Link to Powerpoint (and video of powerpoint):
https://www.youtube.com/watch?v=2MKptyXzBh8
https://docs.google.com/presentation/d/1FPtH-DxLjvSzM-KhXLBVlJ1aOAMYFY8YgChtgjVf6n0/edit?usp=sharing
Presentation itself

Code Function:
The code in this extension is used to help find similar articles to the one you are currently browsing. It does this by taking the article you are looking at, parsing through it, and looking for similar articles in the backend. The code is broken up into three portions. The first part is the crawler, which retrieves all the articles for up to a year past the date input into the function in crawler.py. The second part is the clusterer, which goes through all the articles that you crawled and turns them into a TF-IDF matrix, then clusters them using k means, and then stores the results using metadata so an article can be easily traced to its cluster. This data is then stored for the third part. The third part is the Chrome extension Hacker News recommendations. First, one needs to set up the API, but afterwards, one can activate the chrome extension on any article page, and the extension will return the url's of five similar articles that the user can go and look at.

Software implementation:
The software is implemented in several files. The web crawler is implemented in crawler.py. The main function in Crawler.py is write_stories_for_time_interval. The first three arguments of the function are start_year, start_month, and start_day, which represent the day you want to start crawling from. From there, there is an option num_days to choose how many days to crawl with a default of 365, or a year. The next argument is blacklist_file, which can take a file of websites you do not want to crawl. The default file blacklists most social media websites. The next argument limit is to limit the amount of files per day. This is so people with smaller systems can choose to take in a smaller amount of data, should they choose. The default value for the limit is 250. The final argument is num_threads, which allows the user to use threading to speed up the crawling process. By default the value is 1. The function itself works by first getting the range of dates the user requested with the use of the function all_timestamps. From there, we create a directory for the data, and then start iterating through the dates and getting the articles themselves using the function getStories. Get stories returns the articles as json, so then we need to use parsejson to parse through them. Afterwards, we process the articles, and then write the articles into the file directory. There are also several smaller helper functions: get_html, tag_visible, is_valid_url, url_to_filename, and text_from_html that help all the bigger functions. An important thing to note about the new files that were written: the first line of each file is metadata, while the second line is the article itself.

The other big file is clustering.py. Several things occur here, from clustering to analyzing the data. Let's start with clustering. The clustering function in clustering.py takes in a directory, which is the directory holding all the data that was crawled. It then takes in a destination directory, for where you want the output of this function to go. Afterwards there are some optional arguments. First is num_clusters, which lets you customize how many clusters you want from the means of clustering, although set at default at 5. The second is number_iterations, which customizes how many iterations the k-means cluster goes through, although it's defaulted to 200. Finally, there is a vectorizer_path argument with a default is None. If you already have a TF-IDF matrix, this is where you load in the path. The function starts by getting all the data from the articles, and then forming those articles into a TF-IDF matrix if you did not provide one. Afterwards, the function begins the process of k-means clustering. When that's done, it stores the clusters, the TF-IDF vector, and a third object, a panda dataframe, that links the clusters to the original article URL and title. This function uses several helper functions, mainly get_file_names, parse_metadata, and a normalizer for the tokenizer.

Moving onto the rest of clustering.py, we will start with the class contained within the file, CosSimilarity. A DisSImilarity object holds a path to the data_directory, along with all the data that was stored in clustering.py. From there, it has several functions. It has a function to get CosSimilarity between itself and an article. It also has a parse metadata function. Its two important functions are get_similarity and get_most_similar. Get_similar takes in a url for the article, and then uses a function called classify to predict which cluster this new url/article would belong to. From there, the function performs cosSimilairty on all the articles in the cluster the URL matched with, and returns the similarity, along with a list of URLs and titles. The second function, get_most_similar, takes in all these data points that were returned, and finds the most similar documents. It has an argument for how many documents you want returned, with a default value of 1.

Finally, there are two more functions in clustering.py. Classify, which was mentioned earlier, is what classifies a new document to one of the clusters, and takes in a url, a model, and a vectorizer. The second function is get_similar_docs, which is the function that takes in a url, a cos object, and num_similar documents. This function is the one that goes through get_similarity and passes its return values to get_most_similar, before returning those values back to the API.

The final main python file is API.py. This is the file that runs the api itself when activated. The API being used is fastAPI, so make sure to have it installed on python beforehand. To activate the API, type uvicorn api:app --reload into the terminal. From there, the api will activate the function startup, which just stores all the data the api needs. If you want to use the API directly, go to "http://127.0.0.1:8000/docs/" to enter a nice page where you can type in a URL and the api will classify it and return the results using the other function, read item.

The other part of this extension is the actual extension itself, which is handled in the folder extension. This folder contains a manifest necessary for the extension, which is on manifest version 3.There is a small html file that lays out the pop-up for the extension. That pop-up has a button that is used to trigger the functionality of the extension. Most of the work of the extension takes place in popup.js.

TO understand popup.js, we are going to go in order of how the functions are used. Within the file, there is an eventhandler for when the button is clicked, that begins the process by going to function onClick(). This function gets the current URL, and then passes the URL to getRelated(). This function checks if the url is legit, and then passes the URL to getRelated(). This function encodes the URL to match the format of urls in clustering.py, then sends that url to function sendGet(). SendGet() opens a process request to the server, which then sends the url to the api, at which point the API goes through and finds similar documents, and sends those documents back. Those documents get returned and parsed in getRelated, before being added to a table in updateElements. When that's done, getRelated() toggles the html file to display a table containing all of the similar documents titles and URLs for the user.

Software usage:
To use this software, make sure to have the following python libraries installed:
Os, numpy, time, sklearn, joblib, pandas, sys, crawler, nltk, string, tqdm, itertools, ast, bs4, calendar, datetime, requests, tldextract, validators, fastAPI

Once these have been installed, you will either need to crawl hackernews to get the articles, or you can download a set of data that has already been processed if you would like to go straight to API setup.

Pre-computed:
If you just want to use the following github without acquiring the data yourself, here is file contained data already crawled by this crawler:
https://drive.google.com/file/d/11bZ53ukSD7kwLpItcdEId142TtF-KS4H/view .
 From there, you can download these files:
https://drive.google.com/file/d/1K1K1dWj8UWjoSL3QJW-ZxaKYDGB0gucP/view .
These files are the .pkl files that store the k-means cluster, the TF-IDF vector, and some other ancillary information

Manually get the files:
 To manually get the files, you need to run the main function in crawler.py. There is a date auto populate in there, but you can change that to the beginning of the date range you desire. It will crawl up to 365 days ahead of the initial date entered. This is the longest step of the whole process, and it will take time to crawl the data.

After you have crawled the data, the next step is to run the main function in clustering.py. This function takes in three inputs: the folder containing all the data, the word cluster, and the amount of clusters you would like the data to be divided into. An example would be: py clustering.py <directory_holding_files> cluster <#>. This can vary based on user performance needs. After this is done, it will create and store pickle files that will hold all the info necessary for the chrome extension to work.

Setting up the API:

At this point we move onto the third part, operating the chrome extension. First, you need to follow this link to go to your chrome extension settings, and enable developer tools. From there, unpack the folder holding the manifest.json file in this code package. That will set up the extension in the browser. From there, go back to your interpreter and in the terminal, type uvicorn api:app --reload. This will cause fastapi to activate and start loading all the info needed for the extension to work. After the terminal prints an all good message, feel free to go use the extension. It's all set up and ready to go.

Extension usage:
To use the extension itself, go to the website of your choice, click the extension, and press the button to look for similar results. Within a minute, 5 article titles and urls should pop up for you to look at. You can repeat this as many times as you want, on any article of your choice! Happy browsing!

Contribution:
Trevor worked more on the crawler, clustering, api, js
Alex worked on the crawler, clustering function, investigated different ways to implement query in API, helped on api and js