Scoring Outside the Box

Nascif Abousalh-Neto, JMP Principal Software Developer, SAS

Daniel Valente, Ph.D., JMP Senior Product Manager, SAS

Introduction

Scoring – the process of using a model created by a data analysis application like JMP to make predictions on new data – has been called the unglamorous workhorse of data mining. Like a dark yin to the bright yang of predictive modeling, scoring plays a fundamental role in the ability to implement a model. Scoring requires that the model is first adapted so that it can run where the new data is produced or stored. This process is usually a time-consuming and error-prone endeavor. This paper describes how the new score code generation features in JMP 13 can assist you in extending the reach of your models while minimizing the work required to adapt them.

The case for model deployment

Every data modeling project is a journey.

Every journey is unique, from the original business requirements, particularities of the data, and how the insights they produce will be consumed. But in the end, all projects follow common patterns.

The data mining industry has proposed a process model that captures these patterns. The *Cross Industry Standard Process for Data Mining* (CRISP-DM) identifies six major phases of a data mining (that is, modeling) project and the most frequent dependencies between them. This is shown in Figure 1.

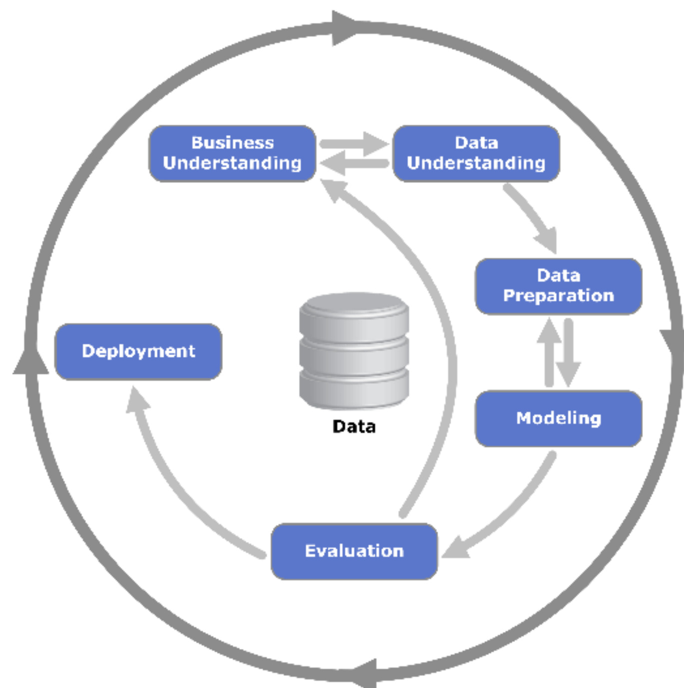

Figure 1: CRISP-DM process diagram (source)

You can think of the CRISP-DM diagram as describing a journey that starts in the business world, takes you inside of an analytics box where data is transformed into knowledge (through understanding, preparation, modeling and evaluation), and then allows you to emerge with newly acquired knowledge that can be put to work to improve the business.

Starting Model Deployment

> *"What people think of as the moment of discovery is really the discovery of the question".*
>
> Jonas Salk

The journey starts outside the analytics box in the Business Understanding phase. After appropriate questions are defined, the analyst can form a plan (a roadmap for the journey ahead), gather data, and start with the insight-finding phases of the project. It is time to step inside the box.

This is the point where JMP comes into play. As a product designed to help customers succeed in their data modeling journeys, JMP has features that are fine-tuned to the specific needs of each phase in the data analysis process:

- *Data Understanding* is achieved through data visualization and descriptive analytics by using the Graph Builder, Distribution, and Explore Outliers.
- *Data Preparation* can rely on Recode, binning, Query Builder, and Formula Editor.
- *Modeling* choices are extensive, with a large number of prediction and specialized models, from bootstrap forests to neural networks.
- *Evaluation* can be performed with the Profiler and Model Comparison platforms.

Most of the time, finding the most useful model is the analyst's end goal. Other times, the goal is to achieve a better understanding of the data and have it captured in a research paper or in a new set of policies or standards. Or perhaps the model will be applied only to other JMP tables—the transition to "work mode" doesn't require the analyst to leave JMP. In general, if the data analyst is also the final consumer of the knowledge captured in the model, the journey ends inside the box. There is no additional boundary to cross.

However, that is not always the case. Business often demands that the journey continues into new territory. The knowledge acquired, in the form of a fitted model, must make its way outside of the analytics box to generate business value; it has to be deployed into production where it can be used in a process called scoring:

> *"The process of using a model to make predictions about behavior that has yet to happen is called "scoring." The output of the model, the prediction, is called a score. Scores can take just about any form, from numbers to strings to entire data structures, but the most common scores are numbers (for example, the probability of responding to a particular promotional offer)."* (source)

After it is deployed, a model will be executed thousands, perhaps millions of times over new, incoming information to perform a myriad of tasks: monitoring new manufacturing data, assessing new risk in a fleet of assets, estimating the price for a good or service about to hit the market, predicting the chances of success of a marketing campaign, or indicating the chances of failure of a component of a system.

The challenges of model deployment

Moving outside of the analytics box has a cost. To be used in the production world, a model has to be translated into a programming language suitable for deployment. This requirement poses a unique set of challenges.

First, there is the translation itself. Manually translating a model into a different programming language is time-consuming because large models may require hundreds, even thousands of lines of code. More importantly, manual translation is very error prone in a way that is quite hard to debug. And at the end of the day:

> *"Your model is not what the data scientists design, it's what the engineers build."* (source)

In addition, translation to what? Different business needs require different IT infrastructures, which in turn dictate different choices of programming language for deployment. When we asked our customers "What do you do with your models after you build them in JMP" the answers reflected that diversity:

- To some customers, deployment into production meant simply applying a model to a new table—they didn't need to get out of JMP but wanted a way to manage models other than copying columns from table to table;
- Others wanted a way to convert the model into a command-line, stand-alone application that is suitable for integration with automated data pipelines, such as those implemented by DAKOTA, Pipeline Pilot, and AspenONE®;
- Many pointed out that 99% of their internal clients use Excel, so they wanted a solution that integrates nicely with it;
- A few mentioned data warehouses and BI systems that require some sort of in-database scoring;
- Finally, we heard about interactive use cases that would be best served by web or mobile applications.

Simplifying model deployment

Our customers indicated that they need to be able to convert a JMP model into a different programing language for deployment. Moreover, the diverse landscape of deployment scenarios implies too many requirements to allow the selection of a single target language.

With this feedback in mind, the JMP R&D team set out to find ways to ease the transition of JMP-generated models for use within and outside of JMP. To that effect, the 13.0 release of JMP Pro introduces the *Formula Depot*, a stand-alone container for formula column scripts in general, and prediction models in particular, with code generation capabilities. The Formula Depot can be launched directly from the **Analyze** > **Predictive Modeling** menu (Figure 2).

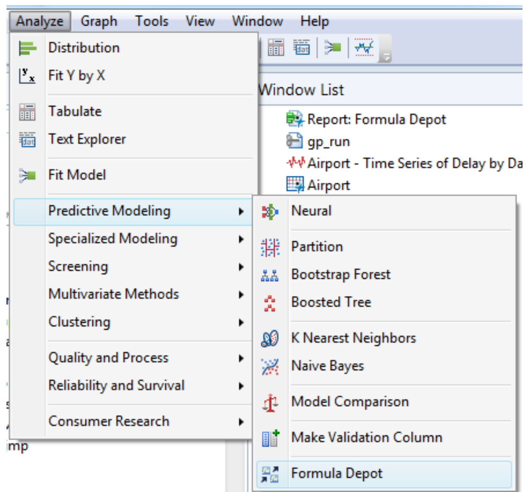

Figure 2: Formula Depot menu location

The Formula Depot can also be launched indirectly through the **Publish** commands, made available in modeling platforms like Neural (Figure 3).

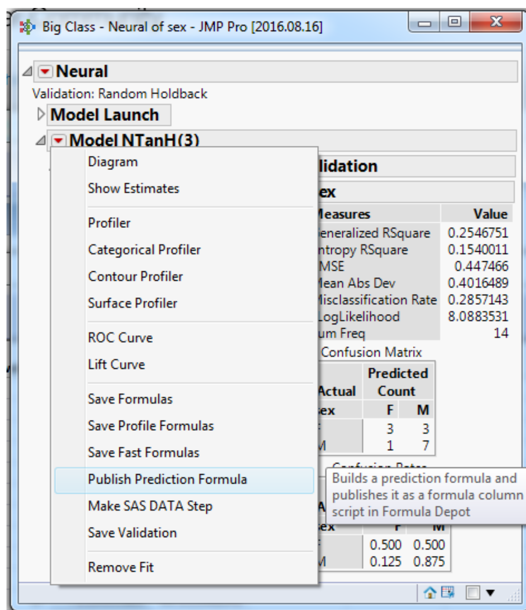

Figure 3: Publishing a model to the Formula Depot

The Formula Depot can store and manage formula column scripts and models independently from their original data tables, allowing for the easy transfer of models to new tables (Figure 4). Stored models can be passed to the Model Comparison and Profiler platforms directly from the Formula Depot interface.

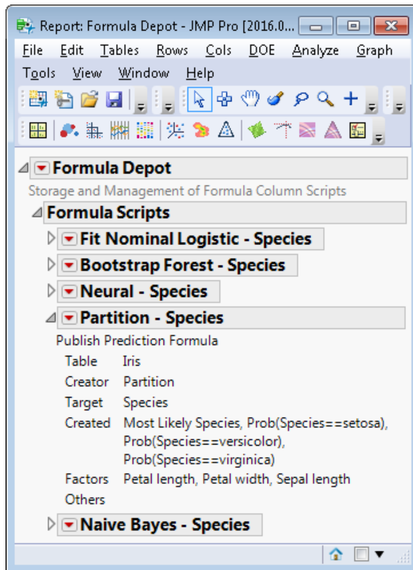

Figure 4: Models stored in the Formula Depot

The Formula Depot provides options to generate score code in different programming languages (Figure 5). Previous versions of JMP supported code generation for **SQL** and **SAS** (DS2) in a few platforms. JMP Pro 13.0 unifies these features under the Formula Depot and adds scoring code generation for the **C**, **JavaScript**, and **Python** programming languages.

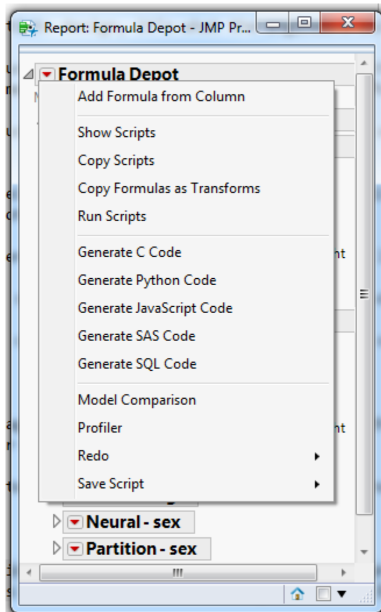

Figure 5: Formula Depot code generation options

Calling one of the **Generate** methods opens an editor window with the generated score code (Figure 6). The user can then save the code for use in the deployment process. When called at the Formula Depot level, the score code combines all selected models; when called from the model level, it generates code only for that model.

Models translated to the *C language* can be compiled into lightweight, performant libraries and applications. C is an ideal language to create streamlined command-line applications for integration into automated data pipelines.

JavaScript translation allows models to be deployed as part of rich-client web applications, scoring data right in a browser. JavaScript models can also be deployed on a server-side application built on Node.js, or as a mobile app based on React Native.

Thanks to its versatility and popularity in Academia, *Python* is challenging R as the open source language of choice for data analysis projects. With its rich library ecosystem, Python allows JMP models to be deployed in a variety of architectures, from REST-based web services built on Flask, to serverless frameworks like AWS Lambda, to iOS apps developed with Pythonista, and of course as traditional desktop applications. JMP models translated to Python can be evaluated and shared as live documents using Jupyter notebooks.

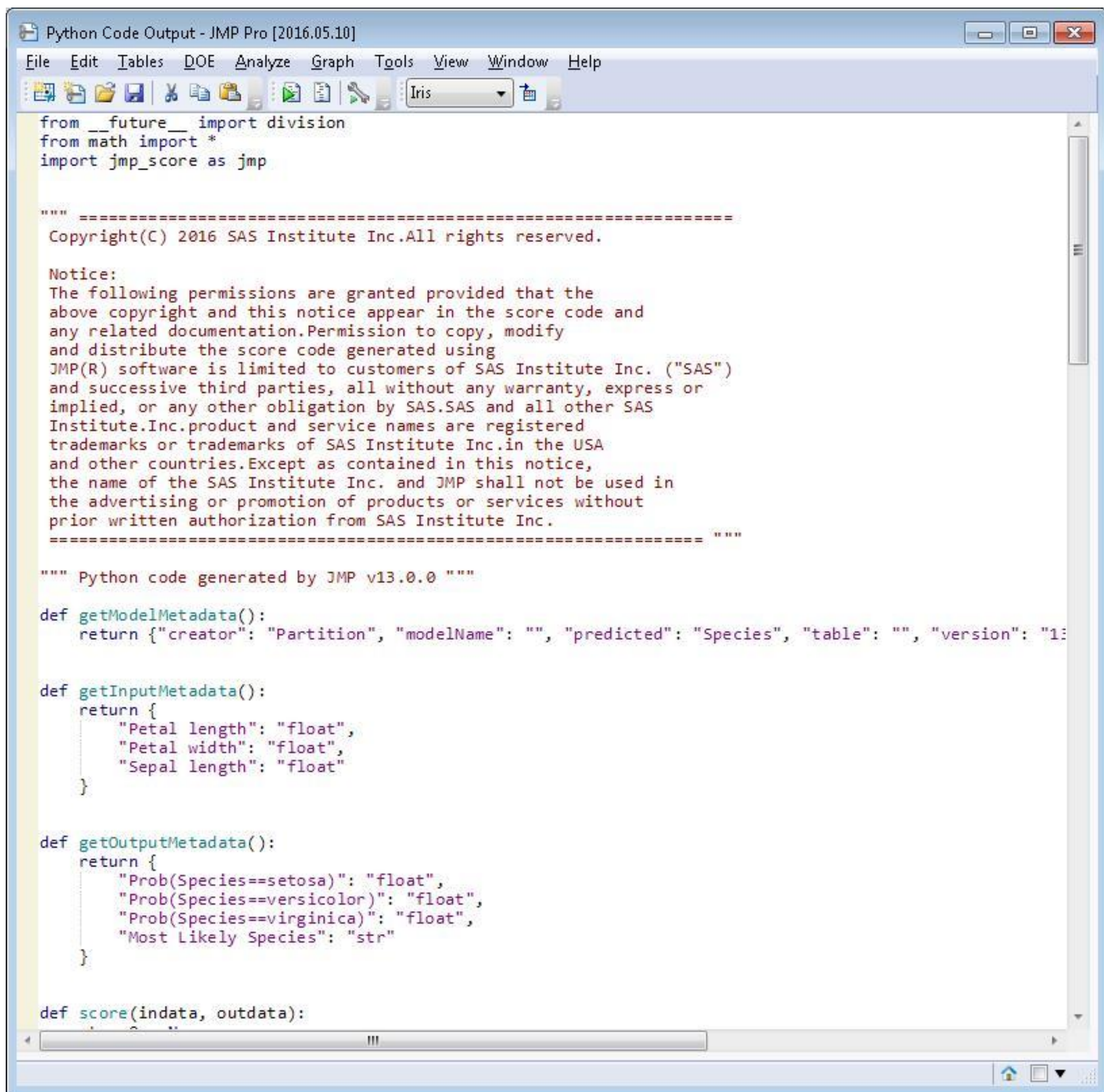

Figure 6: Python code generated from a model in the Formula Depot

Python is also being embraced by major data companies as an extension language. The PL/Python procedural language extension allows Python JMP models to execute inside databases such as PostgreSQL and Massively Parallel Processing (MPP) servers such as Greenplum. Deployment to large-scale data engines, for example Spark and H2O, allows Python JMP models to score large datasets across multiple computational nodes running in parallel. IoT and streaming engines that embraced Python, such as GE Predix and AWS Kinesis, allow JMP models to be applied to new observations as they arrive in the system, in real time.

Scoring support files

Models converted to C, JavaScript, and Python require additional files provided by JMP in order to be compiled or executed. These files are available in the JMP Pro 13.0 install location under the `Scoring` directory (Figure 7).

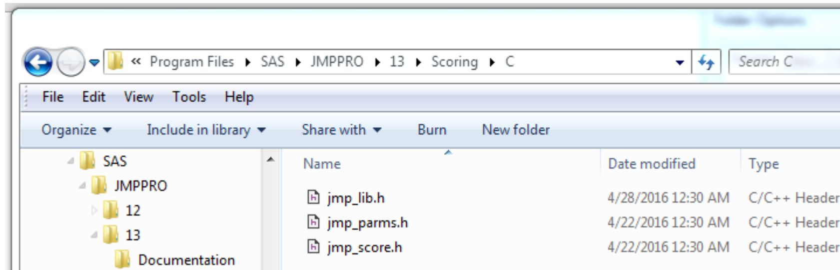

Figure 7: Location of scoring support files

The deployment process responsible for building or moving the generated model into production will likely include a step where the support files of the appropriate language are copied into a staging area or referenced by an include directory configuration.

Streamlining integration: Modules and Metadata

Automated translation to a production programming language is an important step for deployment, but it is not the only one. Additional steps, such as mapping to framework-specific interfaces or copying to a particular server, must be implemented (typically by the production engineer or IT department) to complete the model integration with its new environment.

It is important to streamline these steps. As the external cycle in the CRISP-DM diagram indicates, models have a finite life expectancy and are expected to be replaced multiple times over the course of a long-running data modeling project. Therefore, it is desirable to automate not just the model translation but also the steps required to integrate a new model with an existing framework.

To assist with the integration step, models generated in the languages added by JMP 13.0 are implemented as modules. **Modularization** allows these models to be imported into or referred by a framework-specific adapter code without code modification. The model to be loaded can even be specified dynamically, allowing, for example, an application to switch between a champion and a challenger candidate model at run time.

Module support is a regular feature of the Python language that JMP models leverage. JMP C language models declare public methods with Dynamic Link Library (DLL) storage-class attributes. By default, these models use Microsoft-specific extensions but those can be overwritten in the support files described above.

Modularization is not directly supported by the JavaScript language, but it was achieved by following the CommonJS convention. CommonJS modules are natively supported by Node.js server applications. Web applications can be developed using a build tool like Browserify or Webpack.

Thanks to modularization the task of integrating a JMP module into a new application is quite straightforward. For example, in Python:

```
import Iris_logistic as jmp
indata, outdata = # {"Sepal Width": 5.1, ...}, {}
result = jmp.score(indata, outdata)
```

Metadata helper methods included with the scoring code provide run-time information about the models themselves. These methods support the mapping of the model input and output parameters to the input and output production data sources that will be used as arguments. The methods provide information about the model parameters: how many they are, their names (or positions, depending on the language), their types, and (also depending on the language) how much memory needs to be allocated to store their values.

Metadata methods allow models to be integrated in a data-driven, generic way. They also allow the external application to ask questions about the model itself such as which variable it predicts or which version of JMP was used to generate it. For example, in Python:

```
def getModelMetadata():
  return {"creator": "Boosted Tree", "modelName": "", "predicted": "sex",
"table": "Big Class", "version": "14.0.0", "timestamp": "2016-08-
02T21:03:46Z"}

def getInputMetadata():
    return {
        "height": "float",
        "age": "float",
        "weight": "float"
  }

def getOutputMetadata():
    return {
        "Prob(sex==M)": "float",
        "Prob(sex==F)": "float",
        "Most Likely sex": "str"
  }
```

Conclusion

Deployment of models to production is crucial but also challenging. With its new Formula Depot and code generation features, JMP hopes to facilitate this transition so that your models can reach a larger audience and improve their (and your) value to the business.

JMP will always be the analytics powerhouse you have come to love and appreciate. And if your data journey ever calls for a return to the production world, JMP will be ready to help you score outside the box.

Appendix: Model Support List

JMP models are expressed in JMP Scripting Language (JSL), a language with an incredibly rich set of operators, from simple arithmetic operators like `sum()` to high-level operators like `K Nearest Rows()`. Because some of these operators cannot be directly translated into other programming languages, some JMP models are not completely supported with score generation. However, JMP still generates code for such models.

The JMP scoring code includes placeholders that indicate function calls where additional code is necessary for model implementation. In addition, different languages have different levels of support for constructs like variable declaration, data structures like matrices and vectors, and control flow logic. The consequence is that not all combinations of models and languages are supported in JMP code generation.

The following list indicates, for all of the different models that can be published to the Formula Depot, which languages support a complete translation. This list was compiled for JMP, version 13.0.

Platform/Publish command	**Supported languages**
Analyze->Fit Model	
• Standard Least Squares	
○ Prediction Formula	C, Python, JavaScript, SAS, SQL
○ Prediction Formula (REML)	C, Python, JavaScript, SAS, SQL
○ Parameterized Formula	C, Python, JavaScript, SAS
○ Parameterized Formula (REML)	C, Python, JavaScript, SAS
○ Conditional Formula	C, Python, JavaScript, SAS, SQL
○ Conditional Formula (REML)	C, Python, JavaScript, SAS, SQL
○ Standard Error Formula	-
○ Standard Error Formula (REML)	-
○ Mean Confid Limit Formula	-
○ Indiv Confid Limit Formula	-
• Generalized Regression	
○ Prediction Formula (normal)	C, Python, JavaScript, SAS, SQL
○ Prediction Formula (binomial)	C, Python, JavaScript, SAS
• Nominal Logistic	
○ Probability Formulas	C, Python, JavaScript, SAS
• Ordinal Logistic	
○ Probability Formulas	C, Python, JavaScript, SAS
Analyze->Predictive Modeling	
• Partition (Decision Tree)	
○ Prediction Formula (> numeric)	C, Python, JavaScript, SAS, SQL
○ Prediction Formula (> categorical)	C, Python, JavaScript, SAS
○ Tolerant Prediction Formula	SAS, SQL
○ Difference Formula	C, Python, JavaScript, SAS, SQL
• Neural	

    - Prediction Formula (> numeric)   C, Python, JavaScript, SAS, SQL
    - Prediction Formula (> categorical)   C, Python, JavaScript, SAS
    - Prediction Formula (boosted)   C, Python, JavaScript, SAS, SQL
- Bootstrap Forest
    - Prediction Formula (> numeric)   C, Python, JavaScript, SAS, SQL
    - Prediction Formula (> categorical)   C, Python, JavaScript, SAS
    - Tolerant Prediction Formula (> numeric)   SAS, SQL
    - Tolerant Prediction Formula (> categorical)   SAS
- Boosted Tree
    - Prediction Formula (> numeric)   C, Python, JavaScript, SAS, SQL
    - Prediction Formula (> categorical)   C, Python, JavaScript, SAS
    - Tolerant Prediction Formula (> numeric)   SAS, SQL
    - Tolerant Prediction Formula (> categorical)   SAS
- Naive Bayes
    - Probability Formulas   -
- K-Nearest Neighbors
    - Prediction Formula   -

Analyze->Specialized Modeling

- Gaussian Process
    - Prediction Formula   C, Python, JavaScript, SAS, SQL
    - Variance Formula   -

Analyze->Multivariate Methods

- Discriminant
    - Probability Formulas   -
- Partial Least Squares
    - Prediction Formula   C, Python, JavaScript, SAS, SQL
    - Score Formula   C, Python, JavaScript, SAS, SQL
- Principal Components
    - Components Formulas   C, Python, JavaScript, SAS, SQL
    - Components Formulas (wide)   -

Analyze->Clustering

- Latent Class Analysis
    - Probability Formula   C, Python, JavaScript, SAS

Analyze->Consumer Research

- Uplift
    - Prediction Formula   C, Python, JavaScript, SAS
    - Difference Formula   C, Python, JavaScript, SAS, SQL
    - Tolerant Prediction Formula   SAS