Liam Durcan

Prof. Gardos

DS210

1 May 2025

Final Project Report

A. Project Overview

For my project, I decided to build K Nearest Neighbors from scratch in Rust, as wanted to completely think through the logic behind the algorithm while applying what I've learned in Rust to the world of machine learning. With my K Nearest Neighbors model, I used all of the quantitative/numerical variables from a synthetic dataset of patients to predict whether they have a history of a heart attack. Doing so meant using a 1,000 row dataset from Kaggle. (https://www.kaggle.com/datasets/shriyashjagtap/heart-attack-risk-assessment-dataset?resource=download). One concern of mine was that the data was synthetically generated (i.e. not representing real patients) so I wasn't sure how accurate my model would be given the randomness of the data. However, I was pleasantly surprised to find my model's accuracy in the mid-high 80s (in terms of percent) every time I ran the simulation with 3 neighbors.

B. Data Processing

Both the original dataset and the version I cleaned can be found within my github repo. The most difficult cleaning I did involved normalizing every column of numerical data. This was done so that large differences in one dimension don't weigh more than the differences in another dimension when calculating euclidean distance between vectors (data points). For example, a 40 point difference in systolic blood pressure would increase euclidean distance between two vectors far more than a 20 point difference in diastolic blood pressure, even though these two

differences could be taken equally as seriously as diastolic blood pressure is always much lower. I performed this normalization in excel by subtracting each value in a column from the column's minimum value, and dividing this difference by the difference between the column's max value and min value. This put every number in every numerical column between the values of 0 and 1, giving each column the same range of numbers and possible differences between those numbers. I also replaced the 0s and 1s indicating False/True with the words False or True respectively. This allowed me to easily label them as booleans when reading in the csv.

To read in this data, I used the read_csv function we wrote as part of our homework 8, which uses csv::ReaderBuilder and a series of for loops and match statements to iterate through each row of the csv file and populate Vectors of ColumnVal enums based on the type of data each element contains. These vectors of ColumnVals are within a HashMap as values and the column title in the first row; they thus represent each Column in the DataFrame, as each type of ColumnVal must be the same and there's a new vector for each element in the row.

C. Code Structure

a. Modules

mod reader (reader.rs) - This module contains all the functionality for reading in the csv. The read_csv function uses csv::ReaderBuilder to iterate through the csv and store the data from every column in a HashMap, where the column title is the key and the values are vectors of ColumnVal enums. This essentially recreates the DataFrame struct from homework 8, which I intentionally used so that I could call the restrict_columns() function when narrowing the DataFrame down to just its numerical columns which I wanted to work with for data or down to the final column indicating heart attack history which I wanted to use as my labels.

mod knn (knn.rs) - This module contains all the functions for carrying out K Nearest Neighbors. There are two helper functions that the public k_nearest_neighbors function (called in main.rs) utilizes. The euclidean distance function is useful for finding the distance between any two vectors, while the nearest_vectors function uses this function to find the nearest desired number of training point vectors to the test vector its presented. The main k_nearest_neighbors function then calls the nearest_vectors function with the training data and each point in the testing data and has the nearest reported vectors vote on a label for the testing point, which is then compared to the testing point's actual label.

b. Key Types in reader.rs

The ColumnVal enum represents the different data types that could potentially be read in from the csv, and it has String, bool, f64, and i64 variants. The DataFrame struct is how all of the data from the csv is stored, as the columns HashMap will keep every value from a column in sorted order via a vector (which is the value) and the column title. It goes without saying that num_rows simply contains the number of rows in the csv, which should be equal to the length of each vector of ColumnVals within the columns HashMap. The column_order's main purpose is to store the column titles in the order that they are read in for each row. All variables within the struct are made public for cases where I'd need to access their values in main.rs.

c. Key Functions

pub fn read_csv(&mut self, path: &str, types: &Vec<u32>) -> Result<(), Box<dyn Error>>
This function takes a DataFrame struct (theoretically empty), a file path, and a vector of types for each respective column as inputs. The return value is either nothing (changes to the DataFrame struct were seamlessly made) or raises an error if for example it reads an unfamiliar data type that doesn't fit one of the ColumnVal enums. Simply put, the function reads the first row of the csv as column headers which become keys in the columns hashmap and values in the column_order vector. Iterating through each subsequent row, elements are added to the vectors in the columns HashMap that correspond to their columns title and the num_rows variable

increases by 1 with each iteration. At the end of this for loop, the DataFrame will be fully populated.

pub fn restrict_columns(&mut self, column_names:Vec<String>) -> Result<(), Box<dyn Error>>
Rather than return a new DataFrame, this function alters the original DataFrame and raises an error if one of the provided column names in the argument is not in the DataFrame. Otherwise, the function creates a new columns HashMap and column_order vector to populate. Then, it iterates over all the columns in the column_names vector argument, and adds clones of the key value pairs to the new columns HashMap while pushing a clone of the column title to the new column order vector. At the end of this loop, the self.columns and self.column_order variables are updated with their respective new HashMap and Vector.

fn euclidean_distance(vec_1: &Vec<f64>, vec_2:&Vec<f64>) -> Result<f64, Box<dyn Error>>
This function takes in two f64 vectors and will only return an error if the two vectors it takes in are not of the same length. Otherwise, it iterates over each index in both vectors and computers the squared distance between the two. After summing these distances up, the function takes the square root of this sum and returns Ok(sum). Mathematically, this is the same as finding the euclidean distance or shortest distance between two points, and this is meant to be a helper function for the nearest_vectors function.

fn nearest_vectors(train_data: &Vec<Vec<f64>>, test_point: &Vec<f64>, num_neighbors:u32) -> Vec<usize>
This is a helper function for the public k_nearest_neighbors function below that puts the whole algorithm together. This function will loop through all the f64 vectors in the training data and find their euclidean distance to the testing point, adding a tuple in the form (index of the training vector, distance to test vector) to a vector called distances. After all the distances have been found, the vector of tuples is sorted by putting the distances in ascending order (such that the points closest to the test point are at the front of the vector). This newly sorted distances vector then has its first n elements iterated through, where n is the number of nearby neighbors we're seeking to compare. These elements then have the indices (first element of the tuple) added to a vector neighbor_indices, which is then returned. Thus, this function returns the indices of the training points in the training data that're closest to a given testing point.

pub fn k_nearest_neighbors(train_data: &Vec<Vec<f64>>, test_data:&Vec<Vec<f64>>, train_labels:&Vec<bool>, test_labels:&Vec<bool>, num_neighbors:u32> -> (Vec<bool>, f32)
This function fully executes the k nearest neighbors algorithm as we know it, taking training data, testing data, training labels, testing labels, and a desired number of neighbors. The function

outputs a tuple containing a vector of all predictions it made for test data points and its accuracy. The function's outermost for loop iterates over every vector in test_data and passes it through the nearest_vectors() helper function to get the indices of the closest training data points. From here, the function finds these neighbors' labels from the train_labels vector and stores them in a separate vector called neighbor_labels. To count the frequency of each label, I iterated through the vector and used a HashMap to track each label's number of occurrences. Since HashMaps are unsorted by nature, I then converted each key-value pair into a tuple and collected these pairs into a vector that could be sorted using .sort_by(). The vector was sorted such that votes were in descending order and the most frequently occurring label was the first tuple in the vector of tuples. Once I found the most frequent label among the nearest neighbors, I added it to the vector of predicted labels and compared it to the test point's true label. If the prediction was correct, I added 1 to the correct_predictions variable which I divided by total_predictions at the end of the function to track accuracy. Finally, the vector of predictions and the accuracy were returned as a tuple.

d. Main Workflow

All of these functions operate cohesively in fn main() of main.rs. The data is first read in to a DataFrame struct with the read_csv() function from reader.rs. From here, the numerical data and the column of heart attack history are both isolated by cloning the original DataFrame and calling restrict_columns() on these clones with the relevant column titles as arguments. In between calling these reader.rs functions and the k_nearest_neighbors function, further changes were needed to how the data was handled. The first order of business was to rearrange how the numerical data was stored, as each row needed to be its own vector rather than each column. This meant iterating over each column at a certain index and storing all values at that same column index in a vector before incrementing the index by 1. Additionally, a match statement was needed in order for the values to go from ColumnVal enums to pure f64s that we could do math with. Once all the data was stored as f64s in rows, another for loop and match statement were needed for the labels, which were stored in a separate vector of Booleans once the match statement properly destructures the reference to the enum. Finally, each row of data was paired with its label, shuffled, split into training/testing data, and the k_nearest_neighbors function was

called on it. This involves not only the public function itself, but also both of the behind-the-scenes helper functions mentioned above.

D. Tests

Output of cargo test

```
    Finished `test` profile [unoptimized + debuginfo] target(s) in 8.05s
     Running unittests src\main.rs (target\debug\deps\final_project-8b675e42309e89e0.exe)
running 3 tests
test knn::tests::test_euclidean_distance ... ok
test knn::tests::test_nearest_vectors ... ok
test knn::tests::test_k_nearest_neighbors ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

All three of my tests are carried out within the knn.rs mod file, each one testing one of the three functions I wrote for KNN. The first one, test_euclidean_distance(), is the most straightforward. I defined two vectors, the first one being the zero vector and the second one being [3.0,4.0,10.0,10.0]. When squaring the differences between the vectors' respective components, they sum to 225, the square root of which is 15. Since I thus expected an output of 15.0 from my eucliean_distance() function, I set up my call to assert_eq!() accordingly and got my desired passing output. The second test, test_nearest_vectors() had a test point at the origin and five vectors of hypothetical training data that went into their own vector. I then passed this vector of vectors into the nearest_vectors function with the testing point and an argument indicating that I wanted the three nearest neighbors. I expected the first three vectors in the training data vector to

be the closest vectors because when I wrote the vectors, I intentionally had them get further away from the testing point each time. Thus, I stored my expected output of indices 0,1, and 2 (the first 3 vectors) from the vector of vectors and called assert_eq() to compare my expected output to the actual output of nearest_vectors(). Finally, test_k_nearest_neighbors() was the most elaborate test and the most important one to get right as this is the function I actually use in main.rs. I created two test points and a corresponding vector of two labels, one true and one false. I then came up with 5 testing points, giving only 1 of them a label of false. This meant that for 3 neighbors, both test points would be assigned labels of true so the model would be 50% accurate. Storing my expected accuracy and expected label predictions in the same tuple format that the function outputs, I used assert_eq to compare this tuple to the output of the function itself with the 4 vectors (training points, testing points, training labels, testing labels) and the number of neighbors (3) as arguments. Since the function passed, I know it calculates its accuracy and makes its label predictions how it's intended to.

E. Results

Output of cargo run –release

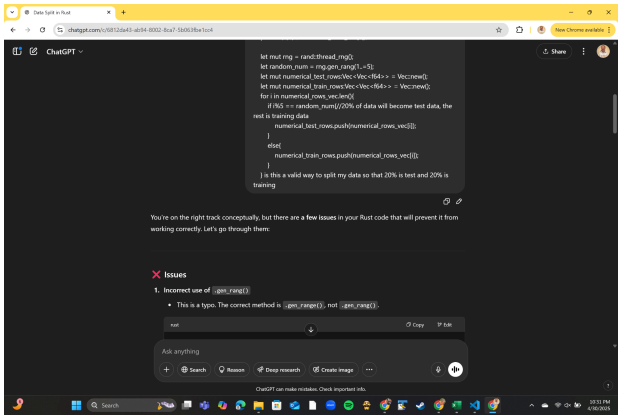
```
Finished `release` profile [optimized] target(s) in 8.85s
Running `target\release\final_project.exe`
KNN Accuracy on Randomized, Synthetic Data: 0.85
```

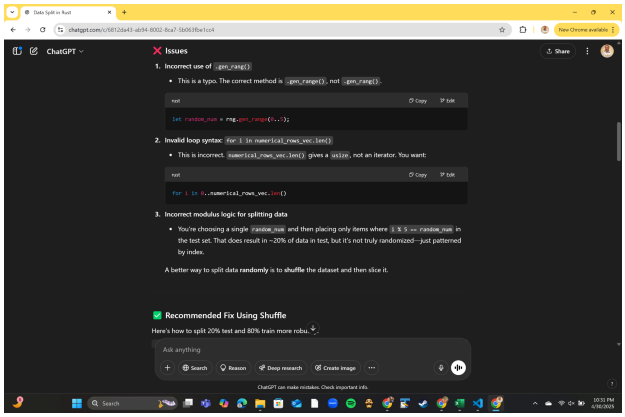
In context, this means that for this set of data that was randomly split up (80% training data, 20% testing data), my K Nearest Neighbors algorithm predicted the labels of the testing data with 85% accuracy.

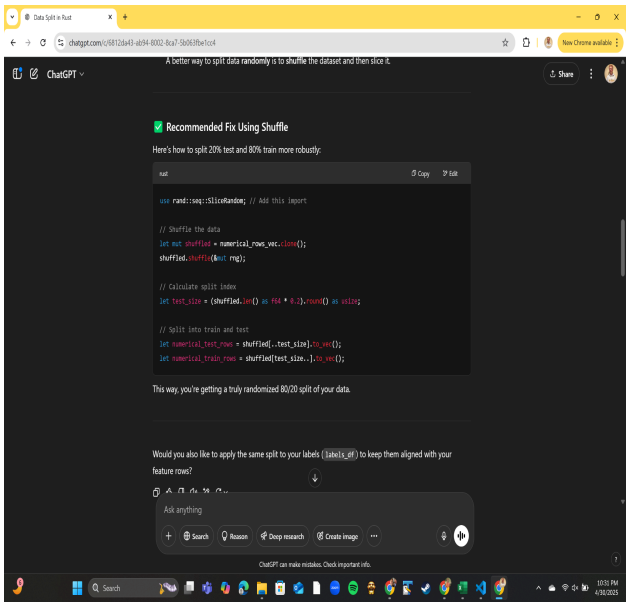
F. Usage Instructions

The project should overall be pretty simple to run. Cargo run or cargo run - -release both work relatively efficiently (using - -release takes the program about 5 or 10 seconds on my computer while just using cargo run takes it closer to 20). My project doesn't involve any user interaction in the terminal beyond simply running the program, but one could adjust the num_neighbors argument in the main.rs call to k_nearest_neighbors if curious about how a different number would affect the accuracy.

G. AI Assistance Disclosure







These three screenshots reveal the main significant interaction I had with ChatGPT/AI while building this project. When trying to debug an error associated with the way in which I split up my data into training and testing data,

ChatGPT proposed a much more concise way of doing so. I debugged my method (see below), which involved picking a random number 1 through 5 and iterating through the dataset and picking out every 5th row starting with that randomly selected index. However, ChatGPT introduced me to the shuffle() method from the rand::seq::SliceRandom crate, which completely randomizes the order of all 1,000 rows. Afterwards, I simply picked the first 20% of the data by slicing the new randomly sorted vector at this point. Also, I adjusted ChatGPT's code for my use case. This meant pairing the vectors of numerical data with their labels in a tuple and storing these tuples in a vector before shuffling/randomizing. This way, the data and their labels wouldn't get jumbled or randomly sorted in different orders like if I were to have shuffled the data and labels vectors separately. Once the vector of tuples was shuffled and split into test and training vectors, I used .iter().clone().unzip() to go through the vectors and unzip clones of the values in each tuple such that the data and labels were again stored separately. From here, references of the four vectors (training data, testing data, training labels, testing labels) could be passed as arguments to the k_nearest_neighbors function.

Here's my original means of splitting the data (which I later commented out when implementing ChatGPT's version).

```
//Split the data so that 20% is test data, 80% is training data
    /*
    let mut rng = rand::thread_rng();
    let random_num = rng.gen_range(1..=5); //Randomly select 1 in every 5
rows
    let mut numerical_test_rows:Vec<Vec<f64>> = Vec::new();
    let mut numerical_train_rows:Vec<Vec<f64>> = Vec::new();
    let mut test_labels: Vec<bool> = Vec::new();
    let mut train_labels:Vec<bool> = Vec::new();
    for i in 0..numerical_rows_vec.len(){
        if i%5 == random_num{//20% of data will become test data, the rest
is training data
            numerical_test_rows.push(numerical_rows_vec[i].clone());
```

```
            test_labels.push(labels[i]);
        }
        else{
            numerical_train_rows.push(numerical_rows_vec[i].clone());
            train_labels.push(labels[i]);
        }
    }*/
```