

# Preventing the Next Vioxx



Zenobia Liendo, Marcus DeMaster, Matt Swan  
W205-2

## Motivation and Prior Work

An adverse drug reaction is an undesirable and unintended, potentially harmful, effect of taking a medication. This effect could happen after taking a single dose, after prolonged use, or may occur as the medication reacts with another medication the patient is currently taking.

*An NIH study reports that 95% of adverse drug reactions are not reported.*<sup>1</sup>

In 2004, the medication Vioxx was withdrawn from the market because of its adverse effects to the cardiovascular system, resulting in heart attacks or death.<sup>2</sup> Unfortunately, since its release, more than 38,000 deaths have been attributed to the drug's use. Some may argue that we didn't know of its adverse effects because of deceptive practices on behalf of Merck, the company's manufacturer.<sup>3</sup>

Regardless, the concept of an independent monitoring system that could potentially identify these adverse drug reactions sooner rather than later is very appealing. Today, there is more opportunity than ever before, as the FDA itself has begun releasing data about adverse drug reactions that have been reported. Additionally, new sources of data, such as social media, now exist to provide insight.

Much research has been done in this field. It has mainly focused on mining publicly available patient data, such as comments tweeted by patients about their experiences. The pharmaceutical journal presents an article mentioning current research on this topic: [“Searching social networks to detect adverse reactions”](#)

## Data Sources

For our project, we focused on two major data sources.

First, ADR report data provided by the FDA. Routinely, when adverse drug reactions are reported, they make their way to the FDA. A patient may report the effect to his primary care physician, who in turn reports it to the FDA. A patient or practitioner may also report the adverse reaction to the drug manufacturer, who is also supposed to then forward the report to the FDA.

Since early last year, the FDA has been hosting the OpenFDA Developer Challenge. They are providing data on adverse drug reactions, recalls, and labeling, with the idea that developers and researchers will leverage the data to find new insights for the benefit of the public at large.

---

<sup>1</sup> <https://www.ncbi.nlm.nih.gov/pubmed/16689555>

<sup>2</sup> <http://www.fda.gov/Drugs/DrugSafety/PostmarketDrugSafetyInformationforPatientsandProviders/ucm106274.htm>

<sup>3</sup> <http://www.npr.org/templates/story/story.php?storyId=5470430>

Second, we've collected data from the Twitter social media website with the belief that active users would be willing to share their adverse reactions with the Twitter community, whether or not they also shared such a reaction with their doctor. This, as we'll see, becomes a two-step process as we must first collect data with regards to various medications, as well as classify those tweets into those describing an adverse drug reaction and those simply remarking about the product.

## Data Collection and Processing

### ***FDA Data***

In our investigation of the OpenFDA API, we first explored the use of the Search API. This is designed for real-time queries and allows you to query specific fields from the entirety of the OpenFDA database. However, OpenFDA has set limits of 5,000 records per query to balance load on their system. Fortunately, we have the ability to use a special download API to get a list of all the OpenFDA JSON files, which contain ADR data<sup>4</sup>.

These files include information on a range of FDA activities, including medical device adverse events, drug and device recalls, drug and device labels, clearance applications, and food enforcement activities. Most of the work in retrieving the ADR data involved the development of a Python script that parses through the JSON contained with the files hosted by the Download API to pull the correct files from the drug/event endpoint. These files are dated by quarter and year for the ADR reports, which allowed us to download the data by time period.

We found several fields of interest to us from the files. We collected report level information such as the *report\_ID*, *patient\_age*, *patient\_sex*, and *report\_date*. Within each report is contained a list of medications currently taken by the patient as well as a list of adverse reactions reported by the patient. Most of the reports don't specify a link from any one drug to any one of the reactions reported by the patient. Therefore, we loaded the drug and reactions fields drawing no connection between the two lists other than the patient's report in which they were delivered. Finally, many of the FDA reports include a list of common brand and generic terms associated with a reported drugs. We collected this information and loaded it into a brands and generics table for later use in joining the FDA reports with the tweets data and displaying selections to end users.

A number of Python packages were required in order to process the FDA data, which came as compressed JSON. This included *Requests* to pull the zipped JSON content from the Download API. *BytesIO* and *ZipFile* were imported to unzip the downloaded objects. And finally, *JSON* was imported to read the JSON into a data object that could be processed in python. *Psycopg2* was imported into this script for loading the desired records from the JSON files into our PostgreSQL database tables.

---

<sup>4</sup> FDA downloads available: <http://download.open.fda.gov/>

We found that the files of interest ranged in size from 15 MB to 300 MB. The larger files tended to kill the loading process when running an m3.medium instance. Switching to a memory-optimized m2.4xlarge process fixed this problem. In total, there were about 5 GB of files for 2015 ADR reports. We don't foresee an issue with scaling to include all years of FDA ADR report data given the table limit in PostgreSQL is 32 TB<sup>5</sup>.

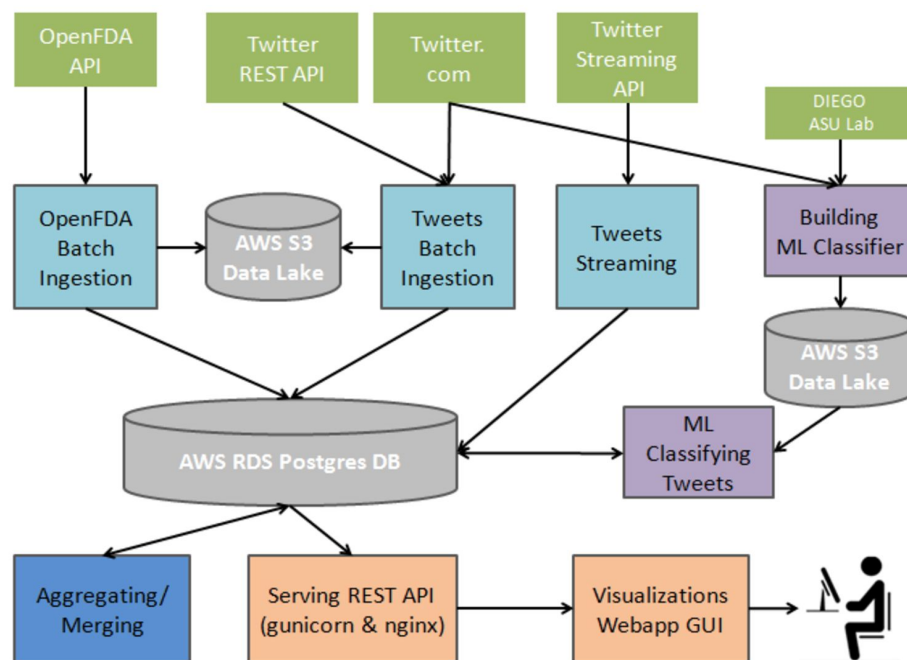
## Twitter Data

Twitter provides two APIs for searching tweets; one API is used to access tweets posted in the last nine days and the other is used to access tweets being posted live for streaming. Both return data in JSON format.

We needed to collect tweets as old as 10 years to compare them to data provided by the FDA. This was a challenge since Twitter APIs limits to tweets nine days old. The solution was to access the existing search at Twitter.com instead and pull historic data by programatically executing searches and scraping the content of the HTML pages returned.

This prototype collected tweets from January 2015 and February 2015 to compare it with FDA data downloaded for 2015 Q1.

## Architecture, Process Flow and Implementation Strategy



**Fig 1. Architectural Diagram**

<sup>5</sup><https://www.postgresql.org/about/>

Our architecture consists of five primary functional areas:

- 1) Data Ingestion
  - Intake and parsing of data in from data sources via flat files or APIs
- 2) Streaming
  - Collecting streaming data from the Twitter social media platform
- 3) Data Classification
  - Responsible for determining which tweets are ADR-related
- 4) Data Integration and Aggregation
  - Integrates data from Twitter and the FDA
- 5) Data Serving
  - Fetches data and processes for user consumption

Additionally, as you'll notice, at the center of our architecture is a PostgreSQL database, hosted on Amazon's RDS service. This strategic decision makes it simple for different members of the development team to access the data in the same state at any given time, rather than trying to keep a distributed database architecture constantly synced.

Let's examine each of these four primary functional areas.

### ***Data Ingestion***

Whether we're discussing ingestion as it relates to the FDA data or the Twitter data, the process on both sides of the data lake is roughly the same. Data is brought in by API and then a data lake is leveraged using Amazon's S3 service to process the data for final insertion into the Postgres database.

### ***Streaming***

The second part of the architecture is concerned with streaming tweets from the Twitter API. This is important because of the aforementioned Twitter-imposed limitation on how far back the API can go to retrieve tweets. It was important to get a streaming service set up so that we could collect as many tweets within that window as possible, as the web scraping efforts - while effective - were a fairly manual process and automation was desired. This service utilizes Apache Storm.

### ***Data Classification***

Once we've collected tweets, whether by web scraping or streaming the API, we must continue to preprocess them, in this case, by classifying the tweets. We've collected tweets based on the appearance of a particular medication name. This part of the architecture then determines

whether or not that tweet is about an adverse drug reaction, or simply a general remark about the drug.

## **Data Serving**

Now that we have the data in our Postgres database, we need a way to get it back out to the public. Here we have both the serving and application layer of the architecture. The serving layer is a *Flask*-based API that runs on a WSGI server provided by *gunicorn* and also a reverse proxy in *nginx*. The application layer sits on the MEAN stack, utilizing *MongoDB*, *Express.js*, *Angular*, and *Node.js*.

## **Architectural Considerations**

### **The Database**

We selected PostgreSQL for our primary database for a few different reasons. First, we didn't feel as though our overall throughput warranted a larger, distributed system like HDFS or Spark. Using an RDBMS like PostgreSQL provided enough power to handle the size of our data and it provides fast query response time. It would also scale with us pretty deep into the application.

Regarding database volumes: the FDA data produces 5GB a year, while tweets mentioning medications have been collected at the rate of 2GB a year for 80 medications ( with an additional 18GB a year in HTML files; those don't go into the database, but rather the S3 AWS repository).

Let's assume we will use a list of 800 medications. We would generate about 25GB of data each year. FDA offers data since 2004, meaning we could potentially benefit from collecting tweets from 2004 to 2016. This would produce a total of 300GB of needed space, with 25GB accumulating each following year. PostgreSQL can handle terabytes of data - the maximum table size is 32TB. At this rate, it would take 28 more years for our application to reach one terabyte of data!

Finally, we felt that an RDBMS provided us the most appropriate tool with which to serve our data. Existing python packages like Flask and psycopg2 could be easily leveraged to connect our frontend to the data in a performance-savvy way.

### **The Streaming**

Apache Storm was an obvious selection here. Storm offered more than enough power to stream tweets from Twitter and was a true streaming solution, rather than trying to patch together some kind of scheduled script to collect Tweets.

```

518989 [Thread-19-tweet-spout] INFO backtype.storm.spout.ShellSpout - ShellLog pid:7951, name:tweet
-spout Empty queue exception
519191 [Thread-19-tweet-spout] INFO backtype.storm.spout.ShellSpout - ShellLog pid:7951, name:tweet
-spout Empty queue exception
519393 [Thread-19-tweet-spout] INFO backtype.storm.spout.ShellSpout - ShellLog pid:7951, name:tweet
-spout Empty queue exception
519496 [Thread-23] INFO backtype.storm.task.ShellBolt - ShellLog pid:7953, name:parse-tweet-bolt pr
ozac: @DebraMessing take some Prozac
519598 [Thread-19-tweet-spout] INFO backtype.storm.spout.ShellSpout - ShellLog pid:7951, name:tweet
-spout Empty queue exception
519800 [Thread-19-tweet-spout] INFO backtype.storm.spout.ShellSpout - ShellLog pid:7951, name:tweet
-spout Empty queue exception
520002 [Thread-19-tweet-spout] INFO backtype.storm.spout.ShellSpout - ShellLog pid:7951, name:tweet
-spout Empty queue exception

```

## The API

We decided on an API for our serving layer because it made our backend more flexible. As an API, the data can be consumed by any software or device that can create http requests. It also added a layer of abstraction over the database. For example, if we allowed everything to connect directly to our RDS database, but then decided we needed to change solutions, we'd have to replace all of those connections.

We used *Flask* for the API because of our familiarity and its overall ease of use. From there, we used a combination of *gunicorn* (WSGI server) and *nginx* (reverse proxy) to serve up the API. This addresses scale issues for a production environment because the reverse proxy acts as a single contact point to the outside world. That means that as the service needs to scale, we can scale out and run the API over additional AWS instances and the *nginx* reverse proxy will act as a load balancer to better distribute the requests.

## The Application

The decision to run the application off the MEAN stack was made primarily to leverage MongoDB's document store as a sort of poor man's cache. As the application requires data on the front-end, it can pass the request to the Node.js backend, which can check the local MongoDB instance for the data.

If it is unavailable, it can then reach out to the API. In this case, it would also need to save the response from the API in the document store so that it would be available for the next request.

```

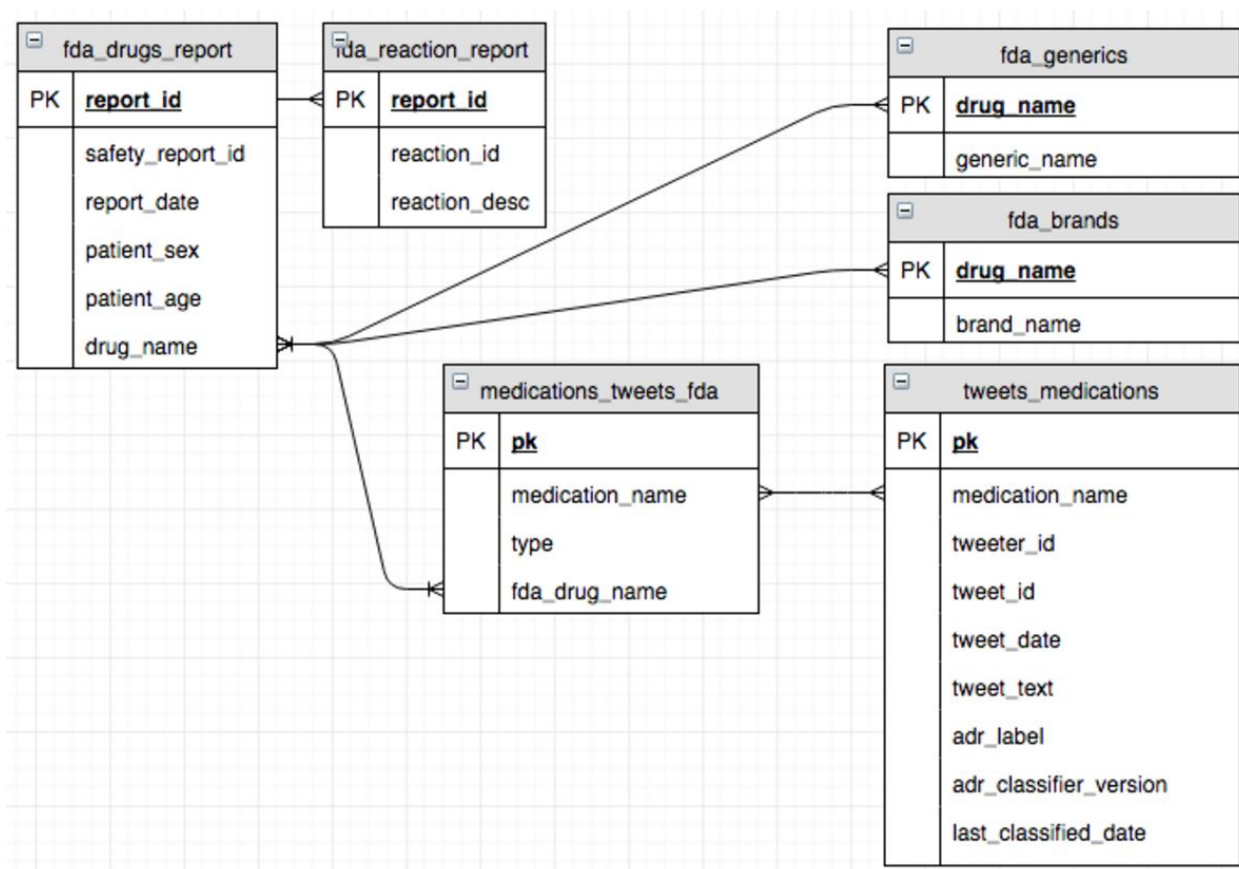
"FOOT WORKS HEALTHY ANTIFUNGAL": [
  {
    "drug": "TOLNAFTATE."
  },
  {
    "type": "brand"
  }
],
"REMEDY WITH PHYTOPLEX": [
  {
    "drug": "MICONAZOLE NITRATE."
  },
  {
    "type": "brand"
  }
],
"OKT3": [
  {
    "drug": "OKT3"
  },
  {
    "type": "brand"
  }
],

```

Because it describes past events, the data is highly unlikely to change and is a good candidate for caching.

## ER Diagram

Our database consists of five loaded tables: *fda\_drugs\_report*, *fda\_reaction\_report*, *fda\_generics*, *fda\_brands*, *tweets\_medications*. It also includes one derived table: *medications\_tweets\_fda*.



The drugs and reactions listed in each FDA ADR report were loaded into separate tables since there was no clear relationship specified between elements in the drug list and the reaction list for each report. We join the *fda\_reaction\_report* table to the *fda\_drugs\_report* through *report\_id* as a one-to-many relationship.

For each drug listed in the FDA ADR reports, there is a supplied list of the generic names and brand names associated with them. Therefore, we join both the *fda\_generics* and *fda\_brands* table to the *fda\_drugs\_report* table on the *drug\_name* field as a one-to-many relationship.

Finally, we found that the drug name listed in the FDA reports may be listed as the brand name or the generic name, while there is just a single corresponding medication name in our tweet



classifier list. Conversely, multiple *medication\_names* from the *medications\_tweets\_fda* table can relate to a *drug\_name*, so we have a many-to-many relationship.

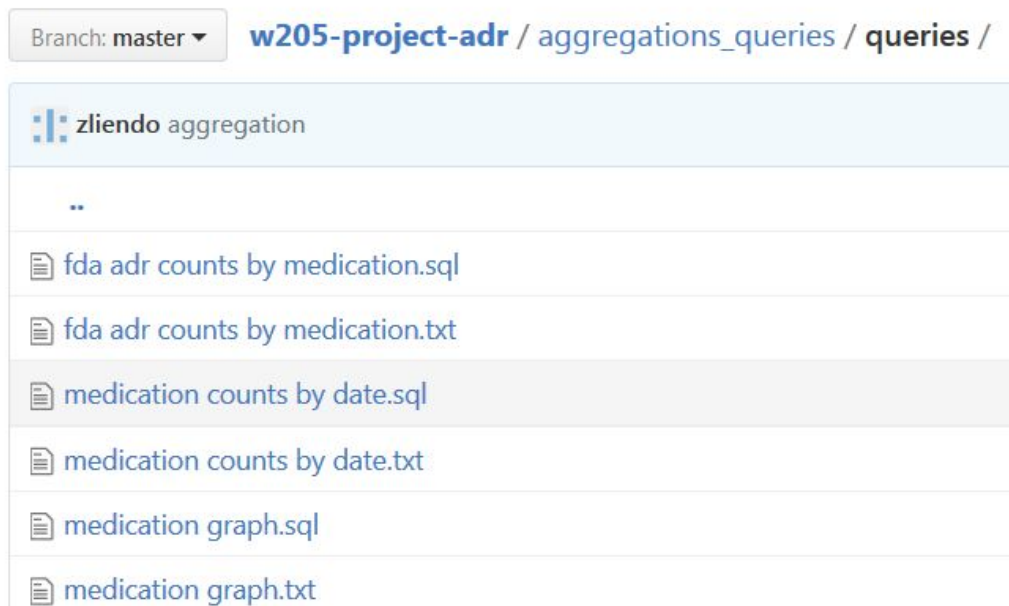
The *tweets\_medications* table includes all the tweets we streamed containing a given *medication\_name*. However, tweets can contain multiple *medication\_name* references. Therefore we have a many-to-many relationship between the *medications\_tweets\_fda* table and *tweets\_medications* table on the *medication\_name* field.

## Aggregation and Queries

For this prototype, we have the following aggregations:

- Creates and populates the *tweets\_medications\_search* table (lookup table), grouping information from the *tweets\_medications* table
- Creates and populates the *medications\_tweets\_fda* (lookup table), joining information from *tweets\_medication\_search*, *fda\_brands* and *fda\_generics* table

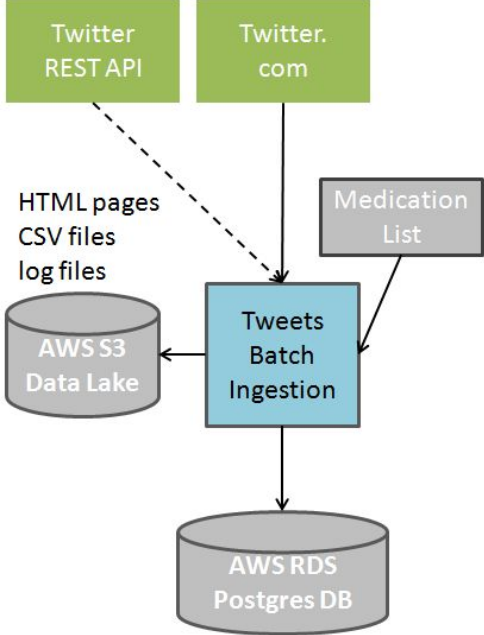
A sample of different type of queries and results can be found at: [Github queries repository](#)



## Twitter Batch Data Ingestion

This module collects historical tweets for a list of medications in a date range. We couldn't use the Twitter REST API because of tweets age limitation, so we instead used the search function on Twitter.com..

The batch process executes the following tasks:

- Reads the list of medications from a file and gathers the date range from input-parameters (different date range format options)
  - Access Twitter.com/search using the medication name and date ranges as keys
  - Receives HTML pages and scrapes page content using *BeautifulSoup* python package. This was a challenge since Twitter html pages are complex and formatting changes overtime.
  - Since the search results can span multiple web pages, this module simulates pagination in the web site by sending multiple search requests to twitter.com. Each request uses the oldest tweet-id from the previous page retrieved as the upper limit.
- 
- ```
graph TD; TwitterREST[Twitter REST API] -.-> Ingestion[Tweets Batch Ingestion]; TwitterCom[Twitter.com] --> Ingestion; MedicationList[Medication List] --> Ingestion; Ingestion --> AWS_S3[(AWS S3 Data Lake)]; Ingestion --> AWS_RDS[(AWS RDS Postgres DB)];
```
- The diagram illustrates the data ingestion workflow. It starts with three inputs: 'Twitter REST API' (green box), 'Twitter.com' (green box), and 'Medication List' (grey box). 'Twitter REST API' has a dashed arrow pointing to 'Tweets Batch Ingestion' (blue box). 'Twitter.com' and 'Medication List' have solid arrows pointing to 'Tweets Batch Ingestion'. From 'Tweets Batch Ingestion', two solid arrows point out: one to 'AWS S3 Data Lake' (grey cylinder) and another to 'AWS RDS Postgres DB' (grey cylinder). A label 'HTML pages CSV files log files' is positioned between the 'Twitter REST API' and 'Tweets Batch Ingestion' boxes.
- Insert tweets into the tweet\_medications db table. This table has unique indexes and constraints on the combination of tweet-id and medication-name to avoid inserting duplicates by mistake (in case the batch process is executed twice by mistake using the same range of dates or overlapping range). The unique index was not only by tweet-id since one tweet can mention more than one medication-name, needing two entries in the tweet\_medication table.
  - Creates three different type of files: \*.HTML to store full html pages retrieved for possible later use (in case we need to pull additional data from each tweet); \*.CSV files to store parsed data; and \*.LOG files capturing log information displayed in the console.
  - Output files are copied to the S3 repository

If, for any reason, the process stops, it can be restarted by reviewing the log files and starting a new ingestion process with a different data range. The option to restart could be automated in a future version. The program could retrieve the last medication, date and tweet-id downloaded and restart from there.

This [README.md](#) file has detail instructions on how to execute the twitter batch ingestion process and what is the expected output in the console.

91.K tweets ingested for Jan and Feb 2015; here a sample:

('prozac', 'Turmeric more Effective than Prozac at Treating Depression')

('prozac', '\*stops taking prozac\*\n\*has mood swings that make me want to vomit & then jump out the window while crying hard enough to fix global warming\*')

('humira', "Sure you might get TB, but your finger won't hurt as much. #Humira")

## Considerations of Scale

Historical Twitter batch data ingestion is a one-time process. Once it is finished and we have all past related medication tweets in our database, it will not be needed anymore since the streaming process will capture all current and future tweets.

We used an AWS EC2 m3.medium instance (1 CPU, 3.75 GB memory) for executing the batch ingestion process for two months of tweets (Jan and Feb 2015, 80 medications). It retrieved ~4GB of information during approximately three hours of processing (thousand of HTTP requests).

Even though this process will be executed only once, it can still take too long. Scaling up or out can be done using AWS resources. The batch process can be parallelized by medication-name and data-range, allowing multiple CPUs processing in parallel since there is no data aggregation or merging at this time.

In theory, this module could be used for other purposes, not only to search for tweets that mention medication-names but other topics of interest for different clients (e.g: list of movies, products, election candidates, etc). The application will need to host a solution to serve several clients at once, each requesting to search Twitter with a different list of search keys which could generate huge volumes of data arriving at the same time.

In this case, further scaling out would be necessary and a queue, such as *Kafka*, would allow us to better manage this process volume by distributing jobs automatically across several machines in a timely manner and having message replay options, if some process fails.

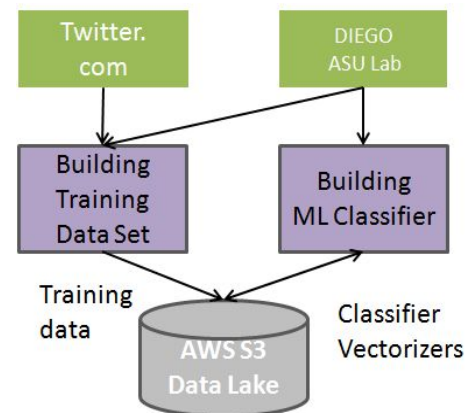
## Building Tweets ADR Classifier

### Building Training Data Set

The [DIEGO lab at Arizona University](#) provided a list of 10K tweet-ids with labels, indicating if the tweet describes an ADR or not.

They also provide a python script to download the corresponding tweets based on the tweet-id. We implemented just few customizations to:

- store training data set in a CSV file for later use
- count number of tweets retrieved successfully
- record a progress log



For details on how to execute the process to generate the training-set : [README.md](#)

### Building ML Classifier

[HLP @ UPenn IBI](#) (formerly DIEGO lab) provides open source code to build a twitter ADR classifier. This code can be found at this [bitbucket public repository](#).

We downloaded their source code and implemented the following customizations:

- Read training data from CSV files
- Split records into training and development sets
- Fixed a minor bug; refactored
- Tuned  $c$  and  $w$  hyperparameters
- Calculated RUC-AUC scores
- Wrote trained classifiers and vectorizers into PKL files for later use<sup>6</sup>
- Added code to print some messages into console for tracking progress

The resulting classifier had the following performance scores:

Precision for the ADR class .. 0.517647058824  
Recall for the ADR class .. 0.494382022472  
ADR F-score .. 0.505747126437  
roc\_auc 0.852958295374

For details on how to re-build this classifier: [README.md](#)

Future Improvements would include collecting more labeled examples for the training set and improving classifier performance scores.

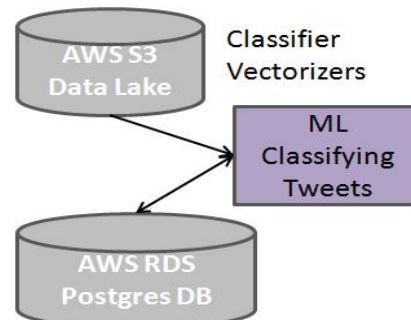
<sup>6</sup> PKL files available in the S3 repository.

## ML Classifying Tweets

This module classifies tweets using the ML classifier built previously.

It takes the following steps:

- Loads vectorizers and classifiers from PKL files into memory
- Reads the database to look for the list of medications with tweets not yet classified
- For each medication found:



- Gets a list of not classified tweets from the database (tweet-id and tweet-text)
- Executes the ML classifier (SVM) for this set of tweets
- Updates the database table (adr\_label) with the result of the classification. In order to keep performance levels high, it uses a batch INSERT and then an UPDATE JOIN to avoid multiple single updates (two db access instead of thousands)

The tweets\_medication db table have fields to keep track on when the tweet was classified and using which version of the classifier.

From 91.4K tweets (Jan and Feb 2015), the classifier found 858 ADR tweets.  
Here is a sample of True Positives :

('zyprexa','catapres zyprexa...you'll gain 10lbs a month on them, you'll also think demonic forces are in control of your body')

('tamiflu', 'Tamiflu makes me nauseous')

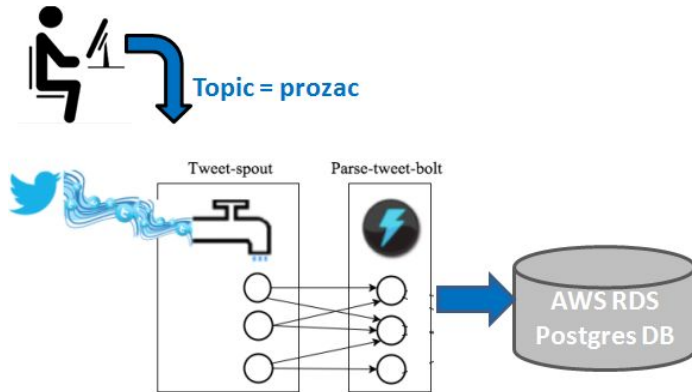
('humira', 'I love humira for making my quality of life better but I get the worst headaches after the injections')

For details on how to run the process to classify tweets: [README.md](#)

Future Improvement would include implementing parallel jobs to classify tweets faster.

## Tweets Streaming

Streaming was implemented using Apache Storm and Streamparse, which addresses future scalability options.



The script that starts Apache Storm receives one parameter for the *medication\_name* to search in Twitter.

For example: `streaming.sh prozac`  
If no *medication\_name* is provided, 'tamiflu' is the default.

Tweets are stored in the *tweets\_medication* table.

**Future improvement:** Providing a list of medications instead of just one (or reading list of medications from the database) and matching arriving tweets to one or more medication-name search keys. Adding a bolt to classify tweets on the fly.

Here is a screenshot of the console while tweets arrive:

```
64779 [Thread-19-tweet-spout] INFO backtype.storm.spout.ShellSpout - ShellLog pid:5325, name:tweet-spout Empty queue
exception
64981 [Thread-19-tweet-spout] INFO backtype.storm.spout.ShellSpout - ShellLog pid:5325, name:tweet-spout Empty queue
exception
65157 [Thread-23] INFO backtype.storm.task.ShellBolt - ShellLog pid:5326, name:parse-tweet-bolt prozac: RT @monalisa
068: #MyShrinkCalled and said that marijuana and whiskey are not acceptable substitutes for Prozac
65252 [Thread-19-tweet-spout] INFO backtype.storm.spout.ShellSpout - ShellLog pid:5325, name:tweet-spout Empty queue
exception
65454 [Thread-19-tweet-spout] INFO backtype.storm.spout.ShellSpout - ShellLog pid:5325, name:tweet-spout Empty queue
exception

445860 [Thread-19-tweet-spout] INFO backtype.storm.spout.ShellSpout - ShellLog pid:5325, name:tweet-spout Empty queue
exception
446062 [Thread-19-tweet-spout] INFO backtype.storm.spout.ShellSpout - ShellLog pid:5325, name:tweet-spout Empty queue
exception
446169 [Thread-23] INFO backtype.storm.task.ShellBolt - ShellLog pid:5326, name:parse-tweet-bolt prozac: i accidenta
y basted my friend's ham with prozac https://t.co/JQ4EZ2yNEi
446263 [Thread-19-tweet-spout] INFO backtype.storm.spout.ShellSpout - ShellLog pid:5325, name:tweet-spout Empty queue
exception
446467 [Thread-19-tweet-spout] INFO backtype.storm.spout.ShellSpout - ShellLog pid:5325, name:tweet-spout Empty queue
exception

exception
448701 [Thread-19-tweet-spout] INFO backtype.storm.spout.ShellSpout - ShellLog pid:5325, name:tweet-spout Empty queue
exception
448811 [Thread-23] INFO backtype.storm.task.ShellBolt - ShellLog pid:5326, name:parse-tweet-bolt prozac: They can't b
acked you imbisole they aren't online god take your prozac please https://t.co/Z9PJvYMIAs
448905 [Thread-19-tweet-spout] INFO backtype.storm.spout.ShellSpout - ShellLog pid:5325, name:tweet-spout Empty queue
exception
449107 [Thread-19-tweet-spout] INFO backtype.storm.spout.ShellSpout - ShellLog pid:5325, name:tweet-spout Empty queue
exception
```

## Lessons Learned

### ***Twitter Data Has Its Limits***

Our biggest challenge out of the gate was in trying to determine ways to deal with acquiring Twitter data for our project. Although the API does have rate limits, there is also an implicit time limit, only allowing someone to fetch tweets from the past week or so.

Acquiring large amounts of past tweets then is a challenge. Eventually, we solved this challenge by scraping the web, as the search on the Twitter website does not suffer from the same up-to-a-week-old-only implicit limit as the API.

### ***Finding Balance Between Data Management and Performance***

When you think about your data design, there is so much more to consider than simply entities and their relationships. Early on in the process, as we were buried in data, our data design became more and more denormalized and we were more concerned with performance. To try to stop while processing gigabytes of incoming FDA data, for example, to try to assign unique “drug\_id” values, and to fetch the matching existing one if there was one, was impossible.

*Schema on write* felt natural because we were in an RDBMS, but we quickly realized that didn’t mean anything. However, once we got a handle on the data and tried to pivot to the serving layer, we realized that we were back at the same impasse. The serving layer had its own needs. The biggest challenge we faced was trying to find an efficient way to connect the Twitter and FDA data, such that it could still be read and aggregated for purposes of serving the data.

### ***What’s in a name?***

Another lesson we learned was simply in creating a consistent vocabulary about the different data points, especially name. Consider the FDA data, which used *medicinal\_product* as its unique name overlord, but also provided *generic* name alternatives, as well as *brand* names. In some cases, the generic names, for example, may only differ by a letter or two, simply because of spelling variations.

Which do we use? Which do we display to users? How do we even talk about these so that everyone knows which one we’re talking about? These were all challenges we faced, and we learned that sometimes it’s worthwhile to approach the problem from various sides. From the data, *medicinal\_product* seems like your common thread, but when you view it from the consumer’s point of view, they likely want to see the generic and brand names because that’s how they’re used to calling it.

In the end, we determined a different vocabulary for different use cases. When it came to the data, *medicinal\_product* became *drug\_name* and served as the connector between FDA and



Twitter data. However, when it came to the user, they didn't need to see all three: medicinal\_product, generic\_name, and brand\_name. In this case, they would see the generic and brand names, but these would remain linked to the drug\_name in the background so that we could find the correct data to display.

## **Future Roadmap**

### ***Complete Initial Visualizations***

Although we were able to complete the end-to-end architecture, from data ingestion all the way out to the web application, we weren't able to fully flesh out the data within this pipeline to get to meaningful visualizations. The first next step would be to complete these.

### ***Add Demographics and Analyze***

Next, we'd want to attach demographic information to our users. This information is available to a certain extent from either data source as the FDA reports record some demographic information, and other information is available by user on the Twitter website.

We'd likely need a third-party to augment our data, but adding demographic data to the ADRs would add some relevant depth. Is a specific demographic more at risk for a particular ADR, for instance? Also, mapping ADRs to different demographics may help us better forecast drug interactions.

### ***Classify Tweets During Streaming***

By adding a second bolt to classify incoming tweets on the fly, we could condense our architecture to no longer have to classify as a separate step. While this was necessary during development as we scraped archived Twitter data, a production instance of our application would run strictly off streamed tweets, so this change would make the process more self-contained.

We would not need to be concerned about scalability here. Since the FDA releases its data no more often than once a quarter, we wouldn't need to be delivering findings in real-time and would be able to keep up with demand over time.

### ***Network diagram visualization; Pathing an ADR***

A really interesting long-term goal with the project would be to eventually add a network diagram visualization. By using a bipartite graph and documenting both the patients and the medications, we could start to get a better understanding of how patients react to medications over time, and also how medications interact with each other.



## Resources

### *Github Repositories*

We do not have a single github repository because for the API and web application, it made sense to separate them out. These repos contain only the code required to make them function. The reason for this is so that if you wanted to stand up a new API server, for instance, you could do so by simply cloning the repository, and wouldn't have additional scripts or data on the server.

Main / Data Ingestion Repository

<https://github.com/zliendo/w205-project-adr>

API Repository

<https://github.com/mattbryanswan/w205-api>

Web Application Repository

<https://github.com/mattbryanswan/w205-webapp>

Upload

Create Folder

Actions ▾

All Buckets / w205final

|                          | Name                                                                                                  | Storage Class |
|--------------------------|-------------------------------------------------------------------------------------------------------|---------------|
| <input type="checkbox"/> |  adr_ml_classifiers  | --            |
| <input type="checkbox"/> |  fda_adr_json_files  | --            |
| <input type="checkbox"/> |  loading_tweets_logs | --            |
| <input type="checkbox"/> |  training_set        | --            |
| <input type="checkbox"/> |  tweet_csv_files     | --            |
| <input type="checkbox"/> |  tweet_html_files    | --            |

### Instructions to Clone and Replicate

Please look at the corresponding README.md file for instructions on how to clone and replicate each of the following modules

|   |                                              |                           |
|---|----------------------------------------------|---------------------------|
| 1 | Tweets Data Ingestion and ADR Classification | <a href="#">README.md</a> |
| 2 | FDA Data Ingestion and Processing            | <a href="#">README.md</a> |
| 3 | Aggregation and Queries                      | <a href="#">README.md</a> |

### Functioning API

The API has been stood up on a t2.micro (free tier) instance. Once configured, we created an upstart script and allowed it to remain accessible. This instance will remain up for review.

Drugs endpoint (returns a list of generic/brand names):

<http://54.173.130.143/drugs>