I recently finished developing a new subreddit recommender system in python using mlxtend's fpgrowth and association_rules functions. Its name is wackysubs.com. It is on the public web.

During training of this recommender I inputted tens of millions of "market basket transactions" in the form of web-scale public user-comment data to particular "subreddits", from two complete months of traffic from year 2018 of social website reddit.com. I aim to retrain my model on freshest possible comment data after I get the first version into production. It's already working good as it is though. New subreddits are made every day because they are end user created without limits. I saw some good newer subreddits that would make it worthwhile in 2022 to get the newer raw data.

Interestingness and speed of query were both very important in this case. The result of the effort is my recommender system which is in the form of a simple looking and responsive web app, to help people discover and navigate to interesting subreddits from among thousands of subreddits, that maybe they did not know about before.

As of Feb 23 2022, I am presently moving it from my dev server running Flask library to a production server at AWS EC2 on Apache, where it's taking me way too long to
1. Get all the permissions and configuration problems solved on the production server;
2. Solve the performance problems a) reducing memory consumed down to less than 2 GB, after initially using 5 GB; and b) also finding the cheapest cloud instance type that gets it running nicely with sufficient memory and processor.

I am sharing this so I can remember how to use this design again in another app but with less R&D time, and also to share so other people can get better results too:

1. The problems encountered;
2. The top tricks I used to solve them including increasing the model training speed, and the application's query speed, and getting more interesting results;
3. Suggestions based on practical experience, for moving an improvement or two into the mlxtend library itself to make it even easier to develop with, faster, use less memory, and make better recommendations than I have achieved when I coded my workarounds strictly on the outside of mlxtend thus far.

Preprocessing pipeline:

1. The raw data compressed files were downloaded using wget. The raw data's format is json text files containing comments posted to reddit.com: The comment text; comment's username; name of subreddit they posted it to; datetime it was posted, et al. Each file has one full month worth of comments. Around the order of 10^7 comments per month is published, by 10^5 unique users, to 10^4 subreddits, if I recall correctly. A new comment json file gets published every month.
2. The app's preprocessing pipeline transforms the raw json file, into one file per user, and then ultimately into one big plain textfile of lines where each line is one unique user's list of subreddits they commented to that month. One line equals one "transaction" or "market basket" in the jargon of association rules mining. Bash shell commands were used extensively to script this part, like jq, gnu-parallel, awk.
3. The file has two lines per unique user if two months worth are analyzed, which works fine too; it's just another "market basket" transaction by the same user. In this app the pipeline simply concatenates two such month files together into one big text file. More lines proved better,

because the model would find and record more patterns and association rules, but it is limited by the available computer power during training.

4. The next preprocessing step has the transaction file passed to mlxtend’s TransactionEncoder using the sparse option (standard dense setting would result in a model that would record fewer interesting patterns), which creates a pandas dataframe having a column for every unique subreddit.
5. Training the model properly begins when the TransactionEncoded dataframe is passed to mlxtend’s fpgrowth function, then its association_rules function.
6. Postprocessing (after preprocessing and training, but before the app loads it) is needed on the trained model dataframe that’s output from assocation_rules as follows: Keep only rows having 3 or fewer antecedents and only one consequent. Run the TransactionEncoder directly on the trained model dataframe, to add a column for every subreddit with 0 and 1 values in the rows. Notice the TransactionEncoder is intentionally re-used for postprocessing of the trained model which is not what it was designed for, but it works great for my app for query speedup versus frozensets (for which feather/arrow lack sufficient support) generated in advance from the frozenset code strings, or frozenset code strings that are far too slow to eval or process as strings or lists of strings or regex matching. Delete all other columns that are unused by the app.

Development Phases:

Phase 0: Data preprocessing pipeline. Input to pipeline: raw json of comments etc. Output from pipeline: Text file where one line is one unique redditor’s “market basket” of subreddits they commented to during the time period of the raw data.
Phase 1: Experimenting to find out if the data science concept would actually work well on this dataset. This was about developing a trained model to generate pretty interesting recommendations.
Phase 2: Performance engineering to achieve interactive speeds, or, How can I get 1 hour queries down to 1 second? And worse, also achieve that level of performance on low-powered low-cost cloud hosts?
Phase 3: Develop the web app, which should look nice, run fast, be easy, look simple, and be fun for the user of the app. I would have to learn flask python web development as well, which is not the same as the Microsoft C# ASP web dev stack I used to use.
Phase 4: Get the web app to work on cloud: Amazon AWS EC2. Spend too much time on software installation, file permissions, and Apache conf files (It was not a goal, but I did that.)
Phase 5: Get the web app to work on Microsoft Azure cloud too. This time, though, use a serverless design (not EC2), or at least do something different to avoid hand-building and hand-administrating a virtual server instance with all that noise like software installation, file permissions, and conf files.

Phase 1 development was the data science phase, where the goals were finding out if the main concept actually works as hypothesized, ie, can we really discover interesting rules and recommendations using this dataset; and also how to maximize the model’s recommendation quality, ie, in this application how many patterns does the model need to find during training, to get to the level where it’s producing truly interesting or new non-obvious recommendations? Did it need a little, or a lot, to work good, subjectively?

As it turned out, the data science concept happily proved valid, because the trained models showed signs of having detected interesting patterns and interesting recommendations: It’s going beyond just simple substring search. The nfl subreddit query, produced the CFB and nba in the results, which make some sense to humans as we know, so the computer model was truly discovering something about shared interests in the human users at reddit who created the raw dataset. A lot of football teams show up in the query results. Many other queries showed interesting association patterns too.

Goodness of results from this model design, it turns out, came from bigger data, in this case inputting two months instead of just one month of raw data; and using a lower min-confidence filter in fpgrowth. These were the two keys to get a good model working great, in terms of interesting results. Finding niche and quirky subreddits was pretty important and really the point of this exercise. It would be essential for the recommender to help the users also find the more quirky, niche, or just newer on-the-way up subreddits as well, where there was not yet front-page bubbling up of their comment chains. Small but good or up-and-coming subreddits don't have that many users yet, compared to the front-page menu subreddits, so the statistical confidence filter would need to be set low to prevent their complete disappearance from this recommender system. Nobody needs a recommender to tell reddit users about news, politics, popular, or AskReddit; these subreddits are already on the default menu at the top of the user interface.

Increasing the goodness was found to make the model TOO big, though, causing dev machine crashes even though 128GB of memory was available; and on the way to the crashes, the training became practically endless, hours into days, as the swap space became fully consumed; and there was no way to know in advance if just a little or half of the swap was needed, and if you let it continue that it will finish but just after a long time and a great model would pop out after the wait. There was a balance to be found. Actually just how big -- and therefore good in the sense of interesting recommendations – could this model get, on my available hardware? It took several attempts to find out.

Phase 2 development was about performance: This is the software R&D and engineering phase, where the goals were to modify the code and the data, in order to find the biggest practical model size, emphasis on practical; while simultaneously achieving the contradictory but equally important goal of greatly speeding the code up to the performance level needed to run it on a website. On a website the app must load and query the trained model interactively in a snappy-feeling public-facing web app, and ideally also do it on lower priced cloud hosts. It was actually not known at the start of this effort whether it was even possible to practically satisfy these very contradictory goals. It turns out yes, it is possible.

You have to understand, at the beginning of the phase 2 performance phase, the first-version correct-but-slow code was running literally 1.1 hours, on a dedicated high powered Xeon workstation even, just to return a single pandas query result from the best model that was produced from the Phase 1 effort.

The software work paid off handsomely: The model was running sub-second queries by the end of phase 2 work. In summary here is how:

I sequentially grew the size of the dataset during development: I used a tiny dev raw dataset and resulting tiny model size for initial code design and training the association rules model within Jupyter Notebook. Next, I used a medium size dev dataset and resulting model size for performance indications and recommendation quality indications, and design improvements, in Jupyter Notebook. Finally, a full size model was used to develop the web app on the dev host, as well as to optimize queries aiming toward the production environment which would mandate the least memory usage, and the fastest queries, for cloud hosts which I discovered would cost me too much money for any increased computer power beyond really little computer processor and memory. My dev host has way more power than my cloud host – this is the challenge.

Fpgrowth is faster and more advanced than Apriori and produces the same output. On web scale data, fpgrowth is necessary to use; it too easily reaches and exceeds available computer power depending on what you do.

The trained model at first was in a pandas dataframe containing “frozenset” representations of frequent set consequents and antecedents. Initial performance was so awfully bad on the query that needed to run on the dataframe when the user clicked “Find,” that this query took like 1 hour 6 minutes to return. For a web app this would be impossibly bad.

I found during training that lower confidence filter values like 0.0003 resulted in way better recommendations, more “niche” and relatively unknown and obscure and quirky subreddits. This is exactly what it needed to do for the users. The lower the confidence the better.
Low confidence values caused the model to become huge, and take massive RAM to train, and take a really long time to train.

Phase 3 was all about the web app, designing and refining a basic but nice to use and nice to look at web app, that only reads the trained model, and uses browser cookies to save user favorites.

Coda: On experimental code versus production code. The code is not refactored and amazingly organized. The art is also not amazing. That is a good thing. I know what unit tests are, and where they belong. They don’t belong here. Here is why. This app is an architectural proof of concept. It answers a question: Is this even possible? It’s an experimental system. Answering a question sooner is deemed the most important thing, because that is how experiments are. Of far lesser importance was refactoring and testability, metaphorically cleaning the sock drawers until they were neat and orderly for the next software developer who needed to efficiently find some socks to wear, or to scale out the number and type of socks it can hold, and reduce defects over time. Not that there is anything wrong with that. I own the web site. I own the app. The priorities are for the owner to dictate. It’s not good to waste time reducing technical debt too soon, that is, optimizing the code’s organization prematurely. It would increase the time, the cost of obtaining the answer to the question. Code optimization is undesirable when it is done too soon, as everyone should know by now. Code organization is a type of code optimization.