Lowering the barrier for ML Monitoring

Wesley Boelrijk

PyData Eindhoven 2022

Who thinks monitoring ML models is important?

Who has implemented ML monitoring so far?

Who am I?

- Lead Machine Learning Engineer at Xccelerated (part of Xebia)
- Responsible for training junior-to-medior ML Engineers
- Currently implementing monitoring for computer vision at Port of Rotterdam

Why monitoring?

Why monitoring?

- problems will occur

Why monitoring?

- problems will occur
- identify them early

Why monitoring?

- problems will occur
- identify them early
- diagnose the cause

Why monitoring?

- problems will occur
- identify them early
- diagnose the cause
- analysis and performance reviews

Tools

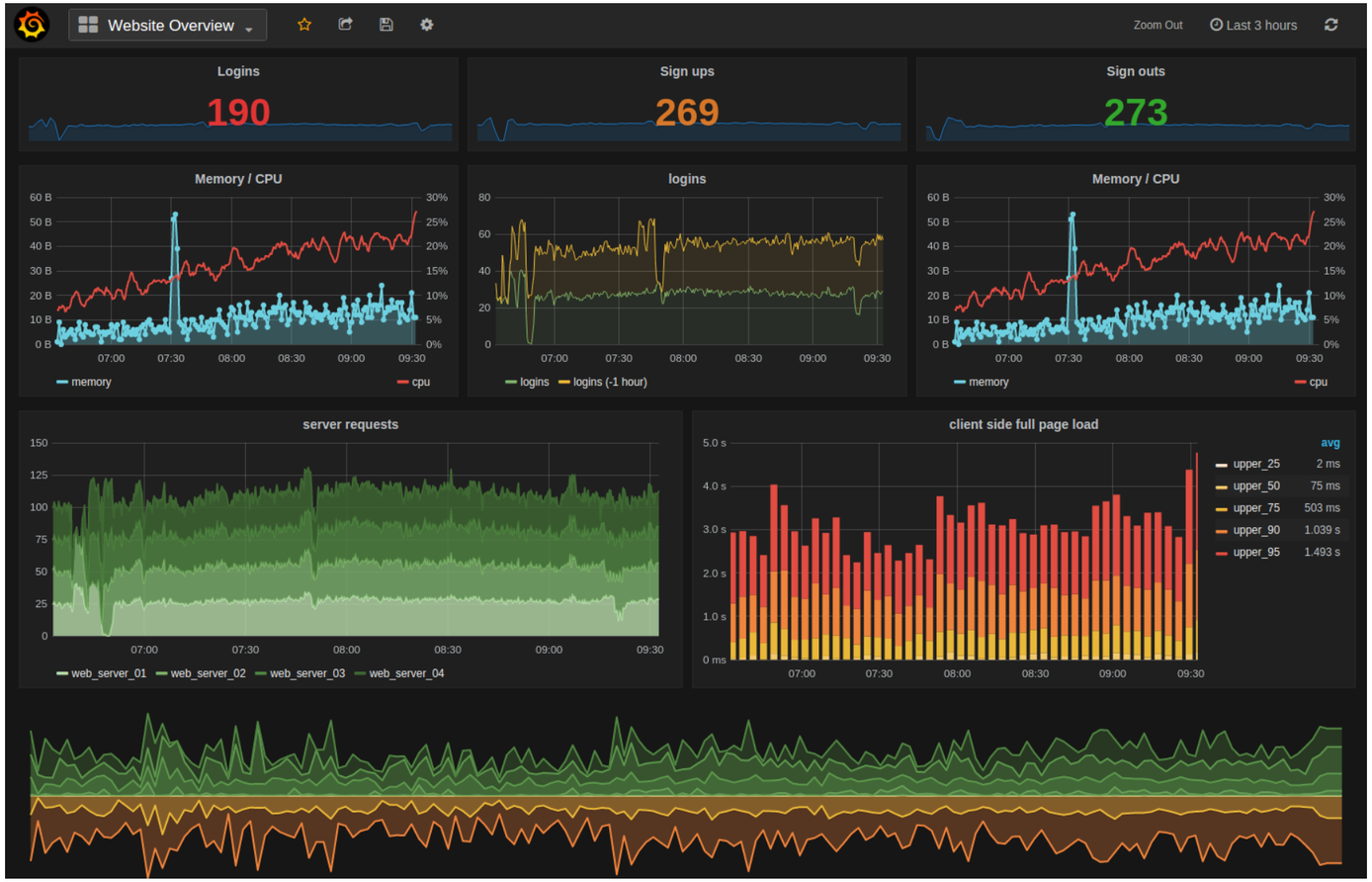
Tools: Prometheus

- backend for monitoring and alerting
- time series database
- HTTP pull model
- PromQL for querying
- open source

Tools: Grafana

- frontend for monitoring
- interactive visualization
- prometheus data source integration
- open source

Tools: Grafana



The scenario

The scenario

- Model exposed in FastAPI app

The scenario

- Model exposed in FastAPI app
  - dummy/random endpoint

The scenario

- Model exposed in FastAPI app
  - dummy/random endpoint
  - model prediction endpoint

The scenario

- Model exposed in FastAPI app
  - dummy/random endpoint
  - model prediction endpoint
- Prometheus

The scenario

- Model exposed in FastAPI app
  - dummy/random endpoint
  - model prediction endpoint
- Prometheus
- Grafana

Local setup with docker-compose

```yaml
version: "3"

services:
  api:
    build: .
    volumes:
     - ./src:/app
     - ./models:/app/models
    ports:
      - "5000:5000"

  prometheus:
    image: prom/prometheus:v2.35.0
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus:/etc/prometheus
      - ./prometheus-data:/prometheus
    command: --web.enable-lifecycle --config.file=/etc/prometheus/prometheus.yaml

  grafana:
    image: grafana/grafana:8.5.2
    ports:
      - "3000:3000"
    restart: unless-stopped
    volumes:
      - ./grafana/provisioning/datasources:/etc/grafana/provisioning/datasources
      - ./grafana-data:/var/lib/grafana
```

Local setup with docker-compose

```yaml
version: "3"

services:
  api:
    build: .
    volumes:
     - ./src:/app
     - ./models:/app/models
    ports:
      - "5000:5000"

  prometheus:
    image: prom/prometheus:v2.35.0
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus:/etc/prometheus
      - ./prometheus-data:/prometheus
    command: --web.enable-lifecycle --config.file=/etc/prometheus/prometheus.yaml

  grafana:
    image: grafana/grafana:8.5.2
    ports:
      - "3000:3000"
    restart: unless-stopped
    volumes:
      - ./grafana/provisioning/datasources:/etc/grafana/provisioning/datasources
      - ./grafana-data:/var/lib/grafana
```

Local setup with docker-compose

```yaml
version: "3"

services:
  api:
    build: .
    volumes:
     - ./src:/app
     - ./models:/app/models
    ports:
      - "5000:5000"

  prometheus:
    image: prom/prometheus:v2.35.0
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus:/etc/prometheus
      - ./prometheus-data:/prometheus
    command: --web.enable-lifecycle --config.file=/etc/prometheus/prometheus.yaml

  grafana:
    image: grafana/grafana:8.5.2
    ports:
      - "3000:3000"
    restart: unless-stopped
    volumes:
      - ./grafana/provisioning/datasources:/etc/grafana/provisioning/datasources
      - ./grafana-data:/var/lib/grafana
```

Traditional software monitoring

Google's 4 golden signals:

Traditional software monitoring

Google's 4 golden signals:

→ Traffic

Traditional software monitoring

Google's 4 golden signals:

- Traffic
- Latency

Traditional software monitoring

Google's 4 **golden signals**:

- Traffic
- Latency
- Errors

Traditional software monitoring

Google's 4 **golden signals**:

- Traffic
- Latency
- Errors
- Saturation (CPU/RAM usage)

Predict endpoint without monitoring

```
@app.post("/predict/model")
def post_model_prediction(features):
    """Get a prediction from the deployed model based on input features"""

    X = parse_features(features)
    prediction = model.predict(X).item()

    return prediction
```

Predict endpoint without monitoring

```
@app.post("/predict/model")
def post_model_prediction(features):
    """Get a prediction from the deployed model based on input features"""

    X = parse_features(features)
    prediction = model.predict(X).item()

    return prediction
```

Prometheus metrics

```
from prometheus_client import Counter, Histogram

REQUEST_COUNT = Counter(
    "request_count", "Total number of requests",
    labelnames=["model_version"]
)

REQUEST_LATENCY = Histogram(
    "request_latency_seconds", "Time spent processing request in seconds",
    labelnames=["model_version"]
)
```

Prometheus metrics

```
from prometheus_client import Counter, Histogram

REQUEST_COUNT = Counter(
    "request_count", "Total number of requests",
    labelnames=["model_version"]
)

REQUEST_LATENCY = Histogram(
    "request_latency_seconds", "Time spent processing request in seconds",
    labelnames=["model_version"]
)
```

Predict endpoint with monitoring

```
@app.post("/predict/model")
@REQUEST_LATENCY.labels(model_version=MODEL_VERSION).time()
def post_model_prediction():
    """Get a prediction from the deployed model based on input features"""
    REQUEST_COUNT.labels(model_version=MODEL_VERSION).inc()

    X = parse_features(features)
    prediction = model.predict(X).item()

    return prediction
```

Expose metrics to prometheus

```
from prometheus_client import generate_latest

@app.get('/metrics', response_class=PlainTextResponse)
def metrics():
    return generate_latest()
```

What makes ML monitoring different?

What makes ML monitoring different?

What makes ML monitoring different?

- Data flows

What makes ML monitoring different?

- Data flows
  - Data availability problems

What makes ML monitoring different?

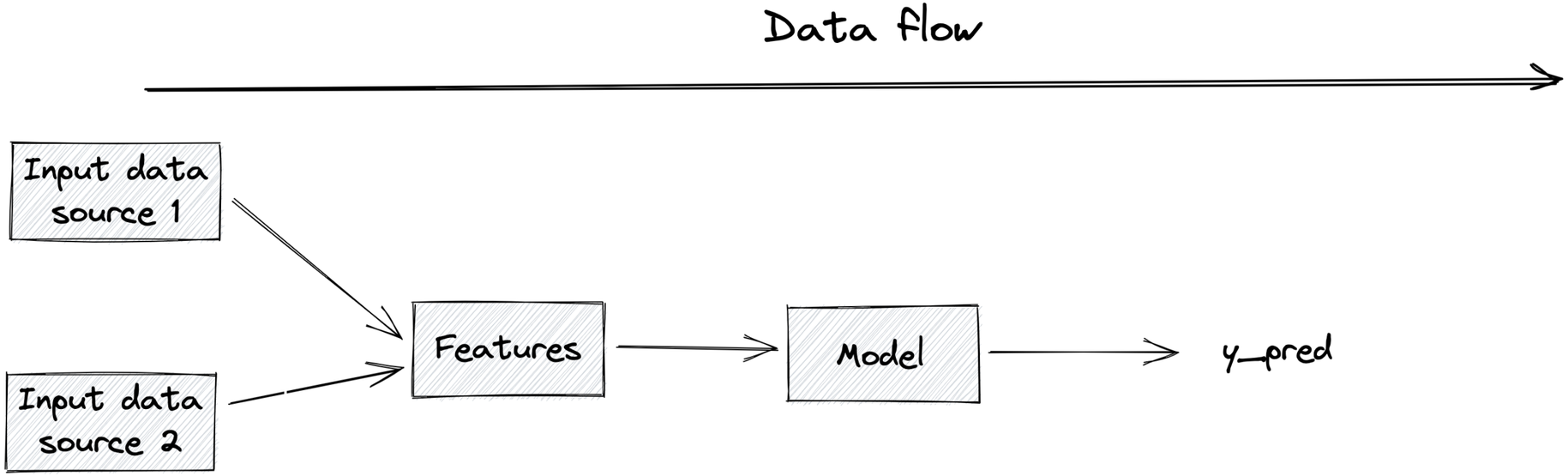
- Data flows
  - Data availability problems
  - Data quality problems

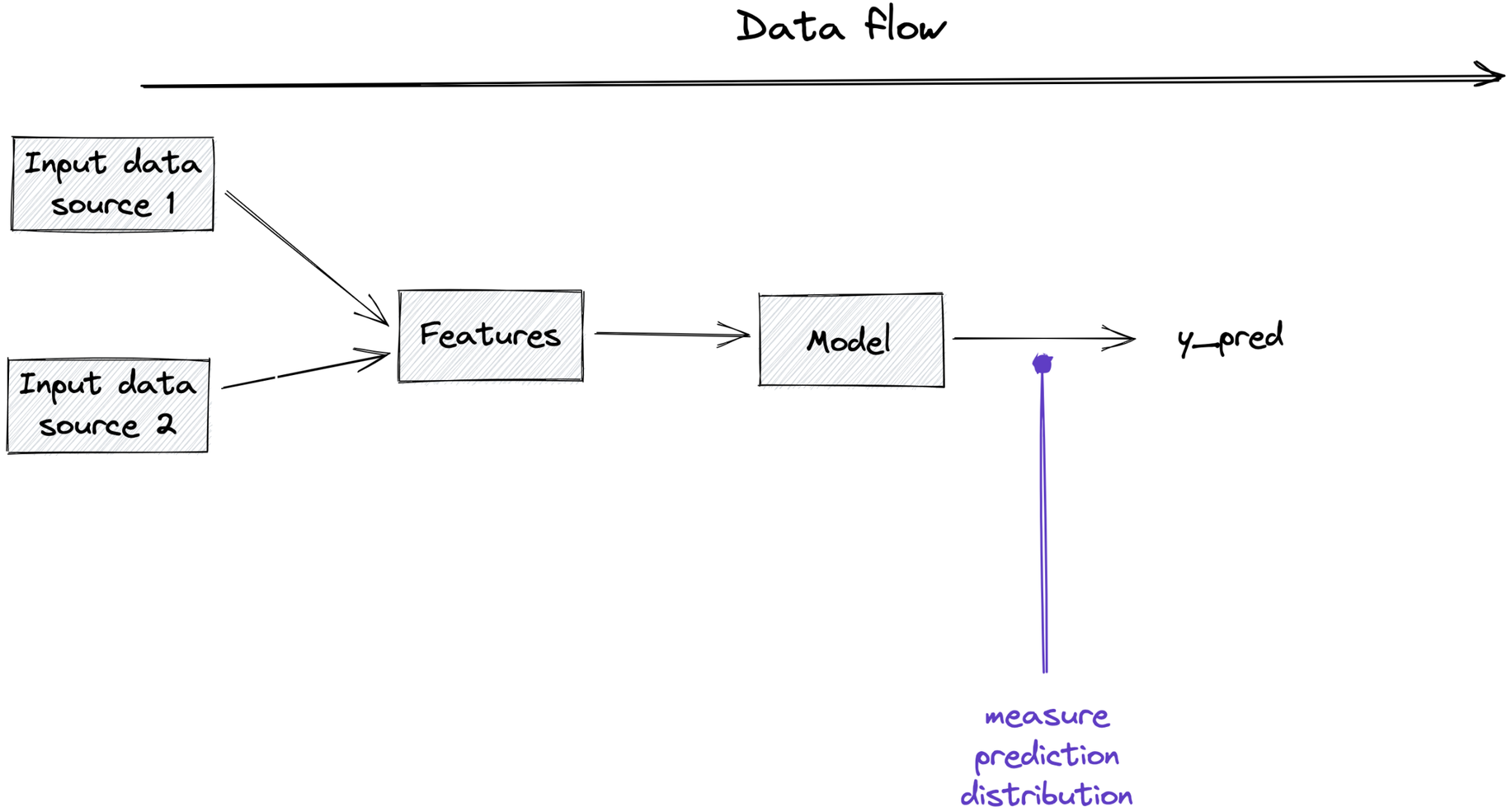
What makes ML monitoring different?

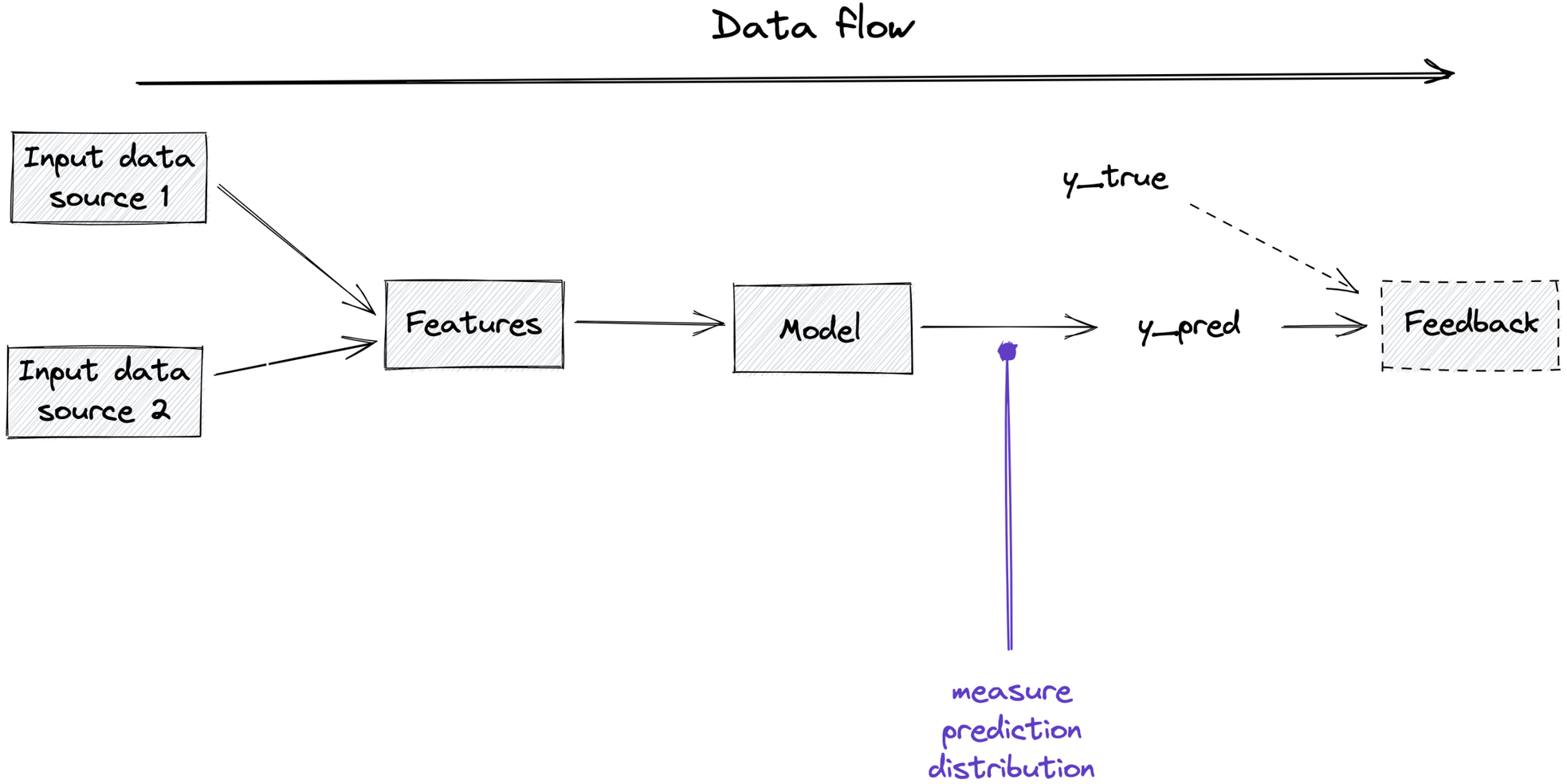
- Data flows
  - Data availability problems
  - Data quality problems
- Black box model

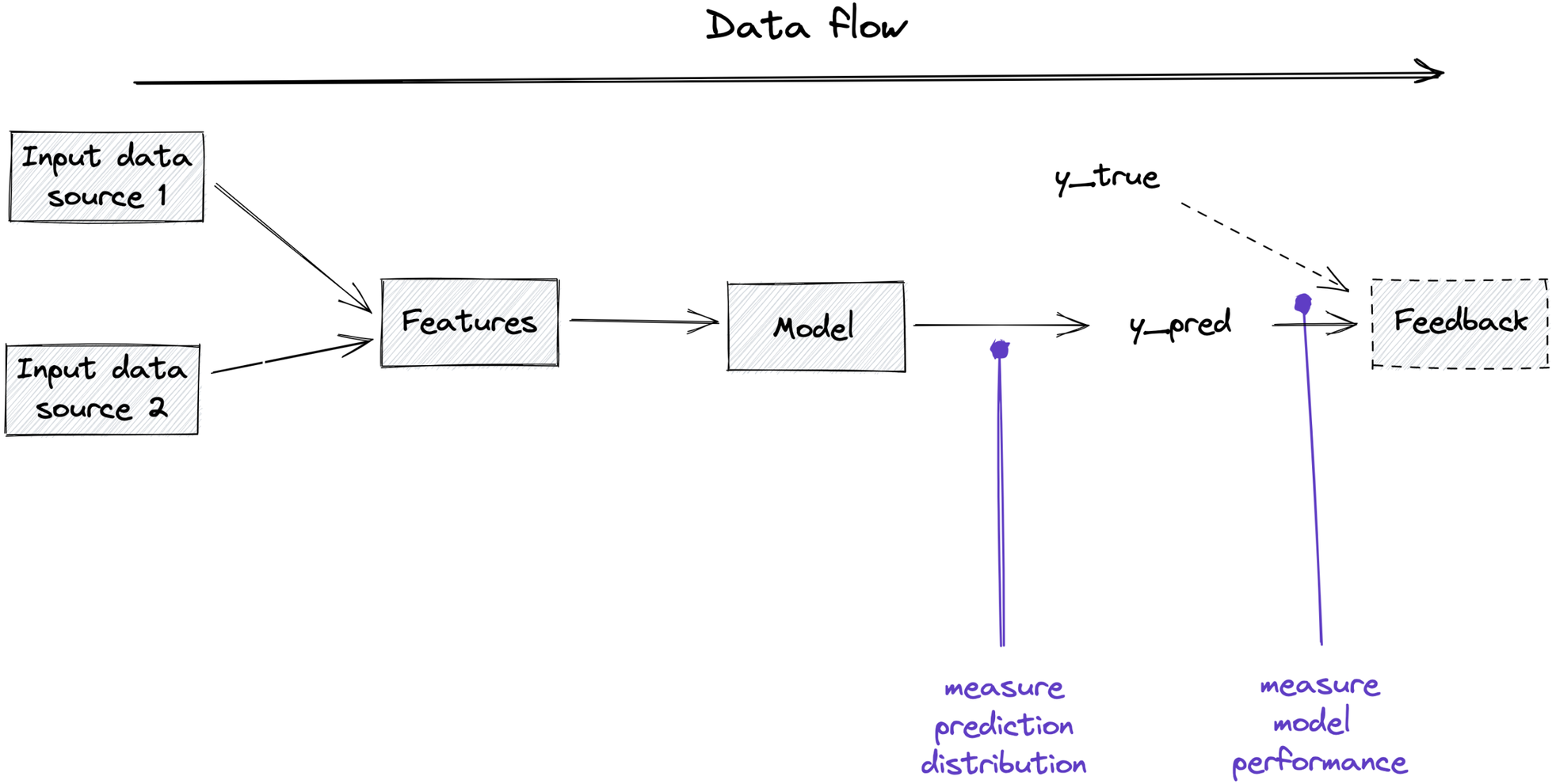
What makes ML monitoring different?

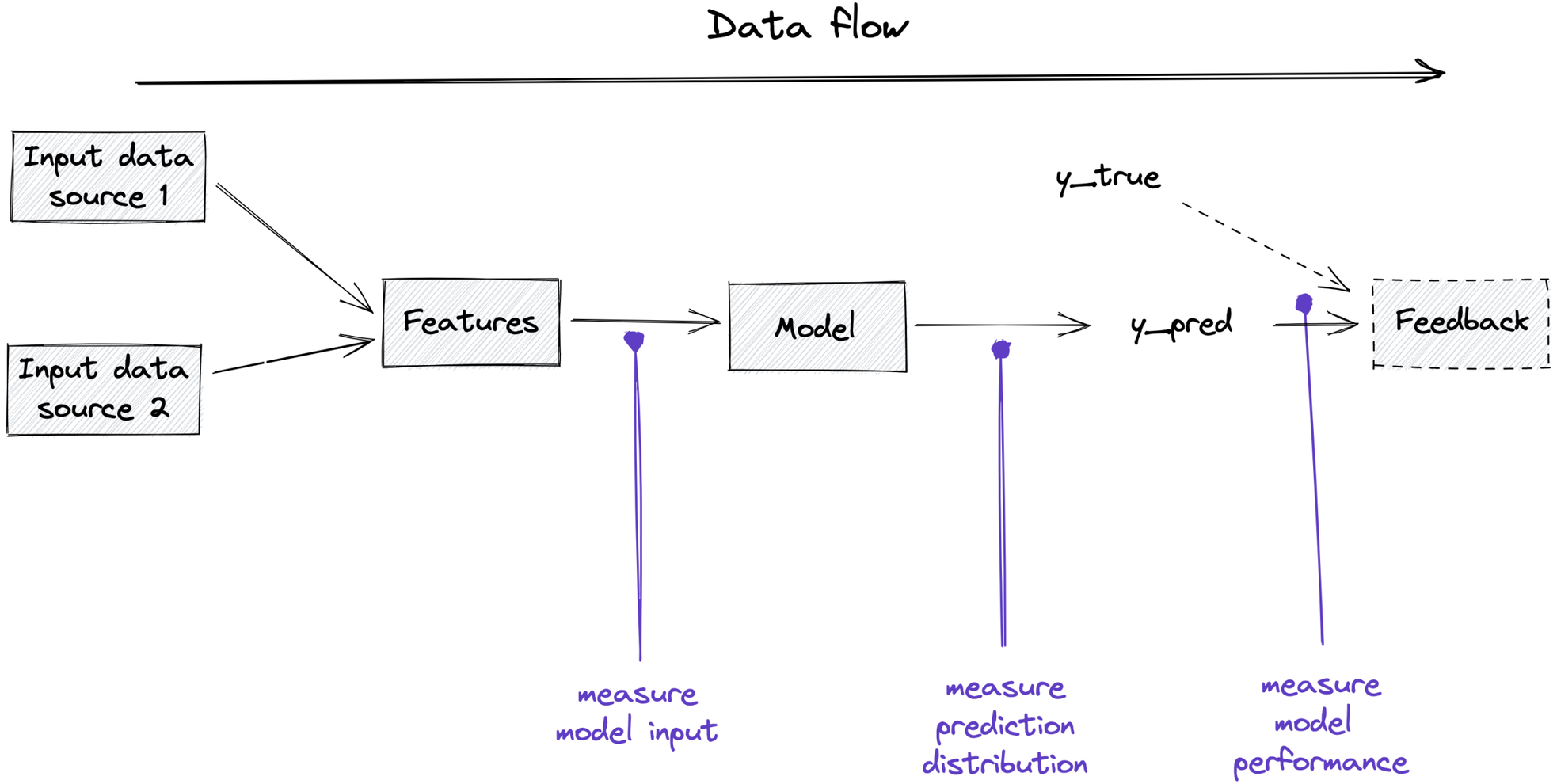
- Data flows
  - Data availability problems
  - Data quality problems
- Black box model
  - Performance decays over time

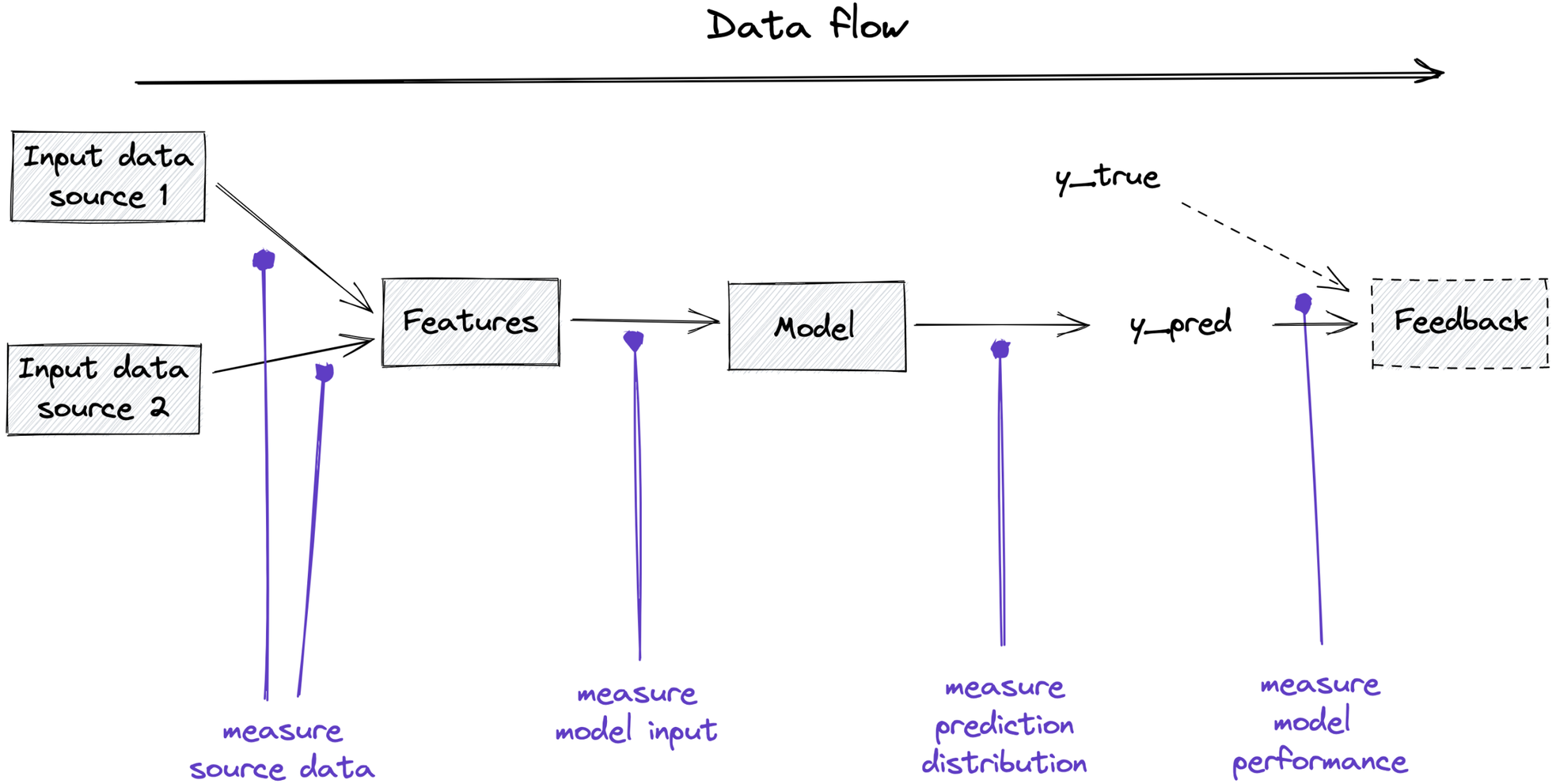

Data flow
Input data source 1
Input data source 2
Features
Model
y_pred

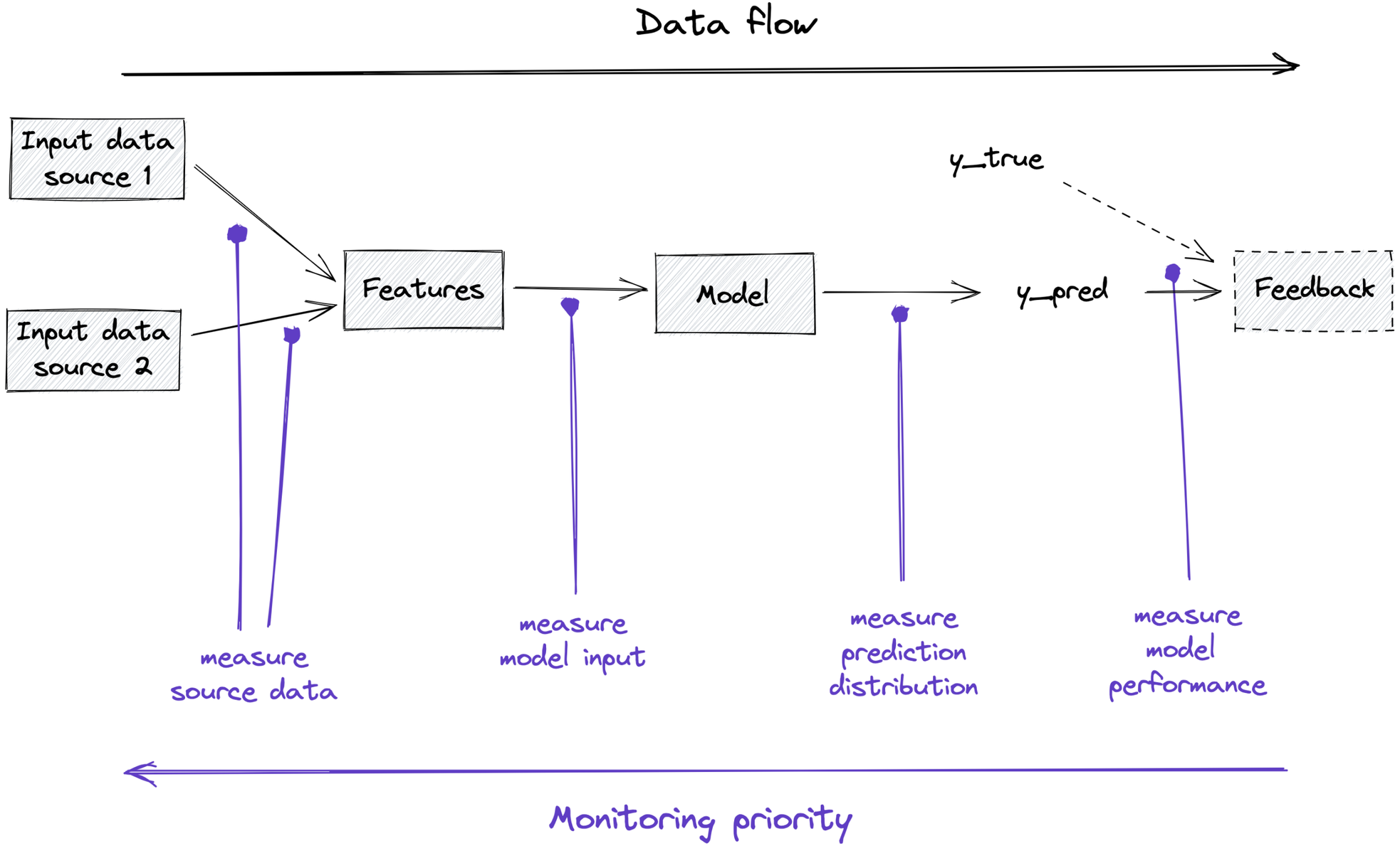

Data flow
Input data source 1
Input data source 2
Features
Model
y_pred
measure prediction distribution


Data flow
Input data source 1
Input data source 2
Features
Model
y_pred
y_true
Feedback
measure prediction distribution


Data flow
Input data source 1
Input data source 2
Features
Model
y_pred
y_true
Feedback
measure prediction distribution
measure model performance


Data flow
Input data source 1
Input data source 2
Features
Model
y_pred
y_true
Feedback
measure model input
measure prediction distribution
measure model performance


Data flow
Input data source 1
Input data source 2
Features
Model
y_pred
y_true
Feedback
measure source data
measure model input
measure prediction distribution
measure model performance


Data flow
Input data source 1
Input data source 2
Features
Model
y_pred
y_true
Feedback
measure source data
measure model input
measure prediction distribution
measure model performance
Monitoring priority

Practical steps for scikit-learn implementations

Prometheus metrics

```
from prometheus_client import Counter, Histogram

REQUEST_COUNT = Counter(
    "request_count", "Total number of requests",
    labelnames=["model_version"]
)

REQUEST_LATENCY = Histogram(
    "request_latency_seconds", "Time spent processing request in seconds",
    labelnames=["model_version"]
)

RESPONSE_DIST = Histogram(
    "response_distribution", "Response distribution of the predictions",
    buckets=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
    labelnames=["model_version"]
)
```

Predict endpoint with monitoring

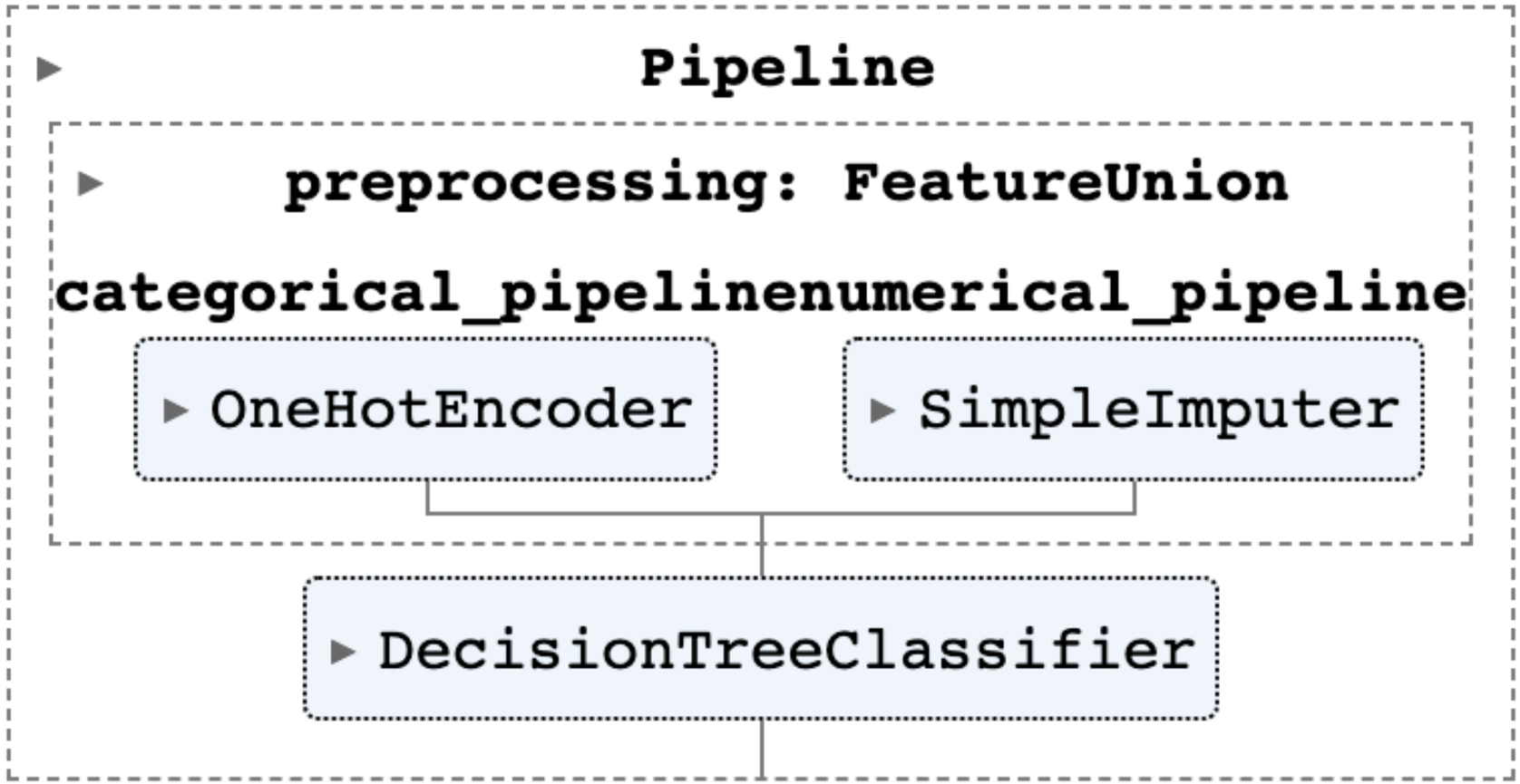
```
@app.post("/predict/model")
@REQUEST_LATENCY.labels(model_version=MODEL_VERSION).time()
def post_model_prediction(features: Features) -> int:
    """Get a prediction from the deployed model based on input features"""
    REQUEST_COUNT.labels(model_version=MODEL_VERSION).inc()

    X = parse_features(features)
    prediction = model.predict(X).item()

    RESPONSE_DIST.labels(model_version=MODEL_VERSION).observe(prediction)

    return prediction
```

Scikit-learn model



One Hot Encoder

```
from sklearn import preprocessing

class OneHotEncoder(preprocessing.OneHotEncoder):

    def transform(self, X):
        transformed_X = super().transform(X)
        features = get_feature_names(X)

        categories = self.inverse_transform(transformed_X)

        for idx, row in enumerate(categories.T):
            for category in row:
                if category is None:
                    category = "missing"
                MODEL_CATEGORICAL.labels(feature=str(features[idx]), category=str(category)).inc()
        return transformed_X
```

One Hot Encoder

```
from sklearn import preprocessing

class OneHotEncoder(preprocessing.OneHotEncoder):

    def transform(self, X):
        transformed_X = super().transform(X)
        features = get_feature_names(X)

        categories = self.inverse_transform(transformed_X)

        for idx, row in enumerate(categories.T):
            for category in row:
                if category is None:
                    category = "missing"
                MODEL_CATEGORICAL.labels(feature=str(features[idx]), category=str(category)).inc()
        return transformed_X
```

Simple Imputer

```
from sklearn import impute

class SimpleImputer(impute.SimpleImputer):

    def transform(self, X):
        features = get_feature_names(X)

        missing = np.isnan(X).sum(axis=0)
        for idx, feature in enumerate(features):
            IMPUTED.labels(feature=feature, method="SimpleImputer").inc(missing[idx])

        return super().transform(X)
```

Simple Imputer

```
from sklearn import impute

class SimpleImputer(impute.SimpleImputer):

    def transform(self, X):
        features = get_feature_names(X)

        missing = np.isnan(X).sum(axis=0)
        for idx, feature in enumerate(features):
            IMPUTED.labels(feature=feature, method="SimpleImputer").inc(missing[idx])

        return super().transform(X)
```

Demo

Model decay

Model decay

→ Often due to drifting distributions

Model decay

- Often due to drifting distributions
- 3 types of drift

Model decay

- Often due to drifting distributions
- 3 types of drift
  - Data drift

Model decay

- Often due to drifting distributions
- 3 types of drift
  - Data drift
  - Concept drift

Model decay

- Often due to drifting distributions
- 3 types of drift
  - Data drift
  - Concept drift
  - Train-serve skew

Data drift

Data drift

- input data changes

Data drift

- → input data changes
- → distributions meaningfully different

Data drift

- input data changes
- distributions meaningfully different
- reflecting changes in underlying patterns

Concept drift

Concept drift

→ learned relationships no longer hold

Concept drift

- learned relationships no longer hold
- requires *y_true* in production

Concept drift

- learned relationships no longer hold
- requires *y_true* in production
- input data might be the same

Concept drift

- learned relationships no longer hold
- requires *y_true* in production
- input data might be the same
- model performance declines (e.g. accuracy or MAE)

Train-serve skew

Train-serve skew

- Mismatch between **train** and **production** phase

Train-serve skew

- Mismatch between **train** and **production** phase
- Possible causes:

Train-serve skew

- Mismatch between **train** and **production** phase
- Possible causes:
  - time gap

Train-serve skew

- Mismatch between **train** and **production** phase
- Possible causes:
  - time gap
  - training data not generalizing

Train-serve skew

- Mismatch between **train** and **production** phase
- Possible causes:
  - time gap
  - training data not generalizing
- Compare distributions on metrics

Train-serve skew

- Mismatch between **train** and **production** phase
- Possible causes:
  - time gap
  - training data not generalizing
- Compare distributions on metrics
- Retraining ?

Wrap up: priorities for monitoring

Wrap up: priorities for monitoring

1. Golden signals -> general app status

Wrap up: priorities for monitoring

1. Golden signals -> general app status
2. Predictions (y_pred) -> model output

Wrap up: priorities for monitoring

1. Golden signals -> general app status
2. Predictions (y_pred) -> model output
3. Feedback observations (y_true) -> concept drift

Wrap up: priorities for monitoring

1. Golden signals -> general app status
2. Predictions (y_pred) -> model output
3. Feedback observations (y_true) -> concept drift
4. Training data (y_train) -> training-serving skew

Wrap up: priorities for monitoring

1. Golden signals -> general app status
2. Predictions (y_pred) -> model output
3. Feedback observations (y_true) -> concept drift
4. Training data (y_train) -> training-serving skew
5. Features and input distributions -> data availability, quality and drift

Thank you

- LinkedIn / wesleyboelrijk
- GitHub / wesleyboelrijk
- GitHub / ... / monitoring-pydata-eindhoven-2022