Versioning for End-to-End Machine Learning Pipelines

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

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

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

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

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

ABSTRACT

End-to-end 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. Those results might come in the form of datasets for data processing pipelines or in the form of model coefficients in case of model training pipelines. Reusing persisted results 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 build upon previous work to produce derivations of datasets to ensure that multiple versions of a pipeline can run in parallel while minimizing the amount of redundant computations. Our extensions include partial derivations to simplify navigation and reuse, explicit support for schema changes of pipelines, and a central registry of running pipelines to coordinate upgrading pipelines between teams.

ACM Reference format:
Tom van der Weide, Dimitris Papadopoulos, Oleg Smirnov, Michal Zielinski, and Tim van Kasteren. 2017. Versioning for End-to-End Machine Learning Pipelines. In *Proceedings of DEEM'17, Chicago, IL, USA, May 14, 2017,* 9 pages.
DOI: http://dx.doi.org/10.1145/3076246.3076248

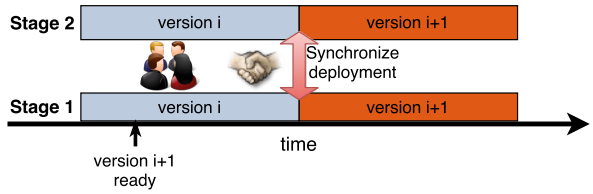
1 INTRODUCTION

At Schibsted 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.

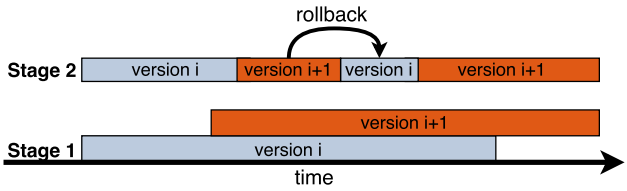
As the product evolves, changes are made to the output data, to the input of our pipeline, as well as to the internal intermediate datasets. To mitigate the consequences of these unstable data dependencies [16], we tie each release to an explicit version of the dataset.

Versioned ingestion presents difficulties, when consumers rely on the schema of a specific dataset. Within a single team maintaining multiple pipelines this is challenging, but across teams this becomes even more complicated. 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 output dataset is analogous to changing an API, which is known to be tedious and disruptive to the development process [5].

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. In [17], we proposed 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 allows customers to rollback at any point. Section 3 describes how this system of dataset versioning enables asynchronous deployments.



(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

DEEM'17, Chicago, IL, USA

978-1-4503-5026-6/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3076246.3076248

While developing machine learning pipelines, a significant amount of time is spent on experimenting with using different features and machine learning algorithms. For example, 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. To automate the bookkeeping of where the results of experiments are stored, [17] proposed using dataset derivations, which are different for each experiment. Section 4 describes how versioning was used to distinguish between experimental outputs. One shortcoming of [17] was that every experiment resulted in a completely different path, which made navigating practically impossible. To address this, Section 4.1 introduces *partial dataset derivations*.

Section 5 extends our previous work with lessons learned and solutions for different ways to store datasets. More specifically, it proposes to explicitly add the data schema version to the path where the data is stored, using different storage schemes for production and experimentation use cases, and using a database to store derivations for easy navigation and analysis. Finally, Section 6 introduces a central registry which keeps track of all the pipelines running in production, and what dataset versions they produce and consume, in order to facilitate version upgrades and monitoring.

2 RELATED WORK

Data dependencies have been argued to carry technical debt similar to code dependencies in [16], with a mitigation strategy to version the incoming signal. The authors have mainly focused on external dependencies (imported features). In addition, we argue that with multi-stage pipelines also internal dependencies are a major source of incompatibilities.

The Spark ML Pipeline API [12] 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. Spark's ML approach was inspired by the work in Scikit-learn [13].

Approaches like [4] and [18, 19] can be used for model life cycle management, but are not designed for the evolution of datasets.

The challenges for managing versioning or branching of relational datasets have been addressed in [11] and [2]. These relational storage systems provide built-in version control designed to facilitate sharing and collaborating on datasets (e.g. across a team of data scientists). They assume a data model whereby each dataset must have a well-defined primary key ; the primary key is used to track records across different versions or branches, and thus is expected to be immutable (a change to the primary key attribute, in effect, creates a new record). This assumption, however, does not necessarily hold for the datasets produced by our pipelines. It is also unclear whether these systems fit in our use-cases, whereby the datasets are continously produced (e.g. hourly, or daily) by processes with various parameters which need to be tracked, as well.

The notion of dataset derivation presented in Section 4 is similar to the lineage API features in [10]. Therein lineage records offer fault-tolerance and job output recovery in case of failures. In our case, derivations are used to uniquely identify datasets, while partial (templated) derivations facilitate experimentation.

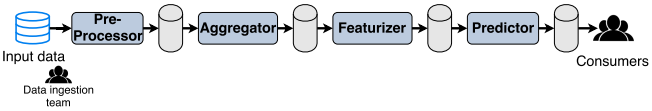


Figure 2: Machine learning pipeline that persists intermediate stages

The registry for running pipelines, which is outlined in Section 6, allows the implementation of ideas that are closely related to the concept of provenance [3], i.e. the ability to understand how each dataset was created, from which computation and what data it originates from. Additionally, by including information about job schedules, it offers insights about possible future interactions between producers and consumers.

3 VERSIONED PIPELINES

For feature generation, training, and prediction we are using an ML pipelines API analogous to the ones described in [12] and [13]. We train predictive models using click data from the last 45 days, which is available in hourly partitions. Data pipelines consist of several preprocessing and cleanup stages, followed by aggregation on user level, and a 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.

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 scheme that has enough expressive power to distinguish between different types of changes. Semantic versioning [14] 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 [15].

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 3.1 (Pipeline Function). A *pipeline function* is a function $f : D \times C \rightarrow D$, 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 [20] 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 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 3.2 (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.

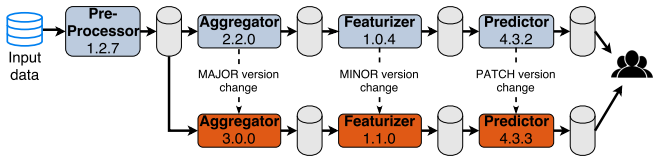


Figure 3: Running multiple pipeline versions

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 3.2, then the version of f and g should be the same. We use the notion of *versioning* to capture this.

Definition 3.3 (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 *preprocessor* 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 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 [6], 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 [3, 7, 9]. 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 4.1 (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. $\langle f, d, c \rangle = \langle g, d, c \rangle$.

See Listing 3 in Section 4.2 for how this definition can be implemented in code.

4.1 Partial dataset derivations

The decision of where and how to store datasets is crucial for multi-stage pipelines with reusable parts. In our previous work [17] we described taking a hash of the dataset derivation and to encode that explicitly in the path. With this approach, every invocation will be stored in a different derivation folder. For navigational purposes, this is rather inconvenient because there is an overhead of looking up whether the pipeline function and configuration for two derivations changed. In Section 5.2 we describe in detail different approaches to determine where to store datasets.

For consumers it may be tricky to determine where they should read data from. Since the path changes with different configuration or input data, the consumer should have intricate knowledge about the producer in order to read its output.

To address these shortcomings, we introduce the notion of a *partial dataset derivation*. Rather than including the exact input dataset specification, it includes a template of what data should be loaded. Given a set of parameters, this template can be expanded into actual dataset specifications.

Definition 4.2 (Partial Dataset Derivation). A *partial dataset derivation* is a tuple $\langle f, c, t \rangle$ where $t \in T$ is a template for which we have a function $\pi : T \times 2^P \rightarrow D$ that maps it to a dataset given a set of parameters.

Listing 4 shows how this definition can be implemented in code. An example of t would contain the paths of all clicks in the last 31 days relative to a given date, that is given as a parameter. The corresponding π would take a date parameter, and fill it in t. Figure 4 depicts this partial dataset derivation. Here t is represented as a set of parameters and inputData, which is a number of path templates that need the parameters in order to be expanded.

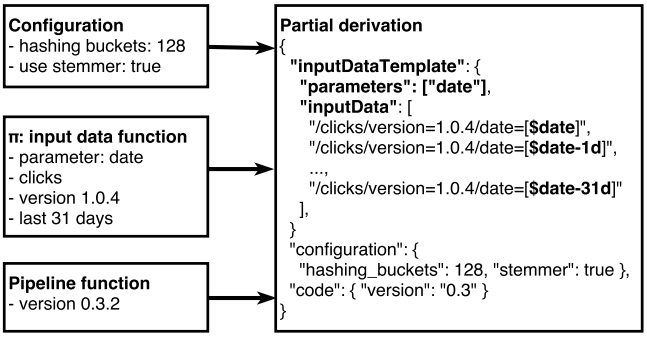


Figure 4: A partial dataset derivation

4.2 Implementation

Machine learning frameworks such as scikit learn [13] and Spark ML [12], use the notions of *pipelines* and *pipeline models*. A pipeline is an *Estimator*, which transforms input data and fits a model to that input data. For example, a pipeline could be a machine learning algorithm that trains a model. The output of *fit* is a pipeline model. A model can be applied to data, by invoking the *transform* method. This will transform the data as specified in the pipeline, and then make predictions for the given input data. The interfaces are as follows.

Listing 1: Pipeline and pipeline model interfaces

```
trait PipelineModel {
  def transform(dataset: InputData): Dataset
}

trait Pipeline {
  def fit(dataset: InputData): PipelineModel
}
```

In addition to Spark ML, we introduce the notion of a *pipeline job*, which wraps a pipeline in an executable program. A pipeline job has a run method, and also a description of what input data it requires. To run, a pipeline job needs job specific parameters, e.g. the date of the input data, or the environment such as staging or production, pipeline specific configuration such as hyper-parameters for the machine learning algorithm, and a storage scheme. Note that the input data is described using a template, which can be expanded with the job parameters into a complete path.

Listing 2: Executable job to run a pipeline

```
trait PipelineJob {
  def inputDataTemplate: InputDataTemplate
  def run(jobParams: JobParams, config: PipelineConfig,
          storageScheme: StorageScheme): Unit
}
```

Typically, pipelines have a configuration with parameters that they use. For example, a pipeline that trains a decision tree might require configuration for the number of trees. In this section, we assume that we have an interface *PipelineConfig* that captures how pipelines can be configured in a particular framework. Note that in

Spark ML, pipelines implement the *Params* trait, which allows you to retrieve all parameters in the pipeline.

Pipelines take one or multiple datasets as input and fit a model. The model can then be used to transform data. For example, an ad click prediction pipeline could take as input several click datasets aggregated by day, and also a dataset with user data. Before fitting a model, the pipeline would union the click data, and join with the user data. To abstract away from the specifics of how to implement this, we assume a trait *InputData*.

Given these classes, we can implement dataset derivations as in Definition 4.1.

Listing 3: Derivation classes as in Definition 4.1

```
case class Derivation(pipelineVersion: Version,
    inputData: InputData, config: PipelineConfig)
```

Similarly, we can now implement a partial dataset derivations as in Definition 4.2. For this we need the class of *InputDataTemplate*, which is shown in Figure 4.

Listing 4: Partial derivation classes as in Definition 4.2

```
case class PartialDerivation(pipelineVersion: Version,
    config: PipelineConfig,
    inputDataTemplate: InputDataTemplate) {

  def apply(jobParams: JobParams): Derivation = {
    val inputData = inputDataTemplate.apply(jobParams)
    Derivation(pipelineVersion, inputData, config)
  }
}
```

Note that a partial dataset derivation can be transformed into a complete derivation given the job parameters. Using the job parameters, the input data template is expanded into a concrete *InputData* instance.

5 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. 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.

5.1 Schema versions

When you need to read a big aggregation of historical data, or when you need to compare data at different points in time, then it may happen that parts of the data are produced with one version, and other parts with another version.

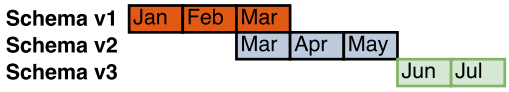


Figure 5: Different schemas can be used at different times

Consider Figure 5, where the data is stored with schema of version v_1 for January, February, and March, but then in March, two pipelines were running in parallel and data was also stored according to schema v_2. In June, a hard switch was made to version v_3. When it is required in an experiment to read data from January until June, the code needs to be able to read data of 3 different schemas.

To read data that was written with different schemas, one has to pay extra attention to ensure correctly handling each version. For example, one field may exist in one version but not in another. Moreover, for downstream consumers it is important to be made aware of schema changes explicitly. Furthermore, the code has to be careful about reading one or both versions of the data in March, since it is possible that the results will be incorrect because of reading the same data twice.

In [17], the schema of a pipeline's output is not represented explicitly in the derivation. Note that it is theoretically not necessary to do so, because with a reference to the pipeline function (and therefore its code) and the input data, it is possible to infer what the output schema will be. Even though with semantic versioning you can communicate breaking changes, our data consumers have requested to make schema changes more explicit. Moreover, a semantic version does not describe how the schema was changed.

To address these use cases, we propose to keep track of the schema versions and to put them explicitly in the path where the data is saved. If the schema version is explicit in the path, then it not only becomes simpler for developers to determine what schema adapters they need to write, it also becomes unmistakably clear when a schema changes. Moreover, we also suggest to publish the actual schema in a file in the path.

5.2 Storage schemes

Given a running pipeline that produces dataset and their corresponding derivations, we need to define where each dataset will be stored. To this end, we introduce the notion of a *storage scheme*, which is a function that takes a dataset derivation and returns a path.

Definition 5.1. A *storage scheme* is a function S that maps a dataset derivation to a file system path s.t. $S(d) = S(d') \Leftrightarrow d = d'$.

What we do not want is that two different datasets, i.e. with a different derivation, are written to the same path. Therefore, the only restriction on storage schemes is that, if the paths of two datasets d and d' are the same, then d and d' must be the same.

Pipelines in production are typically running for a longer period of time with the same code version and configuration. In contrast, when running experiments, the code version and configuration usually changes in every invocation of the pipeline. What storage scheme is most convenient depends on how the pipeline is used.

5.2.1 Production pipelines. For production pipelines, we are using the following scheme. Namely, given an input data template t and corresponding π, as per Definition 4.2, the storage scheme for one invocation of π, i.e. $\pi(t, p) = \langle f, d, c \rangle$, is $/h(f)/h(c)/p$ where h is a hashing function and p are the parameters such as date. In production pipelines, the pipeline function and the configuration are typically stable as long as no new versions are deployed. Using this scheme, it is easy to see how long a certain version has been running because it is easy to list all p for a certain f and c.

Some of our data consumers asked for an explicit schema version. Therefore, we included the schema version as described in Section 5.1 into the storage scheme: $/h(schema(f))/h(f)/h(c)/p$.

When the configuration is static for a longer period of time, it does not provide any useful information. In such cases, one might choose not to include it in the path, e.g. $/h(schema(f))/h(f)/p$. Another option is to consider the configuration as part of the pipeline function, i.e. $/h(schema(f))/h(f, c)/p$.

5.2.2 Experimental pipelines. For experimental pipelines these production storage schemes are not ideal because it is cumbersome to list all experiments that have been run with a certain set of parameters. To easily see all experiments, the scheme $/p/f/h(c)$ might be more useful. For example, suppose the parameters consist of only the date and an experiment is run with two different versions of the code, e.g. using other features, and with two different configurations (hyper parameters). In that case, the following folders may be the output:

```
/2017-02-01/code=3.14.4/config=47eda1
/2017-02-01/code=3.14.4/config=aaf371
/2017-02-01/code=3.14.5/config=47eda1
/2017-02-01/code=3.14.5/config=aaf371
```

Listing all experiments that have been run on data of a particular day now is straightforward and intuitive.

5.3 Derivation database

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 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 6. 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.

5.4 Implementation

In this section we describe how storage schemes can be implemented and we give an example of how the various implementations can be used in an experiment regarding a machine learning pipeline that predicts the likelihood that users will click on ads of various categories.

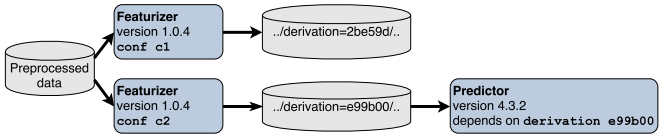


Figure 6: The same pipeline stage with different configuration results in different datasets

The interface for storage schemes has a method *pathFor* that implements Definition 5.1. Moreover, it also has an example implementation of a *save* method, that writes a dataset to the path specified by the storage scheme as well as the derivation to the derivation database as described in Section 5.3.

```
trait StorageScheme {
  def pathFor(pd: PartialDerivation, jp: JobParams): Path

  def save(dataset: Dataset,
           partialDerivation: PartialDerivation,
           jobParams: JobParams): Unit = {
    val path = pathFor(partialDerivation, jobParams)
    write(dataset, path)
    val derivation = partialDerivation.apply(jobParams)
    storeToDatabase(derivation)
  }

  def write(dataset: Dataset, path: Path): Unit
  def storeToDatabase(derivation: Derivation): Unit
}
```

Given all the interfaces that we have described, we can now show how all of this works together in an example of click prediction. Imagine a class *ClickPredictionPipeline*, which defines a series of transformations of click data, and how a random forest classifier should be trained. The data transformations depend on the configuration whether the hashing trick should be applied, and the classifier depends on the configuration of how many trees should be trained. Moreover, the pipeline specifies on what input data sets it depends by using *InputDataTemplate*, as described in Section 4.2.

Given this pipeline, we can now define a pipeline job that executes the pipeline given job parameters (necessary to expand the input data template into actual dataset paths), a pipeline configuration (specifying whether the hashing trick should be used and the number of trees), and a storage scheme such that it can store the output of the pipeline to the right location.

```
object ClickPredictionJob extends PipelineJob {
  override def inputDataTemplate =
      ClickPredictionPipeline.InputDataTemplate

  override def run(jobParams: JobParams, config: PipelineConfig,
                   storageScheme: StorageScheme): Unit = {
    val partialDerivation = PartialDerivation(
        ClickPredictionPipeline.Version, config,
        inputDataTemplate)
    val derivation = partialDerivation.apply(jobParams)
```

Table 1: Example content of derivation database

Derivation	Name	Version	Config	Job params
59ff55818	ClickPrediction	0.3.2	date: 2017-03-04 country: es	trees: 100 hashing: false
d8780980a	ClickPrediction	0.3.2	date: 2017-03-04 country: nl	trees: 100 hashing: false
b3391ebb2	ClickPrediction	0.3.2	date: 2017-03-04 country: es	trees: 50 hashing: true
7693ef20a	ClickPrediction	0.3.2	date: 2017-03-04 country: nl	trees: 50 hashing: true

```
    val pipeline = new ClickPredictionPipeline(config)
    val model = pipeline.fit(derivation.inputData)
    val output = model.transform(derivation.inputData)
    storageScheme.save(partialDerivation, jobParams)
  }
}
```

Suppose that some data scientist wants to run an experiment to compare whether 100 trees without hashing performs better than 50 trees with hashing. Since a model is trained per country, the experiment should be run with the click data for two countries. The code that the data scientist needs to write, looks as follows.

```
object ClickPredictionExperiment {
  def run(): Unit = {
    val configs: Seq[PipelineConfig] = Seq(
      PipelineConfig(num_trees = 100, hashing = false),
      PipelineConfig(num_trees = 50, hashing = true))
    val allJobParams: Seq[JobParams] = Seq(
      JobParams(date = 2017-03-04, country = es),
      JobParams(date = 2017-03-04, country = nl))
    for {
      config <- configs
      jobParams <- allJobParams
    } {
      ClickPredictionJob.run(jobParams, config,
                             StorageScheme.Experimentation)
    }
  }
}
```

This code specifies a number of pipeline configurations that should be compared, as well as various job parameters, which are used to define different input data. The machinery introduced in this paper makes sure that the output of all the runs done by this experiment are persisted in different locations, and that detailed information is written to the derivation database. Table 1 illustrates what data will be stored in the database.

Over time the derivation database provides valuable insights into the experiments that are run on pipelines used inside the company. Moreover, by further integrating this with systems like ModelDB [19], it will be easy to compare the results of experiments.

6 REGISTRY OF RUNNING PIPELINES

One of the most important data sources in Schibsted is user behaviour. The team responsible for producing this dataset wanted to upgrade the schema. Because they did not know which teams were using the old version, they had to produce both versions for months until everything was switched. Since there is a lot of click data, this was both expensive in terms of storage and computation. It also meant that the team now had two pipelines that they had to monitor. Moreover, the switch caused all kinds of bugs in the data consumers, resulting in several stressful weeks for the teams. In this section, we outline the sketch of a system that helps in managing scenarios like this.

A system like GOODS[8] crawls existing datasets and provides an interface to explore what datasets exist, what schema they have, and other metadata. Crawlers like GOODS are made for when there are already many datasets and it is hard to enforce a common library in an organization. Significant effort is put into making it scale. Furthermore, some types of information cannot be crawled easily, but must be inferred, e.g. how often does a batch job run and which fields is it using.

We argue that in many organizations, it is feasible to enforce data applications to use a *common library* for production jobs. By doing so, we can enforce that data applications store metadata to a *central registry*, where they register their data dependencies and outputs. By having this information in a central registry, developers can get insights into what production pipelines are currently running and depending on which version of their data. Also, it is easy to gain insight into what applications are using a particular field, which is valuable in order to assess the impact of having a bug in a certain field.

Since the central registry holds metadata about all datasets, its consumers and the producers, the need for a crawler becomes less urgent. Moreover, because of the added metadata, much stronger inferences can be made, which helps managing running pipelines in production.

This idea is closely related to the concept of provenance, i.e. the ability to answer why questions about computations and datasets [3]. In other words, provenance allows understanding how each dataset was created, i.e. from which computation and what data it originates from. Provenance applies to existing datasets, but not to datasets that will be created in the future. By including information about job schedules, we can also reason about what would happen if a pipeline would be stopped.

6.1 Registering job schedules

As the registry needs to know what jobs are running or scheduled to run in the future, we introduce the notion of a *job schedule*. A job schedule refers to a particular pipeline job, a pipeline configuration, and a storage scheme. Note that as long as a job is scheduled, none of these change. Furthermore, it has a description of the schedule, and a reference to the team that owns this scheduled job.

```
case class JobSchedule(pipelineJob: PipelineJob,
                       config: PipelineConfig,
                       storageScheme: StorageScheme,
                       schedule: Schedule,
                       team: Team)
```

The storage scheme specifies where the job is producing its output. Recall from Listing 2 that pipeline jobs describe on what input data they depend upon. Consequently, if a job schedule is

enabled, then it is possible to inspect what data sources it uses, and also the versions of these sources. Knowing the team is important for notifying them, when jobs producing their input are planned to be shut down.

When a pipeline is deployed, the job schedule is sent to central registry. Similarly, when a pipeline is stopped, the central registry must also be notified. The interface to the registry is as follows.

```
trait CentralRegistry {
  def notifyOfDeployment(jobSchedule: JobSchedule): Unit
  def notifyOfTermination(jobSchedule: JobSchedule): Unit
}
```

This interface describes how programs can notify the central registry. The registry itself should be running on a server. A possible extension of the registry would be that it is also responsible to run the jobs.

6.2 Monitoring

If all teams notify the registry of their deployments, then the registry has an up-to-date view on what pipelines are running, when they are running, and what versioned data sources they use. Consequently, the registry can check whether there are jobs scheduled to run that depend on data versions that are not being produced anymore. This is especially useful for jobs that do not run regularly, e.g. once a month.

Moreover, because the registry knows when jobs are supposed to run and where they should write their output, it can also monitor these output locations to see if the datasets are actually produced and how much delay they have. A system that automatically monitors all pipelines that are running helps developers in not having to create their own monitoring solutions.

6.3 Breaking changes and automatic upgrades

The central registry tells you what teams are using what dataset version. However, it does not tell you what fields of that data are being used. Knowing this is especially important when a breaking change is made to a subset of the fields.

Suppose team T_J develops job J, which consumes fields f_1 and f_2 from dataset D with version v_1. Team T_D, that is responsible for creating D, finds a bug in field f_3 and releases version v_2. Since J is not affected by this change, it could automatically switch to v_2 in theory. By making J consult the central registry to get f_1 and f_2 from D, the registry could instruct J to read v_2 of D. Consequently, the team of J does not need to change anything to switch version.

Suppose team T_J2 develops job J_2, which consumes f_3 from D. The change of T_D will affect job J_2. To become aware of all effects, T_D can query the central registry for all consumers of D with v_1 before it switches to v_2. Since the central registry also knows the team responsible for every job, T_D can communicate with those teams to coordinate a plan to switch to v_2.

Once J_2 has migrated to v_2, team T_J2 stops the schedule for their old v_1 job and starts a new job schedule for the new v_2 job. All changes in job schedules are sent to the central registry. Consequently, the central registry has an up-to-date status about whether there are any jobs still using D with v_1. Once all have migrated, the central registry could notify T_D that it can switch off the v_1 schedule. The central registry can also easily give an overview of all jobs that are scheduled to run, but whose output is not used by any other job.

7 CONCLUSION

In this paper we have described our extension to [17] 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 reuse, 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. Our extension of partial derivation simplifies navigating datasets produced by experiments as well as production pipelines.

We have introduced several schemes of where to store the output of a pipeline. Special storage schemes are designed for experimental and production pipelines. Moreover, we have introduced the concept of partial dataset derivations to have a more user friendly folder structures.

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. We illustrate the implementation of all newly introduced concepts by providing code snippets of key interfaces and functions, which are being applied in use-cases for production jobs and experimentation.

Furthermore, we outline the sketch of a central registry for running pipelines, which is envisioned to simplify the coordination of running pipelines with interdependent teams.

A remaining point for improvement 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.

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] A. Bhardwaj, A. Deshpande, A. J. Elmore, D. Karger, S. Madden, A. Parameswaran, H. Subramanyam, E. Wu, and R. Zhang. Collaborative data analytics with datahub. volume 8, pages 1916–1919. VLDB Endowment, Aug. 2015.

[3] A. Chen, Y. Wu, A. Haeberlen, B. T. Loo, and W. Zhou. Data provenance at internet scale: architecture, experiences, and the road ahead. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

[4] 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.

[5] 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.

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

[7] 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.

[8] A. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. Goods: Organizing google's datasets. In *Proceedings of the 2016 International Conference on Management of Data*, pages 795–806. ACM, 2016.

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

[10] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, New York, NY, USA, 2014. ACM.

[11] M. Maddox, D. Goehring, A. J. Elmore, S. Madden, A. Parameswaran, and A. Deshpande. Decibel: The relational dataset branching system. volume 9, pages 624–635. VLDB Endowment, May 2016.

[12] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(1):1235–1241, Jan. 2016.

[13] 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.

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

[15] 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.

[16] 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.

[17] T. van der Weide, O. Smirnov, M. Zielinski, D. Papadopoulos, and T. van Kasteren. Versioned machine learning pipelines for batch experimentation. In *ML Systems Workshop NIPS 2016*, 2016.

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

[19] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. Model db: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, page 14. ACM, 2016.

[20] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1113–1120, New York, NY, USA, 2009. ACM.