Final Project Report

Joshua Ayaviri

December 11, 2021

1 Algorithm of Choice & Use of Program

This project implements the beginner classification algorithm, k-Nearest Neighbors. Although this implementation is agnostic of the dataset being used, there are a few available datasets in the source code with which one can play around with. The two that are immediately available are the Iris and Glass datasets, stored in the *src/datasets* directory. Along with the two datasets are files which clean and return these datasets in the desired format, which will be further elaborated upon in the document. Since these two datasets are stored locally in the project directory, there is no need to download them.

In order to run this program, the user must enter one of the two following prompts in the command line:

python main.py -manual or *python main.py -automatic*

The command line argument *-manual* allows the user to manually run individual trials of kNN on a given dataset with the ability to tweak any variables at will. The program will not terminate in this mode unless the user quits out of the program forcibly (eg. using CTRL + C on Mac). On the other hand, the *-automatic* argument will run a series of trials of kNN on a given dataset, and output cumulative statistical measures by variable. There are three variables which can be toggled, and their permitted values are as follow:

- distanceMetric: euclideanDistance, manhattanDistance, cosineSimilarity
- k: 1, 3, 5, 7, $\sqrt{n}$
- label: [list of available labels - depends on dataset]

No input is required from the user with this argument. The four statistical measures computed are the **Accuracy**, **Precision**, **Recall**, and **f-Score**. In the case that any one of these cannot be computed (most likely due to a zero in the denominator), the measure will be specified as a string rather than a floating point value. Any other command line argument will terminate the program.

As aforementioned, there are a few available datasets in the *src/datasets* directory, and the user can use any of these by following the last two steps below. However, if the user would like to use this algorithm with any other dataset, **ALL** of the steps below must be followed:

1. Create a file in the *src/datasets* directory called *nameOfDatasetClean.py*
2. Create a function called *cleanData()* with no parameters
3. Clean the data from the file and return it in the following format: list (list (number), string). **NOTE:** This format will treat the training data as a list of tuple in which the first element of the tuple will be the vector representing the features of that given data point, and the second element will be the corresponding label of that data point

4. in *src/main.py*, add the following import:

   from datasets import nameOfDatasetClean

5. In the *manual* and *automatic* functions in *src/main.py*, where the variable *cleanedData* is assigned, replace the present *cleanData* function call with your own

   cleanedData = nameOfDatasetClean.cleanData()

6. Run the program from the command line as described above, using one of *-manual* or *-automatic* as a command line argument

2 Summary of Implementation & Design of Code

Below is a brief summary of each file and its purpose in the *src* directory

- *src/knn.py*: This file stores the implementation of kNN itself. The *knn()* method takes in a training set, which is the list of data points from which the algorithm will classify an element in the query set, a value of k which specifies how many neighbors are to be considered in the voting, a query set, which is a subset of the training population, whose labels are "unknown" to *knn*, and whose labels the algorithm is trying to figure out based on its closets neighbors, and, finally, a distance metric which is to be used in determining the distance or similarity between an element of the training set and an element of the query set.

  The only thing to point out in this implementation is the use of a priority queue or max heap in order to store the current set of the k closest neighbors. The elements in this queue are ordered by distance in descending order, with the farthest neighbor at the root of the queue. Initially, k elements from the training set are chosen at random as the k closest neighbors. Then, *knn* iterates through each element in the training set and checks to see if the current element is closer to the current query element than current farthest neighbor. If this is the case, the neighbor at the root, the farthest neighbor, is extract from the queue, and the current element from the training set is then placed at the appropriate position in the queue. This decision, in retrospect, was not entirely necessary, as k remains mostly constant with respect to the size of the training set. However, a subset of the trials done in the experiment use the square root of n, where n is the size of the training set, as a value of k. As such, the improvement from $O(k)$ to $O(log(k))$ for each element in the training set is minor, and not noticeable in practice.

- *src/priorityQueue.py*: This file stores the representation of the priority queue used in the implementation of *knn*. Behind the scenes, there are three data structures beign used. There are two maps. The first maps from the neighbor's index in the training set [1] to its index in the array used to store the heap itself. The second map is a map in the reverse direction. This allows for quick access of elements by either key or value [2]. Finally, the heap itself is stored not in a binary tree, but in an array whose indices reveal the element's position in the "tree". This array contains the distance from each neighbor to the current query element. Otherwise, the implementation of this priority queue is standard.

- *src/distanceMetric.py*: This file stores all of the available distance metrics. The Minkowski distance metric is included here, but it is not available as an option in *-manual* mode, as it requires an additional parameter. If the user desires to add an additional distance metric, the steps below must be followed:

  1. Write the distance metric as a function in this file. **NOTE:** This distance metric **must only** take in as parameters two vectors, stored as simple arrays of numbers in order to work the program in its current state

[1] It is understood that there is no sense of order in a set, but the training set is being stored in an array, and the easiest way to identify it was to use its index in the array. For all other purposes, the training set can be thought of as a set

[2] I hope to implement a bidirectional map in the future as a slight optimization in favor of this approach

2. Add the following import in *src/main.py*:

from distanceMetrics import nameOfFunction

3. Add to the global dictionary *distanceMetrics* the following entry:

lengthOfDictionary : "nameOfFunction"

4. In this private helper *__chooseDistanceMetric*, add the following entry to the *switcher* dictionary:

lengthOfDictionary :

(lambda firstVector, secondVector : nameOfFunction(firstVector, secondVector))

- *src/trainingData.py*: This file stores all of the information needed to manipulate a dataset. An object of this class, *TrainingData*, is constructed with nothing more than the output of the desired *cleanData()* function of the corresponding dataset. Upon construction, a query set whose size is *sqrtn* is removed from the training set and maintained as the curretn query set. In addition to this, a list of all the labels in the dataset is maintained, and one label from this list is chosen at random as the current label [3]. A user can then perform a variety of operations, the most important of these being the generation of new random query set of a given length. This places all of the elements in the current query set back into the training set, and selects a new random subset of the desired size.

- *src/experimenter.py*: This file stores the function that acts as the middle main between *src/main.py*, the runner of this program, and *src/knn.py*, the implemenation of the algorithm. The function *runTrial()*, as the name suggests, runs a single trial of kNN, taking in a *TrainingData* object, k, the desired size of the new query set, and the desired distance metric. From here, this function creates the appropriate query and corresponding training sets from the *TrainingData* object, runs *knn* once, calculates the four statistical measures of the trial, and outputs them in a dictionary to be used for display and analysis

- *src/database.py*: This file stores the workaround solution for data storage. Initially, I ran into the issue of proper result storage. I wanted to display the results across all trials for each variable, and I could not figure an easy solution for this. My next idea was to connect a SQLite database to the program corresponding to a given dataset, but, in order to avoid the nightmare that this is, I decided to create my own representation of a data store or database in Python that performs the operations I needed. As such, there are two classes in this file. The first class, *SingleTrial* represents a row in a database, with all of the information collected from a single trial. The second class, *ExperimentResults*, stores the entirety of an experiment as a list of *SingleTrial* objects. Given an object of the *ExperimentResults* class, one can obtain the average statistical measures across all trials for a single variable, which greatly improves the readability of the results. This information is then used in the *automatic* function of *src/main.py*, where an experiment, with a trial for each combination of settings described in the previous section of this document, is run until completion

3 Results

All of these results come from the *-automatic mode*.

[3] A note on labels: The term "label" is used through this program. In order to force a binary labelling in multi-labelled datasets, one of the many labels is choosen as corresponding to True in Grouth Truth. A classification of an element in the query set as any other label is then treated as False *during the calculation of the four statistical measures*. As such, this forced binary simplifies the calculation of these measures. The downside of this system is that these calculations only serve to analyse how well the algorithm discriminates against the current label. In reality, however, this implementation of kNN is capable of accurately classifying **ALL** labels. As such, some of the information captured by the algorithm is lost

3.1 Iris Dataset

Results by Distance Metric					
*	Valid Trials	Accuracy	Precision	Recall	f-Score
euclideanDistance	15	0.9611	0.9365	0.9700	0.9468
manahattanDistance	15	0.9611	0.9778	0.9122	0.9369
cosineSimilarity	0	"N/A"	"N/A"	"N/A"	"N/A"

Results by k-Value					
*	Valid Trials	Accuracy	Precision	Recall	f-Score
$k = 1$	6	0.9444	0.9246	0.9667	0.9385
$k = 3$	6	0.9583	0.9444	0.9389	0.9330
$k = 5$		0.9583	1.0000	0.8889	0.9373
$k = 7$	6	0.9861	0.9583	1.0000	0.9762
$k = \sqrt{n} = 12$	6	0.9583	0.9583	0.9111	0.9243

Results by Label					
*	Valid Trials	Accuracy	Precision	Recall	f-Score
setosa	10	0.9750	1.0000	0.9433	0.9687
versicolor	10	0.9583	0.9714	0.9300	0.9431
virginica	10	0.9500	0.9000	0.9500	0.9138

3.2 Glass Dataset

Results by Distance Metric					
*	Valid Trials	Accuracy	Precision	Recall	f-Score
euclideanDistance	21	0.9558	0.9741	0.9150	0.9329
manahattanDistance	21	0.9660	0.9841	0.8789	0.9143
cosineSimilarity	5	0.3000	0.4262	0.4475	0.3877

Results by k-Value					
*	Valid Trials	Accuracy	Precision	Recall	f-Score
$k = 1$	10	0.9357	0.9250	0.9208	0.9706
$k = 3$	10	0.8857	0.8743	0.8786	0.8668
$k = 5$	11	0.9156	0.9394	0.8636	0.8806
$k = 7$	8	0.8482	0.9286	0.7833	0.8429
$k = \sqrt{n} = 14$	8	0.8482	0.9375	0.7685	0.8200

Results by Label					
*	Valid Trials	Accuracy	Precision	Recall	f-Score
"1"	10	0.9714	0.9857	0.9500	0.9650
"2"	15	0.7190	0.8087	0.7206	0.7409
"3"	9	0.9683	1.0000	0.8444	0.8981
"5"	4	0.9643	1.0000	0.7500	0.8333
"6"	3	1.0000	1.0000	1.0000	1.0000
"7"	6	0.9643	0.8778	1.0000	0.9250

4 Conclusion

I believe that this algorithm is a great starting point for ML classification algorithm implementations. It is very straightforward, and a great practice in design, data I/O, and data structures. It is not terribly efficient, and, unfortunately, it suffers from the **Curse of Dimensionality**. The results on these two relatively small

datasets with relatively few features or dimensions is good, but not great. Granted, the average of each statistical measure does not consider the ideal values for each of the other variables we do not intend to change. Instead, it simply queries all of the trials by a given variable. As such, these numbers are on the lower side of what this algorithm and its implementation are capable of. Nevertheless, they serve the intended purpose of developing an intuition for these "dials" and their effects. In my opinion, this algorithm is a great, blind way to gauge the baseline accuracy of any given model. From there, one should take a closer look at the given dataset and use a more appropriate classification algorithm based on its features.