# Debugging Machine Learning in Production

Shreya Shankar

February 10, 2021

# Outline

# My background

- BS and MS from Stanford Computer Science (systems and AI focus)
- Did research in adversarial machine learning and deep learning robustness at Google Brain
- First ML engineer at an applied ML startup
  - Worked with terabytes of time series data
  - Helped build infrastructure for large-scale machine learning and data analytics
  - Responsibilities spanned recruiting, SWE, ML, product, and more

# Evolution of my interests

- In an academic setting, trained many models and cared a lot about deep learning robustness
  - Fairness
  - Generalizability to unknown distributions
  - Security
- In industry, we want to train few models but do *lots* of inference
- What happens beyond the validation or test set?

# The depressing truth about ML IRL

- 87% of data science projects don't make it to production[1]
- Data in the "real world" is not necessarily clean and balanced, like canonical benchmark datasets (ex: ImageNet)
- Data in the "real world" is always changing
- Showing high performance on a fixed train and validation set $\neq$ consistent high performance when that model is deployed

[1] https://venturebeat.com/2019/07/19/why-do-87-of-data-science-projects-never-make-it-into-production/

# Today's talk

- *Not* about how to improve model performance on a validation or test set
- *Not* about feature engineering, machine learning algorithms, or data science
- Discussing:
    - Collaboration behind building ML to deliver business value
    - Components of production ML pipelines
    - Case study of challenges faced post-deployment
- Somewhat motivating MLOps as a discipline

# Many components in production ML pipelines

- Different kinds of people
  - Not thought about as much in ML academia
- Different kinds of technical tools
  - ML academia thinks more (but not enough) about this

# "People" overview

- Lots of stakeholders and many technical dependencies (data, model, evaluation metrics, etc)
  - Different stakeholders speak different languages
  - Domain experts are key[2]
  - Changing data and business requirements
- Modeling is just one part of the pipeline
  - Owned by an ML practitioner or a data scientist
- Models are treated as software
  - Data scientists and ML practitioners may need to manage models they did not create

[2] Shankar, *Get rid of AI Saviorism*.

# "People" overview

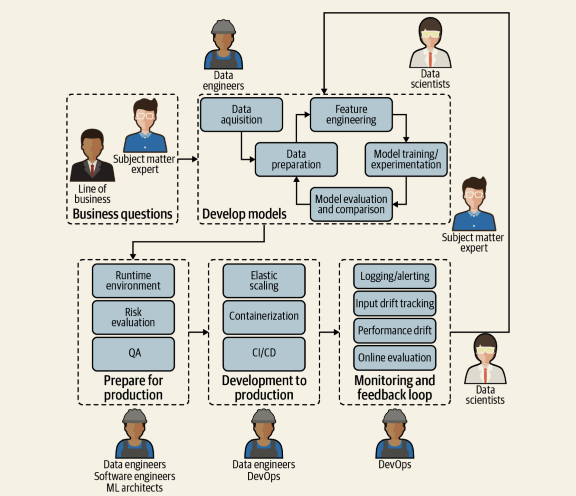
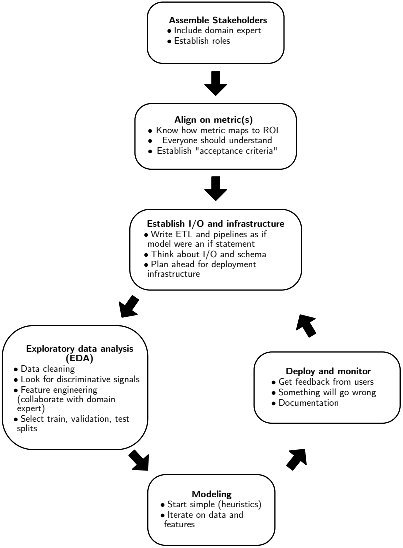


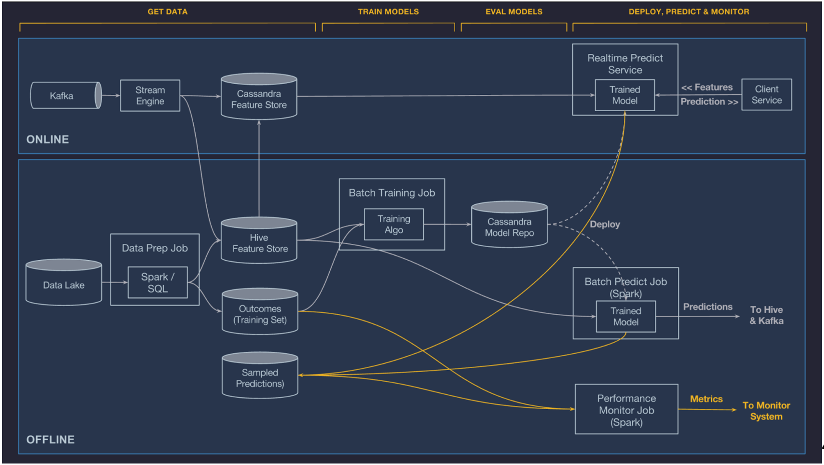

*Figure 1-3. The realistic picture of a machine learning model life cycle inside an average organization today, which involves many different people with completely different skill sets and who are often using entirely different tools.* [3]

[3] Treveil et al., *Introducing MLOps: How to Scale Machine Learning in the Enterprise*.

# Life cycle of an ML project



# Technical system overview



[4]

[4] Hermann and Del Balso, *Meet Michelangelo: Uber's Machine Learning Platform*.

# Technical system requirements

- Scalability and reproducibility of utmost importance
- Versioning all data, models, artifacts
  - Train, evaluation, and test sest
  - Model binary
  - Model parameters and hyperparameters
  - Metrics
  - etc.
- Creating a prediction is easy. "Backtracing" a prediction can be *very* challenging.
  - One ML pipeline can involve retraining and restaging multiple model binaries
  - Every bullet point above practically has its own tool
  - Makes debugging hard!

# Example of what happens post-"deployment"

- Simple toy example using the publicly available NYC Taxicab Trip Data[5]
- Tabular dataset where each record corresponds to one taxi ride
  - Pickup and dropoff times
  - Number of passengers
  - Trip distance
  - Pickup and dropoff location zones
  - Fare, toll, and tip amounts
- Monthly files from Jan 2009 to June 2020
- Each month is about 1 GB of data $\rightarrow$ over 100 GB

[5] https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

# Case study description

- Using publicly available NYC Taxicab Trip Data (data in public S3 bucket `s3://nyc-tlc/trip data/`)
- For this exercise, we will train and "deploy" a model to predict whether the user gave a large tip
- Using Python, Dask, RAPIDS cuML, Saturn Cloud (free trial of compute)
- Goal of this exercise is to demonstrate what can go wrong post-deployment
  - Even if you train and evaluate soundly in an "offline" setting, you can still run into lots of problems!

# Learning

- Let $X$ be the train covariate matrix (features) and $y$ be the train target vector (labels)
  - $X_i$ represents an $n$-dim feature vector corresponding to the $i$th ride and $y_i \in [0, 1]$ where 1 represents the rider tipping more than 20% of the total fare
- Want to learn classifier $f$ such that $f(X_i) \approx y_i$ for all $X_i \in X$
- We do not want to *overfit*, meaning $F_1(X_{train}) \approx F_1(X_{test})$
- We will use a `RandomForestClassifier` with basic hyperparameters from `cuml.dask.ensemble`
  - Very similar to `sklearn` API
  - `n_estimators=100, max_depth=10` copied from some Github repo linked in some Medium post somewhere (cannot find)

# Training and evaluation

- Train on January 2020, validate on February 2020, simulate deployment March 1 2020
- No hparam tuning so no hold-out validation set here
- We will measure $F_1$ score for our metric
  - precision $= \frac{\text{number of true positives}}{\text{number of records predicted to be positive}}$
  - recall $= \frac{\text{number of true positives}}{\text{number of positives in the dataset}}$
  - $F_1 = \frac{2*\text{precision}*\text{recall}}{\text{precision}+\text{recall}}$
- Higher $F_1$ score is better, want to have low false positives and false negatives

# Training code snippet

```
target_col = "high_tip"

taxi_jan = dask_cudf.read_csv(
    "s3://nyc-tlc/trip data/yellow_tripdata_2020-01.csv",
    parse_dates=["tpep_pickup_datetime", "
                                    tpep_dropoff_datetime"],
    storage_options={"anon": True},
    assume_missing=True,
)

# Featurize and fit model
taxi_train = preprocess(df=taxi_jan, target_col=target_col)
rfc = RandomForestClassifier(n_estimators=100, max_depth=10,
                                ignore_empty_partitions=True)
rfc.fit(taxi_train[features], taxi_train[target_col])
```

# Training set evaluation

```
preds = rfc.predict_proba(taxi_train[features])[1]
print(f'F1: {f1_score(taxi_train[target_col].compute().
                                    to_array(), preds.round().
                                    compute().to_array())}')
```

**Output**

F1: 0.6681650475249482

# Test set evaluation

```
taxi_feb = dask_cudf.read_csv(
    "s3://nyc-tlc/trip data/yellow_tripdata_2020-02.csv",
    parse_dates=["tpep_pickup_datetime", "
                                   tpep_dropoff_datetime"],
    storage_options={"anon": True},
    assume_missing=True,
)

# Featurize and evaluate
taxi_test = preprocess(taxi_feb, target_col=target_col)
preds = rfc.predict_proba(taxi_test[features])[1]
print(f'F1: {f1_score(taxi_test[target_col].compute().
                               to_array(), preds.round().
                               compute().to_array())}')
```

**Output**

F1: 0.6658098920024954

# Caveats

- We did no hyperparameter tuning (should hold out a validation set if we are going to do this)
- It is unclear if the classification problem we are solving is actually valuable
- It is unclear if such an $F_1$ score maps to any ROI
- For the sake of the tutorial, we will move on to "deployment"

# "Deployment" simulation

- It is March 2020, and we release our model into the world
  - There are a lot of infra challenges around this I will not get into
  - In practice, we experience *data lag*, where data comes into our databases after some lag
  - This means that on March 1, we might not have data until February 29. So maybe this deployment is in mid-March.
- Streaming or batched inference?
  - Here, simulate inference on each record as it comes in a stream
  - Here, compute metrics in batch
  - In practice, infrastructure considerations

# Challenge: lag

- In this example we are simulating a "live" deployment on historical data, so we have no lag
- In practice there are many types of lag
  - Feature lag: our system only learns about the ride well after it has occurred
  - Label lag: our system only learns of the label (fare, tip) well after the ride has occurred
- When dealing with lag, the evaluation metric will inherently be lagging
- When dealing with lag, at training time the model cannot reflect most recent time

# Simulating live inference and evaluation

```
# Load and sort the march dataframe by date (ascending)

taxi_march = dask_cudf.read_csv(
    "s3://nyc-tlc/trip data/yellow_tripdata_2020-03.csv",
    parse_dates=["tpep_pickup_datetime", "
                                          tpep_dropoff_datetime"],
    storage_options={"anon": True},
    assume_missing=True,
)

taxi_inference = preprocess(taxi_march, target_col=
                                      target_col, start_date='2020-
                                      03-01', end_date='2020-03-31')
                                      .sort_values(by=['
                                      tpep_dropoff_datetime'],
                                      ascending=True).reset_index(
                                      drop=True)
taxi_inference['day'] = taxi_inference.tpep_dropoff_datetime
                                      .dt.day.to_dask_array()
```

# Compute $F_1$ scores

```
# Save predictions as a new column, compute "rolling" and "
                                    daily" F1 scores
taxi_inference['predicted_prob'] = rfc.predict_proba(
                                    taxi_inference[features])[1]
taxi_inference['prediction'] = taxi_inference['
                                    predicted_prob'].round().
                                    astype('int32')
taxi_inference['rolling_f1'] = f1_streaming(taxi_inference,
                                    target_col, 'prediction')
daily_f1 = taxi_inference.groupby('day').apply(
                                    get_daily_f1_score, meta={'day
                                    ': int, 'rolling_f1': float, '
                                    daily_f1': float})
```

# Inspect $F_1$ scores at the end of every day

```
daily_f1.sort_values(by='day').compute()
```

## Output

| | day | rolling_f1 | daily_f1 |
|---|---|---|---|
| 178123 | 1 | 0.576629 | 0.576629 |
| 370840 | 2 | 0.633320 | 0.677398 |
| 592741 | 3 | 0.649983 | 0.675877 |
| ... | | | |
| 1514827 | 28 | 0.659934 | 0.501860 |
| 1520358 | 29 | 0.659764 | 0.537860 |
| 1529847 | 30 | 0.659530 | 0.576178 |

# Deeper dive into March $F_1$ scores

| | day | rolling_f1 | daily_f1 |
|---|---|---|---|
| 178123 | 1 | 0.576629 | 0.576629 |
| 370840 | 2 | 0.633320 | 0.677398 |
| 592741 | 3 | 0.649983 | 0.675877 |
| ... | | | |
| 1507234 | 27 | 0.660228 | 0.576993 |
| 1514827 | 28 | 0.659934 | 0.501860 |
| 1520358 | 29 | 0.659764 | 0.537860 |
| 1529847 | 30 | 0.659530 | 0.576178 |

- Large discrepancy between rolling metric and daily metric
- What you monitor is important – daily metric significantly drops towards the end
- Visible effect of COVID-19 on the data

# Evaluating the same model on following months

```
months = ['2020-04', '2020-05', '2020-06']
month_dfs = {}

for month in months:
    taxi_test = preprocess(...)
    preds = rfc.predict_proba(taxi_test[features])[1]
    print(month)
    print(f'F1: {f1_score(taxi_test[target_col].compute().
                                to_array(), preds.round().
                                compute().to_array())}')
```

## Output

```
2020-04
F1: 0.5714705472990737
2020-05
F1: 0.5530868473460906
2020-06
F1: 0.5967621469282887
```

# Challenge: distribution shift

- Data "drifts" over time, and models will need to be retrained to reflect such drift
- Open question: how often do you retrain a model?
  - Retraining adds complexity to the overall system (more artifacts to version and keep track of)
  - Retraining can be expensive in terms of compute
  - Retraining can take time and energy from people
- Open question: how do you know when the data has "drifted?"

# Challenge: quantifying distribution shift

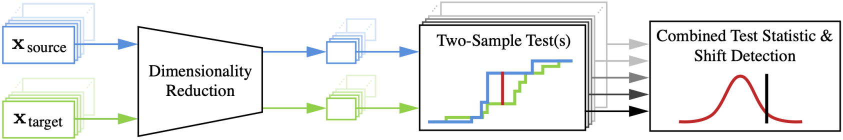



Figure 1: Our pipeline for detecting dataset shift. Source and target data is fed through a dimensionality reduction process and subsequently analyzed via statistical hypothesis testing. We consider various choices for how to represent the data and how to perform two-sample tests. [6]

[6] Rabanser, Günnemann, and Lipton, *Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift*.

# Challenge: quantifying distribution shift

- We only have 11 features, so we will skip the dimensionality reduction step
- "Multiple univariate testing seem to offer comparable performance to multivariate testing" (Rabanser et al.)
  - Maybe this is specific to their MNIST and CIFAR experiments
  - Regardless, we will employ multiple univariate testing
- For each feature, we will run a 2-sided Kolmogorov-Smirnov test
  - Continuous data, non-parametric test
  - Compute the largest difference

$$Z = \sup_z |F_p(z) - F_q(z)|$$

where $F_p$ and $F_q$ are the empirical CDFs of the data we are comparing

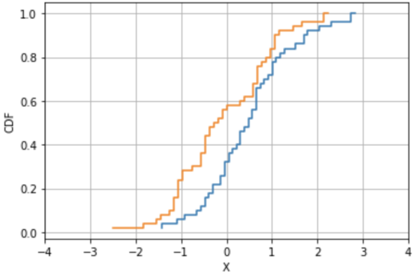
# Kolmogorov-Smirnov test made simple

- For each feature, we will run a 2-sided Kolmogorov-Smirnov test
  - Continuous data, non-parametric test
  - Compute the largest difference

$$Z = \sup_z |F_p(z) - F_q(z)|$$

where $F_p$ and $F_q$ are the empirical CDFs of the data we are comparing

- Example comparing two random normally distributed PDFs



# Challenge: quantifying distribution shift

Let's compare January and February datasets.

```
statistics = []
p_values = []

for feature in features:
    statistic, p_value = stats.ks_2samp(taxi_train[feature].
                                         compute().to_pandas(),
                                         taxi_test[feature].compute
                                         ().to_pandas())
    statistics.append(statistic)
    p_values.append(p_value)

# Make a dataframe of the ks test results
comparison_df = pd.DataFrame(data={'feature': features, '
                              statistic': statistics, '
                              p_value': p_values})
```

# Challenge: quantifying distribution shift

```
comparison_df.sort_values(by='p_value', ascending=True).
                                    head(11)
```

## Output

| | feature | statistic | p_value |
|---|---|---|---|
| 0 | pickup_weekday | 0.046196 | 0.000000e+00 |
| 2 | work_hours | 0.028587 | 0.000000e+00 |
| 6 | trip_time | 0.017205 | 0.000000e+00 |
| 7 | trip_speed | 0.035415 | 0.000000e+00 |
| 1 | pickup_hour | 0.009676 | 8.610133e-258 |
| 5 | trip_distance | 0.005312 | 5.266602e-78 |
| 8 | PULocationID | 0.004083 | 2.994877e-46 |
| 9 | DOLocationID | 0.003132 | 2.157559e-27 |
| 4 | passenger_count | 0.002947 | 2.634493e-24 |
| 10 | RatecodeID | 0.002616 | 3.047481e-19 |
| 3 | pickup_minute | 0.000702 | 8.861498e-02 |

# Challenge: quantifying distribution shift

- For our "offline" train and evaluation sets (January and February), we get *extremely* low p-values!
- Although this method works well when the data has "drifted" and we experience a performance drop, it can also annoyingly flag drift even when the metric is unchanged
- This method can have a high "false positive rate"
  - In my experience, this method has flagged distributions as "significantly different" more than I want it to
  - In the era of "big data" (we have millions of data points), p-values are not useful to look at[7]

[7] Lin, Lucas, and Shmueli, "Too Big to Fail: Large Samples and the p-Value Problem".

# Challenge: establishing more confidence in our training/evaluation pipelines

- Especially with time series data, no use validating just one model
- We want to validate the *process* of training and evaluating models
  - Rolling train/validation sets?
  - Only promote an "architecture" to production if each trained model achieves acceptance criteria on its corresponding validation set?
  - Can get complicated quickly
- Automated retraining and tracing lineage[8]

[8] Hermann and Del Balso, *Scaling Machine Learning at Uber with Michelangelo*.

# Challenge: collaboration

- Not everyone is trained in "ML-speak"
  - "When I say 'algorithm,' I mean a series of steps that I control. Maybe this is what other people also mean, and there is confusion because different stakeholders control different things – for example, the 'algorithm' for me is how I train a model; the 'algorithm' for a client is the entire end-to-end ML product that shows up on their computer in true Software-as-a-Service form."[9]
- ML is inherently "probabilistic"
  - We know we will get some predictions wrong, but we do not know what we will get wrong
  - Hard to communicate this to clients who are mainly experienced in SaaS, not ML

[9] Shankar, *Predictive Modeling: A Retrospective*.

# Areas for future work

- Only scratched the surface with these challenges
- Many more of these problems around deep learning
  - Embeddings as features – if someone updates the upstream embedding model, do all the data scientists downstream need to immediately change their models?
  - "Underspecified" pipelines can pose threats[10]
- Tooling is a can of worms
  - $\geq$ 284 MLOps tools, $\geq$ 180 startups[11]
  - I am honestly just fatigued thinking about more tools at this point
- At the minimum, need a culture shift
  - Assign less hype to fancy models; celebrate "data"ing just as much as modeling
  - Lots to learn from software (modularity, debugging, etc.)

[10] D'Amour et al., *Underspecification Presents Challenges for Credibility in Modern Machine Learning*.

[11] Huyen, *MLOps Tooling Landscape*.

# Miscellaneous

- Code for this talk:
  `https://github.com/shreyashankar/debugging-ml-talk`
- Contact info
  - Twitter: @sh_reya
  - Writing: shreya-shankar.com
  - Email: shreya@cs.stanford.edu
- Thank you!

# References


D'Amour, Alexander et al. *Underspecification Presents Challenges for Credibility in Modern Machine Learning*. 2020. arXiv: 2011.03395 [cs.LG].

Hermann, Jeremy and Mike Del Balso. *Meet Michelangelo: Uber's Machine Learning Platform*. 2017. URL: https://eng.uber.com/michelangelo-machine-learning-platform/.

— .*Scaling Machine Learning at Uber with Michelangelo*. 2018. URL: https://eng.uber.com/scaling-michelangelo/.

Huyen, Chip. *MLOps Tooling Landscape*. 2020. URL: https://huyenchip.com/2020/12/30/mlops-v2.html.

Lin, Mingfeng, Henry Lucas, and Galit Shmueli. "Too Big to Fail: Large Samples and the p-Value Problem". In: *Information Systems Research* 24 (Dec. 2013), pp. 906–917. DOI: 10.1287/isre.2013.0480.

Rabanser, Stephan, Stephan Günnemann, and Zachary C. Lipton. *Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift*. 2019. arXiv: 1810.11953 [stat.ML].

Shankar, Shreya. *Get rid of AI Saviorism*. 2020. URL: https://www.shreya-shankar.com/ai-saviorism/.

— .*Predictive Modeling: A Retrospective*. 2021. URL: https://www.shreya-shankar.com/predictive-modeling-retrospective/.

Treveil, M. et al. *Introducing MLOps: How to Scale Machine Learning in the Enterprise*. O'Reilly Media, Incorporated, 2020. ISBN: 9781492083290. URL: https://books.google.com/books?id=fR64zQEACAAJ.