

Dark-Patterns Detector

Table of Contents:

1. Introduction:

- Overview of the Chrome Extension
- Purpose and Goals
- Target Audience
- Scope and Limitations

2. Installation Instructions

- Configuration Setup
- Requirements
- Dependencies
- AI/ML Integration

3. Manifest.json file

4. Explanation of AI/ML in the Extension

- Determine Category
- Preprocessing Data
- Training Multinomial Naive Bayes Classifier
- Evaluation Metrics

1-Determine Presence

- Preprocessing Data

- Training Bernoulli Naive Bayes Classifier
- Evaluation Metrics

2-Interact with Data

- Reading Data
- Printing Pattern Category Counts
- Model Testing

5.Explanation of the web-development part

6.Explanation of server

Introduction

● Overview of the Chrome Extension

The Chrome extension is designed to enhance users' browsing experience by detecting and addressing dark patterns commonly found in e-commerce websites. It offers a range of features aimed at promoting transparency, empowering users, and protecting their interests while browsing online shopping platforms.

● Purpose and Goals

The primary purpose of the Chrome extension is to identify and mitigate deceptive design techniques known as dark patterns, which are employed by some e-commerce websites to manipulate user behavior. By detecting and highlighting these dark patterns in real-time, the extension aims to raise awareness among users and empower them to make informed decisions while shopping online.

● Target Audience

The target audience for the Chrome extension includes anyone who engages in online shopping activities, ranging from casual consumers to frequent e-commerce shoppers. It is particularly beneficial for individuals who value transparency, privacy, and ethical design practices in online interactions.

● Scope and Limitations

While the Chrome extension offers valuable functionality in detecting and addressing dark patterns, it's essential to recognize its scope and limitations. The extension focuses primarily on identifying common types of dark patterns prevalent in e-commerce websites. However, it may not detect every instance of deceptive design, and its effectiveness can be influenced by factors such as website complexity and updates to dark pattern techniques.

Getting Started

To get started with the development environment for the project, you'll need to follow these steps:

• Installation Instructions

Install Node.js: Ensure Node.js is installed on your system. You can download it from the official website.

Install MongoDB: Download and install MongoDB, a NoSQL database used for storing data. You can find installation instructions on the MongoDB website.

Install Python: If not already installed, download and install Python from the official website.

Install Flask: Flask is a web framework for Python. You can install it using pip, Python's package manager, by running `pip install Flask` in your terminal.

• Configuration Setup

Configure Express: Express is a web application framework for Node.js. Set up Express in your project by installing it using npm, the Node.js package manager. Run `npm install express` in your project.

Configure Tailwind CSS: Tailwind CSS is a utility-first CSS framework. You can use it by including its CDN link or by installing it as a dependency in your project. Install Tailwind CSS using npm: `npm install tailwindcss`.

Configure Other Dependencies: Install other dependencies listed in your project's package.json file using npm.

● Requirements

Ensure your system meets the following requirements:

- Node.js
- MongoDB
- Python
- Flask
- Internet connection (for installing dependencies)
- Dependencies

Project relies on several dependencies, including:

Ejs: A simple templating language for embedding JavaScript into HTML.

Express: A web application framework for Node.js.

Mongoose: An Object Data Modeling (ODM) library for MongoDB and Node.js.

Nodemon: A utility for automatically restarting the server when changes are made to files.

Pandas: A data manipulation library for Python.

Scikit-learn: A machine learning library for Python.

Matplotlib: A plotting library for Python.

Make sure these dependencies are installed in your project environment using npm or pip, depending on the language.

Explanation of Manifest.json in the Extension

This is a manifest file for a Chrome extension named "Dark-Patterns-Detector". Let's break down its components and explain their functionalities:

1. **Manifest Version** : Indicates the version of the manifest file syntax being used.
2. **Name**: The name of the Chrome extension, which is "Dark-Patterns-Detector".
3. **Version**: The version number of the extension, currently set to "1.2".
4. **Description**: Provides a brief description of the extension's purpose, which includes detecting dark patterns, blocking ads, getting real-time counts, reporting incidents, summarizing terms, and checking for data breaches using DarkPatternDetect.
5. **Icons**: Specifies icons for the extension, particularly a 128x128 pixel icon named "Logo.png".
6. **Action**: Defines the default popup HTML file that is displayed when the extension's icon is clicked.

7. **Content Scripts:** Specifies JavaScript and CSS files to be injected into web pages. These scripts and styles are used to modify the behavior and appearance of web pages to detect dark patterns, block ads, etc. They are injected at the end of the document loading process.

8. **Permissions:** Lists permissions required by the extension to perform certain actions. These include accessing active tabs, managing tabs, accessing storage, and using the declarativeNetRequest API for network requests.

9. **Host Permissions:** Grants access to all URLs, allowing the extension to interact with content on any website.

10. **Web Accessible Resources:** Specifies resources that can be accessed by web pages. In this case, it allows the "report.html" file to be accessed by websites with HTTPS URLs.

11. **Declarative Net Request:** Defines a ruleset for declarative network requests, which is used to block or modify network requests based on predefined rules stored in a "**rules.json**" file.

Overall, this manifest file sets up the necessary components and permissions for the **Dark-Patterns-Detector** Chrome extension to function effectively in detecting dark patterns, blocking ads, and providing other related features.

Explanation of AI/ML in the Extension

Determine Category.py:

Purpose: Trains a model to predict the category of a given text pattern (e.g., "Sneak into Basket").

Key Steps:

- **Imports necessary libraries:** Pandas for data manipulation, sklearn for machine learning, joblib for saving models.
- **Loads data:** Reads the "dark_patterns.csv" file containing pattern strings and their categories.

Preprocesses data:

Focuses on relevant columns ("Pattern String" and "Pattern Category")

- **Handles missing values :**Assigns numeric IDs to categories for easier model handling
- **Vectorized text:** Converts text patterns into numerical representations using TF-IDF (term frequency-inverse document frequency) for model compatibility.
- **Splits data:** Divides data into training and testing sets for model evaluation.
- **Trains model:** Uses a Multinomial Naive Bayes classifier to learn patterns and categories.
- **Evaluates model:** Calculates accuracy on the testing set.
- **Saves model and vectorizer:** Stores them for future use in classifying new patterns.

Determine-presence.py:

Purpose: Trains a model to determine if a given text pattern is a "dark pattern" or not.

Key Steps:

- **Imports libraries:** Similar to "Determine Category.py," with additional libraries for confusion matrix visualization (commented out).
- **Loads data:** Reads both "normie.csv" (non-dark patterns) and "dark_patterns.csv" to create a balanced dataset.
- **Preprocesses data:** Similar to "Determine Category.py," but labels patterns as "Dark" or "Not Dark."
- **Vectorized text:** Uses TF-IDF again.
- **Splits data:** Divides into training and testing sets.
- **Trains model:** Uses a Bernoulli Naive Bayes classifier specifically suited for binary classification (dark or not dark).
- **Evaluates model:** Calculates accuracy and prints it.
- **Saves model and vectorizer:** Stores them for future use.

[Interact_df.py:](#)

Purpose: Provides a simple way to explore the pattern categories and their frequencies in the "dark_patterns.csv" dataset.

Key Steps:

- **Imports Pandas:** For data manipulation.
- **Loads data:** Reads the "dark_patterns.csv" file.
- **Preprocesses data:** Focuses on relevant columns and handles missing values.
- **Prints value counts:** Shows the number of patterns in each category, helping visualize the distribution of patterns.

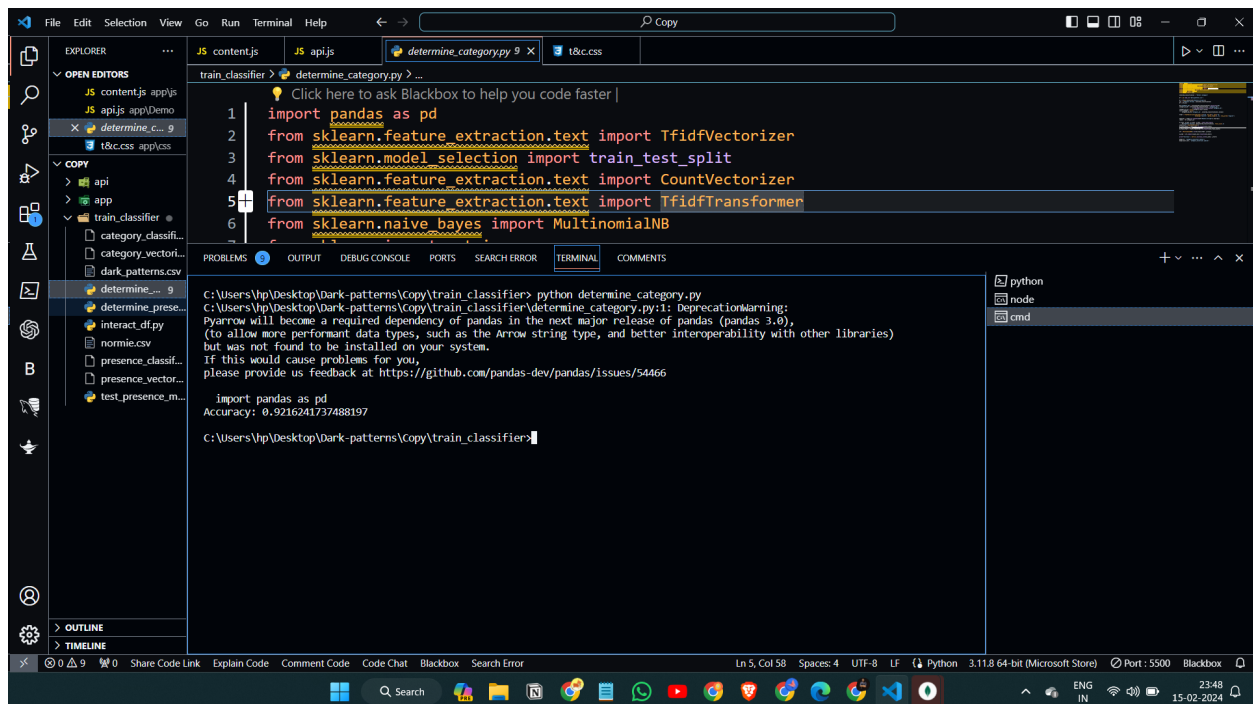
[test_presence_models.py:](#)

Purpose: Experimentation with different machine learning models to potentially improve dark pattern detection accuracy.

Key Steps:

- **Imports libraries:** Similar to previous files, with additional classifiers from sklearn.
- **Loads and preprocess data:** Similar to "Determine-presence.py."
- **Splits data:** Divides into training and testing sets.
- **Creates a list of classifiers:** Includes Naive Bayes, Random Forest, SVM, Decision Tree, SGD, and Logistic Regression.
- **Trains and evaluates each model:** Iterates through the classifiers, trains them on the data, and calculates accuracy scores and confusion matrices.
- **Prints results:** Displays the accuracy of each model, allowing for comparison and potential model selection.

Here are some output of ML model code



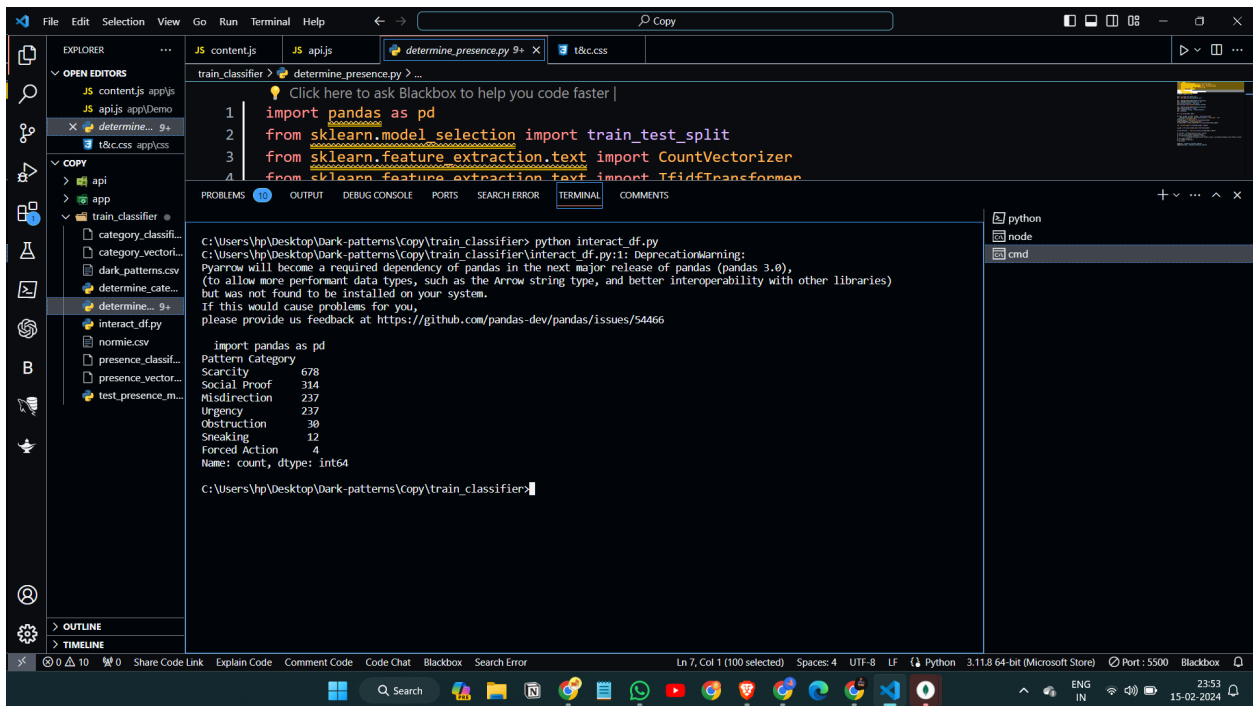
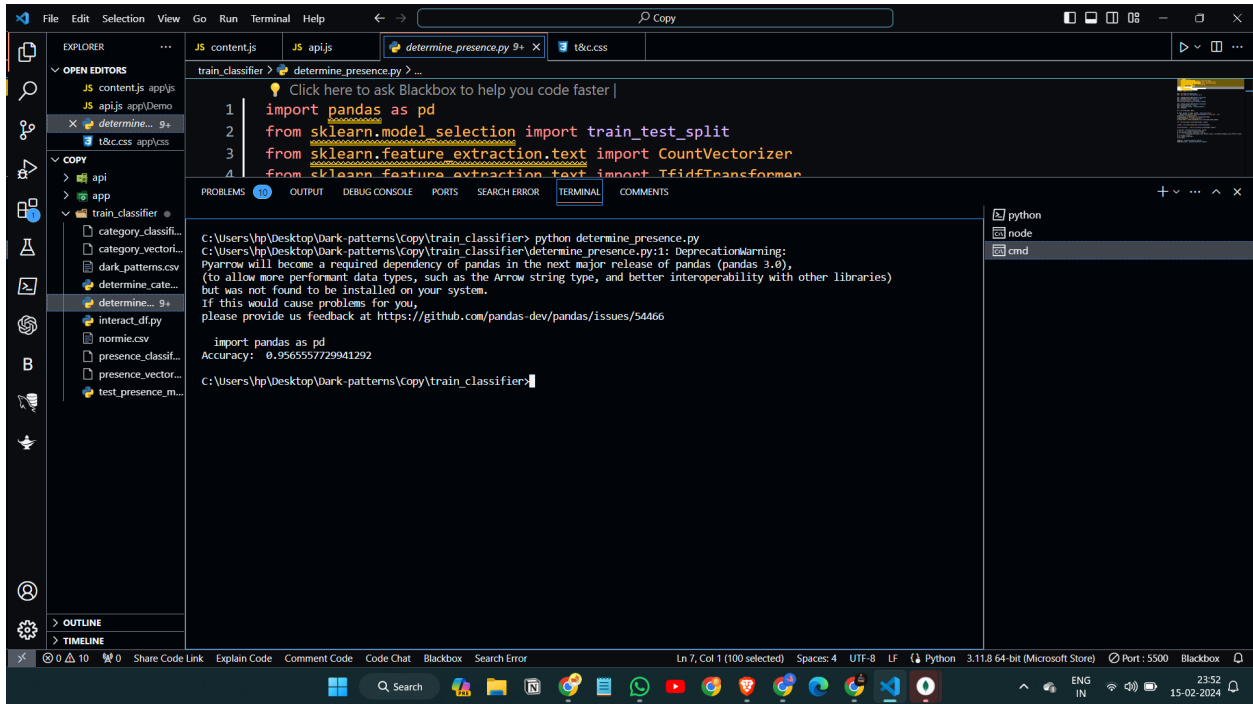
```
1 import pandas as pd
2 from sklearn.feature_extraction.text import TfidfVectorizer
3 from sklearn.model_selection import train_test_split
4 from sklearn.feature_extraction.text import CountVectorizer
5 from sklearn.feature_extraction.text import TfidfTransformer
6 from sklearn.naive_bayes import MultinomialNB

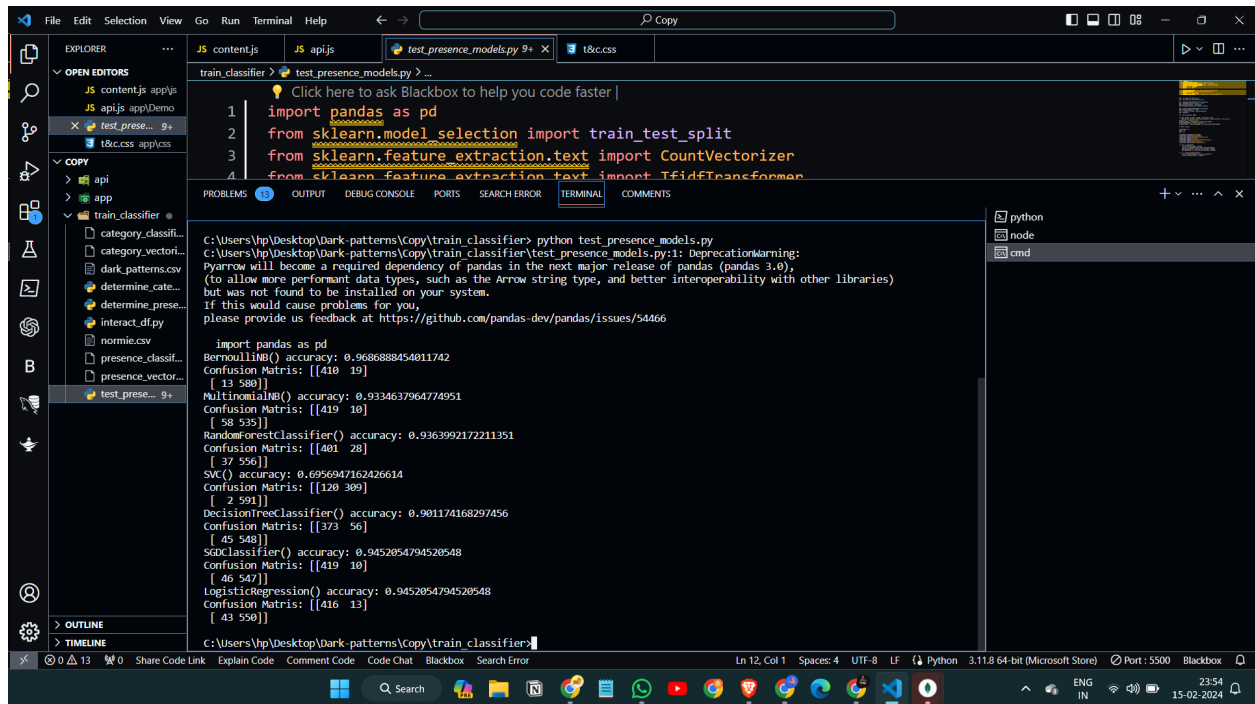
train_classifier > determine_category.py > ...
Click here to ask Blackbox to help you code faster |

C:\Users\hp\Desktop\dark-patterns\Copy\train_classifier> python determine_category.py
C:\Users\hp\Desktop\dark-patterns\Copy\train_classifier\determine_category.py:1: DeprecationWarning:
Pandas will become a required dependency of pandas in the next major release of pandas (pandas 3.0),
(to allow more performant data types, such as the Arrow string type, and better interoperability with other libraries)
but was not found to be installed on your system.
If this would cause problems for you,
please provide us feedback at https://github.com/pandas-dev/pandas/issues/54466

import pandas as pd
Accuracy: 0.9216241737488197

C:\Users\hp\Desktop\dark-patterns\Copy\train_classifier>
```





Explanation of Web-Dev part in the Extension

FILE – Block-Segment

This JavaScript file defines two functions: ``allIgnoreChildren`` and ``segments``.

1. **AllIgnoreChildren:** This function takes an HTML element as input and checks if all of its children are to be ignored based on certain conditions. It iterates through each child of the given element and checks if its tag name (converted to lowercase) is included in an array called `ignoredElements`. If a child's tag name is in the `ignoredElements` array, it skips it and continues to the next child. If all children's tag names are in the ``ignoredElements`` array, it returns `true`, indicating that all children are to be ignored. Otherwise, it returns `false`.

2. **Segments:** This function is a recursive function that segments HTML elements based on certain conditions. It takes an HTML element as input and returns an array of segmented elements.

a. It first checks if the element is falsy (null, undefined, etc.), and if so, it returns an empty array.

b. It then checks if the tag name of the element is not included in the `ignoredElements` array, if it's not a pixel element (`isPixel` function), and if it's shown (`isShown` function).

c. If the above conditions are met, it further checks if the element is a block element (whose tag name is included in the `blockElements` array). If it is a block element and doesn't contain other block

elements (containsBlockElements function), it checks if all of its children are to be ignored (allIgnoreChildren function). If so, it returns an empty array. If not, it recursively calls the segments function for each child of the element and concatenates the results.

d. If the element is a block element but contains text nodes (containsTextNodes function), it returns an array containing only the element.

e. If the element is not a block element, it checks if it contains block elements (containsBlockElements function) but without considering its children. If so, it recursively calls the segment function for each child of the element and concatenates the results.

f. If none of the above conditions are met, it checks if the area of the element relative to the window area is greater than 30%. If so, it recursively calls the segments function for each child of the element and concatenates the results.

g. If none of the conditions are met, it returns an array containing only the element.

Overall, the segments function segments HTML elements based on various conditions and returns an array of segmented elements.

FILE - COMMON.JS

This file contains several utility functions and constants used for manipulating and analyzing HTML elements. Here's a breakdown of each part:

Constants:

BlockElements: Array containing tag names of block-level elements.

ignoredElements: Array containing tag names of elements to be ignored.

Variables:

winWidth and winHeight: Window width and height.

winArea: Area of the window calculated as the product of window width and height.

Functions:

1. Element Size and Position:

getElementArea: Calculates the area of an element based on its bounding rectangle.

getClientRect: Retrieves the bounding rectangle of an element, considering special cases like the <html> element.

2. Element Style:

getBackgroundColor: Retrieves the background color of an element, traversing up the DOM tree if necessary.

3. Randomization

getRandomSubarray: Returns a random subarray of a given array.

elementCombinations: Generates all possible combinations of elements from multiple arrays.

4. DOM Traversal:

getVisibleChildren: Returns visible children of an element based on the `isShown` function.

- **getParents:** Returns all parent elements of a given node.

5. Element Visibility:

isShown: Determines if an element is shown on the screen by checking various conditions like display, visibility, opacity, etc.

isInteractable: Checks if an element is both visible and interactable.

containsTextNodes: Checks if an element contains any non-whitespace text nodes.

isPixel: Determines if an element is a 1x1 pixel element.

- **containsBlockElements:** Checks if an element contains block-level elements.

6. Whitespace Handling:

- **filterText:** Removes whitespace characters from a given text.
- **isWhitespace:** Checks if a given node is a whitespace text node.

These functions and constants are likely used for web scraping, DOM manipulation, or other similar tasks where detailed analysis and manipulation of HTML elements are required.

FILE-CONTENT.JS

Overview

The provided JavaScript snippets are part of browser extensions aimed at identifying and addressing dark patterns on websites. The first snippet involves scraping web pages for dark patterns, analyzing the content, and updating counts of different pattern types. It includes functionality for highlighting elements and responding to user interactions.

The second snippet focuses on creating a dynamic HTML interface within the browser extension. It enables users to control the visibility and position of icons, drag elements around the screen, and access reporting, rating, and data breach functionalities through buttons.

Lastly, the third snippet defines a function named `count`, which generates HTML content for a slider displaying information about various dark patterns. It includes functionality for navigating the slider, attaching event listeners for user interaction, and dynamically generating CSS styles for styling the slider components.

Together, these snippets contribute to a comprehensive browser extension for detecting and addressing dark patterns on websites, providing users with tools to analyze, report, and interact with detected patterns.

Part -1

The provided code in this file is a JavaScript snippet for a browser extension or plugin, likely for analyzing web pages for the presence of certain "dark patterns" – design elements or techniques that manipulate users into taking certain actions.

1. **Constants and Data Structures:** It defines an endpoint for a web server (**endpoint**) and two objects (**descriptions** and **categoriesCount**) used for mapping dark pattern names to descriptions and counting occurrences of each type, respectively.

2. **scrape():** This function appears to be the main functionality of the code. It gathers text content from the DOM, sends it to the specified endpoint using a POST request, receives the analyzed data back, highlights elements on the page based on the analysis, counts occurrences of each dark pattern type, and updates the count.

3. **Highlighting Functions:** These functions (**highlight()** and **removeHighlight()**) are responsible for adding and removing visual highlights to elements on the page to indicate the presence of dark patterns.

4. **Event Listeners:** There's an event listener attached to a button or element with the class `.two`, triggering the `scrape()` function when clicked. There's also an event listener for a button with the class `.three`, toggling the visibility of an element with the id `container`.

5. **Message Passing:** The code sets up message listeners (**`chrome.runtime.onMessage.addListener`**) to receive messages from the browser extension or other scripts, presumably for triggering the analysis (**`analyze_site`**) and updating counts (**`popup_open`**).

6. **Flag for Scraping:** It uses a flag **`isScraped`** to ensure that the scraping function is only executed once until explicitly reset.

7. Additional Functionality: It includes a function **removeHighlights()** for removing highlights and resetting the state after the analysis is done or canceled.

PART -2

The provided code creates a dynamic HTML interface with buttons and icons, and implements functionality for toggling visibility, dragging, reporting, rating, and data breach reporting. Here's a summary of each part:

1. **createDynamicHTML():** This function dynamically creates HTML elements including buttons and icons, and appends them to the document body. Each icon is wrapped in a **<div>** element with specific classes.

2. **Visibility Control:** The **showIcons()**, **hideIcons()**, and **toggleIconsVisibility()** functions control the visibility of the icons. When the main button (with class **.one**) is clicked, it toggles the visibility of the icons with a sliding effect.

3. **Draggable Functionality:** The **enableDragging()**, **disableDragging()**, and **dragElement()** functions enable dragging behavior for the main div (**.maindiv**) when double-clicked. The main div can be dragged around the screen. It also listens for the right arrow key to enable dragging.

4. **Reporting, Rating, and Data Breach Functionality:** Clicking on the respective buttons (**six**, **seven**, and **five**) opens a new window/tab to the corresponding URLs for reporting, rating, and reporting a data breach, respectively.

Overall, this code creates a dynamic interface with various interactive features, providing users with options for reporting, rating, and addressing data breaches, as well as allowing them to manipulate the visibility and position of the interface elements.

PART -3

Here's a breakdown of what the count function does:

1. It creates HTML content for a slider with information about various dark patterns.
2. It defines functions for sliding left and right within the slider, as well as showing a specific item.
3. It attaches event listeners for arrow key navigation and clicking on items in the list to trigger the corresponding functions.
4. It dynamically generates CSS styles for styling the slider and its components.
5. It appends the generated HTML content and CSS styles to the document.

SERVER

Index.js

This is a Node.js Express server application that handles routes for a feedback form, data breach checking functionality, and other related pages. Let's break down the code and explain its functionality:

1. **Dependencies:** The application requires express, path, mongoose (for MongoDB), and likely fetch for making HTTP requests.

2. **Setting Up Express App:**

- The Express app is created.
- EJS is set as the view engine, and the views directory is configured.
- Static files are served from the public directory.
- URL-encoded data is parsed.

3. **Database Connection:** The server connects to a MongoDB database named **Feedback**.

4. **Defining Mongoose Schema:**

- A schema is defined for the **feed** collection, presumably for storing feedback data.

5. **Routes:**

- **GET /rating:** Renders a form for users to provide ratings and feedback.
- **GET /rating/thanks:** Renders a thank you page after submitting feedback.
- **POST /rating:** Handles form submission for feedback. Data is saved to the database.
- **GET/data-breach:** Renders a page for checking data breaches.
- **POST /data-breach:** Handles submission of an email address for data breach checking. This route makes a request to an external API (**data-breach-checker.p.rapidapi.com**) to check for breaches associated with the provided email.
- **GET /reporting:** Renders a reporting page.

6. Server Initialization: The server listens on port 8080.

This application benefits users and the service provider in several ways:

- **Feedback Collection:** Users can submit ratings and feedback, helping the service provider improve their services.
- **Data Breach Checking:** Users can check if their email addresses have been involved in data breaches, enhancing their awareness of potential security risks.
- **Reporting:** Provides a platform for users to report any issues or concerns.

- **Database Storage:** Feedback data is stored in a MongoDB database, allowing the service provider to analyze and act upon user feedback.
- **Thank You Page:** A personalized touch to acknowledge feedback submissions.
- **Server Logging:** Logs server activities to the console, aiding in monitoring and debugging.

Overall, this application facilitates user interaction, feedback collection, data breach checking, and reporting, contributing to a better user experience and service improvement.