"""

MLOps Project: End-to-End Customer Churn Prediction with MLflow

1. Introduction

Project Overview

This project undertakes the development and management of a machine learning model for predicting customer churn within the telecommunications sector. The core focus is on demonstrating a comprehensive MLOps (Machine Learning Operations) lifecycle implementation using MLflow. MLOps practices are crucial for bridging the gap between model development and operational deployment, ensuring reliability, scalability, and continuous improvement of machine learning systems. By systematically applying MLOps principles, this project aims to showcase how MLflow can be leveraged to streamline various stages, including experiment tracking, model training and tuning, robust deployment, ongoing performance monitoring, and meticulous model versioning through a model registry. The telecommunications industry faces significant challenges due to customer churn, making accurate prediction and proactive retention strategies vital for business success. This project provides a practical, end-to-end view of managing an ML model in a way that mirrors real-world deployment scenarios, emphasizing the importance of a structured and automated approach to the ML lifecycle.

Objectives

The primary objectives of this project are designed to provide hands-on experience with MLflow and a deep understanding of effective ML model management. These objectives are:

1. **Experiment Tracking:** To implement and comprehensively demonstrate how MLflow can be utilized to meticulously track a variety of machine learning experiments. This includes the systematic logging of all relevant parameters (e.g., hyperparameters, feature sets), performance metrics (e.g., accuracy, F1-score, AUC), and generated outputs or artifacts (e.g., trained model files, visualizations, preprocessor objects).

2. **Model Training and Tuning:** To develop multiple machine learning models for the churn prediction task and employ MLflow to log distinct training sessions. This involves tracking experiments with varying model architectures, parameters, and showcasing the process of hyperparameter tuning, ensuring all trials and their outcomes are recorded for reproducibility and analysis.

3. **Model Deployment:** To package the optimally trained machine learning model using MLflow's standardized model packaging utilities. Subsequently, to deploy this packaged model as an accessible service, capable of providing real-time or batch predictions, thereby demonstrating a practical pathway from a trained model to a usable application.

4. **Performance Monitoring:** To establish and outline robust mechanisms for continuously monitoring the performance of the deployed model over time. This involves utilizing MLflow to track potential drifts in key model metrics, simulating the arrival of new data, and re-evaluating the model to ensure its predictive accuracy and reliability are maintained in a dynamic environment.

5. **Model Registry:** To effectively utilize MLflow’ s Model Registry for managing the complete lifecycle of the developed models. This includes version control for different model iterations and managing transitions between various stages, such as from 'Staging' to 'Production' and 'Archived', ensuring a governed and auditable model management process.

These objectives collectively aim to provide a holistic understanding of the MLOps lifecycle, emphasizing MLflow's role in achieving efficient, reproducible, and production-ready machine learning solutions.

2. Methodology

Domain and Dataset Selection

- **Domain:** The project is situated within the **Telecommunications** industry. This sector is characterized by high competition and a continuous need to retain customers, making churn prediction a high-value use case.
- **Task:** The specific machine learning task is **Customer Churn Prediction**, a binary classification problem where the goal is to predict whether a customer is likely to discontinue their service (churn) or not.
- **Dataset:** The dataset chosen for this project is the **"Telco Customer Churn"** dataset, widely available and commonly used for this type of prediction task (e.g.,

from Kaggle). This dataset typically includes a range of customer-specific information such as:
    - **Services Subscribed:** Details about services like phone, internet, online security, tech support, streaming TV, and movies.
    - **Account Information:** Tenure with the company, contract type (month-to-month, one-year, two-year), payment method, paperless billing status, monthly charges, and total charges.
    - **Demographic Information:** Gender, senior citizen status, and whether the customer has partners or dependents. The target variable is 'Churn', indicating 'Yes' if the customer churned and 'No' otherwise. The selection of this dataset allows for a realistic exploration of feature engineering, model training, and evaluation relevant to the telecommunications domain. The data acquisition is simulated in the `scripts/01_download_data.py` script, which currently creates a placeholder CSV. For actual use, this script would be modified to download the dataset from its source, for instance, using the Kaggle API.

MLflow Setup and Configuration

The MLflow environment for this project is configured to support all stages of the MLOps lifecycle. The setup is primarily designed for local execution but follows principles applicable to more extensive cloud-based deployments.

- **Installation:** MLflow and its dependencies (including common ML libraries like Scikit-learn, Pandas, NumPy, and Hyperopt for hyperparameter tuning) are installed using `pip`. The specific commands and environment setup are encapsulated in the `scripts/00_setup_env.sh` shell script. This script also initializes a Python virtual environment (`venv`) to ensure dependency isolation and project reproducibility. `bash # Example from 00_setup_env.sh pip install mlflow scikit-learn pandas numpy hyperopt`
- **Tracking Server:** A local MLflow tracking server is utilized. By default, when MLflow scripts are run, if a tracking URI is not explicitly set to a remote server, MLflow uses a local file system backend. For this project, experiments are configured to log to a local `./mlruns` directory relative to where the scripts are executed or if `mlflow.set_tracking_uri("file:./mlruns")` is used within the scripts (as implicitly done by running `mlflow ui` from the project root or by the scripts creating runs).
- **Backend Store:** The backend store, which houses experiment metadata (parameters, metrics, tags, notes, etc.), defaults to a local file store within the `./mlruns` directory. Each run's metadata is stored in its respective subdirectory. For more robust, concurrent setups, a database like SQLite (for local, single-user) or a more production-grade database (e.g., PostgreSQL, MySQL) would be configured.

The current setup uses the default file-based backend store suitable for local development.
- **Artifact Store:** The artifact store, which holds larger outputs like trained models, data files, images, and other arbitrary files, also defaults to a local directory structure within `./mlruns`. Each run has an `artifacts` subdirectory. For production or team-based environments, a remote storage solution like AWS S3, Azure Blob Storage, or Google Cloud Storage would typically be configured as the artifact store.
- **MLflow UI:** The MLflow Tracking UI is an integral part of the setup, allowing for visualization and comparison of experiments and runs, as well as management of registered models. It can be launched by running the command `mlflow ui` from the project's root directory. This command starts a local web server (defaulting to `http://localhost:5000`) that provides access to the logged data.

The scripts, such as `02_train_evaluate.py` and `03_hyperparameter_tuning.py`, explicitly set an experiment name (e.g., `Telco_Churn_Prediction`) using `mlflow.set_experiment()`. If the experiment does not exist, it is created. This organizes all related runs under a common experimental umbrella.

3. Tools Used

This project leverages a suite of Python libraries and tools standard in the machine learning and MLOps landscape:

- **Python (3.x):** The primary programming language used for scripting, data manipulation, model development, and interacting with MLflow.
- **MLflow (version 2.x or as installed by `00_setup_env.sh`):** The central MLOps platform used for:
  - **Tracking:** Logging experiments, parameters, metrics, and artifacts.
  - **Models:** Packaging, managing, and deploying machine learning models.
  - **Model Registry:** Versioning models and managing their lifecycle stages.
  - **Projects:** (Implicitly used) For packaging reusable ML code (though not explicitly creating a separate MLproject file in this iteration).
  - **Serving:** Deploying models locally for real-time predictions.
- **Scikit-learn:** A comprehensive library for machine learning in Python. Used for:
  - Data preprocessing (e.g., `StandardScaler`, `OneHotEncoder`, `ColumnTransformer`).
  - Model implementation (e.g., `LogisticRegression`, `RandomForestClassifier`, `GaussianNB`).
  - Model evaluation metrics (e.g., `accuracy_score`, `precision_score`, `recall_score`, `f1_score`, `roc_auc_score`).

    - Pipeline creation (`Pipeline`) for streamlining preprocessing and modeling steps.
- **Pandas:** A powerful library for data manipulation and analysis, used for loading, cleaning, and transforming the Telco Customer Churn dataset.
- **NumPy:** A fundamental package for numerical computation in Python, used for array manipulations and mathematical operations, often in conjunction with Pandas and Scikit-learn.
- **Hyperopt:** A Python library for distributed asynchronous hyperparameter optimization. Used in `scripts/03_hyperparameter_tuning.py` to find optimal hyperparameters for the RandomForestClassifier, with MLflow tracking each trial.
- **Shell Scripting (Bash):** Used in `scripts/00_setup_env.sh` for automating the setup of the Python virtual environment and installation of required dependencies.
- **JSON:** Used for payload format when interacting with the deployed model endpoint.
- **Requests:** Python library for making HTTP requests, used in `scripts/05_deploy_model.py` to send a test prediction request to the locally served MLflow model.
- **Git & GitHub (Implied):** For version control of the codebase and as a platform for hosting the code repository, as per the project deliverables (though not directly interacted with by the automated scripts).

These tools collectively provide a robust environment for developing, tracking, and deploying the customer churn prediction model while adhering to MLOps best practices.

4. Model Development Process

The model development process for predicting customer churn follows a structured approach, from data acquisition and preprocessing to model training and evaluation, with MLflow integrated at each critical step for tracking and reproducibility. This process is primarily implemented in the `scripts/02_train_evaluate.py` script.

Data Preprocessing

1. **Data Loading:** The Telco Customer Churn dataset is loaded from a CSV file. The `01_download_data.py` script is responsible for making this data available in the `../data/` directory. The `load_and_preprocess_data` function in `02_train_evaluate.py` handles the actual loading using Pandas.
2. **Initial Cleaning:**
    - The `TotalCharges` column, which might be read as an object type if it contains spaces or non-numeric characters, is converted to a numeric type.

Any rows where `TotalCharges` cannot be converted (resulting in `NaN`) are dropped.
    - The target variable `Churn` (originally 'Yes'/'No') is converted into a binary format (1 for 'Yes', 0 for 'No').
    - The `customerID` column, being a unique identifier, is dropped as it provides no predictive value for a general model.
3. **Feature Definition:** Features are categorized into numerical and categorical types:
    - **Numerical Features:** Columns with `int64` or `float64` data types (e.g., `tenure`, `MonthlyCharges`, `TotalCharges`) excluding the target variable.
    - **Categorical Features:** Columns with `object` data types (e.g., `gender`, `Contract`, `PaymentMethod`).
4. **Preprocessing Transformers:** A `ColumnTransformer` from Scikit-learn is defined to apply different transformations to different types of columns:
    - **Numerical Features:** `StandardScaler` is applied to scale these features to have zero mean and unit variance. This helps algorithms that are sensitive to feature magnitudes (e.g., Logistic Regression).
    - **Categorical Features:** `OneHotEncoder` is applied to convert categorical variables into a numerical format. The `handle_unknown='ignore'` parameter ensures that if new categories appear in test data (or future data), they don't cause errors and are encoded as all zeros for those new dummy variables.
5. **Data Splitting:** The preprocessed dataset (features `X` and target `y`) is split into training and testing sets (e.g., 80% training, 20% testing) using `train_test_split`. Stratification by the target variable `y` is used to ensure that the proportion of churners and non-churners is similar in both training and test sets.

Feature Engineering

For this iteration of the project, feature engineering is kept minimal to focus on the MLOps pipeline. The primary transformation is the encoding of categorical variables and scaling of numerical ones. More advanced feature engineering could include: * Creating interaction terms. * Binning numerical features (e.g., tenure into groups). * Extracting information from text-based features if any were present. * Creating ratios (e.g., `MonthlyCharges` to `TotalCharges` for customers with short tenures). These were not implemented but represent potential areas for future model improvement.

Model Selection

Several baseline models are developed and evaluated to establish performance benchmarks. The `02_train_evaluate.py` script implements and compares the following Scikit-learn models:

1. **Logistic Regression:** A linear model commonly used for binary classification tasks. It's interpretable and provides a good baseline. `solver='liblinear'` and `class_weight='balanced'` are used to handle potential class imbalance.
2. **Random Forest Classifier:** An ensemble learning method based on decision trees. It's generally more powerful than logistic regression and can capture non-linear relationships. `n_estimators=100` and `class_weight='balanced'` are used as initial parameters.
3. **Gaussian Naive Bayes:** A probabilistic classifier based on Bayes' theorem with an assumption of normality for features. It's simple and often works well, especially with high-dimensional data.

Each model is encapsulated within a Scikit-learn `Pipeline` that first applies the defined `ColumnTransformer` for preprocessing, followed by the model itself. This ensures that preprocessing steps are consistently applied during training, evaluation, and later during prediction.

Experiment Tracking with MLflow

MLflow is deeply integrated into the model training and evaluation loop (`train_evaluate_model` function in `02_train_evaluate.py`):

- **Experiment Setup:** An MLflow experiment named `Telco_Churn_Prediction` is created (if it doesn't exist) and set as the active experiment for logging.
- **Run Management:** For each model trained (e.g., Logistic Regression, Random Forest), a new MLflow run is initiated using `with mlflow.start_run(run_name=model_name) as run:`.
- **Logging Parameters:**
  - The model's hyperparameters are logged using `mlflow.log_params(model.get_params())`.
  - Custom parameters, like the `model_type`, are also logged using `mlflow.log_param()`.
- **Logging Metrics:** After training the model and making predictions on the test set, various performance metrics are calculated and logged:
  - Accuracy (`accuracy_score`)
  - Precision (`precision_score`)
  - Recall (`recall_score`)

    - F1-score (`f1_score`)
    - Area Under the ROC Curve (AUC) (`roc_auc_score`) These are logged using `mlflow.log_metric()` .
- **Logging Artifacts:**
    - The trained Scikit-learn pipeline (including the preprocessor and the model) is logged using `mlflow.sklearn.log_model(pipeline, model_name)` . This saves the model in MLflow's format, which includes the `MLmodel` descriptor file, the pickled model (`model.pkl`), and environment specifications (`conda.yaml` , `python_env.yaml` , `requirements.txt`). This makes the model easily loadable later for inference or further analysis.
    - (Optionally) Other artifacts like confusion matrices, ROC curve plots, or feature importance plots could also be generated and logged.

This systematic logging allows for detailed comparison of different models and training runs directly within the MLflow UI, facilitating model selection and providing a complete audit trail of the development process.

5. Hyperparameter Tuning

Hyperparameter tuning is a critical step in optimizing machine learning models to achieve the best possible performance. This project demonstrates hyperparameter tuning for the RandomForestClassifier using the Hyperopt library, with all tuning experiments meticulously tracked by MLflow. This process is implemented in the `scripts/03_hyperparameter_tuning.py` script.

Approach (Hyperopt with TPE)

- **Selected Model for Tuning:** The RandomForestClassifier was chosen for hyperparameter optimization due to its potential for high performance and its sensitivity to hyperparameter settings.
- **Tuning Library: Hyperopt:** Hyperopt is a Python library designed for optimizing over awkward search spaces, including real-valued, discrete, and conditional dimensions. It's particularly useful for hyperparameter optimization of machine learning models.
- **Algorithm: Tree-structured Parzen Estimator (TPE):** TPE is a sequential model-based optimization (SMBO) algorithm used by Hyperopt. Unlike grid search or random search, TPE intelligently chooses the next set of hyperparameters to evaluate based on past evaluation results, making the search process more efficient.

- **Search Space Definition:** A search space for the RandomForestClassifier's key hyperparameters is defined using Hyperopt's stochastic expression language (`hp`):
    - `n_estimators` : Number of trees in the forest (e.g., `hp.quniform(