Model Validation & Unit Testing

Goals:

- Discuss the process by which we document & validate our models robustly.
- Describe the creation of unit tests

Review: Lifecycle of a Quant Project

- Steps in a quant project:
  - Data Collection
  - Data Cleaning
  - Data Storage
  - Model Implementation
  - Model Validation
  - Presentation of Results
- The last three lectures were dedicated to the programming side of model implementation. Much of the rest of the course will be dedicated to the modeling side of model implementation.
- But before we get there, we have to discuss one remaining critical component of quant projects, **Model Validation**.

What is Model Validation?

- Model validation is the process of checking that our code is correct and that we have implemented the model in the desired way.
- It is designed to catch certain **special / boundary cases** and make sure the system is robust in handling these cases.
- It is also designed to provide us a baseline of what we expect from the model. We can then monitor this and **catch any unintended deviations from the baseline**.
- We should also use model validation to **uncover the limitations of our model** to let us know when to avoid using it in the future.

Why is Model Validation so important?

- Throughout your careers you will hopefully create models that generate alpha in some form, or help drive the business of the firm's that you work for.
- If you do reach this point in your career, it will place a lot of pressure on your model results being correct.
- In order to make sure you are ready, you should be building a set of guidelines and practices that govern how you implement models now, when there is less pressure.
- This isn't just about finding the bugs in your code. A model that runs with bad data or is used in inappropriate circumstances is as dangerous as a model that has bugs.

Why is Model Validation so important?

- Generally speaking, in order for an institution to rely on a model they need to know it is correct.
- They also need to know when it is safe to use and when they should apply a different model.
- They will also most likely want to know that you are not the only one with knowledge of the model.

Model Validation: Regulatory Requirements

- At banks, model validation fulfills a key regulatory requirement.
- Banks must also attempt to quantify their model risk and model validation is an important component of mitigating this.
- As a result, banks have rigid model validation procedures and entire teams dedicated to validating new models.
- At other institutions, the procedures are less rigid but the principals are equally important, and the model risk is still there.

What are the goals of Model Validation?

- **Provide robust documentation for the model**
- **Catch improper model implementation** (i.e. software bugs)
- **Catch improper model use** (i.e. invalid or unexpected parameters)
- **Explicitly identify model limitations**

Model Validation in a Perfect World

- Ideally, once you complete a model you should hand that model off to another quant for documentation and validation.
- The premise behind this is that the other quant will be less biased about identifying the weaknesses of the model you created, so the assessment is more likely to be complete.
- It is also hard for us to find our own bugs then it is for others to spot them.
- Banks indeed follow this procedure. A separate group will create **documentation**, create **unit tests** and identify **bugs**, underlying **assumptions** and **strengths and weaknesses**.

Model Validation in Smaller Institutions

- In many cases this type of separation is not possible. In those cases the onus will be on you to create these set of procedures.
- In these cases, you will have to identify your own bugs, create your own unit tests and do your own analysis of model strengths and weaknesses.
- You may also still be required to create a document or white paper describing the model in detail. This gives your firm confidence that they are not completely dependent on you to run the model.

Components of Model Validation

- Documentation / White Paper
  - Non-Technical Model Summary
  - Summary of Assumptions
  - Summary of Strengths and Weaknesses
- Units Tests
- Code Review

Documenting our Models

- A document describing the theory behind your model should accompany any model that you create.
- This document should explain:
  - What the model is designed to do
  - The underlying model parameterization
  - The data that you use as an input to the model
  - The algorithm that you use to calibrate the model to the market data.
  - The model's underlying assumptions

Documenting our Models

- It should also provide a robust analysis of the circumstances in which it is appropriate to apply this model, and the circumstances in which another model may be chosen.
- It should identify any sets of parameters with known close form solutions. These will be useful later when creating test cases.
- Model documentation should be updated whenever a new version of the model is released (i.e. a model enhancement).
- Catalogs should be maintained that keep version histories of this documentation as well, and when production switches occurred.

Understanding Model Strengths and Weaknesses

- No model is perfect. All models rely on simplifying assumptions about markets and consequently, each model has limitations.
- As an example, the Black-Scholes Model has many documented uses, but also a long list of limitations that a quant or trader must be aware of when using it.
- We need to make judgments about the appropriateness of these assumptions in different use cases.
- This judgment should also incorporate information on how practical a model is. The Black-Scholes model can be replaced with a model that makes more realistic assumptions, but what challenges does this introduce (i.e. calibration speed of the model).

Understanding Model Strengths and Weaknesses

- Some more complex models, like Heston or SABR may do a better job of matching the volatility skews that we observe in markets, but may still struggle with the skew for shorted dated options.
- Model validation is intended to uncover these types of features in each model by stressing the inputs and considering many market situations.
- If we are unaware of this, it may lead us to a poor model selection decision process.

Identifying Model Assumptions

- The key to understanding model assumptions is understanding the theory behind the model you are implementing.
- All models that rely on Brownian Motions as the driver of uncertainty make certain assumptions about asset prices.
- Usually further assumptions are made about transaction costs, hedging, arbitrage and continutiy of trading.
- We also want to make sure that our implementation is not making additional assumptions beyond what the theory requires.
- Additionally, it is of critical importance to ensure that the assumptions in the theoretical model approximate, or closely resemble, the assumptions being implemented.
- **Question**: when would they not match exactly?

Validating Model Implementation

- Validating how a model was implemented begins by checking the code and attempting to identify divergences between the theory in the model documentation and the implementation.
  - This begins by looking at and understanding the code.
  - This is then followed by separate unit testing of the building blocks of the model.
  - Next, we consider the complete model under simplified assumptions / parameter sets. Ideally, we are looking for cases where the result of the model simplifies to a known closed-form solution.
  - We then relax some of these simplified assumptions on the model parameters and validate the results against an independent implementation of the model.

Validating Model Implementation

- This independent verification could be done either via a third party model (i.e. Bloomberg) or using another technique (i.e. building a simulation to validate results in the Black-Scholes formula)
- We should also verify how the model behaves as a function of its different parameters. For example, the Black-Scholes models behavior should be consistent, and smooth, as we change the input volatility. If not, we likely have an error in our code.
- Lastly we run the model through a wide array of parameterizations and check its behavior, trying to identify anomalies and cases where the results look suspect.
- This wide array of parameters should always include **limiting / boundary parameters**.

Testing the model with simplified parameters

- Some common examples of testing a complex model with simplified parameters might be:
  - Computing an option price with zero time to expiry and comparing to the option's payoff.
  - Computing an option price in a complex model with parameters that force the model to be equivalent to the Black-Scholes model.

Testing the model in special / boundary cases

- Some common examples of testing a complex model in boundary cases might be:
  - Computing an option price with zero & infinite implied volatility
  - Computing option prices where the strike is way in-the-money, or way out-of-the-money.

Validating our Models using Unit Testing

- The process of creating the set of tests to validate our model in a vast array of situations, including in simplified and special cases, is called **unit testing**.
- These unit tests should be designed to span a great deal of possible market conditions.
- For example, if we are working on a model that calibrates a volatility surface, we should make sure it can handle cases of extreme call and put skew, cases where the term structure of volatility is upward sloping or downward sloping, etc.
- Agreeing on the results of these unit tests before putting the model into production is an important means of mitigating model risk.

Using Unit Tests to ensure safe code changes

- Once we have built a comprehensive set of unit tests, and the results of these unit tests have been agreed upon by the creator of the model as well as the validator of the model, the results of the these calculations should then be stored and re-run regularly in an attempt to catch any deviations from these desired results.
- In particular, if we modify the code and then see any such deviations, then they should be explained by some improvement to the model, otherwise, we know that our code contains an error.

Unit Testing in Python

- Python contains a **unittest** module that makes the process of creating these unit tests easy.
- Creation of these unit tests in C++ is more involved, however, it can be done by outputting results of desired calculations to a file and comparing that file to an expected file.
- Python's unittest module includes the functions assertEqual and assert that help us to create test cases. These functions generate exceptions if their conditions are not met.
- For example, we might create a unittest that assert's that the value of an out-of-the-money option at expiry is zero.
- Unit tests should be done on both the building blocks to your models as well as overarching model results.