Written Report

A. Project Overview

Goal: What question are you answering?
 Are there distinct groups of countries with similar happiness-related characteristics over time (2013–2023), and how stable are these groups year to year?

• Dataset: Source, size (link if too large for GitHub).

World Happiness Reports 2013-2023

(Use datasets from 2015 to 2023, exclude 2022)

https://www.kaggle.com/datasets/joebeachcapital/world-happiness-report-2013-2023/datasets/joebeachcapital/world-happiness-report-2013-2013/datasets/joebeachcapital/world-happiness-report-2013-2013/datasets/joebeachcapital/world-happiness-report-201

zip(compressed): 93.2KB Decompressed: 250 KB

B. Data Processing

How you loaded it into Rust.

CSV files are read using csv::Reader crate and each year's file is loaded into a Vec<CountryData>. All data is stored in a HashMap<u32, Vec<CountryData>> mapping year to country data.

• Any cleaning or transformations applied.

I make country names consistent throughout all the datasets by writing a canonical_county_name(). I also filter out the invalid rows by writing a filter_valid_data() method. Moreover, the normalize_features() method can help me to set the value between 0.0 and 1.0 to prevent data inconsistency. Also, I sort countries in alphabetical order in main() to make sure I have stable clustering output, since k-means use fixed seed and depend on input order when selecting initial centers.

C. Code Structure

1. Modules

- Purpose of each and rationale for organization.
 - data_loader.rs: This module reads and loads World Happiness data from multiple years into organized records and fix country name differences.
 This module can help to keep the input format consistent.

- preprocessing.rs: This module removes bad data and adjusts happiness-related values so countries can be compared fairly
- clustering.rs: This module uses a fixed k-means method to group countries and check how tight the groups are.
- main.rs: This module runs the whole process and tracks how countries move between clusters over the years.
- lib.rs: This module acts as the main hub that organizes and exposes the project's core modules. By adding this module, can use cargo test to run tests from outside main.rs.

2. Key Functions & Types (Structs, Enums, Traits, etc)

For each non-trivial item, restate:

- pub struct CountryData

1. What it does:

This struct holds the datasets features that are going to be evaluated and handles inconsistent feature names across different years.

2. Inputs and outputs:

No input or output, since it's a struct.

3. Core logic and key components

This struct holds 6 selected happiness-related features.

It uses serde to automatically map CSC column names to the correct field even if the names do not consist across different year's datasets.

When reading a CSV row, serde fills in the struct fields using the column names that matched.

- fn compute_seed(year: u32, k: usize) -> u64

1. What it does:

This function generates a consistent random seed based on the year and k.

2. Inputs and outputs:

Input: year-year of the data, k-number of clusters

Output: A seed value for randomness as u64

3. Core logic and key components:

The function combines year and k into tuple.

It hashes the tuple using a default hasher.

Then, it returns the result number as the seed.

pub fn canonical_country_name(name: &str) -> String

1.What it does

This function makes sure the countries names are written in a standard way. This can fix the inconsistency of the country names throughout different year's datasets.

2.Inputs and outputs

Input: name, which is a country name as &tr

Output: A fixed version of the country name as String

3. Core logic and key components:

First converts the name to lowercase and trims spaces.

Use a match to check if the name is a form that is known.

Return a standard name if it matches or give back to the original name if it doesn't match.

pub fn load_data(folder_path: &str) -> HashMap<u32, Vec<CountryData>> 1.What it does

This function helps to read CSV file from 2015 to 2023(except 2022 because 2022 misses too many data),

cleans country names, and organizes the data by year.

2.Inputs and outputs

Input: folder_path, which is a string path to the folder that holds the CSV files Output: A HashMap where each yea maps to a list of CountryData

3. Core logic and key components

This function iterates through each year from 2015 to 2023(skip 2022).

It builds the file path for that year's CSV and tries to read the file using CSV reader.

If successfully read, the function turns each row into a CountryData object, standardizes the country names, and saves the list of records under that year in the map.

If there are some errors, the function will print the warning and skip it.

- pub fn filter_valid_data(data: &Vec<CountryData>) -> Vec<CountryData>

1.What it does

This function removes any rows that have missing or invalid numbers.

2.Inputs and outputs

Input: A reference to a list of CountryData

Output: A new list with only valid entries

3. Core logic and key components

This function keeps only the countries where all 5 feature values are real numbers, excluding those with NaN or infinity.

Then, the function returns a clean list with those valid entries.

pub fn normalize_features(data: &mut [CountryData])

1.What it does

This function helps to scale each feature so values can be set between 0.0 to 1.0.

fixing the inconsistency of the value throughout the different year's dataset.

2. Inputs and outputs

Input: A mutable slice of CountryData

Output: None (the function directly changes the input data)

3. Core logic and key components

For each feature, the function finds the smallest and largest values and rescale every value using the formula normalized=(value-min)/(max-min).

The function skips normalization if all values are the same in order to avoid dividing by zero.

- pub fn kmeans

1.What it does

This function groups counties into k clusters using the -means algorithm.

2.Inputs and outputs

Inputs: data-lists of CountryData, k-number of clusters,

iterations-how many times to update clusters, seed-random seed for reproducibility

Output: A list of cluster labels, which country belongs to which group;

A list of cluster centers(average values of each cluster)

3. Core logic and key components

The function randomly picks k countries as starting cluster centers.

Then, it repeats 2 steps iterations times:

Step1-Assign step: each country is assigned to the nearest center(based on 5 feature distance)

Step2-Update step: Each center is moved to the average of the countries in its group.

Lastly, the function returns the final cluster labels and the center positions.

- fn distance(a: &CountryData, b: &CountryData) -> f64

1.What it does

This function calculates how different 2 countries are based on the selected 4 features

2.Inputs and outputs

Input: 2 CountryData entries(a and b)

Output: A f64 number representing the squared distance between them

3. Core logic and key components

For each feature, the function subtract one country's value from the other and square it.

The function then adds all 5 squared differences together.

pub fn calculate_inertia(data: &Vec<CountryData>,centers: &Vec<CountryData>,labels: &Vec<usize>,) -> f64

1.What it does

This function calculates how close each country is to its assigned cluster center 2. Inputs and outputs

Input: data-list of country entries, centers-list of cluster centers, labels-which cluster each country belongs to

Output: The total squared distance as f64

3. Core logic and key components

For each country, the function finds its cluster center and calculates distance to that center.

The function then adds up all the distances and returns the total.

3. Main Workflow

- At a glance, how modules/functions interact to produce your results.
 - Load and read each year's CSV file
 - 2. Filter out invalid value and make the data consistent
 - 3. Try clustering with k=2 to 6 using deterministic seed
 - 4. Print inertia for each k (the first part of the output)
 - 5. Run final clustering with k=4 and track country label transition every year (the second part of the output)

D. Tests

cargo test output (paste logs or provide screenshots).

```
(base) wangxinyan@crc-dot1x-nat-10-239-163-42 DS210_final_project % cargo test
   Compiling ds210_final_project v0.1.0 (/Users/wangxinyan/Desktop/BU/DS210/ds210_final_project)
Finished `test` profile [unoptimized + debuginfo] target(s) in 0.63s
     Running unittests src/lib.rs (target/debug/deps/ds210_final_project-9d563344c40badcb)
running 0 tests
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
     Running unittests src/main.rs (target/debug/deps/ds210_final_project-86c1dc6f019d11d9)
running 0 tests
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
     Running tests/integration_tests.rs (target/debug/deps/integration_tests-ed8aaeee69a370f7)
running 5 tests
test test_inertia_is_positive ... ok
test test_normalize_between_0_and_1 ... ok
test test_remove_invalid_rows ... ok
test test_kmeans_returns_right_size ... ok
test test_country_name_fixing ... ok
test result: ok. 5 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
   Doc-tests ds210_final_project
running 0 tests
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

For each test: what it checks and why it matters.

- Test_country_name_fixing:
 - What it checks:

It make sure country names such as "United States of America" are changed to just "united states"

Why it matters:

Sometimes different files use different names for the same country. This test checks that we fix those names so we can match countries correctly across different year's dataset.

- Test_remove_invalid_rows:
 - What it checks:

It checks if rows with bad or missing numbers are removed from the dataset.

Why it matters:

Bad data can mess up the calculations. This test makes sure I only keep clean and usable data.

- Test_normalize_between_0_and_1:
 - What it checks:

It tests that after we scale the data, all the feature values are set between 0.0 and 1.0.

- Why it matters:

We need to normalize the data so that no feature unfairly affects the clustering because of a bigger scale.

- Test_kmeans_returns_right_size:
 - What it checks:

It checks if the kmeans function gives me the right number of cluster labels and cluster centers.

Why it matters:

If I ask for 2 clusters, I should get exactly 2 centers and every country should be assigned a label.

- Test_inertia_is_positive:
 - What it checks:

It confirms that the total distance between countries and their cluster centers(inertia) is never negative.

- Why it matters:

Distances should never be negative. If this fails, something is wrong with the distance calculation.

E. Results

• All program outputs (screenshots or pasted).

Finding best k for year 2015 k = 2: Inertia = 22.83

- k = 3: Inertia = 17.24
- k = 4: Inertia = 14.37
- k = 5: Inertia = 13.21
- k = 6: Inertia = 12.31

Finding best k for year 2016

- k = 2: Inertia = 25.19
- k = 3: Inertia = 20.78
- k = 4: Inertia = 16.20
- k = 5: Inertia = 14.32
- k = 6: Inertia = 13.94

Finding best k for year 2017

- k = 2: Inertia = 23.24
- k = 3: Inertia = 16.45
- k = 4: Inertia = 16.00
- k = 5: Inertia = 12.03
- k = 6: Inertia = 11.90

Finding best k for year 2018

- k = 2: Inertia = 22.81
- k = 3: Inertia = 16.05
- k = 4: Inertia = 13.60
- k = 5: Inertia = 12.21
- k = 6: Inertia = 11.08

Finding best k for year 2019

- k = 2: Inertia = 22.26
- k = 3: Inertia = 15.44
- k = 4: Inertia = 12.63
- k = 5: Inertia = 11.00
- k = 6: Inertia = 10.61

Finding best k for year 2020

- k = 2: Inertia = 21.93
- k = 3: Inertia = 15.12
- k = 4: Inertia = 12.42
- k = 5: Inertia = 10.74
- k = 6: Inertia = 10.12

Finding best k for year 2021

- k = 2: Inertia = 20.84
- k = 3: Inertia = 18.78
- k = 4: Inertia = 14.97
- k = 5: Inertia = 12.25
- k = 6: Inertia = 11.29

Finding best k for year 2023

- k = 2: Inertia = 18.76
- k = 3: Inertia = 13.42
- k = 4: Inertia = 11.50

```
k = 5: Inertia = 10.58
k = 6: Inertia = 9.26
Cluster Transition Matrix (from -> to):
0 -> 0: 31 (10.95%)
0 -> 1: 49 (17.31%)
0 -> 2: 90 (31.80%)
0 -> 3: 113 (39.93%)
1 -> 0: 44 (20.28%)
1 -> 1: 87 (40.09%)
1 -> 2: 29 (13.36%)
1 -> 3: 57 (26.27%)
2 -> 0: 97 (43.89%)
2 -> 1: 46 (20.81%)
2 -> 2: 20 (9.05%)
2 -> 3: 58 (26.24%)
3 -> 0: 114 (35.51%)
3 -> 1: 10 (3.12%)
3 -> 2: 94 (29.28%)
3 -> 3: 103 (32.09%)
```

- Interpretation in project context (no need for "groundbreaking" results).
 - Essential Concept:
 - K= "Divided into how many groups"
 - Inertia
 - Lower inertia = countries in each cluster are more similar to each other (better fit)
 - Higher inertia = countries are spread out, not tightly grouped
 - Interpreting Result
 - The inertia decreases as k increases but the rate of decreasing slows down after k=4, which supports the theory that as countries are divided into more groups, they are more similar to each other in one group. Based on this result, I chose k=4 to observe the cluster transition from cluster 0 to 3.
 - There are a several things I observed from the cluster transition matrix:
 - Partial stability: Some clusters, like cluster 1 to 1 to cluster 3 to 3, show relatively high percentages(40.09% and 32.09%) of countries remaining in the same cluster over years. This shows that these countries have consistent happiness related characteristics over time.
 - Frequent movement: At the same time, there is also a significant amount of movement between clusters. For instance, transitions like 0 -> 3 or 2 -> 0 39.93

F. Usage Instructions

• How to build and run your code.

cargo build

cargo run

• Description of any command-line arguments or user interaction in the terminal.

No command-line arguments or user interaction in the terminal

Include expected runtime especially if your project takes a long time to run.

Less than 5 seconds

G. Al-Assistance Disclosure and Other Citations

- Cite any **substantive** ChatGPT/GenAl you used (e.g. screenshot or description).
 - You can skip this for common knowledge/debugging use cases
- For each cited snippet, include your own explanation to show understanding.
- You can also provide links to other sources you found useful that are not "common knowledge"

compute_seed()

This code makes sure that the random seed being used in k-means is the same every time I run it for the same year and cluster count. It combines the year and k into a unique key and uses a hashing function to turn that into a number. In this way, the clustering result is stable and can be reproduced across runs.

i want to make my k-means clustering deterministic across different years and values of k. is there a good way to generate a consistant random seed using the year and the number of clusters?

Yes, you can generate a consistent and deterministic random seed based on the year and number of clusters (k) by hashing the two together in a stable way. One good approach is to use a hash function like std::collections::hash_map::DefaultHasher in Rust, which is deterministic within the same execution if you control the seed input yourself.

Recommended Method (Rust Example):

```
rust

Use std::collections::hash_map::DefaultHasher;

use std::hash::{Hash, Hasher};

fn get_seed(year: u32, k: u32) → u64 {

let mut hasher = DefaultHasher::new();

(year, k).hash(&mut hasher); // Tuple implements Hash

hasher.finish()
}
```

This get_seed function:

- Takes year and k as input,
- · Hashes them together deterministically using a tuple,
- Returns a u64 seed you can pass to your RNG (e.g., StdRng::seed_from_u64(seed)).

Why this works well:

- **Deterministic**: Always produces the same seed for the same inputs.
- Compact: Avoids manual string formatting or concatenation.
- Hash-collision resistant enough for typical use in data science experiments.

Tracking Cluster Transitions Over Years

Al suggested that I use 2 hashmaps to compare country cluster labels from 2 years in a row. For each country that appears in both years, I check if the cluster changed, and record the change using (form, to) key in a hash map. This can help me create a transition matrix that shows how countries move between clusters over time.

i wna write a small snippet that tracks how countries move between clusters from one year to the next in my k-means clustering and i already have two HashMap<String, usize> for each year that map country name to cluster label, so how can I count transitions between cluster labels?

You can count cluster transitions between two years by comparing the two HashMap<String, usize>S—one for each year—based on country names. You'll build a matrix or another HashMap<(usize, usize), usize> where each key is a (from_cluster, to_cluster) pair and the value is the count of countries that transitioned that way.

Here's a simple Rust snippet to do that:

```
rust

use std::collections::HashMap;

fn count_cluster_transitions(
    prev_year: &HashMap<String, usize>,
    next_year: &HashMap<String, usize>,
    let mut transitions: HashMap<(usize, usize), usize> {
    let mut transitions: HashMap<(usize, usize), usize> = HashMap::new();

    for (country, &prev_cluster) in prev_year {
        if let Some(&next_cluster) = next_year.get(country) {
            let key = (prev_cluster, next_cluster);
            *transitions.entry(key).or_insert(0) += 1;
        }
    }
    transitions
}
```

Example usage:

Would you like to convert this into a 2D matrix instead of a HashMap?

O O O O O O C <

Testing in integration_tests.rs

This code makes sure every feature value is correctly normalized. It checks that each value is between 0.0 and 1.0 using Rust's range check. If any value is outside the range, the test would not pass.

In my tests, how can I check that normalized values are all between 0.0 and 1.0 in Rust?

To check that normalized values are all between 0.0 and 1.0 in your Rust tests, you can use assert! statements with comparisons. Here's a simple pattern:

Example test code:

```
rust

#[test]
fn test_normalized_values_are_in_range() {
    let mut data = load_sample_data(); // Your own function to load or create data
    normalize_features(&mut data); // Your normalization function

for d in data.iter() {
    assert!((0.0..=1.0).contains(&d.social_support), "Out of range: {}", d.social_sup
    assert!((0.0..=1.0).contains(&d.life_expectancy), "Out of range: {}", d.life_expectancy);
    assert!((0.0..=1.0).contains(&d.freedom), "Out of range: {}", d.freedom);
    assert!((0.0..=1.0).contains(&d.generosity), "Out of range: {}", d.generosity);
    assert!((0.0..=1.0).contains(&d.corruption), "Out of range: {}", d.corruption);
}
}
```

Notes:

- Use assert!((0.0..=1.0).contains(&value)) to verify each value is within bounds (inclusive).
- Add a message to assert! for easier debugging.
- You can make this generic if you want to reuse logic across fields.

Would you like a version that works with closures so you can reuse the test logic across different fields?

日 B P O B C V

- Normalize_column

I use the min-max normalization method to rescale the data to the range 0 to 1. For each feature, the function finds the smallest and largest values and rescale every value using the formula normalized=(value-min)/(max-min). The function skips normalization if all values are the same in order to avoid dividing by zero.

I am doing a project which using the data's value to calculate the inertia and cluster and kmean. however, now i met a problem that values are inconsistance across different datasets because they have different scale. is there any method to normalize it?

Yes — you're encountering a **common problem** in clustering projects: when features have different scales (like one is between 0–1 and another is in the 1000s), it can distort distance-based methods like **k-means**, which relies on **Euclidean distance**. The solution is to **normalize** or **standardize** your features.

Here are three popular normalization methods:

☑ 1. Min-Max Normalization (Rescaling to [0, 1])

Formula:

$$x_{ ext{normalized}} = rac{x - \min(x)}{\max(x) - \min(x)}$$

- Scales all values to a range between 0 and 1.
- Good for when you know the min/max bounds, and values are not too skewed.

2. Z-score Standardization (Standard Scaling)

Formula:

$$x_{ ext{standardized}} = rac{x-\mu}{\sigma}$$

- Centers data to mean = 0 and standard deviation = 1.
- Useful if your data has outliers or different distributions.
- Recommended if your features are Gaussian-like.

3. Max Abs Scaling

Formula:

$$x_{ ext{scaled}} = rac{x}{ ext{max}(|x|)}$$

Useful when data is already centered at zero, just rescales

X How to Pick?

- Use Min-Max if all features are strictly positive and you want values in [0, 1].
- Use Z-score if your features have different means and variances, or when you care about equal weighting.