

Machine Learning to the rescue

Jan Macháček

May 29, 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 continuous integration and continuous 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 Software engineering

...

The Joel test[4] 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. Version control for the ML models and data sets used to train them

Joel: Do you use source control?

2. Single-step / automated data selection, model training, evaluation, and deployment

Joel: Can you make a build in one step?

At least daily training and deployment process

Joel: Do you make daily builds?

3. The results of the BI queries that humans process define what ML should solve

Joel: Do you have a bug database?

4. Versioned, testable; continuously tested and sanity-checked BI

Joel: Do you fix bugs before writing new code?

5. BI that allows any query to be answered in under 10 minutes

Joel: Do you have a spec? Do you have an up-to-date schedule? Do programmers have quiet working conditions? Do you use the best tools money can buy? Do you have testers? Do new candidates write code during their interview? Do you do hallway usability testing?

6. Versioned, testable; continuously tested and sanity-checked analytics (BI)
7. Monitoring on the BI environment to identify queries that use normalised data
8. The results of the BI queries that humans process define what ML should solve
9. Ingestion components decoupled from the rest of the system
10. Versioned, testable; continuously tested and sanity-checked data sets
11. Pre-computed “return constant” model
12. Versioned, testable; continuously tested and sanity-checked model storage with training and validation data set references
13. Model deployer and “debugger”

1. Versioned, testable; continuously tested and sanity-checked analytics (BI)
2. Any BI query can be answered under 10 minutes
3. Monitoring on the BI environment to identify queries that use normalised data
4. The results of the BI queries that humans process define what ML should solve
5. Ingestion components decoupled from the rest of the system

6. Versioned, testable; continuously tested and sanity-checked data sets
7. Pre-computed “return constant” model
8. Versioned, testable; continuously tested and sanity-checked model storage with training and validation data set references
9. 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.

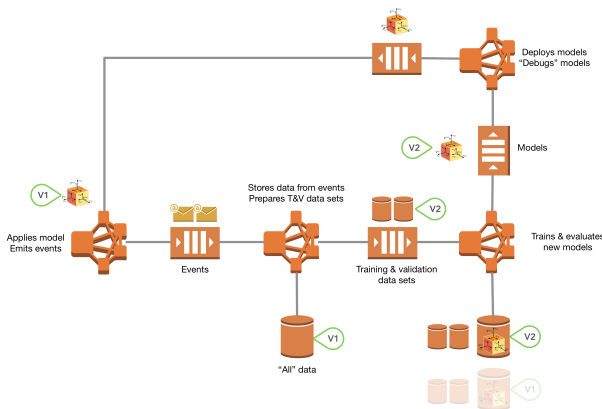


Figure 1: ML pipeline in event-based system

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.

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.

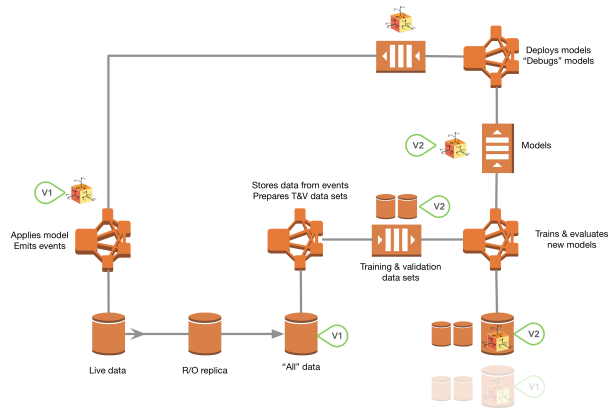


Figure 2: ML pipeline in non event-based system

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.

2 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 RAZCP 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.

2.1 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[1]) 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[3] 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[2]) storage requirements. The loader program performed batch JDBC inserts into the MySQL ta-

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

bles. The events in the *HttpRequest* and *HttpResponse* topics were imported to the matching tables, the events in the *ServiceStateChanged*, *ServiceStateUnchanged*, *ServiceAdded*, and *ServiceRemoved* were pre-processed 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 MiBs⁻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. Remember

These were great first exploratory questions, the next round focused on 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. (Sanity-check: a sequence is made up of multiple requests and responses, and multiple service changed rows; there cannot be more sequences than requests and responses).

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

2.3 Testing

3 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?

References

- [1] Apache Kafka. *Apache Kafka*. <https://kafka.apache.org>.
- [2] MySQL. <https://www.mysql.com/>.
- [3] Protocol Buffers. <https://developers.google.com/protocol-buffers/>.
- [4] 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/>.

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