Machine Learning to the rescue

Jan Macháček

June 1, 2018

Abstract

Machine learning to the rescue!... the first question is "to the rescue of what?"; immediately followed by "when is it indeed rescued?". The answers to these questions are crucial; luckily, the software engineering process is quite used to asking and answering these questions. Careful project analysis and inception, followed by continous integration and continous deployment in development (supported by adequate tests); all overseen by systematic project governance leads to successful software projects. This paper's proposition is that machine learning projects that are to apply established machine learning approaches and algorithms are no different than any other software project; and they must follow all practices of software engineering.

1 Noted

A lot of ML papers and talks ignore the landscape into which a ML component fits. (And for a good reason!–the papers and talks are on machine learning, not system architecture.) The usual steps are

Henrik[1] Define the proble Collect and understand the data Build and deploy simple model that works end-to-end Iterate to optimize and deploy improved models Monitor and back-propagate changes to problem parameters

Business analytics

Data collection & preparation (System to store data from a front-end system in a suitable storage & format) (Normalize, de-duplicate) Data verification (Distribution, correlations, ...; e.g. statistical analysis) Data selection (Training and verification data sets) Model training (The iterative computation) Model evaluation (Check the computed model, and all other models) Model deployment (...)

2 Software engineering

...

The Joel test[5] is a product of one man's biased, ad-hoc, informal, ... view of what makes successful software projects. Annoyingly, high scores on the Joel test correlate with successfully delivered software projects. Spolsky acknowledges that it's possible for a small team of cowboys to deliver amazing software with score of 0, as much as it is possible for a team that scores 12 to be the software equivalent of the Titanic. Nevertheless, high scores on the test correlate with good practices and discipline, which usually leads to good software.

This paper's proposition is that "business" projects that use machine learning are no different than any other software project; and that all practices of software engineering have to be applied to the machine learning subsystems. Specifically, that the 12 points on the Joel test are just as applicable, but with additional 12 points.

1. Versioned, testable; continuously tested and sanity-checked analytics (BI)
2. BI setup so that any previously executed query can be answered under 10 minutes
3. Monitoring on the BI environment to identify queries that use normalised data
4. The output of the BI that humans process define what ML should solve
5. Ingestion system decoupled from the rest of the system
6. Test and verification data set storage with statistical data verification
7. Naïve "return constant" model
8. Versioned computed model storage with training and validation data set references, and the model evaluation
9. Automated or single-click model deployer and "debugger"

But where is the ML that builds the model? That's the code that the engineering teams need to build to replace item 6.

Once the first four steps are known, the engineering teams can implement the remaining steps of the pipeline. If the system that is to take advantage of ML is event-based, the event delivery mechanism provides the decoupling, resulting in architecture shown in Figure 1.

If the front-end system is not event-sourced, the ingestion must be decoupled using a read-only replica of the live data. Notice in Figure 2 the flow of the data: the data is pushed into the read-only replica in the first step to allow the front-end system to control the load on its data store; from the read-only replica, the data is pulled into the ML data store.

Regardless of the approach used (or even if a hybrid approach is deployed), the entire system has to be aware of any back-pressure.

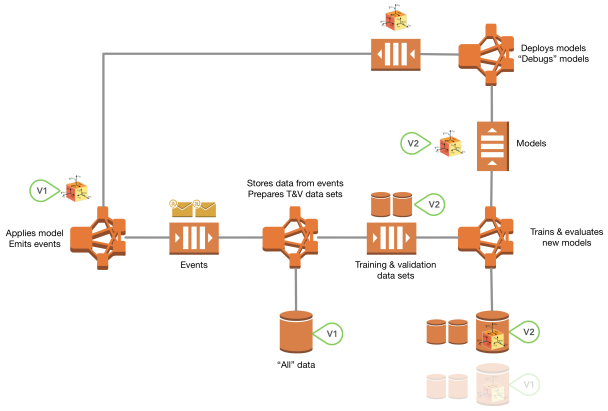

Figure 1: ML pipeline in event-based system

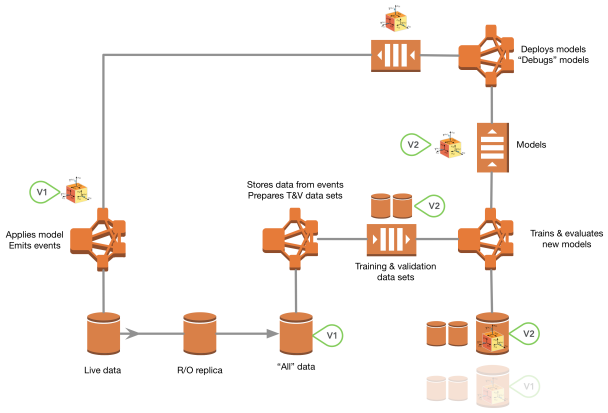

Figure 2: ML pipeline in non event-based system

The ML team maintains the tooling for the pipeline, consults on the best models, researches, ...; but the product teams (that ultimately work on the service that *uses* the model) have the first dibs on implementing the model. Successful implementation of this strategy means that anyone can implement a new model (even if only to just see what will happen!), train it, debug it, and deploy it all within a single day. All the mechanics of data ingestion, storage, versioning; runtime of training a model, evaluation, storage, versioning; debugging and deploying; and the usage is all implemented.

In this sense, the machine learning code is just like any other ordinary code; it is subject to all the high engineering standards and safeguards.

3 Implementation choices

Attach technologies to the blocks in Figure 1 and Figure 2. Show on practical example, attempt to pull out reusable blocks and show examples of good and bad code. Think the recent ML experiments; demonstrating just how much time the initial research and experiments take. Reinventing the experimental wheel must be avoided–the ML team should curate the bootstrapping environment, making it available "on demand" for other teams. The work on the ML core code can only successfully happen when all other pieces are in place.

3.1 Initial research

The practical application of the "ML enablement process" aimed to deliver failure predictions and automatic error recovery based in an event-based architecture system. The system already published seemingly too many events, but the detail in the events were very useful in building the data sets for the ML project. The first step the team needed to take was to build traditional business intelligence database–the goal was to find out whether our human knowledge and experience allowed us to find meaningful information in the data.

The system's journal (Apache Kafka[2]) is configured with 7 days' message retention policy; at the very start of the work, the team was able to download events for the last 7 days. This was approximately 60 GiB of uncompressed Protocol Buffers[4] binary messages. Because of the compactness of the binary wire representation of the Protocol Buffers messages, this translated to 60 GiB of a RDBMS (MySQL[3]) storage requirements. The loader program performed batch JDBC inserts into the MySQL tables. The events in the $HttpRequest$ and $HttpResponse$ topics were imported to the matching tables, the events in the $ServiceStateChanged$, $ServiceStateUnchanged$, $ServiceAdded$, and $ServiceRemoved$ were preprocessed and inserted into the $Service$ table. The tables are not even in the 1st normal form; there is only the primary key, but there are no foreign key constraints; during the batch import, there are no indexes.

The first seven days' worth of data import ended up as 5 103 397 rows in $HttpRequest$, 5 103 397 rows in $HttpResponse$ [1], and 54 566 575 rows in $Service$. This imported database was the starting point for the human analysis code. "How many $statusCode \neq 200$ in $HttpResponse$ are there?"–(72 796); "what is the average response duration?"–(4.0759 ms); "what is the average response duration for successful responses"–(4.9862 ms), and so on. In order to answer these questions, the database needed a few indexes: remember, "any ad-hoc analytics query needs to be answerable within 10 minutes." The $db.m4.4xlarge$ instance was completely sufficient for the first 7 days' worth of data; even for the first 70 days' worth of data. It delivered just over 150 MiB s^{-1} in read and write throughput, and with only a few users running queries, its CPU usage never got above 50 %; it was allocated 5 TiB of SSD storage.

[1] The same number of rows in the $HttpRequest$ and $HttpResponse$ is a very good sanity-check!

HttpRequest	
correlationId	varchar(36)
time	timestamp
method	varchar(10)
uri	varchar(255)
headers	text
entityContentType	varchar(64)
entity	blob

HttpResponse	
correlationId	varchar(36)
duration	integer
statusCode	integer
headers	text
entityContentType	varchar(64)
entity	blob

Status	
correlationId	varchar(36)
hostId	varchar(50)
serviceId	varchar(50)
time	timestamp
state1	integer
state2	integer
lastState1	integer
lastState2	integer
entity	blob
lastEntity	blob

Figure 3: Database schema

The first valuable answer came from the analysis of the frequency of different errors by joining *request* and *response* on *correlationId* where *statusCode* ≠ *200*, grouping and counting by *response.entity*. The top 3 errors turned out to be fairly simple to fix, and the fixes resolved 90 % of all observed errors. Beyond the top 3, the errors had no trivial cause and no trivial fix; the team suspected that the errors were cause by a combination of parameters passed in the requests, the previous state of the device doing the work, the network conditions, ...; something that could not be trivially reproduced nor fixed. Interestingly, the 3 least frequent errors did turn out to have trivial fixes: in the sense that there was no fix, but better failure reporting in low or missing resource conditions, misconfiguration, etc.

These were great first results. The next valuable feature was to predict a failure before it actually happened, which would reduce the pressure & stress in the operations team.

Building time series of requests, responses, and status changes. The tool was simple SQL *create table ... from select ... inner join* ; the sequence of events ended up as 50 182 398 rows, describing 545 357 sequences[2]. The data had sequences such as $(req_1, res_1) \rightarrow \mathcal{I}dle, (req_2, res_2) \rightarrow \mathcal{B}usy, \ldots, (req_n, res_n) \rightarrow \mathcal{E}rror$, or even $(req_1, res_1) \rightarrow \mathcal{I}dle, (req_2, res_2) \rightarrow \mathcal{B}usy$ (without further requests) $\mathcal{E}rror$. This work implements the first part of the overall diagram (viz Figure 4), and was able to satisfy (TODO) X, Y, and Z items on the ML readiness test.

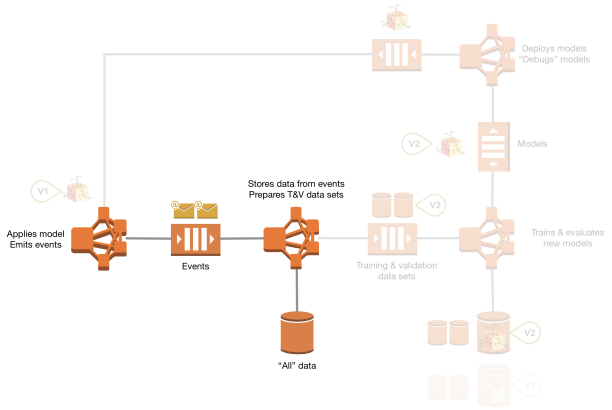

Figure 4: Data collection and BI

Nevertheless, this initial analysis was sufficient to build data sets for failure prediction. the team deployed code

[2]Sanity-check: there are periodic health-checks for every service contributing nearly 80 % of the HTTP requests and responses, and a typical sequence is made up of a single request and response, and multiple (usually > 10) service changed rows; there cannot be more sequences than requests and responses. The 545 357 number of sequences is therefore a sane number of sequences, given the sizes of the tables

more sophisticated than *select from ... group by ... having ...*.

3.2 Buy vs. build (AWS services vs. custom code)

3.3 Testing

Detect

- unstable data (outliers, bad labels, etc.)
- underfitting (too little data)
- overfitting (too little data; model is a memoized map of input -> output)
- unpredictable future (with new data, is the model still stable?)

Seam testing: treat ML models as "legacy black boxes", but evaluate. Use cross-validation (split training and evaluation data sets). Measure and record:

- precision–$TP/allpredictions$
- recall–$TP/TP + FN$
- time taken to train
- model size

Use time-boxed computation to re-evaluate the performance of all previous models against the current validation data set.

Treat the output of the ML box as if it were a scientific experiment: what is the probability that the (good) results happened through chance rather than good performance of the classifier. Statistical tests on the output and the input data.

4 Compromises

How to manage the growing data sizes and time to train the models; particularly with complex models that need a lot of computation to train? Where to keep the old data and old models? Is there ever value in digging out models and data that are 10 versions old?

5 Stability & Maintainability

Detect

- Feedback loops, especially hidden feedback loops
- Legacy data / features (something that's no longer needed, but still included in the training)
- Bundled data / features (deadline is looming, let's just throw in this blob to see what happens... hey!–it worked, we're keeping *all of it*.)
- ε-features: one more thing will increase the accuracy by 0.1 %...
- Data / feature provenance (where did it come from, what version is it?)
- Glue code / hacks: in a mature ML system very little code *does machine learning*. It is better to implement the ML code in a language / framework that is the best match for the system that uses it rather than gluing together other libraries (or even other languages). "It is better to re-implement X in Scala than to use the Java wrapper around the JNI bridge to **R** that implements X already.
- Pipeline spaghetti, particularly when it comes to preparing / assembling data for ML. Usually result of experiments or separation of the ML and engineering teams. The ML team does the minimum engineering work (think *Perl*) to allow it to focus on the ML model; and the engineering team then has to make the write-only code run. Imagine temporary files in people's hard-coded home directories or */**var**/bamtech−ml/exp1* and similar.
- Commented-out / dead experimental code. ML / engineering code that is no longer needed, but may be left behind a forever ***false*** branch, buried in a Docker container. (At some point, someone is going to remove that environmental variable that blocks the dead code, and tears will flow.)

Test

- Remove [detected under-utilised] features and re-evaluate the models
- Statistical test of predictions (sanity checks for biases and "reasonable" predictions)
- Action limits (particularly in RT systems with feedback loops to prevent ML from "turning on X" as the outcome of one prediction, immediately followed by "turning off X" as a result of the prediction that follows.)

References

[1] Henrik Brink. *One does not simply put Machine Learning into Production*. `https://www.youtube.com/watch?v=JKxIiSfWtjI&t=847s`.

[2] Apache Kafka. *Apache Kafka*. `https://kafka.apache.org`.

[3] *MySQL*. `https://www.mysql.com/`.

[4] *Protocol Buffers*. `https://developers.google.com/protocol-buffers/`.

[5] Joel Spolsky. *The Joel Test: 12 Steps to Better Code*. `https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better-code/`.