COMP3105: Assignment 4

Alexander Breeze 101 143 291
Victor Litzanov 101 143 028

Environment:

- Windows 10 20H2 x64
- Python 3.10.7
- Numpy 1.23.3
- Scipy 1.9.1
- Scikit-learn 1.1.3
- Matplotlib 3.6.0

Explanation:

We went with a k-means implementation to learn individual digits. To do this we separated each input image into 3 sub-images. We chose this approach because each label only corresponded to one of three digits in a given input, so lacking labels for each digit forced us to take an unsupervised approach. We did not use a neural net because we wanted to use things that had already been taught in the course, and we did not train on whole 3-digit columns of data because that would have been much harder to learn, and since we would be training on triples we would only have ⅓ as many data points. We tried the same implementation on columns of digits, but couldn't get more than 30% accuracy in any attempt.

We separated the digits simply by cutting each $1 \times d$ vector into thirds, one third for each digit in the column. For y we split each label i into a vector $[-1, i, i]$ such that the top digit in the column is given -1 because the label is not for it and the other 2 digits are given the label i because at least one of them is i. The -1 labels go into group 11, which is removed before finding max. This is only to probabilistically link groups to labels, so it does not need to be 100% accurate. In our case it is 55% accurate, 50% for each digit being the correct one and 10% chance of both digits being the same. With a large enough sample size the correct label is attributed to the correct group with a high degree of certainty. After separating the data and training k-means, we would label each group with the most common label assigned to it. This probabilistic strategy resulted in approximately 22% accuracy to begin with.

To improve accuracy further we pruned the groups. This was mostly to remove groups that contained multiple common labels, essentially sitting between digits. To do this we repeatedly tried removing each group, finding the resulting accuracy on the training set, and then permanently removing the group which increased accuracy the most. We repeated this until accuracy no longer improved. The main advantage of this approach for deciding which groups to remove is that it runs in $O(X \cdot K^2)$ time, which is important for finishing within the time restrictions. Other approaches, while possibly slightly more accurate, would take more time,

even exponential in some cases, which is unacceptable especially when $K > 250$ as in our submission. Removing the worst group each time tends towards a correct final result as accuracy can only increase. When accuracy stops increasing, we stop removing groups.

When classifying we simply use our model to classify each digit, then act accordingly based on the rules of the dataset. For branching on the top digit to decide which of the other 2 digits to read and return, we considered summing the strength of all the values for each possibility, e.g. $strength(0) + ... + strength(4)$ vs $strength(5) + ... + strength(9)$. However, we decided against this for multiple reasons: First, the k-means algorithm is already fairly accurate, so we can simply assume the strongest value is true. Second, there are issues with similarities between digits. For example, if the digit is a 4 then 4 should be the strongest value, but 8 and 9 will also be strong as they look similar to 4. This could result in misclassification since 4 (the strongest value) is outweighed by multiple weaker values which resemble it.

In order to implement our solution, we initially used our k-means implementation from assignment 3. However, it was having some classification issues and so we moved to an equivalent implementation that used Scipy (Henceforth referred to as “M1”). Kmeans randomly initializes K d-dimensional centers, then iteratively adjusts them to the centers of the groups of data such that the average distance from each point to the nearest center is at a local minimum. However, as can be seen in Table 1, approximately ⅓ of all attempts resulted in flukes having accuracy ranging from 15% to 50%. That, in combination with many of the runs taking well in excess of 2 hours forced us to look for a better implementation. We found out that the Lloyd algorithm frequently used by k-means implementations and possibly by M1 is notoriously inefficient and that Elkan would possibly work better while using up less resources. Scikit-learn is one of the libraries that offers this algorithm along with probabilistic cluster selection that aids in speeding up convergence, so that is the library we ultimately stuck with. This new implementation (“M2”) brought down our times to 30% of what they were before with a 10% boost in accuracy. During this switch to M2, we also worked on optimizing the remaining code as much as possible, converting everything to numpy functions and removing unnecessary steps that cost cpu time.

For selecting K, max_iter and num_runs, we first tried some random values and found that large K’s worked well. This is because with small K’s sometimes no group would be attributed to a given digit, since the groups follow whatever they consider to be most important. Also more groups means more pruning steps on average, and as established when pruning accuracy can only increase. We found that $K = 30$ was the minimum to get each label at least once, then saw even better accuracies with $K > 100$. We ran a simulation with the Scipy implementation and achieved the results seen in Table 1.

We also tried $K = 500$ and achieved 80% accuracy, but it took multiple days to train and so it fell well outside of the requirements for a reasonable time. Even the results seen in Table 1 took multiple days to collect. As such, we attempted to find the best K that allowed a compromise between accuracy and speed while also optimizing our code to improve performance. This led to our decision to use scikit-learn.

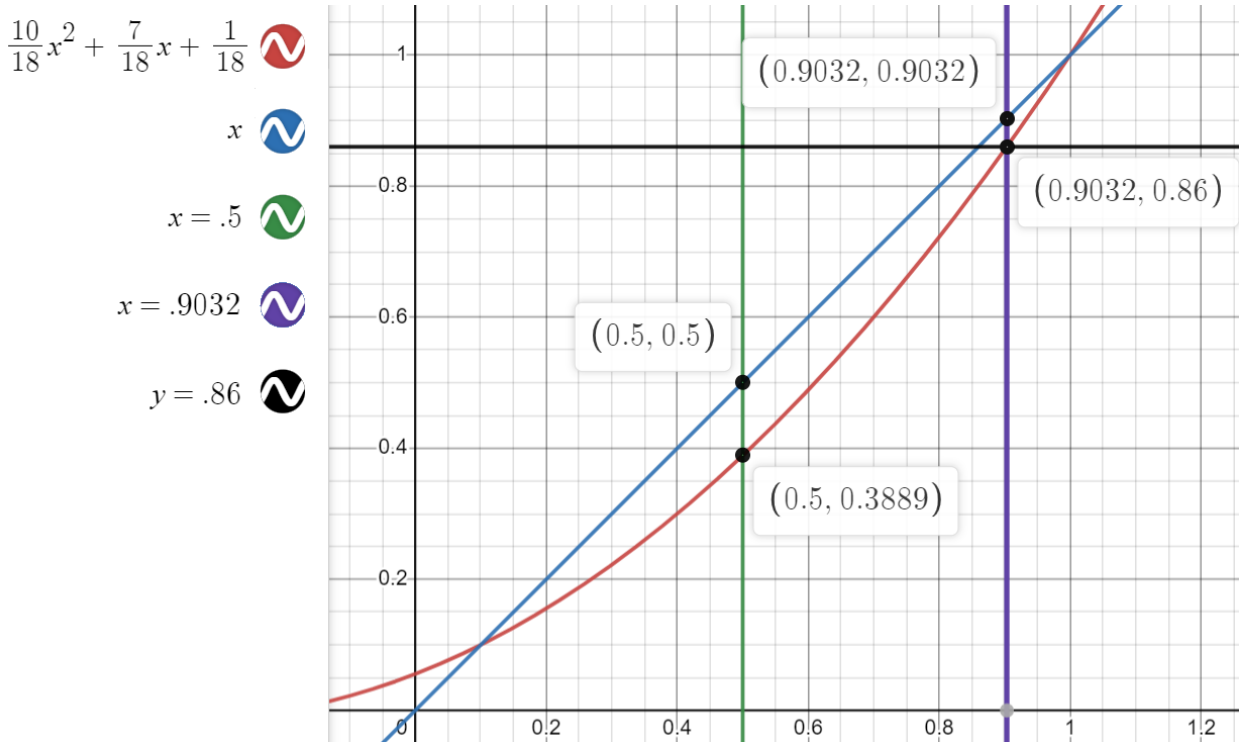
With the boost in performance afforded to us, we increased our search size and tabulated the results seen in Table 5. From these, we could see that the change begins to plateau around $K > 250$. Theoretically past this point we would be getting diminishing returns and to test this, we again tried $K = 500$ to see if the results would be better and if the time would be noticeably less. Sadly, the model's tested upper limit on accuracy is ~87% and unlikely to theoretically be much higher without applying more effective techniques, training for an absurd amount of time, or just switching to another model like a neural network, which is more accurate for these tasks.

We believe the cause of this limit is due to max_iter not mattering very much since the model eventually converges on local minimums regardless. Increasing K has its limits too, since in the pruning step we assign labels to each group probabilistically. With an input of 10000 data points, split into threes with the first digits discarded, that gets us 20000 digits with label accuracies of 60% (50% chance of being correct, 10% chance of both digits being the same, equal probabilities of each of these outcomes). With $K = 500$ we get on average $20000/500 = 40$ labels per group, with 24 being correct and 16 incorrect. As this is the average, it is common to get groups with less labels assigned and a higher ratio of incorrect labels, resulting in the removal of groups being pseudo-random when pruning models that still have a large number of groups. This puts an upper limit on the value of K for which accuracy will improve. Finally, as we train on individual digits instead of columns our model must classify 2 digits correctly to get the right answer unlike NN's that simply classify the 3-digit column input in one step. With a 1-digit classification accuracy of X, this gets us an overall accuracy of:

$$
\begin{aligned}
P(correct) &= P(split\ correctly)\ *\ P(2nd\ digit\ right) \\
&= ((P(1st\ dig.\ right)\ +\ P(1st\ dig.\ lucky))\ *\ P(2nd\ dig.\ right)) \\
&\qquad +\ P(1st\ dig.\ wrong)\ *\ P(2nd\ dig.\ lucky) \\
&= ((X\ +\ ((1\ -\ X)\ *\ \tfrac{4}{9}))\ *\ X)\ +\ (((1\ -\ X)\ *\ \tfrac{5}{9})\ *\ \tfrac{1}{10}) \\
&= (\tfrac{10}{18}X^2 + \tfrac{7}{18}X + \tfrac{1}{18})
\end{aligned}
$$

Meanwhile, as stated, for a single-stage classifier the accuracy is simply X. Below is a graph of both:

$y = overall\ accuracy$
$x = single\ digit\ accuracy$



As can be seen in the graph, making two separate classifications results in reduced accuracy, except for when X is so low that the model correctly classifies less than 10% of the time, at which point it is literally worse than random guessing. Several points are plotted. You can see that, with $X = 0.5$, a dual classification model only achieves 39% accuracy instead of 50%. You can also see that for a model to have an actual accuracy of 86% such as ours it must have a single-digit accuracy of 90%, which means that our model, given the limited label information inherent in the given problem dataset, still manages to achieve approximately 90% digit classification accuracy!

Figures:

Table 1: Accuracies on test data for M1

	Max Iterations						
K	15	30	45	60	75	90	AVG
160	68.24%	67.64%	26.68%	68.90%	70.00%	66.30%	61.29%
170	67.28%	11.30%	66.50%	17.98%	32.78%	68.66%	44.08%
180	22.96%	69.62%	72.08%	70.20%	70.22%	73.10%	63.03%
190	69.20%	69.48%	41.66%	70.10%	66.08%	67.58%	64.02%
200	69.34%	69.78%	71.52%	37.62%	59.58%	41.66%	58.25%
210	72.32%	15.04%	37.54%	73.26%	72.38%	34.12%	50.78%
220	68.02%	73.32%	73.12%	62.80%	36.44%	62.10%	62.63%
230	70.22%	74.00%	68.82%	74.32%	69.40%	66.18%	70.49%
AVG	63.45%	56.27%	57.24%	59.40%	59.61%	59.96%	

Table 2: Times for M1 in hours

	Max Iterations						
	15	30	45	60	75	90	AVG
160	1.42	1.46	1.48	1.08	0.77	0.51	1.12
170	0.49	1.67	1.12	1.69	1.71	1.07	1.29
180	1.91	1.59	1.28	1.23	1.15	1.28	1.41
190	0.82	0.50	2.11	1.55	2.07	1.28	1.39
200	1.44	0.51	1.93	2.35	0.32	2.34	1.48
210	1.18	2.61	2.63	1.74	1.58	2.65	2.06
220	0.62	1.35	1.25	0.57	2.95	2.90	1.61
230	1.26	2.19	0.66	2.02	0.79	0.51	1.24
AVG	1.14	1.48	1.56	1.53	1.42	1.57	

Table 3: Accuracies on test data for M2 on the same ranges

	Max Iterations						
K	15	30	45	60	75	90	AVG
160	80.88%	81.26%	82.22%	80.58%	83.34%	81.72%	81.67%
170	80.70%	82.44%	81.68%	82.58%	82.64%	81.24%	81.88%
180	81.32%	83.02%	82.56%	83.00%	83.12%	82.62%	82.61%
190	83.40%	82.58%	81.36%	83.24%	83.26%	82.98%	82.80%
200	82.14%	81.72%	83.30%	84.30%	83.88%	83.60%	83.16%
210	83.02%	82.20%	82.62%	83.98%	84.42%	84.90%	83.52%
220	84.30%	82.96%	85.64%	83.76%	84.46%	81.64%	83.79%
230	83.14%	83.74%	83.96%	84.26%	84.24%	83.46%	83.80%
AVG	82.36%	82.49%	82.92%	83.21%	83.67%	82.77%	

Table 4: Times for M2 in hours on the same ranges

	Max Iterations						
	15	30	45	60	75	90	AVG
160	0.04	0.23	0.23	0.17	0.20	0.07	0.16
170	0.23	0.09	0.07	0.77	0.34	0.34	0.31
180	0.19	0.79	0.40	0.15	0.31	0.23	0.34
190	0.12	0.10	0.16	0.08	0.42	0.06	0.16
200	0.37	0.39	0.17	0.24	0.08	0.18	0.24
210	0.45	0.11	0.14	0.38	0.12	0.38	0.26
220	0.75	0.46	0.43	0.23	0.10	0.07	0.34
230	0.60	0.58	0.63	0.48	0.16	0.24	0.45
AVG	0.34	0.34	0.28	0.31	0.22	0.20	

Table 5: Accuracies on test data for M2 over large range

	Max Iterations								
K	15	30	45	60	75	90	105	120	AVG
10	27.10%	44.76%	44.86%	44.72%	44.86%	44.94%	44.86%	44.86%	42.62%
20	53.12%	56.30%	55.04%	56.14%	56.58%	53.54%	55.74%	55.90%	55.30%
30	56.76%	59.09%	58.02%	57.56%	60.19%	55.98%	58.96%	59.34%	58.24%
40	62.22%	61.50%	64.03%	68.04%	63.26%	65.68%	61.28%	63.88%	63.74%
50	69.34%	67.40%	63.44%	66.18%	69.16%	64.46%	67.62%	66.08%	66.71%
60	71.00%	71.56%	70.64%	68.92%	23.78%	71.98%	70.54%	69.58%	64.75%
150	82.46%	81.02%	83.18%	81.86%	81.30%	81.54%	81.78%	82.68%	81.98%
160	80.88%	81.26%	82.22%	80.58%	83.34%	81.72%	81.58%	80.20%	81.47%
170	80.70%	82.44%	81.68%	82.58%	82.64%	81.24%	83.04%	82.24%	82.07%
180	81.32%	83.02%	82.56%	83.00%	83.12%	82.62%	81.34%	82.38%	82.42%
190	83.40%	82.58%	81.36%	83.24%	83.26%	82.98%	80.92%	83.12%	82.61%
200	82.14%	81.72%	83.30%	84.30%	83.88%	83.60%	83.56%	82.84%	83.17%
210	83.02%	82.20%	82.62%	83.98%	84.42%	84.90%	83.04%	82.64%	83.35%
220	84.30%	82.96%	85.64%	83.76%	84.46%	81.64%	84.00%	82.42%	83.65%
230	83.14%	83.74%	83.96%	84.26%	84.24%	83.46%	84.02%	84.26%	83.89%
240	83.38%	83.44%	83.64%	83.94%	83.50%	83.16%	83.94%	83.72%	83.59%
250	84.36%	83.72%	83.91%	84.17%	84.22%	84.30%	83.44%	84.58%	84.09%
260	85.04%	80.36%	85.04%	84.88%	84.14%	84.28%	84.17%	84.20%	84.01%
270	85.32%	84.52%	86.28%	85.70%	85.22%	85.60%	85.58%	85.36%	85.45%
280	84.86%	85.70%	84.66%	85.11%	84.48%	84.52%	84.54%	84.50%	84.80%
290	84.22%	84.82%	85.52%	85.78%	84.20%	84.38%	85.80%	84.14%	84.86%
300	84.58%	85.04%	85.86%	84.26%	85.04%	86.10%	85.22%	85.96%	85.26%
AVG	76.03%	76.78%	77.16%	77.41%	75.42%	76.94%	77.04%	77.04%	