Model Version Control Best Practices

Table of Contents

Introduction

Model version control is the practice of managing and tracking different versions of machine learning models throughout their lifecycle. It involves keeping a record of changes made to a model, its associated code, configuration, and other relevant artifacts. The purpose of model version control is to ensure reproducibility, traceability, collaboration, and effective management of machine learning projects. Following are the key aspects and purposes of model version control. Machine learning version control covers three subparts, code for the ML model, data, metrics, parameters associated with each model. Following points list benefits of the maintaining the model version control.

1. **Reproducibility**
    - **Purpose:** Ensure that a model's results can be reproduced consistently. The ability to recreate a specific model version is crucial for debugging, testing, and auditing.
    - **How:** Record the exact code, data, and configuration used to train a model. By versioning these components, you can recreate the same environment and achieve reproducibility.
2. **Traceability**
    - **Purpose:** Keep track of changes made to a model over time. Understand the evolution of the model, including modifications to its architecture, hyperparameters, and training data.
    - **How:** Use version control systems to log changes, associate them with specific model versions, and document the reasons behind each modification.
3. **Collaboration**
    - **Purpose:** Facilitate collaboration among team members working on the same machine learning project. Multiple data scientists, engineers, or researchers may contribute to different aspects of the project concurrently.
    - **How:** Version control systems enable team members to work on different branches, merge changes, and coordinate efforts effectively. This promotes a structured and collaborative development process.
4. **Debugging and Troubleshooting**
    - **Purpose:** Simplify the process of identifying and resolving issues in models. Debugging becomes more manageable when you can pinpoint when and where changes were introduced.
    - **How:** Associate model predictions with specific versions, trace back to the corresponding code, data, and configurations, and identify the cause of discrepancies or errors.
5. **Continuous Improvement**

- **Purpose:** Support the iterative nature of machine learning model development. Continuous improvement relies on the ability to experiment with different configurations and track the performance of each model version.
- **How:** Easily experiment with different hyperparameters, features, or algorithms by creating new model versions. Compare performance metrics and choose the most effective configurations.

6. **Deployment and Rollbacks**
    - **Purpose:** Streamline the deployment process and enable the ability to rollback to a previous model version if needed. Ensure that only well-tested and validated models are deployed in production.
    - **How:** Integrate version control into your deployment pipeline. Rollback to a previous version in case of issues, minimizing downtime and potential negative impacts.
7. **Collaborative Development Workflows**
    - **Purpose:** Establish structured workflows for collaborative model development. Define guidelines for feature branches, code reviews, and integration into the main development branch.
    - **How:** Use branching strategies to isolate changes, conduct code reviews, and follow a systematic process for merging updates into the main model repository.
8. **Model Governance and Compliance**
    - **Purpose:** Meet governance and compliance requirements by maintaining a clear record of model changes. This is particularly important in regulated industries where auditability is crucial.
    - **How:** Maintain detailed documentation, including model metadata, configuration files, and changelogs. Use version control as part of your model governance framework.

When to Use?

Model versioning should be used for all types of machine learning models. It is the process of saving different versions of a machine learning model as the model is trained. This allows developers to keep track of how the model changes over time and compare different versions of the model to see which one performs the best.

Taking snapshots for the entire machine learning pipeline time to time makes it possible to reproduce the same output again, even with the trained weights, which saves the time of retraining and testing. Additionally, version control tools such as Git help software developers collaborate on the same project and avoid conflicts.

In summary, model versioning should be used when you want to:

- Keep track of how the model changes over time.
- Compare different versions of the model to see which one performs the best.
- Reproduce the same output again, even with the trained weights.
- Collaborate with other software developers on the same project and avoid conflicts.

Best Practices

1. Use a Dedicated Version Control System

All machine learning models are generally stored as large, binary and dynamic files. It makes very hard to manage them in generalized code versioning repositories like GIT. Git repositories are more preferred when we are dealing with majorly normal code base/ static files. Using it to manage ML models results in inefficiency and increases complexity.

Instead, ML models should be managed via dedicated repositories and tools like DVC, MLFlow, Weights and Biases etc.

a. DVC (Data Version Control)

Data Version Control, or DVC, is a data and ML experiment management tool that takes advantage of the existing engineering toolset that we are familiar with (Git, CI/CD, etc.). DVC is meant to be run alongside Git. The Git and DVC commands will often be used in tandem, one after the other. While Git is used to storing and version code, DVC does the same for data and model files.

The .dvc file is lightweight and meant to be stored with code in GitHub. When you download a Git repository, you also get the .dvc files. You can then use those files to get the data associated with that repository. Large data and model files go in your DVC remote storage, and small .dvc files that point to your data go in GitHub.

Please refer to the appendix [DVC Integration] for the implementation details for DVC.

b. MLflow

MLflow is an open-source platform to manage the ML lifecycle, including experimentation, reproducibility, deployment, and a central model registry. The MLflow Tracking component is an API and UI for logging parameters, code versions, metrics, and output files when running your machine learning code and for later visualizing the results. MLflow Tracking lets you log and query experiments using Python, REST, R API, and Java APIs.

Please refer to the appendix [MLFlow] for sample code-based implementation for MLFlow.

c. Weights and Biases

Weights & Biases (W&B) is a platform designed to help machine learning practitioners manage and track their experiments effectively. It provides a suite of tools for experiment tracking, visualization, and collaboration, making it easier for practitioners to develop robust and scalable machine learning models.

The main features and benefits of the W&B platform are:

1. **Experiment Tracking:** W&B allows users to track their experiments by automatically logging model hyperparameters, metrics, and output artifacts. This allows practitioners to keep track of their experiments and compare different models and hyperparameters easily.
2. **Visualizations:** W&B provides a suite of visualization tools that allow users to visualize their experiments and results. These include 3D visualizations, confusion matrices, and scatter plots among others. This helps practitioners to better understand their data and identify patterns and trends in experiment results.
3. **Collaboration:** W&B makes it easy to collaborate with team members by providing features for sharing experiments, data, and results. It also provides version control for code and data, ensuring that everyone is working with the same resources.
4. **Integration:** W&B can easily be integrated with a wide range of machine learning frameworks, including TensorFlow, PyTorch, Keras, and Scikit-learn. This allows practitioners to use W&B with their existing workflows and frameworks, without requiring any significant changes to their development process.
5. **Organization:** W&B provides features for organizing experiments, such as projects, runs, and tags. This makes it easy to manage and track experiments across multiple team members and projects.

Please refer to the section [Weights and Biases] for implementation details.

2. Use a Consistent Naming Convention

To keep track of all the versions properly, it is very necessary to keep proper naming convention.

1. One way to use naming is with <model_name>_<data_version>_<metric>_<value> (i.e., resnet50_v1_accuracy_0.85)
2. Along with this, tags and labels can also be used to mark the model as testing, production-ready, experimental etc.
    a. Marking a model with tags allows to filter, sort and organize different models efficiently.

Model Naming Strategies

Model versioning is a critical aspect of managing machine learning models, ensuring traceability, reproducibility, and efficient collaboration. Different versioning strategies can be employed to

uniquely identify and document changes in models over time. Here are four model versioning strategies:

1. **Sequential Versioning**
    a. Sequential versioning involves assigning a numerical identifier to each version of the model in a sequential order.
    b. Each new version receives the next available number, indicating the order of its creation. For example, version 1, version 2, and so forth.
    c. This straightforward approach provides a clear chronological history of model development, making it easy to track changes over time. However, it may lack semantic information about the nature of the changes introduced in each version.

2. **Semantic Versioning**
    a. One challenge in versioning ML models is effectively communicating the changes and compatibility of different versions.
    b. A common solution to this challenge is to use semantic versioning, which is a standardized format for assigning version numbers to software products.
    c. Semantic versioning consists of three numbers: major, minor, and patch.
    d. A major version change indicates a significant alteration in the model's functionality or performance. It implies that the update might break compatibility with previous versions.
    e. A minor version change signifies a minor improvement or addition to the model that does not affect compatibility with previous versions.
    f. A patch version change indicates a bug fix or a minor adjustment that does not alter the model's functionality or performance.

3. **Hash-based Versioning**
    a. Hash-based versioning generates a unique identifier for each model version by hashing its content.
    b. The hash value, typically derived from a cryptographic hash function like SHA-256, serves as a fingerprint for the model's configuration, weights, and code. This ensures that even minor changes result in distinct version identifiers.
    c. While highly precise, hash-based versioning may not convey semantic information about the changes made, making it more suitable for fine-grained tracking rather than high-level documentation.

4. **Tagging and Labeling**
    a. Tagging and labeling involve assigning human-readable tags or labels to specific model versions.
    b. These tags often provide additional context or information about the purpose of a particular version, such as "production-ready," "experiment-1," or "optimized-performance."

c. Tags are particularly useful when collaborating in a team or when marking significant milestones in the model development process. While tags do not inherently convey the sequential order of versions, they offer valuable insights into the model's evolution.

3. Document the Model Version

One of the crucial practices for versioning ML models is to thoroughly and clearly document each model version. Documentation is essential for understanding, reproducing, and collaborating on ML models.

1. Document not only the model code but also the associated data, parameters, metrics, and artifacts for each model version.
2. Include comprehensive information about the model's rationale, assumptions, and limitations in the documentation.
3. Document any challenges, issues, or feedback encountered or received during the development or usage of each model version.
4. Documentation can take various forms, including comments within code, README files, Jupyter notebooks, or detailed reports.
5. Embed comments directly within the code to provide insights into specific sections, functions, or algorithms, enhancing code readability.
6. Use README files to provide a high-level overview of the model, instructions for usage, and details about dependencies and setup.
7. Jupyter/Databricks/Synapse/Fabric notebooks can be utilized to create interactive and executable documents, combining code, visualizations, and explanatory text.
8. Generate detailed reports that comprehensively describe the model, its architecture, training process, and evaluation metrics.
9. Documenting allows for easy reproducibility of experiments and results, ensuring consistency across different environments or by other team members.
10. Clear documentation facilitates collaboration by providing a shared understanding of the model, its intricacies, and the decisions made during development.

4. Storing Code and Model Together

Code and model files should be kept together. It allows better management of the model. Here, Code files basically refers to the scripts, notebooks, or programming files used to develop, train, and evaluate the machine learning model. This includes the code that defines the model

architecture, handles data preprocessing, specifies hyperparameters, and orchestrates the training process.

Model files are files of the trained machine learning model, which is generated by applying the code to a dataset during the training process. This includes the learned weights, parameters, or any other information that represents the knowledge the model gained from the training data.

Please refer to the sample folder structure for the same.

```
project_root/
│
├── code/
│   ├── train_model.py    # Code to train the machine learning model
│   ├── preprocess_data.py # Code for data preprocessing
│   └── utils.py           # Utility functions used in the code
│
├── data/
│   ├── training_data.csv  # Training dataset
│   └── testing_data.csv   # Testing dataset
│
└── models/
    ├── model.pkl          # Trained machine learning model (serialized)
    └── model_info.txt     # Additional information about the model.
```

Benefits of Storing the Code and Model Together

1. **Reproducibility:** Storing the code alongside the model ensures that the exact codebase used to train and create the model is available. This is crucial for reproducibility – the ability to recreate the same model in the future, even if changes have occurred in the codebase or dependencies.
2. **Version Consistency:** Having the code and model together helps maintain version consistency. Different versions of a model might require specific versions of the code or dependencies. By keeping them together, you avoid mismatches that could lead to unexpected behavior.
3. **Ease of Collaboration:** When multiple team members are working on a project, having code and model together facilitates collaboration. Team members can access the complete set of resources needed for model development, making it easier to understand, review, and contribute to the project.
4. **Understanding Model Decisions:** Machine learning models often involve complex algorithms and processes. Storing the code with the model allows anyone examining the model to directly access the code and understand how decisions are made. This transparency is essential for model interpretation and debugging.

5. **Reduced Dependency on Environment:** Models can be sensitive to changes in the environment, such as updates to libraries or changes in system configurations. Storing the code ensures that the model can be recreated in a consistent environment, reducing dependency on specific system configurations.
6. **Efficient Model Deployment:** When deploying a model to production, having the code readily available simplifies the deployment process. The deployment environment can refer to the codebase to understand how the model was trained and make necessary adjustments.
7. **Continuous Integration:** Integrating the model training code into a continuous integration (CI) system becomes more straightforward when the code and model are stored together. CI systems can automate testing, training, and validation processes, ensuring that the code and model remain compatible.
8. **Organized Project Structure:** Storing code and models together often leads to a well-organized project structure. Code, data, and other project-related files are grouped together, making it easier to manage and navigate the project.

5. Implement a Proper Roll Back Mechanism

Once a model is deployed to an environment but the evaluation results and user response for the model is not proper, in such cases, it is crucial to back track the model issues and meanwhile allow user to continue to access the model. A role back mechanism should be in-place with MLOps automation which allows us to take down the inefficient model for the backtracking and re-deploy the previous working model version on the working environment. The rollback mechanism should ensure that switching between model versions does not negatively impact on the availability and performance of your service.

Rollback mechanisms majorly depend on the adapted deployment mechanism for Production. Primarily Blue-Green deployment, Canary deployment strategies are basic strategies. Apart from those, Shadow deployment and Feature Flag also supports the Rollback of ML models.

6. Follow Incremental Development Method

Incremental deployment stands out as a best practice for controlling model versioning in machine learning projects due to its agility and adaptability. By breaking down the development process into smaller, manageable iterations, incremental deployment allows for systematic updates to the model, ensuring that enhancements, modifications, or optimizations can be introduced incrementally. This approach not only facilitates a more flexible and responsive development cycle but also provides a clear versioning structure. Each iteration represents a distinct version of the model, allowing for easy tracking, documentation, and identification of changes. With incremental deployment, the model's evolution becomes a step-by-step journey, minimizing the risk of introducing major errors or disruptions. This practice aligns well with the iterative nature of

machine learning development, enabling data scientists and developers to quickly adapt to evolving requirements and continuously improve model performance without sacrificing stability.

7. Configure Proper Environments

Deployment of Machine Learning models also should be properly setup for CI/CD integration. Proper Dev, QA and Production workspaces should be setup to subscribe to proper development, quality assurance and inferencing of the models.

1. Dev environment should be setup and required resources should be deployed with Dev SKUs (minimal required for Development to optimize the cost).
    a. Developers should be given Contributor access granting them ease in development and creating required resources.
    b. Proper naming convention for the resources should be following providing ease in finding the resources in the Development resource group or across all resource groups in the portal.
    c. To keep track of Cost, for each project, separate resource group should be created.
    d. Proper repositories should be created related to each technology for preservation of the code and ease of CI/CD.
    e. Code getting merged in repository should be thoroughly reviewed. Latest tools like Microsoft Copilot and GPT models also can be integrated to automatically review of the code.
    f. Proper proof of execution of end-to-end run for a stream should be specified.

2. Testing / Quality Assurance environment should be there to test the development changes as there can be automation based changes or some changes or issues which are not detected on Dev environment can be rectified and resolved.
    a. No access should be given to Dev team for the QA.
    b. Proper policies can be defined carry out the runs on QA environment.
    c. All issues encountered in QA should be logged as Bugs and should be fixed before moving to Production.
    d. Depending on the Cost associated, some of the resources can be shared across environment like OpenAI resource where capacity is allocated at tenant level.

3. Production environment should be independent of any interference from Develop team.
    a. Only Infra team should have control over the Production runs and review of pipeline.
    b. Based on Production run documentation, production runs should be carried out by Infra team in an automatic way.
    c. All failures should be sent via mails to Develop team.
    d. All resources should be newly created utilizing the maximum required SKUs as per the consumption of the resource.
    e. No password should be stored as parameter in any resource which can be easily read simply by a reader role on the resource.

    i. All credentials used by the Azure Function App should be stored or utilized from Key Vault or via other authentication mechanism as reader role allows users to view the configurations of the Function App.

f. If there is no role with only reader access available, then one custom role should be created.
g. Proper documentation and knowledge transfer should be given to team managing the infrastructure.

APPENDIX-A

DVC Integration

Folder Structure Setup

```
project_root
|
├── data
|   ├── processed
|   |   ├── train
|   |   └── test
|   └── raw
|       └── [raw data file]
|
├── data_given
|   └── [data from source]
|
├── report
|   └── [placeholder for JSON files]
|
├── saved_models
|   └── [model pickle file]
|
└── src
    ├── get_data.py
    ├── load_data.py
    ├── split_data.py
    └── model_building.py
```

Sample Files Inclusion Hierarchy

```
project_root
|
├── data
|   ├── processed
|   |   ├── test_SAHeart.csv
|   |   └── train_SAHeart.csv
|   └── raw
|       └── SAHeart.csv
|
├── data_given
```

```
│   └── SAHeart.csv
│
├── report
│
├── saved_models
│   └── model.joblib
│
└── src
    ├── get_data.py
    ├── load_data.py
    ├── split_data.py
    └── train_and_evaluate.py
```

Installation

1. Install the DVC using the following command.

```
pip install dvc
```

2. Once installed, initialize it using following code.

```
dvc init
```

3. Now map the storage account/data with following command.
    a. Following code provides the mapping with local system.

```
dvc add . /data_given/SAHeart.csv
```

    b. DVC also supports cloud-based storage systems like Google Cloud, AWS S3 Buckets, and Microsoft Azure Blob Storage.

4. Create and add metrics file scores.json to note down the scored and place it under the **report** folder shown in above folder structure hierarchy.

Build DVC Pipeline

DVC Pipeline consists of the model versioning steps like data loading, pre-processing, data splitting, model building, metrics measurement etc. It contains mapping for dependencies, and output of the model. Internally, it builds up one Directed Acyclic Graph (DAG) which states the series of steps that can be executed in one direction.

Pipeline YAML Creation

Pipeline will follow three major steps: Load Data-> Split Data-> Train and Evaluate

Please refer to the sample YAML file here. **Filename: dvc.yaml**

```yaml
stages:
  load_data:
    cmd: python src/load_data.py --config=params.yaml
    deps:
    - src/get_data.py
    - src/load_data.py
    - data_given/SAHeart.csv
    outs:
    - data/raw/SAHeart.csv

  split_data:
    cmd: python src/split_data.py --config=params.yaml
    deps:
    - src/split_data.py
    - data/raw/SAHeart.csv
    outs:
    - data/processed/train_SAHeart.csv
    - data/processed/test_SAHeart.csv

  train_and_evaluate:
    cmd: [python src/train_and_evaluate.py --config=params.yaml, dvc plots show]

    deps:
    - data/processed/train_SAHeart.csv
    - data/processed/test_SAHeart.csv
    - src/train_and_evaluate.py

    metrics:
    - report/scores.json:
        cache: false

    plots:

    - report/prc.json:
        cache: false
        x: recall
        y: precision
    - report/roc.json:
        cache: false
        x: fpr
        y: tpr
```

```
    - cm.csv:
        cache: false


    outs:
    - saved_models/model.joblib

#
```

1. DAG graph for the pipeline can be visualized using following command.

```
dvc dag
```

2. Update all the steps of the pipline and make pipeline up-to-date/ready to execute.

```
dvc repro
```

3. Track changes with GIT using following command. With this dvc creates a file dvc.lock to track the changes.

```
git add dvc.lock
```

4. Use the following command to send update to remote storage.

```
dvc push
```

5. Above execution will populate the scores.json file. It can be displayed using the following command.

```
dvc metrics show
```

6. Use the following command to push changes to git.

```
git add . && git commit -m "experiment with dvc" && git push origin main
```

Plot the Metrics

1. Use the following command to plot the visuals.

```
dvc plots show cm.csv --template confusion -x chd -y Predicted
```

Please refer to the following links for more details about the setup

Git Repository: Link

Blog: Link

MLFlow – Model Registry

1. Load the Dataset

Following cells load a dataset containing weather data and power output information for a wind farm in the United States. The dataset contains wind direction, wind speed, and air temperature features sampled every eight hours (once at 00:00, once at 08:00, and once at 16:00), as well as daily aggregate power output (power), over several years.

```
import pandas as pd
wind_farm_data = pd.read_csv("https://github.com/dbczumar/model-registry-demo-notebook/raw/master/dataset/windfarm_data.csv", index_col=0)

def get_training_data():
  training_data = pd.DataFrame(wind_farm_data["2014-01-01":"2018-01-01"])
  X = training_data.drop(columns="power")
  y = training_data["power"]
  return X, y

def get_validation_data():
  validation_data = pd.DataFrame(wind_farm_data["2018-01-01":"2019-01-01"])
  X = validation_data.drop(columns="power")
  y = validation_data["power"]
  return X, y

def get_weather_and_forecast():
  format_date = lambda pd_date : pd_date.date().strftime("%Y-%m-%d")
  today = pd.Timestamp('today').normalize()
  week_ago = today - pd.Timedelta(days=5)
  week_later = today + pd.Timedelta(days=5)

  past_power_output = pd.DataFrame(wind_farm_data)[format_date(week_ago):format_date(today)]
  weather_and_forecast = pd.DataFrame(wind_farm_data)[format_date(week_ago):format_date(week_later)]
  if len(weather_and_forecast) < 10:
    past_power_output = pd.DataFrame(wind_farm_data).iloc[-10:-5]
    weather_and_forecast = pd.DataFrame(wind_farm_data).iloc[-10:]

  return weather_and_forecast.drop(columns="power"), past_power_output["power"]
```

2. Train the Model

```
def train_keras_model(X, y):
```

```
  import tensorflow.keras
  from tensorflow.keras.models import Sequential
  from tensorflow.keras.layers import Dense

  model = Sequential()
  model.add(Dense(100, input_shape=(X_train.shape[-1],), activation="relu",
name="hidden_layer"))
  model.add(Dense(1))
  model.compile(loss="mse", optimizer="adam")

  model.fit(X_train, y_train, epochs=100, batch_size=64, validation_split=.2)
  return model

import mlflow

X_train, y_train = get_training_data()

with mlflow.start_run():
  # Automatically capture the model's parameters, metrics, artifacts,
  # and source code with the `autolog()` function
  mlflow.tensorflow.autolog()

  train_keras_model(X_train, y_train)
  run_id = mlflow.active_run().info.run_id
```

3. Register the MLFlow Model using API

```
model_name = "power-forecasting-model"

model_name = get_model_name()

import mlflow

# The default path where the MLflow autologging function stores the TensorFlow
Keras model
artifact_path = "model"
model_uri = "runs:/{run_id}/{artifact_path}".format(run_id=run_id,
artifact_path=artifact_path)

model_details = mlflow.register_model(model_uri=model_uri, name=model_name)

import time
from mlflow.tracking.client import MlflowClient
from mlflow.entities.model_registry.model_version_status import
ModelVersionStatus
```

```
# Wait until the model is ready
def wait_until_ready(model_name, model_version):
  client = MlflowClient()
  for _ in range(10):
    model_version_details = client.get_model_version(
      name=model_name,
      version=model_version,
    )
    status = ModelVersionStatus.from_string(model_version_details.status)
    print("Model status: %s" % ModelVersionStatus.to_string(status))
    if status == ModelVersionStatus.READY:
      break
    time.sleep(1)

wait_until_ready(model_details.name, model_details.version)
```

4. Add Model and Model Version Description using API

```
from mlflow.tracking.client import MlflowClient

client = MlflowClient()
client.update_registered_model(
  name=model_details.name,
  description="This model forecasts the power output of a wind farm based on weather data. The weather data consists of three features: wind speed, wind direction, and air temperature."
)

client.update_model_version(
  name=model_details.name,
  version=model_details.version,
  description="This model version was built using TensorFlow Keras. It is a feed-forward neural network with one hidden layer."
)
```

5. Transition a Model Version and Retrieve Details using API

```
client.transition_model_version_stage(
  name=model_details.name,
  version=model_details.version,
  stage='production',
)
model_version_details = client.get_model_version(
  name=model_details.name,
```

```
  version=model_details.version,
)
print("The current model stage is:
'{stage}'".format(stage=model_version_details.current_stage))

latest_version_info = client.get_latest_versions(model_name,
stages=["production"])
latest_production_version = latest_version_info[0].version
print("The latest production version of the model '%s' is '%s'." % (model_name,
latest_production_version))
```

6. Load Version of Registered Model Using API

```
import mlflow.pyfunc

model_version_uri = "models:/{model_name}/1".format(model_name=model_name)

print("Loading registered model version from URI:
'{model_uri}'".format(model_uri=model_version_uri))
model_version_1 = mlflow.pyfunc.load_model(model_version_uri)

model_production_uri =
"models:/{model_name}/production".format(model_name=model_name)

print("Loading registered model version from URI:
'{model_uri}'".format(model_uri=model_production_uri))
model_production = mlflow.pyfunc.load_model(model_production_uri)
```

7. Create a New Model Version

```
import mlflow.sklearn
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

with mlflow.start_run():
  n_estimators = 300
  mlflow.log_param("n_estimators", n_estimators)

  rand_forest = RandomForestRegressor(n_estimators=n_estimators)
  rand_forest.fit(X_train, y_train)

  val_x, val_y = get_validation_data()
  mse = mean_squared_error(rand_forest.predict(val_x), val_y)
  print("Validation MSE: %d" % mse)
  mlflow.log_metric("mse", mse)
```

```
  # Specify the `registered_model_name` parameter of the
`mlflow.sklearn.log_model()`
  # function to register the model with the MLflow Model Registry. This
automatically
  # creates a new model version
  mlflow.sklearn.log_model(
    sk_model=rand_forest,
    artifact_path="sklearn-model",
    registered_model_name=model_name,
  )
```

8. Fetch the New Model Version ID using MLFlow Model Registry Search

```
from mlflow.tracking.client import MlflowClient
client = MlflowClient()

model_version_infos = client.search_model_versions("name = '%s'" % model_name)
new_model_version = max([model_version_info.version for model_version_info in
model_version_infos])

wait_until_ready(model_name, new_model_version)
```

9. Add Description to the New Model

```
client.update_model_version(
  name=model_name,
  version=new_model_version,
  description="This model version is a random forest containing 100 decision
trees that was trained in scikit-learn."
)
```

10. Transition the New Model Version to Staging and Test the Model

```
client.transition_model_version_stage(
  name=model_name,
  version=new_model_version,
  stage="Staging",
)

forecast_power(model_name, "Staging")
```

11. Deploy the New Model Version to Production

```
client.transition_model_version_stage(
```

```
  name=model_name,
  version=new_model_version,
  stage="production",
)

forecast_power(model_name, "production")
```

Following codes can be used to Archive versions which are not required and delete a version.

12. Archive Version 1 Using MLFlow API

```
from mlflow.tracking.client import MlflowClient

client = MlflowClient()
client.transition_model_version_stage(
  name=model_name,
  version=1,
  stage="Archived",
)
```

13. Delete Version 1 using MLFlow API

```
client.delete_model_version(
   name=model_name,
   version=1,
)
```

14. Delete Model using MLFlow API

```
from mlflow.tracking.client import MlflowClient

client = MlflowClient()
client.transition_model_version_stage(
  name=model_name,
  version=2,
  stage="Archived",
)
```

```
client.delete_registered_model(name=model_name)
```

Weights and Biases

Deploy simple model using W&B and ServiceFoundry

1. Install wandb.

```
pip install wandb -qU
```

2. Login to W & B account.

```
import wandb
wandb.login()
```

3. Install servicefoundry.

```
pip install -U "servicefoundry"
```

4. Login to servicefoundry.

```
sfy login
```

5. Use the following code to train and save the file.

```
import wandb
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
import joblib

# Initialize W&B
wandb.init(project="iris-logistic-regression")

# Load and preprocess the data
X, y = load_iris(as_frame=True, return_X_y=True)
X = X.rename(columns={
        "sepal length (cm)": "sepal_length",
        "sepal width (cm)": "sepal_width",
        "petal length (cm)": "petal_length",
        "petal width (cm)": "petal_width",
})

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
```

```
)

# Initialize the model
clf = LogisticRegression(solver="liblinear")

# Fit the model
clf.fit(X_train, y_train)

# Evaluate the model
preds = clf.predict(X_test)

# Log the model and the evaluation metrics to W&B
joblib.dump(clf, "model.joblib")
wandb.save("model.joblib")

# Finish the run
wandb.finish()
```

6. We will have to create two files to deploy on truefoundry, a app.py file that contains our application code, and a requirements.txt file that contains our dependencies.

```
.
├── app.py
├── deploy.py
└── requirements.txt
```

File: app.py

In this file, we will pass name of the saved file and run path to wandb.restore() function.

```
import os
import wandb
import joblib
import pandas as pd
from fastapi import FastAPI

wandb.login(key=os.environ["WANDB_API_KEY"])

# Retrieve the model from W&B
model_joblib = wandb.restore(
    'model.joblib',
    run_path="sarathis-tfy/iris-logistic-regression/00r5xvyy",
  )

model = joblib.load(model_joblib.name)
```

```
# Load the model
app = FastAPI(root_path=os.getenv("TFY_SERVICE_ROOT_PATH"))

@app.post("/predict")
def predict(
    sepal_length: float, sepal_width: float, petal_length: float, petal_width:
float
):
    data = dict(
        sepal_length=sepal_length,
        sepal_width=sepal_width,
        petal_length=petal_length,
        petal_width=petal_width,
    )
    prediction = int(model.predict(pd.DataFrame([data]))[0])
    return {"prediction": prediction}
```

File: requirement.txt

```
fastapi
joblib
numpy
pandas
scikit-learn
uvicorn
wandb
```

File: deploy.py

In this file, we will use servicefoundry's python SDK to configure the deployment.

```
import argparse
import logging
from servicefoundry import Build, PythonBuild, Service, Resources, Port

# Setup the logger
logging.basicConfig(level=logging.INFO)

# Setup the argument parser
parser = argparse.ArgumentParser()
parser.add_argument("--workspace_fqn", required=True, type=str)
parser.add_argument("--wandb_api_key", required=True, type=str)
args = parser.parse_args()
```

```
service = Service(
    name="fastapi",
    image=Build(
        build_spec=PythonBuild(
            command="uvicorn app:app --port 8000 --host 0.0.0.0",
            requirements_path="requirements.txt",
        )
    ),
    ports=[
        Port(
            port=8000,
            host="ml-deploy-aditya-ws-8000.demo.truefoundry.com",
        )
    ],
    resources=Resources(
        cpu_request=0.25,
        cpu_limit=0.5,
        memory_request=200,
        memory_limit=400,
        ephemeral_storage_request=200,
        ephemeral_storage_limit=400,
    ),
    env={
        "WANDB_API_KEY": args.wandb_api_key
    }
)
service.deploy(workspace_fqn=args.workspace_fqn)
```

7. Deploy the model via ServiceFoundry.
    a. Run following command, here pass the Workspace FQN and Wandb API key.

    ```
    python deploy.py --workspace_fqn "<Your Workspace FQN>" --wandb_api_key "<Your Wandb API Key>"
    ```