

Introduction:

I chose this dataset because I am interested in understanding user behavior and product interactions and how it became an important user experience. I got my dataset from Kaggle and I was able to find a fictional e-commerce dataset that had enough nodes to analyze. My central idea comes from my desire to uncover the relationships between users and products and learn insights about product connectivity. Using this dataset, I want to find out which users are the super buyers (highest transaction activity) and if there is a connection between products or users, how strongly are they connected to each other. I proceed this project by analyzing the relationships that are within a distance of 2 to reveal indirect relationships such as products frequently purchased by users with shared preferences, power-law fit, and how well the distribution fits the assumption. Lastly, with these results, I can use them in real-world situations like understanding trends. While asking questions for one of the TAs, I also referred to lecture 10 notes to refresh my understanding of HashMap and HashSet and Homework 4 for reference.

link: <https://www.kaggle.com/datasets/steve1215rogg/e-commerce-dataset?resource=download>

User_analysis.rs:

I imported HashMap and HashSet for mapping and storing and imported transaction struct from csv_utils. The first public function I made is called analyse_users which will analyze the user behavior and the input will be a slice of struct from Transaction (transaction: &[Transaction]) and the output will be a HashMap where user_id is the key and the value is (number_of_purchase, total_spending). Then it will initialize the user_summary by creating an empty HashMap named user_summary so it can store and accumulate data from each user as the function iterates. Now for the actual for loop, it will loop through all transaction objects in the input slice and it will initialize the entry that checks if the user_id already exists in the user_summary map. If the user exists, it will return a mutable reference to the existing value and if it doesn't, then it will insert a new entry with default tuple and return a mutable reference to it. .or_insert((0, 0.0)) will specify the default value of the parameter I just wrote for new entries. And these will ensure that every user_id is included in the user_summary before updating happens. And for updating the entry, entry.0 += 1 will increment the first value of the tuple by 1 and this way it will track the total number of transactions for the current user_id. And for updating that transaction amount, entry.1 += transaction.final_price will add the final_price from the current struct to the second value of the tuple. Then it will return the summary and this way I know the total spending for each user_id. This is a crucial step for data prep to identify super buyers.

Now for the function that identifies super buyers, this function will only consider with a certain criteria in main.rs. The inputs are user_summary that is a reference to HashMap, purchase_threshold, spending_threshold. And the output will be a Vec<String> containing the ids. It will start with an iterator over the HashMap and filter the iterator to keep only a credit that exceeds the threshold. The closure parameters |(&total_purchase, total_spending)| is basically saying that the first time is ignored and it deconstructs the tuple from the reference to the value. And this is the logic: the


total_purchases must be greater than or equal to purchase_threshold or the total_spending is greater than or equal to spending_threshold. Then it's going to transform each key, value pair that goes through the filter into user_id. Then it will collect the user ids into a Vec<String>

Moving onto the public function degree_distribution, the whole purpose of this function is that it will calculate the degree of each user and each product. So degree of user → number of products they purchased. Degree of product → number of users who purchased it. The input will be transaction: &[Transaction], again a slice of transaction struct and the output is the tuple. It will initialize two empty HashMaps one named user_degree and another one named product_degree and again it's important that these are mutable so I can update these later on. then the for loop is going to loop transaction. Then it will check if user_id exists in the corresponding degree and return a mutable reference to the current degree value and if not, it will insert a user_id to a degree of 0. This whole thing differences the mutable reference and increments the degree by 1. It will keep track of how many products each user has purchased. For product_degree, it will also check if the product_id exists in the corresponding degree and insert 0 if it doesn't. Then increment by 1 for every transaction. Then (user_degrees, product_degrees) will be returned as tuples. By this, it will provide a summary of the connections.

Next, based on my research of what a power-law distribution is, it suggests that a small number of elements have high connections while most elements have few connections. With this concept, it will help me find highly engaged users who buy many products or popular products purchased by many users. So the public function fit_power_law will compute the power-law exponent. The input will be degrees: &HashMap<String, usize> and the output will return an f64 that represents the estimated power-law exponent. To extract a degree value, degree_values will extract all degree values from HashMap and convert into Vec<f64> and will iterate over the degree in the HashMap. Then .map will convert each degree value into f64 then collect the result into Vec<f64>. Now the prep for computation is done. Computing the total number of degrees, it will calculate the total number of nodes by finding the length of degree_values as f64. This is important because n is required to calculate the power-law exponent. And next to find the minimum degree, k_min will find the smallest degree value in degree_values and iterating over Vec<f64> and cloning each value from the iterator. Then .fold(f64::INFINITY, f64::min) will use a folding operation that starts with the largest value and finds the smallest value. This is useful because it is used to normalize the degree values for the logarithmic sum next. Then log_sum will calculate the sum of logarithm by iterating and dividing k by kmin and the .ln() will output the natural log of the normalized value then compute the total sum. Then it will calculate the power-law exponent that simply adds the base value and the degree count divided by log sum. I asked a TA about the concept of this method and utilized AI (screenshot below) to help me guide through these functions and now I know how to find the smallest degree and compute the sum of logs. So, I know that by using power-law analysis, I can see whether a few users account for the majority of


transactions, giving an insight if I have super buyers who “controls” the market. And for products, figure out which products might need more maintenance/ attention to normalize the market in a bigger scope. Overall, I can see the concentration of influence.

rust

 Copy code

```
let k_min = degree_values.iter().cloned().fold(f64::INFINITY, f64::min);
```

rust

 Copy code

```
let log_sum: f64 = degree_values.iter().map(|&k| (k / k_min).ln()).sum();
```

For compute_distance_2_neighbors function, this takes parameters of a slice (same thing from previous functions), and output of a HashMap, HashSet. I first initialized a graph that creates an empty HashMap. Then the for loop will loop and add edges for user and product. Graph.entry will go through the process of checking the existence and adds the transaction.product_id as a neighbor of the user. This will help capture the connection between the user and the product they purchased. Then similar steps for adding products to the user edge. Now this will be the opposite of the line above, it will get the connection between the product and the user. Now the next chunk of the code will get the neighbors of the neighbors. This is where .get comes in useful because it retrieves the HashSet of direct neighbors for the current neighbor. And it let statement of Some(second_neighbor) ensure that the neighbor has direct neighbors before executing. Next for the second_neighbor for loop, it will check if the second neighbor is not the original node itself and it will add the second_neighbor to the distance_2. This is where it builds the set of nodes that are reachable within 2 walks from the current node that the inputs plug in. finally, it will store the neighbors into distance_2_neighbors.

For the public function build_category_connections, I noticed that there is a lack of connectivity with the users or products so to enrich the relationship, I came up with a function that will create connections between users based on the product categories. So it starts with the slice transaction and outputs HashMap, HashSet. I initiated category_to_users and category_graph. It will map each category to the users who purchased products in that category and it will store the final connections between users. Category_to_users.entry will check if the category from the current transaction exists in category_to_use. It will then add the user_id from the transaction to the HashSet for the category. The outer loop will iterate over categories, and the inner loop will iterate over users in a category. To make sure that each user in the category has an entry, connections will check the existence in category_graph and create an empty set only if the user_id is not already in it. And the next inner loop will connect users in the same category and this is done by iterating over other_user and skipping self-connections.

Then add the other_use to the current users' set of connections. Lastly, it will return category_graph. And for the very last function, build_product_connections, it will use the exact same logic and build a relationship between products based on shared users, allowing to study the relationship between products.

Csv_utils.rs:

In this module, the codes will load, validate, and tailor the csv file. The public struct represents a single transaction in the dataset. Deserialize will allow the transaction struct to be automatically populated from a csv row. The fields I created are: user_id that is a String, product_id that is a String, category that is a String, and final_price that will be represented as a f64. The way this works is that when the csv row is generated, the element (rows) values are mapped to the fields of the above transaction struct I just created. So for example, user 1, product 1, clothing, 50.0 will look like: Transaction {user_id: "user1", product_id: "product1", category: "clothing", final_price: 50.0}. Next, the public function clean_and_load_csv will read the csv file. ReaderBuilder from csv crate will open the file at file_path and "?" operator will ensure that any errors like files not being founded are propagated. Then it will create an empty vector to store and I let this be mutable so I can update transactions each time I store the values and this is done by Vec::new(). Next, the iterator will read each row of csv file and will deserialize into a transaction object. Record will extract the transaction object from the result and if the row cant be parsed, then the function will return an error. Then an if statement will filter out the transaction with a finala_price greater than 0.0 are added to the transaction vector to make sure that I only consider the customers who made a purchase. Then OK(transactions) will return the vector. For error handling, if there is an error in the file existence, it will return an error saying "No such file or directory" and this goes the same for having an invalid data, it will output an error message. This step is important because before analyzing the dataset, I want to make sure that all the unnecessary noise is removed/cleaned. This function will later get called in main.rs.

main.rs:

I first imported modules and libraries by mod csv_utils, and mod user_analysis, which are the other two modules for this project. Then I imported clena_andload csv from csv_utils module so it can read the csv file. Secondly, I imported functions to analyze transactions and other properties from user_analysis using crate. Thirdly, I used HashMap to store mappings with users and products. Lastly, std::error::Error will handle errors that will occur when the functions are executed. I first created a helper function for displaying results. The function is called display_top_results and it displays the top n results that I stored in HashMap. The sorted_results will convert the hashmap into a vector (Vec<_>). Then it will sort the vectors in descending order based on values → (|a, b| b.1.cmp(a.1)). Then I put a print statement to print the top_n followed by a label. Then the for loop will display only the top n items from the sorted list. This is done by .take(top_n). If i didn't have this helper function, my outputs would look extremely messy and I would get a long list of unwanted outputs, so I had to create this function to make sure I only get top numbers of the output. I will later on address how many outputs I

want for each function. Moving onto the main function, it will return a result and it will ensure error returning alerts and that is what the `Box<dyn Error>` is for. Then the `file_path` will load the data by calling the `clean_and_load_csv` function I mentioned earlier and `transaction` is a vector of valid transaction objects representing the cleaned dataset. And the question mark is there for error propagation. I threw an `user_summary` that basically summarized the user behavior and it will map each `user_id` to a tuple like number of purchases and total spending (`usize` and `f64`). Now I set a purchase threshold of 5, spending threshold of 1000.0 to identify the significance so anything that exceeds that amount, it will be considered in my functions. Then to identify the super_buyers in my dataset, there will be a vector of `user_ids` meeting these requirements. To extract the super buyers here, the `HashMap` will help map the buyers to their purchase counts using the `user_summary` I addressed above. This is done by `iter().map.collect` And to avoid getting too many outputs, I cut down to only return the top 5 super buyers using the `display_top_results` that I created as a helper function. Moving onto my computing degree distribution this section of code basically outputs the number of edges for each user and product in the graph. It will display the top 5 user degrees by referencing `user_degree` and this applies to the `product_degrees` line below this. Then moving onto the power-law distribution, I got this idea from the piazza post and what this code is doing here is that it will fit a power-law distribution to the degree data and print out the calculated exponents for users and products. How this function actually works will be described later in the `user_analysis.rs` module. For distance-2 neighbors, the `distance_2` will compute the number of neighbors each node has within 2 steps and this really shows what type of relationship they share between users and products. Just like the other functions above, it will take an iterator and convert its elements into a new structure. The node will be cloned and the neighbors are replaced by its length. The reason why it will clone is because `t=`since `HashMap` keys are immutable and ownership cannot be transferred. The result is collected into a newly created structure. Then the category based connections will create edges between users who purchased a product from the same category. And the logic follows the same as above function. Lastly, the product-based connections will connect products purchased by the same users. Then the `Ok(())` indicates a successful execution and errors are propagated. So overall, the key output I'm looking for in each function is that the super buyers will list users with the highest purchase count or spending incorporating the threshold I set in the beginning. The degree distribution will highlight the most connected users and products, the power-law will show whether the graph follows a scale-free structure, and distance-2 neighbors is there to help identify indirect relationship, category- based connection will reveal the commonality based on the category they purchase, and finally product-based connections will show the frequency of purchasing another product together.

Test functions:

I wrote 3 test functions: `test_super_buyers`, `test_compute_distance_2_neighbors`, and `test_category_connections`. Super buyers will test that `identify_super_buyers` will identify users based on the threshold I set up. `Compute_distance` test function will validate `compute_distance_2_neighbors` by making sure that distance of 2 returns as a result. `Category_connections` will call `build_category_connections` by ensuring user connections through shared categories. For the first function, I set up an example amount of transaction

counts and total spending value for each user. Then it will read the threshold values and call `identify_super_buyers`. From what I inserted, user1,2, and 4 will be passed but user 3 will fail because it does not meet the requirements. Notice how I assign user 3 to not go over the threshold. For the second test function, I set it up in a vector form and called the `distance_2_neighbors` function and this is where it will test the distance for all nodes. From the information I plugged in earlier, it will verify that the user can reach user 3 through distance of 2. Finally, for testing `category_connections`, I created an example dataset where user1 and user2 share category 1 and user3 is in a separate category2. It will then call the `category_connections` function that plugs in the example dataset and verify that user1 is connected to user2 with category1. And lastly, `assert! (!)`; will test that user1 is not connected to user3 and it makes sense since they don't belong in the same category. I ran "cargo run" to get my results –When I ran the products by degree, I got some outputs of "941bb2a0-3" to be 1, when I ran the distance-2 neighbors function and one of the top 5 nodes were "c2dbb365" with 0 output, and lastly when I ran products by connection one of the results I got was product of "ad73b872-2" with 1 connection.