Data Science in production

Lecture 2: Coding best practices

Alaa BAKHTI

Goal

- Make experiments reproducible
  - When submitting a research paper that depends on an experimental result, researchers are strongly encouraged to submit code that produces this result (code + dataset) (e.g. NIPS conference)
- Make the code reusable
- The produced code is the ownership of the team, other team members may use it > should be of a good quality.
- How to go from an exploration notebook to a tested and documented python package

IDE - Integrated development environment

- Why?
- Tools: PyCharm, Visual Studio code
- You can get the PyCharm Professional version licence with your EPITA email

PEP & PEP 8 - Style Guide for Python Code

- **PEP**
  - Python Enhancement Proposal
  - like an amendment to the constitution of a country
  - Each PEP determines how the Python language will be changes
  - The PEP index can be found in the PEP0
- Document that describes new features proposed for Python and documents aspects of Python, like design and style, for the community
- PEP 8 provides guidelines and best practices on how to write Python code
- Focus on improving the readability and consistency of Python code
- PEP 8 -- Style Guide for Python Code

Naming convention

- Use the programming language naming conventions (PEP 8 for Python)
    - *“Function names should be lowercase, with words separated by underscores as necessary to improve readability”* - PEP8
- Use descriptive names that describe what the object represents

```
x = [user1, user2, user3]
user_list = [user1, user2, user3]
```

```
def f(y1, y2):
    return np.sqrt(((y1 - y2) ** 2).mean())

def compute_rmse(predictions, targets):
    return np.sqrt(((predictions - targets) ** 2).mean())
```

Linter

- *“A linter is a static code analysis tool used to flag programming errors, bugs, stylistic errors and suspicious constructs”* - wikipedia
- Flake8 is a popular Python linter that can be used to:
  - Enforce the PEP8 coding style compliance.
  - Analyse code and flag programming errors (unused imported packages, undefined variables, …)
  - etc
  - Official Flake8 documentation
- Most IDEs already integrate linters

Can you identify the 7 coding style errors in this code?

```python
import numpy as np

def f( a , b) :
  x, y = 3, 4
  return (a-b) - y**2
```

```
└─ flake8 code-quality.py
code-quality.py:2:1: F401 'numpy as np' imported but unused
code-quality.py:4:1: E302 expected 2 blank lines, found 1
code-quality.py:4:7: E201 whitespace after '('
code-quality.py:4:9: E203 whitespace before ','
code-quality.py:4:14: E203 whitespace before ':'
code-quality.py:5:5: F841 local variable 'x' is assigned to but never used
code-quality.py:6:28: W292 no newline at end of file
```

Refactoring

“Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior” - Martin Fowler

Why?

- Make the code more efficient and maintainable.
- Remove code smell and reduce technical debt

Possible refactoring techniques

- Rename variable, function
- Extract variable, function
- Remove dead code
- Inline variable, function
- A list of refactoring techniques can be found here

Refactoring best practices

- Start with baby steps instead of directly ripping out the old implementation
- Test your code before refactoring to make sure you do not change the code behavior
- For simple refactoring (renaming, extraction, etc), it is better to use your IDE.

Patterns & principles

- **KISS** - Keep It Simple & Stupid
- **DRY** - Don’t Repeat Yourself
- **YAGNI** - You Ain’t Gonna Need It
- Single responsibility

Python basics

Type hinting

Python package, sub-package and modules

- Module: a file containing Python code (e.g. inference.py)
- Package: a directory of Python modules
- Distribution: an archived module/package (tar, zip, whl, etc)

```
my-distribution/
├── dist
│   └── my-distribution-0.1.0.tar.gz
└── my_package
    ├── __init__.py
    └── my_module.py

$ pip install my-distribution
```

Some advices

Encoding categorical variables

- ❌ Never use *pandas.get_dummies* in production
  - Does not guarantee consistent encoding when applied to new data with different feature values (red, blue during training and red, yellow during testing)
  - Not possible to manage exceptions (new feature values for example)
- ✅ Use a persistent encoder instead like *OneHotEncoder* or *OrdinalEncoder*
  - Possible to control what happens when a new feature value is encountered
    - Raise an error *handle_unknown='error'*
    - Ignore *handle_unknown=ignore* and set the new value to
      - zero (one hot encoder): encode(‘red”) [0, 0, 0]
      - Unknown value (ordinal encoder) (e.g. None): encode(‘high school’) = None
  - When to use each encoder (⚠️ without taking the memory & dimensionality constraint into account)?
    - OneHotEncoder for non-ordinal features (countries: France, Egypt, India, etc)
    - OrdinalEncoder for ordinal features (kindergarten, primary school, high skool, etc)
- ⚠️ Use *LabelEncoder* to encode target values and not categorical features

Some useful links about encoding

- Transforming categorical features to numerical features
- Are You Getting Burned By One-Hot Encoding? A common technique for transforming categorical variables into
- Encoding categorical variables
- Feature Engineering - Handling Cyclical Features

Stick to one standard in the project scope

```python
# Double or single quotes for strings
course_name = "Data Science in Production"
school_name = 'EPITA'

# Make changes in dataframe inplace or not
df.drop(['username', 'user_email'], axis=1, inplace=True)
df = df.drop(['username', 'user_email'], axis=1)
```

Keep a copy of your raw data

```
import pandas as pd

df_master = pd.read_csv('path/to/data/file/csv')
df = df_master.copy()
```

Preserve immutability

```
# Mutable
df = df.drop(['username', 'user_email'], axis=1)
# 👆 can be executed only one time, if we re-excute it we willl get an error
# because the columns no longer exists
df = df.drop(['username', 'user_email'], axis=1)

# Immutable
df = df_master.drop(['username', 'user_email'], axis=1) # with df_master
df_without_user_id = df.drop(['username', 'user_email'], axis=1) # with df
```

And finally

- Persist the objects you are using for data preparation so that you use them in production (encoder, scaler, etc). You can use *joblib* for that.

- When you complete the training and evaluation of your model, re-train it on all the dataset before putting it in production

- Fix the seed for the random number generator to get reproducible results
    - the seed determines the state of the PRG (Pseudorandom Number Generator)
    - it makes the random (or better to say pseudorandom) number reproducible.
    - Check this for explanation on how it works
    - https://notebook.community/ageron/ml-notebooks/extra_tensorflow_reproducibility
    -

2 phases in Machine Learning

Training phase

Inference phase

Training and inference pipeline

Training pipeline

Will be used to train the model

Input: training dataset path

Output: trained model, model metrics

Steps

- Read data
- Pre-processing
- Feature engineering
- Model training
- Model evaluation

Inference pipeline

Will be used in production to make prediction

Input: user data

Output: predictions

Steps

- Read data (?)
- Pre-processing
- Feature engineering
- Model loading
- Prediction

Practical work

💎 Create a new commit each time you make a new change

Practical work

Project setup

1. Change the folder structure: add the models folder where you'll store the trained model, encoder, etc
2. Create a virtual environment with [Miniconda]
3. Add project dependencies in a requirements.txt file

```
── my_package
   ├── README.md
   ├── app
   ├── data
   │   ├── test.csv
   │   └── train.csv
   ├── notebooks
   │   └── house-prices.ipynb
   └── requirements.txt
```

Folder structure

Notebook code improvement

5. Add headings to the notebook
6. Seperate the notebook in 2 parts: Training and Inference
7. Seperate the Training section to Read dataset, Preprocessing, Feature engineering, Model training
8. Serate the Preprocessing and Feature engineering to Categorical and Numerical columns
9. Notebook code refactoring
    a. Use the same standards (string, inplace, etc)
    b. Rename variables
    c. Remove mutability if possible
    d. Extract code to functions
10. Add comments if needed
11. Save your trained model locally
12. Load the model in the inference pipeline

Functions extraction in python modules

13. Move the functions to python modules
14. Import and use the functions in the notebook