Abstract

The algorithm I chose to use was k-nearest neighbors. I chose this algorithm because my machine learning problem I am trying to solve is classification and because k-nearest neighbors is considered to be a relatively simple algorithm. My task was to solve a supervised classification problem given glass data which included the refractive index and various element compositions by percent of the glass.

Theoretical Description

K-nearest neighbors is a supervised machine learning algorithm used primarily for classification and regression. The k stands for the number of neighbors that will be used for classification when the algorithm is executed. It is a straightforward algorithm that does not require parameterization. The k-nearest neighbors algorithm begins by using training data to create "neighbors," or cluster or category of data. Then, using data that k-nearest neighbors has never seen, the algorithm attempts to classify based on the nearest neighbor(s) using a distance calculation, depending on what the selected k value is. If the k value is equal to 1, then the algorithm will use purely the closest neighbor for classification. If the k value is 10, then k-nearest neighbors will query the 10 nearest neighbors. When a test data point's nearest neighbors are split between different categories, the traditional k-nearest neighbors algorithm will classify based on the category with the highest neighbor count.

Choosing the best value of k to fit one's data is the primary challenge of using k-nearest neighbors. Finding the best value of k is not trivial since there are not parameters that can be optimized through an optimization algorithm. Instead, what is typically done to minimize the error is to use a model validation method to show the results with the least amount of error while k is changing from execution to execution and select the ones with the lowest error rate.

Another common, and sometimes necessary, method of optimizing k-nearest neighbors is to scale the feature data. This step is sometimes necessary because of how the distance might be calculated. Using Euclidean distance to measure the distance between income in age groups might result in a difference of 10,000 when measuring the amount of money, but for measuring age, the differences are levels of magnitude lower. This causes the algorithm to rely solely on the currency difference because it believes that the age difference is negligible. There are two common techniques of feature scaling: Standardization and Normalization. I chose to use normalization because that method was easier to implement. I found that when I do not normalize my data, the error rate jumps up by 5%-10%. There are various techniques for normalizing data and I chose to use Min-Max Scaling.

Another way of optimizing k-nearest neighbors is to change the type of distance function used. For my implementation, I used Euclidean distance, but I also tried to implement a cosine distance function, but ultimately ran out of time for testing.

I also implemented a cross-validation algorithm to assess the results of my k-nearest neighbors algorithm. Cross validation is a model validation technique used to split data measure how accurately a model will perform. Cross validation is primarily used in predictive models to split the data to evaluate and compare different machine learning algorithms with the same data set. The way I used cross validation in my program was to split my glass data into a training set and a testing set. The type of cross validation I used in my program was k-fold cross validation, and to avoid further confusion with k-nearest neighbors, I will use capital K to denote the fold number. I used cross validation to split my data into a user-defined K folds for processing in the k-nearest neighbors.

Implementation Details

The libraries I used for my program are numpy, matplotlib, and from the Python standard library I imported csv, random, time.

In my project, I used some helper functions. My accuracy function begins on line 8. This function takes two vectors(lists) as parameters and computes the average of the correctly predicted data and returns that number as a float. My Euclidean distance function begins on line 16. It takes in two vectors as parameters and returns the straight-line distance between the two vectors. This function is used heavily in the main k-nearest neighbors algorithm. My normalization function begins on line 34. This function takes in the dataset as a parameter and rescales the data to be within the range 0-1 using this formula in Figure 1

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Figure 1

My k-nearest neighbors implementation begins on line 45. It takes three parameters: a training set, a testing set, and the number of neighbors to be used to classify. Next, for every testing data row, the Euclidean distance function is called and computes the distance between that row and every training data row and appends the training row used and the distance from that training row to a list; each element in this list consists of a 1x10 vector of the features, and the distance from the testing to the training vector. Since this list is essentially a python dictionary without labels, a lambda function is used to sort by distance, with the closest feature at index 0 of the list. Next, the algorithm builds the closest neighbors to the current testing data point up to k neighbors. Then, the algorithm attempts to classify by taking the last column, which is the glass type, and calculates which type has the highest representation using the built in python max() function, and appends it to a list of predictions, which is returned.

My cross validation function begins on line 68. This function takes the dataset and the number of folds as the two parameters. This functions begins by creating an empty matrix to be used as the return value later. Then it creates a copy of my dataset. For every K fold, a random selection of data the size of the fold is chosen using Python's built-in random library to be popped out of the dataset copy and appended to a matrix. For example, if 4 folds are chosen, then the length of data divided by number of folds (214/4) is used to keep track of when each fold begins and ends. This ensures that the same data point will not be copied into different folds. The matrix that contains the popped data is returned after all the folds have been iterated through.

The "main" function of my program begins at line 83. Here, the random seed for my cross-validation is chosen, and I begin opening my data file for parsing. I use a reader from the csv library to parse my data. One difficulty I had with my program was with converting string values to real numbers for my algorithm. Each row in my data file is read and converted into floats, with the exception of the last data point in each row, which is converted to an int type because that is the glass type variable. After each row is converted into usable data for my program, it is appended to a list variable and acts as a dataset matrix, with dimensions [214, 10]. At line 131, my program begins plotting the features of my data using scatter() and subplot() from matplotlib. Here in Figure 2 are the plots of my data using 3x3 subplots with the x-axis as the feature label, and the y-axis for all of the plots excluding the first are weight content by percent.

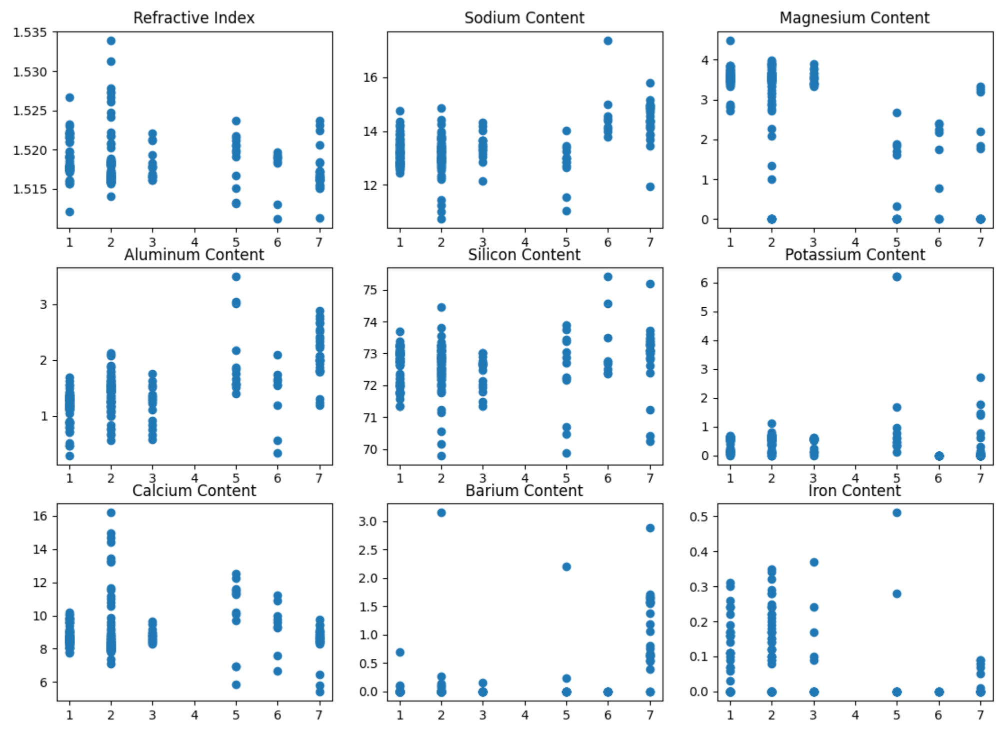

Figure 2

The axis labels were not shown on the subplots on purpose to save space. Refractive index has no units since it is a ratio.

The fold variable, n_folds, used in the cross validation function can be changed here on line 147. On line 148, the value for k can be changed. Next an empty list is created to store the error percentage. Then, the cross validation algorithm is called, which returns the folds as a matrix of matrices. The training and testing set are created based on which cross validation fold the program is currently iterating through, the k-nearest neighbors algorithm is finally used, and the results are compared by calling the accuracy function. The results are displayed as a decimal percentage value for each fold and then the mean accuracy for that fold is displayed. Once each k value has been used, the mean accuracy for each k value is calculated and displayed along with the average error rate as a percentage.

Reproducibility

I coded using PyCharm. To reproduce my results, just run finalProject.py. Initially, Figure 2 will be plotted; to move on, close out of Figure 2. The program will then print the results to the console and display Figure 3. The data I used was the Glass Classification task created by UCI Machine Learning, and it can be found here.

Results

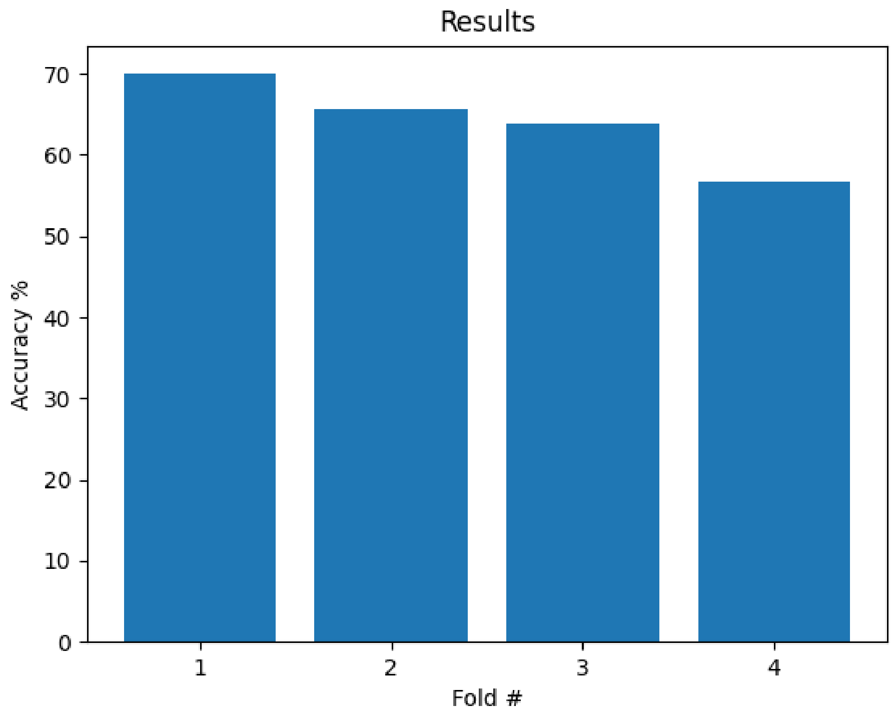

Figure 3

The lowest error rate that I have received through testing was 30%. I am questioning the validity of this result as it was when I set the k to be 1. I get an average resulting error rate between 30%-40% when I run my program. The most consistent error rate was between 33%-36% repeating. I mostly used fold numbers from 5-10, and I tested extensively with k values 1 through 10, and I found that the results are the best when the upper bound for k on the range in the loop on line 152vvv is 5 or below, and when the lower bound is equal to 1. The average time for one k iteration is between 0.42-0.49 seconds. For memory usage, I used Python arrays and lists, as well as numpy arrays.

There were many factors as to why the results of my k-nearest neighbors algorithm has a high error rate. Firstly, the dataset I used had 9 dimensions not including the label. K-nearest neighbors performs poorly as the dimensionality of the data increases. K-nearest neighbors is also sensitive to missing values and outliers, which my dataset contained many of. Had I modified my data, this issue might have been avoided. In addition, I believe that the feature-scaling may have had a small effect on the results. Since k-nearest neighbors almost requires feature-scaling, in hindsight, I do not think that k-nearest neighbors was the correct algorithm to use for this dataset. Another variable that could have affected the data is the distance calculation used. I was unable to get a different distance function to work, such as cosine distance or Chi-Square distance.

By completing this project, I learned about the importance of choosing the correct machine learning algorithm that matches the problem at hand. Similarly, I did not check the dataset I chose thoroughly enough, and that taught me the importance of making sure that the data chosen is complete and that it fits well with my chosen algorithm. I learned the details about the k-nearest neighbors algorithm and its advantages and disadvantages. The same applies for cross validation, which I did not grasp when learning about it in class, but now I understand how cross validation works and why it is important.

References

https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

https://en.wikipedia.org/wiki/Refractive_index

https://en.wikipedia.org/wiki/Cross-validation_(statistics)

https://en.wikipedia.org/wiki/Euclidean_distance

https://medium.com/analytics-vidhya/why-is-scaling-required-in-knn-and-k-means-8129e4d88ed7

https://stats.stackexchange.com/questions/215011/finding-the-optimal-value-of-k-in-the-k-nearest-neighbor-classifier-is-this-cro

https://stats.stackexchange.com/questions/287425/why-do-you-need-to-scale-data-in-knn

https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/

https://en.wikibooks.org/wiki/Data_Mining_Algorithms_In_R/Classification/kNN

StatQuest: K-nearest neighbors, Clearly Explained

Dataset Source