Proceedings of Machine Learning Research 82 (2018) 23-32, 4th International Conference on Predictive Applications and APIs

An Architecture and Domain Specific Language Framework for Repeated Domain-Specific Predictive Modeling

Harlan D. Harris HARLAN@HARRIS.NAME
New York, NY

Editor: Claire Hardgrove, Louis Dorard and Keiran Thompson

Abstract

When repeatedly fitting related predictive models within the same domain, for similar problems, it's helpful to have tools to support an efficient, high-quality workflow. This paper describes a theory of the architecture for such tools and for the interfaces among predictive models and other aspects of a software system. Additionally, it describes an open-source reference implementation of this design, written in R, focusing on a Domain Specific Language for one specific repeated predictive modeling task.

Keywords: Domain Specific Languages, Machine Learning, Software Engineering, Software Architecture

1. Introduction

For data scientists in certain industry roles, especially in SaaS firms, a common problem is the need to fit not just one predictive model, but many related, yet independent models. For the purposes of this paper, consider the hypothetical Acme Systems, a Software as a Service (SaaS) firm that provides applications used by large retail enterprises. One such applications incorporates a model of sales at chain stores—the Acme Chain Model. This model has been sold to many dozens of clients, with variants on the same basic model template.

Although the general problem is similar—sales are probably due to trends, business cycles, pricing changes, etc.—each company the firm works with will have different data, different inventory, and different patterns of cause and effect. Acme's data scientists, then, have to build and integrate custom predictive models for each client. To maintain quality, consistency, and timely fitting and re-fitting of these models, Acme data scientists use a set of domain-specific workflow tools designed to allow the team to efficiently build and deploy models. Importantly, a well-designed workflow can let them apply their growing domain knowledge to solve the problem better and more efficiently at each iteration.

This pattern applies under a specific set of circumstances, which I'll call Repeated Domain-Specific Modeling:

1. The predictive model is a component of a broader software product, with predictions likely being exposed to end users to help them make better decisions.

2. The same general problem is being solved many times, as when a SaaS firm builds separate models per customer.

3. The models are similar enough for investment in shared infrastructure makes sense, but are different enough to require data scientist customization and insight.

This paper has two primary contributions. First, I describe a set of abstractions for thinking about Repeated Domain-Specific Modeling. These abstractions clarify important design choices about representations and interfaces. Second, I provide and describe an implementation of this pattern as an open-source software repository, written in R (R Core Team, 2017). The code[1], available at `https://github.com/HarlanH/featgen-demo`, shows how to use a problem-specific Domain Specific Language (DSL) to describe individual instantiations of a predictive modeling problem, allowing the domain expertise of data scientists to be incorporated, while yielding a highly efficient workflow.

1.1 Related Work

Aspects of this work have been informed by contributions from several sources. There are many workflow tools for Predictive Modeling, both in general and for specific problems. Those tool sets often include APIs for integration with other systems. Commercial examples include SAS, Domino Data Lab, and Azure Machine Learning Studio, while many organizations have built their own solutions using open-source and in-house technologies.

There are several prominent examples of DSLs for Machine Learning *in general*, such as `scikit-learn` and its Pipeline for Python (Buitinck et al., 2013), `mlr` and `caret` for R (Bischl et al., 2016; Kuhn, 2008) and several of the deep-learning frameworks such as TensorFlow (Abadi et al., 2016). (See also Portugal et al., 2016) Although these provide a clean, powerful short-hand for creating predictive models, they are DSLs for the general problem, typically of supervised learning, not for specific business problems.

One public example of Repeated Domain-Specific Modeling was described recently by Uber. Their Michaelangelo framework (Li et al., 2017) allows their team to define high-level feature extractors (what I'll describe as Feature Transformer Generators, below) using a Scala-based DSL, add them to a common Feature Store, and easily re-use and extend them. This framework is specific to Uber and their domain, but shares commonalities with the work presented here.

Some of the theory described below extends work originally presented by (Morra, 2016). Their recently-described and open-sourced Aloha framework (Deak and Morra, 2018) is also a system for Repeated Domain-Specific Modeling. Aloha includes a feature-definition DSL with abstraction layers that separate data extraction from transformation, and a strong focus on the appropriate boundaries between systems.

2. Theory

In theory, machine learning systems are mathematical models mapping between arbitrary representations; in practice, machine learning systems are continuously interacting with

1. Some of the patterns described here were explored by me and my colleagues when employed by The Advisory Board Company (Washington, DC). None of the details of the specific model are relevant here, and all of the open-source code was written entirely from scratch. In practice, the system we built allowed us to effectively create customized models at the rate of one per business day, which had a substantial positive effect on the business.

other systems, human and technical. These interactions, and the interfaces between machine learning systems and other systems, are critically important to the overall effectiveness and value of what's been built.

To build a predictive model, you have to be able to represent entities in the world, such as the properties of the object or event you're predicting, and the outcome or label you wish to predict. Additionally, as a data scientist, you must represent and incorporate your own knowledge into the system. The No Free Lunch theorem (Wolpert and Macready, 1997), which proves that no predictive model can be optimal at all problems, implies that algorithm choice alone, by aligning the underlying problem with the algorithm's Inductive Bias, will have a substantial impact on model quality. Further, feature engineering, or the process of transforming data to make it more usable and useful to the learning algorithm, is a key art in predictive model development (Domingos, 2012).

Repeated Domain-Specific Modeling systems include several parts—the application, the predictive model, the data scientists who will be operationally responsible for keeping the predictive model tuned and accurate, and the users of the application. (See Figure 1) Communication among these parts, be they human-machine interfaces or machine-machine interfaces, must be efficient and consistent for the system as a whole to run smoothly.

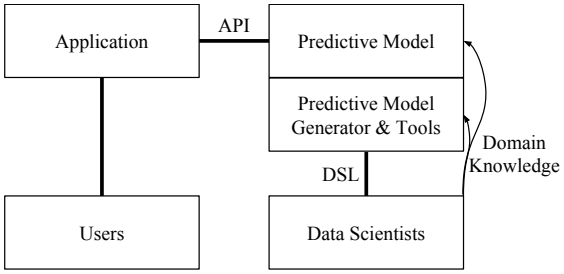

Figure 1: Important components of a Repeated Domain-Specific Modeling system include a well-designed API interface between the model and the main application, a well-designed DSL for efficient expression of the model to the model generator by data scientists, and a modeling framework that allows data scientists to insert their domain knowledge into both the model generating framework and individual models.

2.1 Representations at the Machine Learning and Application Interface

Consider the interface between a machine learning system and a broader software system that interacts with it. The application provides data and entities to be predicted to the machine learning system, then receives a prediction or score, to be provided to a user or used for a decision.

Several well-known ideas from software engineering and systems design apply. First, *Separation of Concerns* says that subsystems should do one thing well, encapsulating processing, and hiding irrelevant details from other subsystems. Second, the *Standard Service Contract* principle from the theory of Service Oriented Architectures (SOAs, Erl, 2007) says that subsystems need to define a commonly-understood language for communication, and

should be clear about what each subsystem should expect from another. Third, Conway's Law (Wikipedia, 2017) says that system structure will inevitably reflect team structure. When data scientists are building predictive systems, this very likely means that a predictive service will be a separate module or service in the overall system.

An implication of these ideas is that the application need not, and should not, know about the internal data processing of the predictive model. Specifically, the application should not know about *feature engineering*, the process of re-representing an entity to be scored in such a way that a predictive model can most effectively make predictions.

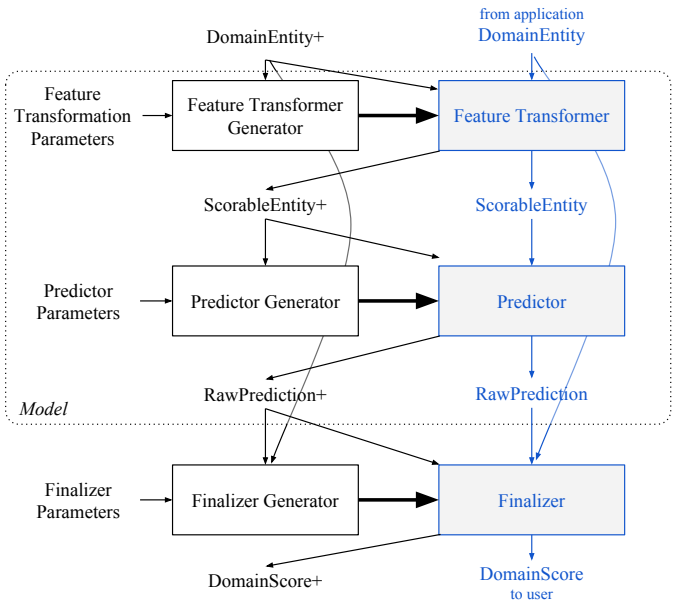

Figure 2: Representations for Repeated Domain-Specific Modeling systems. (Left) Training data flows through a training pipeline, generating both training predictions and a scoring system. (Right) Data to be scored goes through the same translations steps, using the scoring system inferred by the training data. The components within the dotted box are part of the predictive model; outside the box is the application.

Figure 2 describes a framework for thinking about the changes in representation involved in predictive modeling. (See also Morra 2016 and Deak and Morra 2018.) Predictive modeling is essentially two linked processes, Training and Scoring. Importantly, they share a parallel structure with regards to the representations that flow through their processes. And equally importantly, the Training process can be thought of as a system, or a higher-order function, that *generates* the Scoring process as an output of its data processing.

To maintain encapsulation, the application should provide Training and Scoring data to the model in an (agreed upon by contract) format that reflects the semantics and structure used by the application—a `DomainEntity` in Figure 2. The application might provide an API for serving `DomainEntity` objects, while the model might provide an API for real-time

scoring. There are a number of frameworks, commercial and open source, for exposing predictive models as API services. See the demo code for a simple approach.

2.2 Efficiently Contributing Expertise

Most readers will be aware of Drew Conway's influential Venn Diagram of Data Science (Conway, 2013). The oft-neglected third set in that diagram is "substantial expertise." When building a Repeated Domain-Specific Modeling system, it is important that data scientists be able to leverage and incorporate their substantial expertise about the problem as effectively as possible. This need exists at three levels of system building.

1. Data Scientists must be able to *contribute to their own workflow.* Nobody is more qualified to design and build tools and systems for predictive model generation than data scientists. This includes critical components such as tools for data and model validation. (See Harris 2016; Parker 2017.)

2. Data Scientists must be able to *add their domain expertise to the model-generation process.* In a repeated model-building scenario, it's important that general facts that data scientists learn about the domain be represented in the model-generation process. For instance, if you learn that seasonality and holidays are relevant for retail sales, the model-generation framework should directly represent those concepts in the system.

3. Data Scientists must be able to *add their domain expertise to individual models.* In a repeated model-building scenario, it's also important that facts that data scientists learn about specific examples of the domain be easily incorporated into the model. These facts might include data-quality issues or exceptions to general rules.

3. Implementation

As noted above, I have developed an open-source example of what this framework might look like. Most of the steps that might be used in a production environment are included: defining a model with a DSL, fitting a model, viewing a model archive file, generating a standard report about the model, scoring the model via web service, and inspecting the model in production. Two other important steps, tools for interactively validating and exploring the source data, and interactively validating and comparing candidate models, have not been implemented. But see (Harris, 2016) for approaches to those tools.

I will focus the rest of this paper on how a model is defined with the DSL. Readers interested in the other components are encouraged to explore the provided software.

3.1 Domain Specific Language

Listing 1 shows a sample model definition file for a company called Rossman Stores (See Kaggle.com, 2015). The model definition is written using an internal DSL (Fowler, 2010), leveraging the syntax and execution model of an existing language, in this case R. To train a model, the user uses the command line: `acm.R train rossman.R rossman.Rout`. The `acm` executable loads the `rossman.R` script shown and executes it in a context where functions such as `acme_chain_model` and `promo` are available. This is the only code written specifically for Rossman.

```
1  acme_chain_model("rossman") %>%
2    store_type(collapse=list(ab=c("a", "b"))) %>%
3    competition_distance(trans='boxcox', na='median') %>%
4    one_week_ago_sales(cap_to=c(2000,15000)) %>%
5    sales_trend(days_back=4*7, trunc_to=c(-100, 100), na='mean') %>%
6    promo %>%
7    current_sales %>%
8    get_data %>%
9    train(target="current_sales")
```

Listing 1: Sample model-definition file, `rossman.R` in the sample code, for an instantiation of the "Acme Chain Model" for client "Rossman." Defines a human-computer interface allowing efficient specification and customization of the model template for individual clients. The `%>%` pipeline operator, defined in the `magrittr` package (Bache and Wickham, 2014), unrolls nested function calls, passing the results of the previous function as the first argument to the next.

Line 1 executes a function that generates an object, initially with only the name of the customer. (See Figure 3A.) Each subsequent line calls a function with that object, adding additional elements to the object, such as feature generators (Lines 2–7), or the data (Line 8) or the model itself (Line 9). The result of running this script is a model object that gets saved to a file for further processing, such as inspection or deployment.

There are many details that make this DSL powerful, including a mix of feature-specific and general data transformations. (Note that the code in Figure 3B is specific to the Acme Chain Model, while C, D, and E are not.) Next, I'll describe the Feature Transformer Generator (FTG) pattern (see Section 2.1, above), and a lower-level but analogous pattern used to handle missing data.

3.2 Feature Transformer Generators

Figure 3B shows the implementation of the `promo` feature. Importantly, the design of this Acme Chain Model DSL uses FTGs that are based on the semantics of the problem, rather than on the structure of the data provided by the application. This allows the data scientist tuning this model for Rossman Stores to think about their promotion history and policies, avoiding implementation details.

On the other hand, the fact that this FTG is short, and is written in the same language as data scientists use day-to-day, means that people can switch easily from operations—using the framework to fit models—to development—extending the framework based on their accumulation of domain knowledge. This is an example of how this framework allows data scientists to incorporate their expertise at various points in the process.

Figure 3B illustrates the basic FTG pattern. A feature object ("`feat`") is created with standard slots. Line 9 allows the user to add standard parameters to the object, such as missing data handling rules. Line 10 adds the new Feature Transformer (FT) to the model object's list of features, then the function returns with an updated version of the model object. The key logic is Lines 6–8, defining an FT that extracts a `promo` ScorableEntity

```
# A -- the model object structure
> str(model,1)
 $ custname: chr "rossman"
 $ features:List of 6
 $ data    :Classes tbl_df, tbl and 'data.frame': 1115 obs. of 6 variables:
 $ cv_preds:'data.frame': 1115 obs. of  5 variables:
 $ metrics :List of 1
 $ target  : chr "current_sales
 $ model   :List of 8
```

```
# B -- an example of a Feature Transformer Generator
promo <- function(x, ...) {
    feat <- list(
        name = "promo",
        pretty_name = "Promo",
        extract = function(self, data, ...) {
            data_frame(promo=data$activity[[length(data$activity)]]$Promo)
        })
    feat <- list_modify(feat, ...)
    x$features <- list_modify(x$features, promo = feat)
    x
}
# C -- generating ScorableEntity objects for Training
get_data <- function(x) {
    raw_data <- readRDS(glue('{x$custname}.Rdata'))
    x$data <- map_dfc(x$features, function(feat) {
        map_dfr(raw_data, ~ feat$extract(feat, .)
    }
    x
}
# D -- storing NA-handing in Training
if (anyNA(new_cols)) {
    x$features[[pos]]$na_info <<- infer_missing(new_cols, feat$na)
    new_cols <- apply_missing(new_cols, x$features[[pos]]$na_info)
}
# E -- generating Scorable Entity objects, and applying NA rule, in Scoring
newdata <- imap_dfc(predictor_features, function(feat, pos) {
    new_cols <- feat$extract(feat, obj)
    if (anyNA(new_cols))
        new_cols <- apply_missing(new_cols, x$features[[pos]]$na_info)
    new_cols
})
```

Figure 3: DSL implementation code illustrating several key patterns. Extracted from code at `https://github.com/HarlanH/featgen-demo`

from the DomainEntity object. In this case, the logic is easy—just extract a Boolean from an object and create a single-column, single-row data frame. Other features might do much more complex processing (see the `sales_trend` feature for an example), or generate multiple columns.

The `promo` FT can now be used to extract features from a stream of data. Figure 3C shows a substantially simplified version of this process in the implementation of the `get_data` function. In the accompanying demo, the data is stored in an R archive file, but in practice, you might query a web service for a stream of training data. Lines 16–18 iterate over all combinations of features and training data points, generating a data frame that can be used directly by the actual machine learning algorithm.

Not shown is the implementation of `train`, which implements the `Predictor Generator` method, creating a set of scored training data, and a `Predictor` method than can be used for Scoring.

It's also worth noting the role of the `Finalizer`, a component of the *application* that translates the raw prediction to something that an end-user can use to make decisions. It's common, for instance, to translate a raw probability score to a recommended action, or a red/yellow/green risk category. These are User Interface and Design decisions, however, and the code that implements them should be part of the application. Data scientists can and should certainly provide advice on this mapping.

3.3 Missing Data

Figure 3, parts D and E, show aspects of the implementation of a critical component of this pattern — how missing data should be handled. In a SaaS context, the patterns of missing data per client may be quite different, and the best transformations may vary substantially. At a high level, a policy specified in the DSL (or in some cases by default rules set in a FTG) is applied to a new column with missing data by a standard `infer_missing` function. That function does not actually impute the missing data, but instead returns a value that is used for imputation later. In the case of mean imputation, this would be the value of the mean of the column. The identical `apply_missing` function, with identical parameters, is then applied in both Training (D) and Scoring (E), ensuring that both Training and Scoring data have the same distributions.

The demo implementation supports mean, median, min, max, and constant imputation of missing data. The same pattern is used for continuous transformations, such as Box-Cox, which requires a parameter to be fit, stored, and used in production. Other transformations such as collapsing rare categories, or Windsorizing data at percentiles, can be implemented the same way.

4. Discussion

Using the Repeated Domain-Specific Modeling pattern, an organization can create a workflow for creating and deploying many related models, maintaining flexibility while maximizing efficiency and reliability, and ensuring clean separation of concerns and the ability of data scientists to apply their domain knowledge throughout the process.

There are a number of improvements to this framework that could be made. The open source code provided is an example of how to start implementing the pattern, but it is not

a package; anybody wishing to use the code would have to discard the Acme Chain Model pieces, and rewrite those components with FTGs for their own domain. It might be possible to extract general pieces of the code into an actual package that might reduce time-to-value. Also, although writing new FTGs is not generally difficult, additional tooling for doing so and setting up test cases would make it even easier and faster for data scientists to extend the framework as they learn.

Acknowledgments

Thanks to my former colleagues at EAB for helping me think through and implement an earlier version of this pattern, and to Jon Morra and others for their thoughts on earlier versions of the theory components.

References

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. URL `https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf`.

Stefan Milton Bache and Hadley Wickham. *magrittr: A Forward-Pipe Operator for R*, 2014. URL `https://CRAN.R-project.org/package=magrittr`. R package version 1.5.

Bernd Bischl, Michel Lang, Lars Kotthoff, Julia Schiffner, Jakob Richter, Erich Studerus, Giuseppe Casalicchio, and Zachary M. Jones. mlr: Machine learning in R. *Journal of Machine Learning Research*, 17(170):1–5, 2016. URL `http://jmlr.org/papers/v17/15-066.html`.

Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

Drew Conway. The data science venn diagram, 2013. URL `http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram`.

Ryan M. Deak and Jonathan H. Morra. Aloha: A machine learning framework for engineers. In *Proceedings of the SysML Conference*, 2018.

Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

T. Erl. *SOA Principles of Service Design*. The Prentice Hall Service Technology Series from Thomas Erl. Pearson Education, 2007. ISBN 9780132715836. URL https://books.google.com/books?id=mkQJvjR2sX0C.

M. Fowler. *Domain-Specific Languages*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010. ISBN 9780131392809. URL https://books.google.com/books?id=ri1muolw_YwC.

Harlan D. Harris. Workflow tools for helping with model-fitting, 2016. URL https://www.rstudio.com/resources/videos/workflow-tools-for-helping-with-model-fitting/.

Kaggle.com. Rossmann store sales, 2015. URL https://www.kaggle.com/c/rossmann-store-sales.

Max Kuhn. Building predictive models in r using the caret package. *Journal of Statistical Software, Articles*, 28(5):1–26, 2008. ISSN 1548-7660. doi: 10.18637/jss.v028.i05. URL https://www.jstatsoft.org/v028/i05.

Li Erran Li, Eric Chen, Jeremy Hermann, Pusheng Zhang, and Luming Wang. Scaling machine learning as a service. In Claire Hardgrove, Louis Dorard, Keiran Thompson, and Florian Douetteau, editors, *Proceedings of The 3rd International Conference on Predictive Applications and APIs*, volume 67 of *Proceedings of Machine Learning Research*, pages 14–29, 2017. URL http://proceedings.mlr.press/v67/li17a.html.

Jonathan Morra. Data science at eharmony: A generalized framework for personalization. Strata + Hadoop World, 2016. URL https://conferences.oreilly.com/strata/strata-ny-2016/public/schedule/detail/51731.

Hilary Parker. Opinionated analysis development, 2017. URL https://www.slideshare.net/hilaryparker/opinionated-analysis-development. rstudio::conf.

Ivens Portugal, Paulo S. C. Alencar, and Donald D. Cowan. A survey on domain-specific languages for machine learning in big data. *CoRR*, abs/1602.07637, 2016. URL http://arxiv.org/abs/1602.07637.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2017. URL https://www.R-project.org.

Wikipedia. Conway's law, 2017. URL https://en.wikipedia.org/w/index.php?title=Conway%27s_law&oldid=814575211.

David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.