Wypasek Data Science, Inc.

WDS Model Specification Definition

Extended Documentation, History and Examples

Version 0.5.3

Author(s): Christian Wypasek

Abstract

This article is the extended documentation (including draft notes and commentary) for the model specification style of Wypasek Data Science, Inc. (WDataSci, WDS). The spec includes schemas for data and function signatures and model development and implementation. The design presented here is based on over 20 years of best practices while also emphasizing interoperability with other standards. For a technical documentation without commentary, see WDS Model Specification Definition, Technical Documentation.

Contents

1 Preamble

This article is intended to include extended commentary and draft notes for the documentation of the Wypasek Data Science, Inc. (WDS) Model Specification format. A parallel final technical document will be stripped of any superfluous notes and blog-fodder. The sections, footnotes, and items (which can be skipped) will always be prefixed with the contributing authors initials, as defined in Section 3.

This article is also under construction. Parts of the spec and implementations may be available at `https://github.com/wdatasci` before documentation is incorporated here.

2 Purpose

The **WDS Model Spec** format comes from over 20 years of best practices for the development and implementation of models used primarily for loan level asset based finance. These best practices covered the complete life cycle of a data science projects. Therefore, it also includes details and tools for data handling and function signatures. As these practices have evolved, so have external standards such as **PMML** or **PFA**.[1] A redesign or refresh event is always the opportunity to also review what is common in the industry. Even through WDS uses this spec internally, the spec needs to easily and robustly communicate what is used externally.

2.1 Use cases?

Before diving into motivating discussion, when is the WDS Model Spec useful? Whenever part of a model building and implementation process can be run without point-and-click. For example, if one can run a model building script or code block in **batch mode**, for lack of a better term, then that script or code block can likely be automated. Automation might not be in the sense of automated model developement, although it could be.

If a model building process involves excessive user point-and-click in some interface, the process described below is likely not helpful. But then again, users of extensive point-and-click likely have too much free time to worry about things like operational risk.

2.2 Context

The models and examples discussed herein start with a few basic types, but one of the benefits of a having a flexible specification is that a data scientist should be able to rapidly configure and include a new approach. For example,

[1] The official site for PMML and PFA is `https://www.dmg.org`.

one should be able to replace an older component, maybe a logistic regression, with a some other model type, such as decision trees, neural networks, ensembles, etc. As long as the models take inputs from a common space and the outputs are returned to another common space with the same intended purpose, the models should be functionally interchangeable. This does not imply the models must be equally useful, but interchangeability is necessary for rapid and robust comparison.

The motivating models from loan level asset based finance often involve multiple linear components (sometimes referred to as scores). For some projects, WDS uses a specialized form of competing risk proportional hazards model that has four different components, where two are generally multinomial, at least two are time-varying, and one special use. A single project, such as auto life of loan risk and prepayment, often creates an entire suite of models for different segmentations and purposes, such as New-Auto vs Used-Auto, New-Loan vs Seasoned-Loan, Seasoned-Loan with vs without adverse payment history, etc. The competing risk models are also just one part of a larger cash flow model that can include at least two other model types for different events and behaviors during the life of a loan.

A fundamental premise of the spec is a **coffee-cup-and-dougnut** topological approach[2], in the following sense: If the information being encapsulated is well-defined and fully encompasses the details required, than translation into another well-defined standard ought to be fairly straight forward. For example, if the information required to implement a linear score is correctly encapsulated (marked-up), then writing the implementation in a code format or into another information encapsulation, such as PMML, should be formulaic.

After a motivation discussion and practical discussion of the Project/Model/Variable structure, Variable handling details will include naming conventions and standardized individual variable treatments. The article wraps up locations for open-sourced examples and tools.

3 Author Introduction, Notes, and Commentary

As the spec grows, any additional contributors with commentary will be added here.

3.1 Christian Wypasek (CJW)

My own personal development of the style of this specification (and through version 1) has evolved through past work with several firms, including GE Capital, First Union/Wachovia, Merrill Lynch, and Centerbridge Partners. No information contained herein is proprietary to any of these shops. While I have worked with truly excellent people in the past and have shared and discussed many of the topics, the concepts here are ones that I have actively furthered and used.

Many data scientists have finally figured out that model implementation is important. Here, I will use the term *data scientist* not just for a statistician or modeler, but for what encompasses what can be considered multiple roles[3] in a modern organization. As part of an asset based finance firm, a structured product investment team, or a private equity firm, these roles include:

- Data Science Project Manager or Business Interface (part 1)
  Identifying business needs, objectives, and resources available
- Data Engineer (part 1)
  Database design, data cleaning and prep
- Data Scientist or Modeler
  Analysis, model specification, fitting, validation, etc.

[2] In topology, *the coffee cup and the doughnut* is a classic example of the topological equivalence between two surfaces (for a gif animation example, see `https://en.wikipedia.org/wiki/File:Mug_and_Torus_morph.gif`). At a high level, two surfaces are topologically equivalent if one surface can be morphed into the other by stretching, contracting, or deformation, but without introducing any breaks or holes and without closing any holes or joining regions. Or, more precisely, there exists a 1-1 and onto mapping from the points of one surface to the points of another which preserves neighborhoods. Therefore, if two surfaces are topologically similar, then given some degree of closeness, points close on one surface are mapped to points close on the other and any two points far apart on one are not mapped to points arbitrarily close on the other.

[3] For an example, see `https://www.kdnuggets.com/2015/11/different-data-science-roles-industry.html`.

- Data Engineer (part 2)
  Implementation, model integration
- Data Science Project Manager or Business Interface (part 2)
  Final integration into cash flow models and delivery

If a data scientist has limited support and effectively performs all of the roles oneself, then a critical realization is that final implementation design (or the anticipation thereof) should be a part of data and model design and should start early. Best practices have shown that encapsulating the data around model building:

- Creates a standardized understanding about how modeling data and final implementation source data is prepared.
- Standardizes and facilitates rapid model development.
- Reduces implementation and operational risks.
- Creates multiple implementation options.

The spec contained herein targets these objectives, the remainder of this section and *CJW Notes* sections are for additional motivation examples, commentary, or past use details.

3.2 CJW Notes: A personal comment on *data science* vs *machine learning*

When reviewing and writing the previous section, I came across an article[4] in which the author refers to machine learning engineers and researchers as new job descriptions. It never ceases to amaze me how the latest buzz words (I'm sorry, *tags*) get used. But then, I also came across another[5], which gives the 1980s-2010s as the when machine learning became popular. This makes sense since I always thought I was doing machine learning.

Note: Every time I come back to this discussion, I have to re-evaluate these comments. Even over the past several years, public discussion on machine learning has changed (or grown). It is reasonable to believe that *machine learning* does include *all of the above*, but with further subsetting into *supervised* or *un-supervised*.

As far as I am concerned, artificial intelligence, machine learning, deep learning, neural networks, statistics, dynamical systems, stochastics, measure theory and anlaysis, computation, big data, etc., they all just used to be called *math* (or at least in the *mathematical sciences*). On a separate note, mathematicians vs physicists in finance is an entirely different discussion.

One could argue that *modern* machine learning is used in areas with a feature space that is far too complex for normal statistical methods. To which I would have to ask, is it really something new or do we just finally have enough computing power to do cool things? But, hey, they are really cool things, we should just use the right tool for the right job.

Outside of specific really complex feature spaces or without a specific goal (definitely un-supervised), when someone discusses having used machine learning (or for that matter, any technique), the open question remains: Is being used correctly? In the wrong situations and for the wrong purpose, it might just be sloppy data science with a cool name. For example, neural networks have been around for decades. However, in the regulated finance industry, few have been approved because naive use tends to overfit and the financial risk is too great.

During my post-doc, queuing theory motivated concepts led to a method that was effectively feature identification in multi-dimensional random fields. Then, in industry, self-organizing schemes and Bayesian model averaging motivated robust model selection methods from among potentially hundreds of variables. Does that make it worth dropping the machine learning tag? Does that make one a data scientist or a machine learning researcher? In my opinion, a good mathematician or practicioner should be able to solve a problem in more than one way.

In particular, for models that require third party validation and/or where black box models are not acceptable, one might still use advance techniques for feature identification or variable identification (model selection). The technique might not be acceptable as a final product, but these new techniques can be used to guide and inform a

[4] `https://www.kdnuggets.com/2018/12/why-shouldnt-data-science-generalist.html`
[5] `https://www.sas.com/en_us/insights/analytics/what-is-artificial-intelligence.html`

more readily accepted form. To illustrate further, a model that could be useful, but never implemented, is not as good as an other with extremely similar but less powerfull output that is more readily approved.

3.3 CJW Notes: Young modelers

While a Post-Doc in 1996-97, I was looking for a permanent job and had one of the most truly awful interviews of my life. The company's owner made a statement to the effect of never expecting a new employee right out of college with an MS or PhD to really be an independent contributor for at least two years. Needless to say, I was agitated and determined to prove that person wrong. After growing up with a family business where I contributed as teenager, I was determined to make an impact in my first position. My first industry model included learning SAS (from poor code written by someone else), learning the lingo, and being done within two months. Even though I was personally driven, experience has shown that it really does take some time for young or fresh-out-of-school modelers to learn the ropes and get out of bad practices that were easy to pickup in school. This is especially true if they are not mentored for the long term.

Even if a firm has detailed model development, maintenance, and validation protocols, I would advocate that young professionals who want to build models in the real world would benefit from a basic learning exercise:

- Start with an ugly set of data, maybe even something some other first year analyst cobbled together. Generally speaking for loan level asset based finance investing, this will involve historical data on a set of loans or a portfolio and the goal would be to generate a forward forecast based on an outstanding or new portfolio.
- Have them build a model using all of the best skills they learned in school. The smarter students will recognize that this usually requires decomposing the problem and building individual component models. Maybe focus on one, such as default propensity.
- If the young modeler is smart, they will pick up on what the best practices are at the firm. Anything unusual, like "hit the side of the refrigerator before you open it up", will be ignored until adverse actions are realized through non-adherance.
- Have the young modeler demonstrate the variable investigations, testing the model fit, and generating a forecast.
- Since they have now demonstrated all they might have learned in school....
  - Now, they have to explain things to a colleague who wants to *understand how things work* and wants to *see the coefficients* and wants the modeler to explain how he/she used *principal components* because someone else told them that was how it was done.
  - Ok, principal components are totally unrelated or inappropriate to this problem and even if the modeler did somehow use the tool, interpreting the eigenvectors and coefficients will get them a group of frustrated individuals who thought they knew something. And besides, the modeler used a neural network which cannot be understood by looking at it. Oh, this neural network only gives one default estimate probability, but objective was for life of loan cash flows, back to the drawing board....
  - Maybe demonstrating marginal model response by variable is a good idea. Furthermore, why can't the modeler provide a tool so that other colleagues can spend their own time changing a variable value and seeing the response (so they can also figure out how to add their hacks to it....) But, of course, those colleagues may not have R, Python, SAS, or whatever software was used, but everyone has Excel.
  - What? One of the deal leads does not want to learn Knime? How totally un-civilized of them. That person brings in multiple $100M investment deals, get over it. What that person does not have time for is an associate wasting their time.
  - In fact, the next teammate on the project who uses the results is using an Excel-based cash flow model. So, have the young modeler implement the model in a spreadsheet.
  - Then, maybe implement in VBA, since 10k spreadsheet formulas can be replaced with a couple of array valued functions.
  - Oops, something does not match up, can't the modeler just rerun the model on all loans.
  - After going blind comparing model building code to Excel VBA, the young modeler searches for ways to implement R in excel. What is *Managed Code* again and why should anyone care?

  - Scratch that anyways, there is too much data for a workbook without taking forever. The model was built on samples from the database, which involved querying the database, pulling the data into other structures such as workspaces in R, SAS data files, Python pickles, flat files, etc., maybe building additional model fitting data sets, evaluating results, and then pushing results back into a database. Have the modeler create a process for looping through all of the loans, which has since grown to a database of several million...

- After the young modeler has finally gotten the buy-in needed....
  - Hey, the seller finally answered that request for additional data and found another five years of data which included a period with an economic downturn....
  - And, the seller was able to get additional borrower credit information and added new fields for data which fills holes in some other fields...
  - Oh, by the way, a colleague's buddy at another firm suggested something over drinks the previous night. The modeler wasn't there, and what was suggested isn't possible, but since the buddy is *really smart*, shouldn't it be considered.....
  - Oh, by the way, the investment committee is meeting again next week, when can the modeler be done updating everything....
- Ok, the bid was lost. Some other competitor bought into the seller's opinion that losses moving forward will be one half what the portfolio has shown in the past. This is even though the seller wants to start promoting new products to lower credit quality borrowers.
- But, there is another project being put out for bid next week.

If after an appropriate amount of time, a young modeler does not figure out ways to use best practices (maybe even help the firm improve its tool set), or in the worst case, refuses to, they might just need to be managed out.

3.4 CJW Notes: Lessons learned

There are several comments and statements from my childhood that will stay with me forever. The foremost was my father saying I would make a good inventor because I was lazy. This was not just fatherly criticism. He also would say, if you need the right tool for the job, either go get it or make it.[6] In data science, if one really writes significant amounts of code and some pattern is done repeatedly, one tries to automate it. I recognized this early in my first position in industry after graduate school, and not just because SAS programming is so incredibly boring.

In that first position, a supervisor literally told me that our job was to create a model that we could write as a scorecard on paper and then hand that paper to someone else to implement. The firm had recently spent significant funds to fully implement its first scoring model for policy underwriting. After that supervisor moved on, the remaining team was left trying to motivate the company to implement the next new and improved version. After all, why should operations invest significantly in a new implementation when the last was working fine?

To remain relevant, I turned to working with IT on implementation. Back in 1998/1999, the excellent folks in IT came up with a solution where the COBOL based mainframe system could wrap a system request send it to a separate server running an early edition of CCN/Experian's Strategy Manager System.[7] If our team could provide the Strategy Manager configuration that would score a loan, implementation would not have to involve a COBOL programmer cutting and pasting coefficients,[8] or even worse, typing them in by hand. Even though the Strategy

[6] Our family business was a commercial art studio, how does this relate to data science? Especially before computers and on-demand prototyping, items such as demo product packaging for trade shows would have to be created by hand. Solving real world small production problems and standardizing is not just a coding idiom. One of our first direct computer uses and one of my first programs was written on a hard-copy terminal at the local community college to create setting tables for our second-hand stat camera. By 1989, the studio had its first 16-bit Mac, a scanner, a printer, and a typesetter.

[7] Experian's Strategy Manager system has evolved and now appears to be part of Experian PowerCurve system, `http://www.experian.com/strategy-management/strategy-management-systems.html`.

[8] Even by 1999, COBOL programmers were becoming a rare breed. The language was good for dealing with large text-based records on mainframes and all fixed width data field work needed to be configured by hand. I will never forget the contractor who used to be proud of taking 8am classes in college, for when you overslept, oh well.... His programming style was not very different, go for volume and then go back and fix if you have to...

Manager System had a GUI for setting up scorecards, it was still quite tedious. The GUI created a configuration file that was then uploaded to a server.

In a time before XML was common, the configuration file was a plain ASCII file with a custom format. It was relatively easy to decipher which data in the custom format was related to the scoring model and each variable. Furthermore, this portion of the file could be written out directly. The key to making everything work was to tag the critical data, make sure that was the that data could be extracted from the model build, and then use that same data to write out the implementation.

One additional phrase adapted from my childhood is the **Wonder Bread approach**. During a kindergarten field trip to a local Wonder Bread[9] factory, a production line manager held up a fresh ice cream package and talked about "never touched by human hands!" for product safety. What is the Wonder Bread approach for data science? Getting away from handing a piece of paper with coefficients to someone else who would have to type that in. Typing, copy-and-pasting, input data handling, configuration, and output data handling are just some of the places for human operator error. In credit risk modeling, implementation mistakes are operating risks.

To illustrated the Wonder Bread approach, at another position, one colleague had built and implemented a default risk model. A third colleague requested a minor change and the modeler estimated one week as the time frame to update the model, implement, and test. After the modeler went on to another firm, I moved his code into the process described below and was able to fit, implement, and test multiple variations within a few hours.

Speaking for myself, this type of model specification has a proven track record. This is also at least my fifth time writing a model specification. Each time it is important to review what is considered state of the art and determine whether or not to adapt/adopt/extend or create something new. Therefore, this specification takes into account personal best practices, but also incorporates new thoughts so that the framework could also communicate present standards.

4 Motivation And Past Efforts Towards Model Specification

Even in today's modeling world of artificial intelligence, machine learning, big data, or any other latest catch phrase, models for loan level asset based finance, in particular, often need to be reviewed by management, regulators, implementers, and other vested parties. Not only must a modeler prove they developed a model well without over fitting, they must communicate the model in a way so that the implementation treats new subjects in a way similar to how the development process handled the input data.

This is not to say that one technique is the best or another *provincial* or outdated, but consider one modeler who can be able to build an impressive neural network for some aspect of loan level behavior. Compare that modeler to another who might use those same techniques to identify critical effects but then provides a nearly equivalent model that can be implemented in any platform, language, in database, in excel, on server, but also review-able and without a black box. For these reasons, a well designed linear score component model specification will remain critical in a modeler's toolkit.

4.1 A Basic Model Component

At the core of many statistical models is a linear *score* or the sum of the products of coefficients and independent variables. Model types with a basic score construct include not only general linear models, but also generalized linear models such as logistic, Poisson, and other parametric distributions. Furthermore, semi-non-parametric models such as proportional hazards can include several score components.

Consider the ubiquitous case of linear regression where the model form is:

$$Y = \beta_0 + \sum_{i=1}^{n} \beta_i X_i + \epsilon$$

here, in the usual way, Y is the dependent random variable, $\{X_i : i = 1, \ldots, n\}$ are the independent random variables, $\{\beta_i : i = 1, \ldots, n\}$ are the coefficients with β_0 as the intercept term, and ϵ is a noise term with the usual assumptions

[9] `https://www.wonderbread.com`

(independent and identically distributed, normal, etc.). Without any loss of generality and taking the artificial variable $X_0 = 1$, the score is

$$\beta X = \sum_{i=0}^{n} \beta_i X_i$$

and is the dot-product between the vector of coefficients, β, and the vector of independents, X.[10]

In practice, when building a component score, it is common to either add transformations of variables to be considered or replace a single variable, X_i, with a collection of *artificial* variables, say $\{X_{ij} : j = m_i, \ldots, n_i, m_i \in \{0, 1\}, n_i \geq 1\}$. Again, without any loss of generality, one can consider each variable, X_i, as vector-valued and associated with a vector-valued coefficient vector, β_i. (To be clear, if X_i is not replaced with a vector, notionally, $m_i = 1$, $n_i = 1$, and $X_{i1} = X_i$.) A total score can then be decomposed into component or marginal scores:

$$\beta X = \sum_{i=0}^{n} \beta_i X_i = \sum_{i=0}^{n} \left(\sum_{j=m_i}^{n_i} \beta_{ij} X_{ij} \right)$$

Here, the component scores are arranged by variable, but individual scores, $\beta_i X_i$, might be derived through different techniques such as with controlling factors or segmentations.

For an individual variable, the use of artificials can serve multiple purposes including:

- To code missings: for example, $X_{i0} = 1$ if X_i is missing, and 0 otherwise. If no other artificials are used, $X_{i1} = X$ if X is not missing and 0 otherwise.

  Note: by convention, the 0 subscript will always be used for a coded missing indicator.

- To handle segmented versions of the data
- To handle extreme value treatments
- To handle simple discretization treatments and categorical variables
- To handle neighborhood or localized feature effects or non-linearities

In general terms, only a small number of special treatments are commonly used and so it makes sense to have a set of *recipes* to handle the usual suspects. The model specification documented here creates a standardized, extensible, and relatively efficient framework that can used to build up marginal scores into larger component models which in turn are used in larger systems and suites of models.

4.2 Motivating Example and Early Versions of the Specification

4.2.1 CJW Notes, 1997-1999

Circa 1997-1998, while working at GE Capital Mortgage Corp., I started using variable information bundles as a way to facilitate and instruct model building as well as to implement the final model and export final communications. For extremely large data sets, SAS[11] statistical software was the most commonly used language. SAS remains useful for large datasets, but without going into SAS-backwards programming, let us consider a loose example to motivate the concepts.

4.2.2 SAS Example

The example below represents a simple discretization of a variable. This may seem extremely trivial, but in 1997, this was the state of the art in credit scoring.[12] It involves the replacement of a variable X with a set of mutually

[10] Yes, technically if X is row vector valued random variable and β a row vector of coefficients, it should be written $X\beta^T$. Or, if column vector valued for both, $\beta^T X$. Here, the abuse of notation βX will be used.

[11] `https://www.sas.com`

[12] In traditional logistic credit score models, one can easily create a paper scorecard or score sheet like a judge in an athletic competition might use. An individual could then review the attributes of a applicant and create a final score by hand. This is the origin of the term *credit score* and the credit rating agencies still use the term **scorecard**. *Scorecard* is often used in reference to a model segmentation, for example, someone with little credit history will be scored with a different model or scorecard than someone with 40 years of credit usage. Any internet search on the terms FICO and scorecards should bring a significant number of articles for reference.
`https://nortonsafe.search.ask.com/web?q=FICO%20scorecards`
FICO stands for Fair Isaacs Corporation and was one of the first credit scoring firms.

exclusive range indicator artificials. In a simple logistic model, the coefficients of each indicator (scaled) gives one a value (points) to be added or subtracted from the base score.

For this illustration, suppose one has a data set with one row per subject. For each subject, one has a `FICO` variable and some response `Y`. SAS code for creating discretized artificials based on cut points of 620, 660, and 720 in SAS could look like:

```
/*SAS creates a table looping over the rows of
    an input table.  The body of the data step
    is evaluated for each row and can add new
    columns or variables drop unnecessary ones.*/

data modeldata;
    set inputdata;

    X=FICO;
    array XV X0 X1 X2 X3 X4;
    if X eq . then do;
        X0=1; X1=0; X2=0; X3=0; X4=0;
    end;
    else if X lt 620 then do;
        X0=0; X1=1; X2=0; X3=0; X4=0;
    end;
    else if X lt 660 then do;
        X0=0; X1=0; X2=1; X3=0; X4=0;
    end;
    else if X lt 720 then do;
        X0=0; X1=0; X2=0; X3=1; X4=0;
    end;
    else /*if X ge 720 then*/ do;
        X0=0; X1=0; X2=0; X3=0; X4=1;
    end;

run; /* end of datastep instructions */

/*SAS procedures implement the model fitting process.*/

proc reg data=modeldata;
    model Y=X0 X1 /* X2 */ X3 X4;
run;
```

Basic SAS macros (like LaTeXmacros) emit code[13] and their use objective is to eliminate redundant robotic code writing. One could then end up with something short and sweet like:

```
data modeldata;
    set inputdata;

    %VariableBundles;

run;

proc reg data=modeldata;
    model Y= &ModelVariables;
run;
```

Here, `%VariableBundles` is a user defined SAS *macro function* (potentially with arguments) which writes the if-then-else blocks (for clarity here, we will not go into the SAS macro mechanics required). The macro function is also used to add to the user defined SAS *macro variable*, `ModelVariable`, which the `&` operator expands to the code `X0 X1 X3 X4`.

Encapsulating the artificial set construction could be written using the SAS macro language as a sequential finite state parser as in the simple example:

```
/* declare the macro variable in a way which
   can be handled inside macro functions*/

%global ModelVariables;
%let ModelVariables=;
```

[13] In fact, I originally used LaTeX macros for literate programming. However, instead of the Knuthian way, LaTeX files provided the main documentation and emitted code. The code could have been *VarPackets* as described later, SAS, C++ code, or C++ headers, eliminating the need to maintain separate header files.

```
%macro VariableBundles;  /* start of macro function def */

    %global ModelVariables;

    /* a bundle of instructions for the variable FICO */
    %let Name=FICO;
        %let Handle=x;
        %let Treatment=Discretize;
        %let Cuts=620 660 760;
        %let DropArtificials=2;
        %let Coefficients=;  /* place holder */

    /* another macro function defined externally */
    %ProcessVariable(&Name
                    ,&Handle
                    ,&Treatment
                    ,&Cuts
                    ,&DropArtificials
                    ,&Coefficients
                    ,ModelVariables
                    );

    /* add a new bundle for each subsequent variable

    %let Name=NextVariable;
        Stuff......

    %ProcessVariable(&Name,&Handle,&Treatment,&Cuts
                    ,&DropArtificials,&Coefficients,ModelVariables);

    */

%mend; /* end of macro function def */
```

Notice that `X2` was dropped, which is necessary for non-collinearity in the regression model unless a `noint` option is used. The SAS macro function, `%ProcessVariable`, is generalized to handle any variable.

Two things should be immediately recognizable:

- First, adding a new variable is easy, just add a bundle of instructions. (Or, just as easily, comment out a bundle or its subsequent `%ProcessVariable` to not use a variable.)
- Second, `%ProcessVariable` could be used for other tasks.

In the base SAS language, the macro `%VariableBundles` would not be expanded or evaluated until called and `%ProcessVariable` could be redefined between calls. With the same `%VariableBundles`, but with `ProcessVariable` redefined, one could:

- Prepare variables for a model as above.
- Loop through the variables to create summary tables, graphs, distributions, etc., of the associated artificials.

  Note: This would enable not just a point imputation for missing data, but one could impute a distribution across artificials.
- Process the coefficients after the model was fit and create an updated bundle set with the estimated coefficient line for each variable.

  Note: This involves not just emitting SAS code evaluated immediately, but also reusable code into an external .sas file.
- With meaningful coefficients, `%ProcessVariable` could produce efficient hard written scoring code.

Suppose the regression generated coefficients of 1.5, 3, -1, -2 for $X0$, $X1$, $X3$, and $X4$, respectively. The bundle could be updated with:

```
    /* a bundle of instructions for the variable FICO */
    %let Name=FICO;
        %let Handle=x;
        %let Treatment=Discretize;
        %let Cuts=620 660 760;
        %let DropArtificial=2;
        %let Coefficients=1.5 3 0 -1 -2;  /* with a 0 inserted for the dropped variable x2 */
```

To create efficient scoring code, the `%ProcessVariable` could be redefined to produce something like in an external .sas file:

```
MarginalScoreForFICO=0.0;
if      ( FICO eq .   ) then            MarginalScoreForFICO=1.5;
else if ( FICO lt 620 ) then            MarginalScoreForFICO=3;
else if ( FICO lt 660 ) then            MarginalScoreForFICO=0;
else if ( FICO lt 720 ) then            MarginalScoreForFICO=-1;
else                                    MarginalScoreForFICO=-2;
FinalScore=FinalScore+MarginalScoreForFICO;
```

At this point, one could ask oneself, since SAS macros are just writing code, why not have it write out other types of code such as C/C++ (and, yes, written as suspiciously similar as possible to the code above):

```
double MarginalScoreForFICO=0.0;
if      ( isnan(FICO) )                 MarginalScoreForFICO=1.5;
else if ( FICO < 620  )                 MarginalScoreForFICO=3;
else if ( FICO < 660  )                 MarginalScoreForFICO=0;
else if ( FICO < 720  )                 MarginalScoreForFICO=-1;
else                                    MarginalScoreForFICO=-2;
FinalScore+=MarginalScoreForFICO;
```

Or Python:

```
MarginalScoreForFICO=0.0
if      ( FICO is None
          or numpy.isnan(FICO)     ) :  MarginalScoreForFICO=1.5
elif    ( FICO < 620  )              :  MarginalScoreForFICO=3
elif    ( FICO < 660  )              :  MarginalScoreForFICO=0
elif    ( FICO < 720  )              :  MarginalScoreForFICO=-1
else                                 :  MarginalScoreForFICO=-2
FinalScore+=MarginalScoreForFICO
```

Or R, C#, Java, JavaScript, VBA, Lua, Matlab/Octave, etc. The only effective differences in hard written code are small syntax details.[14] Certainly, in many languages a function provider could also be used, but for rapid distribution of compiled code across compute nodes, pedantic code is not unreasonable. Of course, the master source is the code containing `%VariableBundles`, all other code is derived and it is a matter of operational integrity that emitted code blocks are never edited individually.[15]

4.2.3 CJW Notes, 1999-2005

While at First Union/Wachovia, I developed a simple sequential markup pattern, which I called "variable packets". The models ended up being implemented in C++ and the SAS macro implementation was not flexible enough to handle some of the nuances.

4.2.4 Variable Packets

The variable recipes, as used pedantically in SAS as described above, was standardized into a style referred to as **Variable Packets**. An example of the bundled layout is below, but any one familiar with simple markup or markdown, such as yaml (`https://yaml.org`), will recognize the similarities which will be discussed further below. The bundled information and consisted of configuration files with instructions of the pattern:

```
Name: X
<Keyword1> for <Previously Declared Name> [case <CaseKeywordN>]: <Data>
<Keyword2> for <Previously Declared Name> [case <CaseKeyword2>]: <Data>
<Keyword3> for <Previously Declared Name> [case <CaseKeyword3>]: Begin
<Data>
<Keyword3> for <Previously Declared Name> [case <CaseKeyword3>]: End
```

[14] Python is, oddly enough, one of the more difficult because of the indent critical format requirements and the lack of {} groupings.

[15] In C/C++, the use of include directives makes this simple with minimal wrapping code. Other languages, like Java, would require top matter and bottom matter, but individual code files can provide just class method implementation.

Examples of Keywords and information included: Source Variables, Treatments, Critical Values, Coefficients, Use Lists, etc. This type of parameter file effectively requires a customized parser, but still easy enough to write, even in plain old C, Python, Lua, etc. Once a parser has digested the instructions and holds the organized data into an accessible form, it can can be used to generate code like the `%VariableBundle` above.

A more verbose example:

```
Name: FICO
Treatment for FICO: Hats
ZeroC for FICO: 0 1

Name: CLTV
Treatment for CLTV: Hats
ZeroC for CLTV: 0 1

Name: Intercept
Treatment for Intercept: Constant
Documentation for Intercept: Begin
    Obviously, a constant value of 1 as a variable.
Documentation for Intercept: End
PrepCode for Intercept case SAS: Begin
    *Any code that is used in a special way for just SAS;
    Intercept=1.0;
PrepCode for Intercept case SAS: End
PrepCode for Intercept case VBA: Begin
    'Any code that is used in a special way for just VBA
    Intercept=1
PrepCode for Intercept case VBA: End

CriticalValues for FICO: 550 650 750
CriticalValues for CLTV: 70 80 90
```

In this more verbose example, unusual language specific details could be handled if required through `Begin-End` blocks. Complex blocks wrapped in precisely paired `Begin-End` lines are treated as multi-line strings reserved for later processing. This permits embedded sub-models.

One useful feature was the required `for <Previously Declared Name>` qualifier. This allowed the format to break the required sequential nature of the effective finite state machine parser. Additional information such as the coefficients generated during the fitting process could be tagged with the appropriate variable and either appended or included later. Very simple parsers were written in C which could also pretty-print the format, realigned by variable and possibly combining other flat files with additional information, such as the coefficients that were generated in the statistical software. When starting a particular data project, a summary process could also be used to generate the initial bullpen or create additional variables.

For purposes of model development, the format also becomes a bullpen of variables that could be turned on and off easily. The parser tokenizes the data into a node-like structure for each variable and then the model building or implementation code could be emitted. However, breaking the `Name: X` declaration with commenting character such as `#Name: X` would effectively cause the parser to ignore lines or `Begin-End` blocks associated with it. Compare that with a yaml approach where commenting a variable out is more difficult or might take additional instructions:

```
Variables:
  - FICO:
    - Treatment: Hats
    - CriticalValues: [550, 650, 750]
    - ZeroC: [0, 1]
  - CLTV:
    - Treatment: Hats
    - CriticalValues: [70, 80, 90]
    - ZeroC: [0, 1]
  - Intercept:
    - Treatment: Constant
    - Value: 1
```

Yaml might be style focused on human readability, but a small amount of additional markup for a structured set of information does not necessarily destroy readability. Indent-based formats effectively require other software (maybe the parser) or style nuance to combine files from multiple sources while ensuring the proper integrity of the map from variable to the incremental metadata.

4.2.5 CJW Notes, 2005-2009

Even before leaving Wachovia in 2006 and joining Merrill Lynch, it became clear that Variable Packets was just marking up the configuration. XML, DTD/XSD, XPATH, and XSLT provide a framework for specifying the critical information, querying that data, and transforming the data into other formats.

While developing models for the complex cash flows for credit cards while at Merrill Lynch, XML mappings became available in Excel. For highly structured matrix models, the visualization available within the spreadsheet program can be a helpful front end from which XML files can easily be exported and pulled into XSD data-bound applications, providing ideal business-friendly tools for rapid development.

4.2.6 CJW Notes, 2009-2018

At Centerbridge Partners (CBP), we used a Vertica columnar database.[16] In 2009, Vertica version 3.5 was admittedly very difficult and cumbersome to use, but by 2018 and version 9, it had become a preferred tool. One of Vertica's most critical features for data cleanup and model preprocessing for loan level data is the flexibility of **user defined transform functions** (**UTDF**). Vertica's UDTF's provide a fenced environment were a single loan's time series data can be presented in a tabular form to a C++ or Java function where multiple new columns can be created, possibly with rows deleted or added, and then returned directly to the database. Despite these new very powerful features, there could be an excessive amount of redundancy in the SQL, C++, and Java code to implement, in particular for the input and output field specification. Enter again a similar data dictionary type XML file used for interface definition and automating the worst parts of programming.

The approach of this type of specification has enabled a very high level of efficiency. I joined CBP in 2009 as part of a small group of three, an excellent ABS bond trader who led the group, a structuring expert, and myself. Within the first months of joining CBP, starting literally from scratch and while working alone on data prep, model development and implementation, I had created a suite of at least a dozen competing risk models for subprime mortgages to be used for evaluating bonds. The mortgage work led to significant projects involving personal loans, lines of credit, time-shares, credit cards, auto loans, and others.

4.2.7 CJW Notes, 2018+

Starting Wypasek Data Science has been the opportunity for the latest incarnation of the approach. Each iteration must improve on past best practices, but also reviewing what is available while adapting, adopting, extending, and writing new IP.

A reader may recognize some similarities and parallels with PMML. This comes from two things: The first is just coincidence. The core structures are natural concepts, however, there are significant variable level differences. PMML was evaluated in the past, but was and still is inadequate for my purposes.

Secondly, there is a specific intent to enable the export of PMML from the WDS Model Spec XML through a transformation such as XSL. The encapsulated information in the XML must be a superset of the required information for the PMML.

Why not use something easier like json (PFA) or yaml? This is structured information. A coffee-cup-and-doughnut approach implies that you can invertibly map all of the information required. Json or yaml would require additional information implicit in the handling and processing. Of course, this can be done and there is no reason one could not use any other format or even some type of gui for that matter. If two processes are equivalent under a coffee-cup-and-doughnut approach, use the one that gets things done.

4.3 Generalizing the Specification

Author's Note: CJW: For the general XML-based specification, those familiar with PMML might recognize several key similarities. There are two reasons for this. The first is just coincidence. The overall structure including directives (header or extension), dictionary, and component models (model-elements) are natural constructs and things I have had in use for some time. However, the variable handling, especially for regression or score models, is substantially different. This primarily comes the use of this specification in both model construction and implementation. This will be clear later in the handling of artificial variable treatments, such as Hats *below. I had evaluated PMML in the past, but it was and still is inadequate for my purposes. The second reason for similarity is the objective to generate PMML as an output for external communication. If a transformation such as XSL is used to create PMML from the*

[16] `https://www.vertica.com`

WDS Model Spec XML, the encapsulated information in the XML must be a superset of the required information for the PMML.

This model specification will be used for data handling, model creation, model documentation, and model implementation. Since all required information is encapsulated, spec files can either be consumed directly in a model delivery engine or by a code writing engine to produce compilable implementation code. A detail that might not be immediately apparent is that hard-written compilable implementation can be produced in minimal time with minimal cost. The only requirement is the appropriate algorithmic code writers based on the model markup. At this point, as long as the required information is encapsulated, one should also recognize that PMML or PFA are just two other implementation styles. This methodology has already used for to implement in SAS, C/C++, C#/Java, R, Python, and VBA and others will be used. Why so many? The right tool at the right time, the underlying encapsulated mathematical models are the same.

Stepping back for a moment, suppose one has a simple model, say $Y = 3X + 1$, and this is to be evaluated over some set $\{x_i, i = 1, \ldots, N\}$. The conceptual basis of the specification is the following:

- Each value, x, of $\{x_i\}$ is an single instance of the X data element.
- Information about X (and any one value $x \in \{x_i\}$) is called its **Meta Data**, or data about data. Let us refer to the meta data as X^{MD}.
- For Y and the resulting values, $\{y_i = 3x_i + 1, \forall i = 1, \ldots, N\}$, let Y^{MD} represent its meta data.
- For the model evaluation process, (X^{MD}, Y^{MD}), the input and output meta data represent the model data **Signature**.

Use of the term, **Model**, has become so common that it is easy to loose the association that a model is just an abstraction of a process that wrangles data and applies weights or structures (parts of the mechanics) that were determined during what is referred to as **Model Fitting**. In this context, a model is the meta data about a particular mapping from an input space to an output space. A concept that also is generally lost is that Model Fitting essentially performs all of the identical data wrangling (and more). Model Fitting itself represents a *Model*, an abstraction of a process, one whose output space spans those undetermined mechanics for the final Model.

Canned statistical packages also make it easy to loose this association. The internal process of certain regressions is iterative, which implies that many poor or not-quite good enough models of the same structure were evaluated on a significant part of the input space. It follows that the most robust way to build and ensure a consistent implementation integrity is to use the same data wrangling spec for each.

As motivated in Section 4.1, a basic linear component model can be decomposed into marginal scores (each a mini-model). Going further, even the simples serving up of a variable value can have model-like properties, just consider the model $Y = X$. Therefore, as much upward concept inheritance as possible makes sense.

Each level of a specification is metadata for an object which generally includes data about how it is built from simpler objects. Often, this can be strictly hierarchical. However, during development, configuration details of a child element might depend information from the parent. A simple example is a multinomial model where the pre-normalized outcome propensities might depend of different sets of variables. In competing risks in particular, modelling all outcomes on the same set of variables can lead to confounding and the inclusion of many extraneous insignificant coefficients. In a strictly hierarchical structure, a lower level variable element would not indicate its outcome association. This would require a parent model to indicate the associations with multiple lists of variables. A general premise or goal is that the number of times an individual variable's specification and scope is required is as limited as possible. As in the simple examples presented above, if a variable as removed or added during development, it should ideally only have to be changed in one location, often using a simple On/Off attribute.

A model spec defines how to parse marked up data about how a process wrangles data. A parsed valid model does not have ambiguities and so therefore can direct implementation in whatever way is most expedient. One of the benefits of using an XML/XSD/XSL approach is that information can be parsed through common XPATH patterns. In validity checks or transformations, this can be used both up and down the tree. In the multinomial example, outcome association at the variable level can be checked against the possible outcomes in the parent model, and vis-versa. Hence, there are extensions which are contextual which as a programming idioms might be addressed through class inheritance.

Each of the main elements will have a discussion section in what follows, and Figure 4.3 visually depicts an implied recursive structure. There are several conventions and general goals:

- There should be a natural way to include model structures within larger collections, projects, or suites. Conversely, a model structure should have a natural handling of its own component models.[17]
- Models should be able to tell the model delivery engine if they are being used appropriately. Almost any time a block of code of the form `if case A and case B and case C but not case D ... then use Model X` is required for switching, one can create a small scorecard for *applicability*. The model delivery engine can ping models, scoring the applicabilities, until an appropriate or best model is found. The applicability scorecards are (likely) deterministic and minimalistic, but model selection becomes simple and less prone to coding errors.
- Component models do not necessarily have to be limited in structure. For example, the internal structure of a component may be a full PMML, PFA, compilable into byte-code, or other model. It follows that the *Dictionary* element, defining the core field names, data types, and project-specific aliases, be inheritable to component models. For example, a sub-model might add a field as a construction that subsequent child models can use, but the fields in higher levels should also be available. In XML/XSL/XPATH, extracting the master dictionary at the start of file processing is a basic request. Unnecessary redundant dictionary entries down-tree can be difficult to maintain.
- As in the basic model component discussion of section 4.1, there has to be a natural linear score definition.
- A basic linear score model must be able to be expressed as the sum of (possibly scaled) components.
  - The simplest component is a marginal score contribution associated with the grouped artificial treatment of a single variable.
  - However, a single variable might be used in several marginal score recipes, such as might be the case with interactions and control through segmentation variables.
  - Some components may be used for fitting, but not for implementation, such as might be the case when controlling factors are used for a sub-model.
  - A marginal score contribution may also be pulled from another model directly without being re-scaled in the fitting process. So, there must be a systematic way to query other models and select variables within a suite or project.
- Where necessary, a meta data element is labeled *QName* / *MD* to separate the meta data concept from the value of an instance, such as Field vs FieldMD. Model-like metadata will not have a redundant trailing *MD* since it is implied.
- A plural form (ending in either *s* or *Set*) of any element represents a collection of the singular type. For example, a Models element has 0 or more Model child elements. Following the three-part naming convention (See Section 5.2), the term *RecordSetMD* is for a *Record* object, a *Set* qualifier, and a *MD* meta-data representation.
- The common set of attributes for all meta data elements:
  - Name or ID (if required for unique identification)
  - Documentation (optional, but encouraged)
  - SignatureMD (required)
    - Dictionary (Synonyms: Input, Sources) (not required for a FieldMD)
    - Data type (Synonyms: Output, Representation) (required only if not directly inferred from Sources)
  - Global attributes (optional/as needed)
    - Object facets
    - Aliases
  - Contextual attributes (optional/as needed)
    - Applicability
    - Use

4.4 Field and FieldMD

The simplest data elements, **Field** (Value) and **FieldMD** (Meta Data), provide primatives(++).

[17] For example, their are several nuances to the style of competing risk proportional hazards model I developed back in 2000. The component models necessarily include a time varying and static score based on conal ordering of calendar time and loan age, which are then used to create as offsets in a Cox-type regression for other external effects based on loan age ordering, and then all components used to create a robust loan age based baseline function. The three components are bundled with an applicability score.

Separate models are also required for natural segmentations. Taking auto loans as an example, it would be natural to have a model for newly originated new car loans, another for newly originated used car loans, one for seasoned loans that have never been (seriously) delinquent, another for current loans that have had past delinquency, etc. For a given project, each model might have 4 components, but bundled with others (4+ in this example) as part of a model suite.

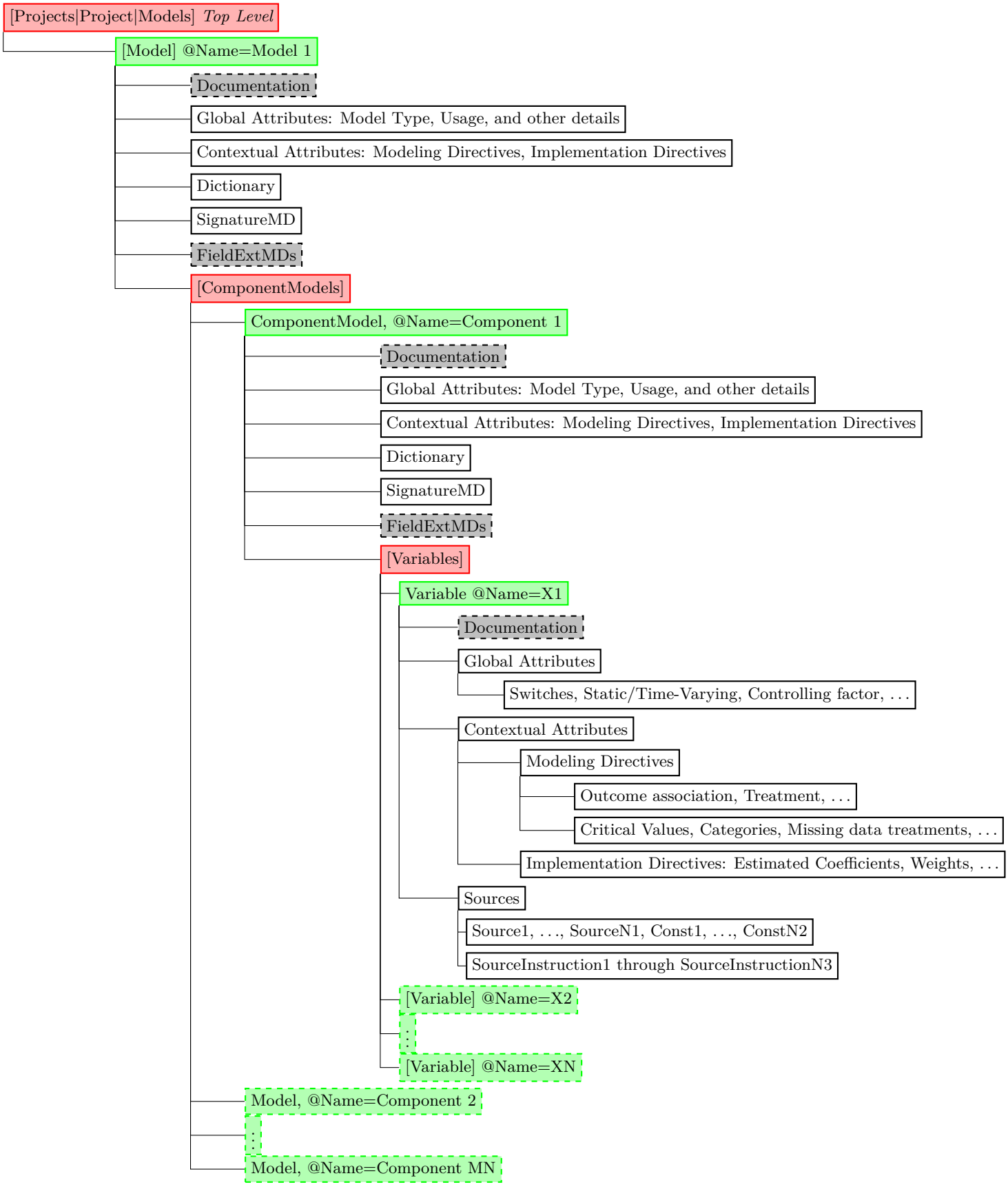

Figure 1: General illustration (not exhaustive) for Section 4.3.

There is a direct relationship between a FieldMD and an SQL column specification since the returned table from a query is one likely data source. For this reason, the synonymous term **Column**, might also be used in UDTF specification, see `https://github.com/wdatasci/WDS-ModelSpec/blob/master/WDS-Vertica/UDTF-Cpp/TSTest` for an example.

FieldMD data includes:

- Name (See Section 5.2)
- Data type (Synonyms: DTyp, Signature, Representation) (See Section 5.5)
- Global attributes
  - Object facets
  - Aliases
- Contextual attributes
  - Use
  - Project scoped aliases and import details

The most basic uses of FieldMD include the specification of SQL table, UDTF, and UDF signature columns, where the minimal set of information required is {Name, DTyp}. Object facets and aliases are useful for meta data organization. Contextual attributes examples include:

- Use in a SignatureMD can be I[nput], O[utput], or IO (both). For large data cleaning UDTFs, a simple markup of the required signature facilitates the programming environment and there may be significant overlap between the input and output records.
- Project scoped aliases arise from large projects which join many disparate tables. The process sounds like a mess and it is. It is very common to receive multiple data files, many with missed truly horrid inconsistent naming conventions, or even different miss-spellings from one table to the next. In these projects, mapping through an alias (and documenting the import details) can be used to standardize the data at the input stage.

4.5 Dictionary

A **Dictionary** (Synonyms: Namespace, DB Field Metadata) is an unordered collection of FieldMDs. As a child element of an object, a dictionary defines the scoped field namespace available to the object and siblings. The collection might be reduced to represent a minimal set or could be used as a bullpen of fields available for use during modeling. A dictionary is a meta data object that include:

- Name (If the dictionary is not standalone, there should be at most one per object.)
- **FieldMDs** (a collection of FieldMD)
  The FieldMDs can either be explicitly listed at the top level or include (without duplicates) all dictionaries, records, and signatures of sibling elements and their children.

4.6 Record, RecordSet, RecordMD, RecordSetMD, SignatureMD

An ordered collection of Fields (Values) is a **Record** (Synonyms: Row, Observation, Tuple) and a collection of Records is a **RecordSet** (Synonyms: Table value, Matrix, DataFrame data). These object correspond to meta data objects **RecordMD** and **RecordSetMD**, but RecordSetMD will be used as an obvious superset. A **SignatureMD** (Synonyms: Function Signature, Input/Output Maps) is a RecordMD with contextual FieldMD attributes as described in Section 4.4. Meta data objects include:

- Name (optional)
- **FieldMDs** (a collection of FieldMD)
  Either full FieldMD objects enumerated explicitly or the Names from the parent dictionary.
- ~~Signature~~ (Directly inferable from Fields)

4.7 FieldExt, Variable

Model-like objects that operate on fields and are scoped by their pair objects include **FieldExt** (Value) corresponding to **FieldExtMD** (Meta Data) (Synonyms: DerivedField, Transformation) and **Variable** (Meta Data).

A FieldExtMD is a Variable without contextual attributes that could be used across siblings in a Model or Project. It returns a single input space value for some later calculation, but may depend on several Fields.

Since a Variable is Model-Like meta data, it is consistent to refer a Variable's result or output. A Variable's result is used contextually. In a scored model implementation, the result is a single value in the model output space (which might be multinomial). Otherwise, a Variable usually returns a vector of size $N \geq 1$, where N is dependent on the treatment (See Section 5.6). The model development process can use this process to generate the coefficients for a marginal score. Unlike a FieldExtMD object that might be shared among siblings, the Variable object can contain contextual information shared back to the parent model.

This treatment does illustrate a significant difference with PMML which primarily represents an implementation. DerivedFields and Transformations in PMML may or may not be used and an artificial treatment such as Hats (See Section 5.6) can create an explosion of additional required DerivedFields needing to be defined (See Section 6.4). The PMML's Scorecard model represents a somewhat different scoring approach, is more compact, and could be a translation of Hats-based coefficients. Here, one objective is that the meta data for the artificials (including their coefficients) associated with a Variable should generally be retained with the variable.[18]

Meta data objects include:

- Name (required)
- **Sources** (Synonym: Fields)
  A collection of **Source** FieldMD, FieldExtMD, or Variable (See note below) references (at least one) and necessary Constants combined through specified treatments (See Section 5.8). These sources could evaluate other model types, in which case the Sources must include all model inputs and the embedded model must properly present its return signature.
  Note: Variables are scoped to the Model level. However, an individual Variable might be scoped to a particular segment (See Section 5.9), or might represent the interaction between other Variables (effectively, the cross product of the artificials of the other variables). For computational consistency, Segmentation Variables must be processed before any dependent Variables and Sourced Variables before their interaction.
- ~~Signature~~ (Directly inferable from Sources)
- Global attributes (For Variables) (examples)
  - On/Off switches
  - Static or Time-Varying
  - Controlling factor information. For example, a Variable may be used in model building, but not used in implementation.
- Contextual attributes (For Variables)
  - Modelling Directives (examples)
    - Outcome association (for mulitnomial models)
    - Treatment (See Section 5.6)
    - Critical Values, Categories
      Note: Including instructions to eliminate collinearities.
    - Missing data treatments (imputation)
  - Implementation Directives
    - Estimated Coefficients

4.8 Models and Projects

As discussed above, a **Model**, is the meta data associated with mapping an input space to an output space. A **Models** object is just a collection of models, and analogously, **Project** and **Projects** objects are synonymous super containers for Models.

A simple model might just contain Variables and directives, but a model in a suite often contains several component models, including one for applicability. The Model meta data includes:

- Name and/or ID
- Documentation and links
- Global attributes
  Model type and other details

[18] One obvious exception is Model level data, such as the estimated coefficient variance matrix. However, in a valid Model, there is no ambiguity between the artificial meta data and the associated columns/rows.

- Contextual attributes
  - Modeling directives
  - Implementation directives
- Dictionary (Accessible FieldMDs scoped to Model)
- SignatureMD (External visibility)
- FieldExts (Accessible FieldExtMDs scoped to Model)
- Either Variables Or Models or Projects
  A standalone Model willl only have Variables. A Project or Composite Model will only have Models. A Composite Model must always have an Applicability scorecard component to be queried by the delivery engine. A Projects object will only have Projects.

5 Fields and Variables

Whether preparing data for modeling or evaluating loan level models in final delivery system, the **covariate picture** per loan is a bundle of attributes (Fields or data elements) that are presented together. Some Fields are *static* in that they never change over the life of the loan, such as origination date, original balance, original term, origination credit score, etc. Some variables are *time varying* as the name implies, such as outstanding unpaid principal balance, current interest rate, remaining term, etc.

5.1 CJW Notes, Naming Conventions

One may somehow find the *Bad Moms/Grandpas/Teachers/etc.* movies funny, but there is nothing funny about bad naming conventions. Just when I think I have seen it all, someone surprises me. In the private equity investing world, it is not uncommon at all to get data from a company about its loans with field names like:

- **For Original Loan Amount**: `LOAN, OA, LOAN_AMOUNT, LOAN_AMT, ORIGINAL_BALANCE`
- **For Unpaid Balance:** `UPB, BALANCE, CURRENT_BALANCE, CURBAL, TRIAL_BALANCE`
- **For Interest Rate:** `IR, INTRATE, INTEREST_RATE, CUR_INT, CURRENT_INT_RATE, INT_RT, COUPON`

I have, in fact, seen some of these fields with multiple conventions used in the same tables. In one deal, the company's system used `cborr` for a borrower ID code (for *core-borrower*, very uncommon), and `co_borr` for the indicator of a co-borrower in the same tables. It also used multiple fields that started with `app`, sometimes for application data, sometime for appraisal data, etc.

In practice, many companies have merged past legacy systems, some with data old enough to have only eight character names. Much of what drove the mortgage financial crisis came from the commoditization of loan level data underlying securitizations such as from LoanPerformance.[19] For decades modelers have been stuck with one programmer's bad choice in a field name using `deliq` for *delinquency* (usually abbreviated `delq`).

An excellent discussion of naming and coding styles using reverse psychology is given by Roedy Green.[20] It is worth reading every once in a while as a reminder, even if one finds Java naming conventions dismal.

There is a very simple rational for the naming convention suggested below. Invariably, a banker will get a list of fields, toss them at you and ask "is this everything you need?" Unless one gets lucky and the person who prepared the data provided a concise well-organized schema with a dictionary, a common thing to do is drop them into a spread sheet, sort them, and start ticking them off. Do I have any or all of the usual:

[19] Now part of CoreLogic `https://www.corelogic.com`, LoanPerformance data was not responsible for the mortgage crisis, but the data transparency gave the market a false sense of performance security before things went bad. Poor modeling of that data also made many shops over compensate when performance did turn. In general, bond prices dropped when delinquencies started climbing and before any losses hit the bonds. By not understanding mixture distributions, many modelers missed the fact that CDRs (conditional default rates) would eventually go down and many bond sellers liquidated too low. That gave value investors an opportunity to by very cheap bonds, at least until the rest of the market players caught up.

Another critical data modeling issue that contributed to the crisis was the lack of data during recessions. The LoanPerformance data as a non-agency database depended on either master trusts or servicers contributing or reporting their data and did not gain significant size until after 1999. For the mortgage servicing data that I personally worked with, there was also a change in many systems around 1996 (possibly related to LSAMS servicing systems). Therefore, few market players had any exposure to mortgage loan performance during recessions, especially with credit data. Exceptions in the mortgage insurance space included GE Capital and PMI.

[20] `https://github.com/Droogans/unmaintainable-code`

- Loan parameters?
- Originator program fields?
- Borrower credit fields?
- Collateral fields?
- Performance fields?

Sometimes referred to as Product, Program, Purpose, Property, and Performance. At this point, one asks oneself, does sorting help or hurt?

5.2 Naming Convention

A **naming convention** is difficult to enforce in a specification schema and therefore generally represents an objective. Ideally, everyone would use a well informed metadata structure, such as ISO/IED 11179.[21] The basic premise is a **three-part naming convention**:[22]

```
Object [Qualifier] Property Representation
```

In general, although it might be acceptable to use CamelCase/StudlyCaps or dot/_ separated values, for this spec the objective will be CamelCase with underscores reserved for a possible trailing modifier:

```
Object [Qualifier] Property Representation [_Modifier]
```

The use case of trailing modifiers becomes clear when handling time varying attributes, such as `PrinBal`, `PrinBal_Lag1`, `PrinBal_Lag2`, etc.

Implicit in the standards are that abbreviations are allowed if well understood, but switching between whole words and abbreviations is not allowed. For example, `OriginationDate` might be a `Loan` property, but either `Origination` is spelled out everywhere or it is abbreviated as `Orig` everywhere.

The only time a word can be dropped is when it is repeating. For example, `LoanOrigDateDate` use a `Loan` object or class, `OrigDate` as the property, but since `Date` is a common representation, `LoanOrigDate` is the final.

Examples of `Object`s or classes of fields commonly used: `Loan, Borr[ower], CoBorr[ower], Prin[cipal], Pmt [Payment], Int[erest]Rate, Collat[eral], Prop[erty]`.

Examples of `Property`(ies) of fields commonly used: `Orig[ination], Cur[rent], Prin[cipal], Int[erest]`

Note, `Prin` *might be an object or a property, as in the case of* `PrinOrigBal` *for an original principal balance and* `PmtPrinAmt` *for the payment amount applied to principal.*

Examples of `Representation` terms commonly used: `Name, Number/Nbr, Rate, Pct, Date, DateTime, Status, Code, Ind[icator]`

Note, as a part of the convention, `Rate` *will be used for values directly interpreted,* `Pct` *will be used for values expressed "per 100".* Therefore, a loan with an 8.25% coupon, would have an `IntRate` field value of 0.0825, but an `IntRatePct` field value of 8.25.

Examples of trailing modifiers commonly used: `_Lag[#], _First, _Last, _Adj[usted]`

Note: One might ask why a modifier such as `_Lag[#]` is required since it could be queried within a panel, but that assumes a panel is always provided. A query sampling incomplete panels would have to provide such data.

5.3 CJW Notes, Aliases

Private equity data work often involves changing data sources on a deal by deal basis. For principal, interest, taxes, and insurance payment, one deal might have the field name, `PITI`, another, `Capital, Interés, Impuestos`

[21] `https://en.wikipedia.org/wiki/ISO/IEC_11179`
[22] `https://en.wikipedia.org/wiki/Data_element_name`

y Seguro. Several colleagues in the past have argued that every source should be pulled into a table *as-is*, so that if there are questions, one can converse directly in the language of the source. While having that cross reference knowledge is extremely important, at some point, data cleaning and merging sources can and will become an absolute nightmare. Experience has also demonstrated repeatedly that if a human being is pulling the data, they will tend to make typos in the field names. There will be even more mistakes if there is an extra layer of hand-written code attempting to piece together data sets when the source tables could have just been read in correctly in the first place. Furthermore, if every deal is *custom*, forget about leveraging all of your past work. If there are clearly equivalent names, use them.

5.4 Aliases

A flexible data import and scrubbing system should be able to handle to input fields which have identical meanings but may have had different source field names or aliases. There is no reason to believe the source field names follow the same or any naming convention. Source field names may also vary from project to project, but, within a particular project scope, field names should be unique and there may be one-to-many mapping from field names to aliases. Therefore, this spec advocates a dictionary repository from which projects can inherit a core set of field names and a project-scope cross table between sources and source field names.

5.5 Data Types

The specification is based on XSD and XSD data types should be accepted, but for practical purposes, only the subset below is generally used. Implementation internals should map to their nearest counterpart. A critical attribute is *nillable* for handling both missing and invalid data. Common implementations will involve languages such as Java or C# which have built-in types that can nillable or null-valued standard data types (`double` vs `Double`).

With the Github publishing of the project WDS_JniPMML_XLL[23], internal representation will use a `DTyp` abbreviation, which generally has 3 characters with the exceptions described below. This is solely to distinguish the types from the standard names which may treated differently in each language.

Core Concept	DTyp	Database(SQL)	XSD Type	Java
String (fixed length)	Str	char	xsd:string	String
String (variable length)	VLS	char/varchar	xsd:string	String
Double	Dbl	real(64), float(64), double	xsd:double (nillable)	Double
Int	Int	integer	xsd:int (nillable)	Integer
Long	Lng	integer, long	xsd:long (nillable)	Long
Boolean	Bln	boolean	xsd:boolean (nillable)	Boolean
Date	Dte	date	xsd:date (nillable)	Date
DateTime	DTm	datetime	xsd:dateTime	java.util.Calendar
Byte	Byt	blob	xsd:base64binary	byte

Additional details:

- TODO - Byte, `Byt`, is not fully implemented as of version 0.5.3.
- TODO - Date/DateTime values, when represented by strings, are ISO format only (i.e., YYYY-MM-DD based). The numeric value passing from C#/Excel to and from Java is not fully implemented yet as of version 0.5.3, but are passed as doubles.
- Strings are normalized, that is, without leadinging and trailing spaces, whitened controls, and with multiple spaces collapsed.
- Fixed length strings, `Str`, when used in a function signature via attribute `DTyp="Str"`, will have an optional attribute `Length="#"` for the fixed length. In an XSD, there are also simple types `Str#` with node `<maxLength value="#"/>`. Pre-defined types in the xml schema are `Str1-Str16`, `Str32`, `Str64`, `Str128`, `Str256`, `Str512`, and `Str1024`. `Str\` is the union of the size specific types.

[23] See `https://github.com/wdatasci/WDS-JniPMML-XLL`.

- Variable length strings, `VLS`, when used in a function signature via attribute `DTyp="VLS"`, will have an optional attribute `Length="#"` for the maximum length length.
- The term *Numeric* (`DTyp=Nbr\`) may be used for Double, Int, Long, or other XSD numeric types such as short, decimal, unsignedInt, unsignedShort, etc.
- List-Of objects (space delimited objects in XML node values) will be allowed with a `_List` modifier, such as `Dbl_List` for a List-Of-Doubles. This is to adhere to the three part

$$+trailing\_modifier$$

naming convention where `Dbl` is the name and representation.

5.6 Variable Treatments

Note: Code and workbook example implementating these treatments is available on the WDataSci's github, ***`https://github.com/wdatasci`****.*

There are some variables with obvious treatments, such as categorical variables or segmentation indicators. For the immediately following notes, let us just consider an effectively continuous variable X.

Back in 1997, simple discretization was still the standard treatment in credit scoring. Discretization is mathematically *pure* in the sense of measure theoretic concepts like integration or expectation.[24] However, in practice, a credit scoring model based on discretized variables has the undesirable effect of a small change in an input variable can create an unexpectedly large change in the score. An example graph is in Figure 5.6. The heights of the indicator functions are varied just for illustration. A scored linear combination is also included. Even though the graph is drawn continuous, the vertical jumps are discontinuities.

To discretize any variable, X, the cut-off points, $c_1, \ldots, c_m$, need to be chosen somehow. These points are sometimes referred to as *cuts*, *knots*, *CriticalValues*, etc. In Figure 5.6, these are just 0, 1, 2, and 3. Notice that 3 cuts yields 4 intervals of the range and 5 artificials since X_0 is by convention a missing indicator. Historically, cut selection used to be done by visual inspection of cross tabulations with the targeted response. If the space of X is coarse enough, it can also be done with a recursive algorithm to minimize SSE and target concepts like no interval too small and a reasonable number of cuts.

Author's Note, CJW: At GE Capital, I was asked, how do we create a linearly interpolated score?[25] The answer was easy. From a PDE class in graduate school, I knew them as *Hat* functions. A simple replacement for discretized artificials, hat functions (**Hats**), were used in the next version of GE Capital Mortgage Insurance's OmniScore, built by Matt Palmgren.[26]

One of the realities in academia is that if you think you come up with something new or at least call it something new in a different area, you can publish it. *Hats* are also:

- Finite element basis functions
- Order 1 B-splines (Bucketized or discretized variables as in Figure 5.6 are also Order 0 B-splines)
- Special cases of fuzzy logic neighborhood functions
- Integrated Haar wavelets

[24] The traditional Riemann integration was motivated by segmenting the range, using a discretized estimate of the target function, adding the areas of each rectangle, and then take the limit under refinement. Probabilistic expectation follows a similar concept, but where segmentation is the over measurable event sets in the probably measure space.

[25] With a scorecard using discretized variables, loan brokers could decipher the scorecard by pinging it and had figured out how to game the system.

[26] Palmgren, M. A., Wypasek, C. J., June 24, 2008, *Methods and apparatus for utilizing a proportional hazards model to evaluate loan risk*, US Patent 7392216

Discretizing a variable X with $n-1$ knots yields $n+1$ indicators $X_0, \ldots, X_n$, that cover the range of a variable X, have the following properties:

$$\begin{aligned} X_i(t) &\in \{0,1\}\ \forall t, i = 0, \ldots, n \\ &\quad X_i(t) \text{ is right continuous between knots } \forall i \\ X_i(t)X_j(t) &= 0\ \forall t, i \neq j \\ \sum_{i=0}^{n} X_i(t) &= 1\ \forall t \end{aligned}$$

Hats extends that slightly, except n knots are used to create $n+1$ artificials with:

$$\begin{aligned} X_i(t) &\in [0,1]\ \forall t, i = 0, \ldots, n \\ &\quad X_i(t) \text{ is continuous and piece-wise-linear everywhere } \forall i \\ X_i X_j X_k &= 0\ \forall t, i \neq j, i \neq k, j \neq k \\ \sum_{i=0}^{n} X_i(t) &= 1\ \forall t \end{aligned}$$

The second property in each group translates to: no two discretized artificials are non-zero at the same time and no more than two hat functions are non-zero at the same time.

Obviously, discretized indicators have discontinuities and hat functions are continuous, but with discontinuities at knot points in its derivative. Higher order B-splines can also be used and will be discussed later. An easy form of artificials which yields second order smooth effects is *iHats* or integrated hat functions with an example in Figure 5.6. As with hats, n knots will imply $n+1$ artificials. For a given set of n knots, $c_1, \ldots, c_n$, if $\hat{X}_i$ is one of the corresponding hat artificials, then

$$X_i(t) = \int_{c_1}^{t} \hat{X}_i(s)ds.$$

This does loose some of corresponding properties above, however, for linear components, it has the effect of fitting hats in the derivative space.

Since discretized variables and Hats are cases of B-splines with orders 0 or 1, why not just use B-splines? Outside of historical reasons for usage and naming, the usual recursive definition of B-splines[27] needs handling for values outside the knot sets and some notational standardization to fit into the framework presented here. We will use the acronym, *BZ#*, for our treatment using B-Splines of order #. So, starting discussion with m knots or critical values, $c_1, \ldots, c_m$,

- All treatments will have a missing indicator, X_0, which here is also includes any invalid data. Extreme values taken as outliers can always be handled by specifying *Clean Limits* , or boundary values, outside of which, the variable value integrity is considered questionable. For example, a FICO credit score of 100 or 8000. Some systems will extreme values as system coded flags.
- A B-Spline Order 0 (Discretized) treatment would have $m-1$ basis functions and a total of $(m-1)+2+1$ artificials, adding 2 end cap indicators for $(-\infty, c_1)$ and $[c_m, \infty)$, and a missing indicator. A B-spline order 0 basis functions would require at least $m = 2$, but this coding of variables allows $m = 1$.
- A B-Spline Order 1 (Hats) treatment would have $m-2$ basis functions. In the recursive treatment, each basis function for each order requires 2 basis functions from the next lower order. There will be $(m-2)+2+1$ artificials, but the left and right end caps will complement $B_{1,1}(x)$ left of c_2 and $B_{m-2,1}(x)$ right of c_{m-1} respectively.
- A B-Spline Order N (BZN) treatment would have $m-(N+1)$ basis functions. As with Hats, the end-caps will be complementary, and there will be $(m-(N+1))+2+1$ artificials.

The theoretical recursive construction B-Splines is easy enough when evaluating an entire space, such as in Figure 5.6, however, it is not necessarily conducive to rapid scoring. For low orders, the formula can still be expressed in a fairly straight forward manner. This can be tricky, since instead of finding all of the lower order basis functions and building up, when given a value, one finds which basis functions have support. For a pseudo-code example of the comparative complexity, consider:

[27] We will use the notation from `https://en.wikipedia.org/wiki/B-spline` where expedient.

```
let CV[1 to m] be the vector of critical values
let CLLeft and CLRight be the left and right clean limits
let x be the variable instance
let A[0 to last] be the vector of artificials to be generated
let last be the number of artificials (not counting the missing indicator since A starts at 0)
let T="Discretized|Hats|iHats|BZ2" be the treatment

if ismissing(x) or x<CLLeft or x>CLRight then A[0]=1
else if x<CV[1] then A[1]=1
else if x>CV[m] then A[last]=1
else {

    find i so that x>=CV[i] and x<CV[i+1]

    if T="Discretized" then
        A[i+1]=1
    else if T="Hats" then
        tmp=(x-CV[i])/(CV[i+1]-CV[i])
        A[i]=1-tmp
        A[i+1]=tmp
    else if T="iHats" then
        tmp=(x-CV[i])^2/(CV[i+1]-CV[i])/2
        A[i+1]=tmp
        A[i]=(x-CV[i]-tmp)
        (all iHats with support left of x need the simple area of the triangle added)
        for j=1 to i-1
            A[j+1]=A[j+1]+(CV[j+1]-CV[j])/2
            A[j]=A[j]+(CV[j+1]-CV[j])/2

    else if T="BZ2" then

        (temporary values...)
        let ia be the artifical index associated with i, here shifted because of the left-cap
        ia=i+1

        xMci = (x - CV[i])
        xMciM1 = 0
        if i>1 then (two basis functions have support)
            xMciM1 = (x - CV[i-1])

        let fo0 be the basis function value associated with found interval i (offset 0)
        let fo1 be the basis function value associated with found interval i (offset -1)
        let fo2 be the basis function value associated with found interval i (offset -2)

        fo0=0
        fo1=0
        fo2=0

        if i< m-1 then (there are no new basis functions after CV[m-2], only the catch all A[last])
            fo0= xMci / (CV[i+2]-CV[i]) * xMci / (CV[i+1]-CV[i])

        if i=1 then (the only left functions is the catch all A[1])
            fo1=1-fo0
        else if i< m-1 then (if i=m-1, then fo1 should be the catch all)
            fo1= xMciM1 / ( CV[(i-1)+2] - CV[(i-1)]) * ( 1 - xMci / ( CV[i+1] - CV[i] ) )
                    + ( 1 - xMci / ( CV[i+2] -CV[i]) ) * xMci / (CV[i+1] - CV[i])

        if i=2 then (the left-most catch all should be)
            fo2=1-fo0-fo1
        else if x>2 then
            fo2=(1-xMciM1)/(CV[(i-1)+2]-CV[i])*(1-xMci/(CV[i+1]-CV[i]))

        if i=m-1 then (there is no value in fo0 and fo1 should catch all)
            fo1=1-fo2

        if ia<last then
            A[ia]=fo0
        else
            A[last]=fo0

        if ia-1<last then
            A[ia-1]=fo1
        else
            A[last]+=fo1

        if i>1 then
            A[ia-2]=fo2

}
```

Figure 2: Simple Discretized Example (With Varying Heights To Illustrate), knots at 0, 1, 2, 3

Figure 3: Simple *Hats* Example, knots at 0, 1, 2, 3

With a few pre-calculated caches of values such as $CV[i+1] - CV[i]$, the artificial values can all be determined quickly and efficiently once the inter-knot interval is found. From the pseudo-code example, it is also clear that if a vector of coefficients, $COEF[.]$, corresponding to the artificials, $A[.]$, is available, the marginal score can be calculated easily.

5.7 The Usual Individual Variable Treatments

The usual variable treatments must have equivalent implementations in each coding language. For this version of the specification, the treatments will be:

Formal Treatment Name	Variable Type	Treatment Aliases	Critical Values	Artificials Produced
None	Numeric	Straight	n.a.	1
Constant	Numeric		n.a.	1
CodedMissings	Numeric		n.a.	2
Categorical	String		$N \geq 1$	$N+1$
NCategorical	Numeric	CategoricalNumeric	$N \geq 1$	$N+1$
Segmentation	String		1	2
NSegmentation	Numeric	Indicator	1	2
Discrete	Numeric	Discretize, Levels, Buckets, BZ0	$N \geq 1$	$N+2$
Hats	Numeric	BZ1	$N \geq 2$	$N+1$
iHats	Numeric		$N \geq 2$	$N+1$
BZ2	Numeric		$N \geq 3$	N
BZ3	Numeric		$N \geq 4$	$N-1$

Critical notes:

- There is no reason this core set cannot be extended, but all consumers would be required to implement the extensions..
- The *None* and *Constant* treatments are the only ones which do not have implicit missing handling and implementation results may be undefined. Therefore, *None* is rarely used. A *CodedMissings* treatment would be used instead, dropping the missing code from the model building process, and assuming the handling logic is provided elsewhere. *Constant* will be used for a term such as an intercept, where a default value may be given or implied.
- The number of artificials produced is also the number of coefficients required to score (per outcome in a multinomial score). Any one or more artificials not used in the model should have a 0-valued place-holder coefficient.
- Here, *Discretize* is strictly an input space partitioning into adjacent mutually exclusive intervals which must cover the entire input space. Uses of the term in other specifications are often equivalent to *CategoricalNumeric* which can be unordered and with a generally small collection of specific values. Ordinal values, either treated individually or as members of contiguous intervals, can be handled with this understanding of *Discretize* without confusion.

Figure 4: Simple *iHats* Example, knots at 0, 1, 2, 3

Figure 5: Simple *BZ2* Example, knots at 0, 1, 2, 3, 4

- *[N]Segmentation* is a special case of *[N]Categorical* with a single critical value or set used for an indicator of a population segment and subsequent variable instructions, see Section 5.9.

5.8 The Usual Variable Source Transformations

There are times when a particular data variable construction may have a limited scope in a model or suite, for example, `amortization_factor = (PrinCurBal)/(PrinOrigBal)`. The `amortization_factor` may not be available in the input space where Fields `PrinCurBal` and `PrinOrigBal` exist.[28]Continuing along the theme that each variable treatment is its own component model, it follows that each variable can have its own dictionary of source fields and transformations. Each can be understood as formula based on a sequence of core sources, say $s_1, s_2, \ldots, s_N$, and possibly a sequence of constants, $c_1, c_2, \ldots, c_M$. More than one transformation can be applied and they will be applied in order to the previous result, denoted below as X. In the case that more than one temporary result is required, X[#], will refer the the calculation # prior on the stack. For the first transformation to be applied or the case where no transformations are applied, $X = s_1$. For a naming and style convention, we will use RPN, with the understanding that all zero divisors result in a *null* value to be treated as missing or invalid data.

Transformation	# Inc Srcs	# Inc Const	Formula
None (X)	1	0	s_1
X[[#]].S[#].add	1	0	$(X[[\#]] + s_{[\#]})$
X[[#]].C[#].add	0	1	$(X[[\#]] + c_{[\#]})$
X[[#]].S[#].sub	1	0	$(X[[\#]] - s_{[\#]})$
X[[#]].C[#].sub	0	1	$(X[[\#]] - c_{[\#]})$
X[[#]].S[#].div	1	0	$(s_{[\#]} > \epsilon)\ ?\ (X[[\#]]/s_{[\#]})\ :\ null$
X[[#]].S[#].S[?].sub.div	1	0	$(s_{[\#]} - s_{[?]} > \epsilon)\ ?\ (X[[\#]]/(s_{[\#]} - s_{[?]}))\ :\ null$
X[[#]].C[#].sub.C[?].div	1	0	$(X[[\#]] - c_{[\#]})/c_{[?]}$
X[[#]].C[#].mul	0	1	$(X[[\#]] * c_{[\#]})$
X[[#]].C[#].min	0	1	$min(X[[\#]], c_{[\#]})$
X[[#]].C[#].max	0	1	$max(X[[\#]], c_{[\#]})$
X[[#]].C[#].pow	0	1	$power(X[[\#]], c_{[\#]})$
X[[#]].S[#].C[?].[eq|le|lt|ge|gt].nullif	1	1	$(s_{[\#]}\ [= \mid \leq \mid < \mid \geq \mid >]\ c_{[?]})\ ?\ X[[\#]]\ :\ null$

In order for any transformation to be added to the usual collection, model specification consumers and/or code writers would need to be updated to have that functionality.

5.9 Segmentation Variables

The term, *segmented*, in modeling an carry several connotations. Some modelers will use the term *segmented linear regression* to refer to piece-wise linear continuous regression, which happens to be an immediate and simple outcome of using the Hats artificial variable treatment described above. Others might use *segmentation* in reference to a population segmentation over which different models are appropriate. In this specification, population segmentation is used to create separate models to be included in a larger suite. The Applicability scorecards help the specify the correct model to use and deliver. A **Segmentation Variable**, in this context of this specification, will be used for the following:

[28]CJW Note: I am not necessarily suggesting it, but since some modelers readily use polynomials for non-linear effects, another example could be `PrinCurBal_Sq=PrinCurBal^2`.

- A binary or low-order multi-nomial indicator that for some population segmentation that calls for possible differing sets of other variable treatments across segments to be fit while other sets of variable treatments may be fit in common across segments. For example, suppose one has population segments A, B, and C, and variables, X and Y, to be fit across segments, variables, S and T, only on segment A, and variables, U and V, only on segment B. Simplistically, one might have something like:

$$\beta_X X + \beta_Y Y + 1_A(\beta_S S + \beta_T T) + 1_B(\beta_U U + \beta_V V)$$

- Suppose W is to be used as a segmentation variable. Then itself has a treatment which gives artificials $W_0, W_1, \ldots, W_N$, in the regular manner and often, $N = 1$. These indicators over the input space of W can also be viewed as segmenting the Intercept variable, or analogously, the coefficients associated with $W_0, W_1, \ldots, W_N$ are the intercepts associated with the possible $N+1$ segments of the population. In the usual way of eliminating co-linearities, if an intercept variable is used, at least one of W_i will need to be dropped from fitting. However, the dropped indicators still exist (in spirit at least) in the modeling input data set.

- Any other variable might have a *Segmented By* designation for W, in which case, the corresponding artificial, W_i, will also need to be noted. Extending the example above, suppose one as the variable:

$$W = 0 * 1_C + 1 * 1_A + 2 * 1_B$$

  Then, W could have a simple CategoricalNumeric treatment with critical values 1 and 2. The 0 value not listed as a critical value would be treated as un-mapped or missing and $W_0 = 1$. Therefore, the net effect of $1_A(\beta_S S + \beta_T T)$ can be given to variables S and T if they include the tagged information *SegmentedBy W, artificial index 1*. Variables U and V would be correspondingly tagged with *SegmentedBy W, artificial index 2*.

  Why not just tag a variable with *SegmentedBy W=<some value>*? Variables are often added, updated, and/or subtracted later. Furthermore, simple binaries are usually used, with an effective treatment of *CategoricalNumeric with Critical Value 1*, which can make the segmentation pattern *artificial index 1* the default.

- Generally speaking, variables do not need to be specified in any given order, but during model development and implementation, segmentation variables should be processed first, but possibly following their own SegmentedBy variables. There are XSLT/XPATH ways to pull out these variables first when processing.

- Once a variable, say S, has been tagged with something like *SegmentedBy W, artificial index 1*, then all artificials for S, including its missing indicator, must be 0 unless $W_1 = 1$. So, if W has itself been tagged with something like *SegmentedBy Z, artificial index 3*, then $S_i = 0 \; \forall \; i$ at least whenever $W_1 \neq 1$ which happens at least whenever $Z_3 \neq 1$.

5.10 A Detailed Example

[Under construction] Consider a simple *made-up* two variable model for competing risks. One Example could be:

```
<?xml version="1.0"?>
<!-- Copyright 2019, 2020, 2021, 2022, Wypasek Data Science, Inc.
    Author: Christian Wypasek (CJW)
-->
<Projects xmlns:xi="http://www.w3.org/2001/XInclude">
    <Project Name="Example1">
        <Documentation>
            <Version>
                <Major>1</Major>
                <Minor>0</Minor>
                <Patch>2</Patch>
            </Version>
            <Date>2020-02-02</Date>
            <Text><par>Simple example model of WDS-ModelSpec.</par>
            </Text>
        </Documentation>
        <Dictionary><xi:include parse="xml" href="[Path to projecct space global source reference]"
            xpointer="FieldMDs"/></Dictionary>
        <Models>
            <Model Name="ExampleModel1"
                Handle="CRM1"
                >
                <SignatureMDs>
```

```
                    <SignatureMD Name="Input">
                    </SignatureMD>
                    <SignatureMD Name="Output">
                    </SignatureMD>
                </SignatureMDs>
                <ModelDirectives>
                    <!--CompRiskSurv models always include:
                            an applicability score, one value per subject
                            a static score, one value per subject and each response
                            a time varying conal ordering score which provides robust scoers across age
                                    and calendar effects, one value per subject, each response,
                                    and each (age, time) in requested time frame
                            a time varying score which provides a robust score across just age effects,
                                    one value per subject, each response, and each (age, time) in
                                    requested time frame
                            a baseline score, one value per
                                    subject, and age in requested time frame
                    -->
                    <Type>CompRiskSurv</Type>
                    <Responses>
                        <Response>EC1</Response>
                        <Response>EC2</Response>
                    </Responses>
                </ModelDirectives>
                <ComponentModels>
                    <!--All models have an applicability score, used to determine at run time which
                        model is appropriate.  The return value of an applicablity score is
                        0 (applicable) or negative.
                    -->
                    <ComponentModel Name="Applicability" Handle="App">
                    </ComponentModel>
                    <ComponentModel Response="Static">
                        <Variables>
                            <Variable Name="FICO" Handle="X">
                                <Treatment>Hats</Treatment>
                                <CleanLimits>
                                    <LeftLimit>150</LeftLimit>
                                    <RightLimit>950</RightLimit>
                                </CleanLimits>
                                <CriticalValues>
                                    <CriticalValue> 600 </CriticalValue>
                                    <CriticalValue> 650 </CriticalValue>
                                    <CriticalValue> 700 </CriticalValue>
                                </CriticalValues>
                                <CoefficientSets>
                                    <CoefficientSet Response="EC1">
                                        <Coefficient Position="0"> 0 </Coefficient>
                                        <Coefficient Position="1"> 0 </Coefficient>
                                        <Coefficient Position="2"> 0 </Coefficient>
                                        <Coefficient Position="3"> 0 </Coefficient>
                                    </CoefficientSet>
                                    <CoefficientSet Response="EC2">
                                        <Coefficient Position="0"> 0 </Coefficient>
                                        <Coefficient Position="1"> 0 </Coefficient>
                                        <Coefficient Position="2"> 0 </Coefficient>
                                        <Coefficient Position="3"> 0 </Coefficient>
                                    </CoefficientSet>
                                </CoefficientSets>
                            </Variable>
                        </Variables>
                    </ComponentModel>
                    <ComponentModel Response="TVC">
                    </ComponentModel>
                    <ComponentModel Response="TV">
                    </ComponentModel>
                    <ComponentModel Response="Baseline">
                    </ComponentModel>
                </ComponentModels>
            </Model>
        </Models>
    </Project>
</Projects>
```

6 CJW Notes: Why this or that, or why not use something else?

6.1 *Why XML? Why not just a config file like "variable packets"?*

A config file always seems like a good idea at the time, until one needs one more feature.

A style such as "variable packets" requires a specialized parser. XML also requires a parser, but parsers are provided in effectively every programming language. Here, we are also exploiting the topology concept of the coffee cup and the doughnut. If the XML is well defined, it is mapping all of the critical features of a model in structured way that can be translated into other forms. Therefore, the XML data could be translated into a variable packet form easily, even with an XSLT transform, but going from a variable packet form to others requires additional code that only grows more convoluted with time.

The section 6.4 also illustrates how other standardized forms might be helpful, but can obfuscate simple constructs.

6.2 *Why not just use PMML?*

As my use of this type of spec has grown, I have periodically investigated what was publicly available. Throughout much of this time, PMML was just inadequate and did not provide the flexibility required. That being said, the maven repo standardized implementation of PMML could be an ideal target, especially for inclusion in Java implementations.

PMML does facilitate the use of piece-wise linear transformations, or Hat-like outcomes, through its normalization transformations[29] Consider the basic outcome of Hats based component such as in Figure 5.6. The final outcome is the from the treatment of one variable, but as of PMML 4.3, for a regression model, either each artificial would need to be constructed outside the model so that it could be used in a vanilla fashion in the regression table, or, the final marginal score could be created as one transformation at which point it could be scaled to use just one coefficient inside the regression table (or maybe with a coefficient of 1).

Starting only with PMML 4.1 (2012), a ScoreCard model type was made available. This model type does permit equations of lines within a given interval for a variable treatment. This is effectively the treatment that the Experian Strategy Manager scorecard had available back in 1999. So, if the knots and coefficients from a Hats based treatment are available, the information can be translated into the necessary PMML format, but this is not likely the way in which one would build the model.

6.3 *Why not something else, like a web-interface, mark-up, or mark-down?*

At somepoint, one is a encapsulating information to be used as instructions, either for building or evaluating models. As discussed in section 2, the **coffee-cup-and-dougnut** topological approach essentially treats similar sets of data the same as long as there is an invertible mapping between formats. Therefore, if the WDS-ModelSpec is complete enough, it should be easily mappable to some other format just as complete and visa-versa.

6.4 Comparison of a simple variable's treatment between WDS Model Spec and PMML

The following example includes subset of what would be required to specify a simple Hats treatment for a variable in the spec under discussion and PMML's Regression model. Note: this could be handled differently in the PMML Scorecard model.

There is a minimal set of information required to perform this process as in VariablePackets:

```
Name: X
Treatment for X: Hats
CriticalValues for X: 0 1 3
Coefficients for X: 0.0 1.25 1.75 2.0
```

[29] See `http://dmg.org/pmml/v4-3/Transformations.html`.

More verbose with markup is the XML version of the WDS Model Spec:

```
<Variable Name="X">
    <Treatment>Hats</Treatment>
    <CriticalValues> 0 1 3 </CriticalValues>
    <Coefficients> 0.0 1.25 1.75 2.0 <Coefficients>
<Variable>
```

Note: A more susinct form XML form for a list of critical values would be:

```
    <CriticalValues><CriticalValue>0</CriticalValue><CriticalValue>1</CriticalValue><CriticalValue>3</CriticalValue></C
<Variable>
```

This does permit easier and cleaner handling in XSLT.

There are several ways to score the model in PMML. The closest to the model fitting process would be to create the Hat functions as variables, as with PMML-4.3 Transformations:

```
<LocalTransformations>
    <DerivedField name="X_1" dataType="double" optype="continuous">
        <NormContinuous field="X">
            <LinearNorm orig="0" norm="-1">
            <LinearNorm orig="1" norm="0">
        </NormContinuous>
    </DerivedField>
    <DerivedField name="X_2" dataType="double" optype="continuous">
        <NormContinuous field="X">
            <LinearNorm orig="1" norm="-1">
            <LinearNorm orig="3" norm="-0.5">
        </NormContinuous>
    </DerivedField>
    <DerivedField name="X_3" dataType="double" optype="continuous">
        <NormContinuous field="X">
            <LinearNorm orig="0" norm="1">
            <LinearNorm orig="1" norm="-.5">
        </NormContinuous>
    </DerivedField>
</LocalTransformations
```

Each of the DerivedFields then receives a separate location in the regression coefficient table. To match the scoring output, the combined final effect could be created as one transformation which would receive only one location in the regression table:

```
TODO: This is part of the problem, the PMML transformation spec is really poor
and not easy to figure out.
```

7 Implementation and Details

7.1 CJW-Notes: TODOs

The formal XSD for the WDS Model Spec is under development. However, whereas the XSD is a formal document, in practice, development of the specification is done through actual use and working throug the details. Examples done for clients can never be shared, but working prototypes will be open sourced when available.

7.2 WDS Github

As draft portions of the spec and tools are open-sourced, they will be made available at `https://github.com/wdatasci`.

7.2.1 WDS-ModelSpec

At `https://github.com/wdatasci/WDS-ModelSpec`, a collection of utilities is available. Many items have parallel implementations in C/C++, Lua, Python, SAS, and others.

7.2.2 CJW-Notes: PMML Evaluation

PMML does have a standard Java-based implementation, `https://github.com/jpmml/jpmml-evaluator`, and there are several other commercial products that offer PMML evaluators. The Java based implementation is obj-orifuscated enough to not be easily translatable into other software.

7.2.3 WDS-JniPMML-XLL

As a project to test PMML implementations in Excel, WDS has open-sourced a JNI-based caller to jpmml-evaluator, available at: `https://github.com/wdatasci/WDS-JniPMML-XLL`. There are several utilities to wrangle data and PMML models in and out of Excel.

Index