Making Video Recommendations Using User Supplied Text

Geoffrey Lawson 6/6/2013

More like this is a Youtube video recommendation service. Users can create channels for types of videos they're interested in, select videos that reflect those interests, and new videos will be recommended for them that reflect trends in the videos liked by the user. Trends or similarities are judged by the words used to describe the videos. The More Like This system is broken up into three main components: the database, the website, and the recommender.

The DBMS I chose for this project was MongoDB. I opted for MongoDB over a traditional SQL database for a few reasons: 1) I wanted a system that could easily scale to incorporate as much data from Youtube as I would be able to acquire, 2) The document layout seemed more conducive to storing video information from Youtube and information about users than tuples, 3) Youtube is inconsistent with the data it serves and fields may appear and disappear which could be difficult to work with in a SQL database, 4) I needed a database that could handle frequently changing data well. I had experience with MongoDB sharded collections so I knew should I ever get to that situation where I needed to break the database up over multiple servers it would be able to handle that gracefully. Concerns 2 and 3 are closely related, as part of the video structure has to do with the inconsistent info Youtube provides. I can't be sure a field will always be present, or up to date when I request information from Youtube. Additionally queues, hashes, and lists occurred frequently in stored information and while I could have created tuples to properly convey order, the built in list structures in MongoDB expedited the project. The last point addresses how the three systems come together. At a high level view the streams collection(I will refer to channels a user has created as streams to differentiate them from channels on Youtube) in the database holds what videos a user should watch next. Those are accessed by the web site, which updates the list to reflect which videos the user has already watched, and what if any feedback they gave about the video. The recommender than recommends videos for streams which are running low on unwatched videos. The new videos are then appended to the list in the appropriate stream in the database so next time the web server asks for new videos there are some waiting for it. I will go into more detail on this process later, but for now it is important to realize that while a user is online, their streams are going

to be accessed from the database frequently. More accurately one stream of theirs will be accessed frequently, then they' shift to watching another stream and that one will be accessed frequently. From my research MongoDB handles large data sets and frequently accessed records well[2,5], which is what my project has.

These are the schemas I used in my Database. It should be noted that since MongoDB is a document store style database not all keys in a record are present. Only one collection should exhibit this behavior however, the videos collection. All others are updated exclusively by the website or myself and should have all keys present. Many keys have been minified to save space[1], names in parenthesis exist to clarify what each field is supposed to be. Additionally documents in a collection aren't required to have the same type stored under a given key. In this database they do and I will label what types a key should return. All collections have an implicit index on the _id key[4]. All other indexes will be explicitly mentioned.

Streams:{ _id: {“u”(user id): String, “id”(stream id)},
“bw”(bonus words): {String,number} List,
“cs”(categories): {String,number} List,
“cur”(currently receiving recommendations): Boolean,
“d”(disliked videos): String List,
“du”(duration info): {“ps”(videos in calculation): Number,
“sx”(total duration): Number,
“sx2”(sum of squared durations): Number,
“m”(mean duration): Number,
“sd”(standard deviation): Number}
“l”(liked videos): String List,
“mw”(minus words): {String,number} List,
“rec”(needs more videos): Boolean
“v”(recent videos watched): String List
“w”(videos to watch): String List

This collection is all of the streams on the site. All streams are unique on the combination of name and user. The bonus words/minus words keys keep track of each word that has been liked or disliked, and the net times they have been liked or disliked. As it is a net score a word can move from one list to the other should they go negative. If a word's net score goes to 0 it is removed. There is a index on {rec:1,curr:-1} so the recommender can quickly find all streams that need a recommendation and currently do not have a process making recommendations for them.

Users: { _id: ObjectID,
“current_sign_in_at”: ISODate,
“current_sign_in_ip”: String,
“email”: String,
“encrypted_password”: String,
“last_sign_in_at”: ISODate,
“last_sign_in_ip”: String,
“sign_in_count”: number}

This collection is maintained by the ruby gem Devise. It keeps track of user login information.

Videos:{ _id: String,
“ t”(title): String,
“u”(URL): String,
“k”(keywords): String List,
“v”(views): Number,
“f”(favorites): Number,
“ch”(channel): String,
“d”(description): String,
“du”(duration): Number,
“c”(Category): String}

This collection is all of the videos currently in my system. This collection is updated every night to get the most recent information about the videos, and to get new videos. If a user attempts to add a video that is not currently in the system, it's information is harvested from Youtube immediately. More videos from that channel won't be harvested until midnight UTC when another video harvest from Youtube is made. It id used is the video tag as supplied by Youtube. They are unique and fairly evenly distributed across their hash space. Keywords for a video are the set of all words in a video's description and title minus the set of words in the stopwords collection. This collection has an index {K:1}.

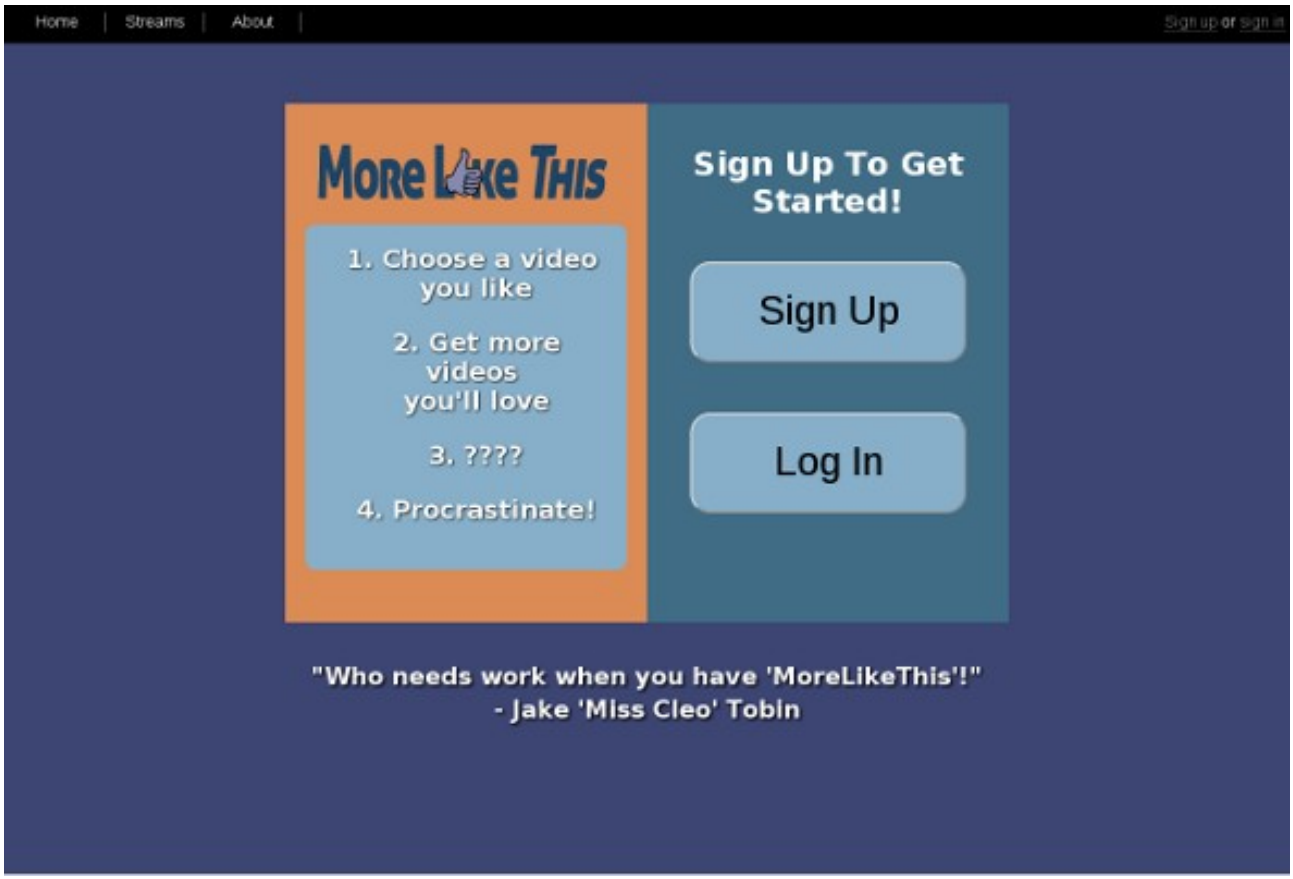
Stopwords:{ _id: String}

This is the list of all stop words used for filtering keywords out of videos. Any word in this collection will not be added as a keyword to a video. This collection is only updated manually. This collection only became necessary once Youtube phased out their own keywords. Originally video makers added their own keywords, but Youtube stopped supporting them about 2 months into this project.

Channels:{ _id: String}

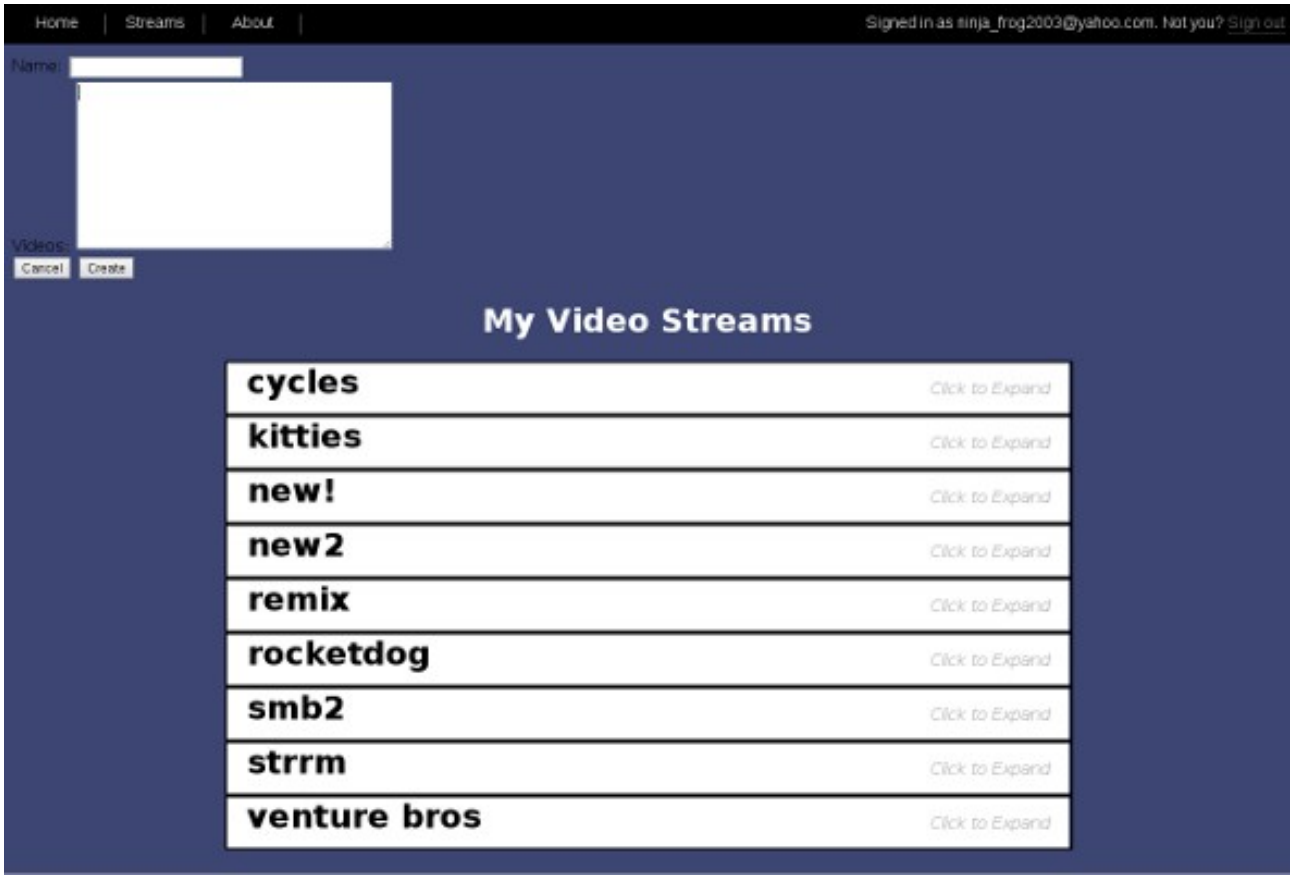
All channels in this collection will have their videos updated in the videos added to the videos collection. This collection was originally created for seeding the database with videos. It started from a list of the 1000 most popular Youtube channels of 2010. It gets expanded with the most active channels of each day as part of the video pull. It is no longer important as the list of relevant channels can be attained from the videos collection.

The next part of the system is the website. The website was created using ruby on rails. I chose to use ruby on rails for the simplicity it offered. I wasn't too interested with putting together a complex website, I only wanted some sort of login mechanism so I could track users and some central serving process so I could send new data to the client without having the javascript directly interface with the database.

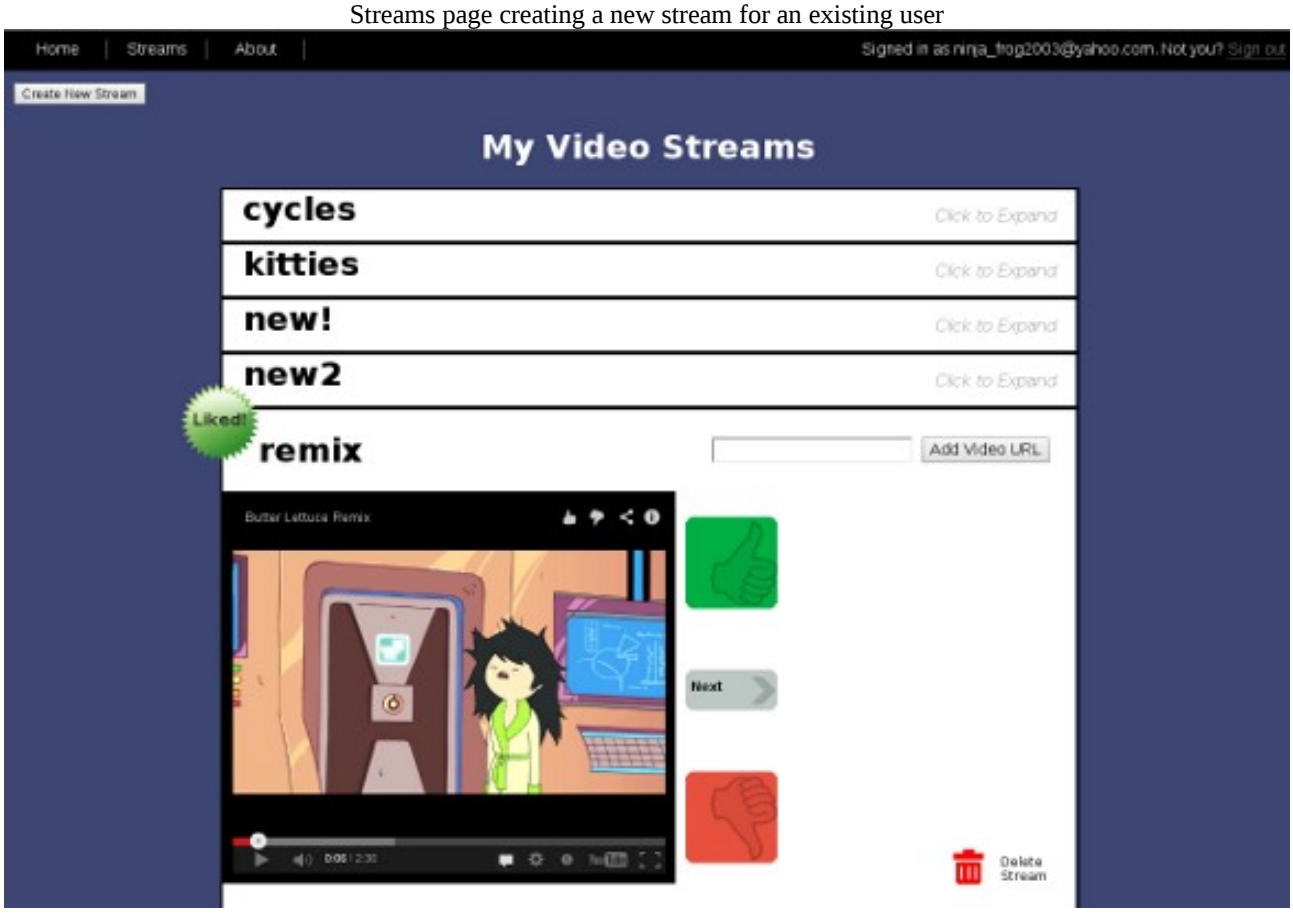


Main page of More Like This.

The main page is where users will find themselves when first navigating to the site. New users can sign up, and returning users can sign in, using the links provided. After logging in or signing up they will be directed to the streams page where they can create new streams or watch existing ones.



Streams page creating a new stream for an existing user



Streams page viewing a liked video on an existing stream

The streams page is the only part of the website that shows any interesting behavior so I will explain how the flow of information works on that page. When the client loads the page for the first

time they must create a new stream. They supply a name for the stream and 1 or more comma separated Youtube video URLs. A javascript function parses the video tags out of the urls and sends them along with the name of the new stream to the server. The server creates the new stream in the database, adds the supplied videos as liked videos, flags it as needing a recommendation, and forks a recommendation process. It then tells the client that the stream has been successfully created and the client creates the stream on the web page. At this point the user will most likely open that stream. The client creates the opened stream on the page, replacing the old stream and asks the server for some videos. If this is the first time a stream is being viewed it is unlikely that it has any videos. Recommendations can take between 30 seconds- 2 minutes so some way to hold the users attention is needed in this time. The solution is server takes one of the liked videos on the stream at random and send it to the client. The client adjusts the stream display to show that the video is a previously liked video. Once a video completes a message is sent back to the server with the id of the video saying it has been watched. The id is appended to the list of recently watched videos. If the list goes over a set amount, ids are shifted off the front. Next the sent id is compared to the id at the front of the recommended videos list. If they match the front is shifted off as it has just been watched. In this case a new video is required as the client only had one video from asking the server last time. It again asks the server, but this time there are videos. The server sends a portion of the recommended videos it has off to the client. These are not removed from the list in the database yet. Videos are only removed from the recommended videos list if the server receives a watched message containing the id of the video at the front of the list. A video completing, the user clicking next, and the user clicking dislike will all send the watched message. If the server sends videos to the client and finds the length of the stream's recommended videos list is less than some non-zero number it will mark the stream as needing a recommendation and fork a recommender process. The recommender process will complete before the user exhausts the recommended videos list. If this is not the case videos are offered at random from the liked videos list similar to before. Rapidly asking for a new video can lead to odd behavior. If the user requests a new video before the server has time

to respond the same video that was being displayed will be offered again to the user. This comes from the client having no videos to offer and thinking that it's request got lost so it re-offers the same video.

Liking a video and adding a video work almost identically. The only difference is the id has to be parsed out of a url client side before being sent to the server when adding a video unlike the like video which pulls it out of the queue. When a video is liked it's id is added to the list of liked videos. Then it's keywords are compared to the words in bonus words and minus words. First if the word is in minus words, it's count is decreased by one. If a word isn't in minus words, it's count is incremented in bonus words if it exists there. If it does not it is added with a score of one. Next the duration of the video is added into the rolling mean and standard deviation calculation. The mean is calculated as sx/ps ps equals the number of videos and sx equals the sum of those videos' durations. SD is calculated as $sqrt(((ps*sx2)-sx^2)/ps^2)$ where ps equals the number of videos, sx2 equals the sum of the squares of all of the video durations, and sx equals the sum of all video durations. Dislike is almost the inverse. The keyword parts works similarly, except words being taken from bonus words and added to minus words. Nothing is done about the duration.

The last part of the System is the recommender. The recommender is a compiled python script. I opted for python as it appeared to execute faster than ruby and I had familiarity with the MongoDB drivers for python from testing. Originally the recommender was part of the server. The server called the recommend function I had made for it. However the server was not multi-threaded and while doing a recommendation it would be unable to server anything. This was obviously undesirable as recommendations could take up to 4 minutes in ruby.

A recommendation starts with the server seeing a stream is low on videos. It marks a video as needing recommendations by setting it's “rec” field to true and it's “cur” field to false and forking a recommender process. The recommender then then runs a find({“rec”:true, “cur”:false}) to get all streams that need a recommendation. This is done in case a recommender died prematurely for some reason. I felt it would be better to be safe and make sure there is some way to pick up streams

that are still looking for recommendations than leave streams forever waiting. It takes the first record and does an atomic find and modify operation to set it's “cur” to false and return the resulting document (find_and_modify({_id: cursor.first[_id], “rec”:true, “cur”:false}, {$set { “cur”: true}})). This query double checks to make sure no other process has taken over this recommendation. Once a stream is locked to the process the recommendation begins.

First a subset of videos is selected from the database. This subset is all videos that aren't in the disliked list, not in the recently watched list, not in the recommended list, have a duration within one standard deviation of the streams mean, and contain at least one keyword from the bonus word list for the stream. This query looks like find({"$and" : [{"k" :{"$in" : stream['bw']}},{'_id ' : {"$nin" : stream['d']}}, { '_id':{"$nin" : stream['w']}}, { '_id' :{"$ nin" : stream['v']}},{ “du”: { “$lte”: stream['du']['m']+stream['du']['sd']}},{“du”: {“$gte”: stream['du']['m']-stream['du']['sd']}}]}). Then each video in that subset has a score calculated. The scoring works as follows: for each keyword a video has that is in the bonus words list, it gets the number of times that word has been liked. Then for each word that is in the minus words list the video gets minus one point. A video gets an additional five points if the video's category matches one of the stream's categories. I originally wanted to have this operation be handled as a map/reduce at it would be an easily parallelizable process, but after some research I learned the MongoDB map/reduce isn't very efficient as it copies the collection[6], and like all map/reduce jobs only gives benefits with multiple nodes[3], which at current my database does not have. The choice to have bonus words effect the score by their frequency and to have minus words not came from the idea that a like meant more than a dislike. When a user likes a video they are expressing two things. First that the video was relevant, and second that it was enjoyable. A dislike can mean either of those statements was false. So If a word comes up a lot in liked videos, it is likely to be indicative of a video the user will like. However another word may be indicative of related content, but due to poor quality videos it doesn't have a net positive score.

After all the videos in the subset are scored, the top 8 are enqueued into the recommended

videos list. The recommender then resets the “curr” and “rec” fields to false on the stream. It continues through it's list of streams that may need recommendations repeating the process for any videos that need it. Once it has no more videos the process exits. This leaves a possible bug in the system. From multiple tests, it seems Ruby 1.9.3 terminates it's child processes when it ends, which can prematurely end the recomenders. This will leave a stream in a state of “rec” and “curr” being true. This would mean that no recommender will ever attempt to recommend videos for this stream. It would be a good idea on starting the server to make sure all “curr” fields are set to false.

In summary, the More Like This system has three parts, the database, the web site server, and the recommender. The database holds a subset of Youtube video information, attempts to update and grow it's video collection every night, and holds recommendations for users as they are too slow to happen in real time. The web site server is the front end that makes sure the user has a constant stream of videos going to them and shields the database from the user. It is responsible for telling the recommender when to run. The recommender looks at the criteria in each stream and makes recommendations mostly off of keyword matching, and also video duration.

Bibliography

1) “A Year With MongoDB” http://blog.engineering.kiip.me/post/20988881092/a-year-with-mongodb. Online June 2013.

2) Clark, BJ “NoSQL If Only It Was That Easy” http://bjclark.me/2009/08/nosql-if-only-it-was-that-easy/ Online June 2013.

3) Dean, Jeffrey and Sanjay Ghemawa “MapReduce: simplified data processing on large clusters” Communications of the ACM 51.1 January 2008

4) “Indexing Overview” http://docs.mongodb.org/manual/core/indexes/ Online June 2013.

5) Kovács, Kristóf “*Cassandra vs MongoDB vs CouchDB vs Redis vs Riak vs HBase vs Couchbase vs Neo4j vs Hypertable vs ElasticSearch vs Accumulo vs VoltDB vs Scalaris comparison” kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis.* Online *June 2013.*

6)”Map-Reduce” http://docs.mongodb.org/manual/core/map-reduce/ Online June 2013.