AI Powered Task Management Application

By

Veluri Niteesh Naidu

BTech CSE (AIML)

GITAM University (Bengaluru)

Introduction

Overview:

The **AI-Powered Task Management Application** is an intelligent task organizer designed to help users effectively manage their daily tasks using artificial intelligence. Unlike traditional task management tools, this application integrates AI-based automation, making task organization smarter, more efficient, and highly adaptive. By incorporating **task prioritization, smart reminders, and NLP-based task categorization**, the system streamlines workflow, ensuring that users stay on top of their schedules with minimal effort.

This project provides an enhanced task management experience by intelligently analysing task urgency, importance, and categorization, ensuring users focus on what matters most. With an easy-to-use interface and AI-driven features, this application improves overall productivity and efficiency in managing tasks.

Purpose and Objectives:

The primary goal of this application is to assist users in organizing their tasks efficiently while leveraging AI-powered automation. The key objectives of this project are:

- **Enhance Productivity:** Automate task management to reduce manual effort and increase efficiency.
- **Task Prioritization:** Assign priorities to tasks based on urgency and importance, ensuring users focus on high-impact activities.
- **Smart Reminders:** Provide timely notifications with motivational messages to keep users engaged and on track.
- **Task Categorization:** Utilize **Natural Language Processing (NLP)** to automatically classify tasks into relevant categories.
- **Progress Tracking:** Offer users insights into their task completion trends using **dynamic visualizations and charts**.

Key Features:

- **Task Management:** Users can **add, edit, delete, and organize tasks** with associated due dates.

- **AI-Based Task Prioritization:** The application assigns priority scores to tasks based on urgency, importance, and deadlines.
- **Smart Reminders with Motivational Messages:** The system sends **desktop notifications** with reminders and motivational lines related to the task description.
- **Progress Tracker:** Users can monitor their productivity through **interactive charts** and detailed insights.
- **AI-Powered Task Categorization:** The application automatically classifies tasks using **NLP techniques**, helping users better organize and search for tasks.

Technologies Used:

- **Frontend:** HTML, CSS, JavaScript, Bootstrap
- **AI/ML:** TensorFlow.js (for NLP-based categorization and task prioritization)
- **Storage:** Local Storage API (for saving tasks persistently in the browser)
- **Notifications:** Web Notifications API (for sending reminders)
- **Data Visualization:** Chart.js (for tracking progress and visualizing task trends.

Benefits of AI Integration:

The integration of AI into this application significantly improves its functionality and usability:

- **Automation:** Reduces the need for manual input by **intelligently categorizing and prioritizing** tasks.
- **Efficiency:** Saves time by **suggesting important tasks** and notifying users at the right moment.
- **Data-Driven Insights:** Provides **analytics on user productivity** and task completion trends.
- **Personalization:** Adapts to **user behaviour**, offering tailored recommendations and reminders based on past task patterns.
- **Improved Time Management:** Helps users allocate time effectively by identifying **critical** and **time-sensitive** tasks.

How it works:

1. **User Inputs Tasks**: The user enters a task with a description, due date, and optional priority level.

2. **AI-Based Prioritization**: The system analyses urgency, importance, and deadlines to assign a **priority score**.
3. **NLP Categorization**: Tasks are automatically grouped into categories based on **keywords, context, and user patterns**.
4. **Smart Reminders**: The system triggers **notifications with motivational messages** based on the **task's priority and deadline**.
5. **Progress Tracking**: A **dynamic chart** visualizes completed and pending tasks, helping users track their productivity trends.
6. **Adaptive Learning**: Over time, the system refines task categorization and prioritization based on **user behaviour and task completion patterns**.

This AI-Powered Task Management Application transforms the way users handle their daily responsibilities by making task organization seamless and automated. With **intelligent reminders, AI-based prioritization, and NLP-driven categorization**, this tool provides a **highly personalized** and **efficient** approach to productivity management.

Installation & Setup

Pre-requisites:

Before setting up the application, ensure you have the following installed:

- A modern web browser (Chrome, Firefox, Edge, etc.)
- A code editor (VS Code, Sublime Text, etc.)
- Basic knowledge of JavaScript, HTML, and CSS

Downloading the Project:

You can download the project files from the following source:

- **GitHub Repository**: [Insert Link Here]
- Alternatively, you can manually create the required files and structure as described below.

Project Structure:

```
AI_Powered_Task_Manager/
|-- datasets/  (Contains training datasets for NLP)
|-- images/    (Stores UI-related images)
|-- js/
|   ├── app.js
|   ├── main-container.js
|   ├── task_prioritization2.js
|   ├── task_categorization.js
|   ├── progressTracker.js
|   ├── sign_in.js
|   ├── sign_up.js
|-- models/    (Contains ML models for task categorization)
|-- index.html
|-- sign_in.html
|-- sign_up.html
|-- sign_in.css
|-- sign_up.css
|-- style.css
```

Setting up the Project:

1. **Clone the Repository**

```
git clone [repository-link]
cd AI_Powered_Task_Manager
```

2. **Open in Browser**
   a. If using VS Code, install the **Live Server** extension and run the project.
   b. Otherwise, open `index.html` directly in a browser.
3. **Include Required Libraries** Add the following CDN links inside your HTML file to include external libraries:
   <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
   <script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs"></script>

4. **Set Up User Authentication**

- Users can sign up via `sign_up.html` and log in via `sign_in.html`.
- The authentication system stores credentials securely in Local Storage.
- The logic for user authentication is handled in `sign_in.js` and `sign_up.js`.

Running the Application:

Once the setup is complete, open `index.html` in a browser. You can now:

- **Sign up or log in** to access personalized tasks.
- **Add tasks** with descriptions and due dates.
- **View AI-based task prioritization**.
- **Receive smart reminders**.
- **Track progress** with dynamic charts.

With the setup complete, you're now ready to explore the AI-powered task management features!

Project Features (Detailed Explanation):

1. Basic Task Management:

This section allows users to perform essential CRUD (Create, Read, Update, Delete) operations on tasks. Users can:

- **Add Tasks:** Input a task description, due date, and optional priority level.
- **Edit Tasks:** Modify existing tasks if needed.
- **Delete Tasks:** Remove tasks permanently when they are no longer needed.
- **Mark as Completed:** Users can mark a task as completed, which updates the progress tracker.

Implementation:

- Tasks are stored in the browser's **Local Storage**, ensuring they persist even after a page refresh.
- The UI updates dynamically using JavaScript DOM manipulation.

2. Progress Tracker (Weekly Tracking with Chart.js):

The application tracks user progress over time by analysing completed and pending tasks. A **Chart.js**-powered graph visually represents:

- **Daily Task Completion Trends**: Displays how many tasks were completed each day.
- **Pending vs. Completed Tasks**: Provides a real-time view of task progress.
- **Week Progress Overview**: Helps users evaluate their productivity trends.

Implementation:

- The number of completed tasks is recorded daily.

- Chart.js updates dynamically whenever a task is added or marked as completed.

3. Task Prioritization with Smart remainders (with Motivation):

This feature ensures users stay on track with their tasks by sending timely notifications, along with motivational quotes based on task descriptions.

Key Features:

- **AI-Powered Prioritization**: The system determines which tasks need immediate attention based on urgency and deadline.
- **Web Notifications API**: Sends real-time alerts to remind users about upcoming or high-priority tasks.
- **Motivational Messages**: Each reminder is paired with an inspiring quote to boost user productivity.

Implementation:

- The priority score is calculated using a combination of **deadline, urgency, and importance**.
- Notifications are scheduled and sent accordingly.

4. Natural Language Understanding for Task Categorization (NLP-based Task Grouping):

The system uses **Natural Language Processing (NLP)** to automatically categorize tasks based on their descriptions.

Key Features:

- **AI-Based Categorization**: Tasks are grouped into categories such as Work, Personal, Shopping, Study, etc.
- **Keyword Extraction**: The AI analyzes task descriptions to determine the most relevant category.
- **Adaptive Learning**: Over time, the system improves categorization accuracy based on user input.

Implementation:

- **TensorFlow.js** is used for text vectorization and NLP processing.
- **Predefined keyword lists** and machine learning models classify tasks into different categories.

Java Script Code Explanations

1. arrow.js (arrow disappearing based on window height):

```
// Get the down arrow element
const downArrow = document.querySelector('.header-down-arrow');

// Add an event listener to handle the scroll event
window.addEventListener('scroll', function () {
    const threshold = window.innerHeight / 10;
    if (window.scrollY > threshold) {
        downArrow.classList.add('hidden'); // Add the hidden class to fade out
    } else {
        downArrow.classList.remove('hidden'); // Remove the hidden class to fade in
    }
});

// Optional: Add a fade-out animation for smoother effect
document.styleSheets[0].insertRule(`
    .hidden {
        opacity: 0;
        transition: opacity 0.5s ease;
    }
`, 0);
```

Code Explanation: Scroll-Dependent Down Arrow Visibility

This JavaScript code manages the visibility of a **down arrow** on a webpage. The arrow disappears when the user scrolls down and reappears when they scroll back up.

1. Selecting the Down Arrow Element

```
// Get the down arrow element
const downArrow = document.querySelector('.header-down-arrow');
```

Explanation:

- The `document.querySelector()` method selects the **element with the class** `.header-down-arrow`.

- The element is stored in the `downArrow` variable for later use.

Purpose:

- This allows us to **manipulate the arrow's visibility** dynamically.

2. Adding a Scroll Event Listener

```
// Add an event listener to handle the scroll event
window.addEventListener('scroll', function () {
```

Explanation:

- The `window.addEventListener()` method **listens for a scroll event** on the webpage.
- Whenever the user **scrolls up or down**, the function inside the event listener executes.

Purpose:

- Detects when the user scrolls to trigger the visibility change of the down arrow.

3. Setting a Scroll Threshold

```
const threshold = window.innerHeight / 10;
```

Explanation:

- `window.innerHeight` retrieves the **height of the visible screen area**.
- Dividing by `10` sets the **scroll threshold** to **10% of the screen height**.

Purpose:

- The arrow should disappear only **after a slight scroll** (not immediately).

4. Checking Scroll Position and Hiding the Arrow

```
if (window.scrollY > threshold) {
```

```
    downArrow.classList.add('hidden'); // Add the hidden class
to fade out
} else {
    downArrow.classList.remove('hidden'); // Remove the hidden
class to fade in
}
```

Explanation:

- `window.scrollY` **gets the number of pixels scrolled down**.
- If the user **scrolls past the threshold**:
  - `downArrow.classList.add('hidden')` → **Hides the arrow** by adding the `.hidden` CSS class.
- If the user **scrolls back up**:
  - `downArrow.classList.remove('hidden')` → **Shows the arrow** by removing the `.hidden` class.

Purpose:

- Enables **dynamic fading** of the arrow based on scrolling behaviour.

5. Adding a Smooth Fade-Out Effect

```
// Optional: Add a fade-out animation for smoother effect
document.styleSheets[0].insertRule(`
    .hidden {
        opacity: 0;
        transition: opacity 0.5s ease;
    }
`, 0);
```

Explanation:

- The `document.styleSheets[0].insertRule()` method **inserts a new CSS rule dynamically**.
- The `.hidden` class:
  - `opacity: 0;` → **Makes the arrow invisible**.
  - `transition: opacity 0.5s ease;` → **Adds a smooth fading effect** over `0.5 seconds`.

Purpose:

- Ensures the arrow **fades out smoothly** instead of disappearing instantly.

2. front_page_animation.js (for front page animation):

```
document.addEventListener("DOMContentLoaded", function () {
    const parts = [
        ...document.getElementsByClassName("info-and-image"),
        ...document.getElementsByClassName("basicfeatures-and-image"),
        ...document.getElementsByClassName("progresstracking-and-image"),
        ...document.getElementsByClassName("customizepriorities-and-image"),
        ...document.getElementsByClassName("aifeatures-and-image"),
        ...document.getElementsByClassName("btn"),

    ];

    const observer = new IntersectionObserver(
        (entries, observer) => {
            entries.forEach(entry => {
                if (entry.isIntersecting) {
                    // Add an animation class when the container enters the
viewport
                    entry.target.classList.add("animate");
                    // Stop observing once the animation is triggered
                    observer.unobserve(entry.target);
                }
            });
        },
        {
            threshold: 0.07,
        }
    );

    // Observe each part
    parts.forEach(part => {
        observer.observe(part);
    });
});
```

Code Explanation: Scroll-Based Animation Using Intersection Observer

This JavaScript code **adds animations** to different sections of a webpage when they enter the viewport. It ensures that each section **animates only once** when it becomes visible to the user.

1. Running the Script After the Page Loads

```
document.addEventListener("DOMContentLoaded", function () {
```

Explanation:

- The `DOMContentLoaded` event ensures the script runs **only after** the full HTML document is loaded.
- This prevents errors from selecting elements that **haven't loaded yet**.

Purpose:

- Ensures the code **only executes when the page is fully ready**.

2. Selecting Elements to Animate

```
const parts = [
    ...document.getElementsByClassName("info-and-image"),
    ...document.getElementsByClassName("basicfeatures-and-
image"),
    ...document.getElementsByClassName("progresstracking-and-
image"),
    ...document.getElementsByClassName("customizepriorities-
and-image"),
    ...document.getElementsByClassName("aifeatures-and-
image"),
    ...document.getElementsByClassName("btn"),
];
```

Explanation:

- `document.getElementsByClassName()` **selects all elements** with the specified class.
- The **spread operator (`...`)** converts the **HTMLCollection** into an **array** for easy iteration.
- The `parts` array **stores all elements** that should have animations.

Purpose:

- Collects **all sections and buttons** that need animations when they appear in the viewport.

3. Creating an Intersection Observer

```
const observer = new IntersectionObserver(
    (entries, observer) => {
```

Explanation:

- `IntersectionObserver` **monitors** when elements enter or leave the viewport.
- `entries` contains **all observed elements** and their visibility status.

Purpose:

- Tracks elements to determine **when to trigger animations**.

4. Checking If an Element Is Visible

```
entries.forEach(entry => {
    if (entry.isIntersecting) {
        // Add an animation class when the container enters
the viewport
        entry.target.classList.add("animate");
        // Stop observing once the animation is triggered
        observer.unobserve(entry.target);
    }
});
```

Explanation:

- **Loop through** all observed elements (`entries`).
- If `entry.isIntersecting === true`, the element is **now visible** in the viewport.
- `entry.target.classList.add("animate")`:
    - Adds the `.animate` CSS class to **trigger animations**.
- `observer.unobserve(entry.target)`:
    - **Stops observing** the element **after animation is applied**.

Purpose:

- Ensures **animations only trigger once** when the element first appears on screen.

5. Setting the Visibility Threshold

```
{
    threshold: 0.07,
}
```

Explanation:

- `threshold: 0.07` means the animation **triggers when 7%** of the element is visible.

Purpose:

- Ensures **animations start early** instead of waiting for the whole element to appear.

6. Observing Each Element

```
// Observe each part
parts.forEach(part => {
    observer.observe(part);
});
```

Explanation:

- `parts.forEach(part => observer.observe(part));`
    - Loops through each element and **activates the observer**.

Purpose:

- Ensures **every targeted section** is monitored for animations.

Final Summary

- **On page load**, the script collects all elements that need animations.

- The **Intersection Observer** detects when an element enters the viewport.
- The `.animate` class is added to **trigger animations** when the section becomes visible.
- The observer **stops tracking** the element after the animation is applied.
- This creates **smooth animations that occur only once per element**.

3. loading.js (for spinner when loading):

```
// Prepare the body tag by adding a "js-loading" class
document.body.classList.add("js-loading");

// Optionally, show a loading spinner
const spinner = document.createElement('div');
spinner.className = 'loading-spinner';
document.body.appendChild(spinner);

// Listen for when everything has loaded
window.addEventListener('DOMContentLoaded', showPage, false);

function showPage() {
    // Remove the "js-loading" class
    document.body.classList.remove("js-loading");
    // Remove the spinner
    spinner.remove();
    // Scroll to the top of the page after loading
    window.scrollTo(0, 0);
}
```

Understanding the JavaScript Code for Page Loading Effect

This JavaScript code improves the user experience by displaying a loading effect while the webpage is loading and removing it once everything is fully loaded. Let's go through it step by step.

Step 1: Adding a "js-loading" Class to the Body

`document.body.classList.add("js-loading");`

- This line adds a CSS class named `"js-loading"` to the `<body>` tag.
- This class can be used to apply CSS styles such as hiding content or showing a loading animation before the page fully loads.

Step 2: Creating and Adding a Loading Spinner

```
const spinner = document.createElement('div');
spinner.className = 'loading-spinner';
document.body.appendChild(spinner);
```

- A new `<div>` element is created and assigned the class `"loading-spinner"`.
- The class `"loading-spinner"` can be styled using CSS to show a loading animation.
- The `appendChild(spinner)` method adds the spinner to the `<body>` of the webpage, making it visible.

Step 3: Waiting for the Page to Load Completely

```
window.addEventListener('DOMContentLoaded', showPage, false);
```

- This line adds an event listener that waits for the `"DOMContentLoaded"` event.
- `"DOMContentLoaded"` fires when the HTML document is completely loaded and parsed, without waiting for images or stylesheets.
- When the page is ready, the `showPage` function is called.

Step 4: Removing the Loading Effect After the Page Loads

```
function showPage() {
    // Remove the "js-loading" class
    document.body.classList.remove("js-loading");

    // Remove the spinner
    spinner.remove();

    // Scroll to the top of the page after loading
    window.scrollTo(0, 0);
}
```

- `document.body.classList.remove("js-loading");`
  - Removes the `"js-loading"` class from the `<body>`, making the page visible again.
- `spinner.remove();`
  - Removes the loading spinner from the page.

- `window.scrollTo(0, 0);`
    - Ensures the page starts at the top after loading, preventing any unwanted scroll positions.

Final Notes

- This script helps in showing a smooth loading effect while the page loads.
- The `"loading-spinner"` class should be styled using CSS to display a spinner animation.
- Using `DOMContentLoaded` ensures that the script runs at the right time, improving user experience.

4.status.js (for knowing the status of the tasks in the week):

```
const completedTasksNumber = document.getElementById('completedTasks');
const pendingTasksNumber = document.getElementById('pendingTasks');
const averageProgress = document.getElementById('averageProgress');

// Helper function for animating numbers
function animateNumber(element, start, end, duration, isPercentage = false) {
    let startTime = null;

    function animationStep(currentTime) {
        if (!startTime) startTime = currentTime;
        const progress = Math.min((currentTime - startTime) / duration, 1);
        const currentValue = Math.floor(progress * (end - start) + start);
        element.textContent = isPercentage
            ? `${currentValue}%`
            : currentValue;
        if (progress < 1) {
            requestAnimationFrame(animationStep);
        }
    }

    requestAnimationFrame(animationStep);
}

function TasksStatus() {
    let completedTasks = 0;
    let pendingTasks = 0;
    let totalProgress = 0;

    // Get tasks and weekly progress data from localStorage
    let tasks = JSON.parse(localStorage.getItem('tasks')) || [];
```

```
    let progress = JSON.parse(localStorage.getItem('weekProgress')) || [];

    // Count completed and pending tasks
    if (tasks) {
        tasks.forEach(task => {
            if (task.startsWith('✔')) {
                completedTasks++;
            } else {
                pendingTasks++;
            }
        });
    }

    // Calculate total progress based on completed tasks in the current week
    if (tasks.length > 0) {
        // Calculate percentage of tasks completed in the week
        totalProgress = (completedTasks / tasks.length) * 100;
    } else {
        totalProgress = 0;  // If no tasks exist, progress is 0%
    }

    // Animate the numbers for completed tasks, pending tasks, and total progress
    animateNumber(completedTasksNumber, 0, completedTasks, 2000);
    animateNumber(pendingTasksNumber, 0, pendingTasks, 2000);
    animateNumber(averageProgress, 0, Math.round(totalProgress), 2000, true);
}

TasksStatus();
```

Understanding the JavaScript Code for Task Progress Tracking

This JavaScript code dynamically updates and animates the number of completed tasks, pending tasks, and average progress using values stored in `localStorage`.

Step 1: Selecting HTML Elements

const completedTasksNumber = document.getElementById('completedTasks');
const pendingTasksNumber = document.getElementById('pendingTasks');
const averageProgress = document.getElementById('averageProgress');

- These lines select three HTML elements where the updated values will be displayed:

- `completedTasksNumber`: Displays the number of completed tasks.
- `pendingTasksNumber`: Displays the number of pending tasks.
- `averageProgress`: Displays the total progress percentage.

Step 2: Creating a Function to Animate Numbers

```
function animateNumber(element, start, end, duration, isPercentage = false)
{
    let startTime = null;

    function animationStep(currentTime) {
        if (!startTime) startTime = currentTime;
        const progress = Math.min((currentTime - startTime) / duration, 1);
        const currentValue = Math.floor(progress * (end - start) + start);
        element.textContent = isPercentage
            ? `${currentValue}%`
            : currentValue;
        if (progress < 1) {
            requestAnimationFrame(animationStep);
        }
    }

    requestAnimationFrame(animationStep);
}
```

How It Works:

- This function gradually updates an element's value from `start` to `end` over a given `duration` (milliseconds).
- It uses `requestAnimationFrame()` for smooth animations.
- The `isPercentage` parameter determines whether the displayed value should have a percentage (%) sign.
- The animation stops once the `progress` reaches `1` (100% of the duration).

Step 3: Defining the Task Status Function

```
function TasksStatus() {
```

```
let completedTasks = 0;
let pendingTasks = 0;
let totalProgress = 0;
```

- The function `TasksStatus()` initializes three variables:
  - `completedTasks`: Stores the number of completed tasks.
  - `pendingTasks`: Stores the number of pending tasks.
  - `totalProgress`: Stores the average progress percentage.

Step 4: Retrieving Data from Local Storage

```
let tasks = JSON.parse(localStorage.getItem('tasks')) || [];
let progress = JSON.parse(localStorage.getItem('weekProgress')) || [];
```

- It fetches the task data from `localStorage`:
  - `"tasks"` contains a list of task names with completed ones marked as `"✔Task Name"`.
  - `"weekProgress"` stores weekly progress data (not used in this function).
- If no data is found, an empty array (`[]`) is used as a fallback to avoid errors.

Step 5: Counting Completed and Pending Tasks

```
if (tasks) {
    tasks.forEach(task => {
        if (task.startsWith('✔')) {
            completedTasks++;
        } else {
            pendingTasks++;
        }
    });
}
```

- Loops through the `tasks` array:
  - If a task name starts with `"✔`, it is considered **completed** and counted.
  - Otherwise, it is counted as a **pending task**.

Step 6: Calculating Total Progress Percentage

```
    if (tasks.length > 0) {
        totalProgress = (completedTasks / tasks.length) * 100;
    } else {
        totalProgress = 0;
    }
```

- The **total progress** is calculated as:
- $$\text{Total Progress} = \left(\frac{\text{Completed Tasks}}{\text{Total Tasks}} \right) \times 100$$
- If there are no tasks, the progress is set to `0%`.

Step 7: Animating the Numbers on the Page

```
    animateNumber(completedTasksNumber, 0, completedTasks, 2000);
    animateNumber(pendingTasksNumber, 0, pendingTasks, 2000);
    animateNumber(averageProgress, 0, Math.round(totalProgress), 2000, true);
}
```

- Calls the `animateNumber()` function for:
  - `completedTasksNumber`: Displays the number of completed tasks.
  - `pendingTasksNumber`: Displays the number of pending tasks.
  - `averageProgress`: Displays total progress as a percentage (`isPercentage = true`).
- The animation runs for `2000ms` (2 seconds).

Step 8: Running the Function

```
TasksStatus();
```

- Calls `TasksStatus()` immediately when the script runs, updating the numbers based on stored data.

Final Notes

- This script dynamically calculates and displays task progress with smooth animations.
- The "✔" symbol is used to mark completed tasks.
- Ensure `localStorage` contains valid `"tasks"` data for accurate results.
- You can modify the duration in `animateNumber()` for faster/slower animations.

5.functions2.js (for knowing the status of the tasks in the week):

```
const mainPart = document.getElementById("main-part");
const alertPlaceholder = document.getElementById('success-alert');
const dateInput = document.getElementById('task-date');
const taskInput = document.getElementById('floatingInput');
const taskListContainer = document.getElementById('task-list-container');
let selectedDate = '';

const appendAlert = (message, type) => {
  const wrapper = document.createElement('div');
  wrapper.innerHTML = [
    `<div class="alert alert-${type} alert-dismissible container" role="alert">`,
    `   <div>${message}</div>`,
    '   <button type="button" class="btn-close" data-bs-dismiss="alert" aria-
label="Close"></button>',
    '</div>'
  ].join('');
  const alertElement = wrapper.firstChild;
  mainPart.prepend(alertElement);
  setTimeout(() => {
    alertElement.remove();
  }, 2000);
};

const getTasksFromLocalStorage = () => {
  return JSON.parse(localStorage.getItem('tasks')) || [];
};

const saveTasksToLocalStorage = (tasks) => {
  localStorage.setItem('tasks', JSON.stringify(tasks));
};

const scheduleWeeklyCleanup = () => {
  const today = new Date();
  /* console.log(today) */

  // Calculate the start of the current week (Sunday at 00:00:00)
  const startOfWeek = new Date(today);
  const currentDay = today.getDay(); // Sunday = 0, Monday = 1, ..., Saturday = 6
  const offset = -currentDay; // Move to the previous Sunday
  startOfWeek.setDate(today.getDate() + offset);
```

```
  startOfWeek.setHours(0, 0, 0, 0);
  /* console.log(startOfWeek) */

  // Calculate time until the next cleanup (end of this Saturday at 23:59:59)
  const endOfWeek = new Date(startOfWeek);
  endOfWeek.setDate(startOfWeek.getDate() + 6);
  endOfWeek.setHours(23, 59, 59, 999);
  /* console.log(endOfWeek) */

  /* const timeUntilNextSaturday = endOfWeek.getTime() - today.getTime();
  console.log(timeUntilNextSaturday)
  console.log(startOfWeek); */
  removeOldTasksFromLocalStorage(startOfWeek);

  /* // Schedule cleanup
  if (timeUntilNextSaturday > 0) {
    setTimeout(() => {
      removeOldTasksFromLocalStorage(startOfWeek);
      scheduleWeeklyCleanup(); // Schedule the next cleanup
    }, timeUntilNextSaturday);
  } */
};

const removeOldTasksFromLocalStorage = (startOfWeek) => {
  const tasks = getTasksFromLocalStorage();
  // Filter tasks based on their assigned date being on or after the start of the
week
  const updatedTasks = tasks.filter(task => {
    const [taskDescription, taskDate] = task.split(' - ');

    const currentYear = new Date().getFullYear(); // Get the current year

    // Map of month names to numbers
    const monthMap = {
      "Jan": 0, "Feb": 1, "Mar": 2, "Apr": 3, "May": 4, "Jun": 5,
      "Jul": 6, "Aug": 7, "Sep": 8, "Oct": 9, "Nov": 10, "Dec": 11
    };

    // Split the date string into day and month parts
    const [day, month] = taskDate.split(" "); // ["05", "Dec"]

    // Format the date as yyyy-mm-dd
    const formattedDate = `${currentYear}-${monthMap[month]+1}-${day.padStart(2,
"0")}`;

    // Create a Date object
    const taskDateObj = new Date(formattedDate)
    /* console.log(taskDateObj) */

    return taskDateObj >= startOfWeek || (!taskDescription.startsWith('✓') &&
taskDateObj < startOfWeek);
  });
  /* console.log(updatedTasks) */
  // Save the updated list back to localStorage
  saveTasksToLocalStorage(updatedTasks);
};
```

```
const renderTasks = () => {

  taskListContainer.innerHTML = ''; // Clear existing tasks

  const tasks = getTasksFromLocalStorage(); // Get tasks from localStorage
  tasks.forEach((task, index) => {
    if (task.trim() === "") return;
    const [taskInputText, taskDate] = task.split(' - ');
    const tasksDate = new Date(taskDate).toLocaleDateString("en-GB");
    /* console.log(taskInputText, tasksDate); */

    // Create the task item elements
    const taskItem = document.createElement('div');
    taskItem.className = 'input-group mb-3 task-item';

    const taskCheckbox = document.createElement('div');
    taskCheckbox.className = 'input-group-text bg-success-subtle';
    taskCheckbox.innerHTML = `<input class="form-check-input mt-0" type="checkbox"
aria-label="Checkbox for following text input">`;

    const taskText = document.createElement('input');
    taskText.type = 'text';
    taskText.className = 'form-control';
    taskText.value = taskInputText.replace("✓ ", ""); // Remove the marker from
the text
    taskText.readOnly = true;

    const date = document.createElement('div');
    date.className = 'date text-dark px-2';
    date.innerHTML = `${taskDate}`;


    const claenderButton = document.createElement('button');
    claenderButton.className = 'btn buttons btn-light disabled';
    claenderButton.innerHTML = `<svg xmlns="http://www.w3.org/2000/svg"
height="24px" viewBox="0 -960 960 960" width="24px" fill="#000000">
                                        <path d="M200-80q-33 0-56.5-23.5T120-
160v-560q0-33 23.5-56.5T200-800h40v-80h80v80h320v-80h80v80h40q33 0 56.5 23.5T840-
720v560q0 33-23.5 56.5T760-80H200Zm0-80h560v-400H200v400Zm0-480h560v-80H200v80Zm0
0v-80 80Zm280 240q-17 0-28.5-11.5T440-440q0-17 11.5-28.5T480-480q17 0 28.5
11.5T520-440q0 17-11.5 28.5T480-400Zm-160 0q-17 0-28.5-11.5T280-440q0-17 11.5-
28.5T320-480q17 0 28.5 11.5T360-440q0 17-11.5 28.5T320-400Zm320 0q-17 0-28.5-
11.5T600-440q0-17 11.5-28.5T640-480q17 0 28.5 11.5T680-440q0 17-11.5 28.5T640-
400ZM480-240q-17 0-28.5-11.5T440-280q0-17 11.5-28.5T480-320q17 0 28.5 11.5T520-
280q0 17-11.5 28.5T480-240Zm-160 0q-17 0-28.5-11.5T280-280q0-17 11.5-28.5T320-
320q17 0 28.5 11.5T360-280q0 17-11.5 28.5T320-240Zm320 0q-17 0-28.5-11.5T600-
280q0-17 11.5-28.5T640-320q17 0 28.5 11.5T680-280q0 17-11.5 28.5T640-240Z"/>
                                    </svg>`

    const editButton = document.createElement('button');
    editButton.className = `btn buttons border-none`;
    editButton.innerHTML = `<svg xmlns="http://www.w3.org/2000/svg" x="0px"
y="0px" width="23" height="23" viewBox="0 0 50 50">
```

```
                                    <path d="M 43.125 2 C 41.878906 2 40.636719
2.488281 39.6875 3.4375 L 38.875 4.25 L 45.75 11.125 C 45.746094 11.128906 46.5625
10.3125 46.5625 10.3125 C 48.464844 8.410156 48.460938 5.335938 46.5625 3.4375 C
45.609375 2.488281 44.371094 2 43.125 2 Z M 37.34375 6.03125 C 37.117188 6.0625
36.90625 6.175781 36.75 6.34375 L 4.3125 38.8125 C 4.183594 38.929688 4.085938
39.082031 4.03125 39.25 L 2.03125 46.75 C 1.941406 47.09375 2.042969 47.457031
2.292969 47.707031 C 2.542969 47.957031 2.90625 48.058594 3.25 47.96875 L 10.75
45.96875 C 10.917969 45.914063 11.070313 45.816406 11.1875 45.6875 L 43.65625
13.25 C 44.054688 12.863281 44.058594 12.226563 43.671875 11.828125 C 43.285156
11.429688 42.648438 11.425781 42.25 11.8125 L 9.96875 44.09375 L 5.90625 40.03125
L 38.1875 7.75 C 38.488281 7.460938 38.578125 7.011719 38.410156 6.628906 C
38.242188 6.246094 37.855469 6.007813 37.4375 6.03125 C 37.40625 6.03125 37.375
6.03125 37.34375 6.03125 Z"></path>
                                    </svg>`;
      /* checkboxInput.checked ? editButton.classList.add('disabled') :
editButton.classList.remove('disabled'); */

    const deleteButton = document.createElement('button');
    deleteButton.className = 'btn buttons';
    deleteButton.innerHTML = `<svg xmlns="http://www.w3.org/2000/svg" x="0px"
y="0px" width="23" height="23" viewBox="0 0 30 30">
                                    <path d="M 14.984375 2.4863281 A 1.0001 1.0001
0 0 0 14 3.5 L 14 4 L 8.5 4 A 1.0001 1.0001 0 0 0 7.4863281 5 L 6 5 A 1.0001
1.0001 0 1 0 6 7 L 24 7 A 1.0001 1.0001 0 1 0 24 5 L 22.513672 5 A 1.0001 1.0001 0
0 0 21.5 4 L 16 4 L 16 3.5 A 1.0001 1.0001 0 0 0 14.984375 2.4863281 z M 6 9 L
7.7929688 24.234375 C 7.9109687 25.241375 8.7633438 26 9.7773438 26 L 20.222656 26
C 21.236656 26 22.088031 25.241375 22.207031 24.234375 L 24 9 L 6 9 z"></path>
                                </svg>`;

    // Set the checkbox status and text-decoration based on the task marker
    const checkboxInput = taskCheckbox.querySelector('input');
    checkboxInput.checked = task.startsWith("✓ ");
    taskItem.style.textDecoration = checkboxInput.checked ? 'line-through' :
'none';

    // Add change event for the checkbox
    checkboxInput.addEventListener('change', () => {
      tasks[index] = checkboxInput.checked ? `✓ ${task.replace("✓ ", "")}` :
task.replace("✓ ", "");
      taskItem.style.textDecoration = checkboxInput.checked ? 'line-through' :
'none';

      calculateProgress();
      window.location.reload();
      /* updateWeeklyProgress(); */
      saveTasksToLocalStorage(tasks);
    });
    checkboxInput.checked ? editButton.classList.add('disabled')  :
editButton.classList.remove('disabled');

    // Attach Flatpickr to the calendar button
    claenderButton.addEventListener('click', () => {
      flatpickr(claenderButton, {
        dateFormat: "d M", // Display format as 30 OCT
        minDate: "today", // Freeze dates before today
        onChange: function(selectedDates, dateStr, instance) {
```

```
          // Update the hidden input with the selected date in desired format
          if (dateStr !== taskDate) {
            updateTaskDate(index, dateStr); // Update task date if changed
          }
        }
      }).open();
      claenderButton.classList.add('disabled')
      hideCompletedTasks();
    });

    // Function to update the task date in localStorage
    const updateTaskDate = (taskId, newDate) => {
      const tasks = getTasksFromLocalStorage();
      const task = tasks[taskId];
      const [taskInputText] = task.split(' - ');

      // Update the task with the new date
      tasks[taskId] = `${taskInputText} - ${newDate}`;
      saveTasksToLocalStorage(tasks);
      renderTasks(); // Re-render the task list
      hideCompletedTasks();
      window.location.reload()
    };


    editButton.addEventListener('click', () => {
      if (taskText.readOnly) {
        // Enable editing
        taskText.readOnly = false;
        taskItem.style.textDecoration = 'none';
        claenderButton.classList.remove('disabled');
        claenderButton.classList.add('date-button')
        taskText.focus();
        editButton.innerHTML = `<svg xmlns="http://www.w3.org/2000/svg"
height="24px" viewBox="0 -960 960 960" width="24px" fill="#000000"><path d="M840-
680v480q0 33-23.5 56.5T760-120H200q-33 0-56.5-23.5T120-200v-560q0-33 23.5-
56.5T200-840h480l160 160Zm-80 34L646-760H200v560h560v-446ZM480-240q50 0 85-35t35-
85q0-50-35-85t-85-35q-50 0-85 35t-35 85q0 50 35 85t85 35ZM240-560h360v-
160H240v160Zm-40-86v446-560 114Z"/></svg>`;
      } else {
        // Save changes
        taskText.readOnly = true;
        claenderButton.classList.add('disabled');
        tasks[index] = `${taskText.value} - ${taskDate}`; // Update the task in
the array
        saveTasksToLocalStorage(tasks); // Save the updated tasks to local storage
        editButton.innerHTML = `<svg xmlns="http://www.w3.org/2000/svg" x="0px"
y="0px" width="23" height="23" viewBox="0 0 50 50">
                                    <path d="M 43.125 2 C 41.878906 2 40.636719
2.488281 39.6875 3.4375 L 38.875 4.25 L 45.75 11.125 C 45.746094 11.128906 46.5625
10.3125 46.5625 10.3125 C 48.464844 8.410156 48.460938 5.335938 46.5625 3.4375 C
45.609375 2.488281 44.371094 2 43.125 2 Z M 37.34375 6.03125 C 37.117188 6.0625
36.90625 6.175781 36.75 6.34375 L 4.3125 38.8125 C 4.183594 38.929688 4.085938
39.082031 4.03125 39.25 L 2.03125 46.75 C 1.941406 47.09375 2.042969 47.457031
2.292969 47.707031 C 2.542969 47.957031 2.90625 48.058594 3.25 47.96875 L 10.75
45.96875 C 10.917969 45.914063 11.070313 45.816406 11.1875 45.6875 L 43.65625
```

```
13.25 C 44.054688 12.863281 44.058594 12.226563 43.671875 11.828125 C 43.285156
11.429688 42.648438 11.425781 42.25 11.8125 L 9.96875 44.09375 L 5.90625 40.03125
L 38.1875 7.75 C 38.488281 7.460938 38.578125 7.011719 38.410156 6.628906 C
38.242188 6.246094 37.855469 6.007813 37.4375 6.03125 C 37.40625 6.03125 37.375
6.03125 37.34375 6.03125 Z"></path>
                                    </svg>`; // Change button text back to "Edit"
(optional)
        window.location.reload()

      }
      hideCompletedTasks();

    });

    // Add delete functionality
    deleteButton.addEventListener('click', () => {
      tasks.splice(index, 1);
      calculateProgress();
      window.location.reload();
      /* updateWeeklyProgress(); */
      saveTasksToLocalStorage(tasks);
      renderTasks(); // Re-render the task list
      hideCompletedTasks();
    });

    // Append elements to the task item
    taskItem.appendChild(taskCheckbox);
    taskItem.appendChild(taskText);
    taskItem.appendChild(date);
    taskItem.appendChild(claenderButton);
    taskItem.appendChild(editButton);
    taskItem.appendChild(deleteButton);

    // Append the task item to the task list container
    taskListContainer.appendChild(taskItem);
  });
};

// Initialize flatpickr on the date button
flatpickr(".date-button", {
  dateFormat: "d M", // Display format as 30 OCT
  minDate: "today", // Freeze dates before today
  onChange: function(selectedDates, dateStr, instance) {
    // Update the hidden input with the selected date in desired format
    document.getElementById("task-date").value = dateStr;
  }
});

const alertTrigger = document.getElementById('task-add');
const date = document.getElementById('date');
if (alertTrigger) {
    alertTrigger.addEventListener('click', () => {
        const taskValue = taskInput.value.trim();
        const selectedDate = document.getElementById("task-date").value; // Use
formatted date or fallback
        const task = `${taskValue} - ${selectedDate ? selectedDate : "No Date"}`;
```

```
        if (taskValue !== '') {
            const tasks = getTasksFromLocalStorage();
            tasks.push(task); // Add task with date
            saveTasksToLocalStorage(tasks);
            appendAlert('Congratulations, your task is added!', 'success');
            renderTasks(); // Update the task list
            taskInput.value = '';
            document.getElementById("task-date").value = ''; // Clear date for
next entry
        } else {
            appendAlert('Please enter a task before adding!', 'danger');
        }
        window.location.reload();
        hideCompletedTasks();
    });
    /* calculateProgress();
    updateWeeklyProgress(); */
};

// Function to enable reordering tasks
const reOrderTasks = () => {
  // Get the parent container of task items
  const taskContainer = document.getElementById('task-list-container');
  // Initialize Sortable on the container
  const sortable = new Sortable(taskContainer, {
    animation: 150, // Smooth animation during drag and drop
    onEnd: () => {
      /* console.log("Reordering tasks..."); */

      // Get the current tasks from local storage
      const storedTasks = getTasksFromLocalStorage()
      /* console.log("Stored tasks from localStorage:", storedTasks); */

      // Extract the reordered tasks
      const reorderedTasks = Array.from(taskContainer.children).map((taskElement)
=> {
        // Extract task description
        const taskDescription = taskElement.querySelector('.form-
control').value.trim();

        // Extract task date
        const taskDate = taskElement.querySelector('.date p').innerText.trim();

        // Check if the task is completed (starts with ✓)
        const isCompleted = taskElement.querySelector('.form-check-
input').checked;

        // Combine task description, date, and completion status
        const formattedTask = `${isCompleted ? '✓ ' : ''}${taskDescription} -
${taskDate}`;

        console.log("Formatted task:", formattedTask);

        // Match the exact task from stored tasks
        const matchedTask = storedTasks.find((storedTask) => storedTask ===
formattedTask);
```

```
        console.log("Matched task from storage:", matchedTask);

        return matchedTask; // Add matched task to the reordered array
      }).filter(Boolean); // Filter out any unmatched tasks

      /* console.log("Reordered tasks:", reorderedTasks); */

      // Save reordered tasks back to local storage
      localStorage.setItem('tasks', JSON.stringify(reorderedTasks));
      window.location.reload()
    },
  });
};

const dateHighlighlighting = () => {
  const today = new Date();
  const previousDay = new Date(today);
  previousDay.setDate(today.getDate() - 1);
  previousDay.setHours(11, 59, 59, 999);
  /* console.log(previousDay) */
  const tasks = getTasksFromLocalStorage()
  const taskElements = document.querySelectorAll('.task-item');
  tasks.forEach((task,index) => {
    const [taskDescription, taskDate] = task.split(' - ');
    const currentYear = new Date().getFullYear(); // Get the current year

    // Map of month names to numbers
    const monthMap = {
      "Jan": 0, "Feb": 1, "Mar": 2, "Apr": 3, "May": 4, "Jun": 5,
      "Jul": 6, "Aug": 7, "Sep": 8, "Oct": 9, "Nov": 10, "Dec": 11
    };

    // Split the date string into day and month parts
    const [day, month] = taskDate.split(" "); // ["05", "Dec"]

    // Format the date as yyyy-mm-dd
    const formattedDate = `${currentYear}-${monthMap[month]+1}-${day.padStart(2,
"0")}`;
    /* console.log(formattedDate) */

    // Create a Date object
    const taskDateObj = new Date(formattedDate);
    taskDateObj.setHours(0, 0, 0, 0);
    /* console.log(taskDateObj) */
    if(!taskDescription.startsWith("✓") && taskDateObj < previousDay) {
      console.log('Task is overdue:', task);
      const dateElement = taskElements[index]?.querySelector('.date');
      console.log(dateElement)
      if (dateElement) {
        dateElement.classList.remove('text-dark');
        dateElement.classList.add('text-danger', 'fw-bold');
      } else {
        console.log("Date element not found for task:", task);
      }
```

```
    }
  })
}

const hideCompletedTasks = () => {
  const tasks = getTasksFromLocalStorage(); // Fetch tasks from local storage
  const today = new Date()/* .toLocaleDateString("en-GB") */; // Get today's date
  /* today.setHours(0, 0, 0, 0); // Set time to 00:00:00 to compare only the date
*/
  /* console.log(today) */
  const previousDay = new Date(today);
  previousDay.setDate(today.getDate() - 1);
  previousDay.setHours(11, 59, 59, 999);

  const taskElements = document.querySelectorAll('.task-item'); // Select all task
items
  /* console.log(taskElements) */

  /* console.log('Tasks from localStorage:', tasks);
  console.log('Task elements in DOM:', taskElements); */

  tasks.forEach((task, index) => {
    const [taskDescription, taskDate] = task.split(' - ');

    const currentYear = new Date().getFullYear(); // Get the current year

    // Map of month names to numbers
    const monthMap = {
      "Jan": 0, "Feb": 1, "Mar": 2, "Apr": 3, "May": 4, "Jun": 5,
      "Jul": 6, "Aug": 7, "Sep": 8, "Oct": 9, "Nov": 10, "Dec": 11
    };

    // Split the date string into day and month parts
    const [day, month] = taskDate.split(" "); // ["05", "Dec"]

    // Format the date as yyyy-mm-dd
    const formattedDate = `${currentYear}-${monthMap[month]+1}-${day.padStart(2,
"0")}`;

    // Create a Date object
    const taskDateObj = new Date(formattedDate);
    /* console.log(taskDateObj) */

    // Hide task if it's completed and the date is before today
    if (taskDateObj < previousDay && taskDescription.startsWith('✓ ')) {
      /* console.log(`Hiding task: ${task}`); */

      if (taskElements[index]) {
        taskElements[index].classList.add('d-none'); // Hide the task in the DOM
      } else {
        console.error(`Task element not found for index: ${index}`);
      }
    }
  });
};
```

```
function completeOrder() {
  const tasks = getTasksFromLocalStorage();
  const completedTasks = tasks.filter(task => task.startsWith('✓ '));
  const uncompletedTasks = tasks.filter(task => !task.startsWith('✓ '));
  const orderedTasks = [...uncompletedTasks, ...completedTasks];
  /* console.log(orderedTasks); */
  saveTasksToLocalStorage(orderedTasks);
  renderTasks();
  hideCompletedTasks();
}

// Call the function after DOM is loaded
document.addEventListener('DOMContentLoaded', () => {
  renderTasks();
  hideCompletedTasks();
  completeOrder();
  reOrderTasks();
  scheduleWeeklyCleanup();
});
window.onload = dateHighlighlighting;
```

1. Selecting Elements

```
const mainPart = document.getElementById("main-part");
const alertPlaceholder =
document.getElementById('success-alert');
const dateInput = document.getElementById('task-date');
const taskInput =
document.getElementById('floatingInput');
const taskListContainer = document.getElementById('task-
list-container');
let selectedDate = '';
```

- **`mainPart`**: This selects the element with the ID `main-part` (likely the section of the page where alerts or messages will be displayed).
- **`alertPlaceholder`**: This selects the element with the ID `success-alert` for showing success notifications or alerts.
- **`dateInput`**: This selects the element with the ID `task-date`, which is the input field where the user selects the task date.
- **`taskInput`**: This selects the element with the ID `floatingInput`, which is the input field where the user enters the task description.

- **`taskListContainer`**: This selects the container (likely a `div`) that will hold the list of tasks.
- **`selectedDate`**: This is a variable that will hold the selected task date.

These variables are used to interact with specific parts of the HTML page.

---

2. Alert Function

```
const appendAlert = (message, type) => {
  const wrapper = document.createElement('div');
  wrapper.innerHTML = [
    `<div class="alert alert-${type} alert-dismissible
container" role="alert">`,
    `   <div>${message}</div>`,
    '   <button type="button" class="btn-close" data-bs-
dismiss="alert" aria-label="Close"></button>',
    '</div>'
  ].join('');
  const alertElement = wrapper.firstChild;
  mainPart.prepend(alertElement);
  setTimeout(() => {
    alertElement.remove();
  }, 2000);
};
```

- **`appendAlert`**: This function is used to create and display an alert with a message. It:
  - Takes two arguments: `message` (the alert's text) and `type` (the style of the alert, e.g., `success`, `error`).
  - It creates a `div` element that contains the alert HTML, including a close button.
  - The alert is added to the `mainPart` element using `prepend`, which inserts the alert at the beginning of the container.
  - The alert automatically disappears after 2 seconds, using `setTimeout` to remove it from the DOM.

---

3. Local Storage Functions

```
const getTasksFromLocalStorage = () => {
  return JSON.parse(localStorage.getItem('tasks')) ||
[];
};

const saveTasksToLocalStorage = (tasks) => {
  localStorage.setItem('tasks', JSON.stringify(tasks));
};
```

- **`getTasksFromLocalStorage`**:
  - Fetches the `tasks` from the browser's local storage.
  - If no tasks are found, it returns an empty array `[]` as a fallback.
  - `JSON.parse` is used to convert the JSON string stored in local storage back into a JavaScript array.
- **`saveTasksToLocalStorage`**:
  - Takes the `tasks` array and saves it to local storage as a string.
  - `JSON.stringify` converts the tasks array into a string that can be stored in local storage.

---

4. Weekly Cleanup Scheduler

```
const scheduleWeeklyCleanup = () => {
  const today = new Date();
  const startOfWeek = new Date(today);
  const currentDay = today.getDay();
  const offset = -currentDay;
  startOfWeek.setDate(today.getDate() + offset);
  startOfWeek.setHours(0, 0, 0, 0);

  const endOfWeek = new Date(startOfWeek);
  endOfWeek.setDate(startOfWeek.getDate() + 6);
  endOfWeek.setHours(23, 59, 59, 999);

  removeOldTasksFromLocalStorage(startOfWeek);
};
```

- **`scheduleWeeklyCleanup`**: This function schedules a cleanup process that will remove tasks older than the start of the current week.
  - **`today`**: Get the current date and time.
  - **`startOfWeek`**: This calculates the start of the week (Sunday). We get the current day of the week using `today.getDay()` and calculate the offset to adjust the date to the start of the week.
  - **`endOfWeek`**: This calculates the end of the week (Saturday), by adding 6 days to the `startOfWeek` and setting the time to 23:59:59.
  - Finally, it calls the `removeOldTasksFromLocalStorage` function to clean up tasks based on the start of the week.

---

5. Remove Old Tasks from Local Storage

```
const removeOldTasksFromLocalStorage = (startOfWeek) =>
{
  const tasks = getTasksFromLocalStorage();
  const updatedTasks = tasks.filter(task => {
    const [taskDescription, taskDate] = task.split(' -
');
```

```
    const currentYear = new Date().getFullYear();
    const monthMap = { "Jan": 0, "Feb": 1, "Mar": 2,
"Apr": 3, "May": 4, "Jun": 5, "Jul": 6, "Aug": 7, "Sep":
8, "Oct": 9, "Nov": 10, "Dec": 11 };
    const [day, month] = taskDate.split(" ");
    const formattedDate = `${currentYear}-
${monthMap[month]+1}-${day.padStart(2, "0")}`;
    const taskDateObj = new Date(formattedDate);

    return taskDateObj >= startOfWeek ||
(!taskDescription.startsWith('✓') && taskDateObj <
startOfWeek);
  });
  saveTasksToLocalStorage(updatedTasks);
};
```

- **`removeOldTasksFromLocalStorage`**: This function removes tasks that are older than the current week.
  - **`tasks`**: Retrieves the tasks stored in local storage.
  - **`updatedTasks`**: Filters the tasks based on their date. It:
    - Splits the task into description and date.
    - Converts the task's date to a `Date` object for comparison.
    - Keeps tasks if the task's date is after or on the start of the week.
  - Finally, it saves the updated task list back to local storage.

---

6. Render Tasks

```
const renderTasks = () => {
  taskListContainer.innerHTML = '';
  const tasks = getTasksFromLocalStorage();
  tasks.forEach((task, index) => {
    if (task.trim() === "") return;
    const [taskInputText, taskDate] = task.split(' - ');
    const tasksDate = new
Date(taskDate).toLocaleDateString("en-GB");

    const taskItem = document.createElement('div');
    taskItem.className = 'input-group mb-3 task-item';

    const taskCheckbox = document.createElement('div');
    taskCheckbox.className = 'input-group-text bg-
success-subtle';
    taskCheckbox.innerHTML = `<input class="form-check-
input mt-0" type="checkbox" aria-label="Checkbox for
following text input">`;

    const taskText = document.createElement('input');
```

```
    taskText.type = 'text';
    taskText.className = 'form-control';
    taskText.value = taskInputText.replace("✓ ", "");
    taskText.readOnly = true;

    const date = document.createElement('div');
    date.className = 'date text-dark px-2';
    date.innerHTML = `${taskDate}`;

    const claenderButton =
document.createElement('button');
    claenderButton.className = 'btn buttons btn-light
disabled';
    claenderButton.innerHTML = `<svg
xmlns="http://www.w3.org/2000/svg" height="24px"
viewBox="0 -960 960 960" width="24px"
fill="#000000"><path d="M200-80q-33 0-56.5-23.5T120-
160v-560q0-33 23.5-56.5T200-800h40v-80h80v80h320v-
80h80v80h40q33 0 56.5 23.5T840-720v560q0 33-23.5
56.5T760-80H200Zm0-80h560v-400H200v400Zm0-480h560v-
80H200v80Zm0 0v-80 80Zm280 240q-17 0-28.5-11.5T440-
440q0-17 11.5-28.5T480-480q17 0 28.5 11.5T520-440q0 17-
11.5 28.5T480-400Zm-160 0q-17 0-28.5-11.5T280-440q0-17
11.5-28.5T320-480q17 0 28.5 11.5T360-440q0 17-11.5
28.5T320-400Zm320 0q-17 0-28.5-11.5T600-440q0-17 11.5-
28.5T640-480q17 0 28.5 11.5T680-440q0 17-11.5 28.5T640-
400ZM480-240q-17 0-28.5-11.5T440-280q0-17 11.5-28.5T480-
320q17 0 28.5 11.5T520-280q0 17-11.5 28.5T480-240Zm-160
0q-17 0-28.5-11.5T280-280q0-17 11.5-28.5T320-320q17 0
28.5 11.5T360-280q0 17-11.5 28.5T320-240Zm320 0q-17 0-
28.5-11.5T600-280q0-17 11.5-28.5T640-320q17 0 28.5
11.5T680-280q0 17-11.5 28.5T640-240Z"/></svg>`;

    const editButton = document.createElement('button');
    editButton.className = `btn buttons border-none`;
    editButton.innerHTML = `<svg
xmlns="http://www.w3.org/2000/svg" width="20"
height="20" viewBox="0 0 20 20"><path d="M12.77 2.23a2 2
0 0 0-2.828 0L10 2.586 13.414 6 14 5.414a2 2 0 0 0 0-
2.828ZM13.207 7.207l-.707.707L4 14.5v2.793h2.793l8.5-
8.5-.707-.707-.5.5-8.5 8.5H6.707l8.5-8.5z"/></svg>`;

    taskItem.append(taskCheckbox, taskText, date,
claenderButton, editButton);
    taskListContainer.appendChild(taskItem);
  });
};
```

- **`renderTasks`**: This function renders tasks to the UI dynamically.

- **Clears existing tasks**: First, it clears any existing tasks in the container by setting `taskListContainer.innerHTML = '';`.
- **Fetches tasks**: Retrieves the tasks from local storage.
- **Loops through tasks**: For each task:
  - Splits the task into description and date.
  - Creates a `div` containing various elements for the task:
    - Checkbox for marking completion.
    - Input field for showing the task text (read-only).
    - Display for the task's date.
    - Calendar and edit buttons.
- Finally, it appends each task item to the container `taskListContainer`.

---

7. Handling Task Events

- The code provides event handling for various actions such as task completion, editing, or updating task dates.
  - Tasks can be marked as completed by checking the checkbox.
  - Editing tasks allows the user to modify the task description.
  - Changing task dates involves updating the `task-date` field.

6.progressTracker.js (for knowing the status of the tasks in the week):

```
//Progress Tracker Graph using Chart.js
let progressChart;

const initializeChart = () => {
    const ctx = document.getElementById('progress-chart').getContext('2d');
    progressChart = new Chart(ctx, {
        type: 'bar',
        data: {
            labels: [], // Labels will update dynamically
            datasets: [{
                label: 'No. of Tasks Completed',
                data: [],
                backgroundColor: 'rgba(75, 192, 192, 0.2)',
                borderColor: 'rgba(75, 192, 192, 1)',
                borderWidth: 1
            }]
        },
        options: {
          plugins: {
              legend: {
                  labels: {
                      color: '#1b263b', // Change text color
                      font: {
                          size: 16 // Change font size
                      },
```

```
                    boxWidth: 0, // Change size of the legend box
                    boxHeight: 0 // Change height of the legend box
                },
                position: 'top' // Move the legend to the bottom
            }
        },
        scales: {
            y: {
                beginAtZero: true,
            }
        }
    }

});
};

initializeChart();

// Formatting Dates Using Flatpickr
flatpickr.formatDate = (date, format) => {
  const options = { day: "2-digit", month: "short" }; // Format: "28 OCT"
  const formattedDate = date.toLocaleDateString("en-US", options).toUpperCase();

  // Swap the day and month order to "29 NOV"
  const [month, day] = formattedDate.split(" "); // Split on space
  return `${day} ${month}`;
};
const getWeekLabels = () => {
  const today = new Date();
  const options = { day: "2-digit", month: "short" };

  // Find the start of the week (Sunday)
  const startOfWeek = new Date(today);
  startOfWeek.setDate(today.getDate() - today.getDay()); // Subtract days to get
to Sunday

  // Generate labels from Sunday to Saturday
  return Array.from({ length: 7 }, (_, i) => {
    const date = new Date(startOfWeek); // Copy the start of the week
    date.setDate(startOfWeek.getDate() + i); // Add i days to get the correct day
    const formattedDate = date.toLocaleDateString("en-US", options);

  // Swap the day and month order to "29 NOV"
  const [month, day] = formattedDate.split(" "); // Split on space
  return `${day} ${month}`; // Format as "d M"
  });
};

/* // Example usage:
console.log(getWeekLabels()); */


const calculateProgress = () => {
  const tasks = getTasksFromLocalStorage(); // Ensure this function retrieves the
data correctly
  /* console.log("All tasks:", tasks); */
```

```
  const today = new Date().toLocaleDateString('en-GB'); // Format today's date as
dd/mm/yyyy
  /* console.log(today)*/
  // Calculate completed tasks for today
  const completedToday = tasks.filter((task) => {
    const [taskDescription, taskDate] = task.split(' - '); // Assuming tasks are
stored as "description - date"
    const currentYear = new Date().getFullYear(); // Get the current year

    // Map of month names to numbers
    const monthMap = {
      "Jan": 0, "Feb": 1, "Mar": 2, "Apr": 3, "May": 4, "Jun": 5,
      "Jul": 6, "Aug": 7, "Sep": 8, "Oct": 9, "Nov": 10, "Dec": 11
    };

    // Split the date string into day and month parts
    const [day, month] = taskDate.split(" "); // ["05", "Dec"]

    // Format the date as yyyy-mm-dd
    const formattedDate = `${currentYear}-${monthMap[month]+1}-${day.padStart(2,
"0")}`;

    // Create a Date object
    const taskDateObj = new Date(formattedDate).toLocaleDateString("en-GB");
    /* console.log(taskDateObj) */
    return taskDescription.startsWith("✓ ") && taskDateObj === today; // Check
both completion and date match
  });

  /* console.log("Completed tasks today:", completedToday); */
  return completedToday.length; // Return the count of tasks completed today
};

// Retrieve progress from localStorage (an array of percentages)
const getProgressFromLocalStorage = () => {
  const progress = JSON.parse(localStorage.getItem("weekProgress")) || [0, 0, 0,
0, 0, 0, 0];

  // Ensure the retrieved progress is a nested array
  const formattedProgress = Array.isArray(progress[0]) ? progress : [progress];

  /* console.log("Formatted Progress from localStorage:", formattedProgress); //
Debugging line */

  return formattedProgress;
};

/* const progress1 = getProgressFromLocalStorage()
console.log(progress1)
console.log(calculateProgress()) */

const saveProgressToLocalStorage = (progress) => {
  /* console.log("Saving Progress to localStorage:", progress); // Debugging line
*/
  localStorage.setItem("weekProgress", JSON.stringify(progress));
```

```
};

// Helper function to get the current week number
const getCurrentWeek = () => {
  const today = new Date();
  const startOfYear = new Date(today.getFullYear(), 0, 1);
  const days = Math.floor((today - startOfYear) / (24 * 60 * 60 * 1000));
  return Math.ceil((days + startOfYear.getDay() + 1) / 7);
};

// Check and reset weekly progress if needed
const resetWeeklyProgressIfNeeded = () => {
  const storedWeek = parseInt(localStorage.getItem("lastUpdatedWeek")) || 0;
  const currentWeek = getCurrentWeek();

  if (storedWeek !== currentWeek) {
    // Reset progress to 0 for the new week
    const resetProgress = [0, 0, 0, 0, 0, 0, 0];
    saveProgressToLocalStorage([resetProgress]);

    // Update the stored week number
    localStorage.setItem("lastUpdatedWeek", currentWeek);
  }
};



const saveDailyProgress = (completedTasks, totalTasks) => {
  // Reset weekly progress if a new week has started
  resetWeeklyProgressIfNeeded();

  // Calculate the progress percentage
  const percentage = calculateProgress(completedTasks, totalTasks);

  // Format today's date as "D MMM"
  const today = new Date();
  const options = { day: "2-digit", month: "short" };
  const formattedDate = today.toLocaleDateString("en-US", options); // Format: "30
Oct"
  const [month, day] = formattedDate.split(" "); // Split on space
  const dateLabel = `${day} ${month}`;

  // Save the progress percentage for the specific date
  saveProgress(dateLabel, percentage);
};


const saveProgress = (dateLabel, percentage) => {
  // Dates that will be tracked for progress
  const dateList = getWeekLabels();

  // Retrieve the current progress array from localStorage
  let storedProgress = getProgressFromLocalStorage();

  // Check if storedProgress is valid, otherwise initialize it
```

```
  if (!storedProgress || !Array.isArray(storedProgress) || storedProgress.length
=== 0) {
    storedProgress = [[0, 0, 0, 0, 0, 0, 0]]; // Default 0 progress for each day
    /* console.log("Progress was uninitialized. Initialized storedProgress:",
storedProgress); */
  }

  // Find the index of the dateLabel in the dateList
  const dateIndex = dateList.indexOf(dateLabel);
  if (dateIndex !== -1) {
    // Update the progress at the found index with the given percentage
    storedProgress[0][dateIndex] = percentage;
    /* console.log("Updated storedProgress:", storedProgress); */
  } /* else {
    console.error("Date label not found in dateList:", dateLabel);
  } */

  // Save the updated progress back to localStorage
  saveProgressToLocalStorage(storedProgress);
};


saveDailyProgress()

// Get the progress for the week
const getProgressForWeek = () => {
  const weekLabels = getWeekLabels(); // Generate labels for the last 7 days
  const storedProgress = getProgressFromLocalStorage();

  // Debugging: Check the value of storedProgress
  /* console.log("Stored Progress from localStorage:", storedProgress); */

  // Ensure storedProgress is an array with at least one sub-array
  if (!storedProgress || !Array.isArray(storedProgress) || !storedProgress[0]) {
    console.error("Stored Progress is not properly initialized.");
    return Array(7).fill(0); // Return an array of 0s if progress is missing
  }

  // Map each label to the corresponding progress percentage or 0 if not present
  return weekLabels.map((_, index) => storedProgress[0][index] || 0); // Access
progress using index
};


// Update the chart with the new labels and completion percentage
const updateChart = () => {
  if (!progressChart) {
    return; // Ensure the chart is initialized
  }

  // Get weekly labels and corresponding progress
  const weekLabels = getWeekLabels();
  const progressData = getProgressForWeek();

  // Calculate the maximum number of tasks for the week
  const maxTasksForWeek = Math.max(...progressData);
```

```
  // Set the y-axis maximum dynamically based on the number of tasks
  const dynamicMaxY = maxTasksForWeek + 4; // Add a buffer for clarity (adjust as
needed)

  // Update chart data
  progressChart.data.labels = weekLabels;
  progressChart.data.datasets[0].data = progressData;

  // Update the y-axis max dynamically
  progressChart.options.scales.y.max = dynamicMaxY;

  // Refresh the chart
  progressChart.update();
};


// Function to update progress every week
const updateWeeklyProgress = () => {
  calculateProgress(); // Update stored progress
  updateChart(); // Refresh the chart with updated data
};

// Initialize weekly progress update
updateWeeklyProgress();


const scheduleWeeklyUpdate = () => {
  const now = new Date();
  const nextMonday = new Date(
    now.getFullYear(),
    now.getMonth(),
    now.getDate() + ((1 + 7 - now.getDay()) % 7)
  );
  const timeUntilNextMonday = nextMonday - now;

  setTimeout(() => {
    resetWeeklyProgressIfNeeded(); // Reset progress for the new week
    updateWeeklyProgress(); // Update chart
    setInterval(() => {
      resetWeeklyProgressIfNeeded();
      updateWeeklyProgress();
    }, 7 * 24 * 60 * 60 * 1000); // Repeat every week
  }, timeUntilNextMonday);
};

// Start the scheduled updates
scheduleWeeklyUpdate();
```

1. Initializing the Chart (`initializeChart`)

- **Objective**: Create a chart to visualize task completion over the week.
- **Code**:

```
let progressChart;

const initializeChart = () => {
    const ctx = document.getElementById('progress-
chart').getContext('2d');
    progressChart = new Chart(ctx, {
        type: 'bar',
        data: {
            labels: [], // Labels will update
dynamically
            datasets: [{
                label: 'No. of Tasks Completed',
                data: [],
                backgroundColor: 'rgba(75, 192, 192,
0.2)',
                borderColor: 'rgba(75, 192, 192, 1)',
                borderWidth: 1
            }]
        },
        options: {
          plugins: {
              legend: {
                  labels: {
                      color: '#1b263b', // Change text
color
                      font: {
                          size: 16 // Change font size
                      },
                      boxWidth: 0, // Change size of the
legend box
                      boxHeight: 0 // Change height of
the legend box
                  },
                  position: 'top' // Move the legend to
the bottom
              }
          },
          scales: {
              y: {
                  beginAtZero: true,
              }
          }
      }
    });
};
```

2. Formatting Dates with Flatpickr (`flatpickr.formatDate`)

- **Objective**: Custom date formatting to swap day and month order.
- **Code**:

```
flatpickr.formatDate = (date, format) => {
  const options = { day: "2-digit", month: "short" };
  const formattedDate = date.toLocaleDateString("en-US",
options).toUpperCase();
  const [month, day] = formattedDate.split(" ");
  return `${day} ${month}`; // Format as "29 NOV"
};
```

3. Getting Weekly Labels (`getWeekLabels`)

- **Objective**: Generate labels for the current week, starting from Sunday.
- **Code**:

```
const getWeekLabels = () => {
  const today = new Date();
  const options = { day: "2-digit", month: "short" };

  const startOfWeek = new Date(today);
  startOfWeek.setDate(today.getDate() - today.getDay());

  return Array.from({ length: 7 }, (_, i) => {
      const date = new Date(startOfWeek);
      date.setDate(startOfWeek.getDate() + i);
      const formattedDate = date.toLocaleDateString("en-
US", options);
      const [month, day] = formattedDate.split(" ");
      return `${day} ${month}`; // Format as "d M"
  });
};
```

4. Calculating Progress for Today (`calculateProgress`)

- **Objective**: Count the number of completed tasks for today.
- **Code**:

```
const calculateProgress = () => {
  const tasks = getTasksFromLocalStorage();
  const today = new Date().toLocaleDateString('en-GB');

  const completedToday = tasks.filter((task) => {
    const [taskDescription, taskDate] = task.split(' -
');
    const currentYear = new Date().getFullYear();

    const monthMap = {
      "Jan": 0, "Feb": 1, "Mar": 2, "Apr": 3, "May": 4,
"Jun": 5,
      "Jul": 6, "Aug": 7, "Sep": 8, "Oct": 9, "Nov": 10,
"Dec": 11
    };
```

```
    const [day, month] = taskDate.split(" ");
    const formattedDate = `${currentYear}-
${monthMap[month]+1}-${day.padStart(2, "0")}`;
    const taskDateObj = new
Date(formattedDate).toLocaleDateString("en-GB");

    return taskDescription.startsWith("✓ ") &&
taskDateObj === today;
  });

  return completedToday.length;
};
```

5. Getting and Saving Progress to Local Storage (`getProgressFromLocalStorage`, `saveProgressToLocalStorage`)

- **Objective**: Retrieve and store weekly progress data in localStorage.
- **Code**:

```
const getProgressFromLocalStorage = () => {
  const progress =
JSON.parse(localStorage.getItem("weekProgress")) || [0,
0, 0, 0, 0, 0, 0];
  const formattedProgress = Array.isArray(progress[0]) ?
progress : [progress];
  return formattedProgress;
};

const saveProgressToLocalStorage = (progress) => {
  localStorage.setItem("weekProgress",
JSON.stringify(progress));
};
```

6. Getting Current Week Number (`getCurrentWeek`)

- **Objective**: Calculate the current week number of the year.
- **Code**:

```
const getCurrentWeek = () => {
  const today = new Date();
  const startOfYear = new Date(today.getFullYear(), 0,
1);
  const days = Math.floor((today - startOfYear) / (24 *
60 * 60 * 1000));
  return Math.ceil((days + startOfYear.getDay() + 1) /
7);
};
```

7. Resetting Weekly Progress (`resetWeeklyProgressIfNeeded`)

- **Objective**: Reset progress if a new week starts.

- **Code**:

```
const resetWeeklyProgressIfNeeded = () => {
  const storedWeek =
parseInt(localStorage.getItem("lastUpdatedWeek")) || 0;
  const currentWeek = getCurrentWeek();

  if (storedWeek !== currentWeek) {
    const resetProgress = [0, 0, 0, 0, 0, 0, 0];
    saveProgressToLocalStorage([resetProgress]);
    localStorage.setItem("lastUpdatedWeek",
currentWeek);
  }
};
```

8. Saving Daily Progress (`saveDailyProgress`)

- **Objective**: Save daily progress as a percentage for the current date.
- **Code**:

```
const saveDailyProgress = (completedTasks, totalTasks)
=> {
  resetWeeklyProgressIfNeeded();
  const percentage = calculateProgress(completedTasks,
totalTasks);

  const today = new Date();
  const options = { day: "2-digit", month: "short" };
  const formattedDate = today.toLocaleDateString("en-
US", options);
  const [month, day] = formattedDate.split(" ");
  const dateLabel = `${day} ${month}`;

  saveProgress(dateLabel, percentage);
};
```

9. Saving Progress for a Specific Date (`saveProgress`)

- **Objective**: Save progress for a specific date.
- **Code**:

```
const saveProgress = (dateLabel, percentage) => {
  const dateList = getWeekLabels();
  let storedProgress = getProgressFromLocalStorage();

  if (!storedProgress || !Array.isArray(storedProgress)
|| storedProgress.length === 0) {
    storedProgress = [[0, 0, 0, 0, 0, 0, 0]];
  }

  const dateIndex = dateList.indexOf(dateLabel);
```

```
  if (dateIndex !== -1) {
    storedProgress[0][dateIndex] = percentage;
  }

  saveProgressToLocalStorage(storedProgress);
};
```

10. Getting Progress for the Week (`getProgressForWeek`)

- **Objective**: Retrieve progress data for the entire week.
- **Code**:

```
const getProgressForWeek = () => {
  const weekLabels = getWeekLabels();
  const storedProgress = getProgressFromLocalStorage();

  if (!storedProgress || !Array.isArray(storedProgress)
|| !storedProgress[0]) {
    return Array(7).fill(0); // Return an array of 0s if
progress is missing
  }

  return weekLabels.map((_, index) =>
storedProgress[0][index] || 0);
};
```

11. Updating the Chart (`updateChart`)

- **Objective**: Update the chart with new progress data for the week.
- **Code**:

```
const updateChart = () => {
  if (!progressChart) return;

  const weekLabels = getWeekLabels();
  const progressData = getProgressForWeek();

  const maxTasksForWeek = Math.max(...progressData);
  const dynamicMaxY = maxTasksForWeek + 4;

  progressChart.data.labels = weekLabels;
  progressChart.data.datasets[0].data = progressData;
  progressChart.options.scales.y.max = dynamicMaxY;

  progressChart.update();
};
```

12. Updating Weekly Progress (`updateWeeklyProgress`)

- **Objective**: Update progress and chart every week.
- **Code**:

```
const updateWeeklyProgress = () => {
  calculateProgress();
  updateChart();
};
```

13. Scheduling Weekly Updates (`scheduleWeeklyUpdate`)

- **Objective**: Automatically reset progress and update weekly.
- **Code**:

```
const scheduleWeeklyUpdate = () => {
  const now = new Date();
  const nextMonday = new Date(now.getFullYear(),
now.getMonth(), now.getDate() + ((1 + 7 - now.getDay())
% 7));
  const timeUntilNextMonday = nextMonday - now;

  setTimeout(() => {
    resetWeeklyProgressIfNeeded();
    updateWeeklyProgress();
    setInterval(() => {
      resetWeeklyProgressIfNeeded();
      updateWeeklyProgress();
    }, 7 * 24 * 60 * 60 * 1000); // Every 7 days
  }, timeUntilNextMonday);
};
```

These snippets will help integrate the weekly progress tracking, chart updates, and automatic resetting mechanism into your task management application.

7.taskPrioritization2.js (for prioritizing the tasks):

```
async function loadPrioritizationCSV() {
  const csvPath = "./datasets/task_management.csv";

  // Load the dataset
  const dataset = tf.data.csv(csvPath);

  return dataset;
}
/* //log the dataset
async function logDataset() {
  const dataset = await loadPrioritizationCSV();
  const records = await dataset.toArray();
  console.log(records);
}
logDataset(); */

// Improved text tokenization with punctuation removal and stemming placeholder
function tokenizeText(text) {
  if (!text) return [];
  const trimmedText = text.trim();
```

```
  // Remove punctuation and split into words
  const tokens = trimmedText.toLowerCase().replace(/[.,!?]/g, ' ').split(/\s/);

  // Stemming or lemmatization could be added here
  return tokens;
}
// Enhanced preprocessing pipeline
async function processDataset() {
  const dataset = await loadPrioritizationCSV();

  // Map and transform dataset records
  const transformedDataset = dataset.map(record => {
    return {
      ...record,
      Task_Description: tokenizeText(record.Task_Description),
      Keywords: tokenizeText(record.Keywords),
    };
  });

  return transformedDataset;
}
/* //log the processed dataset
async function logProcessedDataset() {
  const transformedDataset = await processDataset();
  const records = await transformedDataset.toArray();
  console.log(records);
}
logProcessedDataset(); */

// Combine Description and Keywords for BoW representation
async function createCombinedVocabulary() {
  const transformedDataset = await processDataset();
  const datasetArray = await transformedDataset.toArray();

  // Collect tokens from Description and Keywords
  const allTokens = datasetArray.flatMap(row =>
row.Task_Description.concat(row.Keywords));

  // Remove duplicates and combine with priorityKeywords
  const uniqueTokens = [...new Set(allTokens)];
  return uniqueTokens;
}
/* //log the combined vocabulary
async function logCombinedVocabulary() {
  const vocabulary = await createCombinedVocabulary();
  console.log(vocabulary);
}
logCombinedVocabulary(); */

// Function to create Bag of Words vector
async function createBoWVector(tokens, vocabulary) {
  const vector = new Array(vocabulary.length).fill(0);  // Initialize vector with
zeros

  tokens.forEach(token => {
    const index = vocabulary.indexOf(token);
```

```
    if (index !== -1) {
      vector[index] += 1; // Increment the count
    }
  });
  return vector; // Return the BoW vector
}

// Create separate BoW matrices for description and keywords
async function createDescriptionBoWMatrix() {
  const vocabulary = await createCombinedVocabulary();
  const transformedDataset = await processDataset();
  const datasetArray = await transformedDataset.toArray();

  // Generate Bag-of-Words vectors for description
  const bowMatrix = await Promise.all(
    datasetArray.map(row => {
      return createBoWVector(row.Task_Description, vocabulary);
    })
  );
  return bowMatrix;
}
/* //log the description BoW matrix
async function logDescriptionBoWMatrix() {
  const bowMatrix = await createDescriptionBoWMatrix();
  console.log(bowMatrix);
}
logDescriptionBoWMatrix(); */

async function createKeywordsBoWMatrix() {
  const vocabulary = await createCombinedVocabulary();
  const transformedDataset = await processDataset();
  const datasetArray = await transformedDataset.toArray();

  // Generate Bag-of-Words vectors for keywords
  const bowMatrix = await Promise.all(
    datasetArray.map(row => {
      return createBoWVector(row.Keywords, vocabulary);
    })
  );

  return bowMatrix;
}

/* //log the keywords BoW matrix
async function logKeywordsBoWMatrix() {
  const bowMatrix = await createKeywordsBoWMatrix();
  console.log(bowMatrix);
}
logKeywordsBoWMatrix(); */

// Enhanced normalization of Due_Days
async function normalizeNumericalColumns() {
  const transformedDataset = await processDataset();
  const datasetArray = await transformedDataset.toArray();

  const dueDays = datasetArray.map(row => row.Due_Days);
```

```
  const dueDaysTensor = tf.tensor1d(dueDays);

  // Mean normalization using tf.moments()
  const { mean, variance } = tf.moments(dueDaysTensor);
  const std = tf.sqrt(variance).dataSync()[0];
  const meanValue = mean.dataSync()[0];

  const normalizedDueDays = dueDays.map(value => (value - meanValue) / std);

  return { normalizedDueDays, meanValue, std};
}
/* //log Numerical Columns
async function logNumericalColumns()
{
  const columns = await normalizeNumericalColumns();
  const normalizedDueDays = await columns.normalizedDueDays
  console.log(normalizedDueDays);
}
logNumericalColumns(); */

// Consolidated input feature matrix creation
async function createInputMatrix() {
  const descriptionBoWMatrix = await createDescriptionBoWMatrix();
  const keywordsBoWMatrix = await createKeywordsBoWMatrix();
  const { normalizedDueDays } = await normalizeNumericalColumns();

  const inputMatrix = descriptionBoWMatrix.map((row, index) => {
    return row.concat(keywordsBoWMatrix[index]).concat(normalizedDueDays[index]);
  });

  return inputMatrix;
}
/* //log the input matrix
  async function logInputMatrix() {
  const inputMatrix = await createInputMatrix();
  console.log(inputMatrix);
  }
  logInputMatrix(); */

async function createOutputMatrix() {
  const transformedDataset = await processDataset();
  const datasetArray = await transformedDataset.toArray();

  const priorityScores = datasetArray.map(row => row.Priority_Score);
  return priorityScores;
}
/* //log the output matrix
  async function logOutputMatrix() {
  const outputMatrix = await createOutputMatrix();
  console.log(outputMatrix);
  }
  logOutputMatrix(); */

// Splitting the data into training and testing
async function splitData() {
  const inputMatrix = await createInputMatrix();
```

```
  const outputMatrix = await createOutputMatrix();

  // Split the data into training and testing
  const trainingDataSize = Math.floor(inputMatrix.length * 0.8);
  const trainingInput = inputMatrix.slice(0, trainingDataSize);
  const trainingOutput = outputMatrix.slice(0, trainingDataSize);

  const testingInput = inputMatrix.slice(trainingDataSize);
  const testingOutput = outputMatrix.slice(trainingDataSize);

  return { trainingInput, trainingOutput, testingInput, testingOutput };
}
/* //log the split data
async function logSplitData() {
  const { trainingInput, trainingOutput, testingInput, testingOutput } = await
splitData();
  console.log('Training Input:', trainingInput);
  console.log('Training Output:', trainingOutput);
  console.log('Testing Input:', testingInput);
  console.log('Testing Output:', testingOutput);
}
logSplitData(); */

async function createModelEnhanced(inputShape) {
  const model = tf.sequential();

  // Add dense layers with L2 regularization and batch normalization
  model.add(tf.layers.dense({
    units: 64,
    activation: "relu",
    inputShape: [inputShape],
    kernelRegularizer: tf.regularizers.l2({ l2: 0.02 })
  }));
  model.add(tf.layers.batchNormalization());
  model.add(tf.layers.dropout({ rate: 0.5 }));

  model.add(tf.layers.dense({
    units: 64,
    activation: "relu",
    kernelRegularizer: tf.regularizers.l2({ l2: 0.02 })
  }));
  model.add(tf.layers.batchNormalization());
  model.add(tf.layers.dropout({ rate: 0.5 }));

  /* model.add(tf.layers.dense({
    units: 32,
    activation: "relu",
    kernelRegularizer: tf.regularizers.l2({ l2: 0.01 })
  }));
  model.add(tf.layers.batchNormalization());
  model.add(tf.layers.dropout({ rate: 0.5 })); */

  model.add(tf.layers.dense({ units: 5, activation: "softmax" }));

  model.compile({
    optimizer: tf.train.adam(0.002),  // Adjusted learning rate
```

```
    loss: "sparseCategoricalCrossentropy",
    metrics: ["accuracy"],
  });

  return model;
}

async function trainModel() {
  const { trainingInput, trainingOutput } = await splitData();
  const model = await createModelEnhanced(trainingInput[0].length);

  const trainingInputTensor = tf.tensor2d(trainingInput, [trainingInput.length,
trainingInput[0].length]);
  const trainingOutputTensor = tf.tensor1d(trainingOutput);

  await model.fit(trainingInputTensor, trainingOutputTensor, {
    epochs: 60,
    batchSize: 32,
    validationSplit: 0.2,
    callbacks: [
      {
        onEpochEnd: async (epoch, logs) => {
          console.log(`Epoch: ${epoch} - loss: ${logs.loss} - accuracy:
${logs.acc}`);
        },
      },
    ],
  });

  return model;
}

async function loadModel() {
  try {
      const model = await tf.loadLayersModel('./models/my-prioritization-
model.json');
      console.log('Model loaded successfully');
      return model;
  } catch (error) {
      console.error('Error loading the model:', error);
  }
}
async function saveModel() {
    const model = await testModel();
    await model.save('localstorage://my-model');
    await model.save('downloads://my-prioritization-model');
    console.log('Model saved successfully');
}
/* saveModel(); */

async function testModel() {
  const model = await trainModel();
  const { testingInput, testingOutput } = await splitData();

  const testingInputTensor = tf.tensor2d(testingInput, [testingInput.length,
testingInput[0].length]);
```

```
  const testingOutputTensor = tf.tensor1d(testingOutput);

  const result = model.evaluate(testingInputTensor, testingOutputTensor);
  const loss = result[0].dataSync()[0];
  const accuracy = result[1].dataSync()[0];

  console.log(`Test Loss: ${loss}`);
  console.log(`Test Accuracy: ${accuracy}`);

  return model;
}
/* testModel(); */

// Function to extract keywords from the task description
function extractKeywords(description) {
  const tokens = tokenizeText(description);
  const keywordFrequency = {};

  tokens.forEach(token => {
    if (keywordFrequency[token]) {
      keywordFrequency[token]++;
    } else {
      keywordFrequency[token] = 1;
    }
  });

  // Sort tokens by frequency and select the top N keywords
  const sortedKeywords = Object.keys(keywordFrequency).sort((a, b) =>
keywordFrequency[b] - keywordFrequency[a]);

  return sortedKeywords;
}
// Function to predict task priority
async function predictTaskPriority(model, description, dueDays) {
  // Extract keywords from the description
  const keywords = extractKeywords(description);
  /* console.log('Extracted Keywords:', keywords); */

  // Tokenize the input description and keywords
  const tokenizedDescription = tokenizeText(description);
  const tokenizedKeywords = keywords;

  // Create BoW vectors for description and keywords
  const vocabulary = await createCombinedVocabulary();
  const descriptionBoW = await createBoWVector(tokenizedDescription, vocabulary);
  const keywordsBoW = await createBoWVector(tokenizedKeywords, vocabulary);

  /* console.log('Description BoW:', descriptionBoW);
  console.log('Keywords BoW:', keywordsBoW); */


  // Normalize the dueDays
  const { meanValue, std } = await normalizeNumericalColumns();
  const normalizedDueDays = ((Number(dueDays)) - meanValue) / std;

  /* console.log('Normalized Due Days:', normalizedDueDays); */
```

```
  // Combine the BoW vectors and normalized dueDays into a single input vector
  const inputVector =
descriptionBoW.concat(keywordsBoW).concat(normalizedDueDays);

  /* console.log('Input Vector:', inputVector);
  console.log('Input Vector Shape:', [1, inputVector.length]); */

  // Convert the input vector to a tensor
  const inputTensor = tf.tensor2d([inputVector]);

  // Log the model's expected input shape
  /* console.log('Model Expected Input Shape:', model.inputs[0].shape); */

  // Make the prediction
  const prediction = model.predict(inputTensor);
  const predictedClass = prediction.argMax(-1).dataSync()[0] + 1; // Adding 1 to
match the range 1-10

  return predictedClass;
}

function getDueDays(date) {
  const currentDate = new Date();
  const dueDate = new Date(date);
  const year = new Date().getFullYear();

  // Format the dates as dd/mm/yyyy
  const formattedDueDate = `${dueDate.getDate()}/${dueDate.getMonth() +
1}/${year}`;
  const formattedCurrentDate = `${currentDate.getDate()}/${currentDate.getMonth()
+ 1}/${year}`;

  /* console.log(`Formatted Due Date: ${formattedDueDate}`);
  console.log(`Formatted Current Date: ${formattedCurrentDate}`); */

  // Parse the formatted date strings back into Date objects
  const [dueDay, dueMonth, dueYear] = formattedDueDate.split('/').map(Number);
  const [currentDay, currentMonth, currentYear] =
formattedCurrentDate.split('/').map(Number);

  const parsedDueDate = new Date(dueYear, dueMonth - 1, dueDay);
  const parsedCurrentDate = new Date(currentYear, currentMonth - 1, currentDay);

  /* console.log(`Parsed Due Date: ${parsedDueDate}`);
  console.log(`Parsed Current Date: ${parsedCurrentDate}`); */

  // Calculate the difference in time
  const timeDifference = parsedDueDate - parsedCurrentDate;

  // Convert time difference from milliseconds to days
  const dueDays = Math.ceil(timeDifference / (1000 * 60 * 60 * 24));

  return dueDays;
}
```

```
async function taskList() {
  const tasks = await getTasksFromLocalStorage();
  const tasksPriority = [];
  const model = await loadModel();

  for (const task of tasks) {
    const [taskDescription, taskDate] = task.split(' - ');
    const tasksDate = taskDate;
    const dueDays = parseInt(getDueDays(taskDate));
    const predictedPriority = await predictTaskPriority(model, taskDescription,
dueDays);
    tasksPriority.push({ taskDescription,tasksDate, priority: predictedPriority
});
  }
  return tasksPriority;

}
/* taskList(); */

async function savePriorityToLocalStorage() {
  const tasksPriority = await taskList();
  const prioritizedTasks = tasksPriority.map(({ taskDescription,tasksDate,
priority }) => `${taskDescription} - ${tasksDate} - ${priority}`);
  localStorage.setItem('prioritizedTasks', JSON.stringify(prioritizedTasks));
  /* console.log('Prioritized tasks saved to local storage:', prioritizedTasks);
*/
  await displayTaskList();
}
savePriorityToLocalStorage();

async function displayTaskList() {
  const smartListContainer = document.getElementById('smart-remainders-
container');
  if (!smartListContainer) {
    console.error('Element with ID "smart-remainders-container" not found.');
    return;
  }

  const tasks = await getTasksFromLocalStorage();
  const tasksPriority = JSON.parse(localStorage.getItem('prioritizedTasks'));
  /* console.log(tasksPriority) */

  // Combine tasks and priorities into an array of objects
  const tasksWithPriority = tasksPriority.map(task => {
    const [taskDescription,taskDate, priority] = task.split(' - ');
    return { task: taskDescription,date:taskDate, priority: parseInt(priority, 10)
};
  });

  // Sort the array based on priority
  tasksWithPriority.sort((a, b) => a.priority - b.priority);
  console.log(tasksWithPriority);

  // Clear the container before appending sorted tasks
  smartListContainer.innerHTML = '';
```

```
  let count = tasksWithPriority.length;
  // Display the sorted tasks
  tasksWithPriority.forEach(({ task, priority }) => {
    if (task.trim() === "") return;
    const remainderItem = document.createElement('div');
    remainderItem.className = 'input-group remainder-item';
    const taskInput = document.createElement('input');
    taskInput.type = 'text';
    taskInput.className = 'form-control remainder-text mb-3';
    taskInput.value = task;
    taskInput.disabled = true;
    taskInput.style.backgroundColor = `rgba(255, 0, 0, ${(count) / 10})`;
    count--;
    remainderItem.appendChild(taskInput);
    smartListContainer.appendChild(remainderItem);
  });
  const remainderElements = document.querySelectorAll('.remainder-item')
  /* console.log(remainderElements) */
  const today = new Date();
  const previousDay = new Date(today);
  previousDay.setDate(today.getDate() - 1);
  previousDay.setHours(11, 59, 59, 999);
  /* console.log(previousDay) */
  tasksWithPriority.forEach((task, index) => {
    /* console.log(task) */
    const taskDate = task['date']
    const taskDescription = task['task']
    /* console.log(taskDescription) */
    /* console.log(taskDate) */
    /* const [taskDescription, taskDate] = task.split(' - '); */

    const currentYear = new Date().getFullYear(); // Get the current year

    // Map of month names to numbers
    const monthMap = {
      "Jan": 0, "Feb": 1, "Mar": 2, "Apr": 3, "May": 4, "Jun": 5,
      "Jul": 6, "Aug": 7, "Sep": 8, "Oct": 9, "Nov": 10, "Dec": 11
    };

    // Split the date string into day and month parts
    const [day, month] = taskDate.split(" "); // ["05", "Dec"]

    // Format the date as yyyy-mm-dd
    const formattedDate = `${currentYear}-${monthMap[month]+1}-${day.padStart(2,
"0")}`;

    // Create a Date object
    const taskDateObj = new Date(formattedDate);
    /* console.log(taskDateObj) */



    // Hide task if it's completed and the date is before today
    if (taskDateObj < previousDay && taskDescription.startsWith('✓ ')) {
      /* console.log(`Hiding task: ${taskDescription} ${taskDate}`); */
```

```
      if (remainderElements[index]) {
        remainderElements[index].classList.add('d-none'); // Hide the task in the
DOM
      } else {
        console.error(`Task element not found for index: ${index}`);
      }
    }
  });
}
/* displayTaskList(); */
const motivationalQuotes = [
  "Keep pushing forward!",
  "You can do it!",
  "Stay focused and never give up!",
  "Believe in yourself!",
  "Every step counts!",
  "Success is just around the corner!",
  "Stay positive and work hard!",
  "Your efforts will pay off!",
  "Keep going, you're doing great!",
  "Stay motivated and achieve your goals!",
  "Dream big and dare to fail.",
  "The harder you work for something, the greater you'll feel when you achieve
it.",
  "Don't stop when you're tired. Stop when you're done.",
  "Wake up with determination. Go to bed with satisfaction.",
  "Do something today that your future self will thank you for.",
  "Little things make big days.",
  "It's going to be hard, but hard does not mean impossible.",
  "Don't wait for opportunity. Create it.",
  "Sometimes we're tested not to show our weaknesses, but to discover our
strengths.",
  "The key to success is to focus on goals, not obstacles.",
  "Dream it. Wish it. Do it.",
  "Success doesn't just find you. You have to go out and get it.",
  "The harder you work, the luckier you get.",
  "Don't watch the clock; do what it does. Keep going.",
  "Great things never come from comfort zones.",
  "Success is not for the lazy.",
  "The way to get started is to quit talking and begin doing.",
  "The only limit to our realization of tomorrow is our doubts of today.",
  "The future belongs to those who believe in the beauty of their dreams.",
  "The secret of getting ahead is getting started.",
  "It always seems impossible until it's done.",
  "Don't let yesterday take up too much of today.",
  "You learn more from failure than from success. Don't let it stop you. Failure
builds character.",
  "It's not whether you get knocked down, it's whether you get up.",
  "If you are working on something that you really care about, you don't have to
be pushed. The vision pulls you.",
  "People who are crazy enough to think they can change the world, are the ones
who do.",
  "We may encounter many defeats but we must not be defeated.",
  "Knowing is not enough; we must apply. Wishing is not enough; we must do.",
  "Imagine your life is perfect in every respect; what would it look like?",
  "We generate fears while we sit. We overcome them by action.",
```

```
  "Whether you think you can or think you can’t, you’re right.",
  "Security is mostly a superstition. Life is either a daring adventure or
nothing.",
  "The man who has confidence in himself gains the confidence of others.",
  "The only way to do great work is to love what you do.",
  "The only limit to our realization of tomorrow is our doubts of today.",
  "Creativity is intelligence having fun.",
  "What you lack in talent can be made up with desire, hustle and giving 110% all
the time.",
  "Do what you can with all you have, wherever you are.",
  "Develop an ‘Attitude of Gratitude’. Say thank you to everyone you meet for
everything they do for you."
];

function getRandomQuote() {
  const randomIndex = Math.floor(Math.random() * motivationalQuotes.length);
  return motivationalQuotes[randomIndex];
}

function sendNotification(message) {
  const quote = getRandomQuote();
  const fullMessage = `${message}\n\n${quote}`;
  /* console.log(`Sending notification: ${fullMessage}`); */

  if (Notification.permission === "granted") {
      new Notification(fullMessage, {
          icon: "./images/icon.png",
      });
  } else if (Notification.permission !== "denied") {
      Notification.requestPermission().then(permission => {
          if (permission === "granted") {
              const notification = new Notification(fullMessage, {
                  icon: "icon.png",
              });

              // Use 'onclick' to navigate to the desired link when the
notification is clicked
              notification.onclick = () => {
                  window.location.href = "index.html"; // Redirect to index.html
              };
          }
      });
  }
}

function scheduleNotifications() {
  const now = new Date();
  const times = [
    { hour: 8, minute: 0, message: "Good morning! Here are your tasks for today."
},
    { hour: 12, minute: 0, message: "Midday check-in! How are you doing with your
tasks?" },
    { hour: 16, minute: 0, message: "Afternoon reminder! Don't forget to complete
your tasks." },
    { hour: 20, minute: 0, message: "Evening summary! Here's what you accomplished
today." }
```

```
  ];

  times.forEach(({ hour, minute, message }) => {
    const targetTime = new Date(now.getFullYear(), now.getMonth(), now.getDate(),
hour, minute, 0, 0);
    if (targetTime < now) {
      targetTime.setDate(targetTime.getDate() + 1); // Schedule for the next day
if the time has already passed
    }
    const delay = targetTime - now;
    /* console.log(`Scheduling notification for ${targetTime} with delay
${delay}ms`); */
    setTimeout(() => {
      sendNotification(message);
      setInterval(() => sendNotification(message), 24 * 60 * 60 * 1000); // Repeat
daily
    }, delay);
  });
}

document.addEventListener('DOMContentLoaded', () => {
  if ("Notification" in window) {
    Notification.requestPermission().then(permission => {
      if (permission === "granted") {
        /* alert("Notification permission granted."); */
        scheduleNotifications();
      } else {
        alert("Notification permission denied.");
      }
    });
  } else {
    alert("Notifications are not supported in this browser.");
  }
  /* displayTaskList(); */
});
```

Step-by-Step Explanation of the Code

This code is a JavaScript program that helps in managing and prioritizing tasks using a machine learning model. It loads a dataset, processes it, trains a model, and then uses the model to predict task priorities. Below is a detailed explanation of each part of the code.

1. Loading the Dataset

The first step is to load the dataset from a CSV file.

```
async function loadPrioritizationCSV() {
```

```
  const csvPath = "./datasets/task_management.csv";
  const dataset = tf.data.csv(csvPath);
  return dataset;
}
```

- **Purpose**: This function loads a CSV file containing task data.
- **Explanation**:
  - csvPath is the path to the CSV file.
  - tf.data.csv(csvPath) loads the CSV file using TensorFlow.js.
  - The function returns the dataset.

---

2. Tokenizing Text

Tokenization is the process of splitting text into individual words or tokens.

```
function tokenizeText(text) {
  if (!text) return [];
  const trimmedText = text.trim();
  const tokens = trimmedText.toLowerCase().replace(/[.,!?]/g, ' ').split(/\s/);
  return tokens;
}
```

- **Purpose**: This function splits the text into words and removes punctuation.
- **Explanation**:
  - trimmedText removes extra spaces from the text.
  - toLowerCase() converts the text to lowercase.
  - replace(/[.,!?]/g, ' ') removes punctuation.
  - split(/\s/) splits the text into words based on spaces.

---

3. Processing the Dataset

This function processes the dataset by tokenizing the task descriptions and keywords.

```
async function processDataset() {
  const dataset = await loadPrioritizationCSV();
  const transformedDataset = dataset.map(record => {
    return {
      ...record,
      Task_Description: tokenizeText(record.Task_Description),
      Keywords: tokenizeText(record.Keywords),
    };
  });
  return transformedDataset;
```

```
}
```

- **Purpose**: This function processes the dataset by tokenizing the Task_Description and Keywords fields.
- **Explanation**:
  - loadPrioritizationCSV() loads the dataset.
  - dataset.map() applies the tokenizeText function to each record.

4. Creating a Combined Vocabulary

This function creates a combined vocabulary of all unique tokens from the task descriptions and keywords.

```
async function createCombinedVocabulary() {
  const transformedDataset = await processDataset();
  const datasetArray = await transformedDataset.toArray();
  const allTokens = datasetArray.flatMap(row => row.Task_Description.concat(row.
Keywords));
  const uniqueTokens = [...new Set(allTokens)];
  return uniqueTokens;
}
```

- **Purpose**: This function creates a list of unique words (vocabulary) from the dataset.
- **Explanation**:
  - transformedDataset.toArray() converts the dataset into an array.
  - flatMap() combines all tokens from Task_Description and Keywords.
  - new Set(allTokens) removes duplicate tokens.

5. Creating Bag of Words (BoW) Vectors

This function creates a BoW vector for a given set of tokens.

```
async function createBoWVector(tokens, vocabulary) {
  const vector = new Array(vocabulary.length).fill(0);
  tokens.forEach(token => {
    const index = vocabulary.indexOf(token);
    if (index !== -1) {
      vector[index] += 1;
    }
  });
  return vector;
}
```

- **Purpose**: This function converts tokens into a numerical vector (BoW).

- **Explanation**:
  - vector is an array of zeros with the same length as the vocabulary.
  - For each token, the corresponding index in the vector is incremented.

6. Creating BoW Matrices

This function creates BoW matrices for task descriptions and keywords.

```
async function createDescriptionBoWMatrix() {
  const vocabulary = await createCombinedVocabulary();
  const transformedDataset = await processDataset();
  const datasetArray = await transformedDataset.toArray();
  const bowMatrix = await Promise.all(
    datasetArray.map(row => {
      return createBoWVector(row.Task_Description, vocabulary);
    })
  );
  return bowMatrix;
}
```

- **Purpose**: This function creates a BoW matrix for task descriptions.
- **Explanation**:
  - createCombinedVocabulary() gets the vocabulary.
  - datasetArray.map() creates a BoW vector for each task description.

7. Normalizing Numerical Columns

This function normalizes the Due_Days column.

```
async function normalizeNumericalColumns() {
  const transformedDataset = await processDataset();
  const datasetArray = await transformedDataset.toArray();
  const dueDays = datasetArray.map(row => row.Due_Days);
  const dueDaysTensor = tf.tensor1d(dueDays);
  const { mean, variance } = tf.moments(dueDaysTensor);
  const std = tf.sqrt(variance).dataSync()[0];
  const meanValue = mean.dataSync()[0];
  const normalizedDueDays = dueDays.map(value => (value - meanValue) / std);
  return { normalizedDueDays, meanValue, std };
}
```

- **Purpose**: This function normalizes the Due_Days column using mean and standard deviation.
- **Explanation**:

- tf.tensor1d(dueDays) converts the Due_Days array into a TensorFlow tensor.
- tf.moments() calculates the mean and variance.
- normalizedDueDays is the normalized array.

8. Creating Input and Output Matrices

This function creates the input matrix for the model.

```
async function createInputMatrix() {
  const descriptionBoWMatrix = await createDescriptionBoWMatrix();
  const keywordsBoWMatrix = await createKeywordsBoWMatrix();
  const { normalizedDueDays } = await normalizeNumericalColumns();
  const inputMatrix = descriptionBoWMatrix.map((row, index) => {
    return row.concat(keywordsBoWMatrix[index]).concat(normalizedDueDays[inde
x]);
  });
  return inputMatrix;
}
```

- **Purpose**: This function combines BoW matrices and normalized Due_Days into a single input matrix.
- **Explanation**:
  - descriptionBoWMatrix and keywordsBoWMatrix are combined with normalizedDueDays.

9. Splitting Data into Training and Testing Sets

This function splits the data into training and testing sets.

```
async function splitData() {
  const inputMatrix = await createInputMatrix();
  const outputMatrix = await createOutputMatrix();
  const trainingDataSize = Math.floor(inputMatrix.length * 0.8);
  const trainingInput = inputMatrix.slice(0, trainingDataSize);
  const trainingOutput = outputMatrix.slice(0, trainingDataSize);
  const testingInput = inputMatrix.slice(trainingDataSize);
  const testingOutput = outputMatrix.slice(trainingDataSize);
  return { trainingInput, trainingOutput, testingInput, testingOutput };
}
```

- **Purpose**: This function splits the data into 80% training and 20% testing sets.
- **Explanation**:
  - slice() is used to split the data.

10. Creating and Training the Model

This function creates and trains a neural network model.

```
async function createModelEnhanced(inputShape) {
  const model = tf.sequential();
  model.add(tf.layers.dense({
    units: 64,
    activation: "relu",
    inputShape: [inputShape],
    kernelRegularizer: tf.regularizers.l2({ l2: 0.02 })
  }));
  model.add(tf.layers.batchNormalization());
  model.add(tf.layers.dropout({ rate: 0.5 }));
  model.add(tf.layers.dense({ units: 5, activation: "softmax" }));
  model.compile({
    optimizer: tf.train.adam(0.002),
    loss: "sparseCategoricalCrossentropy",
    metrics: ["accuracy"],
  });
  return model;
}
```

- **Purpose**: This function creates a neural network model with dense layers, batch normalization, and dropout.
- **Explanation**:
  - tf.sequential() creates a sequential model.
  - model.add() adds layers to the model.
  - model.compile() compiles the model with an optimizer, loss function, and metrics.

11. Predicting Task Priority

This function predicts the priority of a task using the trained model.

```
async function predictTaskPriority(model, description, dueDays) {
  const keywords = extractKeywords(description);
  const tokenizedDescription = tokenizeText(description);
  const tokenizedKeywords = keywords;
  const vocabulary = await createCombinedVocabulary();
  const descriptionBoW = await createBoWVector(tokenizedDescription, vocabulary
);
  const keywordsBoW = await createBoWVector(tokenizedKeywords, vocabulary);
  const { meanValue, std } = await normalizeNumericalColumns();
  const normalizedDueDays = ((Number(dueDays)) - meanValue) / std;
```

```
  const inputVector = descriptionBoW.concat(keywordsBoW).concat(normalizedDu
eDays);
  const inputTensor = tf.tensor2d([inputVector]);
  const prediction = model.predict(inputTensor);
  const predictedClass = prediction.argMax(-1).dataSync()[0] + 1;
  return predictedClass;
}
```

- **Purpose**: This function predicts the priority of a task.
- **Explanation**:
  - extractKeywords() extracts keywords from the task description.
  - createBoWVector() creates BoW vectors.
  - normalizeNumericalColumns() normalizes the Due_Days.
  - model.predict() predicts the priority.

---

12. Scheduling Notifications

This function schedules motivational notifications.

javascript

Copy

```
function scheduleNotifications() {
  const now = new Date();
  const times = [
    { hour: 8, minute: 0, message: "Good morning! Here are your tasks for today." },
    { hour: 12, minute: 0, message: "Midday check-in! How are you doing with your t
asks?" },
    { hour: 16, minute: 0, message: "Afternoon reminder! Don't forget to complete yo
ur tasks." },
    { hour: 20, minute: 0, message: "Evening summary! Here's what you accomplishe
d today." }
  ];
  times.forEach(({ hour, minute, message }) => {
    const targetTime = new Date(now.getFullYear(), now.getMonth(), now.getDate(),
 hour, minute, 0, 0);
    if (targetTime < now) {
      targetTime.setDate(targetTime.getDate() + 1);
    }
    const delay = targetTime - now;
    setTimeout(() => {
      sendNotification(message);
      setInterval(() => sendNotification(message), 24 * 60 * 60 * 1000);
    }, delay);
  });
}
```

- **Purpose**: This function schedules notifications at specific times.

- **Explanation**:
  - times contains the times and messages for notifications.
  - setTimeout() schedules the notifications.

13. Displaying Tasks

This function displays the tasks in the UI.

javascript

Copy

```
async function displayTaskList() {
  const smartListContainer = document.getElementById('smart-remainders-container
');
  const tasks = await getTasksFromLocalStorage();
  const tasksPriority = JSON.parse(localStorage.getItem('prioritizedTasks'));
  const tasksWithPriority = tasksPriority.map(task => {
    const [taskDescription, taskDate, priority] = task.split(' - ');
    return { task: taskDescription, date: taskDate, priority: parseInt(priority, 10) };
  });
  tasksWithPriority.sort((a, b) => a.priority - b.priority);
  smartListContainer.innerHTML = '';
  tasksWithPriority.forEach(({ task, priority }) => {
    const remainderItem = document.createElement('div');
    remainderItem.className = 'input-group remainder-item';
    const taskInput = document.createElement('input');
    taskInput.type = 'text';
    taskInput.className = 'form-control remainder-text mb-3';
    taskInput.value = task;
    taskInput.disabled = true;
    remainderItem.appendChild(taskInput);
    smartListContainer.appendChild(remainderItem);
  });
}
```

- **Purpose**: This function displays the tasks in the UI.
- **Explanation**:
  - getTasksFromLocalStorage() retrieves tasks from local storage.
  - tasksWithPriority.sort() sorts tasks by priority.
  - document.createElement() creates HTML elements to display tasks.

14. Saving and Loading the Model

This function saves and loads the trained model.

```
async function saveModel() {
  const model = await testModel();
  await model.save('localstorage://my-model');
  await model.save('downloads://my-prioritization-model');
  console.log('Model saved successfully');
}

async function loadModel() {
  const model = await tf.loadLayersModel('./models/my-prioritization-model.json');
  console.log('Model loaded successfully');
  return model;
}
```

- **Purpose**: These functions save and load the trained model.
- **Explanation**:
  - model.save() saves the model.
  - tf.loadLayersModel() loads the model.

15. Main Execution

The main execution starts when the DOM is loaded.

```
document.addEventListener('DOMContentLoaded', () => {
  if ("Notification" in window) {
    Notification.requestPermission().then(permission => {
      if (permission === "granted") {
        scheduleNotifications();
      } else {
        alert("Notification permission denied.");
      }
    });
  } else {
    alert("Notifications are not supported in this browser.");
  }
});
```

- **Purpose**: This code runs when the page loads.
- **Explanation**:
  - Notification.requestPermission() requests permission for notifications.
  - scheduleNotifications() schedules notifications if permission is granted.

Summary

This code is a comprehensive task management system that:

1. Loads and processes a dataset.
2. Trains a machine learning model to predict task priorities.
3. Displays tasks in the UI.
4. Schedules motivational notifications.

Each function is modular and performs a specific task, making the code easy to understand and maintain.

7.taskCategorization.js (for categorizing the tasks based on the description):

```
async function loadCategorizationCSV() {
    const csvPath =
"./datasets/Updated_Task_categorization_dataset_fixed_work.csv";
    const dataset = tf.data.csv(csvPath);
    return dataset;
}
/* //log the dataset
async function logCategorizationCSV()
{
    const dataset = await loadCategorizationCSV();
    const records = await dataset.toArray();
    console.log(records);
}
logCategorizationCSV(); */

function tokenizeTextFormats(text) {
    if (typeof text !== 'string') {
      throw new TypeError('Expected a string');
    }
    return text.toLowerCase().split(/\W+/).filter(token => token.length > 0);
  }

async function tokenizedDataset() {
    const dataset = await loadCategorizationCSV()

    const transformedCategorizedDataset = dataset.map(record => {
        return {
            ...record,
            Task_Description : tokenizeTextFormats(record.Task_Description),
            Keywords: tokenizeTextFormats(record.Keywords),
            Category : tokenizeTextFormats(record.Category),
        }
    });
    return transformedCategorizedDataset;
}
/* //log tokenizedDataset
async function logTokenizedDataset() {
    const dataset = await tokenizedDataset()
    const datasetArray = await dataset.toArray()
    console.log(datasetArray)
```

```
}
logTokenizedDataset()
 */
async function descriptionVocabulary()
{
    const dataset = await tokenizedDataset();
    const datasetArray = await dataset.toArray();

    const Tokens = datasetArray.flatMap(row => row.Task_Description);

    const uniquedescriptionTokens = [...new Set(Tokens)];
    return uniquedescriptionTokens;
}
/* //log descriptionVocabulary
async function logdescriptionVocabulary() {
    const dataset = await descriptionVocabulary()
    console.log(dataset)
}
logdescriptionVocabulary() */
async function keywordsVocabulary()
{
    const dataset = await tokenizedDataset();
    const datasetArray = await dataset.toArray();

    const Tokens = datasetArray.flatMap(row => row.Keywords);

    const uniqueKeywordsTokens = [...new Set(Tokens)];
    return uniqueKeywordsTokens;
}
/* //log keywordsVocabulary
async function logKeywordsVocabulary() {
    const dataset = await keywordsVocabulary()
    console.log(dataset)
}
logKeywordsVocabulary() */

async function categoryVocabulary()
{
    const dataset = await tokenizedDataset();
    const datasetArray = await dataset.toArray();

    const Tokens = datasetArray.flatMap(row => row.Category)

    const uniqueCategoryTokens =  [...new Set(Tokens)];
    return uniqueCategoryTokens;
}
/* //log categoryVocabulary
async function logcategoryVocabulary() {
    const dataset = await categoryVocabulary()
    console.log(dataset)
}
logcategoryVocabulary() */
// Function to create Bag of Words vector
async function BoWVector(tokens, vocabulary) {
    const vector = new Array(vocabulary.length).fill(0);  // Initialize vector
with zeros
```

```
    tokens.forEach(token => {
      const index = vocabulary.indexOf(token);
      if (index !== -1) {
        vector[index] += 1; // Increment the count
      }
    });
    return vector; // Return the BoW vector
}

async function descriptionBowVector() {
    const vocabulary = await descriptionVocabulary();
    const transformedDataset = await tokenizedDataset();
    const dataset = await transformedDataset.toArray();
    const bowMatrix = await Promise.all(
        dataset.map(row => {
            return BoWVector(row.Task_Description,vocabulary);
        })
    );
    return bowMatrix;
}
/* //log descriptionBowVector
async function logDescriptionBowVector() {
    const bowMatrix = await descriptionBowVector();
    console.log(bowMatrix);
}
logDescriptionBowVector(); */
async function keywordsBowVector() {
    const vocabulary = await keywordsVocabulary();
    const transformedDataset = await tokenizedDataset();
    const dataset = await transformedDataset.toArray();
    const bowMatrix = await Promise.all(
        dataset.map(row => {
            return BoWVector(row.Keywords,vocabulary);
        })
    );
    return bowMatrix;
}
/* //log keywordsBowVector
async function logKeywordsBowVector() {
    const bowMatrix = await keywordsBowVector();
    console.log(bowMatrix);
}
logKeywordsBowVector(); */

async function categoryBowVector() {
    const vocabulary = await categoryVocabulary();
    const transformedDataset = await tokenizedDataset();
    const dataset = await transformedDataset.toArray();
    const bowMatrix = await Promise.all(
        dataset.map(row => {
            return BoWVector(row.Category,vocabulary);
        })
    );
    return bowMatrix;
}
```

```
/* //log categoryBowVector
async function logcategoryBowVector() {
    const bowMatrix = await categoryBowVector();
    console.log(bowMatrix);
}
logcategoryBowVector(); */
// Consolidated input feature matrix creation
async function inputMatrix() {
    const descriptionBoWMatrix = await descriptionBowVector();
    const keywordsBoWMatrix = await keywordsBowVector();

    const inputMatrix = descriptionBoWMatrix.map((row,index) => {
        return row.concat(keywordsBoWMatrix[index])
    });

    return inputMatrix;
}
/* //log the input matrix
async function logInputMatrix() {
    const inputCategorizationMatrix = await inputMatrix();
    console.log(inputCategorizationMatrix);
}
logInputMatrix(); */

async function outputMatrix() {
    const categoryBoWMatrix = await categoryBowVector();

    const outputMatrix = categoryBoWMatrix;
    return outputMatrix;
}
/* //log the output matrix
async function logOutputMatrix() {
    const outputCategorizationMatrix = await outputMatrix();
    console.log(outputCategorizationMatrix);
}
logOutputMatrix(); */

// Splitting the data into training and testing
async function splitCategorizationData() {
  const inputCategorizationMatrix = await inputMatrix();
  const outputCategorizationMatrix = await outputMatrix();

  // Determine the size of the training set (80% of the total data)
  const totalSize = inputCategorizationMatrix.length;
  const trainingSize = Math.floor(totalSize * 0.8);

  // Split the data into training and testing sets
  const trainingInput = inputCategorizationMatrix.slice(0, trainingSize);
  const trainingOutput = outputCategorizationMatrix.slice(0, trainingSize);
  const testingInput = inputCategorizationMatrix.slice(trainingSize);
  const testingOutput = outputCategorizationMatrix.slice(trainingSize);

  return { trainingInput, trainingOutput, testingInput, testingOutput };
}
/* //log the split data
async function logSplitCategorizationData() {
```

```
    const { trainingInput, trainingOutput, testingInput, testingOutput } = await
splitCategorizationData();
    console.log('Training Input:', trainingInput);
    console.log('Training Output:', trainingOutput);
    console.log('Testing Input:', testingInput);
    console.log('Testing Output:', testingOutput);
}
logSplitCategorizationData(); */

async function createCategorizationModel(inputShape) {
    const model = tf.sequential();

    // Add dense layers with L2 regularization and batch normalization
    model.add(tf.layers.dense({
      units: 64,
      activation: "relu",
      inputShape: [inputShape],
      kernelRegularizer: tf.regularizers.l2({ l2: 0.01 })
    }));
    model.add(tf.layers.batchNormalization());
    model.add(tf.layers.dropout({ rate: 0.5 }));

    model.add(tf.layers.dense({
      units: 64,
      activation: "relu",
      kernelRegularizer: tf.regularizers.l2({ l2: 0.01 })
    }));
    model.add(tf.layers.batchNormalization());
    model.add(tf.layers.dropout({ rate: 0.5 }));

    model.add(tf.layers.dense({
      units: 32,
      activation: "relu",
      kernelRegularizer: tf.regularizers.l2({ l2: 0.01 })
    }));
    model.add(tf.layers.batchNormalization());
    model.add(tf.layers.dropout({ rate: 0.5 }));


    model.add(tf.layers.dense({ units: 5, activation: "softmax" }));

    model.compile({
      optimizer: tf.train.adam(0.005),  // Adjusted learning rate
      loss: "categoricalCrossentropy",
      metrics: ["accuracy"]
    });

    return model;
  }
    /* trainCategorizationModel(); */
   async function saveCategorizationModel() {
      const model = await testCategorizationModel();
      await model.save('localstorage://my-categorization-model');
      await model.save('downloads://my-categorization-model');
      console.log('Model saved successfully');
  }
```

```
  /* saveCategorizationModel(); */

async function loadCategorizationModel() {
    const categorizationModel = await tf.loadLayersModel('./models/my-
categorization-model.json');
    /* console.log('Model loaded successfully'); */
    return categorizationModel;
}


  async function trainCategorizationModel() {
    const { trainingInput, trainingOutput } = await splitCategorizationData();
    const model = await createCategorizationModel(trainingInput[0].length);

    const trainingInputTensor = tf.tensor2d(trainingInput, [trainingInput.length,
trainingInput[0].length]);
    const trainingOutputTensor =
tf.tensor2d(trainingOutput,[trainingOutput.length, trainingOutput[0].length]);

    await model.fit(trainingInputTensor, trainingOutputTensor, {
      epochs: 33,
      batchSize: 32,
      validationSplit: 0.2,
      callbacks: [
        {
          onEpochEnd: async (epoch, logs) => {
            console.log(`Epoch: ${epoch} - loss: ${logs.loss} - accuracy:
${logs.acc}`);
          },
        },
      ],
    });

    return model;
  }

async function testCategorizationModel() {
  const model = await trainCategorizationModel();
  const { testingInput, testingOutput } = await splitCategorizationData();

  const testingInputTensor = tf.tensor2d(testingInput, [testingInput.length,
testingInput[0].length]);
  const testingOutputTensor = tf.tensor2d(testingOutput,[testingOutput.length,
testingOutput[0].length]);

  const result = model.evaluate(testingInputTensor, testingOutputTensor);
  const loss = result[0].dataSync()[0];
  const accuracy = result[1].dataSync()[0];

  console.log(`Test Loss: ${loss}`);
  console.log(`Test Accuracy: ${accuracy}`);

  return model;
}
/* testCategorizationModel(); */
// Function to extract keywords from the task description
```

```
function extractCategorizationKeywords(description) {
    const tokens = tokenizeTextFormats(description);
    const keywordFrequency = {};

    tokens.forEach(token => {
      if (keywordFrequency[token]) {
        keywordFrequency[token]++;
      } else {
        keywordFrequency[token] = 1;
      }
    });

    // Sort tokens by frequency and select the top N keywords
    const sortedCategorizationKeywords = Object.keys(keywordFrequency).sort((a, b)
=> keywordFrequency[b] - keywordFrequency[a]);
    /* console.log(sortedKeywords); */
    return sortedCategorizationKeywords;
  }
  async function predictCategory(categorizationModel, description) {
    const descriptionVocab = await descriptionVocabulary();
    const keywordsVocab = await keywordsVocabulary();
    const descriptionCategorizationTokens = tokenizeTextFormats(description);
    const categorizationKeywords = extractCategorizationKeywords(description);
    /* console.log(categorizationKeywords); */
    /* console.log(descriptionCategorizationTokens);
    console.log('Description Vocab Length:', descriptionVocab.length);
    console.log('Keywords Vocab Length:', keywordsVocab.length); */

    const descriptionVector = await BoWVector(descriptionCategorizationTokens,
descriptionVocab);
    const keywordsVector = await BoWVector(categorizationKeywords, keywordsVocab);
    /* console.log('Description BoW Vector:', descriptionVector);
    console.log('Keywords BoW Vector:', keywordsVector); */
    const inputCategorizationVector = descriptionVector.concat(keywordsVector);
    /* console.log('Input Vector Length:', inputCategorizationVector.length);
    console.log('Input Vector:', inputCategorizationVector); */

    const inputCategorizationTensor = tf.tensor2d([inputCategorizationVector], [1,
inputCategorizationVector.length]);

    const prediction = categorizationModel.predict(inputCategorizationTensor);
    const categoryVocab = await categoryVocabulary();
    const categoryIndex = prediction.argMax(1).dataSync()[0];
    /* console.log('Predicted Category:', categoryVocab[categoryIndex]); */
    /* console.log('Prediction Scores:', prediction.dataSync()); */
    return categoryVocab[categoryIndex];
  }

  async function predictCategorizationModel() {
    const categorizationModel = await loadCategorizationModel();
    const categories = [];
    const tasks = await getTasksFromLocalStorage();
    for (const task of tasks) {
      const [taskDescription, taskDate] = task.split(' - ');
      /* console.log('Task Date:', taskDate); */
      const category= await predictCategory(categorizationModel, taskDescription);
```

```
      const taskCategory = category.charAt(0).toUpperCase() + category.slice(1);
      /* console.log('Task Category:', taskCategory); */
      categories.push({taskDescription, taskDate, tasksCategory: taskCategory});
    }
    /* console.log('Categorized Tasks:', categories); */
    return categories;
  }
  async function saveTaskCategories()
  {
    const categories = await predictCategorizationModel();
    const categorizedTasks = categories.map(({ taskDescription,taskDate,
tasksCategory }) => `${taskDescription} - ${taskDate} - ${tasksCategory}`);
    // Sort tasks: not completed tasks (without "✓") at the top
    categorizedTasks.sort((a, b) => {
      const isACompleted = a.startsWith('✓');
      const isBCompleted = b.startsWith('✓');
      return isACompleted - isBCompleted;
    });

    localStorage.setItem('categorizedTasks', JSON.stringify(categorizedTasks));
    /* console.log('Categorized tasks saved to local storage:', categorizedTasks);
*/
    await displayCategorizedTasks();
  }
  saveTaskCategories();
  async function displayCategorizedTasks() {
    const categories = JSON.parse(localStorage.getItem('categorizedTasks'));
    const categorizedTasks = categories.map(category => {
      const [taskDescription, taskDate, taskCategory] = category.split(' - ');
      return {task: taskDescription, date: taskDate, category: taskCategory};
    });
    const categorizedTasksSorting = categorizedTasks.sort((a, b) =>
a.category.localeCompare(b.category));
    /* console.log('Categorized Tasks:', categorizedTasks); */

    const smartCategorization = document.getElementById('smart-categorization-
container');
    smartCategorization.innerHTML = '';

    categorizedTasksSorting.forEach(({ task, category }) => {
      if(task.trim()===" ") return;
      const categoryItem = document.createElement('div');
      categoryItem.className = 'input-group category-item';
      const categoryTask = document.createElement('input');
      categoryTask.disabled = true;
      categoryTask.type = 'text';
      categoryTask.value = task;
      categoryTask.className = 'form-control category-description mb-3';
      categoryTask.style.width = '60%';
      categoryTask.style.backgroundColor = '#dee2ff';
      categoryItem.appendChild(categoryTask);
      const categoryElement = document.createElement('input');
      categoryElement.disabled = true;
      categoryElement.type = 'text';
      categoryElement.value = category;
      categoryElement.className = 'form-control category-text mb-3';
```

```
      categoryElement.style.fontWeight = 'bold';
      if(category === 'Personal') {
        categoryElement.style.backgroundColor = '#e9edc9';
      }
      else if(category === 'Work') {
        categoryElement.style.backgroundColor = '#ffccd5';
      }
      else if(category === 'Education') {
        categoryElement.style.backgroundColor = '#d8e2dc';
      }
      else if(category === 'Health') {
        categoryElement.style.backgroundColor = '#bee9e8';
      }
      else if(category === 'Household') {
        categoryElement.style.backgroundColor = '#e0fbfc';
      }
      categoryItem.appendChild(categoryElement);
      smartCategorization.appendChild(categoryItem);
    });
  const categoryElements = document.querySelectorAll('.category-item')
  /* console.log(categoryElements) */
  const today = new Date();
  const previousDay = new Date(today);
  previousDay.setDate(today.getDate() - 1);
  previousDay.setHours(11, 59, 59, 999);
  /* console.log(previousDay) */
  categorizedTasksSorting.forEach((task, index) => {
    /* console.log(task) */
    const taskDate = task['date']
    const taskDescription = task['task']
    /* console.log(taskDescription) */
    /* console.log(taskDate) */
    /* const [taskDescription, taskDate] = task.split(' - '); */

    const currentYear = new Date().getFullYear(); // Get the current year

    // Map of month names to numbers
    const monthMap = {
      "Jan": 0, "Feb": 1, "Mar": 2, "Apr": 3, "May": 4, "Jun": 5,
      "Jul": 6, "Aug": 7, "Sep": 8, "Oct": 9, "Nov": 10, "Dec": 11
    };

    // Split the date string into day and month parts
    const [day, month] = taskDate.split(" "); // ["05", "Dec"]

    // Format the date as yyyy-mm-dd
    const formattedDate = `${currentYear}-${monthMap[month]+1}-${day.padStart(2,
"0")}`;

    // Create a Date object
    const taskDateObj = new Date(formattedDate);
    /* console.log(taskDateObj) */

    // Hide task if it's completed and the date is before today
    if (taskDateObj < previousDay && taskDescription.startsWith('✓ ')) {
      /* console.log('Task is completed and the date is before today',task); */
```

```
      if (categoryElements[index]) {
        categoryElements[index].classList.add('d-none'); // Hide the task in the
DOM
      } else {
        console.error(`Task element not found for index: ${index}`);
      }
    }
  });
  }
  /* displayCategorizedTasks(); */
```

1. Loading the Dataset

```
async function loadCategorizationCSV() {
    const csvPath =
"./datasets/Updated_Task_categorization_dataset_fixed_work.csv";
    const dataset = tf.data.csv(csvPath);
    return dataset;
}
```

- **Purpose:** This function loads a CSV file that contains task descriptions, keywords, and their categories using TensorFlow.js (`tf.data.csv`).
- **How it works:**
  - `csvPath` is the path to your CSV file.
  - `tf.data.csv(csvPath)` loads the data from the CSV file into a TensorFlow.js dataset.
  - The function returns the loaded dataset.

---

2. Tokenizing Text (Breaking Text into Words)

```
function tokenizeTextFormats(text) {
    if (typeof text !== 'string') {
        throw new TypeError('Expected a string');
    }
    return text.toLowerCase().split(/\W+/).filter(token => token.length > 0);
}
```

- **Purpose:** Tokenizes text by splitting it into individual words (tokens), removing non-word characters, and converting everything to lowercase.
- **How it works:**
  - `text.toLowerCase()` converts the entire text to lowercase.
  - `split(/\W+/)` splits the text by non-word characters (spaces, punctuation, etc.).
  - `.filter(token => token.length > 0)` removes any empty strings that result from splitting.

---

3. Tokenizing Dataset (For Task Descriptions, Keywords, and Categories)

```
async function tokenizedDataset() {
    const dataset = await loadCategorizationCSV()

    const transformedCategorizedDataset = dataset.map(record => {
        return {
            ...record,
```

```
            Task_Description: tokenizeTextFormats(record.Task_Description),
            Keywords: tokenizeTextFormats(record.Keywords),
            Category: tokenizeTextFormats(record.Category),
        }
    });
    return transformedCategorizedDataset;
}
```

- **Purpose:** This function processes the entire dataset by tokenizing task descriptions, keywords, and categories.
- **How it works:**
  - `dataset.map()` applies the `tokenizeTextFormats` function to each record's `Task_Description`, `Keywords`, and `Category`.
  - It creates a new transformed dataset where each text field is tokenized into words.

---

4. Extracting Unique Tokens (Vocabulary)

For Task Descriptions:

```
async function descriptionVocabulary() {
    const dataset = await tokenizedDataset();
    const datasetArray = await dataset.toArray();
    const Tokens = datasetArray.flatMap(row => row.Task_Description);
    const uniquedescriptionTokens = [...new Set(Tokens)];
    return uniquedescriptionTokens;
}
```

- **Purpose:** To get a list of unique words (vocabulary) from the task descriptions.
- **How it works:**
  - `flatMap(row => row.Task_Description)` collects all the task descriptions into a single array of tokens.
  - `new Set(Tokens)` creates a unique set of words.
  - `[...]` spreads the set into an array.

For Keywords:

```
async function keywordsVocabulary() {
    const dataset = await tokenizedDataset();
    const datasetArray = await dataset.toArray();
    const Tokens = datasetArray.flatMap(row => row.Keywords);
    const uniqueKeywordsTokens = [...new Set(Tokens)];
    return uniqueKeywordsTokens;
}
```

- **Purpose:** To get a list of unique words from the keywords field.
- **How it works:** Similar to `descriptionVocabulary()`, but this time, it works with the `Keywords` field.

For Categories:

```
async function categoryVocabulary() {
    const dataset = await tokenizedDataset();
    const datasetArray = await dataset.toArray();
    const Tokens = datasetArray.flatMap(row => row.Category);
    const uniqueCategoryTokens = [...new Set(Tokens)];
    return uniqueCategoryTokens;
}
```

- **Purpose:** To get a list of unique words from the categories.
- **How it works:** Similar to the previous functions but focused on the `Category` field.

---

5. Bag of Words (BoW) Vector

```
async function BoWVector(tokens, vocabulary) {
    const vector = new Array(vocabulary.length).fill(0);  // Initialize vector
with zeros

    tokens.forEach(token => {
        const index = vocabulary.indexOf(token);
        if (index !== -1) {
            vector[index] += 1; // Increment the count
        }
    });
    return vector; // Return the BoW vector
}
```

- **Purpose:** Converts a set of tokens (e.g., task description, keywords) into a vector based on the vocabulary.
- **How it works:**
  - It creates a vector of zeros of length equal to the vocabulary size.
  - It increments the index of the vocabulary that matches each token in the input.

---

6. Creating the Input Matrix (Combining BoW Vectors)

```
async function inputMatrix() {
    const descriptionBoWMatrix = await descriptionBowVector();
    const keywordsBoWMatrix = await keywordsBowVector();

    const inputMatrix = descriptionBoWMatrix.map((row, index) => {
        return row.concat(keywordsBoWMatrix[index])
    });

    return inputMatrix;
}
```

- **Purpose:** Combines the Bag of Words vectors for task descriptions and keywords into a single input matrix.
- **How it works:**
  - `descriptionBoWMatrix.map((row, index) => {...})` takes each row of the description BoW matrix and concatenates it with the corresponding row from the keywords BoW matrix.
  - The result is a combined input matrix that can be fed into a machine learning model.

---

7. Creating the Output Matrix (Category Vectors)

```
async function outputMatrix() {
    const categoryBoWMatrix = await categoryBowVector();
    const outputMatrix = categoryBoWMatrix;
    return outputMatrix;
}
```

- **Purpose:** This function returns the Bag of Words vector for the categories, which serves as the output matrix for the model.
- **How it works:** Simply returns the result of `categoryBowVector()`.

---

8. Splitting Data into Training and Testing

```
async function splitCategorizationData() {
```

```
  const inputCategorizationMatrix = await inputMatrix();
  const outputCategorizationMatrix = await outputMatrix();

  const totalSize = inputCategorizationMatrix.length;
  const trainingSize = Math.floor(totalSize * 0.8);

  const trainingInput = inputCategorizationMatrix.slice(0, trainingSize);
  const trainingOutput = outputCategorizationMatrix.slice(0, trainingSize);
  const testingInput = inputCategorizationMatrix.slice(trainingSize);
  const testingOutput = outputCategorizationMatrix.slice(trainingSize);

  return { trainingInput, trainingOutput, testingInput, testingOutput };
}
```

- **Purpose:** Splits the dataset into training and testing sets (80% training, 20% testing).
- **How it works:**
  - The dataset is split based on the size calculated by multiplying the total size by 0.8.
  - The `slice` method is used to split the dataset into training and testing portions.

---

9. Creating and Training the Model

```
async function createCategorizationModel(inputShape) {
    const model = tf.sequential();

    model.add(tf.layers.dense({
      units: 64,
      activation: "relu",
      inputShape: [inputShape],
      kernelRegularizer: tf.regularizers.l2({ l2: 0.01 })
    }));
    // Additional layers...
    model.compile({
      optimizer: tf.train.adam(0.005),
      loss: "categoricalCrossentropy",
      metrics: ["accuracy"]
    });

    return model;
}
```

- **Purpose:** Creates a neural network model with dense layers to classify the tasks based on their descriptions and keywords.
- **How it works:**
  - `tf.sequential()` creates a sequential model.
  - `tf.layers.dense` adds fully connected layers with 64 units and ReLU activation.
  - The model is compiled with the Adam optimizer and categorical cross-entropy loss function.

---

10. Training the Model

```
async function trainCategorizationModel() {
    const { trainingInput, trainingOutput } = await splitCategorizationData();
    const model = await createCategorizationModel(trainingInput[0].length);

    const trainingInputTensor = tf.tensor2d(trainingInput,
[trainingInput.length, trainingInput[0].length]);
    const trainingOutputTensor =
tf.tensor2d(trainingOutput,[trainingOutput.length, trainingOutput[0].length]);
```

```
    await model.fit(trainingInputTensor, trainingOutputTensor, {
      epochs: 33,
      batchSize: 32,
      validationSplit: 0.2,
    });

    return model;
}
```

- **Purpose:** Trains the model using the training dataset.
- **How it works:**
  - Converts the training data into tensors.
  - The model is trained using `model.fit()` for 33 epochs.

---

11. Making Predictions

```
async function predictCategory(categorizationModel, description) {
    const descriptionVocab = await descriptionVocabulary();
    const keywordsVocab = await keywordsVocabulary();
    const descriptionCategorizationTokens = tokenizeTextFormats(description);
    const categorizationKeywords = extractCategorizationKeywords(description);

    const descriptionVector = await BoWVector(descriptionCategorizationTokens,
descriptionVocab);
    const keywordsVector = await BoWVector(categorizationKeywords,
keywordsVocab);
    const inputCategorizationVector = descriptionVector.concat(keywordsVector);

    const inputCategorizationTensor = tf.tensor2d([inputCategorizationVector],
[1, inputCategorizationVector.length]);

    const prediction = categorizationModel.predict(inputCategorizationTensor);
    const categoryVocab = await categoryVocabulary();
    const categoryIndex = prediction.argMax(1).dataSync()[0];
    return categoryVocab[categoryIndex];
}
```

- **Purpose:** This function uses the trained model to predict the category of a task based on its description and keywords.
- **How it works:**
  - Tokenizes the description and keywords.
  - Creates a Bag of Words vector for both.
  - Combines them and feeds them into the model for prediction.
  - Returns the predicted category.