MinSeok Cho

Prof. Chator

CDS DS210 B1

15 Dec 2024

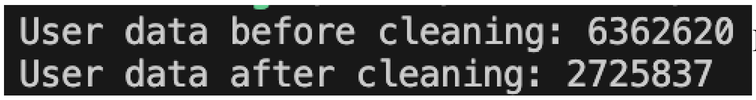
Final Project Write-Up

Dataset Origin: Kaggle (Synthetic Financial Datasets For Fraud Detection)

Here is a brief explanation of the headers of the Synthetic Financial Datasets for Fraud Detection. There are 11 columns and uncountable numbers of rows. The first column is step. This mapped the unit of time in the real world, and 1 represents an hour of time. This dataset includes 31 days of simulation, and the total number of steps is 744. The type represents how the money is transferred. It includes “cash-in,” “cash-out,” “debit,” “payment,” and “transfer.” The amount represents the money amount of the transactions in local currency. As this data includes global data, the numbers here can represent several currencies. The *oldbalanceOrg* represents the initial balance before the transaction from the sender, and the newbalanceOrig refers to the new balance after the transaction from the sender. The *nameDest* represents the customer who is the recipient of the transaction, so this can show where the money went through. The *oldbalanceDest* represents the initial balance of the recipient before getting the transaction, and the newbalanceDest refers to the new balance of the recipient after receiving the transaction. In these columns, M or C will be inserted in the first string to identify whether the recipient is a customer or merchant. The customer may be interpreted as a private, while the merchant can be interpreted as a registered cashier or official corporate. The *isFraud* represents the transactions made by the fraudulent agents inside the simulation. This will represent whether the transaction is actually a fraud or not. The last column *isFlaggedFraud* represents whether the transaction has been identified as potential fraud or not. The system will warn as a fraud if the transfer exceeds 200,000.

I made three modules: analysis.rs, cleaning.rs, and main.rs. The analysis.rs will be in charge of making the graphs and generate the model of calculating the predictions of the possible fraud. The cleaning.rs will be in charge of cleaning the csv file and detect, erase some rows if there are some flaws or input that doesn't fit with the data instructions. The main.rs is the file that will be in charge of combine these two modules and make it work.

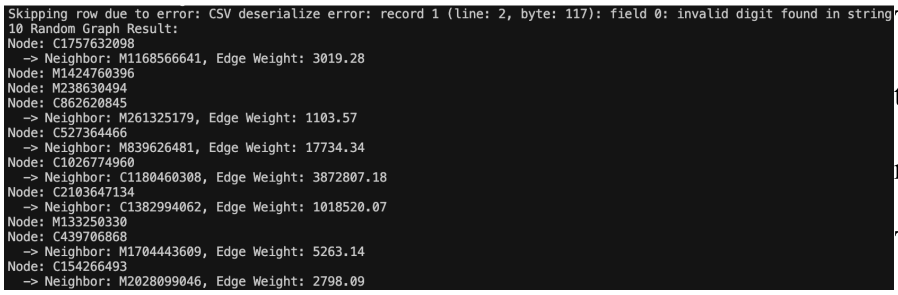
Starting from the cleaning.rs. Two functions are here to clean and reproduce the cleaned csv, and count the rows for the csv files: clean_data, and count_rows. I inserted the count_rows because I wanted to found how many data are erased and cleaned to use for analysis.rs. I first made a struct so that the function and future functions can easily read and follow the steps. This contains all of the rows (or headers) that are in original csv file. The *clean_data* will get *dir* which represents the file name, and the *output* which represents the name we want for our new cleaned data. It will read by the muttable *reader*, and *writer* will help proceed the writing process for new CSV file. Then I will store the possible types that will be useful for our second column of the CSV file. We will call *valid_types*. We will start looping each rows to check whether each row follows the criteria. Each row will be deseriazlied into Transaction struct. If can't be deserialized, it will be skipped (goes to continue - meaning skip the rest of the step for the row). If the type field is not one of the *valid_types*, the row will be skipped. And if the amount exceeds 200,000, I will change isFlaggedFraud to 1. Then check if nameOrig or nameDest starts with M or C. Otherwise, it will skip. After doing these process, the filtered and modified transactions of the records will be serialized and written into new CSV. Then I commanded .flush() in order to emit the result immediately. The function *count_rows* will be similar to the *clean_data*, but it will start counting the rows. As the newly formed data will still have the header, the function will help not count the header as one data.



```
User data before cleaning: 6362620
User data after cleaning: 2725837
```

It turns out that the clean_data helped the original CSV to erase 3,636,783 data which doesn't correspond to the criteria in clean_data function.

Next module is analysis.rs. There are two functions: *build_graph_print_ten* and *run_fraud_prediction*. The first function will guide us to print out some of the graphs that were generated from the CSV file, using the ReaderBuilder. The *graph* and *node_indices* will be here for future use, each representing the graph body and the weight. Next, it will iterate the newly generated CSV rows. The *sender* and *receiver* is here to grab information from the CSV's *nameOrig* and *nameDest*. This will be the possible nodes. The sender_index and the receiver_index are here to identify the positions of the sender and receiver nodes within the graph. The edge_weight will be the transaction amount. And, if the transaction is Fradulent, the weight will be doubled. The doubling will enable the stress on importance of the fraudulent transactions in the graph. Next, we will try starting expressing graph by iterating all the node indices from the graph. The *node_indices* will hold a list of all the node indices. And I decided to return only 10 of the graphs because the code will eventually return every single graphs, so I decided to only print 10 of them. The .choose_multiple helped me on selecting up to 10 elements from the *node_indices* vector. Then we will store selected graphs to *random_nodes*. And, *graph[node].clone()* will get the value from the noe and clone it just for print purposes. The neighbor will get the iterating neighbors gathered from the graph. Along with the neighbors, it will help to print out the weight associated with the graph. The printed result of this function will make you understand 10 of the graphs that starts from senders to receives, and the weight representing the transaction amount.



```
Skipping row due to error: CSV deserialize error: record 1 (line: 2, byte: 117): field 0: invalid digit found in string
10 Random Graph Result:
Node: C1757632098
  -> Neighbor: M1168566641, Edge Weight: 3019.28
Node: M1424760396
Node: M238630494
Node: C862620845
  -> Neighbor: M261325179, Edge Weight: 1103.57
Node: C527364466
  -> Neighbor: M839626481, Edge Weight: 17734.34
Node: C1026774960
  -> Neighbor: C1180460308, Edge Weight: 3872807.18
Node: C2103647134
  -> Neighbor: C1382994062, Edge Weight: 1018520.07
Node: M133250330
Node: C439706868
  -> Neighbor: M1704443609, Edge Weight: 5263.14
Node: C154266493
  -> Neighbor: M2028099046, Edge Weight: 2798.09
```

This is the example graph that represents 10 randomly selected graphs. The first graph, for example, can be interpreted that C1757632098 sent 3019.28 to M1168566641.

The next function - *run_fraud_prediction* - will also start with reading the new CSV file. Just in case if there's an error, the error message will printed out. The *records* will store the recognized data from CSV reading. If there's no data that can be read, the function will halt here. But if it continues, I will make *name_orig_map* and *name_dest_map* so map the corresponding transaction in order of the row iteration. (Why these are needed?) // And I decided to assign the feature columns by numbers. Machine learning to predict whether the transaction is fraudulent or not requires an input variable with feature columns. 0 stands for the step, 1 to 5 represents the one-hot encoding for types, 6 for amount, 7 for nameOrig, 8 for oldbalanceOrg, 9 for newbalanceOrig, 10 for nameDest, 11 for oldbalanceDest, 12 for newbalanceDest, and 13 for ifFlaggedFraud. We don't include isFraud as this is the target variable. The totla feature columns are 14. The *x_data* and *y_data* will be the feature matrix and target vector respectively. Next, we can see the iterated records that saved the interpretable data, and it will fill the features in to x_data and y_data for future training purposes. I first set split ratio to 0.7 and train_size will determine the number of records to allocate to the training set. Based on split_ratio, the training set will be determined. From the x_data and y_data gathered from the iterated above code block, these will be utilized for the training and test appropriately. The train variable will create a

Dataset from x_train and y_train so that it can be sued to train regression model. The max_iter will specify how many iterations will be there for training. For each iteration, it will train via logistic regression model. Next, the variable model will be used for final logistic regression model training. Eventually, the model will make predictions on the test set *x_test* and compare the result with the actual labels from the *y_test*. If there is a matching between the prediction and actual, we will increment the *correct*. And calculate the accuracy.

For this, when I ran with the split_ratio of 0.5, the accuracy was 99.80138%, while the split_ratio of 0.7 was 99.72094%. The reason for this can be from too large data or lack of data for 'testing.' In addition, the overfitting may be the reason. Also, when the change in max_iter was seen, the accuracy didn't change. This was unfortunate and I think this was resulting from the reason that the logistic regression didn't have enough complex shape, or the model converges too quickly.

The main.rs is here to call what the name of CSV is that I am trying to read, and give the name of CSV file that I will use for updated and cleaned CSV file. And with the count_rows function from the cleaning.rs, it will print out the number of rows. I didn't decide to integrate the print of the number of rows because I wanted to differentiate the before and after of the data cleaning.

For analysis.rs test, I tried to capture whether the test_build_graph can execute the build_graph_print_ten function without panicking, and test_run_prediction can execute the run_fraud_prediction without panicking. In cleaning.rs test, I tried to verify whether temporarily made file can be read and can write rows via writeln!. And with the temporary data, it will focus whether it can utilize clean_data. In main.rs, I tested on the main.rs not panicking on the run_application, which compiled every functions of the analysis.rs and cleaning.rs.