CA274 Programming for Data Analysis – Assignment 1 – Adam Tegart

In this assignment I will use the digits dataset given in CA274 and explore the different routes we took in lectures to classify digits using the KNN algorithm.

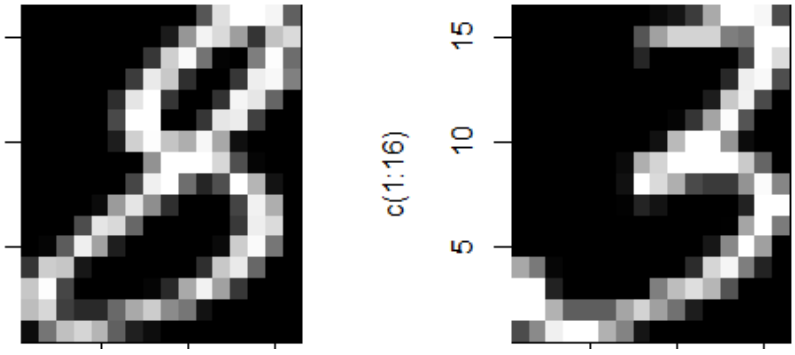
Section 1 – KNN where k = 2:

Running our KNN with k = 2 means that we classify the digit off just the digit closest to it. This is because the 2 nearest digits to any digit always contains itself as the first digit because of our implementation.

results \ labels	0	1	2	3	4	5	6	7	8	9
0	981	0	1	0	1	0	0	1	0	1
1	8	998	1	0	1	0	0	6	2	2
2	1	0	991	7	0	0	0	1	13	2
3	0	0	1	981	0	6	0	0	16	17
4	1	1	0	0	987	0	0	10	0	12
5	0	0	1	3	0	984	3	0	11	7
6	2	0	0	0	1	6	997	0	7	1
7	1	1	2	1	4	0	0	975	1	3
8	4	0	1	5	0	2	0	0	943	5
9	2	0	2	3	6	2	0	7	7	950

Fig 1.1. Confusion matrix of the results vector from classification against the labels vector.

It can be seen in the above confusion matrix that the 8's and 9's are the digits most commonly misclassified, so I have decided to look at their nearest neighbours to understand why exactly they are mistaken for other numbers.



8's mistaken for 3's:

It can be seen here that the 8 is identical to the 3 all along the right hand side.

The only discrepancy is the thin strokes to complete the 8 on the left hand side that are absent on the 3.

This is why the 8 is so similar to the 3.

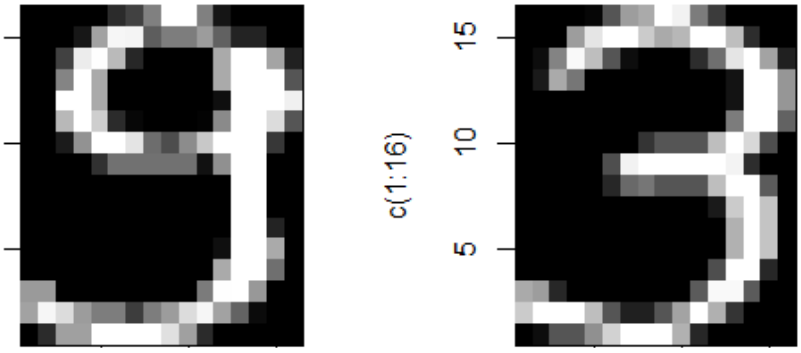


Fig 1.3. 9 misclassified as 3 Screenshot.

9's mistaken for 3's:

Here it is similar to the situation with the 8 and the 3 above. Along the right the digits are identical.

The 3 is missing a small arc to the left of the centre in order to become a 9. As this missing stroke is only small these digits are still very close together.

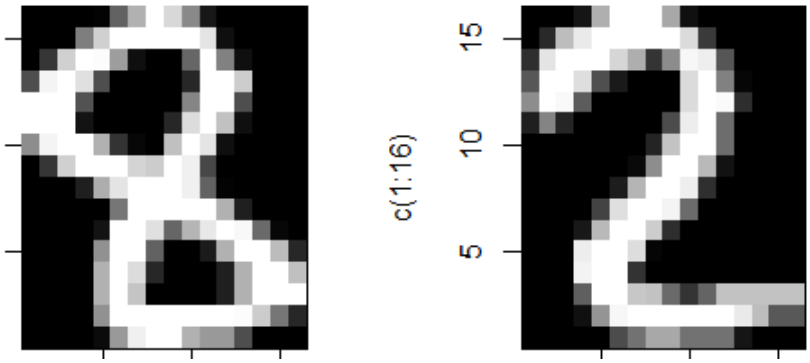


Fig 1.4. 8 misclassified as 2 Screenshot.

8's mistaken for 2's:

It can be seen here that the 2 looks like an 8 that is missing a diagonal stoke from top left to bottom right.

The stroke thickness and shape is identical and for this reason the 8 was classified as a 2.

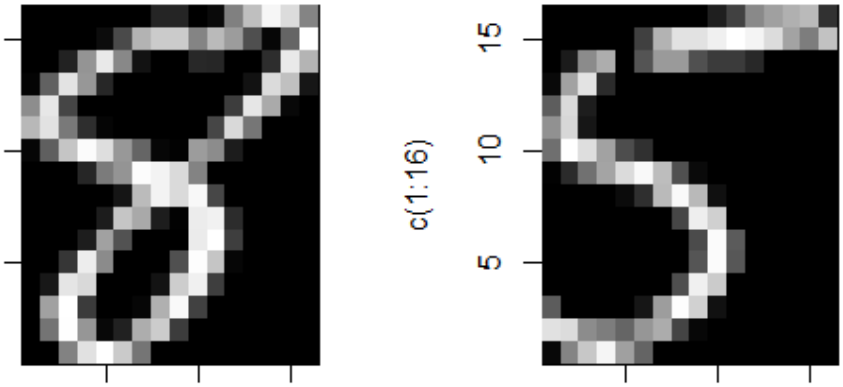


Fig 1.5. 8 misclassified as 5 Screenshot.

8's mistaken for 5's:

The digit 5 can be seen to follow a similar shape to that of an 8.

The only difference is the absence of a diagonal from bottom left to top right in the 5. As they are so close in shape and coverage it is no wonder some 8's are classified as 5's.

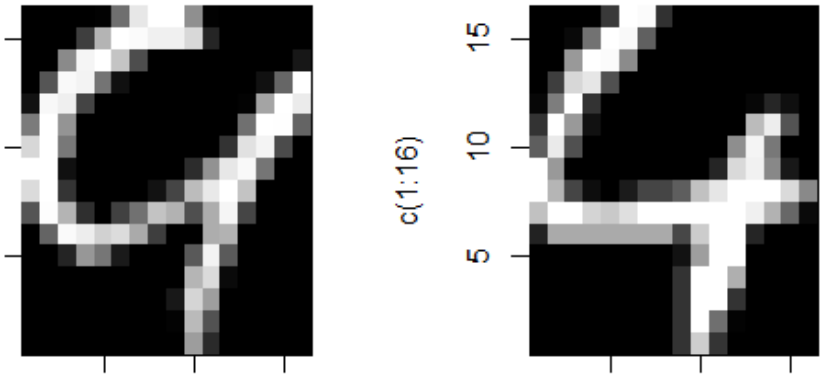


Fig 1.6. 9 misclassified as 4 Screenshot.

9's mistaken as 4's:

In much the same way as the other comparisons, the 9 and 4 have a very similar shape. The only slight difference between them is that the 4 is not rounded and closed at the top.

This particular 9 I found was a very exceptional case. The only indication that it is a 9 is the curvature of the digit.

Possible Improvement:

It is very clear that more complex digits are easier to misclassify because there is more variation between different examples of that digit. If you are using distances between images alone you may need many examples of 8's and 9's to make a difference to the ~95% accuracy as of current. This is because there are so many different possibilities for these digits because they have more strokes and arcs than other classes of digits.

One possible solution to this would be to check for the presence or absence of a stroke in the upper and lower left regions of the digit in the case of the 8 being classified as a 3. Allow me to use a diagram to explain my reasoning.

Below the original image I used can be seen. The premise of the idea is that images must have a similar shape.

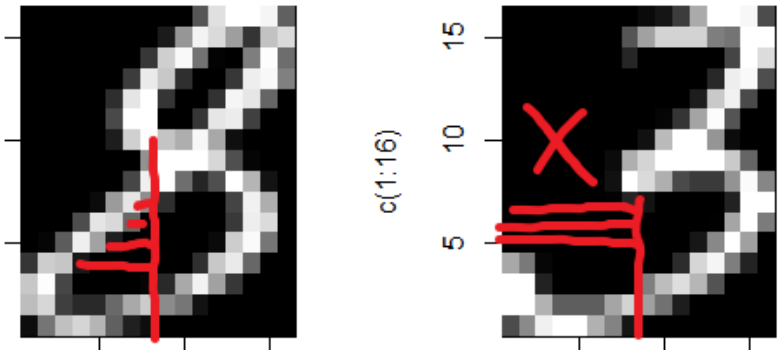


Fig 1.7. Example to explain possible improvement

If an image such as the 8 on the left is being compared to the 3 on the right. Looking at distances shows they're very similar.

If we were to check that there were at least a similar number of bright pixels down the bottom left we would find that there isn't. As a result we would know that there is a stroke missing and these are not likely to be the same digit. This could be applied to any stroke in a digit to help in classification.

This in theory would be wasteful to do for every image being classified, especially considering that there are only a handful where it would be necessary.

In addition to this, the fact that k = 2 means that an image is classified according to only the nearest neighbour. If k is increased I believe that the results will be better as these misclassifications are due to there being very similar shaped digits. In the general case where these digits didn't have such a similar shape by chance, we would see that the nearest neighbours would mainly consist of the same digit type.

Section 2 – 3D Point cloud (Displaying 8's, 3's and 8's misclassified as 3's):

I edited the code to show the 8's, the 3's and the 8's misclassified as 3's when k = 2 in the one point cloud. This was with respect to the first 3 eigenvectors with the most information. The clouds can be seen below, the 3's are red dots, the 8's are light-blue dots and the misclassified 8's are black stars so that they stand out.

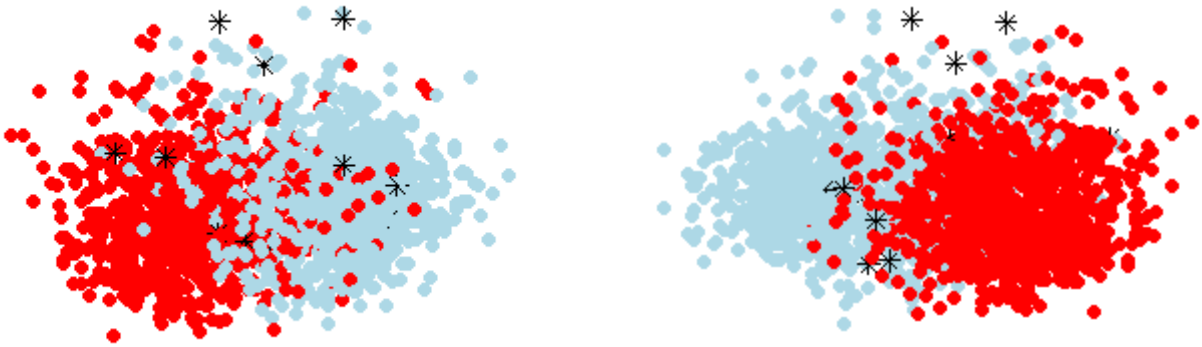


Fig 2.1. Point cloud highlighting position of misclassified 8's (two opposite sides)

It can be seen in the above images that the misclassified 8's lie a distance from the main clusters. They are sat along an outer shell of the main cluster. This is clear as they are found among the stragglers at the top and are also not as deep as the main cluster as they are placed at the front as a result of zorder. This helps to explain why there is some misclassification, but there are some that seem like they are in areas with only 8's in near proximity.

At first I was a little confused at why some of these points were being misclassified when they were buried among other 8's, how could their k-nearest neighbours be 3's? Well, turning the point cloud around some more and exploring the data revealed several 3's (red points) lurking within the groups.

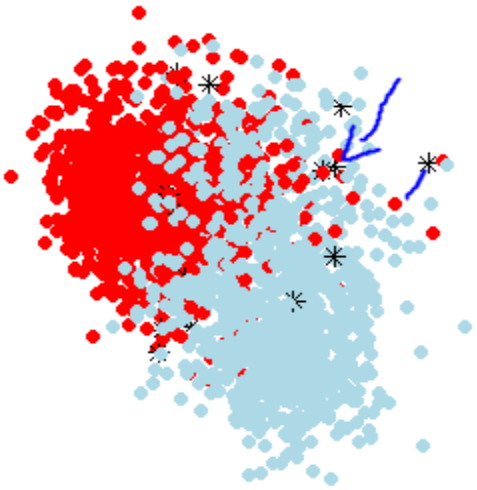


Fig 2.2. Point cloud rotated to show hidden 3's

In some of the cases I could see that a red point did not always appear closest. Above, the line joining the red dot to a black star in the right of the cluster looks like a longer distance than of that between the black star and the light blue dot to the right of the red dot. What I must also consider though is that there is information not shown in these 3 eigenvectors and among the other dimensions the red dot must be closer.

This shows to me that there is an obvious improvement to the algorithm. Should the value of k be increased we would eliminate these unfortunate scenarios where the closest neighbour is a different digit. This would definitely increase the accuracy and not be too expensive computationally for the increase in accuracy.

Running the code to get the accuracy with different values of k produces the following plot, I also used Sys.time() before and after the process and stored the difference to time the process.

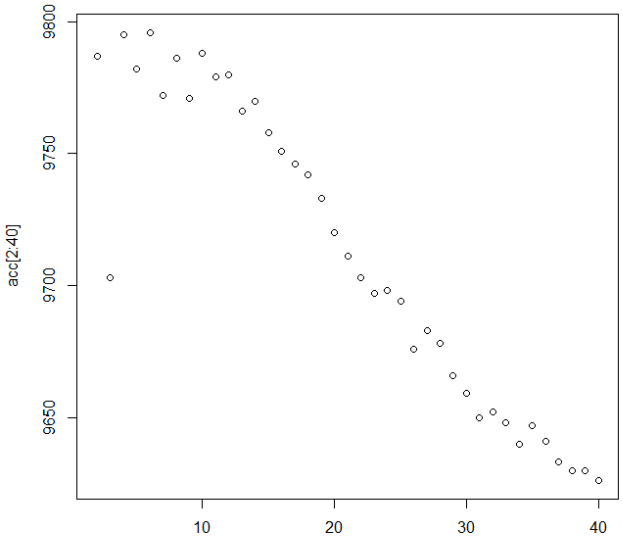


Fig 2.3. Plot of accuracies for different values of k

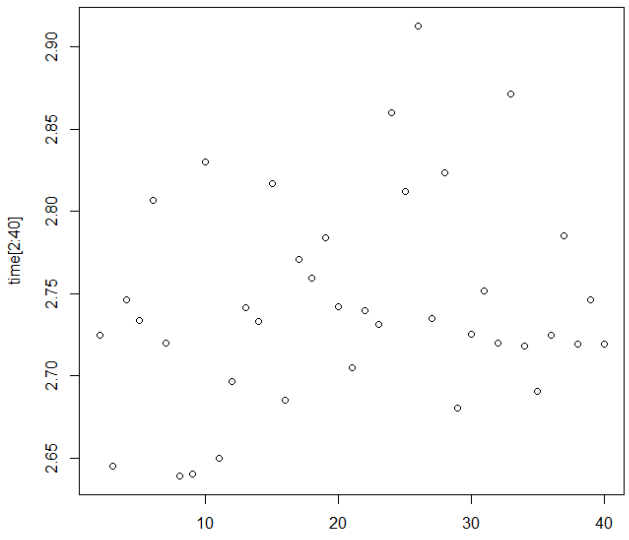


Fig 2.2. Plot of times to classify images with respect to k-nearest neighbours

It can be seen in the first plot that either k = 3 or 5 produces the best accuracy. Looking at the times it took to classify them it can be seen that there less than 0.3 of a second between the slowest and fastest times. This I believe is just variation as there is no correlation between the value of k and the time taken.

For this reason I believe that a value of 3 or 5 for k would have no increase to the cost of the computations but would make a difference to the overall accuracy of the classifications.

Section 3 – Accuracy using different no. of eigenvectors:

The main idea behind using eigenvectors is getting the original images in terms of the eigenvectors. When you multiply the original matrix d by the eigenvectors to get p you are transforming the matrix d onto the eigenvectors. This means that the first row corresponds to the amount the original images moves in the direction of the first eigenvector and contains the most information. If we analyse the distances between images with respect to only these rows we will get a good idea of the nearest neighbours with not nearly as much computation.

Below is the curve representing the accuracy for different eigenvalues. I have every value up to 150 and then in steps of 2 after that to get a nice curve.

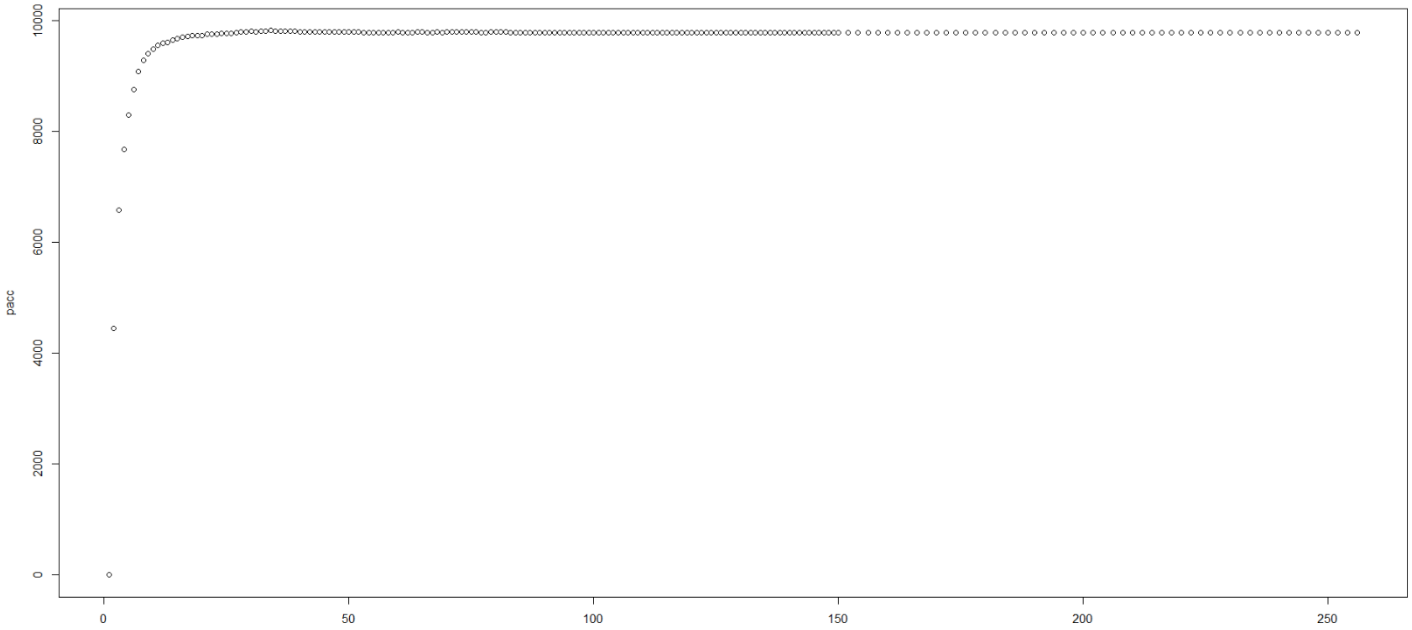


Fig 3.1. Plot of accuracy for different numbers of eigenvectors.

It can be seen that at first there is a drastic and steady increase in accuracy as we introduce more eigenvectors that still are rich in information. The curve follows what looks like a log functions typical shape. With a small number of eigenvectors the code runs much faster than the nbrs256 code. At about 15 eigenvectors the increase in accuracy becomes much smaller. At 34 eigenvectors we reach our max of 98.18%, from here adding eigenvectors decreases the accuracy and causes it to fluctuate around the 97.8% mark. So as a result of new information that is not so vital after 34 eigenvectors the accuracy changes slightly but never getting better than at 34 eigenvectors. So going past 34 offers nothing but an increased amount of computations.

Looking at the max value from using eigenvectors it seems larger than the max accuracy from using all 256 dimensions in the original set. This I believe is as a result of having only vital information that helps to correctly classify digits. One possible issue is that it may be overfitting the dataset, but a test set would be needed in order to check this.

As more eigenvectors are introduced the amount of time required to compute the nearest neighbours increases rapidly. Below a graph of the time for different eigenvectors can be seen.

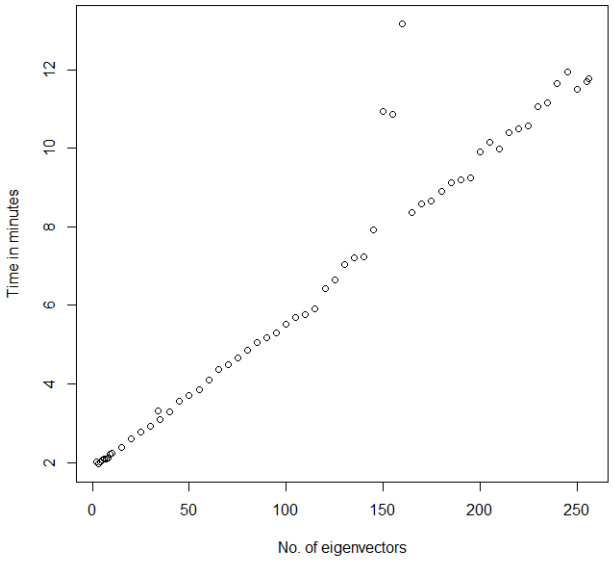


Fig 3.2. Plot of times (in mins) for different numbers of eigenvectors.

It can be seen above that the values follow a linear model. Using R I found that the equation to find the time taken is mins = 0.04*eigenvectors + 1.746 for the above(excluding 150-160). In the above the values for 150,155 and 160 are outliers. This is because during the computations my laptop decided to sleep and this took a toll on the computation time.

In order to find the ideal time for the best accuracy I decided to plot the times against accuracy.

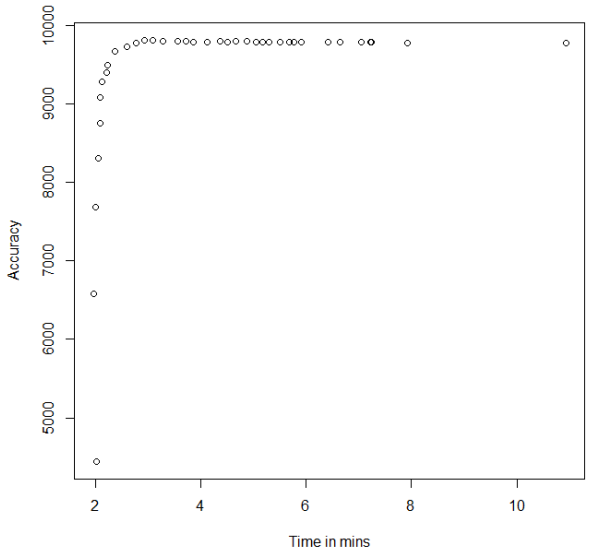


Fig 3.3. Plot of accuracy for different times (times from eigenvalues 1-10, then from 10-150 in steps of 5)

It can be seen above that the plot looks very similar to the accuracy plot. It turns out that you could of course use smaller values between 20-30 and still have a great accuracy with less computations, but anything over 34 eigenvectors is a waste of computations.

This does show that using eigenvectors can provide as good if not a better accuracy with a fraction of the computations.

Section 4 – Limiting computations using boxes:

In order to save on computations, only the distances for images in close proximity to the image we want to classify will be computed.

I modified the code to loop through box dimensions from 200-900 in steps of 100. I timed the computation and put the accuracy and times into 2 different matrices that correspond to one another. These can be seen below.

	200	300	400	500	600	700	800	900
200	9638	9689	9701	9712	9715	9716	9716	9694
300	9662	9703	9713	9718	9720	9721	9721	9720
400	9678	9709	9719	9724	9725	9726	9726	9726
500	9686	9716	9722	9727	9729	9730	9730	9731
600	9687	9715	9722	9725	9727	9728	9728	9729
700	9687	9714	9721	9725	9727	9728	9728	9728
800	9687	9714	9721	9725	9727	9728	9728	9728
900	9670	9709	9720	9725	9728	9728	9728	9728

Fig 4.1a. accuracy of boxes of different dims

	200	300	400	500	600	700	800	900
200	0.2643311	0.3063105	0.3621544	0.4172051	0.4636937	0.5011848	0.5350312	0.5710164
300	0.3035748	0.3960012	0.4853032	0.5615869	0.6311025	0.6946298	0.7482625	0.8044530
400	0.3590310	0.4787733	0.5925703	0.6994660	0.7939516	0.8791364	1.0291534	1.0284858
500	0.4264632	0.5821551	0.7258724	0.8568305	0.9750329	1.0776127	1.1307247	1.2598285
600	0.4589881	0.6298005	0.7953871	0.9479554	1.1153637	1.2413769	1.3179211	1.4499602
700	0.5022263	0.6917654	0.8781796	1.0523587	1.2572671	1.3985494	1.4665848	1.6080116
800	0.5464868	0.7563336	0.9707713	1.1424413	1.3121932	1.4668446	1.5962210	1.9950207
900	0.5783280	0.8197710	1.0518414	1.2667397	1.4511841	1.6219464	1.8097832	1.9845049

Fig 4.1b. times of boxes of different dims

It appears from looking at this data that the biggest box does not produce the best accuracy. I believe this is because you are including images that are not very close to the image we are trying to classify with respect

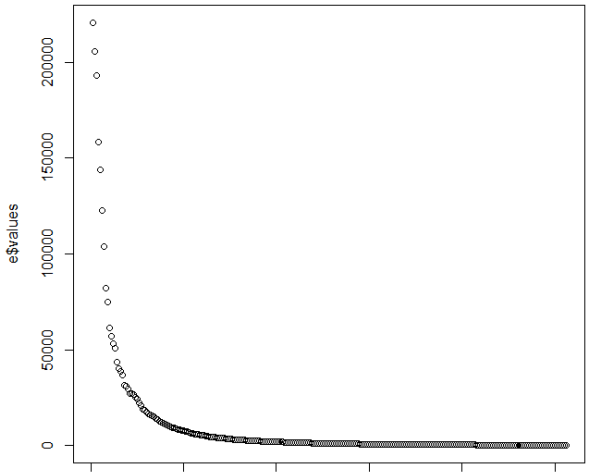


Fig 4.2. Values of eigenvectors

to the first and second eigenvector and are so not close based on a lot of the information about the images.

Looking at the values of eigenvectors shows that using the first two eigenvectors definitely gets some of the information, but this explains why the accuracy isn't the highest when we have a big box in these two dimensions. There are other dimensions that contain quite a lot of the information. When the box is too big a lot of images are let through and those that are close in dimensions that aren't very information rich seem to be considered when classifying.

On the top of the following page a plot of all the times against the accuracies. The colours separate the rows. So red is 200 along the first eigenvector, blue is 300, green is 400, blue violet is 500, chocolate is 600, cyan is 700, magenta is 800 and brown is 900.

I took some points using identify which are show below. They move along the column.

Point no.	19	27	35	36	41	60
x-dim	400	400	400	500	200	500
y-dim	400	500	600	600	700	900
Accuracy	97.19%	97.24%	97.25%	97.29%	97.16%	97.31%
Time	0.59 mins	0.699 mins	0.794 mins	0.975 mins	0.50 mins	1.25 mins

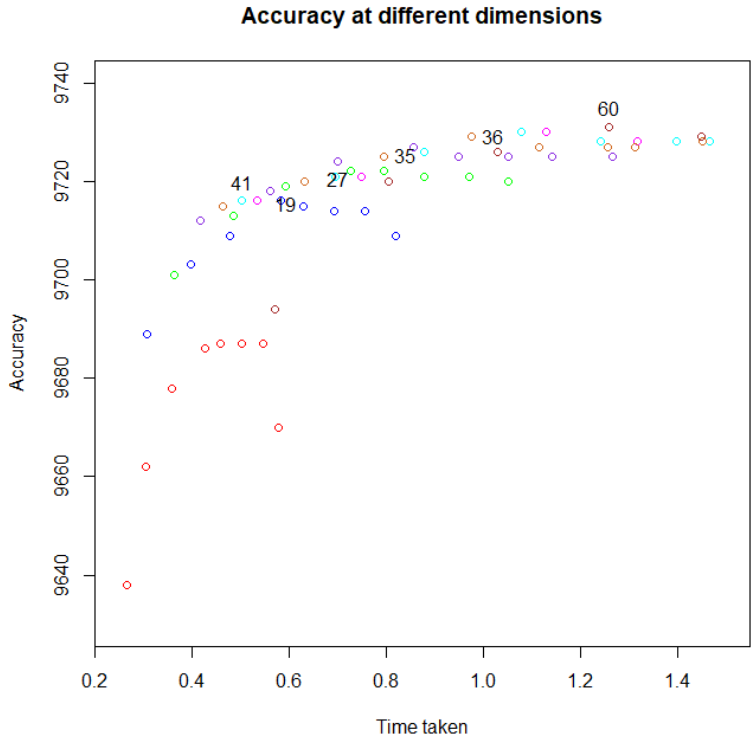


Fig 4.3. Plot of times against accuracy (points from table above are highlighted

It can be seen that all of the points with good accuracy and fast speed lie in a lower range of both the dimensions, typically < 600,600 should give less than a minute. Although if you do want that higher accuracy you shouldn't go as far as 900,900 as that is wasteful and has a lower accuracy, 500,900 at 1.25 mins is as high as you should go dimension wise.

Section 5 – Using blurring and boxes:

The code supplied for blurring was pre-set to a width of 1 and so was quite blurred. I created a matrix for both the accuracy and time taken. Below these two matrices can be seen, dimensions for the box range from 400-900 in steps of 100. This is because at smaller values some images did not have neighbours in this range. The rows correspond to the value of the width in direction of the first eigenvector and the columns correspond to the width of the box in the direction of the second eigenvector.

	400	500	600	700	800	900
400	9647	9645	9647	9647	9647	9647
500	9688	9688	9688	9688	9688	9688
600	9707	9707	9707	9707	9707	9707
700	9718	9718	9718	9718	9718	9718
800	9721	9722	9722	9722	9722	9722
900	9725	9727	9727	9727	9727	9727

Fig 5.1a. blurred accuracy for width 1

	400	500	600	700	800	900
400	0.7411537	0.8462068	0.9320728	1.005385	1.100803	1.082187
500	0.6903073	0.7901388	0.9592789	1.045504	1.021108	1.004779
600	0.7895101	0.9152984	1.0219292	1.090330	1.143515	1.179178
700	0.8875060	1.0367946	1.1572407	1.242488	1.308841	1.346713
800	0.9878438	1.1570476	1.2913344	1.390047	1.463664	1.506346
900	1.0826837	1.2644095	1.4128922	1.517196	1.605355	1.659145

Fig 5.1b. blurred time for width 1 (in mins)

It can be seen from the above tables that changes in width of the accepted values for the first eigenvector(changes along the rows of the table) cause an increase in the accuracy. This I believe is because of the fact that the first eigenvector always contains the most information. This seems to apply to this case more so than the usual as can be seen in the following plot of the eigenvectors for this blurring.

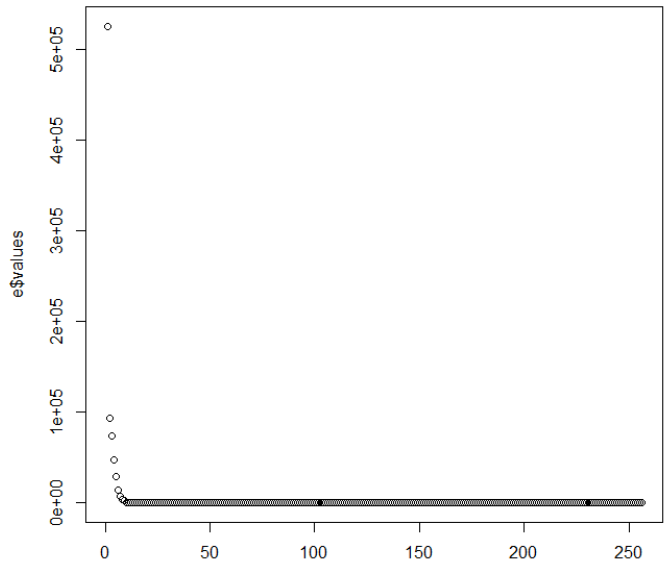


Fig 5.2. Eigenvector values for blur width of 1

It can be seen that even though there is little to no change as you move along the columns there is still an increase in the computation time. This is because along the second eigenvector there is less information than the first (as is seen in the above plot of the eigenvectors) and the images within that range are not as meaningful as the ones in the range of the first eigenvector. It is however a good thing to set a small width to this vector as it did help to eliminate extra computations as can be seen when the width is higher.

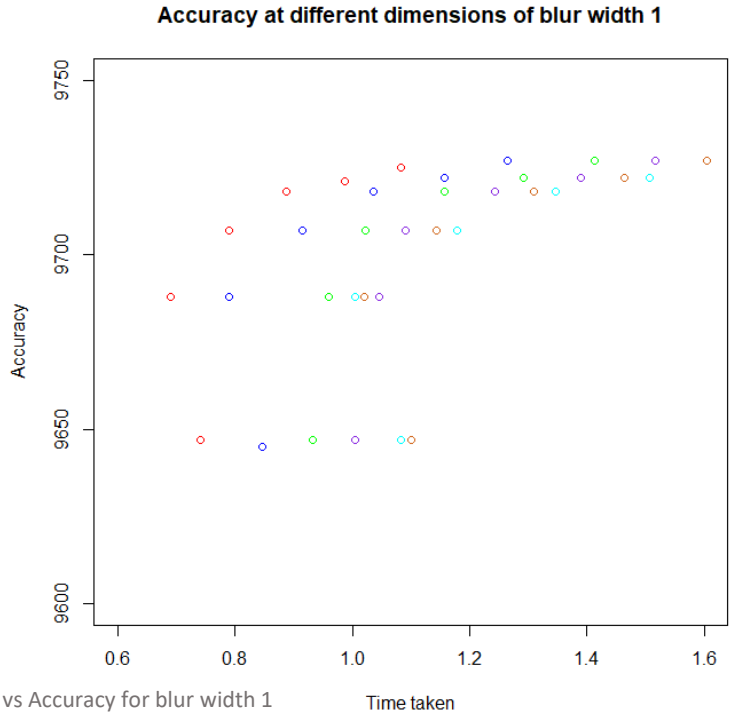


Fig 5.3. Time vs Accuracy for blur width 1

Above each colour represents different value of the width for the second eigenvector, so the columns are different colours. It can be seen that the accuracy for each is very similar in the distribution of the points. At 96.5% we have the first value in every column(the first row). This shows that although the time taken increases the accuracy doesn't as you increase the value of the width of the second eigenvector.

It can also be seen that a similar thing happens with the other rows, so it seems that the best way to keep the time taken down but have a good accuracy is to increase the width for the first eigenvector and keep the width's for the second eigenvector small. This can be seen as the biggest accuracy is achieved at 900,500.

This will not be the case when the level of blurring is not as intense as more information will get through and there will be more in the other eigenvectors. So I would assume that the dimensions of the box in the direction of the second eigenvector will be more important at higher widths in the blurring code.

To test this I used a list for blurred accuracy and blurred time and I had matrices identical to the ones in fig 5.1. populating these lists. For widths of values 3,5,7,9 and 11 I repeated what I did previous and made and filled out a matrix for each level of blurring so that they could be compared for different dimensions and the ideal combination for accuracy and efficiency could be found. For the less blurred details I used dimensions that were smaller as we would not be at as coarse of a level of detail as if the blurring was at width 1. E.g. at width = 9 and 11 I had dimensions of (300, 400, 500, 600, 700, 800).

	400	500	600	700	800	900
400	9691	9697	9700	9701	9701	9701
500	9702	9706	9708	9709	9709	9709
600	9719	9719	9722	9722	9722	9722
700	9720	9721	9723	9724	9724	9724
800	9722	9722	9723	9724	9723	9723
900	9725	9725	9724	9725	9724	9724

Fig 5.4a. blurred accuracy for width 3

	400	500	600	700	800	900
400	0.5426444	0.6454212	0.7462829	0.8258261	0.9020138	0.9565804
500	0.6534936	0.7773839	0.9000342	1.0030075	1.0885041	1.1527071
600	0.7524597	0.8992423	1.0346011	1.1562774	1.2629032	2.0572119
700	0.8395487	1.0061328	1.1588440	1.3023003	1.4147800	1.5272355
800	0.9149668	1.1361127	1.2945433	1.4810796	1.5829503	1.7116236
900	1.0104034	1.2575203	1.4389075	1.7872262	1.7393329	1.8249406

Fig 5.4b. blurred time for width 3 (in mins)

	400	500	600	700	800	900
400	9652	9675	9682	9682	9684	9685
500	9694	9707	9703	9703	9706	9706
600	9707	9720	9722	9720	9722	9722
700	9710	9725	9726	9727	9727	9727
800	9709	9725	9725	9725	9726	9726
900	9712	9727	9726	9725	9725	9725

Fig 5.5a. blurred accuracy for width 5

	400	500	600	700	800	900
400	0.3608531	0.4536819	0.5403361	0.6209814	0.6860118	0.7489200
500	0.4509929	0.5740651	0.6928077	0.7849881	0.8827420	0.9719002
600	0.5429189	0.6840706	0.8231225	0.9504528	1.0714364	1.1406604
700	0.6189596	0.7910284	0.9514032	1.0973096	1.2317944	1.3400608
800	0.6954929	0.8716564	1.0484034	1.1965889	1.3440500	1.4760545
900	0.7549315	0.9705038	1.1627195	1.3766773	1.5642600	1.6169717

Fig 5.5b. blurred time for width 5 (in mins)

	400	500	600	700	800	900
400	9679	9700	9702	9705	9706	9705
500	9701	9712	9706	9708	9708	9708
600	9715	9727	9729	9728	9727	9727
700	9714	9725	9724	9724	9724	9724
800	9719	9726	9727	9725	9725	9725
900	9718	9728	9730	9728	9728	9728

Fig 5.6a. blurred accuracy for width 7

	400	500	600	700	800	900
400	0.3656161	0.4570030	0.5361850	0.6153758	0.7019033	0.824460
500	0.4623924	0.5799444	0.6929319	0.7945221	0.9000797	1.258953
600	0.5565960	0.7219275	0.8309777	0.9478979	1.0606395	1.158541
700	0.6235804	0.7918651	0.9493649	1.0907847	1.3183745	1.337755
800	0.6855602	0.8764328	1.0695774	1.2173082	1.3698133	1.499002
900	0.7465090	0.9575302	1.1516955	1.3311463	1.4915448	1.637404

Fig 5.6b. blurred time for width 7 (in mins)

	400	500	600	700	800	900
400	9691	9714	9715	9722	9720	9720
500	9710	9727	9726	9726	9725	9725
600	9708	9724	9724	9724	9724	9724
700	9717	9726	9727	9726	9726	9726
800	9718	9727	9729	9728	9728	9728
900	9717	9728	9730	9729	9729	9729

Fig 5.7a. blurred accuracy for width 9

	400	500	600	700	800	900
400	0.3665221	0.4522176	0.5430774	0.6151930	0.6817886	0.7413195
500	0.4566633	0.5737392	0.6862726	0.8600042	0.9534041	0.9592206
600	0.5420376	0.6886901	0.8231716	0.9483825	1.0564609	1.1547022
700	0.6444339	0.8191409	0.9678494	1.0937632	1.2199648	1.3433766
800	0.6864145	0.8803172	1.0626640	1.2217387	1.3611994	1.5126952
900	0.7425342	0.9573144	1.1599381	1.3346214	1.5265634	1.6309185

Fig 5.7b. blurred time for width 9 (in mins)

	400	500	600	700	800	900
400	9699	9719	9720	9723	9722	9722
500	9712	9728	9729	9725	9725	9725
600	9717	9727	9725	9725	9726	9726
700	9719	9727	9727	9726	9727	9727
800	9720	9729	9729	9728	9729	9729
900	9719	9728	9728	9727	9728	9728

Fig 5.8a. blurred accuracy for width 11

	400	500	600	700	800	900
400	0.3611135	0.5098753	0.5600864	0.6415667	0.7554318	0.8176192
500	0.5031663	0.5930833	0.7280951	0.8796659	0.9099506	0.9935410
600	0.5859701	0.7146566	0.8428664	0.9589238	1.0578629	1.1679986
700	0.6353019	0.8324515	0.9979353	1.1476890	1.2949551	1.3808015
800	0.7033204	0.9110761	1.0691386	1.2246397	1.3744259	1.5351465
900	0.7399270	0.9521564	1.1476713	1.3246165	1.4926811	1.6445271

Fig 5.8b. blurred time for width 11 (in mins)

In the above matrices for width 5,7,9 and 11 the colnames and rownames are incorrect, they should both be the vector c(300,400,500,600,700,800) but in R an l and 1 look identical and part of my test code was used in my final solution.

This is a lot of data, but what does it all mean?

It seems that blurring the data decreases the accuracy slightly as there are certain images that are not checked as they do not lie in the box. The time also seems to be higher than that of the not blurred data from the previous section for the first few. The upshot of blurring the images is that at a less intense level of blurring at the middle levels there is nice accuracy for a smaller computational time.

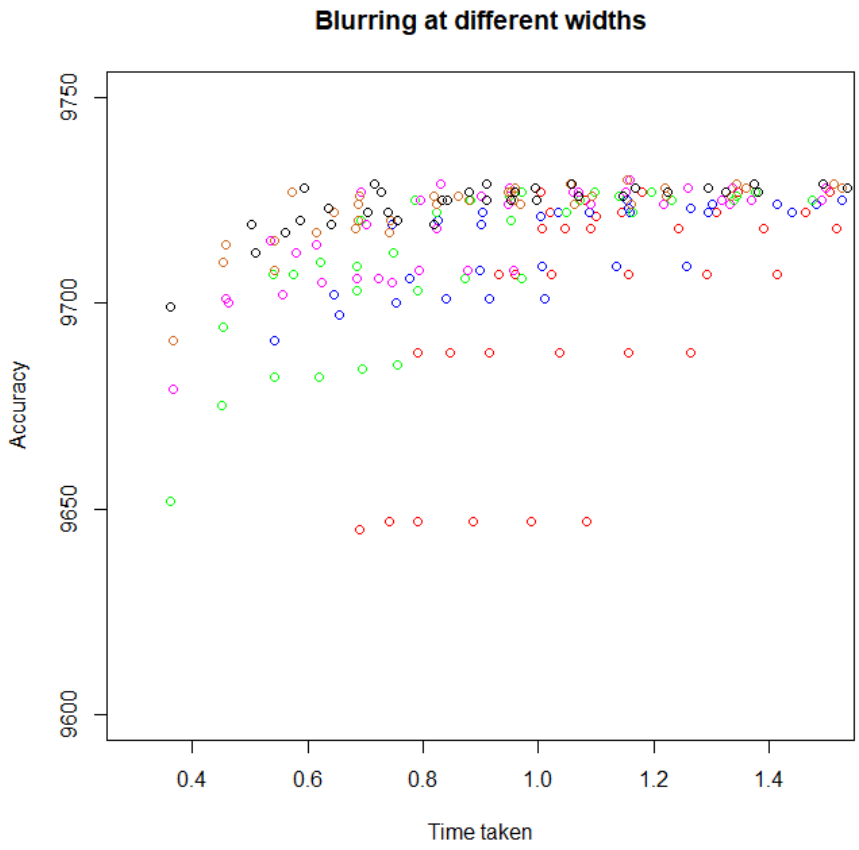


Fig 5.9. Time against Accuracy for different levels of blurring.

In the plot below the colours represent the different levels of blurring with red being 1, blue being 3, green being 5, magenta being 7, chocolate being 9 and black being 11.

It can be seen that the most accurate value is around the 97.3% mark and there are several times this value is seen. As low as 0.6 mins is the first time it is seen. At a moderate level of blurring with dimensions around the 600 or so This value can be found at a relatively low cost.

Overall, It must be concluded that at a higher width of blurring and a reasonably sized box, images can be classified quite accurately and very efficiently. This can be seen as with just boxes we can get an accuracy of 97.31 in 1.25 mins with dimensions 500,900. Using a blurring width of 7 we can get an accuracy of 97.3% in 1.15 with dimensions 800,500. So blurring can be used to save on computation time.