2882. Drop Duplicate Rows

Easy

Leetcode Link

Problem Description

The problem presents a DataFrame named `customers` with columns `customer_id`, `name`, and `email`. We are informed that there are duplicate records based on the `email` column. The task is to remove these duplicate rows, but crucially, we must keep only the first occurrence of each `email`. The DataFrame may contain unique `customer_id` values and `name` values, but some `emails` are associated with more than one customer. The goal is to return a DataFrame where all `email` addresses appear only once, preserving the record of the first customer who had that email.

Intuition

The `drop_duplicates` function from the `pandas` library immediately comes to mind for this task. This function is tailor-made for situations like this, where we need to remove duplicate rows based on one or multiple column values. The function comes with the `subset` parameter, allowing us to specify on which columns to check for duplicates. In this case, we'll set the `subset` to `['email']`, so the function will only consider the `email` column for finding duplicates.

By default, `drop_duplicates` keeps the first occurrence of a duplicated row, which aligns perfectly with our requirements. We do not need to set the `keep` argument as its default value is `'first'`. We can also safely ignore the `inplace` parameter, or set it to `False`, as we want to return a new DataFrame rather than modify the original `customers` DataFrame in place. This provides a simple and efficient solution to the problem that can be implemented and understood with minimal code.

The solution code leverages the described functionality provided by `pandas`, and thus with a single line of code, we achieve the desired output, effectively removing all duplicates based on the `email` while keeping the first occurrence of each.

Solution Approach

The solution is straightforward due to the robust capabilities of the Pandas library in Python, which is designed to handle and manipulate data in DataFrame structures.

The key steps of the solution are:

1. Import the Pandas library to use its functionalities.
2. Define a function `dropDuplicateEmails`, which takes a `customers` DataFrame as its argument.
3. Use the `drop_duplicates` method available in the Pandas library to remove duplicate rows based on specific column values. The `drop_duplicates` method identifies and removes duplicate rows with the following considerations:
   - The `subset` parameter specifies the columns to consider for identifying duplicate rows. In our case, this is set to `['email']` so that the method looks for duplicates only in the `email` column.
   - By default, the `keep` parameter is set to `'first'`, which means that if duplicates are found, the first occurrence is kept while the subsequent duplicates are removed. This behavior is exactly what we need to solve this problem, so we do not need to specify this parameter explicitly.
   - The method returns a new DataFrame with the duplicates removed, which we immediately return from our function.

Therefore, the core algorithm involves no complex loops or conditionals due to the high-level abstraction provided by Pandas. The data structure used is the DataFrame itself, and the pattern applied is the direct use of a library function designed for this exact purpose.

The implementation is as follows:

```python
import pandas as pd

def dropDuplicateEmails(customers: pd.DataFrame) -> pd.DataFrame:
    return customers.drop_duplicates(subset=['email'])
```

This elegant and concise implementation leverages Pandas' capabilities to solve the problem with minimal code and time complexity.

Example Walkthrough

Let's walk through a small example to illustrate how the solution approach effectively removes duplicate email addresses and retains only the first occurrence of each.

Suppose we have a small `customers` DataFrame that looks like this:

customer_id	name	email
1	Alice	alice@example.com
2	Bob	bob@example.com
3	Alice	alice@example.com
4	Charlie	charlie@example.com
5	David	bob@example.com

Here, we can see that `Alice` appears twice with the same email (`alice@example.com`), and so does `Bob` with his email (`bob@example.com`). We would like to have only the first occurrence of each unique email address.

Following the solution, this is what we would do:

1. Import the pandas library so that we can work with DataFrames.
2. Create the `dropDuplicateEmails` function.
3. Use the `drop_duplicates` method on the `customers` DataFrame with `subset=['email']`.

When we apply the function `dropDuplicateEmails` to our DataFrame, the `drop_duplicates` method processes the DataFrame as follows:

- It checks the `email` column for duplicate values since we've set `subset=['email']`.
- When it finds the duplicate values `alice@example.com` and `bob@example.com`, it keeps the first occurrence of each (the rows with `customer_id` 1 and 2) and discards the other occurrences (the rows with `customer_id` 3 and 5).

The DataFrame returned by the `dropDuplicateEmails` function will be:

customer_id	name	email
1	Alice	alice@example.com
2	Bob	bob@example.com
4	Charlie	charlie@example.com

So the final DataFrame correctly contains only one record for each email address, and the first customers with those emails have been kept. The duplicate rows have been removed, achieving our desired result. The elegance of this solution lies in its simplicity and the use of Pandas' high-level functionality, which makes the code clean, readable, and efficient.

Python Solution

```python
import pandas as pd  # Importing the pandas library

def dropDuplicateEmails(customers: pd.DataFrame) -> pd.DataFrame:
    """
    Remove duplicate rows from the customers DataFrame based on the 'email' column.

    Parameters:
    customers (pd.DataFrame): DataFrame containing customer data with an 'email' column.

    Returns:
    pd.DataFrame: A new DataFrame without duplicate emails.
    """
    # Use the drop_duplicates method on the 'customers' dataframe,
    # specifying 'email' as the subset to identify duplicates by the 'email' column only.
    unique_customers = customers.drop_duplicates(subset=['email'])

    # Return the resulting DataFrame with duplicates removed.
    return unique_customers
```

Java Solution

```java
import java.util.HashMap;
import java.util.Map;
import java.util.List;
import java.util.ArrayList;
import java.util.Objects;

class Customer {
    String email;
    // Other customer fields can be added here

    // Constructor
    public Customer(String email) {
        this.email = email;
    }

    // Email getter
    public String getEmail() {
        return email;
    }

    // Equals and hashCode methods are overridden to use 'email' field for comparison
    // and to ensure consistent behavior when used as a key in a HashMap.
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Customer)) return false;
        Customer customer = (Customer) o;
        return Objects.equals(email, customer.email);
    }

    @Override
    public int hashCode() {
        return Objects.hash(email);
    }
}

// A toString method for easy printing of customer information.
@Override
public String toString() {
    return "Customer{" +
            "email='" + email + '\'' +
            '}';
}
}

public class DuplicateEmailsRemover {

    /**
     * Remove duplicate rows from the list of customers based on the 'email' field.
     *
     * @param customers (List<Customer>): A list containing customer objects.
     * @return List<Customer>: A new list without duplicate emails.
     */
    public static List<Customer> dropDuplicateEmails(List<Customer> customers) {
        // Use a HashMap to track unique emails.
        Map<String, Customer> uniqueCustomerMap = new HashMap<>();

        // Iterate over the list of customer objects
        for (Customer customer : customers) {
            // If the email has not been seen before, add the customer to the map.
            if (!uniqueCustomerMap.containsKey(customer.getEmail())) {
                uniqueCustomerMap.put(customer.getEmail(), customer);
            }
        }

        // Return the unique customers as a new list (values of the map).
        return new ArrayList<>(uniqueCustomerMap.values());
    }

    // A main method to test the functionality
    public static void main(String[] args) {
        // Create a list of customers with some duplicate emails
        List<Customer> customers = new ArrayList<>();
        customers.add(new Customer("alice@example.com"));
        customers.add(new Customer("bob@example.com"));
        customers.add(new Customer("charlie@example.com"));
        customers.add(new Customer("alice@example.com")); // Duplicate

        // Remove duplicates
        List<Customer> uniqueCustomers = dropDuplicateEmails(customers);

        // Print out the unique customer list
        uniqueCustomers.forEach(System.out::println);
    }
}
```

C++ Solution

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm> // For std::unique and std::stable_sort
#include "DataFrame.h" // Assuming DataFrame is a fictional class that needs to be included

// Define a custom comparison operator
bool compareByEmail(const Customer& first, const Customer& second) {
    return first.getEmail() < second.getEmail();
}

// Define a custom equality operator
bool equalByEmail(const Customer& first, const Customer& second) {
    return first.getEmail() == second.getEmail();
}

// Remove duplicate rows from the customers DataFrame based on the 'email' column.
DataFrame dropDuplicateEmails(DataFrame& customers) {
    // Assuming 'customers' is a DataFrame containing 'Customer' objects
    // and 'Customer' has a method 'getEmail()' to access the 'email' attribute.

    // First, sort the customers by email using the custom comparison operator
    std::stable_sort(customers.begin(), customers.end(), compareByEmail);

    // Then, use the unique algorithm with a custom equality operator
    // unique algorithm will require the duplicates to be next to each other
    // hence the need for the sort operation beforehand.
    auto lastUnique = std::unique(customers.begin(), customers.end(), equalByEmail);

    // Erase the non-unique elements from the container
    customers.erase(lastUnique, customers.end());

    // Return the resulting DataFrame with duplicates removed.
    return customers;
}
```

Typescript Solution

```typescript
interface Customer {
    [key: string]: any; // An index signature to allow any string as a key and its value can be anything
    email: string;      // Ensure that 'email' is always a string
}

/**
 * Remove duplicate rows from the customers array based on the 'email' property.
 *
 * @param customers - Array containing customer objects with an 'email' property.
 * @returns A new array without duplicate emails.
 */
function dropDuplicateEmails(customers: Customer[]): Customer[] {
    const seenEmails = new Set<string>(); // To track already encountered emails

    // Use filter to exclude duplicates
    const uniqueCustomers = customers.filter(customer => {
        // If seenEmails does not have this email, add it and keep the customer
        if (!seenEmails.has(customer.email)) {
            seenEmails.add(customer.email);
            return true;
        }

        // Otherwise, it's a duplicate; exclude it by returning false
        return false;
    });

    // Return the filtered array without duplicate emails
    return uniqueCustomers;
}
```

Time and Space Complexity

The time complexity of the `dropDuplicateEmails` function primarily depends on the `drop_duplicates` method of the pandas DataFrame, which in turn depends on the size of the input DataFrame.

Time Complexity

For pandas `drop_duplicates` method, the time complexity is generally $O(n)$, where n is the number of rows in the DataFrame because it needs to process each row to check for duplicates.

So, the time complexity for the `dropDuplicateEmails` function is also $O(n)$.

Space Complexity

The space complexity for `drop_duplicates` includes the space required to hold the DataFrame and the temporary data structures used to identify duplicates. Typically, it is also $O(n)$ since a new DataFrame is constructed to store the result without duplicates.

Thus, the space complexity of `dropDuplicateEmails` would also be $O(n)$.