AI Project 1

Carl Sullivan

2023-02-12

kNN Algorithm

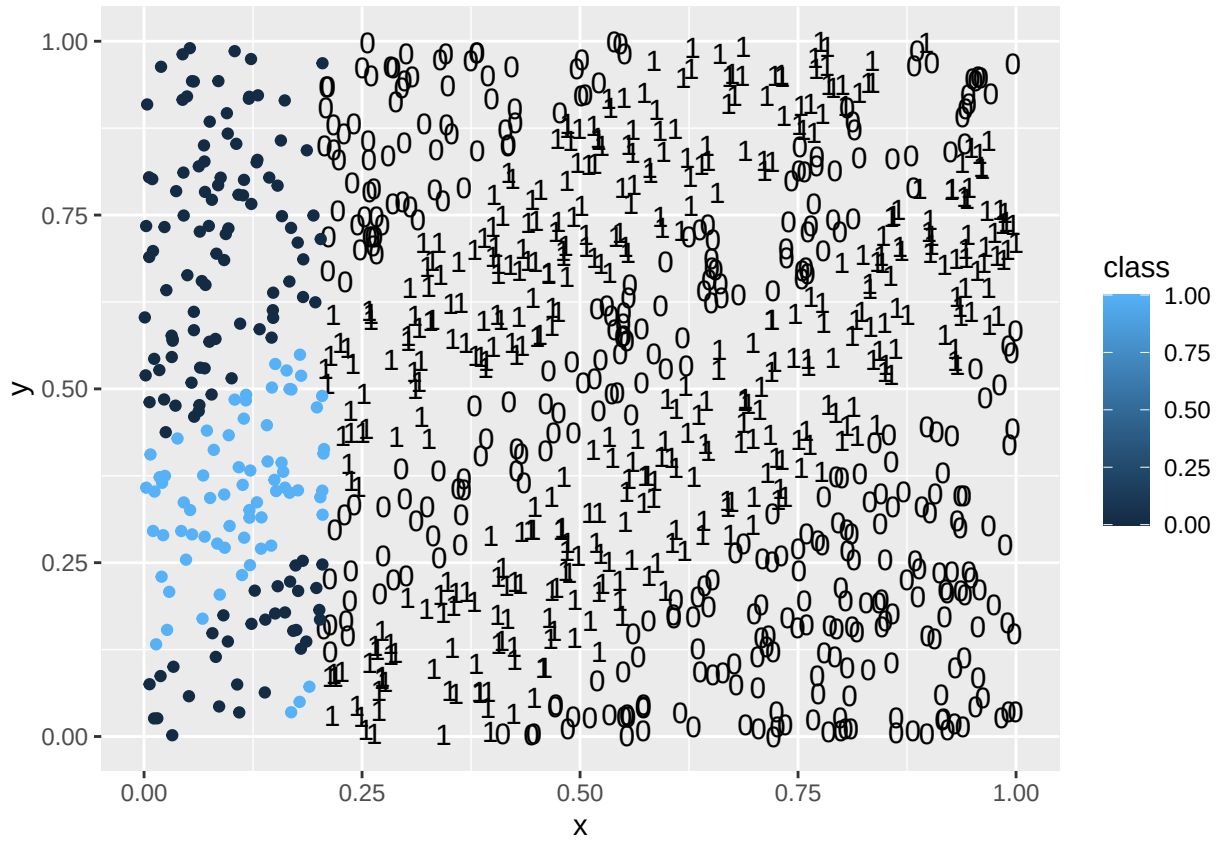
The kNN algorithm is a classification algorithm that takes a training set of data and using a distance formula decides the classification of the new data point based on the most common classification of the neighbors. There are several ways to store the training data, either it can all be stored or a certain number of each classification can be stored then storing only the errors. With storing all of the training data the memory costs can be much higher than when storing the errors but can result in higher accuracy.

To implement this algorithm I used Python and stored the data in a Pandas data frame where I was able to take slices of the frame for each training and testing instance. I considered using dictionaries for the algorithm with the item number being the key and storing an array for the x, y, and class attributes. I went with Pandas though as it gave me an opportunity to practice working with data frames and offered a clear view of the data when debugging. Some troubles I ran into with the data frames though were correct parameters and return types. With data frames, if I took a slice of several rows it would stay as a data frame but when taking a single row using built int Panda functions it can sometimes return a series.

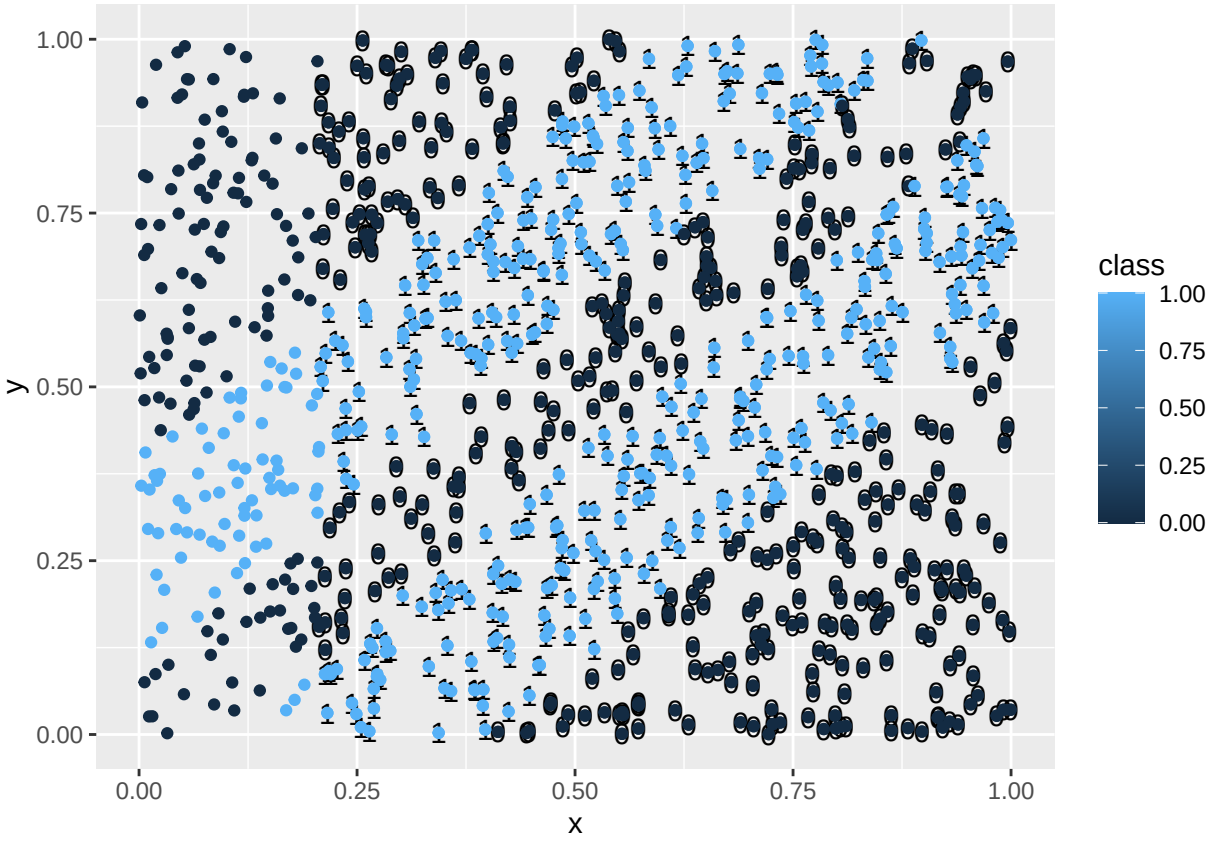
I ended up with two implementations of the kNN algorithm, the first one 'part1_a' I believe is better written than the second but does contain a logical error I have not been able to pinpoint, especially with only storing the errors. This logical error has caused the testing data to perform poorly against the store-all and store-errors data frames, and the only accurate result that is returned with testing the training data against the store-all data frame. The second implementation 'part1_b' returns accurate results but I believe it could be optimized with how I pass data frames and the resulting calculations. It has provided accurate results with the labeled training data across all benchmarks. The testing tables below show the accuracy of 'part1_b'.

Besides the 'labeled-data' I ran several tests with the 'diabetes.csv' and used different attributes for the 'x' and 'y' axis to try and predict which pair would give an accurate prediction. My best attempt was using age as the x-axis and the Diabetes Pedigree Function as the y-axis. Although the results aren't impressive the test accuracy against the store-all was 60% and the test accuracy against the store-errors was 54%. This could potentially be improved by testing all of the columns against each other and averaging out the predictions.

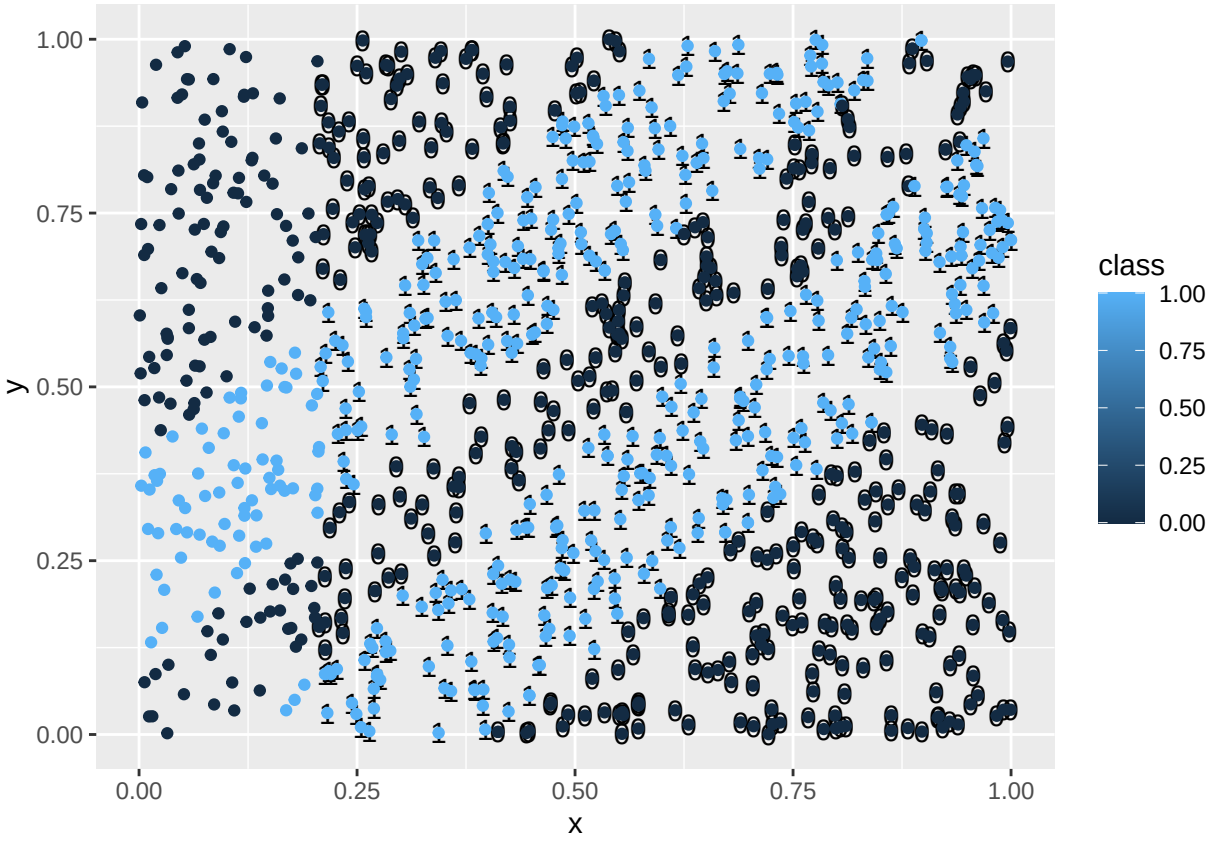
```
library(ggplot2)
library(tidyr)
trainData <- read.csv('part1_b/DataCSV/TrainDF0.csv')
testData <- read.csv('part1_b/DataCSV/TestDF0.csv')
store_all <- read.csv('part1_b/DataCSV/StoreAll0.csv')
store_error <- read.csv('part1_b/DataCSV/StoreError0.csv')

trainData %>%
  ggplot(mapping = aes(x=x,y=y)) +
  geom_text(aes(label= class)) +
  geom_point(data = testData, aes(color = class))
```




```
trainData %>%
  ggplot(mapping = aes(x=x, y=y))+
  geom_text(aes(label= class)) +
  geom_point(data = store_all, aes(color = class))
```




```
trainData %>%
  ggplot(mapping = aes(x=x, y=y))+
  geom_text(aes(label= class)) +
  geom_point(data = store_error, aes(color = class))
```



As seen above, the 'part1_b' kNN algorithm inherits all of the data points, in my train model function the three data frames used are the training set of data, storing all of the training data, and storing the first 5 of each class then only the errors. Before returning the two storage data frames I evaluated the data points in them and they seemed to be working as designed but when the data frames are returned in a list to be stored in the model_train variable the data frames somehow appended to the original data frame that was first passed to the initialization function. I have narrowed it down to this point but cannot find a solution that returns only the new data frames. I suspect it has to do with how Python and Pandas interact with function calls and passing by value and reference.

Alternatively as seen below with 'part1_a' the errors data frame is collecting enough of the data points whose prediction was incorrect.

```
trainData <- read.csv('part1_a/DataCSV/TrainDF0.csv')
store_error <- read.csv('part1_a/DataCSV/StoreError0.csv')

trainData %>%
  ggplot(mapping = aes(x=x, y=y))+
  geom_text(aes(label= class)) +
  geom_point(data = store_error, aes(color = class))
```

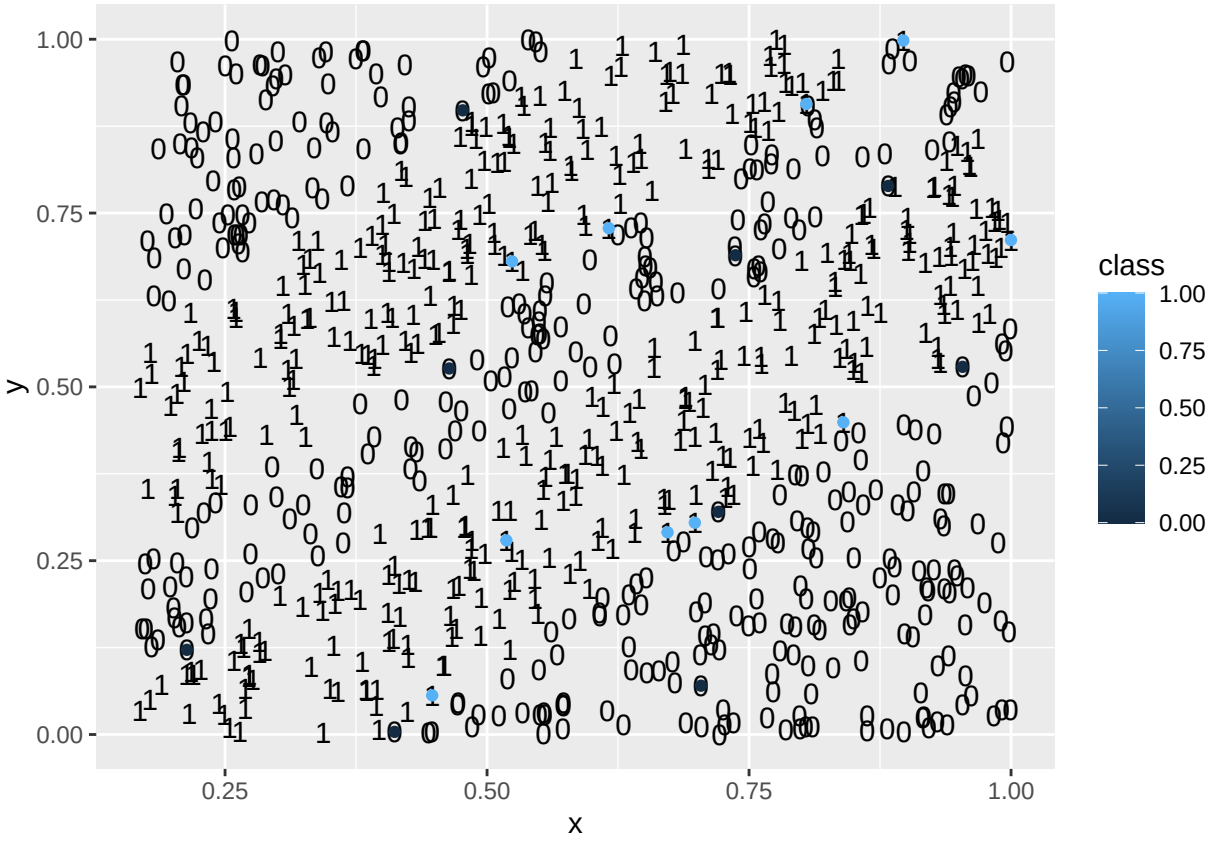


Table 1: kNN Table

Data.File	k.Value	Test_acc_all	Test_acc_error	Train_acc_all	Train_acc_error
Labeled Data	3	89	88	96	89
Labeled Data	5	88	88	95	86
Diabetes (Age vs DPF)	3	60	54	61	45
Diabetes (Age vs BMI)	3	58	54	63	47
Diabetes (Age vs DPF)	5	44	42	62	37
Diabetes (Age vs BMI)	5	46	46	63	44

Simulaged Annealing

The Simulated Annealing algorithm is used to solve optimization problems by a technique of heating and cooling, similar to how metals are hardened. The initial solution is given to the algorithm and from there it is used to find the best solution for a particular problem by iteratively making changes to it. As a new solution is made it is compared to the previous solution and becomes the new solution if better, if it is not better there is a small probability that it could become the new solution. This probability is determined by the temperature function.

For the Simulated Annealing algorithm, I had difficulty understanding the temperature function and how its role was used in determining the probability and solutions. After I understood how the schedule function determines the temperature by slowly decreasing over time. The probability that the next, even though it may not be a better solution than the current is then determined by the difference between the current and the next.

I implemented the schedule function with several different values starting with $\frac{t}{100}$ and increased it by a factor of till $\frac{t}{100000}$ and received the following values below in the table for the various problems. The values in the table are grouped by problem then returning the minimum average for each problem. I used random input vectors that were contained within the parameters of each simulated problem. The full list of the scheduling problems shows the drastic role scheduling plays in finding the optimal solution. Without enough time the annealing doesn't have a chance to find minimal solutions to the problem. The data shows that between 100,000 and 1,000,000 the solutions start to level out.

Table 2: Scheduling Simulated Annealing

eval.problem	schedule	array	result1	result2	result3	result4	result5	average	min
griew	1e+03	[152,137]	1.018023e-01	2.548349e-01	2.207599e-01	2.712990e-01	1.473315e-01	1.992055e-01	1.018023e-01
langermann	1e+04	[2,9]	-1.245884e+00	-1.186182e+00	-1.171712e+00	-1.189639e+00	-1.234542e+00	-1.205592e+00	-1.245884e+00
micha	1e+06	[-77,38]	-1.939006e+00	-1.839670e+00	-1.982513e+00	-1.922269e+00	-1.962829e+00	-1.929257e+00	-1.982513e+00
shekel	1e+05	[5,8]	-7.389049e+04	-4.373225e+04	-4.480767e+04	-5.011577e+04	-5.930341e+04	-5.784054e+04	-7.930341e+04
sphere	1e+06	[5,-1]	2.122300e-03	3.590000e-04	2.760400e-03	2.482500e-03	7.153700e-03	2.975600e-03	3.590000e-04

Genetic Algorithm

The Genetic Algorithm was developed with the idea of natural selection and survival of the fittest, this is due to how it takes two points of data and "breeds" them to make a child with an input of randomness. The solution is found by starting with a population of potential solutions and then iteratively evolving them and keeping the best solutions in the problem.

The Genetic Algorithm was the most difficult for me to implement as there were several helper functions that were necessary for the implementation but weren't specifically intuitive at first. Fitness, crossover, and mutation are three of the helper functions that are used to consistently evaluate and change up the data so that it emulates the original set of data. The functions themselves weren't necessarily difficult the difficult portion was how they fit within the actual algorithm.

In the Genetic Algorithm, I iterated through population sizes of 5, 10, 15, and 20 randomly generated individuals of 52-bit float strings. Next was to select two individuals based on the highest fitness score, the fitness was determined by the given problem in the function. Once the fittest parents were selected the values are reproduced by selecting a random index and dividing the two parents by this index to cross the strings into two new children. The children undergo a random mutation if the random number is under a probability. The mutations iterated through 2.5%, 5%, 7.5%, and 10% mutation rates. Lastly, the algorithm iterated through generations of 50, 75, 100, and 150. As seen in the table below the most common variable values are a population size is 5, generations of 100, and a mutation rate of 0.05.

```
library(knitr)

trials <- read.csv('part2/DataCSV/ga_results.csv')
trials %>%
  group_by(eval.problem)%>%
  slice(which.min(average)) %>%
  kable(caption = 'Genetic Algorithm test tables',format.args = list(scientific = FALSE))
```

Table 3: Genetic Algorithm test tables

eval.problem	bit.len	pop.size	generations	mutation.rate	result1	result2	result3	result4	result5	average	min
bump	52	5	50	0.05	0.9918945	0.9985209	0.9985167	0.9926265	0.9796961	0.9922509	0.9796961
griew	52	20	100	0.10	0.0002528	0.0000668	0.0004712	0.0001382	0.0000002	0.0001858	0.0000002
langermann	52	20	100	0.05	0.0002068	0.0000032	0.0001552	0.0000261	0.0002903	0.0001363	0.0000032
odd_square	52	5	75	0.10	0.0074224	0.0026507	0.0078582	0.0072918	0.0083890	0.0067224	0.0026507
shekel	52	10	100	0.05	0.0469831	0.0466011	0.1899194	0.0468290	0.0462604	0.0753186	0.0462604
sphere	52	5	50	0.05	0.9927192	0.9921088	0.9949082	0.9904186	0.9877572	0.9915824	0.9877572

Conclusion

With each of these algorithms, there were difficulties in implementing them to logically function correctly. But as seen with each result there is an advantage to the different algorithms. The kNN won't find the optimal solution but can be used to classify a data point. With the genetic and simulated annealing algorithms, they both try to find an optimal solution but use very different means. The simulated annealing algorithm bounces around compared to the genetic where a solution is found through survival of the fittest. It is difficult to determine which of the two finds a better solution or finds the solution faster as the scheduling and generations would need to be of similar magnitude.