Deploying predictive models with the Actor framework

Brian Gawalt
Upwork, Inc.
bgawalt@upwork.com

October 20, 2015

Abstract

The majority of data science and machine learning tutorials focus on generating models: assembling a dataset; splitting the data into training, validation, and testing subsets; building the model; and demonstrating its generalizability. But when it's time to repeat the analogous steps when using the model in production, issues of high latency or low throughput can arise. To an end user, the cost of too much time spent featurizing raw data and evaluating a model over features can wind up erasing any gains a smarter prediction can offer.

Exposing concurrency in these model-usage steps, and then capitalizing on that concurrency, can improve throughput. This paper describes how the Actor framework can be used to bring a predictive model to a real-time setting. Two case-study examples are described: a simple text classifier (with accompanying code), and a live deployment built for the freelancing platform Upwork.

1 The practice of machine learning

Imagine a firm has brought a specialist in machine learning onto a new project. The firm wants a product which can provide a quality prediction about some regular event happening in the course of the firm's business. The specialist is handed a pile of relevant historical data, and asked: Among the new customers seen for the first time today, who's likeliest to be a big spender? Or: of all the credit card transactions processed in the last hour, which are likeliest to be fraudulent? Or: when a customer enters a query into our website's Search tool, what results should we be returning?

The specialist starts with the first of two phases of their project. They have to identify a model that can be expected to fit predictions over both the historical data and in a way that will generalize to new data. The familiar version of the steps involved in supervised learning:

1. Identify raw source data, and break it down into distinct observations of the pattern you're trying to learn and predict.

2. For each raw observation, produce a p-dimensional vector of features and a scalar label.

3. Split this collection into disjoint training, validation, and testing sets.

4. For each candidate model (and/or each hyperparameter value of the model/models), fit model parameters to the training vectors and labels, and evaluate the goodness of fit by performing prediction of the validation labels given the validation vectors

5. Select the model whose validation-set predictions came closest to the mark. Use it to then make predictions over the test set. Report this test set performance to vouch for the predictive model you've generated and selected.

Note that this first phase doesn't carry an explicit component of *time urgency*. All else equal, a typical specialist will prefer that the full sequence complete in six hours, and six minutes is better still. But if it takes six days instead, nothing *fundamental* to this first phase has been threatened. The task – finding a model that generates acceptably accurate predictions on new data – is accomplished.

The second phase is to actually put the model's capabilities to use. Given new events and observation that need scoring by the model – is this new customer likely to be a big spender? is this credit card legitimate? – the above featurization and scoring routines need to be run. And in this real-world deployment, it's likely that there are also some strict constraints on how long it takes this sequence to run. All predictions go stale, and some use cases need to act on a prediction within milliseconds of the event itself.

There are some cases where these latency constraints aren't binding. The exact same featurization and scoring routines used to generate and validate the model can be re-run fast enough on new data to produce useful predictions. But this paper focuses on the cases where timeliness requirements exclude the use of the same software developed in the first phase as the backbone of the second phase. What can a lone machine learning specialist do to retool their sequence to run in a production environment?

1.1 Moving to production

If the original software, used to generate and validate the predictive model, is suffering from too-low throughput in producing new predictions, one path forward could be to incorporate more concurrent processing. The three steps to prediction (gathering raw materials, featurizing those materials into vectors, scoring the vectors) can transition from a serial sequence to a pipeline.

Figure 1 demonstrates a modification of the scoring task flow, producing predictions of N events in a sequential and a concurrent pattern. This pipeline

offers a few advantages. Scores are produced with less delay after the raw material gathering (useful in case the information in that material is at risk of going stale or out of date).

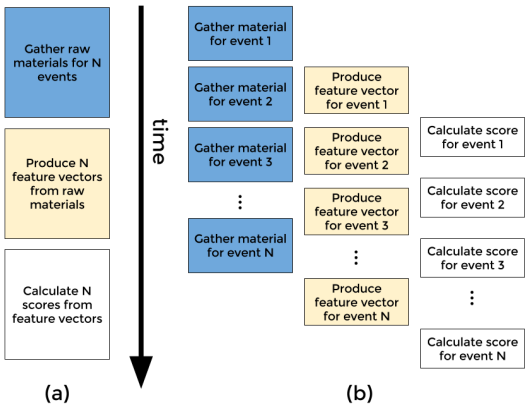

Figure 1: (a) The scoring procedure performed serially. (b) The individual tasks for each event to be scored performed in a concurrent pipeline.

Most importantly, this redesign provides a clear path forward to speed-up in completing all N scoring tasks. If a system can genuinely perform the concurrent tasks in parallel, as a multicore system might, one can easily picture adding "clones" of this pipeline simultaneously processing more and more partitions of the N events.

1.2 Complexity costs

It can be difficult to put together, from scratch, a high-performance concurrent computing system. It's easy to fall into traps of false sharing, deadlocking, and other difficulties. It's not impossible, but it's definitely tricky and time-consuming, and building a new concurrent system for a single project might fail a cost-to-benefits ratio test.

Fortunately, lots of great work has produced platforms to take this plan to a truly wide scale implementation. Apache Storm, Apache Spark (esp. Spark Streaming), and Twitter's Heron all try to distribute this kind of event-driven computing across multiple machines to ensure the highest possible throughput.

Unfortunately, they're complicated systems. Familiarizing oneself with the API, and managing the underlying infrastructure, requires considerably more expertise above and beyond that of our original model in Figure 1(a). If spreading the load over multiple machines is the only way to meet the required service level of the prediction system, this additional complexity will have to be met with additional resources: maybe the firm will have to start allocating more engineers in addition to more hardware, to what originally was a project of just the single machine learning specialist.

This paper is here to specifically recommend a midpoint between a from-scratch concurrent framework and the mega-scale offerings. The Actor framework offers a simple way to reason through concurrent problems, while still being flexible enough to accommodate a variety of approaches to any one problem. The ease

From here, this paper presents a case study where an Actor framework was key to bringing a predictive model to production. It interleaves this story with a summary of what an Actor framework entails, as well as a description of the Actor implementation in the Scala library Akka. It concludes with a reiteration of the advantages (and disadvantages) the Actor framework offers and a discussion of why comfort with Actors can help machine learning specialists with their work and careers.

2 Case Study: Predicting worker availability at Upwork

Upwork is an online labor platform, connecting freelancers and clients for internet-enabled remote work. There are several patterns for how these work arrangements are struck[1], but one important avenue involves a client using the platform's search engine to discover new freelancers they think would be well suited to the project the client has in mind.

When the client issues a query, e.g., "java developer", "restaurant menu design", "paralegal", they expect to see a result set filled with freelancers with relevant experience who are likely to perform well on a new assignment. The client can then invite any of these freelancers to interview for an open position.[2]

This adds a third requirement to the query's result set: listed freelancers should not only be relevant and high-performing, they should also be receptive to an interview invitation at the time of the query. If the system returns a list of the same ten excellent freelancers are returned for every search for "wordpress template," it's a recipe for those ten freelancers being deluged with opportunities they're to busy to fill (and for the freelancers ranked just outside that top ten being unreasonably starved for job invitations).

There was an existing heuristic in place to try and capture this, but it was felt a learned model trained on historical data could easily provide higher accuracy.

[1] Freelancers can send proposals for a client's publicly posted job opening; groups of freelancers form agencies to share many client workloads; clients and freelancers, already working together outside the platform, moving their relationship onto Upwork to take advantage of the site's payment processing, escrow, or other services

[2] Upwork sees the two actions happen in both orders. Sometimes, clients post a job opening, then go searching for freelancers to invite to interview; other times, the client starts by searching and browsing freelancer profiles, and creates the job post once they see someone they'd like to work with.

2.1 Building the predictive model

If the system should only show freelancers available to interview, how should we estimate freelancer availability? This question can be gently reformulated into a question of binary classification: "If Freelancer X were to receive an invitation to interview right now, does their recent use of the site suggest they would accept that invitation, or would they instead reject or ignore it?"

A logistic regression model can help answer this question. Each invitation sent from a client to a freelancer was treated as an example event, labeled as a positive class member if it was accepted and negative otherwise. (There were roughly one million such examples within a relevant and recent time frame.) The goal would be to link the freelancer's recent site activity – within the month prior to receiving the invitation – to the outcome of an acceptance or rejection of the invite.

The raw materials, and derived features, came in four main varieties:

Job application/invitation history In the previous day, how many job applications did Freelancer X create? How many invitations did X receive? How many of each were cancelled by the client, or by the freelancer? How many of each led to filled jobs? Answer these questions again for the time periods: two days, four days, seven days, 14 days, and 30 days.

Hours worked How many hours has Freelancer X billed to any client in the preceding day? two days? ... thirty days?

Server log history In the previous one/two/.../thirty days, how many times has Freelancer X visited pages under the "Job Search," "Message Center," and "View Contractor Profile" sections of Upwork.com?

User profile How many jobs has the freelancer worked in each of Upwork's job categories (e.g., Web Development, Graphic Design, Data Entry)? What is their stated preference for receiving more invitations at this time ("I am available," vs. "I'm not currently available")?

These raw materials (along with a few other sources) could be harvested from three services: a Greenplum relational database for assembling job application and hours-worked data, and an Amazon S3 bucket for the server logs. The user-profile information was an interesting case: the historical state of the profile was logged to a table in Greenplum with each invitation, but a freelancer's present profile state required a call to an internal service's REST API.

Assembling these feature vectors and performing a training-validation-testing split led to a model with a test-set AUC metric of around 0.81. This sufficiently outperformed the existing heuristic's accuracy, and the model was deemed ready for production use. Freelancers would start receiving scores from the model, putting them into buckets of low, medium, and high availability (i.e., high propensity to accept an invitation to interview at this time).

2.2 Throughput constraints in production

An interesting aspect of this modeling task is that hour to hour, any individual freelancer's availability score might swing considerably. Someone who looked dormant three hours ago could decide to log back onto Upwork and start engaging in eager-for-work behavior, sending out job applications and polishing their profile. Another freelancer's behavior might cause a sudden transition from looks-available to looks-overbooked. The more frequently each freelancer could be scored, the more faith Upwork could have that the bucketing used by the search engine was using reliable information. Keeping freshness of the raw material data under four hours was considered a reasonable goal.

Generating those scores meant providing the routine with those four families of raw material. When developing the model using historical data, all four could be gathered and processed in bulk, in advance of the featurization and scoring routines. In a production setting, this remained the case for data from S3 and Greenplum: all that relevant information, for all registered freelancers, could be collected in under an hour.

However, user profiles posed a challenge: given other demands placed on it by other Upwork.com systems, the internal user profile service could only afford to return one user profile every 20 milliseconds to the availability scorer. Any more rapid than that threatened the health of the service. That placed a maximum of 4.3 million scores to be produced per day, one for each up-to-date profile request. With six million total registered freelancers, this put the squeeze on the preliminary goal of every-freelancer, every-four-hours.

2.3 Concurrency under the Actor model

A direct re-use of the routines in the model generation software would involve:

1. bulk collection of the job application and worked-hours data,
2. bulk collection of the server log data,
3. bulk collection of as many user profiles as possible before the data from (1) and (2) could be considered stale,
4. featurization and scoring of each freelancer whose profile was collected in step (3).

Steps (1) and (2) combine to a total of about 40 hours of collection and processing time when performed sequentially. Keeping a buffer of 5 minutes aside for Step (4)'s vectorization and scoring (and saving those scores to disc), that means a 4 hour cycle will involve 195 minutes of harvesting user profiles in Step (3). That means an amortized rate of 146,250 scores produced per hour.

The fastest rate possible (one every 20 ms) would mean 180,000 scores per hour. That leaves room for a potential 23% improvement in throughput by moving to a system where user profiles are harvested concurrently alongside the other routines.

2.3.1 The Actor Model

The Actor model makes it easy to architect a concurrent version of the availability system. An Actor framework involves three components:

The Actors A collection of objects, each holding a message queue, some private state, and some private methods. This state is can only be updated, and these methods can only be called, by the actor object itself, and in response to receiving a message from some other actor. The response an actor takes to each message is fully completed before it begins its response to the next message.

The Messages Simple, immutable objects that contain information or instructions one actor wants another actor to notice.

The execution framework The harness which ensures that the computation each actor calls for is completed in order, each dispatched message reaches the right actor queue, and that the overall workload is shared as evenly as possible over the available resources.