Need for MLflow

Challenges in the traditional ML lifecycle, in points:

- **Two Disconnected Phases:**
  - **R&D (Data Scientists):** Experimentation, model building on local machines (often Jupyter notebooks).
  - **Operations (Ops Teams):** Taking experiments to production, deployment.

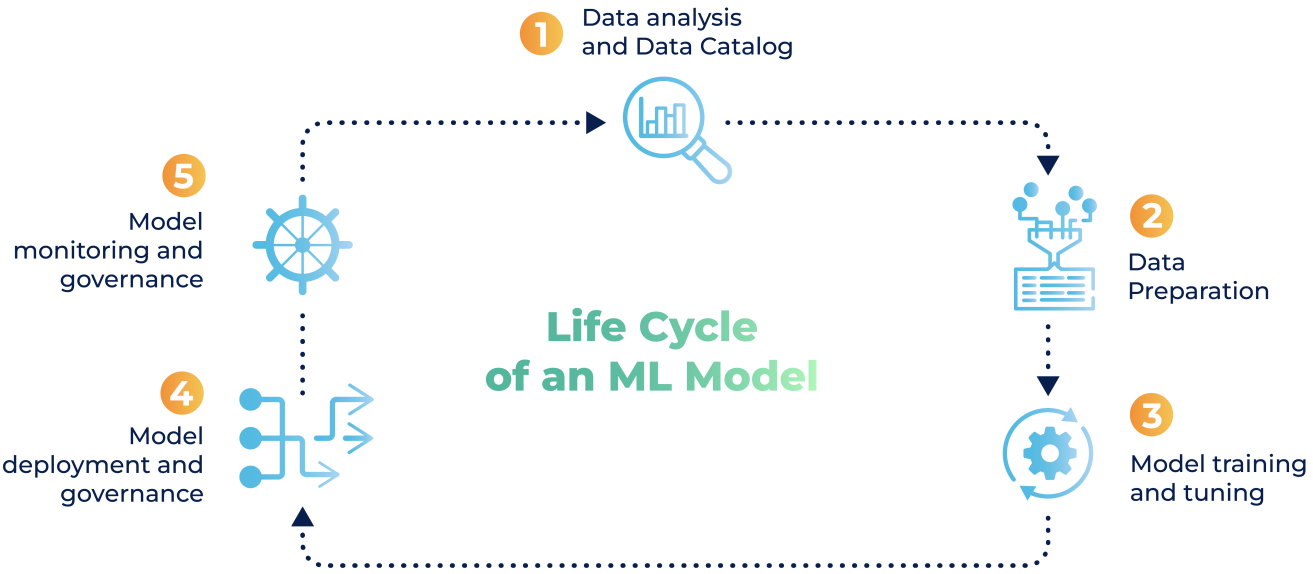
- **The "80% Failure" Problem:** Due to these challenges, 80% of ML models never reach production.
- **Data Scientist Time Waste:** Data scientists spend 30-40% (or even 50%) of their time on non-core activities like explaining code for deployment.
- **Isolation & Lack of Standardization:**
  - Data scientists work in isolation, creating unrobust, unscalable "jumbo notebooks" on local systems.
  - Lack of systematic handoff (emails, USBs, shared drives).
- **Communication & Tooling Gap:**
  - Data scientists and operational teams "don't speak the same language" or use the same tools.
  - Ops teams struggle with dependencies, model inputs/outputs, and scalability concerns.
- **Error-Prone Handoff:** Often leads to code rewriting by engineers, potentially in different languages, changing the model's original intent.
- **"Virtual Wall":** A significant barrier between data scientists (throwing models over the wall) and engineers (trying to understand and deploy).
- **Inefficiency:** This whole process is repetitive, error-prone, and leads to wasted time and resources.

Pre-requisite: The Core Machine Learning Code

Before diving into MLflow, it's crucial to understand that MLflow is **an addition** to your existing machine learning code, not a replacement for it.

- **Necessity of Core ML Code:** MLflow simply wraps around your model-building process.
- **Use Case Selection:** We'll use a basic regression model built with scikit-learn.
- **Target Audience:** Basic understanding of machine learning or experience with simple regression models is assumed. The code will be explained step-by-step.

ML Project Deployment: More Than Just Code

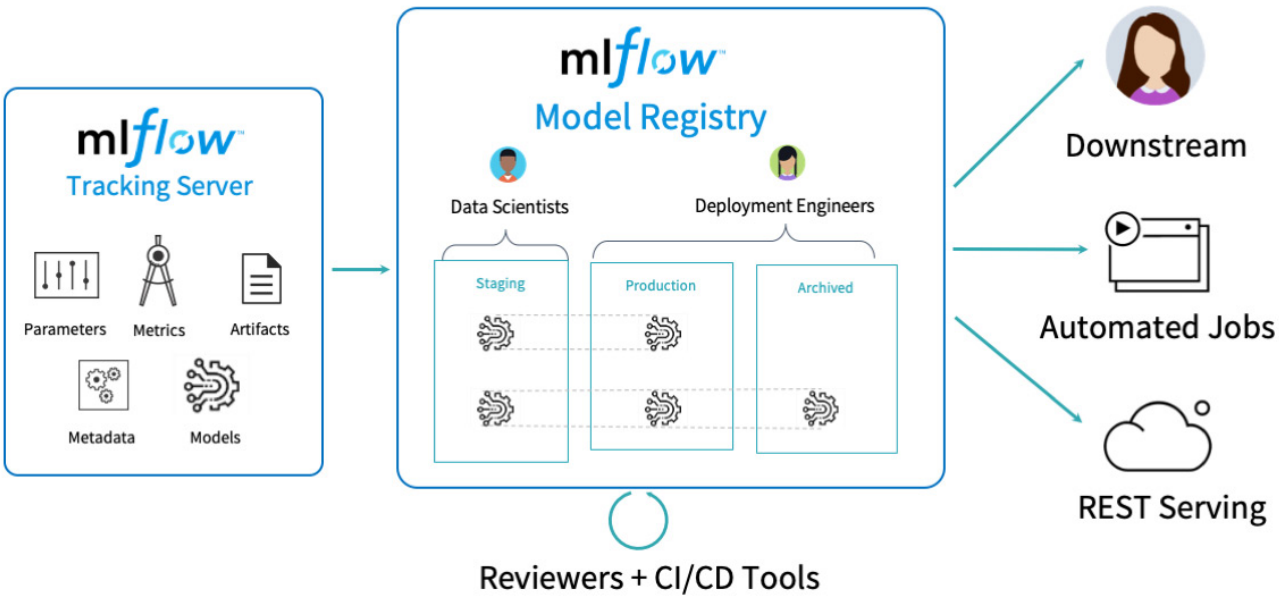


Getting an ML model into production is hard, involving many steps beyond just building the model.

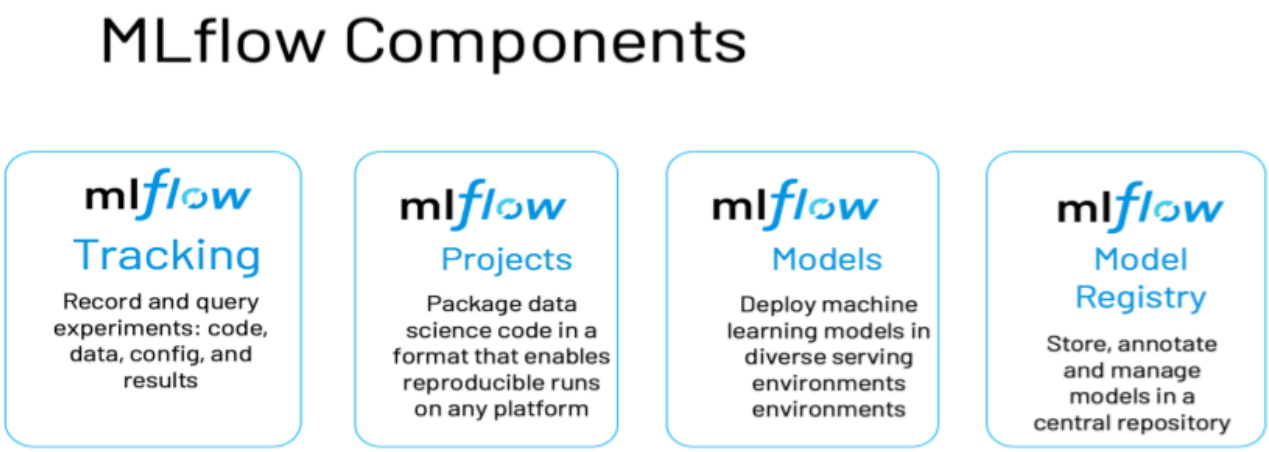
- **Packaging:** Compiling code, managing dependencies.
- **Performance:**
  - **Training:** Scaling for huge data, load balancing, parallelism, GPU support.
  - **Prediction:** Ensuring rapid (milliseconds) and robust predictions for real-time use cases (e.g., fraud detection), even with high data volume.
- **Instrumentation:**
  - **Versioning:** Crucially, versioning **data, algorithms, hyperparameters, and environments**, not just code, for **reproducibility**. Without it, recreating past models is nearly impossible.
  - **Monitoring:** Tracking **model accuracy, data quality, and data drift**, plus system health (latency, CPU/memory). Manual monitoring is too slow; issues are often found too late by users.
- **Automation:** Most tasks (versioning, training, testing) are manual, making retraining (which is inevitable) time-consuming and prone to delays that can jeopardize the project.

In short: The actual ML code is a tiny piece of the puzzle. ML projects are complex, multidisciplinary efforts. Traditional methods lack standardization, leading to major deployment bottlenecks. MLOps aims to solve these challenges through automation and better practices.

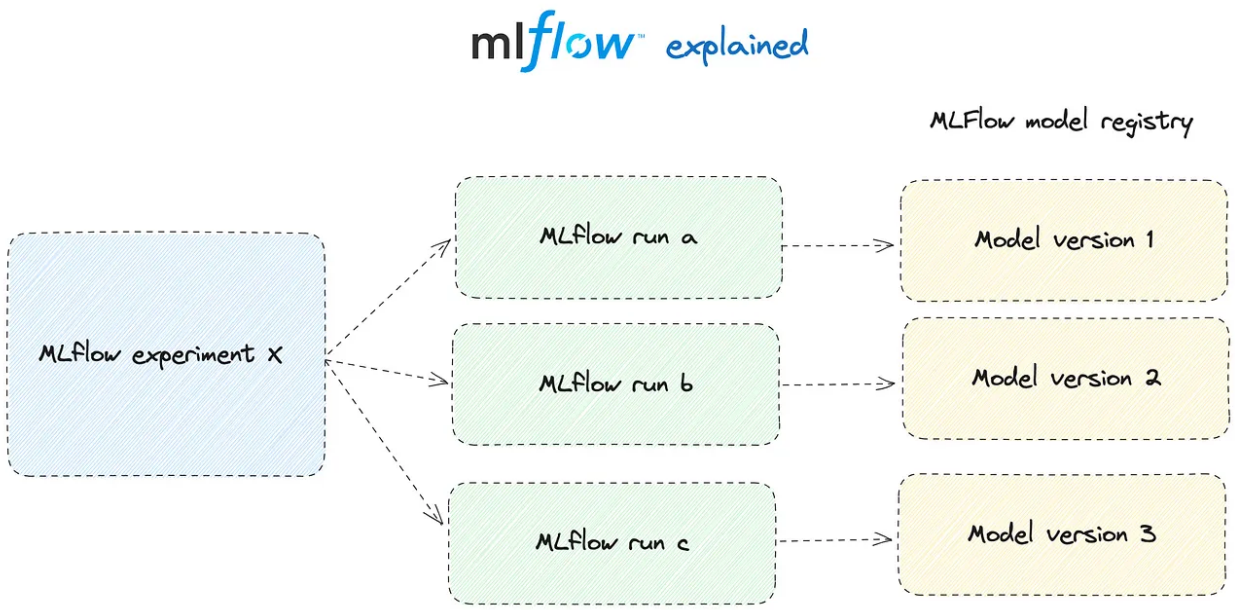
Introduction to MLflow



MLflow components



MLflow Concepts



- **Experiments:** A logical grouping of runs, allowing you to organize and compare a set of related runs. An experiment can contain any number of runs.
- **Run:** A Run in MLflow is a single execution of a machine learning code. Each run can record hyperparameter values, metrics, tags, and artifacts.

Understanding the Core Machine Learning Model: ElasticNet

Our model for predicting wine quality is **ElasticNet**. It's a type of "regularized regression" model, which helps it learn better from data and perform well on new, unseen data.

Regression

- **What it is:** Predicting a **continuous number**.

- **Analogy:** Guessing someone's **exact height** (e.g., 175.5 cm).

- **Goal:** Output a specific numerical value.

Regularization

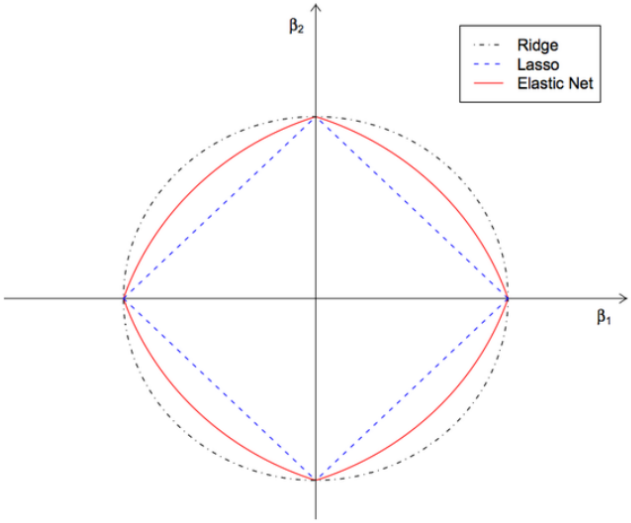
- **What it is:** A technique to **prevent a model from "over-memorizing" (overfitting)** the training data.
- **Analogy:** A student **learning general rules** instead of just memorizing answers for a test. This helps them perform well on new questions.

- **Goal:** Make the model generalize better to **new, unseen data** by adding a "penalty" for complexity.
- **The Challenge of Learning:** Sometimes, models can be too simple, or too complex (getting confused by noise and "over-memorizing"). This is where **regularization** helps.
- **L1 (Lasso) Regularization: The Feature Selector**
    - **Analogy:** Imagine a student with a **strict budget** for facts they can use. They are forced to pick *only the most important ones* and completely ignore irrelevant details.
    - **Function:** In a model, L1 regularization sets the influence of less important features to **exactly zero**, effectively removing them and simplifying the model.
- **L2 (Ridge) Regularization: The Generalizer**
    - **Analogy:** Imagine a student who knows all the facts but tends to give **extreme answers**. This is like telling them to "calm down" and use *all facts in a balanced way*, avoiding over-reliance on any single piece of information.
    - **Function:** L2 regularization shrinks the influence of *all* features, preventing any single one from dominating, leading to a more stable and generalized model.
- **ElasticNet: The Best of Both Worlds**
    - **Analogy:** This is the **super student** who can both pick out the most important facts (L1) AND give balanced answers (L2). It's like having a blending knob to get the perfect mix.

TinyML

Elastic Net Regression

"Comprehensive guide to deploying on edge devices."



- **Function:** ElasticNet combines L1 and L2 penalties. This allows it to:
  - **Select relevant features** (like Lasso).
  - **Maintain generalization properties** (like Ridge).
- **Benefits:** It's robust and versatile, especially for datasets with many features, making it a great choice for our wine quality prediction.
- `alpha` **(Overall Strictness of the Teacher):**
  - **Analogy:** This dial controls the **overall strictness** of the learning process.

  - **Higher** `alpha`**:** More strict, leads to a simpler model (less overfitting).
  - **Lower** `alpha`**:** More relaxed, allows for more complex patterns (risk of overfitting).
  - **Goal:** Find an `alpha` that's strict enough to prevent overfitting but allows useful learning.
- `l1_ratio` **(Blending Knob for L1 and L2):**
  - **Analogy:** This dial acts like a **blending knob on a sound mixer**, adjusting the balance between L1 and L2 styles.
  - `l1_ratio = 1.0`: Pure L1 (Lasso) - aggressive feature selection.

- `l1_ratio = 0.0`: Pure L2 (Ridge) - balanced shrinkage of all features.
- `l1_ratio` between 0.0 and 1.0: A blend (ElasticNet) for both feature selection and generalization.
- **Goal:** Find the right `l1_ratio` for your specific dataset's needs.

The Use Case: Predicting Wine Quality

Our practical example involves predicting wine quality.

- **Dataset:** The well-known Wine Quality dataset, fundamental for regression tasks.
- **Features:** It includes 11 features (e.g., acidity, sugar, chlorides).
- **Target:** The goal is to predict "quality," which is the label.
- **Dataset Details:** A GitHub link for the dataset schema will be provided.

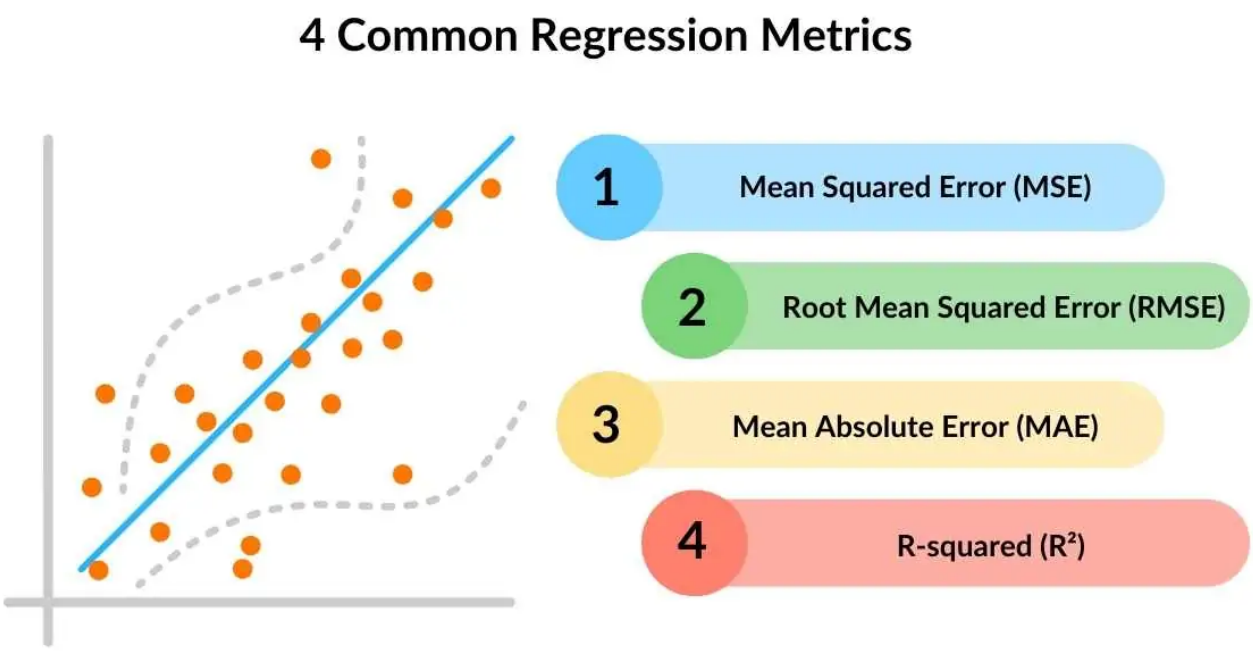
The Pure Machine Learning Code Walkthrough

Before MLflow, we have a standard Python script for model training.

- **Initial Setup:**
  - Importing necessary libraries: `warnings`, `argparse`, `logging`, `pandas` (for data handling), `NumPy` (for math), `scikit-learn` components (for metrics and `ElasticNet`).
  - Basic logging configuration.
- **Command Line Arguments (`argparse`):**
  - A tool to pass `alpha` and `l1_ratio` values directly when running the script from the terminal.
  - Default values for both are set to `0.5`, and they are optional.
- `eval_metrics` **Function:**
  - A custom function to calculate standard regression evaluation metrics: RMSE, MAE, and R2 score.
  - Takes actual and predicted values as input.
- **Main Block Logic (`if __name__ == "__main__":`)**
  - Suppresses warnings and sets a random seed (for reproducibility).
  - **Data Loading:** Reads the `red-wine-quality.csv` dataset.
  - **Data Splitting:** Divides the dataset into 75% training and 25% testing sets using `train_test_split`.
  - **Feature and Label Separation:** Isolates the "quality" column as the target label (`train_y`, `test_y`) and the remaining columns as features (`train_x`, `test_x`).
  - **Hyperparameter Application:** Assigns the `alpha` and `l1_ratio` values (either defaults or from command line) to variables.
  - **Model Instantiation and Training:**
    - Creates an `ElasticNet` model instance using the specified `alpha`, `l1_ratio`, and a `random_state`.
    - Trains the model using the `fit` method on the training data.
  - **Prediction and Evaluation:**
    - Uses the trained model (`lr.predict`) to make predictions on the test dataset.
    - Calls the `eval_metrics` function to compute the RMSE, MAE, and R2 score based on actual vs. predicted wine qualities.
  - **Output:** Prints the hyperparameters used and the calculated evaluation metrics.
- **Current Status:** This code is **pure machine learning**, with no MLflow components integrated yet.

Model Performance and Next Steps

After running our initial code, we got these results for how well our model performed:



- **RMSE:** 0.79
- **MAE:** 0.62
- **R2 Score:** 0.10

Now, let's understand what these numbers mean. They are like different ways to grade your model's test, telling you how well its predictions match the actual correct answers.

Understanding RMSE and MAE

Imagine your model is trying to predict how many cookies each person ate. You have the model's prediction, and you also know the actual number of cookies they ate.

- **RMSE (Root Mean Squared Error):**
    - **What it is:** It's a way to measure the average "mistake" your model made. We calculate the difference between the predicted cookies and the actual cookies for each person, then we square those differences, average them, and finally take the square root.
    - **Analogy:** Think of it like a **penalty system where bigger mistakes are punished much more severely**. If your model is off by just 1 cookie, it gets a small penalty. But if it's off by 2 cookies, it gets *four times* the penalty of being off by 1! This means RMSE really highlights when your model makes a few really big errors.
    - **Goal:** You want **RMSE to be as low as possible (closer to 0)**. A lower RMSE means your model is making smaller mistakes on average, especially avoiding those large, costly errors.
- **MAE (Mean Absolute Error):**
    - **What it is:** This is another way to measure the average "mistake." We calculate the absolute difference (just the positive value of the difference, no negatives) between the predicted cookies and the actual cookies for each person, and then we average all these differences.

    - **Analogy:** This is like a **fairer penalty system where every mistake counts equally, regardless of how big it is.** If your model is off by 1 cookie, it counts as 1. If it's off by 2 cookies, it counts as 2. It doesn't exaggerate bigger errors.
    - **Goal:** You also want **MAE to be as low as possible (closer to 0)**. A lower MAE means your model's predictions are, on average, closer to the actual values.

Understanding R2 Score (Coefficient of Determination)

- **What it is:** This metric tells you how much of the variation in the actual wine quality your model can explain.
- **Analogy:** Imagine you're trying to predict how well students will do on a test.
    - An **R2 of 1.0 (or 100%)** is like saying, "My prediction model can perfectly explain *all* the reasons why students got their specific scores. I know exactly how much influence study time, previous grades, etc., had." This is the ideal scenario.
    - An **R2 of 0.0** means, "My model is no better than just guessing the *average score* for everyone." It explains none of the factors that lead to different scores.
    - An **R2 of -1.0 (or negative)** means, "My model is actually *worse* than just guessing the average score. I'd be better off not even using my model!"
- **Goal:** You want **R2 to be as high as possible (closer to 1.0)**. A higher R2 score means your model is doing a better job at understanding and predicting the variations in wine quality.

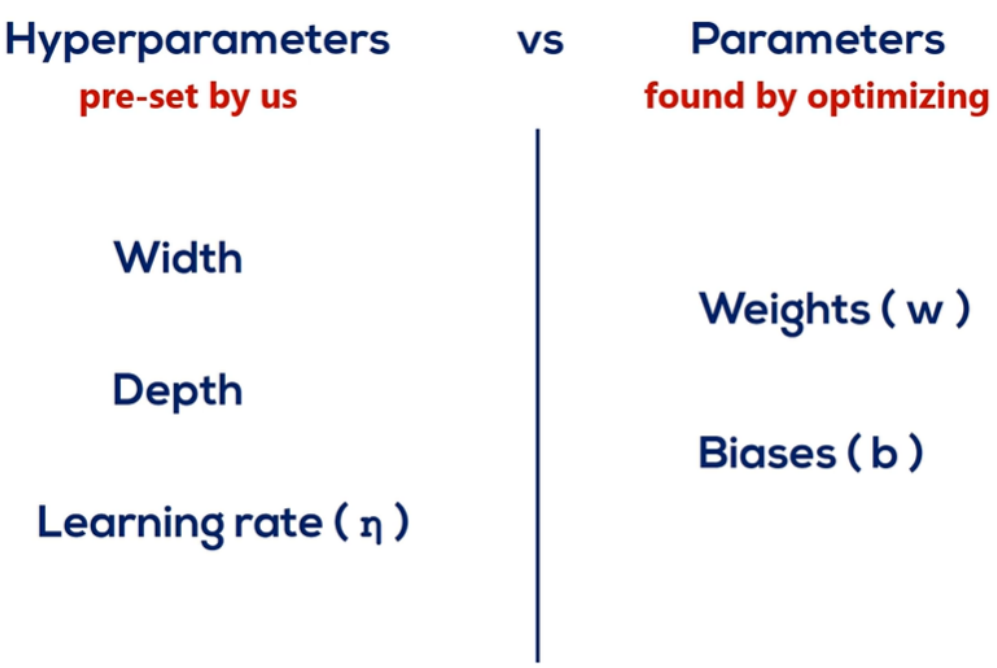
Current Performance and What's Next

Our initial run shows an R2 score of 0.10. This is "not so good," meaning our model isn't explaining much of the wine quality variation yet. However, improving this model's performance isn't our main goal right now.

Here we tracked only hyperparameters, and model metrices but we can even track data, images and other artifacts that we need to keep track of

- **Future Improvement:** We can usually get much better results by **tweaking the hyperparameters** (like `alpha` and `l1_ratio`) we discussed earlier.

Table comparing Parameters and Hyperparameters:



Feature	Parameters (Model Parameters)	Hyperparameters
Who determines?	The **model learns them** directly from the training data.	**You (the ML engineer) set them** before training.
When determined?	**During** the model training process.	**Before** the model training process begins.
What they affect?	The "skill" or "knowledge" of the model; what it has learned.	How the model *learns* its parameters; the learning process itself.
Adjusted by?	Optimization algorithms (e.g., gradient descent).	You, via manual tuning, grid search, random search, etc.
Analogy	The **answers a student writes on a test** after studying.	The **study strategy or teaching method** chosen for the student.
Examples	- Weights and biases in neural networks - Coefficients in linear regression - Cluster centroids in K-Means	- `alpha` and `l1_ratio` in ElasticNet - Learning rate - Number of layers/neurons - `test_size` in data splitting - `random_state`
Are they saved?	Yes, these are what you save when you "save" a trained model.	Often part of the experiment record, but not part of the model itself.
Change per run?	Change with each training run until convergence.	You explicitly change them between different training runs/experiments.

MLflow Folder Structure: Your Experiment Workshop

MLflow organizes your work in a simple, hierarchical way within a default `mlruns` folder:

- `mlruns/`: This is your main **workshop**, holding all experiments.
  - `[experiment_ID]/`: Inside, you'll find numbered folders, each representing a distinct **experiment** (like a project bin).
    - `metal.yaml`: metadata related to the current experiment
    - `[run_ID]/`: Within each experiment, there are unique folders for every **single attempt or "run"** you make (like a specific trial).
      - `artifacts/`: Contains **saved outputs** (your trained model, plots, etc.).
        - `mymodel/`: Contains **conda.yaml, model.pkl, requirements.txt** (conda.yaml,python_env.yaml, requirements.txt to create similar env, trained model, plots, etc.).
      - `metrics/`: Stores **performance scores** (like RMSE, MAE, R2).
      - `params/`: Lists **hyperparameters** used (like `alpha`, `l1_ratio`).
      - `tags`: Holds **metal.yaml** (metadata information of the current run, start time, status, etc.).

It's designed to keep all your machine learning trials neatly organized and easy to review!

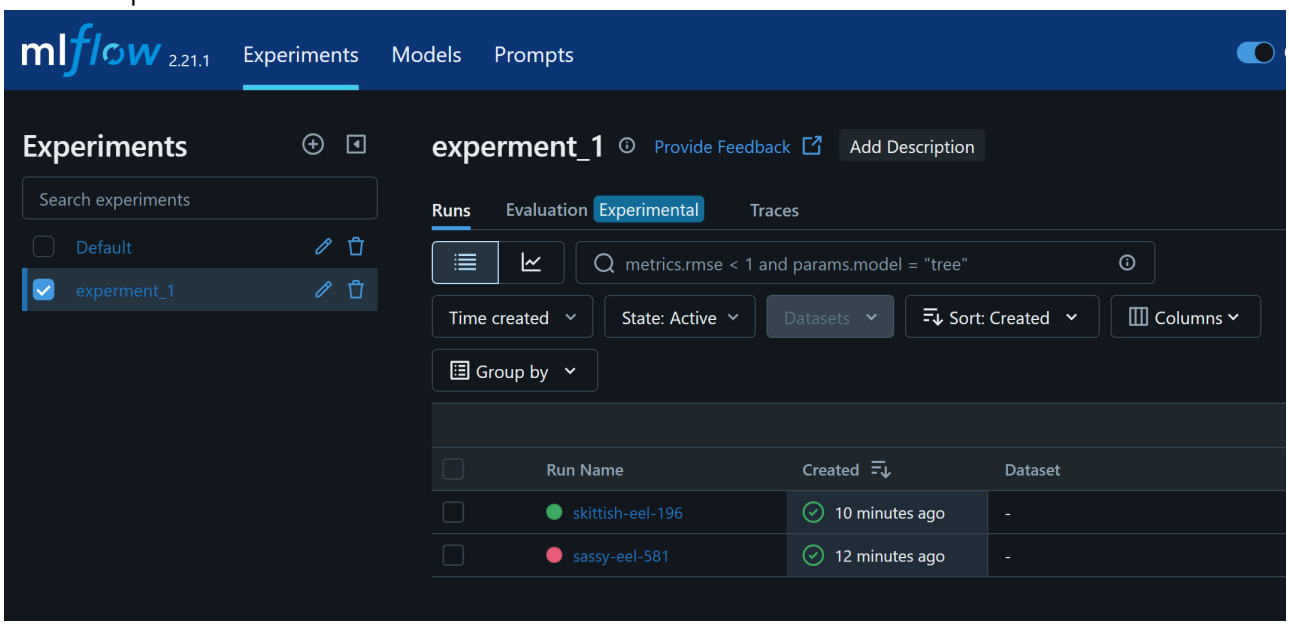
Python code execution

```
python your_script_name.py --alpha 0.8 --l1_ratio 0.2
```

MLflow UI: Your Experiment Dashboard

MLflow UI is your **visual dashboard for machine learning experiments**.

- MLflow experiments and runs



Accessing the UI: Launching Your Dashboard

1. **Activate your Conda environment.**
2. Run `mlflow ui` in your terminal.

3. Copy the link and paste it into your browser. by defualt it runs on localhost:5000

Exploring the UI: What You'll See

- On top it shows experiment ID and artifact location
- Below Runs you can change to either table view or chart view
- You can also sort it, example based on date, metrices etc
- We can show or hide specific columns if we want to from the columns tab
- When you select more than one run, you will get option to compare them

The UI has two main parts:

- **Experiments Section:** Shows all your **experiment runs**.
  - **Experiment List:** Lists your experiments.
  - **Run Details:** Displays individual **runs** in a table (or other views).
    - **Table View:** Compare **metrics** and **parameters** easily.
    - **Insights:** Quickly see the best-performing runs.
    - **Compare Runs:** Select runs to analyze them with various graphs.
  - **Individual Run Details:** Dive deeper into a specific run's details and **artifacts**.
  - **Model Logging vs. Registering:**
    - **Logging:** Tracking a model.
    - **Registering:** Promoting a model to a central repository for wider use.
- **Models Section:** Shows only **registered models**.

Why Use MLflow UI?

MLflow UI helps you **track, compare, and understand your experiments** to find the **best model** for your problem.

References

1. mlflow-log-model