Writing More Production-Ready Data Science Project (Part 1): Object Oriented Programing

Joshua Phuong Le · Follow

11 min read · Feb 27, 2023

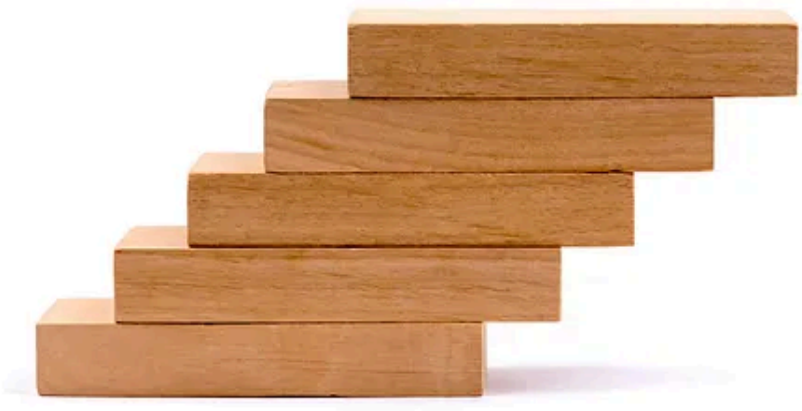


Photo by Volodymyr Hryshchenko on Unsplash

1. Introduction

Six months ago, when I started learning data science with my first Python project (LINK) — a simple text classification problem for the Yelp review data, the focus was learning how to implement basic sk-learn modules to get the results out in a Jupyter notebook environment. Apparently, there was no attention paid to writing clean codes nor adhering to best coding practices. As a result, there were many repeated chunks of codes doing the same thing but with different input arguments. Also, it would be a nightmare to debug, improve, distribute and deploy the whole project as things were messy.

Recently as I focused more on how to make proper data science projects that would be better fit for production, I started to pick up various tools and practices and figured out which were the best for me. I came across a great book from Sergios Karagiannakos named "Deep Learning in Production" and decided to use it as the blueprint to apply the OOP principles and practices to refactor the first project above.

In this 2-part series, I summarize my personal refactoring one part of the original project (logistic regression) into more proper structure so that all important execution can be run within a single `main.py` script (part 1), then move on to making a simple Streamlit app and containerize the whole project and deploy it to Google Cloud for public use (part 2).

Note that this article does not focus on the performance of the model. Its objective is to refactor key stages of a model lifecycle into easier-to-deploy code base structure.

You can find the source codes here:

GitHub - joshuavaple/yelp-review: refactored codes for YELP review regression

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or...

github.com

2. Project Structure and Folder Summary

```
.
├── LICENSE
├── README.md
├── Dockerfile
├── app.yaml
├── pyproject.toml
├── setup.cfg
├── setup.py
├── requirements.txt
├── data
│   ├── exported_models
│   └── yelp.csv
├── notebooks
├── scripts
│   └── main.py
```

```
├── streamlit
│   └── app.py
├── tests
└── yelpreview
    ├── __init__.py
    ├── configs
    │   ├── __init__.py
    │   └── config.py
    ├── dataloader
    │   ├── __init__.py
    │   └── dataloader.py
    ├── executor
    │   ├── __init__.py
    │   ├── inferrer.py
    │   └── trainer.py
    ├── model
    │   ├── __init__.py
    │   ├── base_model.py
    │   └── yelp_model.py
    └── utils
        ├── __init__.py
        ├── config.py
        └── postprocessing.py
```

The project follows the `src` style layout. The first 2 files are standard boiler-plate, `Docerfile` and `app.yaml` are for containerizing and deploying (explained in part-2), the next 4 files are for project packaging. For more information regarding project layout and packaging, check out my previous article here. For the remaining folders, they are functional sub-packages (with `__init__.py` files) or places where we segregate different commands/functionalities, as followed:

- `data/` folder: this is where the dataset is stored. It also contains the sub-folder to store the trained models (this is just a personal preference, you can store the trained models in a separate folder in the root folder).
- `notebooks/` folder: this is where I put draft codes to test before transferring them to proper modules, as the notebook environment is easier for experimenting.
- `scripts/` folder: this stores the main running script to be executed in the terminal —loading data, training, inferring…
- `streamlit/` folder: stores the single streamlit app file.
- `tests/` folder: stores any unit testing scripts.

- `yelpreview/` folder: this is the `src-` equivalent folder, storing the project source codes.

3. Module Details

The overall principle is that we should dedicate tasks along the model lifecycle to specialized classes. Each class handles a distinct job, such as loading configuration, loading data, training model, etc. This is known as Don't Repeat Yourself (DRY). For a simple project, this may make things more complex than absolutely necessary. However, for the intent of this article, I will strictly follow this principle as demonstrated by the book mentioned above.

3.1. Configs

The first sub-package is `configs`, containing one module called config.py. Within this module, we store settings that are reused during our experiments in a json data structure, such as training hyper parameters, path to obtain training data, path to save trained models, etc. The idea is that we should minimize hard-coding, but refer to only one place to re-configure any operation. This makes the project easier to maintain.

One important thing to note is that any relative path stored here is not with respect to the location of this config file, nor is it to the directory of any codes that are using these pieces of config info. It is actually with respect to the terminal directory you run any python script from. In this project, I will always run script from the root folder (./) hence these paths should be configured accordingly. This means, for example, executing `main.py` from root folder will be done via `python ./scripts/main.py` instead of calling `cd` into the `scripts` folder before executing `main.py`.

Here I also put the name of a logistic regression model trained on this dataset for inference later. You should change it to your trained model.

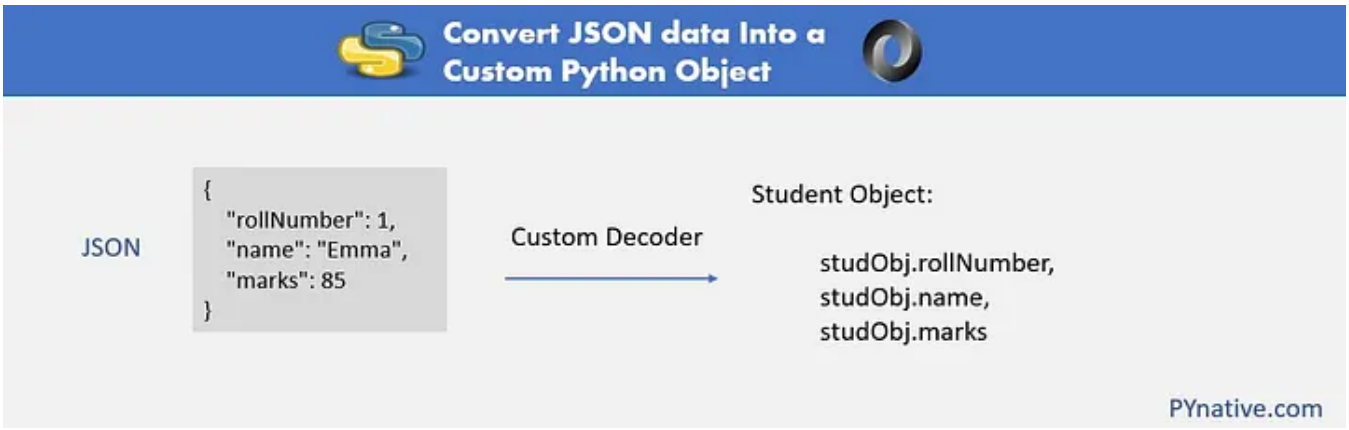
```
# inside .yelpreview/configs/config.py
CFGLog = {
    "data": {
        "path": "./data/yelp.csv",
        "x": "text",
        "y": "stars",
        "test_size": 0.2,
        "random_state": 2022,
        "ngram_range": (1, 2),
```

```
            "min_df": 10

        },
        "train": {
                "solver": "liblinear",
                "max_iter": 1000,
                "random_state": 2022,
                "C": 0.01,
                "penalty": "l2",
        },
        "output": {
            "output_path": "./data/exported_models/",
            "model_name": "20230217_152148_LogReg.pickle",
        }
}
```

3.2. Utils

The next sub-package contains 2 modules. The first one is for a class called Config, which helps us generate configuration objects from the config json data above. The use of Config class to process the json data with `object_hook` (by the class method `from_json`) helps us create a custom Python object and access the values inside the dictionary-like structure simply by using the dot `.` operation, instead of using the `dictionary["key"]` operation, as illustrated below (refer to this LINK for detailed usage). This comes in handy for the other codes where these configurations are being used again and again.



source: pynative.com

```
# inside ./yelpreview/utils/config.py
import json


class Config:
```

```
    """Config class which contains data, train and model hyperparameters"""

    def __init__(self, data, train, output):
        self.data = data
        self.train = train
        self.output = output

    @classmethod # using config to define constructor of the class
    def from_json(cls, cfg):
        """Creates config from json"""
        params = json.loads(json.dumps(cfg), object_hook=HelperDict)
        # init all class instance with data and train attributes
        return cls(params.data, params.train, params.output)


class HelperDict(object):
    """Helper class to convert json into Python object"""
    def __init__(self, dict_):
        self.__dict__.update(dict_)
```

The other module contains a class with just static methods to help us save trained models into pickle files with timestamps.

```
# inside ./yelpreview/utils/postprocessing.py
import datetime
import os
import pickle


class ModelSaving(object):

    @staticmethod
    def get_current_timestamp():
        now = datetime.datetime.now()
        return now.strftime("%Y%m%d_%H%M%S")

    @staticmethod
    def save_model_with_timestamp(vectorizer, model, output_config):
        filename = ModelSaving.get_current_timestamp() + '_LogReg' + '.pickle'
        filepath = os.path.join(output_config, filename)
        with open(filepath, 'wb') as outputfile:
            pickle.dump((vectorizer, model), outputfile)
        return print('Saved vectorizer and model to pickle file: ', filepath)
```

3.3. Dataloader

The next step, which is the data preparation step, is done by another class called `DataLoader`. Here, the input will be the path to the csv dataset, and the output will be the processed data ready for model training. Thus, this class is supposed to handle all pre-processing to transform data from the raw form to the precise form a machine learning algorithm expects. Here, we are using Scikit-learn's `linear_model.LogisticRegression` class, which expects the X and Y inputs as array-like to the training method (`fit`). Thus, it is necessary to vectorize the review texts in this class.

Here, you can see for the first time how the previous Config can be utilized from the variable `data_config`. Essentially, it will be derived analogously from the following fashion in the later part of the codes. As mentioned, here we are accessing the key-value of the original CFGLog dictionary in a more OOP-like manner through the dot operation.

```
cfg = CFGLog
config = Config.from_json(cfg)
data_config = config.data
```

Another important note (which I got it wrong in the original project if you noticed) is that for the training set `x_train`, you must invoke the vectorizer `fit_transform` in order to "learn" or "encode" the vocabulary of the training data into vector embeddings. However for the test set `x_test`, you **must not** use any `fit` method, which will remap the vectorizer's vector space to the vocabulary of the test data. Instead, only `transform` is invoked to simply convert the test data into the existing vector space derived from the training data.

This class returns the original data (lower case x, y...), the pre-processed data (upper case X, Y...) as well as the vectorizer object itself. This is important as the vectorizer is needed to transform new text data for inference later, and will be exported together with the trained logistic regression model into a single pickle file down the line.

```
# inside ./yelpreview/dataloader/dataloader.py
import pandas as pd
from sklearn import model_selection
```

```
from sklearn.feature_extraction.text import CountVectorizer
import numpy as np


class DataLoader:
    """Data Loader class"""

    @staticmethod
    def load_data(data_config):
        """Loads dataset from path"""
        df = pd.read_csv(data_config.path) # data_config will have .path attrib
        return df

    @staticmethod
    def preprocess_data(data_config, dataset = None): #dataset -> df
        """ Preprocess and splits into training and test"""
        x = dataset[data_config.x]
        y = dataset[data_config.y]
        test_size = data_config.test_size
        random_state = data_config.random_state
        # splitting
        x_train, x_test, y_train, y_test = \
            model_selection.train_test_split(x, y,
                                             test_size=test_size,
                                             random_state=random_state)
        # vectorizing
        vectorizer = CountVectorizer(ngram_range = data_config.ngram_range,
                                     min_df = data_config.min_df)
        # fit_transform on training texts: from text to sparse frequency fector
        X_train = vectorizer.fit_transform(x_train)
        X_test = vectorizer.transform(x_test)
        Y_train = np.array(y_train)
        Y_test = np.array(y_test)

        return x, y, x_train, x_test, y_train, y_test, X_train, X_test, Y_train
```

3.4. Executor

The next sub-package contains classes that help us executes the training and inferring steps as followed:

The trainer class below takes in an appropriate model, the processed training data and execute training by invoking the model's `fit` method.

```
# inside ./yelpreview/executor/trainer.py
class LogisticRegressionTrainer():
    def __init__(self, model, X_train, Y_train):
```

```
            self.model = model
            self.X_train = X_train
            self.Y_train = Y_train
        def train(self):
            self.model.fit(self.X_train, self.Y_train)
```

The inferrer class below first loads the pre-trained pickle file specified in the config file by deserializing it into two components — the vectorizer and the trained model itself. Then it uses the vectorizer to transform an input text to vector embeddings (just like how the `DataLoader` class did previously), before making prediction via `model.predict` method. Note the use of `Config` class again.

```
# inside ./yelpreview/executor/inferrer.py
from yelpreview.utils.config import Config
from yelpreview.configs.config import CFGLog
import os
import pickle


class Inferrer:
    def __init__(self):
        self.config = Config.from_json(CFGLog)
        self.saved_path = os.path.join(
            self.config.output.output_path,
            self.config.output.model_name)
        with open(self.saved_path, 'rb') as f:
            self.vectorizer, self.model = pickle.load(f)

    def preprocess(self, document: str):
        """Converts input document string to embeddings by the trained vectoriz
        return self.vectorizer.transform([document])

    def infer(self, document):
        """Converts input document string to embeddings and infer rating result
        document_embedding = self.preprocess(document)
        predicted_rating = self.model.predict(document_embedding)
        print(f'Model in use: {self.saved_path}')
        return predicted_rating
```

3.5. Model

The last sub-package defines arguably the central part of the project, the model itself. Here we find two modules.

The `BaseModel` class is an abstract base model class inherited from `ABC` (Abstract Base Class). By using abstraction, we are declaring a "blueprint" model for any other models to follow, with desired functionalities, without any specific code body. This base model can then be inherited by other models where specific implementations of the `@abstractmethod` are developed. Note that we are making use of the `Config` class here again, according to the previous sample fashion.

```
# inside ./yelpreview/model/base_model.py
from abc import ABC, abstractmethod
from yelpreview.utils.config import Config

class BaseModel(ABC):
    """Abstract Model class that is inherited to all models"""

    def __init__(self, cfg):
        self.config = Config.from_json(cfg)

    @abstractmethod
    def load_data(self):
        pass

    @abstractmethod
    def build(self):
        pass

    @abstractmethod
    def train(self):
        pass

    @abstractmethod
    def evaluate(self):
        pass

    @abstractmethod
    def evaluate_document(self):
        pass
```

The actual logistic regression model makes use of many of the classes we have defined so far. It loads data by the `DataLoader` class, it trains the classifier using the `LogisticRegressionTrainer` class, it has a method to evaluate any test document, as well as exports the trained model together with the vectorizer obtained from the data loading step into a single pickle file.

All along, it uses different portions of the configuration information stored in the Config object. This makes adjusting all these steps in a single configuration file a breeze.

```
from sklearn import linear_model
from datetime import datetime
from .base_model import BaseModel
from yelpreview.dataloader.dataloader import DataLoader
from yelpreview.executor.trainer import LogisticRegressionTrainer
from yelpreview.utils.postprocessing import ModelSaving

class YelpLogisticRegression(BaseModel):
    """Logistic regression model"""

    def __init__(self, config):
        super().__init__(config)
        self._name = 'LogisticRegression'

    def load_data(self):
        """Loads and preprocess data to inputs accepted by the model"""
        self.dataset = DataLoader().load_data(self.config.data)
        self.x, self.y, \
            self.x_train, self.x_test, self.y_train, self.y_test,\
            self.X_train, self.X_test, self.Y_train, self.Y_test, self.vectoriz
            DataLoader().preprocess_data(data_config = self.config.data,
                                         dataset = self.dataset)

    def build(self):
        """Builds the model"""
        # as the model architecture is vanilla logistic regression, there is nc
        # this is useful when you want to customize layers in a deep learning m
        self.model = linear_model.LogisticRegression()
        print('Logistic Regression model built')

    def train(self):
        """Compiles and trains the model with configured training hyper-paramet
        print('Set training hyper parameters')
        self.model = linear_model.LogisticRegression(
            solver = self.config.train.solver,
            max_iter = self.config.train.max_iter,
            random_state = self.config.train.random_state,
            C = self.config.train.C,
            penalty = self.config.train.penalty)
        print('Training is started')
        start_time = datetime.now()
        trainer = LogisticRegressionTrainer(
            model = self.model,
            X_train=self.X_train,
            Y_train=self.Y_train)
        trainer.train()
```

```
        end_time = datetime.now()
        training_time = (end_time - start_time).total_seconds()
        print(f'Training is completed in {"{:.2f}".format(training_time)} secor

    def evaluate(self):
        """Predicts resuts for the test dataset"""
        self.Y_test_pred = self.model.predict(self.X_test)
        self.Y_test_pred_proba = self.model.predict_proba(self.X_test)
        print('Model evaluation on test set completed, check model attributes f

    def evaluate_document(self, document: str):
        """Predicts the rating for an input string given a trained model"""
        document_embedding = self.vectorizer.transform([document]) # vectorizer
        return self.model.predict_proba(document_embedding), self.model.predict

    def export_model(self):
        """Exports the custom model's sklearn linear model"""
        output_config = self.config.output.output_path
        ModelSaving().save_model_with_timestamp(self.vectorizer, self.model, ou
```

4. The (Minimal) Main Running Script

With all steps being properly handled by different classes, the running file where users interface with the program can be kept extremely minimal. Here I included all steps of the project, and they can be commented out as and when needed, and only using the `Inferrer` class to make a test prediction on the sample document.

```
# inside ./scripts/main.py
"""main executing script to train, export and infer"""

from yelpreview.configs.config import CFGLog
from yelpreview.model.yelp_model import YelpLogisticRegression
from yelpreview.executor.inferrer import Inferrer

document = r"Was it worth the for a salad and small pizza Absolutely not Bad se

def run():
    """Builds model, loads data, trains and evaluates"""
    # load data and train model
    # mymodel = YelpLogisticRegression(CFGLog)
    # mymodel.load_data()
    # mymodel.build()
    # mymodel.train()
    # predicts reqsults on test set
    # mymodel.evaluate()
    # export to pickle
    # mymodel.export_model()
```

```
    # inference
    inferrer = Inferrer()
    print(inferrer.infer(document))

if __name__ == '__main__':
    run()
```

As mentioned in section **3.1.**, relative paths are configured so that they work when executing any scripts from the project root folder. Hence the proper way to run this script is below, where `projectroot` is the parent folder of the highest hierarchy files such as `LICENSE`, `README.md`, etc. The predicted class is 2-star.

```
projectroot/ python ./scripts/main.py
```

```
>>
Model in use: ./data/exported_models/20230217_152148_LogReg.pickle
[2]
```

5. Conclusion

Refactoring a poorly structured project was no easy task as many concepts need to be picked up and debugging done. However, this is extremely rewarding for me as I can apply a similar structure to future projects, and rest assured that deployment to web app and cloud service will be successful (as you will see in part 2 later).

This article would not have been possible without the help from the book mentioned in the Intro, please consider supporting the author by purchasing it or visit his website https://theaisummer.com/ for more quality content.

BECOME a WRITER at MLearning.ai

Mlearning.ai Submission Suggestions

How to become a writer on Mlearning.ai

medium.com

Machine Learning | Mlops | Data Science | Text Analytics | Ml So Good

Follow

Written by Joshua Phuong Le

143 Followers

I'm a data scientist having fun writing about my learning journey. Connect with me at https://www.linkedin.com/in/joshua3112/

More from Joshua Phuong Le

Joshua Phuong Le

Guide to Python Project Structure and Packaging

TIPS:

11 min read · Feb 3, 2023

394 5

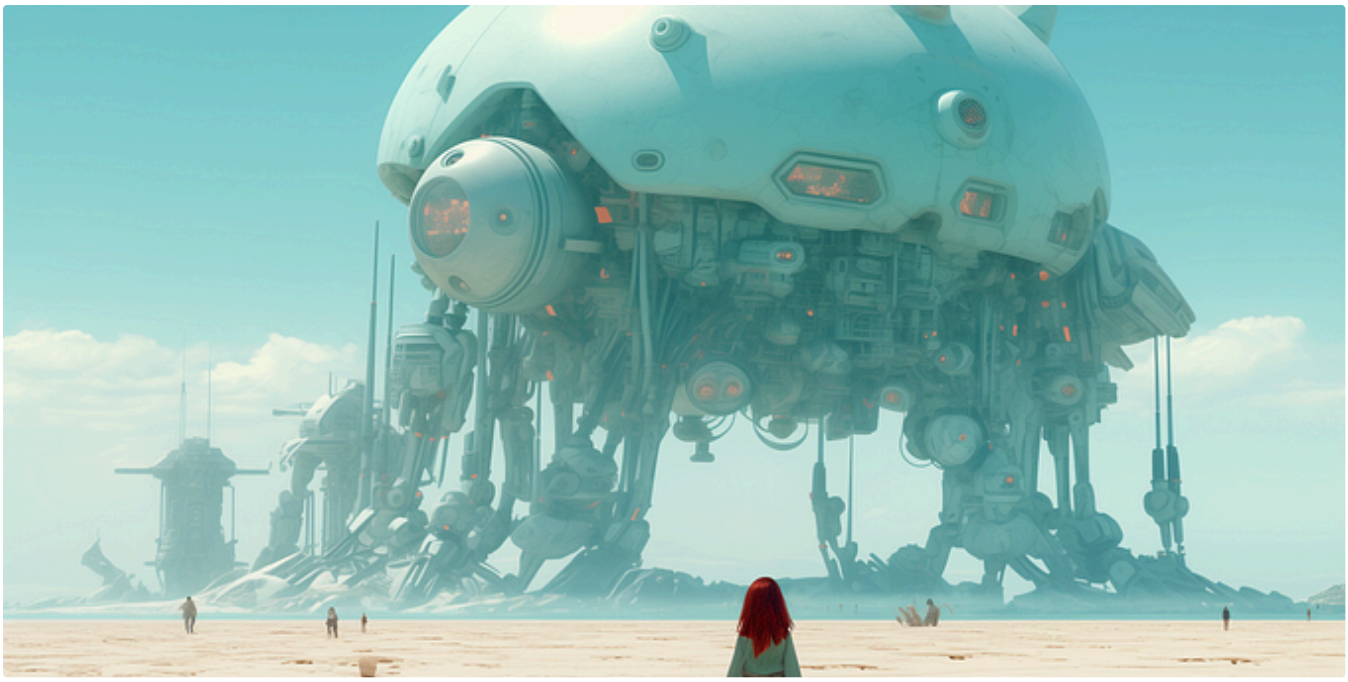
Maximilian Vogel

The ChatGPT list of lists: A collection of 3000+ prompts, GPTs, use-cases, tools, APIs, extensions...

Updated Jan-11, 2024. Added GPT Store, new resources.

11 min read · Feb 7, 2023

11.7K 141



Maximilian Vogel

The Generative AI List of Lists: 5000 Models, Tools, Technologies, Applications, & Prompts

A curated list of resources on generative AI. Updated June 1st, 2024—new models

16 min read · Jan 16, 2024

1.4K 9

Joshua Phuong Le

Building Custom Datasets for PyTorch Deep Learning Image Classification

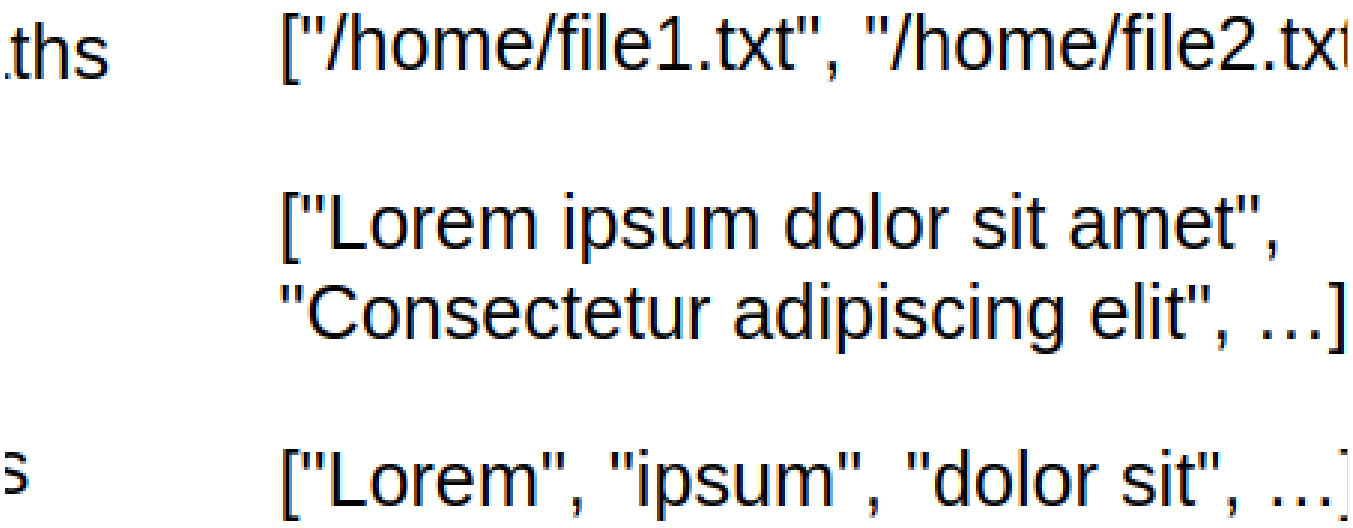
Learn how to use your own custom dataset for training a deep learning image classifier.

9 min read · Nov 22, 2022



See all from Joshua Phuong Le

Recommended from Medium



Iftimie Alexandru

Data pipeline recipes in Python

As a Computer Vision engineer, one of my jobs was to train Neural Network models. To train those models I had access to very large datasets...

10 min read · Dec 24, 2023

61 1

Vinícius Ramos

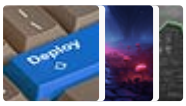
Python Poetry Simplified: A Step-by-Step Guide for Data Scientists

What is Poetry?

4 min read · Mar 24, 2024

14

Lists



Predictive Modeling w/ Python

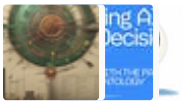
20 stories · 1251 saves

Practical Guides to Machine Learning

10 stories · 1506 saves

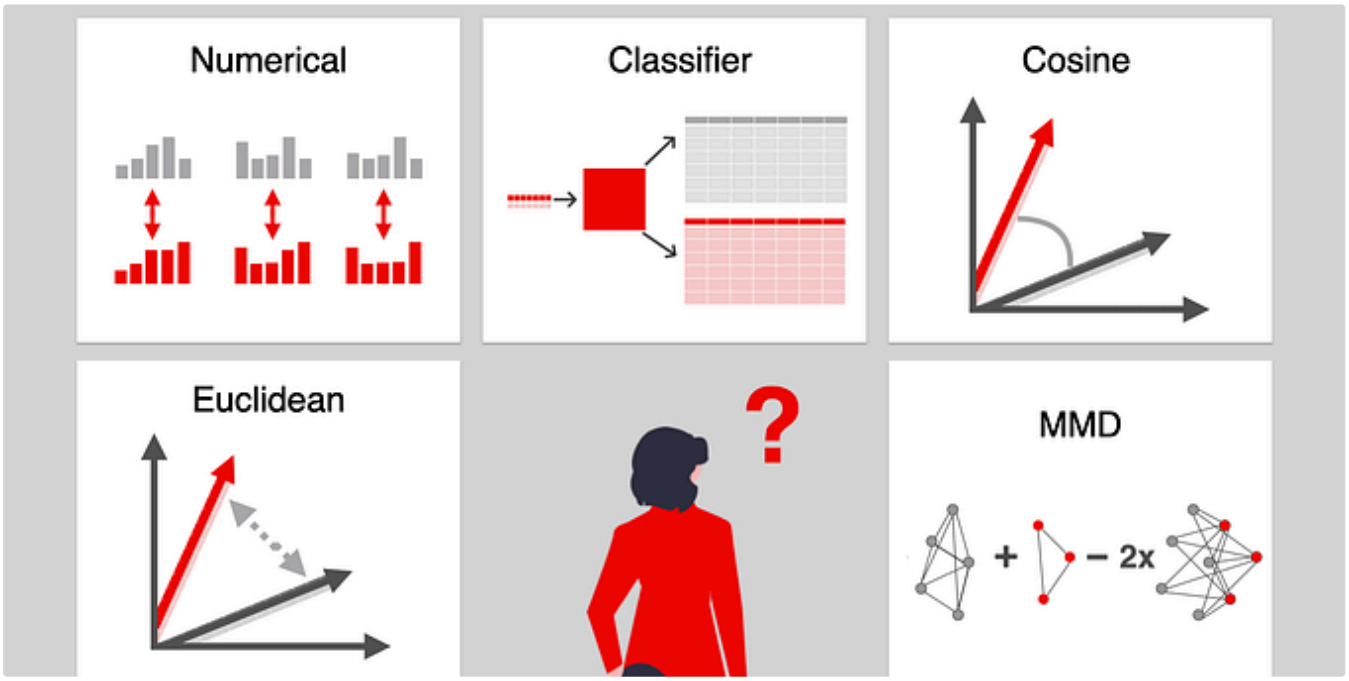
Natural Language Processing

1489 stories · 1001 saves



data science and AI

40 stories · 175 saves



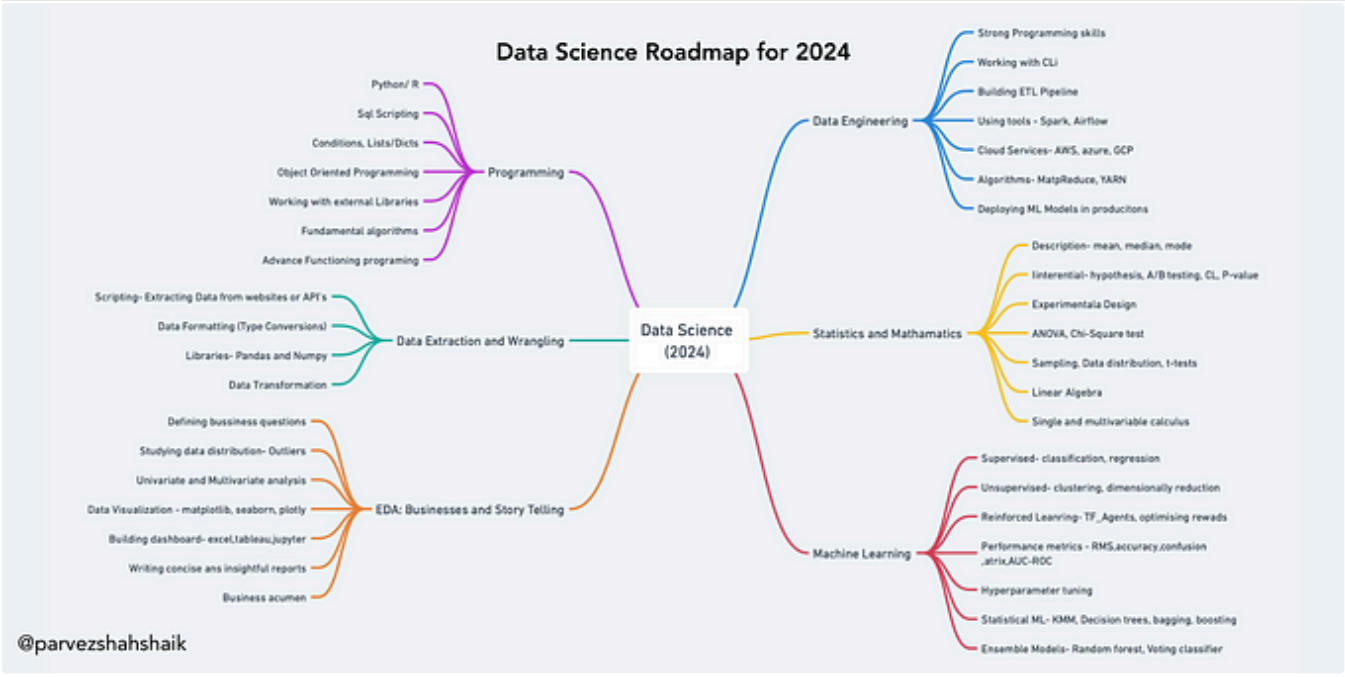
Elena Samuylova in Towards Data Science

How to Measure Drift in ML Embeddings

We evaluated five embedding drift detection methods

10 min read · Jun 14, 2023

508 4



Parvez Shah Shaik in Bootcamp

DataScience RoadMap for 2024

This article aims to guide you in building a strong portfolio showcasing your data science skills. It provides a framework, resources, and...

8 min read · Dec 27, 2023

239 5

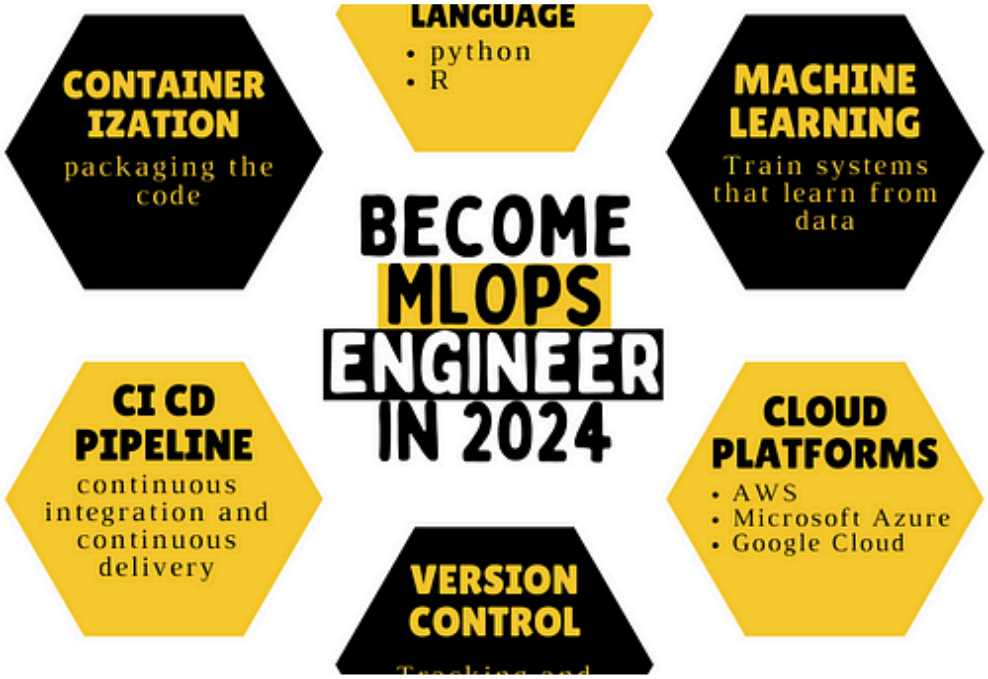
Sachinsoni

Understanding DVC : A practical guide to Data Version Control

Does managing data for your data science projects feel like a constant battle? You're not alone! But what if there was a tool that could...

7 min read · Mar 6, 2024

27



Asad iqbal

MLOps Roadmap | How To Become MLOps Engineer in 2024

A Comprehensive MLOps roadmap to become MLOps engineer in 2024

9 min read · Apr 1, 2024

335 3

See more recommendations