

MinSeok Cho

Prof. Chator

CDS DS210 B1

15 Dec 2024

Final Project Citation

.flush(): [Rust Language](#)

Q: I think the ML doesn't involve all the learnings from the column. How should I code in Rust?

Ans:

If your machine learning model doesn't utilize all the data from the columns, you need to ensure your Rust implementation processes the input correctly and includes all the relevant features in the dataset for learning. Here's a structured approach:

Steps to Code This in Rust

1. **Read All Columns**:

Ensure your `FileStructure` struct captures all the relevant columns of your CSV file. If some columns are ignored during deserialization, you might be missing important data for training.

```
``rust
#[derive(Debug, Deserialize)]
struct FileStructure {
    feature1: f64,
    feature2: f64,
    feature3: f64,
    // Include all necessary columns
    label: String, // Assuming the label column exists
}
``
```

2. **Vectorize the Data**:

Convert the deserialized records into a matrix-like structure suitable for training. Libraries like `ndarray` or `tch` (Torch for Rust) can be used to manipulate numerical data.

```
``rust
use ndarray::Array2;

fn process_data(records: Vec<FileStructure>) -> (Array2<f64>, Vec<String>) {
```

```

let num_records = records.len();
let num_features = 3; // Number of features (adjust as necessary)
let mut features = Array2::zeros((num_records, num_features));
let mut labels = Vec::new();

for (i, record) in records.iter().enumerate() {
    features[[i, 0]] = record.feature1;
    features[[i, 1]] = record.feature2;
    features[[i, 2]] = record.feature3;
    labels.push(record.label.clone());
}

(features, labels)
}
'''

```

3. ****Prepare the Model**:**

Choose or implement a machine learning model. For example, if you're using `linfa` (a Rust crate for ML), you can apply logistic regression, SVM, or other algorithms. Here's an example with `linfa`.

```

'''rust
use linfa::prelude::*;
use linfa_logistic::LogisticRegression;

fn train_model(features: Array2<f64>, labels: Vec<String>) {
    // Convert labels to numeric representation
    let labels: Vec<f64> = labels
        .iter()
        .map(|label| if label == "fraud" { 1.0 } else { 0.0 }) // Adjust based on your dataset
        .collect();

    // Convert labels to an array
    let labels = Array2::from_shape_vec((labels.len(), 1), labels).unwrap();

    // Fit the logistic regression model
    let model = LogisticRegression::default()
        .fit(&features, &labels)
        .expect("Failed to train the model");

    println!("Model trained successfully!");
}
'''

```

4. **Pipeline**:

Combine the data loading, processing, and training into a single flow:

```
```rust
pub fn run_fraud_prediction(file_path: &str) -> Result<(), Box<dyn Error>> {
 let mut reader = ReaderBuilder::new()
 .has_headers(true)
 .from_path(file_path)?;
 let mut records = Vec::new();

 for result in reader.deserialize::<FileStructure>() {
 let rec = result?;
 records.push(rec);
 }

 let (features, labels) = process_data(records);
 train_model(features, labels);

 Ok(())
}
```
```

Tips for Improvement

- **Inspect Data**:

Check the structure of your dataset using tools like `println!("{:?}", records);` before training to ensure no features are missing.

- **Use CSV Headers**:

If you need specific columns, ensure the struct field names match the headers in the CSV file. Alternatively, use dynamic column indexing if the header names are unknown or variable.

- **Feature Scaling**:

Normalize or standardize numerical features before training, which is crucial for many ML algorithms.

- **Libraries**:

- Use `ndarray` for numerical operations.
- Use `linfa` for ML models.
- Use `tch` if you require deep learning capabilities.

This pipeline will ensure that all relevant features from your dataset are included and utilized for machine learning. If specific issues persist, share additional details about your dataset or desired outcome for tailored advice.

Q: How should I utilize the feature columns in training?

Ans:

To utilize these feature columns during training, follow these steps:

1. Organize Feature Columns

Each feature corresponds to a specific index in the training data matrix (`x_data`). Ensure the columns are correctly processed and aligned with the model's expectations.

Column Summary:

| Index | Feature | Type | Transformation |
|-------|------------------|-----------------|----------------------|
| 0 | `step` | Numeric | Use directly |
| 1-5 | `type` | Categorical | One-hot encoding |
| 6 | `amount` | Numeric | Use directly |
| 7 | `nameOrig` | Text/Identifier | Encode as numeric ID |
| 8 | `oldbalanceOrig` | Numeric | Use directly |
| 9 | `newbalanceOrig` | Numeric | Use directly |
| 10 | `nameDest` | Text/Identifier | Encode as numeric ID |
| 11 | `oldbalanceDest` | Numeric | Use directly |
| 12 | `newbalanceDest` | Numeric | Use directly |
| 13 | `isFlaggedFraud` | Numeric | Use directly |

2. Construct Training Data

Extract and transform the data into a numerical matrix (`x_data`) and corresponding labels (`y_data`).

Example:

Here's how to process the feature columns and construct `x_data` and `y_data`:

```
```rust
use ndarray::{Array2, Array1, s};

fn process_training_data(records: &[Record]) -> (Array2<f64>, Array1<usize>) {
 let n = records.len(); // Number of rows
 let num_features = 14; // Total number of features
 let mut x_data = Array2::<f64>::zeros((n, num_features));
```

```

let mut y_data = Array1::<usize>::zeros(n);

let possible_types = vec!["CASH-IN", "CASH-OUT", "DEBIT", "PAYMENT", "TRANSFER"];
let mut name_orig_map = std::collections::HashMap::new();
let mut name_dest_map = std::collections::HashMap::new();

for (i, record) in records.iter().enumerate() {
 // Step
 x_data[[i, 0]] = record.step as f64;

 // Type (One-hot encoding)
 for (j, ttype) in possible_types.iter().enumerate() {
 x_data[[i, 1 + j]] = if &record.ttype == ttype { 1.0 } else { 0.0 };
 }

 // Amount
 x_data[[i, 6]] = record.amount;

 // nameOrig (Encoded as numeric ID)
 let orig_id = *name_orig_map.entry(record.nameOrig.clone()).or_insert_with(|| name_orig_map.len() as f64);
 x_data[[i, 7]] = orig_id;

 // oldbalanceOrg
 x_data[[i, 8]] = record.oldbalanceOrg;

 // newbalanceOrig
 x_data[[i, 9]] = record.newbalanceOrig;

 // nameDest (Encoded as numeric ID)
 let dest_id = *name_dest_map.entry(record.nameDest.clone()).or_insert_with(|| name_dest_map.len() as f64);
 x_data[[i, 10]] = dest_id;

 // oldbalanceDest
 x_data[[i, 11]] = record.oldbalanceDest;

 // newbalanceDest
 x_data[[i, 12]] = record.newbalanceDest;

 // isFlaggedFraud
 x_data[[i, 13]] = record.isFlaggedFraud as f64;

 // Target column: isFraud

```

```

 y_data[i] = record.isFraud as usize;
 }

 (x_data, y_data)
}
'''

3. Split Data into Training and Testing Sets
Divide the data into training and testing sets for evaluation.

'''rust
let split_ratio = 0.8;
let train_size = (n as f64 * split_ratio) as usize;

let x_train = x_data.slice(s![0..train_size, ..]).to_owned();
let y_train = y_data.slice(s![0..train_size]).to_owned();
let x_test = x_data.slice(s![train_size.., ..]).to_owned();
let y_test = y_data.slice(s![train_size..]).to_owned();
'''

4. Train the Model
Use the training data (`x_train` and `y_train`) to train your model.

Example with `linfa` (Logistic Regression):
'''rust
use linfa::prelude::*;
use linfa_logistic::LogisticRegression;

let train_dataset = Dataset::new(x_train, y_train);
let model = LogisticRegression::default()
 .fit(&train_dataset)
 .expect("Failed to train the model");
'''

5. Evaluate the Model
Use the testing data (`x_test` and `y_test`) to evaluate the model's performance.

```

#### Predictions and Accuracy:

```
```rust
let y_pred = model.predict(&x_test);
let mut correct = 0;
let total = y_test.len();

for (pred, actual) in y_pred.iter().zip(y_test.iter()) {
    if pred == actual {
        correct += 1;
    }
}

let accuracy = correct as f64 / total as f64;
println!("Fraud Detection Accuracy: {:.2}%", accuracy * 100.0);
```
```

---

### \*\*6. Key Points for Training\*\*

1. \*\*Feature Importance\*\*:

- Ensure each column is relevant to the task.
- For example, `nameOrig` and `nameDest` might not add meaningful information unless patterns exist in their numeric IDs.

2. \*\*Normalization\*\*:

- Normalize or scale numeric features like `amount`, `oldbalanceOrg`, etc., for better model convergence.

3. \*\*Categorical Handling\*\*:

- Ensure one-hot encoding for `type` is applied consistently during training and testing.

4. \*\*Regularization\*\*:

- Add regularization in your logistic regression model to handle potential overfitting due to large feature sets.

5. \*\*Iterative Training\*\*:

- Fine-tune hyperparameters like learning rate, number of iterations, and regularization strength for better results.

---

### \*\*Conclusion\*\*

Your feature columns are well-structured for training. By:

1. Processing each column appropriately (numerical, categorical, or text).
2. Ensuring consistent transformations.

### 3. Splitting data into training and testing sets.

You can train and evaluate the model effectively, leveraging the rich feature set to detect fraudulent transactions. Let me know if you need help fine-tuning the implementation!

Q: How should I apply one-hot encoding here?

Ans:

One-hot encoding is a process of converting categorical variables into a binary matrix (0s and 1s) where each category is represented as a unique binary vector. In Rust, you can implement one-hot encoding by iterating over the possible categories and encoding each value dynamically.

Here's how you can do one-hot encoding in your fraud detection project:

---

### \*\*1. Define the Possible Categories\*\*

Before encoding, you need to define the list of all possible categories for the column. For example:

```
```rust
let possible_types = vec!["CASH-IN", "CASH-OUT", "DEBIT", "PAYMENT", "TRANSFER"];
```
```

---

### \*\*2. Encode the Categories Dynamically\*\*

Iterate over the records and encode the categorical column into one-hot vectors.

#### Implementation:

Here's how you can perform one-hot encoding for the `type` column:

```
```rust
for (i, record) in records.iter().enumerate() {
    // One-hot encoding for the 'type' column
    for (j, category) in possible_types.iter().enumerate() {
        x_data[[i, 1 + j]] = if &record.ttype == category { 1.0 } else { 0.0 };
    }
}
```
```

#### Explanation:

1. Iterate through the rows of your dataset (`records`).
2. Compare the value of `record.ttype` with each category in `possible\_types`.



3. If the category matches, set the corresponding position in the matrix to `1.0`; otherwise, set it to `0.0`.

---

### ### \*\*3. Generalize One-Hot Encoding\*\*

You can generalize one-hot encoding to work with any categorical column dynamically:

```
```rust
fn one_hot_encode(
    value: &str,
    possible_values: &[&str],
    output: &mut Vec<f64>,
) {
    for category in possible_values {
        output.push(if value == *category { 1.0 } else { 0.0 });
    }
}
```
```

#### #### Usage:

```
```rust
let mut one_hot_vector = Vec::new();
one_hot_encode(&record.ttype, &possible_types, &mut one_hot_vector);
```
```

---

### ### \*\*4. Integrate One-Hot Encoding into Feature Extraction\*\*

Integrate the one-hot encoding process into your feature extraction pipeline:

```
```rust
fn extract_features(records: &[Record], possible_types: &[&str]) -> Array2<f64> {
    let n = records.len();
    let num_features = 14; // Adjust this if you add/remove features
    let mut x_data = Array2::<f64>::zeros((n, num_features));

    for (i, record) in records.iter().enumerate() {
        // Encode other features...

        // One-hot encoding for 'type'
        for (j, category) in possible_types.iter().enumerate() {
            x_data[[i, 1 + j]] = if &record.ttype == category { 1.0 } else { 0.0 };
        }
    }
}
```
```

```
}
```

```
// Encode other features...
```

```
}
```

```
x_data
```

```
}
```

```
...
```

```

```

### \*\*5. Example: Encoding the `type` Column\*\*

Given the following dataset:

| Transaction ID | Type     | Amount |
|----------------|----------|--------|
| 1              | CASH-IN  | 100.0  |
| 2              | CASH-OUT | 200.0  |
| 3              | TRANSFER | 300.0  |

The one-hot encoding for the `type` column with `possible\_types = ["CASH-IN", "CASH-OUT", "DEBIT", "PAYMENT", "TRANSFER"]` would look like:

| Type     | CASH-IN | CASH-OUT | DEBIT | PAYMENT | TRANSFER |
|----------|---------|----------|-------|---------|----------|
| CASH-IN  | 1       | 0        | 0     | 0       | 0        |
| CASH-OUT | 0       | 1        | 0     | 0       | 0        |
| TRANSFER | 0       | 0        | 0     | 0       | 1        |

```

```

### \*\*6. Ensure Consistency Between Training and Testing\*\*

To avoid issues with missing categories during testing:

- Use the same `possible\_types` list for both training and testing datasets.
- Ensure that new categories in the testing data are handled (e.g., by ignoring or treating them as unknown).

```

```

### \*\*7. Advanced: Use Sparse Representations\*\*

If the number of categories is large, one-hot encoding can lead to a sparse matrix with many zeros. Consider using:

- **Label Encoding**: Assign integers to each category, but this may introduce a sense of order where none exists.
- **Embeddings**: Represent categories as dense vectors (useful in deep learning).

---

### ### Complete Code Example

Here's a complete example of feature extraction with one-hot encoding:

```
```rust
fn process_training_data(records: &[Record]) -> Array2<f64> {
    let possible_types = vec!["CASH-IN", "CASH-OUT", "DEBIT", "PAYMENT", "TRANSFER"];
    let n = records.len();
    let num_features = 14; // Adjust based on features
    let mut x_data = Array2::::zeros((n, num_features));

    for (i, record) in records.iter().enumerate() {
        // One-hot encoding for 'type'
        for (j, category) in possible_types.iter().enumerate() {
            x_data[[i, 1 + j]] = if &record.ttype == category { 1.0 } else { 0.0 };
        }

        // Encode other features...
        x_data[[i, 6]] = record.amount; // Numeric column
        // Continue for other columns...
    }

    x_data
}
```
```

---

This approach ensures that categorical columns like `type` are properly transformed into numerical representations, making them suitable for training machine learning models. Let me know if you need help integrating this into your code!

Q: The code gives every possible graphs, but I want to print only like ten random graphs. I should I modify? Don't give the exact code yet, just guide me.

Ans:

Certainly! Here's a guided approach:

#### 1. **Random Sampling**:

- Introduce randomness to select a subset of nodes or edges.
- Use a crate like `rand` to sample nodes or edges randomly from the graph.

2. **Filter Nodes and Edges**:

- After building the graph, create a new subgraph containing only the randomly selected nodes and the edges that connect them.

3. **Modify the Graph Construction**:

- During graph construction, keep a running count of nodes and edges and stop adding to the graph once you've reached your desired size. Alternatively, filter nodes/edges after the graph is complete.

4. **Use DOT Representation**:

- Format and print only the subgraph instead of the entire graph. Use the 'Dot' struct to render the selected nodes and edges.

5. **Steps in Code**:

- Add randomness to node/edge selection.
- Build the subgraph containing the random selection.
- Print only the subgraph.

By focusing on these steps, you can efficiently modify the existing function to print only a subset of the graph. Let me know if you want further clarification on any specific step!

Q: Why are feature columns needed in this code?

Ans:

Feature columns are needed in this code because they represent the **input variables** used by the machine learning model to predict whether a transaction is fraudulent or not. The features capture the essential information from the dataset, transforming raw data into a numerical format that the model can process. Here's why they are crucial:

---

### **1. Represent the Transaction Data**

- Feature columns encode various aspects of a transaction (e.g., step, type, amount, balances, origin/destination names, etc.).
- They turn raw data into structured, numerical input, which the logistic regression model can process.

For example:

- 'step' (time-related information) helps the model understand temporal patterns in fraud.
- 'amount' (transaction value) provides insights into financial behavior.

---

### **2. Provide the Basis for Predictions**

- Features are the **independent variables** that the model uses to make predictions about the target variable ('isFraud').
- The model learns patterns or relationships between features and the target during training.

For example:

- A high transaction amount combined with certain transaction types (like "TRANSFER") might correlate with fraudulent behavior.

---

### ### \*\*3. Enable Machine Learning Models to Work\*\*

- Machine learning models, such as logistic regression, require input in the form of numerical vectors.
- Features encode data in a format the model can process, turning categorical/textual information (like `type`, `nameOrig`, `nameDest`) into numerical or one-hot encoded representations.

---

### ### \*\*4. Facilitate Meaningful Learning\*\*

Each feature column contributes specific information:

- **step**: Encodes the temporal sequence of transactions, useful for identifying time-based fraud patterns.
- **One-hot encoding of `type`**: Captures the transaction type (e.g., CASH-IN, PAYMENT) as a categorical feature.
- **amount**: Represents the transaction size, critical for distinguishing large, suspicious transactions.
- **nameOrig` and `nameDest**: Encoded as numerical IDs to indicate sender and receiver entities.
- **Balances (`oldbalanceOrig`, `newbalanceOrig`, `oldbalanceDest`, `newbalanceDest`)**: Represent changes in account balances, helping the model identify abnormal patterns.
- **isFlaggedFraud**: Indicates if the transaction was flagged as fraud by the system, serving as an additional signal.

---

### ### \*\*5. Support Generalization\*\*

- Carefully chosen feature columns allow the model to generalize beyond the training data, identifying fraud patterns in unseen transactions.
- By using relevant features, the model avoids overfitting to noise or irrelevant details in the data.

---

### ### \*\*6. Example\*\*

Suppose a transaction has the following raw data:

| Step | Type     | Amount | nameOrig | oldbalanceOrig | newbalanceOrig | nameDest | oldbalanceDest | newbalanceDest | isFlaggedFraud |
|------|----------|--------|----------|----------------|----------------|----------|----------------|----------------|----------------|
| 10   | TRANSFER | 10000  | C12345   | 50000          | 40000          | M67890   | 0              | 10000          | 0              |

The corresponding feature vector might look like this:

|      |         |          |       |         |          |        |          |                |                |          |                |                |                |
|------|---------|----------|-------|---------|----------|--------|----------|----------------|----------------|----------|----------------|----------------|----------------|
| Step | CASH-IN | CASH-OUT | DEBIT | PAYMENT | TRANSFER | Amount | nameOrig | oldbalanceOrig | newbalanceOrig | nameDest | oldbalanceDest | newbalanceDest | isFlaggedFraud |
|------|---------|----------|-------|---------|----------|--------|----------|----------------|----------------|----------|----------------|----------------|----------------|

```

-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----		
10	0	0	0	0	1	10000	0.0	50000	40000	1.0	0	10000	0

```

### ### \*\*7. Summary\*\*

Feature columns are needed to:

1. **Encode the raw transaction data** into a numerical format.
2. **Provide the inputs** for training and prediction.
3. **Capture relevant patterns** in the data to enable the model to distinguish fraudulent transactions from legitimate ones.
4. **Ensure compatibility** between the dataset and the machine learning model.