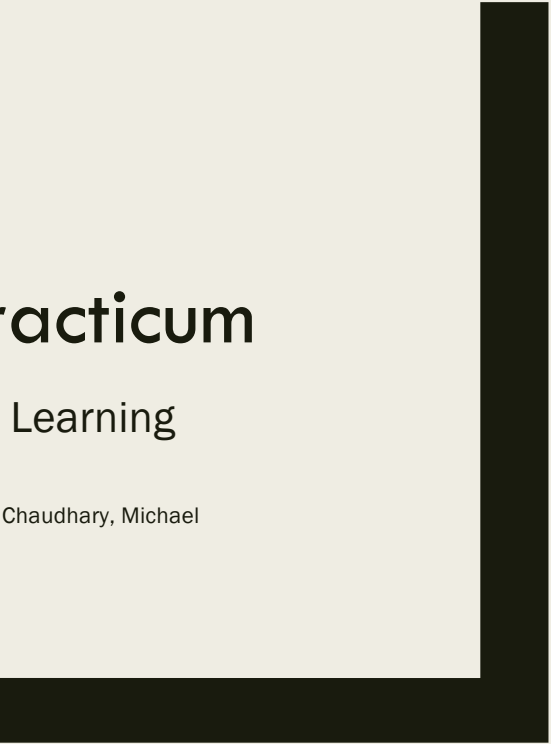# CSCI 8360: Data Science Practicum

Lecture 4: "Hidden Technical Debt in Machine Learning Systems"

D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, Dan Dennison

Dr. Shannon Quinn

What is "technical debt"?

- Coined by Ward Cunningham in 1992
- *Refers to long-term costs incurred by moving quickly in software engineering*
- Debt metaphor
- *Not necessarily a bad thing, but **always** needs to be serviced*
- Goal: **NOT** to add new functionality
- *Enable future improvements, reduce errors, improve maintainability*
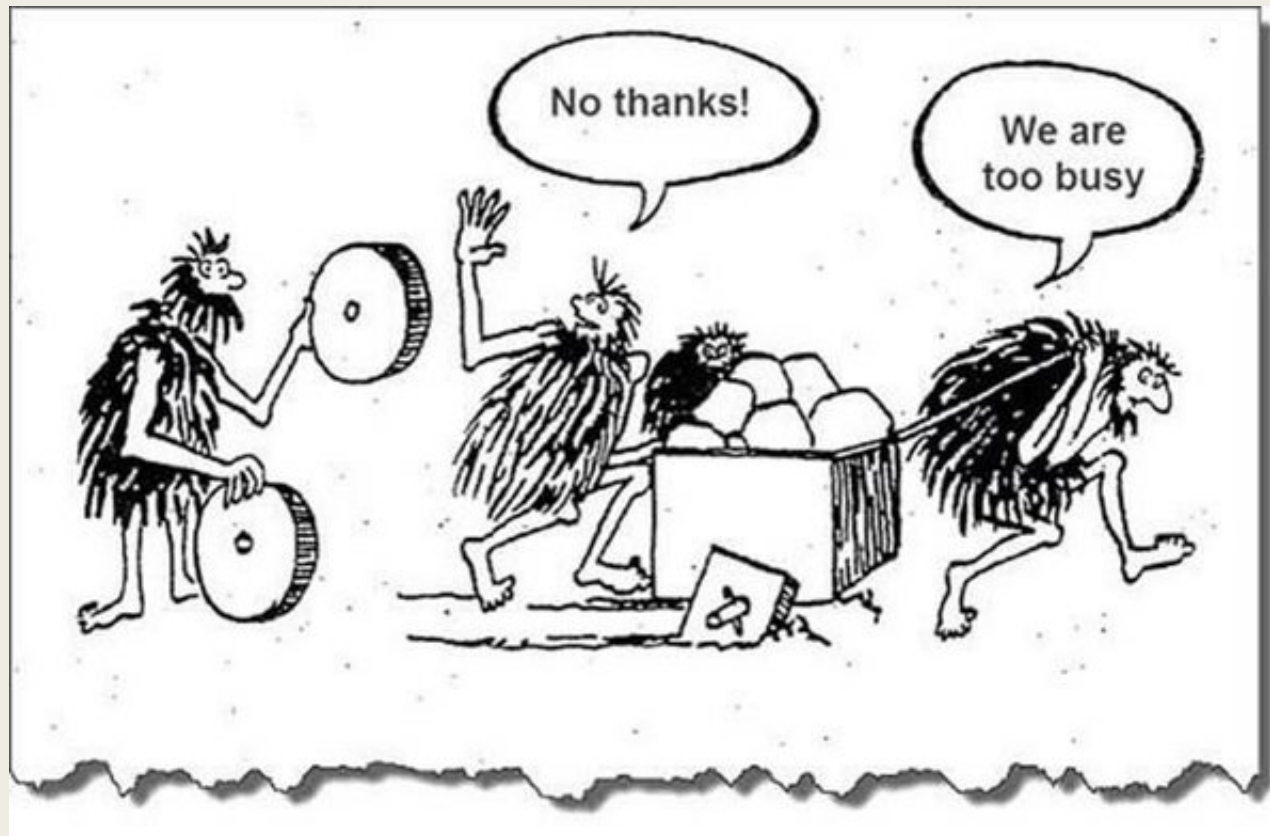
What is "technical debt"?
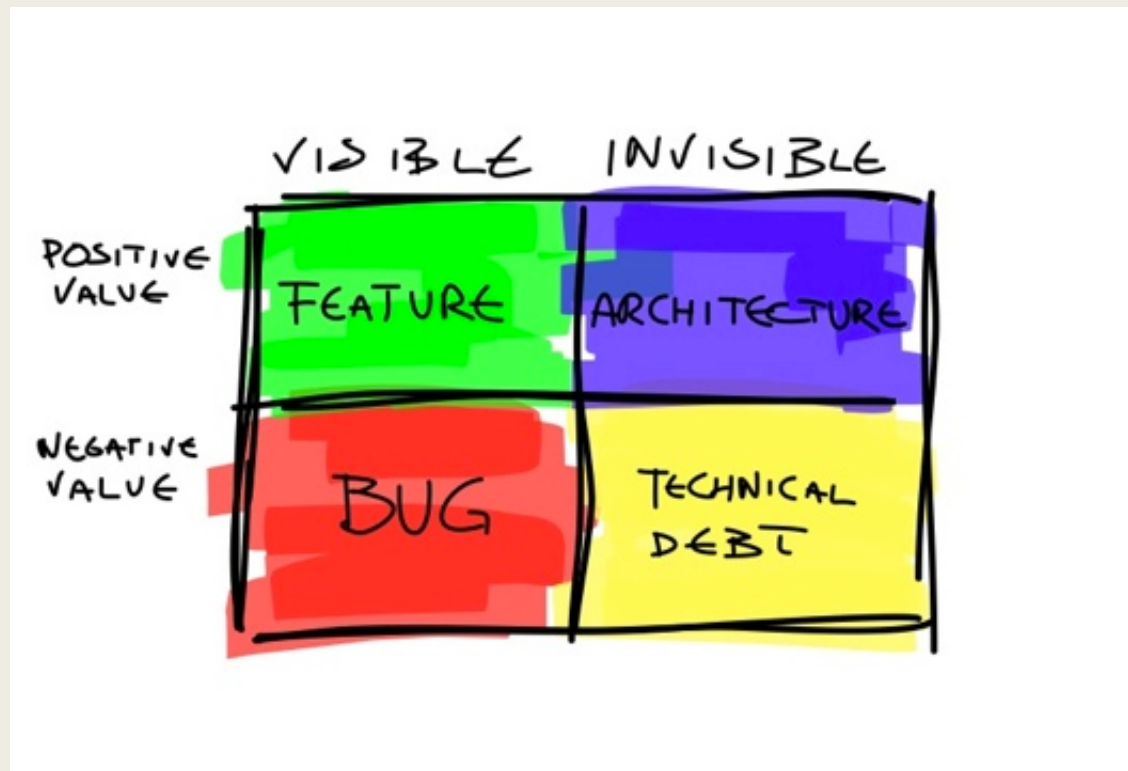
Technical debt

From Wikipedia, the free encyclopedia

Technical debt (also known as **design debt**[1] or **code debt**) is "a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution[2]".

Technical debt can be compared to monetary debt.[3] If technical debt is not repaid, it can accumulate 'interest', making it harder to implement changes later on. Unaddressed technical debt increases software entropy. Technical debt is not necessarily a bad thing, and sometimes (e.g., as a proof-of-concept) technical debt is required to move projects forward. On the other hand, some experts claim that the "technical debt" metaphor tends to minimize the impact, which results in insufficient prioritization of the necessary work to correct it.[4][5]
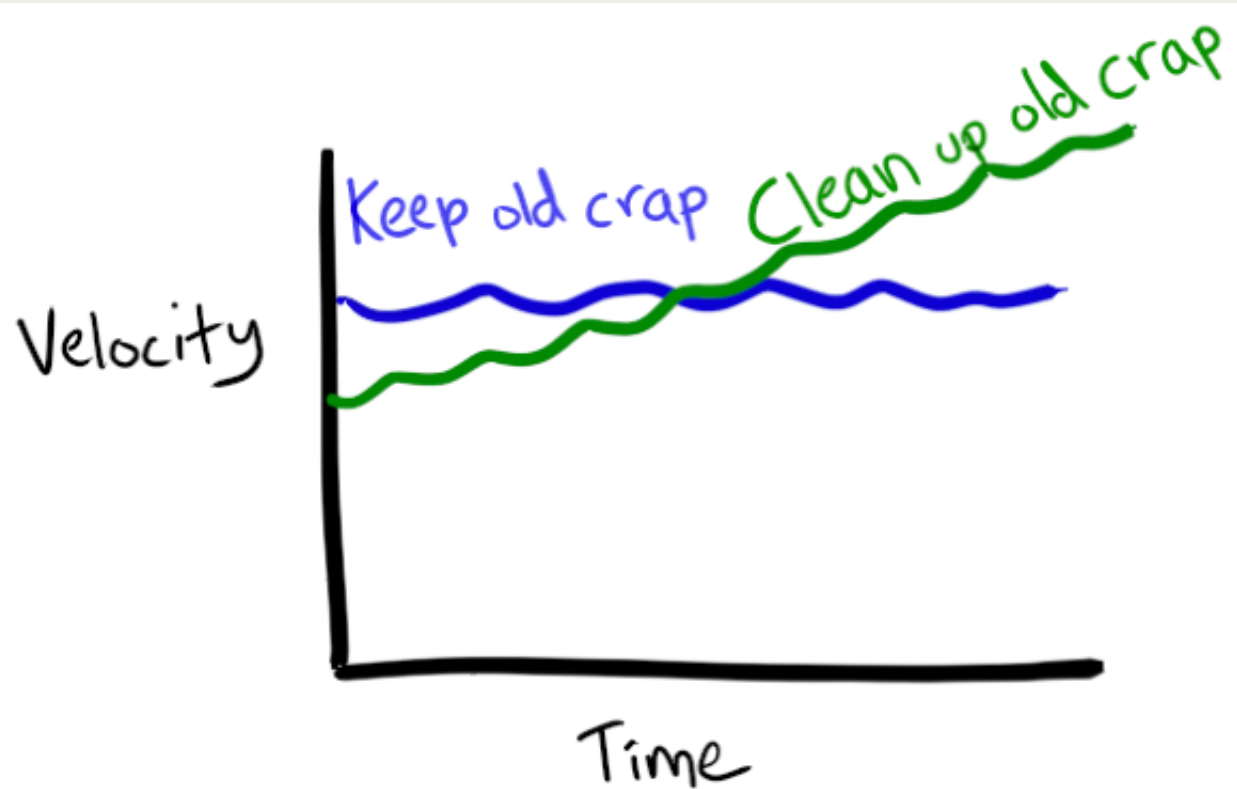
What is "technical debt"?

What is "technical debt"?

What is "technical debt"?

What is "technical debt"?
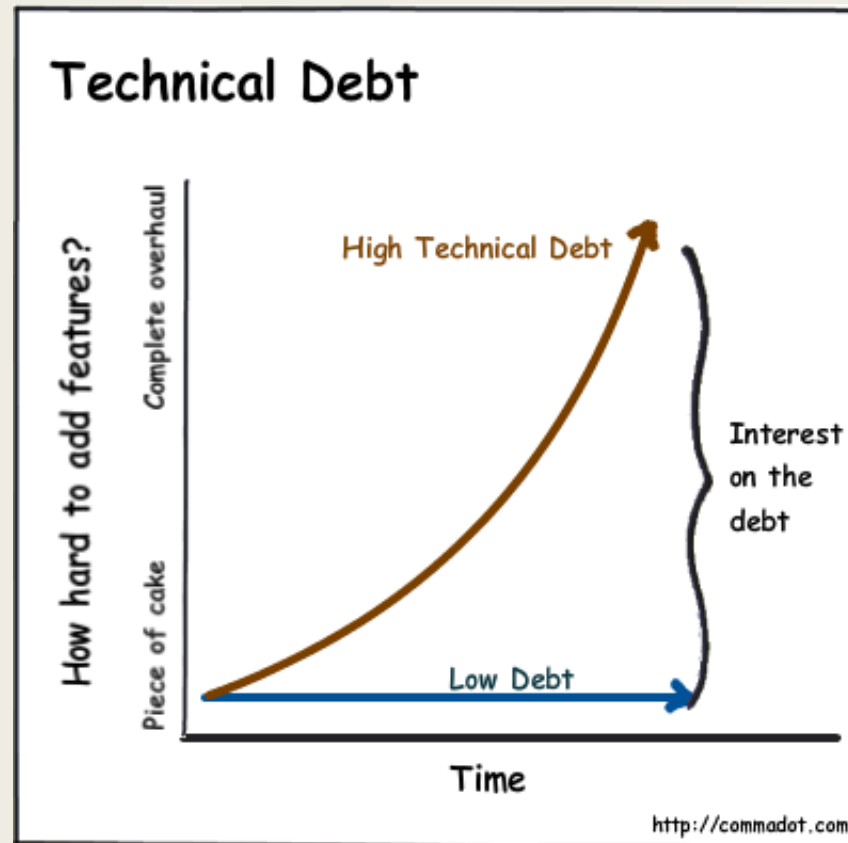
Causes of technical debt

Technical Debt and Machine Learning

- All the maintenance problems of "traditional" code
- *Plus an additional set of ML-specific concerns*

- Debt can exist at *system* level, instead of [strictly] code level
- *Data influences ML system behavior!*
- *"Traditional" abstractions and boundaries can be corrupted*

- "Traditional" methods for paying down code-level debt are not sufficient to address ML-specific issues at system level

1: Model Complexity

- Traditional software engineering: strong abstraction boundaries, encapsulation, and modular design

- Machine learning: desired behavior relies *specifically* on external data
  - *The real world does not fit into tidy abstraction rules*

1: Model Complexity

- ■ **Entanglement**
- – *ML systems mix signals*
- – *If we change distribution of one input feature, weights of other d – 1 features may change as well*
- – *"Changing Anything Changes Everything"*

- ■ **Correction Cascades**
- – *We have model m_a for problem A, but need a solution for slightly different A'*
- – *Tweak m_a to m'_a...then need to solve A'', and so on*

- ■ **Undeclared Consumers**
- – *Output of your model m_a may be input to some downstream system*
- – *Changes in your model m_a will drastically affect performance of downstream consumers*

2: Data Dependencies

- Traditional software engineering identifies "dependency debt" as a key contributor to overall technical debt

- Data dependencies in ML systems carry similar debt-building capacity, but with the added joy of being more difficult to detect

2: Data Dependencies

- **Unstable Data Dependencies**
- *Consumed input changes over time*
- *Input signal comes from another ML system that updates itself*
- *Engineering ownership of input signal is distinct from ML system consuming it*

- **Underutilized Data Dependencies**
- *Input signals that provide little modeling benefit*
- *Legacy features, bundled features, correlated features, ϵ-features*

3: Feedback Loops

■ A key feature of ML systems is influencing its own behavior

■ **Analysis debt**

– *Difficult to predict behavior of a given model before release**

* Tay?

3: Feedback Loops

- **Direct Feedback Loops**
  - *Model directly influences selection of its own future training data*
  - *Supervised algorithms*
- **Hidden Feedback Loops**
  - *Two systems indirectly influence each other through the world*
  - *Related but distinct recommendation systems—improving one leads to changes in the other*

4: ML System Anti-patterns

- How much code in a machine learning system is, well, *machine learning*?

- The non-ML code is "plumbing"—the majority, but nonetheless typically an afterthought.

4: ML System Anti-patterns
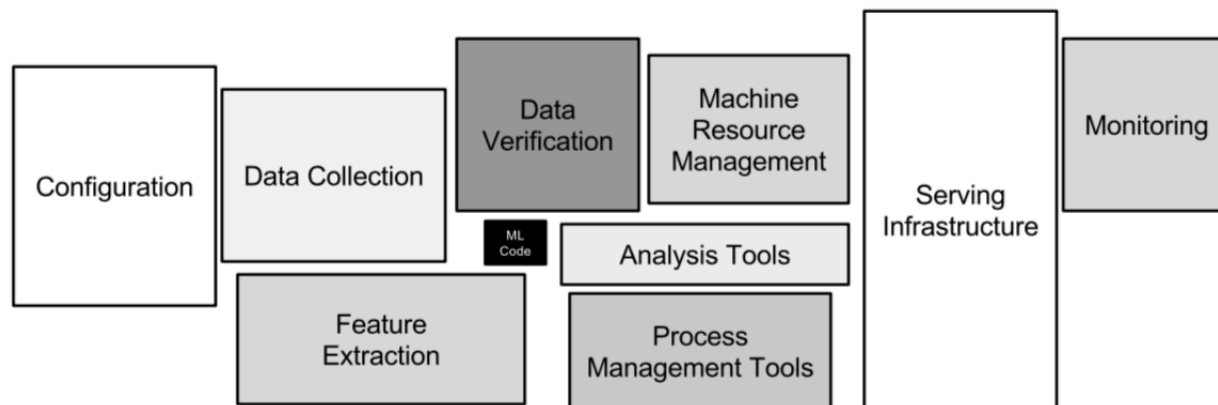
Figure 1: Only a small fraction of real-world ML systems is composed of the ML code, as shown by the small black box in the middle. The required surrounding infrastructure is vast and complex.

5: Configuration Debt

■ Large systems have wide range of configurable options

– *Features used*

– *Data selection*

– *Algorithm hyperparameters*

– *Verification methods*

– *Pre- and post-processing routines*

■ May reach the point of **# of lines of configuration >>> # of lines of code**

5: Configuration Debt

- It should be easy to specify a configuration as a small change from a previous configuration.

- It should be hard to make manual errors, omissions, or oversights.

- It should be easy to see, visually, the difference in configuration between two models.

- It should be easy to automatically assert and verify basic facts about the configuration: number of features used, transitive closure of data dependencies, etc.

- It should be possible to detect unused or redundant settings.

- Configurations should undergo a full code review and be checked into a repository.

6: Changes in the External World

- Related to #2 "Data Dependencies" and #3 "Feedback Loops"

- **Fixed Thresholds in Dynamic Systems**
- *What p(x) will be used to separate "spam" from "not spam"?*
- **Monitoring and Testing**
- *What to monitor?*
- *Prediction Bias (distribution of predicted labels)*
- *Action Limits (automated alerts)*
- **Upstream Producers**
- *Any upstream data producers should also be thoroughly and frequently tested*

7: Other ML-related Debt

- **Data Testing Debt**
- *If data == code in ML systems, and code should be tested, then some data should be tested as well to monitor for changes in input distributions*

- **Reproducibility Debt (Open Science!)**
- *If frameworks are re-run in identical configurations, should produce identical results*

- **Process Management Debt**
- *Mature systems may have dozens or even hundreds of simultaneous ML models*

- **Cultural Debt**
- *Create a healthy project team culture with a breadth of expertise that rewards good engineering practices*

Conclusions

- Measuring technical debt
- *No clear metric*

- Simply "moving fast" is not evidence of low debt or good practices
- *Moving quickly often **introduces** technical debt!*

- Useful questions for consideration:
1. *How easily can an entirely new algorithmic approach be tested at full scale?*
2. *How precisely can the impact of a new change to the system be measured?*
3. *Does improving one model or signal degrade others?*
4. *How quickly can new members of the team be brought up to speed?*

Questions?
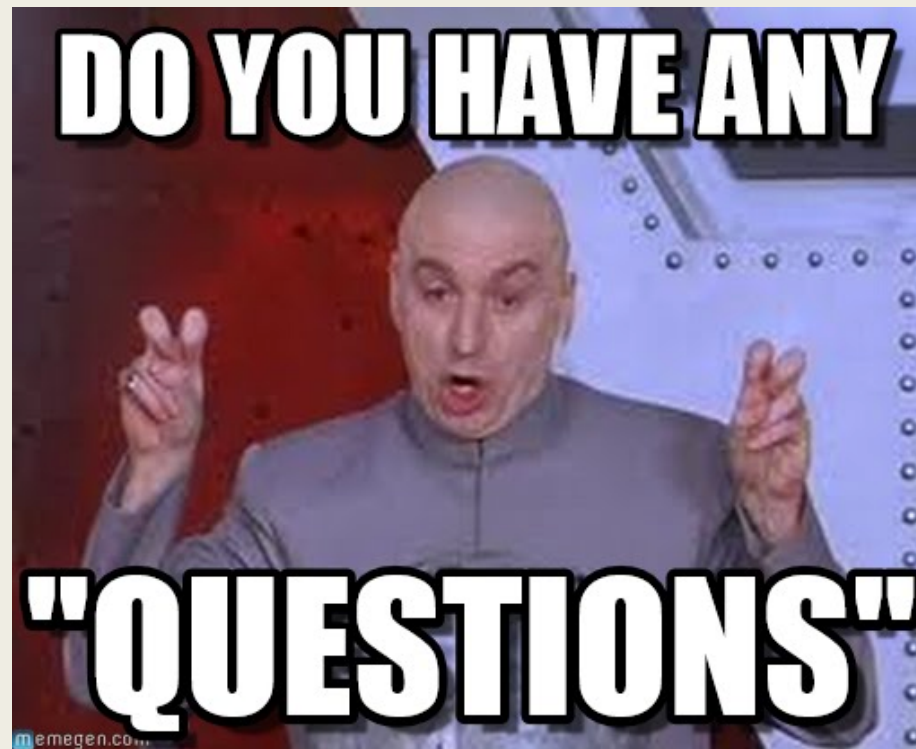
PO Review

- How was it?

P0 Review: Good Version Control Habits

DON'T

- Hard-code program arguments *(read the instructions)*

- Include data or program output in version control

- Use Jupyter notebooks for program code

- Leave commit comments of "p0 update"

- Start at the last minute

DO

- Use tickets (Issues)

- Leave descriptive commit messages ("Fixed bug from ticket #3")

- Create CONTRIBUTORS, LICENSE, and README files

- Create a logical directory structure ("src", "examples", etc)

- Use Jupyter notebooks and toy data sets as examples for new users

```
function foo(bar) {
  const baz = bar(false)
  // we no longer do this for some good reason
  // if (baz === 'foobar') {
  // return baz
  // } else {
  // return bar.foobar()
  // }
  return baz
}
```
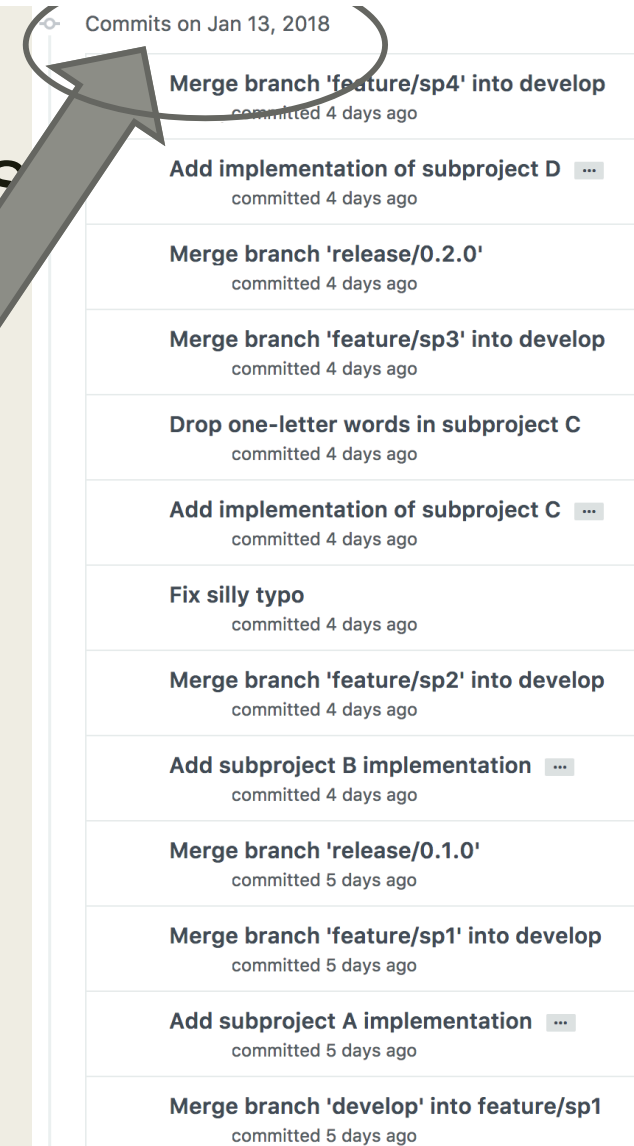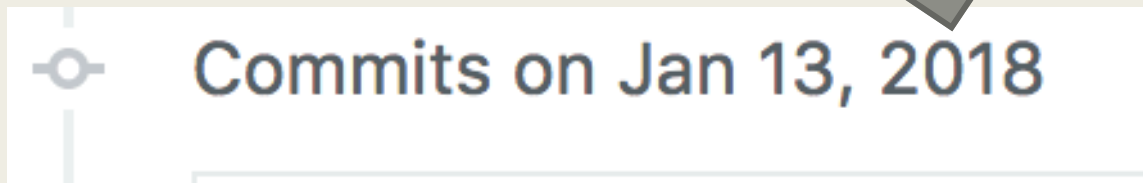
```
function foo(bar) {
    return bar(false)
}
```

https://blog.kentcdodds.com/please-don-t-commit-commented-out-code-53d0b5b26d5f
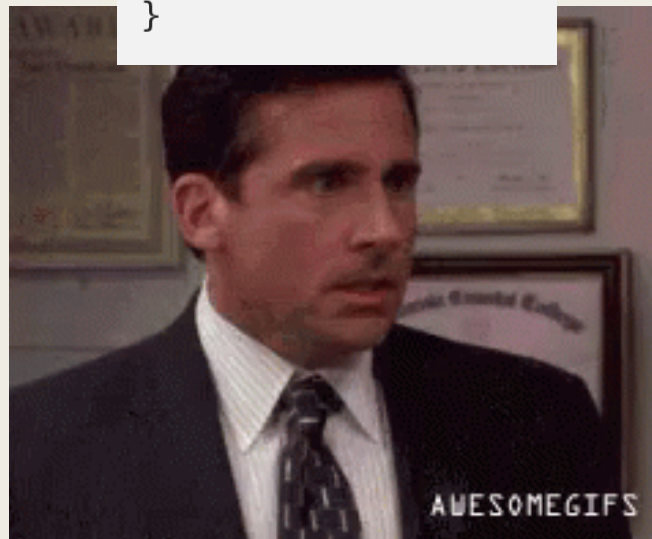
P0 Review: Commit Messages

- Actual commit messages from a student

- ...that's just 1 day's worth

Commits on Jan 13, 2018

Commits on Jan 13, 2018

Merge branch 'feature/sp4' into develop
committed 4 days ago

Add implementation of subproject D
committed 4 days ago

Merge branch 'release/0.2.0'
committed 4 days ago

Merge branch 'feature/sp3' into develop
committed 4 days ago

Drop one-letter words in subproject C
committed 4 days ago

Add implementation of subproject C
committed 4 days ago

Fix silly typo
committed 4 days ago

Merge branch 'feature/sp2' into develop
committed 4 days ago

Add subproject B implementation
committed 4 days ago

Merge branch 'release/0.1.0'
committed 5 days ago

Merge branch 'feature/sp1' into develop
committed 5 days ago

Add subproject A implementation
committed 5 days ago

Merge branch 'develop' into feature/sp1
committed 5 days ago

This repo contains various gyrations of word count implemented with Spark's Python API. Completed for CSCI8360: Data Science Practicum at the University of Georgia.

The project is divided into four subprojects. Given a set of documents, we can: 1. Perform a case-insensitive count of all words and report the forty most frequent words as a JSON dictionary (sp1.py) 2. Report the forty most-frequent words excluding stopwords (sp2.py) 3. Report the forty most-frequent words while stripping leading/trailing punctuation (sp3.py) 4. Report the top 5 words in each document based on their TF-IDF score (sp4.py)

Getting Started

These instructions will get you a copy of the project up and running on your local machine for development and testing purposes.

Prerequisites

This project uses Apache Spark. You'll need to have Spark installed on the target cluster.
The `SPARK_HOME` environment variable should be set, and the Spark binaries should be in your system path.

Dependencies are managed using the Conda package manager. You'll need to install Conda to get setup.

Installing Dependencies

The environment.yml file is used by Conda to create a virtual environment that includes all the project's dependencies (including Python!)

Navigate to the project directory and run the following command

```
conda env create
```

This will create a virtual environment named '   -p0". Activate the virtual environment with the following command

```
conda activate        -p0
```

After the environment has been activated, the subprojects can be run as follows (replace sp1.py with the script for whichever subproject you wish to run)

```
python sp1.py
```

Configuration variables

These scripts support user-defined configuration variables that tell the scripts the location of the text data and the location of the cluster on which to run.

The config.json file is ignored, so to get started, create a copy of `config.example.json` and rename it to `config.json`. The following variables should be set

- `APP_NAME` - If running on a cluster with a GUI, this name will show up while the job is running (defaults to '   -p0')
- `CLUSTER_URI` - The location of the cluster on which the jobs should be run (defaults to 'local')
- `DATA_LOCATION` - The local/remote directory, file name, or HDFS from which the text files should be read (defaults to 'testdata')

Built With

- Python 3.6
- Apache Spark
- PySpark - Python API for Apache Spark
- Conda - Package Manager

Contributing

There are no specific guidelines for contributing. If you see something that could be improved, send a pull request! If you think something should be done differently (or is just-plain-broken), please create an issue.

Versioning

This project uses the GitFlow workflow to organize branches and "releases".

Authors

-

See the contributors file for details.

License

This project is licensed under the GNU GPL v3 - see the LICENSE.md file for details

P0 Review: Tickets

Filters ▾ | 🔍 is:issue is:open | Labels | Milestones | New issue

☐ ⊙ **1 Open** ✓ 6 Closed | Author ▾ | Labels ▾ | Projects ▾ | Milestones ▾ | Assignee ▾ | Sort ▾

☐ ⊙ **Subproject C receiving abnormally low score** `bug` | 💬 2
#7 opened 4 days ago by

P0 Review

- Average score on AutoLab as a function of hours-to-deadline

- Yes, R^2 isn't great

- But seriously: *submit early, submit often*