Versioned machine learning pipelines for batch experimentation

Tom van der Weide
Schibsted Media Group
tom.vanderweide@schibsted.com

Oleg Smirnov
Schibsted Media Group
oleg.smirnov@schibsted.com

Michal Zielinski
Schibsted Media Group
michal.zielinski@schibsted.com

Dimitris Papadopoulos
Schibsted Media Group
dimitris.papadopoulos@schibsted.com

Tim van Kasteren
Schibsted Media Group
tim.vankasteren@schibsted.com

Abstract

Real-world Machine Learning pipelines that run in shared environments are challenging to implement. Production pipelines typically consist of multiple interdependent processing stages. Between stages, the intermediate results are persisted to reduce redundant computation and to improve robustness. Reusing persisted datasets improves efficiency but at the same time creates complicated dependencies. Every time one of the processing stages is changed, either due to code change or due to parameters change, it becomes difficult to find which datasets can be reused and which should be recomputed.

In this paper we propose how to produce derivations of datasets to ensure that multiple versions of a stage can run in parallel while minimizing the amount of redundant computations. Furthermore, we show how to use such versioning to reduce the time spent in experimentation on setting up the data and keeping track of results.

1 Introduction

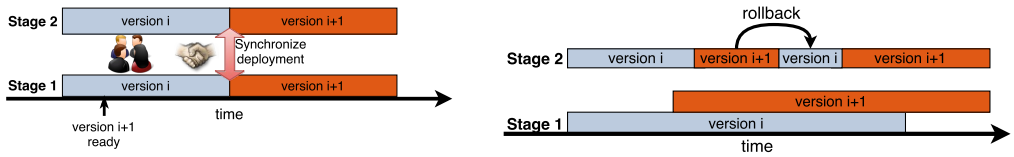
At Schisted Media Group we predict demographic traits of our users such as age, gender, and home location, based on their click behavior. Other teams use these predictions for targeted ads and for audience insights. Our models are trained daily and predictions are made on an hourly basis.

As the product evolves, changes are made to the output data, to the input of our pipeline, as well as to the internal intermediate datasets. To mitigate the consequences of these unstable data dependencies [11], we tie each release to an explicit version of the dataset[1].

Versioned ingestion presents difficulties, when consumers rely on the schema of a specific dataset. After communicating the changes, data consumers need to change their ingestion procedure. Finally, when both parties are ready, the new versions of the producer and the consumer code need to be deployed at the exactly same time. This process of synchronous deployments, as shown in Figure 1a, requires coordination and makes such changes expensive and error-prone. Changing version of

[1]For instance, the version could appear in the storage path.

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.



(a) Synchronous deployments require careful coordination between teams

(b) Asynchronous deployments do not require synchronization and support rolling back

Figure 1: Synchronous and asynchronous deployments

output dataset is analogous to changing an API, which is known to be tedious and disruptive to the development process [3].

If the data consumer detects a problem with the deployed version, they cannot simply rollback; they need to agree and synchronize rollback with the data producer. We propose a system, as depicted in Figure 1b, where it is possible to deploy and run multiple versions simultaneously, so that producers and consumers do not have to synchronize their releases. This would allow customers to rollback at any point. In Section 3 we show how dataset versioning enables asynchronous deployments.

The development process normally involves experimenting with using different features and various machine learning algorithms. We might want to compare Gradient Boosted Trees to Random Forests on one set of features and then compare Random Forests to itself on different features. In our pipeline, we persist featurized data prior to model training. While experimenting with new features, we need to manually do bookkeeping to record where what experiment is stored. In Section 4 we describe how we use versionization to distinguish between experimental outputs.

2 Related work

Data dependencies have been argued to carry technical debt similar to code dependencies in [11], with a mitigation strategy to version the incoming signal. The authors have mainly focused on external dependencies (imported features). We argue that with multi-stage pipelines internal dependencies are a major source of incompatibilities.

The Spark ML Pipeline API [7] allows to compose pipelines consisting of transformers and estimators. A *transformer* transforms data, e.g. it could add a column that contains the logarithm of another column. An *estimator* fits a model to data and produces a parametric representation. For example, linear regression is an estimator. When fitted to a certain dataset, a model consisting of the coefficients and bias is learned. This model can then be applied as a transformer and can be used to create predictions on new data. Approaches like [2] can be used for model life cycle management, but are not designed for the evolution of datasets.

Spark's ML approach was inspired by the work in [8]. The authors of the Scikit-learn framework defined, implemented and popularized the notion of *estimator* and *transformer* classes with corresponding *fit* and *transform* methods.

3 Versioned Pipelines

For feature generation, training, and prediction we are using a ML pipelines API analogous to the ones described in [7] and [8]. We train predictive models using click data from the last 45 days, which is available in hourly partitions. Data pipeline consists of several preprocessing and cleanup stages, followed by aggregation on user level, and featurization step.

The data of the latest 45 days is used to train models and make predictions. Since we generate predictions hourly, we need to read 45 days of data every hour. Because the preprocessing of data is deterministic and we have to read its result every hour for the coming 45 days, persisting the preprocessed data saves a lot of compute. Since every hour of data is being used $45 \times 24 = 1080$ times (45 days times 24 hours), persisting the intermediate result prevents us from preprocessing the same data for 1079 times. We apply the same trick as in [1] to prevent recalculation on both the aggregation and featurization step. See Figure 2 for what this looks like.

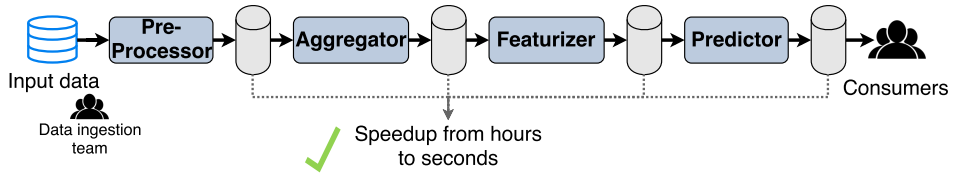


Figure 2: Machine learning pipeline that persists intermediate stages

With this approach we get a significant increase in performance, but we do have to be careful with how we use the intermediate persisted data. Namely, if the code in a pipeline changes, e.g. the aggregation changes, then we must ensure that the next pipeline reads the output of the latest version. This is especially important when we are running multiple deployments simultaneously.

For our customers it is important that we provide versioned predictions, but they should not be bothered by what version of input data we use or what versions the intermediate stages have. Consequently, we propose that pipelines declare both what data they use and what version of the data they require. Note that this is the same as what is done with code dependencies.

This section is organized as follows. We start in Section 3.1 with explaining a popular versioning schema called semantic versioning. Next, Section 3.2 explains how to managing different versions of the code. Section 3.3 then goes into detail about considerations for updating versions of code. Finally, Section 3.4 shows how we can now run multiple versions in parallel.

3.1 Semantic Versions

As pipeline stages are being further developed, their output schema or the content of the output might change. Typically, version numbers are used to denote the state of the software. Downstream consumers are developed with respect to a particular version. Consequently, if a pipeline stage changes, then its consumers may also need to change. For example, if a column was removed that a consumer was using, then the consumer needs to evolve.

Not all changes in a pipeline stage require that the downstream pipeline stages must evolve. To better communicate the severeness of a change, we use a versioning schema that has enough expressive power to distinguish between different types of changes. Semantic versioning [9] is a popular formalised version policy. In semantic versioning version numbers have the structure `MAJOR.MINOR.PATCH`, e.g. 3.1.7, where we increase `MAJOR` for incompatible API changes, `MINOR` for new backwards-compatible functionality, and `PATCH` for backwards-compatible bug fixes. Although semantic versioning allows a programmer to express the kind of changes that were made, careful attention has to be paid to comply to these rules [10].

For example, in a pipeline that generates features, we increase `MAJOR` when features are removed or altered, `MINOR` when new features are added (note that adding a feature maintains backwards compatibility), and `PATCH` when data quality bugs of the features are resolved (e.g. fix how some feature was calculated).

3.2 Code Versions

This section introduces some notation around pipelines in order to more concisely express our ideas. First we define what a pipeline does.

Definition 1 (Pipeline Function) *A* pipeline function *is a function $f : D \times C \to D$, where D is the set of all possible datasets, and C the set of all possible configurations, i.e. a function that takes a dataset and configuration as parameter and returns a dataset.*

Pipelines can be configured from outside the code. For example, a pipeline that contains a machine learning algorithm may be configured with hyper-parameters, or a featurization pipeline with how many buckets to use with the hashing trick or whether words have to be stemmed.

The implementation in code of a pipeline defines its corresponding pipeline function. Consequently, two different implementations correspond to two different pipeline functions. One approach to determine changes in code is to use the hash of the source code files. However, this means that

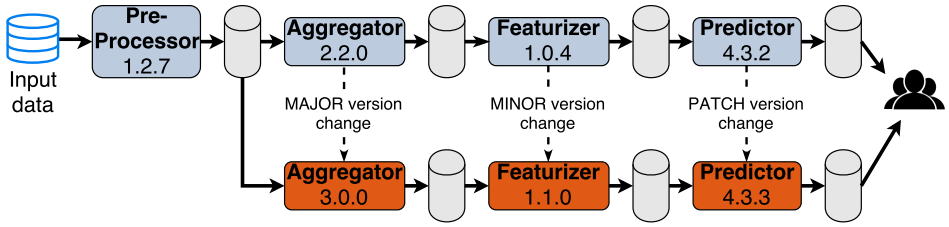


Figure 3: Running multiple pipeline versions

the smallest code change results in a different version, which prevents us from reusing previously constructed datasets. This is especially cumbersome when refactoring code. To mitigate this we introduce the notion of *pipeline equivalence*.

Definition 2 (Pipeline Equivalence) *We say that two pipeline functions f and g are* equivalent*, i.e. $equiv(f, g)$, if and only if $f(d, c) = g(d, c)$ for all $d \in D$ and $c \in C$.*

Consider that a new commit c is added to a branch with pipeline function f resulting in a pipeline function g. If commit c refactored the code, added some comments, or changed something not used by pipeline function f, then f and g will always produce exactly the same output. Although the bytecode of the two functions may be different, we consider them to be equivalent.

3.3 Updating Versions

As code evolves, so will the versions of the pipelines be increased. This section addresses the question of how version changes should be incorporated in pipelines that depend on them. Updating the version of a pipeline should depend on whether the output of that pipeline has changed. In other words, if pipeline functions f and g are equivalent, as per Definition 2, then the version of f and g should be the same. We use the notion of *versioning* to capture this.

Definition 3 (Versioning) *A* versioning *is a function $v : F \rightarrow V$ s.t. $v(f) = v(g) \Rightarrow equiv(f, g)$, where F is the set of all pipeline functions and V the set of all versions.*

Consequently, two pipeline functions with the same version must be equivalent, i.e. produce the same output. Furthermore, pipeline functions that are not equivalent cannot have the same version.

Note that it is not necessary that equivalent pipelines have the same version. Namely, if they would not be the same, then this would mean that they would save their output in different locations. Although less efficient, we will not run into any problems. If two not equivalent pipeline functions would save their output in the same location, then problems would arise.

3.4 Running Multiple Versions

One of the goals addressed in this paper is to allow for deploying multiple versions simultaneously. In Section 3.3 we prescribe that when two pipelines are not equivalent, then their versions must be different. Since the version is used in the path where the output is stored, we can ensure that they do not conflict with one another.

Consider the example depicted in Figure 3. Here several pipeline stages have multiple versions running in parallel. The data *pre-processor* did not change and consequently only needs to be deployed once. The *aggregator*, however, underwent a major change, which is reflected in its version. Since the path of its output contains the version, the two running versions will not conflict. Because of the major change in the *aggregator*, the *featurizer* also needed a change, which was backwards-compatible and therefore only requires a minor version change. In this example, this only resulted in a patch version change in the *predictor*.

4 Derivable Datasets

While running experiments with different features or machine learning algorithms, it is time-consuming to keep track of what code and parameters were used in each experiment. Moreover, you

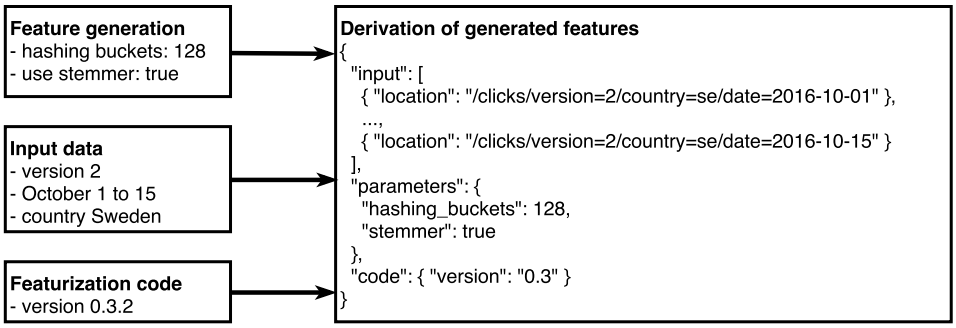


Figure 4: A derivation specifies all parameters needed to reconstruct it

have to be careful that one experiment does not overwrite the output of another experiment. This section describes how we create *derivations* of datasets to make this process more user friendly.

Our idea to create derivations is partially inspired by NixOS [4], a purely functional Linux distribution, where a build is a pure function whose outcome only depends on its parameters. In other words, a build can be derived from its parameters. This idea has also been applied to data. Namely, the provenance of datasets has been defined in terms of the source and transformations that have been applied [6, 5]. In this section, we will apply these ideas to datasets used in machine learning pipelines.

First, we define what derivations of datasets should consist of.

Definition 4 (Dataset Derivation) *A* dataset derivation *is a tuple* $\langle f, d, c \rangle$*, where* f *is a pipeline function,* $d \in D$*, and* $c \in C$*. A derivation uniquely defines a certain output dataset.*

To address having a different derivation for even the smallest change, derivations should not distinguish between equivalent classes of pipeline functions. This means that if f and g are equivalent, then any derivations with the same input data and configuration should be the same, i.e. $\langle f, d, c \rangle = \langle g, d, c \rangle$.

4.1 Persisting Output

With no automatic system in place, versions are updated manually when code changes to ensure that different pipeline functions are not written to the same location. Moreover, to improve predictive models, many experiments are run that have different configurations or changes in the pipeline code [12]. This means that the experimenter needs to update the version for each experiment and needs to manually keep track of all this. This is an error-prone process that should be automated. We propose using derivations of datasets to automatically change the version of a dataset.

The output of a pipeline is written to disk and we need to make sure that the output of two different derivations is not written to the same location. This can be achieved by encoding the derivation into the location, for example, by applying a hash function to the derivation and to use the hash to uniquely identify the derivation. Schematically, we can think of applying a hash function to all relevant parameters as shown in Figure 4.

To compare experiments you need to know what parameters were used. Furthermore, in a production system derivations are crucial to debug problems. Storing derivations in a database enables debugging and better analysis. For example, users could search for particular types of datasets as well as to do more in depth analysis of how a machine learning system is used.

While experimenting with, for example, different parameters of the featurization pipeline, you do need to specify explicitly in downstream pipelines on what data they depend. Consider Figure 5. Here there have been two runs of the Featurizer with the same code version, but with different configuration. Since the experimentation mode is on, their outputs are written to different locations in the storage. Next, we would like to see how the different configurations affect the prediction performance. Consequently, the predictor must specify what derivation to use.

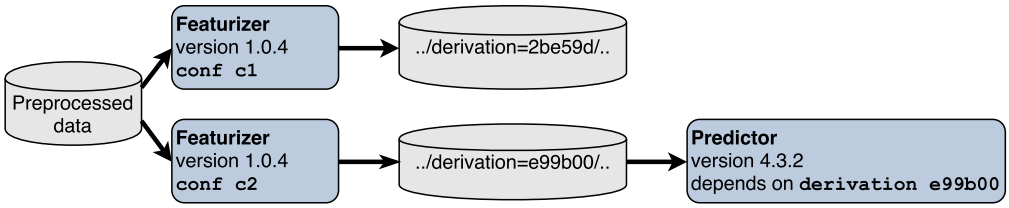


Figure 5: The same pipeline stage with different configuration results in different datasets

4.2 Implementation in Spark

Spark ML pipeline stages implement the *Params* trait, which provides functionality to set and retrieve parameters. For example, an estimator like a decision tree has parameters for the maximum depth and bins. Spark uses this mechanism to load and store pipelines.

To create the derivation of a dataset produced by a pipeline, we collect (i) all parameters using the *Params* trait, (ii) all paths of all input datasets, and (iii) the code version of the pipeline. From all this, a JSON object is created, of which we use the hash in the path of the output. Finally, we persist the JSON object such that the derivation can be inspected.

Suppose we have a daily job that reads page views and then makes predictions. For a specific day, it reads data from */clicks/date=2016-11-11/* and writes the predictions to */predictions/derivation=bb3efca/date=2016-11-11/*. Since the input path of this daily prediction pipeline changes daily, there will be a different derivation hash every day. However, for navigation it is handy if the derivations of both day predictions were the same given that the parameters did not change. Since the output path includes the date, we could use path templates, e.g. */clicks/date=[date]*, as input paths in the derivation. Because such path templates are static over time, all daily predictions will be stored with the same derivation hash, until the parameters or code version changes. Consequently, for every daily job there will be a separate sub-folder in the derivation folder making it easy to see what predictions were created with these parameters and code version.

5 Conclusion

In this paper we have described our approach to address our problems of running multiple deployments simultaneously and having a lot of manual setup to run experiments. Because our pipeline is cut up into several smaller pipelines, which all persist their output for caching, we described how we version the datasets. This elaborate versioning enables us to run multiple pipelines in parallel.

To facilitate experimentation within our platform, we have introduced the concept of derivations of datasets. A derivation specifies all the elements necessary to recreate a dataset. Derivations are used to write different experiments in different locations and to automatically do the administration of what parameters have been used in an experiment.

Because the output path of a pipeline is changed automatically when its parameters are changed, the workflow for running experiments and for deploying multiple versions in parallel has been simplified greatly, by automating error-prone manual setup. Our approach improves upon using timestamps to distinguish between datasets since subsequent pipeline stages can refer to upstream dependencies using derivations rather than having to know when a particular dataset was created.

A weak point of this system is that the version of the code still has to be set manually. It has proven to be difficult to decide when to change the major, minor, or patch part of the version. Also, to ensure that derivations change as little as possible, we use templates for the input paths. However, this is more complicated than expected when a range of similar input paths are required, e.g. all page views partitioned by date in a certain date range.

References

[1] M. Aly, A. Hatch, V. Josifovski, and V. K. Narayanan. Web-scale user modeling for targeting. In *Proceedings of the 21st International Conference on World Wide Web*, pages 3–12. ACM, 2012.

[2] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. In *Conference on Innovative Data Systems Research (CIDR)*, 2015.

[3] D. Dig and R. Johnson. How do apis evolve? A story of refactoring. *Journal of software maintenance and evolution: Research and Practice*, 18(2):83–107, 2006.

[4] E. Dolstra, A. Löh, and N. Pierron. Nixos: A purely functional linux distribution. *Journal of Functional Programming*, 20(5-6):577–615, 2010.

[5] M. Greenwood, C. Goble, R. D. Stevens, J. Zhao, M. Addis, D. Marvin, L. Moreau, and T. Oinn. Provenance of e-science experiments-experience from bioinformatics. In *Proceedings of UK e-Science All Hands Meeting 2003*, pages 223–226, 2003.

[6] D. P. Lanter. Design of a lineage-based meta-data base for gis. *Cartography and Geographic Information Systems*, 18(4):255–261, 1991.

[7] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, and others. Mllib: Machine learning in apache spark.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[9] T. Preston-Werner. Semantic versioning 2.0.0. `http://semver.org/spec/v2.0.0.html`.

[10] S. Raemaekers, A. van Deursen, and J. Visser. Semantic versioning and impact of breaking changes in the maven repository. *Journal of Systems and Software*, 2016.

[11] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[12] M. Vartak, P. Ortiz, K. Siegel, H. Subramanyam, S. Madden, and M. Zaharia. Supporting fast iteration in model building. *LearningSys*, 2015.