Code organization and refactoring

INTRODUCTION TO DATA VERSIONING WITH DVC

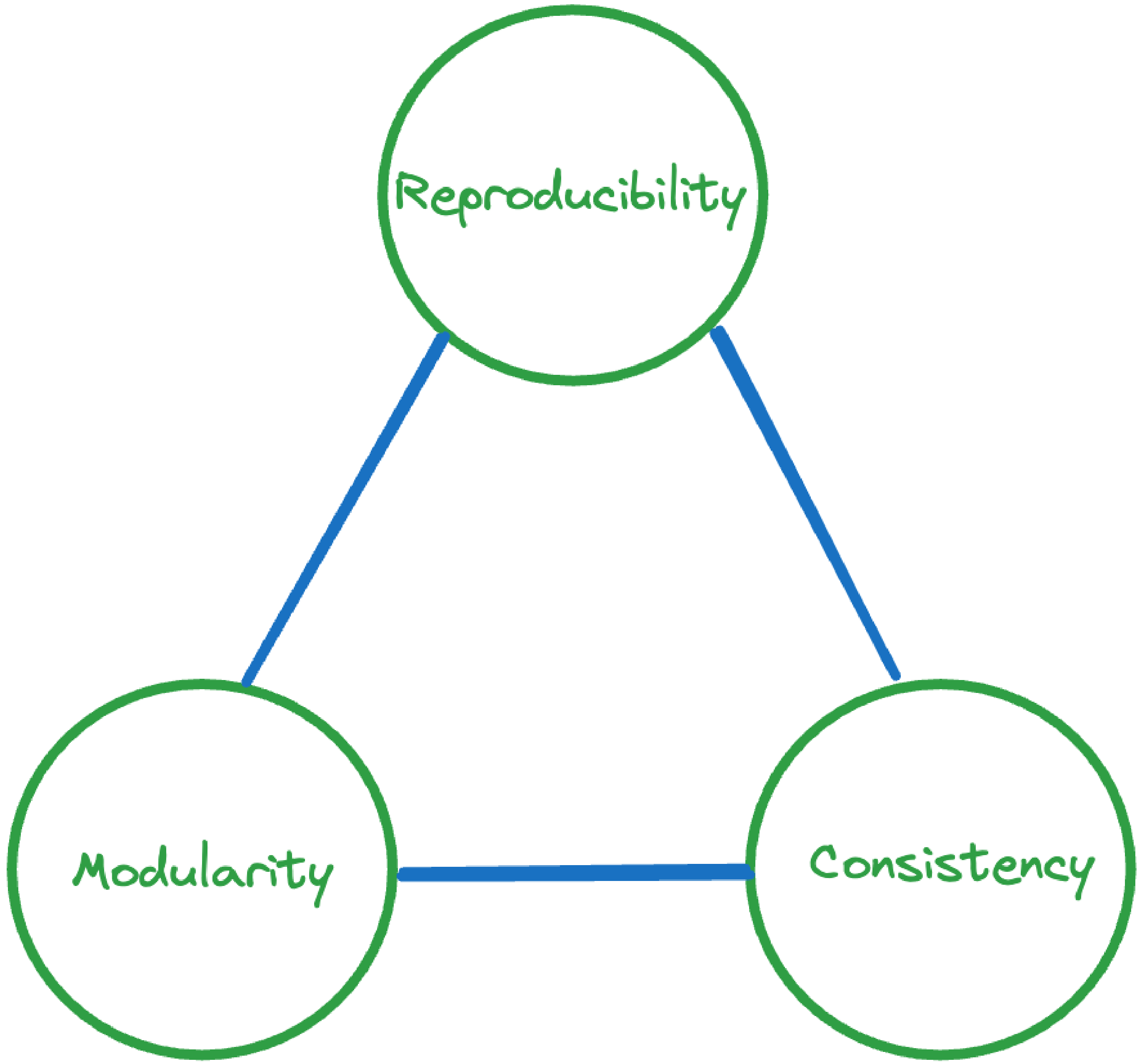
Ravi Bhadauria
Machine Learning Engineer

Prototyping vs production code

- Prototyping code allows rapid iteration
- But not suitable for production
  - Untested and prone to errors
  - Not modular with many repeated code blocks
  - Likely not reproducible

Features of good production code

- **Reproducible:** recreate same outputs in different environments and time
- **Modular:** written as distinct, independent, and testable modules
- **Consistent:** Single source of truth for all parameters
  - A configuration/parameter file



Configuration files and YAML

- Files should be in supported format
  - YAML, JSON, TOML, Python
  - Default is `params.yaml`
- We'll work with YAML
  - **YAML Ain't Markup Language**
  - Allows a standard format to transfer data between languages or applications
  - Simple and clean format
  - Valid file extensions: `.yaml` or `.yml`

[1] https://dvc.org/doc/command-reference/params#description

YAML Syntax

- Specify parameters as dictionaries
  - Keys and values separated by `:`
- Comments start with `#`
- Data types:
  - Integer, Floats, Strings
- Data structures:
  - Arrays
  - Nested Dictionaries
- Indentation is important

```yaml
# Key-value pairs
a: 1
b: 1.2
c: "String value"
```

```yaml
# Arrays
a: [1, 2.2, 3, 4.8]
b:
  - 5
  - "String value"
```

```yaml
# Nested dictionaries
a:
  b: "Some value"
  c: "Some other value"
```

Example configuration file

```
# Data preprocessing paramters
preprocess:
  ...
  target_column: RainTomorrow
  categorical_features:
    - Location
    - WindGustDir
    - ...
# Model training/evaluation paramters
train_and_evaluate:
  rfc_params:
    n_estimators: 2
    ...
```

Example modular function

```
# In model.py
def evaluate_model(model, X_test, y_test):
    """Evaluate a model on a test set and return metrics."""
    y_pred = model.predict(X_test)
    precision = precision_score(y_test, y_pred)
    ...
    return { "accuracy": accuracy, "precision": precision,
        "recall": recall, "f1_score": f1 }
```

```
# In entry-point code (train_and_evaluate.py)
from model import evaluate_model
metrics = evaluate_model(model, X_test, y_test)
```

Sample project code layout

```
-> tree .
.
├── params.yaml # Configuration file
├── metrics_and_plots.py # Helper functions
├── model.py # Model definition
├── preprocess_dataset.py # Driver code to preprocess
├── train_and_evaluate.py # Driver code to train
└── utils_and_constants.py # More helper functions
```

Let's practice!

INTRODUCTION TO DATA VERSIONING WITH DVC

Writing and visualizing DVC pipelines

INTRODUCTION TO DATA VERSIONING WITH DVC

Ravi Bhadauria
Machine Learning Engineer

DVC Pipelines

- Sequence of stages defining Machine Learning workflow and dependencies
    - Versioned and tracked with Git
- Defined in `dvc.yaml` file
    - Input data and scripts (`deps`)
    - Parameters (`params`)
    - Stage execution commands (`cmd`)
    - Output artifacts (`outs`)
        - Special data e.g. `metrics` and `plots`

Adding preprocessing stage

- Create stages using `dvc stage add`

```
dvc stage add \
-n preprocess \
-p params.yaml:preprocess \
-d raw_data.csv \
-d preprocess.py \
-o processed_data.csv \
python3 preprocess.py
```

```
stages:
  preprocess:
    cmd: python3 preprocess.py
    params:
      # Keys from params.yaml
      - params.yaml
        - preprocess
    deps:
    - preprocess.py
    - raw_data.csv
    outs:
    - processed_data.csv
```

Adding training and evaluation stage

- Add a training step using output from previous step

```
dvc stage add \
-n train_and_evaluate \
-p train_and_evaluate \
-d train_and_evaluate.py \
-d processed_data.csv \
-o plots.png \
-o metrics.json \
python3 train_and_evaluate.py
```

- Directed Acyclic Graph (DAG)

```
stages:
  train_and_evaluate:
    cmd: python3 train_and_evaluate.py
    params:
      # Skip specifying parameter file
      # Defaulted to params.yaml
      - train_and_evaluate
    deps:
    - processed_data.csv
    - train_and_evaluate.py
    outs:
    - plots.png
    - metrics.json
```

Updating stages

- Running `dvc stage add` multiple times

```
ERROR: Stage 'train_and_evaluate'
already exists in 'dvc.yaml'.
Use '--force' to overwrite.
```

- Use `dvc stage add --force`

```
dvc stage add --force \
-n train_and_evaluate \
-p train_and_evaluate \
-d train_and_evaluate.py \
-d processed_data.csv \
-o plots.png \
-o metrics.json \
python3 train_and_evaluate.py
```

Visualizing DVC pipelines

```
# Print DAG on terminal
dvc dag
```

```
# Display DAG up to a certain step
dvc dag <target>
```

```
      +------------+
      | preprocess |
      +------------+
             *
             *
             *
  +--------------------+
  | train_and_evaluate |
  +--------------------+
```

Visualizing DVC pipelines

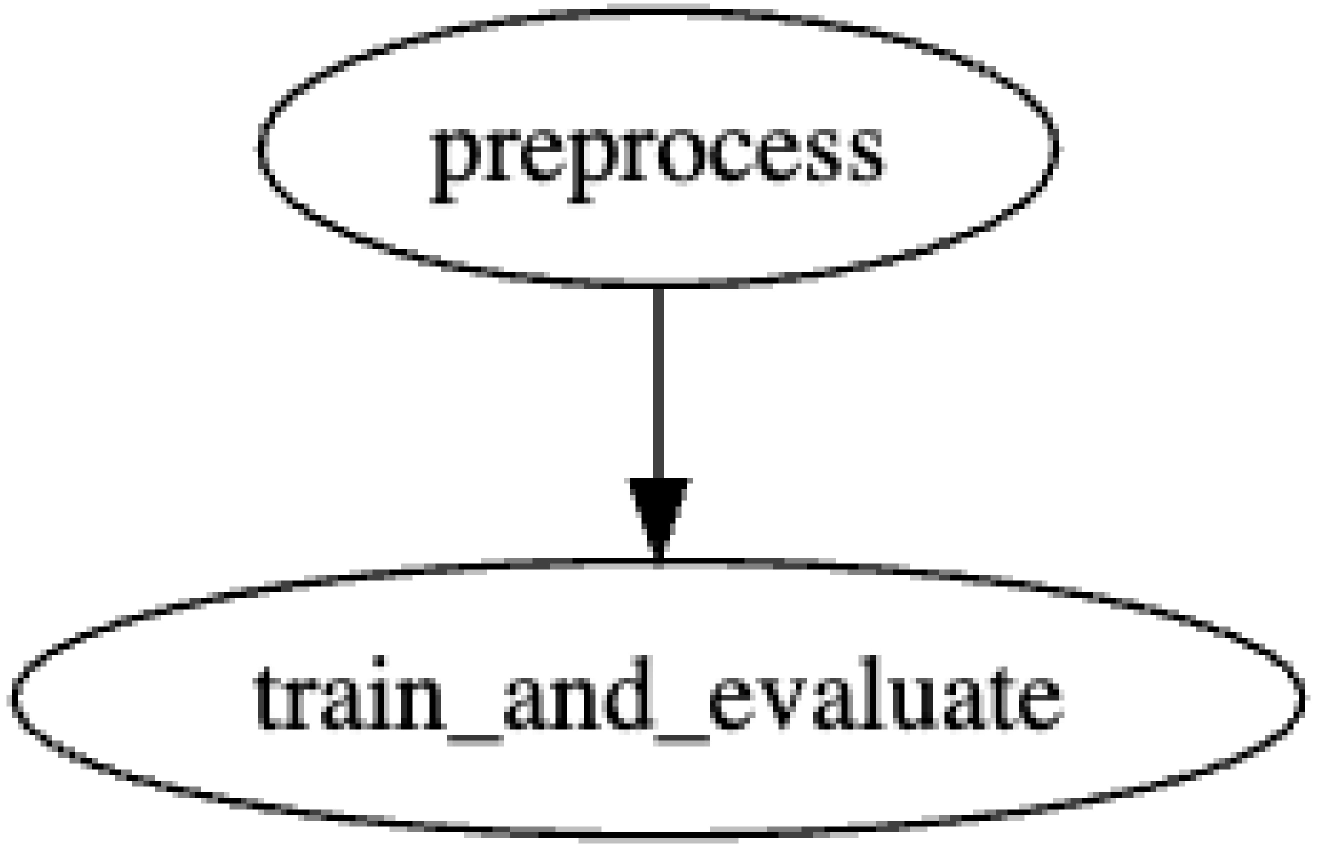
```
# Display step outputs as nodes
dvc dag --outs
```

```
                    +-------------------------------+
                    | processed_dataset/weather.csv |
                    +-------------------------------+
                          ***                 ***
                       ***                       ***
                     **                             **
          +---------------+                    +------------+
          | metrics.json |                    | plots.png |
          +---------------+                    +------------+
```

Visualizing DVC pipelines

```
dvc dag --dot
```

```
strict digraph  {
"preprocess";
"train_and_evaluate";
"preprocess" -> "train_and_evaluate";
}
```



[1] https://dreampuf.github.io/GraphvizOnline/

Let's practice!

INTRODUCTION TO DATA VERSIONING WITH DVC

Executing DVC pipelines

INTRODUCTION TO DATA VERSIONING WITH DVC

Ravi Bhadauria
Machine Learning Engineer

Recap

- Preprocessing stage

```
stages:
  preprocess:
    cmd: python3 preprocess.py
    params:
      - preprocess
    deps:
    - preprocess.py
    - raw_data.csv
    outs:
    - processed_data.csv
```

- Training and evaluation

```
stages:
  train_and_evaluate:
    cmd: python3 train_and_evaluate.py
    params:
      - train_and_evaluate
    deps:
    - processed_data.csv
    - train_and_evaluate.py
    outs:
    - plots.png
    - metrics.json
```

Reproducing a pipeline

- Reproduce the pipeline using `dvc repro`

```
$ dvc repro
```

```
Running stage 'preprocess':
> python preprocess.py
Running stage 'train_and_evaluate':
> python train_and_evaluate.py
Updating lock file 'dvc.lock'
```

- A state file `dvc.lock` is generated
  - Similar to `.dvc` file, captures MD5 hashes

```
$ git add dvc.lock && git commit -m "first pipeline run"
```

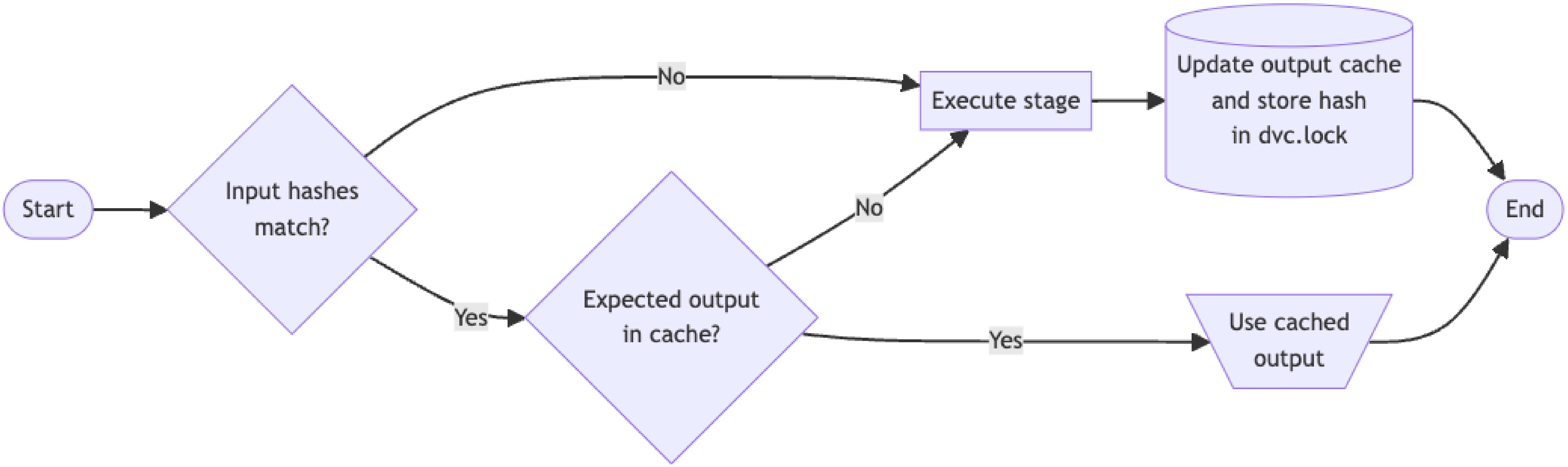
Using cached results

- Using cached results to speed up iteration

```
$ dvc repro
```

```
Stage 'preprocess' didn't change, skipping
Running stage 'train_and_evaluate' with command: ...
```

Stage caching in DVC



Dry running a pipeline

- Use `--dry` flag to only print commands without running the pipeline

```
$ dvc repro --dry
```

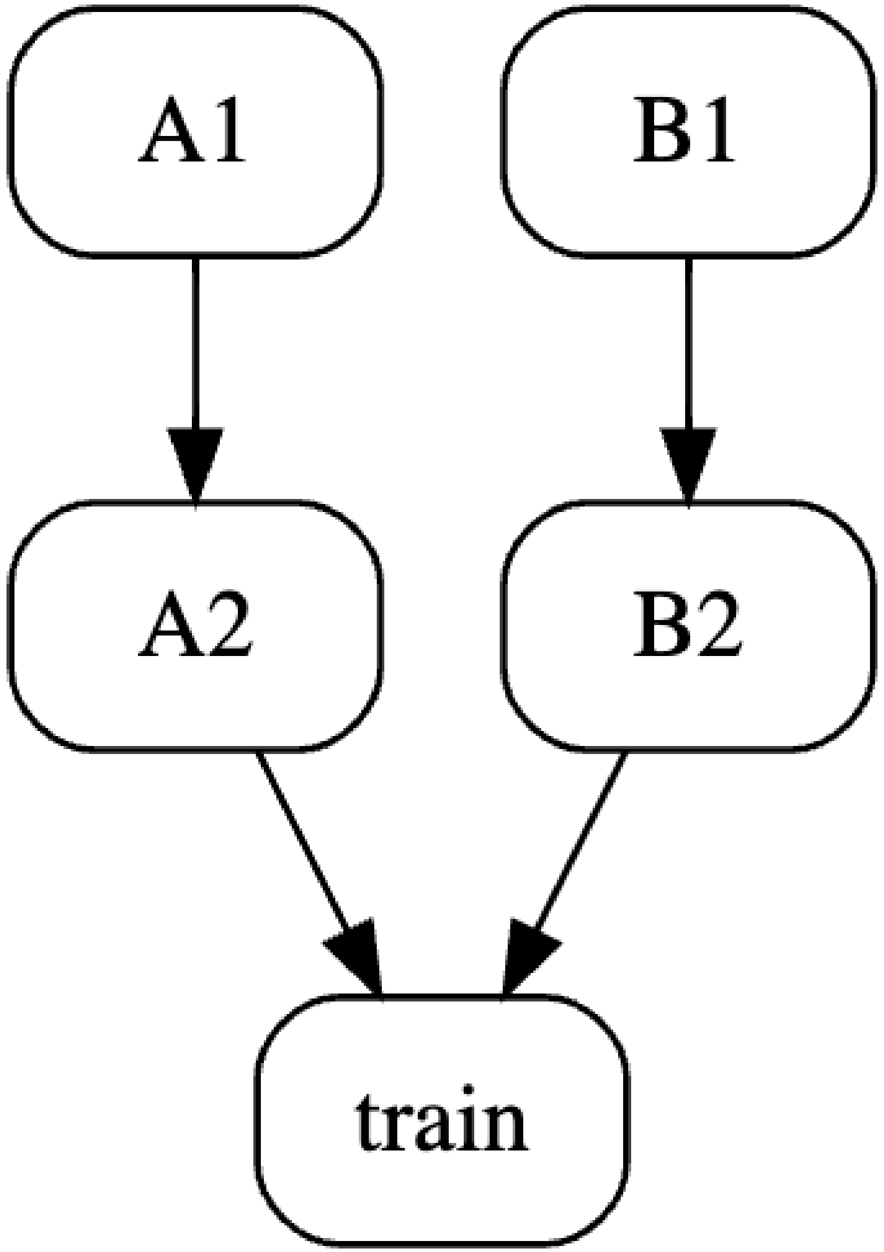
```
Running stage 'preprocess':
> python3 preprocess_dataset.py

Running stage 'train_and_evaluate':
> python3 train_and_evaluate.py
```

Additional arguments

- Running specific files `dvc repro linear/dvc.yaml`
  - Multiple `dvc.yaml` in one folder are not allowed
- Running specific stages `dvc repro <target_stage>`
  - This will *also* run upstream dependencies
- Force run a pipeline/stage `dvc repro -f`
- Not storing execution outputs in cache `dvc repro --no-commit`
  - Use `dvc commit` later

Parallel stage execution



- Run independent steps concurrently

```
# Run A2 and its upstream dependencies
$ dvc repro A2
```

```
# Run B2 and its upstream dependencies
$ dvc repro B2
```

- Use caching to speed up execution

```
$ dvc repro train
```

```
Stage 'A2' didn't change, skipping
Stage 'B2' didn't change, skipping
Running stage 'train' with command: ...
```

Let's practice!

INTRODUCTION TO DATA VERSIONING WITH DVC

Evaluation: Metrics and plots in DVC

INTRODUCTION TO DATA VERSIONING WITH DVC

Ravi Bhadauria
Machine Learning Engineer

Metrics: changes in dvc.yaml

- Configure DVC YAML file to track metrics across experiments
- Change from `outs`

```
stages:
  train_and_evaluate:
    outs:
    - metrics.json
    - plots.png
```

- To `metrics`

```
stages:
  train_and_evaluate:
    outs:
    - plots.png
    metrics:
      - metrics.json:
        cache: false
```

Printing DVC metrics

```
$ dvc metrics show
```

```
Path          accuracy    f1_score    precision    recall
metrics.json  0.947       0.8656      0.988        0.7702
```

Compare metrics across runs

- Change a hyperparameter and rerun `dvc repro`

```
$ dvc metrics diff
```

```
Path          Metric     HEAD    workspace    Change
metrics.json  accuracy   0.947   0.9995       0.0525
metrics.json  f1_score   0.8656  0.9989       0.1333
metrics.json  precision  0.988   0.9993       0.0113
metrics.json  recall     0.7702  0.9986       0.2284
```

Plots: changes in dvc.yaml

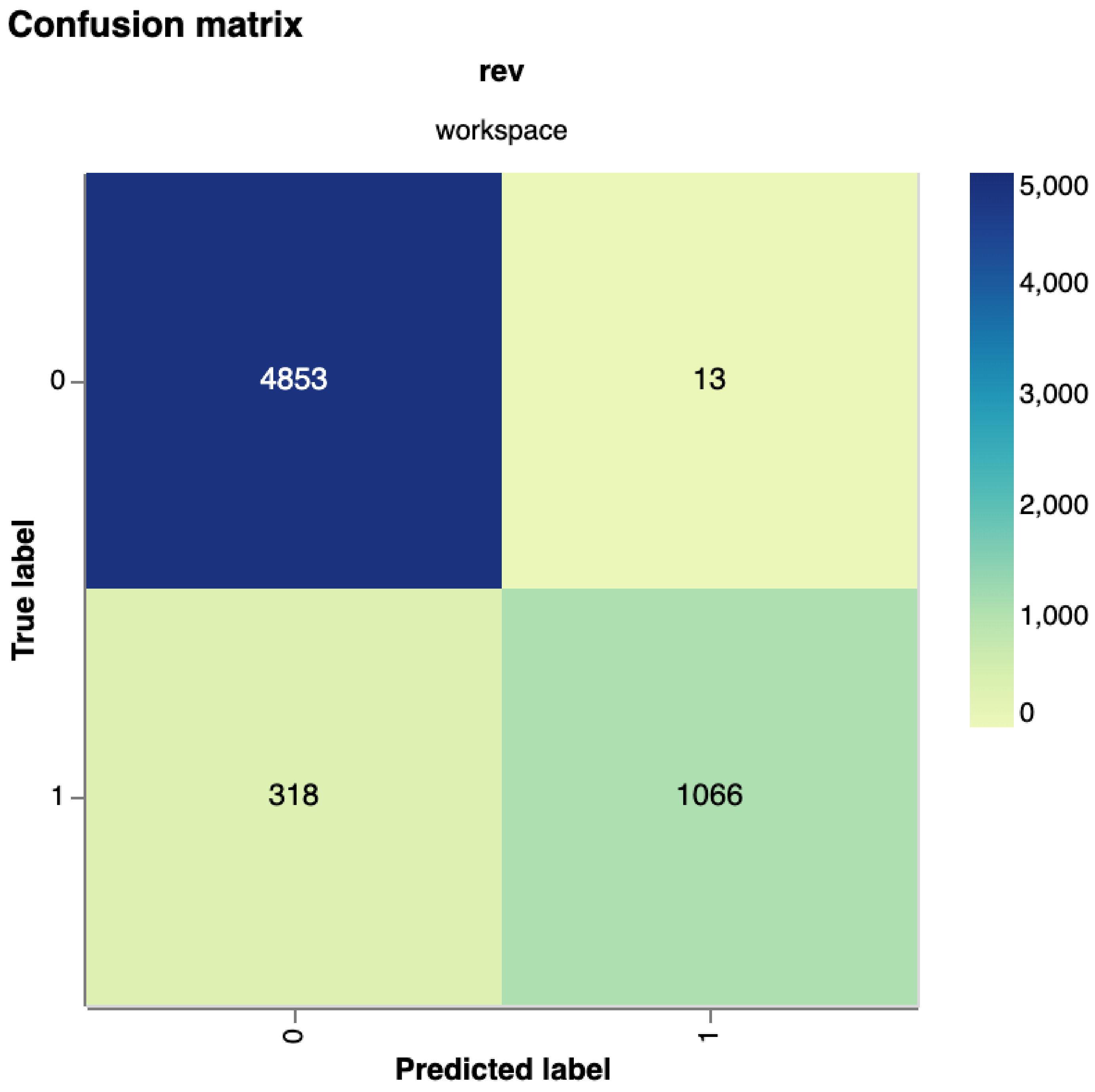
```
stages:
  train_and_evaluate:
    ...
    plots:
    - predictions.csv: # Name of file containing predictions
        template: confusion # Style of plot
        x: predicted_label # X-axis column name in csv file
        y: true_label # Y-axis column name in csv file
        x_label: 'Predicted label'
        y_label: 'True label'
        title: Confusion matrix
        cache: false # Save in Git
```

[1] https://dvc.org/doc/user-guide/experiment-management/visualizing-plots#plot-templates-data-series-only

Printing DVC plots to file

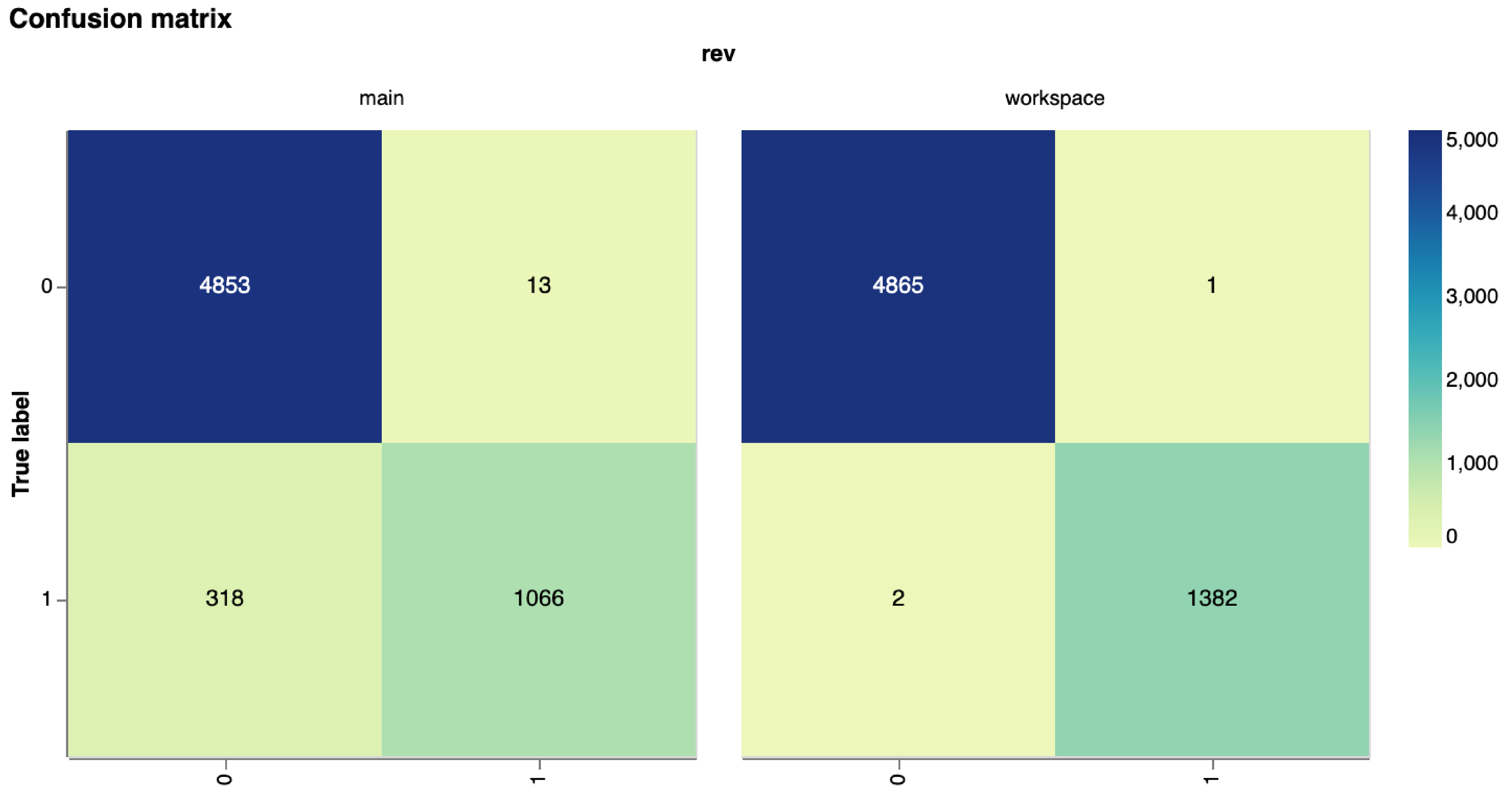
```
$ dvc plots show predictions.csv
```

```
file:///path/to/index.html
```



Comparing DVC plots

```
# comapre plot in predictions.csv against branch main
$ dvc plots diff --target predictions.csv <branch name or commit SHA>
```



Let's practice!

INTRODUCTION TO DATA VERSIONING WITH DVC

Congratulations!

INTRODUCTION TO DATA VERSIONING WITH DVC

Ravi Bhadauria
Machine Learning Engineer

Data versioning and DVC

- Anatomy of Machine Learning Model
  - Code, data, and hyper-parameters precisely define a model
  - All three need to be tracked and versioned
- Git and DVC
  - Git helps with tracking code, DVC helps with tracking data
  - Git tracks metadata about the actual data
- DVC enables us
  - To version data and models
  - Run reproducible experiment pipelines
  - Track changes in metrics and plots

DVC setup, cache, and remotes

- Setup
  - Install using `pip install dvc`
  - Initialize using `dvc init`
  - Use `.dvcignore` to control file patterns to track
- Cache
  - Add files using `dvc add`
  - Track metadata using `.dvc` files
  - Remove using `dvc remove`, clean with `dvc gc`
- Remotes
  - Configure using `dvc remote add`, list using `dvc remote list`
  - Upload and download data using `dvc push` and `dvc pull`

DVC pipelines

- Anatomy of the `dvc.yaml` file
  - Use `dvc stage add` to add stages
  - Components include `steps` , `commands` , `dependencies` , `params` , and `outputs`
  - Track metrics and plots using the `metrics` and `plots` keys
- Visualize and run DAG
  - Visualize using `dvc dag`
  - Run with `dvc repro`
- Show and compare metrics and plots
  - Visualize using `dvc plots show` and `dvc metrics show`
  - Compare using `dvc plots diff` and `dvc metrics diff`

Thank you!

INTRODUCTION TO DATA VERSIONING WITH DVC