Ivan Wiandt
Collaborators: None
Sources:

1) https://doc.rust-lang.org/stable/std/iter/
2) https://stackoverflow.com/questions/26033976/how-do-i-create-a-vec-from-a-range-and-shuffle-it
3) https://users.rust-lang.org/t/solved-efficient-rust-random-shuffle-of/25350

Note: Run using –release
Github link: https://github.com/ivszendvics/210final/tree/main

Overview

My goal for this project was to make a simple KNN algorithm for a dataset. In this case, it was to figure out whether a new datapoint representing a student had depression based on their answers to questions (i.e. workload, stress, etc)

1) Dataset I'm using: https://www.kaggle.com/datasets/adilshamim8/student-depression-dataset?resource=download

The csv file associated is also in the github.

The csv file has information about each student as shown below (screenshot is cropped for readability)

	A	B	C	D	E	F	G	H
	id	Gender	Age	City	Profession	Academic Pressı	Work Pressure	CGPA
	2	Male	33	Visakhapatnam	Student	5	0	8.9
	8	Female	24	Bangalore	Student	2	0	5
	26	Male	31	Srinagar	Student	3	0	7.1
	30	Female	28	Varanasi	Student	3	0	5.9
	32	Female	25	Jaipur	Student	4	0	8.1

And also a value at the very end labelled "depression" which is just a 0 or 1 depending on whether they have depression or not (0 is yes 1 is no)

f Depression
1
0
0
1
0
0
0

Data Processing

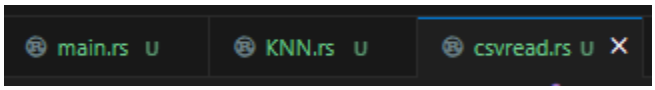
Since the file was a csv, I loaded it into rust using a csv file loader similarly to most of our previous homeworks/projects (in the csvread module). The only thing somewhat different is that I put the values into a struct “Student” defined earlier in the same module.

```rust
pub fn csvread(path: &str) -> Result<Vec<Student>, Box<dyn Error>> {
    let mut rdr = ReaderBuilder::new()
    .delimiter(b',')
    .has_headers(true)
    .from_path(path);

    let mut data = Vec::new();
    for result in rdr?.records(){
        let r = result.unwrap();

        let student = Student{
            id: r[0].parse().unwrap_or(0),
            gender: parse_gender(&r[1]), //not using this
            age: r[2].parse().unwrap_or(0.0),
            city: r[3].to_string(), //not using
            profession: r[4].to_string(), //or this
            academic_pressure: r[5].parse().unwrap_or(0.0),
            work_pressure: r[6].parse().unwrap_or(0.0),
            cgpa: r[7].parse().unwrap_or(0.0),
            study_satisfaction: r[8].parse().unwrap_or(0.0),
            job_satisfaction: r[9].parse().unwrap_or(0.0),
            sleep_duration: parse_sleep_duration(&r[10]),
            dietary_habits: parse_dietary_habits(&r[11]),
            degree: r[12].to_string(), //or this
            suicidal_thoughts: parse_suicidethoughts(&r[13]),
            workstudy_hours: r[14].parse().unwrap_or(0.0),
            financial_stress: r[15].parse().unwrap_or(0.0),
            family_history: parse_fam_history(&r[16]),
            depression: r[17].parse().unwrap_or(0.0),
        };
        data.push(student);

    }
    Ok(data)


}
```

The data was largely uniform and formatted the same, so I didn’t have to do much cleaning. The only thing I had to parse differently was the values that had something like a “yes” or “no” answer, which I changed to a value (for yes/no it was 1.0, 0.0, for more complicated stuff I assigned a float value as I saw fit). The functions for parsing are earlier in the same module.

Code Structure

The code is split into 3 modules (not including lib or the tests)



Csvread is what was in the previous section, just reading out the csv data in a vector of Student structs

The student struct was made as follows:

```rust
pub struct Student {
    pub id: i32,
    pub gender: f32,
    pub age: f32,
    pub city: String,
    pub profession: String,
    pub academic_pressure: f32,
    pub work_pressure: f32, //I think all the values for this are 0? b
    pub cgpa: f32,
    pub study_satisfaction: f32,
    pub job_satisfaction: f32,
    pub sleep_duration: f32, //just changing this to a f32 value inste
    pub dietary_habits: f32, //changing this too
    pub degree: String,
    pub suicidal_thoughts: f32,
    pub workstudy_hours: f32,
    pub financial_stress: f32,
    pub family_history: f32,
    pub depression: f32,
}
```

I ended up disregarding anything labelled as a string because I didn't know how to properly encode them into a value (ex. How to turn their city into a representative value that would work with the distance function)

KNN is where the functions for the actual KNN function are provided - this takes the two train and test datasets from splitting the full dataset (both of format Vec<Student>) and finds the closest points to each point in the test set. It takes the nearest K points and uses those to assign it a value of either 0 or 1 (depression or no depression).

I made a helper function targets_features that just splits the vector of Student structs into a vector of vector of values for each label as shown below:

```rust
pub fn targets_features(data: Vec<Student>) -> Result<(Vec<Vec<f32>>, Vec<u8>), Box<dyn Error>> {
    let mut features = vec![];
    let mut targets = vec![];

    for value in data {
        features.push(vec![
            value.gender,
            value.age,
            value.academic_pressure,
            value.work_pressure,
            value.cgpa,
            value.study_satisfaction,
            value.job_satisfaction,
            value.sleep_duration,
            value.dietary_habits,
            value.suicidal_thoughts,
            value.workstudy_hours,
            value.financial_stress,
            value.family_history,
        ]);


        targets.push(value.depression as u8);
    }

    Ok((features, targets))
}
```

The Knn function works as follows:
It first splits the given data into features and targets using the targets_features function

```rust
pub fn knn(traindata: Vec<Student>, testdata: Vec<Student>, k: usize) -> Vec<u8> {
    let (trainfeatures, traintargets) = targets_features(traindata).unwrap();
    let (testfeatures,testtargets) = targets_features(testdata).unwrap();
```

It then begins iterating over every testfeature in order to find the relevant distances.

```rust
    testfeatures.iter().map(|test_point| { //iterate over every testfeature
        let mut distances: Vec<(f32, u8)> = trainfeatures.iter()
```

It does this by zipping together the training dataset features and targets and then finding the distance between the given test points and the points in the features of the training dataset function using the distance function defined in this module (i left it out because it seems fairly trivial, although the implementation was goofy because I was using an iterator). It collects it into a vector of tuples holding a total distance value and a label (either 1 or 0, no depression or has depression).

```
.zip(&traintargets) //zips together trainfeatures and traintargets into one iterator
.map(|(train_point, &label)| (distance(test_point, train_point), label)) //map the d
.collect();
```

It then sorts that vector to find the closest values

```
distances.sort_by(|a, b| a.0.partial_cmp(&b.0).unwrap()); //sort by distance
```

It then counts the first K values (defined by whatever value you give it, for my example I used 101 because this is a fairly large dataset), incrementing count_0 by 1 if the label for that point is 0, or incrementing count_1 by 1 if the label for that point is 1.

```
let mut count_0 = 0; //number of zeroes (depression)
let mut count_1 = 0; //number of ones (no depression)
for &(burger, label) in distances.iter().take(k) { //iterates over first k values (k neighbors) in sorted distances
    if label == 1 { count_1 += 1; } else { count_0 += 1; } //checks if 0 or 1 (depression value) and shoves into count
}
```

It then predicts the value of the testpoint using those counts and collects it into the final vector

```
    if count_1 > count_0 { 1
    } else { 0
    } //assigns either 1 or 0 depending on w
}).collect() //shoves every prediction into t
```

There are also commented descriptions next to most of this code, I just had to cut a lot of it out in the screenshots so it was readable.

In the final main function, it reads out the csv file, splits it into 90-10 training and testing set, calls knn, finds the number of correct guesses, and returns the accuracy.

It first starts by just reading the file

```
fn main() {
    let mut data = csvread::csvread("student_depression_dataset.csv").expect("error");
```

It then uses shuffle to randomly reorganize the vector of Student structs

```
    data.shuffle(&mut thread_rng()); //mixing the Vec<Student> randomly
```

It splits the shuffled data 90-10

```
let split_index = (data.len() as f32 * 0.9).round() as usize; //finding index to split the th

let (train_data, test_data) = data.split_at(split_index); //splitting the mixed data
```

It then uses the knn function to find the predicted values of every point (I used k=101 because this was a fairly large dataset, around 27000 students)

```
let predicted = KNN::knn(train_data.to_vec(), test_data.to_vec(), 101);
```

I then split the test dataset into targets and features (just so i could compare the predicted values with the actual target values in the test set), and found the number of correct predictions by zipping the predicted and actual target values together before comparing them and counting them.

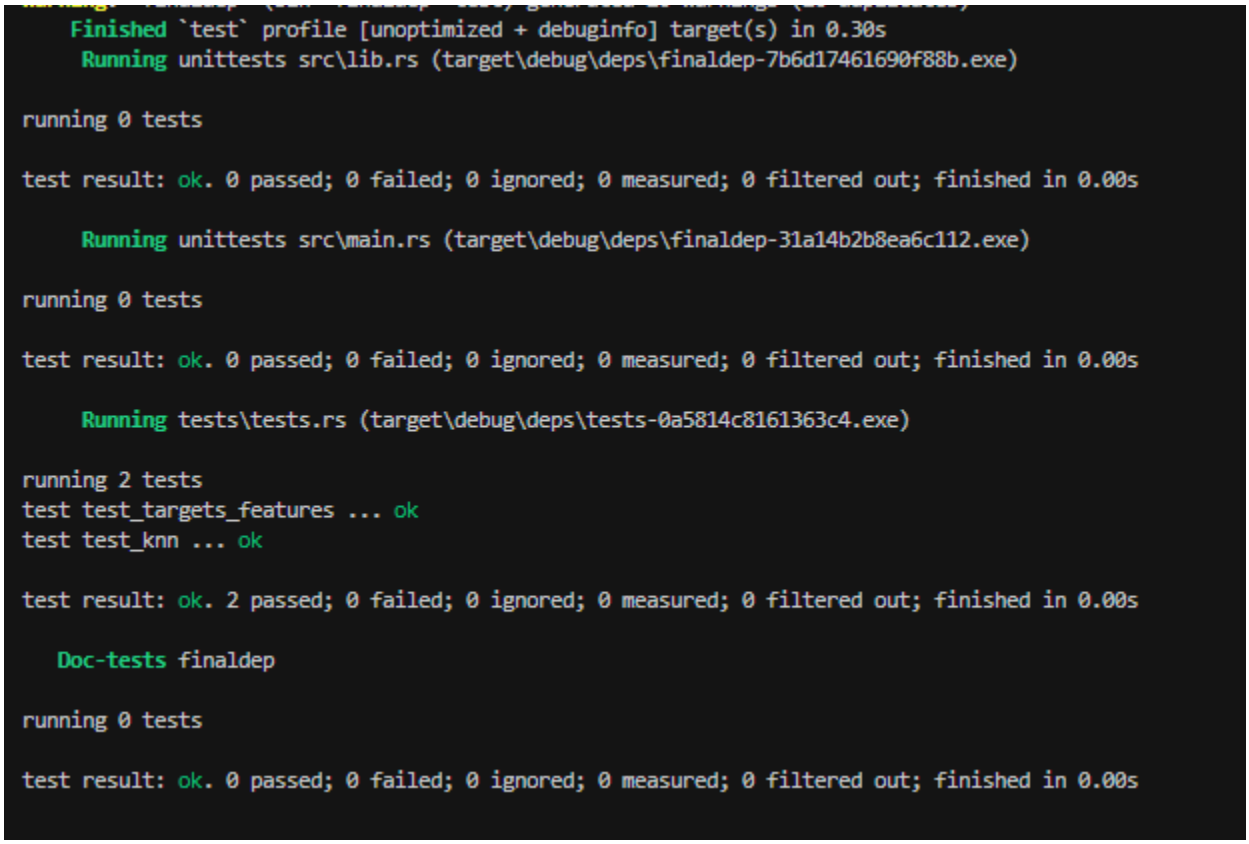
```
let correct = predicted.iter().zip(test_targets.iter()).filter(|(a, b)| a == b).count();
```

I then just took this number correct over the total number of students in the test set to get a final accuracy

```
let accuracy = correct as f32 / test_targets.len() as f32; //finding fraction correct
println!("Accuracy: {:?}", accuracy * 100.0); //acc
```

Tests

This is the output from running cargo test:



The first test is as follows:

```rust
#[test]
fn test_targets_features() { //testing whether targets_features works as it shoul
    let mock_data = vec![
        Student {
        id: 0,
            age: 0.0,
            gender: 0.0,
            city: "burger".to_string(),
            profession: "burger".to_string(),
            degree: "burger".to_string(),
            academic_pressure: 0.0,
            work_pressure: 0.0,
            cgpa: 0.0,
            study_satisfaction: 0.0,
            job_satisfaction: 0.0,
            sleep_duration: 0.0,
            dietary_habits: 0.0,
            suicidal_thoughts: 0.0,
            workstudy_hours: 0.0,
            financial_stress: 0.0,
            family_history: 0.0,
            depression: 0.0,
        },
        Student {
            id: 0,
            age: 0.0,
            gender: 0.0,
            city: "burger".to_string(),
            profession: "burger".to_string(),
            degree: "burger".to_string(),
            academic_pressure: 0.0,
            work_pressure: 0.0,
            cgpa: 0.0,
            study_satisfaction: 0.0,
            job_satisfaction: 0.0,
            sleep_duration: 0.0,
            dietary_habits: 0.0,
            suicidal_thoughts: 0.0,
            workstudy_hours: 0.0,
            financial_stress: 0.0,
            family_history: 0.0,
            depression: 1.0,
        },
    ];

    let (features, labels) = targets_features(mock_data).unwrap();
    assert_eq!(features.len(), 2); //there should be 2 vectors in the feature vec
    assert_eq!(labels, vec![0, 1]); //the two labels for depression or no depress
}
```

I made 2 simple student structs to test whether targets_features worked as it should, since this is mainly what pulls the data from the Vec<Student> into vectors that can actually be used.

The second test is as follows:

```rust
fn test_knn() { //testing whether the knn prediction
    let train_data = vec![
        Student {
            id: 0,
            age: 20.0,
            gender: 0.0,
            city: "burger".to_string(),
            profession: "burger".to_string(),
            degree: "burger".to_string(),
            academic_pressure: 0.0,
            work_pressure: 0.0,
            cgpa: 0.0,
            study_satisfaction: 0.0,
            job_satisfaction: 0.0,
            sleep_duration: 0.0,
            dietary_habits: 0.0,
            suicidal_thoughts: 0.0,
            workstudy_hours: 0.0,
            financial_stress: 0.0,
            family_history: 0.0,
            depression: 0.0,
        },
        Student {
            id: 100,
            age: 22.0,
            gender: 0.0,
            city: "burger".to_string(),
            profession: "burger".to_string(),
            degree: "burger".to_string(),
            academic_pressure: 1.0,
            work_pressure: 1.0,
            cgpa: 1.0,
            study_satisfaction: 1.0,
            job_satisfaction: 1.0,
            sleep_duration: 1.0,
            dietary_habits: 1.0,
            suicidal_thoughts: 0.0,
            workstudy_hours: 1.0,
            financial_stress: 1.0,
            family_history: 1.0,
            depression: 1.0,
        },
    ];
```

```rust
    let test_data = vec![
        Student {
            id: 101,
            age: 22.0,
            gender: 0.0,
            city: "burger".to_string(),
            profession: "burger".to_string(),
            degree: "burger".to_string(),
            academic_pressure: 2.0,
            work_pressure: 2.0,
            cgpa: 2.0,
            study_satisfaction: 2.0,
            job_satisfaction: 2.0,
            sleep_duration: 2.0,
            dietary_habits: 2.0,
            suicidal_thoughts: 0.0,
            workstudy_hours: 1.0,
            financial_stress: 1.0,
            family_history: 1.0,
            depression: 1.0,
        },
    ];

    let predictions = knn(train_data, test_data.clone(), 1);
    assert_eq!(predictions.len(), 1);
    let predicted_label = predictions[0] as u8;
    assert!(predicted_label == 0 || predicted_label == 1); //t
}
```

I made a vector with 2 student structs, both with different depression values and a lot of simple values, to use as a training data set. I then made test data set with just 1 struct, with values a lot closer to the second struct in the training data. I then checked to make sure that knn assigns the testdata features a depression label of 1.0 (since that's the label of the struct it's closer to).

Outputs

The final output of all of this is the following:

```
warning: `finaldep` (bin "finaldep") generated 26 warnings (run `cargo fix --bin "finaldep"` to apply 21 suggestions)
    Finished `release` profile [optimized] target(s) in 0.04s
     Running `target\release\finaldep.exe`
Accuracy: 80.03584
```

The knn classifier has an accuracy of ~80.036%. In the context of this problem, it takes the provided information (gender, age, gpa, work hours, etc.) and correctly guesses whether or not that person has depression ~80.036% of the time.
It is important to note that the values are NOT normalized. Most of the differences between respective values were relatively small, so I didn't think it was necessary. It's possible if they were normalized the accuracy would go up.

Usage Instructions

Just use cargo run –release. It shouldn’t take super long to run. I tried using just regular cargo run as well but that took incredibly long and I didn’t wait for it to finish.
The directory levels should just be kept the same as it is in the github project. Src for the 3 modules and lib, tests folder and csv outside of said src folder.