Paper SAS4318-2020

Risk Modeling on the Fast-Track: Pythonistas (and Others) Harness the Power of the SAS® Risk Engine

Dave Stonehouse, Joshua Johnstone, and Katherine Taylor, SAS Institute Inc.

ABSTRACT

With the availability of the SAS® Risk Engine on SAS® Viya®, SAS® provides a first-class platform for developers to create and deploy risk models. The SAS Risk Engine is a complete, scalable solution that delivers a comprehensive set of risk models and risk metrics. The SAS Viya version of the engine accelerates development of custom risk applications using Python and other languages. Developers can focus their effort on the specific value-added algorithms related to their business context while leveraging key features from SAS, like risk simulation and stress testing. With the SAS Risk Engine on SAS Viya, the path from designing your model to deploying it in a high-performance environment has never been easier. In this paper, we demonstrate the power of developing risk applications in Python based on the SAS Risk Engine with examples from both market and credit risk.

INTRODUCTION

Developing risk management solutions is a complex task that requires deep domain expertise. Increasingly, it also requires considerable computer programming expertise. The complexity of measuring and managing a firm's risk, coupled with the time pressure for risk-aware decisioning creates substantial difficulty when building risk applications from scratch.

To solve this, SAS introduces the Python package SASRiskPy, which is a toolkit for quickly building and deploying risk applications for a wide variety of business use cases. Powered by the SAS Risk Engine on SAS Viya, SASRiskPy combines the flexibility of open-source development with the reliability and analytics breadth of SAS.

SASRiskPy is a logical, business-concept-driven framework that offers both turn-key risk analytics as well as modular analytics building blocks that you can use to create custom-made applications. You can interleave your favorite open-source libraries, the full SAS® Analytics stack, and the capabilities of the SAS Risk Engine in a single Python program.

With SASRiskPy, risk analysts can focus on business concepts while developing. Applications are scalable, concise and readable, and can be easily shared for collaboration. SASRiskPy is an interface to the SAS Risk Engine, which is an enterprise-class software platform that incorporates performance, governance, and security.

In this paper, we talk about the value of the SAS Risk Engine and the power of calling the SAS Risk Engine from an open-source interface. We illustrate the ease of working with SASRiskPy through two simple examples.

WHY YOU WANT A (GREAT) RISK ENGINE

A RISK ENGINE?

If you are a risk analyst, whether in a specialized risk area like market or credit risk or in a centralized risk or quant team, you are asked to develop systems that provide a view into

the amount and the sources of risk. Classic examples include metrics like value at risk (VaR) or expected credit loss (ECL).

When creating a system like this, the analyst and development team face these common challenges:

Challenge	Description
Time to value	Projects often have tight timelines that are driven by regulatory deadlines.
Performance and scalability	The production system must produce results very shortly after the period ends. The results can also get quite large.
Maintenance	At project's end, there is a lot of code to maintain and probably enhance, but there might be turnover in the team.
Many models	The number of models for needed for risk factors, pricing, and so on might be high.
Complex dependencies	Dependencies might include collateral, margin accounts, thresholds, rebalancing, and so on.

Table 1. Challenges for Risk Analysis Projects

A risk engine is a software solution that addresses these challenges and more. It is an accelerator that can be used to solve a variety of risk management problems.

TURBO SPEED – SAS RISK ENGINE ON SAS VIYA

For more than 20 years, risk analysts at banks, insurance and energy companies have used risk engines created by SAS. With the SAS Risk Engine for SAS Viya, users can get faster time to value with the flexibility to work in their language of choice: Python, SAS, R, and others.

Additional benefits of the SAS Risk Engine include:

- a scalable platform to take advantage of grid computing
- built-in analytics for a variety of risk use cases, including market risk, credit risk, counterparty credit risk, liquidity risk, asset and liability management, energy risk, insurance risk, and capital optimization
- integration with the SAS® Analytics stack and visual exploration tools such as SAS® Visual Analytics
- a seamless handoff between the model development team and the production implementation team

SAS RISK ENGINE OBJECTS

The basic structure of the SAS Risk Engine is a combination of objects and analytics. Objects are inputs into analytics. You provide objects and the SAS Risk Engine provides analytics. You can also write custom analytics for the SAS Risk Engine to run.

For example, you can use a historical market data object as an input to a simulation analytic to produce many simulated market outcomes. You can then use the simulated market output as an object, use it with a portfolio object, and use these inputs to a risk analysis analytic to produce the simulated portfolio values for Value at Risk.

The ability to chain together these objects and analytics as needed is very powerful. It saves a lot of time in developing custom risk applications.

The following two tables present some of the commonly used objects and analytics of the SAS Risk Engine.

Object	Description
Portfolio	The set of instruments that a company has, such as loans, trades, insurance contracts or energy holdings.
Market States	The current risk factor values for a moment in time. These values can include the current state and future states as calculated by a simulation.
MethodLib	A set of methods or functions used for a variety of tasks.
Values	The set of instruments in the portfolio that are evaluated at each of the provided market states or scenarios.
Results	Requests for risk metrics at any level of portfolio aggregation.
Scenarios	States for a set of risk factors in the future, often chosen to be an adverse case.
Counterparties	Entities to which you have a credit exposure.
Scores	The set of counterparties evaluated at each of the provided market states or scenarios.
SessionContext	Information about the connection to the SAS server.

Table 2. Risk Engine Objects in the Examples

Analytics	**Description**
Market State Generation	Run simulations to produce a set of risk factors for risk analysis.
Counterparty Scoring	Run scoring methods against counterparties to produce a set of score results.
Portfolio Evaluation	Run evaluation methods against the portfolio to produce values at market states or scenarios.
Querying Results	Aggregate values and compute risk metrics.

Table 3. Risk Engine Analytics in the Examples

Now let us move on to the examples.

MARKET RISK EXAMPLE

Please see the appendix of this paper for a link to the full code for this example. The code snippets in these examples are for illustrative purposes.

The goal of this example is to calculate market risk metrics such as VaR and Expected Shortfall (ES) of a portfolio of bonds. We start from nothing but input data (consisting of a portfolio and some market data), some simple bond pricing methods, and a connection to our SAS server. We perform a Monte Carlo simulation to generate market states for the risk factors in the historical data set, price the bonds in the portfolio at each of the generated market states, then query the results to get a Pandas DataFrame of risk metrics.

This example depends on the following Python packages: riskpy, pandas (as ‘pd’), SWAT, and datetime (as ‘dt’).

GENERATING MARKET STATES

Our first step is to create a MarketData object to describe our economic data. This object will reference Excel files for current and historical market data:

```
market_data = riskpy.MarketData(
                current    = 'data/current.xlsx',
                historical = 'data/history.xlsx')
```

Now, we create an instance of a MarketStates object to set up for market state simulation. Here, we pass in the MarketData object we just created, an as-of date for the simulation, and the number of desired replications. We also pass in a session context, which describes our connection to the SAS server.

```
states = riskpy.MarketStates(
                session_context = conn,
                market_data     = market_data
                as_of_date      = dt.datetime(2020,3,2),
                num_draws       = 100)
```

We then call states.generate(), and the SAS Risk Engine does the rest:

```
states.generate()
```

With just a few lines of code, we have run a full Monte Carlo simulation, and now our states are contained in the states object. Under the covers, the SAS Risk Engine calculates covariances between the risk factors from the historical data and simulates market states for those risk factors. All of these actions are distributed and performed in-memory using the power of the SAS Risk Engine. Later, we will attach these market states to a Values object to price the portfolio.

SETTING UP FOR THE RISK ANALYSIS

We need a few essential pieces to run our analysis, the first of which is a portfolio. Here, we create a Portfolio object, reading in positions data as a Pandas DataFrame:

```
portfolio = riskpy.Portfolio(data = 'data/portfolio.xlsx')
```

Then, we create a MethodLib object, which is a file containing the bond pricing functions:

```
methods = riskpy.MethodLib(method_code = 'methods/bond_methods.sas')
```

RUNNING THE RISK ANALYSIS

Now, we have all the pieces and are ready to price the portfolio at the market states. The Values object connects the various objects that we have created along the way: Portfolio, MarketStates, and MethodLib:

```
values = riskpy.Values(
        session_context  = conn,
        portfolio        = portfolio,
        market_states    = states,
        method_lib       = methods)
```

By calling evaluate() on the Values object, we tell the SAS Risk Engine to price the portfolio at the current market state and the simulated market states:

```
values.evaluate()
```

The resulting prices are now associated with our Values object.

VIEWING RESULTS

Finally, we would like to compute various risk metrics for our portfolio. We create a Results object and attach the Values object to it, then simply call results.query() to return a DataFrame containing VaR and ES for the aggregated portfolio.

```
results = riskpy.Results(values=values, alpha=0.05)
stats_05_df = results.query()[['VAR', 'ES']]
```

Suppose we would like to see our statistics at a different confidence level. It is easy to do so.

```
results.alpha = 0.01
stats_01_df = results.query()[['VAR', 'ES']]
```

The power of the SAS Risk Engine allows us to reaggregate our portfolio to generate risk metrics without having to rerun the entire analysis.

We can use well-known plot packages for Python to visualize the results of our risk analysis. Figure 1, built with the Seaborn and Matplotlib packages, shows VaR and ES on a density plot of losses. Please see the full example and documentation for additional detail.

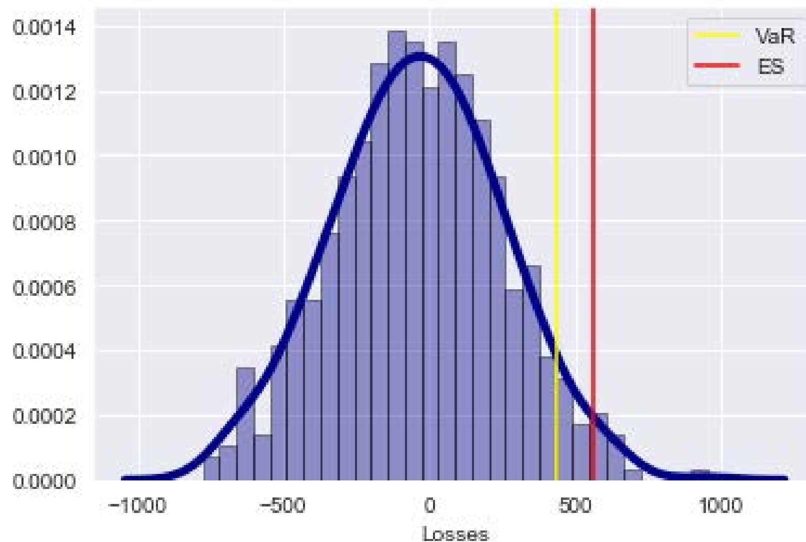

Figure 1: VaR and ES Superimposed on a Histogram and Density Plot of Losses.

CREDIT RISK EXAMPLE

The SAS Risk Engine is also used for a variety of credit risk applications. For various reasons (including regulatory), institutions need to determine how much money they expect to lose from customers who default. The goal of this business example is to illustrate how to calculate credit losses over time for a couple of macroeconomic scenarios.

One calculation is the probability of default, which can be a complex function, often repeated for different exposures to the same counterparty or for exposures of a similar nature.

Our input data includes our portfolio, counterparties, and scenarios. We will score the counterparties to get the probability of default for each group of loans, evaluate the loans for each scenario, and then query the results at an aggregate level.

Figure 2 illustrates the system used to accomplish this goal.

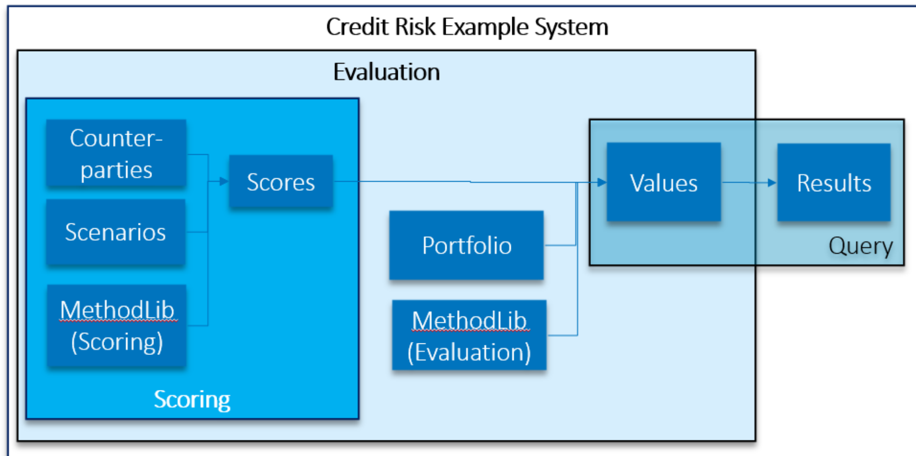

Figure 2. Credit Risk Example System

LOAD SCENARIOS

Our first step is to load our scenarios from a spreadsheet and create a Scenarios object, associating it with our current market data:

```
mkt_data = riskpy.MarketData(
    current      = pd.DataFrame(data={'uerate': 6.0}, index=[0]),
    risk_factors = ['uerate'])
my_scens = riskpy.Scenarios(
    name        = "my_scens",
    market_data = mkt_data,
    data        = os.path.join(data_dir, 'uerate_scenario.xlsx'))
```

The scenarios input defines two macroeconomic scenarios for unemployment.

SCORING

Now we solve for probability of default. We load the counterparty data from Excel and associate it with a Counterparties object:

```
loan_groups = riskpy.Counterparties(data=pd.read_excel(
   path("datasources","CreditRisk",'uerate_cpty.xlsx')))
```

A scoring method calculates scoring values (in our case, probability of default) for a counterparty for a given scenario. We create a MethodLib object that is associated with the location of the method code and map the method to the type in our counterparty data:

```
scoring_methods = riskpy.MethodLib(
   method_code=path("methods","CreditRisk",'score_uerate.sas'))
```

Now we need a Scores object to associate the three objects needed for scoring:

```
my_scores = riskpy.Scores(counterparties=loan_groups,
                          scenarios=scens,
                          method_lib=scoring_methods)
```

We are finished with configuration for scoring. Now we tell the server to generate our probabilities of default:

```
my_scores.generate(session_context=conn)
```

Our probabilities of default are now associated with our Scores object.

SETTING UP FOR THE RISK ANALYSIS

The next two steps are similar to the Market Risk example This code reads our portfolio data from Excel and associates it with a Portfolio object:

```
portfolio = riskpy.Portfolio(
    data=path("datasources","CreditRisk",'retail_portfolio.xlsx'),
    class_variables = ["region", "cptyid"])
eval_methods = riskpy.MethodLib(
    method_code=path("methods","CreditRisk",'credit_method2.sas'))
```

RUNNING THE RISK ANALYSIS

We now have all the ingredients (Portfolio, Scenarios, MethodLib and Scores) for the SAS Risk Engine to generate values, so we configure them in a Values object and evaluate:

```
 my_values = riskpy.Values(
                          session_context=conn,
                          portfolio=portfolio,
                          output_variables=["Expected_Credit_Loss"],
                          scenarios=my_scens,
                          scores=my_scores,
                          method_lib=eval_methods,
                          mapping={"Retail": "ecl_method"})
my_values.evaluate()
```

In addition to the input objects, we specify the output variable Expected_Credit_Loss to indicate that the evaluation method also computes the expected credit loss.

VIEWING RESULTS

Now that we have values, we want to get to our question of how much we lose in our two scenarios. For this, we create a Results object and query it:

```
results = riskpy.Results(
    session_context=conn,
    values=my_values,
    requests=["_TOP_", ["region"]],
    out_type="values"
)
results_df = results.query().to_frame()
```

The query returns aggregate results for the whole portfolio as well as rolled up values for each region and returns the results in a Pandas DataFrame. Figure 2, built with Matplotlib, shows the cumulative differences in our losses over 3 horizons for each scenario.

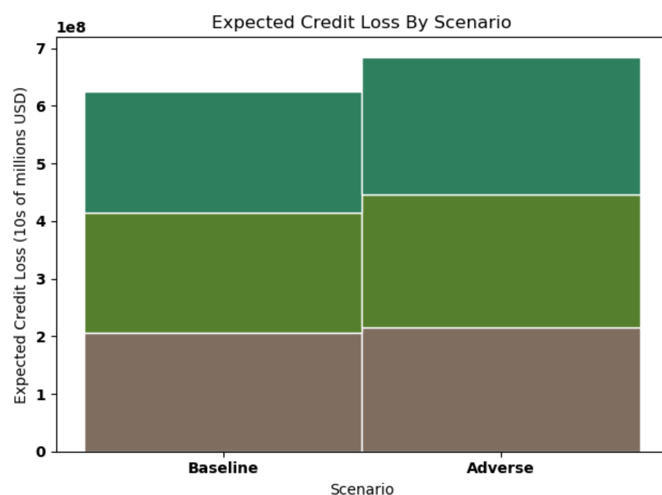

Figure 2: Expected Credit Loss Over 3 Horizons For 2 Scenarios

CONCLUSION

When building risk applications, leveraging the SAS Risk Engine speeds the development time and results in smarter, more performant solutions. Accessing the SAS Risk Engine from an open-source interface, like the SASRiskPy package, extends the power of SAS Analytics to a wider group of developers, bringing together Python coders with SAS programmers and graphical tools users. Faster time-to-value and the benefit of full team collaboration significantly improves the development of risk management applications, enabling the real-time risk management and decisioning that firms need.

APPENDIX

To experiment with using Python and the SAS Risk Engine to create risk modeling systems, please go to the GitHub repository associated with this paper: https://github.com/sascommunities/sas-global-forum-2020/tree/master/papers/4318-2020-Stonehouse

REFERENCES

SAS Institute Inc. 2017. SAS Software / python-swat. Accessed January 29, 2020. Available https://sassoftware.github.io/python-swat/

Foreman C. 2019. "SWAT's it all about? SAS Viya® for Python Users." *Proceedings of the SAS Global Forum 2019 Conference*. Cary, NC: SAS Institute Inc. Available https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2019/3610-2019.pdf

RECOMMENDED READING

SAS Institute Inc. 2020. *SAS Risk Engine 8.2: Programmer's Guide*. Cary, NC: SAS Institute, Inc. Document is available only to SAS Risk Engine customers.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Dave Stonehouse
SAS Institute Inc.
Dave.Stonehouse@sas.com

Joshua Johnstone
SAS Institute Inc.
Joshua.Johnstone@sas.com

Katherine Taylor
SAS Institute Inc.
Katherine.Taylor@sas.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.