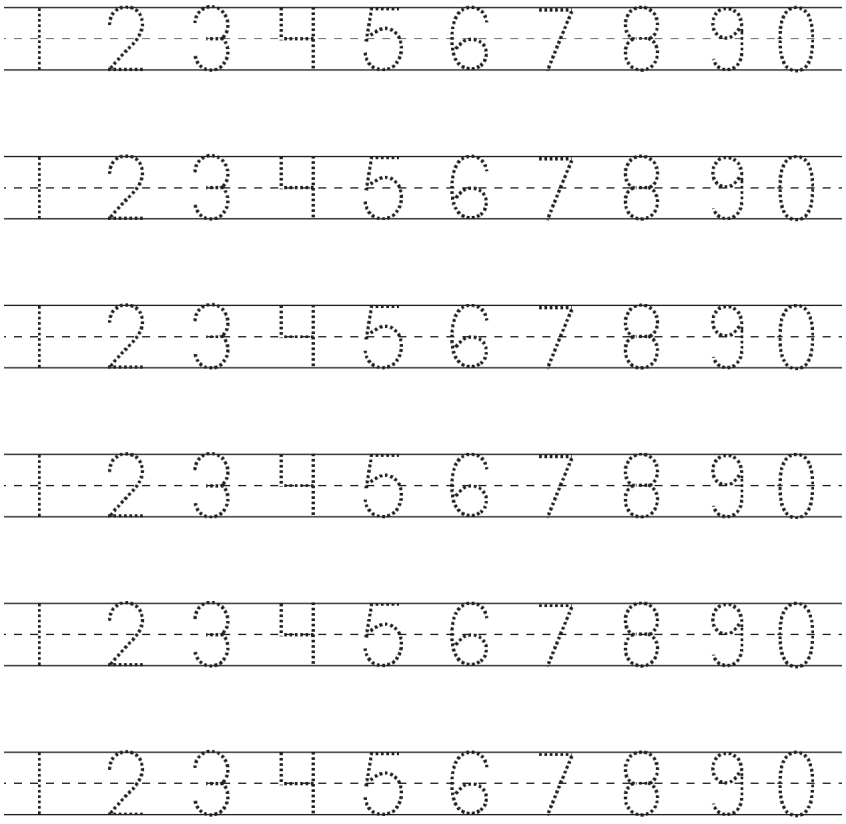
Multiple Methods of Classification in R: Classifying Handwritten Numbers 0-9

Regression and Machine Learning F

Kevin Shao & Jeffrey Cheng

16 April 2018



Kevin Shao & Jeffrey Cheng
Mr. Piper
Machine Learning
14 May 2018

1 Abstract

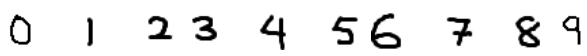
The MNIST dataset (http://yann.lecun.com/exdb/mnist/) is a classic dataset in the field of computer vision. The task is to classify simple black-and-white images of handwritten digits as its value (0-9). The each input observation comes in the form of 784 (28x28) numbers, ranging from 0-255, indicating the intensity of the color at the pixel (0 is no intensity, 255 is full intensity). There are four files: training images, test images, training labels, and test labels. Each is in the form of a data file. The format is very simple and can be found on the website, so it won?t be expanded on here.

We used a simple Python script to port the data to a CSV file.

Because the explanatory variables represent individual pixels within different images, it is not really helpful to perform a five number summary.

2 Preprocessing

Aside from porting to CSV format, there was not much general preprocessing to be done. Most of the data processing was specific to each classification method. Our training data set consisted of 59999 observations while our test set had 9999 observations. All observations had 785 columns, the first column begin the true number followed by the rest of the pixel data. An example of each number is shown below:



3 LDA

There were two main problems when attempting to do LDA on the dataset. Firstly, since the training and test sets are so big, the time it would take R to complete the LDA classification would be substantially long. Thus to counteract this problem, we took a small subset of our training and test sets. Since they were already randomized, there is no need to take a random sample, thus we took the first 2000 observations of the training set and the first 500 observations of the test set.

The second problem is that some of the pixels remain white through all the images. So, when R attempts to do lda, it indicates that the variables are constant. To counteract this problem, we took the sums of the columns of the dataset, which corresponds to the total grayscale of a specific pixel throughout all 2000 training images. If the sum was greater than an arbitrary number, say 1000, we used the column in our lda. If the sum were less than that arbitrary number, then we removed it from the data set. Using 1000 as our arbitrary number for now, we obtained a test and training error and corresponding confusion matrices.

Training Error	Test Error
0.041	0.246

d \r	0	1	2	3	4	5	6	7	8	9
0	186	0	1	0	0	0	0	0	1	0
1	0	214	3	0	1	1	1	6	3	0
2	0	0	187	0	0	1	0	0	0	0
3	1	1	0	185	0	2	0	0	0	0
4	0	0	2	0	203	1	0	1	0	6
5	0	2	0	2	2	171	0	0	1	1
6	1	0	0	0	1	1	199	0	1	0
7	0	0	1	1	0	0	0	210	0	3
8	3	2	3	0	1	2	0	0	165	2
9	0	1	1	3	6	1	0	7	1	198

Training Set Confusion Matrix

The d and the r refer to data and reference. So, the numbers running along the top of the matrix are the true numbers while the numbers along the left are what lda predicts. There are no consistent errors in the data set. The greatest errors come from confusing 4s with 9s, a task not teachers can consistently get right (for all those times that Dr. Brown couldn't read my handwriting).

It is interesting how much lower the test error is than the training error. We thought this was due to the fact that the lda model that we created was still more tailored to the training set we created. If our training set were bigger, the discrepancy between training and test error would decrease. Nonetheless, the both errors we got, were much lower than the 90% error we would obtain from randomly classifying.

d\r	0	1	2	3	4	5	6	7	8	9
0	35	0	0	0	1	1	2	0	0	0
1	0	66	6	0	1	1	1	1	1	1
2	1	0	36	2	1	3	4	2	2	0
3	1	0	1	34	1	3	0	1	1	1
4	0	0	2	0	41	1	2	0	4	7
5	3	1	1	5	0	36	2	0	2	2
6	0	0	2	0	1	1	32	0	0	0
7	0	0	2	1	0	0	0	33	2	5
8	1	0	4	1	0	3	0	0	27	1
9	1	0	1	3	9	1	0	11	1	37

Test Set Confusion Matrix

We wondered if we could improve upon our error if we changed the arbitrary value of 1000 we initially chose. So, we wrote a function that takes in the value of for which the sum of pixels in the training set needs to be greater than and returns the test and training errors.

n	Training	Test	n	Training	Test	n	Training	Test	n	Training	Test
50	0.0340	0.274	550	0.0365	0.240	1050	0.0400	0.240	1550	0.0425	0.238
100	0.0340	0.272	600	0.0390	0.243	1100	0.0400	0.244	1600	0.0435	0.238
150	0.0335	0.264	650	0.0385	0.245	1150	0.0405	0.246	1650	0.0435	0.234
200	0.0375	0.256	700	0.0385	0.238	1200	0.0405	0.242	1700	0.0450	0.234
250	0.0345	0.260	750	0.0395	0.238	1250	0.0405	0.232	1750	0.0445	0.236
300	0.0345	0.258	800	0.0400	0.242	1300	0.0420	0.236	1800	0.0445	0.236
350	0.0350	0.253	850	0.0405	0.244	1350	0.0420	0.232	1850	0.0445	0.236
400	0.0350	0.246	900	0.0410	0.244	1400	0.0425	0.236	1900	0.0450	0.232
450	0.0355	0.248	950	0.0410	0.244	1450	0.0425	0.236	1950	0.0450	0.232
500	0.0370	0.244	**1000**	**0.0410**	**0.246**	1500	0.0425	0.238	2000	0.0455	0.230

The bolded text represents the n = 1000 that we initially chose. There is a clear trend, the lower the n, which in turn represents less columns being discarded thus leading to a greater number of explanatory variables, the lower the training error. However, while the test error has a less linear trend, the general trend is the exact opposite of the training error. This makes sense since we are only using the most important explanatory variables when n is larger.

Is there another way to use lda to classify the variables? Since even using the reduced number of variables takes a long time for R to process, we wondered if the total number of colored pixels was related to the true number. After all, some numbers can be written using less pixels (read: ink) than others. Thus, we took a slightly larger training and test set, 10000 and 2500, respectively, and ran lda on the data.

number	fill	number	fill
0	194.87712	5	149.38631
1	89.11003	6	157.02761
2	168.83653	7	133.80748
3	165.40174	8	173.62288
4	143.07959	9	145.44683

Test Error
0.778

d\r	0	1	2	3	4	5	6	7	8	9
0	79	1	63	47	25	34	46	10	83	16
1	2	274	17	22	52	15	15	98	6	31
2	8	0	5	2	2	6	1	0	4	2
3	69	0	53	52	32	40	32	17	62	31
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	27	2	49	25	31	29	32	19	24	24
7	24	10	85	96	124	88	90	109	59	134
8	6	0	1	1	2	4	1	0	0	2
9	4	0	4	9	7	5	8	3	4	4

Test Set Confusion Matrix

The test error is so high because the group means shown above are so close to each other. This is also reflected in the confusion matrix, not one observation in the test set was classified as either a 4 or 5. While some numbers, such as one, were consistently correctly classified due to its unique pixel count, there were too many numbers that had a pixel count of around 140, which led to severe misclassification.

4 KNN

In contrast with other methods for machine learning, KNN is a form of lazy classification. That is, it defers all computation to the testing phase. In fact, there is no training phase; all the work is done upon testing an image, whereupon the k closest neighbors are computed, and the test image is classified most frequently occurring class. Proximity, in our implementation, is simply n-dimensional Euclidian distance.

For such a large dataset, especially for a manual implementation, it was important to bear in mind the complexity of the computation. Our essential code, the part that actually executes KNN, is as follows.

```
knn <- function (input, k = 7) {
  differences = sweep(arr, 2, input, "-", check.margin = FALSE )
  differences = differences * differences
  distances = rowSums(differences)
  indices = order(distances)[1:k]
  votes = train[indices, 1]
  prediction = strtoi(names(which.max(table(votes))))
  return (prediction)
}
```

As input to our function, we take input, a 784 dimensional vector of pixel intensities, along with an optional k value, which defaults to 7 (reducing the chance of ties, as most ambiguous digits are between two possibilities, such as 4 and 9, 3 and 8, etc).

In the first line of the function, we take advantage of the built-in R "sweep" function. The variable "arr" is a 60000×784 matrix, where the i-th row is the 784 dimensional vector representing the pixel intensities of the i-th training observation. In essence, the "sweep" function applies a given function, in this case subtraction (hence the "-"), along a given axis, in this case vertically (hence the 2), on each row of a given matrix, in this case "arr", by a given vector, in this case our input. The result, then, is a 60000×784 matrix, where the i-th row is the 784 dimensional vector representing the differences in pixel intensities between the i-th training observation and the test observation. The next line squares the matrix element-wise, and the following one collapses the matrix into a 60000 dimensional vector by summing each row. With three short lines, then, we have arrived at the squared distance from the test input to each training point. Since the square root function is monotonically increasing and only the order of the distances is relevant, it is unnecessary to take the computationally expensive step of taking the square root.

Next, we use the R function "order" to sort the distances from least to greatest, and take the first k of them (hence the [1:k]). We then convert it into a table, with "table(votes)", then take the most frequent appearance, with "which.max". After some simple parsing, we return our prediction.

With this implementation of KNN and choice of k, we arrive at a test error rate of 3%. The confusion matrix is as follows.

d\r	0	1	2	3	4	5	6	7	8	9
0	974	0	11	0	1	5	6	0	6	5
1	1	1133	8	3	8	0	3	25	4	6
2	1	2	988	2	0	0	0	3	0	0
3	0	0	2	976	0	8	0	0	11	6
4	0	0	1	1	945	2	3	1	7	8
5	1	0	0	12	0	866	2	0	12	4
6	2	0	2	1	5	4	944	0	1	1
7	1	0	16	7	1	1	0	988	6	11
8	0	0	4	4	1	2	0	0	2916	2
9	0	0	0	4	21	4	0	10	5	963

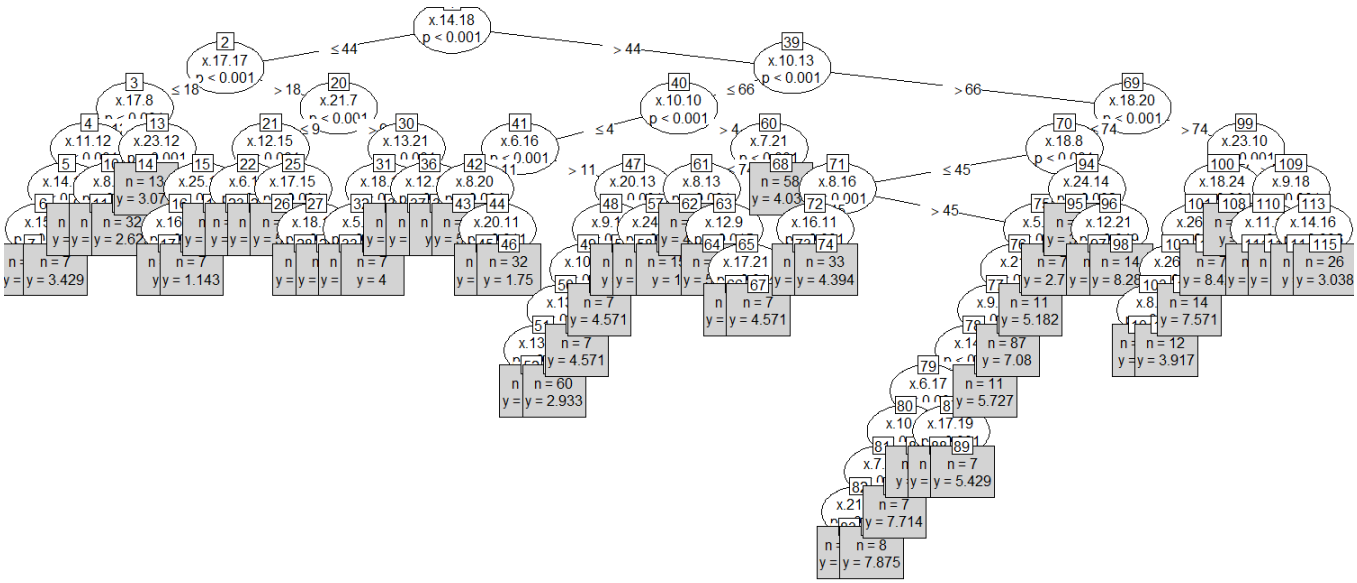
Test Set Confusion Matrix

As is surprisingly human, the principal misclassifications are misclassifying 7's as 1's, 4's as 9's, 2's as 7's and 8's as 3's or 5's. Interestingly, their converses are generally far more accurate.

The main drawback to KNN is the runtime complexity. After all, even with our vectorized approach (using the sweep method), our algorithm runs with complexity $O(784n)$, which, in this case, translated to taking 5 seconds to classify a single digit, far inferior to human speeds.

5 Random Forest

We needed to change the names of the variables since randomForest cannot read a string that looks like an integer for some reason. Thus, we just preppended an 'x' before each name. The naming scheme for each variable is simple, they are in the format of "x.a.b" where x is the aforementioned preppended letter, a is the row the pixel corresponds to on the actual image, and b is the column the pixel corresponds to. Completing the random forest classification, we are able to plot the result.



The extremely dense formatting of the boxes aside, we only need to look at the topmost split to show why random forest is a terrible way to classify our data. The first split occurs at 14.18, and begins the classification based on the color of that pixel. Essentially, all the splits are performed by looking at the greyscaling of a **specific** pixel. Even if the training error happened to be low, if each number that was drawn was just shifted one pixel over, then the entire model would fail. The only feasible way for this model to work is to take each image, crop all the whitespace around

it, rescale it back to size and give it artificial greyscaling, so the position and size of each image would match the others. However, this is next to impossible, which makes using a different classification method the better approach.

d\r	0	1	2	3	4	5	6	7	8	9
0	70	0	0	0	0	0	0	0	0	0
1	65	154	0	0	0	0	1	0	0	1
2	31	45	64	3	0	0	1	2	0	1
3	16	13	83	80	3	3	3	4	2	1
4	8	4	31	74	102	46	14	6	9	2
5	1	2	14	23	92	106	75	21	32	10
6	0	1	5	7	13	24	106	54	57	26
7	0	1	0	4	4	0	0	134	49	42
8	0	0	1	0	0	1	0	3	23	65
9	0	0	0	0	0	0	0	0	0	62

Training Set Confusion Matrix

Another problem with random forests is that it treated our classes as continuous variables, so we rounded to obtain integer values. This is shown in our confusion matrix, as all the error is from misclassifying numbers that are neighbors numerically, not by shape.

Training Error	Test Error
0.5495	0.9000

The test error is 90%, which is essentially just random classification. This result is consistent with what we predicted above. The training error is surprisingly semi-decent, but still nowhere close to as accurate as other methods above.

d\r	0	1	2	3	4	5	6	7	8	9
0	3	2	6	3	5	1	5	2	3	5
1	5	10	5	13	7	9	2	5	8	7
2	2	6	2	5	6	6	3	5	3	5
3	7	9	10	1	8	3	7	3	6	5
4	6	8	3	6	7	4	4	8	6	9
5	2	7	8	3	3	4	6	2	3	6
6	5	5	8	7	3	10	4	4	4	5
7	7	10	7	4	5	5	8	12	3	7
8	3	5	1	1	3	1	4	4	4	2
9	2	5	5	3	8	7	0	3	0	3

6 Conclusion

For LDA, our results of 25% test error is quite decent for our methods. The main roadblock to LDA was complexity: matrix inversion, which is at the core of LDA, is a computationally expensive routine, so we were forced to use a smaller dataset than possible. Even then, the training phase was sluggish. However, the test phase was instantaneous, in contrast to KNN, but at the cost of a slightly lowered accuracy.

For KNN, the findings were essentially opposites of LDA. Our test error was uncannily accurate, with a mere 3% error rate, and this method requires no training phase. However, its drawback was an extremely sluggish test speed. Of course, reducing the training size could have sped it it up at the cost of accuracy, harkening back to the classic speed-performance tradeoff.

For random forests, we arrived at the conclusion that this method is simply ineffective for dealing with our task. Since classifying images involves so many predictors, the tree that the algorithm arrives at is cluttered and nonsensical. Our test error was 90%, no better than a random guess, despite a training error of 54%.