Matthew Morris
Professor Kontonthanassis
DS210
4 May 2025

DS210 Final Project Writeup

Project Overview

The objective of my final project is to find clusters within my data and analyze them. I used a novel method that we had not learned in class, K Means clustering in order to achieve my goal. While initially I wanted to find deeper correlations and properly detect fraud–specitically through finding deeper "rings" of correlated accounts, I found that through the exploration of my data, there was no way to properly detect fraud, as most accounts had majority normal transactions and very few "fraudulent" transactions. Thus, it became hard to properly detect a specific manner in which fraudulent transactions were performed. The question I transitioned to that I was to is: "utilizing my dataset and K-means clustering, what buying habits in particular stand out, and how can we view the data from a fraud perspective?"

This is my dataset. It is 51,000 rows, giving insight into a variety of different transactions behaviors along these categories: Transaction_ID User_ID, Transaction_Amount, Transaction_Type, Time_of_Transaction, Device_Used, Location, Previous_Fraudulent_Transactions, Account_Age, Number_of_Transactions

Data Processing

I performed very minimal cleaning to this dataset. Through this dataset, I firstly omitted the non numerical values, as it is hard to assign values and normalize specific locations or device used. The dataset was also very high quality with essentially no need for cleaning NaN values or omitting unfilled data. I loaded the dataset via CSV, and wrote some code for a csv reader in order to manipulate the data in the form of structs.

Code Structures

1. Main Module (main.rs)
Purpose: Orchestrates the entire fraud detection process and handles result presentation in print statements, taking all the data that was generated in helper functions.

Rationale: Acts as the entry point and coordinator, separating high-level workflow from implementation details.

2. K-Means Module (kmeans.rs)
Purpose: Implements the clustering algorithm and cluster analysis.
Rationale: Encapsulates all clustering-related functionality, making it reusable and maintainable.

3. CSV Reader Module (csv_reader.rs)
Purpose: Handles data loading and feature vector conversion.
Rationale: Separates data I/O concerns from business logic, making the system more modular.

Main Module Components:

ClusterMetrics Struct
Purpose: Holds aggregated statistics across all clustersComponents:

- total_transactions: Total number of transactions analyzed
- total_fraud: Total number of fraudulent transactions
- fraud_rate: Overall fraud rate across all clusters

print_cluster_analysis Function
Purpose: Displays detailed information about a single clusterInputs:

- cluster: ClusterAnalysis data
- rank: Cluster's rank based on fraud rate
- fraud_rate: Percentage of fraudulent transactions

Outputs: Formatted console outputCore Logic:

1. Displays basic cluster statistics
1. Shows average feature values
1. Identifies common transaction characteristics
1. Determines risk level

calculate_metrics Function
Purpose: Computes overall statistics across all clustersInputs: Vector of cluster analyses with fraud ratesOutputs: ClusterMetrics structCore Logic:

1. Aggregates total transactions and fraud counts
1. Calculates overall fraud rate

K-Means Module Components:

ClusterAnalysis Struct

Purpose: Represents analysis results for a single clusterComponents:

- size: Number of transactions in cluster
- fraud_count: Number of fraudulent transactions
- unique_users: Number of unique users
- avg_features: Average values for each feature
- most_common_tx_type: Most frequent transaction type
- most_common_payment: Most frequent payment method

analyze_clusters Function

Purpose: Performs K-Means clustering and analyzes resultsInputs:

- Vector of transactions
- Number of clusters

Outputs: Vector of ClusterAnalysis objects

Core Logic:

- Converts transactions to feature vectors
- Creates dataset matrix
- Performs K-Means clustering
- Analyzes each cluster's characteristics

CSV Reader Module Components

Transaction Struct

Purpose: Represents a single transaction record

Components:

- Transaction details (amount, type, time)
- User information (ID, account age)
- Fraud indicators (fraudulent flag, previous fraud)
- Payment information (method, recent transactions)

to_feature_vector Method:

Purpose: Converts transaction data into numerical features

Inputs: None (uses self)

Outputs: Vector of f64 values

Core Logic:

- Extracts numerical features
- Handles missing values with defaults

read_transactions Function

Purpose: Loads transaction data from CSV
Inputs: File path
Outputs: Vector of Transaction objects
Core Logic:

- Opens and reads file
- Parses CSV records
- Converts to Transaction objects

Main Workflow

- Data Loading (main.rs → csv_reader.rs)
- Main module calls read_transactions
- CSV Reader loads and parses data
- Returns vector of Transaction objects
- Clustering (main.rs → kmeans.rs)
- Main module calls analyze_clusters
- K-Means module:
- Converts transactions to features
- Performs clustering
- Analyzes each cluster
- Returns cluster analyses
- Analysis & Presentation (main.rs)
- Calculates fraud rates for each cluster
- Sorts clusters by fraud rate
- Displays detailed analysis
- Shows overall metrics

Module Interactions

Main.rs adminsters the helper modules.
-> CSV_Reader.rs
-> Kmeans.rs

Tests

Output:
running 6 tests
test tests::tests::test_transaction_feature_vector_with_none_values ... ok

test tests::tests::test_transaction_feature_vector ... ok
test tests::tests::test_empty_transactions ... ok
test tests::tests::test_single_transaction ... ok
test tests::tests::test_cluster_analysis_properties ... ok
test tests::tests::test_clustering_basic ... ok

test result: ok. 6 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.01s

---

Test_transaction_feature_vector_with_none_values:
- Tests how the system handles missing data (None values)
- Sets all transaction fields to None
- Verifies that missing values are properly defaulted to 0.0

Test_transaction_feature_vector:
- Clones test_transaction (set of 3 fake transactions)
- Tests if a transaction is correctly converted into a feature vector
- Verifies all 5 features (9 minus nonnumerical values)
- Checks each feature individually

Test_empty_transactions:
- Tests how the system handles empty input
- Verifies that clustering succeeds with empty input
- Ensures empty input returns empty clusters
- Tests how function reacts to edge case

Test_single_transaction:
- Tests what happens if only one transaction happens

Test_cluster_analysis_properties:
- Tests the necessary criteria for clusters: positive size, fraud doesn't exceed cluster size, all 5 avg features are present and are valid numbers

Test_clustering_basic:
- Tests basic clustering functionality

Results:

Output:

K-Means clustering (k=10)
Fraud Detection Analysis Results:

High-Risk Clusters (Sorted by Fraud Rate):

Cluster Rank 1 (Fraud Rate: 6.3%)
Size: 508 transactions
Fraudulent: 32 (6.3%)
Unique Users: 471

Feature Analysis:
Avg Amount: $49997.80
Avg Time: 10.6
Avg Prev. Fraud: 1.98
Avg Acct Age: 62 days
Avg Recent Transactions: 7.4

Common Characteristics:
Transaction Type: Bank Transfer
Payment Method: Credit Card

Risk Level: Low Risk

Cluster Rank 2 (Fraud Rate: 5.2%)
Size: 5584 transactions
Fraudulent: 289 (5.2%)
Unique Users: 2976

Feature Analysis:
Avg Amount: $3032.61
Avg Time: 11.0
Avg Prev. Fraud: 2.00
Avg Acct Age: 61 days
Avg Recent Transactions: 7.5

Common Characteristics:
Transaction Type: Bill Payment
Payment Method: UPI

Risk Level: Low Risk

Cluster Rank 3 (Fraud Rate: 5.1%)
Size: 5247 transactions
Fraudulent: 270 (5.1%)
Unique Users: 2902

Feature Analysis:
Avg Amount: $4716.75
Avg Time: 10.7
Avg Prev. Fraud: 2.02
Avg Acct Age: 60 days
Avg Recent Transactions: 7.4

Common Characteristics:
Transaction Type: Bill Payment
Payment Method: Debit Card

Risk Level: Low Risk

Cluster Rank 4 (Fraud Rate: 5.1%)
Size: 6405 transactions
Fraudulent: 329 (5.1%)
Unique Users: 3190

Feature Analysis:
Avg Amount: $127.46
Avg Time: 11.0
Avg Prev. Fraud: 2.00
Avg Acct Age: 60 days
Avg Recent Transactions: 7.5

Common Characteristics:
Transaction Type: ATM Withdrawal
Payment Method: Debit Card

Risk Level: Low Risk

Cluster Rank 5 (Fraud Rate: 5.1%)
Size: 5447 transactions
Fraudulent: 278 (5.1%)
Unique Users: 2940

Feature Analysis:
Avg Amount: $3603.83
Avg Time: 10.9
Avg Prev. Fraud: 1.95
Avg Acct Age: 60 days
Avg Recent Transactions: 7.5

Common Characteristics:
Transaction Type: POS Payment
Payment Method: UPI

Risk Level: Low Risk

Cluster Rank 6 (Fraud Rate: 5.1%)
Size: 5568 transactions
Fraudulent: 283 (5.1%)
Unique Users: 2995

Feature Analysis:
Avg Amount: $1285.98
Avg Time: 11.0
Avg Prev. Fraud: 1.98
Avg Acct Age: 61 days
Avg Recent Transactions: 7.5

Common Characteristics:
Transaction Type: Bill Payment
Payment Method: Debit Card

Risk Level: Low Risk

Cluster Rank 7 (Fraud Rate: 4.9%)
Size: 5626 transactions
Fraudulent: 277 (4.9%)
Unique Users: 2977

Feature Analysis:
Avg Amount: $703.60
Avg Time: 11.0
Avg Prev. Fraud: 1.99
Avg Acct Age: 61 days
Avg Recent Transactions: 7.6

Common Characteristics:
Transaction Type: ATM Withdrawal
Payment Method: Credit Card

Risk Level: Low Risk

Cluster Rank 8 (Fraud Rate: 4.9%)
Size: 5574 transactions
Fraudulent: 271 (4.9%)
Unique Users: 3034

Feature Analysis:
Avg Amount: $2447.96
Avg Time: 10.7
Avg Prev. Fraud: 2.01
Avg Acct Age: 60 days
Avg Recent Transactions: 7.5

Common Characteristics:
Transaction Type: Bill Payment
Payment Method: Debit Card

Risk Level: Low Risk

Cluster Rank 9 (Fraud Rate: 4.4%)
Size: 5645 transactions
Fraudulent: 249 (4.4%)
Unique Users: 3018

Feature Analysis:
Avg Amount: $1861.10
Avg Time: 11.0
Avg Prev. Fraud: 2.00
Avg Acct Age: 59 days
Avg Recent Transactions: 7.5

Common Characteristics:
Transaction Type: Bank Transfer
Payment Method: Net Banking

Risk Level: Low Risk

Cluster Rank 10 (Fraud Rate: 4.3%)
Size: 5396 transactions
Fraudulent: 232 (4.3%)
Unique Users: 2891

Feature Analysis:
Avg Amount: $4162.41
Avg Time: 10.9
Avg Prev. Fraud: 2.00
Avg Acct Age: 60 days
Avg Recent Transactions: 7.5

Common Characteristics:
Transaction Type: Bill Payment
Payment Method: UPI

Risk Level: Low Risk
Overall Metrics:
Total Transactions: 51000
Total Fraudulent: 2510
Overall Fraud Rate: 4.92%

This code/output shows the the result for 10 k-means clusters. While a lot, I felt that this was a good amount to test the functionality and hopefully note the minute changes in the data across 51000 points. I rated the fraud risk as such: <20% = Low Risk, 20-50% = Medium Risk, >50% = High Risk. From there, I also print out the statistics from each cluster, because I want to know why the data decides to cluster. From there, I note that this cluster (see data below) contains the most fraudulent transactions.

Cluster Rank 1 (Fraud Rate: 6.3%)
Size: 508 transactions
Fraudulent: 32 (6.3%)
Unique Users: 471

Feature Analysis:
Avg Amount: $49997.80
Avg Time: 10.6
Avg Prev. Fraud: 1.98
Avg Acct Age: 62 days
Avg Recent Transactions: 7.4

It demonstrates that typically accounts are not very old, and that the avg amount taken out is ~$50000, which is a significant amount of money, indicating that fraudsters go for large sums of money typically, and often repeat (avg prev. Fraud: 1.98)

It's really interesting to note and compare the different clusters of transactions and their feature values, because they vastly differ. In retrospect, I should have utilized silhouette coefficient to truly nail down how many clusters I should have created without oversaturating the clustering.

Usage Instructions:

- Just download the code and use cargo run –release. The kmeans function will take some time in normal cargo run, but the code should not take more than a minute or so with cargo run –release. There is no user input, as this solely takes this dataset into account.
-

AI Assistance Disclosure and Other Citations:

- https://rust-ml.github.io/book/3_kmeans.html
- https://docs.rs/linfa-clustering/latest/linfa_clustering/struct.KMeans.html
- https://github.com/rust-ml/linfa

Gemini queries:

how does kmeans work in rust?

Show thinking

Okay, let's break down how K-Means clustering typically works within the Rust ecosystem, primarily focusing on the approach used in your code with the linfa crate.

In Rust, you generally don't implement complex algorithms like K-Means from scratch for typical applications. Instead, you leverage existing, well-tested libraries. The most prominent one for classical machine learning in Rust is **linfa**.

Here's the typical workflow and how it maps to the K-Means algorithm steps, using linfa:

1. **Data Preparation (Your Transaction::to_feature_vector()):**
    - **K-Means Requirement:** K-Means operates on numerical data points in a multi-dimensional space. Each data point needs to be represented as a vector of numbers.
    - **Rust Implementation:** You first need to load your raw data (e.g., from your fraud_detection.csv). Then, you must convert each record (like a Transaction) into a numerical feature vector (Vec<f64>). This is precisely what your (unseen) Transaction::to_feature_vector() method does. This step involves:
        - Selecting relevant columns.
        - Handling missing values (imputation, removal).

- **Encoding categorical features:** Converting text like "Credit Card" or "Location" into numbers (e.g., using one-hot encoding, label encoding - though label encoding is often less suitable for K-Means).
- **Scaling numerical features:** Often important for K-Means, as features with larger ranges can dominate the distance calculation (e.g., using StandardScaler or MinMaxScaler, which linfa also provides). Your current code doesn't explicitly show scaling, but it's a common step.
-

2. **Data Representation (ndarray::Array2):**
   - **K-Means Requirement:** The algorithm needs the data organized, typically as a matrix where rows are data points (samples) and columns are features.

Rust Implementation: The linfa ecosystem heavily relies on the ndarray crate. You collect your individual feature vectors (Vec<f64>) into a 2D array, usually ndarray::Array2<f64>. Your kmeans.rs code does this:

Rust
```rust
let mut data = Array2::zeros((features.len(), n_features));
for (i, feature_vec) in features.iter().enumerate() {
    for (j, &value) in feature_vec.iter().enumerate() {
        data[[i, j]] = value;
    }
}
```

   -

3. **Dataset Creation (linfa::DatasetBase):**
   - **K-Means Requirement:** The library needs the data in its expected format.

Rust Implementation: linfa uses a Dataset or DatasetBase struct to wrap the ndarray data along with optional targets or weights. For unsupervised clustering like K-Means, you typically just wrap the feature array:

Rust
```rust
use linfa::prelude::DatasetBase;
let dataset = DatasetBase::from(data);
```

   -

4. **Model Configuration (KMeans::params):**

- **K-Means Requirement:** You need to specify parameters, most importantly k (the number of clusters). Other parameters control the initialization method, stopping criteria, etc.
-

Rust Implementation (linfa-clustering): linfa uses a builder pattern. You start with KMeans::params(n_clusters) and chain methods to set options:

Rust

```
use linfa_clustering::KMeans;
let model_params = KMeans::params(n_clusters) // Set k
    .max_n_iterations(200)      // Stop after this many iterations
    .tolerance(1e-5)           // Stop if centroids move less than this
    .init_method(linfa_clustering::KMeansInit::KMeansPlusPlus); // Smart initialization
```

-
    - KMeansInit::KMeansPlusPlus is generally preferred over random initialization as it often leads to better and more consistent results.
    -

5. **Training the Model (.fit()):**
    - **K-Means Requirement:** This is where the core iterative algorithm runs:
        - Initialize k centroids (based on init_method).
        - **Repeat:**
            - **Assignment Step:** Assign each data point in the dataset to its nearest centroid (usually using Euclidean distance).
            - **Update Step:** Recalculate each centroid's position as the mean (average) of all points assigned to it.
            -
        - **Until:** Convergence criteria are met (centroids move less than tolerance, or max_n_iterations is reached).

Rust Implementation: You call the .fit() method on the configured parameters, passing the dataset. linfa handles the internal iterations. Rust's error handling (Result) is used because the fitting process could potentially fail (e.g., if k is invalid for the data).

Rust

```
use linfa::prelude::Fit;
let model = model_params.fit(&dataset)?; // The '?' handles potential errors
```

- The model variable now holds the trained K-Means model, including the final positions of the centroids.
-

6. **Using the Model (.predict(), .centroids()):**
   - **K-Means Requirement:** Once trained, you typically want to know which cluster each point belongs to, and where the final cluster centers are.
   - **Rust Implementation:**

Cluster Assignments: Use .predict() on the trained model to get an ndarray::Array1<usize> where each element is the cluster index assigned to the corresponding data point in the input dataset.

Rust
```rust
use linfa::prelude::Predict;
let predictions = model.predict(&dataset);
```

-

Centroids: Access the final centroid coordinates (often useful for understanding cluster centers, like you are doing with avg_features).

Rust
```rust
let centroids = model.centroids(); // Returns an ndarray::Array2<f64>
```

- *(Note: Your code recalculates the centroids manually as avg_features, which is also valid and sometimes done for precision based on final assignments, though using model.centroids() is direct).*
-

Rust-Specific Aspects:

- **Type Safety:** Rust's static typing helps ensure that you're passing correctly typed data (e.g., Array2<f64>) to the algorithm.
-
- **Error Handling:** The use of Result<T, E> (e.g., in .fit()) makes error handling explicit and robust.
-
- **Performance:** Rust, combined with ndarray (which can use optimized backends like BLAS), allows for high-performance K-Means implementations suitable for large datasets.
- **Ecosystem:** You rely on crates (linfa, ndarray, linfa-clustering) managed by Cargo, Rust's package manager.

•

In essence, K-Means in Rust works by leveraging the linfa library, which provides a high-level API (params, fit, predict) that internally executes the standard K-Means iterative steps (assignment, update) on data typically stored in ndarray arrays.

Sources and related content

The impact of neglecting feature scaling in k-means clustering - PMC

pmc.ncbi.nlm.nih.gov

What is k-means clustering? - IBM

www.ibm.com

KMeansInit in linfa_clustering - Rust - Docs.rs

docs.rs

KMeans in linfa_clustering - Rust - Docs.rs

docs.rs

smartcore::cluster::kmeans - Rust - Docs.rs

docs.rs

how do i configure a csv and features for this purpose?

Show thinking

Okay, configuring your CSV data and defining features for K-Means in Rust (using linfa) involves two main parts: preparing the CSV file itself and then implementing the logic in your Rust code to process that CSV data into the numerical feature vectors K-Means requires.

1. Configuring the CSV File

- **Standard Format:** Use a standard CSV (Comma Separated Values) format. Ensure consistent use of delimiters (usually commas) and quoting (usually double quotes " around fields containing commas or newlines).

Header Row: Crucial: Have a clear header row as the first line. This names each column (e.g., Transaction_ID, User_ID, Transaction_Amount, Location, Fraudulent). This makes parsing and feature selection much easier in your Rust code. Your example fraud_detection.csv already does this well.

Code snippet

```
Transaction_ID,User_ID,Transaction_Amount,Transaction_Type,Time_of_Transaction,Device_Used,Location,Previous_Fraudulent_Transactions,Account_Age,Number_of_Transactions_Last_24H,Payment_Method,Fraudulent
T1,4174,1292.76,ATM Withdrawal,16.0,Tablet,"San Francisco",0,119,13,Debit Card,0
T2,1234,50.00,Online Purchase,8.5,Mobile,"New York",1,250,2,Credit Card,0
...
```

-
- **One Row Per Sample:** Each row after the header should represent a single data point (in your case, one transaction).
-
- **Include All Relevant Data:** Include all columns that might be useful, *even if not directly used as features*. For instance, keep Transaction_ID and User_ID in the CSV, as you might need them later to identify which transactions fall into which clusters. Keep the Fraudulent column to analyze the purity of your clusters after they are formed.

2. Defining Features (in Rust Code)

This is the most critical part and happens within your Rust code, specifically in the function/method that converts a row from the CSV (likely parsed into a Transaction struct) into a Vec<f64> for linfa (the conceptual Transaction::to_feature_vector() method).

K-Means Requirements to Keep in Mind:

- **Numerical Only:** K-Means *only* understands numbers.
- **Distance-Based:** It uses distances (like Euclidean). This means feature scaling is very important.
-

Steps to Configure Features in Your Rust Code:

1. **Column Selection:**
    - **Identify Candidates:** Choose columns from your Transaction struct that you believe are relevant for grouping similar transactions.
        - *Good Candidates:* Transaction_Amount, Time_of_Transaction, Previous_Fraudulent_Transactions, Account_Age, Number_of_Transactions_Last_24H, Transaction_Type, Device_Used, Location, Payment_Method.
    - **Exclude Identifiers:** Do *not* include Transaction_ID or User_ID as features. They don't represent transaction behavior for grouping.
    - **Exclude Target (for Clustering):** Do *not* include the Fraudulent column *as a feature* in the vector used for K-Means clustering itself. You are trying to find natural groups *without* knowing the fraud label beforehand.
2. **Handle Missing Values:**
    - Before converting to numbers, decide how to handle rows with missing data in your selected columns (e.g., empty strings, NULL, NaN).
    - **Strategies:** Remove the row, or impute the value (replace with mean, median, or mode of the column). Imputation is often preferred if you have a lot of data.
3. **Convert to Numerical & Encode:**
    - **Numerical Columns:** (Transaction_Amount, Time_of_Transaction, Previous_Fraudulent_Transactions, Account_Age, Number_of_Transactions_Last_24H)
        - Parse them into f64.
        - *Consider Transformations:* For skewed data like Amount, a log transform (log(amount + 1)) might help. For Time_of_Transaction, consider cyclical encoding (converting time t out of 24 hours into two features: sin(2 * PI * t / 24) and cos(2 * PI * t / 24)) if the cyclical nature is important.
    - **Categorical Columns:** (Transaction_Type, Device_Used, Location, Payment_Method)
        - These *must* be converted to numbers.

        - **One-Hot Encoding (Recommended for K-Means):** This is usually the best approach. Create a new binary (0 or 1) feature for each possible category within the column.
            - Example Payment_Method: If you have "Credit Card", "Debit Card", "PayPal". This becomes 3 features: is_CreditCard (0 or 1), is_DebitCard (0 or 1), is_PayPal (0 or 1).
            - *Pros:* Doesn't impose artificial order. Works well with distance measures.
            - *Cons:* Increases the number of features (dimensionality). Can be complex for features with many unique categories (like Location if it's very granular).
        - *Other Encodings (Use with Caution):* Label Encoding (mapping categories to 0, 1, 2...) is generally **bad** for K-Means because it implies an order and distance ("Debit Card" (1) is closer to "Credit Card" (0) than "PayPal" (2)) which is usually meaningless.
4. **Feature Scaling (Very Important!):**
    - Because K-Means uses distance, features with larger values/ranges (like Transaction_Amount) will dominate the distance calculation compared to features with smaller ranges (like Number_of_Transactions_Last_24H).
    - **You MUST scale your numerical features** to have a similar range or distribution.
    - **Common Techniques:**
        - **StandardScaler (Z-score):** Rescales features to have a mean of 0 and a standard deviation of 1. Good general choice. (linfa-preprocessing::StandardScaler)
        - **MinMaxScaler:** Rescales features to a specific range, typically [0, 1]. (linfa-preprocessing::MinMaxScaler)
        -
    - **When to Scale:** Apply scaling *after* handling missing values and *after* any transformations (like log transform) or encoding (though you typically scale the *original* numerical features and the *results* of one-hot encoding separately or together depending on strategy).
5. **Construct the Final Vector:**
    - Combine all the processed, encoded, and scaled numerical values into a single Vec<f64>.
    - **The order matters!** Ensure you add the features to the vector in the *same order* for every single transaction. This order defines what "Feature 0", "Feature 1", etc., mean.

Conceptual Example (Transaction::to_feature_vector()):

Rust

```
// Inside your Transaction struct's impl or a helper function

// Assume 'transaction' is your parsed struct instance
// Assume you have already handled missing values

// 1. Select & Process Numerical Features
let mut features = vec![];
let log_amount = transaction.transaction_amount.ln_or(0.0); // Example: Log transform amount
let time_sin = (2.0 * std::f64::consts::PI * transaction.time_of_transaction / 24.0).sin(); // Cyclical
time
let time_cos = (2.0 * std::f64::consts::PI * transaction.time_of_transaction / 24.0).cos(); // Cyclical
time
let prev_fraud = transaction.previous_fraudulent_transactions as f64;
let age = transaction.account_age as f64;
let tx_count = transaction.number_of_transactions_last_24h as f64;

// Add numerical features (BEFORE SCALING in this conceptual example)
features.push(log_amount);
features.push(time_sin);
features.push(time_cos);
features.push(prev_fraud);
features.push(age);
features.push(tx_count);

// 2. Encode Categorical Features (Example: One-Hot Encoding Payment Method)
let is_credit = if transaction.payment_method == "Credit Card" { 1.0 } else { 0.0 };
let is_debit = if transaction.payment_method == "Debit Card" { 1.0 } else { 0.0 };
// ... add flags for other payment methods ...
features.push(is_credit);
features.push(is_debit);
// ... push other flags ...

// ... Encode other categoricals (Type, Device, Location) similarly ...
```

// 3. Feature Scaling (Conceptual - Usually done on the whole dataset matrix)
// let scaler = StandardScaler::fit(&all_features_matrix).unwrap();
// let scaled_features_matrix = scaler.transform(&all_features_matrix);
// You would typically perform scaling on the entire dataset matrix *after* constructing
// all feature vectors, using tools from linfa_preprocessing.
// The `features` Vec above would be one row in that matrix BEFORE scaling.

// Return the unscaled (or scaled, depending on where you implement scaling) vector
// The order here defines Feature 0, Feature 1, etc.
features // This vector needs scaling applied before passing to KMeans::fit

In summary: Prepare a clean CSV with headers. The main work is in your Rust code: select columns, handle missing data, convert *everything* to numerical values (using encoding like One-Hot for categoricals), apply transformations if needed, and critically, **scale** the resulting features before feeding them into linfa's K-Means.

Sources and related content

The Ultimate Guide to CSV Formatting - Knack

www.knack.com

kmeans - k-means clustering - MATLAB - MathWorks

www.mathworks.com

When to use MinMaxScaler vs StandardScaler - Kaggle

www.kaggle.com

M

Generate Audio Overview