Applied Machine Learning for Business Analytics

Lecture 10: Get Machine Learning Models in Production

Lecturer: Zhao Rui

Logistics

- Appreciate if you keeps video on!

Agenda

1. From Notebooks to Python Scripts
2. Interfaces of ML Systems
3. MLOps
4. Building ML Pipelines with better tools

1. From Notebooks to Python Scripts

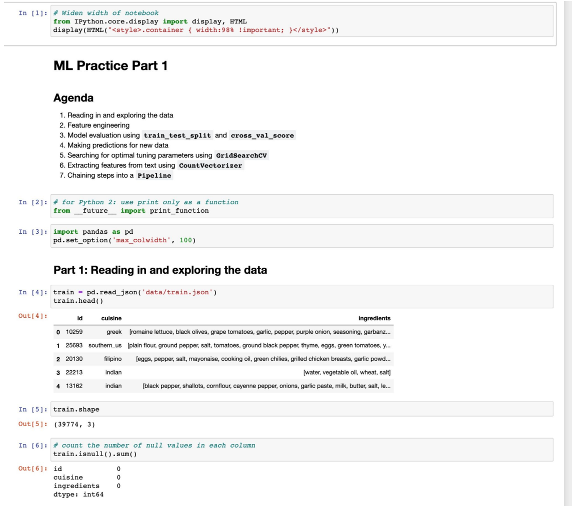
Virtual Environment

- Virtual Environment is required to isolate the packages necessary for applications from our other projects that may have different dependencies
- requirements.txt
  - Set up the development environment
  - pip freeze will dump all dependencies of all our packages into the file
  - Try pipreqs, pip-tools
- setup.py
  - Redistribute the whole packages
  - Contains metadata, requirements and entry points

https://stackoverflow.com/questions/43658870/requirements-txt-vs-setup-py

Organized Code

- Code should be **readable**, **reproducible**, **scalable** and **efficient**,
- Notebooks are only suitable for POC
- The code can be organized based on utility i.e., working pipeline components



In [1]:
```
# Widen width of notebook
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:98% !important; }</style>"))
```

ML Practice Part 1

Agenda

1. Reading in and exploring the data
2. Feature engineering
3. Model evaluation using `train_test_split` and `cross_val_score`
4. Making predictions for new data
5. Searching for optimal tuning parameters using `GridSearchCV`
6. Extracting features from text using `CountVectorizer`
7. Chaining steps into a `Pipeline`

In [2]:
```
# for Python 2: use print only as a function
from __future__ import print_function
```

In [3]:
```
import pandas as pd
pd.set_option('max_colwidth', 100)
```

Part 1: Reading in and exploring the data

In [4]:
```
train = pd.read_json('data/train.json')
train.head()
```

Out[4]:

	id	cuisine	ingredients
0	10259	greek	[romaine lettuce, black olives, grape tomatoes, garlic, pepper, purple onion, seasoning, garbanz...
1	25693	southern_us	[plain flour, ground pepper, salt, tomatoes, ground black pepper, thyme, eggs, green tomatoes, y...
2	20130	filipino	[eggs, pepper, salt, mayonaise, cooking oil, green chilies, grilled chicken breasts, garlic powd...
3	22213	indian	[water, vegetable oil, wheat, salt]
4	13162	indian	[black pepper, shallots, cornflour, cayenne pepper, onions, garlic paste, milk, butter, salt, le...

In [5]:
```
train.shape
```

Out[5]:
```
(39774, 3)
```

In [6]:
```
# count the number of null values in each column
train.isnull().sum()
```

Out[6]:
```
id             0
cuisine        0
ingredients    0
dtype: int64
```

Cookiecutter Data Science Template

- One of templates we can use is:
  - https://drivendata.github.io/cookiecutter-data-science/

Cookiecutter Data Science

A logical, reasonably standardized, but flexible project structure for doing and sharing data science work.

Cookiecutter Data Science Template

pip install cookiecutter
cookiecutter https://github.com/drivendata/cookiecutter-data-science
cd churn_model

```
(churn_model)  ruizhao@Ruis-MBP ?~ ?cookiecutter https://github.com/drivendata/cookiecutter-data-scienc
e
You've downloaded /Users/ruizhao/.cookiecutters/cookiecutter-data-science before. Is it okay to delete an
d re-download it? [yes]: yes
project_name [project_name]: churn_model
repo_name [churn_model]: churn_model
author_name [Your name (or your organization/company/team)]: BT5153
description [A short description of the project.]: ML Pipeline Demo
Select open_source_license:
1 - MIT
2 - BSD-3-Clause
3 - No license file
Choose from 1, 2, 3 [1]: 1
s3_bucket [[OPTIONAL] your-bucket-for-syncing-data (do not include 's3://')]:
aws_profile [default]:
Select python_interpreter:
1 - python3
2 - python
Choose from 1, 2 [1]: 1
```

Metadata

Cookiecutter Template

- The folder will be generated following the template
- Easier for us to understand and modify the code base

```
├── LICENSE
├── Makefile           <- Makefile with commands like `make data` or `make train`
├── README.md          <- The top-level README for developers using this project.
├── data
│   ├── external       <- Data from third party sources.
│   ├── interim        <- Intermediate data that has been transformed.
│   ├── processed      <- The final, canonical data sets for modeling.
│   └── raw            <- The original, immutable data dump.
│
├── docs               <- A default Sphinx project; see sphinx-doc.org for details
│
├── models             <- Trained and serialized models, model predictions, or model summaries
│
├── notebooks          <- Jupyter notebooks. Naming convention is a number (for ordering),
│                         the creator's initials, and a short `-` delimited description, e.g.
│                         `1.0-jqp-initial-data-exploration`.
│
├── references         <- Data dictionaries, manuals, and all other explanatory materials.
│
├── reports            <- Generated analysis as HTML, PDF, LaTeX, etc.
│   └── figures        <- Generated graphics and figures to be used in reporting
│
├── requirements.txt   <- The requirements file for reproducing the analysis environment, e.g.
│                         generated with `pip freeze > requirements.txt`
│
├── setup.py           <- Make this project pip installable with `pip install -e`
├── src                <- Source code for use in this project.
│   ├── __init__.py    <- Makes src a Python module
│   │
│   ├── data           <- Scripts to download or generate data
│   │   └── make_dataset.py
│   │
│   ├── features       <- Scripts to turn raw data into features for modeling
│   │   └── build_features.py
│   │
│   ├── models         <- Scripts to train models and then use trained models to make
│   │   │                 predictions
│   │   ├── predict_model.py
│   │   └── train_model.py
│   │
│   └── visualization  <- Scripts to create exploratory and results oriented visualizations
│       └── visualize.py
│
└── tox.ini            <- tox file with settings for running tox; see tox.readthedocs.io
```

Config

Config directory or file should be created for the following:

- Hyper-parameters for training
- Specifications for model locations, logging and other hand-coded information
- Running a small test for training

Config Template

params.yaml


```yaml
external_data_config:
  external_data_csv: data/external/train.csv

raw_data_config:
  raw_data_csv: data/raw/train.csv
  model_var: ['churn','number_vmail_messages','total_day_calls','total_eve_minutes','total_eve_charge','total_intl_minutes','number_customer_service_calls']
  train_test_split_ratio: 0.2
  target: churn
  random_state: 111
  new_train_data_csv: data/raw/train_new.csv

processed_data_config:
  train_data_csv: data/processed/churn_train.csv
  test_data_csv:  data/processed/churn_test.csv

mlflow_config:
  artifacts_dir: artifacts
  experiment_name: model_iteration1
  run_name: random_forest
  registered_model_name: random_forest_model
  remote_server_uri: http://localhost:1234

random_forest:
  max_depth: 30
  n_estimators: 42

model_dir: models/model.joblib

model_webapp_dir: webapp/model_webapp_dir/model.joblib
```

Logging is important for ML Sys

- Life is short. You need logs
- Do not rely too much on print statements
  - For example, print(‘aaaaaa’)
- Logging is the process of tracking and recording key events that occur in the applications
  - Inspect processes
  - Fix issues
  - More powerful than print statement

Logging 101

- Logger:
  - The main object that emits the log messages from the whole project
  - Can be specified to each module
- Handler:
  - Used for sending log records to a specific location and specifications for that location (name size, etc)
  - Different handlers have different rules to save logs in local files
- Formatter
  - Used for style and layout of the log records
- Levels (according to different priorities)
  - CRITICAL
  - ERROR
  - WARNING (Default setting for root logger)
  - INFO
  - DEBUG

Levels in Logs

```python
import logging
import sys

# Create super basic logger
logging.basicConfig(stream=sys.stdout, level=logging.INFO)

# Logging levels (from lowest to highest priority)
logging.debug("Used for debugging your code.")
logging.info("Informative messages from your code.")
logging.warning("Everything works but there is something to be aware of.")
logging.error("There's been a mistake with the process.")
logging.critical("There is something terribly wrong and process may terminate.")
```

PROBLEMS 3 OUTPUT TERMINAL DEBUG CONSOLE

```
(base) ruizhao@Ruis-MBP ~/Desktop python test.py
INFO:root:Informative messages from your code.
WARNING:root:Everything works but there is something to be aware of.
ERROR:root:There's been a mistake with the process.
CRITICAL:root:There is something terribly wrong and process may terminate.
(base) ruizhao@Ruis-MBP ~/Desktop
```

Best Practices in Logging

- Logger in each module
    - Examples:

```
| app.py
| package_a
|     module_a.py
```

```
# app.py
import logging
logging.basicConfig(format='%(asctime)s - %(name)s - %(levelname)s:%(message)s')
from package_a import module_a

logger = logging.getLogger(__name__)
logger.warning('from app')


# module_a.py
import logging

logger = logging.getLogger(__name__)
logger.warning('from module_a')
$ python app.py
2019-12-24 21:53:21,915 - package_a.module_a - WARNING:from module_a
2019-12-24 21:53:21,916 - __main__ - WARNING:from app
```

Best Practices in Logging

- Logger in each module
  - Easy to identify the error source
  - But at the same time: it is important to throw the pot

"甩锅" ("throw the pot/pass the buck")

"你背" ("let you carry the pot", i.e., "lay the blame on you")

"你的锅" ("it's your pot"/ "it's for you to get the blame")

Best Practices in Logging

- Log all the details that you want to generate from the inside
    - It could be useful during development and model running check
- Should log messages outside of small functions and inside larger workflow
    - Logger could be placed within main.py and train.py since the smaller functions defined in other scripts are used here

Logging Configuration

- Coding directly in scripts
- Using a config file
  - `logging.config.fileConfig()`
- Using the dictionary type
  - `logging.config.dictConfig()`
  - Can be put in config/config.py

Suitable for complex projects

Documenting Your Code

- Document our code is a way to organize our code
- What is more, make others and ourselves in the future to easily use the code base
- Most common documenting types:
  - Comments
  - Typing
  - Docstrings
  - Documentation

Comments

- Good code should not need comments because it is readable
- When do you need comments:

Ayush Goel, Learner, Worker

Answered Nov 21, 2013

Found this in the production code we use currently :

```
// This is black magic
// from
// *Some stackoverlow link
// Don't play with magic, it can BITE.
```

4.6K views · View 39 upvotes

Typing

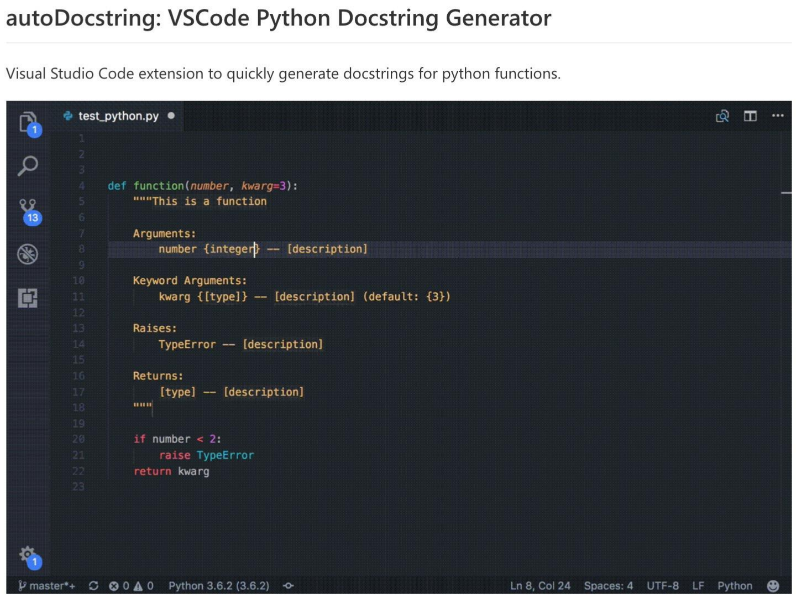
- Make our code as explicit as possible
  - Naming for variables and functions should be self-explaining
- Typing: Define the types for our function’s inputs and outputs

Starting from Python 3.9+, common types are **built in**

```python
from typing import List, Tuple, Dict
def add(a: int, string: str, f: float, b: bool) -> Tuple[List, Tuple, Dict, bool]:
    list1 = list(range(a))
    tup = (string, string, string)
    d = {"a": f}
    bl = b
    return list1, tup, d, bl
print(add(5, "hhhh", 2.3, False))
```

Docstrings

- Docstrings could be placed in functions and classes
- Use Python Docstrings Generator extension in VS Code



https://marketplace.visualstudio.com/items?itemName=njpwerner.autodocstring

Documents

- The above are all placed inside scripts. The documentation is a separated doc.
- Some open-source packages could be used to automatically generate the documentation
  - mkdocs (generates project documentation)
  - mkdocs-material (styling to beautiful render documentation)
  - mkdocstrings (fetch documentation automatically from docstrings)

Styling

- Code is read more often than it is written
- Follow consistent style and formatting conventions -> make code easy to read
- Most conventions are based on PEP8 conventions.
- We have lots of pipeline tools in place to automatically and effortlessly ensure that consistency

PEP 8 -- Style Guide for Python Code

PEP:	8
Title:	Style Guide for Python Code
Author:	Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Nick Coghlan <ncoghlan at gmail.com>
Status:	Active
Type:	Process
Created:	05-Jul-2001
Post-History:	05-Jul-2001, 01-Aug-2013

Styling Tools

- Those tools could be used with configurable options:
    - Black: an in-place reformatter that (mostly) adheres to PEP8.
    - isort: sorts and formats import statements inside Python scripts.
    - flake8: a code linter with stylistic conventions that adhere to PEP8.

```
# Black formatting
[tool.black]
line-length = 100
include = '\.pyi?$'
exclude = '''
/(
      \.eggs         # exclude a few common directories in the
    | \.git          # root of the project
    | \.hg
    | \.mypy_cache
    | \.tox
    | \.venv
    | _build
    | buck-out
    | build
    | dist
  )/
'''
```

Formatting done by Black

```python
# in:

def very_important_function(template: str, *variables, file: os.PathLike, engine: str, header: bool = True, debug: bool = False):
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, 'w') as f:
        ...

# out:

def very_important_function(
    template: str,
    *variables,
    file: os.PathLike,
    engine: str,
    header: bool = True,
    debug: bool = False,
):
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, "w") as f:
        ...
```

Makefile

- Makefile is an automation tool that organizes our commands
- Syntax:

```
# Makefile
target: prerequisites
<TAB> recipe
```

```
# Styling
.PHONY: style
style:
	black .
	flake8
	isort .
```

In the command Line, call “make style”

Makefile

- Different rules can be configured here

```
Makefile M ×
Makefile
PYTHON_INTERPRETER = python3

ifeq (,$(shell which conda))
HAS_CONDA=False
else
HAS_CONDA=True
endif

#################################################################################
# COMMANDS                                                                      #
#################################################################################

## Install Python Dependencies
requirements: test_environment
	$(PYTHON_INTERPRETER) -m pip install -U pip setuptools wheel
	$(PYTHON_INTERPRETER) -m pip install -r requirements.txt

## Code styling using flake8, black, isort
style:
	black src
	flake8 src
	isort src
## Make Dataset
data: requirements
	$(PYTHON_INTERPRETER) src/data/make_dataset.py data/raw data/processed

## Delete all compiled Python files
clean:
	find . -type f -name "*.py[co]" -delete
	find . -type d -name "__pycache__" -delete
```

```
(churn_model) ✘ ⚙ ruizhao@Ruis-MBP ~/churn_model main ± make
Available rules:

clean                  Delete all compiled Python files
create_environment     Set up python interpreter environment
data                   Make Dataset
requirements           Install Python Dependencies
style                  Code styling using flake8, black, isort
sync_data_from_s3      Download Data from S3
sync_data_to_s3        Upload Data to S3
test_environment       Test python environment is setup correctly
```

https://github.com/rz0718/churn_model/blob/main/Makefile

2.2 Interfaces of ML Systems

Batch prediction vs. online prediction

- Batch prediction:
  - Generate predictions periodically
  - Predictions are stored somewhere (e.g. SQL tables, CSV files)
  - Retrieve them as needed
  - Allow more complex models
- Online prediction:
  - Generate predictions as requests arrive
  - Predictions are returned as responses

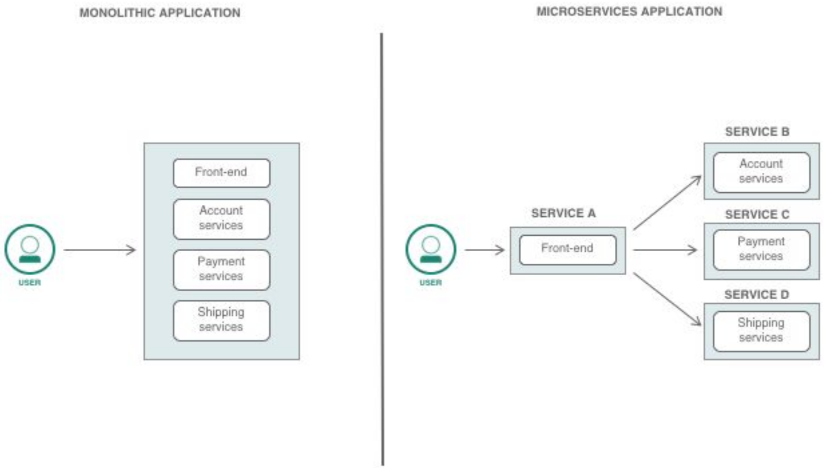
	Batch prediction	**Online prediction**
Frequency	Periodical	As soon as requests come
Useful for	Processing accumulated data when you don't need immediate results (e.g. recommendation systems)	When predictions are needed as soon as data sample is generated (e.g. fraud detection)
Optimized	High throughput	Low latency
Input space	Finite: need to know how many predictions to generate	Can be infinite
Examples	● TripAdvisor hotel ranking ● Netflix recommendations	● Google Translation ● Tiktok feed

Hybrid: batch & online prediction

- Online prediction is default, but common queries are precomputed and stored
- DOORDASH
  - Restaurant recommendations use batch predictions
  - Within each restaurant, item recommendations use online predictions
- NETFLIX
  - Title recommendations use batch predictions
  - Row orders use online predictions

Monolith vs Microservices

- Monolith: all application logics in one instance
- Microservices:
  - break application logics into smaller services
  - each service in its own container

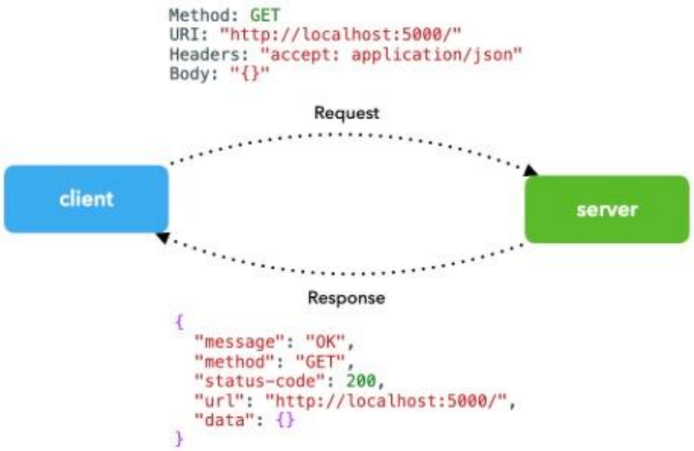


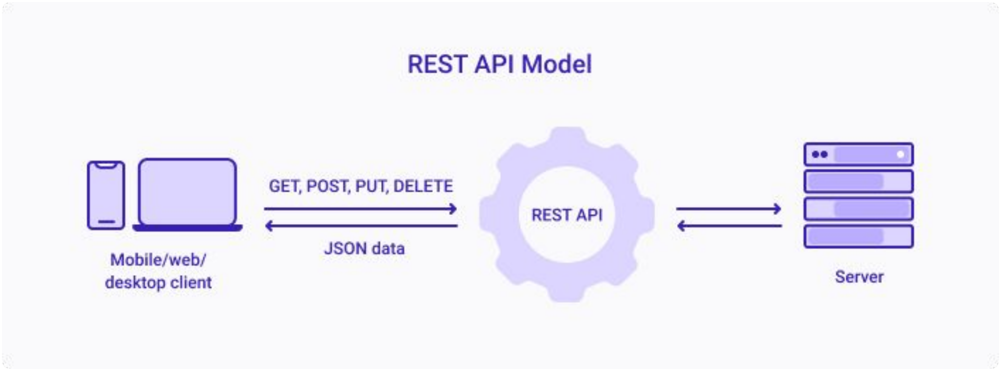
Microservices

- **Reduced complexity**: each developer works on a smaller codebase
- **Faster development cycle**: easier review process
- **Flexible stack**: different microservices can use different technology stacks

Deploy ML model in RestAPI

- Wrap ML Models in a Rest API
- Deploy them as a **Microservice**





Export a model

- Train models
- Turn models into binary formats
    - scikit-learn, XGBoost -> joblib, pickle
    - TensorFlow -> .save()
    - PyTorch -> .save()
    - Here, MLflow is recommended that is a board project focused on improving the lifecycle of machine learning projects.
- How to deploy the model?

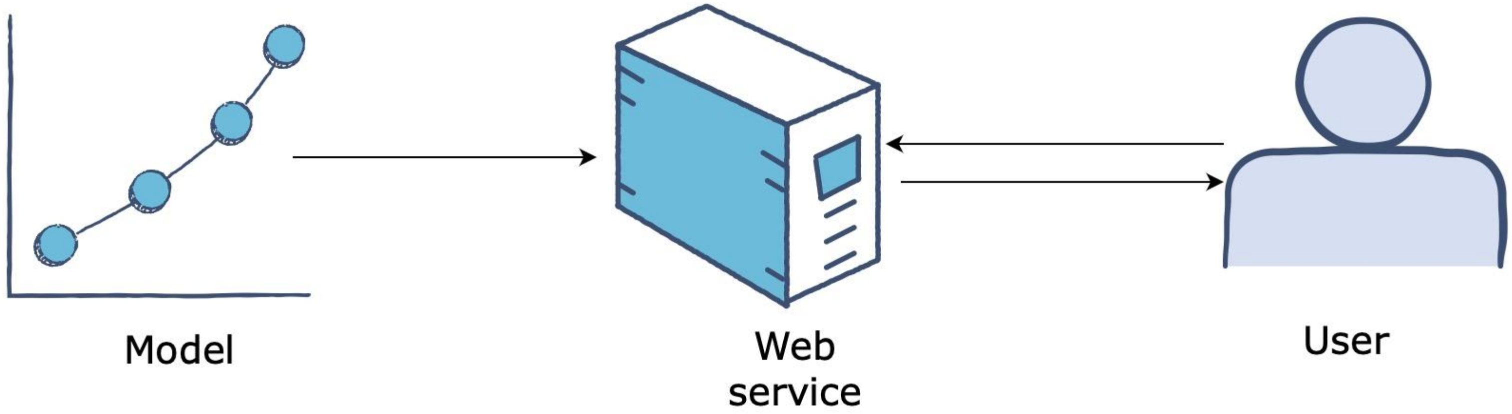
```
import mlflow.keras

model_path = "models/keras_games_v1"
mlflow.keras.save_model(model, model_path)

loaded = mlflow.keras.load_model(model_path, custom_objects={'auc': auc})
loaded.evaluate(x, y, verbose = 0)
```

Model as a web endpoint

- A model as an endpoint:
  - Prediction in response of a set of parameters
  - Here, parameters are feature vectors, images or model inputs
  - Other systems can easily use the predictive model which provides a real-time result



SKLearn Model Endpoint using Flask

```python
import flask

model_path = "models/logit_games_v1"
model  = mlflow.sklearn.load_model(model_path)

app = flask.Flask(__name__)

@app.route("/", methods=["GET","POST"])
def predict():
    data = {"success": False}
    params = flask.request.args

    if "G1" in params.keys():
        new_row = { "G1": params.get("G1"),"G2": params.get("G2"),
                    "G3": params.get("G3"),"G4": params.get("G4"),
                    "G5": params.get("G5"),"G6": params.get("G6"),
                    "G7": params.get("G7"),"G8": params.get("G8"),
                    "G9": params.get("G9"),"G10":params.get("G10")}

        new_x = pd.DataFrame.from_dict(new_row,
                              orient = "index").transpose()
        data["response"] = str(model.predict_proba(new_x)[0][1])
        data["success"] = True

    return flask.jsonify(data)

if __name__ == '__main__':
    app.run(host='0.0.0.0')
```

```python
import requests

new_row = { "G1": 0, "G2": 0, "G3": 0, "G4": 0, "G5": 0,
            "G6": 0, "G7": 0, "G8": 0, "G9": 0, "G10": 1 }

result = requests.get("http://52.90.199.190:5000/", params=new_row)
print(result.json()['response'])
```

Python Web Frameworks

- Flask
  - Suitable for quickly prototype
- Django
  - First choice to build robust full-stack websites
- FastAPI
  - Good at speed or scalability but quite new

A proper deployment also need a WSGI server that provides scaling, routing and load balancing.

Model as a serverless function

- Reduces the DevOps overhead of deploying models as web services.
- GCP Cloud Functions or AWS Lambda
- With serverless function environments,
  - Write a function that the runtime supports
  - Specify a list of dependencies
  - Deploy the function to production
  - The rest is fully managed by cloud platform such as provisioning servers, scaling up more machines to match demand, managing load balancers, and handling versioning.

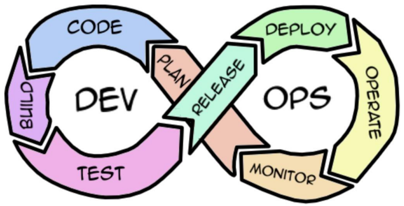
Rapidly moving from prototype to production for your prediction models

https://towardsdatascience.com/data-science-in-a-serverless-world-d04632e07a67

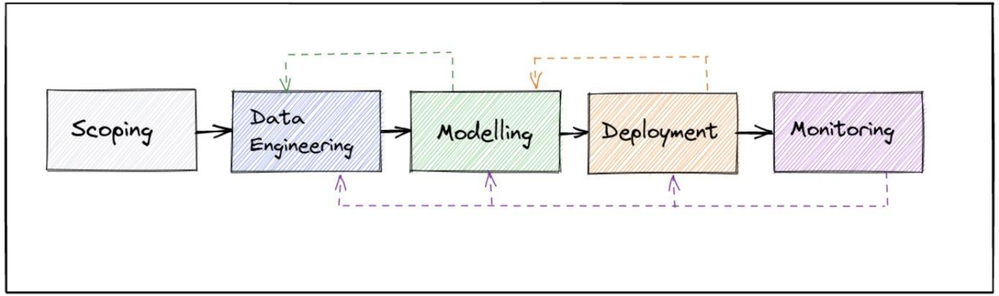
3. MLOps

MLOps=ML + DevOps

- MLOps:
  - A sequence of steps implemented to deploy an ML Model to the production environment
  - It is easy to create ML models that can predict based on the data you fed
  - It is challenging to create such models are are reliable, fast, accurate, and can be used by a large number of users



DevOps (Software Features)



ML Project Lifecycle

Image Credits

MLOps Concepts: I

- Development Platform
  - Enable smooth handover from ML Training to deployment
  - A collaboration platform for performing ML experiments
  - Enable secure access to data sources
- Versioning
  - Track the version of data and code
- Model Registry
  - An overview of deployed & legacy ML Models and their version history, and the deployment stage of each version
- Model Governance
  - Access control to training process related to any given models
  - Access control for who can request/reject/approve transitions between deployment stages (dev to staging to prod) in the model registry

MLOps Concepts: II

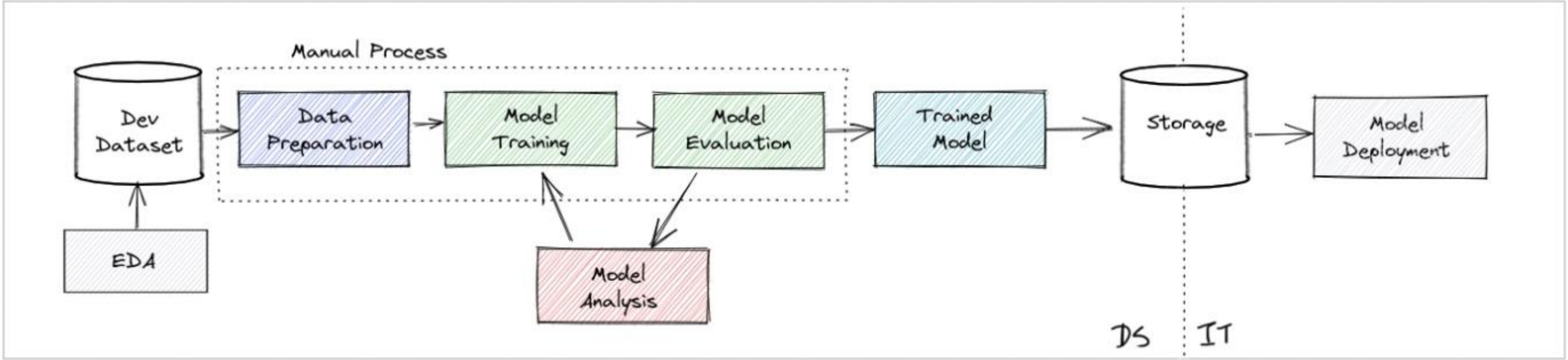
- Monitoring
  - Track performance metrics
    - ML metrics: F1 score, MSE, …
    - Ops metrics: uptime, throughput, response time
  - Drift detection
    - Concept drift: *when the relation between input and output has changed*
    - Label drift: *changes in predictions, but the model still holds*
    - Feature drift: *change in the model's outcomes compared to training data*
    - Prediction drift: *change in the distribution of model input data*
  - Outlier detection
    - If the new input is totally different from any training samples, we can identify this sample as potential outlier and the risk on the trustworthy of the model's prediction

MLOps Concepts: III

- Model Unit Testing: when we create, change or retrain a model, we should automatically validate the integrity of the model
    - Should meet minimum ml performance metrics on a test set
    - Should perform well on synthetic use case-specific datastest
- Devops Concepts:
    - CI/CD
    - Unit Test
    - Code Structure
    - Documentation

Manual MLOPs

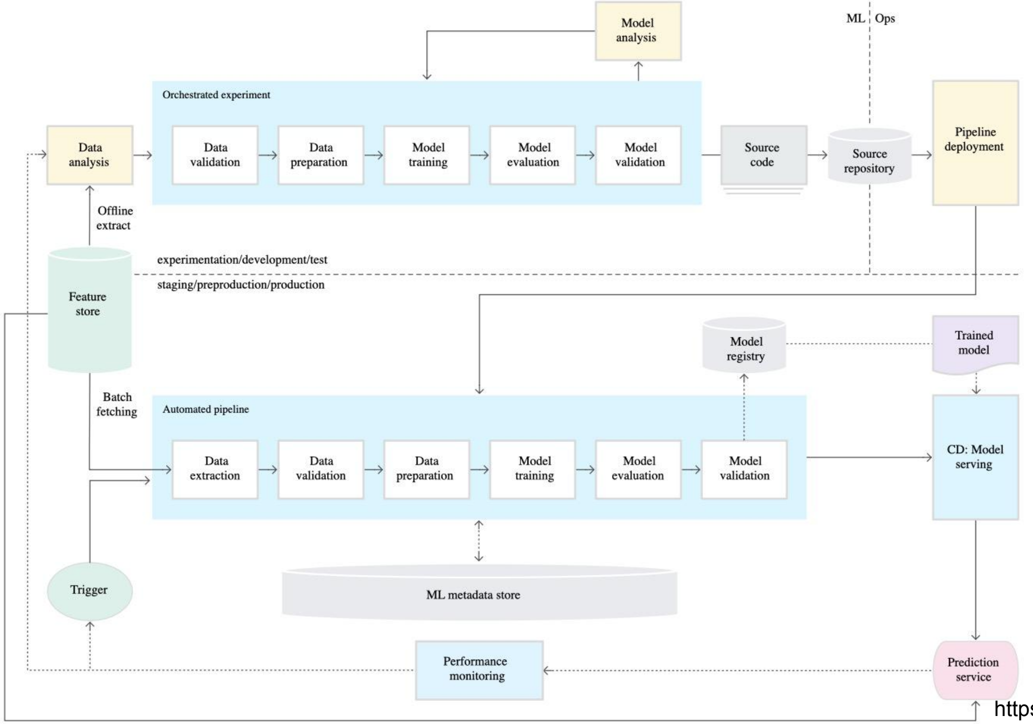
All the work are done manually



https://towardsdatascience.com/a-gentle-introduction-to-mlops-7d64a3e890ff

MLOPs

Automated Pipeline



https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning

4. Building ML Pipelines with better tools

ML Pipeline for Churn Prediction

- Machine learning models are trained to predict churn
  - Kaggle data: https://www.kaggle.com/c/customer-churn-prediction-2020
- Tools used for the ML pipeline
  - Flask: create API as interfaces of models
  - MLFlow: for model registry
  - Github: for code version control
  - Data Version Control (DVC): version control of the datasets and to make pipeline
  - Cookiecutter: Project templates

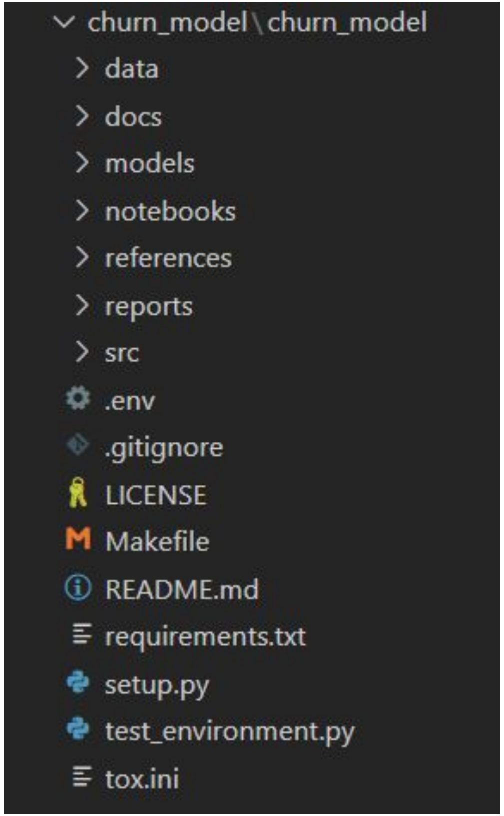
Git Repo: https://github.com/rz0718/churn_model

Create Virtual Environment

```
conda create -n churn_model
conda activate churn_model
```

Create Project Structure using the cookiecutter

```
pip install cookiecutter
cookiecutter https://github.com/drivendata/cookiecutter-data-science
cd churn_model
```



Create a Github repo

```
git init -b main
git add .
git commit -m "Init project"
git remote add origin <your_github_repo>
git branch -m main
git push -u origin main
```

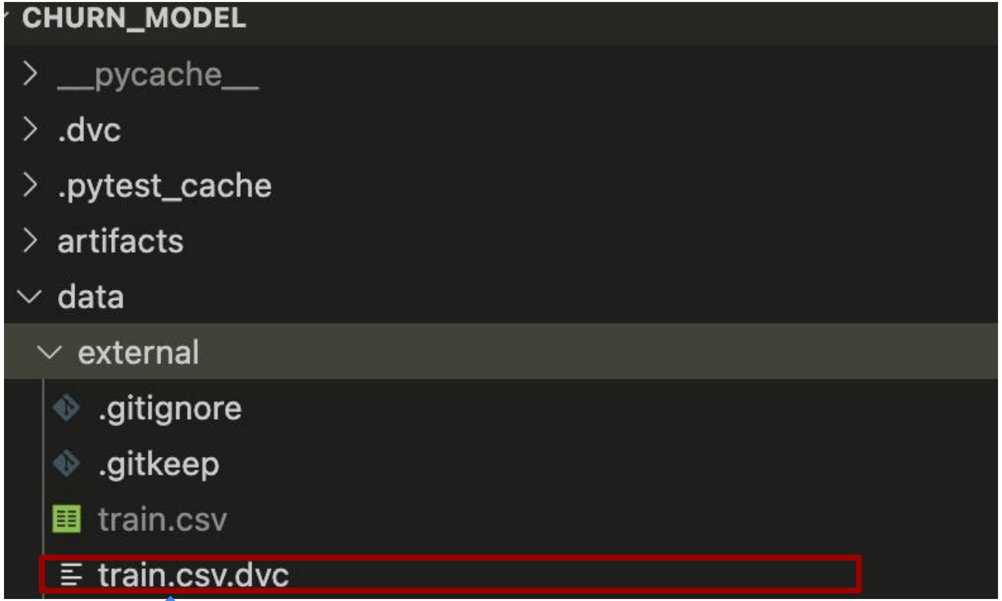
rz0718 add lib list		d23e7b6 3 hours ago 3 commits
.dvc	finish local deploy	3 hours ago
artifacts/1	finish local deploy	3 hours ago
data	finish local deploy	3 hours ago
docs	Init Repo with template	4 hours ago
models	finish local deploy	3 hours ago
notebooks	Init Repo with template	4 hours ago
references	Init Repo with template	4 hours ago
reports	Init Repo with template	4 hours ago
src	finish local deploy	3 hours ago
tests	finish local deploy	3 hours ago
webapp	finish local deploy	3 hours ago
.dvcignore	finish local deploy	3 hours ago
.gitignore	Init Repo with template	4 hours ago
LICENSE	Init Repo with template	4 hours ago
Makefile	Init Repo with template	4 hours ago
README.md	Init Repo with template	4 hours ago
app.py	finish local deploy	3 hours ago
dvc.lock	finish local deploy	3 hours ago
dvc.yaml	finish local deploy	3 hours ago
params.yaml	finish local deploy	3 hours ago
requirements.txt	add lib list	3 hours ago
setup.py	Init Repo with template	4 hours ago
test_environment.py	Init Repo with template	4 hours ago

Code Version Control

Track data version with DVC

pip install dvc
dvc init
dvc add <path_for_the_data_file>

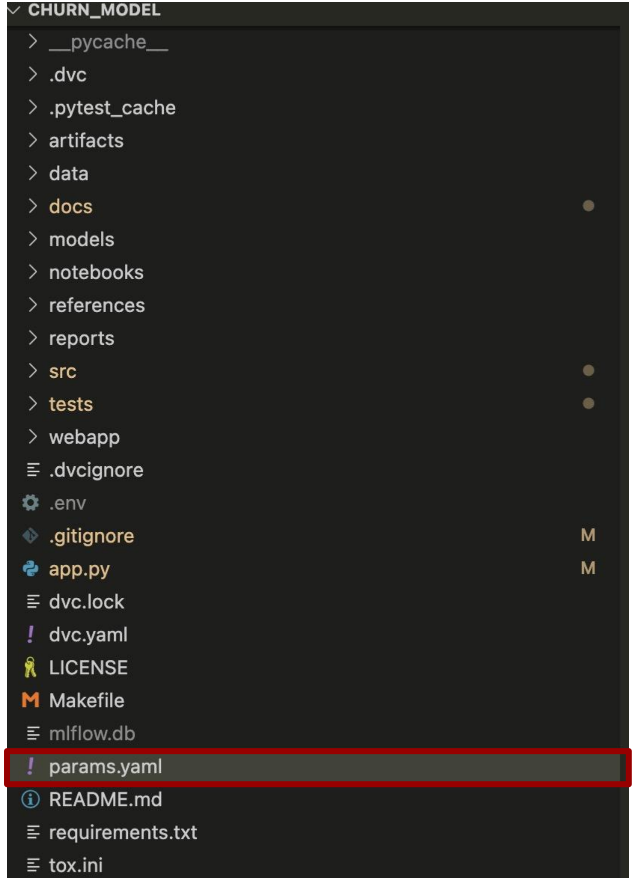
Data Version Control



Used by dvc to track train.csv

Write config file: params.yaml

- Store all the configurations related to this project
- Put the yaml file under main folder

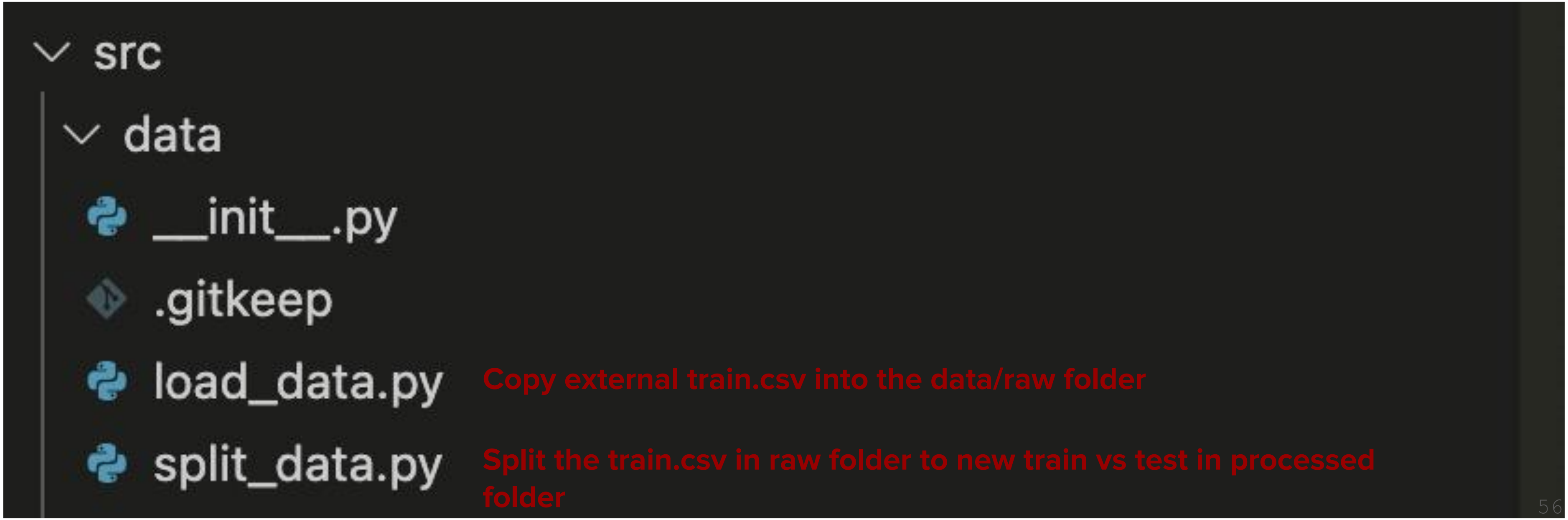


Prepare Source Code Inside the Src Folder

- Add data loading related scripts into the folder of data
- Add modeling related scripts into the folder of models

```
├── src                <- Source code for use in this project.
│   ├── __init__.py    <- Makes src a Python module
│   │
│   ├── data           <- Scripts to download or generate data
│   │   └── make_dataset.py
│   │
│   ├── features       <- Scripts to turn raw data into features for modeling
│   │   └── build_features.py
│   │
│   ├── models         <- Scripts to train models and then use trained models to make
│   │   │                 predictions
│   │   ├── predict_model.py
│   │   └── train_model.py
│   │
```

Data Folder

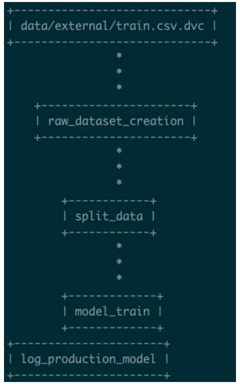

src
data
__init__.py
.gitkeep
load_data.py
Copy external train.csv into the data/raw folder
split_data.py
Split the train.csv in raw folder to new train vs test in processed folder

Model Folder

- MLflow is used to track the model performances
- model_selection.py is used to select the best model from model registry and save the best model in the root/model directory

Pipeline Creation with DVC

- With all scripts in src folder, create the dvc.yaml to define the pipeline
- Each stage in yaml files contains:
  - cmd: bash command to execute the script
  - deps: the dependencies to execute the step
  - outs: output from the cmd line (model or data)
  - params: parameters used in the script
- With deps, we can create DAG
  - Call “dvc dag”




```yaml
stages:
  raw_dataset_creation:
    cmd: python src/data/load_data.py --config=params.yaml
    deps:
    - src/data/load_data.py
    - data/external/train.csv
    outs:
    - data/raw/train.csv

  split_data:
    cmd: python src/data/split_data.py --config=params.yaml
    deps:
    - src/data/split_data.py
    - data/raw/train.csv
    outs:
    - data/processed/churn_train.csv
    - data/processed/churn_test.csv

  model_train:
    cmd: python src/models/train_model.py --config=params.yaml
    deps:
    - data/processed/churn_train.csv
    - data/processed/churn_test.csv
    - src/models/train_model.py
    params:
    - random_forest.max_depth
    - random_forest.n_estimators

  log_production_model:
    cmd: python src/models/model_selection.py --config=params.yaml
    deps:
    - src/models/model_selection.py
    params:
    - random_forest.max_depth
    - random_forest.n_estimators
    outs:
    - models/model.joblib
```

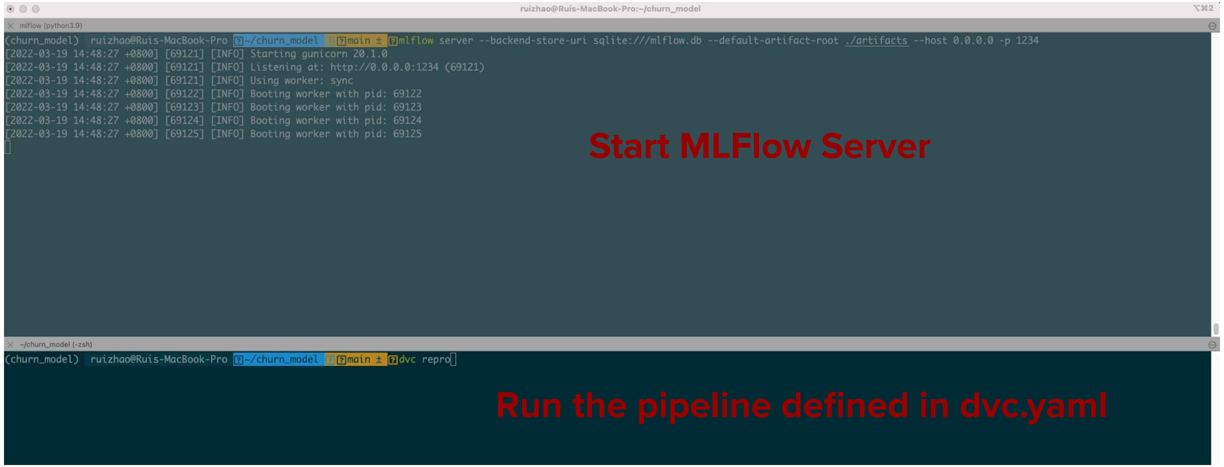
https://github.com/rz0718/churn_model/blob/main/dvc.yaml

Execute the pipeline

- Use two terminals to execute:

mlflow server --backend-store-uri sqlite:///mlflow.db --default-artifact-root ./artifacts --host 0.0.0.0 -p 1234

dvc repro

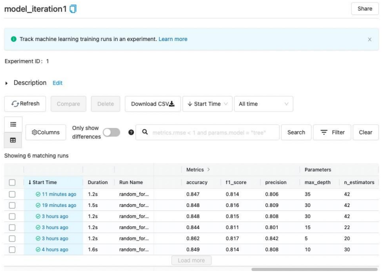


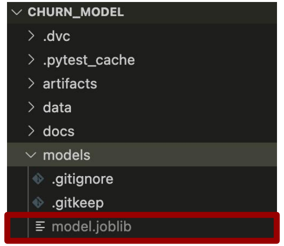
Why DVC

- DVC only conduct the action if dependencies are changed
- For example, run dvc repro again

```
(churn_model) ruizhao@Ruis-MacBook-Pro ?~/churn_model ??main ± ?dvc repro
'data/external/train.csv.dvc' didn't change, skipping
Stage 'raw_dataset_creation' didn't change, skipping
Stage 'split_data' didn't change, skipping
Stage 'model_train' didn't change, skipping
Stage 'log_production_model' didn't change, skipping
Data and pipelines are up to date.
```

- Change the hyper-parameters in params.yaml, the last two stages will be executed. We can use mlflow dashboard to track

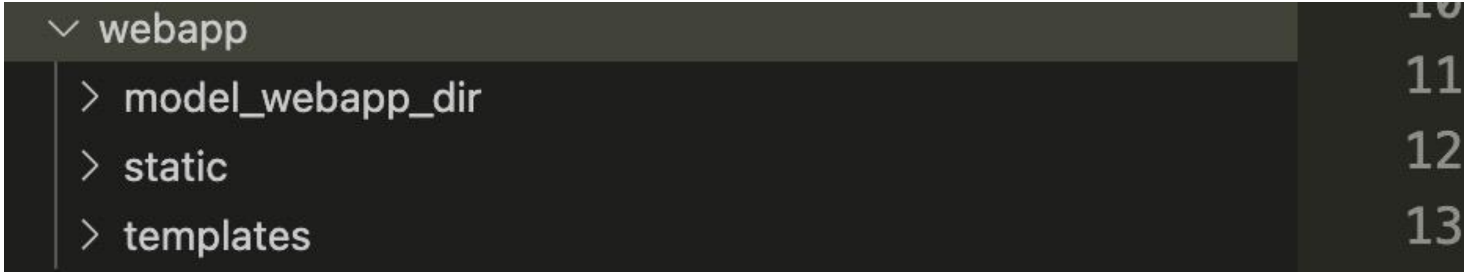




Update the model file

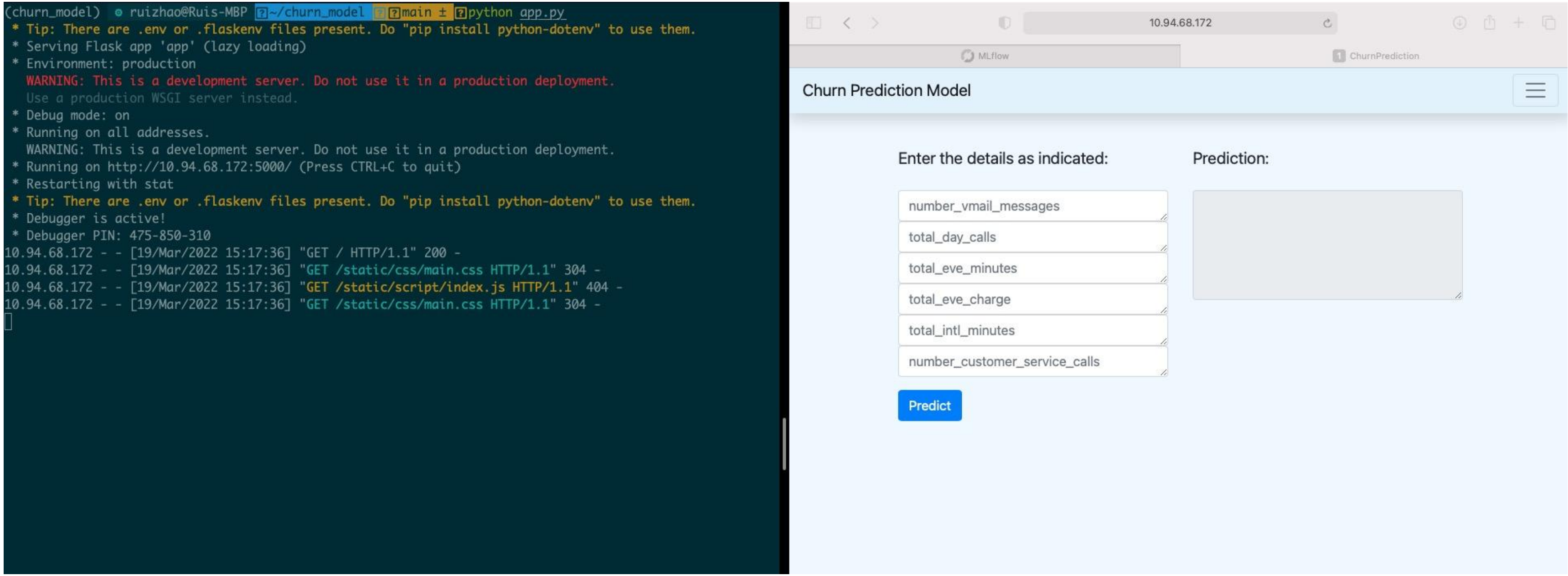
Build a web app using Flask

- Web App is able to:
  - Allow users to enter feature values
  - Call the backend ml model to take the inputs and predict the outcome
- Create a folder: webapp
  - Have required HTML, CSS and JS codes (front-end)
  - The model file (joblib) is required to be put in the model_webapp_dir



Create app.py

- Create app.py under the main folder
  - Connect backend to frontend
  - Send the responses to the UI after predicting the label



What we are missing

- Unit/Load tests
- Deploy the application in a real environment (not local env.)
- CI/CD
  - Push the change to git repo
  - It can be immediately deployed in production after passing the test
  - The answers from industries at this moment are:
    - Containers
    - Kubernetes
- Model Monitoring