Iris & CIFAR Linear Classification with Genetic Algorithms

Alejandro Soto González
School of Computer Engineering
Instituto Tecnológico de Costa Rica
Cartago, Cartago
Email: al.soto25@gmail.com

***Abstract*—The purpose of this project is to see the test the k-Nearest Neighbors algorithm with 2 hyper-parameters on CIFAR-10's set of images. In this instance, the hyper-parameters used were of course *k* and the *distance formula*. For k, the values of 1, 3 and 5 were tested, while for the formulas, Chebyshev's, Manhattan's and Levenshtein's were used. Both hyper-parameters were permuted to see how the results would vary and how they would fare among each other in terms of speed and accuracy. However, as expected, Levenshtein distance formula was not able to compare the whole set, so a much smaller set was used to have a notion about what the formula could do. At the end, while a bit slower than Chebyshev's formula, Manhattan proved to be the most well rounded.**

1. Introduction

Genetic Algorithms have always been a great optimization technique and since we'll be using a linear classifier, it might be a good way to experiment a bit with how good they are in optimizing the classifier with a **modified CIFAR-10 set** and **Iris**. As for hyper-parameters, this algorithm has a lot of flexibility, so most parameterizable things were taken advantage of; from the classic populatio, generations and mutation percentage, to crossover method and fitness score calculation. As much combinations as time allowed were allowed, always looking for the best results and heading that way more as more experiments were made. Due to time restraints, the GPU implementation could not be properly made and preferred to leave it out of the experiments for now since it wasn't the main idea of the project, however duration could be exponentially lowered if correctly implemented.

2. Methodology

2.1. Environment

The whole project was made in **Python** using **numpy** library to speed up vector operations, as well as other auxiliary libraries to measure the time like **timeit** and **matplotlib** to get graphs to see loss per generation. It was computed in my personal desktop computer which has the following specs (Only relevant ones to experiment given):

Processor	Intel Core i5-6600K @ 3.50GHz
Memory	16 GB @ 2400MHz
Storage	480 GB SSD

2.2. Genetic Algorithm

As with any Genetic Algorithm, the steps implemented for this project are: initialization, selection, crossover and then repeat until stop condition is met. Each step of the algorithm has several ways to be implemented and heaily depends on the problem at hand, so the ones that were believed were most fit for the problem were tried.

2.2.1. Initialization. This step is actually different for every single problem since it's basically adapting the data to the whole algorithm. Since first experiments were set to be with Iris, the whole base was made with that dataset and later the modified CIFAR-10 set was modified to match the same structure. Since the classification method is linear with a bias trick, both sets had to be slightly modified (added the 1 to the last column) and then, the rest of the W was initialized using a normal distribution with a mean of 0 for every case and a variance between 0.5 and 1.

2.2.2. Selection. For test and selection, the main score function that was used was the Hinge Loss Function, which is defined as:

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$

$$s = f(x_i, W)$$

This means that the total loss L for the gene W and all the images x is the sum of maxes between zero and the difference plus one between the result of the classification s_j for the known class against each of the remaining classes s_y, with j being the known image label and i being each of the images.

Since the loss function returns a score in a way where bigger is worse, as opposite to a normal fitness function, a slight modification was done to the loss scores, substracting the highest loss of the generation to all the genes to have a higher score for better suited genes.

Formulas used in all cases were the distance between two points (in this case color difference per channel per pixel), as it appears to be different variations for distance between lines or strings for some of these.

2.2.3. Chebyshev's Distance Formula. Distance *d* between two points *a*, *b* is defined as the **absolute magnitude of the differences between coordinates of a pair of objects**, which in this experiment would replace coordinates for color difference per channel per pixel, with the objects being images. The big O notation of this implementation is whatever numpy's vector sum function is, which we'll assume its $O(n)$.

$$d_{a,b} = (max|x_{ai} - y_{bi}|), \forall i \in a, b, \exists x \in a, \exists y \in b$$

2.2.4. Manhattan's Distance Formula. Similar to Chebyshev's formula, Manhattan's distance *d* between two points *a*, *b* is defined as the **sum of the differences between coordinates of a pair of objects**. The case for the big O notation of this implementation is also the same as Chebyshev's, which was assumed to be $O(n)$.

$$d_{a,b} = \sum_i |x_{ai} - y_{bi}|, \forall i \in a, b, \exists x \in a, \exists y \in b$$

2.2.5. Levenshtein's Distance Formula. This formula is completely different to the other two and it's mostly used for comparing strings, giving different weights depending if a character needs to be inserted, changed or deleted. The big O notation in this case is $O(n^2)$ as both vectors compares are always the same size, making it exponentially slower than the other two. A distance *d* between two strings *a*, *b* and indexes *i*, *j* for each one respectively is defined as:

$$d_{a,b}(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0 \\ min\{ & \\ d_{a,b}(i-1,j) + 1 & \\ d_{a,b}(i,j-1) + 1 & \text{else} \\ d_{a,b}(i-1,j-1) + 1_{a_i \neq b_j}\} & \end{cases}$$

$$\forall i \in a_1, \ldots, a_i, \forall j \in b_1, \ldots, b_j$$

2.3. Test/Train Sample

For the algorithm training and testing, the **CIFAR-10** image set was used, which includes a total of **50,000 images (40,000 for training and 10,000 for testing)** as well as **10 classes**. All of the images in the set were already classified to easily compare the results of the algorithm against the actual result.

3. Experiments

Initially a total of 9 experiments were set, as that would be the permutation of all the hyper-parameters. Scheduled with the whole 10,000 image test set against the training set of 40,000 images.

1) Classification by Chebyshev's distance with k = 1
2) Classification by Manhattan's distance with k = 1
3) Classification by Levenshtein's distance with k = 1
4) Classification by Chebyshev's distance with k = 3
5) Classification by Manhattan's distance with k = 3
6) Classification by Levenshtein's distance with k = 3
7) Classification by Chebyshev's distance with k = 5
8) Classification by Manhattan's distance with k = 5
9) Classification by Levenshtein's distance with k = 5

However after the first batch of results from the first 3 experiments, the experiments had to be changed due to resource limitations. Mostly in the size sample of the testing images for Levenshtein's distance formula.

1) Classification by Chebyshev's distance with k = 1
2) Classification by Manhattan's distance with k = 1
3) Classification by Chebyshev's distance with k = 3
4) Classification by Manhattan's distance with k = 3
5) Classification by Chebyshev's distance with k = 5
6) Classification by Manhattan's distance with k = 5
7) 5 image sample classification by Levenshtein's distance with k = 1
8) 5 image sample classification by Levenshtein's distance with k = 3
9) 5 image sample classification by Levenshtein's distance with k = 5

On each of these experiments, the accuracy and time elapsed were the main means of measurements used to compare them between each other.

4. Results

As previously noted in the Experiments section, the first batch of results changed the way the experiment was tested, specially for Levenshtein's formula. After waiting for around 2 hours and realizing that barely 4 images of the 10,000 were classified. So with the current environment, it would take approximately 32 years to finish the classification of the whole set. Because of this, the accuracy for the Levenshtein's formula was based on a a sample of only 5 images, just to be compared alongside the other 2.

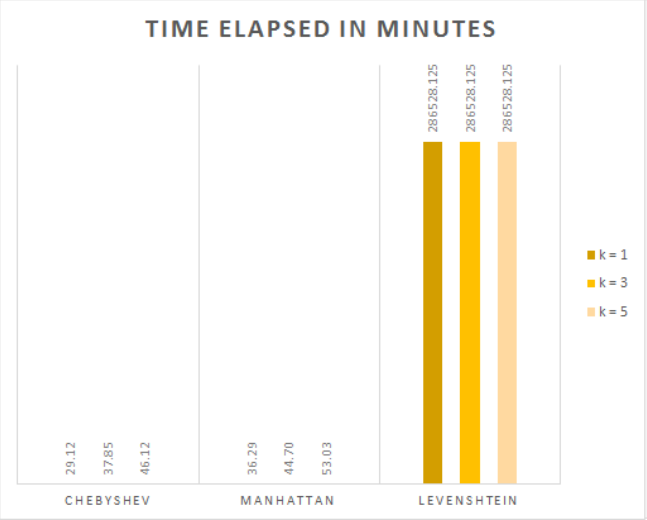

Figure 1. Graph of time elapsed in minutes for the permutation of all hyper-parameters.

In the Figure 1, the time that all formulast last is compared. However, the amount of time Levenshtein's formula lasted was assumed, calculated from what it lasted classifying 5 images.

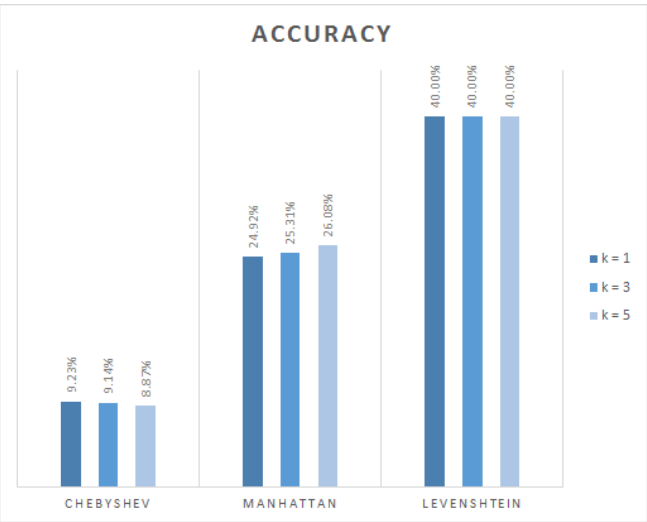

Figure 2. Graph of accuracy for the permutation of hyper-parameters.

Figure 3. Graph of amount of total misses for the permutation of hyper-parameters.

In the last 2 graphs, the overall accuracy and total misses for the comparison of the whole 10,000 test image set is seen. As Levenshtein's formula resulted impossible to compute with the available resources, it's accuracy was again assumed based on the performance classifying 5 random images. It has to be noted that with such a small sample, results are completely unreliable and not a good source of information about how it would actually bare against the others.

To keep the experiment a bit more precise, the Levenshtein's formula was removed from the results as most of them were assumed based on a very low amount of images classified. So the next 3 graphs show the actual complete results of the experiment with the whole set of images.

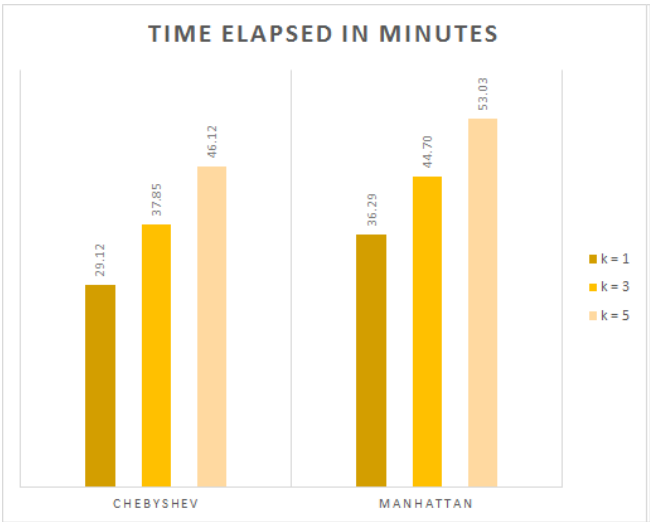

Figure 4. Graph of accuracy for the permutation of hyper-parameters.

Figure 5. Graph of amount of total misses for the permutation of hyper-parameters.

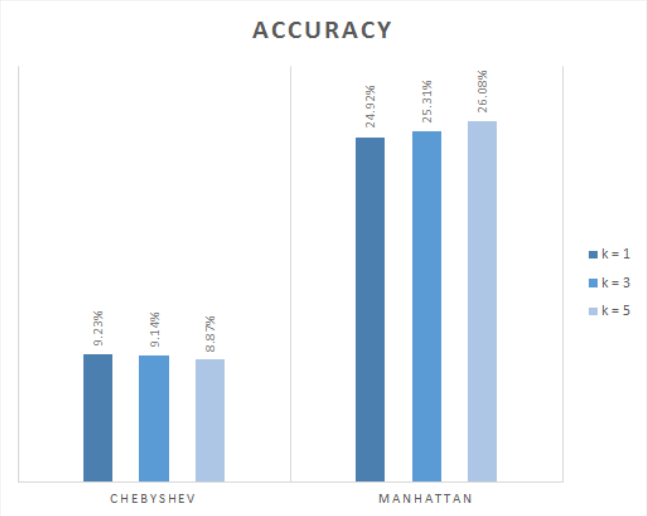

Figure 6. Graph of accuracy for the permutation of hyper-parameters.

5. Conclusion

It's a shame that the resources available to complete the experiment were not enough to see the actual performance of Levenshtein's formula, as it looks very promising at first sight in terms of accuracy. So hypothetically, if enough resources were given and assuming that the result of the 5 image classification is actually an accurate representation of what the algorithm can do in the whole set, it seems like the better choice. It gave pretty consistent results for different values of **k** as well which is something to have in mind as well.

Moving into the formulas that could actually finish classifying the whole set, we have Chebyshev's and Manhattan's. The first one seemed to perform better in terms of time, but was very disappointing when it came to accuracy, with a 9.23% in the best case. The latter was the other way around, performed a bit worse in time, however it's accuracy was more than double than Chebyshev's, with a 24.92% in the worst case. It's important to note as well that while Chebyshev's algorithm seemed to perform slightly worse with higher values of k, Manhattan's was the other way around.

It's clear the the most well rounded algorithm in this specific experiment is Manhattan's distance as the 7 minute difference in time elapsed is not big enough of a drawback to justify the low accuracy that Chebyshev's formula gave. And Levenshtein's 40% assumed accuracy is not good enough either to justify the amount of resources it takes to compute it. This by no means show that any of these formulas are any good at this specific task, as there are algorithms giving 90% or better results, so a completely different implementation is recommended.

References

[1] X. Décoret (2006), *Manhattan distance of a point and a line*, DOI: http://artis.imag.fr/ Xavier.Decoret/resources/maths/manhattan/html/

[2] D. Jurafsky (Unknown Date) *Minimum Edit Distance*, DOI: https://web.stanford.edu/class/cs124/lec/med.pdf

[3] K. Teknomo (2017) *Chebyshev distance*, DOI: http://people.revoledu.com/kardi/tutorial/Similarity/ChebyshevDistance.html