ChatGPT

Smart Bookmarks: Intelligent Folder Recommendations

Building a **Smart Bookmarks** Chrome extension involves integrating with the browser's bookmarks system and adding AI-driven logic to suggest where to save pages. The extension must use the Chrome Bookmarks API (with the `"bookmarks"` permission) to read/write bookmarks developer.chrome.com , and can leverage other APIs (e.g. `chrome.history`, `chrome.contextMenus`, `chrome.storage`) to gather data and present a UI. Crucially, **intercepting Chrome's native "Add Bookmark" UI is not possible**; instead, the extension provides its own bookmarking interface (e.g. a context-menu item or toolbar button) stackoverflow.com . When the user chooses to "Smart Bookmark" a page, the extension gathers the page content and user data, runs an ML/NLP evaluation, and then suggests existing folders (or creates a new one) for that bookmark. The following sections outline the design, APIs, and techniques for implementing this system.

Bookmark Flow: Custom Triggers

- **Cannot override native bookmark**: Chrome doesn't allow extensions to intercept the default bookmark button/keyboard (e.g. Ctrl+D) or inject UI into it stackoverflow.com . Instead, provide your own bookmark action. For example, use the **chrome.contextMenus API** to add a right-click menu item ("Smart Bookmark this page") developer.chrome.com developer.chrome.com , or add an **extension toolbar button** (browser action) that the user clicks to bookmark.
- **Context menu approach**: In `background.js` , create a menu item with `chrome.contextMenus.create({ title: "Smart Bookmark", contexts: ["page"], id: "smart-bookmark" })` developer.chrome.com . When clicked (`chrome.contextMenus.onClicked`), the extension receives the current `tab` info and can trigger analysis and prompting.
- **Extension icon/popup**: Alternatively, define an `"action"` with a default popup HTML. When the user clicks the extension icon, the popup script can fetch page info and compute folder suggestions. Either way, the extension controls the bookmarking flow from end to end.

Chrome Extension APIs & Permissions

To implement Smart Bookmarks, the extension needs the following permissions and APIs:

- `chrome.bookmarks` **API** (permission `"bookmarks"`) allows reading and modifying the bookmark tree developer.chrome.com. You can retrieve the full hierarchy with `chrome.bookmarks.getTree()` developer.chrome.com, search bookmarks, create folders/bookmarks, and move items. For example, to **create a new folder** under the bookmarks bar:

```js
chrome.bookmarks.create(
  { parentId: bookmarkBarId, title: "New Folder N
  function(newFolder) { console.log("Created fold
);
```

(This pattern is shown in Chrome's docs developer.chrome.com.) To add a new bookmark:

```js
chrome.bookmarks.create(
  { parentId: folderId, title: docTitle, url: pag
  callback
);
```

- `chrome.contextMenus` **API** (permission `"contextMenus"`) lets the extension add items to the right-click menu developer.chrome.com developer.chrome.com. Use it to create a "Smart Bookmark" menu. Context menu items can have submenus, so you could dynamically populate it with suggested folders. (Remember to declare `"contextMenus"` permission in `manifest.json` developer.chrome.com.)

- `chrome.history` **API** (permission `"history"`) can access browsing history developer.chrome.com. For deeper personalization, the extension might query recent visits (`chrome.history.search`) to identify topics the user often visits. However, note that `"history"` permission will prompt a warning to the user.
- `chrome.tabs` **and** `chrome.scripting` **APIs** allow retrieving the current page's title, URL, and content. For example, use `chrome.tabs.query({active: true, currentWindow: true})` to get the current tab, then `chrome.scripting.executeScript` to run a script that extracts `document.body.innerText` or other metadata. Alternatively, declare a content script (`"content_scripts"`) that automatically runs on all pages and sends text to the background via `chrome.runtime.sendMessage`. The Chrome docs note that content scripts "run in the context of web pages" and "can read details of the web pages" via the DOM developer.chrome.com. Use this to grab enough page content (e.g. body text or meta tags) for analysis.
- `chrome.storage` (permission `"storage"`) can cache computed embeddings or settings. If embeddings of folder contents or history are expensive to compute, store them with `chrome.storage.local` to reuse later. Also use storage for user preferences (e.g. which ML model to use, threshold values).
- **Manifest example**: A sample `manifest.json` might include:

```
json                                                    Copy
```

```
{
  "manifest_version": 3,
  "name": "Smart Bookmarks",
  "permissions": ["bookmarks", "contextMenus", "h
  "background": { "service_worker": "background.j
  "action": { "default_popup": "popup.html" },
  "content_scripts": [
    {
      "matches": ["<all_urls>"],
      "js": ["content.js"]
    }
  ]
}
```

(This declares needed permissions like `"bookmarks"` and `"contextMenus"` developer.chrome.com developer.chrome.com .)

Extracting and Analyzing Page Content

Efficient content extraction is key for good recommendations. Options include:

- **Content scripts**: As noted, a content script can grab the page's DOM. For example, `document.title` gives the page title, and `document.body.innerText` or specific element text can provide content. Keep text reasonably short (strip out common boilerplate, adverts, etc.) to speed up processing. The content script can then send this data to the background (`chrome.runtime.sendMessage`), where analysis happens.

- **Page metadata**: Don't overlook `<meta>` tags: descriptions and keywords can be quick signals. You might extract `<meta name="description">` or use Open Graph tags. These are easily obtained via content script (e.g. `document.querySelector('meta[name="description"]').content`).
- **Browser history**: To incorporate user interest, use `chrome.history.search({ text: "", startTime: cutoffTime, maxResults: N }, callback)` to fetch recent sites developer.chrome.com. You can analyze the titles or URLs of, say, the last 50 visited pages to detect trending topics. For privacy, only use this if truly helpful and inform the user.
- **Data to feed ML**: Ultimately, the AI logic will consider at least: the current page's title and text (and possibly URL), the names and/or contents of existing bookmark folders, and possibly topics from browsing history. All these should be converted into a comparable representation (e.g. text embeddings or keywords) for similarity search or classification.

ML/NLP Techniques for Folder Suggestion

Smart Bookmarks uses AI to *match a page to folders* or *create a new folder name*. Approaches include:

- **Semantic similarity (embeddings)**: Compute vector embeddings for the page content (and title) and for each existing folder. A simple method: concatenate a folder's name with the titles (or keywords) of bookmarks in it to form a "folder document." Embed each folder document with a text embedding model (e.g. TensorFlow.js Universal Sentence Encoder or OpenAI's text embeddings API). Then compute cosine similarity between the page's embedding and each folder embedding. Suggest folders with highest scores. If **no folder exceeds a threshold**, decide to create a new folder. TensorFlow.js even has tutorials for running ML in extensions (e.g. image classification with MobileNet) tensorflow.org ; similarly you could load a USE model in the extension.
- **Clustering or classification**: If the user has many bookmarks, group them by topic. For instance, run **k-means clustering** on all bookmark titles/content embeddings. Label each cluster by its most common keywords or use a summary from an LLM. When a new page arrives, find which cluster (topic) it belongs to and place it in the corresponding folder (if user's folders roughly match those clusters). Alternatively, treat existing folders as "classes": use supervised classification by using each bookmark's known folder as a label. A lightweight classifier (like logistic regression on TF-IDF vectors) could be trained periodically in background, but this is complex. Embedding-based nearest-neighbor is simpler and more flexible.

- **LLM / GPT-based suggestions**: A large language model (via API) can directly suggest folder placement or names. For example, send the page title/content and list existing folder names as context, and ask "Which folder is most appropriate for this page? If none match, suggest a new folder name." The model (e.g. GPT-4) can read context and output a recommendation. Example prompt:

  > **User:** I have these bookmark folders: **"Recipes", "Work Projects", "Travel", "Technology".**
  > **Page Title:** *"Best practices in web security"*
  > **Page Snippet:** *"This article covers web security fundamentals, secure coding, and OWASP guidelines."*
  > **Q:** Which folder fits this page, or suggest a new folder name?

  The LLM might reply: *"New folder: "Cybersecurity"."* or suggest "Technology" if we choose an existing one. This approach can leverage modern LLMs' semantic understanding (especially GPT-4's) to handle nuance and naming. However, sending data to a cloud API has privacy and cost implications, so give users an option or use content-based anonymization.

- **Keyword extraction / summarization**: A simpler offline method is to extract top keywords (e.g. via TF-IDF or a JS keyword library) from the page. For example, pick the top 2–3 unique nouns. Use those words to search among folder names or generate a title for a new folder. E.g., if keywords are "quantum" and "computing," either pick the "Technology" folder or create a folder named "Quantum Computing."

- **Model choices**: For on-device embedding, use TensorFlow.js models (like Universal Sentence Encoder) or light transformers (e.g. DistilBERT in tfjs). For cloud options, OpenAI's *text-embedding-3-small* or *text-embedding-3-large* can embed text, and GPT-3.5/GPT-4 can classify or name topics. Hugging Face Inference API also offers text classification/embedding endpoints. The exact model depends on performance constraints.

Implementation Architecture

Putting it all together, a high-level architecture:

1. **Trigger**: User clicks the extension button or context-menu item.
2. **Data Gathering**: Background script (or popup script) requests the current tab's title, URL, and content. For example, using `chrome.scripting.executeScript({func: () => document.title}, …)` and similar for text. Optionally fetch recent history (via `chrome.history.search`).
3. **Compute Suggestions**:
    - **Load existing bookmarks**: `chrome.bookmarks.getTree()` gives the hierarchy developer.chrome.com . Traverse to find existing folder nodes (ignore special unmodifiable ones). For each folder, prepare a text representation (e.g. join the titles of its bookmarks and the folder name).

- **Embed texts**: Convert page content and each folder text into embedding vectors. (This may involve asynchronous model calls.)
- **Similarity scoring**: For each folder, compute cosine similarity with the page. Sort folders by score.
- **Decide on new folder**: If top score < threshold (say 0.7), plan to create a new folder. For naming it, you could take the page's title (e.g. first few words) or ask an LLM for a concise topic name.

4. **User Prompt / UI**: Present the top suggestions to the user. Possible UI designs:
   - **Context submenu**: Dynamically create child menu items under the "Smart Bookmark" menu for each suggested folder name (and one "New Folder..." option) before the user opens the menu. This requires creating/removing context menu items on each page load, which is feasible but somewhat complex.
   - **Popup HTML**: In the extension popup, list suggestions as clickable entries. For example:

```pgsql
Where should this bookmark go?
1. Technology (score 0.92)
2. Security (score 0.88)
3. [Create new folder "Web Security"]
```

   The user selects one.

- **Notifications**: Less ideal, since Chrome notifications have limited interaction. Better to use extension UI.

5. **Save Bookmark**: Once the user picks a folder (or new), call `chrome.bookmarks.create()` or `chrome.bookmarks.move()`:
   - If new folder: `chrome.bookmarks.create({ parentId: parentFolderId, title: suggestedName })` to make folder, then add bookmark into it.
   - If existing folder: move or create the bookmark under that folder. E.g. `chrome.bookmarks.create({ parentId: chosenFolderId, title: pageTitle, url: pageURL })`.
   - (Optionally, remove the bookmark from its default location if this flow was invoked after a native bookmark event.)

Code snippets:

```js
// Background: create context menu
chrome.runtime.onInstalled.addListener(() => {
  chrome.contextMenus.create({ id: 'smartBmk', title:
});

// When user selects it:
chrome.contextMenus.onClicked.addListener(async (info
  if (info.menuItemId === 'smartBmk') {
    // 1. Get page info
    let [ { title }, { url } ] = await Promise.all([
      chrome.tabs.get(tab.id),
```

```
    // can also do scripting for content if needed
  ]);

  // 2. Fetch page text via content script messagin
  let [pageText] = await chrome.scripting.executeSc
    target: { tabId: tab.id },
    func: () => document.body.innerText
  });

  // 3. Load bookmark tree and compute suggestions
  chrome.bookmarks.getTree(tree => {
    const folders = extractFolders(tree);
    // e.g. for each folder do: folderText = folder
    // compute embeddings & similarities with pageT
    // determine best folder or new folder name
    // show prompt UI (e.g. by sending message to p
  });
 }
});
```

(In practice you'd separate this into functions and handle asynchrony carefully. Use Promises or async/await.)

Performance and Privacy

- **Performance**: Running NLP models in the extension can be heavy. Use async calls so the browser UI isn't blocked. For instance, TensorFlow.js models load in Web Workers (or off the main thread). Compute embeddings only on demand, not continuously. Cache embeddings for frequently visited sites or folders in `chrome.storage.local` to avoid recalculating. Limit history fetched to a small window.

- **Resource limits**: Chrome extensions have limited memory and CPU. A 512-dimensional embedding model (like USE) may consume tens of MB, so consider using a lighter model or offloading to a server/API for complex tasks. TensorFlow.js offers **lite** models (USE Lite) that trade some accuracy for speed and size.
- **Privacy**: Be transparent if sending data off-device. If you call a remote API (OpenAI, Hugging Face, etc.), the page's title/text is transmitted. Users should know their bookmarks' content might be processed externally. For sensitive users, allow a mode that only uses local algorithms (like keyword matching). You might also strip or anonymize the text (e.g. remove names, emails). On-device models (tfjs) keep data local and are more private. Always handle data securely and avoid logging sensitive info.
- **Permissions & Trust**: Requiring `"history"` or running ML on all pages may trigger user concerns. Explain in the extension's description why each permission is needed. For example, history can reveal browsing interests to tailor suggestions developer.chrome.com ; allow users to disable that feature if desired.

User Experience

- **Non-intrusive suggestions**: The AI analysis should run quickly. A brief loading spinner or message ("Finding best folder…") is fine, but the user won't tolerate multi-second lags. Consider pre-computing folder embeddings on startup.

- **Clear options**: In the UI, list suggested folders with context (e.g. number of matching keywords). Also include an explicit "Create new folder" option with a default name (editable?). For example, after scoring, you might show "Create folder '<auto-name>'" as a choice. Let the user rename if needed.
- **Learning from feedback**: If a user repeatedly moves a bookmark out of a suggested folder, adjust the logic. For instance, lower the similarity threshold or refine embeddings (by including that bookmark's content under the "correct" folder's representation). Over time the system can adapt to the user's vocabulary.
- **Fallback**: If analysis fails (e.g. no network for API call, or model error), fallback to default behavior: create a bookmark in a default folder (e.g. "Unsorted" or the Bookmarks Bar). You might also fall back to a simple keyword search among folder names.

Example Implementation Tips

- **Getting bookmarks/folders**: Use `chrome.bookmarks.getTree()` developer.chrome.com and traverse the nested nodes to find folder objects. Note some root nodes ("Bookmarks Bar", "Other Bookmarks") are special but can still be used as parents. Avoid folders with `unmodifiable: true`.
- **Searching bookmarks**: You can also use `chrome.bookmarks.search({ title: 'Foo' })` to quickly check if a folder named "Foo" exists. Then use its `id`.

- **Moving bookmarks**: If the user invoked the extension via a native bookmark event (e.g. they pressed Ctrl+D unknowingly), you may catch `chrome.bookmarks.onCreated` event and move that new bookmark. Example:

```js
chrome.bookmarks.onCreated.addListener((id, bookm
  if (bookmark.url) {
    // Run suggestion logic and then:
    chrome.bookmarks.move(id, { parentId: suggest
  }
});
```

  However, note [1] that you *can't intercept* the bookmarking *before* it happens; you'd be reacting *after*. This approach can still re-arrange the bookmark into a better folder right after creation (though without user prompt, or with a subsequent prompt).

- **UI examples**: For a quick UI, you could use `chrome.windows.create({ url: chrome.runtime.getURL("popup.html"), type: "popup", width: 300, height: 400 })` to show a small window with suggestions after the context menu is clicked. Your `popup.html` can list folder buttons generated by JavaScript.

Conclusion

By combining Chrome's bookmarks and context-menu APIs with NLP models, a Smart Bookmarks extension can greatly streamline organizing saved pages. The extension uses content scripts to extract page text, then applies **ML/NLP** (embeddings, classification, or an LLM) to match the page against the user's existing folder hierarchy. If a suitable folder is found, it suggests moving the bookmark there; if not, it generates a fitting folder name. Core implementation steps include gathering the bookmark tree (`chrome.bookmarks.getTree()` developer.chrome.com), computing similarities, and prompting the user via an extension UI (context menu or popup) to confirm placement. All features should be carefully optimized to respect performance and privacy — for example, using on-device models (TensorFlow.js) to avoid sending data externally tensorflow.org or warning users about any cloud-based analysis. With this approach, the extension gives a **complete, intelligent bookmarking workflow** that feels native to the browser while harnessing modern NLP techniques for smart suggestions.

References: Chrome extension APIs (Bookmarks, ContextMenus, History) and permissions documentation developer.chrome.com developer.chrome.com developer.chrome.com developer.chrome.com developer.chrome.com ; a TensorFlow.js example of ML in a Chrome extension tensorflow.org ; StackOverflow advice on bookmark interception stackoverflow.com ; and Chrome content-script docs developer.chrome.com . Each of these guides aspects of the Smart Bookmarks design (intercepting actions, adding UI, extracting page data, and running ML locally or via API).

Citations

Permissions | Chrome for Developers
https://developer.chrome.com/docs/extensions/reference/permissions-list

javascript - Is there a way to detect in browser if user cli...
https://stackoverflow.com/questions/15038261/is-there-a-way-to-detect-in-browser-if-user-clicked-on-save-as-or-bookmarked-t

chrome.contextMenus | API | Chrome for Developers
https://developer.chrome.com/docs/extensions/reference/api/contextMe

chrome.contextMenus | API | Chrome for Developers
https://developer.chrome.com/docs/extensions/reference/api/contextMe

chrome.bookmarks | API | Chrome for Developers
https://developer.chrome.com/docs/extensions/reference/api/bookmarks

chrome.bookmarks | API | Chrome for Developers
https://developer.chrome.com/docs/extensions/reference/api/bookmarks

chrome.history | API | Chrome for Developers
https://developer.chrome.com/docs/extensions/reference/api/history

Content scripts | Chrome Extensions | Chrome for Dev...
https://developer.chrome.com/docs/extensions/develop/concepts/conte scripts

chrome.history | API | Chrome for Developers
https://developer.chrome.com/docs/extensions/reference/api/history

Deploy TensorFlow.js in a Chrome extension
https://www.tensorflow.org/js/tutorials/deployment/web_ml_in_chrome

chrome.history | API | Chrome for Developers

https://developer.chrome.com/docs/extensions/reference/api/history

All Sources

developer.chrome stackoverflow tensorflow