Building systems to monitor model health in production systems

Mohammed Ridwanul Islam (@mohammedri_)
Product Manager @Dessa

DESSA™

Hello DataCouncil 👋

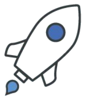
Focus on building SDK and infrastructure tools for Machine Learning Engineers

Previously: Data Platform Engineering @Shopify

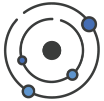
DESSA™

We build Continuous Delivery Tools for Machine Learning

Foundations suite

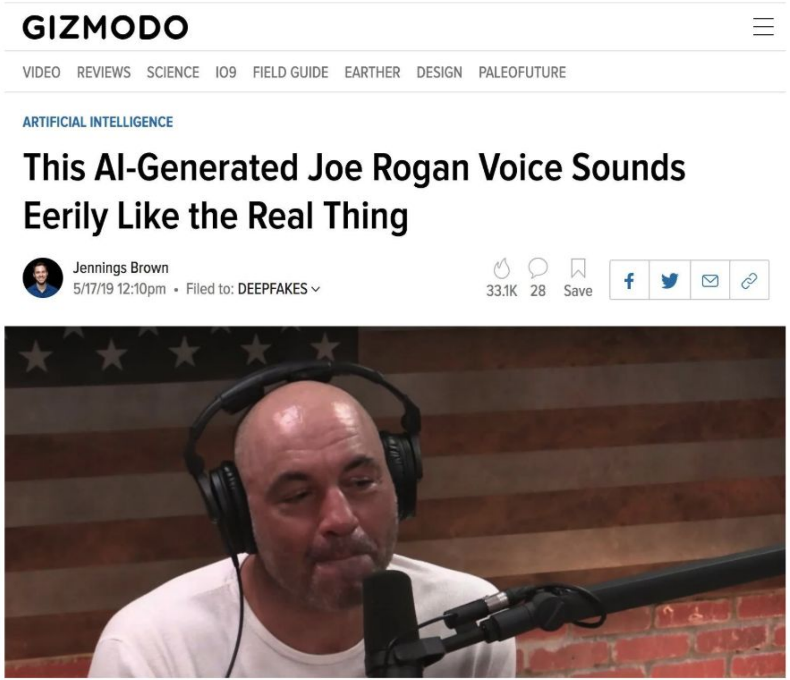


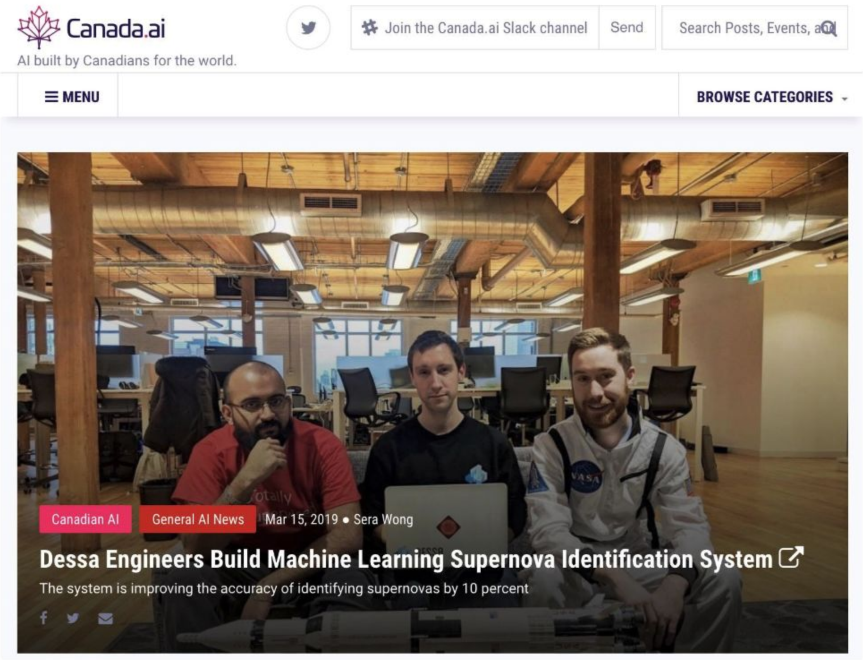
Foundations Atlas
For Machine Learning Development across teams



Foundations Orbit
For post-production model monitoring & maintenance

We also have fun with Machine Learning





DESSA™

Agenda

- Why do we care about ML monitoring?
- Why current systems are not good enough to monitor ML?
- What do we monitor?
- How we do monitoring at Dessa?
- Our systems approach and design
- Q&A

Why care about Model monitoring?

Personal goal: see more and more companies *RELEASE* products that are powered by Machine Learning *SUCCESSFULLY*

Lots of companies are building models but few are able to move them out of POC.

Why?

- Teams are getting burnt after putting models in production
    - Real life-example: F100 Enterprise
    - Lack of automated ways of maintaining SLAs - reduces trust on models in production

- Models drop in performance rapidly & managing more than 3 models per team becomes quite difficult manually

- Monitoring, diagnosis, rollback and rebuilding strategies are currently quite ad-hoc and not scalable to many models

Data Infrastructure Engineers responsible for building strong monitoring and performance management systems that are easy to use by Data Scientists.

What usually goes wrong

Accuracy (or your metric of choice) declines without explanation 😩

- Is this a bug in the upstream data that the model is consuming?
  - If so, which part of the pipeline is broken? Is it the feature engineering steps?
- Is this a bug in the training source code?
  - Was a new model deployed?
  - Was a new model trained with a drastically different dataset?
  - Are we overfitting / underfitting? Has the model gone stale?
- Has the wrong model been put into production? How do I rollback?
- Is there an adversarial user?
- Where do I start? Who is responsible? Who knows this model / dataset best?

Source: XKCD, https://xkcd.com/1838/

Solution: continuous testing

- Create tests & alerts on upstream datasets & models for quality
- Centralized model management with efficient rollback strategies
- Evaluate models using real outcomes and maintain pre-defined SLAs through re-training & recalibration
- Make it easy for your scientists!

Monitoring needs for ML

Traditional software vs. ML

- Source code changes (track using CI tools, build pipelines etc.)
- Infrastructure / Network Changes (track using DataDog, Splunk et. al.)

- Source code changes
- Infrastructure / Network Changes
- Bugs in the Data Pipeline
- Dataset shifts
    - Co-variate shifts
    - Prior probability shifts
    - Concept Drift
    - Uncertainty
    - Explainability
    - + more!

What needs to be monitored?

Bugs in the data pipeline

- Upstream changes impacting downstream data consumption e.g.
    - Schema changes:
        - Type changes
        - Column name changes
        - New features added/deleted
    - Systematic errors introduced

Bugs in the data pipeline - monitoring strategies

- Schema checks:
  Compare schemas of incoming data to a reference dataset

- Special Value Checks:
  E.g. % of nulls in a column, NaNs, -99999, -1, etc.

- Min-max ratio checks

- Uniqueness checks

- Special knowledge checks

Dataset Drift

Input Drift

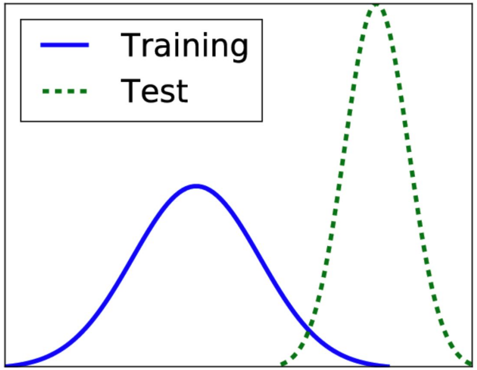
Concept Drift

Input Drift

Input shift refers to the change in the distribution of the input variables present in the training and inference data

1. May or may not correspond to a drop in model performance
2. Indicates either a shift in the underlying data distribution or bugs in the data pipeline
3. Monitoring input drift can help you be proactive in choosing better features in the future

$$p_{train}(X) \neq p_{inference}(X)$$



On the Reduction of Biases in Big Data Sets for the Detection of Irregular Power Usage - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Example-of-covariate-shift-training-and-test-data-having-different-distributions_fig1_324168804 [accessed 12 Nov, 2019]

Input Drift - various monitoring strategies

1. Population Stability Index (PSI)
    a. Based on Kullback–Leibler (KL) Divergence
    b. Complex to interpret

2. L-Infiniti
    a. Much more easily interpretable e.g. “allow changes of only up to 1% for each (feature)” [1]

3. Similarity score
    a. Lots of other approaches - many papers have been written

[1] Eric Breck et. al., *Proceedings of SysML* (2019) (to appear) DATA VALIDATION FOR MACHINE LEARNING, 2019

Input Drift - various monitoring strategies

Choose the metric that is easiest to understand and use.

"A first-cut solution here might be to use typical distance metrics such as KL divergence or cosine similarity and fire an alert only if the metric crosses a threshold. The problem with this solution is that product teams have a hard time understanding the natural meaning of the metric and thus tuning the threshold to avoid false positives…"

...(distribution distance metrics instead) has a simple natural interpretation of the largest change in probability for a value in the two distributions, which makes it easier for teams to set a threshold (e.g., "allow changes of only up to 1% for each value")

Concept Drift

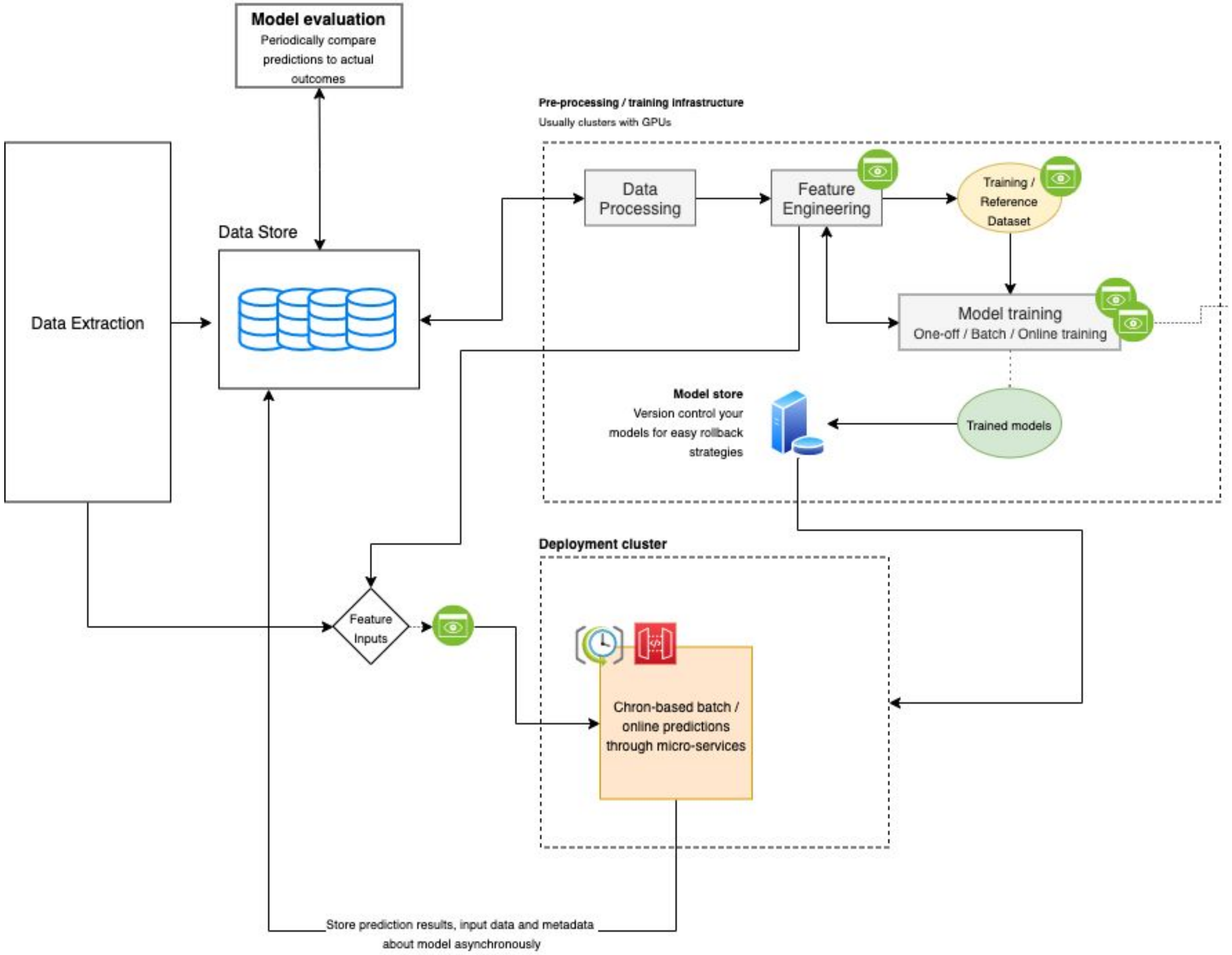
Occurs when the function mapping input to target has changed

1. Corresponds to a drop in model performance
2. Indicates that a model may need to be trained
3. Change can take many forms[2]:
   a. A gradual change over time.
   b. A recurring or cyclical change.
   c. A sudden or abrupt change.

$$p_{train}(Y|X) \neq p_{inference}(Y|X)$$

Source:
[2]: Jason Brownlee, https://machinelearningmastery.com/gentle-introduction-concept-drift-machine-learning/

We need to be able to codify expectations from our datasets


Model evaluation
Periodically compare predictions to actual outcomes
Pre-processing / training infrastructure
Usually clusters with GPUs
Data Extraction
Data Store
Data Processing
Feature Engineering
Training / Reference Dataset
Model training
One-off / Batch / Online training
Model store
Version control your models for easy rollback strategies
Trained models
Deployment cluster
Feature Inputs
Chron-based batch / online predictions through micro-services
Store prediction results, input data and metadata about model asynchronously

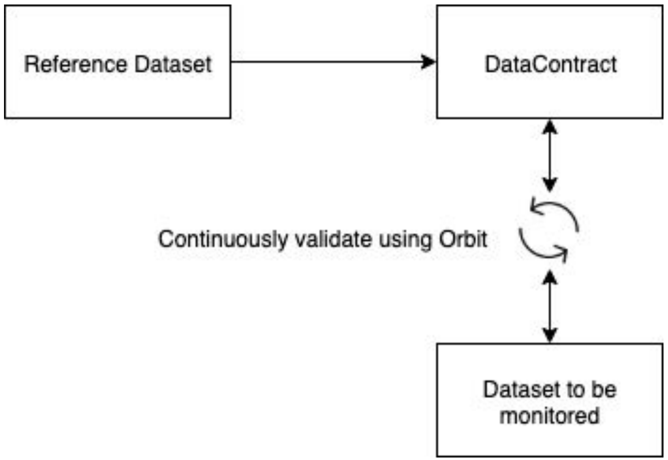
Orbit = Python SDK + Scheduler + GUI

DataContracts as expectations using the Orbit SDK

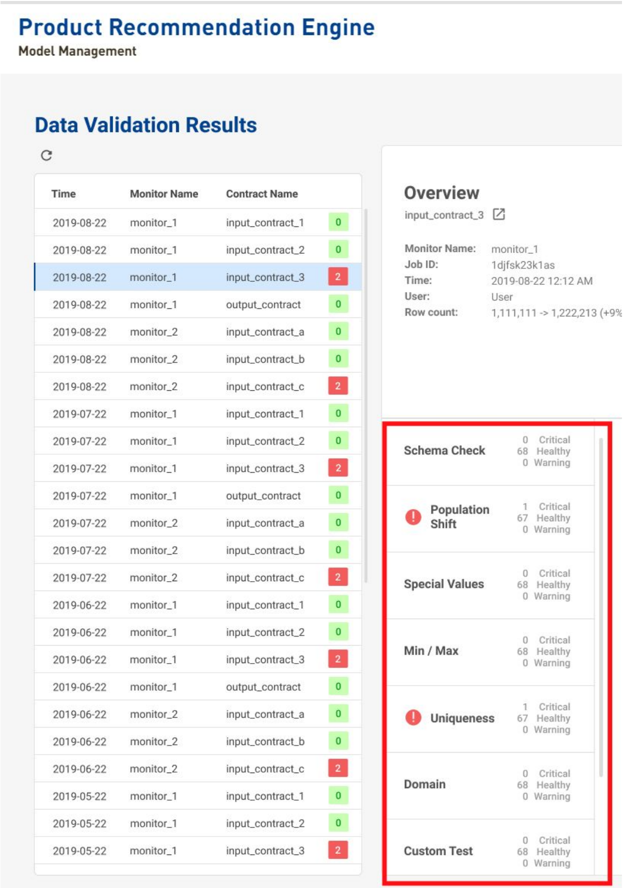
To monitor bugs in the pipeline and Dataset Drifts

Orbit Python SDK - DataContracts

- DataContracts = Data Expectations
- Every DataContract object ~= configurable metadata store, versioned & immutable
- Uses reference dataframe to compare incoming data and generate validation reports
- Configure and set expectations for any *reference* data set



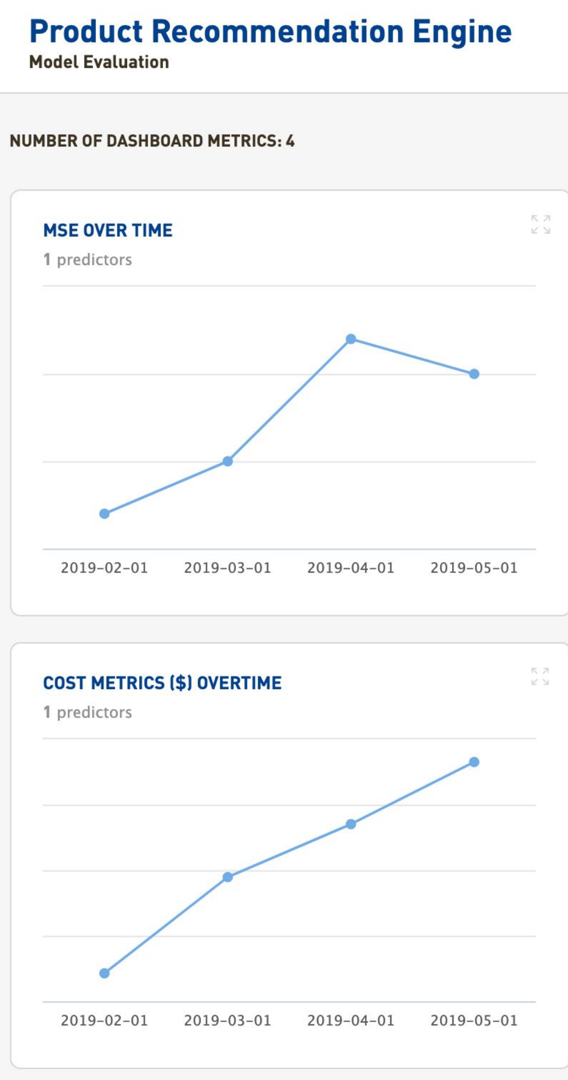
```python
from foundations_orbit import DataContract

data_contract = DataContract(data_contract_name,
reference_dataframe)

data_contract.save(dir_path_to_save) # Save contracts in a
registry

contract.distribution_test.configure(["col1","col2"],
threshold=0.2, method=my_custom_method())
contract.distribution_test.configure(["col1","col3"],
threshold=0.1, method='l-infinity')
contract.special_value_test.configure(["col1"],
thresholds={np.nan: 0.1, -1: 0.2})

validation_report =
data_contract.validate(dataframe_to_validate)
```

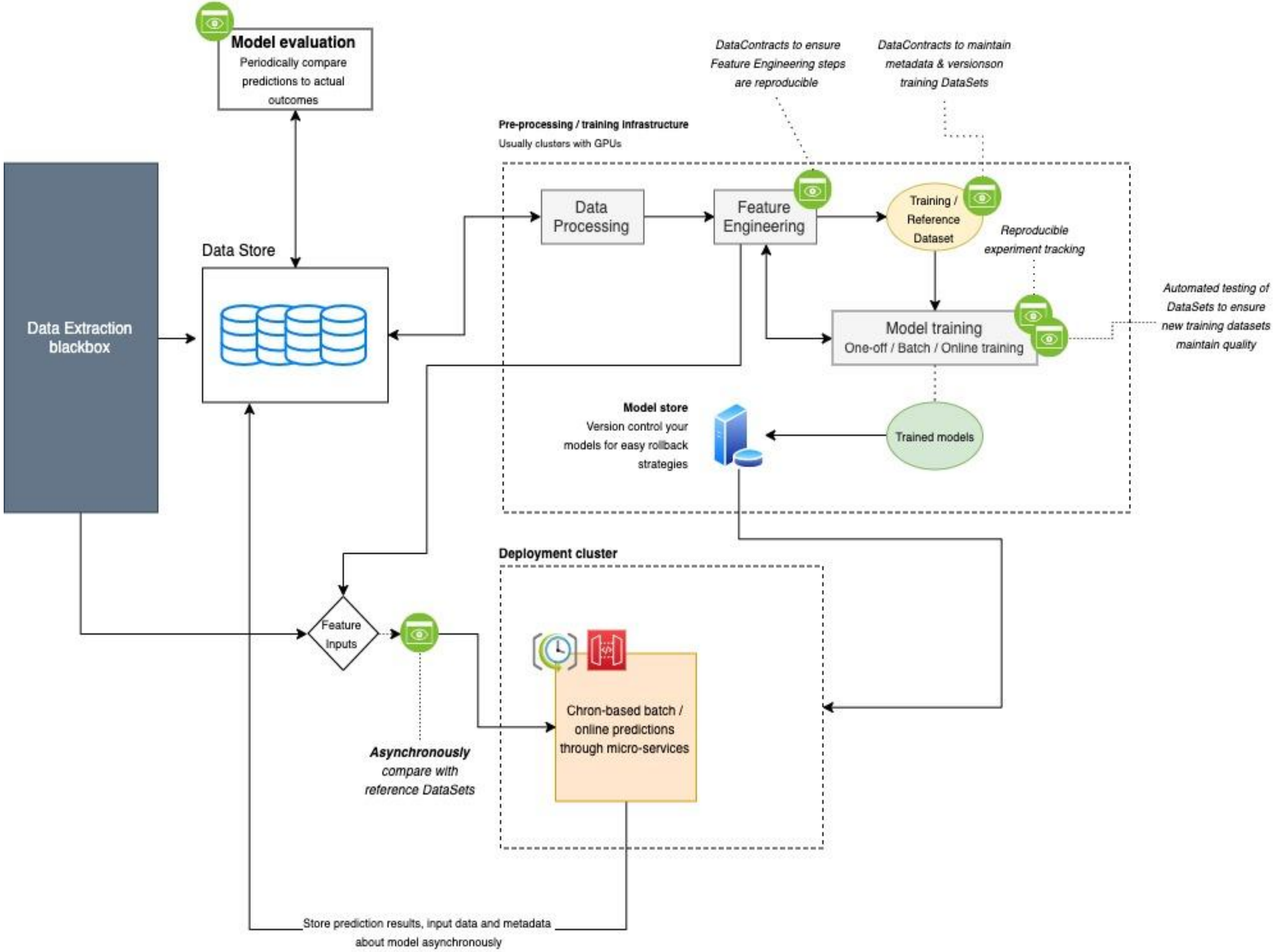


Concept Drift - monitoring strategies

- Continuously monitor the metric you are optimizing for
- Continuously evaluate metric with actual outcome

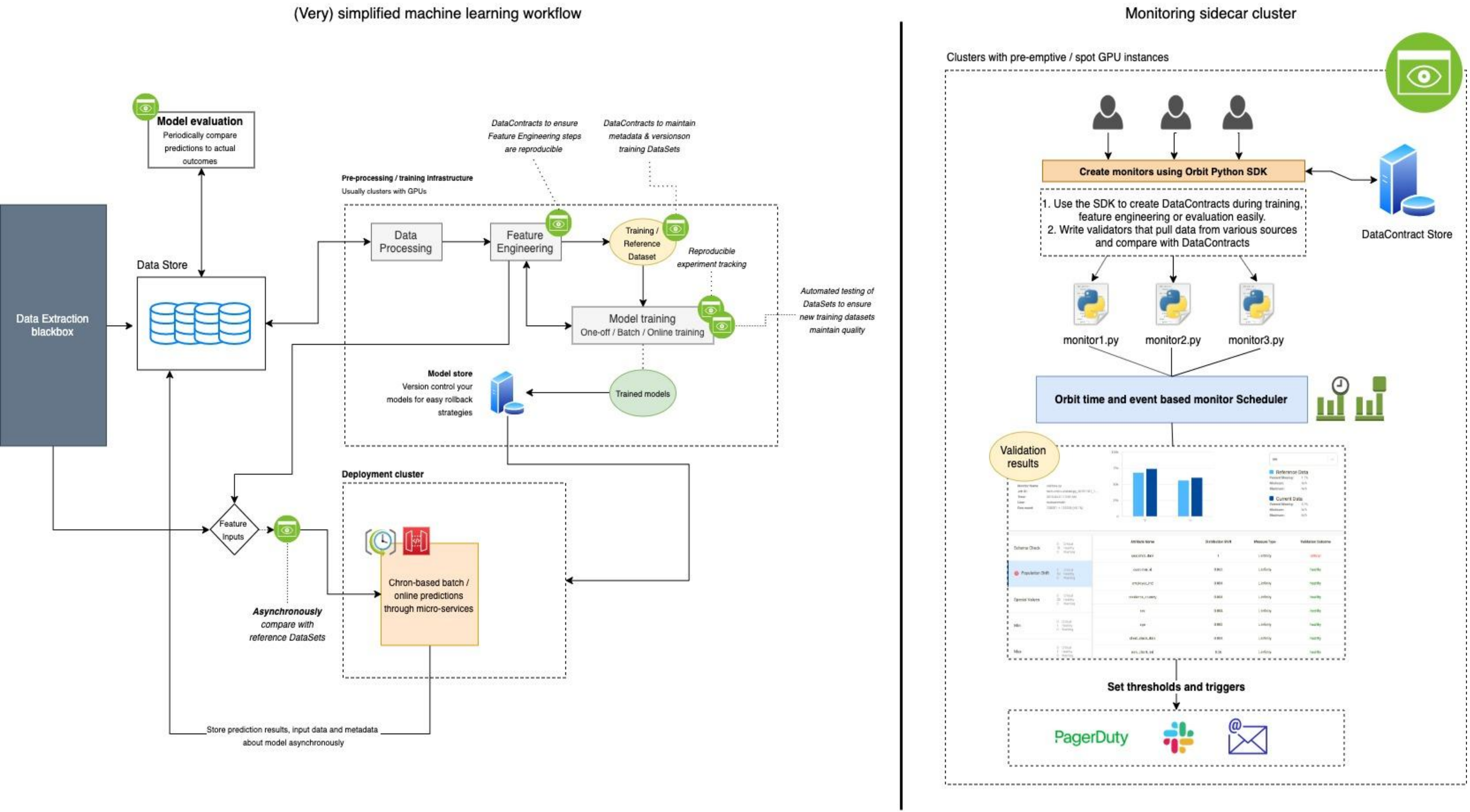
```
## Logic to calculate these metrics

foundations.track_production_metrics("accuracy",
{str(eval_date): accuracy})
foundations.track_production_metrics("roc_auc",
{str(eval_date): roc_auc})
foundations.track_production_metrics("revenue",
{str(eval_date): revenue})
foundations.track_production_metrics("n_active_custs",
{str(eval_date): n_active_custs})
```




Model evaluation
Periodically compare predictions to actual outcomes
Data Store
Data Extraction blackbox
Pre-processing / training infrastructure
Usually clusters with GPUs
DataContracts to ensure Feature Engineering steps are reproducible
DataContracts to maintain metadata & versionson training DataSets
Data Processing
Feature Engineering
Training / Reference Dataset
Reproducible experiment tracking
Automated testing of DataSets to ensure new training datasets maintain quality
Model training
One-off / Batch / Online training
Model store
Version control your models for easy rollback strategies
Trained models
Deployment cluster
Feature Inputs
Asynchronously compare with reference DataSets
Chron-based batch / online predictions through micro-services
Store prediction results, input data and metadata about model asynchronously

Other criterias for our monitoring system

- Sidecar + Async architecture
- Built-in-standard monitoring such as schema checks, min-max’s, L-infiniti etc.
- Ability to define custom monitors through lambda functions
- Chron based scheduler for monitoring
- Event based monitoring
- Dashboards for easy viewing and the ability to set thresholds and alerts


(Very) simplified machine learning workflow
Monitoring sidecar cluster
Model evaluation
Periodically compare predictions to actual outcomes
DataContracts to ensure Feature Engineering steps are reproducible
DataContracts to maintain metadata & versionson training DataSets
Pre-processing / training infrastructure
Usually clusters with GPUs
Data Extraction blackbox
Data Store
Data Processing
Feature Engineering
Training / Reference Dataset
Reproducible experiment tracking
Automated testing of DataSets to ensure new training datasets maintain quality
Model training
One-off / Batch / Online training
Model store
Version control your models for easy rollback strategies
Trained models
Deployment cluster
Feature Inputs
Chron-based batch / online predictions through micro-services
Asynchronously compare with reference DataSets
Store prediction results, input data and metadata about model asynchronously
Clusters with pre-emptive / spot GPU instances
Create monitors using Orbit Python SDK
1. Use the SDK to create DataContracts during training, feature engineering or evaluation easily.
2. Write validators that pull data from various sources and compare with DataContracts
DataContract Store
monitor1.py
monitor2.py
monitor3.py
Orbit time and event based monitor Scheduler
Validation results
Set thresholds and triggers
PagerDuty

Data Scientists workflow

DESSA™

- Create a DataContract for a reference or training data set

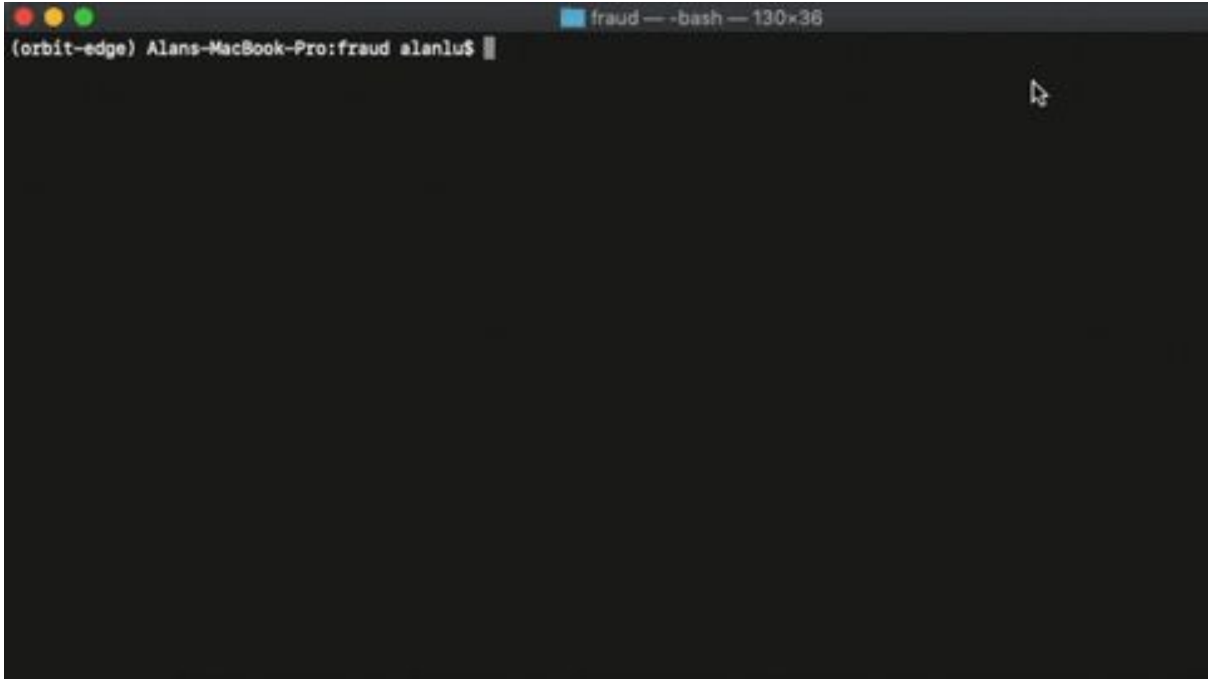
```python
# load all training data
train_dfs = []
for date in load_dates:
    date_without_time = date.date()
    # load one month of train data from a GCP bucket as a pandas dataframe
    df = db.load(f'{data_key}-labelled-{date_without_time}')
    # don't include customers who got promos in the training data
    train_df = df[df['promo'] == 0]
    train_df = train_df.drop(['promo'], axis=1)
    train_dfs.append(train_df)
train_df = pd.concat(train_dfs, axis=0)

# split features from target
y_train = train_df['churn']
x_train = train_df.drop(['cust_id', 'date', 'churn', 'predicted_churn_probability'], axis=1)

# drop nans
x_train = x_train.fillna(0)  # ensure the model can still train if NaN values are present

# train model
model = RandomForestClassifier()
model.fit(x_train, y_train)

# save model
pickle.dump(model, open("fitted_objects/model.pkl", "wb"))
```

- Place DataContract in validation pipelines separate as monitoring code

train_driver.py | train.py | model_eval.py | validate.py

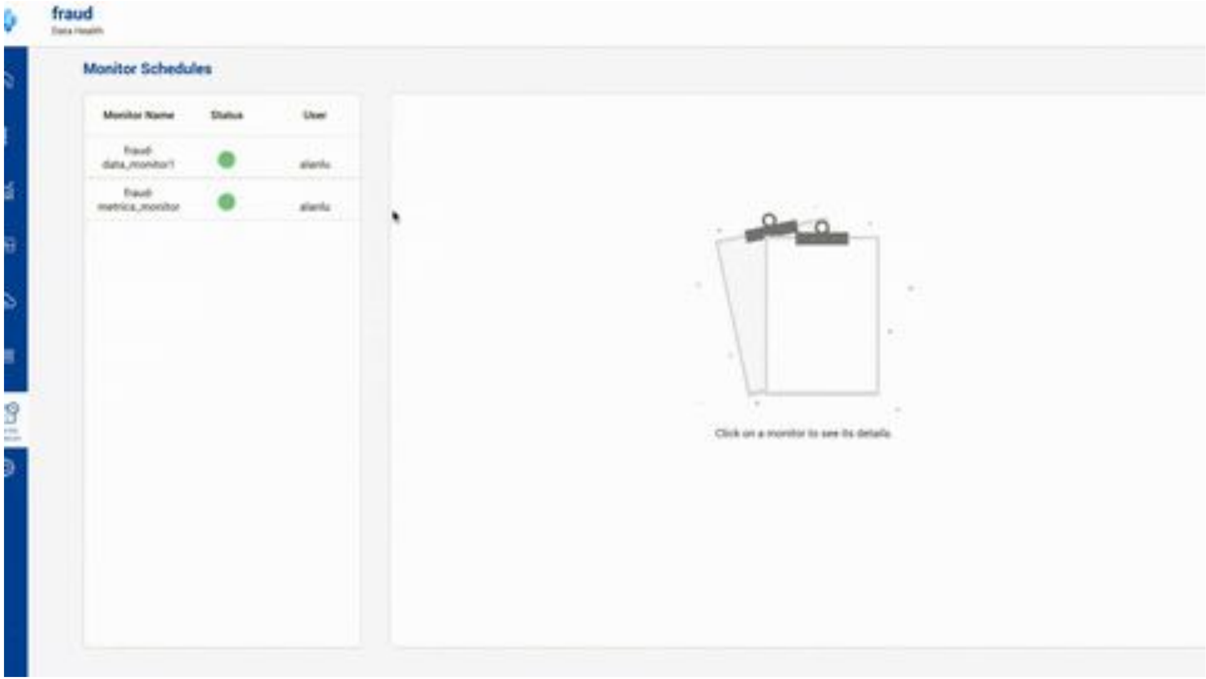
```python
import ...

def validate(inference_date):

    inference_date = inference_date.split(' ')[0]
    # load inference data from a GCP bucket as a pandas dataframe
    inference_df = db.load(f'{data_key}-inference-{inference_date}')
    # drop non-feature columns
    x_inference = inference_df.drop(['cust_id', 'date'], axis=1)
    x_inference = x_inference.fillna(0)

    # load model from local directory
    model = pickle.load(open("fitted_objects/model.pkl", "rb"))

    # run inference, and create a dataframe to store the predicted probabilities
    probs = model.predict_proba(x_inference)[:, 1]
    probs_df = pd.DataFrame(probs, columns=["predicted_churn_probability"])

    # save predictions to a GCP bucket
    db.save(f'{data_key}-predictions-{inference_date}', probs_df)
```

- Deploy validation pipelines w/ DataContracts to Orbit via a REST backend



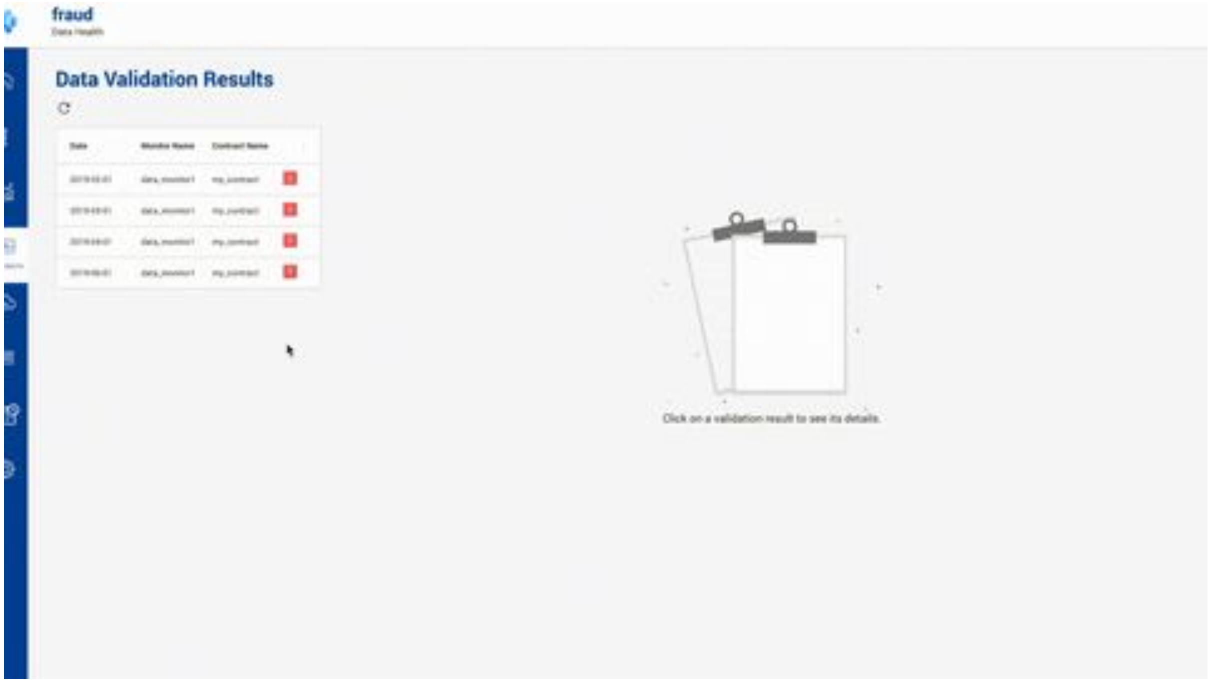
DESSA

- Trigger monitoring using chron scheduler or event based triggers e.g. when a new dataset available



DESSA™

- Dig into reports or set triggers on data quality



DESSA™

Summary

Thank you :)

@mohammedri_

DESSA™