Team Objectives

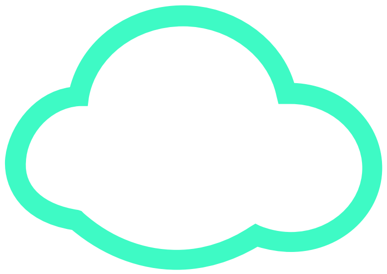
Build a distributed machine learning pipeline in Pandas and Dask using gigabytes of retail data from a large retail company. The final deliverable will be a presentation in Jupyter / PowerPoint describing your approach, modeling techniques, and final results.

1. Load data from a .pkl into Jupyter notebook running on an Amazon EC2 instance

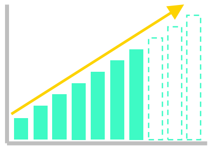
2. Explore the data and generate new features for modeling

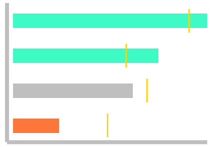
3. Build a tuned modeling pipeline to produce accurate forecasts

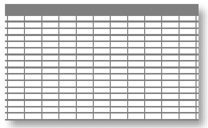
4. Interpret the results and write a summary for a non-technical audience

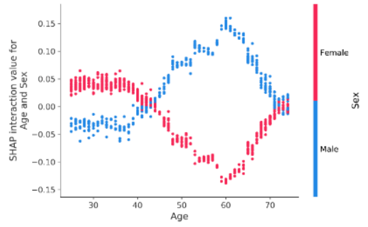


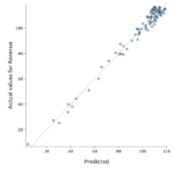
AWS EC2

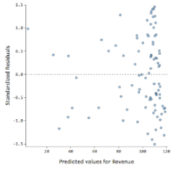












Data Overview

The dataset we'll be using is real, historical Walmart data taken from the M5 Forecasting Competition on Kaggle. The data was briefly cleaned and aggregated to the week level to reduce sparsity.

About the M5 Competition

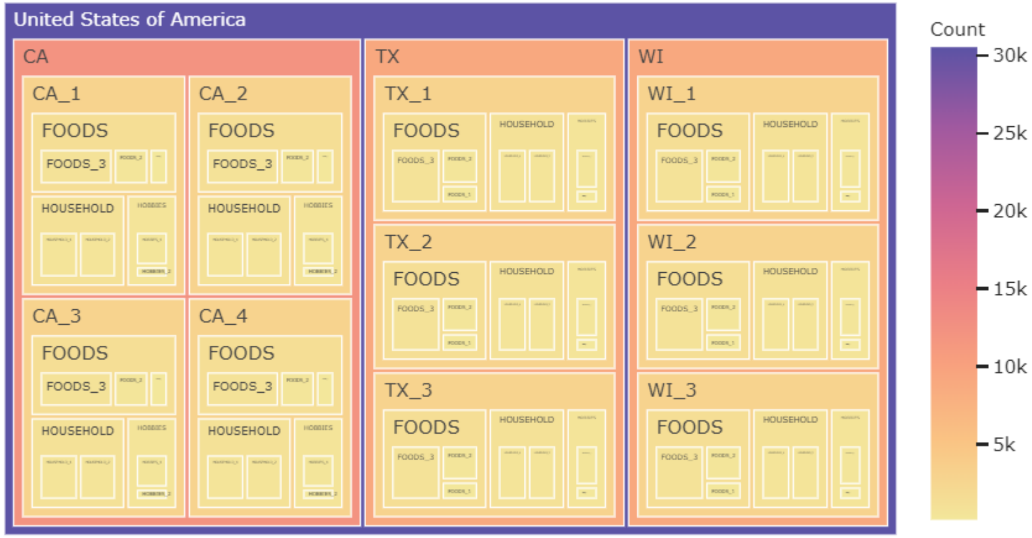
- Purpose is to advance the theory of forecasting and improve its utilization by business and non-profit organizations
- Preeminent forecasting competition in the academic world
- First competition (now called the M1) took place in the 1980's)
- Latest competition (M5) closed just last week

Data Overview

- Hierarchical sales data from Walmart, the world's largest company by revenue
- Covers stores in three US States (California, Texas, and Wisconsin)
- Includes item level, department, product categories, and store details
- See the Word doc below for details:

Microsoft Word Document

Walmart: Distribution of items



More EDA available here

Team Expectations

Rules of the Road

- Machine learning is a team sport; be respectful of others' contributions and code reviews
- Try to fix your own bugs, but don't be afraid to ask for help if you get stuck
- Commit early, commit often with useful descriptions
- Show up on time and ready to walk-through your code
- Update the Trello board before each Zoom meeting
- Everyone will present the weekly status update at least once
  - Accomplishments
  - Roadblocks
  - Decisions Needed
  - Next Steps

Things We'll Learn

- Real-world software engineering skills
  - CI/CD basics
    - Writing good tests
    - Automating them with GitHub Actions
  - Functional programming basics, including taking advantage of
    first-class objects
  - Writing fast, efficient python code
- Machine learning skills
  - Feature engineering
  - Algos (XGBoost, LightGBM)
  - Validation (and how not to do it)
- Collaborating with other programmers on GitHub and Trello

Guiding Principles & Technology Stack

Guiding Principles

1. Other teams *do* things; **we build things**
2. **Encourage continuous feedback and code reviews**; we're all here to learn
3. **Eliminate unnecessary meetings and PMO activities** ("Talk less, do more")
4. **Proactively manage technical debt** (beware the "broken window" theory)
5. **Minimize our collective WTF's per minute** (next slide)
6. **Automate your tests** for reuse (short-term pain => long-term gain)

Tech. Stack

Languages / Frameworks

Trello

Task Management Platform

jupyter GitHub

Collaboration Tech.

Amazon EC2

Data Hosting Service

Minimizing WTF's Per Minute: Guidelines from *Clean Code*

- **All object names should be descriptive and meaningful**
  - Long names are fine if that's what it takes to make them descriptive (can easily search for names using tab key)
  - Objects / vectors / tables should be lower-case nouns [e.g., training_df]
  - Methods / functions should start with verbs [e.g., get_file_path()]

- **Comments should only be included as a last resort when your code isn't expressive enough**
  - Comments are always failures; they suggest that you haven't found a way to express yourself using code
  - When writing a comment, ask yourself: can I refactor my code or change variable / function names to clearly express what I'd want to convey in a comment?

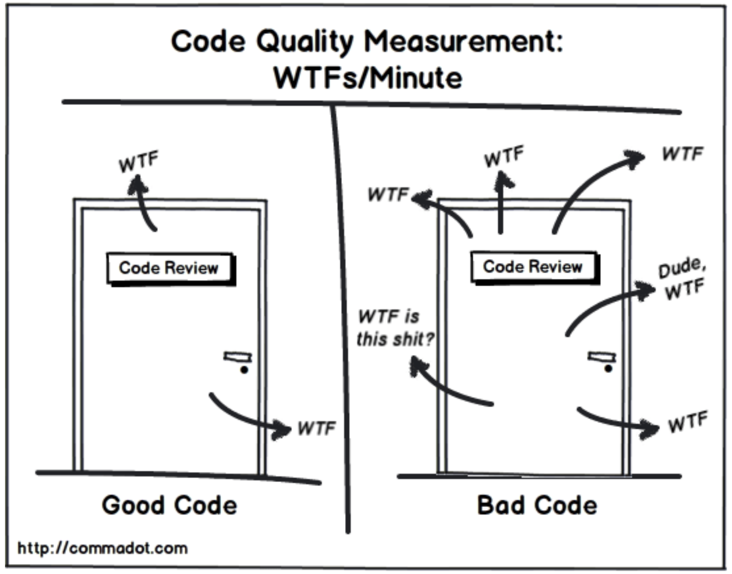
- **Unused code is to be deleted, not commented out**
  - If you're worried about losing a piece that you may need later, commit a new change in GitHub, delete the unwanted code, and then commit a new change

- **The vast majority of data transformations should sit within expressively-named functions**
  - Makes code easier to follow
  - Enables us to test each piece independently
  - Allows us to quickly locate and fix broken code in production
  - Avoids having interim tables take up valuable RAM

- **All functions should do one and only one thing and contain no more than 3 parameters maximum**
  - Functions should be as small as possible
  - If you have a function longer than 5 lines, you probably should split it up until multiple functions

- **No hardcoding, period**
  - Anything hardcoded should be in the parameters table at the beginning of the script

- **Minimize the number of for() and while() loops used in each script**
  - Spark internals are designed to simplify your code on the back-end as long as you avoid UDFs



Appendix:
“Hacker Laws”

“Hacker Laws” (1 of 3)

(source)

	Summary	Implications
Amdahl's Law	Parallelization increases processing speed according to a hyperbolic tangent curve	There’s a drastic speed difference between something that’s 95% parallelized and something that’s only 50%
The Broken Windows Theory	Visible signs of crime or lack of care increases the odd for future crime / disorderliness	Be intolerant of bad code
Brooks' Law	Adding human resources to a late development project makes it later	Staff the team right from the beginning. Focus on quality, not quantity, of DB practitioners
Conway's Law	Technical boundaries of a system will reflect the structure of the organization that built it	Use a client-server model for powering other use cases. Don’t bolt-on new applications.
Dunbar's Number	We can only keep so much in memory	Refactor early and often. Write simple systems. Ruthlessly cut deprecated code.
Gall's Law	Highly-complex systems are more likely to fail	Write clean, tight functions. Limit dependency between modules.
Goodhart's Law	When a measure becomes a target, it ceases to be a good measure.	Be vigilant about overfitting.
Hofstadter's Law	It always takes longer than expected.	Be careful not to over scope engagements. Leave room to breathe so you can cover the team if disaster strikes.
Hutber's Law	Improvements to a system will lead to a deterioration in other parts of that same system.	Limit dependencies between modules. Ensure you’re constantly rerunning the pipeline during development to avoid upstream issues.
The Hype Cycle & Amara's Law	We overestimate tech. in the short-run and underestimate in the long-run.	Be purposeful when building your roadmap to describe exactly what the team will / will not do. Avoid scope creep caused by tech. hype.

“Hacker Laws” (2 of 3)

(source)

	Summary	Implications
Hyrum's Law (The Law of Implicit Interfaces)	As your number of users scale, you should expect all attributes of your API to be used.	Test *all* of your code.
Kernighan's Law	Debugging is twice as hard as writing the code in the first place.	Write simple, not clever code. Make it as easy to read as possible.
Murphy's Law / Sod's Law	Anything that can go wrong will go wrong.	Test *all* of your code.
Occam's Razor	Prefer simple solutions.	Write simple code. If something feels complicated, ask yourself if there’s an easier way to think about it
Parkinson's Law	Work expands so as to fill the time available for its completion	Timebox efforts to 3-5 hours per week. Work smarter, not harder.
Premature Optimization Effect	Premature optimization is the root of all evil.	Get it to work and write the test before you worry about optimizing.
Putt's Law	Management often necessarily doesn’t understand technical details.	Simplify your messaging. Partner with people who have different strengths than you do.
Reed's Law	The utility of large networks scales exponentially with the size of the network	Build a big, diverse team
The Law of Conservation of Complexity (Tesler's Law)	Some complexities cannot be reduced.	Don’t make the user journey more complex by making the code easier.
The Law of Leaky Abstractions	All non-trivial abstractions introduce bugs.	Be sure your abstractions are necessary.
The Law of Triviality	People tend to spend more time on trivial or cosmetic issues than on substantial ones	Be judicious with your time. Avoid unnecessary PMO like the plague

“Hacker Laws” (3 of 3)

(source)

	Summary	Implications
The Pareto Principle (The 80/20 Rule)	80% of value comes from 20%	Work with leadership to identify the 80% of value we should be going after
The Robustness Principle (Postel's Law)	Be conservative in what you do, be liberal in what you accept from others	Avoid using rigid functions. Understand that most users will try to enter weird objects as inputs
The Single Responsibility Principle	Every module should only do one thing.	Write simple code. Don’t try to do too much in one function
The Open/Closed Principle	Interfaces should be open for extension but closed for modification	Write extensible functions (e.g., take other functions as parameters). Try not to be too prescriptive.
The DRY Principle	Don’t repeat yourself	Don’t write the same code more than once. Use functions and variable effectively
The KISS principle	Keep it simple.	Do everything you can to fight complexity
YAGNI	Only build functionality your users are asking for.	Try to avoid pursuing ideas that aren’t explicitly being asked for by someone (either a PMD or a client)
The Fallacies of Distributed Computing	Distributed computing offers many advantages but also some unforeseen consequences	Educate the team on Spark to help them solve known issues, particularly OOM failures and latency limitations

