Applied Machine Learning for Business Analytics

Lecture 10: Get Machine Learning Models in Production

Lecturer: Zhao Rui

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 DS 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 DS template

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

```
(bt5153env) rz@RuisPeralMacPro lab_lecture10 % cookiecutter https://github.com/drivendata/cookiecutter-data-science
project_name [project_name]: cuisine_tag
repo_name [cuisine_tag]: cuisine_tag
author_name [Your name (or your organization/company/team)]: rz_msba
description [A short description of the project.]: lecture10_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 structure frame 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

Avoid hard coding

Config template

main spam_detection / params.yaml

rz0718 init folder

1 contributor

35 lines (27 sloc) | 810 Bytes

```yaml
external_data_config:
  external_data_csv: data/external/sms.tsv

raw_data_config:
  raw_data_csv: data/raw/train.csv
  model_var: ['label', 'message']
  train_test_split_ratio: 0.2
  target: label
  text: message
  random_state: 111
  new_train_data_csv: data/raw/train_new.csv
  label_encoding: {'ham':0, 'spam':1}

processed_data_config:
  train_data_csv: data/processed/spam_train.csv
  test_data_csv:  data/processed/spam_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: 35
  n_estimators: 42

count_vectorizer:
  max_features: 5000

model_dir: models/model.joblib

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

https://circleci.com/blog/what-is-yaml-a-beginner-s-guide/

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 > 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**

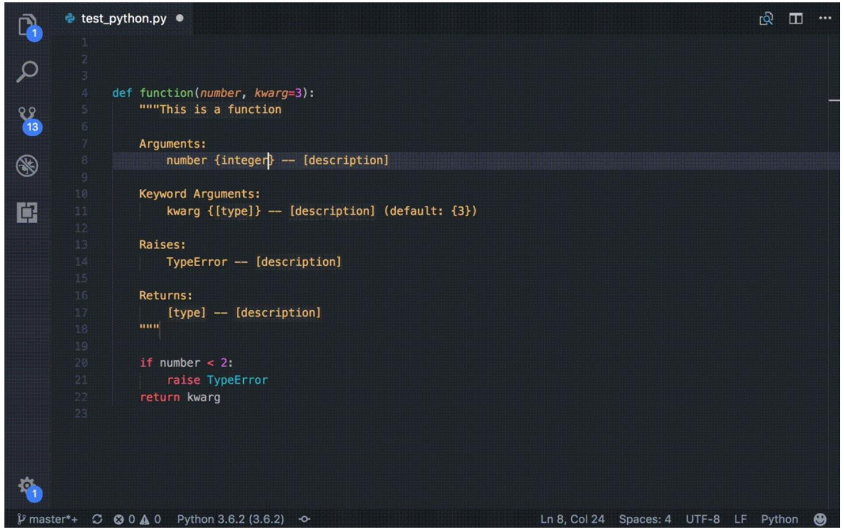
```python
from typing import List, Tuple, Dict
def add(a: int, string: str, f: float, b: bool) -> Tuple[List, Tuple, Dict, bool]:
    list1 = list(range(a))
    tup = (string, string, string)
    d = {"a": f}
    bl = b
    return list1, tup, d, bl
print(add(5, "hhhh", 2.3, False))
```

Docstrings

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

autoDocstring: VSCode Python Docstring Generator

Visual Studio Code extension to quickly generate docstrings for python functions.



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 in Makefile
    - Example here

```
(bt5153env) rz@RuisPeralMacPro spam_detection % make
Available rules:

clean                  Delete all compiled Python files
create_environment     Set up python interpreter environment
data                   Make Dataset
lint                   Lint using flake8
requirements           Install Python Dependencies
test_environment       Test python environment is setup correctly
(bt5153env) rz@RuisPeralMacPro spam_detection % make clean
find . -type f -name "*.py[co]" -delete
find . -type d -name "__pycache__" -delete
```

2 Interfaces of ML Systems

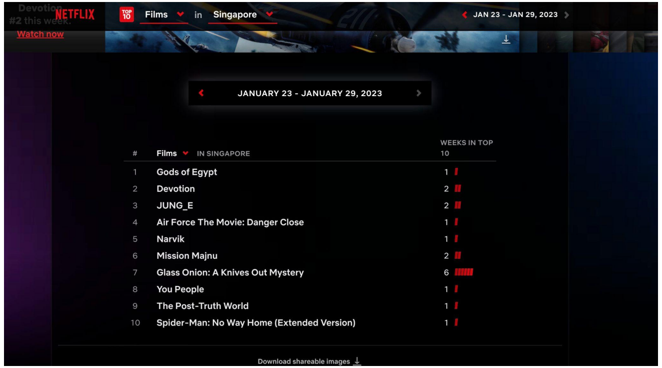
How to deploy ML models

- **Batch Deployment**
  - Generate Predictions at defined frequencies
- **Real-time Deployment**
  - Generate predictions as requests arrive
- Streaming Deployment
  - Generate predictions when specific events trigger
- Edge Deployment
  - Generate predictions on users' side

They are also called as **online prediction**

Batch deployment

- Frequency: Periodical
- Processing accumulated data when you do not need immediate results
  - Predictions can be pre-computed and stored in a database. Then, can be easily retrieved when needed
  - However, predictions can be quickly outdated if we can not use recent data.
- Applications:
  - TripAdvisor hotel ranking
  - Netflix recommendation

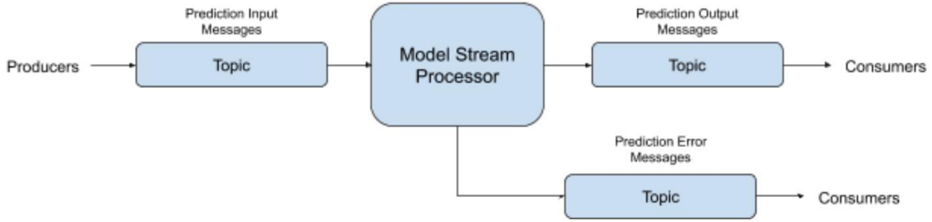


Real-time deployment

- Frequency: as soon as requests come
  - A synchronous process when a user/customer requests a prediction
- The process starts with users' requests
  - Users' requests is pushed to a backend service (usually through HTTP API calls)
  - Then, it is pushed it to a ML service
  - ML service would either take features from the request or collect recent contextual information to return predictions
- Multi-threaded processes and vertical scaling by additional servers could handle latency and concurrency issues.
  - Multiple users raise additional parallel requests
- Applications:
  - Google translation

Streaming deployment

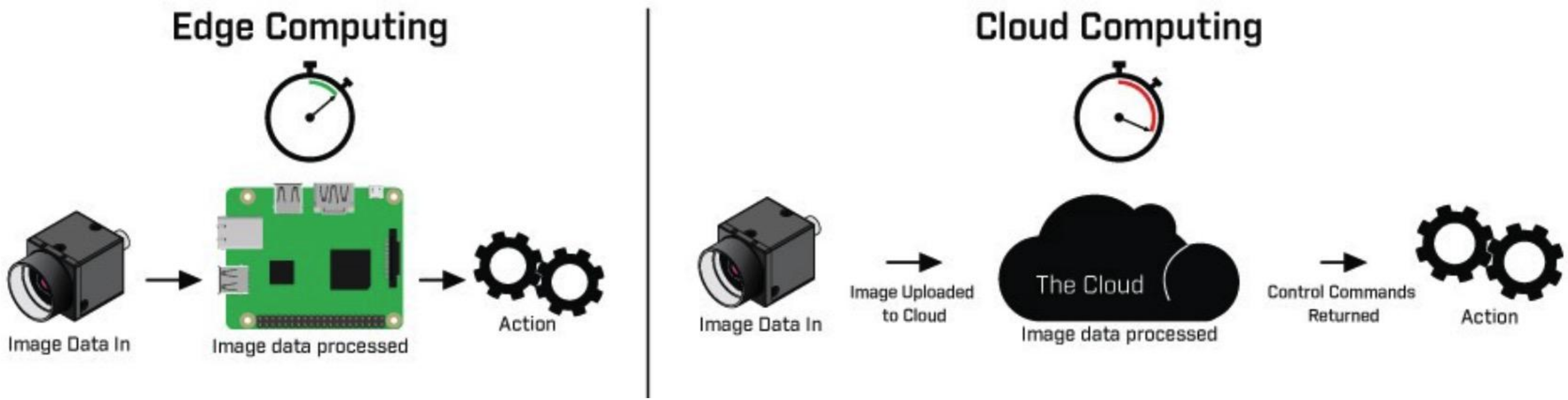
- Frequency: based on events
    - A more synchronous process compared to real-time deployment
- Events can trigger the start of prediction process
    - Users' requests is pushed to a backend service (usually through HTTP API calls)
    - For example, you are at tiktok page, the recommendation process would be triggered. And by the time your scroll, the recommendation results will be ready to be refreshed
    - Massage brokers like Kafka are always used as the queueing process
- Applications:
    - Facebooks Ads
    - Tiktok recommendation



Source: https://www.tekhnoal.com/streaming-ml-model-deployment.html

Edge deployment

- Model is directly deployed on the client side
  - Web browser, Mobile phone, Car, IoT hardwares
  - Can be fastest and offline predictions (without internet)
  - Models’ complexity are limited due to the smaller hardware



Source: https://www.kdnuggets.com/2018/09/deep-learning-edge.html

Batch vs Online deployment

Batch deployment

- Pro:
  - The most simple deployment approach
- Cons:
  - It is not efficient since most predictions might not be used at the end
  - It can not react to data changes

Real-time deployment

- Pro:
  - The model takes in account near real-time data and make fresh predictions
- Cons:
  - Has some steep learning curve

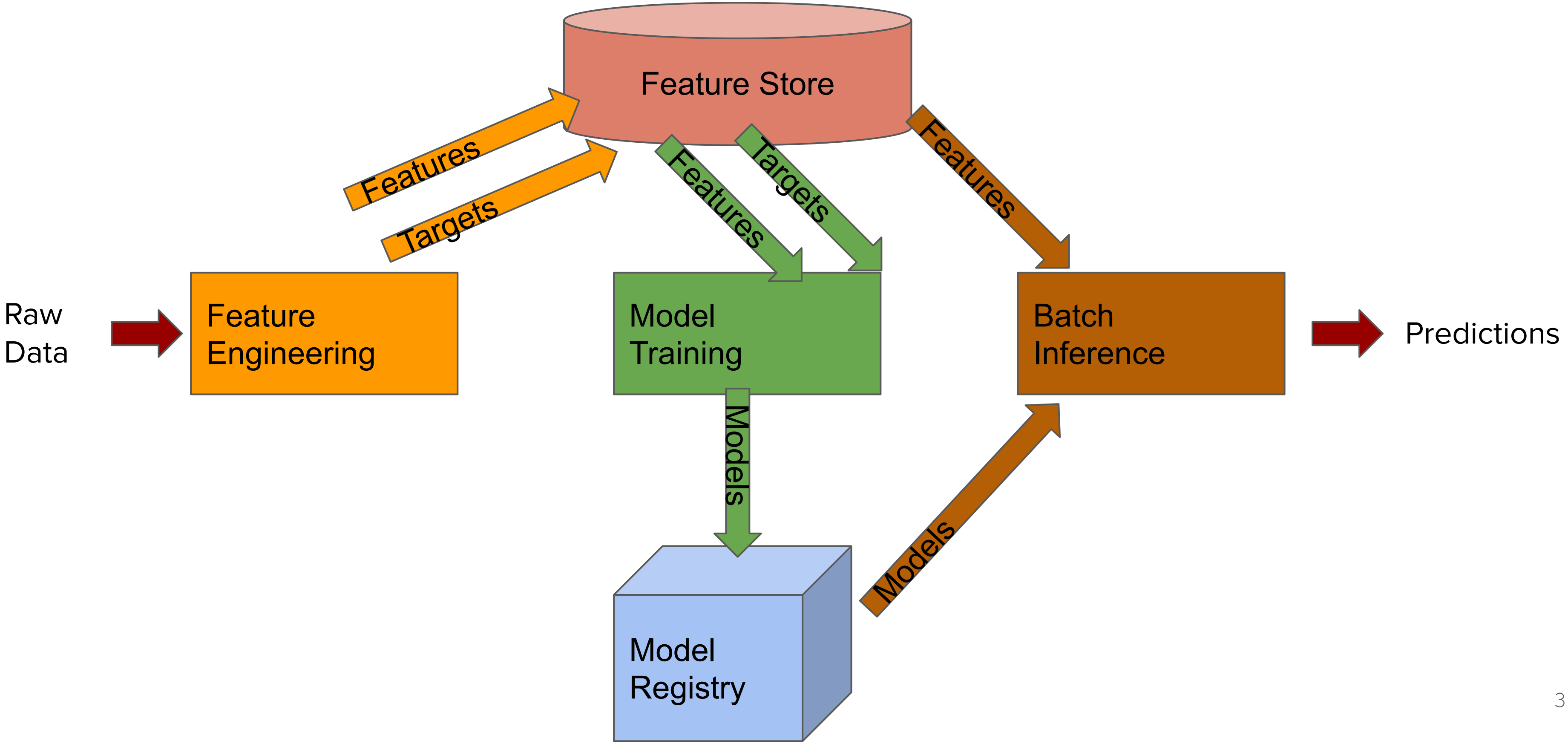
Hybrid: batch & real-time prediction

- Real-time prediction is default, but common queries are precomputed and stored
- Food delivery services
  - Restaurant recommendations use batch predictions
  - Within each restaurant, item recommendations use online predictions
- Streaming services
  - Title recommendations use batch predictions
  - Row orders use online predictions

Batch deployment

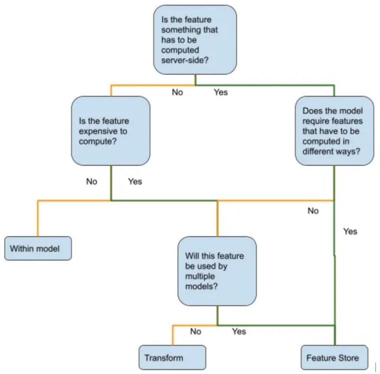
- A batch deployment usually work as on a fixed schedule (every 9:30 am), raw data are processed, and then model predictions are generated
- 3 pipeline architecture is usually used:
  - Feature pipeline
  - Training pipeline
  - Batch prediction pipeline

Batch deployment



Batch deployment: feature engineering

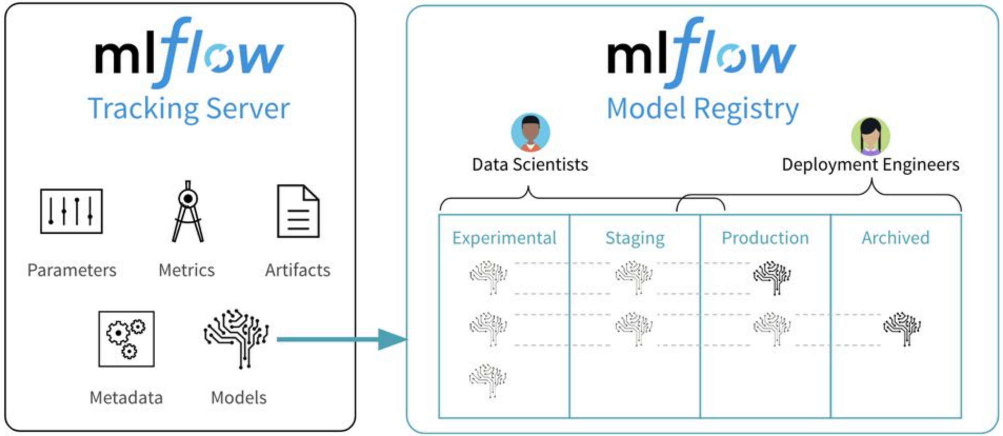
- Read raw data and generates features and labels
- Two engineering change would be applied:
  - Automation: feature pipeline to be executed in a fixed interval
    - Cron job
    - Airflow
    - GitHub action
  - Persistence: a place to store features generated by the script (instead of csv files on disk).
    - Feast
    - Other feature store tools



Source:
https://towardsdatascience.com/do-you-really-need-a-feature-store-e59e3cc666d3

Batch deployment: model training

- Read raw data and generates features and labels
- Turn models into binary formats
  - scikit-learn, XGBoost -> joblib, pickle
  - TensorFlow -> .save()
  - PyTorch -> .save()
  - We can save the trained model in the **model registry** (such as mlflow)



Batch deployment: batch inference

- Create a new script to do the following things:
  - Loads the production model from the model registry
  - Loading the most recent feature batch
  - Make model predictions and save them in databases
- The above script should also be scheduled

Deploy ML model in RestAPI

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

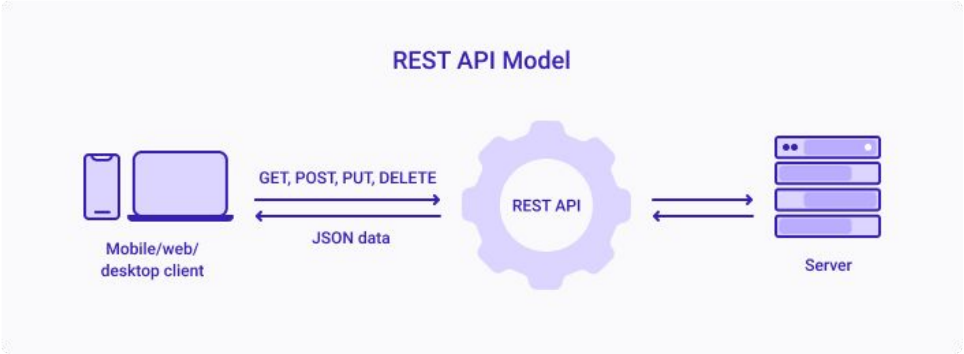
Model Preparation

1. Data collection & clean
2. Feature & model selection
3. Model training

Serialization & De-serialization

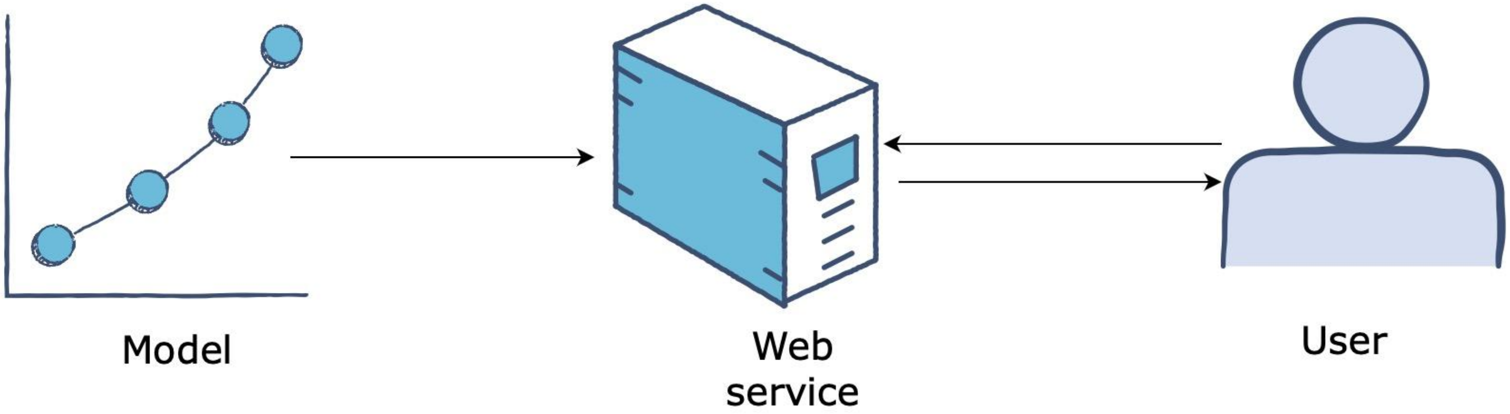
1. Save model from memory to disk
2. Load model from disk to memory

Web Service



Model as a web endpoint

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



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.

Build a web app using Flask

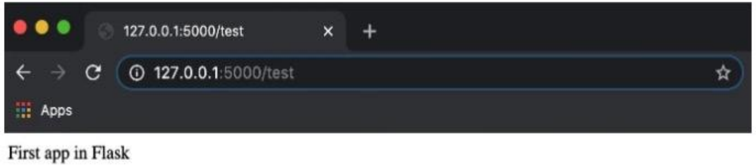
- Flask: a lightweight web framework for Python
  - Create an API call which can be used from front-end
  - Build a full-on web application

```python
from flask import Flask

# we define a variable called app
app = Flask(__name__)

# tells Flask what URL a user has to browse to call the function below.
# you will need to browse the url : '/ml-model'
@app.route("/test")
def run_model():
     #run model
     result = "First app in Flask"
     return result
if __name__ == '__main__':
     app.run(host='0.0.0.0', port=5000)
```

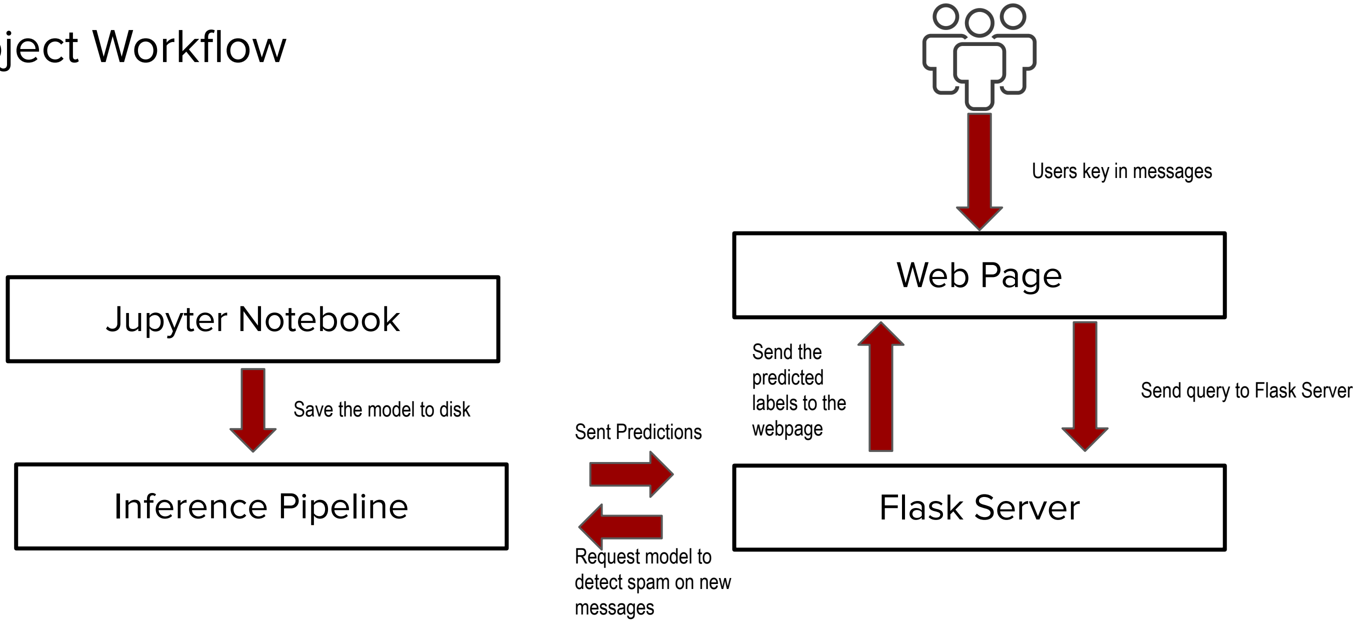
Code snippet



Web Server

Build a spam detection web app

- Spam detection from notebook needs to be deployed in order to be used by our end-users
- Project Workflow



Project folder

- Create a project folder:
  - Have required HTML, CSS and JS codes ([front-end](#))
  - The model file (joblib) is required to be put in the model_webapp_dir

```
webapp
├ model_webapp_dir
│ └ model.joblib
├ static
│ ├ css
│ │ └ main.css
│ └ sctipt
│ │ └ index.js
└ templates
│ ├ 404.html
│ ├ base.html
│ └ index.html
app.py
```

Tree is generated via
https://marketplace.visualstudio.com/items?itemName=Shinotatwu-DS.file-tree-generator

Frontend design

- Created index.html for web page design
  - Collect text from users
  - Display predictions whether it is spam or ham.

```html
<div class="container-fluid masthead">
    <br>
    <br>
    <br>
    <br>
    <div class="container">
        <div class="row">
            <div class="col">
                <form method="POST">
                    <!-- Input block-->
                    <div class="form-group">
                        <label for="translation">
                            <blockquote class="blockquote">
                                <p class="mb-2">Enter the text:</p>
                            </blockquote>
                        </label>
                        <textarea class="form-control" name="message" rows="1"
                            placeholder="message"></textarea>
                    </div>

                    <!-- Select output language here. -->
                    <div class="form-group">
                    </div>
                    <button type="submit" class="btn btn-primary mb-2">Predict</button></br>
                    </br>
                </form>
                <!-- Translated text returned by the Translate API is rendered here. -->
            </div>
            <div class="col">
                <form>
                    <div class="form-group">
                        <label for="translation-result">

                            <blockquote class="blockquote">
                                <p class="mb-2">Prediction:</p>
                            </blockquote>

                        </label>
                        <textarea readonly class="form-control" id="exTextarea" rows="5">{{ response }}</textarea>
                    </div>
                </form>
            </div>
        </div>
    </div>
```

Code Snippet

Web UI

Create app.py

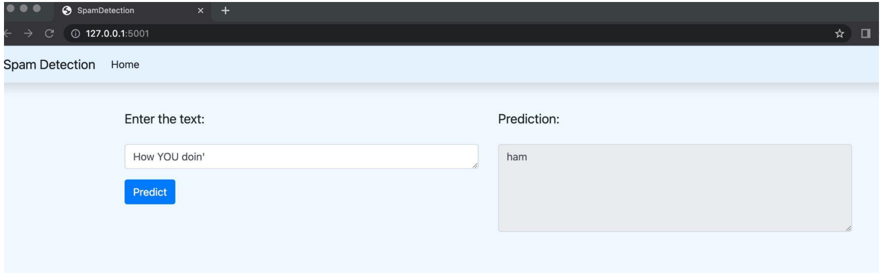
- Create app.py under the main folder
  - Connect backend to frontend
  - Send the responses to the UI after predicting the label

```python
@app.route("/", methods=["GET", "POST"])
def index():
    if request.method == "POST":
        try:
            if request.form:
                dict_req = dict(request.form)
                response = form_response(dict_req)
                return render_template("index.html", response=response)
        except Exception as e:
            print(e)
            error = {"error": "Something went wrong!! Try again later!"}
            error = {"error": e}
            return render_template("404.html", error=error)
    else:
        return render_template("index.html")
```

Code Snippet

```
(bt5153env) rz@RuisPeralMacPro spam_detection % python app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5001
 * Running on http://192.168.1.214:5001
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 111-509-677
127.0.0.1 - - [22/Feb/2023 22:56:17] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [22/Feb/2023 22:56:17] "GET /static/script/index.js HTTP/1.1" 404 -
127.0.0.1 - - [22/Feb/2023 22:56:17] "GET /static/css/main.css HTTP/1.1" 200 -
127.0.0.1 - - [22/Feb/2023 22:56:18] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [22/Feb/2023 22:56:37] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [22/Feb/2023 22:56:37] "GET /static/css/main.css HTTP/1.1" 200 -
127.0.0.1 - - [22/Feb/2023 22:56:37] "GET /static/script/index.js HTTP/1.1" 404 -
```

Run app.py



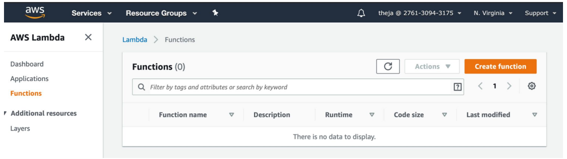
Web Server is alive

Create app.py

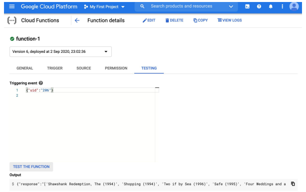
- Check the full implementation with ML pipeline in our github page
- Other examples:
    - Keras + Image Classification + Flask
    - Test REST API using Postman
    - Using Gunicorn to provide a WSGI server for applications
    - Use streamlit that HTML & CSS are not required

Severless deployments

- Reduces the DevOps overhead of deploying models as web services
  - We have to take care of provisioning and server maintenance
  - Worry about scale. Would one server be enough?
  - Reduce the efforts and deployment time when the team size is small
- 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.



Lambda Functions

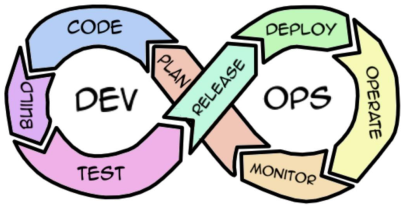


Cloud Functions

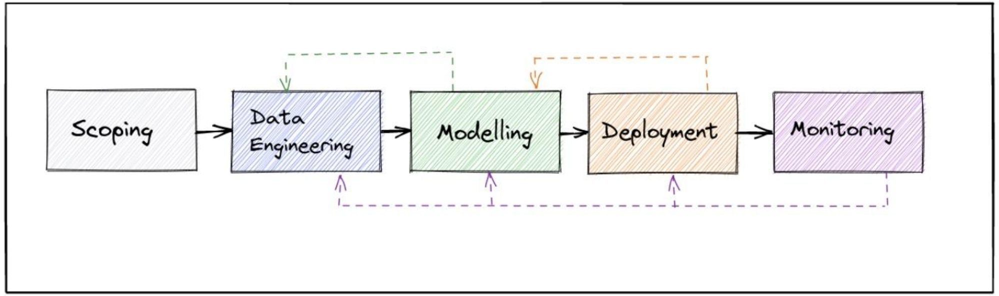
3. MLOps

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

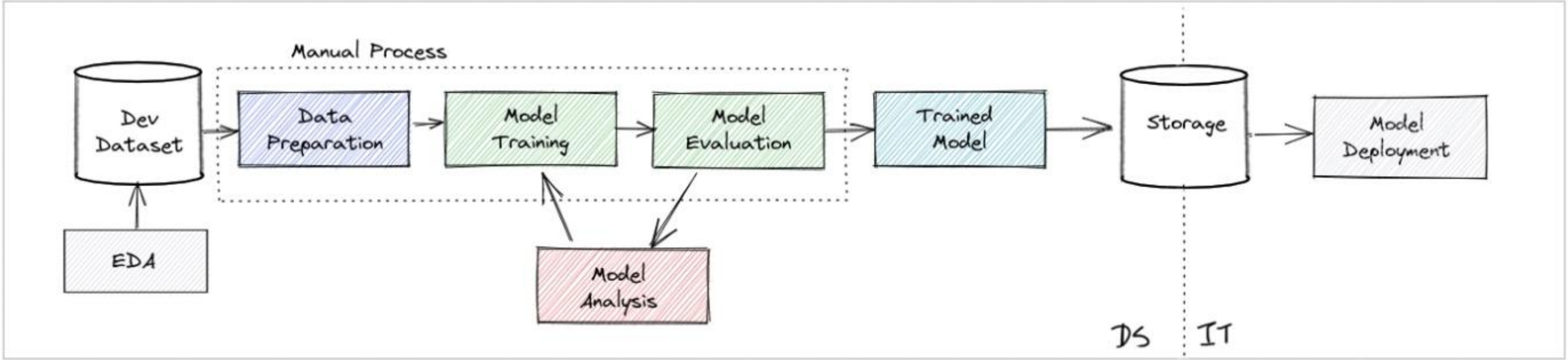
- Monitoring
  - Track performance metrics
    - ML metrics: F1 score, MSE, …
    - Ops metrics: uptime, throughput, response time
  - Drift detection
    - Concept drift: *when the relation between input and output has changed*
    - Label drift: *changes in predictions, but the model still holds*
    - 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

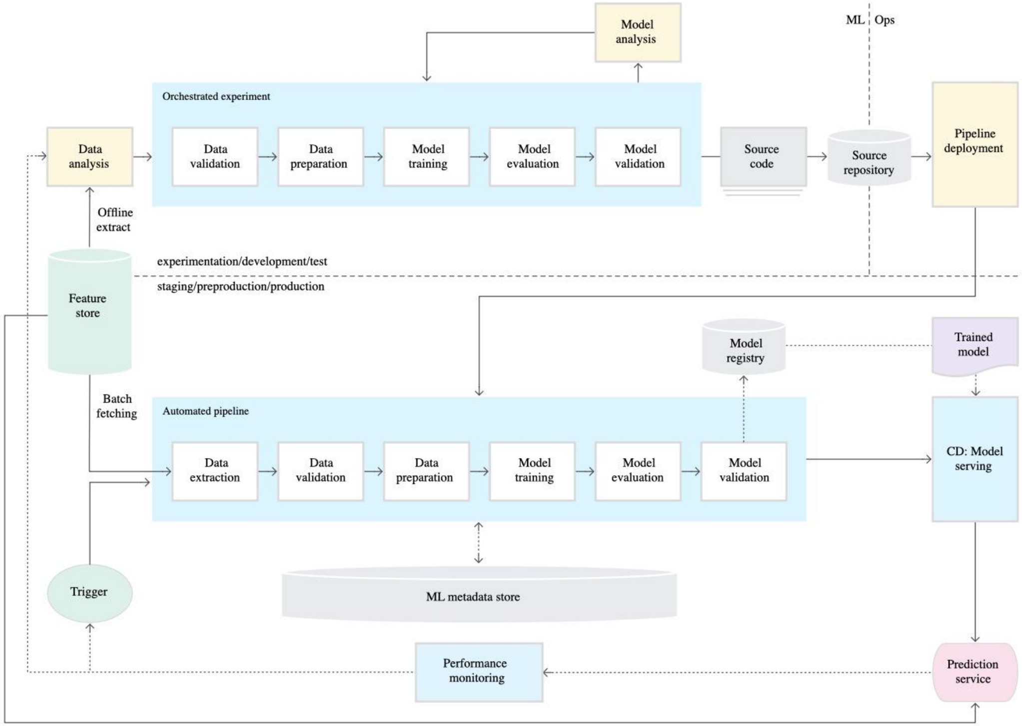
All the work are done manually



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 spam detection

- 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 spam_detection
conda activate spam_detection
```

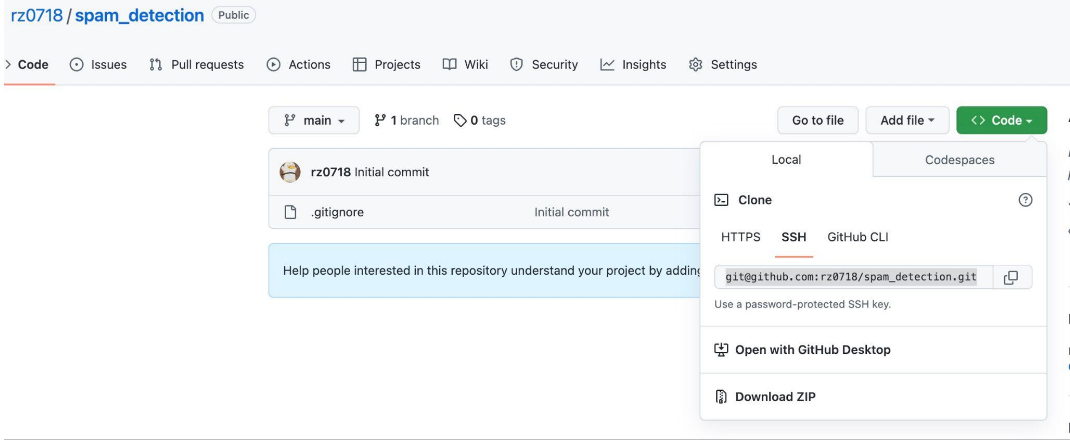
Create project structure using the cookiecutter

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

```
You've downloaded /Users/rz/.cookiecutters/cookiecutter-data-science before. Is
it okay to delete and re-download it? [yes]: yes
project_name [project_name]: spam_detection
repo_name [spam_detection]: spam_detection
author_name [Your name (or your organization/company/team)]: rz_nus
description [A short description of the project.]: endtoend ml pipeline
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
```

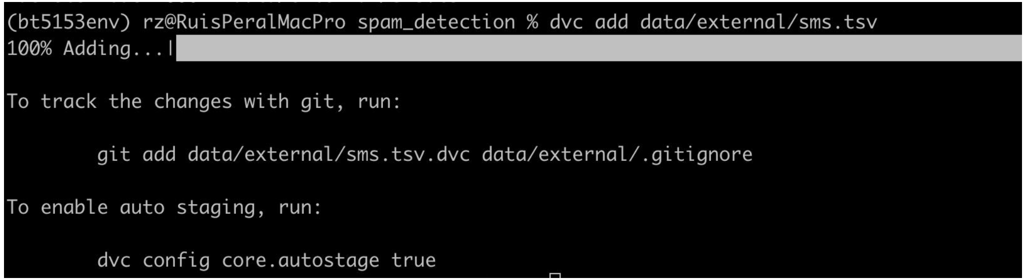
Create a github repo

```
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
```

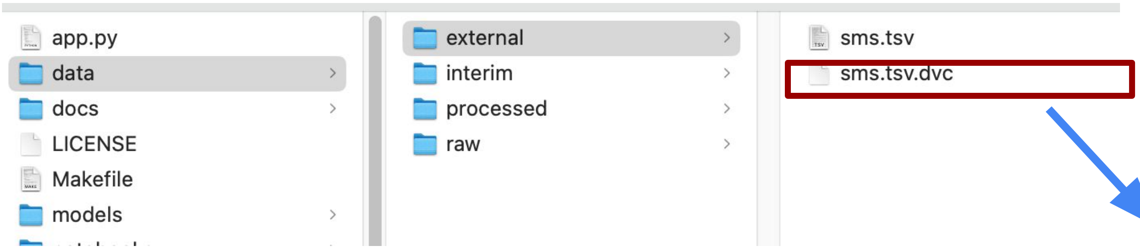


Track data version with DVC

pip install dvc
dvc init
dvc add <path_for_the_data_file>



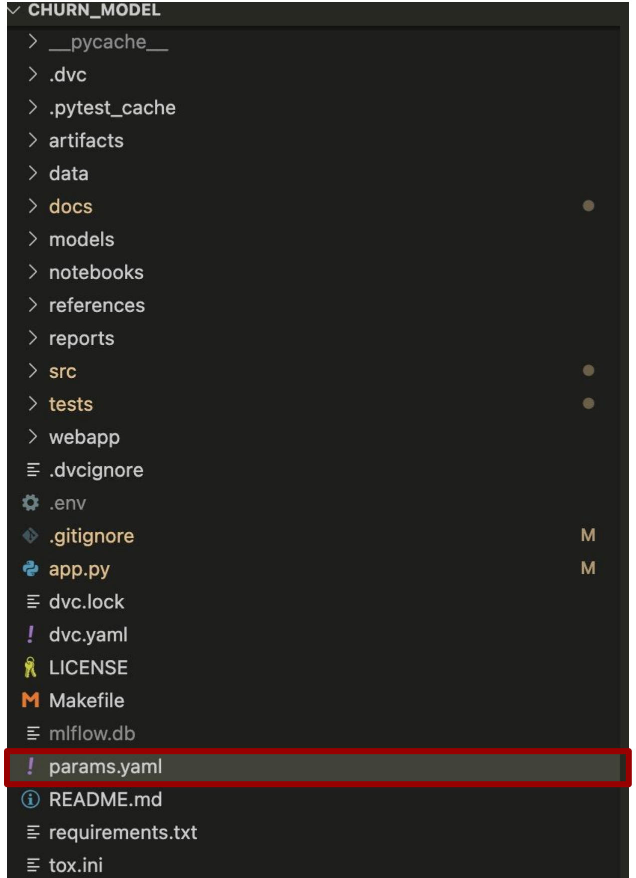
Data Version Control



Used by dvc to track sms.tsv

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 older

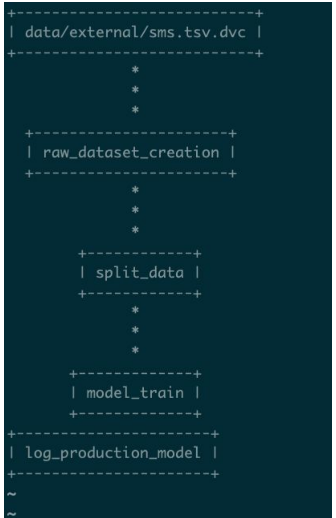
- src
  - data
    - __init__.py
    - .gitkeep
    - load_data.py — Copy external train.csv into the data/raw folder
    - split_data.py — Split the train.csv in raw folder to new train vs test in processed folder

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”



Pipeline creation with DVC

```
+-------------------------+
| data/external/sms.tsv.dvc |
+-------------------------+
             *
             *
             *
  +---------------------+
  | raw_dataset_creation |
  +---------------------+
             *
             *
             *
       +------------+
       | split_data |
       +------------+
             *
             *
             *
       +-------------+
       | model_train |
       +-------------+
+---------------------+
| log_production_model |
+---------------------+
~
~
```

39 lines (36 sloc) | 1000 Bytes

```yaml
stages:
  raw_dataset_creation:
    cmd: python src/data/load_data.py --config=params.yaml
    deps:
    - src/data/load_data.py
    - data/external/sms.tsv
    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/spam_train.csv
    - data/processed/spam_test.csv

  model_train:
    cmd: python src/models/train_model.py --config=params.yaml
    deps:
    - data/processed/spam_train.csv
    - data/processed/spam_test.csv
    - src/models/train_model.py
    params:
    - random_forest.max_depth
    - random_forest.n_estimators
    - count_vectorizer.max_features

  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
    - count_vectorizer.max_features
    outs:
    - models/model.joblib
```

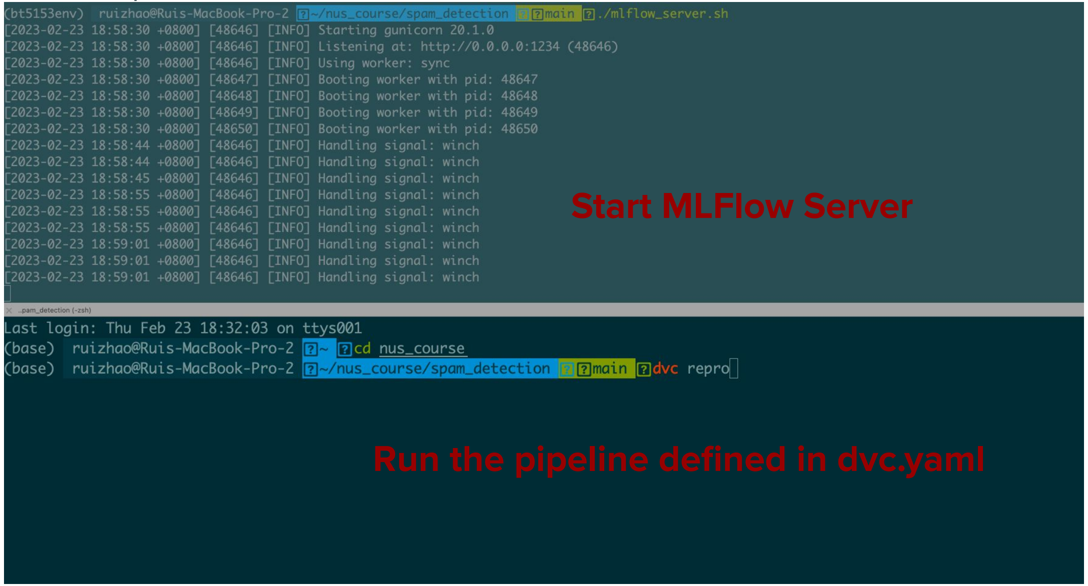
Code Snippet: dvc.yaml

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

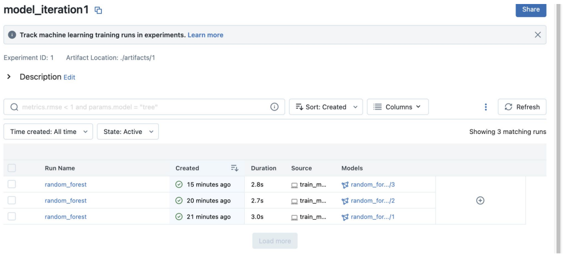


Why DVC

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

```
(bt5153env)  ruizhao@Ruis-MacBook-Pro-2 ?~/nus_course/spam_detection ? ?main ?dvc repro
'data/external/sms.tsv.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
  - Model_train
  - Log_production_model



Build ml pipeline using DVC and MLflow

- Check the full implementation with ML pipeline in our github page

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

Next Class: Causal Inference