Prismatic Architecture - Using Machine Learning on Social Networks to Figure Out What You Should Read on the Web

Monday, July 30, 2012 at 8:15AM

Todd Hoff in Example, Machine Learning

This post on Prismatic's Architecture is adapted from an email conversation with Prismatic programmer Jason Wolfe.

What should you read on the web today? Any thoroughly modern person must solve this dilemma every day, usually using some occult process to divine what's important in their many feeds: Twitter, RSS, Facebook, Pinterest, G+, email, Techmeme, and an uncountable numbers of other information sources.

Jason Wolfe from Prismatic has generously agreed to describe their thoroughly modern solution for answering the "what to read question" using lots of sexy words like Machine Learning, Social Graphs, BigData, functional programming, and in-memory real-time feed processing. The result is possibly even more occult, but this or something very much like it will be how we meet the challenge of finding interesting topics and stories hidden inside infinitely deep pools of information.

A couple of things stand out about Prismatic. They want you to know that Prismatic is being built by a small team of four computer scientists, three of them very strong young PHDs from Stanford and Berkeley. Halfhazard methods won't do. They are bringing brain power to solving the information overload problem. But these PHDs are also programmers, working on everything from websites, iOS programming, as well as the sexy BigData/ML backend programming.

One of the things that excited me about Prismatic as an architecture is that a problem that will need to be solved over and over again in the future is applying Machine Learning to great streams of socially mediated information in real-time. Secrecy prevents them saying very much about their Machine Learning tech, but we do get a peak behind the curtain.

As you might expect, they are doing things a little differently. They've chosen Clojure, a modern Lisp that compiles to Java bytecode, as their programming language of choice. The idea is to use functional programming to build fine-grained, flexible abstractions that are composed to express problem-specific logic.

One example of functional power is their graph library, which they use all over the place. For example, a graph of computations can be described to create the equivalent of a low-latency, pipelined set of map-reduce jobs for each user. Another example is the use of subgraphs to compactly describe service configuration in a modular way.

Given this focus on functional elaboration, they avoid large frameworks like Hadoop, going for a smaller, more reliable, easier to debug, easier to extend, and easier to understand codebase.

A criticism of Prismatic's approach is the long training periods needed to get results. First, they say it doesn't take that long at all to start getting good content. Second, I would add, start thinking about these types of systems using the Long Sight. ML based recommenders will start being trained from childhood and will stay with you your entire life. A scalable digital analog of your mind will act as both information gatekeeper and wingman.

In a Tech Crunch article Prismatic founder Bradford Cross pithily describes Prismatic as being “built around a complex system that provides large scale, real-time, dynamic personalized re-ranking of information, as well classifying and grouping topics into an ontology.” Now let’s see what that system looks like...

Stats

- Every day millions millions of new news articles, blog posts, and other shared content are fetched and analyzed for tens of millions of social signals from Twitter, Facebook, Google Reader, and around the web.
- Within seconds of a user signing up for Prismatic enough of their historical activity is fetched and analyzed to make pretty solid topic and publisher suggestions. Their social networks are digested to find friends who seem to share the most relevant content. Every interaction users make on Prismatic is also a learning opportunity.
- Within a few tenths of a second after visiting a home feed, a feed is produced of the most interesting stories by matching a model of user interests against all of this content.
- Millions of article impressions are served to users every week

Platform

- Hosted on EC2/Linux.
- 99.9% of backend pipeline and API servers are written in Clojure, a modern Lisp that compiles to Java bytecode.
- All heavy lifting happens inside the JVM.
- MongoDB
- MySQL

- S3
- Dynamo
- A focus on building custom code to efficiently solve particular problems.

Data Storage and IO

- Rather than a typical architecture where each service reads and writes live data to and from a database or distributed filesystem, systems are designed around the data.
- Most of data flows through a backend pipeline from one service to the next without any round-trips to disk.
- At the end of the pipeline, API machines rely heavily on custom in-memory data structures that are periodically snapshotted to disk.
- By keeping the data as close to the CPUs as possible, most API requests are served with very little to no IO. This allows serving up feeds with very low latency, which keeps the API machines CPU bound, and makes gracefully scaling to handle more users much easier.
- A number of off-the-shelf storage solutions are used: MongoDB, MySQL, and Amazon's S3 and Dynamo. Each has been chosen carefully to match the size, access patterns, and other characteristics of the data to be stored.

Services

- At a high level, the architecture is split up into about 10 distinct service types, in roughly 5 different categories: Data Ingest - Backend; Onboarding; API - Client Facing; Other Services - Client Facing; Batch and Other Services.
- Each service is designed to do one kind of thing, be horizontally scalable in a particular way, and often be limited by a particular

resource type or two (IO, CPU, RAM).

- Economics favor quite large machines, so services are singletons, but there’s no expectation major bottlenecks will prevent horizontally scaling.

Data Ingest - Backend

- Tons of great content (news articles, blog posts, other web pages, etc) are created each day, and Prismatic wants to know as much of it as possible.
- For each piece of content it must be known who is sharing it and what they are saying about it so relevant commentary can be shown alongside an article and so a user can be shown content shared by their friends or people with similar tastes.
- The first step in this process is ingesting and analyzing the content and social data. At the top of the ingest pipeline are pollers:
    - A RSS poller loops through feeds looking for new articles
    - Twitter and Facebook pollers connect to the corresponding APIs and fetch comments/tweets from users and their friends.
    - These services are pretty simple, and mostly stateless.
    - The only real state, and the interesting part, is in figuring out what's been seen already and intelligently prioritizing what to poll next.
    - One of the most difficult problems is actually figuring out what piece of content each social interaction is about, before it flows into the rest of the pipeline. This is hard since people may share many versions of the same article (shortened link, mobile and regular versions, etc).
- From here, RSS entries and comments/shares/tweets (with properly canonicalized URLs) flow into a barrier, where it is decided what to actually fetch and analyze.
    - Spam and other junk are eliminated later in the pipeline, but

the best viagra spam article is the one no further cycles are wasted on.
    - URLs that make the cut are passed on to the fetching/analysis pipeline
    - URLs from this phase go into the fetching/analysis pipeline. This is where a lot of the magic starts happening.
    - A queue of URLs is kept, each of which runs through a 'graph' that successively elaborates the URL, fetching its HTML, applying machine learning algorithms to extract the text of the article, identify the best images, extract the publisher, label with applicable topics, and so on.
    - A whole lot of engineering has gone into making these algorithms run fast, fit into memory, and perform well. The problem of handling all the URLs itself is of course embarrassingly parallel.
- At the end of the ingest phase is the doc master, whose job is to receive the elaborated articles and social context about them (which continues to trickle in over time), match them up, cluster the articles (in an online fashion) into story clusters, decide on the current active set of docs, and manage the indices of the API machines.

Onboarding - Backend

- Onboarding is ingesting data about new users so they can be provided with a great personalized experience within seconds of signing up for the app. This has two major components:
    - Figuring out suggested topics and publishers for a user based on their activity on Twitter, Facebook, or Google Reader,
    - Analyzing the social graph of a user to try to deduce which friends' shares are the most powerful signals that they will like an article.
- These services are embarrassingly parallel, this time across users.

Aside from the intricacies of efficient social graph analysis, which Prismatic doesn’t want to say too much about, the key is how these services have been tuned to have quite low latency and reasonable throughput:

- For example, for users who are active on Twitter, Facebook, or Google Reader, it’s possible to compute more than a hundred accurate topic and publisher suggestions in 15 seconds or less. This is fast enough that suggestions can start being computed after users OAuth in, and usually be ready with personalized suggestions by the time the user has finished creating their account (selecting a handle and password) and reached in walkthrough.
- In these 15 seconds, quite a lot happens:
    - The user's recent posts are fetched on Twitter and Facebook, articles they've liked on Google Reader, and so on. This in itself may take 10 or more seconds to finish.
    - Identification of a collection of up to a thousand or so unique URLs shared by the user, her friends, etc.
    - These flow into a version of the fetching/analysis pipeline, where all of the URLs are fetched and elaborated with the same ML stack above
    - The results of this analysis are aggregated, post processed, and saved off to DynamoDB and S3

- The latency of this process is really critical, so these processes can’t be executed serially -- they have to be pipelined and parallelized as much as possible.
- Throughput is also important because the process is quite heavyweight and when there are a lot of signups it takes a lot of effort to keep latency down. This is where Prismatic’s stream processing and aggregation libraries really pull their weight, allowing the performance of the equivalent of a low-latency,

pipelined set of map-reduce jobs for each user, for multiple users in parallel, using close to full capacity of a machine.

API - Client-facing

- The Onboarding and Data Ingest services come together in the API machines.
- Not much will be said say about the actual feed generation process, other than it's a complex and highly optimized process with lots of stages that matches the user's 'fingerprint' and query against the index, retrieves, ranks, and pages the resulting feeds in a few hundred milliseconds.
- The major design goals/challenges here from a systems perspective are:
    - Recent articles must be indexed and available for low-latency feed generation
    - The index is quite large (many gigabytes), and must be kept up-to-date in real time, so that users can find breaking stories
    - Users should be load balanced across machines, and it should be relatively easy and fast to spin new machines up (and shut them down) in response to load
    - Generating good user feeds requires more than just the index -- we also need the user's 'fingerprint' -- their interests, social connections, articles they've recently seen, and so on.
    - This fingerprint is quite large, and constantly changes as the user views and interacts with content on the site.
- The solution to the first few problems involve the doc-master.
    - The doc-master machine organizes our current document set, preprocess the docs, and every few minutes writes a set of prepared index files to S3.
    - When a new API machine spins up, it first reads these files and dumps them into memory, giving it a nearly-up-to-date

copy of the index.
    - The doc-master also publishes commands for index changes (document/comment additions and deletions, story clustering changes, etc) in real-time to all of the live API machines
    - When a new machine comes up, it replays the history of the last few minutes of changes to bring its index up to the present.
    - Other general (non-user-specific) state needed by the machine is also read from S3 into memory and periodically refreshed, or stored in dynamo if the data is larger and access is less frequent.
- The remaining problem is how to efficiently serve up feeds for a user without incurring the IO cost and latency of fetching and updating the fingerprint on each request.
    - The approach here is to use sticky sessions that bind the user to an API machine
    - When the user first signs in, their data is all loaded into memory in an expiring, flushing write-back cache
    - Throughout the user's session, this data stays in memory and is used to generate their feeds
    - All user actions pass through this same API machine throughout a session
    - Updates to less crucial parts of the fingerprint are batched and flushed to redundant storage every few minutes, when the session expires, or when the machine shuts down
    - More crucial updates are done synchronously or at least with a direct write-through cache.
- A high IO hit is paid for reading the user's fingerprint only once throughout the course of their session, which is amortized across the (typically) relatively large number of feed clicks, pages, article clicks, shares, and so on the user takes, and limit the number of writes back of this data. When a machine comes down the data is flushed and the user is moved to another machine -- in the worst

case a few minutes of non-critical data will be lost for some users, which is consider worth it for the benefits in simplicity and scalability. This loss is always recoverable via later batch jobs over raw event logs.

Other services - Client facing

There are a few other separate client-facing services:

- Public-feeds - which does smart caching of topic and publisher feeds for non-logged in users, fetching them from the regular API on demand and allowing multiple ages of each feed to be paged through.
- Auth - which handles account creation, sign-in, and so on. Mostly a thin layer in front of the SQL database, which stores critical user data that needs to be periodically snapshotted and backed up.
- An URL-shortener

Batch and other services

- Other services for machine language training, data archival, and event tracking/analytics.
- MongoDB is used to store server metrics and user analytics, largely because it supports a nice low-hassle story for sending raw events of different shapes, maintaining the right indices, and keeping online roll-ups for counts.

Graph Library

- This is a really nice way to declaratively describe a graph of computations, which plays to Clojure's strengths. A graph is just a Clojure map, where the keys are keywords and values are keyword functions that take values computed by other functions in the map, as well as external parameters.

- It is used all over the place; for instance, the doc-analysis pipeline is a graph, where each elaboration of a document may depend on the previous (e.g., identifying topics for a document depends on having already extracted its text); the newsfeed generation process is a graph composed of many query and ranking steps; and each of the production services is itself a graph, where each resource (e.g. data store, memory index, HTTP handler, and so on) is a node that can depend on others.
- This makes it easy to do things like:
    - Compactly describe service configurations in a modular way (e.g., using subgraphs)
    - Produce charts diagramming out the dependencies of a process or service
    - Measure the computation time and errors that happen at each node in a complex computation like document analysis, or activity on each resource of a complex service like API
    - Smartly schedule the different nodes of such computations on different threads (or in theory even machines)
    - Easily write tests of our entire production services, by replacing a few nodes in the graph with mocks (e.g., faking out storage) and so on.

Machine Learning on Documents and Users

Documents and users are two areas where Prismatic applies ML (machine learning):

ML on Documents

- Given an HTML document:

  - learn how to extract the main text of the page (rather than the sidebar, footer, comments, etc), its title, author, best images, etc
  - determine features for relevance (e.g., what the article is about, topics, etc.)
- The setup for most of these tasks is pretty typical. Models are trained using big batch jobs on other machines that read data from s3, save the learned parameter files to s3, and then read (and periodically refresh) the models from s3 in the ingest pipeline.
- All of the data that flows out of the system can be fed back into this pipeline, which helps learn more about what's interesting, and learn from mistakes over time.
- From a software engineering perspective one of the most interesting frameworks Prismatic has written is the 'flop' library, which implements state-of-the-art ML training and inference code that looks very similar to nice, ordinary Clojure code, but compiles (using the magic of macros) down to low-level array maniuplation loops, which are as close to the metal as you can get in Java without resorting to JNI.
- The code can be an order of magnitude more compact and easy to read than the corresponding Java, and execute at basically the same speed.
- A lot of effort has gone into creating a fast running story clustering component.

ML on Users

- Guess what users are interested in from social network data and refine these guesses using explicit signals within the app (+/remove).
- The problem of using explicit singnals is interesting as user inputs should be reflected in their feeds very quickly. If a user removes 5 articles from a given publisher in a row, then stop showing them

articles from that publisher right now, not tomorrow. This means there isn't time to run another batch job over all the users, The solution is online learning: immediately update the model of a user with each observation they provide us.

- The raw stream of user interaction events is saved. This allows rerruning later the user interest ML over the raw events, in case any data is lost through slightly loose write-back caches on this data when a machine goes down or something like that. Drift in the online learning can be corrected and more accurate models can be computed.

Lessons Learned

- **Find your story**. Think very carefully about the entire pipeline and all of the data that flows through it. Work around scalability challenges with problem-specific solutions. Each service has its own scaling story built in and the services communicate in a way that should make it easy to scale each component without putting too much pressure on the others. Starting instead with a system built around Hadoop jobs, where all our data was stored in raw form in a distributed database/filesystem, Prismatic would be telling a different story.
- **Find embarrassingly parallel opportunities and exploit them**.
- **Do work in parallel while the user is waiting**. The onboarding process, for example, ingests data about new users so they can be provided with a great personalized experience within seconds of signing up for the app.
- **Use functional programming** to build fine-grained, flexible abstractions. These abstractions are composed to express problem-specific logic.
- **Avoid large, monolithic frameworks** like Hadoop. This yields a smaller code base, which, all things equal, is less error prone,

simpler to understand, and easier to extend. Built own libraries simply because much of the functionality of open-source code is locked up into monolithic frameworks and not easily re-usable, extensible, or debuggable when something breaks or fails to scale.

- **The right people are vital for making this all work**. The current backend team consists of three CS Ph.Ds, working on everything from ML (machine language) algorithms to low-level systems engineering to web and iPhone client code.
- **All code is put into production early and often** though an investment in tools that make building and debugging production services easy and fun.
- **Keep it simple**. Don't take on the burden of complex libraries or frameworks for simple stuff, and don't get fancy when a much simpler solution is good enough. For example, use a simple HTTP-based messaging protocol rather than one of the popular frameworks. When it makes sense, be happy to pay a bit more for a managed solution such as S3 or Dynamo that just works.
- **Invest in building great tools and libraries**. For instance, Prismatic’s 'flop' library allows them to write number-crunching machine learning algorithms that are as fast as Java, in 1/10 of the code. 'Store' abstracts away many unimportant details of key-value storage, allowing the writing of higher-level abstractions for caching, batching, and flushing that work in a variety of circumstances, across a variety of storage engines. And 'graph' makes writing, testing, and monitoring distributed stream processing services a breeze.
- **Think carefully about each type of data**, and don't expect to find a one-size-fits-all solution for IO and storage.

Related Articles

- Prismatic wants to be the newspaper for a digital age

- Prismatic Blog
- Keynote on Clojure
- Prismatic Hopes to Create a New Category of Social News
- “In the Studio,” Prismatic’s Bradford Cross Wants to Reinvent Social News
- Software Engineering at Prismatic
- How Prismatic deals with data storage and aggregation
- DataSift Architecture: Realtime Datamining At 120,000 Tweets Per Second

---