Deploying models in production

Giorgio Alfredo Spedicato, PhD FCAS CSPA C.Stat[1]

Leitha SRL, Unipol Group

2024-11-01

Disclaimer

The views and opinions expressed in this presentation are those of the author and do not necessarily reflect the position of the organization of which he belongs.

Intro

- Deploying a model in production is a complex task that requires a deep understanding of the model, the data and the infrastructure.
- Actuaries were traditionally involved in the first two aspects, but the third one is becoming more and more important.
- Modern *Python frameworks* make the third aspect *easier*, at least when presenting a POC to the stakeholders.

MLOps: Enhancing the ML Model Lifecycle

- **Definition**: MLOps integrates practices to automate and manage the entire machine learning model lifecycle, improving deployment efficiency and reliability in production.
- **Why It's Needed**: Bridges the gap between development and production, enabling better collaboration between data scientists and engineers for models that are more performant, scalable, and maintainable.
- **Key Benefits**: Unified workflow reduces errors, accelerates deployment, and provides continuous monitoring to keep models up-to-date and accurate.

MLOps: instruments

- **Model training**: Python pipelines, requirements files, MLflow
- **Version control**: Git, GitHub, GitLab
- **Model deployment**: FastAPI, Streamlit, Docker

Project presentation

- **Objective**: Deploying an insurance quote model in production
- **Tools**:
  - **Python pipelines**: to train the frequency, severity and pure premium models
  - **Git**: to version control the code
  - **Docker**: to create a container with the models
  - **FastAPI**: to deploy the models
  - **Streamlit**: to create a user interface

Python pipelines

Structure

- Ingests the data, and clean
- Fits the models, saves them
- Assess the performance of the models, uses MLflow to log the results and artifacts

Running the pipeline

- see the `main.py` file
- the directory `steps` contains the steps of the pipeline
- It can be run as `python main.py`

Requirements

- Python packages specified in a `requirements.txt` file
- The `requirements.txt` file is used to create a virtual environment
- The Dockerfile uses the virtual environment to create a container

MLflow

- MLflow is an open source platform for managing the end-to-end machine learning lifecycle.
- It is used to log the results and artifacts of the models, that can be inspected in the MLFlow UI at `http://localhost:5000` after running the pipeline
- It is organized in experiments and runs

Streamlit walkthrough

Why streamlit?

- Streamlit is an open-source app framework for Machine Learning and Data Science projects.
- It creates a user interface for the models
- It is easy to use and to deploy

Streamlit code

- the `st.` functions are used to create the user interface
- the session state is used to track user's choices, e.g. button clicks
- the models are loaded, cached and used to make predictions

Running the app

- The app is run with `streamlit run quote-page.py`
- An initial section allows to input che policyholder's data (pre-defined values exists)
- A button click will send the data to the predict pipeline and show the results

FastAPI Walkthrough

Why FastAPI?

- FastAPI is a modern framework for building APIs, ideal for deploying machine learning models.
- It enables exposing models as a REST API, allowing other applications to use them, for example, via Python's `requests` library.
- It uses Python type hints to validate the input and output of the API, improving code readability and maintainability.

Structure of a FastAPI App

- The application is defined in a Python file, typically named `app.py`.
- The app runs with the ASGI server `uvicorn`, allowing FastAPI to handle asynchronous requests, which increases scalability and speed.
- The app structure includes a startup event for loading models, endpoints for handling requests, and dependencies to manage model usage.

Key Components of the Insurance Prediction API

1. Input Schema

- The input is structured using the `Insured` model, a class that describes the characteristics of the insured person, such as vehicle power (`VehPower`), vehicle age (`VehAge`), driver age (`DrivAge`), and other attributes relevant for calculating insurance premiums.
- Each field includes a title, description, and validity limits using `Field` from `pydantic`. This ensures input data consistency and completeness.

2. Output Schema

- The output is managed by the `PredictionResponse` class, defining the key variables predicted by the model: frequency (`Frequency`), severity (`Severity`), and pure premium (`Pure_Premium`).
- The `PredictionResponse` class ensures that the API always returns a standard format, simplifying integration with other applications.

4. Prediction Endpoint

- The primary endpoint of the application is `/predict/`, which receives the insured person's data and returns a prediction.
- Within the endpoint, the `Frequency` model and `Severity` model are used to calculate the pure premium:
  - **Frequency** (`Frequency`): predicts the number of claims.
  - **Severity** (`Severity`): predicts the average cost of claims.
  - **Pure Premium** (`Pure_Premium`): calculated as `Frequency * Severity`, representing the expected premium for the customer.
- The endpoint uses FastAPI's `Depends` to load the model dependencies and then uses `Predictor` to perform predictions.

FastAPI in Action

- The API can be run locally using the command `uvicorn app:app --reload`, enabling rapid development and testing.
- Once the application is online, you can interact with the prediction endpoint using tools like `curl` or directly from FastAPI's interactive documentation available at `/docs`.

Docker

Why Docker?

- Docker is a platform for developing, shipping, and running applications.
- It allows to create containers with the models and the dependencies
- It is used to deploy the models in production

Dockerfile

- The Dockerfile is used to create the container
- Classical steps are:
  - load a base python image
  - copy models and code
  - install requirements
  - Run the app

Docker commands

- `docker build -t myimage .` to build the image
- `docker run -d --name mycontainer -p 8000:8000 myimage` to run the container that exposes the app on port 8000
- `docker stop mycontainer` to stop the container

References I