🐸FrogID ML - Repo Structure & Git Approach

Code Repository Structure

We have created a template ML project repo structure that is a variation on the structure provided by Databricks MLOps.

The ML Project template repo is here: https://github.com/austmus/frogid-mlops-project-template

To create a new project from this template, follow these instructions from GitHub: https://docs.github.com/en/repositories/creating-and-managing-repositories/creating-a-repository-from-a-template

The essential workflow is:

1. In GitHub, start a new repo from the template frogid-mlops-project-template as per: https://docs.github.com/en/repositories/creating-and-managing-repositories/creating-a-repository-from-a-template, picking a new name
2. Clone the repo to your development environment (Local or Databricks)
3. Early exploration work in notebooks etc would be done in the adhoc_notebooks folder
4. Transition to modular a pipeline structure and modularise code within appropriate mlops folders
5. Productionise code with environment config, CI/CD, good code logging and error handling, monitoring etc.

Template ML Project Structure:

https://github.com/austmus/frogid-mlops-project-template

```
frogid-mlops-project-template/        <- Root directory
│
├── mlops/                  <- Contains python code, notebooks and ML resources related to one
ML project
│   │
│   ├── requirements.txt        <- Specifies Python dependencies for ML code (for example:
model training, batch inference)
│   │
│   ├── utils/          <- ADDED: Common utilities code modules that the team creates over time
and uses across projects
```

```
│    │
│    ├── databricks.yml          <- databricks.yml is the root bundle file for the ML project that can
be loaded by databricks CLI bundles. It defines the bundle name, workspace URL and resource
config component to be included
│    │
│    ├── training/               <- Training folder contains Notebook that trains and registers the
model with feature store support
│    │
│    ├── feature_engineering/     <- Feature computation code (Python modules) that
implements the feature transforms. Includes preprocessing, feature transformations etc. The
output of these transforms get persisted as Feature Store tables. Most development work
happens here
│    │
│    ├── validation/              <- Optional model validation step before deploying a model
│    │
│    ├── evaluation/              <- Detailed evaluation codebases
│    │
│    ├── monitoring/              <- Model monitoring, feature monitoring, etc.
│    │
│    ├── ml_config/               <- ML pipeline custom config files
│    │
│    ├── deployment/              <- Deployment and Batch inference workflows
│    │    │
│    │    ├── batch_inference/     <- Batch inference code that will run as part of scheduled
workflow
│    │    │
│    │    ├── model_deployment/    <- As part of CD workflow, deploy the registered model by
assigning it the appropriate alias
│    │
│    │
│    ├── tests/                   <- Unit tests for the ML project, including the modules under `features`
│    │
│    ├── resources/               <- ML resource (ML jobs, MLflow models) config definitions
expressed as code, across dev/staging/prod/test
│        │
│        ├── model-workflow-resource.yml              <- ML resource config definition for model
training, validation, deployment workflow
│        │
│        ├── batch-inference-workflow-resource.yml      <- ML resource config definition for batch
inference workflow
│        │
│        ├── feature-engineering-workflow-resource.yml  <- ML resource config definition for
feature engineering workflow
│        │
│       ├── ml-artifacts-resource.yml              <- ML resource config definition for model and
experiment
│       │
│       ├── monitoring-resource.yml                <- ML resource config definition for quality
monitoring workflow
│
├── adhoc_notebooks/            <- ADDED: For any exploratory analysis or early work before
formalizing ML pipelines
│
├── templates/                  <- ADDED: For storing any template code
│
├── .github/                    <- Configuration folder for CI/CD using GitHub Actions. The CI/CD
workflows deploy ML resources defined in the `./resources/*` folder with databricks CLI bundles
│
├── docs/                       <- Contains documentation for the repo
│
├── cicd.tar.gz                 <- Contains CI/CD bundle that should be deployed by
deploy-cicd.yml to set up CI/CD for projects
```

Creating New ML Pipelines

Each new experiment would be made up of new scripts, config files, and would re-use some modules. Instead of overwriting a pipeline, the code for multiple ml pipelines for different experiments and iterations would accumulate in the mono repository, but all themed around the same '**problem'.**

The naming convention for files that make up a new pipeline should align with the Experiment name: {problem}_{target}_{data_selection}. Except we can drop the problem part and use the model target x data selection label {target}_{data_selection}. Ideally the user can find a short name for the unique experiment. Long filenames are annoying, while long experiment registry names are not so annoying.

Git Branching

In general, the approach is to use Feature Branches for a new model experiment, typically owned by a single individual. The team will generate multiple versions of ML pipelines and other code in the repository and create a new branch to work on it before it is merged to main later.

Branch types

- **Feature branches** (short name for experiment) - for a new experiment or other significant change
- **Main** branch (main): The primary branch where code is merged after testing.
- **Release** branch (naming convention TBC) : Branch from which production deployments occur

Workflow

Data scientists develop new ML pipeline experiments in the development environment using development feature branches they create.

To validate ML pipelines before merging to main, data scientists will need to follow the **Databricks Asset Bundle Validation and Deployment** approach to be able to run tests in the CI/CD workflow that confirm databricks compatibility:

The below shell commands provide the essential requirements:

Validation

```
Unset
databricks bundle validate
```

To validate the databricks.yml file and associated configurations. This ensures the bundle is correctly structured and formatted before deployment.

Deployment

```
Unset
databricks bundle deploy -t <target>
```

To deploy the bundle to a specified target, typically dev since deployment to higher environments are managed by CICD

Run the deployed bundle

```
Unset
databricks bundle run -t <target-name> <job-name>
```

To test the job by triggering it in the target environment. Can also achieve by running it from databricks-workflows

Merge to main branch

These are merged to the main branch through a pull request (PR) that needs to be approved.

When mature, the team will apply automation so that a PR against the main branch triggers the code to run in the staging environment with unit tests. When tests pass, code is merged to the main branch, which automatically deploys to the staging environment.

Deploy to production

To promote to production, a release branch is created from the main branch and deployed either by a manual or automated deployment process.