 1000. One epoch is one forward pass and one backward pass of all the training examples.

Test results:

Classification report:

	precision	recall	f1-score	support
2	0.9897	0.9697	0.9796	198.0000
3	0.9578	0.9578	0.9578	166.0000
5	0.9573	0.9812	0.9691	160.0000
accuracy	0.9695	0.9695	0.9695	0.9695
error rate	0.0305	0.0305	0.0305	0.0305
macro avg	0.9683	0.9696	0.9689	524.0000
weighted avg	0.9697	0.9695	0.9695	524.0000

Confusion matrix:

	2	3	5
2	192	4	2
3	2	159	5
5	0	3	157

Train results:

Classification report:

	precision	recall	f1-score	support
2	1.0	1.0	1.0	731.0
3	1.0	1.0	1.0	658.0
5	1.0	1.0	1.0	556.0
accuracy	1.0	1.0	1.0	1.0
error rate	0	0	0	0
macro avg	1.0	1.0	1.0	1945.0
weighted avg	1.0	1.0	1.0	1945.0

Confusion matrix:

	2	**3**	**5**
2	731	0	0
3	0	658	0
5	0	0	556

Conclusions

The result of the neural network surpassed the other models and gave a test error rate of 3.05%. the non-linearity of the neural network ensure that it would work well with image data sets, such as our data set. Also, the usage of max pooling layer contributed to avoid overfitting.

To sum up all the model, the performance of the models ordered from the best from the worst are as follows:

Rank	**Model**	**Test error rate**	**Train error rate**
1	Neural Network (CNN, in our case)	0.0305	0
2	K nearest neighbor with k=4	0.0477	0
3	Random forest	0.0611	0
4	Multi class logistic regression	0.0706	0.0123
5	Linear Discriminant Analysis	0.0954	0.0242