Machine Learning

|Launch, Monitor, and Maintain Your System

Prof. Dr. Mostafa Elhosseini

https://youtube.com/@ProfElhosseiniTechVerse

Agenda

- Launch, Monitor, and Maintain Your System

Deployment

- Save the best model you trained,
- Transfer the file to your production environment, and
- Load it

Saving Your Model

- To save the model

```
import joblib

joblib.dump(final_model, "my_california_housing_model.pkl")
```

- pkl is a file extension that stands for "pickle“
- The pickle module in Python is used for serializing and deserializing Python object structures, meaning it can convert Python objects to a byte stream, and vice versa. This process is also known as pickling and unpickling.

Saving Your Model

- The **pickle module** in Python is used for **serializing** and deserializing Python object structures, meaning it can convert **Python objects** to a **byte stream**, and vice versa.
- This process is also known as **pickling** and unpickling.
  - To **save trained models to disk** after training and load them later when needed for predictions
  - To **save data preprocessing states**. For example, parameters of **scalers**, **encoders**, or any other transformation applied to training data can be saved and reused later to ensure that the same transformation is applied to new data during model deployment.

Tip

- It’s often a good idea to **save every model you experiment with** so that you can **come back easily** to any model you want.
- You may also **save the cross-validation** scores and perhaps the actual predictions on the validation set.
- This will allow you to easily compare scores across model types, and compare the types of errors they make

Deployment

- Once your model is transferred to production, you can **load** it and **use** it
- You must first **import** any custom classes and functions the model relies on

```
import joblib

# extra code – excluded for conciseness
from sklearn.cluster import KMeans
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.metrics.pairwise import rbf_kernel

def column_ratio(X):
    return X[:, [0]] / X[:, [1]]

#class ClusterSimilarity(BaseEstimator, TransformerMixin):
#    [...]

final_model_reloaded = joblib.load("my_california_housing_model.pkl")

new_data = housing.iloc[:5]  # pretend these are new districts
predictions = final_model_reloaded.predict(new_data)
```

Deployment

- The model will be used within a website
  - The user will **type in some data** about a new district and
  - **Click** the Estimate Price button.
  - This will **send a query** containing the data to the **web server**, which will forward it to your **web application**, and
  - Finally, your code will simply call the model's predict() method

Deployment

- you can wrap the model within a dedicated web service that your web application can query through a REST API.

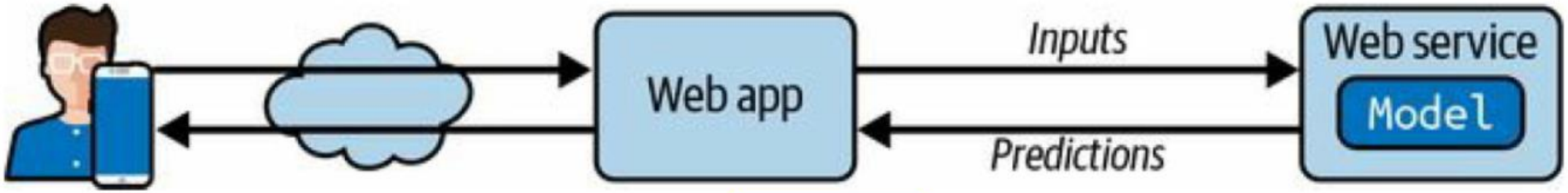

Figure 2-20. A model deployed as a web service and used by a web application

- A REST API (Representational State Transfer Application Programming Interface) is an architectural style and approach to communications often used in web services development. It enables different software systems to communicate with each other over the internet using standard **HTTP** methods.

Deployment

- you can wrap the model within a dedicated web service that your web application can query through a REST API.
  - This makes it easier to upgrade your model to new versions without interrupting the main application.
  - It also simplifies scaling, since you can start as many web services as needed and load-balance the requests coming from your web application across these web services.
  - Moreover, it allows your web application to use any programming language, not just Python

Deployment

- Another popular strategy is to **deploy your model to the cloud**, for example on **Google's Vertex AI** (formerly known as Google Cloud AI Platform and Google Cloud ML Engine):
- Just save your model using joblib and upload it to Google Cloud Storage (GCS), then head over to Vertex AI and create a new model version, pointing it to the GCS file. That's it!
- This gives you a simple web service that takes care of load balancing and scaling for you. It takes JSON requests containing the input data (e.g., of a district) and returns JSON responses containing the predictions.
- You can then use this web service in your website

Deployment

The typical process to set up a machine learning model as a web service involves the following steps:

- **Model Training**: First, you develop and train your machine learning model using your preferred ML framework (like TensorFlow, PyTorch, Scikit-Learn).
- **Model Serialization**: Once the model is trained and validated, you **serialize** the model using a format like pickle (for Python objects) or a more specialized format like ONNX (Open Neural Network Exchange), depending on the requirements and the framework used.

Deployment

The typical process to set up a machine learning model as a web service involves the following steps:

- **Developing the Web Service**: Create a web service using a framework suitable for handling HTTP requests. Popular choices for Python include Flask and FastAPI. This service will handle incoming HTTP requests, pass them to the model, and send back predictions.

Deployment

The typical process to set up a machine learning model as a web service involves the following steps:

- **API Definition**: Define RESTful endpoints that accept data in a structured format (e.g., JSON), process this data as needed (e.g., preprocessing, reshaping), pass it to the model for prediction, and return the prediction results.
- **Deployment**: Deploy the web service on a server or a cloud platform that can scale according to the demand. Common deployment targets include AWS (Amazon Web Services), GCP (Google Cloud Platform), Azure, or Heroku.

Monitoring

- You also need to **write monitoring code** to check your system's **live performance** at regular intervals and trigger alerts when it drops.
  - Drop very **quickly** - if a **component breaks** in your infrastructure,
  - Decay **very slowly** - go **unnoticed** for a long time - because of **model rot** - if the model was trained with last year's data, it may **not be adapted** to today's data

Monitoring

- Model's performance can be inferred from **downstream metrics**.
- if your model is **part of a recommender system** and it suggests products that the users may be interested in, then it's easy to **monitor** the number of recommended products **sold each day**. If this number drops (compared to non-recommended products), then the prime suspect is the model.
- This may be because the **data pipeline is broken**, or perhaps the model **needs to be retrained** on fresh data.

Monitoring

You may also need human analysis to assess the model's performance.

- For example, suppose you trained an image classification model to **detect various product defects** on a production line.
- How can you get an alert if the model's performance drops, before thousands of defective products get shipped to your clients?
- One solution is to **send to human raters** a sample of all the pictures that the model classified (especially pictures that the model **wasn't so sure about**).

Monitoring

- Depending on the task, the **raters** may need to be **experts**, or they could be **nonspecialists**, such as **workers** on a crowdsourcing platform (e.g., Amazon Mechanical Turk).
- In some applications they could even be the **users** themselves, responding, for example, via surveys or repurposed captchas

Monitoring

- You need to **put in place a monitoring system** (with or without human raters to evaluate the live model), as well as all the relevant processes to define **what to do in case of failures** and **how to prepare for them**. Unfortunately, this can be a lot of work.

Monitoring

- Regularly gather and label fresh data, potentially with human raters' help.
- Develop a script to automatically train the model and adjust hyperparameters. Schedule this to run at regular intervals (daily, weekly, etc.).
- Create a script to assess both the new and old models against the updated test dataset. Deploy the new model if performance remains stable or improves, investigating any declines.

Monitoring

- Regularly check the quality of model input data to detect early signs of issues, such as missing features or unexpected data value changes. Set up alerts for anomalies.
- Keep backups of all models and datasets to enable quick rollbacks or comparisons between different versions if necessary.