Integrating data science and ML engineering

Designing a collaboration and handoff process between data scientists and ML engineers

Challenge:
Different needs of data science vs ML engineering

- In both we want *agility* – but this is achieved in different ways:
  - In data science, we achieve agility through the *explorative and iterative notebook* workflow.
  - By contrast, for a production software system to stay agile, we require *clean code, type safety, moving away from notebooks, and loose coupling between components*.
- Challenge: **the former does not automatically scale into the latter**

➔**What leads to agility in model training leads to a lack of agility in model deployment and maintenance**.

➔***Different best practices / quality standards*** for each

➔ Need to find a **good process for *collaboration* and *handoff***

Note on terminology: What about ML scientists?

- “ML scientist” vs “ML engineer”
  - Confusingly, ML scientists are often called "ML engineers".
  - But “engineering” refers to an approach to building reliable, production-grade systems.

- “ML scientist” vs “data scientist”
  - Basically, a ML scientist is a data scientist working with models requiring a higher level of advanced ML expertise.
  - Like data scientists, their emphasis is on getting a ML model to work, rather than ongoing maintainability.
    - E.g., they may spend time on performance optimization of compute bottlenecks, but are less concerned with the readability and maintainability of their code, or how easy it is to run it on another machine.

  -> For our purposes, we can subsume both under the same category. I will use the term “data scientist” to refer to both.

Conflicting best practices in data science and ML

- Starting point: Need to *acknowledge* this dilemma:
  - unique *needs* of both sides (previous slide)
  - unique *talents* of both sides (division of labor)

- Next step: *Codify best practices / quality standards separately* for each side.

- Keep in mind: How to structure *incentives*

Data science vs engineering

- *Difference* from engineering:
  - explorative and iterative -> notebook workflow
  - Higher importance of *domain expertise*
    - Do these data make sense?
    - What way of computing this feature makes the most sense from a domain perspective?
- How data science *supplements* engineering:
  - Exploring the data by someone with domain expertise can:
    - surface problems
    - create new ideas
  - Any resulting changes should be addressed:
    - using *production-grade fixes* rather than hacks (implemented by engineers, based on insights from data scientists)
    - at the *source* (rather than each data scientist reinventing the wheel by performing the same data cleaning downstream)

Why not all engineering best practices apply to data science

- Applying a specific engineering best practice on data science productivity can be:
  - **Counterproductive**: specific needs of the data science process
  - **Productive** - but there may be **hurdles** to adoption:
    - adoption cost (can we lower it sufficiently?)
    - not well-known enough (educate!)
    - incentives encourage short-term focus (same problem as in software engineering)
  - **Neutral** to data scientist productivity (We may as well adopt them early on to make handoff easier)

1) Why some engineering best practices may be counterproductive in data science

- Some engineering practices are too constraining (their cost is not offset by large enough concomitant benefit)
  - Code is **short-lived** -> maintainability is less important
  - Lesser need to *foresee* possible problems; instead, take a close look at *actual* data, and react to problems as we observe them.
  - **Interactive** data science workflow provides *different ways of addressing certain problems*. E.g.:
    - Knowing the data schema beforehand is less important, because we can just take a look at the data and fix any problems we observe.
    - In a notebook, type hints are less important for readability because we can interactively inspect what variable looks like if we're not sure.

2) Why other engineering best practices are relevant to data science

- Even though data science code is more short-lived, and long-term maintainability is thus less important, ***changeability*** *is still important due to the iterative nature of the explorative workflow*.

Shared best practices

- Must have:
- Should have:
- Would-like to have:
- (Does not need to have:)

Data science best practices

- Must have:
  - Quick & easy dev setup
  - Iterative and interactive workflow
- Should have:
  - Clean code
  - Invest time to find good *data-science* tools for job
  - Use engineering tools determined to be worth the investment (engineering team should do evaluation, recommendation, and setting up)
    - leverage power of IDE (rather than notebook in browser)
- Would-like to have:
  - Leverage design patterns to achieve loose coupling between components
  - Trusted test suite (automated unit and integration/acceptance tests); automated data validation, static analysis
- (Does not need to have:)

ML Engineering best practices

- Must have:
  - Clean code
  - Leverage design patterns to achieve loose coupling between components
  - Trusted, automated test suite
    - unit tests
    - integration/acceptance tests
    - data validation
    - static analysis
  - Type safety:
    - code: use type hints, and force in CI pipeline
    - data: use explicit data schemas
  - Observability:
    - code: structure & centralized logging, monitoring and alerting, distributed tracing if using micro service architecture
    - model performance: comparison between different models, comparison of same model over time (model drift?), performance for different subsets of the population/bias (if substantial, does this vary by model?).
    - data: data quality
  - Infrastructure-as-code
  - CI/CD
  - Data lineage
  - Invest time to find good tools for the job
    - Avoid reinventing the wheel -> Leverage managed services wherever possible (unless “unfair” pricing)
    - leverage power of IDE (rather than notebook in browser)

- Should have:
  - Profile code and optimize performance bottlenecks
    - (Why we do not consider this a must-have: Engineers' time is very expensive, and so is delaying features or accumulating technical debt, so unfortunately performance optimization has sometimes to be sacrificed for these even more important goals.
  - Periodically reevaluate if there are better tools for the job
    - e.g., Pandas alternatives (such as Polars)
    - e.g., Spark vs Presto
    - e.g., End-to-end (Sagemaker) versus best-off-breed MLOps tools

- Would-like to have:

- (Does not need to have:)

Best practices for data science:

- How engineering complements data science:
  - There are plenty of cases where engineering principles can benefit data scientists, but:
    - there is a substantial adoption cost:
      - if you can bring adoption cost down, it may make sense to include some engineering best practices as data science best practices
    - There is a knowledge gap
    - the adoption threshold seems too high

Collaboration between data science and ML

- Recommendation: Create **cross-functional teams** of data scientists and ML engineers
  - ***Vertical* slicing** in Agile (cross-functional teams / slice by *value*)
    - Data pipeline, model training, deployment
  - Handoff between data science and ML is challenging (even *with* the best possible process)
    - Requires collaboration
    - Collaboration is much easier *within* a team (rather than *between* teams)
    - Remember the lessons from DevOps (don't just throw models over the wall) and microservices.
  - Collaboration early in the ML lifecycle reduces the risk of introduces bugs during productionization
    - e.g., if ML engineers refactor data transformation code before it is used in model training pipeline, we can make sure that we apply exact same transformations in training and inference, thereby eliminating a common source of training-serving skew
- Challenges:
  - *Organizational inertia*: If data scientists are located on the business side, it's tempting to just add a new MLOps team within IT/engineering
  - *Cultural differences* between data scientists and engineering
    - It's tempting to separate both sides into different teams.
    - But: This perpetuates these differences and makes collaboration hard.
    - Remember again the lessons from DevOps!

What exactly needs to be handed off?

➔We need to productionize:

- Code
- Models
- Data (transformations)
- Data and model understanding

Handoff, 1): Productionizing *models*

- Easiest (compared to code and data)
- ***Model registry*** can serve as a convenient hand-off point
- Potential problems:
  - Changes to model interface (incl. schema of input data)
  - Dependency management
- Solution: *Formal contracts* that are *automatically enforceable*
  - Standard process for environment creation
  - Better yet: Use containers
  - Define model interface and data schemas in shared libraries
    - Class interface can be cheaply enforced through static analysis (mypy) in CICD pipeline (and IDE plugin)
    - Data schema checks sometimes require running the process for validation, so it's more expensive. Make this part of acceptance tests.
      - E.g., pandera
  - Note that steps also bring huge side-benefits in terms of documentation and system understandability

Handoff, 2: Productionizing *code*

- Harder than productionizing models.
- Question: To what extent can you expect data scientists to follow software engineering best practices?
  - Remember: Different *needs* and *talents* -> It's not realistic to expect data scientists to become engineers.
  - That said, there is also an *overlap* in best practices -> In some areas, data scientists can become more productive by leveraging insights from software engineering
    - E.g., power of IDE, type hints, basic understanding of what makes code changeable (because even though data science code is typically short-live, changeability is still highly important due to the iterative nature of the workflow).

  -> Need to find the right balance, taking into consideration:
  - Which parts of software engineering best practices would also benefit data scientists
  - How much more costly it is to add something after versus before the handoff
  - How hard it is / how costly it is (in terms of one-time investment) for data scientists to learn required skills
  - -> The right point on this trade-off varies:
    - over time
      - can't expect data scientists to learn a bunch of new skills at once
      - easier to get buy-in from data scientist if we start with quick wins that help them see the benefits for themselves
    - between companies (and even between teams)
      - Ability to attract top talent (pay premium, corporate brand, etc.)
      - Different specialties within data science require different level of engineering knowledge

- Suggested steps:
  1. Make it as easy as possible for data scientists to follow software development best practices
  2. Standards?
  2. Leverage automatic code improvement tools:
  3. Manual re-factoring by ML engineers
  4. Find production-ready solution for any hacks

-> **Challenge: Balance quality control with ability of data science team to self-serve**

Handoff 2): Productionizing *code*
1) Make it as easy as possible for data scientists to follow software development best practices

- e.g., provide templates for recommended IDE configuration, etc.
  - Ideally, laptops should already come preconfigured with recommended settings for each job function.
- This is an example of how to properly structure incentives!
- ML Engineers should make themselves available to help (and coach) with things in their area of expertise. And vice versa.

Handoff 2): Productionizing *code*

2) Leverage automatic code improvement tools:

-> Automate what can be automated

- linting and formatting (AutoPEP-8, Black, YAPF)
- adding type hints (e.g., monkeytype)

Handoff 2): Productionizing *code*

- 3. Manual re-factoring by ML engineers
  - Performance: e.g., convert panda map/apply to vectorized operation of possible;
  - Reliability (e.g., Type-safety: add type hints, add data schemas (especially if read from external source))
  - Maintainability: Readability,
  - Testability: extract functions
- 4. Find production-ready solution for any hacks
  - E.g., unofficial data sources need to be productionized

Handoff, 3): Productionizing *data*

- *How* do you productionize data?
  - Outside the scope: data governance strategy, etc.
    - Should be defined at the organizational level
    - If non-existent or not sufficient, ML engineering may want to be part of this discussion to communicate the needs of ML.
  - In scope:
    - Bring *unofficial* data sources into the organization's official data governance realm
      - E.g., Excel files
    - Productionize data **transform *code***
      - ➔ Reduces to the previous problem of productionizing code
    - Productionize **features** feature store
      - ➔ Store transformations in **feature store**

Handoff, 3): Productionizing *data*

- Additional challenges: Productionizing *data* **requires wider support from leadership** due to upstream dependencies
  - Assign *data owners* who are domain experts
  - Collaborate with data owners to fix any data problems that data scientists discovered at the source
  - Ideally, the *general* data engineering ("silver tables") is handled by dedicated teams/data engineers.
  - In the short term, ML engineers may have to lend a hand in order to show the value of this model (and because they have an interest in it).
  - In the long term, ML engineers should only be productionizing data transformations that are related to feature engineering or are very specific to their use case ("gold tables")

Handoff, 3): Productionizing *data*

- This is the hardest problem of the 3
- Goals:
  - Ensure data **quality**
    - Trustworthiness
    - Data problems should be *addressed at the source*, by someone *familiar with the domain* -> data users should be able to *rely* on quality
  - **efficiency**: Don't make people reinvent the wheel
    - Data problems should be fixed once at the source, not by every user
      - Quicker for a domain expert to validate data quality and fix potential problems
      - Domain expert can do a better job at detecting problems and making decisions on how to fix any issues found
      - Has to be done only once rather than multiple times
    - It should be easy to *share* relevant data and features, and to *discover* them in the first place easy
  - Standardize feature calculations

    -> **comparability** across use cases
  - data **lineage** / provenance
  - Potentially: Low **latency** for real-time inference

Handoff, 4): Data and Model understanding

- Goals:
  - Don’t throw away insights!
  - Don’t make MLOps engineers reinvent the wheel!
- Data scientists accumulate data and model understanding because:
  - They tend to be closer to the business side than engineers, so they have more domain knowledge
  - Due to the *exploratory* nature of the data science process, they often discover surprising insights
  - They perform data and model validation (formal or informal) to check their work

How to structure Handoff?

- Need to find a handoff process that *works for both sides*
  - Acknowledge different needs of both sides
  - Danger: Compromises will be required on both sides; be careful the outcomes are not driven by the prevailing balance of power
- Especially if handoff is *between teams:* **Define stable, formal contract. Enforce automatically** if possible.
  - e.g., data schemas, API schema, gives interfaces/data classes defined in shared libraries, Gherkin scenarios. Enforced in CI/CD pipeline.
  - …because this decouples teams from each other, thus reducing blockers and communication inefficiencies
  - Requires an *engineering* mindset
  - Still beneficial - though less important - within cross-functional team

How to make handoff work?

- **Quick feedback loop**
  - In the best case, static analysis plug-ins in the IDE tells us when we are violating an interface right as we are writing the code.
- **Align incentives**
  - If one side breaks the contract, it **should be clear which side did so, and accordingly has to fix it**.
    - E.g., test is run automatically in pipeline and blocks merging.
  - We ideally want to **avoid bothering the other side with alerts they did not cause**.

Maturity levels

- Level 0: Only ad-hoc communication between teams of what and how to hand off.
  - Characteristics: handoff often doesn't go smoothly:
    - "it works on my machine problem" (e.g., because there is no easy way to re-create an identical virtual environment
    - ML team is not able to satisfy frequent request for changes (e.g., adding an additional feature)
- Level 1: Formal handoff process that divides responsibilities
- Level 2: Formal handoff process + collaboration throughout the ML lifecycle to minimize handoff
  - "shifting left": Where needs of engineering are not opposed to needs of data science, it's cheaper to introduce engineering requirements earlier in the ML lifecycle