Identifying Consumer Trends in Different Seasons using Rust graphs.

In this project, I used the Consumer Shopping Trends Dataset from Kraggle. The dataset contains information about the consumer and their purchases. The primary goal of my project is to analyze shopping trends, especially focusing on how these trends vary across different seasons.

The Dataset contains:
Customer ID - Unique identifier for each customer
Age - The age of the customer
Gender - Gender of the customer (Male/Female)
Item Purchased - The item purchased by the customer
Category - Category of the item purchased
Purchase Amount (USD) - The amount of the purchase in USD
Location - The location where the purchase was made
Size - Size of the purchased item
Color - The color of the purchased item
Season - Season during which the purchase was made
Review Rating - Rating given by the customer for the purchased item
Subscription Status - Indicates if the customer has a subscription (Yes/No)
Shipping Type - The type of shipping chosen by the customer
Discount Applied - Indicates if a discount was applied to the purchase (Yes/No)
Promo Code Used - Indicates if a promo code was used for the purchase (Yes/No)
Previous Purchases - The total count of transactions concluded by the customer at the store, excluding the ongoing transaction
Payment Method - Customer's most preferred payment method
Frequency of Purchases - Frequency at which the customer makes purchases (e.g., Weekly, Fortnightly, Monthly)

There are a total of 3900 customers in this dataset.

By constructing and analyzing a graph where nodes represent items and edges represent relationships (category similarity), the project aims to:

- Identify key items (nodes) that are central to consumer purchasing patterns.
- Understanding how these patterns change with seasons provides information on seasonal variations in shopping patterns.

The sample output for this project:
Item 'Blouse': Degree Centrality: 0.4167
Item 'Sweater': Degree Centrality: 0.4167
Item 'Jeans': Degree Centrality: 0.4167
…

Season Spring:
  Item 'Blouse': Seasonal Degree Centrality: 0.0100
  Item 'Sweater': Seasonal Degree Centrality: 0.0030
  Item 'Jeans': Seasonal Degree Centrality: 0.0100
…

Season Summer:
  Item 'Blouse': Seasonal Degree Centrality: 0.0031
  Item 'Sweater': Seasonal Degree Centrality: 0.0010
  Item 'Jeans': Seasonal Degree Centrality: 0.0105
…

The output prints out the degree centrality of all the items and then the degree centrality of all the items for each season. This can help firms in the fashion industry to know the spending expenditure of items throughout the year and they can effectively market for particular products.

We can look at the degree of centrality for the item "coat":
Season Spring:
Item 'Coat': Seasonal Degree Centrality: 0.0100
Season Fall:
Item 'Coat': Seasonal Degree Centrality: 0.0031
Season Winter:
Item 'Coat': Seasonal Degree Centrality: 0.0103
Season Summer:
Item 'Coat': Seasonal Degree Centrality: 0.0010

We can see the seasonal degree centrality is the highest in winter and then in spring. This suggests that this item is more central to the customers' purchases during these times. This could be due to the colder weather, making coats more essential and, therefore, more frequently purchased.
The lower centrality scores in fall and summer show that the item is less central to purchasing patterns in these seasons. It's likely due to the warmer weather, resulting in reduced demand for coats.

For the fashion industry: In winter and spring, a fashion retailer could allocate more marketing resources to promote coats during these seasons. This could include advertising campaigns, in-store displays, and online promotions.
During fall and summer, when the demand for coats is lower, retailers could offer discounts to clear out remaining stock and make room for seasonally appropriate clothing.
The code:

Item Struct: Defines the structure of each record in your dataset. Each Item instance represents a unique purchase record from the dataset.

read_csv Function: Reads a CSV file and converts each record into an Item instance. Returns a Result which is either Ok(Vec<Item>) containing a vector of Item instances or an Error if any occurs during file reading or parsing.

Graph Module:

create_nodes Function: Creates nodes for a directed graph, where each node represents a unique item purchased. It uses item_purchased as the key to ensure uniqueness and associates each item with a NodeIndex. iterates over the items slice. For each item, it checks if a node for that item already exists in the graph; if not, adds a new node. Returns a HashMap where keys are item names (String) and values are their corresponding NodeIndex in the graph.

create_edges Function: It adds edges between nodes in the graph. In this implementation, an edge is added between two items if they belong to the same category, but are not the same item. This models the relationship between items of the same category.

build_graph Function: Combines the creation of nodes and edges to build the complete graph. It returns both the graph and the mapping between item names and their respective nodes. Initializes a new DiGraphMap. Calls create_nodes to add nodes to the graph. Calls create_edges to add edges between nodes. Returns the constructed graph and the item-node mapping.

Centrality module

calculate_degree_centrality Function: Calculates the degree centrality for each node in the graph. For each node in the graph, count the number of edges it has. Calculates the degree centrality for each node as the number of edges divided by the total number of nodes minus one. Returns a vector of centrality scores corresponding to each node.

calculate_seasonal_degree_centrality Function: Calculates degree centrality scores for items, broken down by seasons. For each season, it calculates centrality scores for items belonging to that season. It Returns a HashMap where keys are seasons and values are vectors of tuples (item name, centrality score).

Main Function:
It performs the following steps: Reads the CSV file and checks for errors. Builds the graph from the read data. Creates a reverse mapping from NodeIndex to item names for easy lookup. Calculates and prints the degree centrality of each item. Calculates and prints the seasonal degree centrality.

Main.rs

```
use std::{error::Error, collections::HashMap};
use csv::Reader;
use petgraph::graph::NodeIndex;

mod graph;
mod centrality;

#[derive(Debug, Clone, PartialEq,Eq, Hash)]
//the struct for the csv file
struct Item {
   customer_id: usize,
   age: usize,
   gender: bool,
   item_purchased: String,
   category: String,
   purchase_amount: usize,
   location: String,
   size: String,
   color: String,
   season: String,
   review_rating: usize,
   subscription_status: bool,
   shipping_type: String,
   discount_applied: bool,
   promo_code_used: bool,
   previous_purchases: usize,
   payment_method: String,
   preferred_payment_method: String,
   frequency_of_purchases: String,
   edges:Vec<String>,
}



//reads the csv file and returns a vector of items
fn read_csv(file_path: &str) -> Result<Vec<Item>, Box<dyn Error>> {
   let mut reader = Reader::from_path(file_path)?;
   let _headers = reader.headers()?.clone();

   let data: Vec<Item> = reader
       .records()
```

```rust
        .filter_map(|result| {
            result.ok().and_then(|record| {

                Some(Item {
                    customer_id: record[0].parse().unwrap_or_default(),
                    age: record[1].parse().unwrap_or_default(),
                    gender: record[2].parse().unwrap_or(false),
                    item_purchased: record[3].to_string(),
                    category: record[4].to_string(),
                    purchase_amount: record[5].parse().unwrap_or_default(),
                    location: record[6].to_string(),
                    size: record[7].to_string(),
                    color: record[8].to_string(),
                    season: record[9].to_string(),
                    review_rating: record[10].parse().unwrap_or_default(),
                    subscription_status: record[11].parse().unwrap_or_default(),
                    shipping_type: record[12].to_string(),
                    discount_applied: record[13].parse().unwrap_or_default(),
                    promo_code_used: record[14].parse().unwrap_or_default(),
                    previous_purchases: record[15].parse().unwrap_or_default(),
                    payment_method: record[16].to_string(),
                    preferred_payment_method: record[17].to_string(),
                    frequency_of_purchases:record[18].to_string(),
                    edges: Vec::new(),
                })
            })
        })
        .collect();

    Ok(data)
}


fn main() {
   match read_csv("/Users/krisma/Desktop/project/shopping_trends.csv") {
       Ok(items) => {
           let (graph, item_node_mapping) = graph::build_graph(&items);

           let degree_centrality = centrality::calculate_degree_centrality(&graph);

           // Create a reverse mapping from NodeIndex to item name
           let reverse_mapping: HashMap<NodeIndex, String> = item_node_mapping
               .iter()
```

```
                .map(|(item, &node)| (node, item.clone()))
                .collect();

            // Print the degree centrality along with the item name
            for node in graph.nodes() {
                if let Some(item_name) = reverse_mapping.get(&node) {
                    let centrality = degree_centrality[node.index()];
                    println!("Item '{}': Degree Centrality: {:.4}", item_name,
centrality);
                }
            }

            let seasonal_centrality =
centrality::calculate_seasonal_degree_centrality(&graph, &items, &item_node_mapping);

            // Print the seasonal degree centrality for each node with item names
            for (season, centrality_scores) in seasonal_centrality.iter() {
                println!("Season {}:", season);
                for (node, centrality) in graph.nodes().zip(centrality_scores.iter()) {
                    if let Some(item_name) = reverse_mapping.get(&node) {
                        println!("  Item '{}': Seasonal Degree Centrality: {:.4}",
item_name, centrality);
                    }
                }
            }
        }
        Err(e) => println!("Error reading CSV file: {:?}", e),
    }
}



//test module
#[cfg(test)]
mod tests {
    use super::*;

    fn create_test_items() -> Vec<Item> {
        vec![
            Item {
                customer_id: 1,
                age: 30,
```

```rust
        gender: true,
        item_purchased: "Shirt".to_string(),
        category: "Clothing".to_string(),
        purchase_amount: 100,
        location: "Hawaii".to_string(),
        size: "M".to_string(),
        color: "Grey".to_string(),
        season: "Spring".to_string(),
        review_rating: 3,
        subscription_status: true,
        shipping_type: "Express".to_string(),
        discount_applied: false,
        promo_code_used: false,
        previous_purchases: 3,
        payment_method: "Venmo".to_string(),
        preferred_payment_method: "Credit Card".to_string(),
        frequency_of_purchases: "Every 3 Months".to_string(),
        edges: Vec::new(),


    },
    Item {
        customer_id: 2,
        age: 25,
        gender: false,
        item_purchased: "Pants".to_string(),
        category: "Clothing".to_string(),
        purchase_amount: 150,
        location: "New York".to_string(),
        size: "L".to_string(),
        color: "Black".to_string(),
        season: "Winter".to_string(),
        review_rating: 4,
        subscription_status: false,
        shipping_type: "Standard".to_string(),
        discount_applied: true,
        promo_code_used: true,
        previous_purchases: 5,
        payment_method: "Credit Card".to_string(),
        preferred_payment_method: "Credit Card".to_string(),
        frequency_of_purchases: "Once a Year".to_string(),
        edges:Vec::new(),
```

```rust
            },
            // Add more test items as needed
        ]
    }

    #[test]
    fn test_create_nodes() {
        let items = create_test_items();
        let mut graph = petgraph::graphmap::DiGraphMap::new();
        let nodes = graph::create_nodes(&mut graph, &items);

        assert_eq!(nodes.len(), items.len());
        assert!(nodes.contains_key("Shirt"));
        assert!(nodes.contains_key("Pants"));
    }
    #[test]
    fn test_read_csv() {
        let file_path = "/Users/krisma/Desktop/project/shopping_trends.csv"; let data =
read_csv(file_path).unwrap();
        assert_eq!(data.len(), 3901); // Num of rows in CSV file
    }

    #[test]
    fn test_create_edges() {
        let items = create_test_items();
        let mut graph = petgraph::graphmap::DiGraphMap::new();
        let item_nodes = graph::create_nodes(&mut graph, &items);

        graph::create_edges(&mut graph, &items, &item_nodes);

        let shirt_node = item_nodes.get("Shirt").unwrap();
        let pants_node = item_nodes.get("Pants").unwrap();

        assert!(graph.contains_edge(*shirt_node, *pants_node));
    }


    #[test]
    fn test_centrality() {
            let items = create_test_items();
            let (graph, item_node_mapping) = graph::build_graph(&items);
```

```rust
        let degree_centrality = centrality::calculate_degree_centrality(&graph);

        let shirt_node = item_node_mapping.get("Shirt").unwrap();
        let pants_node = item_node_mapping.get("Pants").unwrap();

        let shirt_centrality = degree_centrality[shirt_node.index()];
        let pants_centrality = degree_centrality[pants_node.index()];

        //'Shirt' and 'Pants' are the only two items and connected,
        // their centrality should be 1/(2-1) = 1.0
        assert_eq!(shirt_centrality, 1.0);
        assert_eq!(pants_centrality, 1.0);
    }



}
```

Graph.rs

```rust
use std::collections::HashMap;
use petgraph::graph::NodeIndex;
use petgraph::prelude::DiGraphMap;

use crate::Item;
//creates nodes for each purchased item
pub(crate) fn create_nodes(
    graph: &mut DiGraphMap<NodeIndex, ()>,
    items: &[Item],
) -> HashMap<String, NodeIndex> {
    let mut item_nodes: HashMap<String, NodeIndex> = HashMap::new();

    for item in items {
        item_nodes.entry(item.item_purchased.clone())
            .or_insert_with(|| graph.add_node(NodeIndex::new(graph.node_count())));
    }

    item_nodes
}

//creates edges between nodes based on the same category
pub(crate) fn create_edges(
```

```
    graph: &mut DiGraphMap<NodeIndex, ()>,
    items: &[Item],
    item_nodes: &HashMap<String, NodeIndex>,
) {
    for item in items {
        let node = item_nodes.get(&item.item_purchased).unwrap();

        // Example: Connect items with the same category
        for other_item in items {
            if item.category == other_item.category && item.item_purchased !=
other_item.item_purchased {
                let other_node = item_nodes.get(&other_item.item_purchased).unwrap();
                graph.add_edge(*node, *other_node, ());
            }
        }
    }
}

//builds the graph based on the nodes and the edges
pub(crate) fn build_graph(items: &[Item]) -> (DiGraphMap<NodeIndex, ()>,
HashMap<String, NodeIndex>) {
    let mut graph = DiGraphMap::new();

    let item_node_mapping = create_nodes(&mut graph, items);
    create_edges(&mut graph, items, &item_node_mapping);

    (graph, item_node_mapping)
}
```

Centrality.rs

```
use std::collections::{HashMap, HashSet};
use petgraph::graph::NodeIndex;
use petgraph::prelude::DiGraphMap;

use crate::Item;

// Calculates the degree centrality of each node in the graph.
pub(crate) fn calculate_degree_centrality(graph: &DiGraphMap<NodeIndex, ()>) ->
Vec<f64> {
    let num_nodes = graph.node_count() as f64;
```

```rust
    let degrees: Vec<usize> = graph.nodes().map(|node|
graph.neighbors(node).count()).collect();

    degrees.iter().map(|&degree| degree as f64 / (num_nodes - 1.0)).collect()
}

// Calculates the seasonal degree centrality of each node in the graph.
pub(crate) fn calculate_seasonal_degree_centrality(
    graph: &DiGraphMap<NodeIndex, ()>,
    items: &[Item],
    item_node_mapping: &HashMap<String, NodeIndex>,
) -> HashMap<String, Vec<f64>>{
    let mut seasonal_centrality: HashMap<String, Vec<f64>> = HashMap::new();

    for season in items.iter().map(|item| &item.season).collect::<HashSet<&String>>() {
        let subgraph_nodes: Vec<NodeIndex> = items
            .iter()
            .filter(|item| &item.season == season)
            .filter_map(|item| item_node_mapping.get(&item.item_purchased))
            .copied()
            .collect();


        let num_nodes = subgraph_nodes.len() as f64;
        let mut centrality_scores = Vec::new();

        for node in subgraph_nodes {
            let degree = graph.neighbors(node).count() as f64;
            centrality_scores.push(degree / (num_nodes - 1.0));
        }

        seasonal_centrality.insert(season.clone(), centrality_scores);
    }

    seasonal_centrality
}
```

Bibliography

https://docs.rs/petgraph/latest/petgraph/graph/index.html
https://lib.rs/crates/graphrs
https://docs.rs/rustworkx-core/latest/rustworkx_core/
https://users.rust-lang.org/t/using-petgraph-graphmap-to-create-sub-graphs/79041
https://docs.rs/petgraph/latest/petgraph/graph/struct.Graph.html