Practical Full Stack Machine Learning

A Guide to Build Reliable, Reusable, and Production-Ready Full Stack ML Solutions

ALOK KUMAR

Practical Full-stack Machine Learning

A Guide to Build Reliable, Reusable, And Production-Ready Full Stack ML Solutions

Alok Kumar

www.bpbonline.com

FIRST EDITION 2022

Copyright © BPB Publications, India

ISBN: 978-93-91030-42-1

Distributors:

BPB PUBLICATIONS
20, Ansari Road, Darya Ganj
New Delhi-110002
Ph: 23254990/23254991

DECCAN AGENCIES
4-3-329, Bank Street,
Hyderabad-500195
Ph: 24756967/24756400

MICRO MEDIA
Shop No. 5, Mahendra Chambers,
150 DN Rd. Next to Capital Cinema,
V.T. (C.S.T.) Station, MUMBAI-400 001
Ph: 22078296/22078297

BPB BOOK CENTRE
376 Old Lajpat Rai Market,
Delhi-110006
Ph: 23861747

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

www.bpbonline.com

About the Author

Alok is an author, speaker, open source contributor and a ML practitioner. He is currently leading the India Innovation center at Publicis Sapient to leverage emerging technologies to solve real world challenges.

He has extensive experience in leading strategic initiatives and driving cutting edge fast-paced data driven solutions ranging from products to platforms. His work has won several reputed awards. The inspiration to write the book on full-stack ML came from the observation of the struggle of scaling, productioning ML systems and teams.

Beyond work, He is passionate about democratizing knowledge.He manages multiple not-for-profit learnings and creative groups in NCR. He can be reached at linkedin (**https://www.linkedin.com/in/aloksaan/**) and twitter (**https://twitter.com/Aloksaan**).

About the Reviewer

Abhijeet Prakash has 6 years of extensive experience in Artificial Intelligence, Machine Learning and, Full Stack Development, using tools like Selenium, BS4, Google Colab, Apigee, FastAPI AWS, Google Cloud Platform with programming languages like Ruby, Node, Python, etc. Abhijeet pursued MCA from the Department of Mathematical Science and Information Technology, B.U. He has worked with various banks, NBFCs, and Fin-Tech companies. He worked as Machine Learning Engineer and currently working as a Full Stack AI Engineer. Abhijeet also wrote a book on Ethical Hacking.

Acknowledgments

I would like to thank my family for all their help, patience, and support. Without their support and assistance, I couldn't have imagined completing this book.

All the tools used in the book are coming from open source community and I would like to take this opportunity to thank the community that has helped democratize the AI knowledge so much.

Finally, I would really like to thank the team at BPB for all their help and support in my journey writing this book. It has been a pleasure to write this book and the team at BPB are certainly a big part of that. Many thanks to my development coordinator - Priyanka. Her continuous and regular follow ups helped me tremendously to keep us on track and focussed.

Preface

A successful data science project is not just about building powerful models, but the efficient execution of the entire project lifecycle. Unfortunately, data science has been made like ART and data scientist as ARTIST that uses hard to guess and unexplainable tricks.

If you find it difficult to decide on the correct initialization of hyper parameters or re-running someone else model training code, then you share the pain and frustration of data scientist community. Experience may teach you few tricks but there are limitations on how much we can remember and recall them accurately at the time of need.

Also, in the grand scheme of things, ML training is a part of a bigger data machine. This means inputs and outputs will always be reliant on other parts of the system. We would like you to pause here and think about the question – How would you revert your productionized model that is failing to meet the requirements or a step before it – how would you know it is failing or the accuracy has gone down the accepted limit? Again, how many such tricks across the whole pipeline would you able to learn, remember and recall?

The objective of this book is to introduce you to a collection of powerful, open-

Source tools and concepts needed to build an effective data science pipeline so that you don't have to remember the tricks but only remember the right tools, which We guess and, in my experience, is much easier to do.

To ensure that you share the excitement with me – consider this example.

You want to buy stocks and hence forth decided to seek advice from advisors. Being an experienced investor, you want to take the advice from others as well. Here is how it may look like. The percentage is the accuracy rate.

- Financial Advisor – 75%
- Stock Market Trader – 70%
- Market Research Team – 75%
- Social Media Expert – 60%

As clearly seen, all specialists' predictions are below 75%, however if you combine all their predictions, you get a completely different picture

Accuracy Rate = 1- (25%* 30% * 25% * 40%) = 99.25%

This is the power of "ensembling" and in machine learning world, ensemble learning is essentially a combination of multiple machine learning techniques performed together.

Now, to experiment different ensembling techniques, you can write it from scratch, or you can use ML-Ensemble library. ML-Ensemble combines a Scikit-learn high-level API with a low-level computational graph framework to build memory efficient, maximally parallelized ensemble networks in as few lines of codes as possible. We hope this drives home the idea and purpose of this book.

Since the book is about building effective pipelines and systems, we have organized the books around common steps of data science project. The steps look like this:

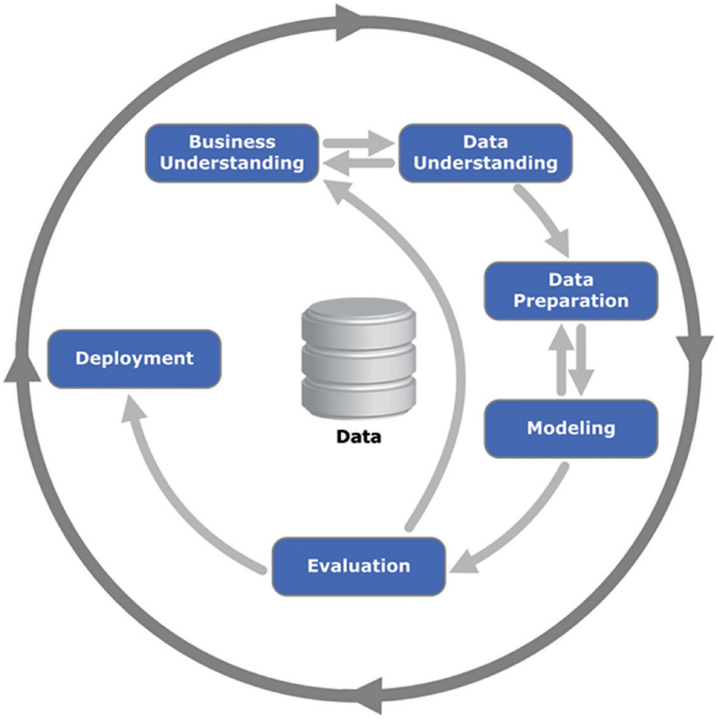


Figure 0.1: *CRISM DM common steps*

I am sure you would have seen a picture or diagram like this before. The steps are so common that it doesn't turns any head. It seems so logical that it is hard to believe that considerable effort was spent to build this intuitive process. Interestingly, this was created before data science became the sexiest job.

…

CRISP DM or Cross Industry standard process for data mining is a process methodology for data mining applications.

While there are other methodologies too, CRISP DM is one of the popular choices.

You would find variations of process adapted by lot of data mining tools without giving any attribution to it.

The objective of CRISP DM is to provide an Industry independent repeatable process for data mining work. Lately CRISP DM has slowly begun to fall aside but the underlying basics are still strong and useful.

The book chapters are loosely organized around different steps of CRISP DM. The intent is to provide a frame to group the different tools and libraries. Our mind has the amazing ability to recall things easily that are grouped/connected. Here are how the chapters are organized

Chapter 1: Organizing your data science project

Data science projects are experimental in nature and how you organize your project has a huge impact on the ease and speed of your experiments. A machine learning model is code plus data and hence both need to be organized properly. Getting started is not just about **organizing** your project but also deciding on the **environment, framework, baseline, target** metrics and **workflow**. In this chapter, we will explore concepts, tools and ideas that would help put the best foot forward.

Chapter 2: Preparing your data for Data science project

Description : Data collection and preparation are the foundation for trusted Machine learning/Deep learning models. a considerable amount of effort is spent at this step. The focus of this chapter will be to learn best practices, tools on data analysis and pre-processing for machine learning projects.

Chapter 3: Building your architecture for your data science projects

Building your architecture is not about using the latest popular and viral algorithm to build your model but training a model that meets the real-world expectations and challenges. The focus of this chapter will be to learn best practices on algorithm selection, hyperparameters initialization/ tuning and debugging techniques to enhance the model performance.

Chapter 4: Bye-Bye Scheduler, welcome airflow

Apache airflow is an open-source project to programmatically author, schedule and monitor workflows. The key benefit of machine learning pipelines is the automation it offers for different steps. Every new training dataset must go through the steps outlined in the CRISP DM process. Most of the team either do it manually or duct tape the steps making it extremely brittle. You need a pipeline if your model has users. If you are still not convinced then chap-5 will do that job. The objective of this chapter would be a gentle introduction of Airflow.

Chapter 5: Managing ML pipeline with MLflow

Majority of us would have experienced the frustration of running someone else code plus model. The libraries dependencies, hidden configurations and undocumented setup steps make it extremely difficult to treat someone else model like a black box. MLflow is an open-source project that helps you train, reuse, and deploy models with any library and package them into reproducible steps that other data scientists can use as a "black box," without even having to know which library you are using. The objective of this chapter is to introduce you to MLflow and how you can use it in your situations.

Chapter 6: Feature stores for ML

Feature store could be imagined as a warehouse of features. It is central vault for storing documented, access-controlled features. Feature store is an emerging concept with the objective of removing the challenges in taking ML models to production.

The focus of this chapter will be to learn about feature stores via an open-source feature store called **feast**.

Chapter 7: Serving ML as API

The focus of this chapter will be to learn how we can deploy ML model as an API. We will use **fastAPI** which is a modern, high-performance Python web framework perfect for building RESTful APIs. fastAPI can handle both synchronous and asynchronous requests and has built-in support for data validation, JSON serialization, authentication, and authorization. As a bonus, you will also learn about **Ray-serve**.

The chapters are independent and can be read in any sequence. This is very helpful because maybe you are interested in few chapters or may be some chapters knowledge are required immediately.

This may sound cliché, but it is true – the best way to learn and remember is to practice and repeat. Try preparing notes for each chapter and revisit them frequently. Repetition helps *retentions*.

Now decide on the chapter you want to start with and get going.

Happy learning!!

Downloading the coloured images:

Please follow the link to download
the ***Coloured Images*** of the book:

https://rebrand.ly/cd277f

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors if any, occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at:

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Table of Contents

Chapter 1
Organizing Your Data Science Project

You don't need to worry about organizing your project if:

- You are the only person working on the project.
- Your first trained model meets the project requirements.
- Your data does not require pre-processing at all.

If any of the preceding points are not true, then you need to think about effectively organizing your projects. A well-organized project enables better collaboration, repeatability, and re-usability. Project organization is not just about code base but the environment as well. Not everyone can afford environments like Google, Amazon, and Facebook. Cloud is not always the best answer, as marketed by various cloud vendors. If you are running model training continuously for weeks, then buying a GPU machine is a lot cheaper than doing it in cloud.

Structure

In this chapter, we will discuss the following topics:

- Project folder and code organization
- GPU 101
- On-premises vs. cloud

- Deciding your framework
- Deciding your targets
- Baseline preparation
- Managing workflow

Objective

After studying this chapter, you will be able to:

- Set up the project code base effectively. We will explore a library called **`cookiecutter`** that simplifies and standardizes the process a lot.
- Apprehend the best practices of selecting GPU. The options in the market are overwhelming and also quite confusing.
- Learn the best practices to decide infrastructure – cloud or on-premises.
- Select the framework (Think of TensorFlow, PyTorch, etc.) and hardware.
- Apply the tools for workflow management setup. We will explore sacred and **`omniboard`** projects to keep track of experiments.
- Decide on the target and define the metrics around it.
- Define your baseline.

1.1 Project folder and code organization

A good folder structure separates data processing, model definition, and model training. The following is a good method to organize the folders:

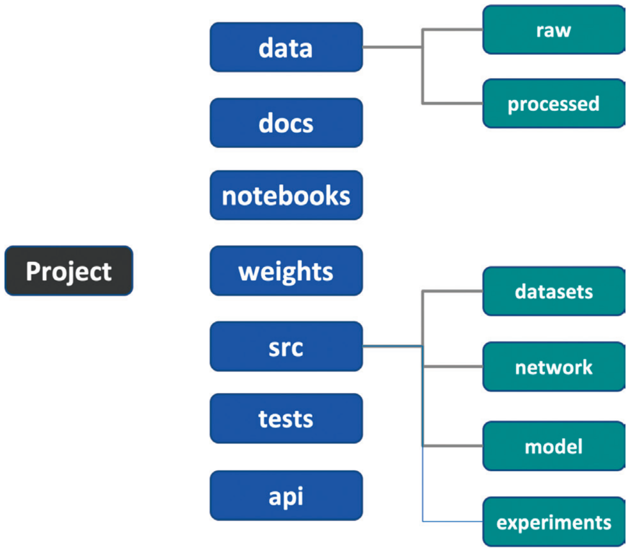


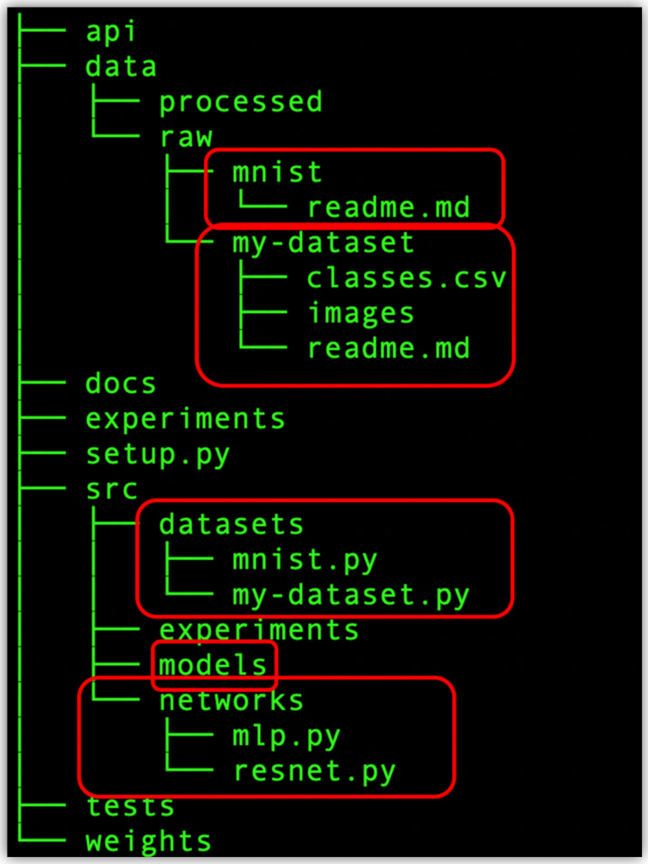
Figure 1.1: *Folder and code organization*

Let us go through each folder to understand its purpose:

- **`data`**: The purpose of data folders is to organize the data sources. This one has two folders which are as follows:
    - **`raw`** – This is the original, immutable data dump. This data should never be modified and should be the single source of truth.
    - **`processed`** – These are the final, canonical data sets for modelling or training. This will be the data that is cleaned, transformed, and extended to make it amenable for training.

  With this organization, you can immediately see some benefits such as:
    - Data becomes immutable via the **`raw`** folder.
    - We don't need to prepare the data every time due to the **`processed`** folder.
- **`docs`**: This contains project documentation. We don't need to convince you about the value of project documentation. We recommend using sphinx (https://www.sphinx-doc.org/en/master/) for documentation.
- **`weights`**: The weights folder stores the trained, serialized models, model predictions, or model summaries.
- **`src:`** This contains all the Python scripts required for your project. We recommend splitting it across 3 folders.
- **`datasets:`** This folder contains Python scripts to process the data.
- **`network:`** This folder has scripts to define the architectures. Only the computational graph is defined.
- **`model:`** This folder has the script to handle everything else needed in addition to the network.
- **`experiments:`** This folder contains the parameter/hyperparameters combinations that you would like to try.
- **`api:`** This folder exposes the model through a rest end point.

Let us go a further level down to look at some of the files. We have kept the minimal number of files for brevity.



```
├── api
├── data
│   ├── processed
│   └── raw
│       ├── mnist
│       │   └── readme.md
│       └── my-dataset
│           ├── classes.csv
│           ├── images
│           └── readme.md
├── docs
├── experiments
├── setup.py
├── src
│   ├── datasets
│   │   ├── mnist.py
│   │   └── my-dataset.py
│   ├── experiments
│   ├── models
│   └── networks
│       ├── mlp.py
│       └── resnet.py
├── tests
└── weights
```

Figure 1.2: *Folder structure with sample files*

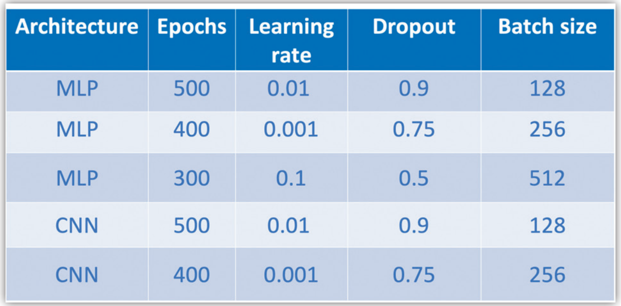
This is a condensed view of the structure to help focus on the idea. Let us explore some of the folders. The **`data/raw`** folder has a folder for MNIST (handwritten digit dataset) and some proprietary dataset. Let's call it "`my-dataset`." Essentially, each different dataset is maintained in its own respective folder.

The separation isolates different structures; for example, one of the image datasets may contain **`image`** class names mentioned in a csv but the actual images are dumped in a single folder. Maybe the second dataset that you collected from somewhere else has the image class name mentioned in the file name itself. It is worth reiterating that this data should never be changed.

The **`data/processed`** folder should maintain the processed data. In this case, it could be normalized data ready for training. The **`models`**' folder, as the name suggests, would contain the trained models.

For deep learning projects, it further helps to separate the core network from the rest of the stuff. The **`network`** folder defines the network architectures used. Think of it as a block which only defines the computational graph without caring about the input and output shapes, model losses, and training methodology. The loss function and the optimization function can be managed by the model. However, how does this help?

Imagine that you want to perform the following experiments:



Architecture	Epochs	Learning rate	Dropout	Batch size
MLP	500	0.01	0.9	128
MLP	400	0.001	0.75	256
MLP	300	0.1	0.5	512
CNN	500	0.01	0.9	128
CNN	400	0.001	0.75	256

Figure 1.3: *Hyperparameters values grid*

A scalable way to try all these experiments without modifying your network is to write a separate script – let's call it "*try-experiments*". A Python style pseudocode to run an experiment would look as follows:

```
Python try-experiment '{network : MLP, Epochs : 500,  learning_rate : 0.01,
Droput:0.9, Batch_size : 128}'
```

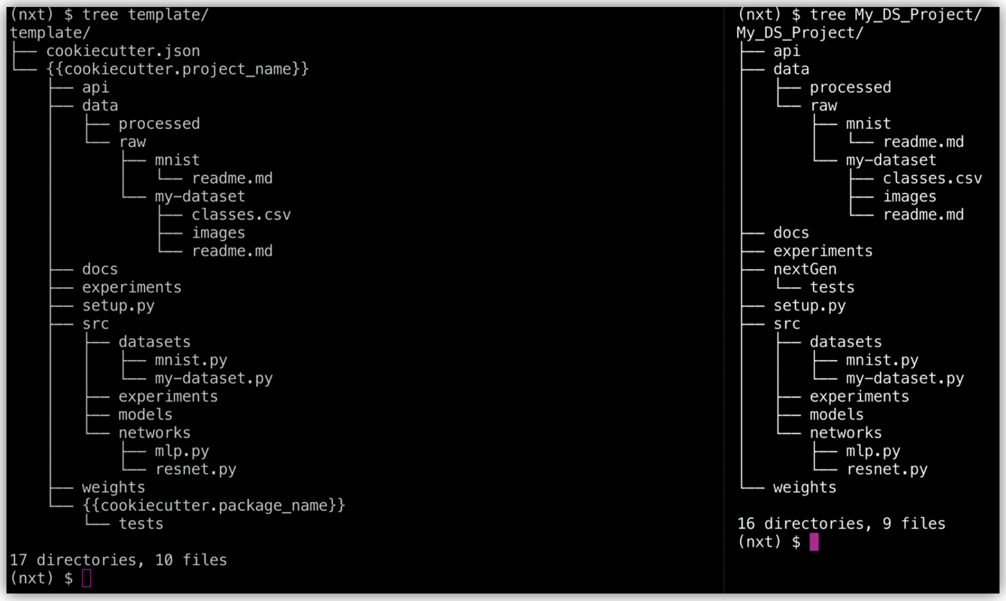
Put all these "*try-experiment*" calls in a shell script and you have a clean automated way to try different parameters. The credit for this idea goes to *Josh Tobin* (http://josh-tobin.com/).

Wouldn't it be nice if the whole folder setup could be automated? Of course, this can be done via custom scripts, but then script maintenance becomes a problem. Is there any clean solution?

Cookiecutter is a CLI tool (https://cookiecutter.readthedocs.io/en/1.7.2/index.html) to create an application boilerplate from a template. It uses a templating system—Jinja2 (https://jinja.palletsprojects.com/)—to replace or customize folder and file names as well as file content.

The whole thing may sound complicated, but using and understanding it is very intuitive. Think of cookiecutter this way: Earlier, we defined a bunch of folders/subfolders to organize our project. Now you want every new data science project to use that structure (template). So, you create a template from that structure and then anyone can pass that template to the cookiecutter CLI to generate the same folder/

file's structure. The left-hand side of the picture given below shows a template and the right-hand side shows the generated project.



```
(nxt) $ tree template/
template/
├── cookiecutter.json
└── {{cookiecutter.project_name}}
    ├── api
    ├── data
    │   ├── processed
    │   └── raw
    │       ├── mnist
    │       │   └── readme.md
    │       └── my-dataset
    │           ├── classes.csv
    │           ├── images
    │           └── readme.md
    ├── docs
    ├── experiments
    ├── setup.py
    ├── src
    │   ├── datasets
    │   │   ├── mnist.py
    │   │   └── my-dataset.py
    │   ├── experiments
    │   ├── models
    │   └── networks
    │       ├── mlp.py
    │       └── resnet.py
    ├── weights
    └── {{cookiecutter.package_name}}
        └── tests

17 directories, 10 files
(nxt) $
```

```
(nxt) $ tree My_DS_Project/
My_DS_Project/
├── api
├── data
│   ├── processed
│   └── raw
│       ├── mnist
│       │   └── readme.md
│       └── my-dataset
│           ├── classes.csv
│           ├── images
│           └── readme.md
├── docs
├── experiments
├── nextGen
│   └── tests
├── setup.py
├── src
│   ├── datasets
│   │   ├── mnist.py
│   │   └── my-dataset.py
│   ├── experiments
│   ├── models
│   └── networks
│       ├── mlp.py
│       └── resnet.py
└── weights

16 directories, 9 files
(nxt) $
```

Figure 1.4: *Project generation from a template*

Let us build a template to get a hang of things:

1. Create a folder that contains the template. For example, **`myTemplate`**.
2. Inside the **`myTemplate`** directory, we create the directory tree to be copied into the generated project. In order to generate a name for this project, we put the project name in templating tags **`{{`** and **`}}`**. In the preceding example (*figure 1.4*), we have used two template tags for **`{{cookiecutter.project_name}}`** and **`{{cookiecutter.package_name}}`**. The command snippets are kept minimal for brevity as shown below:

```
mkdir {{cookiecutter.project_name}}
cd {{cookiecutter.project_name}}

mkdir {{cookiecutter.package_name}}
cd {{cookiecutter.package_name}}

mkdir test
---------
---------
```

3. All the tags are like variables that get defined in **`cookiecutter.json`**. The JSON for the previous template looks like this:

   ```
   {
       "project_name": "Your project name",
       "author_name": "Your name (or your organization/company/team)",
       "description": "A short description of the project.",
       "directory_name": "Directory name",
       "package_name":"Package test script",
      "open_source_license": ["MIT", "BSD-3-Clause", "No license file"],
       "python_interpreter": ["python3", "python"]
   }
   ```

   Cookiecutter will replace the value of the **`project_name`** provided by the user while generating the project. You can define the tags names as whatever you want. In the documentation, they call it tags but we prefer to imagine them as variables that get replaced with actual values during run time.

4. Now we can run Cookiecutter and create a new project from our template. Move to a directory where you want to create the new project. Then run Cookiecutter and hand it the directory where the template lives. For our example, it would look like this:

   ```
   $ cookiecutter template/
   ```

5. It is not much of a saving if you define the folder structure as template and then generate new projects using that template. It is possible to upload the whole template to a version control repository (like Git) and then use that instead as your template. An example would be helpful here as shown below:

   ```
   cookiecutter https://<<your repo>>/<<user>>/myTemplate
   ```

Over a period of time, you will have a bunch of templates for your organization and that is not a bad thing. Think of different templates for different project types - CV (computer vision) or NLP problems. Given that it is open source, there are a lot of templates for use to reuse or to use as a starting point. So next time, don't copy-paste your project—cookie-cut it!

1.2 GPU 101

Before we start, you might be wondering—why do I need a GPU, or why a CPU is insufficient for machine learning training needs?

To answer this question, we need to look at some major differences between a CPU and a GPU:

CPU	GPU
Few fast processors	Thousands of slow processors
General purpose computations	Massive parallel computations

Table 1.1: *CPU vs GPU*

You might know that CPUs usually have few fast processors or cores, whereas GPUs have thousands of comparatively slow processors or cores. CPUs are designed for general purpose computations, whereas GPUs are designed for massively parallel computations.

To understand and appreciate the value of GPU, let us look at the requirements of a **DL** (**deep learning**) project:

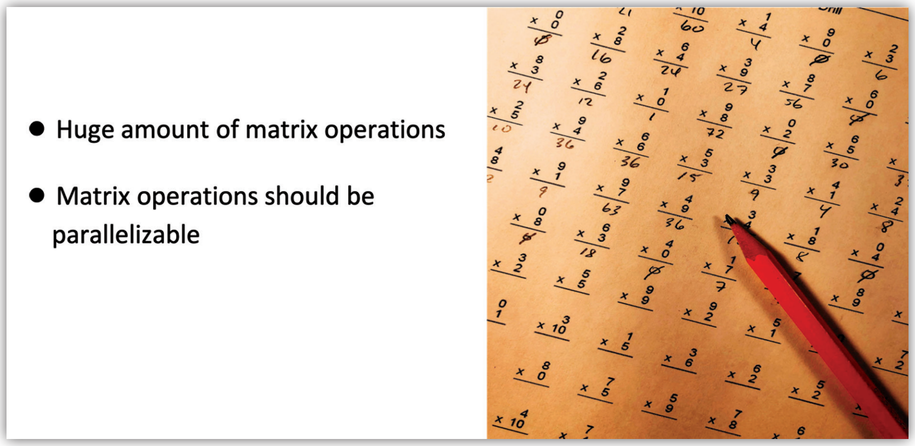

- Huge amount of matrix operations
- Matrix operations should be parallelizable

Figure 1.5: *DL Computational requirements.*
Image credit: *Chris Liverani on Unsplash*

Deep learning requires a massive amount of matrix calculations. Fortunately, these matrix operations are parallelizable, and hence, you need the massive parallel computation power of GPUs to perform the operations quickly.

Maybe you are stubborn, and so, despite all the benefits of GPUs, what if you still insist to do training on a CPU? Well, you end up waiting more than actually experimenting.

Figure 1.6: *The ling training cycles*
Image credit *- [Massimo Sartirana](#) on [Unsplash](#)*

A GPU enables us to have faster turnaround time for training and to be honest, in a lot of cases, the success of your data science project depends on it. Now that we understand the value of GPUs, the next question is to decide on the GPU brand to pick. This is easy at the moment. Nvidia and AMD are the two primary vendors in the GPU space.

Figure 1.7: *The two GPU vendors*

For hassle-free software support of GPU acceleration, we recommend Nvidia. The primary reason for this is a technology called Cuda. Nvidia early-on invested lots of their engineering effort in the development of low-level optimizations using the Cuda library.

Due to this early lead, most of the deep learning frameworks like TensorFlow and PyTorch have better support of Nvidia graphic cards. If you already have AMD GPU and you are willing to troubleshoot some starting hiccups, you can try out GPU acceleration on AMD graphic cards also. There is experimental support for some of these libraries on OpenCL as well as by using AMD Radeon through the Compute (https://github.com/RadeonOpenCompute) project.

OpenCL is an alternative to the CUDA library for general purpose GPU acceleration. For this book, we recommend sticking to Nvidia. It's no surprise that there are lots of options in the Nvidia space as well. To understand the different options, we prefer using three parameters:

- Processing power (clock speed, core count, and GPU generation)
- Video memory (GPU memory)
- GPU bandwidth

The processing power of a GPU is driven by three parameters - clock speed, core count (or CUDA core count) and GPU generation. A better clock speed usually means faster processing.

A high core count essentially means more parallelization potential. And lastly, usually due to improvements in fabrication technology each year, the newer generation GPUs are more efficient and powerful than the older generation.

The video memory for a GPU is basically random-access memory that is required to load the training data before any computations. You may encounter a GPU that is out of memory error frequently if the memory is low. While there are some workarounds to avoid the memory errors, it is always better to have a GPU with a large GPU memory.

The last thing that you need to look out for is whether your machine has sufficient bandwidth available for proper GPU support. Parameters like available PCI Express lanes will reveal this information. PCIe lanes are essentially how the PCIe devices communicate to the system through that PCIe bus. It's similar to how multi-lane highways work—with higher traffic areas needing more lanes to handle the volume, certain components require higher throughput, thus requiring more lanes. In the context of GPU, it is about how fast data can be delivered to a GPU.

If you are in the market for building a new workstation, these are some of the GPU options that you can choose from:

GPU	**Architecture**	**RAM (GB)**	**Tensor cores**	**Year**
K80	Kepler	24		2014
Titan X	Maxwell	12		2015
P100	Pascal	16		2016
1080 Ti	Maxwell	11		2017

V100	Volta	16	120	2017
Titan V	Volta	12	110	2017
2080 Ti	Turing	11	60	2080
Titan RTX	Turing	24	130	2018
RTX 8000	Turing	48	160	8000

Table 1.2: *NVIDIA GPU options*

Let us understand the chart so that you can confidently decide.

- NVIDIA comes with a new architecture every year.
  - Kepler
  - Maxwell
  - Pascal
  - Volta
  - Turing
- Tensor cores are just more heavily specialized to the types of computation involved in machine learning compared to the CUDA core. It uses much less computation power at the expense of precision than CUDA cores, but that loss of precision doesn't have that much of an effect on the final output. Therefore, for machine learning models, tensor cores are more effective at cost reduction without changing the output that much. Tensor cores were introduced from the volta architecture. Technically speaking, CUDA cores operate on a per-calculation basis; each individual CUDA core can perform one precise calculation per revolution of the GPU. Tensor cores, on the other hand, can calculate with entire 4×4 matrices operation being calculated per clock.

Here are a few key take-aways. A lot of these recommendations are from Tim Dettmer's post https://timdettmers.com/2019/04/03/which-gpu-for-deep-learning/. We encourage you to read the post.

- Kepler/Maxwell are 2-4 times slower than Pascal/Volta and not worth buying.
- Best GPU overall: RTX 2070.
- GPUs to avoid: Any Tesla card; any Quadro card; any Founders Edition card; Titan RTX, Titan V, Titan XP.
- Cost-efficient but expensive: RTX 2070.
- Cost-efficient and cheap: RTX 2060, GTX 1060 (6GB).

- If you have little money: GTX 1060 (6GB).
- If you have almost no money: GTX 1050 Ti (4GB). Alternatively: CPU (prototyping) + AWS/TPU (training); or Colab.
- If you do Kaggle: RTX 2070. GTX 1060 (6GB) or GTX Titan (Pascal).
- If you are a competitive computer vision or machine translation researcher: GTX 2080 Ti with the blower fan design. For very large network, go for RTX Titans.
- If you are an NLP researcher: RTX 2080 Ti 16-bit.
- If you want to build a GPU cluster—this is really complicated, you can get some ideas from my multi-GPU blog post.
- If you have started deep learning and are serious about it: RTX 2070.
- If you want to try deep learning, but are not serious about it: GTX 1050 Ti (4 or 2GB). This often fits into your standard desktop and does not require a new PSU. If it fits, do not buy a new computer!
- If you started deep learning and are serious about it: Start with a GTX 1060 (6GB).
- If you want to try deep learning, but are not serious about it: GTX 1050 Ti (4 or 2GB).
- The other frame to consider while buying a GPU is its fitment for common problems.

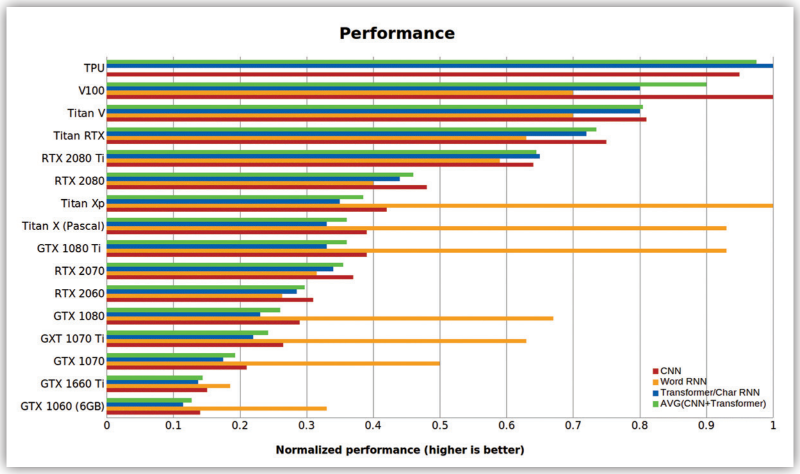


Figure 1.8: *Normalized performance data of GPUs and TPU. Higher is better.*
Credit - *https://timdettmers.com/2019/04/03/which-gpu-for-deep-learning/*

Here are a few takeaways based on the above chart (Credit: https://lambdalabs.com/blog/choosing-a-gpu-for-deep-learning/)

- Buy large memory GPU if your focus is on language models. This is because language models are more memory-bound and image models are more computationally-bound.
- GPUs with higher VRAM have better performance because using larger batch sizes helps saturate the CUDA cores.
- GPUs with higher VRAM enable proportionally larger batch sizes. Back-of-the-envelope calculations yield reasonable results: GPUs with 24 GB of VRAM can fit ~3x larger batches than a GPU with 8 GB of VRAM.
- Language models are disproportionately memory intensive for long sequences because attention is quadratic to the sequence length.
- Cost efficiency analysis is another important frame for selecting a GPU:

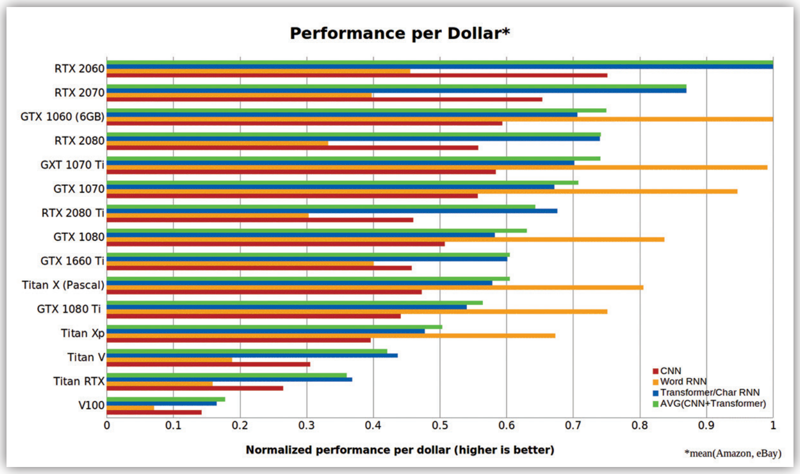
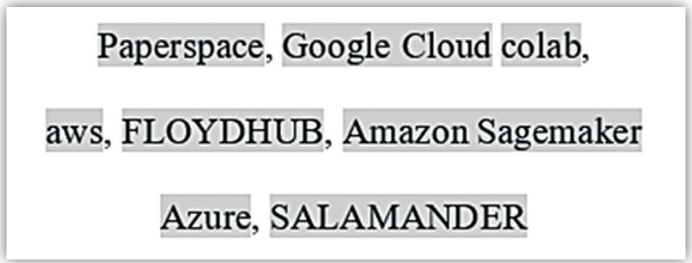


Figure 1.9: *Normalized performance/cost numbers for convolutional networks (CNN), recurrent networks (RNN), and transformers. Higher is better.*

Credit - https://timdettmers.com/2019/04/03/which-gpu-for-deep-learning/

It's no surprise that if budget is not an issue, then the choice is much easier. Otherwise, pick the GPU which has better price-to-performance ratio. In case you are not ready to invest upfront in a new GPU, fortunately, these are some of the popular companies from whom you can rent cloud GPU instances. All these companies have good availability of Nvidia-powered GPU instances:


Paperspace, Google Cloud colab,

aws, FLOYDHUB, Amazon Sagemaker

Azure, SALAMANDER

Figure 1.10: *Cloud GPU options*

GPU instances on AWS/Azure and TPU on Google are viable alternatives. TPU should be preferred for image-related tasks and transformers. For other task types, the cloud GPU should work fine. The other aspect that you need to consider for cloud GPU is framework lock-in. TensorFlow is the reliable choice for TPU but that means switching over to PyTorch may not be that straightforward.

For starters, some free GPU computation services are also available using platforms like Google Colab. To keep the cost low on cloud GPU instances, you can consider—what Amazon calls—spot instances and Google—preemptible instances.

Spot-instance or preemptible instance pricing makes high-performance GPUs much more affordable for deep learning. These instances are at excess capacity and can be taken away when the demands surges. If your training process can withstand possible instance pre-emptions, then spot instances can reduce the costs significantly. For more detailed comparison of various cloud GPU options, I encourage you to check online.

Now that you have the knowledge of different frames to decide between cloud or local GPU, let us switch to a related topic.

1.3 On-premises vs cloud

If you are starting out your data science project, you need infrastructure for many of your needs ranging from training to deployment. That infrastructure could be built by you or your team on-premises or in cloud. For example, AWS, Azure and Google cloud, etc.

So, how do you decide which ones to use? Let us begin with looking at the benefits of cloud infrastructure:

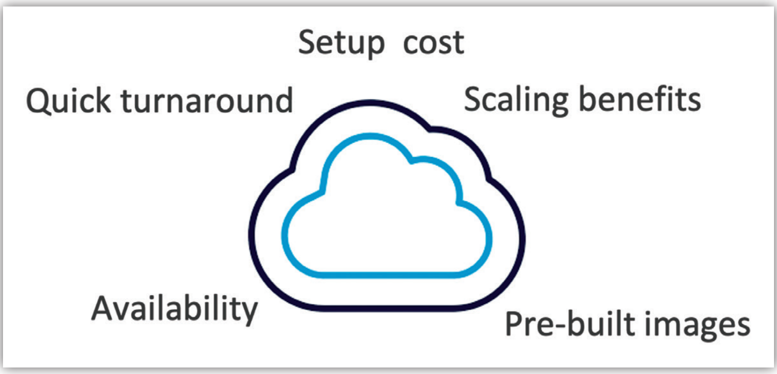


Figure 1.11: *Cloud benefits*

Let's learn about cloud benefits:

- One of the top benefits of using cloud infrastructure over on-premises hardware is quick turnaround and setup cost.
- Consider a case where you quickly want to train or experiment a deep learning network. With cloud infrastructure, you can just quickly spawn a couple of GPU instances with per hour billing. There is no need to order or build a machine or chase your IT guys for making the machine as per your specification.
- The other major benefits of cloud infrastructure is quick horizontal or vertical scaling. You can increase the number or size of your compute infrastructure with a few clicks or even make it auto scale.
- The other major benefits of cloud infrastructure are software configuration. With pre-built software images like Amazon AMI, you can get most of the software already optimally configured. For example, if you want to use TensorFlow and PyTorch, you don't need to manually install these libraries. You can just use any of the pre-built machine images containing all major software and dependencies already installed. You would agree that this is a major productivity boon.
- Last but not the least, by using cloud infrastructure, you ensure much better availability of your infrastructure.

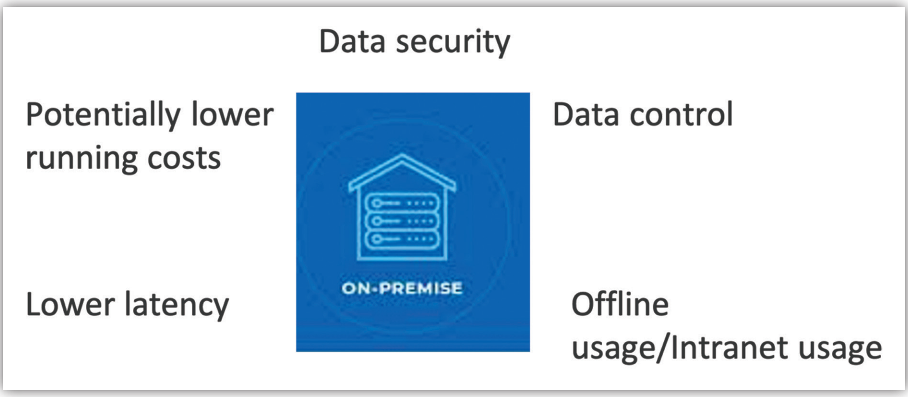


Figure 1.12: *On-premises benefits*

Let us now shift our attention to the on-premises option and explore some of its benefits and use cases:

- First and foremost, if you are a heavy user of infrastructure and if you or your organization has the necessary skills to deploy on-premise infrastructure then, on-premise provides increased opportunity to lower the costs. For example, if you are doing very frequent deep learning trainings and testing, you can rack thousands of dollars of fees with cloud GPU instances. By using on-premises consumer grade GPU, however, you can save money for the long term.
- Another major benefit of using on-premise infrastructure is better data security and control.
- Since your data does not leave your intranet or your machine, you have a little less to worry about. For highly sensitive use cases, you can even train your ML model on fully offline machines as well.
- On-premises is the most acceptable option if your organization or clients have a very strict policy of data control or physical security
- Another plausible use case for on-premise infrastructure is lower latency. If you are working on use cases like real time object detection on streaming data, real time speech processing, and so on; by using-on premise infrastructure, you can ensure lower latency.
- Last but not the least, by using on-premise infrastructure, you can cater to use cases in which is no internet access or in cases where you need to deploy ML application in the company network only.
- So, how do you decide between on-premise and cloud? Go for cloud when:
- You need minimum infrastructure management hassle.

- You have low usage needs. For example, you need a high-end machine for training for a limited period.
- You are on deadline and need to quickly experiment different architectures.
- Your model has a huge number of parameters to train.
- You know that you will have to add more power or scale quickly. While the pre-built or self-built machines are cheap compared to cloud builds, it is very difficult to maintain after a certain point. Clustering, monitoring, and workflow management get too unmanageable in on-premise options.

Go for on-premises when:

- You have client requirements for strict policy around data sharing and control.
- You have dedicated people who can manage infrastructure and software issues. On-premises saves cost in the long term.
- You have frequent long-running tasks.
- Your systems have real-time data needs.

By the way, it must be no surprise that going the hybrid mode is also a reasonable option. This is generally practiced in a mature and heavy usage team. Here, you have some on-premises dedicated hardware for running non-priority batch jobs for your experimentations and then cloud is used for cases when you need quick ramp up of infrastructure.

For example, use cloud for hyperparameter search experiments and then use them in local (on-premises training). To set up on-premises machines, building a machine with 4 GPU is not that difficult and is the cheapest as well. There are very good resources on the internet to setup your deep learning box. Search for "how to build a deep learning rig." You also have the option of buying pre-built machines from Lambda labs and NVIDIA. While NVIDIA machines are expensive, Lambda labs costs are typically 20% more than a self-built machine and that is not a bad deal given that they pre-install most of the popular frameworks. With the hardware taken care of, it is now time to think about software.

1.4 Deciding your framework

Compared to the previous decades, we now have a large number of tools and libraries available to choose from. We have TensorFlow and PyTorch as the two most popular frameworks for deep learning.

In the language space, we have options like Python, R, MATLAB, and C++. For domain specific use cases, we have many frameworks like NLTK, Spacy, Genism SciKit-learn, etc. The following image shows some of the popular choices. You can debate on why something is included or not included but that is beside the point. The problem we are after is that with all these choices available, how do we choose right tool for the task?

Figure 1.13: *Popular open-source tools for ML and DL.*

Let us start with the language tool first. R and Python are the two most popular programming languages for data analysis and we will choose one of them. R is a statistical analysis tool and was developed in 1992. It was the preferred language for most data scientists for years. Python was developed in 1989 as a general-purpose programming tool with a focus on readability and efficiency.

To make a decision, let us consider a few questions. These are as follows:

1. Which language has better toolsets for all the different steps of a data science pipeline?
2. Python being a generic programming language has robust ecosystems for all the steps in the pipeline.
3. Which one has better support for production situations?
4. Python being a full-fledged programming language with a tested, reliable, and powerful ecosystem is the best choice.
5. Which one has better support for software engineering?
6. Again, being a general-purpose language, Python is better-suited for integration with different components.

I believe that by now you must already know which is my recommendation.

Let us now look at deep learning frameworks. For our purpose, let us evaluate deep learning frameworks on two properties:

- Good for development - How easy is it to develop and debug?
- Good for production - How easy is it to scale?

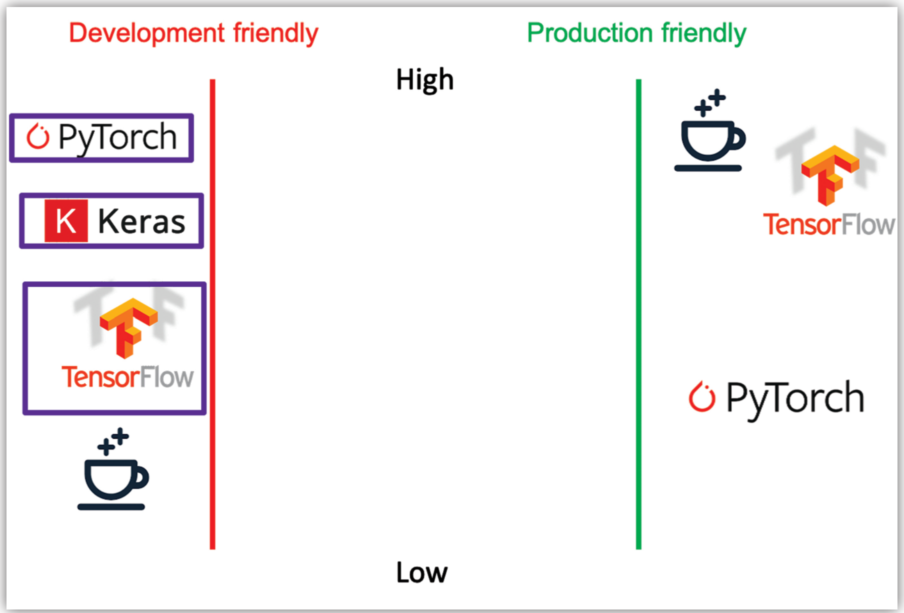


Figure 1.14: *Framework comparison. High is good.*

Caffe was the first one to receive industry adoption. Built in C++, it has speed and performance. For this reason, it is still a reasonable choice for situations that have real-time result need.

With Google investment, TensorFlow is also a good choice for production. It really enables what we call "build once and deploy everywhere." You can deploy the TensorFlow model on servers, laptops, and now mobiles as well. But again, it has a little steep learning curve, making it not the optimum choice during development. The code we write in TensorFlow gets converted into a computation graph to run efficiently on server, making it harder to debug.

However, with Keras, the developer experience has improved a lot. And the best part is that it is good for production as well because what we write in Keras gets converted into the TensorFlow graph.

PyTorch is a recent one from Facebook and has become quite popular in a short span of time. Its style is Pythonic, making it very developer friendly. My recommendation would be to choose between Keras/TensorFlow and PyTorch.

With TensorFlow 2.0's release, due to eager mode execution, Keras TensorFlow behaves quite like PyTorch. With caffe2, PyTorch models could be compiled in an optimized format to run efficiently on different environments. We will stick to TensorFlow in this book.

Given Python as our language choice, in the libraries space, a numerical analysis package like NumPy, data processing package like Pandas, and visualization library Matplotlib are pretty much the default choices. The following image has some of the most popular ones. This list is in no way complete but it is good enough for you to be familiar with and replace it with newer alternatives as you gain more confidence.

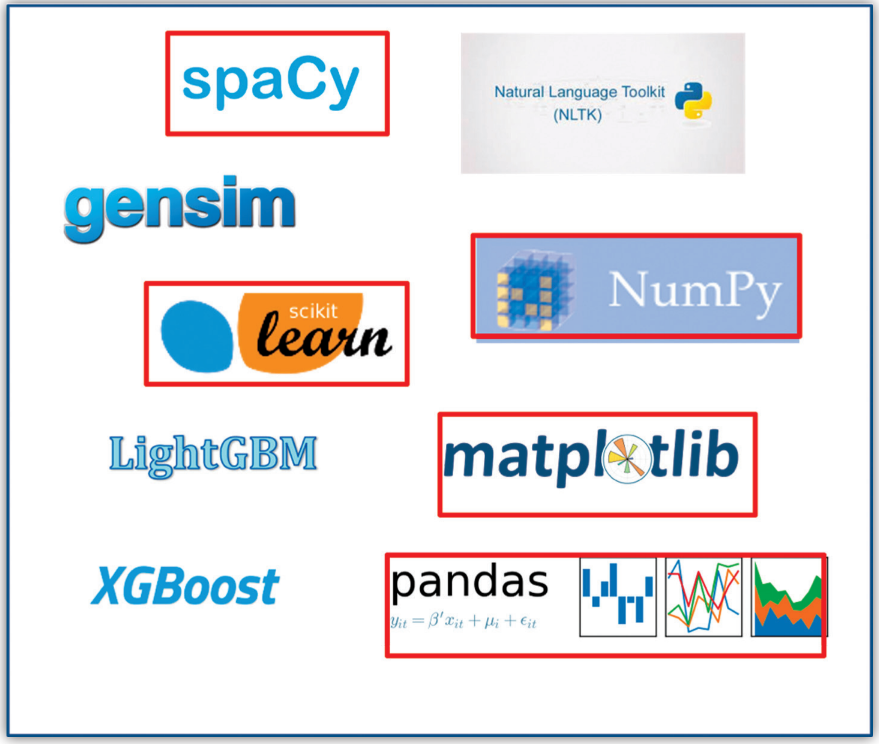

spaCy
Natural Language Toolkit (NLTK)
gensim
NumPy
scikit learn
LightGBM
matplotlib
XGBoost
pandas
$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$

Figure 1.15: *Task specific ML libraries*

The purpose of the libraries are as follows:

Library	Purpose
NLTK	NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, and wrappers for industrial-strength NLP libraries. https://www.nltk.org/

spaCy	spaCy is an open-source software library for advanced natural language processing, written in the programming languages Python and Cython. It features NER, POS tagging, dependency parsing, word vectors, etc. NLTK is mostly used for research and experimentations, given the plethora of algorithms available in it. Whereas, spaCy's focus area is on the production use cases where the performance is important. It is not at all appropriate to compare two libraries with a single statement. I encourage you to check out the spaCy documentation for details: (https://spacy.io/). In my experience, it is okay to have spaCy as your default choice for NLP tasks and then change it if required.
Gensim	Gensim is an open-source library for unsupervised topic modeling and natural language processing by using modern statistical machine learning. It is implemented in Python and Cython. It is designed to handle large text collections using data streaming and incremental online algorithms, which differentiates it from most other machine learning software packages that target only in-memory processing. For topic modelling related tasks, Gensim is a reasonable default choice. https://radimrehurek.com/gensim/
Scikit-learn	Scikit-learn (formerly scikits.learn and also known as sklearn) is a free software machine learning library. It features various classification, regression, and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy. It is a good tool to build your baselines (more on baseline later). https://scikit-learn.org/stable/
LightGBM	LightGBM is a gradient boosting framework that uses tree-based learning algorithms. Gradient boosting is a powerful ensemble machine learning algorithm. It is popular for structured predictive modelling problems, such as classification and regression on tabular data, and is often the main algorithm or one of the main algorithms used in winning solutions to machine learning competitions, like those on Kaggle. https://lightgbm.readthedocs.io/en/latest/

XGBoost	XGBoost is an open-source software library which provides a gradient boosting framework for C++, Java, Python, R, Julia, Perl, and Scala. XGBoost is also a decision tree-based ensemble machine learning algorithm that uses a gradient boosting framework. When it comes to small-to-medium structured/tabular data, decision tree-based algorithms are considered best-in-class right now. https://xgboost.readthedocs.io/en/latest/
Numpy	NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. If you have any experience with ML and Python, then you must know Numpy.
Matplotlib	Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications by using general-purpose GUI toolkits. While there are other choices as well for visualization, people generally start from Matplotlib and move to others depending on the project requirements. https://matplotlib.org/
Pandas	Pandas is a Python library for data manipulation and analysis. It offers robust data structures and operations for manipulating numerical tables and time-series. https://pandas.pydata.org/

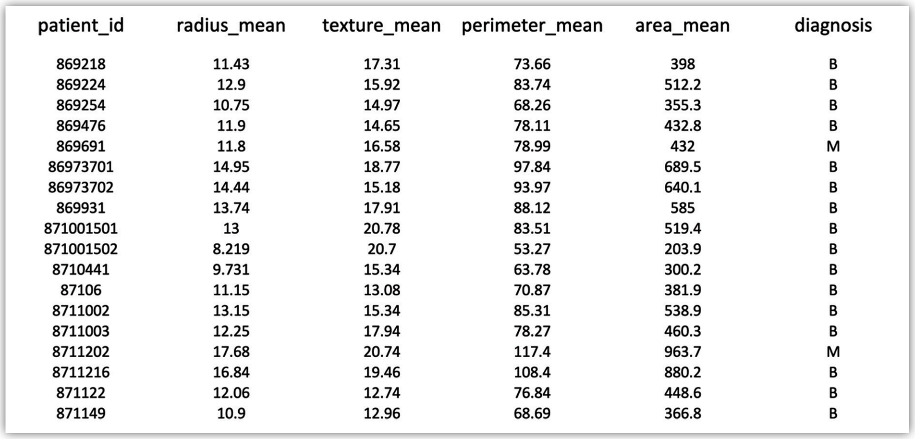
Table 1.3: *Popular ML libraries*

Don't feel overwhelmed if you don't know some of these libraries. We will cover a few of them in detail later. The first step in knowing is to know "*what*" is out there and the table given above answers that question . So far, you have learnt the techniques to decide on your project structure, infrastructure, framework, and libraries but the project setup is not done yet. Now, we will look at a few techniques to decide on the project targets (metrics).

1.5 Deciding your targets

For any data science project, you have a lot of different metrics available to ascertain your trained model behavior; for example, accuracy, precision, recall, IOU (for object detection), and work error rate (for speech recognition), etc. Starting a data science project without choosing the right metric for measurement is like throwing a dart blindfolded and hoping it will hit the right target.

Let us look at this dataset image from Kaggle; the problem statement here is breast cancer detection:



patient_id	radius_mean	texture_mean	perimeter_mean	area_mean	diagnosis
869218	11.43	17.31	73.66	398	B
869224	12.9	15.92	83.74	512.2	B
869254	10.75	14.97	68.26	355.3	B
869476	11.9	14.65	78.11	432.8	B
869691	11.8	16.58	78.99	432	M
86973701	14.95	18.77	97.84	689.5	B
86973702	14.44	15.18	93.97	640.1	B
869931	13.74	17.91	88.12	585	B
871001501	13	20.78	83.51	519.4	B
871001502	8.219	20.7	53.27	203.9	B
8710441	9.731	15.34	63.78	300.2	B
87106	11.15	13.08	70.87	381.9	B
8711002	13.15	15.34	85.31	538.9	B
8711003	12.25	17.94	78.27	460.3	B
8711202	17.68	20.74	117.4	963.7	M
8711216	16.84	19.46	108.4	880.2	B
871122	12.06	12.74	76.84	448.6	B
871149	10.9	12.96	68.69	366.8	B

Figure 1.16: *Sample dataset*

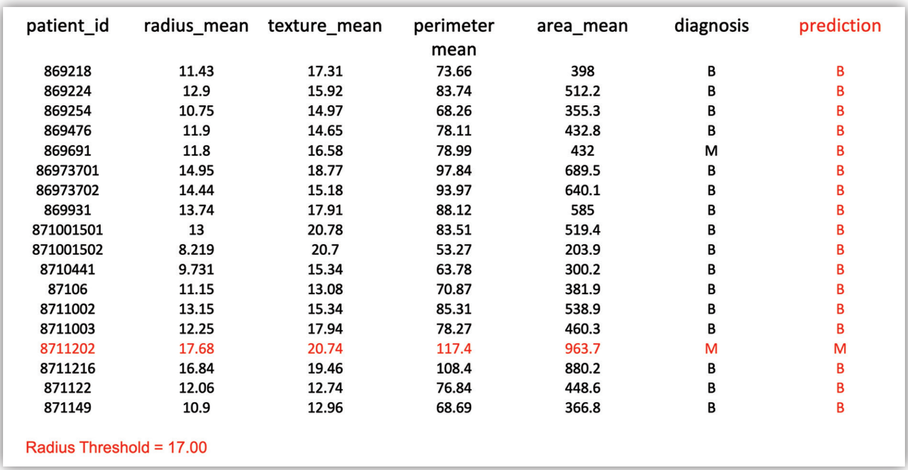
Source Kaggle: *https://www.kaggle.com/uciml/breast-cancer-wisconsin-data*

In this dataset, different objective parameters of the tumor have been measured. Tumors are classified into two categories - benign and malignant. Benign tumors are completely harmless, indicating negative diagnosis of breast cancer. Malignant tumors, on the other hand, are cancerous tumors indicating positive cancer diagnosis.

We have on the first column, patient ID, followed by multiple parameters about tumors. (Note that I have skipped a lot of parameters which are present in the actual dataset for brevity). The final column gives the actual diagnosis for each tumor.

Let's assume that you have been hired by a medical diagnosis company to build an ML model that would get deployed in a diagnostic machine. The machine would automatically scan for these parameters and then use your model to predict whether a person has breast cancer or not. As the starting point, you can take a quick look and pick one of the parameters like Radius mean.

You can decide to put a threshold of around 17 here, so if the Radius mean is greater than 17, you can give the diagnosis of malignant tumor, otherwise benign tumor.



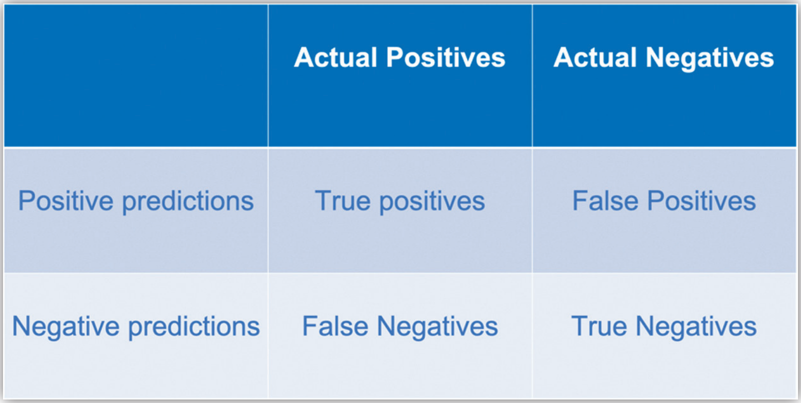
patient_id	radius_mean	texture_mean	perimeter mean	area_mean	diagnosis	prediction
869218	11.43	17.31	73.66	398	B	B
869224	12.9	15.92	83.74	512.2	B	B
869254	10.75	14.97	68.26	355.3	B	B
869476	11.9	14.65	78.11	432.8	B	B
869691	11.8	16.58	78.99	432	M	B
86973701	14.95	18.77	97.84	689.5	B	B
86973702	14.44	15.18	93.97	640.1	B	B
869931	13.74	17.91	88.12	585	B	B
871001501	13	20.78	83.51	519.4	B	B
871001502	8.219	20.7	53.27	203.9	B	B
8710441	9.731	15.34	63.78	300.2	B	B
87106	11.15	13.08	70.87	381.9	B	B
8711002	13.15	15.34	85.31	538.9	B	B
8711003	12.25	17.94	78.27	460.3	B	B
8711202	17.68	20.74	117.4	963.7	M	M
8711216	16.84	19.46	108.4	880.2	B	B
871122	12.06	12.74	76.84	448.6	B	B
871149	10.9	12.96	68.69	366.8	B	B

Radius Threshold = 17.00

Figure 1.17: *Radius mean parameter with a threshold.*
Source Kaggle: https://www.kaggle.com/uciml/breast-cancer-wisconsin-data

If we run this threshold model, we get some sort of prediction. The last column in *figure 1.17* represents a prediction made by our model. A helpful starting point for many of the machine learning classification problems before deciding the metric for measurement is to look at the confusion matrix.

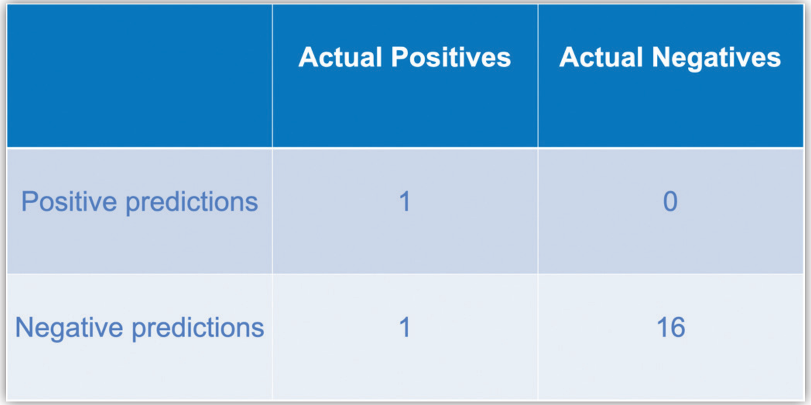
This is how the confusion matrix looks where each column represents the actual diagnosis/prediction and each row represent the values predicted by our model:



	Actual Positives	Actual Negatives
Positive predictions	True positives	False Positives
Negative predictions	False Negatives	True Negatives

Figure 1.18: *Confusion matrix*

A cross product of both will give you the confusion matrix. For example, if for any observation, the predicted and actual diagnosis are positive, then you call it to be true positive. Similarly, if the model predicted positive diagnosis when it was negative diagnosis, you call that observation false positive. For our naive model, this is the how confusion matrix looks like:

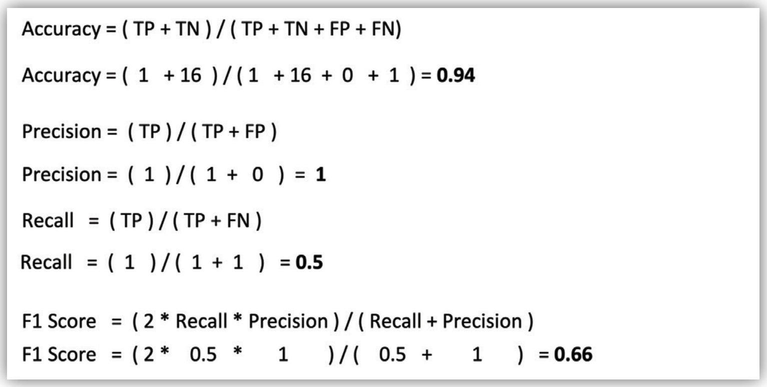


	Actual Positives	Actual Negatives
Positive predictions	1	0
Negative predictions	1	16

Figure 1.19: *Confusion matrix for the threshold based naïve model*

We have only one positive prediction predicted by our model and that prediction had an actual diagnosis as positive as well. Hence, the number of true positives is 1. Similarly, we have zero false positives, one false negative, and 16 true negatives in our predictions.

One of the most popular metrics is accuracy and its formula is shown in *figure 1.20*.



Accuracy = (TP + TN) / (TP + TN + FP + FN)

Accuracy = (1 + 16) / (1 + 16 + 0 + 1) = **0.94**

Precision = (TP) / (TP + FP)

Precision = (1) / (1 + 0) = **1**

Recall = (TP) / (TP + FN)

Recall = (1) / (1 + 1) = **0.5**

F1 Score = (2 * Recall * Precision) / (Recall + Precision)

F1 Score = (2 * 0.5 * 1) / (0.5 + 1) = **0.66**

Figure 1.20: *Metrics formula: - TP (True positive), TN (True negative), FP (false positive) and FN (False negative)*

Entering these values for our naïve model (threshold based) gives an accuracy score of 0.94. Our model is right 94% percent of the time now. Now look at another popular metric called precision which tells the exactness, or in other words, measures what proportion of the positive identification was correct. The precision formula is shown in *figure 1.20*.

Filling these values gives us the precision as 1. Looking at the accuracy and precision, you might think that we have achieved very good results. But if you are selling a device that diagnoses based on our model, it can encounter a big problem.

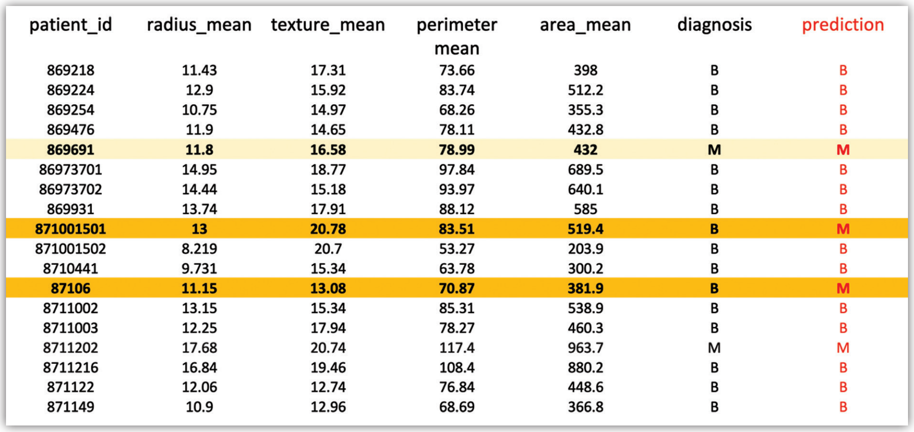
Statistically, we will achieve 100 % accuracy with a precision of 1 if we make our model classify every case positive. The machine will never miss a case but this is not practical. Every patient will be subjected to a biopsy.

Let us look at another metric.

Recall is another measure that tries to answer what proportion of actual positives were identified correctly. The recall formula and its calculated value for our naïve model is shown in *figure 1.20*.

You can clearly see that recall as 0.5 is not that good. We are missing 50% of positive diagnosis. We need to work on improving this. There is another very important metric called **F1 Score** that combines precision and recall into 1 value. F1 score measures harmonic means of precision and recall. Again, the formula and F1 score for our model is shown in *figure 1.20*.

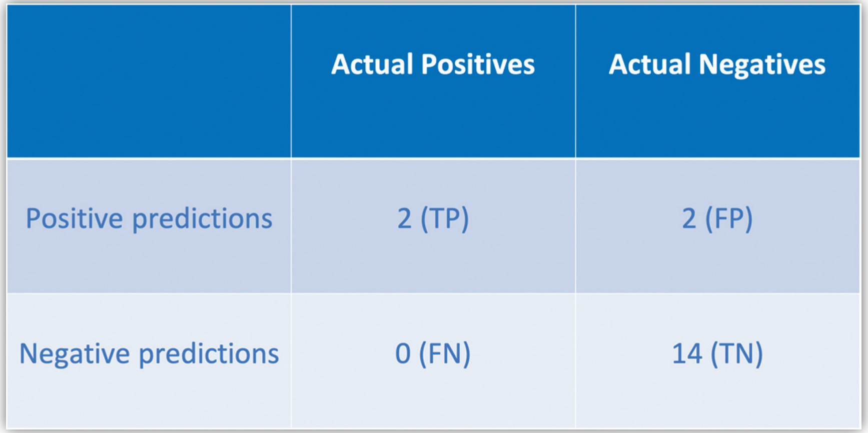
The F1 score for our case turns out to be 0.66 or 66 per cent. Now, to improve our model, let us modify it to include other parameters. Let us imagine that the new predictions look like *figure .1.21*.



patient_id	radius_mean	texture_mean	perimeter mean	area_mean	diagnosis	prediction
869218	11.43	17.31	73.66	398	B	B
869224	12.9	15.92	83.74	512.2	B	B
869254	10.75	14.97	68.26	355.3	B	B
869476	11.9	14.65	78.11	432.8	B	B
869691	**11.8**	**16.58**	**78.99**	**432**	**M**	**M**
86973701	14.95	18.77	97.84	689.5	B	B
86973702	14.44	15.18	93.97	640.1	B	B
869931	13.74	17.91	88.12	585	B	B
871001501	**13**	**20.78**	**83.51**	**519.4**	**B**	**M**
871001502	8.219	20.7	53.27	203.9	B	B
8710441	9.731	15.34	63.78	300.2	B	B
87106	**11.15**	**13.08**	**70.87**	**381.9**	**B**	**M**
8711002	13.15	15.34	85.31	538.9	B	B
8711003	12.25	17.94	78.27	460.3	B	B
8711202	17.68	20.74	117.4	963.7	M	M
8711216	16.84	19.46	108.4	880.2	B	B
871122	12.06	12.74	76.84	448.6	B	B
871149	10.9	12.96	68.69	366.8	B	B

Figure 1.21: *Post-modifications predictions.*

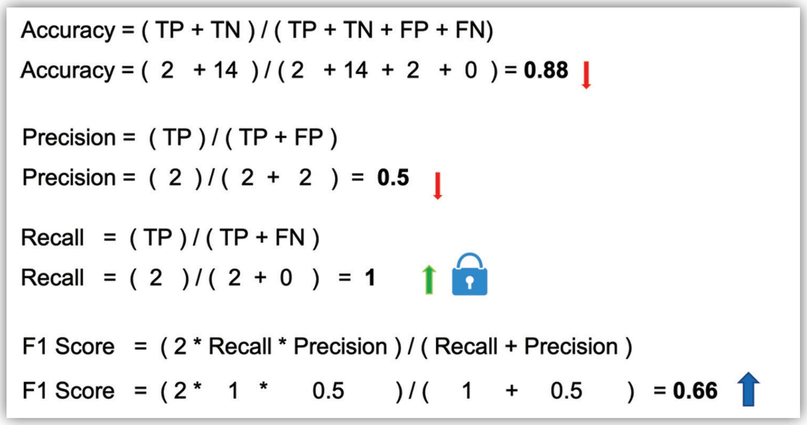
By using the new model, we can predict both the positive diagnosis correctly (`Patient_id` - 869691 and 8711002). But in doing so, our model has now started giving wrong positive diagnosis for two negative cases (`Patient_id` - 871001501 and 87106). Let us recalculate the confusion matrix:



	Actual Positives	Actual Negatives
Positive predictions	2 (TP)	2 (FP)
Negative predictions	0 (FN)	14 (TN)

Figure 1.22: *Confusion matrix*

Let us calculate the accuracy, precision, recall, and F1 score:



Accuracy = (TP + TN) / (TP + TN + FP + FN)

Accuracy = (2 + 14) / (2 + 14 + 2 + 0) = **0.88** ↓

Precision = (TP) / (TP + FP)

Precision = (2) / (2 + 2) = **0.5** ↓

Recall = (TP) / (TP + FN)

Recall = (2) / (2 + 0) = **1** ↑ 🔒

F1 Score = (2 * Recall * Precision) / (Recall + Precision)

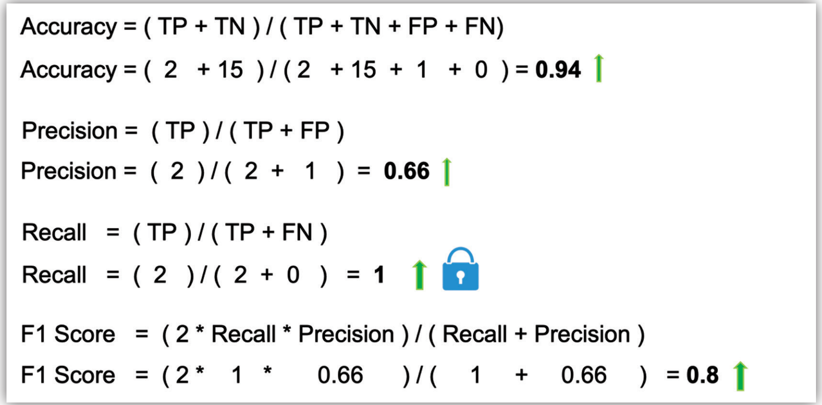
F1 Score = (2 * 1 * 0.5) / (1 + 0.5) = **0.66** ↑

Figure 1.23: *Metrics values calculation*

As you can see, the accuracy of this model has decreased drastically. Similarly, the model precision has also reduced by almost half to 50%. But still, this could be a better result if you are that medical diagnosis company. You are not missing positive diagnosis at least!

For our diagnosis machine, precision and recall are important metrics as our targets. Just to be clear here, a bad precision means people being diagnosed with cancer without having it. In this case, the doctor may do multiple tests to be sure. But still, you would agree that it is better than missing actual cancer cases.

Let us try to improve our model further, and for that, let us lock down recall and then try to improve precision.



Accuracy = (TP + TN) / (TP + TN + FP + FN)

Accuracy = (2 + 15) / (2 + 15 + 1 + 0) = **0.94** ↑

Precision = (TP) / (TP + FP)

Precision = (2) / (2 + 1) = **0.66** ↑

Recall = (TP) / (TP + FN)

Recall = (2) / (2 + 0) = **1** ↑ 🔒

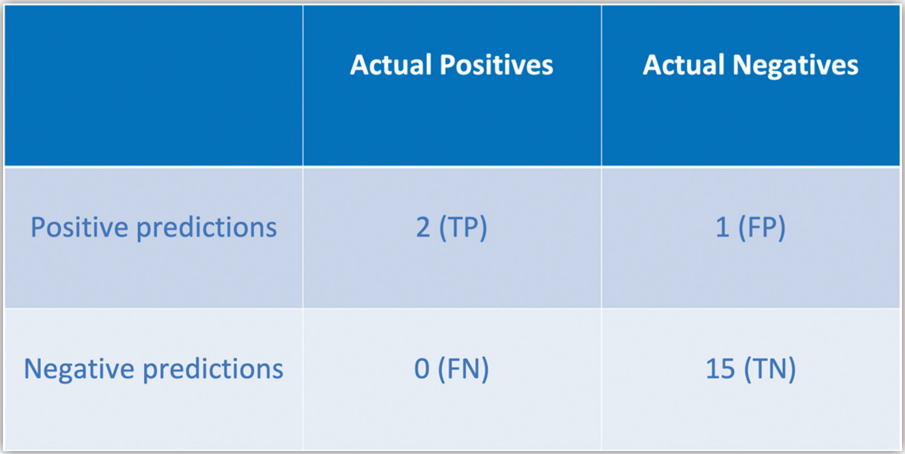
F1 Score = (2 * Recall * Precision) / (Recall + Precision)

F1 Score = (2 * 1 * 0.66) / (1 + 0.66) = **0.8** ↑

Figure 1.24: *Metrics calculation*

F1 score is the same as before. We need to increase the F1 score without decreasing recall (locked). Let us assume that somehow we manage to improve our model and now we have one less missed diagnosis.

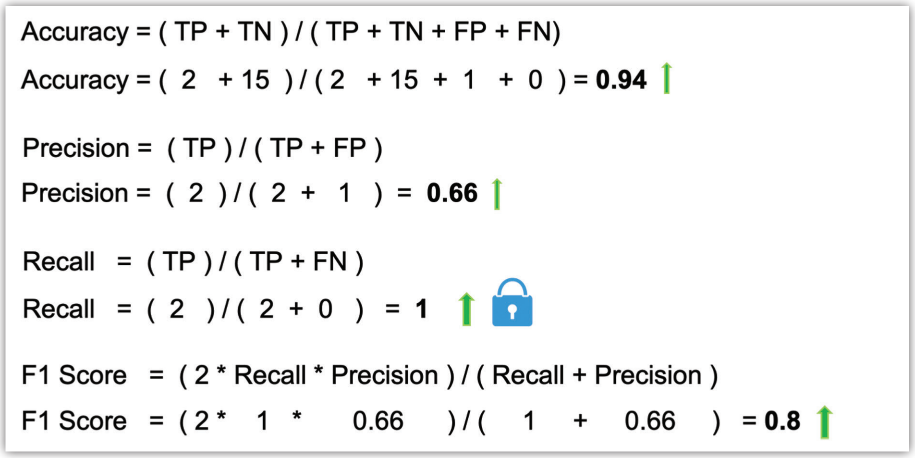
This is how the updated confusion matrix looks now:



	Actual Positives	Actual Negatives
Positive predictions	2 (TP)	1 (FP)
Negative predictions	0 (FN)	15 (TN)

Figure 1.25: *Confusion matrix (reduced FP)*

Let us recalculate the metrics by using *figure 1.25* values:



Accuracy = (TP + TN) / (TP + TN + FP + FN)

Accuracy = (2 + 15) / (2 + 15 + 1 + 0) = **0.94** ↑

Precision = (TP) / (TP + FP)

Precision = (2) / (2 + 1) = **0.66** ↑

Recall = (TP) / (TP + FN)

Recall = (2) / (2 + 0) = **1** ↑ 🔒

F1 Score = (2 * Recall * Precision) / (Recall + Precision)

F1 Score = (2 * 1 * 0.66) / (1 + 0.66) = **0.8** ↑

Figure 1.26: *Recalculated metrics.*

So now, we can improve accuracy and precision without reducing recall. New the F1 score of our model is 0.8 which is significantly better than before. It is evident that that for our problem at hand, recall and F1 score are better metrics to rely on compared to accuracy or precision.

The key takeaway here is to not blindly follow a single metric. Check different metrics with their pros and cons and then decide the right one for your product.

1.6 Preparing baseline

A baseline provides a point of reference to compare models that you construct. Technically speaking, a baseline is a method that uses heuristics, simple summary statistics, randomness, or machine learning to create predictions for a dataset.

You can use these predictions to measure the baseline's performance (for example, accuracy). Another way to think about it is that baselines give you a lower bound on expected the model performance.

But the questions is - why is baselining important? In order to understand the importance of baseline, let us look at *figure 1.27*:

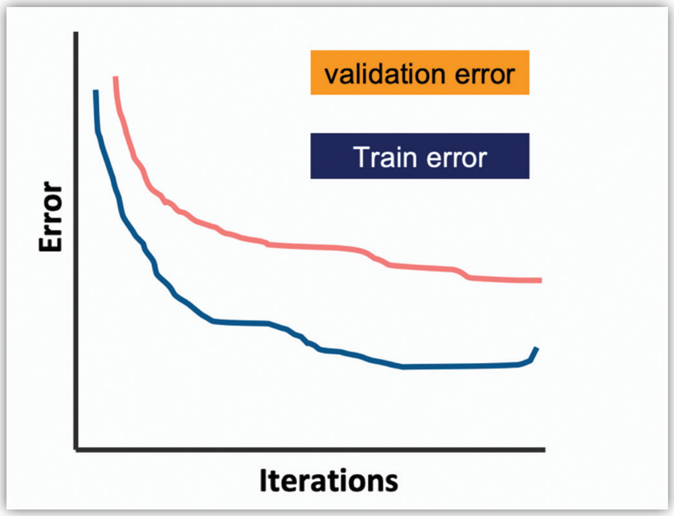


Figure 1.27: Error vs iteration graph without baseline

Now, let me ask a question - Is this a good model? What should we do next?

If you are feeling confused then I would concur that it is indeed confusing. We don't know if it is good or bad and what should we do next. Let us now add a baseline:

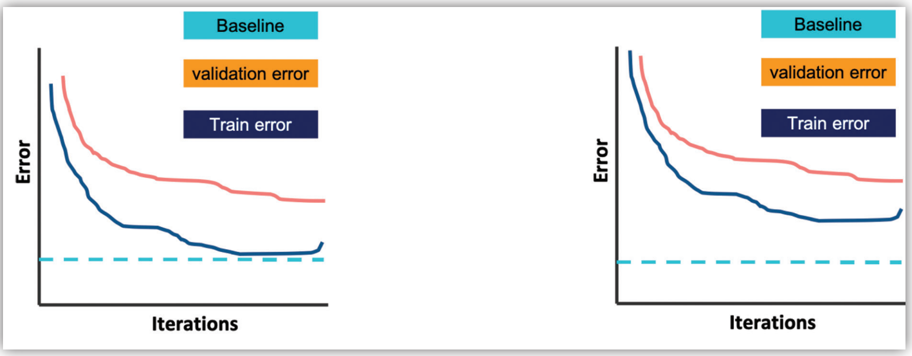


Figure 1.28: Error vs iteration's graph with different baselines

Let us assume that the dotted line in the left-hand side picture is the baseline (let us say that error level of human for a task). We can now clearly see that while the training error is almost as good as human performance level, the validation is quite off from the baseline. This clearly seems to be a case of training overfitting and we might decide to add more data for training. With the baseline, it became quite clear as to what to do next.

Let us shift focus to the right-hand side picture of *figure 1.28*. As we can clearly see, both the training and testing error are quite off from baseline. The next step here could be to try different architecture or train for longer. Again, with baseline, it becomes quite clear as to what to do next.

Without a proper baseline, you don't know if you are making any progress. Now that we understand the value of baselining, let us look at the different baselining techniques.

There are generally two ways to prepare the baseline:

- Collect them
    - A lot of the time, the business shares this information as part of the requirements.
    - It helps to look at the state of the art or the published result in the similar domain. For example, if you are building a recommender, then it is beneficial to know what performance levels the others were able to achieve. Do note that this may not be an apple-to-apple comparison as your data distribution could be different from the published version.
- Extract them
    - It can be done via custom programs. For example, we write an OpenCV app to detect humans in a photo using hardcoded rules. They may not be perfect, but are still a good start point.
    - We could build Simple ML models; for example:
        - Standard models with bag-of-words classifier for NLP problem.
        - We can use the results from linear classifier/regression.
        - We can build a basic neural network; for example, VGG without any batch normalization
    - We can use scikit-learn dummy estimators.

Let us understand scikit-learn dummy estimators a little more. They are quite useful but are largely unknown. The dummy estimator gives you a measure of the "*baseline*" performance, that is, the success rate that one should expect to achieve even if it's simply guessing.

Depending upon the problem type, you can use dummy regressor or classifier. Here are some simple strategies for classification. Please refer to this documentation for full details: (https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html)

- Stratified generates random predictions by respecting the training set class distribution.
- `most_frequent` always predicts the most frequent label in the training set.

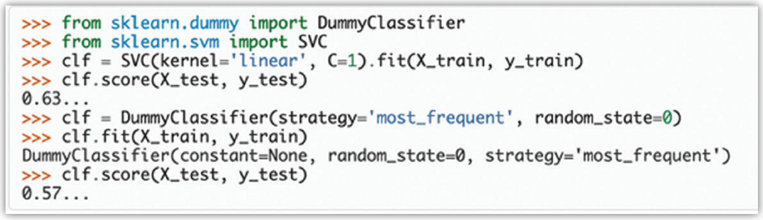
- Prior always predicts the class that maximizes the class prior (like **`most_frequent`**) and **`predict_proba`** returns the class prior.
- Uniform generates predictions uniformly at random.
- Constant always predicts a constant label that is provided by the user.

DummyRegressor also implements four simple rules of thumb for regression. Here are a few simple rules for regression. Please refer to the documentation for full details:

(https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyRegressor.html)

- Mean always predicts the mean of the training targets.
- Median always predicts the median of the training targets.
- Quantile always predicts a user-provided quantile of the training targets.
- Constant always predicts a constant value that is provided by the user.

Using the dummy classifier is quite easy. Here, we are comparing SVC and the dummy classifier **`most_frequent`** strategy.



```
>>> from sklearn.dummy import DummyClassifier
>>> from sklearn.svm import SVC
>>> clf = SVC(kernel='linear', C=1).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.63...
>>> clf = DummyClassifier(strategy='most_frequent', random_state=0)
>>> clf.fit(X_train, y_train)
DummyClassifier(constant=None, random_state=0, strategy='most_frequent')
>>> clf.score(X_test, y_test)
0.57...
```

Code 1.1: *Dummy classifier with the most frequent strategy*

Next, we will look at tools for effective workflow management.

1.7 Managing workflow

You would have surely felt frustrated when:

- You were unable to reproduce past results.
- You were unable to figure out why a program was working or not working.
- You were unable to recall what options and tricks you had tried earlier that had worked.

All these pain points boil down to the need for an effective experiment management system. But how does an experiment management system address the above problems?

Before we highlight the value of an effective experiment management, let us be clear on the connection between ML and experimentation. The machine learning workflow is inherently experimental and iterative. We run a bunch of experiments, trying different architectures, loss functions, optimizers, hyper parameters, sampling methods, and so on. In the absence of a system, things quickly get out of hand.

By the way, if you are thinking of Git, then do realize that Git manages the code version and not the relationship between the code and the respective results. An effective workflow management system must allow the following:

- Keeping a track of all the parameters of an experiment.
- Quick running of the experiment with different settings.
- Enabling us to save configurations of individual runs.
- Reproducing past results.

While the toolset space is still evolving, we have open-source options available. At an uber level, there are two common types of tasks for an effective workflow management: log the details and then visualize the details:

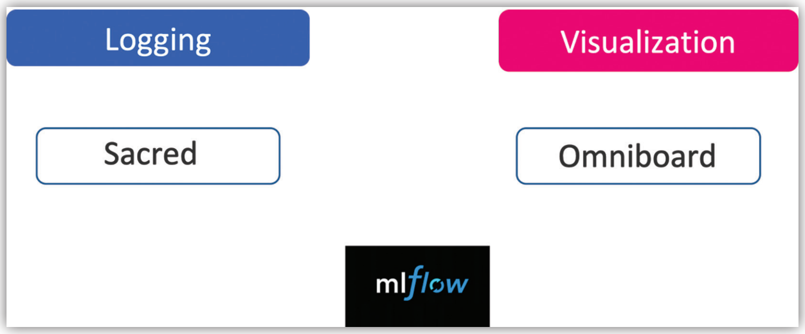


Figure 1.29: *Workflow management tools and framework*

From this figure, we learn the following:

- Sacred is a tool to configure, organize, log, and reproduce computational experiments.
- Omniboard is a web dashboard build exclusively for sacred.
- MLflow is an open-source platform from databricks for managing the end-to-end machine learning lifecycle.

Let us start with Sacred. The ability to conveniently make experiments configurable is at the heart of Sacred. If the parameters of an experiment are made configurable, then:

- You can maintain the parameter values separately and thus keep track of the changes made to it.
- You can easily run your experiment for different values.
- You can save configurations for individual runs in files or in a database.

It should not be a surprise that by being able to do the above three, you get the power of reliable reproducibility.

To best understand Sacred (https://sacred.readthedocs.io/), let us look at an experiment. Experiment is the central class of the sacred framework. It can be installed via PIP, or you can clone the repository from here: (https://github.com/IDSIA/sacred). Here is how the "**hello world**" of a sacred experiment would look like. This is a Python file tested with Python 3.7.3:

```
from sacred import Experiment --1


ex = Experiment() --2


@ex.config --4
def my_config(): --3
    recipient = "Sacred"
    message = "Hello %s!" % recipient


@ex.automain --5
def my_main(message):
    print(message)
```

Code 1.2: *Sacred hello world*

Let us unpack the code:

- We import the **sacred** library.
- We instantiate an experiment.
- **my_config** is a normal Python function and has two local variables - **recipients** and **message**.
- It has been decorated with a decorator defined inside Sacred and what it does - it takes the two local variables (**recipient** and **message**) and adds them to the global configurations of the experiment so that the variables are available to the other parts of the experiment.

- The decorator **`@automain`** designates a function as the starting point. Note that the local variable **`message`** from **`my_config`** has been passed to it as a parameter. Just to make sure that you understand it well, a local variable **`message`** from **`my_config`** function was made available to the function **`my_main`** and that magic was made possible by **`@config`** decorator.

If you run this script – **`python <Listing 1.2>.py`**, you would get a result like this:

```
WARNING - hello_sacred - No observers have been added to this run
INFO - hello_sacred - Running command 'my_main'
INFO - hello_sacred - Started
Hello Sacred!
INFO - hello_sacred - Completed after 0:00:00
```

Other than a few logging statements, nothing much seems to be interesting here. However, I would like to draw your attention towards the idea of making local variables available globally. What is the benefit of this? Well, think of these local variables as parameters that you would like to experiment with. Think of different learning rates, loss functions, and optimizers that you would like to try.

In case you don't know about decorators, then think of a decorator as a function that accepts another function as a parameter in order to modify that function behavior without modifying its code.

There are three different ways of adding configuration to an experiment in **`sacred`**. It is through Config Scopes, Dictionaries, and Config files. Config scopes is a regular function decorated with **`@ex.config`**. All variables from its local scope are collected and added to the configuration entries of the experiment as shown below:

```
from sacred import Experiment

ex = Experiment()

@ex.config
def my_config(): --1
    x = 5 #some integer --2
    conf = {'foo': '10', 'bar':'ten'}

@ex.main
def my_main(x, conf): --3
    print(f"x = {x}")
    print(f"foo = {conf['foo']}")

ex.run()
```

Code 1.3: *Configuration (parameters) specified via config scope.*

- **`my_config`** is a normal function and decorated with config decorator.
- It has two local variables, one integer, and another dictionary. This is no different from what we have already seen earlier with the only difference being of one parameter being a dictionary.
- We pass the local parameters to our main designated function (**`@ex.main`**).

This is the same code as *code 1.2* other than the local dictionary parameter.

Note that the functions used as a config scopes cannot contain any `return` or `yield` statements.

Configurations can be added as a dictionary using the **`ex.add_config`** method as shown below:

```
from sacred import Experiment

ex = Experiment()

ex.add_config({ --1
  'foo': 12,
  'bar': 'twelve'
})

@ex.main
def my_main(foo, bar): --2
    print(f"foo = {foo}")
    print(f"bar = {bar}")
ex.run()
```

Output:

```
WARNING - my_experiment - No observers have been added to this run
INFO - my_experiment - Running command 'my_main'
INFO - my_experiment - Started
INFO - my_experiment - Completed after 0:00:00
```

```
foo = 12
bar = twelve
```

Code 1.4: *Configuration (parameters) specified via dictionary.*

- We add a configuration entry to the experiment. Right now, it is JSON but it supports **`pickle`** and **`yaml`** as well.
- Once added, we can access these parameters anywhere. In this case, it is the main designated function.

If we can use **`json`**, then I guess the **`json`** files containing parameters should not be a surprise to you. Config files as **`json`**, **`pickle`**, or **`yaml`** constitute the third way of managing the parameters separately from the code. If we were to imagine it in terms of ML parameters, then now we have means to maintain hyperparameters as separate files. Hurray, you have a scalable way to maintain all the different combinations of hyperparameters.

Observing an experiment

So far, we have looked at different means to configure and manage parameters. These were like input but we are interested in capturing the outputs as well. We would like to observe an experiment so that the results can be analyzed and reproduced if required. By attaching an observer, we can observe all the information about the run while it is still running.

Now, there are 4 types of observers shipped with **`sacred`**.

- The Mongo observer which stores all information in a MongoDB.
- The File Storage observer stores the run information as files in each directory and will therefore only work locally.
- The TinyDB observer provides another local way of observing experiments by using TinyDB to store run information in a JSON file.
- The SQL observer connects to any SQL database and will store the relevant information there.
- And of course, you can build a custom observer.

Let us look at File observer as it is easy to setup. This observer is meant to be used during initial experiment development:

```
conf.json --1
{
  "foo": 12,
  "bar": "twelve"
}

ex = Experiment("my_observer_experiment")
ex.add_config('conf.json')

@ex.main
def my_main(foo, bar):
    print(f"foo = {foo}")
    print(f"bar = {bar}")

from sacred.observers import FileStorageObserver --2

ex.observers.append(FileStorageObserver.create(' test_runs')) --3

ex.run() --4
```

Output:

```
INFO - my_observer_experiment - Running command 'my_main'
INFO - my_observer_experiment - Started run with ID "1"
INFO - my_observer_experiment - Completed after 0:00:00
foo = 12
bar = twelve
```

On the terminal/command line, use the tree command to view the run results:

```
tree test_runs/1

test_runs/1 --5
├── config.json
├── cout.txt
├── metrics.json
└── run.json

cat test_runs/1/run.json --6

{
  "artifacts": [],
  "command": "my_main",
  "experiment": {
    "base_dir":,
    "dependencies": [
      "ipython==7.6.1",
      "numpy==1.16.4",
      "sacred==0.7.5"
    ],
    "mainfile": null,
    "name": "my_observer_experiment",
    "repositories": [],
    "sources": []
  },
  "heartbeat": "2019-12-03T18:02:29.325318",
  "host":
  },
  "meta": {
    "command": "my_main",
    "options": { --7
      "--beat_interval": null,
 "--capture": null,
 "--comment": null,
 "--debug": false,

    }
  },
  "resources": [],
  "result": null,
  "start_time": "2019-12-03T18:02:29.319136",
  "status": "COMPLETED",
  "stop_time": "2019-12-03T18:02:29.323065"
}
```

Code 1.5: *Sacred experiment observation result*

Let us understand the code:

1. Let us define our experiments with the parameters defined in a JSON file. Make sure that this JSON file lies in the same folder containing the Python file as well.
2. Let us import the **`fileStorage`** observer. Note that these are your Python script statements.
3. We need to link the file storage observer to the experiment observer object. **`Test_run`** is the folder name that will store all the results.
4. Now run the experiment. **`Python <Listing 1.5>.py`**
5. Each run is given a number, and within that numbered folder, you would find a bunch of files.
    - **`Config.json`** contains the parameters values
    - **`metrics.json`** stores metrics that you would define for your experiment; for example, accuracy, loss, etc.
    - **`run.json`** stores dependencies and a bunch of other information required to run the experiment.
6. Let us look at **`run.json`** to see some captured information. It maintains the library versions used. This helps in experiments re-running in the future.
7. We can control the logged information by adjusting the sacred options.

We have omitted a lot of details for the sake of brevity. I would strongly urge you to keep the book aside and go through the online documentation— (https://sacred.readthedocs.io/en/stable/observers.html#)—to make sure that you understand the output.

Right now, the **`config`** filename is hardcoded (**`ex.add_config('conf.json')`**), but if you replace it with the Python command line parameter, then you have a recipe of running multiple experiments in the batch mode. Imagine a script that looks like this:

```
python SVM.py config1.json
python SVM.py config2.json
...................................
```

You should try this as an exercise. Let us quickly recap what we have done so far with Sacred. We extracted the parameters from code to a separate JSON files. Then, we added observability to the algorithm. We chose the file observer, that is, whether all the output would get persisted in the local file system. Now you have the all the details – which parameter values produce which output. Isn't this amazing!

Switchin0g over to a Mongo observer is straight-forward. Change the observer to MongoObserver. You need to make sure that Mongo is up and running. Refer to this link in case you need Mongo installation steps: (https://docs.mongodb.com/manual/administration/install-community/). To run Mongo locally, you can use this guide (https://docs.mongodb.com/guides/server/install/)

```
from sacred.observers import MongoObserver
ex.observers.append(MongoObserver(url='my.server.org:27017',
                                  db_name='MY_DB'))
```

Code 1.6: *Sacred Mongo observer setup.*

It is no surprise that to view the observation results, you need to query the mongo DB **`> db.runs.find()[0]`**. We used 0 index to get the first record as it was a fresh install. Let us now go through the Sacred experiments using real ML code. We will first build an SVM-based classifier and then Keras-based MNIST classifier. In case you don't know about SVM, you may want to read this post beforehand: (https://scikit-learn.org/stable/modules/svm.html). We are only using SVM for code simplicity; feel free to try any other algorithm that you are comfortable with.

```
# ML_SVM_conf.json --1

{
    "C" : 1.0,
    "gamma" : 0.7,
    "kernel" : "rbf",
    "seed" : 42
}

#put all the below remaining code in a python file
# python version – 3.7.3
#sklearn version - '0.21.2'

from sacred import Experiment
from sacred.observers import FileStorageObserver
from sklearn import svm, datasets, model_selection

ex = Experiment("svm", interactive=True) --2

ex.observers.append(FileStorageObserver.create('ML_SVM_runs')) --3
ex.add_config('ML_SVM_conf.json') --4

@ex.capture --5
def get_model(C, gamma, kernel):
    return svm.SVC(C=C, kernel=kernel, gamma=gamma)

@ex.main  --6
def run():
    X, y = datasets.load_breast_cancer(return_X_y=True)
   X_train, X_test, y_train, y_test = model_selection.train_test_split(X, 
y, test_size=0.2)
    clf = get_model()  # Parameters are injected automatically.
    clf.fit(X_train, y_train)
    return clf.score(X_test, y_test)

ex.run() –7
```

Code 1.7: *Sacred experiment with SVM.*

Let us unpack the code.

1. Define the hyper parameters in a JSON file (**`ML_SVM_conf.json`**).
2. Instantiate the **`Experiment`** object. The interactive flag needs to be set if it is running inside Jupiter notebook.
3. Set up the file-based observer. The name of the output folder would be "**`ML_SVM_runs`**".
4. We specify the configuration file name. Again, as a best practice, avoid hardcoding and consider replacing it with a command line parameter.
5. Sacred automatically injects configuration values for captured functions. This means that whenever this function is called, its parameters will have the default values from config. It is because of this reason that a statement like **`clf = get_model()`** works.
6. Designate the starting point.
7. Run the experiment – **`python <Listing 1.7>.py`**. This has been tested with Python 3.7.3.

Once it has been run, check the different JSON files under the **`ML_SVM_runs`** folder. I will skip the details as you have already seen it before. In case you run into problems, check out the source code repository. In case you don't know about SVM, then please read about it here: (https://scikit-learn.org/stable/modules/svm.html).

```
Keras MNIST classifier
# NN_conf.json --1

{
    "batch_size" : 128,
    "num_units_first_layer" : 512,
    "num_units_second_layer" : 512,
    "dropout_first_layer" : 0.2,
    "dropout_second_layer" : 0.2,
    "learning_rate" : 0.001
}

#Listing 1.8 goes in a python file.
# Keras version - '2.2.4'
#python - Python 3.7.3

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import RMSprop
from keras.callbacks import Callback

from sacred import Experiment
from sacred.observers import MongoObserver

class LogMetrics(Callback):
def on_epoch_end(self, _, logs={}):
training_metrics(logs=logs)

ex = Experiment("NN", interactive=True)

ex.observers.append(<MongoConnection param>,db_name='db')) --2
ex.add_config('NN_conf.json')

@ex.capture
def training_metrics(_run, logs):
_run.log_scalar("loss", float(logs.get('loss')))
_run.log_scalar("acc", float(logs.get('acc')))
_run.log_scalar("val_loss", float(logs.get('val_loss')))
_run.log_scalar("val_acc", float(logs.get('val_acc')))
_run.result = float(logs.get('val_acc'))

@ex.main --3
def run(batch_size,
num_units_first_layer,
num_units_second_layer,
dropout_first_layer,
dropout_second_layer,
learning_rate):

num_classes = 10
epochs = 2

# the data, shuffled and split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
model = Sequential()
```

```
model.add(Dense(num_units_first_layer,     activation='relu',     input_
shape=(784,)))
model.add(Dropout(dropout_first_layer))
model.add(Dense(num_units_second_layer, activation='relu'))
model.add(Dropout(dropout_second_layer))
model.add(Dense(10, activation='softmax'))

model.summary()

model.compile(loss='categorical_crossentropy',
optimizer=RMSprop(lr=learning_rate),
metrics=['accuracy'])

history = model.fit(x_train, y_train,
batch_size=batch_size,
epochs=epochs,
verbose=1,
validation_data=(x_test, y_test),
callbacks=[LogMetrics()])
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
result = score[1]
return result

ex.run() --4
```

***Code 1.8:** Sacred experiment with Keras based classifier.*

1. Define the hyper parameters in a JSON file (**`NN_conf.json`**).
2. This time, we are using a Mongo-based observer. You need to replace the connection string for Mongo. In case you are new to Mongo, you can stick to in case you need Mongo installation steps he file system observer. My intent is to show you both the options.
3. The designated starting point of the script. Do recall that Sacred can automatically inject the configurable values as parameter value for the **`@ex.main`** and **`@ex.capture`** designated functions.

4. Run the experiment. `python <Code 1.8>.py`

To view the results, as expected, you need to connect to MongoDB.

Visualizing the experiments running is not so easy and that is where Omniboard (https://github.com/vivekratnavel/omniboard) comes into picture. Omniboard is a web dashboard for the Sacred machine learning experiment management tool.

It connects to the MongoDB database used by Sacred and helps in visualizing the experiments and metrics/logs collected in each experiment. Omniboard is written with React, Node.js, Express, and Bootstrap.

While you can setup all the applications separately, we would recommend using Docker. With Docker, the setup reduces to one step:

```
docker-compose up
```

`The docker-compose.yml file can be copied from here -` https://vivekratnavel.github.io/omniboard/#/quick-start?id=docker-compose

In case you don't know about Dockers, just follow the instructions from the website. We will cover Docker from the ML perspective in later chapters. Next, you need to modify your ML code and point the Mongo observer connection parameter to Mongo running inside Docker.

To summarize – we are running a Mongo DB inside a docker. The frontend dockered omniboard and the ML code observer are pointing to the same dockered Mongo.

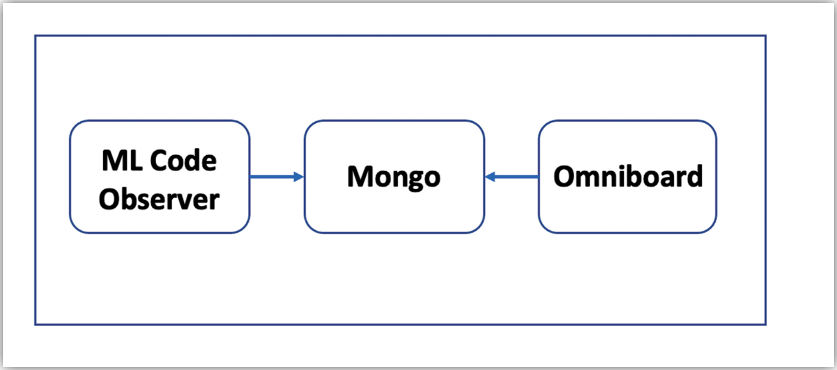
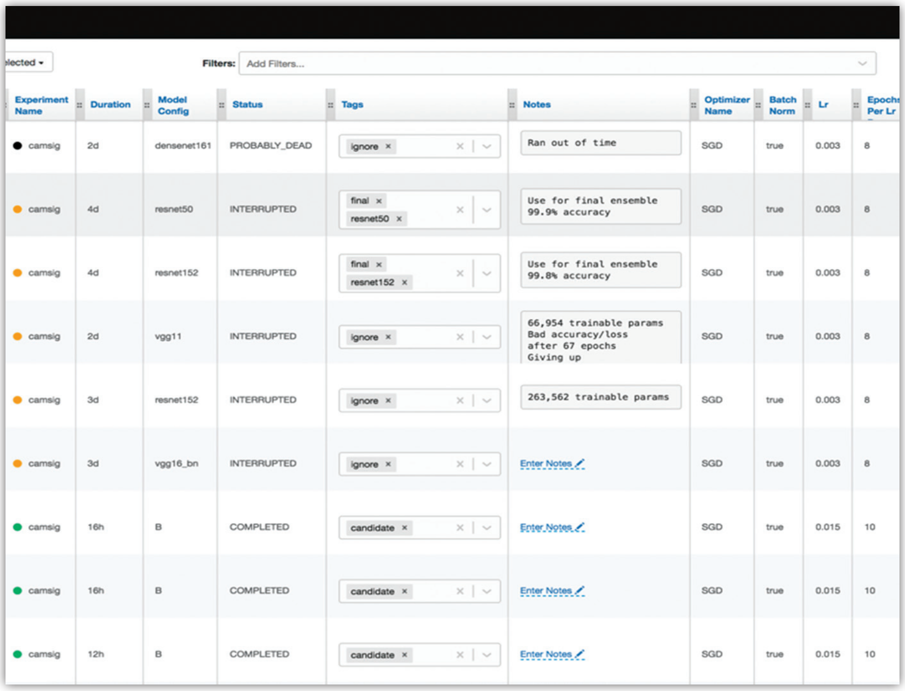


Figure 1.30: *ML and Omniboard integration*

When you run the Omniboard in the browser (Check the port for the Omniboard service in `docker-compose.yml`), you would see something like *figure 1.31*:

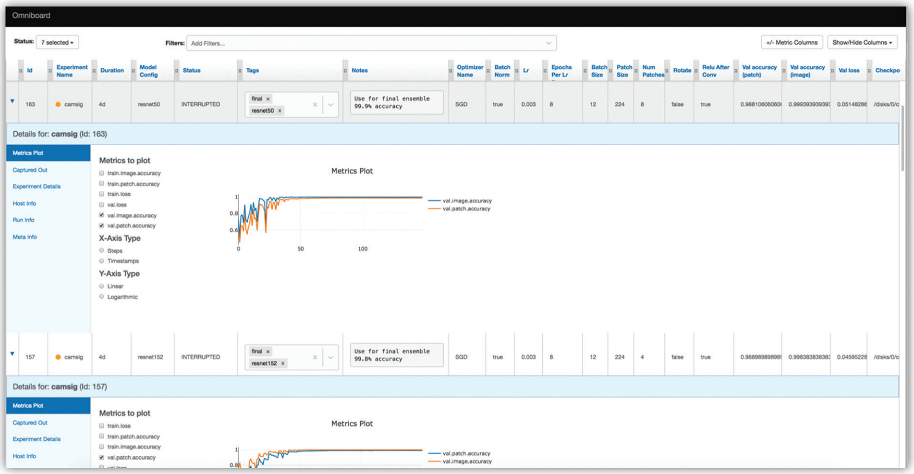


lected ▾ Filters: Add Filters...

Experiment Name	Duration	Model Config	Status	Tags	Notes	Optimizer Name	Batch Norm	Lr	Epochs Per Lr
● camsig	2d	densenet161	PROBABLY_DEAD	ignore ×	Ran out of time	SGD	true	0.003	8
● camsig	4d	resnet50	INTERRUPTED	final × resnet50 ×	Use for final ensemble 99.9% accuracy	SGD	true	0.003	8
● camsig	4d	resnet152	INTERRUPTED	final × resnet152 ×	Use for final ensemble 99.8% accuracy	SGD	true	0.003	8
● camsig	2d	vgg11	INTERRUPTED	ignore ×	66,954 trainable params Bad accuracy/loss after 67 epochs Giving up	SGD	true	0.003	8
● camsig	3d	resnet152	INTERRUPTED	ignore ×	263,562 trainable params	SGD	true	0.003	8
● camsig	3d	vgg16_bn	INTERRUPTED	ignore ×	Enter Notes	SGD	true	0.003	8
● camsig	16h	B	COMPLETED	candidate ×	Enter Notes	SGD	true	0.015	10
● camsig	16h	B	COMPLETED	candidate ×	Enter Notes	SGD	true	0.015	10
● camsig	12h	B	COMPLETED	candidate ×	Enter Notes	SGD	true	0.015	10

Figure 1.31: *Omniboard screen (Source: Omniboard documentation)*

The rows represent the multiple runs and you can deep dive into a run detail by clicking on the left column row. On expansion, the dashboard shows the run related details:



Omniboard

Status: 7 selected ▾ Filters: Add Filters... +/- Metric Columns Show/Hide Columns ▾

	Id	Experiment Name	Duration	Model Config	Status	Tags	Notes	Optimizer Name	Batch Norm	Lr	Epochs Per Lr	Batch Size	Patch Size	Num Patches	Rotate	Relu After Conv	Val accuracy (patch)	Val accuracy (image)	Val loss	Checkpo
▾	163	● camsig	4d	resnet50	INTERRUPTED	final × resnet50 ×	Use for final ensemble 99.9% accuracy	SGD	true	0.003	8	12	224	8	false	true	0.98810606060(	0.99939393939:	0.05148286	/disks/0/c

Details for: **camsig** (Id: 163)

Metrics Plot | Captured Out | Experiment Details | Host Info | Run Info | Meta Info

Metrics to plot
- ☐ train.image.accuracy
- ☐ train.patch.accuracy
- ☐ train.loss
- ☐ val.loss
- ☑ val.image.accuracy
- ☑ val.patch.accuracy

X-Axis Type
- ○ Steps
- ○ Timestamps

Y-Axis Type
- ○ Linear
- ○ Logarithmic

	Id	Experiment Name	Duration	Model Config	Status	Tags	Notes	Optimizer Name	Batch Norm	Lr	Epochs Per Lr	Batch Size	Patch Size	Num Patches	Rotate	Relu After Conv	Val accuracy (patch)	Val accuracy (image)	Val loss	Checkpo
▾	157	● camsig	4d	resnet152	INTERRUPTED	final × resnet152 ×	Use for final ensemble 99.8% accuracy	SGD	true	0.003	8	12	224	4	false	true	0.98898989898!	0.99838383838:	0.04595228	/disks/0/c

Details for: **camsig** (Id: 157)

Metrics Plot | Captured Out | Experiment Details | Host Info

Metrics to plot
- ☐ train.loss
- ☐ train.patch.accuracy
- ☐ train.image.accuracy
- ☑ val.patch.accuracy

Figure 1.32: *Omniboard expanded view (Source: Omniboard documentation)*

Understanding the report is quite easy if you remember the JSON files produced by Sacred (**`config.json`**, **`metric.json`**, etc.). In *figure 1.32*, the metric plot tab is selected and is showing the accuracy vs iteration chart. Customization is allowed and you can specify what columns to hide, etc. Any customization in terms of access and authorization need to be custom-build.

A quick summary to ensure that you understand the full picture: We extracted the hyperparameters values to a separate configuration file and logged other details via Sacred. The data was persisted in a MongoDB. We then used omniboard to visualize the data.

There are two other popular choices for workflow management:

- **MLflow** (https://mlflow.org/). We will explore MLflow in detail.
- **Kubeflow** (https://www.kubeflow.org/).

So how do you decide which one to go for? You may find the answer anti-climactic. Both serve different purposes and hence it is not a question of either "`mlflow`" or "`kubeflow`" but both. Here are a few tools use cases that would help you in seeing the point:

MLflow	• Applicable during the initial experimentation and research. • Perfect during EDA. • Perfect for managing experiments.
Kubeflow	• Applicable during deployment and model serving in production. • Perfect for managing data pipeline. • Captures the "last mile" of the pipeline.

Table 1.4: *MLflow vs Kubeflow*

MLflow is like Sacred but with more features and cloud option. You may wonder then—why did we look at Sacred at all? Well, Sacred is easier to get started and provides reasonable features for small teams (3-4 people). The moment that you are getting into multiple teams, it would be time to switch to MLflow.

We have reached the end of this chapter and we hope that you now understand what it means to get started effectively. Revision and repetition is the best way to remember the ideas for long term. I would like you to try these exercises:

1. Write a Keras-based MNIST classifier that uses MLP and ResNet network. The project structure should be similar to what we learnt earlier, and all the hyper- parameters should come from a configuration file.

2. Setup the first classifier on Google colab and train it.
3. For a gender classifier system, what metrics would you consider and why?

Conclusion

Getting started is not just about organizing your project but also deciding on the environment, framework, baseline, target metrics and the workflow management tool. The cookiecutter tool helps to setup the project code base effectively. It creates project templates that can be used across teams. We learned the need of a GPU and the various parameters to choose the right one. The Nvidia GPU should be preferred, and we explored different Nvidia options. In case you don't want to invest upfront, then there are cloud options. Preemptible instances are the cheap way to try powerful cloud GPU.

We went through different use cases to decide between on-premises and cloud. On-premises provides full control over security but is difficult to scale. Cloud is the default choice for scalability, but heavy usages can lead to hefty bills.

While there are many frameworks and languages, Python + TensorFlow + Keras is a good scalable choice. Don't blindly follow a single metric. Check different metrics with their pros and cons and then decide the right one for your product. Baselines give us a lower bound on the expected model performance. Baselines could be collected or extracted. For any serious ML work, workflow management tool is required. Logging and visualization are two common tasks of any workflow management system. Sacred and OmniBoard are good starting points. MLflow and Kubeflows are not mutually exclusive but work at different stages of the ML project.

In the next chapter, we will learn the tricks and tools for data preparation activities and strategies.

Questions

1. Which library would you use to setup a template for your DS/ML/DL project?

   Ans: cookiecutter

2. What tool can you use to track and visualize experiments?

   Ans: Sacred and OmniBoard.

3. What cloud instances can you use to keep the cost low?

   Ans: Preemptible instances

4. Which pipeline platform can you use to support your initial experimental and research?

   Ans: Kubeflow

Points to remember

- Getting started is not just about organizing your project but also deciding on the environment, framework, baseline, target metrics, and workflow management tool.
- The cookiecutter tool helps to setup the project code base effectively. It creates project templates that can be used across teams.
- The Nvidia GPU should be preferred.
- Consider cloud options if you don't want to invest upfront.
- Preemptible instances are the cheap way of trying a powerful cloud GPU.
- On-premises provides full control over the security but is difficult to scale.
- Cloud is the default choice for scalability, but heavy usage can lead to hefty bills.
- While there are many frameworks and languages, Python + TensorFlow + Keras is a good scalable choice.
- Don't blindly follow a single metric. Check different metrics considering their pros and cons and then decide the right one for your product.
- Baselines gives us a lower bound on expected model performance. Baselines can be collected or extracted.
- For any serious ML work, a workflow management tool is required. Logging and visualization are two common tasks of any workflow management system.
- Sacred and omniboard are good starting points.
- MLflow and Kubeflows are not mutually exclusive but they work at different stages of the ML project.

Further reading

- https://www.kubeflow.org/docs/components/pipelines/overview/pipelines-overview/
- https://aws.amazon.com/sagemaker/
- https://docs.docker.com/get-started/

Chapter 2
Preparing Your Data

Data collection and preparation are the foundation of trusted machine learning/deep learning models, and hence, a considerable amount of effort is spent on this step. The focus of this chapter will be on learning the best practices and tools for data analysis, and pre-processing for machine learning/deep learning projects.

Structure

In this chapter, we will learn the following topics:

- Data exploration with facets
- Missing data conundrum: Imputation techniques
- Scaling data
- Outlier treatment
- Feature engineering
- Data collection
- Large scale data processing with Dask
- Data distribution
- Training data management with Snorkel
- Data versioning

Objective

After studying this chapter, you will be able to:

- Learn the first step for data clean-up that is to know where to clean and about data exploration with the **facets tool**.
- Apply imputation techniques along with a few tools on the missing data which is a common and exciting problem in statistical analysis and machine learning.
- Implement the outlier treatment along with **AutoImpute**. An outlier is any data point which differs greatly from the rest of the observations in a dataset.
- Work with automated feature extraction using **Featuretools**. Feature extraction or engineering, also known as feature creation, is the process of constructing new features from existing data to train a machine learning model.
- Apprehend Data augmentation using **Augmentor**, which is a strategy that enables practitioners to significantly increase the diversity of data available for training models without collecting new data.
- Learn about programmatic labelling, building, and management of training data with Snorkel. It is the data that at the majority of times decides the success and failure of a ML project.
- Explore on data distribution techniques through two frames – storage and distribution.
- Understand large-scale local machine data pre-processing techniques using **Dask**. If you search for large scale data processing, the majority of the results will point to big data tools like Hadoop, Spark, and algorithms like MapReduce. However, the golden rule for working with a large dataset is to first maximize the throughput on the local machine.
- Apply data versioning using DVC, as in the context of ML, a deployed model is code plus data. While we already have the **robust toolset** for codes like GitHub, how do we version control data?

2.1 Data exploration with facets

Data, like oil, is precious but needs to be freed from impurities before it becomes useful. Inaccurate, incomplete, and biased data (biased distribution) can severely impair the model learning, leading to inaccurate and unreliable predictions. Data clean-up is not an easy task as it not only involves knowing where to clean but it needs to be done in a generalized way so that it is scalable.

Facets (https://pair-code.github.io/facets/) is an open source project from Google Research that helps in visualizing and understanding data. It can slice the data in all sorts of manners, helping see us to see how the dataset is laid out. By allowing to find out where the data doesn't quite look the way you expected it would, facets can help reduce mishaps down the road.

There are two parts of facets as follows:

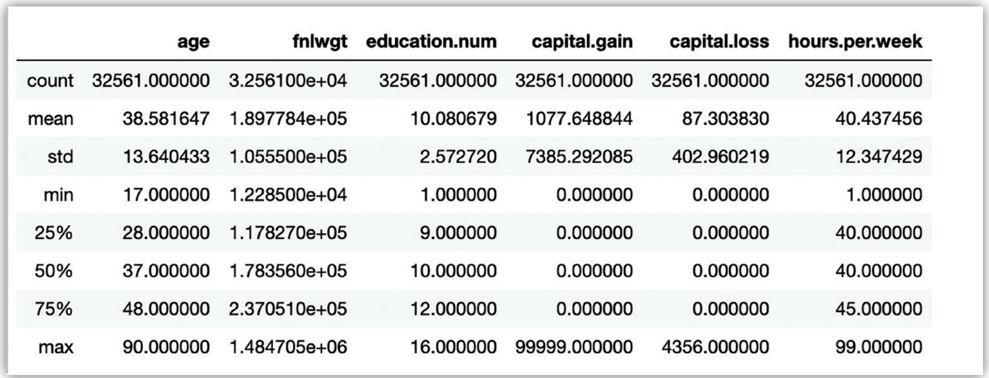
- Facets overview
- Facets dive

Overview and Dive visualizations are implemented as Polymer web components, backed by TypeScript code and can be easily embedded into Jupyter notebooks or webpages.

In order to understand facets overview, let's look at the output of the pandas dataframe.describe() function. The "describe" function displays a table of statistics about the DataFrame. This is super useful for sanity checking the dataset. Seeing if the distribution of the data looks reasonable and whether the properties are on expected lines is a great time-saver.

I have used census dataset—a classic dataset extracted from the 1994 US census by Berry Backer. Its target is to predict whether a household annual income is more or less than $50K based on various census statistics (Dataset download - https://www.kaggle.com/uciml/adult-census-income?select=adult.csv).

```
import pandas as pd
from pandas import DataFrame

df = pd.read_csv('adult.csv')
df.describe()
```

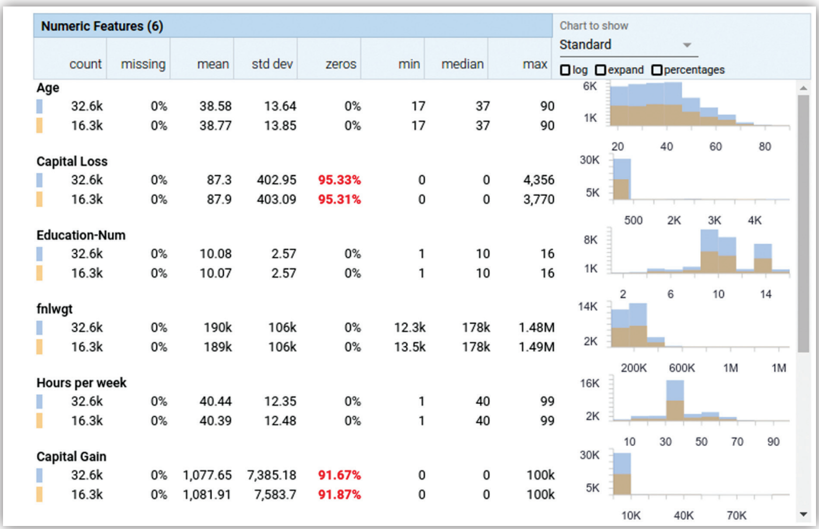


	age	fnlwgt	education.num	capital.gain	capital.loss	hours.per.week
count	32561.000000	3.256100e+04	32561.000000	32561.000000	32561.000000	32561.000000
mean	38.581647	1.897784e+05	10.080679	1077.648844	87.303830	40.437456
std	13.640433	1.055500e+05	2.572720	7385.292085	402.960219	12.347429
min	17.000000	1.228500e+04	1.000000	0.000000	0.000000	1.000000
25%	28.000000	1.178270e+05	9.000000	0.000000	0.000000	40.000000
50%	37.000000	1.783560e+05	10.000000	0.000000	0.000000	40.000000
75%	48.000000	2.370510e+05	12.000000	0.000000	0.000000	45.000000
max	90.000000	1.484705e+06	16.000000	99999.000000	4356.000000	99.000000

Figure 2.1: *Dataset summary from pandas describe function*

Facets overview gives an upgraded view of the summary. It splits the columns of your data into rows of information, showing information like percentage missing, min, max as well as stats like mean, median, and standard deviation. *Figure 2.2* shows the facets overview for the same census dataset. I would like you to follow along the following steps:

1. Open this link: https://pair-code.github.io/facets/index.html#facets-overview.
2. Download the data - https://www.kaggle.com/uciml/adult-census-income?select=adult.csv.
3. Go to the bottom of the page and by using the `"Load your own data"` button, upload `adult.csv` from step 2.



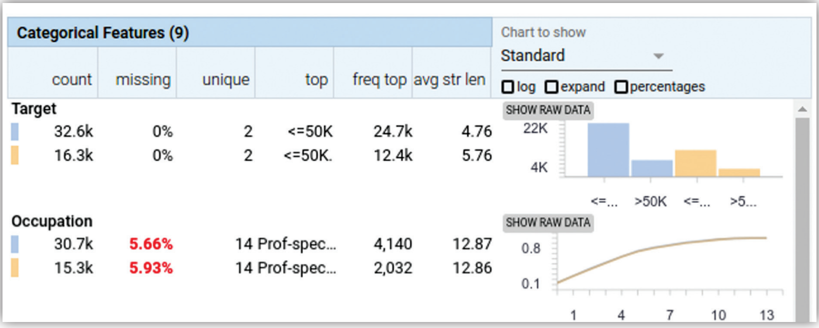
Numeric Features (6)

Chart to show: Standard ☐ log ☐ expand ☐ percentages

	count	missing	mean	std dev	zeros	min	median	max
Age								
🟦	32.6k	0%	38.58	13.64	0%	17	37	90
🟧	16.3k	0%	38.77	13.85	0%	17	37	90
Capital Loss								
🟦	32.6k	0%	87.3	402.95	95.33%	0	0	4,356
🟧	16.3k	0%	87.9	403.09	95.31%	0	0	3,770
Education-Num								
🟦	32.6k	0%	10.08	2.57	0%	1	10	16
🟧	16.3k	0%	10.07	2.57	0%	1	10	16
fnlwgt								
🟦	32.6k	0%	190k	106k	0%	12.3k	178k	1.48M
🟧	16.3k	0%	189k	106k	0%	13.5k	178k	1.49M
Hours per week								
🟦	32.6k	0%	40.44	12.35	0%	1	40	99
🟧	16.3k	0%	40.39	12.48	0%	1	40	99
Capital Gain								
🟦	32.6k	0%	1,077.65	7,385.18	91.67%	0	0	100k
🟧	16.3k	0%	1,081.91	7,583.7	91.87%	0	0	100k

Figure 2.2: *Facets dataset summary*

The missing percentage of values that are zeros works for categorical features as well as shown in the following figure:



Categorical Features (9)

Chart to show: Standard ☐ log ☐ expand ☐ percentages

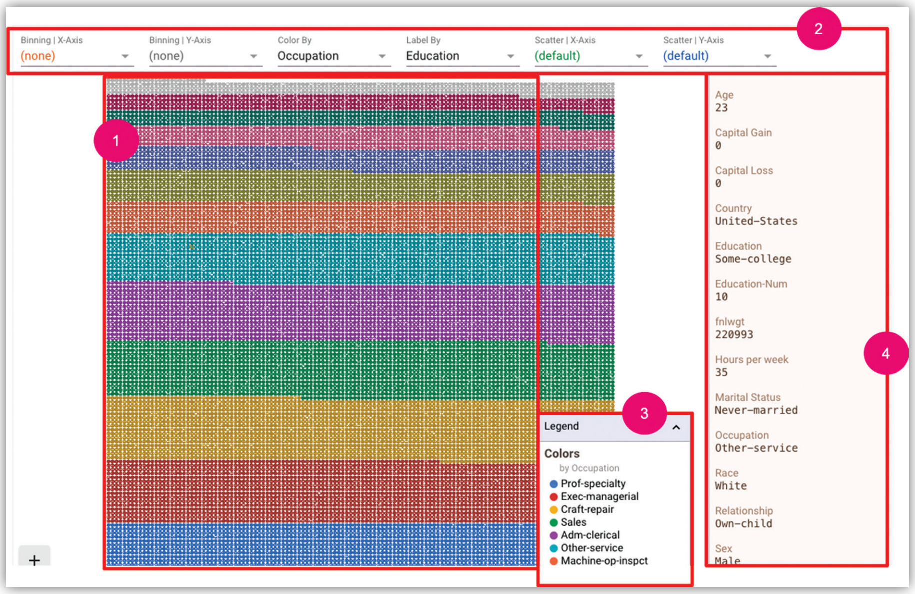
	count	missing	unique	top	freq top	avg str len
Target						
🟦	32.6k	0%	2	<=50K	24.7k	4.76
🟧	16.3k	0%	2	<=50K.	12.4k	5.76
Occupation						
🟦	30.7k	5.66%	14	Prof-spec...	4,140	12.87
🟧	15.3k	5.93%	14	Prof-spec...	2,032	12.86

Figure 2.3: *Facets missing values for categorical features*

These insights about the data help a lot in catching cases where most of the values are zero or missing. You should always test for these distributions on your test and training set for each feature. This will help you uncover data distribution mismatch between training and testing data. If you paid attention to the practices, then you must know that for good accuracy, training and test data must have the same type of distribution.

The data distribution matching between training and testing is essential to avoid problems at later stages, but it gets so cumbersome that practically, it has to be skipped in the real world—imagine the effort required to build these charts! This is where facets come into picture; they reduce the friction to perform that analysis with just a few clicks.

If facets overview is about high-level summary, then dive is all about zooming all the way in to see an individual piece of data. You can facet the data by rows and columns across any of the features of your dataset. *Figure 2.4* shows the dive view. While you can deploy the tool locally, for this book, I would suggest using the online version (https://pair-code.github.io/facets/index.html#facets-dive):


Binning | X-Axis
(none)
Binning | Y-Axis
(none)
Color By
Occupation
Label By
Education
Scatter | X-Axis
(default)
Scatter | Y-Axis
(default)

Age
23
Capital Gain
0
Capital Loss
0
Country
United-States
Education
Some-college
Education-Num
10
fnlwgt
220993
Hours per week
35
Marital Status
Never-married
Occupation
Other-service
Race
White
Relationship
Own-child
Sex
Male

Figure 2.4: *Facets dive components*

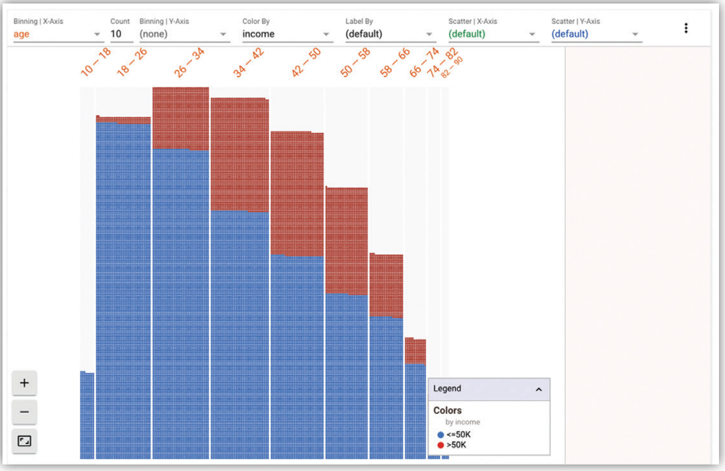
The interface is divided into a few sections which are as follows:

1. The main area in the center; it is a zoomable display of data.
2. The top panel is where you can change the arrangement of your data with various drop downs to control, like faceting, positioning, and color.

3. The legends are for the center display.
4. A detailed view of a specific row of data. You can click on any dot in the center visualization to see the detailed information about the point.

Let's explore a dataset to see all this stuff in action. I will use the facets online version to explain the visualization and use the census dataset—a classic dataset extracted from the 1994 US census by Berry Backer. Its target is to predict whether a household annual income is more or less than $50K based on various census statistics. I would like you to follow along with me with the following steps:

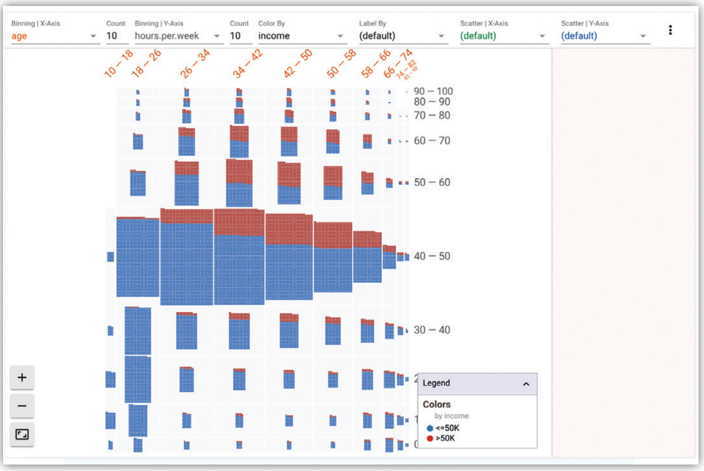
1. Open this link: https://pair-code.github.io/facets/index.html#facets-dive
2. Download the data - https://www.kaggle.com/uciml/adult-census-income?select=adult.csv.
3. Go to the bottom of the page and by using the **Load your own data** button, upload `adult.csv` from step 2.
4. Select **income** from **Color by** column.
5. Select **age** from the X-axis (first) column.
6. Steps 4 and 5 would split the data by age range and color the data points based on the target value. Blue is less than 50K and red is more than 50K (*figure 2.5*):



Binning | X-Axis: age
Count: 10
Binning | Y-Axis: (none)
Color By: income
Label By: (default)
Scatter | X-Axis: (default)
Scatter | Y-Axis: (default)

***Figure 2.5:** Data split based on age range*

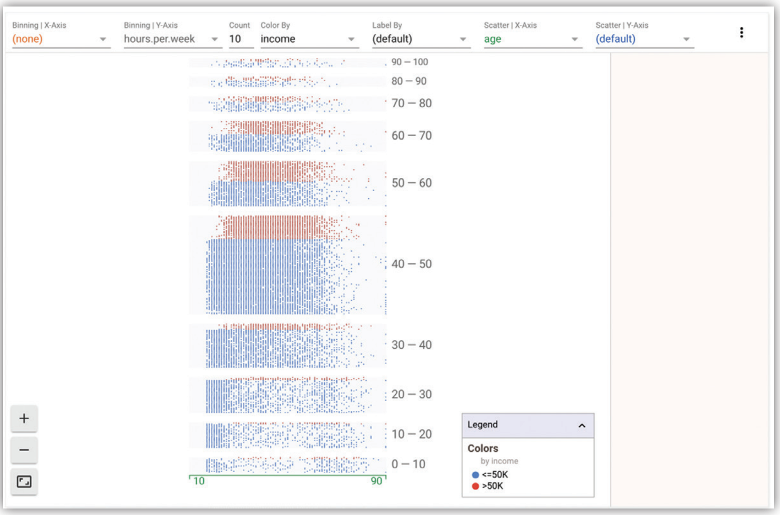
7. Now, let's explore how the working hours change with age.
8. For Y-Axis, select **hour per week** from the dropdown.
9. Make sure that the **X-Axis** has the **age** value and **Color by** has the **income** value.



Binning | X-Axis age
Count 10
Binning | Y-Axis hours.per.week
Count 10
Color By income
Label By (default)
Scatter | X-Axis (default)
Scatter | Y-Axis (default)

Figure 2.6: Age with hours per week

10. There are a few things that stand out in *figure 2.6*. The total working hours are the highest and remain constant during the middle age. They reduce as we grow older.
11. Let's continue the tool exploration. We will switch over to the scatter plot for detailed view.
12. To facet by age, select **age** from the scatter x-axis drop down.
13. For the vertical field, select **hours per week** from the **Binning | Y-Axis** drop down. To be clear, the intent is to view the working hours across different age groups. Now we can indeed see that the **hours per week** rise in the middle of the chart and lower on either side (*figure 2.7*).



Binning | X-Axis (none)
Binning | Y-Axis hours.per.week
Count 10
Color By income
Label By (default)
Scatter | X-Axis age
Scatter | Y-Axis (default)

Figure 2.7: Detailed view

Do play around with faceting and see if you can find other interesting insights. "Facets" is a useful tool for peering into your data and seeing the relationships between different features as well as ensuring that there aren't any missing or unexpected values in your dataset.

2.2 Missing data conundrum: Imputation techniques

Figure 2.8: *Image source - https://pixabay.com/illustrations/data-data-loss-missing-data-process-2764822/*

What is missing data? In simple terms, it is data in which the values are missing for some of the attributes. Missing data is a common and exciting problem in statistical analysis and machine learning. The problem of missing data is relatively common and can have a significant effect on the conclusions that can be drawn from the data. But what are the consequences of missing data?

Missing data presents various problems. These are as follows:

- **Reduced statistical power** - The absence of data reduces statistical power.
- **Bias** - The lost data can cause bias in the estimation of parameters.
- **Reduce the representativeness** - It can reduce the representativeness of the samples.
- **Complicated analysis** - It may complicate the analysis of the study.

Each of these distortions may threaten the validity of the trials and can lead to invalid conclusions. Before we get into the techniques for handling missing data, let's understand why it happens in the first place.

There could be many reasons for missing data such as the following:

- Subjects in long studies may drop out before the study is completed because they may have moved out of the area, died, may no longer see any personal benefit to participating, or may not like the effects of the treatment.
- Surveys could suffer missing data when participants refuse or do not know the answer to or accidentally skip an item.
- Experimental studies experience missing data when a researcher is simply unable to collect an observation. Bad weather conditions may render observation impossible in field experiments. A researcher becomes sick or equipment fails. Data may be missing in any type of study due to accidental or data entry error. A researcher drops a tray of test tubes. A data file becomes corrupt.

The types of missing data fit into four classes, which are based on the relationship between the missing data mechanism and the missing and observed values. These classes are important to understand because the problems caused by missing data and their solutions are different for the four classes.

- **Missing completely at Random (MCAR):** This means that the nature of the missing data is not related to any of the variables, whether missing or observed. An example of MCAR is a weighing scale that ran out of batteries. Some of the data will be missing simply because of bad luck.
- **Missing not at Random (MNAR):** This is also known as non-ignorable because the missingness mechanism cannot be ignored. They exist when the missing values are neither MCAR nor MAR. The missing values on the variable are related to that of both the observed and unobserved variables. For example, people with high salaries generally do not want to reveal their incomes in surveys.
- **Missing at Random (MAR):** MAR means that the missingness can be explained by variables on which you have the full information. However, it's not a testable assumption. Take political opinion polls, for example. Many people refuse to answer to them. If you assume that the reasons people refuse to answer are entirely based on demographics, and if you have those demographics of each person, then the data is MAR. There's really no way to know if that is the full explanation.

But how does knowing the class helps? Well, "Missing completely at Random" and "Missing at Random" are both considered ‘ignorable' because we don't have to include any information about the missing data itself when we deal with the missing data. This means that we can drop those columns without any problems.

There are several methods for handling missing data, and we will explore a few of them next.

- **Deletion** - There are two types of deletion. These are as follows:
  - o **Case deletion:** Case-wise deletion (complete-case analysis) removes all data for an observation that has one or more missing values, particularly if the missing data is limited to a small number of observations. Think of deleting rows that have missing values. List-wise deletion is the most frequently used method in handling missing data, and thus, it has become the default option for analysis in most statistical software packages. Do note that case-wise deletion methods may produce biased parameters and estimates.
  - o **Pair-wise deletion:** In pair-wise deletion, the missing cases are removed by an analysis-by-analysis method. It eliminates information only when the data-point needed to test a particular assumption is missing. If there is missing data elsewhere in the data set, the existing values are used in the statistical testing. Since a pair-wise deletion uses all the information observed, it preserves more information than the list-wise deletion, which may delete the case with any missing data.
- **Imputation:** Imputation is the process of replacing the missing data with estimated values. Instead of deleting any case that has any missing value, this approach preserves all cases by replacing the missing data with a probable value estimated by other information available.
  - o **Mean, median, and mode imputation:** Computing the overall mean, median, or mode is a very basic imputation method; it is the only tested function that takes no advantage of the time series characteristics or the relationship between the variables. It is very fast but has clear disadvantages. One disadvantage is that the mean imputation reduces variance in the dataset.
  - o **Regression imputation:** This is an imputation technique that uses information from the observed data to replace the missing values with predicted values from a regression model. This approach has several advantages because the imputation retains a great deal of data over the list-wise or pair-wise deletion and avoids significantly altering the standard deviation or the shape of the distribution.
  - o **k-Nearest Neighbor (kNN) imputation:** For k-Nearest Neighbor imputation, the missing values are based on the kNN algorithm. These values are obtained by using similarity-based methods that rely on distance metrics (Euclidean distance, Jaccard similarity, Minkowski norm, and so on). They can be used to predict both discrete and continuous attributes.

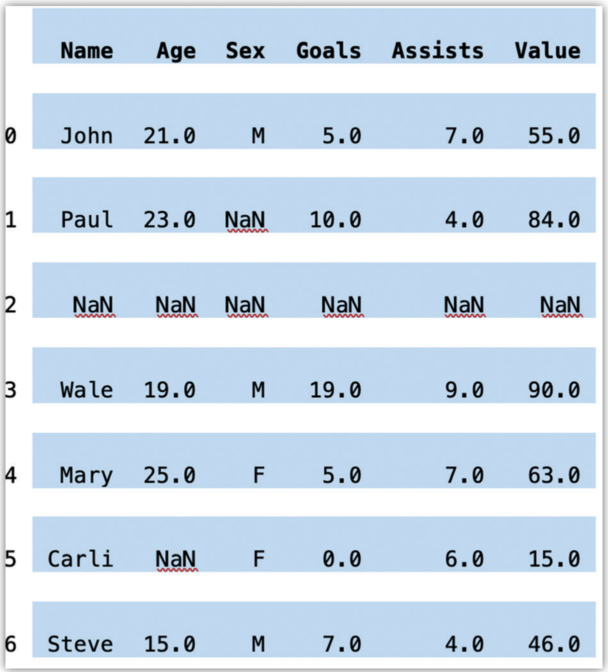
Next, we will explore some of the imputation techniques in code. We will look at 3 libraries – pandas, sklearn, and Autoimpute.

Let's start with pandas as follows (To install pandas, you can use pip command – **pip install pandas**):

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import Imputer
import warnings
warnings.filterwarnings('ignore')
```

```
#Toy Dataset --1
data = {'Name': ['John','Paul', np.NaN, 'Wale', 'Mary', 'Carli', 'Steve'],
        'Age': [21,23,np.nan,19,25,np.nan,15],
        'Sex': ['M',np.nan,np.nan,'M','F','F','M'],
        'Goals': [5,10,np.nan,19,5,0,7],
        'Assists': [7,4,np.nan,9,7,6,4],
        'Value': [55,84,np.nan,90,63,15,46]}


df=pd.DataFrame(data, columns =['Name','Age','Sex','Goals', 'Assists', 'Value']) --2
```

```
df.head(n=7) --3
```

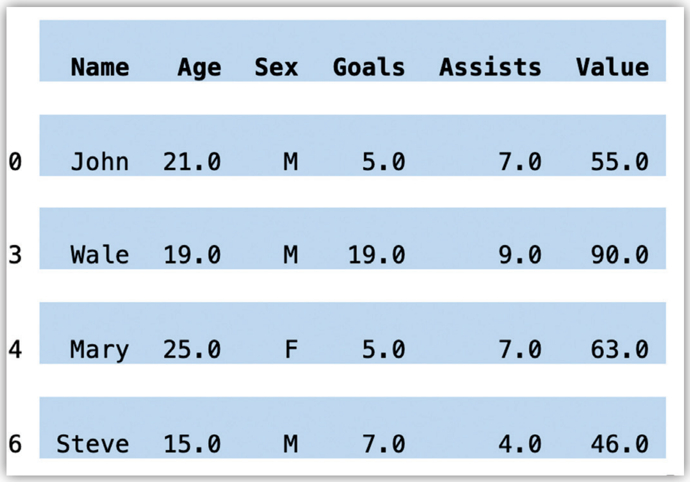


```
    Name   Age  Sex  Goals  Assists  Value
0   John  21.0    M    5.0      7.0   55.0
1   Paul  23.0  NaN   10.0      4.0   84.0
2    NaN   NaN  NaN    NaN      NaN    NaN
3   Wale  19.0    M   19.0      9.0   90.0
4   Mary  25.0    F    5.0      7.0   63.0
5  Carli   NaN    F    0.0      6.0   15.0
6  Steve  15.0    M    7.0      4.0   46.0
```

***Figure 2.9:** Raw dataset with missing values*

```
df.dropna() --4
```

#Output

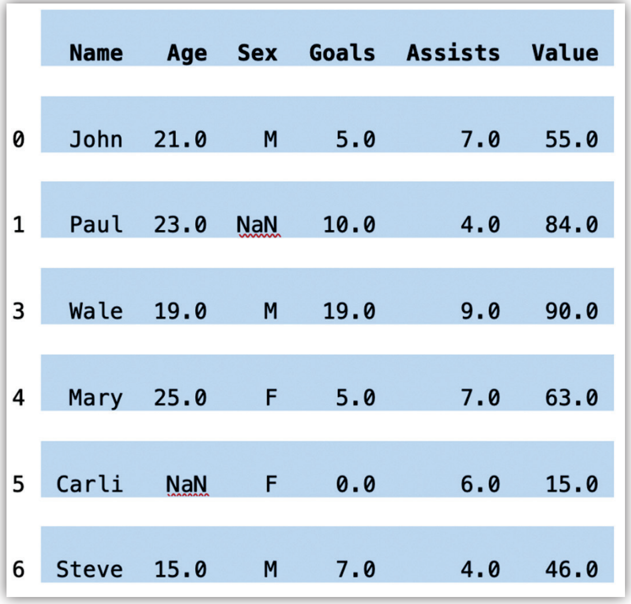


```
     Name   Age  Sex  Goals  Assists  Value
0    John  21.0    M    5.0      7.0   55.0
3    Wale  19.0    M   19.0      9.0   90.0
4    Mary  25.0    F    5.0      7.0   63.0
6   Steve  15.0    M    7.0      4.0   46.0
```

Figure 2.10: *Dropped row with missing values*

```
drop_rows=df.dropna(how='all') --5
```

#output



```
     Name   Age  Sex  Goals  Assists  Value
0    John  21.0    M    5.0      7.0   55.0
1    Paul  23.0  NaN   10.0      4.0   84.0
3    Wale  19.0    M   19.0      9.0   90.0
4    Mary  25.0    F    5.0      7.0   63.0
5   Carli   NaN    F    0.0      6.0   15.0
6   Steve  15.0    M    7.0      4.0   46.0
```

Figure 2.11: *(Dropped row with all columns empty)*

Let's unpack the code to understand the steps as follows:

1. Toy dataset to explore the missing data imputation techniques.
2. Load the toy dataset in a DataFrame.
3. It shows some of the samples.
4. Drop all rows that have a column value missing. This is case deletion.
5. Drop all rows that have all column values missing.

If you look closely at the imputation techniques (mean/mode/median, regression, and KNN), they are organized around two different approaches. In the scikit-learn world, they are as follows:

- **Univariate:** As the name suggests, in univariate, the imputed value for a feature comes from the known values of that feature only. For example, the missing "age" will be imputed on the basis of the known "age" values. Now, it shouldn't come as a surprise that mean/mode and median fall under the univariate category.
- **Multivariate:** Multivariate is a sophisticated approach where a feature missing value is predicted based on an entire feature set. Doesn't it sound like regression?

Let's check out univariate - mean imputation technique with sklearn as follows (To install sklearn, use **`pip - pip install scikit-learn`**).

```
from sklearn.impute import SimpleImputer  --1
X = np.array([[1,2], --2
             [np.nan,3],
             [7,6]])
imp = SimpleImputer(missing_values=np.nan, strategy='mean') --3
imp.fit(X) --4
print(imp.transform(X)) --5
```

Let's unpack the code as follows:

1. The SimpleImputer class provides basic strategies for imputing missing values. It enables you to provide a constant value or use the statistics (mean, median, or most frequent).
2. A random dataset. Feel free to try with another dataset.
3. Initialize the SimpleImputer with a placeholder for missing values (the value that needs to be imputed) and the strategy (here, mean).
4. Fit the imputer on the model. In this case, fitting is all about calculating the mean.
5. Transform the missing values in X. Notice the value X[1][0]

```
[[1. 2.]
[4. 3.]
[7. 6.]]
```

In multivariate, you use the IterativeImputer class that models each feature with missing values as a function of other features, and uses that estimate for imputation. It is a round robin process where at each step, a feature column (with missing values) is designated as target y and the other feature columns are treated as inputs X. A regressor is then fit on (X, y) for the known y. Then, the regressor is used to predict the missing values of y. This is repeated for max_iter imputation rounds. The results of the final imputation round are returned as follows:

```
from sklearn.experimental import enable_iterative_imputer     --1

from sklearn.impute import IterativeImputer   --2

X = np.array([[1, 2], --3

              [3, 6],

              [4, 8],

              [np.nan, 3],

              [7, np.nan]])

imp = IterativeImputer(max_iter=10, random_state=0) --4

imp.fit(X)  --5

print(np.round(imp.transform(X))). --6

#Output

[[ 1. 2.]

[ 3. 6.]

[ 4. 8.]

[ 2. 3.]

[ 7. 14.]]
```

Let's unpack the code as follows:

1. Multivariate estimator is experimental as of now, and hence, you need to enable by importing enable_iterative_imputer.
2. Importing the IterativeImputer.

3. A dummy dataset with few missing values (np.nan).
4. Initializing the imputer with max_iter and a random state.
5. Fit it on all the features
6. Transform the input (replace the missing values).

You can build a composite estimator by using simple iterative imputer in a pipeline.

Autoimpute (https://pypi.org/project/autoimpute/) is a Python package for analysis and implementation of imputation methods. You can install it via PIP, and it supports a bunch of imputation techniques. It makes the imputation methods more accessible. The following table shows various imputation techniques:

Univariate	Multivariate	Time Series
Mean	Linear Regression	Linear
Median	Binomial Logistic Regression	Quadratic
Mode	Multinomial Logistic Regression	Cubic
Random	Stochastic Regression	Polynomial
Norm	Bayesian Linear Regression	Spline
Categorical	Bayesian Mean matching	Time-weighted
	Predictive Mean Matching	Next Obs Carried Backward
	Local Residual Draws	Last Obs Carried Forward

***Table 2.1:** Autoimpute imputation techniques. Source - Documentation*

These are four-step approaches, as claimed by the authors, that should be used with Autoimpute. These are as follows:

- Assess the extent of the missing value problem with descriptive and visual measures. Do you recall that missing values can also be visualized with facets?
- Examine the factors related to the missingness of data (Remember MAR, MCAR, etc.). We have already covered the importance of this step before.
- Try various imputation methods on your dataset and select the most appropriate one.
- Measure the impact of the imputation to the fit.

The preceding four steps also provide a map to understand Autoimpute features. I will not cover all the steps since that can be done more effectively with the framework documentation (https://kearnz.github.io/autoimpute-tutorials/).

Autoimpute is easy to use. If you know scikit-learn, then the Autoimpute syntax will be a cakewalk for you. It inherits from sklearn's BaseEstimator, TransformerMixin, and implement fit and transform methods, making them valid transformers in a sklearn pipeline.

Let's apply the imputation technique to the dataset as follows:

```
from autoimpute.imputations import SingleImputer  --1

imputer = SingleImputer(strategy={"Age":"mean","Sex":"categorical"}). --2

data_imputed = imputer.fit_transform(df) –3

print(imputer.imputed_) –4
```

Let's unpack the code as follows :

1. Autoimpute offers two classes. These are as follows:
   a. **SingleImputer:** Impute missing values only once.
   b. **MultipleImputer:** Impute missing values multiple times.
2. **Initializing the SingleImputer.** Strategy parameter maps the imputation technique to a field. Here, the mean strategy is being applied to a numerical field Age and categorical strategy to a categorical field Sex. It is possible to specify a blanket strategy, for example, SingleImputer(strategy = "mean") but then it will not work because Age and Sex have different field types.
3. It fits to data and then transforms it. If you know TransformerMixin from sklearn, then you don't need any explanation.
4. The imputed_ attribute returns a dictionary where each key is a column and its value is a list with the index of each imputation for that column. These indices represent where the data was originally missing but has now been imputed. You can use these indices to find the location of the imputations within a transformed dataset. Pretty cool, isn't it?

With the knowledge of **`SingleImputer`**, understanding **`MultiImputer`** would be easy. We will follow the usual pattern of having the code followed by the explanation as follows:

```
from autoimpute.imputations import MultipleImputer --1

#Toy dataset --2

toy_df = pd.DataFrame({

    "age": np.random.choice(np.arange(20,80), 50),
    "gender": np.random.choice(["Male","Female"], 50),

    "education": np.random.choice(["Graduate","masters"], 50),

   "employment": np.random.choice(["Unemployed","Employed", "Part Time",
"Self-Employed"], 50),

    "salary": np.random.choice(np.arange(50_000, 1_000_000), 50),

    "weight": np.random.choice(np.arange(100, 300, 0.1), 50),

})

print(toy_df)

toy_df.loc[toy_df.sample(frac=0.1).index, 'gender'] =np.nan --3

imp = MultipleImputer(  --4

    n=10,

      strategy={"salary":  "pmm",  "gender":  "bayesian  binary  logistic",
"age": "norm"},

       predictors={"salary":  "all",  "gender":  ["salary",  "education",
"weight"]},

    imp_kwgs={"pmm": {"fill_value": "random"}},

    visit="left-to-right",

    return_list=True

)

imp.fit_transform(toy_df) --5
```

Let's unpack the code as follows :

1. Importing the MultiImputer feature from autoImpute.
2. Setting up a toy dataset.
3. Setting up a few missing values in the gender feature.
4. Instantiating the MultiImputer:
    a. **strategy:** The strategy argument specifies the imputation technique. The following list shows some of the strategies available to impute a column within a DataFrame. Please read the documentation for the full list.
        i. default predictive
        ii. multinomial logistic

iii. Bayesian least squares

iv. mean

v. median

vi. Mode

vii. random

Predictive default is the default strategy if none is specified. Depending on the column's data point, the predictive default chooses the preferred strategy to use. For example, pmm for numerical, multinomial logistic for categorical). Note that some of these strategies are for categorical data while others are for numeric data. You can specify the strategies in three ways:

- **string:** It broadcasts the strategy across every column in the DataFrame:

  si_str = MultipleImputer(strategy="mean")

- **tuple:** Strategies gets applied to the corresponding column position:

  si_list = MultipleImputer (strategy=["mean", "binary logistic", "median"])

- **dictionary:** This is where the key is the column that we want to impute, and the value is the strategy to use:

  si_dict = MultipleImputer (strategy={"gender":"categorical", "salary": "pmm"})

We are using the dictionary style strategy and would recommend using it as it makes it more readable and less prone to unexpected behavior.

b. **img_kwgs: img_kwgs:** This is the means to specify additional parameters to the strategy. Here, we are providing an additional value to the pmm strategy. "pmm" stands for predictive mean matching and you can learn more about it here - https://statisticalhorizons.com/predictive-mean-matching. Think of it as a way to customize the strategy. So, how do you know which additional parameters are available strategy-wise? Well, consider this an exercise and unsurprisingly, the answer is there in the documentation. (https://autoimpute.readthedocs.io/en/latest/user_guide/strategies.html). It is very important to get familiar with the documentation in order to use the library efficiently.

c. **predictors:** Predictors specify what the imputation model depends on if the imputation model is multivariate predictive. For example, to impute gender, salary, education, and weight should be used. The selected fields are hypothetical but you must have gotten the point. If no predictors are specified, then all columns are used.

   d. **n:** The number of imputations to generate.

   e. **return_list:** If True, then imputations are done all at once, not evaluated lazily, and this will return M*N where M is the number of imputations and N is the size of the original DataFrame.

5. Fit in the data and transform it.

You may wonder that if **`MultipleImputer`** returns multiple imputations, then how do you select one? There is no automated way for it. The recommended way is to plot the original data along with the imputed data. You may want to check the **`autoimpute.visuals`** module in the documentation.

For the sake of completion, there is another feature which enables you to examine the missingness of the data. Let's follow the usual pattern of code and then the explanation as follows:

```
from autoimpute.visuals import plot_md_percent, plot_md_locations  --1


#Toy dataset --2

toy_df = pd.DataFrame({

    "age": np.random.choice(np.arange(20,80), 50),

    "gender": np.random.choice(["Male","Female"], 50),

    "education": np.random.choice(["Graduate","masters"], 50),

   "employment": np.random.choice(["Unemployed","Employed", "Part Time", 
"Self-Employed"], 50),

    "salary": np.random.choice(np.arange(50_000, 1_000_000), 50),

    "weight": np.random.choice(np.arange(100, 300, 0.1), 50),

})

--3

toy_df.loc[toy_df.sample(frac=0.2).index, 'gender'] =np.nan

toy_df.loc[toy_df.sample(frac=0.2).index, 'salary'] =np.nan

toy_df.loc[toy_df.sample(frac=0.2).index, 'age'] =np.nan

plot_md_percent(toy_df)  --4
```

```
plot_md_locations(toy_df) -5
```

Let's unpack the code as follows:

1. Import the necessary modules.
2. Set up the toy dataset.
3. Randomly remove some of the values to create missingness.
4. Call the `plot_md_percent` method. This plots the percentage of missing data by column within a DataFrame. This is shown as follows:

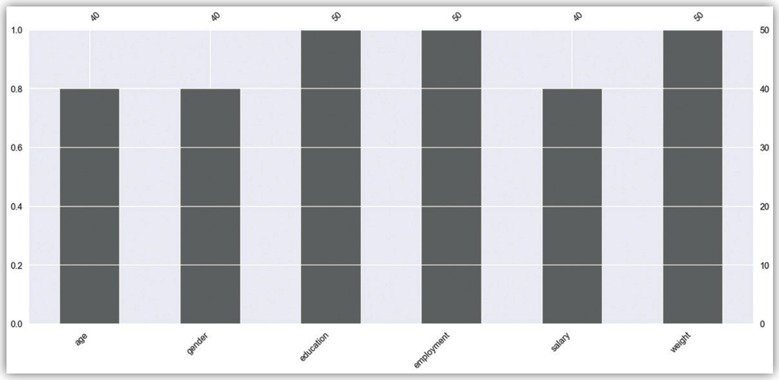


Figure 2.12: *Percentage of missing data*

5. Call the `plot_md_locations` method. This plots the locations where the data is missing within a DataFrame as shown in the following figure:

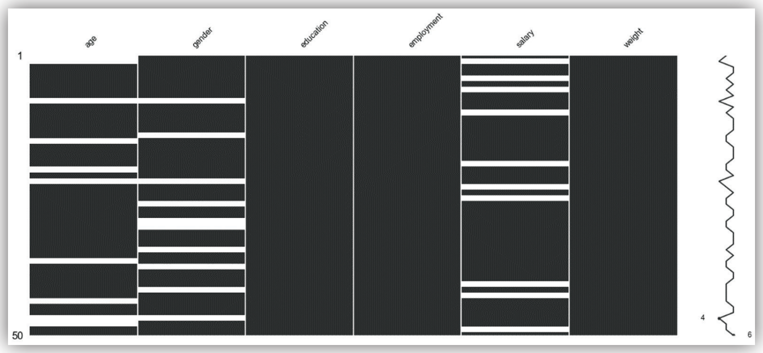


Figure 2.13: *Location of missing data*

Prefer facets over AutoImpute for missing data exploration.

It is important to take a step back and recall what we started from. All the features that have been explained are around the following four-step approach for handling missing data:

1. Explore the missing data (`plot_md_percent` and `plot_md_location`).
2. Identify the reasons for missingness (MAR, MCAR, etc.).
3. Impute with SingleImputer and MultiImputer.
4. Analyze the impact of imputation (`autoimpute.visuals` module).

2.3 Scaling data

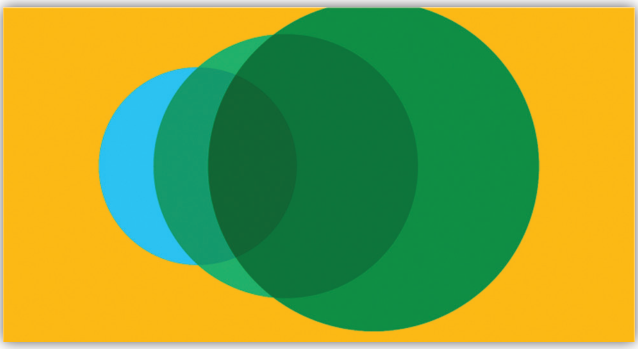


Figure 2.14: *Image source - https://theodi.org/*

The numeric data is already in a format that is easily used by mathematical models. But this doesn't obviate the need for feature engineering. Good features should not only represent salient aspects of the data, but should also conform to the assumptions of the model. Hence, transformations are often necessary. It is important to note that numeric feature engineering techniques are fundamental and can be applied whenever data is converted into numeric features. There are a few sanity checks to be always performed regardless of where the data comes from. These are as follows:

- **Does magnitude matters?**

  The first sanity check is to check whether the data magnitude matters.

  Does it matter if they are +ive or -ive? Do we only need to know the magnitude at a very coarse granularity? This sanity check is particularly important for automatically accrued numbers such as counts—the number of daily visits to a website, the number of reviews garnered by a restaurant, etc.

- **Scale of the features**

  The next one is to check the scale of the features. What are the largest and the smallest values? Do they span several orders of magnitude? Models that are smooth functions of input features are sensitive to the scale of the input. This helps you in deciding the need for data normalization. Algorithms like k-means clustering, nearest neighbors' methods, radial basis function

(RBF) kernels, and anything that uses the Euclidean distance are sensitive to magnitude, and hence, the features for these algorithms should be normalized beforehand.

Logical functions, on the other hand, are not sensitive to input feature scale. Hence, models based on space-partitioning trees (decision trees, gradient boosted machines, random forests) are not sensitive to scale.

- **Distribution of numeric features**

  Feature distribution helps in getting a sense of the data you are dealing with. Distribution helps you to identify erroneous data like outliers. You already know that outliers are not good in quite a few ML algorithms.

  Even if outliers are absent, some of the algorithms are sensitive to data distribution. For instance, the training process of a linear regression assumes the prediction errors to be Gaussian distributed. This could become a problem if the target predictions have an exponential range.

  It really helps to know whether your data is skewed or normally distributed. A lot of tests assume the data to be normally distributed. Knowing that beforehand, would help you decide the right transformation strategy.

Let's now look at a few strategies for engineering numerical values. When data can be produced at a high volume and velocity, it's very likely to contain a few extreme values.

Binarization is a technique for converting raw numbers into binary. Let's understand this with an example. In the Million Song Dataset (http://millionsongdataset.com/), the raw listen count is not a robust measure of user taste. Users have different listening habits. Some people might listen to their favorite songs quite frequently while others might listen to them at a lesser frequency. We can't necessarily say that someone who listens to a song 20 times must like it twice as much as someone else who listens to it 10 times.

A more robust representation of user preference is to binarize the count and clip all counts greater than 1 to 1. The binary target is a simple and robust measure of user preference.

Binning maps a continuous number to a discrete one. You can think of the discretized numbers as ordered sequence of bins that represent a measure of intensity. For example, rather than having the raw count, you can group the counts in bins and get rid of the raw count values.

An important factor for binning is to decide the size of each bin. There are two options for it which are as follows:

- o **Fixed-width binning:** In fixed-width binning, each bin contains a specific numeric range. The ranges can be custom designed or automatically segmented, and they can be linearly scaled or exponentially scaled. I'm sure you would have seen age ranges like these before:

  0–12 years old

  12–17 years old

  18–24 years old

- o **Quantile binning:** Fixed-width binning is easy to compute. But if there are large gaps in the counts, there will be many empty bins with no data. This problem can be solved by adaptively positioning the bins based on the distribution of the data. This can be done using the quantiles of the distribution. Quantiles are values that divide the data into equal portions. For example, the median divides the data in halves; half of the data points are smaller and half are larger than the median. The quartiles divide the data into quarters, the deciles into tenths, etc. To calculate the bins, the pandas library is a reasonably good choice.

Log transformation - To understand log transformation, we need to understand log function. Log function compresses the range of large numbers and expands the range of small numbers. The larger the x, the slower the log(x) increments. The following is a visual distribution of news article popularity with and without log transformation. Note that we have focused on only one property—the number of words in the article. The distribution looks much more Gaussian after the log transformation:

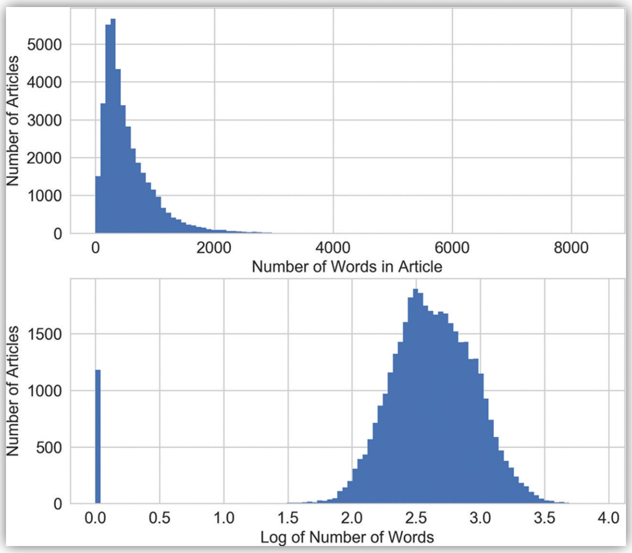


Figure 2.15: *Log transformation*

Transform generalization - It is possible to generalize the transformation. A simple generalization of both the square root transform and the log transform is known as the Box-Cox transform. Models that are smooth functions of the input, such as linear regression, logistic regression, or any that involve a matrix, are affected by the scale of the input. If your model is sensitive to the scale of the input features, then feature scaling helps. As the name suggests, feature scaling changes the scale of the feature. It is also called feature normalization. Feature scaling is usually done individually to each feature.

Min-max scaling squeezes (or stretches) all feature values to be within the range of [0, 1]. *Figure 2.16* illustrates this idea:

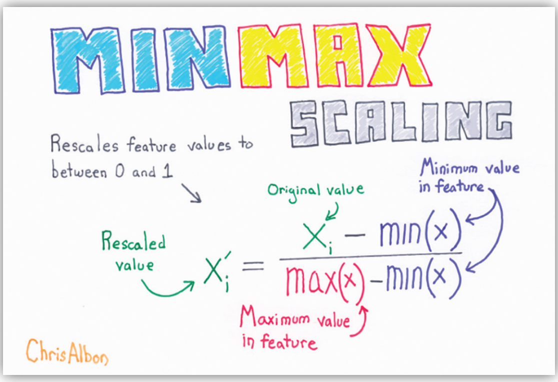


Figure 2.16: *Min-max scaling*

Standardization (Variance Scaling)

It subtracts off the mean of the feature (over all data points) and divides by the variance. Hence, it can also be called variance scaling. The resulting scaled feature has a mean of 0 and a variance of 1. The following figure illustrates the point:

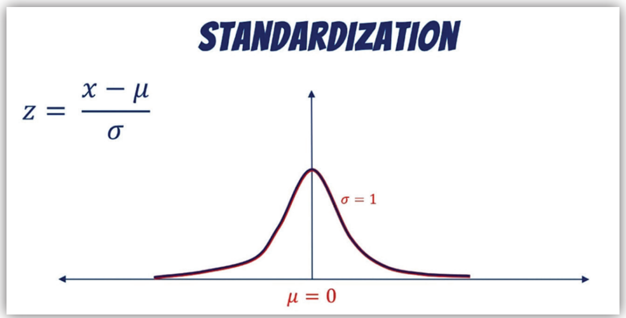


Figure 2.17: *Standardization*

While on one hand we can extract features from features—for example—extracting a weekend/weekday from a date field. On the other hand, it is also possible to combine multiple features together into more complex features. A lot of the times, combined features help the model to generalize better and learn more accurately. The process of combining features into new features is known as **interaction**.

An example would be helpful here: Tenure and monthly charges can be multiplied to create a new feature, that is, yearly charges. Feature scaling is a must when you have features that have very different scales. For instance, the age of employees will be less than 100 while their salaries might be in the thousands.

I have shared some code examples of the above ideas in the associated chapter notebook. Since they include the commonly used sklearn, numpy, pandas, matplotlib, and so on, we will not get into detailing out the code.

2.4 Outlier treatment

Figure 2.18

An outlier is a rare chance of occurrence within a given data set. It is an observation point that is distant from other observations. In simple terms, an outlier is any data point which greatly differs from the rest of the observations in a dataset. When a data point deviates markedly from the others in a sample, it is called an outlier. Any other expected observation is labelled as an inlier.

There can be multiple reasons for a data point to be an outlier. A few of them are as follows:

- **Data entry errors** – These may be caused by a human while entering the data.
- **Measurement error** – These may occur if the instrument developed some snag and gave the incorrect value.

- **Intentional** – These may occur if someone doesn't want to reveal the information. A lot of the times, we do not want to reveal our salary and would put zero in surveys or leave it empty.
- **Incorrect distribution –** These may occur if the point represents the usual occurrence of another distribution.
- **Sampling error** - These may occur if the data has been extracted or sampled from the wrong place.

Based on these causes, you can clearly see that outliers can come from many sources and hide in many dimensions. To explain the reason behind why a data point is an outlier, we need to first locate the possible outliers in the data. At a broad level, we group the approaches in two categories:

- **Univariate:** A univariate outlier is a data point that consists of an extreme value on one variable. Here, it is feature-specific.
- **Multivariate:** A multivariate outlier is a combination of unusual scores on more than one variable. Here, it is an instance with multiple attributes (outlier as a whole).

While we are discussing outlier detection approaches, it is very important to keep in mind the context and the answer the question *"Why do I want to detect outliers?" or "What happens if I don't detect the outliers?"*

Quite a few ML algorithms are sensitive to the range and distribution of feature values. Outliers, if present, can skew and mislead the training process, resulting in longer training times, lesser accurate models, and poorer results ultimately.

The following are some of the popular methods to handle outlier problems:

- Z-score or extreme value analysis (parametric)
- Probabilistic and statistical modelling (parametric)
- Linear regression models (PCA, LMS)
- Proximity based models (non-parametric)
- Information theory models.
- High dimensional outlier detection methods (high dimensional sparse data)

Fortunately, PyOD (https://pyod.readthedocs.io/en/latest/) wraps a lot of algorithms for us. PyOD is a Python toolkit for detecting outlying objects in multivariate data. Here are a few reasons why pyOD is a useful library for outlier detection:

- At the time of writing of this book, it supports more than 30 algorithms.

- It provides optimized performance with JIT and parallelization whenever possible.

The PyOD toolkit features can be grouped in three major functional groups which are as follows:

- **Individual detection algorithms:** There are round 30 supported algorithms at the time of writing this book.
- **Outlier ensembles and outlier detector combination:** This is based on a very intuitive idea; rather than relying on one detector, try with different detectors and then combine them. It essentially combines (for example, calculate the average or vote) the outlier score of various detectors.
- **Utility functions:** Utility function for manipulating data.

Let's explore a KNN-based detector. KNN is essentially a proximity-based algorithm that uses the distance to the nearest neighbor as the outlier score:

```
from pyod.models.knn import KNN   # kNN detector --1

from pyod.utils.data import generate_data

from pyod.utils.data import evaluate_print

from pyod.utils.example import visualize


# Let us generate some sample data

contamination = 0.1  # percentage of outliers --2

n_train = 200  # number of training points

n_test = 100  # number of testing points


X_train, y_train, X_test, y_test = generate_data(

    n_train=n_train, n_test=n_test, contamination=contamination) --3


# train kNN detector

clf_name = 'KNN'

clf = KNN() --4

clf.fit(X_train)  --5



# get the prediction labels and outlier scores of the training data
```

```
y_train_pred = clf.labels_  # binary labels (0: inliers, 1: outliers) --6

y_train_scores = clf.decision_scores_  # raw outlier scores  -7


# get the prediction on the test data

y_test_pred = clf.predict(X_test)  # outlier labels (0 or 1) -8

y_test_scores = clf.decision_function(X_test)  # anomaly scores --9


# evaluate and print the results

print("\nOn Training Data:")

evaluate_print(clf_name, y_train, y_train_scores)

print("\nOn Test Data:")

evaluate_print(clf_name, y_test, y_test_scores)


visualize(clf_name, X_train, y_train, X_test, y_test, y_train_pred,

          y_test_pred, show_figure=True, save_figure=False)  --10
```

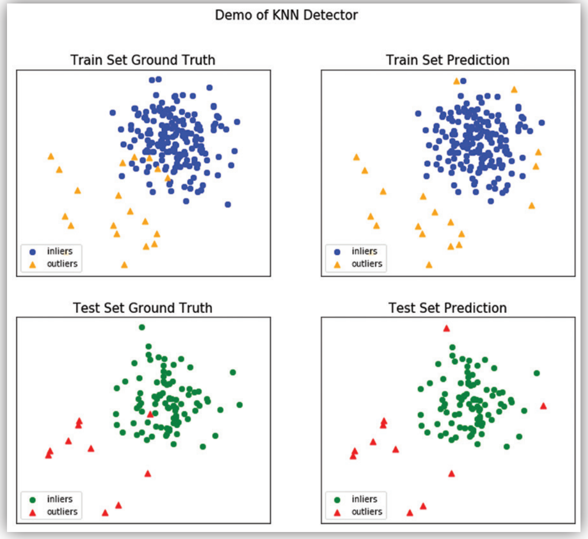


Figure 2.19: *KNN detection visualization*

Let's unpack the code as follows:

1. We import the necessary libraries. pyod.models contains the different algorithms and pyod.utils contains the different utility functions. Note that we will use the KNN algorithm.
2. Contamination here is the proportion of outliers in the data set. It is used when fitting to define the threshold on the decision function.
3. This is a utility function to generate synthesized data. Normal data is generated by a multivariate Gaussian distribution and the outliers are generated by a uniform distribution.
4. KNN is initialized.
5. Fit the model.
6. Get the prediction labels. These are the binary labels of the training data. 0 stands for inliers and 1 for outliers/anomalies.
7. Get the decision scores. `decision_scores_` is the outlier scores of the training data. The higher, the more abnormal. Outliers tend to have higher scores.
8. It is predicts the outlier labels (0 or 1).
9. `decision_function` predict raw anomaly score using the fitted detector.
10. As the name suggests, it helps in visualizing the results.

Let's now shift our focus to another PyOD functional group—outlier detection combination. Outlier detection robustness can often suffer from model instability due to its unsupervised nature. One of the workarounds is to combine the various detector outputs. Detector combination is a subfield of outlier ensembles.

The following are some of the combination mechanisms available in PyOD:

- **Average:** Average scores of all detectors.
- **Maximization:** Maximum score across all detectors. This works like a voting system.
- **Average of Maximum (AOM):** Divide the base detectors into sub-groups and take the maximum score for each sub-group. The final score is the average of all sub-group scores.
- **Maximum of Average (MOA):** Divide the base detectors into sub-groups and take the average score for each sub-group. The final score is the maximum of all sub-group scores.

As always, the code followed by the explanation is as follows:

```
import numpy as np --1
```

```
from sklearn.model_selection import train_test_split

from scipy.io import loadmat

from pyod.models.knn import KNN

from pyod.models.combination import aom, moa, average, maximization

from pyod.utils.utility import standardizer

from pyod.utils.data import generate_data

from pyod.utils.data import evaluate_print

X, y = generate_data(train_only=True)  # load data. --2

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4)
--3

X_train_norm, X_test_norm = standardizer(X_train, X_test). --4
n_clf = 20  # number of base detectors
k_list = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140,
            150, 160, 170, 180, 190, 200]  --4
train_scores = np.zeros([X_train.shape[0], n_clf])
test_scores = np.zeros([X_test.shape[0], n_clf])
--5
for i in range(n_clf):
    k = k_list[i]

    clf = KNN(n_neighbors=k, method='largest')
    clf.fit(X_train_norm)

    train_scores[:, i] = clf.decision_scores_
    test_scores[:, i] = clf.decision_function(X_test_norm)
    print('Base detector %i is fitted for prediction' % i)
train_scores_norm, test_scores_norm = standardizer(train_scores,
                                                   test_scores)  --6
y_by_average = average(test_scores_norm)  --7
evaluate_print('Combination by Average', y_test, y_by_average)
```

```
y_by_maximization = maximization(test_scores_norm)  --8
evaluate_print('Combination by Maximization', y_test, y_by_maximization)
y_by_aom = aom(test_scores_norm, n_buckets=5)  --9
evaluate_print('Combination by AOM', y_test, y_by_aom)


y_by_moa = moa(test_scores_norm, n_buckets=5) --10
evaluate_print('Combination by MOA', y_test, y_by_moa)
```

Let's unpack the code as follows. While the code is complete, I will focus only on the new parts:

1. Import the necessary libraries. The new thing here is the combinations import.
2. Generate the synthetic data. You have already seen this before.
3. Split the data into test and train with 60% reserved for training and the remaining 40% for testing.
4. Standardizer is a PyOD utility function that, as the name suggests, standardizes the value. It conducts Z-normalization on data to make it a zero mean with a unit variance.
5. Initialize 20 KNN outlier detectors with different k values ranging from 10 to 200.
6. After fitting and predictions, the scores need to be standardized as well. The output scores are standardized into zero average and unit standard before combination. This step is crucial to adjust the detector outputs to the same scale.
7. We combine the scores by average mechanism. Recall that the input scores are from the 20 detectors.
8. We combine the results by maximum mechanism.
9. We combine the results with AOM mechanism.
10. We combine the results with MOA mechanism.
11. Evaluate all the combinations by ROC and precisions.

Charts are important tools to understand the outliers. Use box plots, histograms, and scatter plots to identify outliers.

Now the only remaining piece of the puzzle is outlier treatment. Fortunately, the tricks to deal with outlier treatment are very similar to missing values – imputation techniques, deletion, transformation, binning, etc.

Deletion is self-explanatory and you have already learned about various imputation techniques, so let's take a look at transformation and binning. Transforming variables can help eliminate outliers. For example, the natural log of a value reduces the variation caused by extreme values. The image on the left shows a sample set and the image on the right shows the log transformed value:

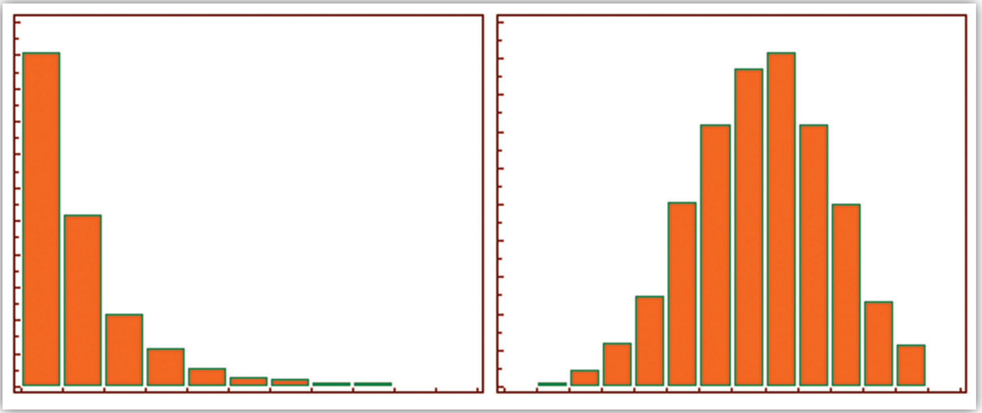


Figure 2.20: *Outlier treatment with log transformation*

Binning is also a form of variable transformation. It essentially transforms continuous variables into discrete variables by creating a set of contiguous intervals that span the range of the variable's values. Since they are in the same interval, they no longer differ from the rest of the values at the tails of the distribution. You can have the following binning types :

- **Equal width:** This is an unsupervised binning method. It divides the data into k intervals of equal size. This is quite helpful for skewed variables as it spreads data equally. Quantile is a good way to create bins of equal width.
- **Equal frequency:** This is also an unsupervised binning method. It groups approximately the same number of values in a bin.

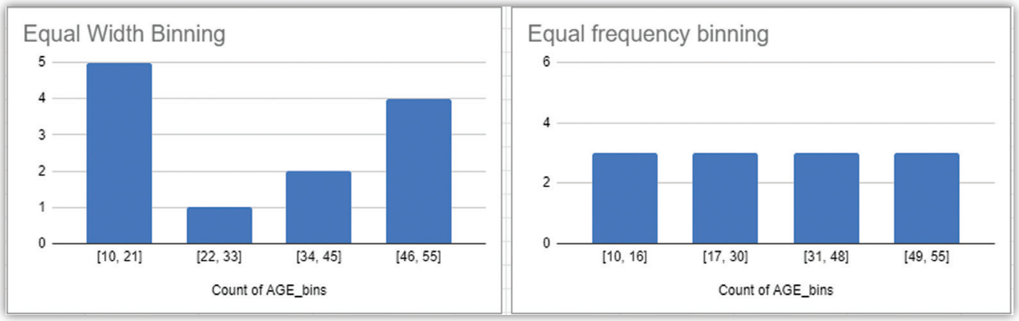


Figure 2.21: *Equal width vs equal frequency binning.*

- Instead of linear models, consider using tree-based methods like random forests and gradient boosting techniques, which are less impacted by outliers.
- Use MAE instead of RMSE as a loss function.

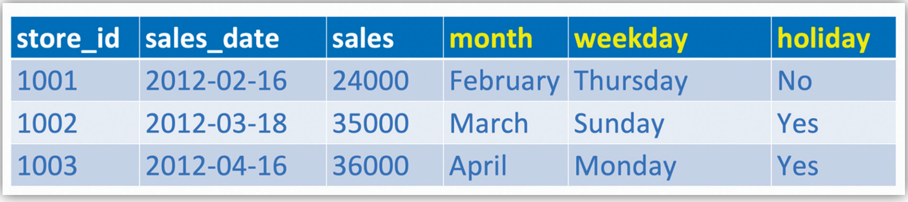
2.5 Feature engineering

Feature extraction or engineering, also known as feature creation, is the process of constructing new features from existing data to train a machine learning model. One of the most important success factors for a data science project is the features used. Learning is easy if you have many independent features that each correlate well with the class. However, feature extraction is a drawn-out manual process, relying on domain knowledge, intuition, and data manipulation. This process can be extremely tedious, and the final features will be limited by both human subjectivity and time.

We will learn feature engineering or automated feature engineering via the featuretools Python library. The process of feature extraction can get complicated when the data is spread across multiple tables. At a high level, we can group the extraction operations across the following two categories:

- Transformation
- Aggregation

A transformation acts on a single table. Imagine the table to a pandas DataFrame. Transformation operations create a new feature from one or multiple columns. Let's take a look at the following dummy dataset which has stores sales summary data. I have kept the minimum number of fields for brevity:

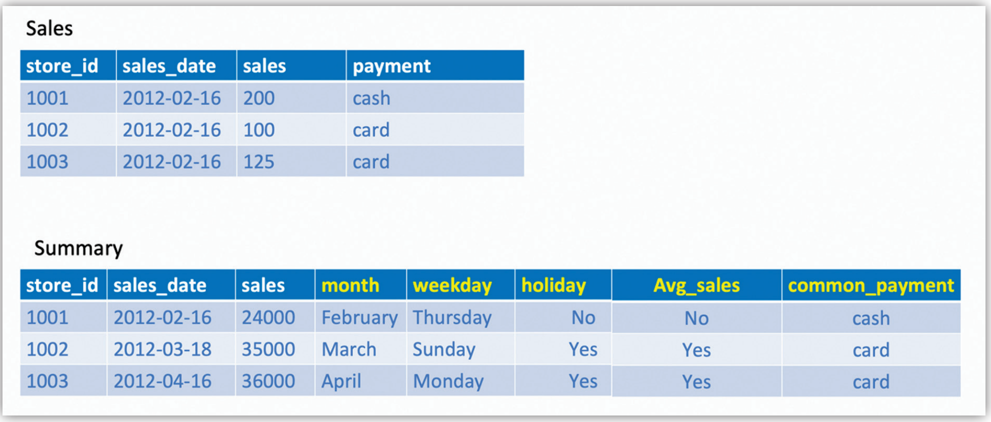


store_id	sales_date	sales	month	weekday	holiday
1001	2012-02-16	24000	February	Thursday	No
1002	2012-03-18	35000	March	Sunday	Yes
1003	2012-04-16	36000	April	Monday	Yes

***Figure 2.22:** Dummy dataset with generated features*

We have extracted month and weekday from the sale_date column. With additional information like month and weekday, we provide new data relationships for learners to pick up. It doesn't need to stop here. We can figure out if that date was a holiday. We know by experience that holidays can lead to increase in sales and we would like our learner to consider this relationship while training.

Aggregation is performed across tables and uses a one-to-many relationship to group observations and then calculate statistics.



Sales

store_id	sales_date	sales	payment
1001	2012-02-16	200	cash
1002	2012-02-16	100	card
1003	2012-02-16	125	card

Summary

store_id	sales_date	sales	month	weekday	holiday	Avg_sales	common_payment
1001	2012-02-16	24000	February	Thursday	No	No	cash
1002	2012-03-18	35000	March	Sunday	Yes	Yes	card
1003	2012-04-16	36000	April	Monday	Yes	Yes	card

Figure 2.23: *Dummy data with aggregated features*

The sales table stores individual sales data. It has been joined with the existing sales summary data to extract new features like average sales or the common payment option used. The process would first involve grouping on the basis of `store_id`, and then calculating the aggregates and joining with the sales summary DataFrame on the `store_id` column. While all these activities may seem trivial while using pandas, it becomes a problem when there are hundreds of features spread across multiple tables.

Well, this is where featuretools (https://www.featuretools.com/) comes into the picture. featuretools is an open-source Python library that can automatically create many features from a set of related tables. It is based on a method known as deep feature synthesis (DFS). The "deep" here has nothing to do with deep learning. I will come back to DFS shortly.

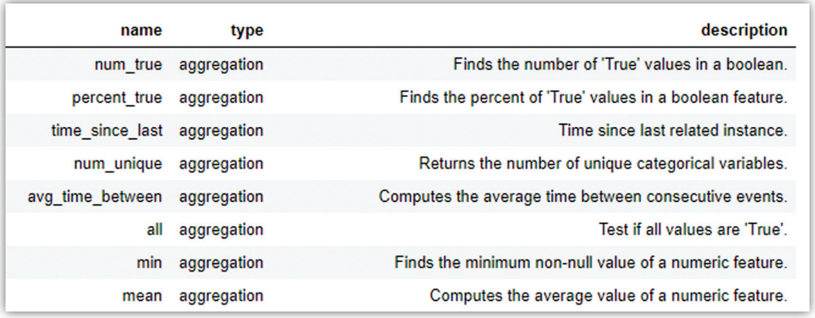
To use featuretools, we need to be aware of a few of its components which are as follows:

- **Entities:** Think of an entity as a table or a pandas DataFrame.
- **Entitysets:** A collection of Entities and the relationship between them is called Entitysets. An intuitive way to think about table relationships is to imagine one-to-many relationships between the tables. An analogy could be a parent-child relationship. For a single parent row, there could many rows in the child table.
- **Feature primitives:** These are basic operations that can be performed to form new features. Calculating the mean or finding the minimum or maximum value of a column are some examples of feature primitives.

Let's take a closer look at feature primitives. Feature primitive (or primitive operations) can be of the following types:

- **Aggregation:** An operation performed across a parent-child relationship. The `avg_sales` and `common_payment` features generated earlier by joining and grouping is an example of aggregation.
- **Transformation:** When done on a single or multiple columns in a single entity. Extracting features like month and weekday are a few examples of it. Do remember that primitives can be stacked.

The following are a few examples of feature primitives, and yes, it is possible to write custom primitives:



name	type	description
num_true	aggregation	Finds the number of 'True' values in a boolean.
percent_true	aggregation	Finds the percent of 'True' values in a boolean feature.
time_since_last	aggregation	Time since last related instance.
num_unique	aggregation	Returns the number of unique categorical variables.
avg_time_between	aggregation	Computes the average time between consecutive events.
all	aggregation	Test if all values are 'True'.
min	aggregation	Finds the minimum non-null value of a numeric feature.
mean	aggregation	Computes the average value of a numeric feature.

***Figure 2.24:** Common feature primitives.*

You now understand all the building blocks to understand DFS. Deep feature synthesis is the method to create new features. To make features with specified primitives, we use the DFS function (stands for deep feature synthesis). We pass in the entity set, the `target_entity` which is the table in which we want to add the features, the selected transformations, and aggregations. Pictorially, this is how it looks:

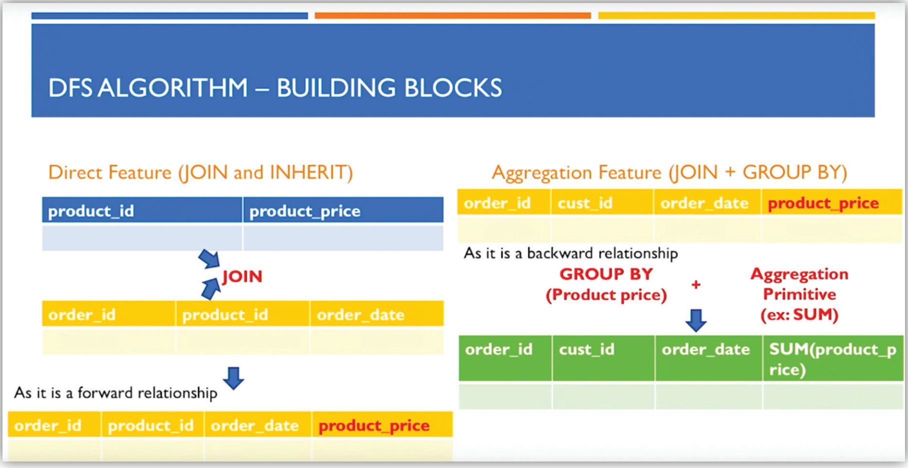


***Figure 2.25:** DFS algorithm steps.*

***Source** – YOW! Data 2019 Ananth Gundabattula*

On the left-hand side, we have joined the product entity with the order entity. This leads to a feature addition, that is, `product_price` to the order entity.

On the right-hand side, we have done the aggregation which is essentially a group by and then applying an aggregation function. The order-id and the product price from the order entity have been grouped and the sum primitive operation has been applied to it.

Deep feature synthesis stacks multiple transformation and aggregation operations to create features from the data spread across many tables. Let's explore featuretools in code now:

```
#!pip install featuretools

# For conda

# conda install -c conda-forge featuretools

import featuretools as ft --1

data = ft.demo.load_mock_customer()  --2

data.keys(). -3

es = ft.demo.load_mock_customer(return_entityset=True)  --4

feature_matrix, feature_defs = ft.dfs(entityset=es,

                                          target_entity="customers",

                                          agg_primitives=["count"],

                                          trans_primitives=["month"],

                                       max_depth=1)  --5
```

Let's unpack the code as follows:

1. Import the necessary library.
2. We will use the toy dataset available with the tool.
3. The toy dataset has four entities. Think of the entities as tables.
   a. **Customers:** Unique customers who had sessions.
   b. **Sessions:** Unique sessions and associated attributes.
   c. **Transactions:** List of events in this session.
   d. **Products:** List of products involved in the transactions.
4. Load and return the entityset. "Id" are the indexes here. Note the relationship between the entities:

```
Entityset: transactions
  Entities:
    transactions [Rows: 500, Columns: 5]
    products [Rows: 5, Columns: 2]
    sessions [Rows: 35, Columns: 4]
    customers [Rows: 5, Columns: 4]
  Relationships:
    transactions.product_id -> products.product_id
    transactions.session_id -> sessions.session_id
    sessions.customer_id -> customers.customer_id
```

5. Generate the features which are available in `feature_matrix`. The target_entity is the table where the features will be added. Note the two primitive (operations):

   "**count**" is an **aggregation primitive** because it computes a single value based on many sessions related to one customer.

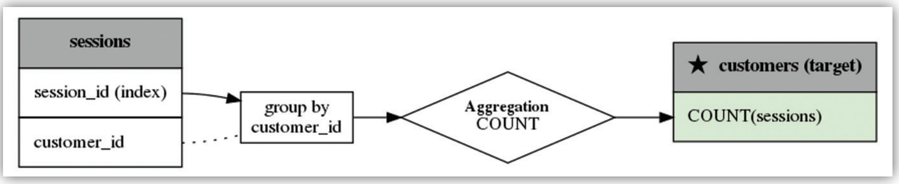


***Figure 2.26:** Aggregative primitive.*

"month" is called a **transform primitiv**e because it takes one value for a customer and transforms it to another.

But how are these helpful? Well, imagine that you are interested in knowing the total number of sessions or months that the customer signed up for. Since primitives are specified as list, you can specify multiple of them. For example, `agg_primitives=["mean", "sum", "mode"]`.

Let's take a step forward. We will now build deep features by stacking primitives:

```
feature_matrix, feature_defs = ft.dfs(entityset=es,
                                   target_entity="customers",
                                   agg_primitives=["mean", "sum", "mode"],
                                   trans_primitives=["month", "hour"],
                                   max_depth=2)
```

If you look at the feature definitions (`feature_defs`), you will get a result like this. Note the highlighted stacked primitives:

```
[<Feature: zip_code>,

 <Feature: MODE(sessions.device)>,

 <Feature: MEAN(transactions.amount)>,

 <Feature: SUM(transactions.amount)>,

 <Feature: MODE(transactions.product_id)>,

 <Feature: MONTH(join_date)>,

 <Feature: MONTH(date_of_birth)>,

 <Feature: HOUR(join_date)>,

 <Feature: HOUR(date_of_birth)>,

 <Feature: MEAN(sessions.MEAN(transactions.amount))>,

 <Feature: MEAN(sessions.SUM(transactions.amount))>,

 <Feature: SUM(sessions.MEAN(transactions.amount))>,

 <Feature: MODE(sessions.MODE(transactions.product_id))>,

 <Feature: MODE(sessions.MONTH(session_start))>,

 <Feature: MODE(sessions.HOUR(session_start))>,

 <Feature: MODE(transactions.sessions.customer_id)>,

 <Feature: MODE(transactions.sessions.device)>]
```

The highlighted stacked primitives are as follows:

1. Calculating the sum of all transaction amounts per session to get the total amount per session.
2. Then apply the mean to the total amounts across multiple sessions to identify the average amount spent per session.

In featuretool lingo, this is called a **"deep feature"** with the depth of two. In the real world, you would need to prepare the entityset from datasets like pandas DataFrame. Next, we will look at an end-to-end example. I will stick to the toy dataset as it saves us from downloading external data.

Let's take a look at the toy dataset as follows:

```
data = ft.demo.load_mock_customer()
```

Let's create a DataFrame for each entity. In the real world, you may get this DataFrame from your data pipeline:

```
products_df = data['products']

transaction_df = data['transactions']

sessions_df = data['sessions']

customers_df = data['customer']
```

Let's merge some of the entities:

```
merged_transactions_df = transaction_df.merge(sessions_df).
merge(customers_df)
```

Now that our data is in DataFrame, let's build an entityset as follows:

```
#STEP 1. Initialize an EntitySet

es = ft.EntitySet(id="customer_data")


#STEP 2. Add entities

es = es.entity_from_dataframe(entity_id="transactions",

       dataframe=transactions_df, index="transaction,

time_index="transaction_time",

variable_types={"product_id": ft.variable_types.Categorical,

"zip_code": ft.variable_types.ZIPCode})
```

While the code is self-explanatory, `time_index` requires an explanation. It is the time when the instance is known. For example, a customer can have multiple transactions done over a period of time. The `time_index` will contain the session start time for all the different sessions. Also, in case you were wondering, `variable_types` are essentially column names.

Let's check the `customer_data` entityset with the `es.plot()` method:

Transactions
transaction_id : index
session_id : numeric
transaction_time : datetime_time_index
amount : numeric
customer_id : numeric
device : categorical
session_start : datetime
join_date : datetime
date_of_birth : datetime
product_id : categorical
zip_code : zipcode

# STEP 3 - Add relationship

Now, we need to connect `customer_data` with products, but for that, the products need to be added first:

```
es = es.entity_from_dataframe(entity_id="products",
                                      dataframe=products_df,
                                    index="product_id")
```

If you now plot with es.plot, you will see transactions and products entities. Let's define a new relationship. Note that the parent comes before the child. The product is the parent here:

```
new_relationship = ft.Relationship(es["products"]["product_id"],
                                        es["transactions"]["product_id"])
```

Then, as it's no surprise, you add that relationship to the entityset:

```
es = es.add_relationship(new_relationship)
```

If you plot using es.plot, you will see the entities with the relationship:

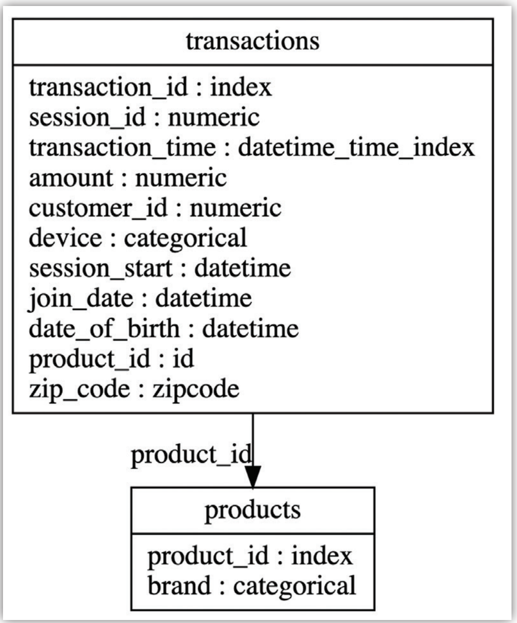


Figure 2.27: *Connected entityset*

# STEP 4 – Build the features

```
feature_matrix, feature_defs = ft.dfs(entityset=es,
                                      target_entity="products")
```

Go ahead and train your model on this expanded feature set.

2.6 Data collection

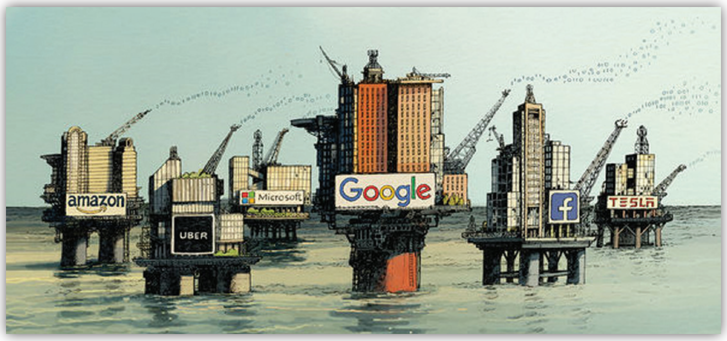


Figure 2.28: *Image credit - https://www.economist.com/*

AI and machine learning are considered to be biggest innovation since the microchip. We are surrounded with products and services that are powered by machine learning.

Much of the recent success has been due to better computation infrastructure and large amounts of training data.

Data is now being called the new oil. However, machine learning has challenges too. Among the many challenges, data collection is becoming one of the critical bottlenecks. It is a known fact that the maximum time spent in an ML project is not on modelling but data collection and pre-processing.

To understand the challenges, we need to first expand our understanding of the data collection process. *Figure 2.30* shows the high-level landscape of the data collection process that was proposed in a research paper https://arxiv.org/pdf/1811.03402.pdf by Yuji Roh, Geon Heo, and Steven Euijong Whang.

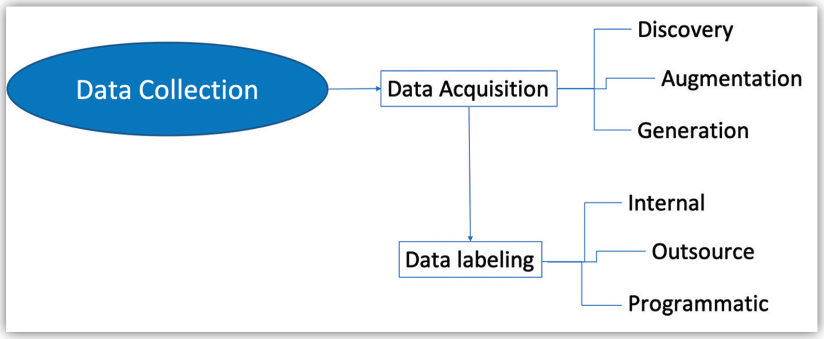


Figure 2.29: *Data collection process landscape*

The key takeaway here is that the data collection process should be looked from both the machine learning and data management perspectives. Data acquisition, as you can guess, is the process of collecting dataset. The training dataset determines the output of our system and the quality of the user experience. You have already seen that data collection can be done via search, augmentation, or synthetic data generation.

In case you are looking for public dataset, then you can use any one or all of the following options:

- **Online**
    - Google dataset search - https://toolbox.google.com/datasetsearch. The Google dataset search is the search engine for dataset. It is a companion of sorts to Google Scholar, the company's popular search engine for academic studies and reports.
    - Google dataset - https://ai.google/tools/datasets/

      These are Google datasets for the broader research community.
    - Kaggle dataset - https://www.kaggle.com/datasets. Kaggle has more than 1000 machine learning ready datasets.

- **Licensed data**

  It is now possible to buy data. Amazon data-exchange and Google offer more than a thousand licensable data products from over 80 data providers. There is a diverse catalog of free and paid offerings in categories such as financial services, healthcare/life sciences, geospatial, weather, and mapping.

  o Amazon data exchange - https://aws.amazon.com/data-exchange/

  o Google commercial datasets - https://cloud.google.com/commercial-datasets/

- **Behind firewall**

  What about data acquisitions that are inside firewalls? Well, this is a problem which almost the majority of the organizations are facing. How to make data available for exploration and insight is a big problem and an opportunity. If you have come across "*data lakes*," then you must understand the problem and the solutions that organizations are trying to build. We will not talk about data lakes in this book.

Now that you know where to look for data, let's look at the augmentation strategy. This is very helpful for deep learning trainings where you have very little data. Data augmentation is a strategy that enables practitioners to significantly increase the diversity of the data available for training models without collecting new data. Augmentation is quite popular for image datasets.

The following are a few augmentation operations which you would have probably already seen:

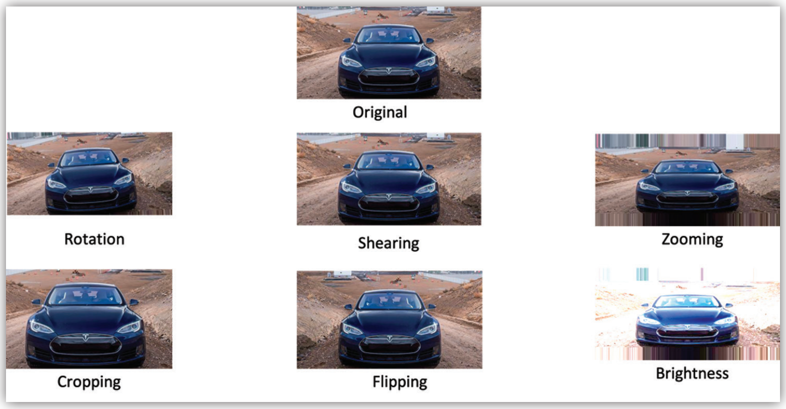


Figure 2.30: *Augmentation stratégies. Image credit - https://www.geeksforgeeks.org/*

Let's explore image augmentation in code. You would have seen the image augmentation in keras via data generators. The code looks like this:

```
datagen = ImageDataGenerator(
        rotation_range = 40,
        shear_range = 0.2,
        zoom_range = 0.2,
        horizontal_flip = True,
        brightness_range = (0.5, 1.5))
```

Now, we will explore an image augmentation library that makes us platform and framework independent. Augmentor (https://github.com/mdbloice/Augmentor) is an image augmentation library in Python for machine learning. It offers finely-grained control over augmentation and implements the most real-world relevant augmentation techniques.

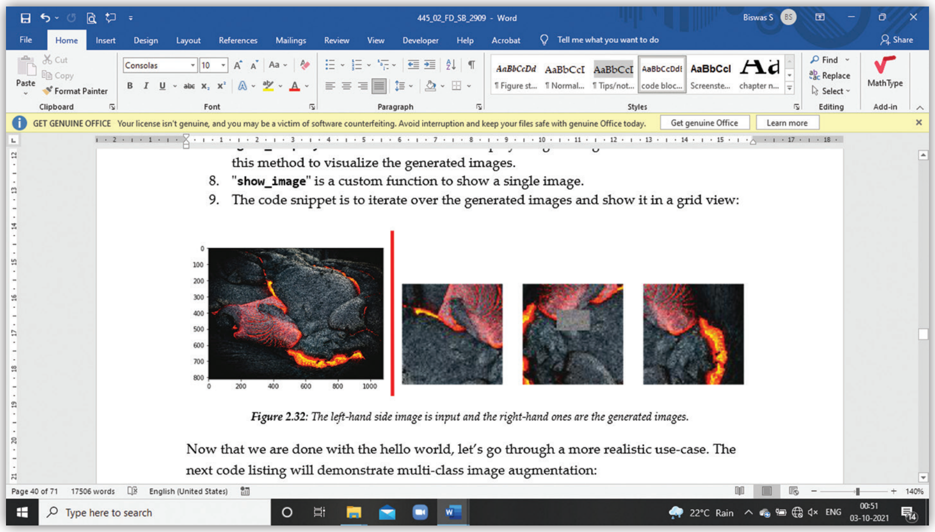
It works by building an augmentation pipeline that defines a series of operations to perform on a set of images. Operations that you saw are added one by one in the pipeline and when executed, it generates augmented dataset. Let's explore the steps through code as follows:

```
#!pip install Augmentor
--1
import matplotlib.pyplot as plt
%matplotlib inline
import Augmentor
import os
import numpy as np
from glob import glob
import PIL
input_images_path = './data/images/volcano'   --2
output = "output". --3
p = Augmentor.Pipeline(input_images_path, output). --4
--5
p.rotate90(probability=0.5)
p.rotate270(probability=0.5)
p.flip_left_right(probability=0.8)
```

```
p.flip_top_bottom(probability=0.3)
p.random_erasing(probability=0.5, rectangle_area = 0.2)
p.crop_random(probability=1, percentage_area=0.5)
p.resize(probability=1.0, width=120, height=120)
#Sample images from the pipeline
p.sample(3)  --6
def  grid_display(list_of_images,  list_of_titles=[],  no_of_columns=6,
figsize=(10,10)):  --7
    fig = plt.figure(figsize=figsize)
    column = 0
    for i in range(len(list_of_images)):
        column += 1
        #  check for end of column and create a new figure
        if column == no_of_columns+1:
            fig = plt.figure(figsize=figsize)
            column = 1
        fig.add_subplot(1, no_of_columns, column)
        plt.imshow(list_of_images[i])
        plt.axis('off')
        if len(list_of_titles) >= len(list_of_images):
            plt.title(list_of_titles[i])
#Show a single image
def show_image(file_path):  --8
    img = PIL.Image.open(file_path)
    plt.imshow(img)
#Generated samples. --9
output_images=os.path.join(input_images_path, output, "*.jpeg")
print(output_images)
images = [ PIL.Image.open(f) for f in glob(output_images) ]
grid_display(images)
```

Let's unpack the code as follows:

1. Import the necessary libraries.
2. Specify the input folder path. This folder would contain images that are to be augmented. Feel free to download the images from the internet and change the path if required.
3. As expected, specify the output folder.
4. The augmentor is initialized by initializing the pipeline. The pipeline requires the input and output location.
5. Next, add a bunch of operations. The operation names are self-explanatory, and the probability parameter specifies the probability of that operation being applied in the pipeline. It essentially introduces randomness to generated outputs.
6. Augmented images are generated by calling the sample method and it takes the number of sample (augmented images) to be generated as the parameter.
7. `"grid_display"` is a custom function to display images in a grid format. We will use this method to visualize the generated images.
8. `"show_image"` is a custom function to show a single image.
9. The code snippet is to iterate over the generated images and show it in a grid view:



this method to visualize the generated images.

8. "**show_image**" is a custom function to show a single image.
9. The code snippet is to iterate over the generated images and show it in a grid view:

Figure 2.32: The left-hand side image is input and the right-hand ones are the generated images.

Now that we are done with the hello world, let's go through a more realistic use-case. The next code listing will demonstrate multi-class image augmentation:

445_02_FD_SB_2909 - Word
File Home Insert Design Layout References Mailings Review View Developer Help Acrobat Tell me what you want to do
GET GENUINE OFFICE Your license isn't genuine, and you may be a victim of software counterfeiting. Avoid interruption and keep your files safe with genuine Office today.
Get genuine Office
Learn more
Page 40 of 71 17506 words English (United States)
Type here to search
22°C Rain ENG 00:51 03-10-2021

Figure 2.31: *The left-hand side image is input and the right-hand ones are the generated images.*

Now that we are done with the hello world, let's go through a more realistic use-case. The next code listing will demonstrate multi-class image augmentation:

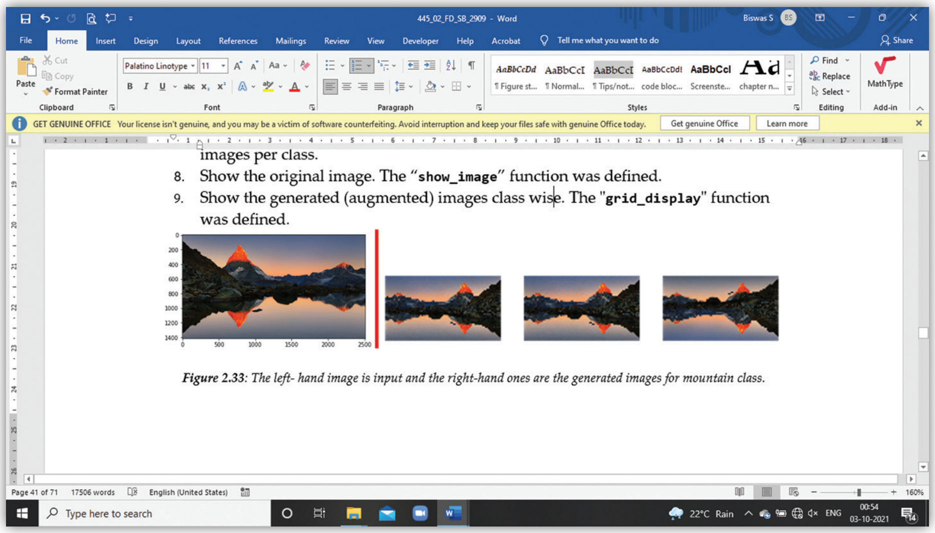
```
pipelines = {} --1

sec_input_images_path = './data/images/mountain'. --2

pipelines['volcano'] = Augmentor.Pipeline(input_images_path) --3

pipelines['mountain'] = Augmentor.Pipeline(sec_input_images_path) --4

--5

pipelines['volcano'].rotate(probability=0.5,  max_left_rotation=5,  max_
right_rotation=5)

pipelines['volcano'].random_distortion(probability=1,      grid_width=4,
grid_height=4, magnitude=8)

--6

pipelines['mountain'].flip_left_right(probability=0.5)

pipelines['mountain'].flip_top_bottom(probability=0.5)

--7

for p in pipelines.values():

   print("Class %s has %s samples." % (p.augmentor_images[0].class_label,
len(p.augmentor_images)))

    p.sample(3)

#original image  --8

file = os.path.join(sec_input_images_path, "photo-mountain.jpeg")

show_image(file)

#Generated samples  --9

output_images=os.path.join(sec_input_images_path, output, "*.jpeg")

print(output_images)

images = [ PIL.Image.open(f) for f in glob(output_images) ]

grid_display(images)
```

Let's unpack the code as follows:

1. Initialize the pipeline.
2. Specify the image classes. There are two classes – `volcano` and `mountains`. The `"volcano"` class image(s) come from code and `"mountain"` new class

(here, folder) needs to be set up. Download the image(s) from the internet and change the path if required.

3. Add the mountain class to the pipeline dictionary and initialize its pipeline. By default, the augmented images will go to the `"output"` folder.
4. Add the volcano class to the pipeline dictionary and initialize its pipeline.
5. Add augmentation operations to mountain class. Again, this is something you have already seen.
6. Add augmentation operations to volcano class.
7. The individual pipelines are executed and sampled to generated three augmented images per class.
8. Show the original image. The “`show_image`” function was defined.
9. Show the generated (augmented) images class wise. The `"grid_display"` function was defined.



images per class.

8. Show the original image. The “**show_image**” function was defined.
9. Show the generated (augmented) images class wise. The "**grid_display**" function was defined.

Figure 2.33: The left- hand image is input and the right-hand ones are the generated images for mountain class.

Figure 2.32: *The left- hand image is input and the right-hand ones are the generated images for mountain class.*

2.7 Large scale data processing with Dask

If you search for large scale data processing, the majority of the results will point to big data tools like Hadoop, Spark, and algorithms like map reduce. However, the golden rule for working with a large dataset is to:

1. First, maximize the throughput on your local machine.

2. Then, when you hit the limit on your local machine, switch to big data tools.

To scale locally, we need to solve the following two problems:

1. **How to handle datasets larger than RAM?**

   Pandas and NumPy require full dataset in the memory. *Figure 2.34* shows the richness of the Python numerical ecosystem. However, most of these tools only work on in-memory data. Pandas, NumPy, and scikit need all the data to be in the memory before processing. We need distributed pandas and NumPy!

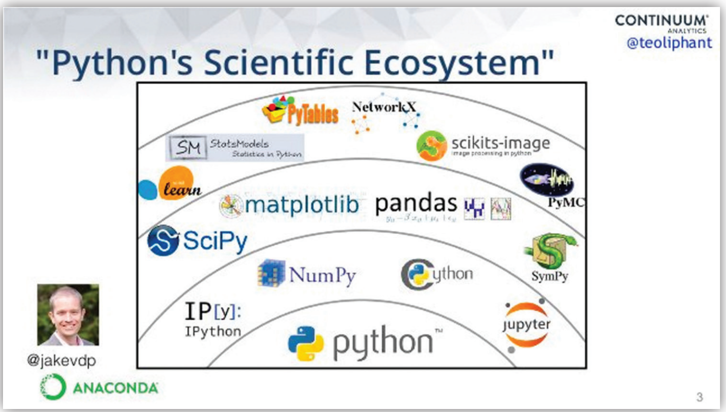


Figure 2.33: *Python scientific ecosystem*

2. **How do we distribute and parallelize the task?**

   The Python single execution mode makes it harder for task distribution and parallelized execution. On launching, the Python application is assigned to one core and it runs on that core until completion or termination.

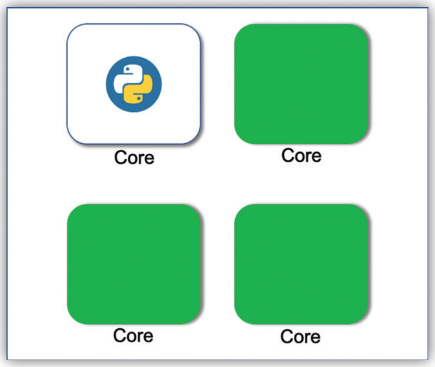


Figure 2.34: *Python single core execution*

It is quite evident that the execution speed can be considerably increased by splitting the job and then allocating the sub-tasks to different cores. Behind the scenes, Python's multiprocessing package launches a new Python process for each core. Each process is independent and separate from the others. Note that we have split the tasks across cores but there is further parallelization opportunity via the threads inside a core. Unfortunately, multi-threading in Python requires you to handle lot of concurrency issues yourself.

Dask (https://Dask.org/) addresses both of the preceding limitations. At the core, it is a flexible library for parallel computing in Python for analytics tasks.

From a developer standpoint, you can view Dask at two levels as follows:

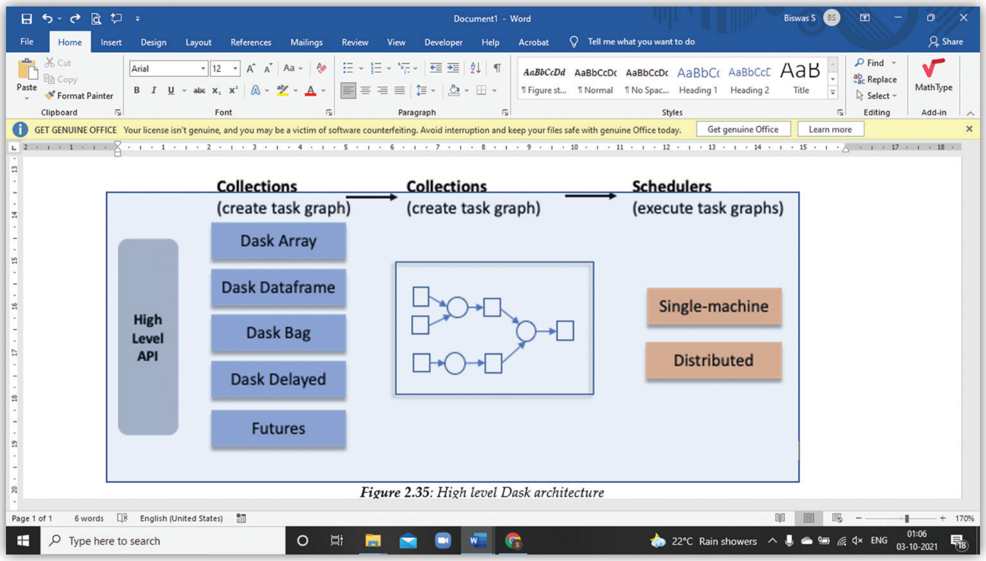


Figure 2.35: High level Dask architecture

At the high level, Dask provides high-level Array, Bag, and DataFrame collections that behave like NumPy, lists, and pandas, but can execute in parallel on datasets that are bigger than the machine RAM.

Using Dask's high-level collections are drop-in replacements for NumPy and pandas for large datasets.

At the low levels, Dask provides features that execute computation graphs in parallel via dynamic task schedulers. Dask's schedulers are an alternative to direct the use of threading or multi-processing libraries in complex cases or other task scheduling systems like Luigi or IPython parallel.

The best part of Dask is its Pythonic ways of doing things which means that the learning curve is minimal. The following is a code snippet to read a csv file with **Dask DataFrame:**

```
import Dask.dataframe as dd
df = dd.read_csv(file_path)
```

How would you read a csv file in pandas?

```
import pandas as pd
df = pd.read_csv(file_path)
```

Other than a different import, there are no changes at all. Let's explore the high-level collections a little more.

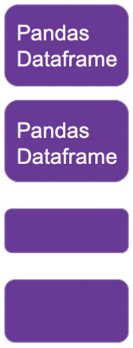


Figure 2.36: *Dask DataFrame*

The preceding image shows a Dask DataFrame. You need to appreciate the Dask power here. It is managing (along a single axis) and giving a single view to the collection of pandas DataFrame. Technically, you can assign each of the collection DataFrame to different, thread, cores, or machines.

For Dask array, it would feel like a copy-paste of NumPy:

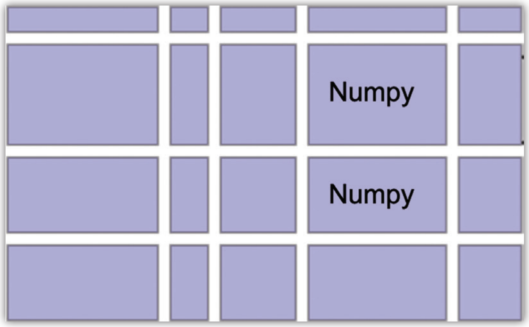


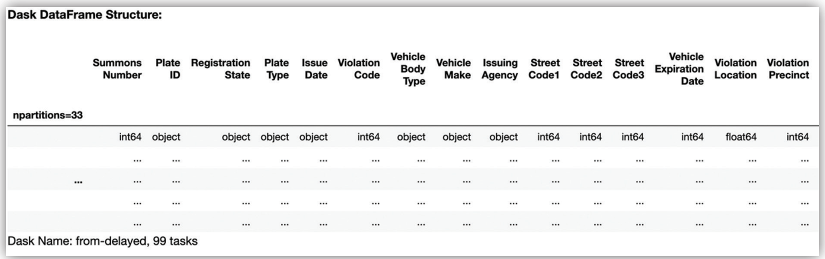
Figure 2.37: *Dask array*

Dask array is managing the collection of NumPy array partitioned along an axis and giving you a single view of the collection. You will no longer be tied to the memory limit of your machine.

With the unified view of the collection, you can now distribute the underlying items (individual NumPy or pandas, etc.) to a different machine or execute them in parallel on a single machine. We will come back to the parallel and distributed mode in a bit. For now, let's continue with Dask DataFrame.

We will use the Kaggle dataset (https://www.kaggle.com/new-york-city/nyc-parking-tickets#Parking_Violations_Issued_-_Fiscal_Year_2017.csv) for the exploration.

```
file_path = './data/nyc-parking-tickets/*2017.csv'

df = dd.read_csv(file_path)

df
```



```
Dask DataFrame Structure:
               Summons  Plate  Registration  Plate   Issue   Violation  Vehicle  Vehicle  Issuing  Street  Street  Street  Vehicle     Violation  Violation
               Number   ID     State         Type    Date    Code       Body     Make     Agency   Code1   Code2   Code3   Expiration  Location   Precinct
                                                                        Type                                               Date
npartitions=33
               int64    object object        object  object  int64      object   object   object   int64   int64   int64   int64       float64    int64
               ...      ...    ...           ...     ...     ...        ...      ...      ...      ...     ...     ...     ...         ...        ...
       ...     ...      ...    ...           ...     ...     ...        ...      ...      ...      ...     ...     ...     ...         ...        ...
               ...      ...    ...           ...     ...     ...        ...      ...      ...      ...     ...     ...     ...         ...        ...
               ...      ...    ...           ...     ...     ...        ...      ...      ...      ...     ...     ...     ...         ...        ...
Dask Name: from-delayed, 99 tasks
```

Figure 2.38: *Dask DataFrame output*

Note that "dd here is Dask. When you run the code, the output may not be what you expected. Pandas would show you the few top records, whereas here, we see the metadata. The top row of the result is the column names, and under each column, we see the guesstimated data type. It is guesstimate because not all data is accessible to Dask in one shot. In order to be reliable, it uses random sampling to guess the data type. The best practice is to play it safe and explicitly mention the data type.

The **`npartitions`** shows how many partitions the data frame will be split into. Rather than working on the entire DataFrame, a Dask worker thread will load one partition at a time. You may have guessed it but it is worth highlighting that by using partitions, Dask is able to work with datasets that are larger than the memory size. Each partition has three tasks – read the raw data, split the raw data into blocks, and initialize the underlying DataFrame. Each partition under the hood is a DataFrame.

1. **Selection and filtering**

```
#Columns selection
```

```
nyc_df[['Issue Date', 'Plate Type']].head(n=2)
```

```
#Row selection
```

```
temp_rows = nyc_df.loc[:200].head(n=2)
```

The preceding code snippets require no explanation because this is how you do selection and filtering with pandas. The column names Issue Date and Plate Type were specified during the DataFrame initialization. Please check the code for details.

2. **Data cleaning process – Removing rows and columns**

   Let's drop those columns that have more than 50% of values missing. The first step would be to calculate the total missing values column.

   ```
   missing_count = temp_rows.isnull().sum()
   ```

   If you divide with total rows, you get the missing values percentage column wise.

   ```
   percent_missing = ((missing_count / temp_rows.index.size) * 100)
   ```

   Then you identify the column index that have more than 50% values missing.

   ```
   columns_to_drop = list(percent_missing[percent_missing >= 50].index)
   ```

   And then you simply drop those columns

   ```
   nyc_df_stage1 = nyc_df.drop(columns_to_drop, axis=1)
   ```

   Again, none of these will be a new thing having pandas knowledge.

3. **Data cleaning process – Imputing missing values**

   Let's fill the missing "violation county" with the most common one as follows:

   ```
   count_of_violation_county = nyc_df_stage1['Violation County'].
   value_counts().compute()
   ```

   The value_counts method counts the unique occurrences of data. Sorting the values and selecting the top value would give the most common country.

   ```
   most_common_county = count_of_violation_county.sort_
   values(ascending=False).index[0]
   ```

   Now replace the missing value with the most common country in the dataset.

   ```
   nyc_df_stage2 = nyc_df_stage1.fillna({'Violation County': most_
   common_county})
   ```

4. **Data cleaning process – Dropping rows with missing values**

   Let's drop those rows that have columns with missing values around 10 percent.

   ```
   drop_rows = list(percent_missing[(percent_missing > 0) & (percent_
   missing < 10)].index)
   ```

#Subset argument specifies which columns to check for null values

```
nyc_df_stage3 = nyc_df_stage2.dropna(subset=drop_rows).compute()
```

The compute function indicates that it needs to be executed explicitly. Taking a step back, if you need the pandas features for datasets larger than the RAM size, then Dask DataFrame is your tool.

We will now shift our attention to the Dask parallel code execution features. Dask can execute algorithms in parallel using an interface called `Dask.delayed` interface. Let's explore the idea with code as follows:

```
def add_one(x):
    return x + 1
def multiply_two(x):
    return x * 2
def plus(x, y):
    return x + y
def is_even(x):
    return not x % 2
data = [1, 3, 7, 12]
step1 = [add_one(i) for i in data]
step2 = [multiply_two(j) for j in step1]
total = sum(step2)
print(total)
```

The code is trivial and would execute sequentially when run. However, when asked to optimize it, you can see that while step 2 is dependent on step 1's output, the individual list items calculation can be done in parallel. One of the requirements for parallel and distributed execution is to not execute the instructions immediately but to delay them. For delayed execution, you need some sort of data structure that allows it to be treated as a task (or job). The task(s) can then be distributed or executed in parallel. The sequence of execution of tasks is defined in a computation graph. The `Dask.delayed` interface decorates functions so that they can be executed lazily (add it as a task). Let's change:

```
import Dask
data = [1, 3, 7, 12]
step1 = [delayed(add_one)(i) for i in data] --1
step2 = [delayed(multiply_two)(j) for j in step1] --2
```

```
total = delayed(sum)(step2) --3

total.visualize() --4

total.compute()  --5
```

Let's unpack the code:

1. Wrap the add_one method and call in the Dask.dealyed method. Note that the method execution is deferred, and step (read task) is added in the computation graph.
2. `Multiple_two` is the same thing as step 1 but is dependent on step 1's output.
3. The sum cannot be executed until steps 1 and 2 are complete. So, it cannot run for sure in parallel to the other steps, but it too can be converted into a task and added to the task group.
4. Until now, there has not been actual execution of code but only task creation. You can view the computation graph via the visualize method. A picture is worth a thousand words.

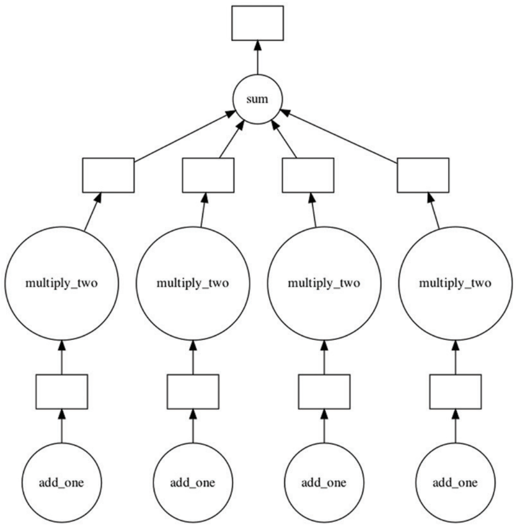
   

   Figure 2.39: *Computation graph*

   Circles are the tasks and the squares represent the results. Note that there are four `add_one` circles for the four list elements indicating that it can be done in parallel. The computation graph essentially shows you the execution blueprint.
5. Execute the computation graph. Do recall that for distributed and parallel execution, the task blueprint and its execution need to be separated.

The following is another example to make you more comfortable with the idea:

```
    data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The first code snippet is for sequential mode (execution). It is a trivial example where the number is multiplied by two if even, otherwise one is added to it. Finally, the whole list is summed up as follows:

```
results = []
for x in data:
    if is_even(x):  # even
        y = multiply_two(x)
    else:          # odd
        y = add_one(x)
    results.append(y)
total = sum(results)
```

The parallelized version would be as follows:

```
results = []
for x in data:
    if is_even(x):  # even
        y = delayed(multiply_two)(x)
    else:          # odd
        y = delayed(inc)(x)
    results.append(y)
total = delayed(sum)(results)
total.visualize()
total.compute()
```

Again, the best way to visualize the execution is to look at the computation graph:

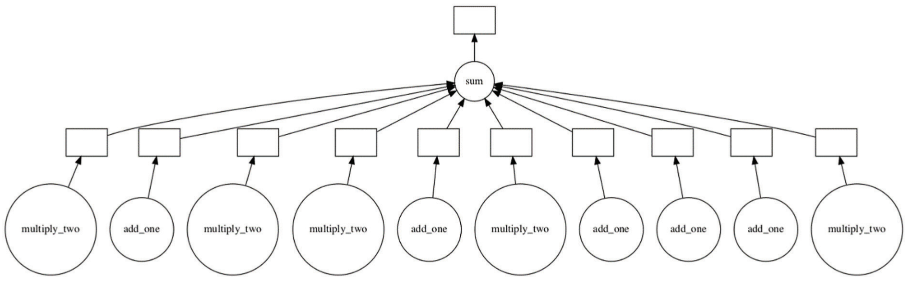


Figure 2.40: *Computation graph for the preceding code*

It is worthwhile to take a step back and think about the different pieces. There are two parts to large dataset processing: dataset size that is beyond the RAM size and then efficient processing of data. Dask DataFrame and arrays etc. allow the handling of data in chunks and the delayed interface enables distributed and parallel execution.

Dask-ML

What would you get if you combine Dask with scikit-learn? You get a scalable ML algorithm that can run across machines. Yes, scikit-learn is parallel, but scikit-learn only provides parallel computing on a single machine with joblib. I will come back to joblib in a moment.

Let's build a classifier using Dask and LogisticRegression. We will use the `make_classification` dataset that generates a random n classification problem:

```
from Dask_glm.datasets import make_classification

from Dask_ml.linear_model import LogisticRegression

from Dask_ml.model_selection import train_test_split


X, y = make_classification()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)


lr = LogisticRegression()

lr.fit(X_train, y_train)

lr.predict(X_test)
```

The code is literally standard scikit-learn code and is using joblib. Joblib (https://pypi.org/project/joblib/) natively provides thread-based and process-based parallelism. It is the joblib that supports the `n_jobs= parameter` during the normal use of scikit-learn. While this works well for modest data sizes but not for large computations such as random forests, hyper-parameter optimization, and more.

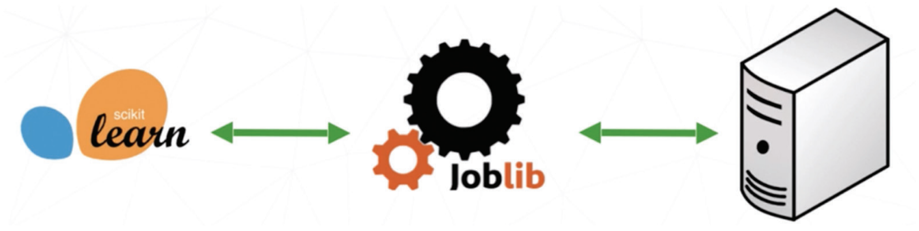


Figure 2.41: Dask thread and process-based processing on a single machine using joblib

Please read the documentation for installing `Dask_ml`.

Dask can extend this parallelism to many machines in a cluster to support large computations. Before we look at the code let's take a look at *Figure 2.43* to see how the distributed computation looks like:

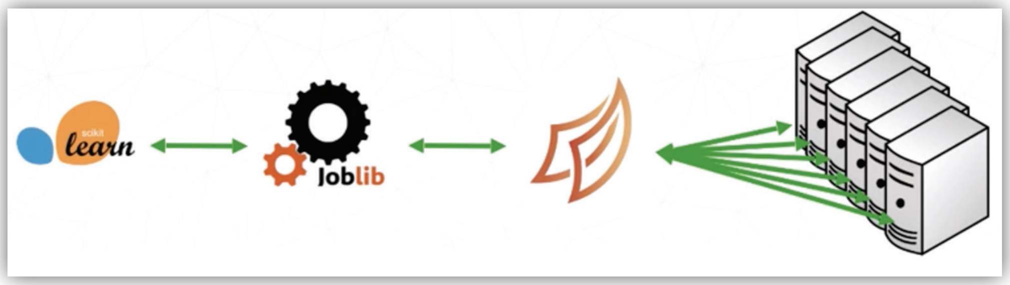


***Figure 2.42:** (Dask parallel execution in a clustered environment)*

Here, Dask is talking to scikit-learn via joblib on one side and using a cluster to train the model on the other side. Let's go through the code to see things in action:

```
from Dask_ml.model_selection import GridSearchCV

parameters = {'penalty': ['l1', 'l2'], 'C': [0.5, 1, 2]}

lr = LogisticRegression()

est = GridSearchCV(lr, param_grid=parameters)

est.fit(X_train,y_train)
```

This should seem familiar. We are using the grid search to find the best value for the logisticRegression parameters - penalty and coefficient C. The dataset is the same as the one we used earlier. Now, let's train it on a cluster as follows:

```
from Dask_ml.model_selection import GridSearchCV

parameters = {'penalty': ['l1', 'l2'], 'C': [0.5, 1, 2]}

lr = LogisticRegression()

est = GridSearchCV(lr, param_grid=parameters)

import joblib #--1

from Dask.distributed import Client #--2


client = Client() #--3

with joblib.parallel_backend('Dask'): #--4
```

```
    est.fit(X_train, y_train) #--5
```

Let's unpack the code here. You would notice that there is no change in the scikit-learn grid search code and we are using the same data as follows:

1. We have imported the joblib library. Recall that joblib runs the scikit-sklearn function on different threads or processes on a local machine. We are importing this to register the new backend which is Dask here. It is essentially to change.
2. We have imported the client from Dask `Dask.distributed` to connect to the Dask cluster.
3. The client is being initialized to connect to the Dask cluster. But when did we start the cluster? On the local machine, it runs automatically when you initialize the client with no parameter.
4. With the `joblib.parallel_backend` context, we are specifying to use the Dask backend or cluster for training.
5. Then, you fit the estimator on the cluster rather than on threads or processes.

While Dask is powerful and compatible with scikit-learn, there are a few limitations that you need to be aware of such as the following:

1. Dask DataFrames are made up of smaller pandas DataFrames.
2. Functions such as insert and pop that alter the structure of the DataFrame are not supported.
3. Although operations like join/merge, groupby, and rolling are supported, they generally require setting index, making them performance bottlenecks.
4. Operations that were slow on pandas like iterating through row-by-row remain slow on Dask DataFrame as well.

2.8 Data distribution

We are going to take a look at data distribution through the following two frames:

- **Storage** - This frame will help you to learn the best practices of data storage. This will help you get an answer to "where to store what." It helps you to navigate through the jungle of different products and platform which are out there.
- **Training and validation data distribution** - This will help you get answers to questions like "What should be the data distribution for training and validation" and "How big should your training and validation dataset be?"

It is important to note that our context will be "supervised" DL problems here. You can imagine data storage being built from the following three blocks:

- **Physical storage**

  Think of the physical storage as your file system. It is where you store your binary files like images, sound files, or texts files. The file system could be your local disk.

  - **NFS** – Network file system that is accessible over network.
  - **Distributed file system** – It is like HDFS (Hadoop distributed file system). It is important to remember the difference between NFS and HDFS. In NFS, the file is stored remotely. Whereas in HDFS, a file is split and saved on all the machines in the cluster.

  With choices comes the challenge of selecting the right option. Access pattern should be an important design consideration for selecting the appropriate storage. The file system can be fast but not good for parallel activities. Of course, other important factors would be security, backup, or recovery.

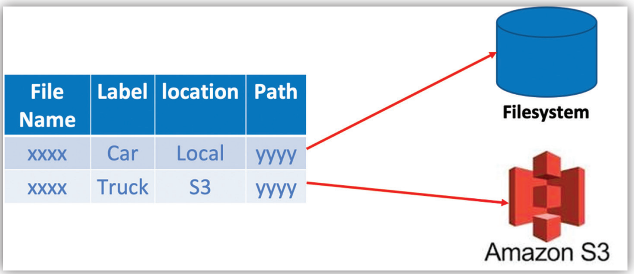
- **Object storage**

  When the files are treated as objects, it essentially indicates an interface - an API layer to access the object. Amazon s3, Microsoft Azure, and Block Blobs are some examples of it. The API layer helps you treat the file system as a service, enabling you to add versioning and redundancy. While it allows parallel access, will not be fast.

  Ceph (https://ceph.io/ceph-storage/) is an on-premise object storage solution in case cloud is not an option for you. Do check out their page for more information if required.

- **Meta data**

  As the name suggests, meta data is the data about the data. Think of it as a table that stores the information about your images or sound files along with the labels. This should ring a bell to think about SQL databases. The fundamental unit will be a row; a column will have different values.



File Name	Label	location	Path
xxxx	Car	Local	yyyy
xxxx	Truck	S3	yyyy

Figure 2.43: *Data metadata storage architecture*

In order to keep the retrieval fast, a good practice is to store the binary file reference and not the actual file. For example, the row will have the label and the URL for s3 file or will have a path of the actual file. The SQL interface for databases makes the data addition, selection, and manipulation a breeze.

Best Storage practices

1. Use object storage for managing the files. They reduce the maintenance overhead considerably.
2. Use SQL databases to manage the metadata. You would have seen a raw version of this in Kaggle competitions. Meta data are generally in XLS and binary files are dumped inside folder.
3. At the training time, copy the data to the local or networked filesystem. This is required because you don't want any bottleneck between your GPU and data.

Training and validation data distribution

While you would agree on the need for effective dev / test data split, you may question this step being considered as the pre-processing step. The reason for including it here is because it helps to think about the data distribution strategy before we begin experimenting. It is of utmost importance to have the same dev / test distribution across different experiments. With changing goals—changing dev / test data here—it is hard to know if we are making any real progress.

We will learn a few guidelines to effectively distribute the data, but before that, let's agree on the term's definition. Typically, data will be split across train, dev, and test sets. The dev set has different names such as hold out set or cross validation set.

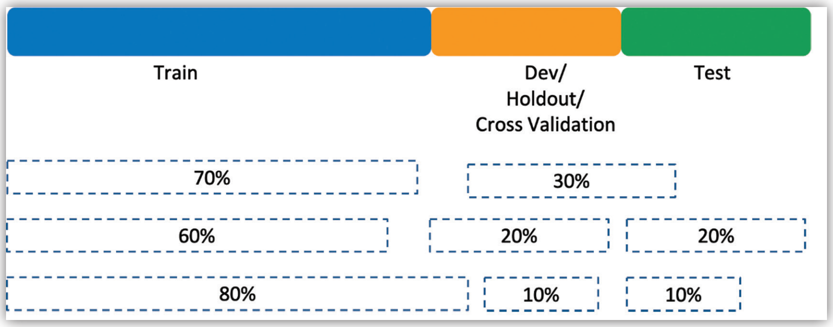


Figure 2.44: *Data split across sets*

A few years ago, a 70-30 % split between train and test or 60-20-20 split between train, dev, and test used to be a standard practice. However, in today's world, it may not always be a good choice.

Deep learning algorithms are hungry for data, and hence, it is okay to give them more during training and reduce the dev/test data size. This is okay because the purpose of the dev dataset is to evaluate between different algorithms, and for that, 30 or 20% of the data is not required. Also, the test dataset purpose is to give a confidence estimate for the final model and that too 20% of the data is generally not required. It is okay if you split your data in the 80 - 10 -10.

- Have separate test sets with real world or production data, or data that you would expect in the future.
- The dev and test sets should be just big enough to accurately represent the performance of the model.

Distribution mis-match

Let's take an example of visual search. Imagine that you build a service that uses images to search the inventory of trolley bags. For example, as a user, you saw a nice trolley and you want to buy the same.

Now, you mined the data on the web to train your model, but once in production, the customers complained about incorrect results. What went wrong?

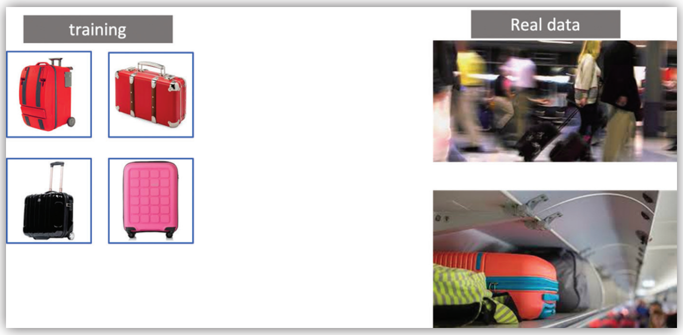


Figure 2.45: *Data mismatch*

Well, it may be the case that the training was done on high quality images from the web but not the real-world pictures were taken from mobile phones and were blurry with a lot of other noises. This is a case of distribution mismatch.

There are two takeaways here which are as follows:

1. Be aware that data mismatch is a real problem and that you need to monitor it.
2. The dev and test sets should come from the same distribution and distribution must reflect the real-world data.

2.9 Training data management with Snorkel

To understand the value of Snorkel, we need to go a little back in time. A few years ago, to train an accurate model, a data scientist had to identify and create the right features. While you know of a tool – featuretools – it is still a painful task. It is not easy to write good features.

Fortunately, deep learning has removed the need for feature engineering and is a commodity for many tasks. But just like in life, nothing comes for free. Deep learning requires very large training datasets. To prepare datasets like imagenet (http://www.image-net.org/), thousands of hours have been spent. Data labelling is a business now. People who call data as the new oil are wrong because labelled data is the new oil.

The key idea of Snorkel is to model the process of training set creation. In simple terms, it is a pragmatic way to build and manage training datasets. Snorkel exposes API for three types of operations which are as follows :

- **Labelling data** - Using heuristic rules or distant supervision techniques.
- **Transforming data** - Rotating or stretching images to perform data augmentation.
- **Slicing data** - Slicing data into subsets.

A picture is worth a thousand words.

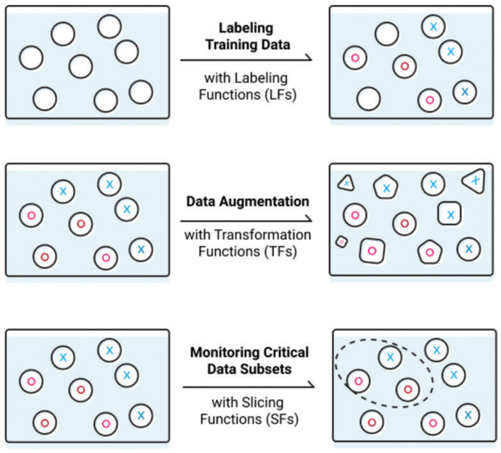

Labeling Training Data
with Labeling Functions (LFs)
Data Augmentation
with Transformation Functions (TFs)
Monitoring Critical Data Subsets
with Slicing Functions (SFs)

Figure 2.46: *Snorkel operations. Image source – Snorkel documentation*

Training a machine learning model without any hand-labelled training data via Snorkel involves five basic steps. These are as follows:

1. Writing labelling functions (LFs).
2. Modelling and combining LFs.
3. Writing transformation functions (TFs) for data augmentation.
4. Writing slicing functions (SFs) for data subset selection.
5. Training a final ML model.

To explore the steps, we will train a model for spam classification and will use a public dataset of YouTube comments (http://www.dt.fee.unicamp.br/~tiago/youtubespamcollection/). Fortunately, Snorkel examples have helper functions to read this dataset and load them into a pandas DataFrame. The code examples are inspired from documentation examples in order to ensure consistency and continuity.

As a pre-requisite, you need to ensure that the following steps are maintained:

1. Snorkel is installed - pip install snorkel
2. textblob is installed - pip install textBlob
3. Git cloned tutorials – Git clone https://github.com/snorkel-team/snorkel-tutorials.git
4. You have the `"data"` folder containing the dataset (YouTube comments). When you get clone tutorials, you get this `data` folder. Make sure that you copy the folder to the folder containing your notebook.
5. `getting_started.py` (https://github.com/snorkel-team/snorkel-tutorials/blob/master/getting_started/getting_started.py) and utils.py (https://github.com/snorkel-team/snorkel-tutorials/blob/master/getting_started/utils.py) are copied to your Jupyter notebook folder.

It is now time to get into the details of each step.

Writing Labelling Functions (LFs)

In this step, we programmatically label the unlabeled data. The purpose of the labelling function is to output a label for a subset of training data. We will use following labels:

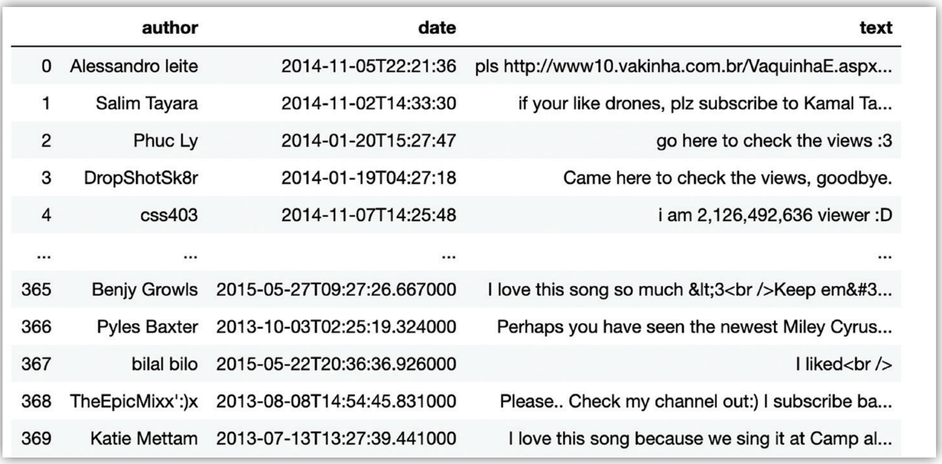
```
ABSTAIN = -1

NOT_SPAM = 0

SPAM = 1
```

The intent of a labelling function is to enable the users to inject domain knowledge (think of rules) to label the data. They don't need to be perfectly accurate. In fact, Snorkel will automatically estimate their accuracies and then reweight and combine their output labels, leading to high-quality training labels.

Let's write a few labelling functions for our unlabeled dataset as follows:

```
import os
from utils import load_unlabeled_spam_dataset
df_train = load_unlabeled_spam_dataset()
```

The `load_unlabeled_spam_dataset` method reads the csv files and dumps them all in a pandas DataFrame . If you check the value of `data_train`, it would be as follows:



	author	date	text
0	Alessandro leite	2014-11-05T22:21:36	pls http://www10.vakinha.com.br/VaquinhaE.aspx...
1	Salim Tayara	2014-11-02T14:33:30	if your like drones, plz subscribe to Kamal Ta...
2	Phuc Ly	2014-01-20T15:27:47	go here to check the views :3
3	DropShotSk8r	2014-01-19T04:27:18	Came here to check the views, goodbye.
4	css403	2014-11-07T14:25:48	i am 2,126,492,636 viewer :D
...	...	...	...
365	Benjy Growls	2015-05-27T09:27:26.667000	I love this song so much <3<br />Keep em&#3...
366	Pyles Baxter	2013-10-03T02:25:19.324000	Perhaps you have seen the newest Miley Cyrus...
367	bilal bilo	2015-05-22T20:36:36.926000	I liked<br />
368	TheEpicMixx':)x	2013-08-08T14:54:45.831000	Please.. Check my channel out:) I subscribe ba...
369	Katie Mettam	2013-07-13T13:27:39.441000	I love this song because we sing it at Camp al...

Figure 2.47: *YouTube comments DataFrame.*

Let's define a few labelling functions. Recall that the objective of labelling function is to inject some domain knowledge programmatically.

Keyword matching

```
from snorkel.labeling import labeling_function
@labeling_function()
def lf_keyword_my(x):
    """Many spam comments talk about 'my channel', 'my video', etc."""
    return SPAM if "my" in x.text.lower() else ABSTAIN
```

The code is self-explanatory. As a real-world rule, a lot of spam has texts like `"my channel,"` `"my-video,"` etc. Note how decorators are being used to designate a function a labelling function.

Regular expression is as follows:

```
import re


@labeling_function()
def lf_regex_check_out(x):
    """Spam comments say 'check out my video', 'check it out', etc."""
    return SPAM if re.search(r"check.*out", x.text, flags=re.I) else ABSTAIN
```

Again, the code is self-explanatory. As a real-world rule, a lot of spam has texts starting with "check out", etc.

Custom heuristic

```
@labeling_function()
def lf_short_comment(x):
    """Non-spam comments are often short, such as 'cool video!'."""
    return NOT_SPAM if len(x.text.split()) < 5 else ABSTAIN
```

Third party model

```
from textblob import TextBlob
@labeling_function()
def lf_textblob_polarity(x):
     """Combine a sentiment analysis  with the heuristic that non spam
       comments are often positive.
       """
   return NOT_SPAM if TextBlob(x.text).sentiment.polarity > 0.3 else ABSTAIN
```

Writing labelling function is generally an iterative process but it is still much better than doing manual labelling.

Labels cleaning

The next step is to apply the labelling functions which generates a label matrix, L_train, where each row corresponds to a data point and each column corresponds to a labelling function. Labelling functions can have unknown accuracies and correlations.

In order to produce clean and integrated training labels, a model called LabelModel needs to be trained to learn the LF accuracies and combine their output labels.

This class learns a model of the labelling functions' conditional probabilities of outputting the true (unobserved) label Y, P(lf | Y), and uses this learned model to re-weight and combine their output labels. Think of it as a model to automatically estimate their accuracies and correlations, reweight and combine their labels, and produce a final set of clean, integrated training labels.

```
--1

from snorkel.labeling.model import LabelModel

from snorkel.labeling import PandasLFApplier

-2

lfs = [lf_keyword_my, lf_regex_check_out, lf_short_comment, lf_textblob_
      polarity] -

--3

applier = PandasLFApplier(lfs)

L_train = applier.apply(df_train)


label_model = LabelModel(cardinality=2, verbose=True) --4

label_model.fit(L_train, n_epochs=500, log_freq=50, seed=123)  --5

df_train["label"]     =     label_model.predict(L=L_train,     tie_break_
      policy="abstain") --6
```

Let's unpack the code:

1. Import the necessary libraries. You already know `LabelModel` and PandasLFApplier, as the name suggests, is a LF applier for a pandas DataFrame.
2. Define the set of labelling functions.
3. Apply the labelling functions. The shape of `L_train` is ((1956, 4). Recall that rows are the datapoint and columns are the labelling functions (4 in our case).
4. Initialize the LabelModel. Cardinality represents the number of classes.
5. Train the label model.
6. Return predicted labels with ties broken according to the policy. Policies to break ties include the following :
    a. "abstain": return an abstain vote (-1)
    b. "true-random": randomly choose among the tied options
    c. "random": randomly choose among tied option using deterministic hash

Transformation for data augmentation

Data augmentation is a strategy for artificially augmenting the existing labelled training datasets by creating transformed copies of the data points. You have seen image augmentation in action with Augmentor. Data augmentation leverages information about domain invariances into ML models via the data rather than by trying to modify their internal architectures. These days, data augmentation is increasingly used in a range of settings, including text.

A text data augmentation strategy of randomly replacing a word with a synonym has been implemented as following:

```
import random

--1

import nltk

from nltk.corpus import wordnet as wn

from snorkel.augmentation import transformation_function

from snorkel.augmentation import ApplyOnePolicy, PandasTFApplier

--2

nltk.download("wordnet", quiet=True)

--3

def get_synonyms(word):

    """Get the synonyms of word from Wordnet."""

    lemmas = set().union(*[s.lemmas() for s in wn.synsets(word)])

    return list(set(l.name().lower().replace("_", " ") for l in lemmas)
      - {word})

--4

@transformation_function()

def tf_replace_word_with_synonym(x):

    """Try to replace a random word with a synonym."""

    words = x.text.lower().split()

    idx = random.choice(range(len(words)))

    synonyms = get_synonyms(words[idx])

    if len(synonyms) > 0:
        x.text = " ".join(words[:idx] + [synonyms[0]] + words[idx + 1 :])
        return x

--5
tf_policy = ApplyOnePolicy(n_per_original=2, keep_original=True)
tf_applier = PandasTFApplier([tf_replace_word_with_synonym], tf_policy)
df_train_augmented = tf_applier.apply(df_train)
```

Let's unpack the code:

1. Import the necessary libraries. For finding a word synonym, we will use the nltk wordnet corpus. In case you don't know what corpus is, then think of it as a dictionary of English, similar to a traditional thesaurus. The augmentation will be implemented as a transformation function, and hence, the transformation function would have been imported.
2. Download the wordnet corpus.
3. It returns the synonym(s) of a word. WordNet groups English words into sets of synonyms called synsets and the synsets function returns the synonyms set. In linguistics, the canonical form or morphological form of a word is called a lemma. To find a synonym as well as an antonym of a word, you call the lemmas function in WordNet. In short, a Synset represents a group of lemmas that all have a similar meaning while a lemma represents a distinct word form.
4. Designate a transformation function that calls `get_synonyms` inside.
5. Apply the transformation. As an exercise, compare the size before and after transformation using the DataFrame shape property.

Slicing function

The slicing function is part of the Snorkel slicing package. It outputs binary masks indicating whether a data point is in the slice or not. Each slice represents some noisily defined subset of the data (corresponding to an SF) that needs to be monitored programmatically. It has primarily two functions which are as follows:

- Monitor specific slices. In the real-world, certain subsets are more important than others. We will take a look at an example shortly.
- Improve model performance over them by adding representational capacity targeted on a per-slice basis. Go ahead and check the documentation for the same (https://www.snorkel.org/use-cases/03-spam-data-slicing-tutorial).

By now, you know the pattern – define a function and decorate it with a Snorkel function:

```
from snorkel.slicing import slicing_function

@slicing_function()
def short_comment(x):
    """Ham comments are often short, such as 'cool video!'"""
    return len(x.text.split()) < 5

sfs = [short_comment]
```

The code is easy to understand. You will notice that `Short_comment` is a heuristic and will not cover all the cases. For such cases, you use the model.

These are the currently available packages in Snorkel:

- Snorkel analysis package
- Snorkel augmentation package
- Snorkel classification package
- Snorkel labelling package
- Snorkel map package
- Snorkel preprocess package

We are almost done and left with training a classifier. Recall that the Snorkel objective is to build training data. Once the training data is ready, you can use your favorite ML/DL framework to train the model:

```
train_text = df_train_augmented.text.tolist()
X_train = CountVectorizer(ngram_range=(1, 2)).fit_transform(train_text)


clf = LogisticRegression(solver="lbfgs")
clf.fit(X=X_train, y=df_train_augmented.label.values)
```

The preceding code is a standard scikit code and it works seamlessly with the programmatically labelled data. Here is a wonderful list of data-labelling tools for various data types in case your data has no text:

https://github.com/heartexlabs/awesome-data-labeling

2.10 Data versioning

Figure 2.48

Ask any software engineer and they can talk at length about version control. In fact, one won't be considered a serious developer if they don't know about version control. In the context of ML, a deployed model is code plus data. While we already have a robust toolset for code like GitHub, how do we version control data?

When a model goes into production, it needs to be versioncontrolled but the data that is used to train that model needs to be version controlled as well. In the absence of a version control, how would you rollback an erring model? The objective of this section is to make you aware of code + data versioning strategies using DVC (https://dvc.org/). DVC is an open-source version control system for machine learning projects.

Let's build a maturity model for how data version is controlled:

Level	Description
0	There is no versioning. You train a model and put it in production. For every new training, you use whatever training data you have.
1	The basic snapshot versioning is in place. If you have seen date-wise data folders, then you already understand the level 1 maturity. I have seen projects where Gitcommit labels were used as folder names. This offers a basic mapping between the code and data but is quite manual, and hence, prone to mistakes.
2	In this, data is versioned as a mix of code and data. Rather than creating data copies, run-wise data reference files are created, and they are checked-in along with code.

Table 2.2: *Data versioning maturity model*

Since Level 2 is the ideal state, let's take a look at an example to illustrate the functioning of it. Image you have your files in amazon s3 with unique ids. For each run or training session, we can create JSON, XML, or csv files with all the ids used. This represents the reference snapshot of training data without creating data copies. The JSON, XML, or CSV will then be checked-in along with the code. At any point, we can rebuild the model using the data id mentioned in the JSON, XML, or CSV files.

You already have a partial taste of it as you have participated in Kaggle competitions. There, physical files are dumped inside a folder, and a CSV contains the mapping of files and corresponding classes. Did you know that Git-lfs (https://git-lfs.github.com/) can store large JSON or CSV files? You would agree that this is a clean way to manage the data versions and we recommend using it.

DVC

By now, you already know that DVC is an open-source project for managing data versions. If you know Git, then working with DVC is much easier. DVC works with Git and they are often in sequence. If Git is about code then DVC is about data and model.

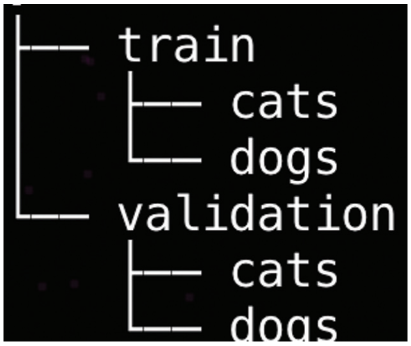
Like Git, DVC can store code locally and on a hosting service. GitHub is a popular hosting service for Git. DVC supports most major cloud providers including AWS, GCP, Azure and you can set up a DVC remote repository in your intranet or local machine. Again, drawing parallel from Git, in a common Git workflow style, you get a local copy of the remote repository. You can modify the files and then upload your changes to share with the team members.

When you store your data and models in the remote repository, a .dvc file is created. A .dvc file is a small text file that points to your actual data files in remote storage.

The simplest and fastest way to install DVC is via PIP. For the code walkthrough, we will use the Git clone https://github.com/iterative/example-versioning.git. This contains a small dataset of cats and dogs. Follow these steps to set up:

1. Git clone https://github.com/iterative/example-versioning.git. This needs to run on command line.
2. `cd example-versioning`. You need to be inside the `example-versioning` folder.
3. dvc get https://github.com/iterative/dataset-registry tutorial/ver/data.zip. This downloads the data.
4. `unzip data.zip.`

5. `rm -f data.zip`. Post-extraction, your data folder would look as follows:



```
├── train
│   ├── cats
│   └── dogs
└── validation
    ├── cats
    └── dogs
```

Figure 2.49: *Data setup*

6. Change to the parent folder (`example-versioning`) and initialize the DVC using `dvc init` command. This is similar to `git init`. DVC initialization create a `.dvc` folder that holds configuration information just like the `.git` folder for Git. This folder is not modified manually but we will look inside to understand DVC.
7. In case you want to block the anonymous usage collection, then run the following command: `dvc config core.analytics false`
8. As discussed earlier, DVC needs a remote repository. You can add a remote repository via this command: `dvc remote add -d remote_storage ../remote_data`. Note that for our purpose, `remote_data` is a folder created at the same level as example-versioning.

   Hint – Remote data could be local or a cloud file system.
9. To version control files, it needs to be tracked. You can use DVC add command to track files.
   a. DVC add data/train/
   b. DVC add data/validation/

   DVC under the hood
   a. Adds the train/ and validation/ folders to .gitignore.
   b. This creates two files with the .dvc extension, train.dvc, and validation.dvc.
   c. This copies the train/ and val/ folders to a staging area.

But why these steps?

a. By adding the train and validation folders, DVC ensures that large images files are not copied to GitHub.

b. `.dvc` files are small configuration files that contain information to track files in DVC. In our case, we have separate DVC files for train and validation each. These are YAML files which can be created and edited easily. If you look into DVC files right now, you will see following entries:

```
outs:
- md5: 023be85d0ac831e6a338227ada969c4c.dir
  path: train
```

A . dvc file contains the following fields:

   i. **Outs –** An output entry in turn contains the following fields:

      a. `path` – Path to file or directory

      b. `Deps` – Check the documentation here: https://dvc.org/doc/user-guide/dvc-files-and-directories#dvc-files

      c. `Wdir` - Check the documentation here: https://dvc.org/doc/user-guide/dvc-files-and-directories#dvc-files

      d. `md5` - Hash value for the file or directory being tracked with DVC. Any small change in the file would change the hash. This is a robust way to track changes. Along with the changes, if the files have same hash, then only the copy is maintained.

      e. `meta` - Check the documentation here: https://dvc.org/doc/user-guide/dvc-files-and-directories#dvc-files

c. DVC copies the actual images files to the staging area called cache. But when and where is this cache created? When you run the DVC init command, it creates a hidden folder called.dvc (ls -a shows hidden folder). If you look inside the .dvc folder, you would find three folders and one file which are as follows:

   i. `cache` – When you run the DVC add command, it copies all the images here.

   ii. `config` – step 8 and 9 entries coming in the config file.

   iii. `Plots` – This would contain the built-in plot templates. It contains templates for "linear", "scatter", "smooth" and confusion matrix. Please read the documents for details.

   iv. `tmp`

10. It is time to add the `.dvc`, .gitignore files to Git. Remember, the small files like `.dvc` and the code go into Git but the large image files go to DVC cache. It is worth reiterating that while Git takes care of the code and the actual data metadata, DVC takes care of the larger file.
11. Imagine that you trained a model and saved the final weights in model.h5 file.
12. Again, this needs to be tracked in DVC because this is a large file. Recall the thumb rule: large file in DVC and metadata in Git. In order to track the model file, you add it via the DVC add command which generates the corresponding `.dvc` file. You add the model.h5.dvc to Git.
13. Until now, the data has been in cache. To push the data from cache to remote, DVC offers the push command: DVC push. Can you recall the Git command to push changes to remote? Git push!
14. Next is a visual representation of the pipeline. To understand the DVC data versioning, you need to imagine DVC and Git side-by-side as follows:

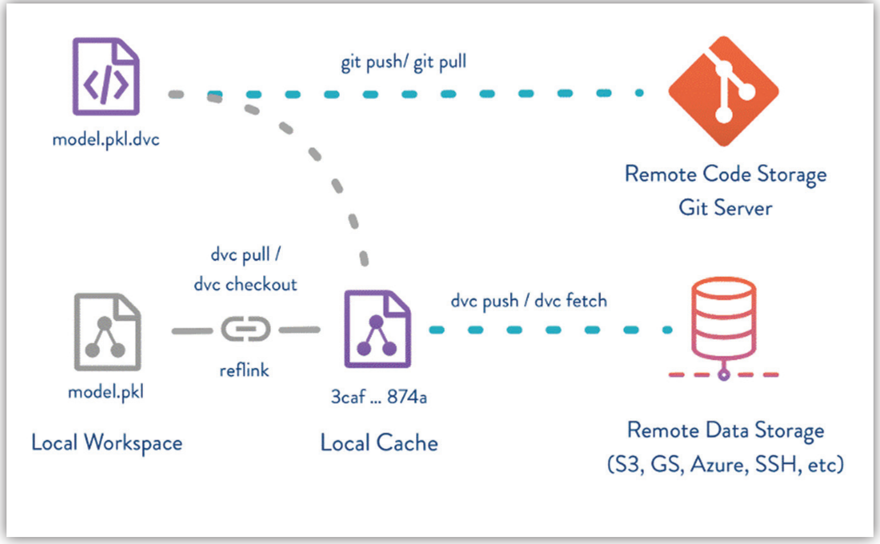


Figure 2.50: *DVC vs Git pipeline (image source - https://dvc.org/)*

Next is step 9 for `model.pkl`. The step output adds the `pkl` file to gitignore and generates the corresponding `.dvc` file. You then add Git and push the .dvc file to the remote server. Do you realize that the data and the model have been versioned now?

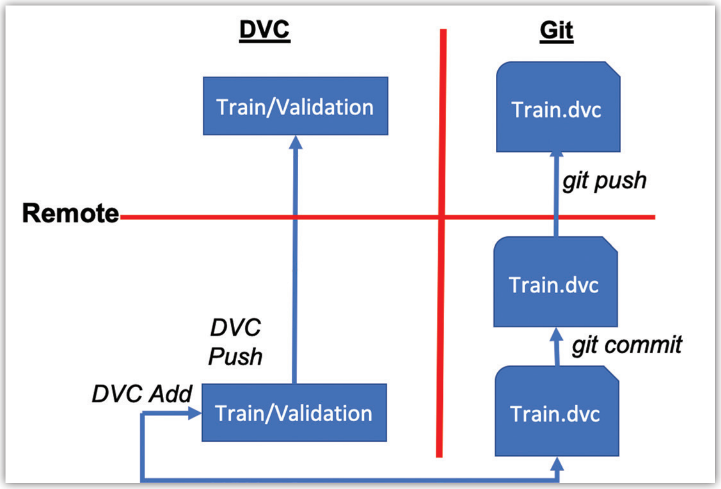


Figure 2.51: *Data and metadata movement across commands.*

15. In case your teammates want to use the above versioned data, they need to do a Git pull first. The Git pull will fetch the `.dvc` file and then they will have to use the DVC `fetch` command to fetch the actual images – `dvc fetch data/train/train.dvc`.
16. Let's say that you would like to add new data to your training and validation set, which you can actually do by downloading the new data - dvc get https://github.com/iterative/dataset-registry tutorial/ver/new-labels.zip
17. Add the new data with dvc commit command and repeat the cycle.
18. So, how do you switch between versions? Well, a picture is worth a thousand words as follows:

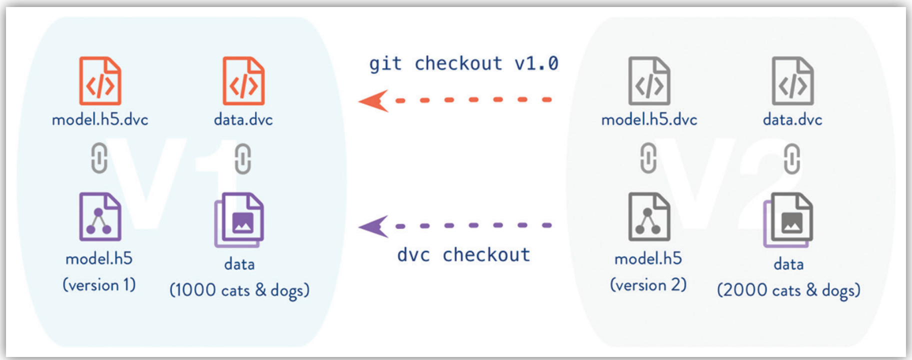


Figure 2.52: *DVC checkout process*

Let's now switch gears to multi-user scenarios. A lot of times, due to large file size, it is not practically possible to store it on the local machine. The solution is to maintain the cache at the high-capacity machine. By taking the cache off from the local machine, you make it shareable.

Recall that by default, cache is created under the `.dvc` folder at the time of DVC initialization (DVC init). To change the default cache location, use the dvc cache command `dvc cache dir path/to/your/shared-cache`. You also need to set group permission on the created or downloaded cache files as follows:

```
dvc config cache.shared group
```

Unsurprisingly, all the preceding details are maintained in the `.dvc/config` file. Configs are important and should be added to Git. With the cache moved to a shared location, it is time to change it to a cloud storage. This is straight-forward. The following are the commands for the Amazon s3 bucket. Add S3 remote to be used as the cache location for s3 files:

```
dvc remote add s3cache s3://mybucket/cache
```

Ask DVC to use the s3cache remote as the s3 cache location:

```
dvc config cache.s3 s3cache
```

Add data on s3 directly:

```
dvc add s3://mybucket/mydata
```

Create the stage with external S3 output:

```
dvc run -d data.txt -o s3://mybucket/data.txt aws s3 cp data.txt s3://
mybucket/data.txt
```

The following figure shows a visual representation of what we have set up so far:

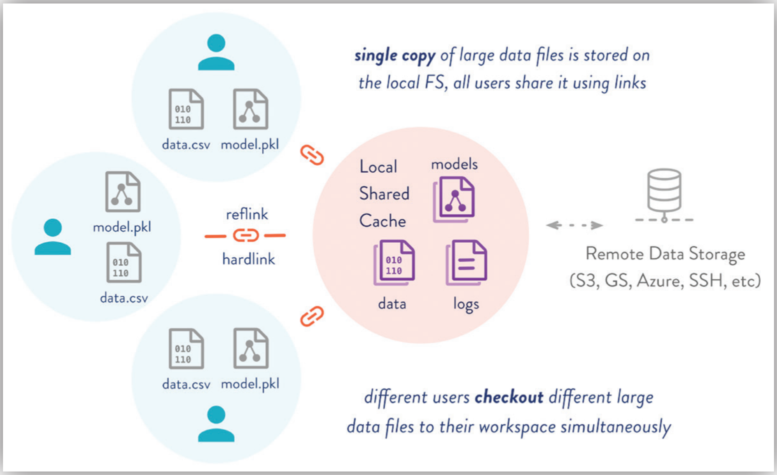

single copy of large data files is stored on the local FS, all users share it using links
data.csv
model.pkl
model.pkl
data.csv
reflink
hardlink
Local Shared Cache
models
data
logs
Remote Data Storage (S3, GS, Azure, SSH, etc)
data.csv
model.pkl
different users checkout different large data files to their workspace simultaneously

Figure 2.53: *DVC setup for team. Image source documentation*

This brings an end to the DVC tour. With DVC, experiments become reproducible and traceable.

Conclusion

Data collection and preparation are the foundations for trusted machine learning/deep learning models. A considerable amount of effort is spent at this step. With all the tools that you have learned about in the chapter; you can now not only do it effectively but make it repeatable as well. You learned quite a few tools in the chapter and that may feel overwhelming. It is okay to feel that way. The best way to build confidence is to write the code yourself. See if you can recall the code from memory.

In the next chapter, you will learn the best practices and tools for building effective architecture for your project. Until then, happy learning!

Points to remember

- Unlabelled, unbalanced data can throw off the analysis. Before fixing the issues, it is important to identify what and where to fix. Facets is an open-source tool to explore the data. It has two parts - overview and dive.
- Missing data can have a significant effect on the conclusions that can be drawn from the data. We learned different techniques like deletion and imputation for handling missing data. We also went through Autoimpute that can impute missing values using various strategies.
- Algorithms like k-means clustering, nearest neighbor methods, radial basis function (RBF) kernels, and anything that uses the Euclidean distance are sensitive to magnitude, and hence, the features for these algorithms should be normalized beforehand. We also learned different numerical data scaling techniques.
- Feature extraction or engineering, also known as feature creation, is the process of constructing new features from existing data to train a machine learning model. Featuretools can automate the process of new feature creation.
- It is a known fact that the maximum spent time in an ML project is not on modelling but on data collection and pre-processing. You have learned different sources for data collection and data generation. We looked at how Augmentor can automate the process of image data generation.
- Dask enables distributed pandas and lumpy features. It allows parallel execution for speedy and scalable data processing.

- We explored data distribution through two frames - storage and training/validation distribution. We learned the challenges of data-mismatch and the various strategies to handle it.
- Labelled data is one of the most intensive tasks. We explored how Snorkel can help to automate that task.
- For traceability and reproducibility, data versioning is required. We learned how DVC, along with git, can solve the problem of data versioning. We learned the strategies to scale DVC from a single machine to cloud storage.

Questions

1. What are the two parts of facets modules? Could you describe the purpose of each module?

   Hint – Check the facets section.
2. What are the two imputers available in the Autoimpute tool?

   Hint – Check the facets section.
3. Try creating features for https://www.kaggle.com/heeraldedhia/groceries-dataset using featuretools.

   Hint – Check the featuretools section and replace the example dataset with the groceries dataset.
4. Try training an iris species classifier using Dask powered logistic regression. Dataset - https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html

   Hint – Refer to the Dask-ML section.
5. **Set up a Google cloud storage for your DVC setup.**

   Hint - https://dvc.org/doc/user-guide/setup-google-drive-remote#setup-a-google-drive-dvc-remote

Chapter 3
Building Your ML Architecture

Building the ML architecture is not about using the latest most popular and viral algorithm to build the model. It's about training a model that meets the real-world expectations and challenges. The focus of this chapter will be on learning the best practices, tools on algorithm selection, hyperparameters initialization/tuning, and debugging techniques to enhance the model performance.

Structure

In this chapter, we will discuss the following topics:

- Building intuition for algorithm
- Hyperparameters selection and optimization using Keras Tuner
- Transfer learning for computer vision problems
- Model ensembling using ML-ens

Objective

After studying this chapter, you will learn the following:

- Building intuition for selecting the algorithms and learning the way to achieve model-problem fit.

- Selecting hyperparameters and optimizing using Keras Tuner.
- Transfer learning for computer vision problems.
- Transfer learning for NLP.
- Ensemble model using ML-ens.

3.1 Building intuition for algorithms

How does it feel when you have a bunch of tools but are not sure when and where to use them? *Figure 3.1* depicts this as follows:

Figure 3.1: *Image credit and source (Creative common license https://pixabay.com/)*

You can map this to the machine learning world where you know a bunch of algorithms but are unsure about their suitability for different problems. The objective of this chapter is to help you build intuition for model-problems fit. This knowledge would help you select a reasonable starting point. The trick to building the intuition is to look at different ML grouping strategies. Strategies generally map well to different problem domains. Let us explore a few strategies.

3.1.1 How algorithms learn?

A common way is to group them based on how are learned as follows:

- **Supervised learning** – Train the model using labelled data. Supervised learning is typically done in the context of classification, when we want to map input to output labels; or regression, when we want to map input to a continuous output.

- **Unsupervised learning** - In unsupervised learning, we have less information about objects. In particular, the train set is unlabeled. Two common use-cases for unsupervised learning are exploratory analysis and dimensionality reduction
- **Semi-supervised learning** - They use labelled and unlabeled data. It is helpful for those who cannot afford labelling their data. The technique involves a small portion of labelled examples and many unlabeled examples from which a model must learn and make predictions on new examples.
- **Reinforcement learning** – The idea of RL is to help learn a software agent (train a model) using a carrot and stick (reward and punishment) system. The system rewards desired actions and punishes undesired one. Reinforcement learning differs from supervised learning in the way that in supervised learning, the training data has the answer key with it so the model is trained with the correct answer itself, whereas in reinforcement learning, there is no answer but the reinforcement agent decides what to do to perform the given task. In the absence of a training dataset, it is bound to learn from its experience.

One important takeaway from this grouping is to consider your domain data availability as well for selecting the appropriate ML algorithm.

3.1.2 What do algorithms learn?

It is also possible to group them based on what they learn. They can be grouped into parametric and non-parametric based on what they learn as follows:

- **Parametric models** have a fixed set of parameters. They capture all its information about the data within those fixed parameters. In simple terms, to predict an unknown value, all that is needed are the parameters. For example, in the case of a linear regression with one variable, you have two parameters—the coefficient and the intercept. Knowing these two parameters will enable you to predict a new value.

  A few examples of parametric models are as follows:

  o Linear regression

  o Linear SVM

  o Logistic regression

  o Naive Bayes

  o Perceptron
- **Non-parametric** algorithms, on the other hand, use a flexible number of parameters. This number grows as it learns more from the data. They

are quite useful in situations in which you have a lot of data but no prior knowledge of them.

A flexible number of parameters makes it computationally slower, but then very few assumptions are made about the data.

A few examples of non-parametric models are as follows:

- o Decision trees
- o KNN
- o SVM with Gaussian Kernels
- o DNN

In case you haven't noticed, the trade-offs between parametric and non-parametric algorithms are in computational cost and accuracy.

3.1.3 Algorithm use cases

Another way to group them is based on use cases. This is quite useful since by knowing the use cases, you can understand any algorithm at a high level even if you haven't used it. Do note that this grouping is not perfect, and some algorithms can fall in multiple categories.

Also, the examples are not exhaustive as the intent is to give an intuition.

- **Regression algorithms:** Regression model predicts dependent values based on independent variables. It is mostly used for finding out the relationship between variables and forecasting.

  The following are few examples of it:

  - o Linear regression
  - o Logistic regression

- **Instance-based algorithm:** Instance-based learning is a family of learning algorithms that, instead of learning via generalization, compares training data instances with new data instances.

  The following are few examples of it:

  - o kNN
  - o SVM

- **Tree algorithms:** Decision tree methods build a model based on decisions made on actual attribute values. Think of a tree on which each node represents a feature, each link (branch) represents a decision (rule), and each leaf represents an outcome.

The following are few examples of it:

- o ID3
- o C4.5
- o CART

- **Bayesian algorithms:** As the name suggests, Bayesian algorithms apply the Bayes' theorem for classification and regression problems.

  The following are few examples of it:

  - o Naive Bayes
  - o Gaussian Naive Bayes
  - o Multinomial Naive Bayes

- **Clustering algorithms:** Clustering, as the name suggests, groups the data points. It should not come as a surprise that the data belonging to a group will have certain common features.

  Given a set of data points, we can use a clustering algorithm to classify each data point into a specific group.

  The following are few examples of it:

  - o K-means
  - o Hierarchical clustering

- **Dimensionality reduction algorithms:** Dimensionality reduction algorithms reduce the number of features by extracting a set of principal variables. It can be divided into feature selection and feature extraction.

  - o Principal Component Analysis (PCA)
  - o Linear Discriminant Analysis (LDA)

- **Ensemble algorithms:** Ensemble algorithms are also called meta-algorithms because they combine several machine learning techniques into one model to improve the prediction. Consider using this technique when you have to combine the predictions of multiple smart models.

  The following are few examples of it:

  - o Bootstrapped aggregation (Bagging)
  - o AdaBoost
  - o Weighted average (Blending)
  - o Stacked generalization (Stacking)
  - o Gradient boosting machines (GBM)

- **Deep learning algorithms:** Deep learning algorithms are essentially large neural networks. Deep neural networks essentially obviate the need for explicit feature engineering.

The following are some examples of it:

o **Convolutional Neural Network (CNN)**

o **Recurrent Neural Networks (RNNs)**

o **Long Short-Term Memory Networks (LSTMs)**

Figure 3.2 is a quick cheat-sheet that shows a visualization of different groupings:

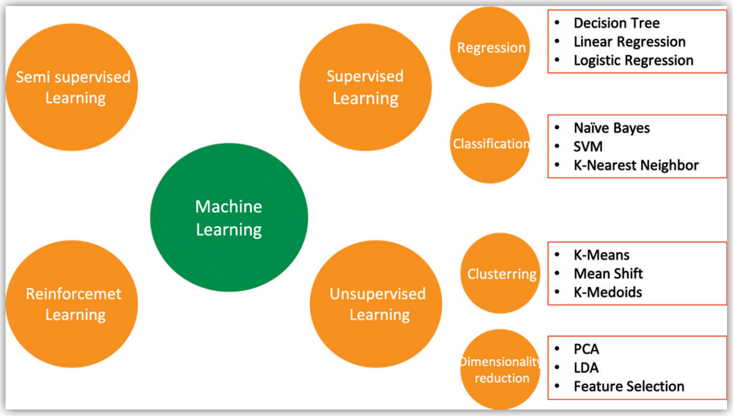


Figure 3.2: *ML algorithm types*

3.2 Hyperparameters selection and optimization using Keras Tuner

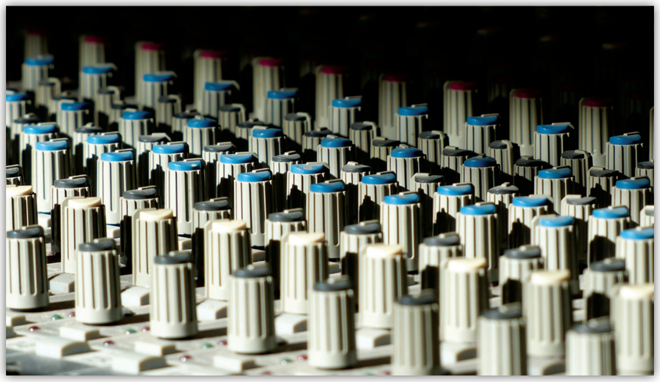


Figure 3.3: *Creative common license. The paradox of choices*

Deep learning models have a lot of tunable hyperparameters and finding the best configuration values for each of the hyperparameters in a high dimensional space is a difficult task. It is important to note that the best hyperparameter value is not just about finding the best score but also about ensuring that the best score is found with minimal resources like compute, money, and time.

The objective of this section is to help find a few strategies to search the hyper parameters configurations efficiently and automatically as much as possible. Before we deep dive, let us expand our definitions of parameters as follows:

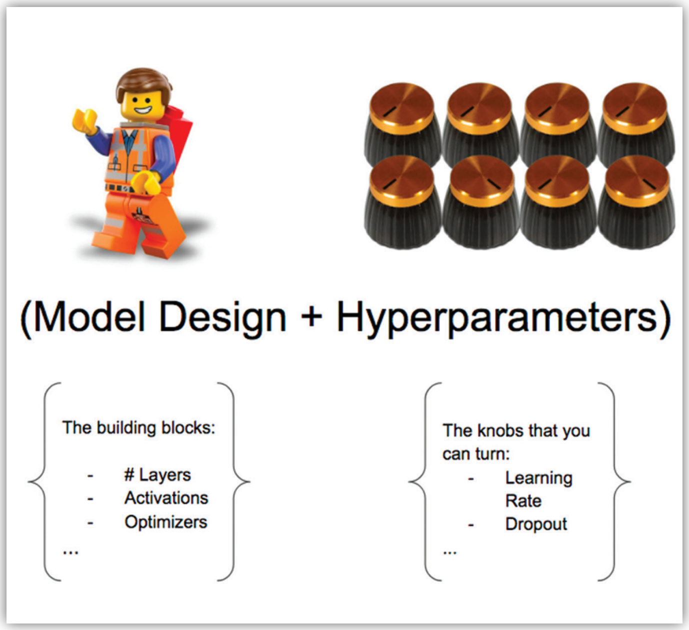


Figure 3.4: *Model building blocks (Image source - https://blog.floydhub.com/)*

To optimize our network, we not only need to consider hyperparameters like learning rate, epochs, etc., but also the model building blocks like layers, activations, and optimizers. Let us go through a few strategies.

3.2.1 Grid Search

Grid-searching is the process of searching the hyperparameter space to compute the optimal parameter value. It evaluates the model for every parameter combination. However, the naive way of trying every combination makes it quite computationally expensive.

Figure 3.5 illustrates the algorithm as follows:

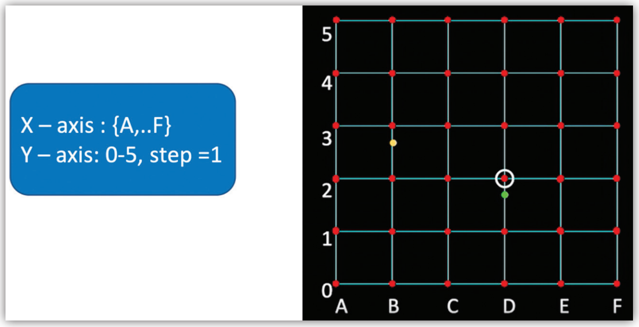


Figure 3.5: *Grid search*

Each dimension of the grid maps to a hyperparameter. The axis range maps to the range of hyperparameter values we want to try. Try all the possible parameter combinations and wait for the results to establish the best one.

3.2.2 Random Search

Random search tries to address some of the grid search problems. Random search was proposed in this paper: http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf by Bergstra and Bengio.

The difference between grid and random search is the step 1 that we saw in the grid search. The random search picks the point randomly. *Figure 3.6* explains this point:

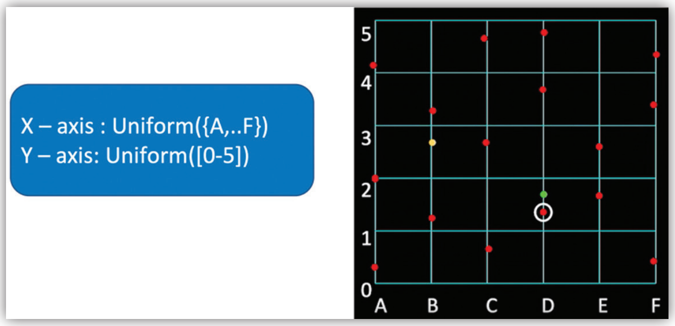


Figure 3.6: *Random Search*

Random combinations of the hyperparameters are used to find the best solution for the built model. *Figure 3.7* is another visualization of random search that drives home the idea of random search.

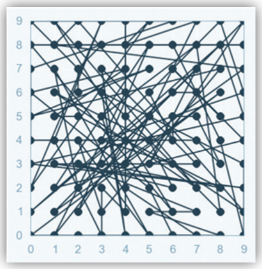


Figure 3.7: *Random search*

You may wonder - does this random combination work? Surprisingly, it does. Let us compare grid search and random search side-by-side as shown in *Figure 3.8*:

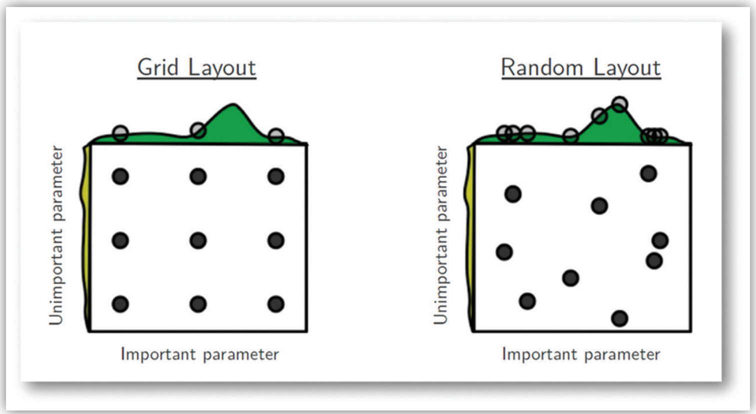


Figure 3.8: *Grid vs. random search*

We have 3×3 set of parameters and let us say that there are about nine set parameters in each search technique. With grid search, only three values per hyperparameter get tested. However, with random search, all nine trials explore distinct values of parameters.

While techniques like grid search or random search offer an automated way to search for optimal hyperparameter values, they have big flaws. Grid search cannot optimize models with more than four dimensions due to the curse of dimensionality. While random search is more capable than grid search, it's naive approach is time-consuming and expensive. Furthermore, it is unreasonable beyond 10 dimensions. Essentially, each new guess is independent from the previous run. We will now look at the Bayesian optimization that solves the grid and random search limitations to a great extent.

3.2.3 Bayesian optimization

Bayesian optimization is roughly an automated hand tuning the search keeping in mind the past results. It was initially popularized to break free from grids. Technically speaking, it is about forming a probabilistic model mapping hyperparameters to a probability of a score on the objective function. Basically, we want to model the loss function from hyperparameters. In literature, it is called surrogate model. We will come back to surrogate in a moment.

In simple terms, we want to find the model hyperparameters that yield the best score on the validation set metric without running the training for each hyperparameter combination. Let us build the intuition around the steps.

You already know that Bayesian optimization keeps track of past evaluations while choosing the hyperparameter set to evaluate next. How does this help? Choosing its parameter combinations based on past results helps in identifying and focusing on those parameter space areas that will give the most promising validation scores. This means that less iterations would be required to get to the optimal set of hyperparameter values. You know that less iterations means less time!

There are three building blocks of Bayesian optimization. These are as follows:

1. Parameter search space
2. An objective function
3. A surrogate (selection function)

The steps of Bayesian optimization algorithm are described below:

1. Select a combination of hyperparameter values and train the machine learning model with it.
2. Compute the score of the model.
3. Based on the score, pick up the next hyperparameter combination
4. Terminate when a stopping criterion is met.

Let us illustrate this with an example (read it row wise) as shown in *Figure 3.9*:

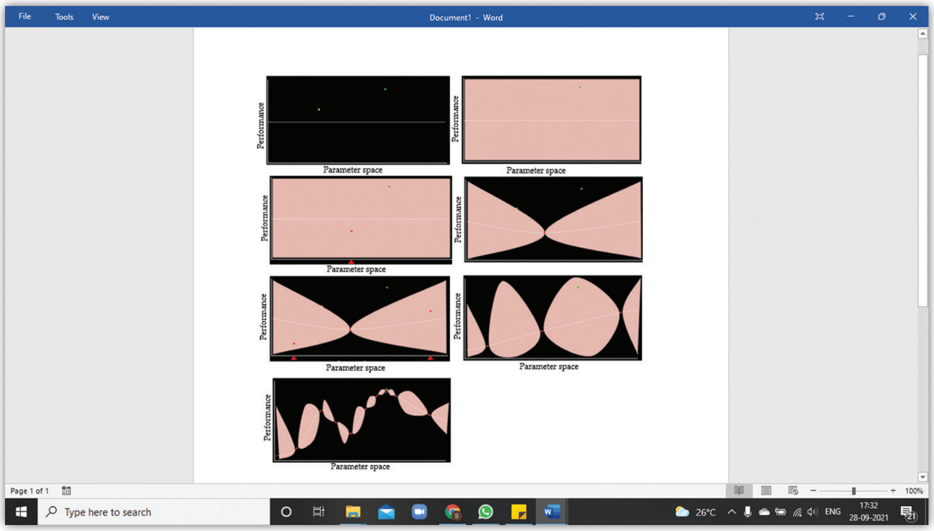


Figure 3.9: *Bayesian Optimization iteration*

In the first figure in *figure 3.9*, we have a uni-dimensional parameter space on the x-axis and performance on the y-axis. Higher performance (y-axis) is better. Green is the optimum point. If it helps, we can imagine that our goal is to mine for gold in an unknown land and that point green is the place having the highest gold deposits. We want to find that place with the minimal number of drillings as it is a costly process.

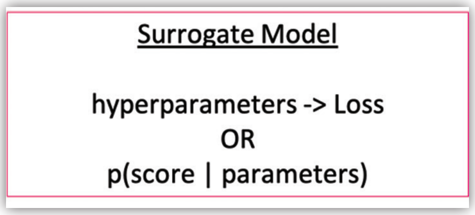
At first, we know nothing. This pipeline is our estimate of how each point in the high parameter space will affect our performance and the pink is our uncertainties. This is the second image in the first row. Since we know nothing, we can pick any point in the parameter space, fit a model, and evaluate. The red point is that selection in the second row, first figure. Now, we can adjust the uncertainty area which is the model of that loss. The second column image in the second row shows that adjustment. The rest of the images are the iterations of the steps followed until we reach the optimal point.

Let us go back to the building blocks of Bayesian optimization.

The objective function here evaluates the hyperparameter combinations. Examples of objective functions are accuracy, root mean squared error, and so on. We are sure that you would have guessed by now what it does. It takes in a set of hyperparameters and outputs a score that indicates how well a set of hyperparameters performs on the validation set.

It is not practical to evaluate each set of hyperparameter combination by means of the objective function as you need to train a model each time. Hence, calling this evaluation should ideally be restricted to a minimum.

Surrogate is the probabilistic model for the objective function that you are trying to minimize or maximize. It helps in selecting the next hyperparameter combination to try. Combinations that perform best on the surrogate function are then forwarded to actual objective function for evaluation. Just to be clear—the surrogate model reduces the number of calls to the actual objective functions by choosing the promising ones. It would be fair to think of the surrogate function as an approximation of the objective function:



Surrogate Model

hyperparameters -> Loss
OR
p(score | parameters)

Figure 3.10: *Surrogate Model*

The common choices for surrogate model are as follows:

1. Gaussian process regression
2. Random forest regression
3. Tree-structured Parzen Estimator

There are two pitfalls of Bayesian method that you need to know.

1. Since Bayesian search is an informed search, it is sequential in nature, making it difficult to scale via parallelization.
2. Surrogate models have their own parameters and tuning them appropriately might become a challenging task.

Fortunately, you don't have to write the search algorithm from scratch. There are open-source library offering the strategies. As a side note, it is ok if you have the intuition of the algorithm but don't understand the algorithm implementation completely. The purpose of the libraries is to abstract those details. Remember, In order to use compiler optimizations you don't need to understand how that was written in the compiler.

3.2.4 Keras Tuner

Keras Tuner is a distributable hyperparameter optimization framework to perform a hyperparameter search. You can define a search space and then leverage algorithms like Bayesian Optimization, Hyperband, and random search to find the best hyperparameter values. Let us go through different algorithms to understand the library usage.

Random search:

```
--1

from tensorflow import keras

from tensorflow.keras import layers

from kerastuner.tuners import RandomSearch

(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_
data() --2

--3

X_train = X_train.astype('float32') / 255.

X_test = X_test.astype('float32') / 255.

--4

X_train = X_train[:10000]

y_train = y_train[:10000]

def build_model(hp):

    model = keras.Sequential()

    model.add(layers.Flatten(input_shape=(28, 28)))

    for i in range(hp.Int('num_layers', 2, 20)):  --4

       model.add(layers.Dense(units=hp.Int('units_' + str(i), 32, 512, 32),

                               activation='relu'))  --5

    model.add(layers.Dense(10, activation='softmax'))

    model.compile(

        optimizer=keras.optimizers.Adam(

            hp.Choice('learning_rate', [1e-2, 1e-3, 1e-4])), --6

        loss='sparse_categorical_crossentropy',

        metrics=['accuracy'])

    return model
```

```
tuner = RandomSearch(  --7

    build_model,

    objective='val_accuracy',

    max_trials=3,

    executions_per_trial=2,

    directory='mnist_dir')

tuner.search_space_summary() --8

--9

tuner.search(x=X_train,

             y=y_train,

             epochs=3,

             validation_data=(X_test, y_test))

--10

tuner.results_summary()

--11

models = tuner.get_best_models(num_models=2)

best_model = tuner.get_best_models(num_models=1)[0] --12

accuracy = best_model.evaluate(X_test, y_test) --13
```

Let us unpack the code as follows:

1. Import the necessary library. As a side note, this code used tensorflow 2.3.1. To install other packages, you may want to run the following commands:

   ```
   pip install keras-tuner
   pip install tensorflow_datasets
   ```
2. Download the mnist dataset and split it into train and test.
3. Normalize the data.
4. `build_model` function builds the model. Note the `hp` parameter. `hp` is an alias for Keras Tuner's HyperParameters class. When you build a model for

hypertuning, you also define the hyperparameter search space in addition to the model architecture. The model you set up for hypertuning is called a hypermodel

Keras Tuner offers the following ways to build a hyper model:

a. By using a model builder function.
b. By subclassing the HyperModel class of the Keras Tuner API.
c. By using two pre-defined HyperModel classes – HyperXception and HyperResNet for computer vision applications..

In the preceding code, we have used the model builder function. The network here is a simple network without any convolutions. The input is flattened and then a bunch of dense layers are added on top of it. You know that the number of layers affects the learnability and hence we need to tune it for best results.

The search space for the numbers of layer has been defined using the hp.init function. There are other hyperparameter functions like hp.choice. You can find the full list here: https://keras-team.github.io/keras-tuner/documentation/hyperparameters/.

In Keras Tuner, hyperparameters have a type (Float, Int, Boolean, and Choice) and a unique name. You can also specify certain other options to guide the search as follows:

a. **name:** Str. Name of parameter. It must be unique.
b. **min_value:** Float. Lower bound of the range.
c. **max_value:** Float. Upper bound of the range.
d. **step:** Optional. Float, for example, 0.1. Smallest. The smallest meaningful distance between two values. Whether the step should be specified is Oracle dependent since some Oracles can infer an optimal step automatically.
e. **sampling:** Optional. One of `"linear," "log," "reverse_log."` It acts as a hint for an initial prior probability distribution for how this value should be sampled; for example, "log" will assign equal probabilities to each order of magnitude range.
f. **default:** The default value to return for the parameter. If unspecified, the default value will be `min_value`.

5. We define the search space for each layer units (neurons), again using the `hp.int` function.

6. We specify the search space for optimizers.
7. It is time to define the tuner.
8. Keras Tuner offers the following search methods:
   a. Random search
   b. Hyperband
   c. Bayesian optimization

   The tuner parameters used are as follows:
   a. The model-building function.
   b. Objective to optimize (whether to minimize or maximize is automatically inferred for built-in metrics).
   c. The total number of trials (`max_trials`) to test. It represents the number of hyperparameter combinations that will be tested by the tuner.
   d. The number of models that should be built and fit for each trial. (`executions_per_trial`).

   The purpose of having multiple executions per trial is to reduce results variance and therefore be able to assess the performance of a model more accurately.
9. The utility function to view the search space.
10. Start the search for the best hyperparameter configuration. The call to search has the same signature as `model.fit()`. Under the hood, models are built iteratively by calling the model-building function, which populates the hyperparameter space (search space) tracked by the `hp` object. The tuner progressively explores the space, recording the metrics for each configuration.
11. Print the result summary.
12. Get the best models as determined by the objective when the search is over. The models are loaded with the weights corresponding to their best checkpoint (at the end of the best epoch of best trial).

    This method is only a convenient shortcut. For the best performance, it is recommended to retrain the model on the full dataset using the best hyperparameters found during search (`get_best_hyperparameters`).
13. Get the best model, that is, a single model.
14. Evaluate the model with the best hyperparameters on the test set.

Hyperband

Hyperband is an optimized version of random search which uses early stopping to speed up the process of hyperparameter tuning. To save the time, hyperband runs configures for once or twice to get an initial sense. Then it picks up the best performer and run them for longer. Deep learning algorithms and tree ensembles like gradient boosting are some good candidates for tuning with Hyperband. You don't need to fully understand the mechanics of hyperband to use it. However, if interested, you can check the paper here: (https://arxiv.org/abs/1603.06560) code-wise using hyperband is not much different from what you saw earlier.

```
h_tuner = kt.Hyperband(build_model,

                     objective = 'val_accuracy',

                     max_epochs = 10,

                     factor = 3,

                     directory = 'mnist_hyperband_dir',

                     project_name = 'hello_hyperband')

h_tuner.search(x=X_train,

             y=y_train,

             epochs=3,

             validation_data=(X_test, y_test))

best_hps = h_tuner.get_best_hyperparameters(num_trials = 1)[0]


h_model = h_tuner.hypermodel.build(best_hps)

h_model.fit(X_train, y_train, epochs = 10, validation_data = (X_test,
y_test))

# Retrieve the best model.

best_model = h_tuner.get_best_models(num_models=1)[0]

# Evaluate the best model.

accuracy = best_model.evaluate(X_test, y_test)
```

Bayesian optimization

The Hyperband tuning algorithm under the hood uses adaptive resource allocation and early stopping. For a small number of epochs, the algorithm trains quite a few models and carries forward only half of the well-performing models to the next round. The number of models to be trained is calculated by computing 1 + logfactor(`max_epochs`) and rounding it up to the nearest integer.

Again, code-wise, it is not going to be much different as follows:

```
bo_tuner = kt.BayesianOptimization(
                    build_model,
                     objective = 'val_accuracy',
                     directory = 'mnist_BO_dir',
                    max_trials=5,
                    seed=42,
                    executions_per_trial=2,
                     project_name = 'hello_BO'
        )
```

`#max_trials` is the Total number of model configurations to test.

```
bo_tuner.search(x=X_train,
             y=y_train,
             epochs=3,
             validation_data=(X_test, y_test))
best_bo = bo_tuner.get_best_hyperparameters(num_trials = 1)[0]
bo_model = bo_tuner.hypermodel.build(best_bo)
bo_model.fit(X_train, y_train, epochs = 10, validation_data = (X_test, y_test))
# Retrieve the best model.
best_model = bo_tuner.get_best_models(num_models=1)[0]

# Evaluate the best model.
accuracy = best_model.evaluate(X_test, y_test)
```

Tuning is required to extract the maximum value from the model. Tuning hyperparameters effectively comes with experience and libraries like Keras Tuner democratizes that knowledge for everyone.

3.3 Transfer learning for computer vision problems

Figure 3.11: *Image source:- Image src - https://aiplindia.com/*

We humans have the amazing ability to transfer knowledge across tasks. If you learn cycling, then riding bikes becomes easier. We essentially adapt our cycling skills to biking.

In the deep learning world, the ability to adapt a model trained on task A for task B is called transfer learning. Transfer learning has revolutionized the way in which DL networks are trained these days. It helps in overcoming the challenge of less data availability. With transfer learning, it is now possible to train a deep neural network even if you have less data. The lack of data is compensated by transferring the knowledge of some other pre-trained model. It is now proven that features learned from very large datasets, such as "imagenet," are highly transferable to a variety of image recognition tasks. We will look at a few strategies for transfer learning.

Before we explore the strategies, let us look at the typical structure of a CNN. We have stacks of convolutions and pooling layers with fully connected layers at the top. A picture would be helpful here. This is a CNN for image classifier task. Convolutions and pooling are stacked with two fully connected neural networks and a "softmax" layer at the top.

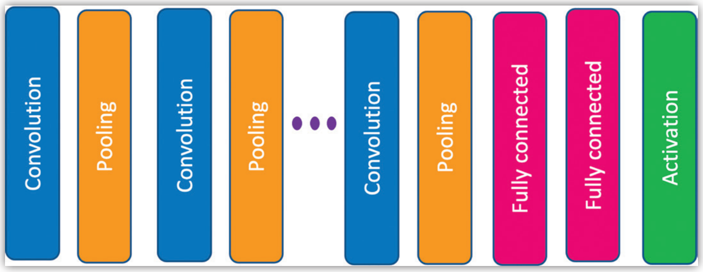


Figure 3.12: *CNN architecture*

We will use *figure 3.12* architecture as our starting point. Training the top layer is the easiest way to transfer the knowledge.

Training the top layer

We chop off the fully-connected layer from the already trained model and replace it with a randomly initialized one. Only the parameters in the fully connected layers are trained while keeping the other parameters fixed as shown in *figure 3.13*:

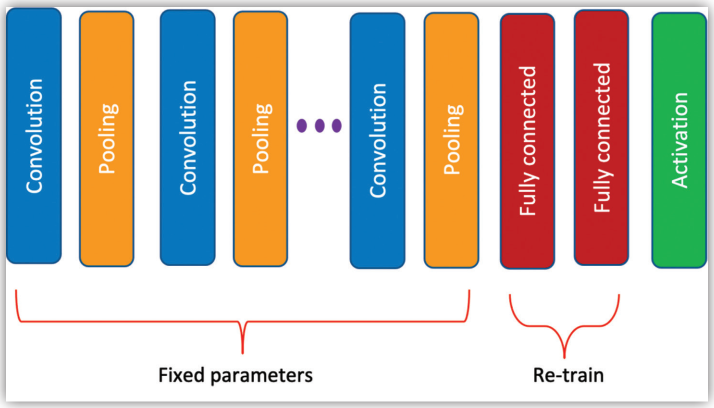


Figure 3.13: *Training the fully connected layer only*

This method treats the frozen layers as feature extractor, and we are essentially transferring the features to the new task. This is suitable when the data and tasks are like the data and tasks of the original model that they were trained on. If you have very little data for your task, then this is the only solution left for you.

Fine-tuning

In case you have more data, you can unfreeze the transferred parameters and train the entire network:

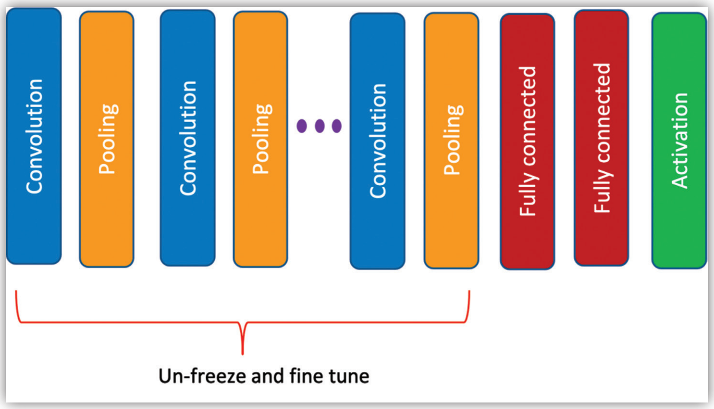


Figure 3.14: *Fine-tuning features*

In this mode, transferred parameters are used to initialize the network in the place of random initialization which gives our model a warm start and speeds up for convergence. To preserve the initialization from pre-training, it is a common practice to lower the learning rate by order of magnitude. It is a common practice to start with frozen parameters and training only the fully-connected layers. This prevents early changes to the transferred parameters. Once the randomly initialized layers or fully connected layers converge, then other parameters unfreeze and the entire network is re-trained.

This strategy really shines when you have less data for your task, but your task has a lot of similarities to some other pre-trained tasks. Before we explore other techniques, let us take a step back and understand why the transfer learning even works.

When a deep learning network is trained on a large dataset, the early layer parameters resemble each other irrespective of the task. The early layers tend to learn the edges, corners, textures, and patterns. These features are generic in nature and apply to a majority of the tasks. It doesn't matter whether it is a bird or a car, they will have the same features or parameters for edges and corners, etc.

The features become specific to tasks as we move towards the output layers. For example, a higher layer may be able to identify headlights of a car as shown in *figure 3.15*:

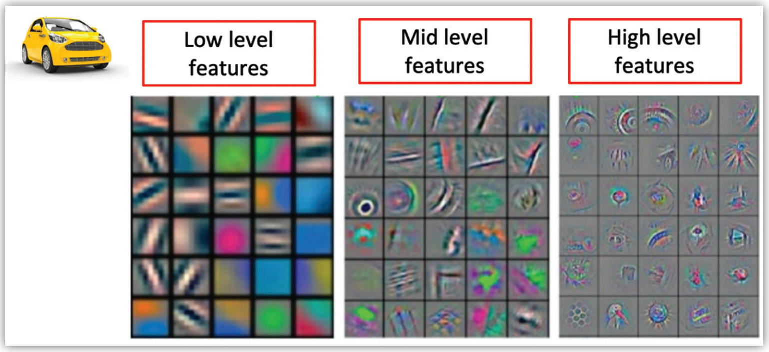


Figure 3.15: *(features visualization)*

Now that you know that layer parameters move from generic to specific, you can choose how many layers or up to what layers you want to transfer the weights. We can get creative here.

For example, rather than freezing, you can control the rate of change of weights across different layers.

Generalizing transfer learning

In general, it's possible to transfer the first n layers from a pre-trained network to the target network and randomly initialize the rest. *Figure 3.16* illustrates the n layers transfer idea as follows:

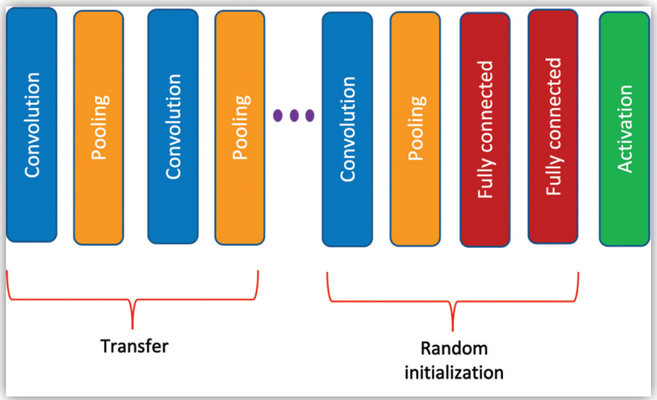


Figure 3.16: *Training in layers*

Recall that in the first strategy (training top layers – fully connected layers), we had frozen all the non-fully connected layers. But in *figure 3.16*, along with fully connected layers, some convolutional layers are being trained as well. Just to be explicit, the transferred layers are the frozen layers whose weights are reused.

Training the bottom layers

Technically, this is the reverse of the top layer(s) training strategy. Rather than randomly initializing the top layer(s), we can randomly initialize the first few layers and transfer the remaining ones.

Figure 3.17 illustrates this idea as follows:

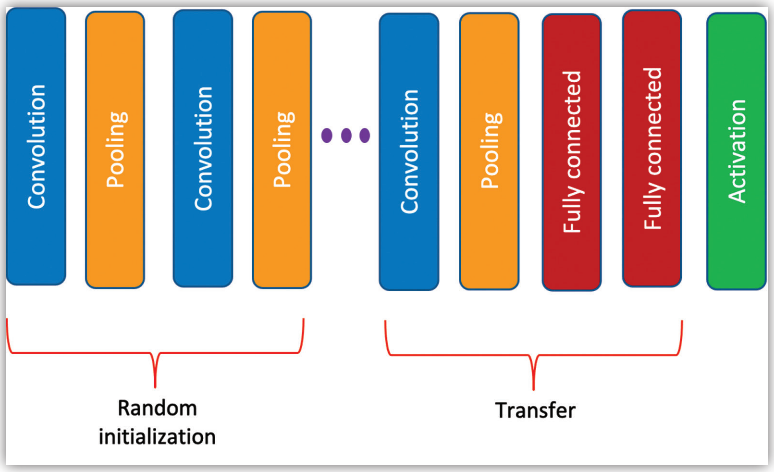


Figure 3.17: *(Bottom layers training strategies)*

But where is this required? Well, this is helpful when the network is the same but the input has changed. For example, let us say that the input image to an image classifier now includes the depth information channel. This means that we need to retrain the initial layers so that it can learn to highlight the edges, corners, and patterns from the inputs that have depth data. Note that for the later layers, nothing changes.

It is time to explore some of these strategies in code. We will train a dog and a cat classifier. The data for training and validation can be downloaded from Kaggle: (https://www.kaggle.com/c/dogs-vs-cats/data):

```
--1

from tensorflow.keras.applications.vgg16 import VGG16

import tensorflow as tf
```

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

--2

image_size = 150

train_dir ='data/train'

valid_dir = 'data/valid'

#Load the VGG model --3

vgg_conv   =   VGG16(weights='imagenet',   include_top=False,   input_
shape=(image_size , image_size, 3))

vgg_conv.summary() --4

--5

#Freeze all the layers. We only want to train the fully connected layers

for layer in vgg_conv.layers:

    layer.trainable = False

--6

# Check the trainable status of the individual layers

for layer in vgg_conv.layers:

    print(layer, layer.trainable)

#Create the model

model = tf.keras.models.Sequential() --7

#Add the vgg convolution block

model.add(vgg_conv) --8

# Add two fully connected layers

model.add(tf.keras.layers.Flatten()) --9

model.add(tf.keras.layers.Dense(1024, activation=tf.nn.relu)) --10

model.add(tf.keras.layers.Dense(2, activation=tf.nn.sigmoid)) --11

model.summary()
```

```
train_batch_size = 16.

valid_batch_size =10

train_datagen = ImageDataGenerator(  --12

      rescale=1./255)

test_datagen = ImageDataGenerator(rescale=1./255)


train_generator = train_datagen.flow_from_directory(

        train_dir,  # this is the target directory

        target_size=(150, 150),  # all images will be resized to 150x150

        batch_size=train_batch_size,

        class_mode='categorical')  --13

# this is a similar generator, for validation data

validation_generator = test_datagen.flow_from_directory(

        valid_dir,

        target_size=(150, 150),

        batch_size=valid_batch_size,

        class_mode='categorical') --14

#Compile the model

model.compile(loss='categorical_crossentropy',

              optimizer='rmsprop',

              metrics=['accuracy']) --15

history = model.fit_generator(

  train_generator,

  epochs=5,

  validation_data=validation_generator

) --16
```

```
model.save_weights('first_try_v2.h5')  # always save your weights after
training or during training --17

--18

acc = history.history['accuracy']

val_acc = history.history['val_accuracy']

loss = history.history['loss']

val_loss = history.history['val_loss']

# Unfreeze the layers except the last 2 layers --19

for layer in vgg_conv.layers[-2:]:

    layer.trainable = True

history = model.fit_generator(

  train_generator,

  epochs=5,

  validation_data=validation_generator

) --20

model.save_weights('second_try_v2.h5')  # always save your weights after
training or during training
```

Let us unpack the code as follows:

1. Import the necessary libraries.
2. We specify a fixed size for the images. Ensure that if you download the data to some other location, update the training and validation data folder path.
3. We are using a VGG16 model because it is light and runs easily on a non-GPU machine as well. VGG16 (also called as OxfordNet) is a convolutional neural network architecture named after the Visual Geometry Group from Oxford, who developed it. It was the winner of ImageNet competition in 2014. Even though it has been outperformed by Inception and ResNet, it is still a good model for learning and experimenting.
4. Use the summary method to view the architecture. Note that you won't see the top fully connected layers because during initialization it was set to false.
5. We are setting off the trainable flag to ensure that during training, weights are not updated. We only want to train the fully connected layers.

6. A check to ensure that the flag has been switched off. Think of this as sanity code that ensures that layers are set up the way that you want them to be.
7. It is time to initialize the model. We will use the Sequential model for simplicity flag.
8. The convolutions and the pooling layer from VGG16 are added to it. Recall that top layers are not present.
9. It is time to add the fully connected layer, and for that, we need to flatten it first.
10. Add a dense layer with Relu activation.
11. Add another final dense layer with only two outputs as we have only two classes – cats and dogs ¬– with sigmoid activation.
12. We are setting up ImageDatagenerator to generate the batch of images. Notice the standardization (division by 255).
13. It's no surprise that the ImageData generator will read the data from the folders as we have downloaded the images locally.
14. This is step 12 and 13 for validation set.
15. It is time to compile the model.
16. We are fitting the model with very small epochs to ensure that it runs quickly on CPU as well. The accuracy numbers are not so relevant with such a small number of epochs, but you get the full picture.
17. Post training, it is a good idea to save the weights.
18. In case you are interesting in plotting the metrics, then the relevant data is present in the history dictionary.
19. Now that the top layers are trained, it is time to open the convolution layers for training. It is important to take a step back and recall what we just did here – we first only trained the top layers, and then we trained the whole VGG16 layers. Now that you know how to freeze and unfreeze layers, trying other strategies should not be that difficult.
20. For the sake of completion, we are fitting the model again. Remember that convolution layers weight will be updated too this time.

Note: The preceding code has been written and tested with Keras 2.2.4

Model distillation

Till now, we have learned the different ways to transfer weights from a pre-trained network to a new network having the same shapes. But what about transferring knowledge between networks having different shapes?

Model distillation is a technique for transferring knowledge between models having different shapes. This transfer trick can also be viewed as a method for model compression in which a small model is trained to mimic a pre-trained larger model. In a lot of literature, you would find this referred to as "teacher-student" where the larger model is called the teacher and the small model is called the student as expected.

The key idea is that the soft activations of the output layer of a teacher network is more capable of learning. Think of soft activations as probabilities instead of Boolean or categorical values. The student will then learn from the teacher or mimic the pre-trained teacher model.

The intuition to remember here is that the output distribution of a large well-trained network captures extra information about the high-level features and its output informs the student to consider the dark knowledge available in the probability distribution. *Figure 3.18* shows the high-level architecture as follows:

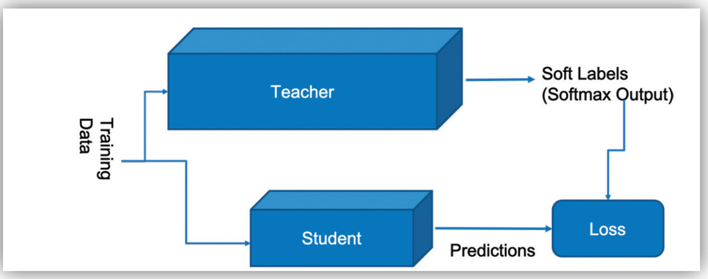


Figure 3.18: *Teacher-student architecture*

This approach is quite practical and intuitive. We need a large complex network during training time to discover the complex relationship and output probability distribution. Then, the student will try to replicate the output of the teacher's network.

Consider a tiger image that is labelled as one hot encoded data, which essentially assigns a value of 1 to the correct class and 0 to all other classes. By passing it through a softmax function that outputs probability distribution, you may get a probability of 80% for a tiger, and let's say 15% for a cat, and 5% for a dog. Rather than using the hardcoded label, using the soft labels (probabilities) helps the student network to pick the nuances and the grey area.

The idea was first presented in the 2015 paper "Distilling the Knowledge in a Neural Network" by Hinton et al. (https://arxiv.org/abs/1503.02531).

There are benefits worth remembering which are as follows:

- **Data flexibility:** It reduces the dependency on new data. Rather than assigning labels to new data, the teacher model can assign soft labels to it.
- **Model compression:** To learn well, the common strategy in neural networks is to stack more but complex and large networks requiring very high computing power. However, once the network is trained, you already have a reduced decision surface. Now you can use a smaller network that can learn faster from the reduced decision surface. The smaller network can then be deployed on mobile phones or laptops.

3.4 Transfer learning for NLP

A new era of NLP was started with the publication of word2vec in the public domain. Word vectors, or word embeddings as are popularly called, heralded the dawn of transfer learning in NLP.

With word embeddings, it was now possible to capture many different properties of a word and replace feature engineering for NLP tasks. *Figure 3.19* shows the vector representation and the relationship among words.

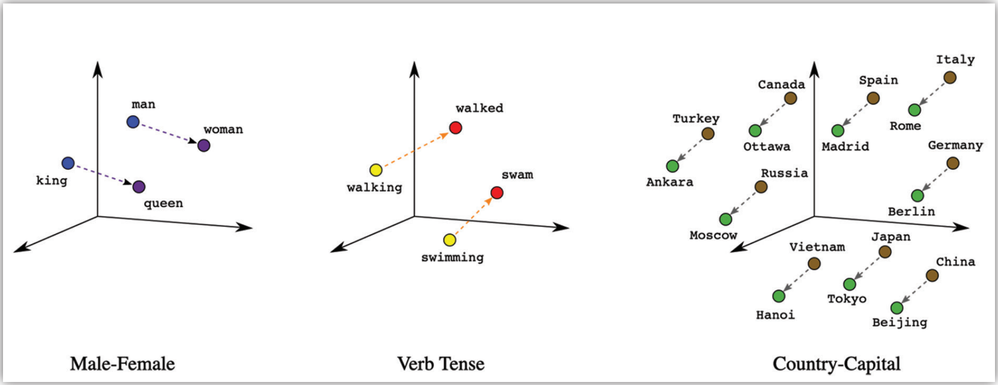


Figure 3.19: *Word vectors representation. Image source: https://developers.google.com)*

For the sake of simplicity, we can group vectors into three types. I also think of the three types as an evolution of the word vectors. These are as follows:

- **Word vectors:** This help to represent the words cat and dog in a standard vector format learned from a large text corpus.

  bank = [0.7, 0.4, -0.3]

  money = [0.6, 0.3, -0.2]

- **Sentence or doc vectors:** This is like a word vector but done for the entire sentence or document. The example clarifies the point that the vectors represent the entire sentence and not the collection of individual word vectors.

  Open an account in the bank = [0.2, .-1, -0.7]

  How is a river bank formed? = [0.8, .0, 0.3]

- **Word in-context vector:** This is again a word vector, but the same word can have different embedding depending upon the context.

  Open an account in the bank = [0.7, .-3, -0.4]

  How is a river bank formed? = [0.2, 0.1, 0.8]

While word2Vec started it all, beyond word2Vec, there are a few other popular embeddings as well.

Word2vec - It is powerful because we don't have to manually label that *"Einstein was a scientist," "Cricket is a sport,"* and *"India is a country."* Word2vec can learn and much more all its own. Our world is full of unlabeled, uncategorized, and unstructured natural language text.

GloVe (https://nlp.stanford.edu/projects/glove/) - GloVe uses an unsupervised learning algorithm to obtain vector representations for words but its training is very different from word2vec. GloVe doesn't use a neural network like word2vec. Even though word2vec first popularised the idea of word vectors, GloVe should be the preferred way to train you for a word vector model.

FastText (https://fasttext.cc/) - FastText is from Facebook and takes the concept from word2vec one step further. FastText trains a vector representation for every n-character gram which includes words, misspelled words, partial words, and even single characters. FastText breaks words into several n-grams. For instance, the tri-grams for the word apple are app, ppl, and ple. Let's say you don't have apple vector in your training dataset and you search for similar vectors to apple. Word2Vec would throw an error because apple is not there; however, fastText would be able to return some similar vectors due to partial n grams.

If you still have doubts about the connection between embeddings and transfer learning, then think of it this way – word2Vec, GloVe, or fastText are pretrained vectors. For your NLP tasks, you would leverage the pretrained vectors.

It is time to explore the concepts in code. We will build a sentiment classifier using Glove embeddings. We will use the dataset from the UCI Machine learning repository (https://archive.ics.uci.edu/ml/datasets/Sentiment+Labelled+Sentences). It contains sentences labelled with positive and negative sentiments.

```
import os

--1

raw_data_folder = './data/sentiment labelled sentences'

yelp_filepath = os.path.join(raw_data_folder,'yelp_labelled.txt')

import pandas as pd --2

df = pd.read_csv(yelp_filepath, names=['sentence', 'label'], sep='\t')

df.head() --3

--4

sentences = df['sentence'].values

y=df['label'].values

from sklearn.model_selection import train_test_split

#Split the data in train and test

sentences_train, sentences_test, y_train, y_test = train_test_split(

        sentences, y, test_size=0.25, random_state=1000)

--5

# Tokenize the texts

from keras.preprocessing.text import Tokenizer

tokenizer = Tokenizer(num_words=5000)

tokenizer.fit_on_texts(sentences)

#Our vocabulary

tokenizer.index_word --6

--7

X_train = tokenizer.texts_to_sequences(sentences_train)

X_test = tokenizer.texts_to_sequences(sentences_test)

# Adding 1 because of reserved 0 index for padding

 vocab_size = len(tokenizer.word_index) + 1 --8

# Pad the texts
```

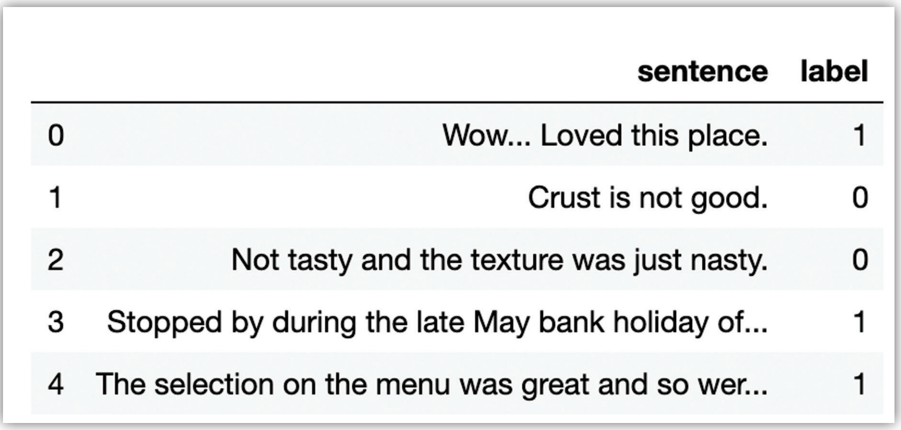
```
from keras.preprocessing.sequence import pad_sequences --9

maxlen = 100

X_train = pad_sequences(X_train, padding='post', maxlen=maxlen)

X_test = pad_sequences(X_test, padding='post', maxlen=maxlen)

# Download the embedding from here http://nlp.stanford.edu/data/glove.6B.
zip, put it inside the data folder and extract the data from it (unzip).

import numpy as np

#Since we have only a limited number of words in our vocabulary,

#we can skip most of the words in the pretrained word embeddings:

def create_embedding_matrix(filepath, word_index, embedding_dim): --10

   vocab_size = len(word_index) + 1  # Adding again 1 because of reserved
0 index

    embedding_matrix = np.zeros((vocab_size, embedding_dim))

    with open(filepath) as f:

        for line in f:

            word, *vector = line.split()

            if word in word_index:

                idx = word_index[word]

                embedding_matrix[idx] = np.array(

                    vector, dtype=np.float32)[:embedding_dim]

    return embedding_matrix

embedding_dim = 50

embedding_matrix = create_embedding_matrix('data/glove.6B/glove.6B.50d.
txt',

                                    tokenizer.word_index, embedding_dim)

embedding_matrix.shape

nonzero_elements = np.count_nonzero(np.count_nonzero(embedding_matrix,
axis=1))
```

```
nonzero_elements / vocab_size

# Build the network

import tensorflow as tf

from keras import layers

--11

model = tf.keras.models.Sequential()

model.add(tf.keras.layers.Embedding(vocab_size, embedding_dim,

                           weights=[embedding_matrix],

                           input_length=maxlen,

                           trainable=False))

model.add(tf.keras.layers.GlobalMaxPool1D()) #To downsample the incoming
feature vectors

model.add(tf.keras.layers.Dense(10, activation='relu'))

model.add(tf.keras.layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='adam',

              loss='binary_crossentropy',

              metrics=['accuracy'])

model.summary()

history = model.fit(X_train, y_train,

                    epochs=10,

                    verbose=False,

                    validation_data=(X_test, y_test),

                    batch_size=10) --12

loss, accuracy = model.evaluate(X_train, y_train, verbose=False) --13

print("Training Accuracy: {:.4f}".format(accuracy))

loss, accuracy = model.evaluate(X_test, y_test, verbose=False)

print("Testing Accuracy:  {:.4f}".format(accuracy))
```

```
--14
# Let us additionally train the word embeddings
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Embedding(vocab_size, embedding_dim,
                           weights=[embedding_matrix],
                           input_length=maxlen,
                           trainable=True))
model.add(tf.keras.layers.GlobalMaxPool1D())
model.add(tf.keras.layers.Dense(10, activation='relu'))
model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
model.summary()
history = model.fit(X_train, y_train,
                    epochs=50,
                    verbose=False,
                    validation_data=(X_test, y_test),
                    batch_size=10)
loss, accuracy = model.evaluate(X_train, y_train, verbose=False)
print("Training Accuracy: {:.4f}".format(accuracy))
loss, accuracy = model.evaluate(X_test, y_test, verbose=False)
print("Testing Accuracy:  {:.4f}".format(accuracy))
```

Let us unpack the code as follows:

1. In case you unzip the data to a different location, make sure to update the location to explore the data.
2. We will use the Python pandas library for data analysis.
3. The data looks as follows:



	sentence	label
0	Wow... Loved this place.	1
1	Crust is not good.	0
2	Not tasty and the texture was just nasty.	0
3	Stopped by during the late May bank holiday of...	1
4	The selection on the menu was great and so wer...	1

Figure 3.20: *Sentences with labels*

4. Extract the sentences and the labels. This will be split into the train and test set. As the saying goes - reuse and not recreate. We are using the sklearn `train_test_split` method to split the dataset.
5. It is time to tokenize the text. Tokenize converts words into tokens. It is the first step to encode texts into numbers. TensorFlow Keras contains a library called preprocessing that provides several extremely useful tools to prepare data for machine learning. One of these is a tokenizer that will allow you to take words and turn them into tokens. The Tokenizer object is initialized with the number of words that it can tokenize. This will be the maximum number of tokens to generate from the corpus of words. Think of it as the vocabulary size. Once the tokenizer is initialized, calling `fit_on_texts` creates the tokenized word index.
6. You can check the tokenized word with the tokenizer `index_words` property. It prints key/value pair for the words in the corpus as follows:

```
{1: 'the',
 2: 'and',
 3: 'i',
 4: 'was',
 5: 'a'……
```

7. Now that we have tokenized the words and converted them into number. It is time to convert the text sequence into number sequence. This is pretty straightforward with the `tokeniser.texts_to_sequences` method. All you need to do is to pass the list of sentences.

8. The vocabulary size needs to be increased by one as the index 0 is reserved for padding. We will talk about padding next.
9. Just to be clear as to what we have done – by using tokenization, we converted words into numbers and then the text sequence was converted into number sequences. One sample training sequence may look like this [10, 39, 1, 192, 1, 355, 4, 1, 50, 29, 39, 27, 22]. However, the sequences could of be varying length. This would not work with neural networks as it expects all the input data to be of same shape. So, we need to pad the sequences if it is less or chop if it is more. Again, the Keras preprocessing `pad_sequences` method does this job for us. The Maxlen parameter specifies the desired maximum length of sequences. It is possible to specify where to pad the default value – the beginning or ending of the sequence. Check the documentation for details.
10. Next, we need to download the pertained embedding GloVe from here (http://nlp.stanford.edu/data/glove.6B.zip). Download and extract the data. Since we have a very limited vocabulary (5000 words), we don't need all the embeddings. The `create_embedding_matrix` function prunes all non-vocabulary vectors.
11. It is time to build the network. We are adding the embedding layer. It must be clear now why the `create_embedding_matrix` method was needed. We have added a `globalMaxpooling1D` layer to down sample the incoming feature vectors. The trainable parameter indicates that the vector should be updated during training.
12. With the network in place and compiled as well, it is time to fit the model.
13. Evaluate the model on test set. All this should be familiar if you have used Keras before.
14. Let us again train the network, but this time, let the training update the embedding as well. In case you didn't guess it, all that is required is to set the trainable parameter to true. This time, the embedding will be adjusted to reflect the training dataset vocabulary relationship.

With pre-trained word embeddings, your dependency on a large dataset gets reduced considerably.

Pre-trained language model

It is now time to look at generic pre-trained language models. 2019 will probably be remembered as the ImageNet moment in NLP. Transfer learning that truly democratised deep learning for computer visions has found its way for NLP tasks via pre-trained language models. An example would really help understand the excitement here. Let us predict the missing word here:

The food was good, but the price was ______.

To predict, the language model needs to know a lot about languages. To predict "high" or "exorbitant," the language model should not only learn food attributes but also negations or contrast. Here, the language model is learning to capture not only the semantic meaning but also high-level concepts like negations, long term dependencies, etc.

I am sure that you can imagine how such language models can improve the solution for common NLP tasks like sentence classification. In case you haven't noticed, it is these generic language models that are transferrable. The general process of knowledge transfer involves training a language model on very large text corpus like Wikipedia and then fine-tuning it for downstream tasks such as sentence classification or sentiment analysis. To reiterate, the objective of language modelling is to predict the next word given a stream of input words.

Figure 3.21 shows the high-level process. You need to read it from bottom to top.

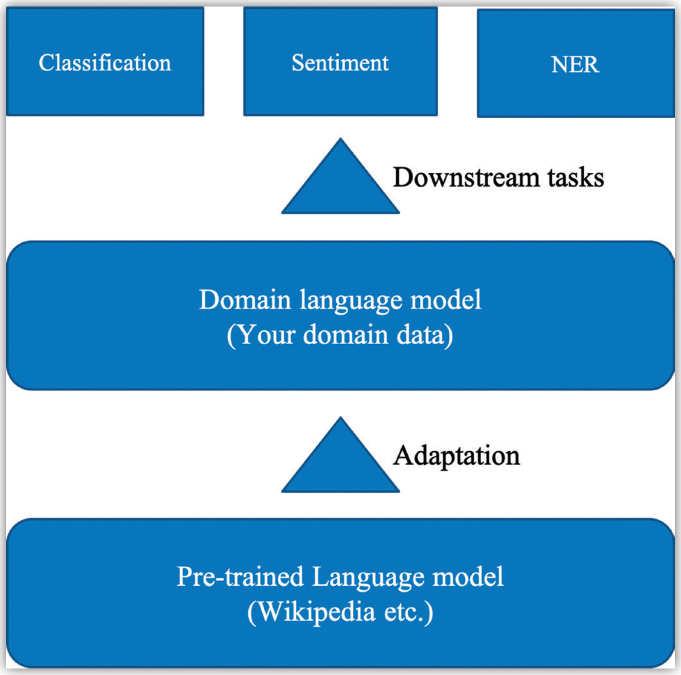


Figure 3.21: *Pre-trained language model adaptation*

Think of it as a stack where the lowest level is learning the generic language features. The model coming out from the bottom stack is generic and can be used by different tasks. The layers above the generic or pre-trained models are task specific. The middle layer is fine-tuning the pre-trained models using the domain dataset and then using it for the task at hand.

It is sometimes hard to believe the success of language modelling. How is a model that can predict the next word useful for tasks like sentiment analysis? Well, the intuition comes from how neural networks are trained. Given enough data to a massive corpus like Wikipedia, many parameters, and enough compute, a model learns a lot of structure and syntax about the language. The knowledge can then be used for tasks like sentiment analysis.

The following is a list of some of the generic pretrained models:

- ULMFiT
- Transformer
- Google's BERT
- Transformer-XL
- OpenAI's GPT

If you are still not convinced about the power of pretrained language models, I encourage you to search for the OpenAI GPT controversy. Now that you have an idea of how it works, let us understand the fine-tuning strategy to adapt pre-trained networks for downstream tasks:

- **Reuse pretrained model internals as is** - This could be as simple as adding one or more layers on top of a pre-trained model. This is a common pattern with BERT.
- **Tune the pretrained model internals** - This is required when the pre-trained model needs to be adapted to a structurally different task. This helps in initializing the target task model as much the possible.
- **Retain the pre-trained weights** - This essentially treats the pre-trained weight as features for downstream tasks.
- **Progressive tuning** - Training all layers at the same time leads to instability. Selectively freezing and unfreezing helps in stable tuning. This is like transfer learning techniques for CV problems.
- **Differential learning rate** - As the lower layer generally contains generic information, it helps to tune them slowly with a lower learning rate. This is essentially decaying learning rate for each layer in the network. Again, you have seen this strategy in the transfer learning for CV video.

For sharing and accessing pretrained models, you can use hubs. Hubs are central repositories that provide a common API for accessing pretrained models. The two most common hubs are as follows:

- TensorFlow Hub
- PyTorch Hub

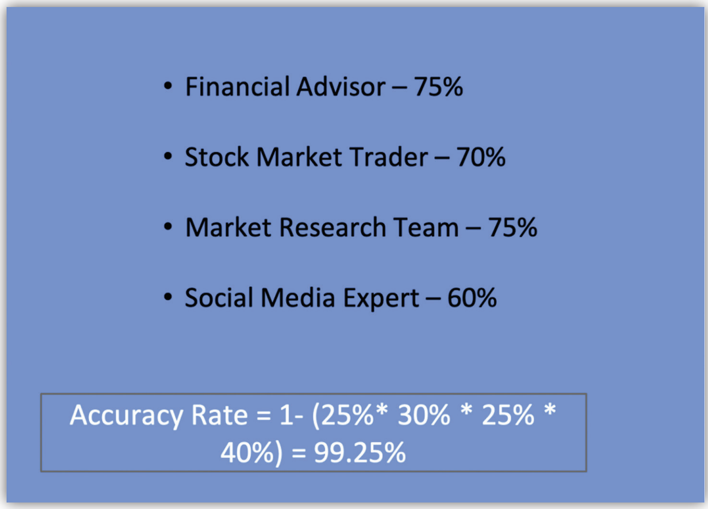
I encourage you to explore this notebook that predicts the movie review sentiment using BERT https://github.com/google-research/bert/blob/master/predicting_movie_reviews_with_bert_on_tf_hub.ipynb

3.5 Model ensembling using ML-ens

One stick can easily break but when tied together, they are a force to reckon with. We can take the power of collectives to lot of situations as well. If you ask a complex question to a thousand random people, their collective answer, well most of the time, will be better than an expert's answer. In the machine learning context, if you aggregate the predictions of diverse learners (such as classifiers or regressors), you will often get better predictions than with the best individual learners. A group of learners is called an ensemble; thus, this technique is called ensemble learning.

We practice ensembling in our daily life without knowing it. Suppose you want to invest in a company but are not sure about its performance, and hence decide to look for advice. So, you reach out to a financial advisor who has an accuracy rate of 75% in the past. Not to rely only on one financial advisor, you decide to check with others as well.

Assume a case where all the individual advisors advice you to buy that stock. What will be the accuracy rate of this collective advice? *Figure 3.22* shows the answer:



- Financial Advisor – 75%
- Stock Market Trader – 70%
- Market Research Team – 75%
- Social Media Expert – 60%

Accuracy Rate = 1- (25%* 30% * 25% * 40%) = 99.25%

Figure 3.22: *The power of collectives*

The typical structure of an ensemble contains different learners. It takes the data and trains it on different algorithms. The final predictions combine the individual predictions. *Figure 3.23* shows the common structure visually:

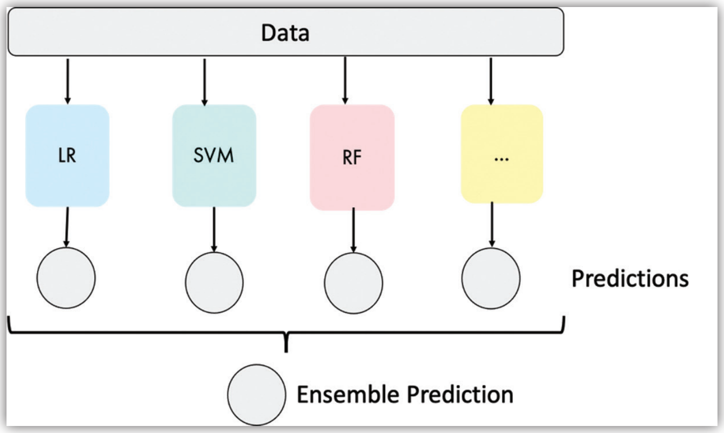


Figure 3.23: *Ensemble common architecture*

There are quite a few ensembling techniques and it helps to have some sort of grouping. Grouping helps to easily remember and recall. There is another hidden benefit - it enables you to tinker with new ideas within the category frame easily:

- The first set of techniques are around providing data to models.

  – Full sampling, random sampling etc.
- The second set of techniques are around combining different models' outputs.

  – Whether they should execute in parallel or sequentially.
- The third set of techniques are around training different models.

  – The number of runs, hyperparameters, etc.

Let us explore a few techniques in each of the categories.

Bagging

It belongs to the first set of techniques. The best way to understand it is to take a look at the architecture as follows:

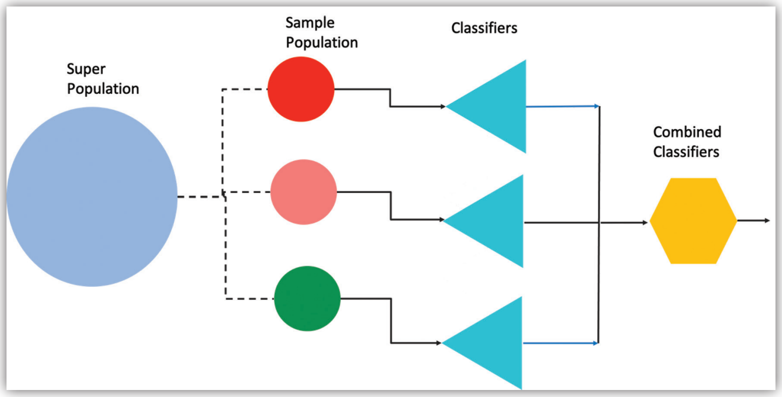


Figure 3.24: *Classifier using bagging technique*

In the first step, we produced multiple different training sets (called bootstrap samples) by sampling with replacement from the original dataset. In the second step, we give those bootstrap samples to the classifier. The final output prediction is combined across the projections of all the sub-models.

K-fold cross-validation

This also belongs to the first set of techniques. Again, we will take a look at an architecture to understand it better as follows:

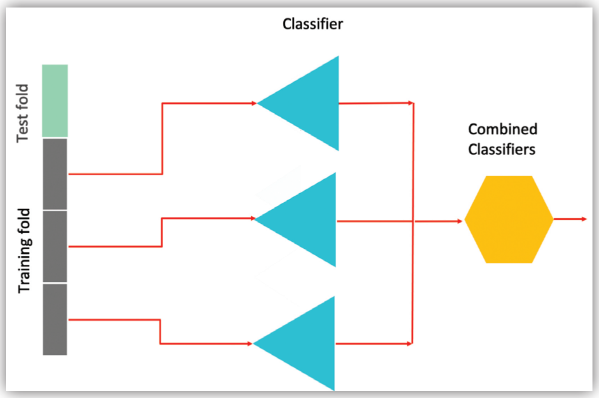


Figure 3.25: *Classifier with k-fold classification*

In the first step, we produce multiple different training sets (called training folds). This is essentially sampling without replacement. From the second step onwards, it is like bagging. Those folds are given to the classifier and the final output prediction is combined across the projections of all the sub-models.

Boosting

Boosting is a form of sequential learning technique. The algorithm works by training a model with the entire training set, and the subsequent models are constructed to fix the error of their predecessor model. It is equivalent to the old proverb "Work on your weaknesses."

Figure 3.26: *Boosting. Image source: Pinterest*

Based on how you fix the previous learner error(s), we will look at two boosting techniques.

Weight-based Boosting

Consider *figure 3.27*:

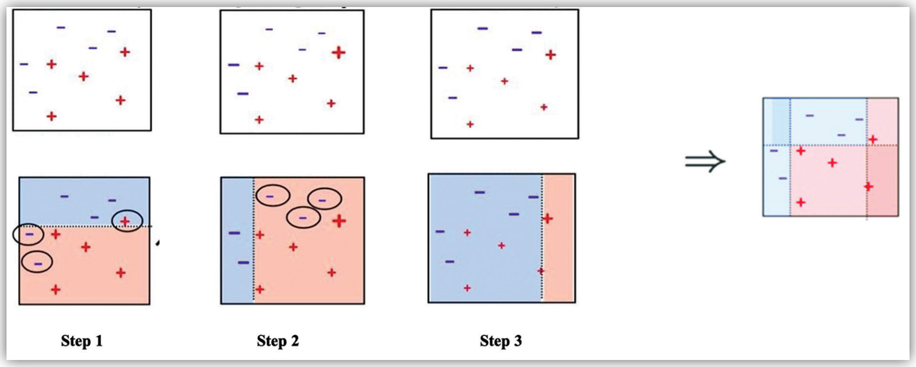
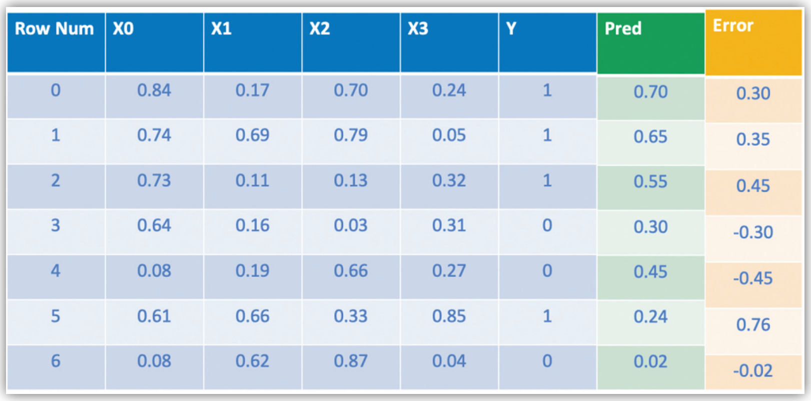


Figure 3.27: *Weight-based boosting*

In the first step, every observation / sample has equal weight (the top-left white box). The weak learner draws the top-horizontal line to discern + from – but fails for 3 (bottom-left colored box). We next update the weights of those incorrectly classified observation (first box, second column). The increased weight leads to increased chance of them getting included in the training of the next layer. We repeat the steps (steps 2 and 3), and finally. by combining the learners from different steps, we get our final model.

Residual-based boosting

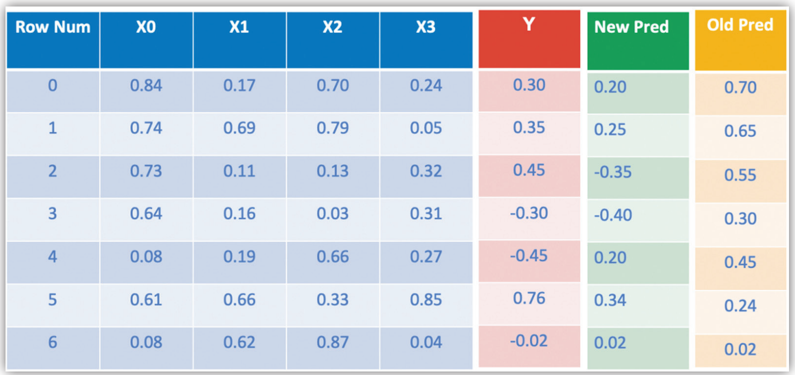
Instead of tweaking the instance weights at every iteration, this method tries to fit a new predictor to the residual errors made by the previous predictor. If the preceding statement doesn't make sense, don't worry. Let us look at an example (*figure 3.28*). X0 to X3 are the features and Y is our truth or target value. Let us assume that our learner predicted the following values:



Row Num	X0	X1	X2	X3	Y	Pred	Error
0	0.84	0.17	0.70	0.24	1	0.70	0.30
1	0.74	0.69	0.79	0.05	1	0.65	0.35
2	0.73	0.11	0.13	0.32	1	0.55	0.45
3	0.64	0.16	0.03	0.31	0	0.30	-0.30
4	0.08	0.19	0.66	0.27	0	0.45	-0.45
5	0.61	0.66	0.33	0.85	1	0.24	0.76
6	0.08	0.62	0.87	0.04	0	0.02	-0.02

Figure 3.28: *Data + prediction + error*

Now, we use the error or the residue as our new target value as shown in *figure 3.29*. The whole cycle is run again.



Row Num	X0	X1	X2	X3	Y	New Pred	Old Pred
0	0.84	0.17	0.70	0.24	0.30	0.20	0.70
1	0.74	0.69	0.79	0.05	0.35	0.25	0.65
2	0.73	0.11	0.13	0.32	0.45	-0.35	0.55
3	0.64	0.16	0.03	0.31	-0.30	-0.40	0.30
4	0.08	0.19	0.66	0.27	-0.45	0.20	0.45
5	0.61	0.66	0.33	0.85	0.76	0.34	0.24
6	0.08	0.62	0.87	0.04	-0.02	0.02	0.02

Figure 3.29: *Error as the new target*

Again, the final prediction is the sum of all predictions. AdaBoost, Gradient Boosting, and XGBoost are popular boosting algorithms. I would encourage you to read the documentations of the respective algorithms.

Stacking

Stacking is the last one in the varying combination category that we will look at now.

Imagine that you are playing KBC (a quiz show) with Amitabh Bachchan and he asks you to answer a history question. You ask two of your friends, one of whom is a history major and the other is a computer science major.

Whom would you trust more? Your previous knowledge would have enabled you to trust your history major friend's answer, or technically-speaking, to give higher value to one of the underlying learners. Stacking is based on the same idea; instead of using trivial functions (such as hard voting) to aggregate the predictions of all learners in an ensemble, why don't we train a model to perform this aggregation? Let us look at its architecture to understand this better.

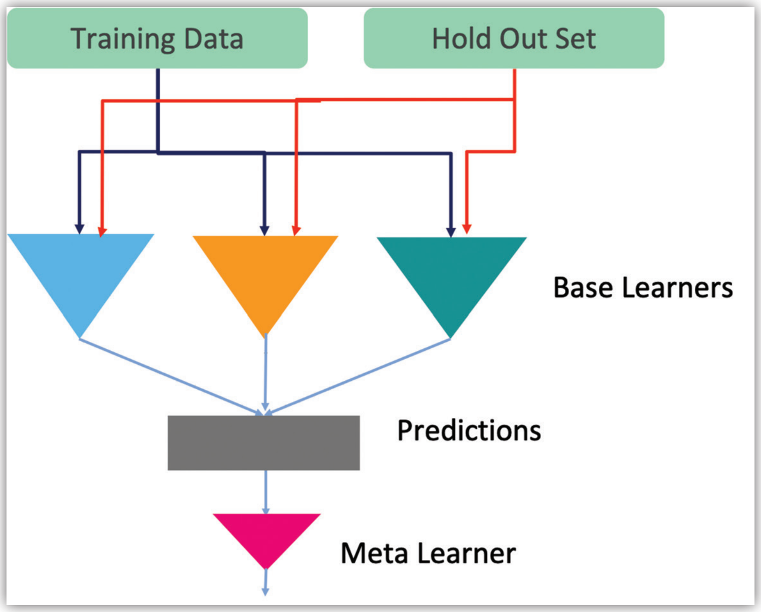


Figure 3.30: *Stacking architecture*

The highlight of the architecture is the meta learner. The base learners' outputs are used to help the meta learner learn.

Varying models

In varying models (also called mix modelling), unlike the mixing training data approach, we give the same dataset to different machine learning models and then combine the results in different ways to get better performing models. Here, instead of relying on a single model, we train our dataset of different machine learning models together (see *figure 3.31*), and then combine the results from these varying models using different techniques.

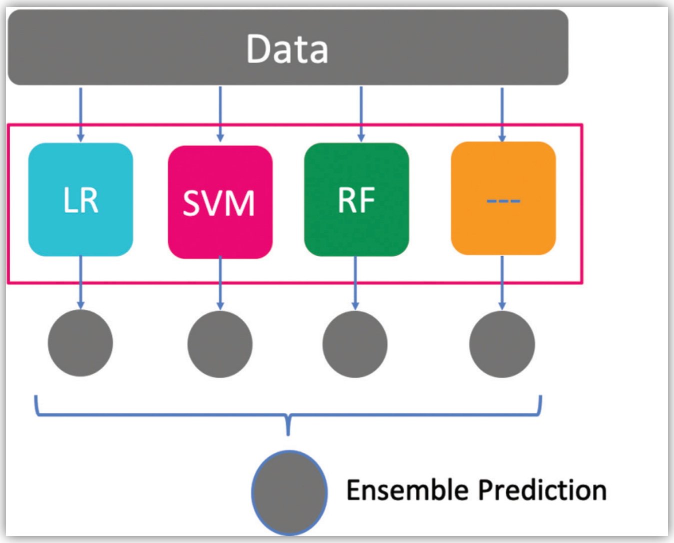


***Figure 3.31:** Mix modelling*

Let us go through a code example for varying models:

```
from sklearn.model_selection import train_test_split

from sklearn.model_selection import GridSearchCV

from sklearn.datasets import load_breast_cancer

import numpy as np

X, y = load_breast_cancer(return_X_y=True)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, stratify=y, random_state=123)


### k-Nearest Neighbors (k-NN)
```

```
from sklearn.neighbors import KNeighborsClassifier


knn = KNeighborsClassifier()

params_knn = {'n_neighbors': np.arange(1, 25)}

knn_gs = GridSearchCV(knn, params_knn, cv=5)

knn_gs.fit(X_train, y_train)

knn_best = knn_gs.best_estimator_


### Random Forest Classifier

from sklearn.ensemble import RandomForestClassifier


rf = RandomForestClassifier(random_state=0)

params_rf = {'n_estimators': [50, 100, 200]}

rf_gs = GridSearchCV(rf, params_rf, cv=5)

rf_gs.fit(X_train, y_train)

rf_best = rf_gs.best_estimator_


### Logistic Regression

from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression(random_state=123, solver='liblinear',
penalty='l2', max_iter=5000)

C = np.logspace(1, 4, 10)

params_lr = dict(C=C)

lr_gs = GridSearchCV(log_reg, params_lr, cv=5, verbose=0)

lr_gs.fit(X_train, y_train)

lr_best = lr_gs.best_estimator_
```

```
# combine all three Voting Ensembles
from sklearn.ensemble import VotingClassifier


estimators=[('knn', knn_best), ('rf', rf_best), ('log_reg', lr_best)]
ensemble = VotingClassifier(estimators, voting='soft')
ensemble.fit(X_train, y_train)
```

I won't go into the full code explanation as it is pretty much standard scikit-learn ML code. However, let us understand the approach as follows:

1. Train three different algorithms – KNN (K Nearest Neighbors), random forest and logistic regression on the breast cancer dataset.
2. The output of all of three are then combined using a voting classifier class implemented in the scikit-learn library.

If you measure the resultant accuracy of each of the individual models as well as the combined model on the test dataset, we can get a very good boost in accuracy. The voting mechanism could be "soft" or "hard." I would encourage you to check the documentation for details.

It is also possible to mix the same model but with different hyperparameters (also hyperparameter tuning ensembles). Here, instead of relying on different models for making ensemble models, we used a good machine learning model but trained this same model with different hyperparameter settings. *Figure 3.32* visualizes the idea with random forest:

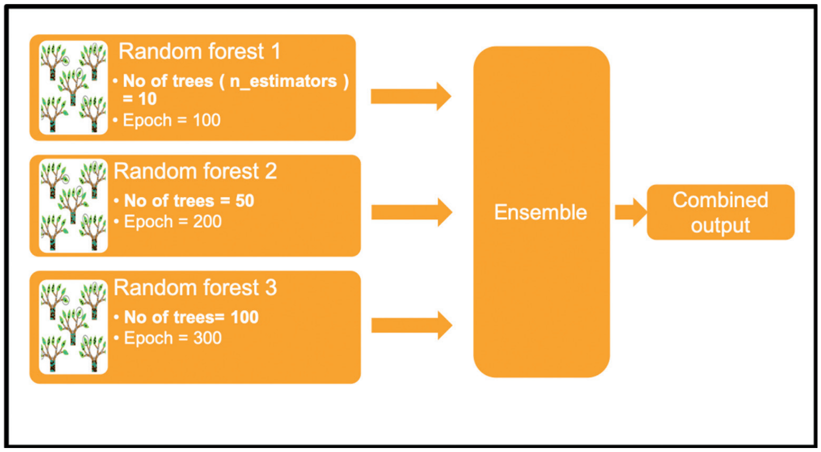


Figure 3.32: *Random forest with different parameters*

This was a quick tour of different ensembling techniques. Next, we will explore ML-ens (https://pypi.org/project/mlens/).

ML-Ensemble

ML-Ensemble, also known as mlens, is an open-source Python library for building scikit-learn compatible ensemble estimators. You can install it via pip:

```
pip install mlens
```

The API style to build the ensembles is very similar to libraries like Keras. It offers a very simple and straightforward way to build deep ensembles with complex interactions. But why do we need a separate library for ensembling? Well, scikit-learn does not support stacking directly. You can still write it but you will have to maintain it yourself. Ml-ens offers a generic way to ensemble estimators and has a reasonable documentation. It is worth exploring even if you decide to not use it in production code. The API helps to experiment with different ensembles very quickly.

Let us build a stacked ensemble via ml-ens. Do recall that stacking combines multiple classification or regression estimators via a meta-learner. The first level estimators are trained based on a complete training set, then the meta-learner is trained on the outputs of the first layer estimators' predictions.

Let us set up the data first. We will use the *make_moons* dataset. In case you didn't know, *make_moons* is a simple toy dataset that makes two half interleaving circles as follows:

```
# ---Data setup----

import numpy as np

from sklearn.metrics import accuracy_score

from sklearn.datasets import make_moons

seed = 42


X, y = make_moons(n_samples=10000,noise=0.4, random_state=seed)


# --- 1. Initialize ---

from mlens.ensemble import SuperLearner
```

```
ensemble = SuperLearner(scorer=accuracy_score, random_state=seed)

# --- 2. Build the first layer ---

ensemble.add([RandomForestClassifier(random_state=seed),      SVC(random_
state=seed)])

# --- 3. Attach the final meta learner

ensemble.add_meta(LogisticRegression())

# --- Train ---

ensemble.fit(X_train, y_train)

# --- Predict ---

preds = ensemble.predict(X_test) https://arxiv.org/abs/1603.06560
```

Now, let's go through the code. The ensembling is essentially a three-step process as follows:

1. We first initialize the ensemble which is SuperLearner here.
2. In the second step, we add the intermediate estimators. Here, we are adding two classifiers: RandomForrest and SVM. Note that they will execute in parallel.
3. Then, we add the meta learner which is LogisticRegression here.

Finally, we call the fit method and do the predictions. Visually, it can be represented as follows in *figure 3.33*:

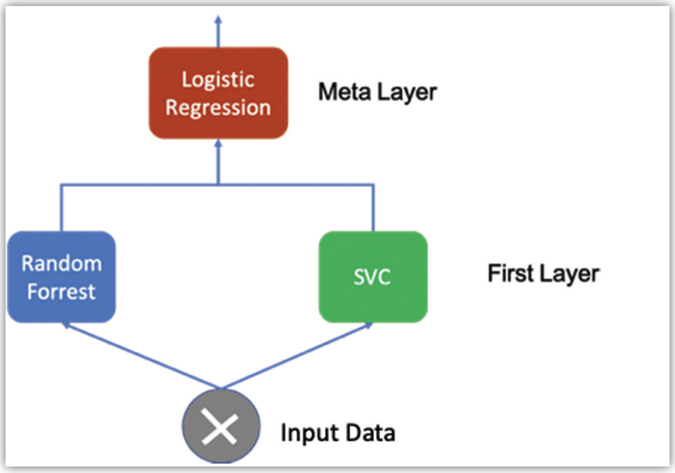


Figure 3.33: *Single-layer stacked ensemble*

Doesn't it feel like setting up a neural network where we stack layers to build the network? Check the performance of the estimator in the layers, and call the data attribute as follows:

```
print("Fit data:\n%r" % ensemble.data)
```

Fit data:

```
                                    score-m  score-s  ft-m   ft-s   pt-m   pt-s
layer-1  randomforestclassifier      0.84     0.00    0.06   0.00   0.01   0.00
layer-1  svc                         0.86     0.00    0.14   0.00   0.06   0.00
```

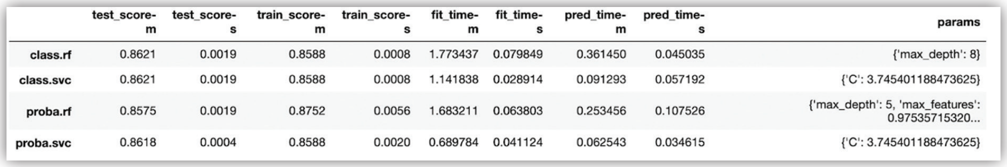
The first column - score-m - contains the score. The suffix -m is used to denote mean values and `-s` is used to denote standard deviation across folds for brevity. ft and pt stand for fit time and prediction time respectively. I encourage you to read the documentation for further details. Do note that we provided the scoring function during SuperLearner initialization. If we can add two estimators at the first layer, then it shouldn't be a surprise that you can add more estimators at any layer.

Multi-layer ensembles

Adding multiple layers is equally straightforward. We just need to call the add function to add a new layer. Note that layers are executed in sequence. However, within a layer, estimators can run parallelly as follows:

```
ensemble = SuperLearner(scorer=accuracy_score, random_state=seed, verbose=2)

# Build the first layer

ensemble.add([RandomForestClassifier(random_state=seed), LogisticRegression(random_state=seed)])

# Build the 2nd layer

ensemble.add([LogisticRegression(random_state=seed), SVC(random_state=seed)])

# Attach the final meta estimator

ensemble.add_meta(SVC(random_state=seed))
```

Figure 3.34 is a visual representation of the ensemble:



	test_score-m	test_score-s	train_score-m	train_score-s	fit_time-m	fit_time-s	pred_time-m	pred_time-s	params
class.rf	0.8621	0.0019	0.8588	0.0008	1.773437	0.079849	0.361450	0.045035	{'max_depth': 8}
class.svc	0.8621	0.0019	0.8588	0.0008	1.141838	0.028914	0.091293	0.057192	{'C': 3.745401188473625}
proba.rf	0.8575	0.0019	0.8752	0.0056	1.683211	0.063803	0.253456	0.107526	{'max_depth': 5, 'max_features': 0.97535715320...
proba.svc	0.8618	0.0004	0.8588	0.0020	0.689784	0.041124	0.062543	0.034615	{'C': 3.745401188473625}

Figure 3.34: *Multi-layer stacked architecture*

Ensemble model selection

The power of the ensemble comes from the diversity of the base learners. But often, it is difficult to figure out the base learners parameters correct values. It is important to remember that a superior ensemble can be built by combining base learners with a lower predictive power. This is because the uncorrelated error from the base learners help the meta estimator to overcome the weakness of the base estimators. Thus, the base learners' parameters hyper tuning (treating the base learner's parameter as the parameters of the ensemble), is a very important step in appropriate model selection in ensemble. Remember that the objective here is to exploit the learning capacity of an ensemble.

Meta learner is another critical part of the ensemble and hence needs to be selected appropriately. So, how would you select an appropriate meta learner? It is not an easy task given that the entire ensemble must be evaluated.

One trick used by mlens is to process the lower layer and then cache the results. This essentially makes the lower layer of an ensemble a pre-processing pipeline. This way, the model selection would only be restricted to higher levels or meta learners. To use an ensemble for this purpose, you need to set the `model_selection` parameter to true in the SequentialEnsemble constructor. We will see this in code shortly. Under the hood, this ensures that the ensemble predict is called on test folds.

Before we look at the end-to-end code for model selection, there are some more concepts that we need to understand.

The scoring function

The different learners may measure error or accuracy. However, for ensembling to work, they need to be made consistent to behave in a similar fashion. To do this, we need to wrap the scoring function inside the mlens `make_scorer()` function as follows:

```
from mlens.metrics import make_scorer

accuracy_scorer = make_scorer(accuracy_score, greater_is_better=True)
```

The true value of the parameter `greater_is_better` indicates accuracy whereas false indicates error/loss. The `make_scorer` wrapper is a copy of the scikit-learn sklearn`.metrics.make_scorer()`. Under the hood, sklearn `make_scorer` is a factory function that wraps scoring functions for use in `GridSearchCV` and `cross_val_score`. It takes a score function such as `accuracy_score, mean_squared_error, adjusted_rand_index` or `average_precision` and returns a callable that scores an estimator's output. So that we are not lost here, do remember that our objective is to find the appropriate meta learners. Now that you know how to make scoring consistent, let us understand how the library manages the processing pipeline.

Evaluator

ml-ens Evaluator class allows you to grid search several models in parallel across several pre-processing pipelines. The evaluator class pre-fits transformers, thus avoiding fitting the same pre-processing pipelines on the same data repeatedly. Let us go through the code to understand it better. I have skipped some of the common and obvious codes for brevity.

```
from mlens.model_selection import Evaluator

from scipy.stats import randint

from sklearn.naive_bayes import GaussianNB

from sklearn.neighbors import KNeighborsClassifier

Now, we need to name the estimators:

ests = [('gnb', GaussianNB()), ('knn', KNeighborsClassifier())]
```

Then, we prepare the parameters list. This is no different from what you do during grid or random search. Note that `gnb` is not included because it doesn't have any parameters:

```
pars = {'n_neighbors': randint(2, 20)}

params = {'knn': pars}
```

We can now run an evaluation over these estimators and parameter distributions by calling the evaluate method as follows:

```
evaluator = Evaluator(scorer=accuracy_scorer, cv=10)

evaluator.fit(X, y, ests, params, n_iter=10)
```

You can check the results and summary via Evaluator `cv_results` and summary properties.

Pre-processing

The pre-processing feature of ml-ens helps you to compare the models across a set of pre-processing pipelines. It does this via a class that acts as a transformer, allowing you to use lower or incoming layers as a "pre-processing" step, so that you need only evaluate the meta learners iteratively. Let us look at the code to understand it better:

```
from sklearn.preprocessing import StandardScaler

preprocess_cases = {'none': [],

                   'sc': [StandardScaler()] }
```

We have specified a dictionary of pre-processing pipelines to run through. Each entry in the dictionary is a list of transformers to apply sequentially. It's now time to look at an end-to-end example to see all the pieces in action as follows:

```
from mlens.model_selection import Evaluator

from mlens.ensemble import SequentialEnsemble #--1

from mlens.metrics import make_scorer

from scipy.stats import uniform, randint


base_learners = [RandomForestClassifier(random_state=seed),

                SVC(probability=True)] #--2


proba_transformer = SequentialEnsemble(

                       model_selection=True, random_state=seed).add(

                           'blend', base_learners, proba=True) #--3

class_transformer = SequentialEnsemble(

                       model_selection=True, random_state=seed).add(

                      'blend', base_learners, proba=False) #--4
```

```
preprocessing = {'proba': [('layer-1', proba_transformer)],

                 'class': [('layer-1', class_transformer)]} #--5


meta_learners = [SVC(random_state=seed), ('rf',
RandomForestClassifier(random_state=seed))] #--6

params = {'svc': {'C': uniform(0, 10)},

          'class.rf': {'max_depth': randint(2, 10)},

          'proba.rf': {'max_depth': randint(2, 10),

                            'max_features': uniform(0.5, 0.5)}

          } #--7

scorer = make_scorer(accuracy_score) #--8

evaluator = Evaluator(scorer=scorer, random_state=seed, cv=2) #--9


evaluator.fit(X, y, meta_learners, params, preprocessing=preprocessing,
n_iter=2)#--10


from pandas import DataFrame

df = DataFrame(evaluator.results) #--11
```

Figure 3.35: *DataFrame output from the previous code*

Let us unpack the code:

1. We import the `SequentialEnsemble` class. The `SequentialEnsemble` allows users to build ensembles with different classes of layers. The different classes of layers are `"blend," "subset," and "stack."` The three different classes are three different ways of mapping the training set to prediction set used by the meta learner.
2. Use Randomforrest and SVM as our base learners.

   We are setting up two competing ensemble bases as pre-processing transformers. This one is a blended ensemble base with proba. Note that by proba, it means whether the layer should predict class probabilities. Here the estimators `predict_proba` method will be called.

3. This one is a blended ensemble without proba. Note that the `model_selection` parameter is set to true. This modifies how the transform method behaves and ensures what predict is called on test folds.
4. Now, we need to set up a preprocessing mapping. Each pipeline in this map is fitted once on each fold before evaluating the candidate meta learners.
5. It's time to set up candidate meta learners. Here, the estimators will run on all pre-processing pipelines.
6. Here, we are setting up parameter mappings. Note that the distributions are differentiated between cases for the random forest.
7. We are wrapping the score function. You already know why that is.
8. It's time to instantiate the evaluator.
9. Call the evaluator fit method.
10. This is not required but you can load the evaluator result in a DataFrame to view the results in a formatted way. Again, `-s` and `-m` suffixes stand for mean and standard deviation.

Let us do a quick recap. Ml-ens offers Keras style API build ensembles. The SuperLearner class helps in building stacking ensembles. Ml-ens offers different kinds of stacking layers like "stack," "blend," and "subset." Running an entire ensemble several times just to compare different meta learners can be prohibitively expensive. ML-Ensemble implements a class that acts as a transformer, allowing you to use ingoing layers as a "pre-processing" step, so that you need only evaluate the meta learners iteratively. You can find the project documentation here: http://ml-ensemble.com/info/index.html

3.6 Conclusion

ML is iterative and experimental in nature. Rather than speculating or going by "what I know," it really helps to get started simply and then iterate. A neural network setup with transfer learning should be preferred over handcrafting neural network for applied AI (business situations). Ratherthan going by a hyperparameter value that probably worked in your last project, it helps to use automated hyperparameter searching techniques. Ensembling techniques are the winners in Kaggle competitions but have somehow received that respect in the enterprises. With libraries like Ml-ens, the barrier for ensembling experimentation has been considerably lowered. Use of high-quality libraries speeds initial development that results in fewer bugs, reduces reinvention-of-the-wheel, and cuts long-term maintenance costs. In the next chapter, we will explore Airflow which is an ML workflow management system. In fact, the remaining chapters will focus on scaling ML systems. See you in the next chapter, and until then, happy learning.

3.7 Points to remember

- The trick to building the intuition is to look at different ML grouping strategies. Strategies generally map well to different problem domains.
- ML algorithm can be grouped into supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning based on how they learn.
- ML algorithm can be grouped into parametric and non-parametric model.
- The best hyperparameter value is not just about finding the best score but also ensuring that the best score is found with minimal resources like compute, money, and time.
- Grid search, random search, and Bayesian optimization are a few automated techniques for searching the hyper parameter space.
- Grid search cannot optimize models with more than four dimensions due to the curse of dimensionality.
- Random search is more capable than grid search. It's naive approach is time-consuming and expensive. Furthermore, it is unreasonable beyond ten dimensions.
- Bayesian optimization searches the parameter space, keeping in mind past results
- Parameter space, an objective function, and a surrogate (selection method) are the three building blocks for Bayesian optimization.
- The objective function evaluates the hyperparameter combinations. Examples of objective functions are accuracy, root mean squared error, and so on.
- Surrogate is the probabilistic model for the objective function that you are trying to minimize or maximize. It helps in selecting the next hyperparameter combination to try.
- Keras Tuner is a distributable hyperparameter optimization framework to perform a hyperparameter search. You can define a search space and then leverage algorithms like Bayesian Optimization, Hyperband, and Random Search to find the best hyperparameter values.
- Training the top layer is the easiest way to transfer the knowledge for CV tasks.

- You can fine-tune the weights by unfreezing the transferred parameters and then retraining.
- One can generalize the transfer learning by transferring the first n layers from a pre-trained network to the target network and randomly initializing the rest.
- Rather than training the top, reverse is also possible. One can train the bottom layers and transfer the remaining ones.
- Model distillation is a technique for transferring knowledge between models having different shapes.
- Word embeddings enables transfer learning for NLP tasks.
- Word2Veec, GloVe and fastText are a few ready-to-use pre-trained word embeddings (vector).
- Pre-trained language models are revolutionizing transfer learning strategy.
- The objective of language modelling is to predict the next word, given a stream of input words.
- The general process of knowledge transfer involves training a language model on very large text corpus like Wikipedia and then fine-tuning it for downstream tasks such as sentence classification or sentiment analysis.
- A group of learners are called an ensemble.
- Ensembling techniques can be grouped into three broad categories as follows:
  - o The 1st set of techniques are about providing data to models.
  - o The 2nd set of techniques are about combining different models' output.
  - o The 3rd set of techniques are about training different models.
  - o ML-Ensemble, also known as ml-ens, is an open-source Python library for building scikit-learn compatible ensemble estimators.

3.8 Questions

1. What are parametric and non-parametric models?
2. What is the difference between grid search and random search?
3. How does Bayesian optimization address the shortcomings of grid and random search?
4. What are the three building blocks of Bayesian optimization?
5. What is transfer learning?
6. What is model distillation?
7. What are the high-level techniques for ensembling?

Chapter 4
Bye-Bye Scheduler, Welcome Airflow

Apache Airflow is an open-source project to programmatically author, schedule, and monitor workflows. The objective of this chapter is to give a gentle introduction of Airflow.

Structure

In this chapter, you will learn the following:

- Introducing Airflow
- Executers in Airflow
- Anatomy of DAG
- Understanding scheduling
- Airflow operators
- Managing task branching and dependencies
- Sharing data between tasks using XCom
- Understanding Airflow variables

Objective

By the end of this chapter, you will know the following:

- What is Airflow and why we need it.

- What is DAG and it is core to Airflow.
- Installing and configuring airflow. Docker will be used as it is almost the default production choice.
- Using Airflow UI.
- Defining Airflow workflows with complex task dependencies.
- Scheduling tasks.
- Exploring different operators.
- Managing variables in Airflow.

4.1 Introducing Airflow

Machine learning is essentially a pipeline in which you collect data, clean it, transform it, and eventually train the model. If you know about `ETL (extract, transform and load)`, then you would know what I mean. Even though the real world is a lot more completed than this, even the basic ML systems still require some sort of workflow. If you imagine the steps as tasks, you would need some sort of system to execute and manage the tasks.

Figure 4.1: *Image source: Airflow logo*

Airflows provides a framework to manage the tasks at scale. If you are not convinced of the value of a workflow management system, then think of the following scenarios:

1. What happens if the data collection (extraction) fails?
2. What happens if the transformation fails?
3. What happens if the training fails?
4. It is not just about extraction. How do you indicate "transformation" to begin only after extraction (collection) is successful?
5. What do you do if the task was partially successful?
6. How do you manage tasks when they need different technology stacks?
7. How do you optimize task execution by executing them in parallel?

While you can continue adding to the list, I am sure you would agree with the involved challenges and the need for a framework that offers a seamless, repeatable, and scalable workflow management system. Airflow provides a Python framework to develop and manage data pipelines. If you look at the website, then the definition would be clear to you. Airflow is a framework to programmatically author, schedule and monitor workflows.

Workflow, tasks, and execution are a few terms that you need to be comfortable with in order to understand Airflow. Let us understand the concepts using an example: Imagine that you need to generate the sales report every week and the process looks like `figure 4.1`:

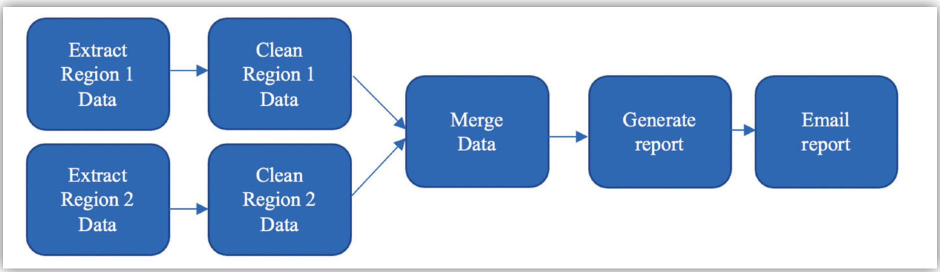


Figure 4.2: *Sales report generation workflow*

Congratulations! You just orchestrated a workflow with each step constituting a task. If you think that the initial steps of extraction and cleaning can be executed in parallel, then yes, you are correct. Some of the tasks in the `figure 4.2` workflow can be executed in parallel and post-merging, they need to be executed sequentially. The situation can get complicated if the regions have different technology stacks – imagine one of them using MySQL and another one using MySQL databases. Based on `figure 4.2`, it is clear that while executing, Airflow must ensure the following safeguards:

1. Tasks are executed in the correct sequence, or that tasks dependencies are understood and maintained.
2. In the event of failures, it must send notifications.
3. If a task fails, attempt retry.
4. Tasks executions are monitored.
5. Tasks executions are properly logged.

The idea of a workflow manager is not new but with hyperfocus on `"data"` has led to innovation in the space. The upstream tasks appear on the left of the arrow and the downstream ones appear on the tip of the arrow. To be explicit, the downstream tasks are dependent on the upstream tasks.

To execute the tasks, the workflow is modelled like a **DAG (Directed acyclic graphs)**. So what is a DAG? DAGs are acyclic graphs which means that they are not allowed to contain a circle. But how this is helpful? With a circle, there is no way to know when the workflow would end. A DAG highlights independent and dependent tasks, paving the way for parallel execution wherever possible.

There are a few properties of DAG that are worth remembering. These are as follows:

- They should be directed (Edges should have a direction).
- They must be acyclic. Downstream tasks should not be connected to upstream tasks.
- They must be graph structures (vertices and edges).

Let us reimagine *figure 4.2* as a DAG:

- Tasks are the vertices.
- Edges are the dependencies.
- None of the downstream tasks are connected to the upstream tasks.

In Airflow, we essentially define DAGs with Airflow managing the DAG (graph) execution for us. It would soon become clear in case it is still not clear. DAGs are the core to workflow. There is one more interesting thing worth discussing from *figure 4.1*. The use case seems to fit better for the batch mode or scheduled executions, that is, Airflow executes the report generation at scheduled intervals. Airflow is better for batch mode execution, a job that has a finite set of steps with a clear beginning and end. If you have infinite or real time processing needs, then Spark (https://spark.apache.org/) is most probably your solution.

Let us look at a DAG (*code 4.1*) to see the concepts in code:

```
import airflow.utils.dates

from airflow import DAG

from airflow.operators.bash_operator import BashOperator

from datetime import datetime, timedelta


default_args = {

        "owner": "airflow",

        "start_date": airflow.utils.dates.days_ago(1)

    }
```

```
with   DAG(dag_id="sleep_dag",   default_args=default_args,   schedule_
interval="@daily") as dag:


    t1 = BashOperator( --1

            task_id="t1",

            bash_command="echo Hello"

        )


    t2 = BashOperator( --2

            task_id="t2",

            bash_command="echo World"

        )


    t1 >> t2 --3
```

Code 4.1: *Airflow DAG Hello world*

Don't worry about the imports and the DAG initialization. We will come back to it. Let us discuss points 1, 2, and 3:

1. In point 3, we have defined a DAG – task `"t1"` which should be executed before task `"t2."`
2. The double angular sign is essentially a way to define an edge – dependency between the two tasks. But what are `t1` and `t2`?
3. `"t1"` and `"t2"` are two dummy tasks (here, bash commands) but could have been something complicated as well. Like BashOperator, there are quite a few ready-to-use task types (or operators in the Airflow world) that simplify the DAG writing process.

Let us play a pop quiz. Can you guess which of the following are acyclic graphs?

1. T1>>T2>>T3
2. T1>>T2>>T3>>T1
    a. T1>>T2>>T3>>T4
    b. T2>>T5>>T6

Only 2 is not an acyclic graph because it has a loop. Also, how do you think tasks will be executed in the following situations?

1. T1 >> [T2, T3] >> T4
2. T1 >> [T2, T3] >> [T4, T5] >> T6

The following is an explanation of the preceding steps:

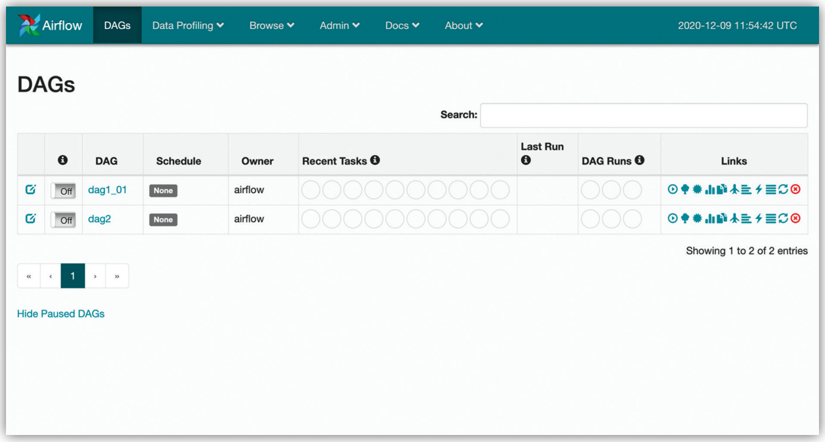
1. Task t1 executes first and then T2 and T3 can execute in parallel. This is also called fan-out pattern. Task T4, however, is dependent on the upstream list (T2 and T3). You may be wondering whether T4 should wait for both to complete or for just one to finish. They are excellent questions but hold on for now.
2. I guess, now you can interpret it as T1, then (independent T2, T3), then (independent T4, T5), and finally T6.

We hope the exercises made DAG and its execution clear. In case you have doubts, don't worry. We will keep coming back it. It is time to set up Airflow.

4.2 Installing Airflow

Before we install Airflow, let us get a feel of it. At the core, Airflow has the following 3 parts:

- **Webserver** – The webserver provides the web interface to manage and view the workflows. By default, it would be available on port 8080 (http://localhost:8080/). The page would look as shown in *figure 4.3*:



DAGs

Search:

		DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
	Off	dag1_01	None	airflow				
	Off	dag2	None	airflow				

Showing 1 to 2 of 2 entries

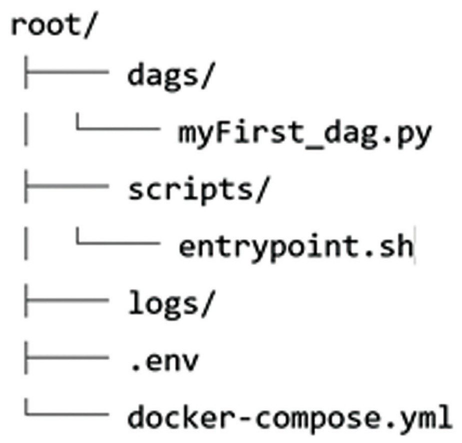
Hide Paused DAGs

Figure 4.3: Airflow management dashboard.

- **Scheduler** – Scheduler is responsible for scheduling the workflow (technically DAG definition) execution. *Code 4.1* is an example of a workflow which is called a DAG in the Airflow lingo. Scheduler is responsible for managing the workers executing the tasks.

- **Database** – Database stores the metadata about the workflow, tasks, schedules, and execution status.

Given that Airflow is a generic workflow management, it supports quite a few installations configuration options. We will use Docker-based Airflow installation as it makes it platform-independent and is the almost default choice for production. We will cover Docker in a later chapter. For now, in case you don't know about Docker, please follow the instructions. We will use the following code structure for our exploration:



```
root/
├─── dags/
│   └─── myFirst_dag.py
├─── scripts/
│   └─── entrypoint.sh
├─── logs/
├─── .env
└─── docker-compose.yml
```

Let us write the **`docker-compose.yml`** file:

```
#docker-compose.yml


version: '3.8'

services:

    postgres:

        image: postgres

        environment:

            - POSTGRES_USER=airflow

            - POSTGRES_PASSWORD=airflow

            - POSTGRES_DB=airflow
```

```
scheduler:
    image: apache/airflow
    command: scheduler
    restart_policy:
        condition: on-failure
    depends_on:
        - postgres
    env_file:
        - .env
    volumes:
        - ./dags:/opt/airflow/dags
        - ./logs:/opt/airflow/logs
webserver:
    image: apache/airflow
    entrypoint: ./scripts/entrypoint.sh
    restart_policy:
        condition: on-failure
    depends_on:
        - postgres
        - scheduler
    env_file:
        - .env
    volumes:
        - ./dags:/opt/airflow/dags
        - ./logs:/opt/airflow/logs
        - ./scripts:/opt/airflow/scripts
    ports:
        - "8080:8080"
```

We will not go into detail, but the following points require highlighting:

1. We are using postgres as the backend for the Airflow.
2. The `entrypoint.sh` contains the commands to run the web server and initialize the database.

   Prepare the `entrypooint.sh` file:

   ```
   #entrypoint.sh


   #!/usr/bin/env bash
   airflow initdb
   airflow webserver
   ```

   The environment variables would look as follows:

   ```
   #.env
   AIRFLOW__CORE__SQL_ALCHEMY_CONN=postgresql+psycopg2://
   airflow:airflow@postgres/airflow
   AIRFLOW__CORE__EXECUTOR=LocalExecutor
   ```

   For `myFirst_Dag.py` file, use code 4.1. Don't forget to remove the pointers (1, 2, and 3).

So in terms of steps, all you need to run are the following steps:

1. Run the following command from the application **`root`** folder to run the Docker. It automatically picks up the **`docker-compose.yml`** file:

   ```
   docker-compose up
   ```

2. Run the following command to stop the Docker:

   ```
   docker-compose up
   ```

3. Run the following command to get the container-id. The container ID will be required to log into the Docker session. The first column of the result contains the container ID:

   ```
   docker ps
   ```

4. We will run the Airflow command from inside Docker. Replace the container ID retrieved at step 3:

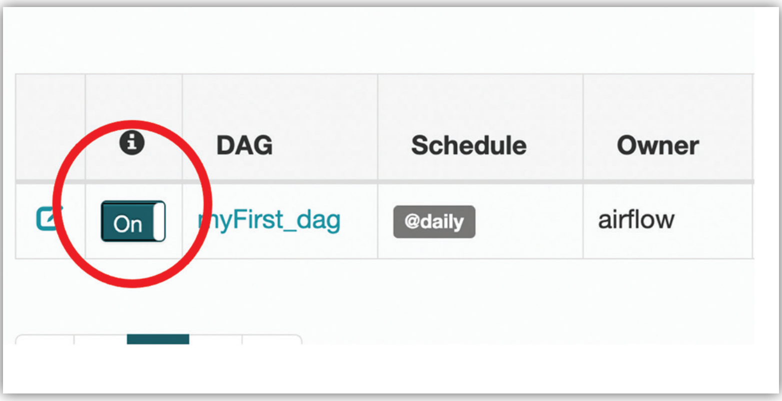
   ```
   Docker exec -it <<container id>> /bin/bash
   ```

5. Go to your browser and open this URL: http://localhost:8080/. If it opens the Airflow page (*figure* 4.3), then congratulations! You just successfully installed

Airflow. In case you are looking for non-Docker setup, then the Airflow documentation (https://airflow.readthedocs.io/en/stable/installation.html#getting-airflow) will be the best place to go.

4.3 Running Airflow

Before you run the DAG, make sure that the schedule flag is set to ON (Toggle it if off). Refer to *figure 4.4*:



		DAG	Schedule	Owner
	On	myFirst_dag	@daily	airflow

Figure 4.4: *The schedule state of the DAG. Must be set to ON.*

There are two ways to run the DAG manually. We will discuss the scheduling later.

4.3.1 Running Airflow via UI

1. Click on the DAG (**`myFirst_Dag`**). It will open the DAG page. Click on the **`Trigger DAG`** option. Refer to *figure* 4.5:

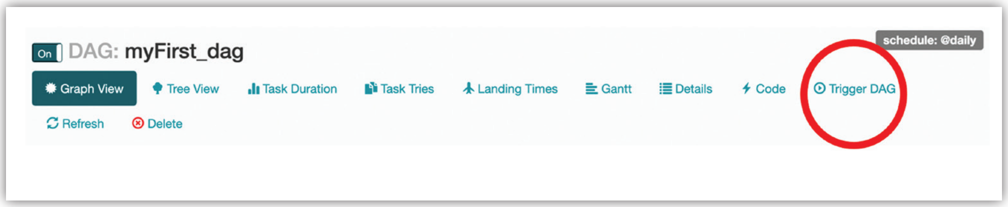


Figure 4.5: *Trigger the DAG from UI.*

2. In order to view the status of the DAG, come back to the **Graph view** page (the highlighted tab in green in *figure* 4.5). If you see **success**, then awesome! You just had your first Airflow run.

4.3.2 Running Airflow via command line

1. Connect to the running Airflow Docker. The steps to connect to the running Docker are provided in the installation section.
2. In the Docker, go to the ""**`dag`**" folder

   ```
   cd /root/airflow/dags
   ```

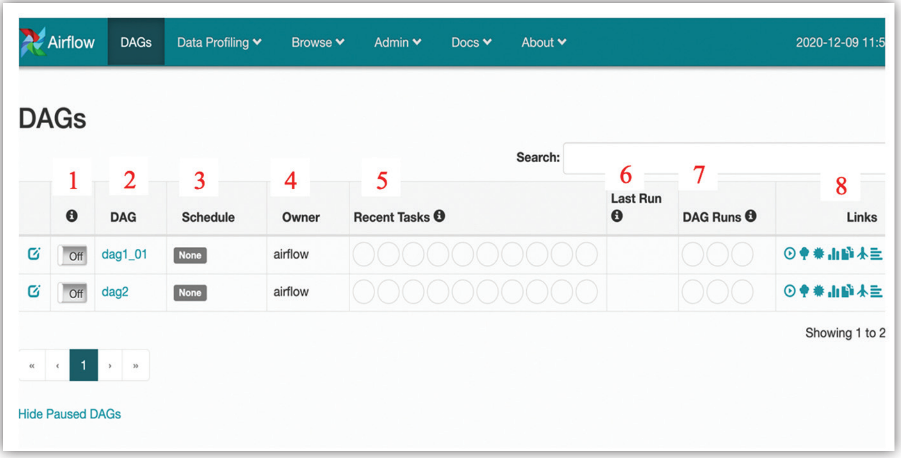
   Run the following command. The **`trigger_dag`** is the command. The **`myFirst_dag`** is the DAG ID specified in the DAG (refer to **`myFirst-Dag.py`**) and **`-e`** is the scheduled date (since when it should run from). For testing purpose, use your current – 1 date. Again, we will come to the detail of it later. At the moment, we are gently getting our feet wet. To check the status, go back to the UI and open the DAG **`Runs`** page (http://localhost:8080/admin/dagrun/). You will find the status of your triggered DAG:

   ```
   airflow trigger_dag myFirst_dag -e <give a data in yyyy-mm-dd>
   ```

Just to make sure you are not lost: We defined the workflow (`myFirst_Dag.py`) that had two simple tasks, and then we triggered them manually via the UI and command line. So which one is better? I will share my thumb rule – stick to command line during development; it's just fast!

4.3.2.1 Airflow UI tour

Let us spend some time understanding the UI. This goes a long way in the effective usage of the tool.



DAGs

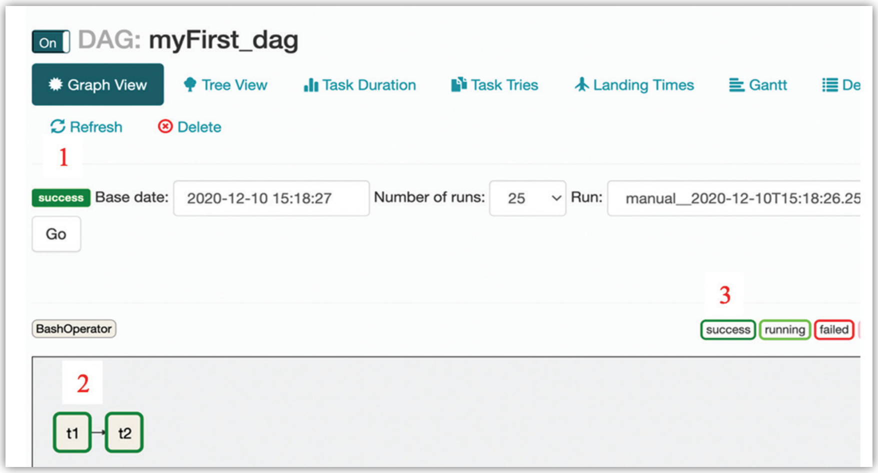
		DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
	Off	dag1_01	None	airflow				
	Off	dag2	None	airflow				

Figure 4.6: DAG dashboard elements

Let us understand the columns as follows:

1. The first page is the `DAG` dashboard. This gives you an overview of the task's status. For example, whether they are running, have failed or succeeded, when the DAG is scheduled and when it was last executed. For each DAG, the information is spread across eight columns to be read from left to right.
2. The second column `DAG` (containing the ON/OFF toggle) gives you a toggle to pause or schedule a DAG. Do recall that a DAG needs to have this in the ON state to get triggered even manually.
3. The `Schedule` column tells you about the scheduled execution of the DAG. It could be daily, monthly, weekly, hourly, etc. You get the idea!
4. `Owner` tells you who can see the DAG.
5. The `recent tasks` column shows a summary of the last scheduled run. Each circle shows the status of the task – queued, running, failed, etc.
6. `Last run` shows the last time the DAG was triggered.
7. `DAG Runs` shows the history of the DAG runs, essentially how it was run in the past.
8. The `links` column provides a bunch of links to switch between the different views of the DAG. Hover over each of the link buttons and they will show their purpose in the tool tip. One of them you already know – `Trigger DAG`. Click on it and trigger the DAG. You will see an entry in the DAG run column. The corresponding tasks will be shown in the recent task column. I would like you to stop reading NOW and quickly try it.

Let us now click on the DAG Name (column 2) to understand the different views. This will take you to the graph view as shown in *figure 4.7*:



Base date: 2020-12-10 15:18:27 Number of runs: 25 Run: manual__2020-12-10T15:18:26.25

Figure 4.7: *DAG graph view*

Let us understand the columns as follows:

1. Indicates the DAG run status. Here, it is successful. Do note the color legend as well.
2. This shows the DAG structure. This structure was coded in code 4.1. A few things you should be able to deduce easily. T1 and T2 are sequentially connected and are the only tasks. This structure is created when you place the DAG file in the DAG directory (Airflow/DAGs). It is read by the Airflow and then visualized in the UI. It's no surprise that this is one view that will be used quite heavily during the development process as it makes the dependency and flow quite clear.
3. When you run a DAG, it goes through different stages. The color codes along with the textual status inform you about the current state of the run. In figure 4.6, it is green and success because it got successfully completed.

All tasks' logs are collected by Airflow, and they are organized visually for searchability on the UI. Click on any of the tasks to view the logs as shown in *figure 4.8*.

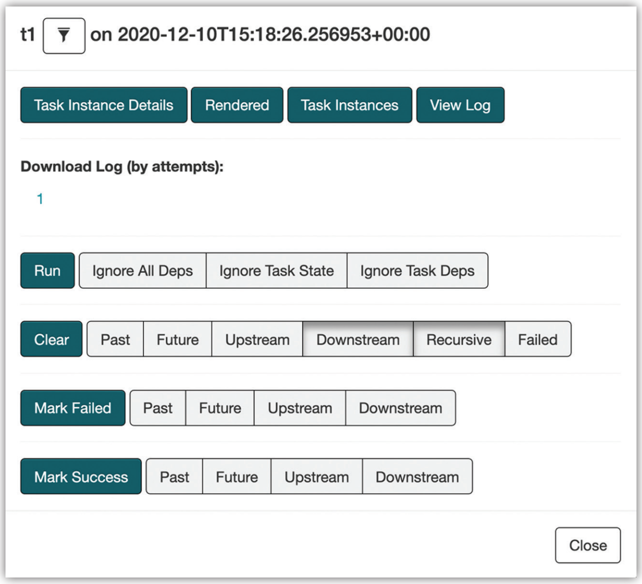
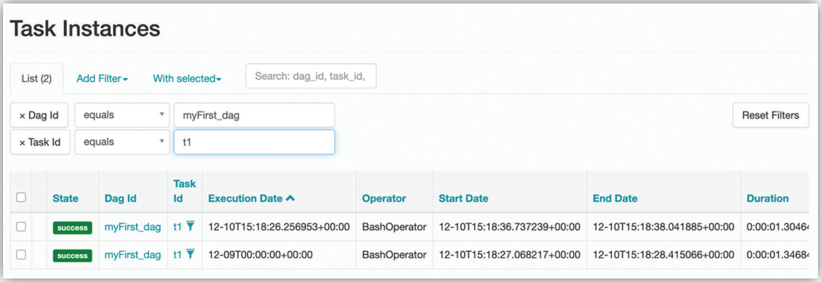


Figure 4.8: *Task execution details*

Let us quickly go over the purpose of each tab as follows:

1. **Task Instance Details** – It shows a summary of what the task does such as its attributes and rendered templates. We will cover templates later, but for now, think of it as a placeholder in DAG to replace with the run time values.
2. **Task Instances** – This is a historical view of the task. Refer to *figure 4.9*:

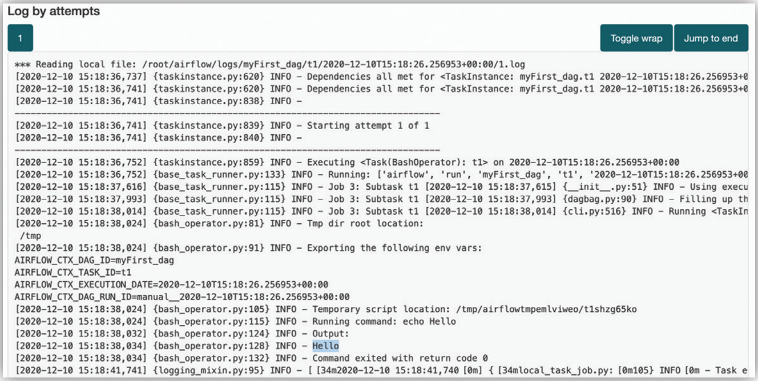


Filter		Value
× Dag Id	equals	myFirst_dag
× Task Id	equals	t1

	State	Dag Id	Task Id	Execution Date	Operator	Start Date	End Date	Duration
☐	success	myFirst_dag	t1	12-10T15:18:26.256953+00:00	BashOperator	12-10T15:18:36.737239+00:00	12-10T15:18:38.041885+00:00	0:00:01.3046
☐	success	myFirst_dag	t1	12-09T00:00:00+00:00	BashOperator	12-10T15:18:27.068217+00:00	12-10T15:18:28.415066+00:00	0:00:01.3468

Figure 4.9: *Task history view*

3. **View Log** - The logs are quite verbose but give you the complete picture. Notice the highlighted text – that was our task output.



```
*** Reading local file: /root/airflow/logs/myFirst_dag/t1/2020-12-10T15:18:26.256953+00:00/1.log
[2020-12-10 15:18:36,737] {taskinstance.py:620} INFO - Dependencies all met for <TaskInstance: myFirst_dag.t1 2020-12-10T15:18:26.256953+0
[2020-12-10 15:18:36,741] {taskinstance.py:620} INFO - Dependencies all met for <TaskInstance: myFirst_dag.t1 2020-12-10T15:18:26.256953+0
[2020-12-10 15:18:36,741] {taskinstance.py:838} INFO -
--------------------------------------------------------------------------------
[2020-12-10 15:18:36,741] {taskinstance.py:839} INFO - Starting attempt 1 of 1
[2020-12-10 15:18:36,741] {taskinstance.py:840} INFO -
--------------------------------------------------------------------------------
[2020-12-10 15:18:36,752] {taskinstance.py:859} INFO - Executing <Task(BashOperator): t1> on 2020-12-10T15:18:26.256953+00:00
[2020-12-10 15:18:36,752] {base_task_runner.py:133} INFO - Running: ['airflow', 'run', 'myFirst_dag', 't1', '2020-12-10T15:18:26.256953+00
[2020-12-10 15:18:37,616] {base_task_runner.py:115} INFO - Job 3: Subtask t1 [2020-12-10 15:18:37,615] {__init__.py:51} INFO - Using execu
[2020-12-10 15:18:37,993] {base_task_runner.py:115} INFO - Job 3: Subtask t1 [2020-12-10 15:18:37,993] {dagbag.py:90} INFO - Filling up th
[2020-12-10 15:18:38,014] {base_task_runner.py:115} INFO - Job 3: Subtask t1 [2020-12-10 15:18:38,014] {cli.py:516} INFO - Running <TaskIn
[2020-12-10 15:18:38,024] {bash_operator.py:81} INFO - Tmp dir root location:
 /tmp
[2020-12-10 15:18:38,024] {bash_operator.py:91} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_ID=myFirst_dag
AIRFLOW_CTX_TASK_ID=t1
AIRFLOW_CTX_EXECUTION_DATE=2020-12-10T15:18:26.256953+00:00
AIRFLOW_CTX_DAG_RUN_ID=manual__2020-12-10T15:18:26.256953+00:00
[2020-12-10 15:18:38,024] {bash_operator.py:105} INFO - Temporary script location: /tmp/airflowtmpemlviweo/t1shzg65ko
[2020-12-10 15:18:38,024] {bash_operator.py:115} INFO - Running command: echo Hello
[2020-12-10 15:18:38,032] {bash_operator.py:124} INFO - Output:
[2020-12-10 15:18:38,034] {bash_operator.py:128} INFO - Hello
[2020-12-10 15:18:38,034] {bash_operator.py:132} INFO - Command exited with return code 0
[2020-12-10 15:18:41,741] {logging_mixin.py:95} INFO - [ [34m2020-12-10 15:18:41,740 [0m] { [34mlocal_task_job.py: [0m105} INFO [0m - Task e
```

Figure 4.10: *Task execution log*

4. **Run** – Run allows to run the task with conditions that you can set on the right (ignore all deps, etc.).
5. **Clear** – This removes that task instance from the database. I would recommend using it to rerun the task.
6. **Mark Failed** – This sets the task status to fail. This could be helpful during testing to figure out if the exception handler is working fine.
7. **Mark Success** – This is the reverse for "failed." It is helpful during development when you don't want to wait for a long running task to complete.

After the graph view comes the tree view as follows:

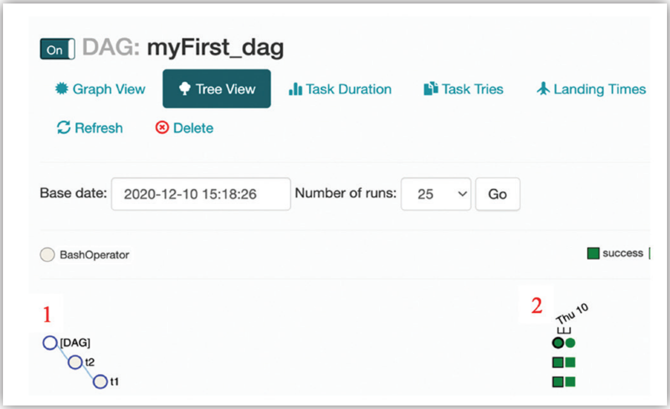


Figure 4.11: DAG run tree view

1. It shows the task dependency
2. It shows a summary of the past runs represented with circles and the associated tasks with square. There are two columns because the DAG was run twice. If you hover over the circle, the **`run_id`** field contains the indication of whether the DAG was run manually or via scheduler. Note the color coded legends. Green, here, indicates success.

It is time to look at the next tab – Task Duration. If you run the DAG multiple times, then this tab shows the duration of each task over the past N runs. This is very helpful to identify where your DAG is taking the most time.

The "*Task Tries*" shows the number of times your tasks have been executed over many runs. The Landing time allows you to compare how DAGs have performed over time.

The Gantt view allows to view the time duration of each task and how they executed, sequential or parallel. We will come back to this when we discuss the execution model. *Figure 4.12* shows a view of it. Notice that the tasks `t1` and `t2` executed sequentially:

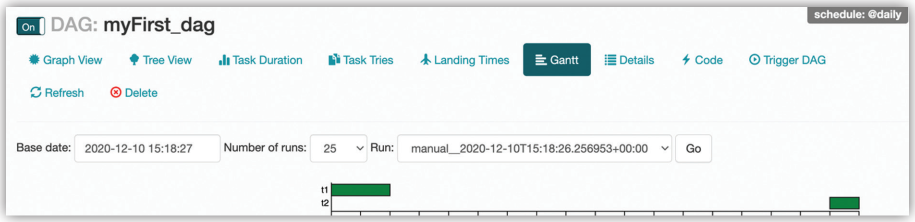


Figure 4.12: DAG run Gantt view

`Code` shows the DAG code. Use this to validate if your DAG changes have been read by Airflow. `Delete`, as the name suggests, deletes the DAG along with the metadata from the database.

So, you now have a good understanding of the Airflow UI. We will keep coming back to it as we move along. I suggest you take a break and then come back.

4.3.2.2 Airflow command line tour

While there are quite a few commands along with options, we will focus on a few important ones as follows. If you'll recall, you have already seen few commands before:

1. **airflow initdb:** This initializes the DB. If you have changed the DB, then this needs to be run again. We used this command in the entrypoint.sh file.
2. **airflow webserver:** This is used to run the webserver. We used this command in the entrypoint.sh file.
3. **airflow resetdb:** This rebuilds the meta database. Do remember that it resets everything. It is literally like a fresh install.
4. **airflow upgradedb:** Use this to upgrade the DB. For example, if a new version of Airflow is launched, then use this to upgrade the database. Also, if you change the executor (we will come back to it) in the configuration, then you need to upgrade the db.
5. **airflow scheduler:** Runs the Airflow scheduler. We used this in entrypoint. sh. This points to dag folder - `$AIRFLOW_HOME/dags`. It is so because of the predefined location for DAGs; they are automatically picked up and parsed.
6. **airflow lists_dags:** Lists all the DAGs parsed by scheduler. It is the same list that you saw on the DAG dashboard page. Here's a sweet trick – to check the location of a DAG file, go to the DAG details page and click on the `Details` tab and check the filepath attribute.
7. **airflow trigger_dag:** You have already seen this before. While triggering, you can specify the scheduled date. This leads to an important fact – Airflow stores the date in the UTC format.
8. **airflow list_dag_runs <Dag id>:** This shows the history of the DAG runs for that specified DAG ID.
9. **airflow list_tasks <Dag id>:** As the name suggests, it lists the tasks of the specified DAG_id. Can you recall which screen shows this information on UI?

10. **airflow test <Dag id> <test id>** - This is useful during development as it allows to execute a task without checking for dependency or storing its state in the database.

Now that you know how to execute commands, let us understand how the commands are executors. We will next look at executors.

4.4 Executors in Airflow

As the name suggests, executors execute the scheduled tasks. But there is more to them than just executing tasks. Let us look at the task execution process as follows. Understanding the process helps in understanding the different types of executors better.

1. When a task is scheduled by turning on the toggle, all the details of the task like status, scheduled time, etc. are stored in the database.
2. Next, the schedular periodically checks if there is any task to run. If a task is ready for trigger, then it is added to the queue of the **executor**.
3. **Executor** is responsible for the task execution. It reads the task from the queue and then specifies how to execute it. If the executor is sequential, then the tasks will be executed in a sequential manner. It is important to note that executor only provides the context for the job and doesn't execute the job. Airflow offers the following types of executors:

Sequential

Think of a task getting executed in a linear fashion. It uses SQLite as the backend. It is the default option when you install Airflow.

Local

Local executor allows you to execute tasks that can run in parallel on a single machine. For each task, a new process will be spawned. Under the hood, it uses the Python multi-processing features. It is limited by the number of cores that you have on your machine. While it offers parallelism cheaply, it comes with limitations as well. A single machine means a single point of failure. This would not work if you had multi-node architecture.

SQLite cannot be used as the DB for the local executor as it doesn't support concurrent write. PostgreSQL (https://www.postgresql.org/) relational DB is a popular open-source choice for it. Due to client server architecture, it supports concurrent access and is ACID compliant. To change from the default sequential executor, you need to change the executor entry in the **`airflow.cfg`** file. This file is the configuration file for the Airflow setup. It will be present under the **`$AIRFLOW_HOME`** folder:

```
[core]

executor = LocalExecutor
```

You also have the option of specifying in the environment variable. Look at the environment variables that we defined and used during set up. We specified **`LocalExecutor`** and postgreDB as the backend. To be explicit, the Docker + Airflow setup precedingly mentioned already uses Local executor and PostgreSQL DB and that you need to change it.

I would recommend you to go through the configuration documentation page for Airflow: (https://airflow.readthedocs.io/en/1.9.0/configuration.html#) for further details.

With Local executor, we get the benefit of concurrency and parallelism. Let us explore some of the configurations that tweak the concurrency and parallelism.

Consider the DAG in *figure 4.13*:

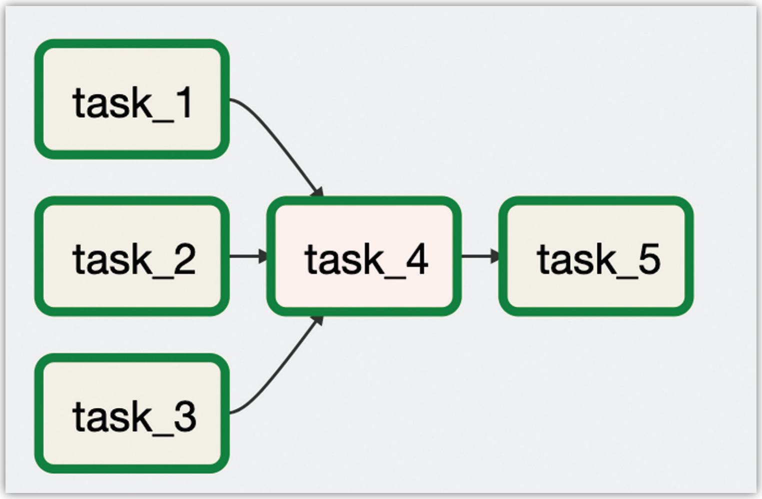


Figure 4.13: *DAG consisting of 5 tasks*

As can be seen tasks 1, 2 and 3 are independent; however, 4 and 5 are sequential. If you check the **Gantt view** for the preceding DAG with our settings, it looks like *figure 4.14*:

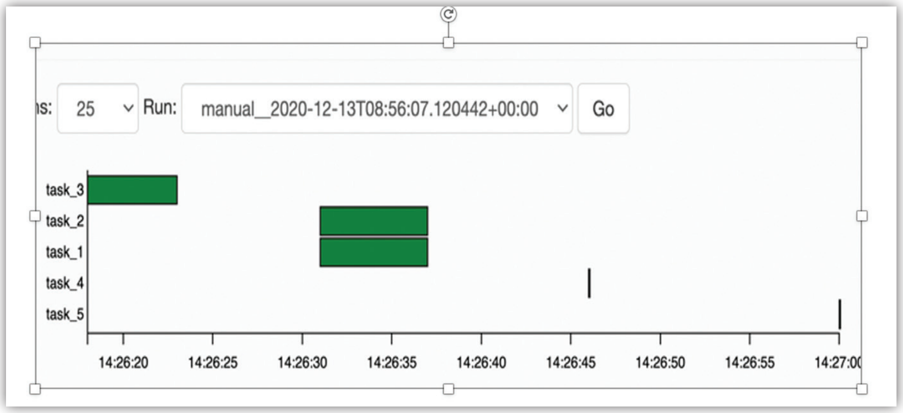


Figure 4.14: *Execution chart for DAG (Figure 4.13)*

While tasks 1 and 2 were executed in parallel, task 3 was executed beforehand which you may consider weird given that as per DAG, all 1, 2, and 3 can execute in parallel. Since you know that local executor can execute tasks in parallel, were why 1, 2 and 3 not executed in parallel? What is going on here?

The number of allowed parallel tasks is controlled with the parallelism variable in **`airflow.cfg`**:

```
# The amount of parallelism as a setting to the executor. This defines

# the max number of task instances that should run simultaneously

# on this airflow installation

parallelism = 2
```

This explains why there were only two parallel task executions in the Gantt view (*figure 4.14*). Can you guess what the Gantt view would like if you set the parallelism to 1? Well, that is no surprise as shown in *figure 4.15*:

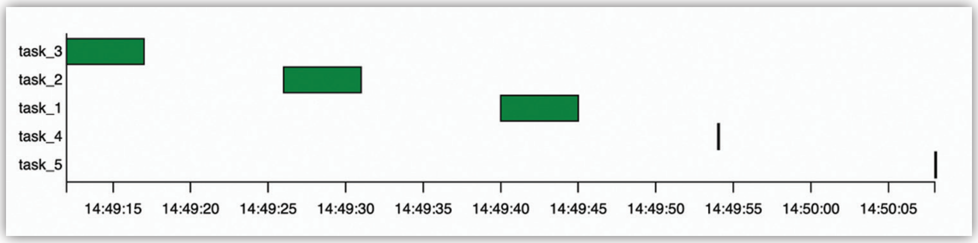


Figure 4.15: *DAG execution with parallelism value set to 1.*

Finally, to conclude, let us try with the value set to 3:

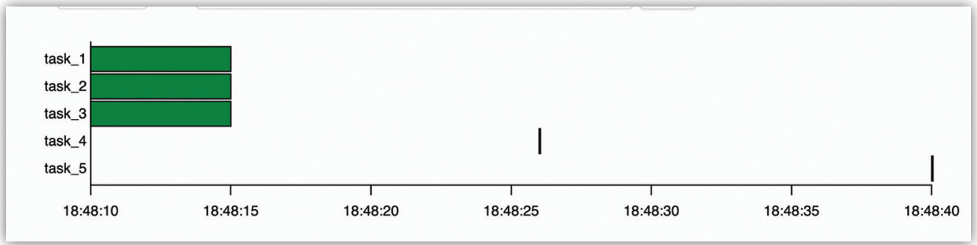


Figure 4.16: *Gantt view for a DAG with parallelism set to 3*

Let us look at another concurrency parameter name *dag_concurrency*:

```
# The number of task instances allowed to run concurrently by the scheduler

Dag_concurrency = 2
```

As the comments explain, it controls the number of tasks that can run in parallel in DAG run. A pop quiz for you: if we leave the parallelism parameter value as it is but change the **`dag_concurrency`** value to 2, how would Gantt like it?

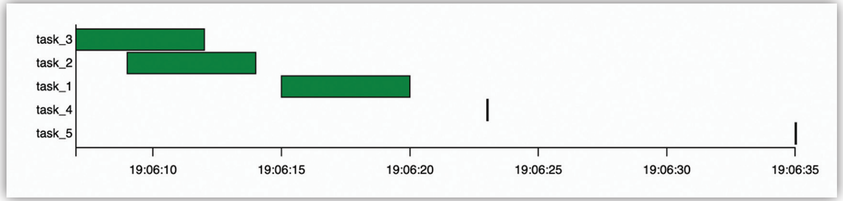


Figure 4.17: *Gantt view with parallelism set to 3 and dag_concurrence set to 2*

You can see that while tasks 2 and 3 were executed partially parallel (indicating independency), task 1 had to wait even though it was also independent . As an exercise, what happens if you set the parallelism to 1 and `dag_concurrency` to 2? Think about the answer and then validate it.

It is important to remember the global and local effect of parallelism and `dag_concurrence`. Parallelism affects all the DAG runs whereas `dag_concurrency` is restricted to a particular DAG run. For example, if parallelism is set to 4 and `dag_concurrence` is set to 2, then two DAGs can run with two tasks from each DAG respectively in parallel.

Let us look at another configuration option – `max_active_runs_per_dag`. This controls the maximum number of active DAG runs active per DAG:

```
max_active_runs_per_dag = 16
```

Imagine a scenario where you are running a sales report for the past one year month-wise. In Airflow, it is possible to run a DAG in the past. The scheduler triggers a backfill of all DAG run instances for which it has no status recorded, starting with the `start_date` that you specify in your `default_args`. Scheduling is a full section in itself and we will come back to it.

For our scenario, we are instructing Airflow to schedule the DAG from one year which means Airflow needs to run the DAG 11 times for the past months. `max_active_runs_per_dag` controls how many such DAG runs can be executed in parallel. The scheduler will not create new runs once this limit is hit.

To get it clearly, let us change the value 1 and see the results. For you to try, please change the DAG initialization code to *code 4.2*:

```
default_args = {

        "owner": "airflow",

        "start_date": airflow.utils.dates.days_ago(12)

    }


with DAG(dag_id="myFirst_dag", default_args=default_args,

max_active_runs=1, schedule_interval="@daily", catchup = True) as dag:


    t1 = BashOperator(

            task_id="t1",

            bash_command="echo Hello"

        )


    t2 = BashOperator(

            task_id="t2",

            bash_command="echo World"

        )


    t1 >> t2

with DAG(dag_id="myFirst_dag", default_args=default_args,
max_active_runs=1, schedule_interval="@daily", catchup = True) as dag:
```

Code 4.2: *DAG with controlled active runs*

We are yet to go through the details of the DAG initialization, but the following few details can be easily understood here:

- **`max_active_runs:`** The maximum number of active runs for this DAG. The scheduler will not create new active DAG runs once this limit is hit. Defaults to `core.max_active_runs_per_dag` if not set.
- `default_args` has `start_date` as 12 days before the current date.
- `Schedule_interval` sets the frequency and that means code 4.2 will run 12 times or 12 DAG runs.
- `Catchup` flag decides if the backfilling needs to be done or not. This is a bit strange in the beginning as we are used to thinking of scheduled jobs as something that will execute "*from now onwards*." With Airflow, it becomes – even though I am setting it up today, I would like it to execute from the past date or in Airflow terms – "*backfill*"

The tree view of the *code 4.2* would be as shown in *figure 4.18*:

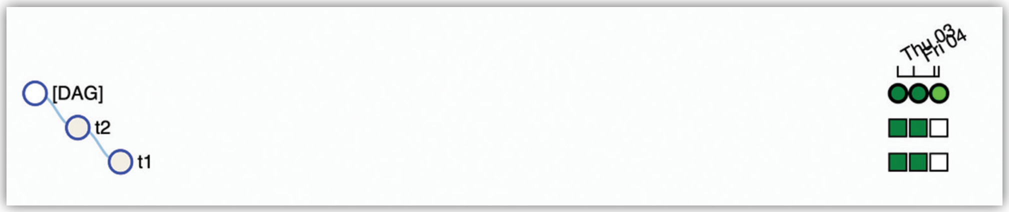


Figure 4.18: *Dag run with max runs set to 1.*

The shinning green (3rd column) is only kicked off when the second column DAG run is complete. If you change the **`max_active_runs 2`**, then you have two active DAG runs as shown in *figure 4.19*:

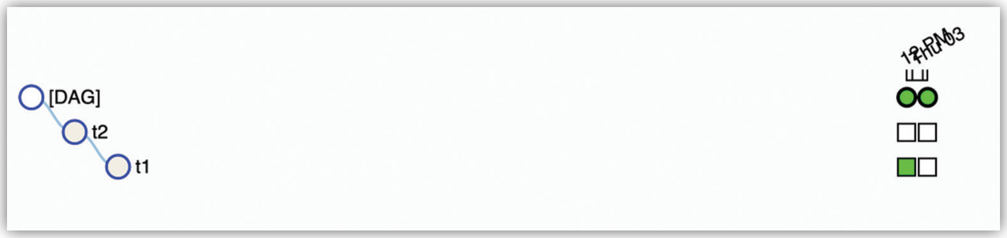


Figure 4.19: *(Two active DAG runs).*

Celery

If you have 100s of DAGs to run, then `LocalExecutor` is not sufficient. You need to distribute the work on multiple worker nodes. This is what celery offers. Celery is an asynchronous distributed task queue to distribute tasks among worker nodes. Celery is a widely used Python package that makes it very easy to run jobs or tasks in the background.

A worker is created using the `airflow worker` command and that starts an Airflow daemon managing one to many celery processes to execute a task for a given DAG. So, with the increase in load, you add a new worker (think of a new machine) and you apply the multiprocessing paradigm, that is, spawn multiple new processes on worker nodes. If you understand horizontal scaling, then you'll know that the idea is quite similar here. If a particular worker node goes down, then the celery redistributes that task to other available worker nodes which is not possible in local executor.

Since it is a multi-worker management, you need a message broker like RabbitMQ or Redis. (https://docs.celeryproject.org/en/stable/getting-started/introduction.html). An architecture diagram as shown in *figure 4.20* would be helpful here:

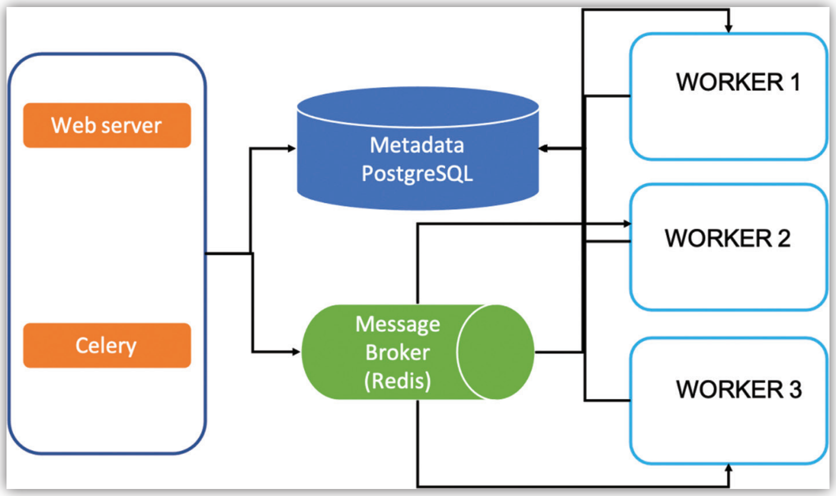


Figure 4.20: *Airflow + celery architecture.*

Let us do a dry run of the architecture as follows to have the flow clear in our mind:

1. Web server and scheduler are co-located. As soon as a task is scheduled, the webserver makes that entry in the metadata DB.
2. When a task is ready to trigger, it pushed to the message broker by celery.

3. All the worker nodes pull the task from the message broker periodically. The worker node updates the status of the task in the metadata DB and executes it. Also, when the task is complete, the worker updates the final status.

Due to the distributed architecture, it is possible to define the affinity of certain tasks to a particular node. Think of tasks that are CPU intensive. Then maybe you configure and designate worker 1 with a high-end processor and large memory to process CPU bound jobs fast. I guess you have now enough reason to accept that Airflow + celery is a good choice for production situations.

Nothing comes free in the world. The following are a few drawbacks that you need to be aware of in order to build the appropriate solution:

1. It needs separate message broker setup and maintenance. This also means a new failure point
2. There is complexity in the architecture.
3. Tied to points 1 and 2, there is a higher cost.

To see these ideas in action, let us run our DAG with a celery executor. Since celery is a different Python package and includes components like queue, database, etc., setting it up correctly could be little tricky.

We will use Docker to make things easier for us. We will use puckel's docker-Airflow (https://github.com/puckel/docker-airflow) to run it as at the time of writing this book, we couldn't find a complete Docker on the Airflow Docker page.

This contains six containers which are as follows:

- **Webserver –** Runs the Airflow UI.
- **Redis –** This is the message broker to queue tasks and execute them.
- **Worker –**The Celery worker which keeps on polling on the Redis process for any incoming tasks; then processes them and updates the status in Scheduler. These are the right-hand side blocks in *figure 2.12*.
- **Flower –** Flower is a web-based tool for monitoring and managing celery clusters (https://flower.readthedocs.io/en/latest/).
- **Scheduler –** Airflow Scheduler is used to queue tasks on Redis. These are the queued tasks that are picked and processed by Celery workers.
- **Postgres –** The backend database for Airflow to persist, manage, and display DAGs' state and other information.

To run the Dockers, make sure that you have Docker installed on your machine and then use the following commands:

```
$ git clone https://github.com/puckel/docker-airflow
```

```
cd docker-airflow
```

You might have to modify the mounting path in case your DAG location is different. We are mapping **`dags`** folder from host to docker **`dags`** location:

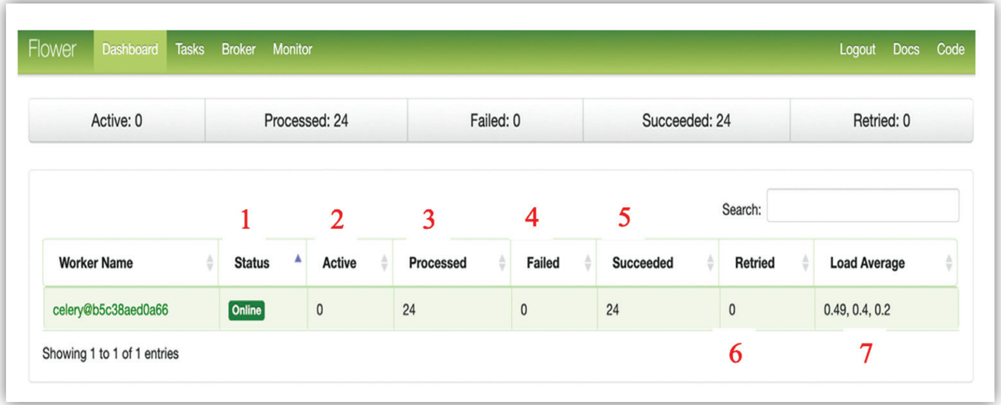
```
volumes:

      - ./dags:/usr/local/airflow/dags
```

To run, we will use the `Docker-compose` command:

```
docker-compose -f docker-compose-Celery.yml
```

We will stick to the *code 4.2* DAG. To view the execution, we need two browser tabs this time – one for Airflow (http://localhost:8080) and the other for flower (http://localhost:5555). Flower would show the celery execution.

Run the DAG (*code 4.2*) from Airflow UI (Ensure that the toggle is set to ON). Since you have already used Airflow UI, let us take a look at flower UI:



Active: 0 | Processed: 24 | Failed: 0 | Succeeded: 24 | Retried: 0

Worker Name	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@b5c38aed0a66	Online	0	24	0	24	0	0.49, 0.4, 0.2

Showing 1 to 1 of 1 entries

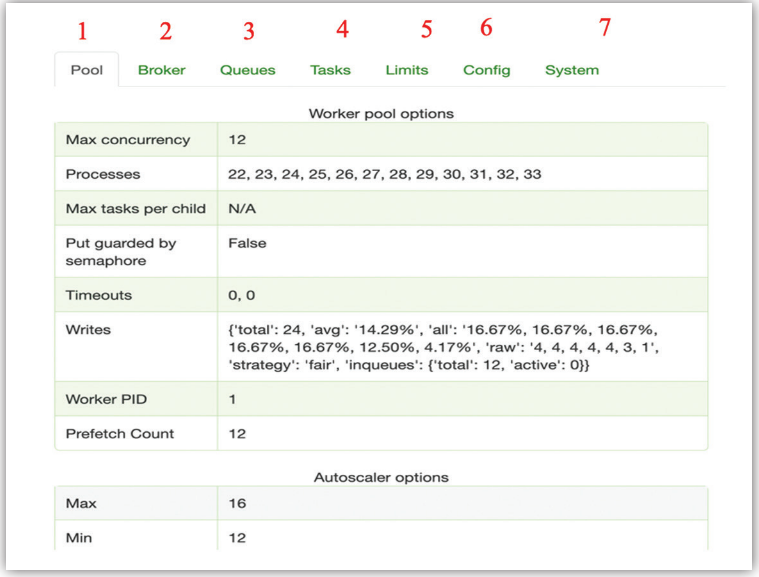
Figure 4.21: *Flower UI showing the status of the celery tasks*

Let us explore the UI as follows:

1. The status could be online or offline
2. The number of active tasks. It is 0 because DAG has finished running.
3. The total number of processed tasks.
4. The number of failed tasks.
5. The number of succeeded tasks.
6. The number of retired tasks. This is 0 because there were no failures.

7. The load average column has 3 values corresponding to the number of active processes averages over the last 1, 5, and 15 minutes respectively. Those numbers are important as they give information about the system load of your worker. So, how do you interpret those numbers? Well, in general, since each CPU core can handle one process at a time, the system isn't overloaded until it goes over 1.0 per processor. For example, I have 6 cores in total on my machine and if it stays below 6, it is good. Any value more than 4-5 generally indicates a problem. In case you run into high CPU usages, then you may consider adding a high CPU + RAM worker and directing the high CPU oriented task to that worker.

   Let us explore the various tabs of flower. Click on the name of the worker in the flower UI as shown in *figure 4.22*:



Worker pool options

Max concurrency	12
Processes	22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33
Max tasks per child	N/A
Put guarded by semaphore	False
Timeouts	0, 0
Writes	{'total': 24, 'avg': '14.29%', 'all': '16.67%, 16.67%, 16.67%, 16.67%, 16.67%, 12.50%, 4.17%', 'raw': '4, 4, 4, 4, 4, 3, 1', 'strategy': 'fair', 'inqueues': {'total': 12, 'active': 0}}
Worker PID	1
Prefetch Count	12

Autoscaler options

Max	16
Min	12

Figure 4.22: *Flower tabs*

8. The worker pool gives you the maximum number of processes allowed to run.
9. The message broker used and that is redis in our case.
10. The queue where the tasks are sent and then pulled by the worker to execute.
11. The tasks processed by the worker.
12. On the limits tab, you can set the time limit before the process executing the task gets terminated and rate the limit to define the maximum number of tasks per minute.
13. **Configuration**, as the name suggests, shows the configuration of celery.

14. **System** tab shows you the system usage statistics.

    For celery executor, there are a few more parallelism and concurrency related configurations that are important to know. Again, they get defined in airflow.cfg file

    ```
    Worker_concurrency = 16
    ```

    The above concurrency will be used by celery when starting a new worker. We will go through the steps to create a new worker shortly but at a high level, this essentially defines the number of task instances a worker would take. This should be set based on the available resources and your task needs.

Let us add a new worker node. Note that in the `docker-compose`, file we had only one worker defined. So, the new worker will be a new machine and that means we need to add a new container to the already running network. For us, the existing network would come from `docker-compose-celery.yml` that you saw earlier.

When you run the `docker compose` command, it creates a default network for you. Creating network "`code_default`" with the default driver.

The new worker needs to attach to this network. To do this fortunately, there is a docker command for it. Make sure that you specify the full path of the Docker host as follows:

```
docker run --network code_default --expose 8793 -v <Full_path_of_host_
dags>/dags/:/usr/local/airflow/dags   -v   <Full_path_of_cfg>/airflow.cfg:/
usr/local/airflow/airflow.cfg  -dt python:3.7
```

We have specified the network that it should connect to. Then we exposed the port used by Airflow to serve the worker local log files to the Airflow main web server. A tiny web server subprocess is started when an Airflow worker is started. It serves the workers local log files to the Airflow main web server, which then builds pages and sends them to the users. In other words, this defines the port on which the logs are served. It needs to be unused, and should be open and visible from the main web server to connect onto the workers:

```
worker_log_server_port = 8793
```

This value is defined in the `airflow.cfg` file. Next, we need to bind two volumes corresponding to the `dags` folder and the `airflow.cfg` file. This is required because each of the workers will have Airflow installed and it important for all the workers to have the same configurations.

If you check flower (http://localhost:5555), you will only see one worker and that is because we are yet to set up Airflow on the new worker. It is time to get inside the new worker container. Use the `docker exec` command to connect to an already running container as follows:

1. Connect to the new worker docker container:

   ```
   Docker exec -it <container_id> /bin bash
   ```
2. Add the path to Airflow dags:

   ```
   export AIRFLOW_HOME=/usr/local/airflow/
   ```
3. Add the users:

   ```
   useradd  -ms /bin/bash -d $AIRFLOW_HOME airflow
   ```
4. Install Apache airflow:

   ```
   pip install "apache-airflow[celery, crypto, postgres, redis]==1.10.9"
   ```
5. Initialize the DB:

   ```
   airflow initdb
   ```
6. Change the ownership to Airflow user:

   ```
   chown -R airflow: $AIRFLOW_HOME
   ```
7. Log in as Airflow:

   ```
   su – airflow
   ```
8. To ensure that the worker is connected to the Airflow backend, change the DB connection string in its `airflow.cfg` file:

   ```
   sql_alchemy_conn=postgresql+psycopg2://{DATABASE_USER}:{DATABASE_
   USER_PASSWORD}@{DATABASE_HOST}:{DATABASE_PORT}/airflow
   ```
9. We need to change Celery `result_backend` in `airflow.cfg`. When a job finishes, it needs to update the metadata of the job. Therefore, it will post a message on a message bus or insert it into a database. In our case, it is the PostgreSQL database. This status is used by the scheduler to update the state of the task. Based on our configuration, it would look as follows:

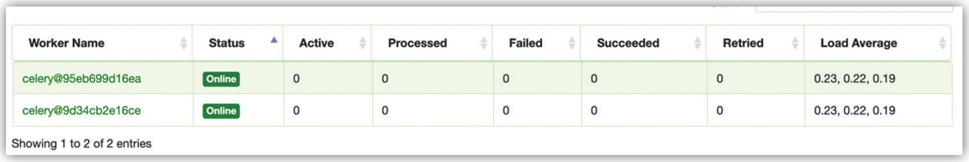
   ```
   result_backend = db+postgresql://airflow:airflow@postgres:5432/airflow
   ```
10. We also need to update the message broker URL to point to redis. With our configuration, it would look as follows:

    ```
    broker_url = redis://:redispass@redis:6379/1
    ```
11. Export the variable again:

    ```
    export AIRFLOW_HOME=/usr/local/airflow/
    ```
12. Run the airflow worker command to add the node:

    ```
    airflow worker
    ```

13. Go to http://localhost:5555 and if you see two workers, then congratulations! You just added a new worker to the Airflow setup.



Worker Name	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@95eb699d16ea	Online	0	0	0	0	0	0.23, 0.22, 0.19
celery@9d34cb2e16ce	Online	0	0	0	0	0	0.23, 0.22, 0.19

Showing 1 to 2 of 2 entries

Figure 4.23: *Two celery workers*

If you thought that the steps were cumbersome, then there is an extremely easy way to add worker nodes, thanks to Docker!

```
docker-compose -f docker-compose-celery.yml scale worker=2
```

Now that we know how to add workers, let us take advantage of it. It is possible to dedicate certain workers for some specific tasks. For example, CPU bound tasks can be sent to a machine with a high RAM and a powerful core/processor. Or, all Spark (https://spark.apache.org/) tasks should only be sent to a machine running the Spark process. To summarize, we would like to specialize workers either from a resource or an environment perspective.

To dedicate a worker for type(s) of work, we need to attach queues to the worker. This is done by specifying -q option to the airflow worker command.

```
airflow worker -q queue_1, queue_2
```

By default, the worker listens on the default queue and is specified via the `default_queue` attribute specified in **`airflow.cfg`**:

```
default_queue = default
```

Let us now dedicate workers to certain queues. We will define two queues – one dedicated for IO and the other for CPU bound operations respectively. This is as follows:

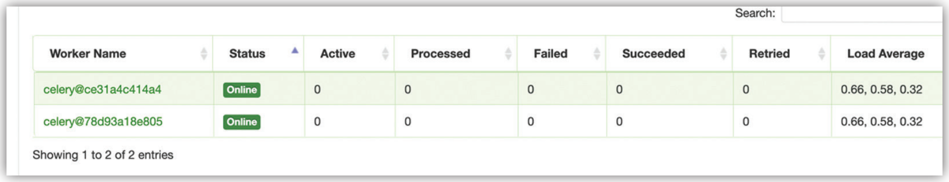
1. In the Docker compose file **`docker-compose-celery.yml`**, add the following to the worker service:

   ```
   command: worker -q worker_IO, worker_CPU,default
   ```

2. Run the docker-compose command:

   ```
   docker-compose -f docker-compose-celery.yml up --scale worker=2
   ```

3. Open the flower UI – http://localhost:5555 and you would see two workers as follows:

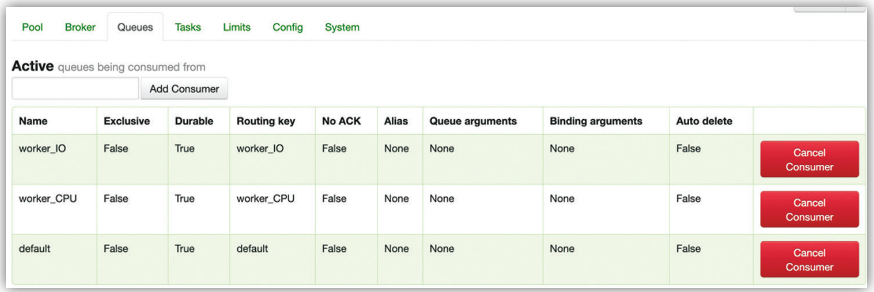

Search:

Worker Name	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@ce31a4c414a4	Online	0	0	0	0	0	0.66, 0.58, 0.32
celery@78d93a18e805	Online	0	0	0	0	0	0.66, 0.58, 0.32

Showing 1 to 2 of 2 entries

Figure 4.24: *Two workers listening to specific queues*

4. Click on any worker, and then on the next page, select the **`Queue`** tab:



Active queues being consumed from

Name	Exclusive	Durable	Routing key	No ACK	Alias	Queue arguments	Binding arguments	Auto delete	
worker_IO	False	True	worker_IO	False	None	None	None	False	Cancel Consumer
worker_CPU	False	True	worker_CPU	False	None	None	None	False	Cancel Consumer
default	False	True	default	False	None	None	None	False	Cancel Consumer

Figure 4.25: *Named worker queues*

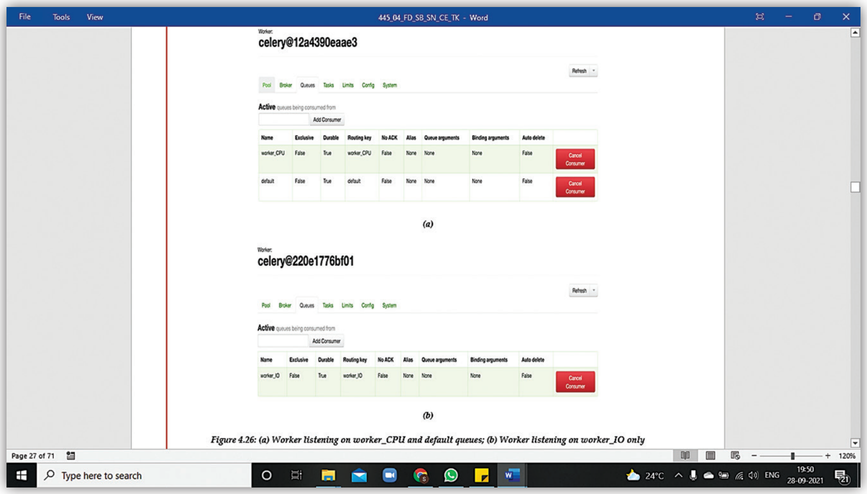
5. We need to modify the DAG to point the tasks to the respective queue. Recall that by default, all tasks go to the default queue specified the **`default_queue`** parameter in **`airflow.cfg`**. The partial DAG would look like *code 4.3*:

```
with DAG(dag_id="myQueue_dag", default_args=default_args,

max_active_runs=2, schedule_interval="@daily", catchup = True) as dag:
t1_IO = BashOperator(
      task_id="t1_IO",
      bash_command="echo Hello IO",
      queue="worker_IO" --1
  )

 t2_CPU = BashOperator(
      task_id="t2_CPU",
      bash_command="echo World of CPU",
      queue="worker_CPU" --2
  )

t3_Default = BashOperator(
      task_id="t3_Default",
      bash_command="echo Default Of the World"
  ) --3


t1_IO >> t2_CPU >> t3_Default
```

Code 4.3: *With tasks pointing to specific queues*

Code 4.3 is not very different from the earlier DAG that you have seen other than the new **`queue`** attribute:

1. For the task `t1_IO`, we have specified the `worker_IO` queue and that means all the IO tasks would go this container (worker).
2. For the task `t2_CPU`, we have specified the `worker_CPU` queue and that means all the CPU bound tasks would go to this container (worker).
3. For the task `t3_default`, we have not specified any queue and that means it goes to the default `worker_IO` queue specified in the `airflow.cfg`.
4. Next, we need to define which queue each worker should consume. This is required because the command for creating the queues has been executed for each worker using `docker-compose`. They all are listening to the same queues. On the flower page (*figure 2.24*), remove those queues that you don't want your worker to listen to by clicking on the `Cancel Consumer` button (last column). Mine looks like *figures 4.26(a)* and *(b)*:



Worker:
celery@12a4390eaae3

Active queues being consumed from

Name	Exclusive	Durable	Routing key	No ACK	Alias	Queue arguments	Binding arguments	Auto delete	
worker_CPU	False	True	worker_CPU	False	None	None	None	False	Cancel Consumer
default	False	True	default	False	None	None	None	False	Cancel Consumer

(a)

Worker:
celery@220e1776bf01

Active queues being consumed from

Name	Exclusive	Durable	Routing key	No ACK	Alias	Queue arguments	Binding arguments	Auto delete	
worker_IO	False	True	worker_IO	False	None	None	None	False	Cancel Consumer

(b)

Figure 4.26: (a) Worker listening on worker_CPU and default queues; (b) Worker listening on worker_IO only

Figure 4.26: *(a) Worker listening on worker_CPU and default queues; (b) Worker listening on worker_IO only*

Limiting **parallelism** with task prioritization

It is no secret that systems get overwhelmed when too many processes request it at the same time. Imagine you have a DAG that has three tasks to be executed in parallel. Let us say that all three tasks pull data from a rest service in parallel (recall the parallelism parameter) but this API only allows two requests at a time leading to throttling error.

So, how can you limit the number of requests? This is where pools come into the picture. A pool is a great way to limit the number of concurrent instances of a specific type of task to avoid overwhelming the source or destination.

By using pools, you can limit the number of tasks requesting the API by defining several slots and assigning the two tasks to this pool. With this, all the three tasks won't execute in parallel as shown below:

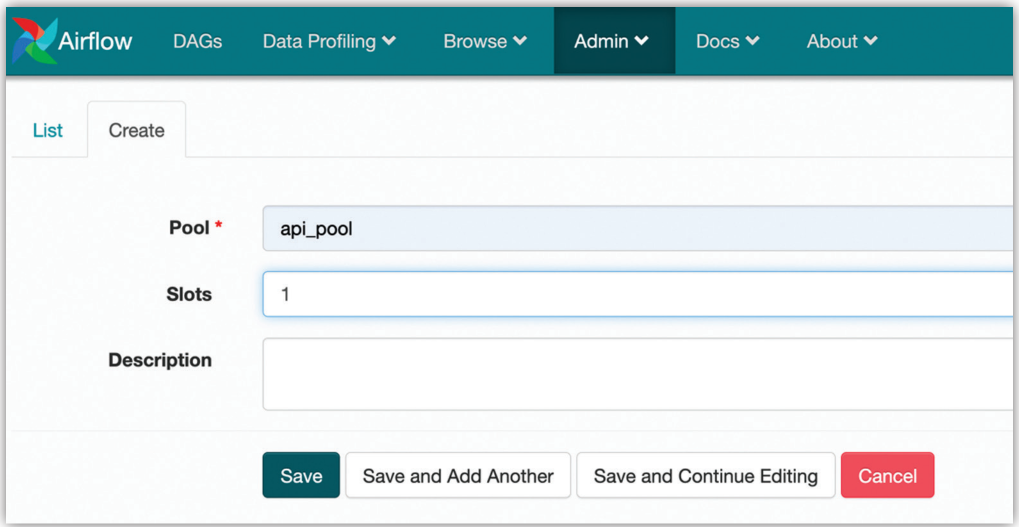
1. Let us define a DAG that has a few parallel tasks:

```
import airflow.utils.dates

from airflow import DAG

from airflow.operators.bash_operator import BashOperator

from datetime import datetime, timedelta


default_args = {

        "owner": "airflow",

        "start_date": airflow.utils.dates.days_ago(12)

    }


with DAG(dag_id="myPool_dag", default_args=default_args,

max_active_runs=2, schedule_interval="@daily", catchup = True) as dag:


    t1 = BashOperator(

            task_id="t1",

            bash_command="echo Hello t1"
```

```
      )


     t2 = BashOperator(

            task_id="t2",

            bash_command="echo Hello t2"

      )


     t3 = BashOperator(

            task_id="t3",

            bash_command="echo Hello t3"

      )


    t4 = BashOperator(

            task_id="t4",

           bash_command="echo Hello t4"

      )


     t5 = BashOperator(

            task_id="t5",

            bash_command="echo Finally All."

     )

     [t1,t2,t3,t4] >> t5
```

Code 4.4: *DAG with 4 parallel tasks*

2. Pools are created via the airflow UI (**`Menu | Admin | Pools`**). Let us create one with a single slot. This means only that task can be executed at a time:



Pool * api_pool

Slots 1

Description

Figure 4.27: *Pool creation page*

3. Go back to your DAG and add the `pool` parameter to the task. This means tasks 1, 2, 3, and 4 are assigned to `api_pool` and that pool executes one task at a time:

```
t1 = BashOperator(

        task_id="t1",

        pool='api_pool',

        bash_command="echo Hello t1"

    )


t2 = BashOperator(

        task_id="t2",

        pool='api_pool',

        bash_command="echo Hello t2"

    )
t3 = BashOperator(
        task_id="t3",
        pool='api_pool',
        bash_command="echo Hello t3"
    )


t4 = BashOperator(
        task_id="t4",
        pool='api_pool',
        bash_command="echo Hello t4"
    )
```

Code 4.5: *Pool assigned tasks*

4. Execute the DAG and if you check the Gantt view it would look like *figure 4.28*:

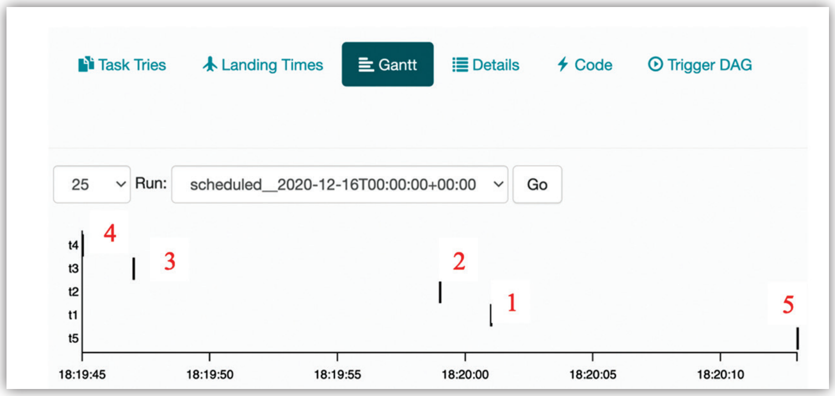


Figure 4.28: *pooled tasks*

5. Had they not been pooled; the execution would have looked like *figure 4.29*. 1, 2 were executed in parallel and then 3, 4 were executed in parallel. But shouldn't 1,2, 3, and 4 have executed in parallel? Well, that is because parallelism parameter in airflow.cfg was set to 2. Recall that parallelism defines the maximum number of tasks that can simultaneously happen:

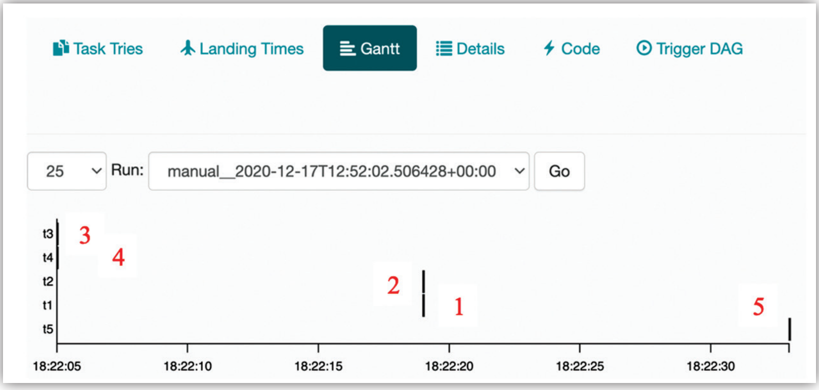


Figure 4.29: *Non-pooled tasks*

6. Look at *figure 4.29*. While tasks 1, 2, 3, and 4 were executed sequentially, they were not in the order that we expected them to. Let us say we want to order the execution sequence in a pool. priority_weight parameter allows you to define the tasks priority. It essentially defines priorities in the queue and in what sequence tasks get executed as the pool slots become available.

   Code 4.6 shows that parameter in the DAG. I have only kept the tasks code to avoid repetition.

```
t1 = BashOperator(

          task_id="t1",

          pool='api_pool',

          priority_weight = 1,

          bash_command="echo Hello t1"

      )


  t2 = BashOperator(

          task_id="t2",

          pool='api_pool',

          priority_weight = 2,

          bash_command="echo Hello t2"

      )
```

```
    t3 = BashOperator(

            task_id="t3",

            pool='api_pool',

            priority_weight = 3,

            bash_command="echo Hello t3"

        )


    t4 = BashOperator(

            task_id="t4",

            pool='api_pool',

            priority_weight = 4,

            bash_command="echo Hello t4"

        )
```

Code 4.6: *Tasks with priority_weight*

By default, the tasks are not assigned to any pool and the execution mode is decided by the executor settings.

So, how do you decide between `LocalExecutor` and celery? Start with `Localexecutor` and switch to celery when you hit the limits of Local executor. Just because celery is scalable doesn't means everyone needs it.

- **K8s – (https://kubernetes.io/):** The purpose of the *KubernetesExecutor* is to take advantage of Kubernetes to scale and maximize the resources optimization. Under the hood, each task that needs to run, talks to Kubernetes API to lunch pods for the task execution. I prefer to remember with this simple equation – `TASK = POD`. I leave this executor as an exercise for you to understand (https://airflow.apache.org/docs/apache-airflow/stable/executor/kubernetes.html). With the understanding gained so far, please trust me, it is possible. Do note that it is a recent addition to Airflow (Apache Airflow 1.10) which means there might be some issues. At the moment, Celery executor is the popular choice for production.

Now that you have a good understanding of the setup and its configurations, we will switch gears to understand the DAG. As a developer, most of your time will be spent there.

4.5 Anatomy of a DAG

In order to understand the DAG, let us relook at one previously built DAG (code 4.1):

```
import airflow.utils.dates

from airflow import DAG --1

from airflow.operators.bash_operator import BashOperator --2

from datetime import datetime, timedelta


default_args = {

        "owner": "airflow",

        "start_date": airflow.utils.dates.days_ago(1)

    } --3


with   DAG(dag_id="myFirst_dag",   default_args=default_args,   schedule_
interval="@daily") as dag: --4


    t1 = BashOperator( --1

            task_id="t1",

            bash_command="echo Hello"

        ) --5


    t2 = BashOperator( --2

            task_id="t2",

            bash_command="echo World"

        )


   t1 >> t2  --6
```

Let us unpack the code as follows:

1. The DAG modules from the Airflow contains the functions to build to run the DAG. It is a mandatory import for any DAG.
2. **`BashOperator`** are responsible to execute bash scripts. Another example would be **`PythonOperator`** that helps you to execute Python code snippets. At a high level, DAG is a collection of tasks and Airflow operators executes those tasks. We will look at different operator types as we move along.

   Bash and Python are generic operators but there are other use case specific operators as well. For example, **`EmailOperator`** sends out notifications and **`HTTPOperator`** calls API endpoints. Tasks are generally considered as a unit of work and operators help perform that unit of work. Since we are only calling bash commands in the tasks, only **`BashOperator`** needs to be imported.

   If "task" creates confusion, then think of units of work as operations then DAG is an orchestration of operators performing different operations. With the construct of operator, it becomes easier to understand and manage the execution:

   a. Which operator should execute and when?
   b. What is the state of an operator?
   c. What to do when an operator completes or fails?

   While there are many more pieces, the preceding points clearly highlight the value of pre-defined operators.
3. The tasks may need certain common arguments. To pass those values to the tasks, one option would be to pass those values to each task constructor. Fortunately, Airflow let's you define **`default_args`** in a JSON format and then it can be used during DAG initialization and that in turn gets passed along to the tasks. Here, we have default arguments:

   a. **Owner** – Owner of the DAG
   b. **Start_date** – The start date specifies the date from which the DAG should start running.
4. The DAG object is instantiated. The **`with`** style object initialization makes the object lifetime management easier for the developer. You would find this style in almost all the DAG examples. There are certain constructor parameters that are being passed along to the DAG object initialization.

   a. **`dag_id`** – This is the unique identifier of the DAG and this is the name you see on the DAG dashboard.
   b. **`default_args`** – This defines the default arguments parameter.

   c. **`schedule_interval`** – This defines the interval at the which the DAG should run. If you have ever scheduled a job, then you know what this means. If not, then **`@daily`** sets the DAG to run daily once. If it is daily, then you would expect other options, and yes, there are options. Scheduling requires more understanding as it must cater to a lot of different use cases. We will explore it later.

   The non with block style DAG initialization is shown in code 4.7:

```
dag = DAG(

dag_id="myFirst_dag", default_args=default_args, schedule_
interval="@daily"

)

    t1 = BashOperator(

            task_id="t1",

            bash_command="echo Hello",

        dag=dag

        )
```

Code 4.7: *DAG initialization without "with" block*

   If DAG is initialized separately, then it needs to pass to all the tasks explicitly.

5. It is time to define the tasks. To keep things simple for now, our task is to just execute a bash command. You already know now that to execute a bash command, a bash operator will be required. Hence, we have two instances of `BashOperator` that execute different tasks. The instantiated **`Operator`** object is called a task.

   To execute and track the task, there are certain details that needs to be provided to the task which will vary from task to task. For example, **`HTTPOperator`** would require a URL as well. In our case, we are specifying two parameters to our task initialization:

   a. **`task_id`** – The unique ID of the task or the task identifier. No two tasks in a DAG can have the same **`task_id`**.

   b. **`bash_command`** – Since this is a **`bashOperator`**, we need to specify the bash command.

6. Finally, we specify the task execution sequence. "*Binary right shift operator*" are used to define the dependencies between tasks. The way to specify the

dependencies is quite unique. The following are a few examples and we touched part of it earlier as well:

a. T1 >> T2: Execute T1 and then T2

b. T1 >> T2 >> T3: This is self-explanatory. Three tasks are executed in sequence.

c. [T1, T2, T3] >> T4: T1, T2 and T3 are independent but T4 is dependent on the preceding list.

d. Start >> [T1, T2,T3] >> [T4,T5,T6] >> Stop: Start and then execute T1, T2, T3 independently. Once complete, T4, T5, T6 are executed independently and then finally, the "stop" task is executed. It is possible for DAG to have branches but that is managed via operators. We will discuss more about it later.

   It is important to remember that while the syntax is in Python, it is not a Python file. Try running the DAG file with Python and you will get errors. The DAG file is parsed by Airflow and the DAG class in initialized.

Along with the code, it helps to understand the execution model of Airflow. Tasks are started either via the push or pull model. In the push model, the central model pushes the task to the workers for execution. Whereas, in the pull model, the workers pull the central process for work. While both have benefits and challenges, Airflow uses the push model and scheduler is the central process.

4.6 Understanding scheduling

We have already seen that Airflow can be scheduled via the **`schedule_interval`** and **`start_date`** parameters. In this section, we will go deeper to understand the different aspects of scheduling to meet real world scheduling needs.

Let us start imagining a DAG that downloads sales data from different regions and then processes them. We will simulate the activities to focus on the concepts. Code 4.8 shows the DAG for it:

```
import airflow.utils.dates

from airflow import DAG

from airflow.operators.bash_operator import BashOperator

from datetime import datetime, timedelta


default_args = {
        "owner": "airflow",
        "start_date": airflow.utils.dates.days_ago(2)
    }

with DAG(dag_id="mySchedule_dag", default_args=default_args,
schedule_interval="@daily") as dag: --1

    download_sales_data = BashOperator(
            task_id="download_sales_data",
            bash_command="echo Download Data"
        )

    process_sale_data = BashOperator(
            task_id="process_sales_data",
            bash_command="echo process_data"
        )

    download_sales_data >> process_sale_data
```

Code 4.8: *Daily scheduled DAG*

4.6.1 Understanding scheduling preset

The **`@daily`** **`schedule_interval`** parameter that you saw earlier is quite self-explanatory. *Table 4.1* shows some of the other common and popular cron *presets*. Think of presets as macros:

Preset	Meaning
`@once`	Schedule once and only once
`@hourly`	Run hourly or the beginning of every hour
`@daily`	Run once a day at midnight

`@weekly`	Run once a week at midnight or Sunday morning
`@monthly`	Run once a month at midnight of the first day of the month
`@yearly`	Run once a year at midnight of January 1

***Table 4.1:** Airflow presets*

It is not necessary for DAGs to have schedules. Hence, you can specify **`schedule_interval = None`** and that means no scheduling for the DAG. The schedule helps inform Airflow on how to create the DAG runs. The DAG will be instantiated for each schedule, while creating a DAG run for each of the schedules.

It is important to understand the construct of a DAG run. DAG runs have a state associated to them (running, failed, or successful) and inform the scheduler about which set of schedules should be evaluated for task submissions. You may wonder why the metadata at the DAG run level is required. Well, without the metadata at the DAG run level, the Airflow scheduler would have much more work to do in order to figure out what tasks should be triggered and come to a crawl.

4.6.2 Understanding cron scheduling

Cron scheduling is another way to schedule the DAG. It is helpful for situations that can't be handled via preset. For example, how would you schedule the DAG to run at 10.30 am? If you know about cron on Unix, then the syntax will be quite familiar to you.

The cron schedule consists of five components which are as follows:

```
* * * * * command(s)
- - - - -
| | | | |
| | | | ----- Day of week (0 - 7) (Sunday=0 or 7)
| | | ------- Month (1 - 12)
| | --------- Day of month (1 - 31)
| ----------- Hour (0 - 23)
------------- Minute (0 - 59)
```

It may feel convoluted if you are seeing it for the first time, but it is quite flexible. Let us understand the syntax. Note that the five fields may contain one or more values, separated by a comma or a range of values separated by a hyphen:

- *: The asterisk operator means any value or always. If you have the asterisk symbol in the Hour field, it means the task will be performed each hour.
- ,: The comma operator allows you to specify a list of values for repetition. For example, if you have 1,3,5 in the hour field, the task will run at 1 am, 3 am, and 5 am.

- -: The hyphen operator allows you to specify a range of values. If you have 1-5 in the day of week field, the task will run every weekday (from Monday to Friday).
- /: The slash operator allows you to specify values that will be repeated over a certain interval between them. For example, if you have */4 in the hour field, it means the action will be performed every four hours. It is the same as specifying 0,4,8,12,16,20. Instead of asterisk before the slash operator, you can also use a range of values. 1-30/10 means the same as 1,11,21.

We can simulate some of the preset via the cron syntax as follows:

Preset		Meaning
`@once`		Schedule once and only once
`@hourly`	0 * * * *	Run hourly or at the beginning of every hour
`@daily`	0 0 * * *	Run once a day at midnight
`@weekly`	0 0 * * 0	Run once a week at midnight or Sunday morning
`@monthly`	0 0 1 * *	Run once a month at midnight of the first day of the month
`@yearly`	0 0 1 1 *	Run once a year at midnight of January 1

Table 4.2: *cron syntax*

Let us take a look at some more examples:

- 0 0 1 * * = midnight on the first of every month
- 45 23 * * SAT = 23:45 every Saturday
- 0 0 * * MON,WED,FRI = run every Monday, Wednesday, and Friday at midnight
- 0 0 * * MON-FRI = run every weekday at midnight
- 0 0,12 * * * = run every day at 00:00 AM and 12:00 P.M

4.6.3 Understanding frequency based scheduling

One of the limitations of cron based scheduling is to express frequency-based scheduling. For example, how you would specify a task(s) to run every alternate day, or it could be a little more sophisticated like running thrice a month on the 1st, 15th, and 30th. The challenge here is to define scheduling intervals in terms of a relative time interval. In order words, it is not stateless, but we would like to remember when the previous job was run. To specify frequency-based schedule, Airflow offers the timedelta function:

```
from datetime import timedelta

dag = DAG(

    dag_id="sales_report",

    schedule_interval=timedelta(days=2)

    start_date=dt.datetime(year=2020, month=1, day=1),

)
```

Code 4.9: *DAG with scheduling done with timedelta function*

The above DAG would run after every two 2 days starting from 1 January 2020. In the example, besides days as the parameter for timedelta function, you can also use minutes or hours as shown as follows:

- **`timedelta(minutes = 20):`** Run every 20 min.
- **`timedelta(hours = 20):`** Run every 20 hours.

Now that you understand the different options for setting up the interval, let us understand the other piece – When should it start from and if there is a start, how do you specify the end data? You have already seen the start date parameter multiple times. It's no surprise that Airflow offers an end parameter as well. `Code 4.9` shows a DAG end date:

```
from datetime import timedelta


dag = DAG(

    dag_id="sales_report",

    schedule_interval=timedelta(days=2)

    start_date=dt.datetime(year=2020, month=1, day=1),

    end_date=dt.datetime(year=2020, month=12, day=1),

)
```

Code 4.10: *DAG with start and end date*

There is a certain quirkiness about how interval and start date are treated in Airflow. Airflow schedules the first execution of a DAG to run at the first schedule interval after the start date (start + interval). The subsequent runs will continue executing at schedule intervals following this first interval.

Let us draw the scheduled timeline for *code 4.10* as shows:

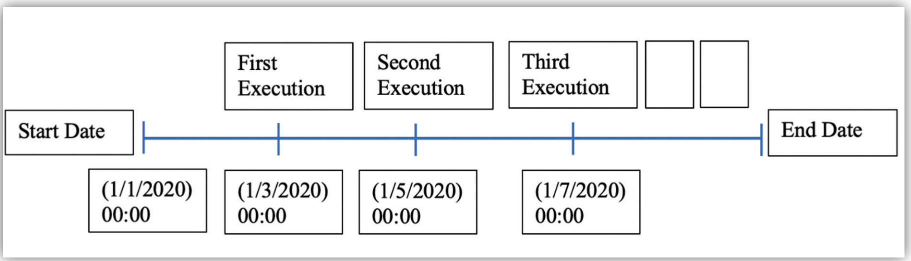


Figure 4.30: *DAG scheduling logic*

Note that the first execution is not on 1st January but on 3rd January (start date + interval).

4.6.3 Understanding dynamic time

There would be debugging or auditing requirements where you would like to know the interval in which a task is running. To cater to such requirements, Airflow provides extra parameters to tasks. **`execution_date`** is one such parameter. **`execution_date`** represents the date and time for which our DAG is being executed. It is important to note and remember that **`execution_date`** is not a date but a timestamp, which reflects the start time of the schedule interval for which the DAG is being executed. The end time of the schedule interval is made available by another parameter called the **`next_execution_date`**. Using these two dates, you could calculate the length of a task's schedule interval. We are already having a torrent of date related parameters and it is important to understand them. The **`execution_date`** and **`start_date`** are a little confusing, so following frame would help you remember the difference:

- **`execution_da`**te is the start date and time when you expect your DAG to be triggered.
- **`start_date`** – is the date and time when it is triggered.

Airflow sets **`execution_date`** based on the left bound of the schedule period that it is covering, and not based on when it fires (which would be the right bound of the period). *Figure 4.30* illustrates the idea. The first run would trigger after midnight on 3rd January.

This left-bound labelling is a common concept in ETLs but gets a little confusing in cron-like scheduling context. *Table 4.3* shows some of the other popular date-related variables. You can check the full list here: https://airflow.apache.org/docs/apache-airflow/stable/macros-ref.html.

Variable	Description
`{{ds}}`	The execution date as YYYY-MM-DD
`{{ ds_nodash }}`	The execution date as YYYYMMDD
`{{ prev_ds }}`	The previous execution date as YYYY-MM-DD if **`{{ ds }}`** is 2018-01-08 and **`schedule_interval`** is **`@weekly`**, **`{{ prev_ds }}`** will be 2018-01-01
`{{ next_ds }}`	The next execution date as YYYY-MM-DD if **`{{ ds }}`** is 2018-01-01 and schedule_interval is **`@weekly`**, **`{{ next_ds }}`** will be 2018-01-08
`{{ yesterday_ds }`	The day before **`execution_date`** as YYYY-MM_DD
`{{ tomorrow_ds }}`	The day after **`execution_date`** as YYYY-MM_DD
`{{ prev_execution_date }}`	The previous execution date (if available)
`{{prev_start_date_success}}`	Start date from prior successful DAG run (if available)
`{{ next_execution_date }}`	The next execution date

Table 4.3: *Popular date-related variables.*

If you are a visual thinker, then *figure 4.31* would help you understand the different date variables:

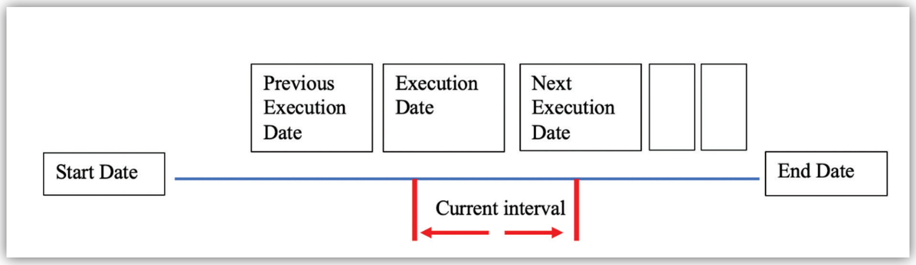


Figure 4.31: *Airflow provided dates*

Are you wondering what those curly braces are? {{**`variable_name`**}} is an example of using Airflow's Jinja-based templating syntax for referencing Airflow's specific parameters. Think of these as Airflow provided variables that get replaced with actual values at run time.

The following examples would be helpful here:

```
import airflow.utils.dates

from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta


default_args = {
        "owner": "airflow",
        "start_date": airflow.utils.dates.days_ago(2)
    }


with DAG(dag_id="mySchedule_dag", default_args=default_args,
schedule_interval="@daily") as dag:

    download_sales_data = BashOperator(
            task_id="download_sales_data",
            bash_command="echo Download Data for {{execution_date}}" --1
        )


    process_sale_data = BashOperator(
            task_id="process_sales_data ",
          bash_command="echo process_data for {{next_execution_date}}" --2
        )
    download_sales_data >> process_sale_data
```

Code 4.11: *DAG with date macros*

Let us unpack the highlighted points as follows:

1. In the **`bash_command`** we have used the macro **`{{execution_date}}`**. This is jinja templating and during processing it gets replaced with actual value. For me, this was the log entry:

   `Running command: echo Download Data 2020-12-21T12:28:32.845399+00:00`

2. Same as before but a different macro **`{{next_execution_date}}`** was used. For me, this was the log entry:

   ```
   Running command: echo process_data for 2020-12-
   21T12:28:32.845399+00:00
   ```

Because the **`execution_date`** parameters are often used in this fashion to reference dates as formatted strings, Airflow also provides several shorthand parameters for common date formats. For example, **`ds`** and **`ds_nodash`** parameters are different representations of the **`execution_date`**, formatted as YYYY-MM-DD and YYYYMMDD respectively. Similarly, **`next_ds`**, **`next_ds_nodash`**, **`prev_ds`** and **`prev_ds_nodash`** provide shorthand for the next and previous execution dates, respectively.

It is important to remember that not all parameters in operators are templated, and so, Jinja templates cannot be used everywhere by default.

Also, Jinja templates can be affected by other parts of your DAG. For example, if the DAG is scheduled to run **`@once`**, **`next_execution_date`**, and **`previous_execution_date`** macros will be None since the DAG is defined to run just once.

To find out which parameters support templates, the documentation is the place to look at. For example, in **`BashOperator`**, only **`bash_command`** can be templated. There would be a section for templating. The following is the template section entry for **`BashOperator`**.

You can use Jinja templates to parameterize the **`bash_command`** argument.

4.6.4 Understanding backfilling and catch up

You now know that scheduling involves the interval, start, and optional end date. The start date could be arbitrary and could go in the past as well. This means that we can have a past interval starting from past dates to process past data. This process is called backfilling.

Airflow will schedule and run any past schedule intervals that have not yet been run by default. However, this can be controlled by the **`catchup`** parameter. If the **`catchup`** is set to False, past intervals will not be executed and only the current interval will be executed. Catch up parameter could be specified in the **`airflow.cfg`** (configuration file) file:

```
catchup_by_default = True
```

Or while initializing the DAG class.

```
with DAG(dag_id="myPool_dag", default_args=default_args, schedule_
interval="@daily", catchup = False) as dag:
```

Backfilling is a powerful concept. Imagine a situation where you change the processing logic (task) and now you want to run it for the past data as well. However, for the past processing past data needs to be available. Generally, during development, I keep it off to save on the execution time.

4.7 Airflow operators

In a DAG, we define the different tasks or the complete workflow. It is the operators that get the task done. An operator map to a single task in a workflow. As a good practice, it helps to have operators atomic in behavior, that is, they are self-contained and can stand on their own and don't need to share resources with any other operators.

Airflow provides operators for many common tasks, including the following:

- **`BashOperator`** - Executes a bash command
- **`PythonOperator`** - Calls an arbitrary Python function
- **`EmailOperator`** - Sends an email
- **`SimpleHttpOperator`** - Sends an HTTP request
- **`MySqlOperator`, `SqliteOperator`, `PostgresOperator`, `MsSqlOperator`, `OracleOperator`, `JdbcOperator`**, etc. - Executes an SQL command
- **`Sensor`** - Waits for a certain time, file, database row, S3 key, etc.

This list does not end here. In addition to these basic building blocks, there are many more specific operators: **`b`, `HiveOperator`, `S3FileTransformOperator`, `PrestoToMysqlOperator`, `SlackOperator`**. Beyond the core operators there are lots of community provided operators as well. The **`airflow/contrib/`** directory contains community-built operators. Note that operators are only loaded if they are used in a DAG.

Again, if you are a visual thinker, then the nodes in the DAG are the operator. All operators derive from **`BaseOperator`** and inherit many attributes and methods that way. You can broadly classify the operators into the three following types:

- Operators that perform an action or tell another system to perform an action.
- Transfer operators move data from one system to another.
- Sensors will keep on running until a certain condition is met. Think of sensors as event driven operators. Examples could be, let us say a specific file landing in HDFS or S3, a partition appearing in Hive, or a specific time

of the day. Sensors are derived from **BaseSensorOperat**or and run a poke method at a specified **poke_interval** until it returns **True**.

You have already seen BashOperator in action. Let us explore few more.

4.7.1 Python operator

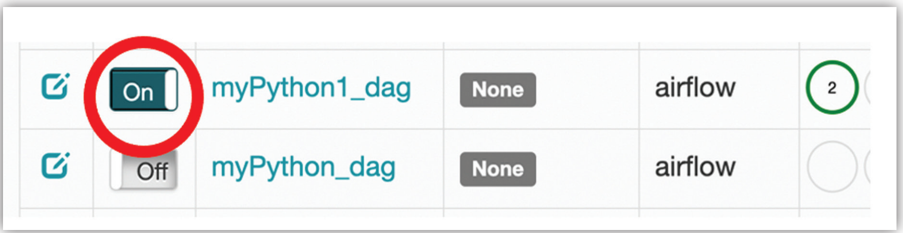
PythonOperator helps to execute the Python code or Python callable. The following example would be the best way to understand it:

```
--1

import airflow.utils.dates

from airflow import DAG


from datetime import datetime, timedelta

from airflow.operators.python_operator import PythonOperator

from airflow.operators.bash_operator import BashOperator


def _download_sales_data(): --2

    print("Downloading the sales data")


default_args = {

        "owner": "airflow",

        "start_date": airflow.utils.dates.days_ago(2)

    }


with DAG(dag_id="myPython1_dag", default_args=default_args,

schedule_interval=None) as dag:


    download_sales_data = PythonOperator( --3

            task_id="download_sales_data",
            python_callable=_download_sales_data
        )


    process_sales_data = BashOperator(
            task_id="process_sales_data",
            bash_command="echo process_data"
        )


    download_sales_data >> process_sales_data
```

Code 4.12: *DAG with python operator*

Let us unpack the code as follows:

1. All the imports you have already seen before except the Python operator. **`Airflow.operators`** contains most of the operators.
2. We define a Python callable function. At the moment, it is very basic and not accepting any parameters. Notice the preceding _ that indicates this function should not be called directly. Note that it is a convention and not a rule.
3. Next, we instantiate the **`pythonOperator`**. While **`task_id`** is something that you have seen before, **`python_callable`** is a new parameter. It is reference to an object that is callable, and in our case, it is a function. Different operators will have different parameters and the best source to know about it is to check the documentation.
4. There is one more thing about the DAG. The **`schedule_interval`** is set **`None`** and that means it needs to be triggered manually. Do remember that before triggering a DAG, it must be in an ON state.



	On	myPython1_dag	None	airflow	2
	Off	myPython_dag	None	airflow	

Figure 4.32: *DAG State*

Just as a reminder, to view the output of your print statement, you need to go to the **view log** tab. To reach the **view log** page, click on the DAG, go to the **graph view**, click on the task in the task pane, and in the pop up, click on the **view log** (check *figure 4.8*) tab.

Let us now **`pass`** parameters to the Python function:

```
import airflow.utils.dates

from airflow import DAG


from datetime import datetime, timedelta

from airflow.operators.python_operator import PythonOperator

from airflow.operators.bash_operator import BashOperator


def _download_sales_data():

    print("Downloading the sales data")


def _download_region_sales_data(region): --1

    print(f"Downloading the sales data for {region}")


default_args = {

        "owner": "airflow",

        "start_date": airflow.utils.dates.days_ago(2)

    }


with DAG(dag_id="myPython1_dag", default_args=default_args,

schedule_interval=None) as dag:


    download_sales_data = PythonOperator(

            task_id="download_sales_data",

            python_callable=_download_sales_data
        )


    download_region_sales_data = PythonOperator(

            task_id="download_region_sales_data",

            python_callable=_download_region_sales_data,

            op_kwargs={'region': 'east'},  --2


        )


    download_sales_data >> download_region_sales_data >> process_sales_data

    process_sales_data = BashOperator(

            task_id="process_sales_data",

            bash_command="echo process_data"

        )

    download_sales_data >> download_region_sales_data >> process_sales_data
```

Code 4.13: *Python callable with parameters*

Let us unpack the new things as follows:

- **`_download_region_sales_data(`**) function accepts a new parameter named "**`region`**."
- We are passing the function parameter value via the **`op_kwargs`** parameter. In fact, you pass values via **`op_args`** and **`op_kwargs`** parameters.

Airflow passes in an additional set of keyword arguments: one for each of the Jinja template variables (remember the {**`{variables}`**}) and a **`templates_dict`** argument. Let us see that in action in code 4.14:

```
import airflow.utils.dates

from airflow import DAG


from datetime import datetime, timedelta

from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator


def _download_region_sales_data(**kwargs):  --1


    region = kwargs.get('templates_dict', None).get('region', None) --2

   sales_date = kwargs.get('templates_dict', None).get('region_executing_
date', None) --3


    print(f"Downloading the {region }sales data for {sales_date}") --4


default_args = {

        "owner": "airflow",

        "start_date": airflow.utils.dates.days_ago(2)

    }


with DAG(dag_id="myPython_templ_dict_dag", default_args=default_args,

schedule_interval=None) as dag:


    download_region_sales_data = PythonOperator(

            task_id="download_region_sales_data",

            python_callable=_download_region_sales_data,

            templates_dict={. --5

          "region": "EAST",

                    "region_executing_date": "{{ds}}"

          },

         Provide_context = True. --6
    process_sales_data = BashOperator(

            task_id="process_sales_data",

            bash_command="echo process_data"

        )

download_region_sales_data >> process_sales_data
```

Code 4.14: *Parameter passing with templates_dict*

Let us unpack the code as follows:

1. We define the function **`_download_region_sales_data`** that receives all the context variables in this dictionary.
2. We extract the **`templates_dict`** and the **`region`** key value. It returns **`None`** if the key is absent. The keys are hypothetical; the important part is to learn the retrieval.
3. We extract the **`templates_dict`** and the **`region_executing_date`** key value. It returns None if the key is absent. The keys are hypothetical; the important part is to learn the retrieval.
4. We print the dynamically prepared statement. You would be able to view this statement in the view logs tab.
5. While defining the Python operator `download_region_sales_data`, we build the `templates_dict`. It is a JSON and that explains the retrieval process at step 2. The `region_executing_date` key is new and interesting here. We are populating its value via the default variables.

   Enabling this kind of syntax makes it quite useful and flexible. Imagine you are processing day-wise files. To pass day-wise files for processing, all you need is to make the input file name dynamic. The following code demonstrates the idea.
6. In order to pass the context, `provide_context` needs to be set to True. When `provide_context` argument is set to True, Airflow passes in an additional set of keyword arguments: one for each of the Jinja template variables and a `templates_dict` argument. Recall that `templates_dict` argument is templated and each value in the dictionary is evaluated as a Jinja template. We will come back to Jinja template in detail later. For now, just imagine it to be like a placeholder that would get replaced at runtime.

4.7.2 Email operator

Email operator, as the name suggests, is for sending out mails. This would be a commonly used operator as notifications are an integral part of workflows. We will keep our use case restricted to notifications on retires, failures, etc. Let us go through the steps as follows:

1. The first step is to configure the mail server. We will stick to SMTP (simple mail transfer protocol). Go to the [smtp] section in `airflow.cfg`. Update the following entries:

   ```
   smtp_host = smtp.gmail.com

   smtp_starttls = True

   smtp_ssl = False

   smtp_user = YOUR_EMAIL_ADDRESS

   smtp_password = 16_DIGIT_APP_PASSWORD

   smtp_port = 587

   smtp_mail_from = YOUR_EMAIL_ADDRESS
   ```

   It is also possible to update the values via environment variables. We are yet to discuss the environment variables in the Airflow context, so for now accept that Airflow is able to read the following environment variables automatically:

   ```
   AIRFLOW__SMTP__SMTP_HOST=smtp.gmail.com

   AIRFLOW__SMTP__SMTP_PORT=587

   AIRFLOW__SMTP__SMTP_USER=your-mail-id@gmail.com

   AIRFLOW__SMTP__SMTP_PASSWORD=yourpassword

   AIRFLOW__SMTP__SMTP_MAIL_FROM=your-mail-id@gmail.com
   ```

   The password has to be generated via Google for Google smtp server (https://security.google.com/settings/security/apppasswords).

   To configure the email that will be sent out, update the [email] section in the `airflow.config` file:

   ```
   [email]

   email_backend = airflow.utils.email.send_email_smtp
   ```

```
subject_template = /path/to/my_subject_template_file

html_content_template = /path/to/my_html_content_template_file
```

Since the email content is template based, you can use Jinja templating. Here is how a sample template may look like. Notice the templated variables being used as follows:

```
Try {{try_number}} out of {{max_tries + 1}}<br>

Exception:<br>{{exception_html}}<br>

Log: <a href="{{ti.log_url}}">Link</a><br>

Host: {{ti.hostname}}<br>

Log file: {{ti.log_filepath}}<br>

Mark success: <a href="{{ti.mark_success_url}}">Link</a><br>
```

2. Next is to build the operator. Code 4.15 shows a DAG with email-operator:

```
from airflow import DAG
from airflow.operators.email_operator import EmailOperator --1
from airflow.operators.python_operator import PythonOperator
from datetime import datetime, timedelta


default_args = {
    "owner": "airflow",
    "start_date": datetime(2019, 11, 1)
}


def _error_function(): --2
    raise Exception('Alert!Alert!')
with DAG(dag_id="myEmail_dag", default_args=default_args,
    schedule_interval=None) as dag:
        send_completion_mail = EmailOperator( --3
            task_id="send_completion_mail",
            to='receivers@mail.com',
            subject='Task Completion',
            html_content='<p> Your Job was completed <p>')
        Dummy_task = PythonOperator(  --4
            task_id='Dummy_task',
            python_callable=_error_function,
            email_on_failure=True,
            email='receivers@mail.com')
Dummy_task >> send_completion_mail
```

Code 4.15: *DAG with email operator*

Let us unpack the code as follows:

1. We need to import EmailOperator first. The parent module is something that you have already seen before.
2. We define a Python callable function. This function raises an exception so that we can simulate the condition for email notification to fire off.
3. We define the email operator. All the fields are self-explanatory. Remember templated fields? Well, the following are the template fields in EmailOperator: `template_fields= ['to', 'subject', 'html_content', 'files']`
4. `Dummy_task` is a Python operator; the callable raises an exception and then the operator sends out an email due to failure. The email on failure behavior is due to `email_on_failure` flag set to True.

To be explicit, email is being sent out in two scenarios in the following DAG:

1. When the task completes. Demonstrated with the shift operator. Mail is sent out when the `dummy_task` is complete.
2. When the task fails, configured via the `email_on_failure` attribute.

   `Dummy_task >> send_completion_mail.`

4.7.3 Setting up notifications at DAG

You can set the notifications at the DAG level, and that means they will be applicable/ available to all the tasks. This can be done by setting up the email in the `default_args`:

```
from airflow import DAG

from datetime import datetime, timedelta

default_args = {

    "owner": "airflow",
    "start_date": datetime(2019, 11, 1),
    "email": [<<emails array>>]  --1
}
def _error_function():
    raise Exception('Alert!Alert!')


with DAG(dag_id="myEmail_dag", default_args=default_args,
    schedule_interval=None) as dag:

        Dummy_first_task = PythonOperator(
            task_id='Dummy_task',
            python_callable=_error_function,


      )

        Dummy_second_task = PythonOperator(
            task_id='Dummy_task',
            python_callable=_error_function,


       )
Dummy_first_task >> Dummy_second_task
```

Code 4.16: *Notification at the DAG level.*

If any of the tasks fail in code 4.16, email will be sent out to the email list (highlight #1). Note that none of the tasks have explicitly set a failure notification.

4.7.3.1 Setting up notification at DAG task

For some of the use cases, notification may not be required for all the tasks. To be specific, we want email alerts for specific task(s). It is straightforward actually. You don't set it at the DAG level but use it in the tasks where it is required. Code 4.17 demonstrate the idea as follows:

```
from airflow import DAG

from datetime import datetime, timedelta


default_args = {

    "owner": "airflow",

    "start_date": datetime(2019, 11, 1),

}


with DAG(dag_id="myEmail_dag", default_args=default_args,

    schedule_interval=None) as dag:


    Dummy_first_task = PythonOperator(

            task_id='Dummy_task',

            python_callable=_error_function,

      )

        Dummy_second_task = PythonOperator(

            task_id='Dummy_task',

            python_callable=_error_function,

          email_on_failure=True --1

       )

Dummy_first_task >> Dummy_second_task
```

Code 4.17: *Isolated error notification task*

The alert setup has been removed from DAG default parameters and set at the task level (code 4.17, #1).

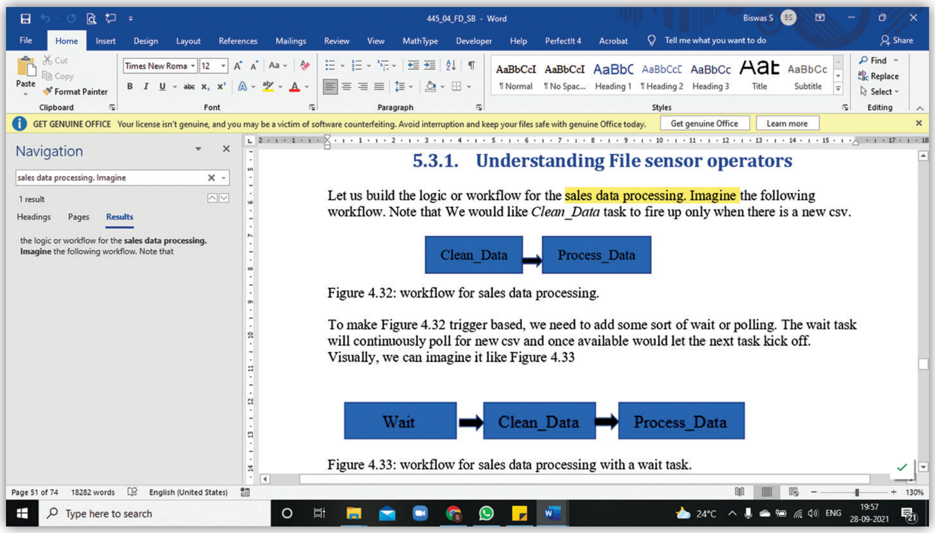
If you are looking for traceable alerts, then consider Slack. Check out the airflow.providers.slack.operators in the documentation.

4.7.4 Understanding sensor operators

Till now, we have seen how tasks can be scheduled using different mechanisms. However, there are certain tasks that don't have fixed schedule. Think of tasks that are event dependent for example,; a sales reporting task should only be executed when the sales data dump is available. We will now take a look at a few operators that are trigger based.

4.7.4.1 Understanding file sensor operators

Let us build the logic or workflow for the sales data processing. Imagine the following workflow. Note that we would like `Clean_Data` task to fire up only when there is a new CSV:


445_04_FD_SB - Word
File
Home
Insert
Design
Layout
References
Mailings
Review
View
MathType
Developer
Help
Perfectlt 4
Acrobat
Tell me what you want to do
Share
GET GENUINE OFFICE Your license isn't genuine, and you may be a victim of software counterfeiting. Avoid interruption and keep your files safe with genuine Office today.
Get genuine Office
Learn more
Navigation
sales data processing. Imagine
1 result
Headings
Pages
Results
the logic or workflow for the sales data processing. Imagine the following workflow. Note that
Page 51 of 74
18282 words
English (United States)
Type here to search
24°C
ENG
19:57
28-09-2021

5.3.1. Understanding File sensor operators

Let us build the logic or workflow for the sales data processing. Imagine the following workflow. Note that We would like *Clean_Data* task to fire up only when there is a new csv.

Figure 4.32: workflow for sales data processing.

To make Figure 4.32 trigger based, we need to add some sort of wait or polling. The wait task will continuously poll for new csv and once available would let the next task kick off. Visually, we can imagine it like Figure 4.33

Figure 4.33: workflow for sales data processing with a wait task.

Figure 4.33: *Workflow for sales data processing*

To make *figure 4.33* trigger based, we need to add some sort of wait or polling. The wait task will continuously poll for new CSV, and once available, it would let the next task kick off. Visually, we can imagine it as shown in *figure 4.33*:

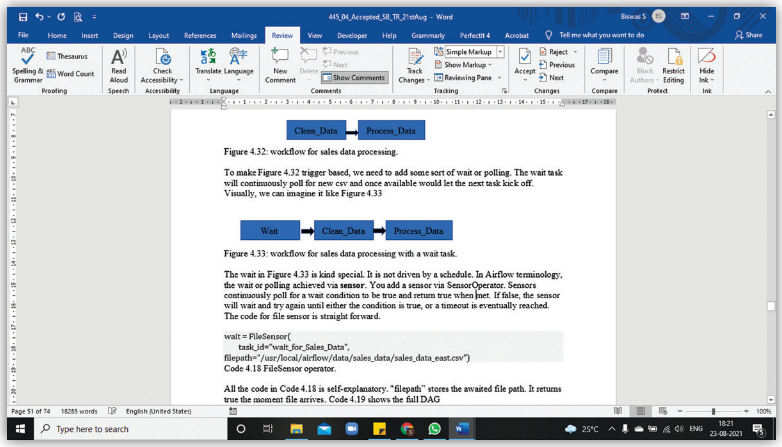


Figure 4.32: workflow for sales data processing.

To make Figure 4.32 trigger based, we need to add some sort of wait or polling. The wait task will continuously poll for new csv and once available would let the next task kick off. Visually, we can imagine it like Figure 4.33

Figure 4.33: workflow for sales data processing with a wait task.

The wait in Figure 4.33 is kind special. It is not driven by a schedule. In Airflow terminology, the wait or polling achieved via **sensor**. You add a sensor via SensorOperator. Sensors continuously poll for a wait condition to be true and return true when met. If false, the sensor will wait and try again until either the condition is true, or a timeout is eventually reached. The code for file sensor is straight forward.

```
wait = FileSensor(
    task_id="wait_for_Sales_Data",
filepath="/usr/local/airflow/data/sales_data/sales_data_east.csv")
```

Code 4.18 FileSensor operator.

All the code in Code 4.18 is self-explanatory. "filepath" stores the awaited file path. It returns true the moment file arrives. Code 4.19 shows the full DAG

***Figure 4.34:** Workflow for sales data processing with a wait task.*

The wait in *figure 4.34* is kind of special. It is not driven by a schedule. In Airflow terminology, the wait or polling achieved via sensor. You add a sensor via SensorOperator. Sensors continuously poll for a wait condition to be true and return true when met. If false, the sensor will wait and try again until either the condition is true or a timeout is eventually reached.

The code for file sensor is straightforward:

```
wait = FileSensor(

       task_id="wait_for_Sales_Data",             filepath="/usr/local/
       airflow/data/sales_data/sales_data_east.csv")
```

***Code 4.18:** FileSensor operator*

All the code in *code 4.18* is self-explanatory. The filepath stores the awaited file path. It returns true the moment the file arrives. Code 4.19 shows the full DAG:

```
import airflow.utils.dates

from airflow import DAG

from airflow.contrib.sensors.file_sensor import FileSensor --1

from airflow.operators.dummy_operator import DummyOperator --2


default_args = {

        "owner": "airflow",
        "start_date": airflow.utils.dates.days_ago(1)
    }


with DAG(dag_id="myFile_Sensor_dag", default_args=default_args,
schedule_interval="@daily", catchup = False) as dag:


    wait_for_Sales_Data = FileSensor( --3
        task_id="wait_for_Sales_Data",
        filepath="/usr/local/airflow/data/sales_data/sales_data_east.csv"
        )
    clean_data = DummyOperator(task_id="clean_data") --4
    process_data = DummyOperator(task_id="process_data")


    wait_for_Sales_Data >> clean_data >> process_data
```

Code 4.19: *DAG with file sensor*

Let us unpack the code as follows:

1. We import the FileSensor.
2. We import a new operator type called DummyOperator. It literally does nothing. The task is evaluated by the scheduler but never processed by the executor. It has some interesting use case for to group tasks. Here, we are using it because we don't care about what happens next once the file sensors returns true.
3. The file sensor is initialized. It inherits from BaseSensorOperator. As described earlier, its purpose is to wait for a file or folder to land in the filesystem. If the path given is a directory, then this sensor will only return true if any files exist inside it (either directly or within a subdirectory).
4. We then instantiate two dummy tasks. Other than the task-id, there is no other attribute because they have nothing to do.
5. When you trigger the DAG. It will wait for the wait task to complete.

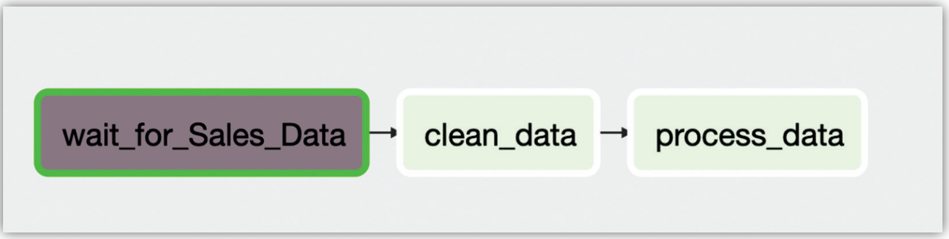


Figure 4.35: *Triggered DAG with fileSensor waiting for a file.*

6. To complete, you need to place the `sales_data_east.csv` file under the /usr/local/airflow/data/sales_data/ folder.
7. If you are using the local executor Docker compose file, then run the docker ps command to find the container ID of the Airflow server. Use `docker exec -it <container_Id> /bin/bash` to connect to the Docker. Once inside, you need to create the respective folders and the file as follows:

   ```
   cd ~

   mkdir data data/sales_data

   cat hello > data/sales_data/ sales_data_east.csv
   ```

If the fileSensor is able to sense the presence of the file, you will find some similar-looking entry in the log (View log tab):

```
INFO - Success criteria met. Exiting.

{taskinstance.py:1048} INFO - Marking task as SUCCESS.dag_id=myFile_
Sensor_dag, task_id=wait_for_Sales_Data,
```

Inside the log, you would also find poking statements.

```
{file_sensor.py:60} INFO - Poking for file /usr/local/airflow/data/sales_
data/sales_data_east.csv
```

Here, the sensor is poking for the file availability. It is a name given by Airflow for the polling operation. So, how frequently it should poke? Well, it is configurable and the default value is 1 minute.

4.7.4.2 Understanding custom pooling

The simplicity of FileSensor comes with the limitation of adding custom logic for event triggering. Everything has a trade-off. If you need custom logic in your poking, then you can use `PythonSensor`. The PythonSensor is like the PythonOperator in its structure and operation. You supply a Python callable (function/method/etc.)

to execute. However, the callable is limited to returning a Boolean value. True is returned to indicate that the condition is met successfully; False to indicate it is not. Let us re-write code 4.19 with PythonSensor:

```
import os

import airflow.utils.dates

from airflow import DAG

from airflow.contrib.sensors.python_sensor import PythonSensor

from airflow.operators.dummy_operator import DummyOperator


def _wait_for_file(region):  --1

     file_path = "/usr/local/airflow/data/sales_data/sales_data_" + region
+ ".csv"

    isFile = os.path.isfile(file_path)

    return isFile


default_args = {

        "owner": "airflow",

        "start_date": airflow.utils.dates.days_ago(1)

    }


with DAG(dag_id="myPython_Sensor_dag", default_args=default_args,

schedule_interval="@daily", catchup = False) as dag:


    wait_for_Sales_Data = PythonSensor(  --2

        task_id="wait_for_Sales_Data",

        python_callable=_wait_for_file,

        op_kwargs={'region': 'east'},

        )
```

```
clean_data = DummyOperator(task_id="clean_data")

process_data = DummyOperator(task_id="process_data")


wait_for_Sales_Data >> clean_data >> process_data
```

Code 4.20: *DAG with Python sensor*

Let us unpack the code as follows. With your acquired understanding of PythonOperator, this would be a cake walk.

1. We define a Python callable that checks for a file existence and returns True or False. It takes a hypothetical parameter; I did it to show the similarity with PythonOperator.
2. We instantiate the PythonSensor. All the parameters are same as PythonOperator.
3. The rest of the tasks are dummy tasks described earlier in *code 4.19*.
4. When you run the DAG, the execution will show a similar pattern that you have seen earlier:

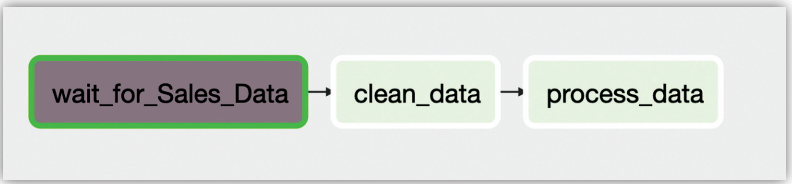


Figure 4.36: *DAG with Python sensor*

5. To complete, you need to place the `sales_data_east.csv` file under the `/usr/local/airflow/data/sales_data/` folder.
6. If you are using the local executor Docker compose file, then run the ps command to find the container ID of the airflow server. Use `docker exec -it <container_Id> /bin/bash` to connect to the Docker. Once inside, you need to create the respective folders and the file:

```
cd ~

mkdir data data/sales_data

cat hello > data/sales_data/ sales_data_east.csv
```

If the **`PythonSensor`** is able to sense the presence of the file, you will find some similar-looking entry in the log (View log tab)

```
{base_sensor_operator.py:123} INFO - Success criteria met. Exiting.

{taskinstance.py:1048} INFO - Marking task as SUCCESS.dag_id=myPython_
Sensor_dag, task_id=wait_for_Sales_Data,
```

Til now, we have seen the happy path, but you know that the real world is never so ideal. Imagine that the system providing the region-wise sales data went down leading to the task continuously polling blocking the execution.

The sensor's timeout argument stores the maximum number of seconds that a sensor is allowed to run. If, the number of running seconds gets higher than the number set to timeout, at the start of the next poke, the sensor will fail. The default value for sensor timeout is set to 7 days. Since the DAG is scheduled to run daily, it may lead to a problematic situation called sensor deadlock if the task continuously fails. The deadlock would mean that Airflow would not be able to add new tasks. Here is a quick sequence of an event leading up to a deadlock. While the existing runs are failing, new DAG runs are added every day, the sensors for those respective days are started, and as a result, more and more tasks start running. Since there's a limit to the number of tasks, Airflow can handle and run (on various levels), the entire Airflow system will be stalled.

There are two solutions to avoid and handle deadlocks. These are as follows:

1. By setting up the mode parameter: Mode parameter controls how the sensor operates. It accepts two options: poke (default) or reschedule. When set to poke, the sensor takes up a worker slot for its whole execution time and sleeps between pokes. It is the poke mode that causes the deadlock. It doesn't mean that poke mode is always a bad choice. Use this mode if the expected runtime of the sensor is short or if a short poke interval is required. Note that the sensor will hold onto a worker slot and a pool slot for the duration of the sensor's runtime in this mode.

   In the reschedule mode, the worker slot is released if during the poke, the callable returns false. Due to failure, it is rescheduled later. As a good practice, the poke interval should be more than one minute to prevent too much load on the scheduler.

2. The second option is to use a small DAG and then call them explicitly. Splitting a large DAG into smaller DAG is a separation of concern and helps in avoiding the deadlock problem. The added benefit of having small DAG is to reduce redundancy and increase reuse.

   While having small DAG leads to simplification, it adds a little overhead. One DAG might need to call others. Fortunately, Airflow offers an operator called `TriggerDAGRunOperator` to do the same (Are you surprised?). We will next look at `TriggerDAGRunOperator`.

4.7.5 Understanding TriggerDAGRunOperator

TriggerDAGOperator at a high-level needs the following two things:

1. A controller – A controller is a task that decides the outcome based on a condition (Think of a Python callable)
2. A target – It is the target DAG that is kicked off based on the controller outcome.

Let us build two DAGs for the following workflow:

Imagine a DAG called Triggering DAG invoking another DAG named Triggered DAG that has two tasks – task1 and task2.

Triggering DAG will trigger the 2nd DAG with a message. Code 4.21 shows the DAG code:

```
from airflow import DAG

from airflow.operators.dagrun_operator import TriggerDagRunOperator --1

from airflow.operators.python_operator import PythonOperator

from airflow.operators.bash_operator import BashOperator

from airflow.utils.dates import days_ago

############################### Controller DAG #################

dag_controller = DAG(

    dag_id="trigger_controller_dag",

    default_args={"owner": "airflow"},

    start_date=days_ago(2),

    schedule_interval="@once"

)


def send_msg(context, dag_run_obj):  --2

    dag_run_obj.payload = {'message': 'Hello World'}

    if True:

        return dag_run_obj
```

```
trigger = TriggerDagRunOperator(  --3
    task_id="trigger_dagrun",
   trigger_dag_id="trigger_target_dag",  # Ensure this equals the dag_id
of the DAG to trigger
    python_callable=send_msg,
    dag=dag_controller,
)
############################### Target DAG #################


dag_target = DAG(
    dag_id="trigger_target_dag",
    default_args={"owner": "airflow"},
    start_date=days_ago(2),
    schedule_interval=None --4
)


def run_this_func(**kwargs):  --5
    print("Remote value of {} for key=message".format(kwargs["dag_run"].
conf["message"]))


run_this = PythonOperator(task_id="run_this", provide_context=True,
    python_callable=run_this_func, dag=dag_target) --6


bash_task = BashOperator(
    task_id="bash_task",
    bash_command='echo "Here is the message: $message"',
    env={'message': '{{ dag_run.conf["message"] if dag_run else "" }}'},
    dag=dag_target,
)


run_this >> bash_task
```

Code 4.21: *DAG with TriggerDAGRunOperator*

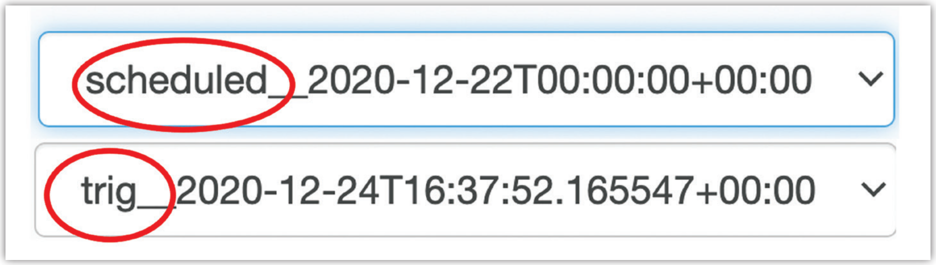
The controller and the target DAGs are defined in the same Python to save you from switching between different files. As a good practice, you may want to maintain them separately.

Let us unpack the code as follows:

1. To use the `TriggerDAGRunOperator`, we need to import it first.
2. This is the controller task and is in the form of Python callable.
3. We define the controller task (callable). It returns the `dag_run_obj` with a payload message that we would like to get delivered to target DAG. It's no surprise that the `dag_run_obj` can also be passed with context parameters.
4. Next, we define the target DAG. The only highlight to remember is keep the `schedule_interval = None`. This will ensure that Airflow doesn't schedules it. This is required because we would like the target DAG to be explicitly triggered by controller DAG.
5. `run_this` is a callable for the target DAG. The callable signature is something that you have seen before during parameter passing. Just to reiterate, a DAG run is an object representing an instantiation of the DAG in time. The conf is the configuration passed to the triggered DAG as a JSON blob. Note that the parameters from `dag_run.conf` can only be used in a template field of an operator.
6. Finally, we define the task as a Python operator that will be executed (triggered). The context needs to be set to True because the target DAG is invoked with the conf JSON blob.

One of the common use cases of trigger DAG that I know of, is to use it to extract the safety check from the main business logic. They provide a reusable way to add fail safe to the end of the workflow.

One last thing – how do you differentiate between a scheduled DAG and a triggered DAG? Scheduled DAG run names will have *scheduled_ in it whereas a triggered DAG will have trig_ in it as shown in figure 4.37*:



scheduled__2020-12-22T00:00:00+00:00

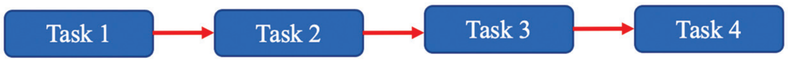
trig__2020-12-24T16:37:52.165547+00:00

Figure 4.37: *Scheduled and triggered DAG runs naming convention*

4.8 Managing task branching and dependencies

So far, you have seen linear or sequential progression of tasks execution - one after other. However, in the real-world, dependencies between tasks can take different paths. Like our life where one decision can change everything, one task may change the task(s) or their order of execution. In this section, we will learn about implementing complex dependencies between tasks.

Before we get into the details, let us revisit some of the dependencies that we have seen before:



$$\begin{tikzcd} \text{Task 1} \arrow[r] & \text{Task 2} \arrow[r] & \text{Task 3} \arrow[r] & \text{Task 4} \end{tikzcd}$$

Figure 4.38: *Sequentially organized tasks*

We can specify the tasks execution in the following two ways:

```
Task 1 >> Task 2 >> Task 3 >> Task 4
```

Or

```
Task 1 >> Task 2

Task 2 >> Task 3

Task 3 >> Task 4
```

You have seen few other complex dependencies:

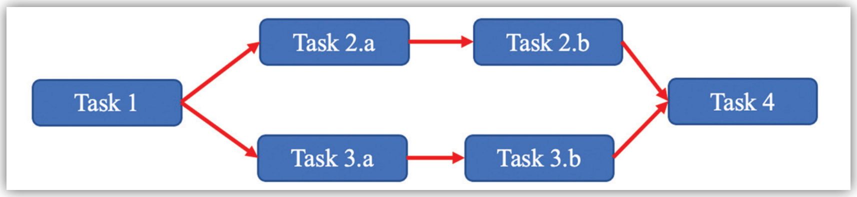


***Figure 4.39:** Fan in/out dependencies*

Figure 4.39 shows a fanning in/out of tasks. After task 1, the execution needs to split across two paths (Fan out), and they merge at task 4 (Fan in). The execution can be specified in the following way:

Task 1 >> Task 2.a >> Task 2.a >> Task 4

Task 1 >> Task 3.a >> Task 3.b >> Task 4

Let us twist the requirements. We now want to execute either the task 2 path or task 3 path. The task dependency may look as follows:

```
Task 1 >> Task 2.a >> Task 2.a >> Task 4
```

OR

```
Task 1 >> Task 3.a >> Task 3.b >> Task 4
```

To build the branching workflow, we need some decision maker in between. It wouldn't be wild to imagine the decision maker as another task. The new workflow may look *figure 4 39*:

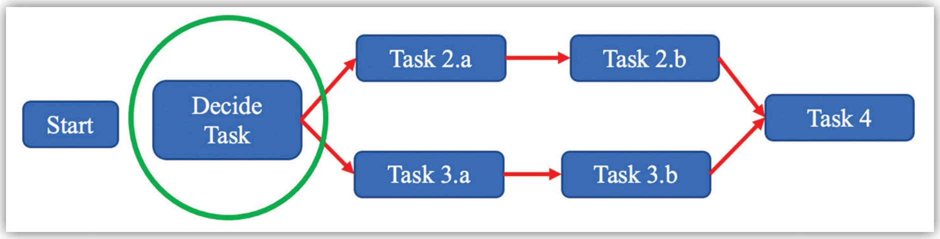


***Figure 4.40:** Workflow with a branching task.*

With your current knowledge, you can build these kinds of complex dependencies. Recall the following two operators:

- **PythonOperator:** Write your branching logic and branch work in a Python callable. While this would work technically, you would compromise with "separation of concerns" benefits.

- **DAGTriggerRunOperator:** You can implement Task 2* and Task 3* as separate DAGs and then invoke via DAGTriggerRunOperator. While this works, it may end up creating way too many small DAGs.

Both the options have their drawbacks. Fortunately, there is another way out BranchPythonOperator. This is like PythonOperator as it accepts a Python callable but returns the ID of the downstream task. The returned task ID would be executed next after the branching task:

```
def _decide_task(**context): --1
    year = context["execution_date"].year
    if year < CUT_OFF_YEAR: --2
         return "Task_2a" --3
     else:
         return "Task_2b"


decide_task = BranchPythonOperator( --4
        task_id=' decide_task',
        provide_context=True,
        python_callable=_decide_task,
)
```

Code 4.22: *Branching using BranchPythonOperator*

Let us unpack the code as follows:

1. `_decide_task` is the callable to decide the next task. Note that it is returning the next `task_id`.
2. We have implemented a hypothetical branching condition using the `execution_date`. If it is less than some cut-off date, it will return `Task_2a`. Think of a transition from an old system to a new system as a use case. Under the hood, `"execution_date"` is based on the Python pendulum package (https://pendulum.eustace.io/). Pendulum is a wonderful date time package that is timezone aware.
3. If the execution date is more than cut-off, return the other task – `Task_2b`.

4. It is time to instantiate the `BranchPythonOperator`. All the parameters are those you have already seen before and hence should not feel unfamiliar to you.

Let us go through a complete example as follows to see the ideas in action:

```
import airflow.utils.dates

from airflow import DAG

from airflow.operators.bash_operator import BashOperator

from airflow.operators.dummy_operator import DummyOperator

from airflow.operators.python_operator import BranchPythonOperator


from datetime import datetime, timedelta


default_args = {

        "owner": "airflow",

        "start_date": airflow.utils.dates.days_ago(12)

    }


def _decide_task(**context):

    year = context["execution_date"].year

    if year < 2022:

        return 'task_2a'

    else:

        return 'task_3a'


with DAG(dag_id="myPython_branch_dag", default_args=default_args,

schedule_interval="@daily") as dag:
```

```
start = BashOperator(
        task_id="start",
        bash_command="echo Start"
    )
decide_task = BranchPythonOperator(
   task_id='decide_task',
   provide_context=True,
   python_callable=_decide_task,
    )
task_2a = BashOperator(
        task_id="task_2a",
        bash_command="echo execute task_2a"
    )

task_2b = BashOperator(
            task_id="task_2b",
            bash_command="echo execute task_2b"
        )
task_3a = BashOperator(
        task_id="task_3a",
        bash_command="echo execute task_3a"
    )

task_3b = BashOperator(
            task_id="task_3b",
            bash_command="echo execute task_3b"
        )
    finish = BashOperator(
                task_id="finish",
                bash_command="echo Finish"
            )


start >> decide_task >> task_2a >> task_2b
start >> decide_task >> task_3a >> task_3b
task_2b >> finish
task_3b >> finish
```

***Code 4.23:** End-to-end example of branching using BranchPythonOperator*

Let us unpack the code as follows:

1. Since `BranchPythonOperator` is an operator, it needs to be imported.
2. The `_decide_task` is the Python callable that has the condition logic to return the next task IS. The logic extracts the year from the execution data. Year is available as a property in the pendulum date time type. It returns the next task ID based on the condition. Recall that to use `execution_date`, the context needs to be provided to the callable. You already know that is done with the property `property_context`.
3. Next, we define a bunch of tasks. All the tasks are bash operator except the `BranchPythonOperator` to keep things simple.
4. At last, we define the tasks dependencies. The best way to visualize the task dependencies is to use the graph view of Airflow and we discussed that also in detail. *Figure 4.40* shows the graph view for DAG (*code 4.23*):

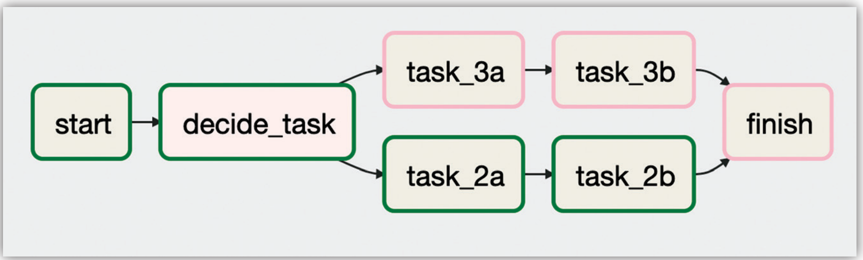


***Figure 4.41:** Graph view for DAG in Code 4.23*

Again, from the graph view, it is not apparent that `decide_task` is a branching operator and a good naming convention should be used to name the task. When you execute the DAG, the following tasks will be executed unless you are running this in 2022:

`Start` → `decide_task` → `task_2a` → `task_2b`

Note that finish is not executed. But why?

Well, the finish operator expects both `task_2a` and `task_3b` to complete. However, due to the conditional branching, `task_3b` was skipped, leading to finish being skipped as well. In fact, all the downstream tasks will be skipped. The reason for this behavior is the default behavior of Airflow. Airflow expects all upstream tasks of a task to finish before executing it. In *figure 4.41*, Finish has two upstream tasks – `task_2b` and `task_3b`, but only one will execute. Hence, Finish will be skipped.

What you just saw is called a trigger rule. Expecting all the upstream tasks to finish first is an example of a trigger rule. The trigger rule essentially defines the dependency settings. *Table 4.4* shows the list of other trigger rules as follows:

Trigger_rule	Description
`all_success`	All the upstream tasks have succeeded. This is default behavior.
`all_failed`	All parents are in a failed or upstream_failed state.
`one_failed`	Fires as soon as at least one parent has failed; it does not wait for all parents to be done.
`one_success`	Fires as soon as at least one parent succeeds; it does not wait for all parents to be done.
`none_failed`	All parents have not failed (failed or upstream_failed), that is, all parents have succeeded or been skipped.
`none_failed_or_skipped`	All parents have not failed (failed or upstream_failed) and at least one parent has succeeded.
`none_skipped`	No parent is in a skipped state, that is, all parents are in a success, failed, or upstream_failed state.
`dummy`	Dependencies are just for show; trigger at will.

Table 4.4: *Trigger rules list*

Can you guess the `trigger_rule` that will enable the Finish task to execute as well? If you guessed `one_success`, then well done, you are indeed correct. Code 4.24 shows the code snippet to add the trigger rule. It is added to finish task because we want to change the relationship between the finish task and its upstream task as follows:

```
finish = BashOperator(
            task_id="finish",
            trigger_rule = "one_success",
            bash_command="echo Finish"
        )
```

Code 4.24: *trigger_rule for BashOperator*

It shouldn't come as a surprise that all operators have this property.

There is another operator related to the conditional skipping of task – `LatestOnlyOperator`. Imagine a situation where you are running and backfilling (back dated runs) a job that sends out newsletter to your users. You don't want to send an old newsletter to your audience. One possible solution is to use the code that's branching in your Python callable but then you are going against the principal of separation of concerns. To think of it in a more general way, we want to execute a task for the latest run only. `LatestOnlyOperator` provides that logic out of the box. Code 4.25 shows the idea and concept in code:

```
from airflow.operators.latest_only_operator import LatestOnlyOperator

is_latest_run = LatestOnlyOperator(task_id="No_ newsletters_during_
backfills", dag=dag)

upstream_task >> is_latest_run >> send_newsletters >> other_downstream_task
```

Code 4.25: *Skipping task with LatestOnlyOperator*

The operator lets the downstream task run if the current time is between the most recent execution time and the next scheduled execution time. If no, then the downstream task(s) get skipped.

4.9 Sharing data between tasks using XCom

XCom (cross communication) enables task to exchange messages. The other way to think about it is to imagine it as a means for intra-task communication or sharing state between tasks. XCom is a structure containing the following:

- Key
- value
- timestamp
- DAG/Task id that created the message

Any data that can be pickled can be used in XCom. As a side note, it is important to remember that small or lightweight values should be used with XCom. Heavy/large size data exchange should be avoided via XCom. For large data exchanges, an external system like database or filesystem should be used.

XCom offers pull and push patterns for message exchange. A pushed XCom message is generally available to other tasks. Tasks can push XComs using the `xcom_push()` method. They are automatically pushed when a task returns a value either from its operator's execute() method, or from a PythonOperator's python_callable function.

To pull messages, XCom offers the xcom_pull() method. You can filter the messages using key, source task_ids, and dag_id. By default, xcom_pull() filters for the keys that are automatically given to XComs when they are pushed by being returned from execute functions (as opposed to XComs that are pushed manually).

xcom_pull can work for single or multiple task IDs. If it is passed as a single `task_id`, then the most recent XCom value from that task is returned. If a list of `task_ids` is passed, then a corresponding list of XCom values is returned.

Figure 4.42 shows the process of push and pull at a high level:

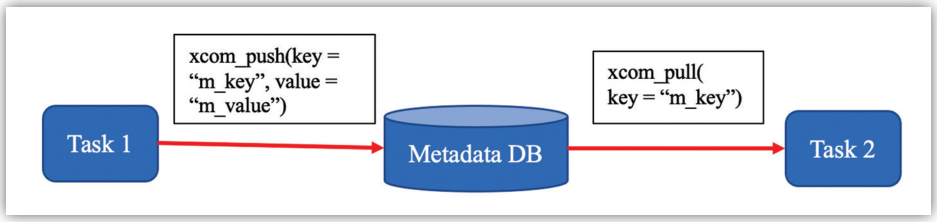

xcom_push(key = "m_key", value = "m_value")
xcom_pull(key = "m_key")
Task 1
Metadata DB
Task 2

***Figure 4.42:** XCom push and pull process*

Code 4.26 demonstrates XCom features and behavior in code:

```
import airflow

from airflow.models import DAG

from airflow.operators.dummy_operator import DummyOperator

from airflow.operators.python_operator import

BranchPythonOperator, PythonOperator

from airflow.operators.bash_operator import BashOperator

args = {

    'owner': 'Airflow',
    'start_date': airflow.utils.dates.days_ago(1),

}

def _push_xcom_return_value():

    return 'Hello_XCom'

def _get_xcom_return_value(**kwargs):

    print(kwargs['ti'].xcom_pull(task_ids='task0'))

def _push_next_task(**kwargs):

    kwargs['ti'].xcom_push(key='next_task', value='task3')

def _get_next_task(**kwargs):

    return kwargs['ti'].xcom_pull(key='next_task')

def _get_multiple_xcoms(**kwargs):

    print(kwargs['ti'].xcom_pull(key=None, task_ids=['t0', 'task2']))

with DAG(dag_id='my_xcom_dag', default_args=args, schedule_interval=None)
as dag:

    task0 = PythonOperator( --1

        task_id='task0',

        python_callable=_push_xcom_return_value

    )

    task1 = PythonOperator( --2

        task_id='task1',

        provide_context=True,

        python_callable=_get_xcom_return_value

    )

    task2 = PythonOperator( --3

        task_id='task2',

        provide_context=True,

        python_callable=_push_next_task
    )
    branching = BranchPythonOperator( --4
        task_id='branching',
        provide_context=True,
        python_callable=_get_next_task,
    )
    task3 = DummyOperator(task_id='task3') --5
    task4 = DummyOperator(task_id='task4') --6
    task5 = PythonOperator( --7
        task_id='task5',
        trigger_rule='one_success',
        provide_context=True,
        python_callable=_get_multiple_xcoms
    )
    task6 = BashOperator( --8
        task_id='task6',
         bash_command="echo value from xcom: {{ ti.xcom_pull(key='next_
task') }}"
    )
    --9
    task0 >> task1
    task1 >> task2 >> branching
    branching >> task3 >> task5 >> task6
    branching >> task4 >> task5 >> task6
```

Code 4.26: *XCom usages*

Let us unpack the code. Since there are quite a few tasks, a graphical view of DAG as follows would help in understanding the task dependencies:

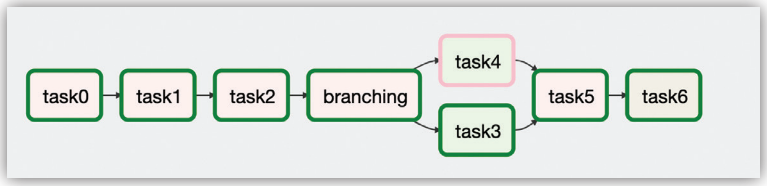
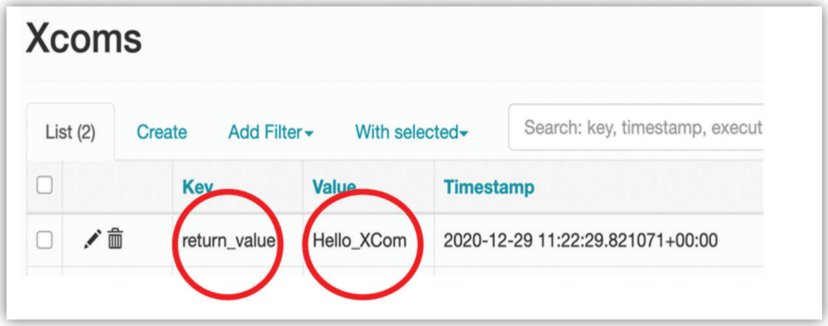


Figure 4.43: *Graph view for tasks sharing messages*

In code 4.26, some of the tasks are pushing information and a few of them are pulling and based in the message taking actions. Let us unpack the code as follows:

1. task0 is a Python operator that calls a Python callable and returns a string. This string is passed along as an XCom message. If you look at the XCom screen (admin | Xcoms), you will see an entry for "Hello_XCom" with the key as return_value. This is the default key name. This key-value was persisted in the metadata database as follows:



Key	Value	Timestamp
return_value	Hello_XCom	2020-12-29 11:22:29.821071+00:00

Figure 4.44: *XCom entry for the pushed message*

2. task1 is again a Python operator that calls a Python callable reading the XComs from task0. kwargs['ti'] is the task instance and we call the xcom_pull method with task0 as the parameter. This is a filtering of XCom and it essentially reads the XComs from task0. You already know that the task instance is called context and is made available to the command by setting the provide_context to True.

   Can you recall where can you see this print statement? Well, you click the DAG on the dashboard, and click on the graph view and click on the task0 in the bottom pane and click on the view log on the pop up.

3. task2 is again a Python operator that pushes a new XCom.

```
kwargs['ti'].xcom_push(key='next_task', value='task3')
```

   If you go to XCom's screen (Admin | XComs), you will see this new key value pair along with other details:

4. Next (branching) is a Python branch operator that reads the next_task key and executes that task next. Recall that the Python branch operator returns the ID of the next task. Note that while reading the XCom, we have not used any filter like task id:

   ```
   return kwargs['ti'].xcom_pull(key='next_task')
   ```
5. task3 is a simple branch implemented as a dummy operator.
6. task4 is another simple branch implemented as a dummy operator.
7. task5 is a Python operator having a callable reading multiple XComs. Since that is available in the context (task instance) and needs to be passed, this time, we are using a list of task ids as the filter:

   ```
   print(kwargs['ti'].xcom_pull(key=None, task_ids=['t0', 'task2']))
   ```
8. task6 is a bash operator that reads the XCom in the template. This is to show you the availability of XComs in templates as well.
9. At last, you specify the task dependency. You already know by now that the graph view from Airflow is a good way to visualize the task dependencies (*figure 4.44*).

You can change the XCom behavior of serialization and deserialization of tasks' result. Following are the steps to do it:

1. Change xcom_backend parameter in Airflow config. The provided value should point to a class that is a subclass of BaseXCom.
2. Override serialize_value and deserialize_value methods in the subclass.

All the XComs will have to be cleaned from the metadata DB as they are not cleaned automatically.

4.10 Understanding Airflow variables

If you look in the documentation, then variables are defined as a generic way to store and retrieve arbitrary content or settings as a simple key value store within Airflow. I prefer to imagine it to store and read data at runtime and avoid hardcoding in the DAG.

Variables can be created via UI (Admin | Variables), CLI (command line interface), and code.

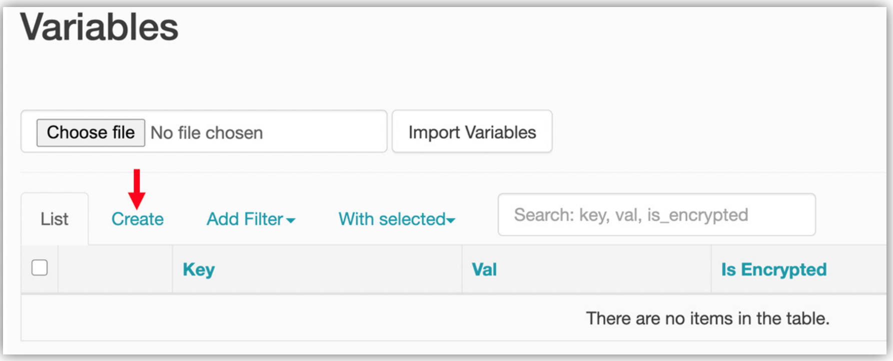
4.10.1 Variable creation

Here, we will see various variables used in Airflows along with their creation.

4.10.1.1 UI

To create a variable via UI, follow these steps:

1. Admin | Variables. The screen would look like *figure 4.45*:



Variables

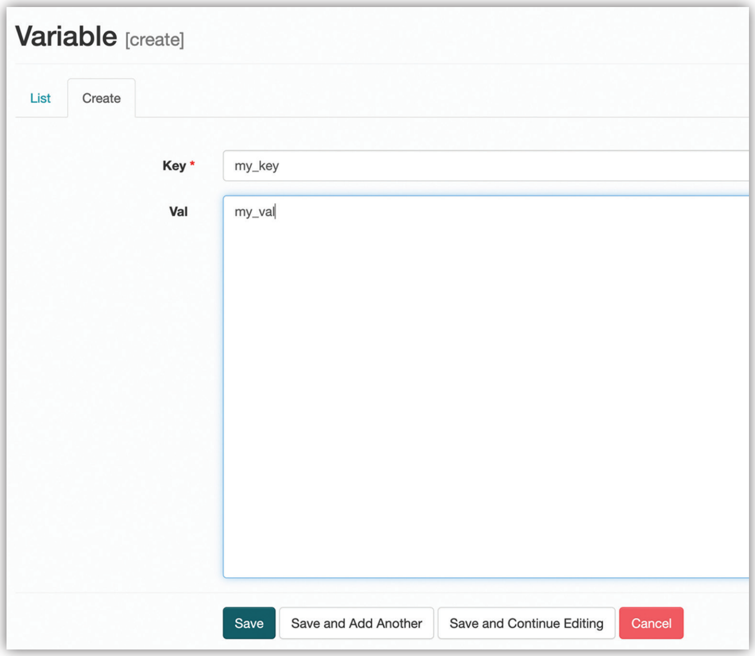
Search: key, val, is_encrypted

	Key	Val	Is Encrypted

There are no items in the table.

Figure 4.45: *Variables List screen*

2. Click on Create. Enter your key and value. Click on Save. This would save the variable in the metadata database.



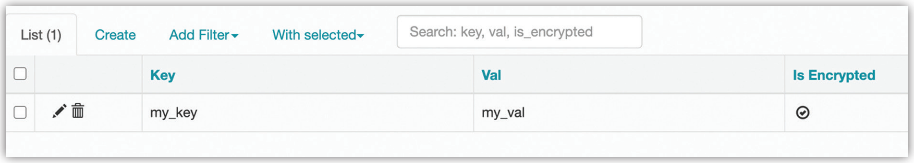
Variable [create]

Key * my_key

Val my_val

Figure 4.46: *Variable creation page*

3. Go back to Airflow | Variable page and you would see your variable. *Figure 4.47* lists the variable created at step 3.



Key	Val	Is Encrypted
my_key	my_val	✓

Figure 4.47: *Variable list page with created variables*

4.10.1.2 CLI

The following are the steps to create the variable(s) via command line:

1. Go to terminal and type the following command. You can view the variable on the Admin | Variable page:

   ```
   airflow variables -s my_cli_key my_cli_val
   ```

2. You can explore the command by listing the help:

   ```
   airflow@36024047e87e:~$ airflow variables -h
   [2020-12-29 18:24:33,448] {settings.py:253} INFO - settings.
   configure_orm(): Using pool settings. pool_size=5, max_overflow=10,
   pool_recycle=1800, pid=33491
   usage: airflow variables [-h] [-s KEY VAL] [-g KEY] [-j] [-d VAL]
                            [-i FILEPATH] [-e FILEPATH] [-x KEY]
   ```

   optional arguments:

   ```
     -h, --help            show this help message and exit
     -s KEY VAL, --set KEY   VAL
                             Set a variable
     -g KEY, --get KEY       Get value of a variable
     -j, --json              Deserialize JSON variable
     -d VAL, --default VAL
                             Default value returned if variable does
                             not exist
     -i FILEPATH, --import   FILEPATH
                             Import variables from JSON file
     -e FILEPATH, --export   FILEPATH
                             Export variables to JSON file
     -x KEY, --delete KEY    Delete a variable
   ```

4.10.1.3 Code

You can create variables from directly inside the DAG:

```
from airflow.models import Variable

my_var = Variable.set("my_code_key", "my_code_value")
```

4.10.1.4 Environment variables

Airflow variables can also be created and managed using environment variables. It is important to follow the naming convention and the naming convention is `AIRFLOW_VAR_{VARIABLE_NAME}`, all uppercase. So, if your variable key is FOO, then the variable name should be `AIRFLOW_VAR_FOO`:

```
export AIRFLOW_VAR_FOO=BAR

# To use JSON, store them as JSON strings

export AIRFLOW_VAR_FOO_BAZ='{"hello":"world"}'
```

Now that the variable has been created, using it in the code is pretty straightforward:

```
from airflow.models import Variable

foo = Variable.get("foo")

bar = Variable.get("bar", deserialize_json=True)

baz = Variable.get("baz", default_var=None)
```

Code 4.26: *Using variables in code*

The second call is for variables that are in the JSON format. With the deserialize_json flag switched ON, Airflow will deserialize it into the bar as such. There is a benefit of using JSON. Since variables are maintained in database, it means each variable access would incur a separate DB call. To cut down on the DB call, one possible option is to club the values in one JSON. So, you specify your value in JSON, and Airflow can deserialize it for you. Consider it as a variable optimization technique.

The third call uses the default_var parameter with the value None, which either returns an existing value or none if the variable isn't defined. The get function will throw a KeyError if the variable doesn't exist, and no default is provided.

To use a variable inside a Jinja template, use the following format:

```
echo {{ var.value.<variable_name> }}
```

In case the value is a json object, it needs to be deserialized:

```
echo {{ var.json.<variable_name> }}
```

4.10.2 Hiding variable values

At times, you may want to hide the values of your variable. Password is a good example of it. The process is pretty straightforward. All you need is to prefix your key with the following strings. For example, `google_secret_access_key`:

- `'password'`
- `'secret'`
- `'passwd'`
- `'authorization'`
- `'api_key'`
- `'access_token'`

Prefer environment variable if possible because it saves the database round trip if stored in the metadata database.

Let us do a quick recap. Workflows are an integral part of the data processing pipeline and framework as workflow simplifies the setup, maintainability, and monitoring of the systems leading to standardization and repeatability of the process.

4.11 Conclusion

Airflow enables you to automate a considerable amount of data processing tasks. Consider Airflow when you are looking for automated ETL tasks. With its "configuration as code" approach, it allows you to create complex a workflow with various task dependencies in a standard way. The biggest challenges with data pipelines are monitoring, dependency management and management. Airflow UI offers a great way to monitor tasks status, progress, and errors. The scheduler offers a scalable way to schedule jobs and define the dependencies between tasks. The common and yet very important features like retires, error handling, notifications, etc. are available out of the box leading to considerably reduced time for implementation. It has integration module (read operators) for a lot of third-party systems making it easier to include other systems. Being written in Python lowers the barrier for getting started.

In the next chapter, we will explore MLflow – an open-source platform for managing end-to-end ML lifecycle.

4.12 Points to remember

- Airflow is a workflow management system.

- Consider Airflow for batch-mode operations. Consider Spark (https://spark.apache.org/) for streaming data processing needs.
- There are multiple ways to install but prefer Docker-based installation as it makes the setup repeatable and almost production-ready.
- At the bare minimum, Airflow has three parts which are as follows:
  - o Webserver – To run the Airflow web app.
  - o Scheduler – To run the scheduled jobs.
  - o Backend – A database to manage all the task metadata.
- To run Airflow, you need to:
  - o Initialize the database.
  - o Run the web server.
- Use docker-compose to group different Airflow components.
- DAG (directed acyclic graph) is a graph with nodes and edges. Edges connect the nodes and only go in one direction. There is no loop back.
- Airflow tasks are represented as DAG nodes and edges represent the relationship or dependencies between task.
- All the DAGs should be maintained in a predesignated folder. By default, it is the "dags" folder.
- A DAG must be switched ON before you can schedule/trigger. By default, DAGs are in off state.
- A DAG can be run via UI, command line, or trigger_dag CLI command.
- Executor is responsible for the task execution. It reads the task from the queue and then specifies how to execute it. If the executor is sequential, then the tasks will be executed in a sequential manner. It is important to note that executor only provides the context for the job and doesn't execute the job. Airflow offers the following types of executors:
  - o `SequentialExecutor`
  - o `LocalExecutor`
  - o `CeleryExecutor`
  - o `DaskExecutor`
  - o `KubernetesExecutor`
- If celery executor is being used, then consider using flower (https://flower.readthedocs.io/en/latest/) as the celery monitoring tool.
- Celery requires a message broker to queue tasks. Redis (https://redis.io/) or rabbitMQ (https://www.rabbitmq.com/) are good choices.

- Some of the commonly used operators are as follows
  - o `BashOperator` - Executes bash commandos.
  - o `PythonOperator` - To run Python code snippets.
  - o `BranchPythonOperator` - To run Python code snippets but that return a task ID.
  - o `DummyOperator` - A dummy task that does nothing.
  - o `FileSensor` - To check a file sensor. It is an example of an event driven operator.
  - o `pythonSensor` – It works like a sensor but custom logic can be applied.
  - o `DAGTriggerRunOperator` - Use it to trigger DAG programmatically.
- Airflow uses ninja templating to provide dynamic values to commands/ tasks.
- Trigger_rules defines the task dependencies. The default is "all_success" that expects all upstream tasks to complete successfully.
- Use XCom to exchange data between tasks.
- XComs are structures containing the following:
  - o Key
  - o value
  - o timestamp
  - o DAG/Task id that created the message
- XComs are not suitable for large scale data transfer.
- Use external systems like database for large size data transfer between tasks.

  XCom needs to be deleted explicitly as it is not removed by the system.
- Variables are a generic way to store and retrieve arbitrary content or settings as a simple key value store within Airflow.
- Variables can be created via the following:
  - o UI
  - o terminal
  - o Code
  - o Environment variables
- Except environment variables, variables created via code, CLI, and UI get persisted in the metadata database.
- In terms of performance, environment variables will be better as they save the database round trip.

4.13 Questions

1. What is a DAG?
2. What are the three major parts of Airflow?
3. What are executors?
4. What are the different executors in Airflow?
5. Which operator would you use to execute Python code snippets?
6. Which operator would you use to execute bash commands?
7. What CLI command is used to trigger a workflow?
8. Which file contains Airflow configuration?

CHAPTER 5
Organizing Your Data Science Project

You don't need to worry about organizing your project if:

- You are the only person working on the project.
- Your first trained model meets the project requirements.
- Your data does not require pre-processing at all.

If any of the preceding points are not true, then you need to think about effectively organizing your projects. A well-organized project enables better collaboration, repeatability, and re-usability. Project organization is not just about code base but the environment as well. Not everyone can afford environments like Google, Amazon, and Facebook. Cloud is not always the best answer, as marketed by various cloud vendors. If you are running model training continuously for weeks, then buying a GPU machine is a lot cheaper than doing it in cloud.

Structure

In this chapter, we will discuss the following topics:

- Project folder and code organization
- GPU 101
- On-premises vs. cloud

You may not want to read about the iterative nature of ML projects again, but it is worth re-iterating until we really believe in it. Like a classical software project, the ML projects also have a lifecycle, albeit very different.

The focus of this chapter will be on learning about MLflow (https://mlflow.org/), an open-source platform to simplify the management of machine learning project lifecycle. At a high level, the lifecycle includes the abilities of experimentation, reproducibility, and deployment.

Structure

In this chapter, you will learn the following:

- Understanding ML pipeline challenges
- MLflow components

Objective

By the end of this chapter, you will know the following:

- The challenges of ML lifecycle in multi-ML teams organizations.
- Achieving "write once use everywhere" with MLflow.
- Understanding the different components of MLflow.
- Using MLflow across different stages of development.

5.1 Understanding ML pipeline challenges

To understand the ML project challenge, let us compare it with the traditional software development lifecycle. But before that, I would like to share an experience from the early days. In one of the projects, the metrics of a new training run was published and it was not good at all. One of the data scientists was entrusted to identify the new things to try. After a day, he wrote an email to the developer, sharing with her a new model to try. It was an interesting and weird situation. Nobody had any idea about what was new in that model. It is not important as to what happened at the end, but there are multiple problems worth highlighting here. These are as follows:

- What features were being used?
- Was there any change in feature engineering?
- What hyperparameters had been changed?
- What data was used – same, new, or modified?
- What architecture was tried?

I am sure that you would be able to add many more to the list. But the underlying problem is the lack of transparency in the process. This ad hoc black box model is an invitation to the following problems:

- How would you retrain if you had a new dataset?
- How would you iterate over the black box model if it only marginally improved the metrics?
- How would you debug it?

Think of these problems at scale. It will be a nightmare. Let us now go back to the point about comparing a traditional software release with an ML release. Let us remind ourselves about the high-level steps in a traditional software release which are as follows:

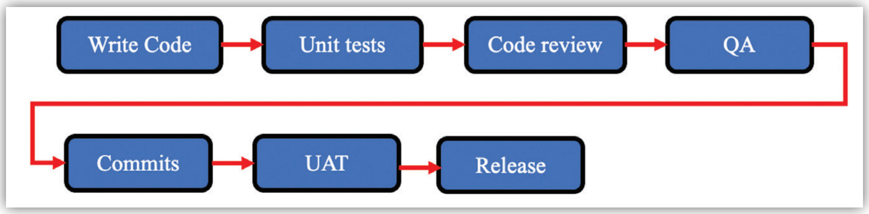


Figure 5.1: *Traditional software release lifecycle*

I accept that this flow is very simple and would feel quite familiar if you have ever written any commercial software. But since people can interpret it differently depending upon their experience, I will briefly detail out the steps as follows:

1. Based on the requirements, you write code(s).
2. Then, you write test cases for the functional code.
3. The code is reviewed by peers and seniors for any potential issues.
4. It is released to QA or testing teams.
5. Finally, the code is committed for release.
6. Stakeholders do the acceptance testing.
7. The software is released in production.

The ML process is quite different, and hence, it wouldn't be wise to apply these traditional principles and rules to this programming paradigm. To make things concrete, let us now look at the lifecycle of an ML project as follows:

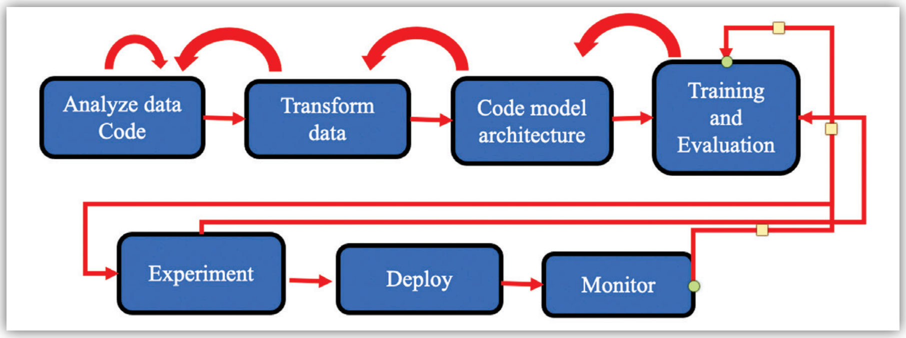


Figure 5.2: *ML release lifecycle*

Let us unpack the steps in *figure 5.2* as follows:

1. We first need to analyze the data. You can clearly see the difference in terms of activity! The data analysis may involve, let us say, Spark (https://spark.apache.org/). Even the tools are so different here. A considerable amount of data analysis has been done here in an iterative way.
2. Next, the data needs to be transformed to make it compatible with the ML architecture (algorithm). The one hot encoding, normalization, and standardization are a few examples of it. In a nutshell, it needs to be put in a format that can be used by ML models.
3. Then, you code the model architecture. This coding is connected to step 2 because in the beginning, you generally start with the simplest model and then iterate. The model code may need special data transformation. There is a feedback loop at this level as well.
4. The next feedback loop comes into play when you fully train and validate the model. Based on the outcomes, you may move to step 2 or 3.
5. Based on the step 4 feedback loop, you try with different hyperparameters or model architectures. It is not uncommon to have 100s of experiments being done here. Within a very small team, I have seen people managing such experiments with excel spreadsheets.
6. Finally, you deploy it.
7. The ML model output will change as the data distribution changes. This is called drift and model needs to be retrained. There could be other reasons as well. For example, there could be newer architecture that is being claimed to perform better. Maybe the underlying data is going through changes. There are just so many changes.

The steps shouldn't be surprising to you, but you can see the stark difference between not only the process but the toolset as well. Let us compare the two side by side in *figure 5.3*.

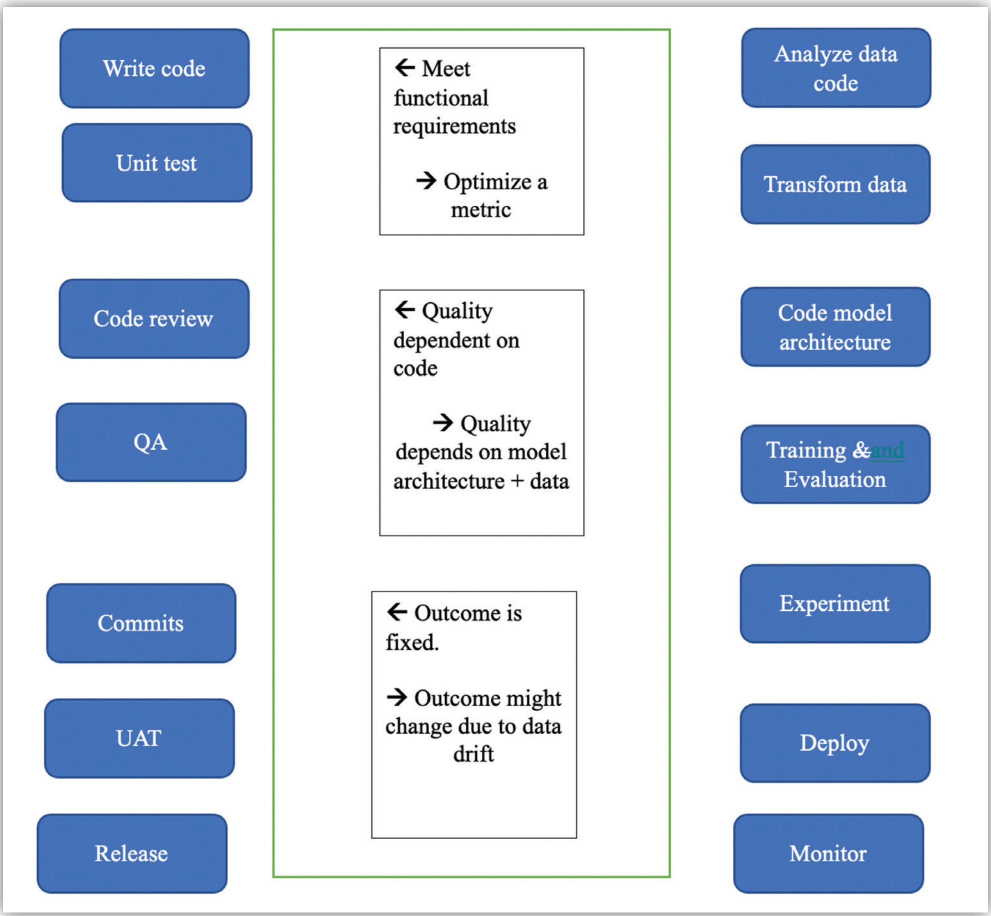

Write code
Unit test
Code review
QA
Commits
UAT
Release
← Meet functional requirements
→ Optimize a metric
← Quality dependent on code
→ Quality depends on model architecture + data
← Outcome is fixed.
→ Outcome might change due to data drift
Analyze data code
Transform data
Code model architecture
Training & and Evaluation
Experiment
Deploy
Monitor

Figure 5.3: *Process and output comparison*

Let us think about the differences a little bit more as follows:

1. In a traditional software (L.H.S.), the objective is to meet some functional requirements. The path to those functional requirements is pretty much known upfront. However, in ML application (R.H.S.), the objective is to minimize or maximize some metrics. The path to achieving that objective is not at all clear.
2. The quality is entirely dependent on how well the code has been written. To ensure correctness, the unit test provides the coverage and they are pretty much deterministic. Unless the code changes, the quality would not change. Whereas an ML application is code + data, which means the quality would change with data. Think of phone cameras that are upgraded so fast. Photo quality changes drastically from generation to generation. A model trained

on a certain phone camera generation may not work with the photo the same phone in the next generation camera.

3. The outcome is pretty much defined and deterministic. No code changes would mean no change in behavior. However, in ML applications, this claim has no meaning. With change in data distribution (called as drift) the output would vary. Even if you don't change the code, the model would require regular trainings.

You would agree that for an ML application, a different workflow system is required that supports the iterative nature of the steps and keeps track of things. The following are a few popular choices. This list is not exhaustive but does not indicate the seriousness and need for such a platform:

- **Kubeflow (https://www.kubeflow.org/)** – Kubeflow is a pipeline for building, deploying, and managing ML projects on Kubernetes.
- **Facebook FBLearner** – This is the pipeline used inside Facebook to manage the building and deploying of FB ML projects
- **Uber's Michelangelo** – The Michelangelo platform is used inside Uber to build and manage the ML project pipeline.
- **Google TFX (https://github.com/tensorflow/tfx)** – TFX (Tensorflow Extended) is a Google ML platform to build and manage Google ML products and projects.

5.2 MLflow components

At a high level, MLflow's features can be grouped in the following four components:

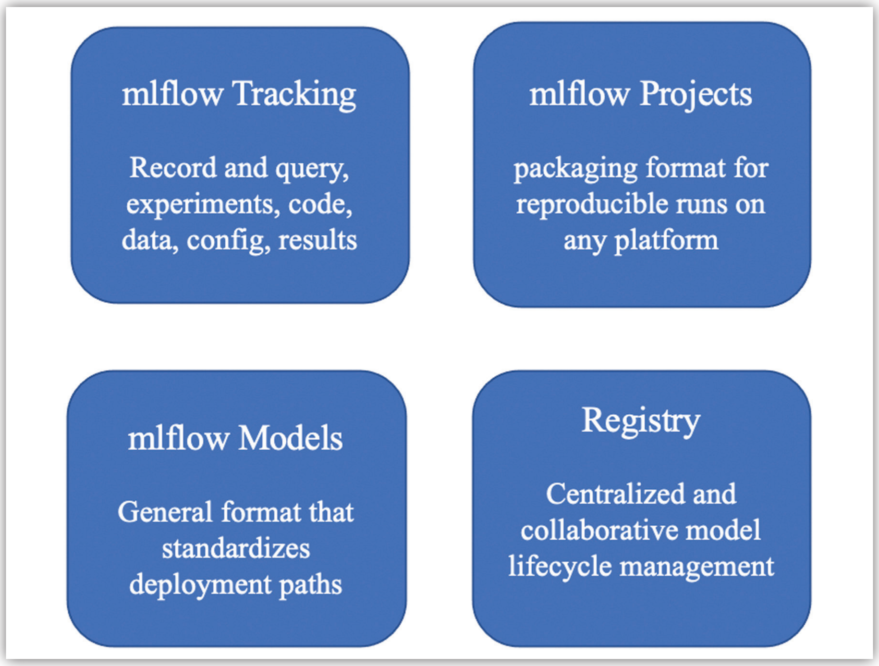

mlflow Tracking
Record and query, experiments, code, data, config, results
mlflow Projects
packaging format for reproducible runs on any platform
mlflow Models
General format that standardizes deployment paths
Registry
Centralized and collaborative model lifecycle management

***Figure 5.4:** MLflow components*

- **MLflow tracking:** MLflow tracking offers the UI and API for logging values. The logged value could be the parameters, code versions, metrics, and artifacts. The UI offers visualization of the logged parameters. The tracking is possible in any environment (for example, a standalone script or a notebook) and you can log the parameters and the results to the local files or to a server across multiple runs and then compare the results.
- **MLflow projects:** MLflow Project enables you to share code with others. To make it sharable, it offers a standard format for packaging. Each project has a descriptor file to specify the dependencies and details around how to run the code. It is simply a directory with code or a Git repository. If you have experience with the Python Anaconda package manager, then you would already have some idea. Think of how a conda YAML file specifies the environment and dependent packages. The tracking API in a Project MLflow automatically remembers the project version (Think of Git commit) and any other parameters. You can run your MLflow project from GitHub; or your internal Git repository; or chain the steps to create a workflow.
- **MLflow models:** The MLflow Model is a convention-based packaging format and tools that let you easily deploy the same model (from any ML library) to batch and real-time scoring on platforms such as Docker, Apache Spark, Azure ML, and AWS SageMaker. Think of a model being served through a REST API or batch inference on Apache Spark. I prefer to imagine it as an

abstraction over a model so that the pipeline features remain independent of the underlying ML library.

Under the hood, the models are saved as a directory containing arbitrary files and a descriptor file that lists several "flavors" that the model can be used in. Think of flavors as conventions that deployment tools can use to understand the model without having to integrate the underlying ML library. The following examples would be helpful here:

    - o A TensorFlow model loaded as a TensorFlow DAG.
    - o A Python function to apply to input data.
    - o A model supporting the "Python function" flavor deployed to a Docker-based REST server.
    - o A model supporting the "Python function" flavor deployed to cloud platforms such as Azure ML and AWS SageMaker.
    - o A model used as a user-defined function in Apache Spark for batch and streaming inference.

- MLflow registry: MLflow Registry is a recent addition to the MLflow world. It offers a centralized model store, a set of APIs, and UI, to collaboratively manage the full lifecycle of an MLflow model. Think of the traditional software deployment process that includes "staging" like construct but having more ML semantics baked into it. There are two dimensions to it. These are as follows:
    - o The first one is the concept of stages. So, models don't just get created and then deployed; they often go through stages, such as staging environments, AB testing and deployment. The MLflow registry offers these abstractions in the registry to represent these different maturity levels of deployment.
    - o The second dimension is around use cases where a handoff between different people are required. Imagine a situation where a data scientist shares a new model to be tested and that request needs to be approved. Such an authorization process is also built into the model registry.

We will get into the details of each of these components shortly, but before that, let us be clear on the uses cases that the preceding components are supposed to solve:

- **How to keep track of experiments.** How do you keep track of data, code, and the parameters used to get results?
- **How to reproduce results.** It is not just about tracking the code and the parameters but also the whole environment used. If it is Python, think of all

the packages and their respective versions used. This is especially challenging if you want another data scientist to use your code, or if you want to run the same code at scale on another platform (for example, in the cloud).

- **There's no standard way to package and deploy models.** To ensure consistency and repeatability, it is necessary that packaging and deployment are done in a single way across the board.
- **There's no central store to managing models (their versions and stage transitions).** During the ML project life cycle, the data science team creates many models. How would another ML team use your model? How would you keep track of model progress across different stages (testing, staging, production etc.)? How would you ensure that your model consumers are not impacted during transitions to newer models?

5.2.1 MLflow tracking

Let us go through an example using a linear regression model. I have deliberately kept the model simple to focus on the mlflow components. We will use the diabetes dataset available under sklearn only and will use only the first column. Installing mlflow is straightforward: `pip install mlflow`:

```
#1

import matplotlib.pyplot as plt

import numpy as np

from sklearn import datasets, linear_model

from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

from urllib.parse import urlparse

import mlflow

import mlflow.sklearn


#2

import logging

logging.basicConfig(level=logging.WARN)

logger = logging.getLogger(__name__)
```

```
def eval_metrics(actual, pred): #3

    mse = mean_squared_error(actual, pred)

    rmse = np.sqrt(mean_squared_error(actual, pred))

    mae = mean_absolute_error(actual, pred)

    r2 = r2_score(actual, pred)

    return mse, rmse, mae, r2


# Load the diabetes dataset #4

diabetes_X, diabetes_y = datasets.load_diabetes(return_X_y=True)


# Use only one feature #5

diabetes_X = diabetes_X[:, np.newaxis, 2]


# Split the data into training/testing sets #6

diabetes_X_train = diabetes_X[:-20]

diabetes_X_test = diabetes_X[-20:]


# Split the targets into training/testing sets #7

diabetes_y_train = diabetes_y[:-20]

diabetes_y_test = diabetes_y[-20:]


# Create linear regression object

regr = linear_model.LinearRegression() #8


with mlflow.start_run(): #9

    # Train the model using the training sets
    regr.fit(diabetes_X_train, diabetes_y_train) #10


    # Make predictions using the testing set
    diabetes_y_pred = regr.predict(diabetes_X_test) #11
    (mse, rmse, mae, r2) = eval_metrics(diabetes_y_test, diabetes_y_pred) #12
    #print the values
    # The coefficients
    print('Coefficients: \n', regr.coef_)
    # The mean squared error
    print('Mean squared error: %.2f'
      % mse)
    # The root mean squared error
    print('Rroot Mean squared error: %.2f'
      % rmse)
    # The mean absolute error
    print('Mean absolute error: %.2f'
      % mae)
    # The coefficient of determination: 1 is perfect prediction
    print('Coefficient of determination: %.2f'
      % r2)
#13
    mlflow.log_param("coefficient", regr.coef_)
    mlflow.log_metric("mse", mse)
    mlflow.log_metric("rmse", rmse)
    mlflow.log_metric("mae", mae)
    mlflow.log_metric("r2", r2)


    tracking_url_type_store = urlparse(mlflow.get_tracking_uri()).scheme #14

    if tracking_url_type_store != "file": #15

  mlflow.sklearn.log_model(regr, "model", registered_model_
  name="LinearRegressionModel") #16


    else:

        mlflow.sklearn.log_model(regr, "model") #17
```

Code 5.1: *Tracking with MLflow for Linear regression model.*

Let us unpack the code as follows:

1. We import the necessary libraries. We are using LinearRegression from sklearn and, hence, they need to be imported. We would like to track metrics like MSE (mean square error), RMSE (root mean square error) etc., and, hence they are imported too. Mlflow.sklearn provides an API for logging and loading scikit-learn models. This model, under the hood, exports scikit-learn models.
2. The logging module is not directly used here but should be used in the place of print statements. Using print statement for a production code is almost like a command.
3. eval_metric function calculates the metrics that we would like to observe. It is returning mse, rmse, mae, and r2. These are popular loss functions for regression models, and I assume you know about them.
4. We will use the diabetes dataset available in sklearn. For learning, I prefer to stick to the sklearn dataset as it is seamlessly integrated in the framework. It helps to focus on the core idea rather than getting drowned in data cleaning and setup code.
5. To keep things simple further, we will use only one column. As a practice, I would encourage you to adapt this example to multiple columns.
6. We will split the dataset between training and testing.
7. The splitting needs to be done for target values as well.
8. We initialize the LinearRegression. This will not be surprising.

9. mlflow.start_run() launches a new run. A run is a term for running a training. Under the hood, it returns the currently active run (if one exists), or starts a new run, and returns an mlflow.ActiveRun object usable as a context manager (Python context manager) for the current run.

   Just as a side note, you do not need to call start_run explicitly: calling one of the logging functions with no active run automatically starts a new one. We will take a look at the logging function shortly.

   If there is start_run, then you would expect end_run as well. Yes, mlflow.end_run is available.

   There is another use case for start_run. If you have to launch multiple mlflow runs in the same program: say for example, during a hyperparameter search; then the context_manager (Python context manager) returned by start_run makes it easy to do so. You can simply "scope" each run to just one block of code. The following example would be helpful here:

   ```
   for epoch in range(0, 3):

       mlflow.log_metric(key="quality", value=2*epoch, step=epoch)
   ```

10. With the context manager initialized, we fit the regression model on the training dataset.
11. We make the predictions using the predict method. Again, this is standard stuff.
12. We then calculate the metrics that we are interested in tracking.
13. It is time to log the parameters and the metrics.
    - `mlflow.log_param()` logs a single key-value param in the currently active run. The key and value are both strings. You can use mlflow.log_params() to log multiple params at once.
    - `mlflow.log_metric()` logs a single key-value metric and the value must always be a number. MLflow remembers the history of the values for each metric.

The following table lists some of the other commonly used loggers:

Function name	Description
`mlflow.log_metrics()`	Logs multiple metrics at once.
`mlflow.set_tag()`	Sets a single key-value tag in the currently active run. The key and value are both strings.
`mlflow.set_tags()`	To set multiple tags at once.

`mlflow.log_artifact()`	Logs a local file or directory as an artifact, optionally taking an artifact_path to place it within the run's artifact URI. Run artifacts can be organized into directories, so you can place the artifact in a directory this way.
`mlflow.log_artifacts()`	Logs all the files in a given directory as artifacts, again taking an optional artifact_path. `mlflow.get_artifact_uri()` returns the URI that artifacts from the current run should be logged to.

Table 5.1: *MLflow logging functions*

While we are talking about logging, it begs the question - where are the values logged? MLflow runs can be recorded to local files, to a SQLAlchemy compatible database, or remotely to a tracking server. By default, the MLflow Python API logs run locally to files in an mlruns directory wherever you run your program. You can then run mlflow UI to see the logged runs.

To log runs remotely, set the MLFLOW_TRACKING_URI environment variable to a tracking server's URI or call mlflow.set_tracking_uri(). MLflow supports the following different kinds of remote tracking URIs:

- Local file path (specified as `file:/my/local/dir`) where data is just directly stored locally.
- Database encoded as `<dialect>+<driver>://<username>:<password>@<host>: <port>/ <database>`. MLflow supports the dialects mysql, mssql, sqlite, and postgresql.
- HTTP server (specified as `https://my-server:5000`), which is a server hosting an MLflow tracking server.
- You can also use the Airflow managed environment by Databricks. (https://databricks.com/product/managed-mlflow).

Eventually we log the model as scikit-learn model as mlflow artifact for the current run. Just to ensure that you realize the power of this abstraction, look at the following list. Each of the frameworks have a log_model method. Irrespective of which framework you use, the abstraction knows how to persist the model.

There are certain key concepts that are important to understand and internalize with respect to tracking. For reproducibility, it is important to keep track of important hyperparameters (like a knob) that impact model performance. Along with the hyperparameters, metrics indicating the model performance should be tracked as well. For useful traceability, it is helpful and required to keep track of artifacts as well. Think of notes and annotations that you learnt, jotted down, and would like to make available to others.

The mlflow tags and notes function provides the means to track the artifacts. Imagine that as an ML developer, you trained a model and after a few months, someone else would like to extend it. All the hyperparameters and metrics along with artifacts would help them to get started from the point that you left off and not start from zero. Would you not agree that is big time-saver?

In the real world, different teams may have different training ecosystems. Some may be training in cloud, some may be on premise, and a few may be on local machines. Regardless of where the training is being done, mlflow tracking API provides a single, consistent way of capturing and logging all the details.

The next step is to aggregate the collected information. mlflow offers a pluggable centralized tracking server that can run on different infrastructural environment. mlflow offers its own UI to view the aggregated data and API to expose the collected aggregated data.

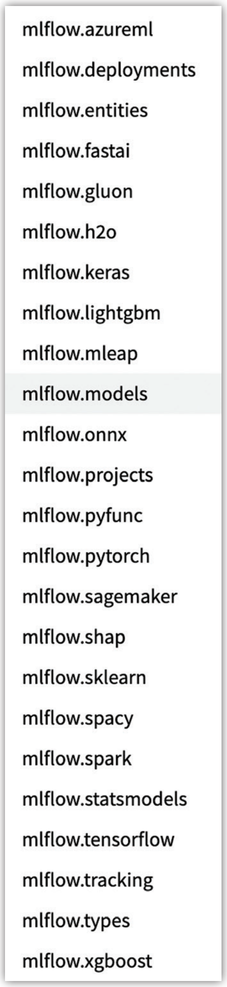

mlflow.azureml
mlflow.deployments
mlflow.entities
mlflow.fastai
mlflow.gluon
mlflow.h2o
mlflow.keras
mlflow.lightgbm
mlflow.mleap
mlflow.models
mlflow.onnx
mlflow.projects
mlflow.pyfunc
mlflow.pytorch
mlflow.sagemaker
mlflow.shap
mlflow.sklearn
mlflow.spacy
mlflow.spark
mlflow.statsmodels
mlflow.tensorflow
mlflow.tracking
mlflow.types
mlflow.xgboost

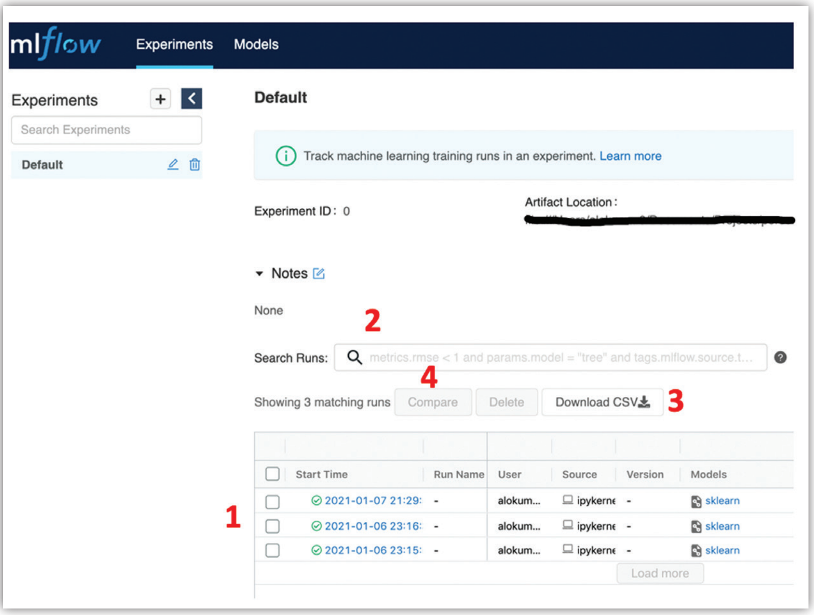
Figure 5.5: *MLflow library abstractions*

Another important aspect of reproducibility is to offer an abstraction over the models. Based on *figure 5.5*, mlflow.tensorflow.log_model directive will take the TensorFlow graph, persist it in an MLflow model format that can be saved in a central repository. The others can later download the model from the central repository and use it in their environment. To log the PyTorch model, the function call is similar. We will come back to abstractions in a bit.

For now, let us look at mlflow UI. To run the UI, type the command in your favorite terminal. Make sure that this command is run from the folder containing the mlruns directory:

```
mlflow ui
```

This will launch the Gunicorn server. Green Unicorn (https://gunicorn.org/) is a Python WSGI (web server gateway) to run Python web applications. Open http://localhost:5000 in your browser. *Figure 5.6* shows the default UI from my machine:



Track machine learning training runs in an experiment. Learn more

Experiment ID: 0

Artifact Location: [redacted]

Notes

None

Search Runs: metrics.rmse < 1 and params.model = "tree" and tags.mlflow.source.t...

Showing 3 matching runs

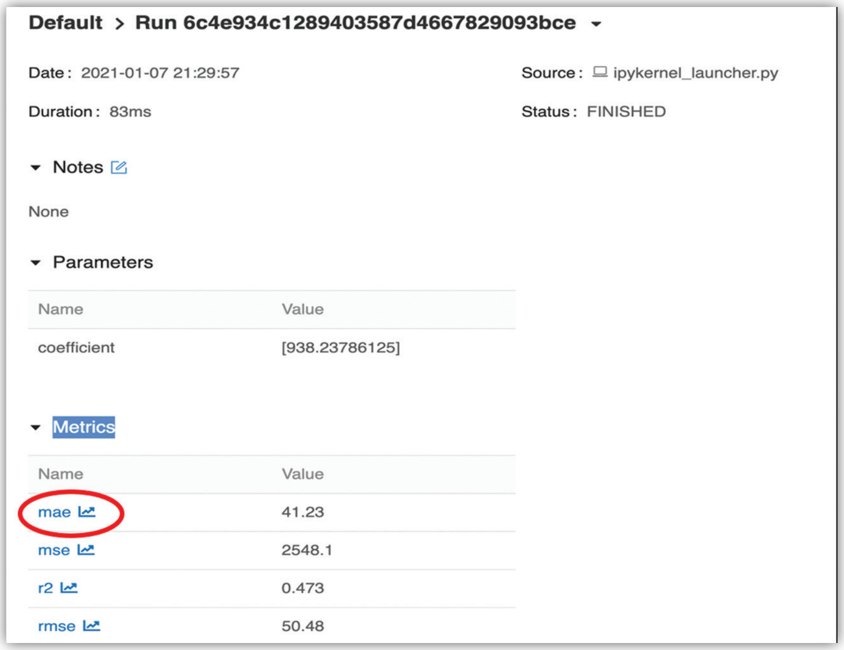
Start Time	Run Name	User	Source	Version	Models
2021-01-07 21:29:	-	alokum...	ipykerne	-	sklearn
2021-01-06 23:16:	-	alokum...	ipykerne	-	sklearn
2021-01-06 23:15:	-	alokum...	ipykerne	-	sklearn

Figure 5.6: *Default MLflow UI*

You are seeing three runs of Code 5.1. The UI lets you visualize, search, and compare runs.

At a high level, the UI offers the following key features:

1. Running the code.
2. Searching for runs by parameter or metric value.
3. Downloading run results.
4. To compare runs, select the run checkboxes and click on the compare button.
5. Visualizing run metrics. While you can write your visualizations using the mlflow API, it also offers its own visualization. Click on any of the runs as shown in *figure 5.7*:



Default > Run 6c4e934c1289403587d4667829093bce

Date : 2021-01-07 21:29:57

Source : ipykernel_launcher.py

Duration : 83ms

Status : FINISHED

Notes

None

Parameters

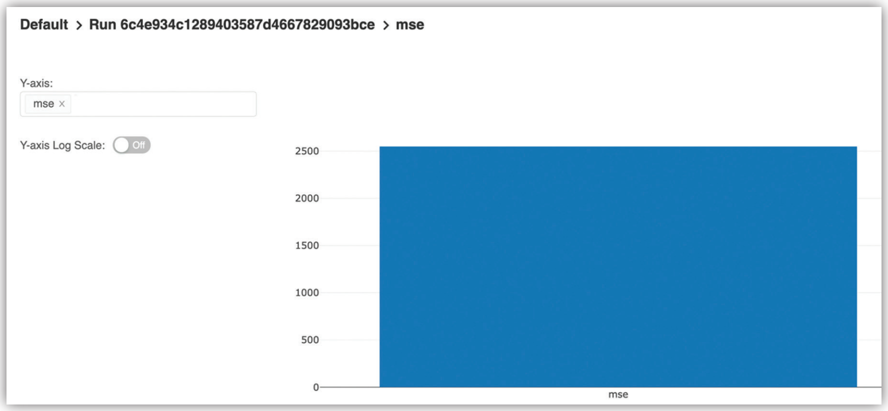
Name	Value
coefficient	[938.23786125]

Metrics

Name	Value
mae	41.23
mse	2548.1
r2	0.473
rmse	50.48

Figure 5.7: *Run detail page*

6. Click on any of the metrics and you land on the visualization page. The visualization is trivial because I didn't make any changes across different runs:



Default > Run 6c4e934c1289403587d4667829093bce > mse

Y-axis: mse

Y-axis Log Scale: Off

Figure 5.8: *MLflow in-built visualization*

Let us go through another example while only focusing on tracking:

```
import os

from random import random, randint
from mlflow import log_metric, log_param, log_artifacts


if __name__ == "__main__":


    # Log a parameter (key-value pair)

    log_param("param1", randint(0, 100))

    # Log a metric; metrics can be updated throughout the run

    log_metric("foo", random())

    log_metric("foo", random() + 1)

    log_metric("foo", random() + 2)


    # Log an artifact (output file)

    if not os.path.exists("outputs"):

       os.makedirs("outputs")


    with open("outputs/test.txt", "w") as f:

          f.write("hello world!")


   log_artifacts("outputs")
```

Code 5.2: *MLflow tracking API with artifacts*

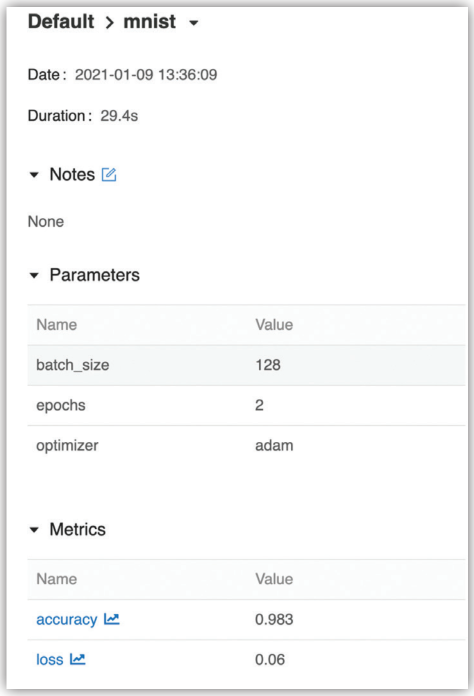
The preceding code shows how simple the tracking gets with mlflow. Along with the metrics, you have the artifacts available as well for reference.

What about deep learning networks? How the tracking would look like, let us say, a fully connected neural network. The following code shows a Keras-based fully connect neural network for the famous MNIST dataset. The network is trivial, and I would like you to pay attention to the mlflow tracking API:

```
   #--1

import numpy as np
```

```
from tensorflow import keras

from tensorflow.keras import layers

import mlflow


num_classes = 10 #--2

input_shape = (28, 28, 1) #--3


# the data, split between train and test sets

(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data() #--4


# Scale images to the [0, 1] range #--5

x_train = x_train.astype("float32") / 255

x_test = x_test.astype("float32") / 255


# Make sure images have shape (28, 28, 1)  #--6

x_train = np.expand_dims(x_train, -1)

x_test = np.expand_dims(x_test, -1)


print("x_train shape:", x_train.shape)

print(x_train.shape[0], "train samples")

print(x_test.shape[0], "test samples")


#--7

# convert class vectors to binary class matrices

y_train = keras.utils.to_categorical(y_train, num_classes)

y_test = keras.utils.to_categorical(y_test, num_classes)
```

```
#--8

model.compile(loss="categorical_crossentropy", optimizer="adam",
metrics=["accuracy"])


#--9

with mlflow.start_run(run_name = "mnist"):

    batch_size = 128

    epochs = 2

   #--10

   mlflow.log_param("batch_size",batch_
size)

    mlflow.log_param("epochs",epochs)

   mlflow.log_param("optimizer","adam")


 #--11

     model.fit(x_train, y_train, batch_
size=batch_size,        epochs=epochs,
validation_split=0.1)

     score = model.evaluate(x_test, y_
test, verbose=0)


    test_loss = score[0]

    test_accuracy = score[1]


 #--12

    mlflow.log_metric("loss", test_loss)

    mlflow.log_metric("accuracy", test_accuracy)
```


Default > mnist

Date : 2021-01-09 13:36:09

Duration : 29.4s

Notes

None

Parameters

Name	Value
batch_size	128
epochs	2
optimizer	adam

Metrics

Name	Value
accuracy	0.983
loss	0.06

Figure 5.9: Keras run detail

```
    print("Test loss:", test_loss)

    print("Test accuracy:", test_accuracy)
```

***Code 5.2:** MLflow tracking API for Keras-based fully connected network*

Let us unpack the code as follows:

1. Standard Keras modules are imported. We also imported mlflow. Note that I used keras/tensorflow 2.
2. We will train a classifier for the MNIST dataset, and it has 10 classes. Each image is 28×28 with a single channel.
3. Let us split the data between train and test.
4. Make sure that the input images have the size 28×28×1.
5. We convert class vectors to binary class matrixes.
6. Compile the model.
7. Initialize the mlflow run by calling start_run. You have seen this before and it returns a context_manager.
8. We log the parameter for tracking using the log_param method. Again, you are already familiar with the function.
9. We fit the model and evaluate on the test dataset.
10. Eventually, we log the "metrics" – loss and accuracy.

If you have experience with Keras, then you would notice that there is no change other than the mlflow tracking calls. It would be no surprise that the same pattern is applicable to all the supported deep learning framework. *Figure 5.9* shows the Keras run detail on mlflow UI. As an exercise, try different parameters and then compare the runs on mlflow UI.

Backend stores

The last topic that you need to be familiar with in tracking are backend stores. The tracking services of mlflow consists of two components.

The first part is an entity or metadata store that is responsible for collecting and aggregating all the lightweight metadata associated with a training session. The parameters like epoch, batch size, optimizer etc. fall in this category. The backend could be your file system or SQL databases like SQLLite, MySQL, and Postgres.

The second part of the backend is responsible for storing heavy artifacts like model, training data, or files etc. The following options are available for the heavy artifacts. Refer to this documentation: (https://mlflow.org/docs/latest/tracking.html#mlflow-tracking-servers) for details on each of the options:

- Amazon S3
- Azure Blob storage
- HDFS
- Google cloud storage
- Databricks' file system
- FTP
- SFTP
- NFS

5.2.2 MLflow projects

An MLflow project at a high level is a format to package data science code in a reusable and reproducible way. At the core, the format is a convention for organizing and describing the code so that the automated tools or data scientists can understand and run it. The purpose of convention should be clear now since repeatability can be achieved with standardization only.

Let us look at an example to understand it more.

Imagine that your organization has multiple ML projects going on. The teams are not only using different toolsets like TensorFlow, PyTorch, etc. but also different environments like private cloud, public cloud, local machine, etc. This leads to the challenge of ensuring consistency and reproducibility. How would you ensure that local notebook code works the same way in the cloud as well? The underlying problem is to ensure that all the dependencies, test, and trained data are packaged in a consistent way so that it can run seamlessly in different environments. The idea is to ensure the same state and context across the board. If you are coming from a traditional software background, then you know the desire and the benefit of writing once and running everywhere. To achieve this, a self-contained packaging format is required that can be used across different environments. An MLflow project provides that self-contained format listing all the dependencies, data requirements, etc. Another way to think about it is to imagine it as a self-contained execution that bundles the following:

1. All the machine code that is required to train the model.
2. All the dependencies that are required to run the code - the libraries, the languages, and the different configurations.

3. All the training and validation data
4. The config specification.

If you have used conda YAML file or PIP requirement.txt file, then you understand the value of a specification file. *Figure 5.10* shows the idea visually:

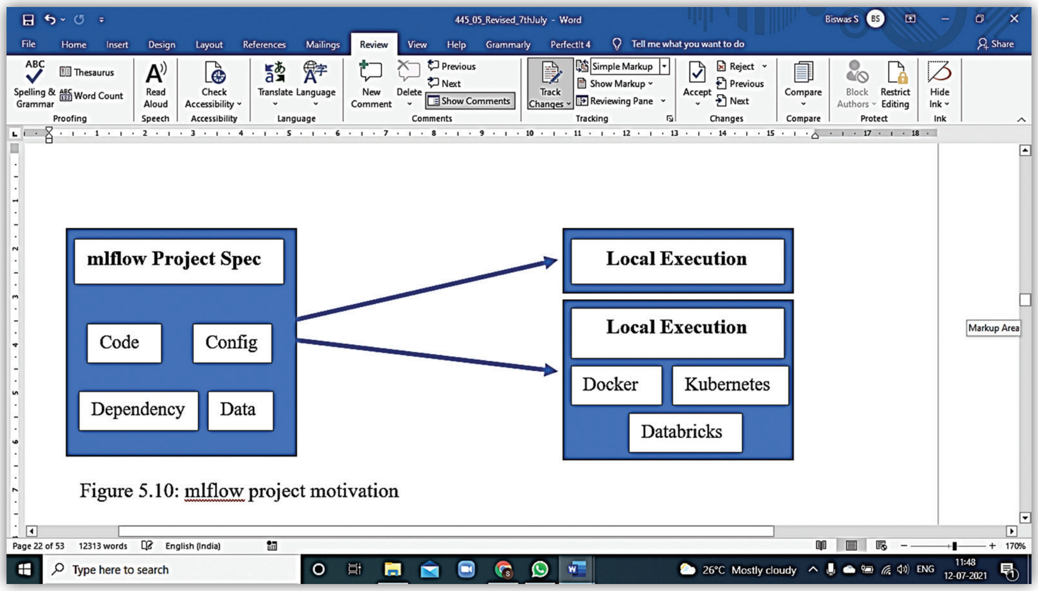


***Figure 5.10:** MLflow project motivation*

So, what does an MLflow project look like? Under the hood, it is simply a directory that contains optional configuration file, training code (it could be a git repository as well), library dependency specification, and other data required by the training session. You can specify the library dependencies in multiple ways. For example, if you use Anaconda environment, then you can include a YAML-formatted Anaconda environment specification to enumerate the library dependencies. They can also include a Docker container and MLflow will execute that training code within the specified container. Another option is to use an MLproject file that is also a YAML file but allows you to add more details of the project. We will discuss MLproject in detail later. To run the MLflow projects, there is an MLflow CLI as well.

Let us now look inside an MLflow project. The following structure shows the folder structure of a simple MLflow project:

```
myProject/
├──MLproject
├──conda.yaml
├──main.py
└──model.py
```

The following code shows the content of an ML project:

```
conda_env: conda.yaml  #--1


entry_points:  #--2
  main:
    parameters:
      lambda: {type: float, default: 0.1}
    command: "python main.py {lambda}"
```

Code 5.3: *MLproject configuration*

MLproject is a specification/convention for mlflow to recreate an experiment in a target environment. Let us unpack the entries in the file as follows:

1. The conda file specified via conda_env lists down the dependencies required by the code. Conda environments support both Python packages and native libraries (for example, CuDNN). If a conda environment is specified, then it needs to be activated before the project runs.

   By default, MLflow will use the system path to find and run the conda binary. You can use a different conda installation by setting the MLFLOW_CONDA_HOME environment variable. If the environment variable is present, then MLflow attempts to run the binary mentioned at $MLFLOW_CONDA_HOME/bin/conda.

   It is possible to use Docker files in place of conda environment, and for that, you need to add a top level entry called docker_env in the MLproject file. The value of this entry would be the location of the Dockerfile. It is important to remember that the Dockerfile must be accessible on the local machine:

   ```
   docker_env:
     image: mlflow-docker-example-environment
     volumes: ["/local/path:/container/mount/path"]
     environment: [["NEW_ENV_VAR", "new_var_value"],
   ```

```
"VAR_TO_COPY_FROM_HOST_ENVIRONMENT"]
```

Code 5.4: *Docker entry in MLproject*

Only the image key is required, and it contains the Docker file with the default tag as latest. The rest of the keys like volumes, environment is only required if you want to mount something in the Docker and set up specific environment variables. As an exercise, can you read the documentation to figure out how to specify remote images?

2. entry_points specifies the commands that can be run within the project along with the parameter details. Most projects contain at least one entry point that you would like other users to call. MLproject allows to specify multiple entry points. For example, you might have a single Git repository containing multiple featurization algorithms. You can also call any .py or .sh file in the project as an entry point. If required, you can specify parameters for the entry points via the parameters entry. The "lambda" is a random parameter of float type (It could have been a learning rate). You could have guessed it but it is always better to be explicit – parameters will be optional; they can have a default value and could be multiple parameters. Finally, the command that would train the model. So, for anyone else, all that they need is to run the command to run the training.

   Let us explore the command a bit more. When specifying an entry point in an MLproject file, the command can be any string in Python format string syntax (https://docs.python.org/2/library/string.html#formatstrings). All the parameters declared in the entry point's parameter fields are passed into this string for substitution. Even if you call the project with additional parameters that are not listed in the parameters field, MLflow passes them using key value syntax. This means that you declare types and defaults for just a subset of your parameters. While this is cool, it should be avoided. Being explicit always helps in programming. Code 5.4 shows the short form of specifying the parameters but mlflow supports the long format as well. Which one you should go for is less of a good practice question and more of a style question. So, how do you know what parameter types are supported? *Table 5.16* shows the list of supported data types as follows:

Type	Description
string	A text string
float	A real number. MLflow validates that the parameter is a number.

path	This specifies the local file path on the system. MLflow will convert it into an absolute path. If a distributed filesystem path is specified, then it is downloaded locally. You use this parameter when the code needs the data locally.
uri	This is applicable for distributed file systems or storage. Think of your training code reading data from Spark.

Table 5.2: *MLproject Entry_point parameter data type*

The only remaining piece of the puzzle is how to execute the commands. You can use MLflows UI, CLI, or as API to execute the project. For now, we will focus on the CLI and the API mode. But before that, let us put in place the training code (main.py) so that you can run the project. The following code has the main.py code. It is a dummy code, but you get the idea:

```
import mlflow

import warnings

import sys


if __name__ == '__main__':

    # suppress any deprecated warnings

   # warnings.filterwarnings("ignore", category=DeprecationWarning)


    mY_lambda = float(sys.argv[1])


    parameters = [{'alpha': 0.3, 'lambda':mY_lambda},

                 {'alpha': 0.4, 'lambda':mY_lambda},

                 {'alpha': 0.5, 'lambda':mY_lambda}]


    # Iterate over three different runs with different parameters

    for param in parameters:

      print(f"Running with param = {param}")
```

Code 5.5: *Dummy main.py code*

MLflow provides two ways to run projects: the mlflow run command-line tool or the mlflow.projects.run() Python API. The following code shows a few examples calling the run:

```
mlflow run . -e main -P lambda=0.2 #--1


mlflow run git://<YOUR_GIT_PROJECT> -P lambda=0.2  #--2


#--3
mlflow.run("git://<YOUR_GIT_PROJECT>", parameters = {..})
```

Code 5.6: *Different ways to run the command.*

1, 2, and 3 are different ways to execute the command.

3. This CLI command runs the code when it is available locally. You specify the parameter if you want to use a value different from the default value. Recall that while defining the parameter, you can specify the default value. If not noticed, the command can be run without parameters, and in that case, default values will be used. If you check the run entry in the UI, you would see the "lambda" parameter automatically tracked.
4. The second is similar to the first except that it accepts your Git project link.
5. The third option is the API to call the run. I don't think I need to explain the parameters. However, the code would look as follows:

```
ml_project_uri = "<YOUR GIT PROJECT>"
parameters = [{'alpha': 0.3},
                 {'alpha': 0.4},
                 {'alpha': 0.5}]


# Iterate over three different runs with different parameters
for param in parameters:


      print(f"Running with param = {param}"),

      res_sub = mlflow.run(ml_project_uri, parameters=param)

      print(f"status={res_sub.get_status()}")

      print(f"run_id={res_sub.run_id}")
```

Code 5.7: *Run API example*

There are quite a few options that can be specified with the run command and I encourage you check the documentation for the same (https://mlflow.org/docs/latest/cli.html#cli). From time to time, I have asked you to read the documentation, and there is a reason for it. In order to learn a framework effectively, it is important to be comfortable with the documentation structure as you will be using that a lot for your serious work. Can you imagine using something without knowing it?

6. Next is the conda.yaml file. If you are familiar with Python anaconda, then you fully understand the purpose of it. The PIP equivalent of this is requirement.txt. It contains all the libraries dependencies for your code. In case you are not comfortable with conda yaml or pip requirement.txt then think of this way. To run the project, you need Python and mlflow. Where would you specify these dependencies. Again, it is worth repeating that this file or the environment would be run before the ML code is run.
7. The following code shows the conda file of our example:

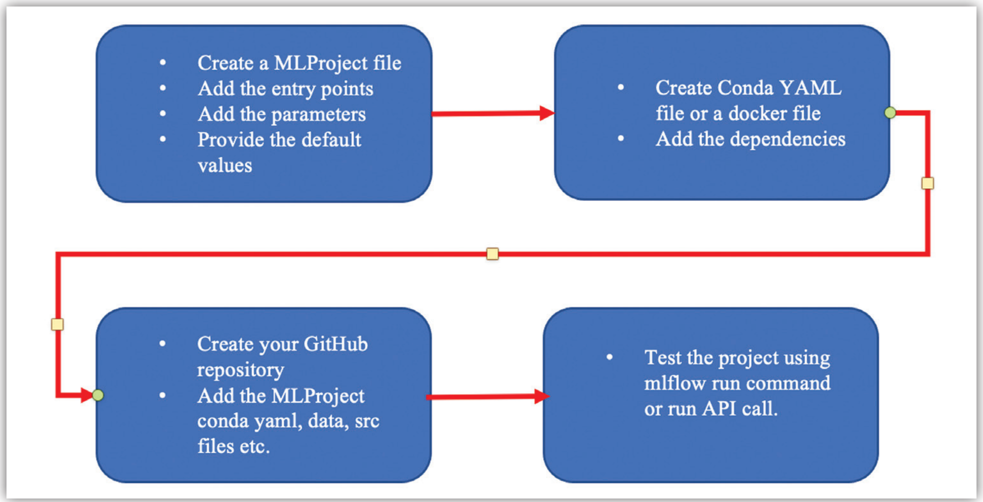
```
name: mlflow-env


channels:

  - defaults


dependencies:

  - pip=20.2.4

  - python=3.7

  - scikit-learn=0.20.3

  - pip:

    - mlflow
```

Code 5.8: *Conda YAML file containing package dependencies*

The channel specifies the location from where the packages will be installed. "dependencies" list the Python packages to be installed. The "pip" lists what will be installed via the PIP command.

So that you are not lost in the details, let us recap what we covered:

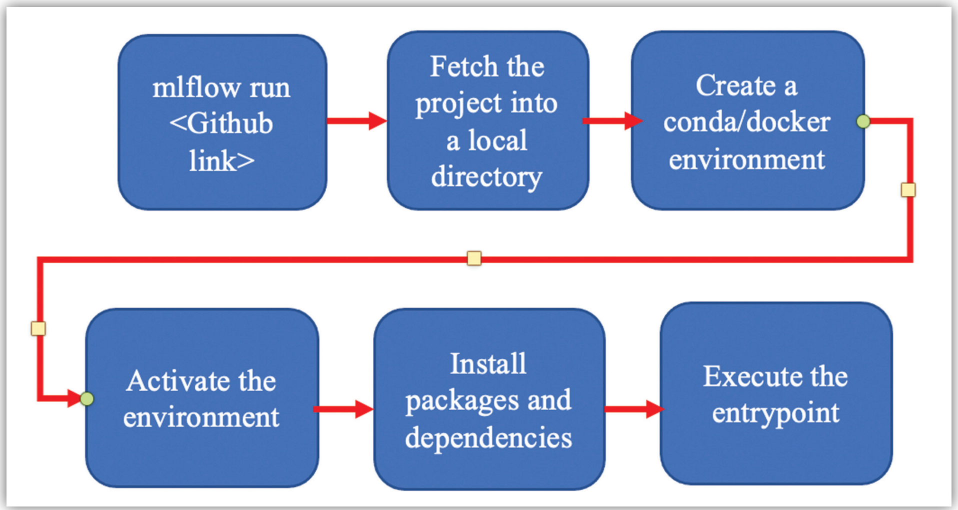
- The steps to build an MLproject as shown in *figure 5.11*:



• Create a MLProject file
• Add the entry points
• Add the parameters
• Provide the default values
• Create Conda YAML file or a docker file
• Add the dependencies
• Create your GitHub repository
• Add the MLProject conda yaml, data, src files etc.
• Test the project using mlflow run command or run API call.

Figure 5.11: *Steps to build an MLproject*

- Execution lifecycle of an MLproject as shown in *figure 5.12*:



mlflow run <Github link>
Fetch the project into a local directory
Create a conda/docker environment
Activate the environment
Install packages and dependencies
Execute the entrypoint

Figure 5.12: *MLProject execution model*

5.2.3 Multistep workflow

By combining the MLproject and the tracking ideas, mlflow allows you to build multi-step workflows with different projects as separate entry points. Each call to mlflow.projects.run() would return a run object that can then be used with mlflow.tracking to determine when the run has ended and get its output artifacts. The artifacts produced at a previous step can then be passed into the next step that takes path or uri parameters. All the steps can coordinate in a single Python program that looks at the results of each step and decides what to submit next using custom code.

It is worthwhile to be aware of what becomes possible with multistep workflows:

1. **Code modularization** – The steps code becomes reusable for other workflows. Think of steps for feature engineering, data validation, etc. Combine this with the idea of versioning and each step could be versioned and locked offering a scalable governance model.
2. **Hyperparameter tunning –** Different runs can now be launched in parallel.
3. **Cross validation** – The same training code can now run different splits of training and validation datasets.

Let us build a multistep workflow. It will have two dummy steps: download and train. They represent the typical step of downloading the data and then using it for training. The output of download will be passed to the training step. The project structure would look like the following code. It would be good to create a separate folder for it:

```
├── MLproject
├── conda.yaml
├── download_data.py
├── main.py
└── train_data.py
```

Let us go through each of the files as follows:

```
MLproject

name: mlflow-env


channels:

  - defaults


dependencies:

  - pip=20.2.4

  - python=3.7
  - scikit-learn=0.20.3

  - pip:

    - mlflow
```

Code 5.10: *Multistep workflow MLproject file*

You already understand an MLproject as follows:

1. Conda entry specifies the package dependencies.
2. It has three entry points – download_data, train_data and main. Do you remember how a specific entry point can be run? If you don't specify any entry point, then the main entry point will be run by default

   ```
   mlfow run .  -e download_data
   ```
3. download_data, as the name suggests, downloads the data. It will be a dummy operation for us. We will look inside download_data shortly.
4. train_data, as the name suggests, trains the data. It could be your TensorFlow, PyTorch, or whichever framework you like. The only parameter it accepts is the downloaded file.
5. The main entrypoint is the aggregator and calls the other entrypoints. When you look at the code, it would become clear. For now, it accepts two parameters. The parameters have no significance training wise but I have added them to show you the complete flow.

   ```
   Conda_env

   name: multistep

   channels:

     - defaults

     - anaconda

     - conda-forge

   dependencies:

     - python=3.7

     - pip=20.2.4

     - pip:

       - tensorflow==2.3.1
   ```

```
          - mlflow==1.13

          - click
```

Code 5.11: Multistep workflow conda environment specification

Not much explanation is required because all we are doing is specifying the packages that must be installed beforehand. In case you don't know about the click Python module, then check them out. It really simplifies how you handle the command line parameters inside the Python code.

download_data entrypoint

The download_data calls the download_data Python file:

```
"""

Dummy python application mocking data download operation

"""

import os

import mlflow


def download_data():

    with mlflow.start_run() as mlrun:


        print("Downloading Data\n")


        #Create a features.txt artifact file

       features = "rooms, zipcode, median_price, school_rating, transport"


        with open("features.txt", 'w') as f:

            f.write(features)


        mlflow.log_artifact("features.txt","data")
```

```
if __name__ == "__main__":

    download_data()
```

Code 5.12: *Download data task Python file*

Again, you have seen all the code before. We are creating a feature.txt as a representation for data download. Once the data is downloaded, it is saved in the artifact store. You may recall that log_artifact logs a local file or directory as an artifact of the currently active run. If no run is active, this method will create a new active run.

`train_data entrypoint`

This entry point calls the train_data Python file and it accepts the file path as the parameter:

```
import os

import mlflow

import click


@click.command()

@click.option("--data-path", default="features.txt")

def train_data(data_path):


    with mlflow.start_run() as mlrun:


        print("training features = : %s" % data_path)


        # As a dummy transformation, let us rename the txt file

        artifact_uri = mlflow.get_artifact_uri(artifact_path=data_path)


       print("Artifact uri for training data : {}".format(artifact_uri))

      # Write your training code here
```

```
if __name__ == "__main__":

    train_data()
```

Code 5.13: *Training task Python file*

The only new thing in the preceding code is how it is retrieving the artifacts using the mlflow.get_artifact_uri. This method retrieves the absolute URI of the specified artifact in the currently active run. If the path is not specified, the artifact root URI of the currently active run will be returned. What you log via log_artifact gets retrieved via get_artifact_uri.

`main entrypoint`

This maps to main.py and is responsible for orchestrating the download and training tasks as follows:

```
import os

import sys

import click


import mlflow


@click.command()

@click.option("--max-iter", default=10, type=int)

@click.option("--keras-units", default=20, type=int)

def workflow(max_iter, keras_units):


    with mlflow.start_run() as active_run:

        #print(f"Parameters are max_iter={max_iter} and keras_units={keras_
        units}")


        print("Launching 'download'\n")

        download_run = mlflow.run(".", "download_data", parameters={})
        #--1

        download_run = mlflow.tracking.MlflowClient().get_run(download_run.
        run_id) #--2

        file_path_uri    =    os.path.join(download_run.info.artifact_uri,
        "data")

        print(f"The downloaded file URI is {file_path_uri}") #--3


        print("Launching 'train'\n")

        train_run = mlflow.run(".", "train_data", parameters={"data_path":
        file_path_uri}) #--4

      train_run = mlflow.tracking.MlflowClient().get_run(train_run.run_id)

        data_path_uri    =    os.path.join(download_run.info.artifact_uri,
        "data","features.txt")


        print(f"Training the model with - {data_path_uri}")


if __name__ == "__main__":

    workflow()
```

Code 5.14: *Tasks pipeline orchestration*

This maps to main.py and is responsible for orchestrating the download and training tasks. You have already seen the majority of the code. mlflow.run is an API equivalent for the command line run command.

1. The downloaded task doesn't accept any parameter and hence it is empty. The run method returns an object that holds information about the current run and can be used to store artifacts. This way, when the next call to run that executes the next task or entrypoint is made, you get access to the previously stored artifacts.
2. mlflow.tracking.MlflowClient.get_runs fetches the run from backend store. The returned "Run" contains a collection of run metadata in the form of RunInfo along with RunData that contains run parameters, like tags, and metrics etc.
3. From the Info object, the artifact URI is retrieved. This URI points to the

artifact stored at the previous step. This is how you access the data from a previous step.

4. The URI is then passed to the next step as a parameter. There are two ways to run the whole workflow. They are as follows:
   a. mlflow run: This looks at the MLproject file and picks up the default entrypoint which is the main.
   b. mlflow run: -e main -P max_iter=30 -P keras_units=30 This is for if you want to be explicit with the entry point.

This completes the exploration of the MLproject. Again, to reiterate, the whole idea and purpose of MLproject is to offer a standard way of how projects are shared so that you don't have to make the statement ”It is working on my machine."

5.2.4 MLflow models

MLflow models like MLproject offer an abstraction over the model. MLflow models would seem quite like MLproject in terms of ideas and structure. With your understanding of MLproject, understanding MLflow models would be easier. To understand the motivation, let us imagine a situation where the organizations have different ML teams with their own framework choices. The first challenge here is how to enable the teams to share the models easily. A lot of times, the same model could be used in real time or batch mode; for example, real time as an API and in batch mode in Spark. The challenge here is to use the same model with different serving tools. One of the solutions that I have seen is teams creating one to one pipeline between models and environments. This may work initially but IT quickly goes out of control. *Figure 5.13* illustrates the problem visually:

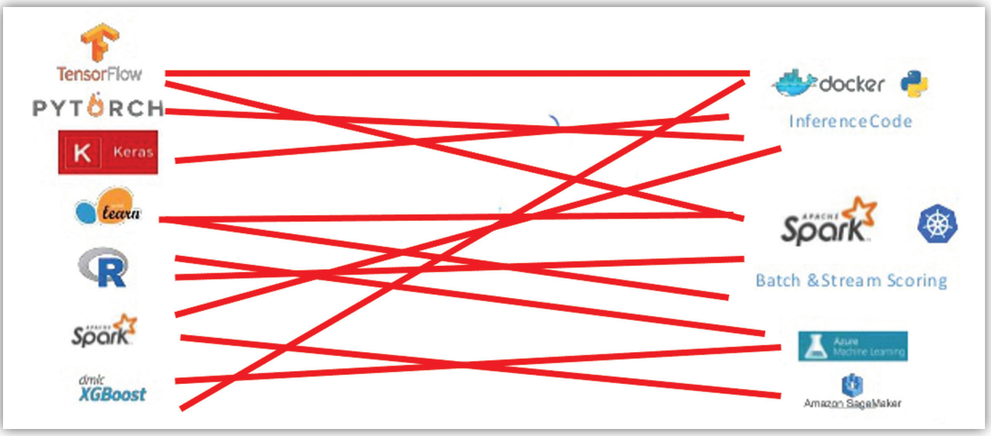


Figure 5.13: *Brittle pipelines between models and production environments*

What we need is something that enables one to many connections between models and serving tools. By introducing an abstraction in between, it is possible to allow to seamless one to many connections or the same model but different serving tools. *Figure 5.14* demonstrates this idea visually. With an abstraction in between, it is now possible to add new tools on both sides:

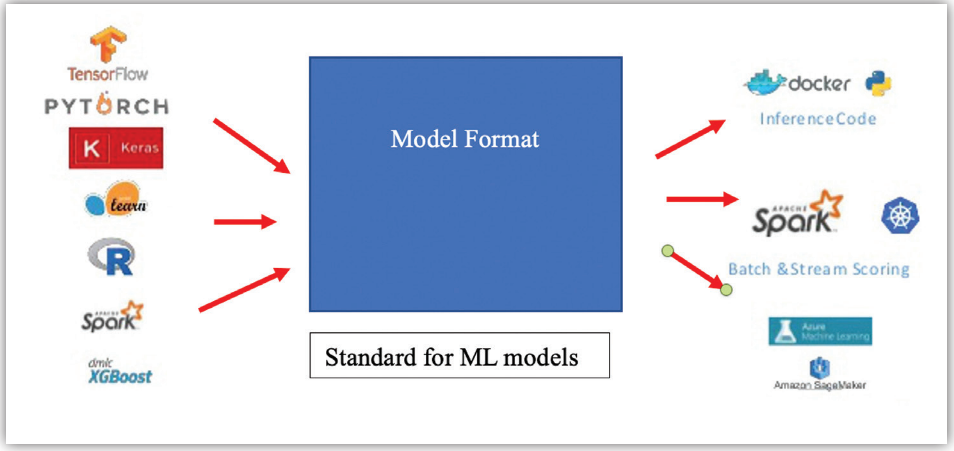


***Figure 5.14:** Model abstraction for N×M combinations*

Think of the abstraction as a Docker file for model that offers a unified way to access the code or functionality inside. What we need is something that enables one-to-many connections between models and serving tools. By introducing an abstraction in between, it is possible to allow to seamless one to many connections or the same model but different serving tools, as demonstrated by *figure 5.15*. Another analogy could be to think of Apache arrow project (https://arrow.apache.org/). At a high level, it offers a language-agnostic in-memory data structure:

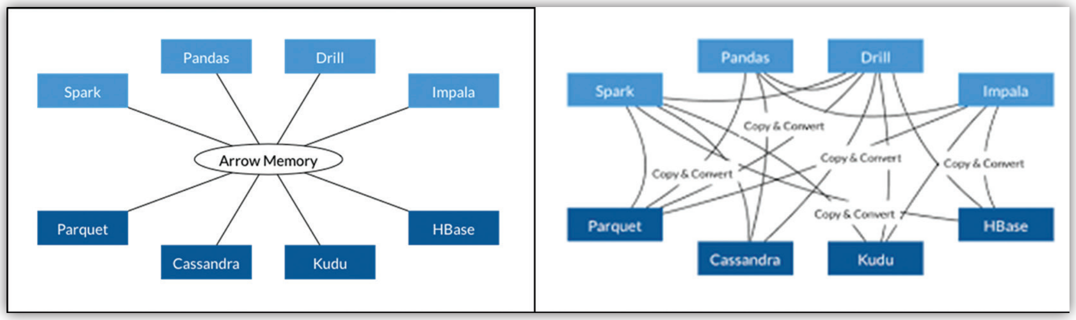


***Figure 5.15:** Before and with Arrow comparison*

You can clearly see the difference and the benefits with abstraction (right-hand side is with Arrow). There are three following benefits that are worth highlighting:

- All the connecting systems use the same memory format.
- No communication overhead.
- Projects can share with each other.

Note: I'm not saying that the Arrow and MLflow model format are the same but rather used the analogy to explanation the idea and motivation.

So how does the MLflow model look like? It is a directory containing a bunch of configuration files (Does this ring a bell about MLProject?) and serialized model artifact. The following list shows high level elements:

- Packaging format for ML formats
    - o Any directory with MLmodel (like MLProject) file
- Dependencies for reproducibility.
    - o Conda environment for dependent packages
- Model creation utility for serializing models.
    - o Save models from any framework in MLflow format.
- Deployment API
    - o CLI/Python/ R/Python

We will explore all the elements in detail. Let us first train an estimator and for the sake of simplicity, we will stick to sklearn LinearRegression:

```
from pprint import pprint


import numpy as np

from sklearn.linear_model import LinearRegression


import mlflow


def yield_artifacts(run_id, path=None):  #--1


    client = mlflow.tracking.MlflowClient()

    for item in client.list_artifacts(run_id, path):

        if item.is_dir:

            yield from yield_artifacts(run_id, item.path)

        else:
            yield item.path

def fetch_logged_data(run_id):  #--2

    client = mlflow.tracking.MlflowClient()
    data = client.get_run(run_id).data

#--3
   tags = {k: v for k, v in data.tags.items() if not k.startswith("mlflow.")}
    artifacts = list(yield_artifacts(run_id))
    return {
        "params": data.params,
        "metrics": data.metrics,
        "tags": tags,
        "artifacts": artifacts,
    }

def main():
    # enable autologging
    mlflow.sklearn.autolog()  #--4

    # prepare training data

    X = np.linspace((1,2),(10,20),10)
    y = np.dot(X, np.array([1, 2])) + 3

    # train a model
    model = LinearRegression()
    with mlflow.start_run() as run:
        model.fit(X, y)
        print("Logged data and model in run {}".format(run.info.run_id))
    # show logged data
    for key, data in fetch_logged_data(run.info.run_id).items():
        print("\n---------- logged {} ----------".format(key))
        pprint(data)


if __name__ == "__main__":
    main()
```

Code 5.15: *LR model with tracking*

Let us Zoom in on the MLflow specific parts:

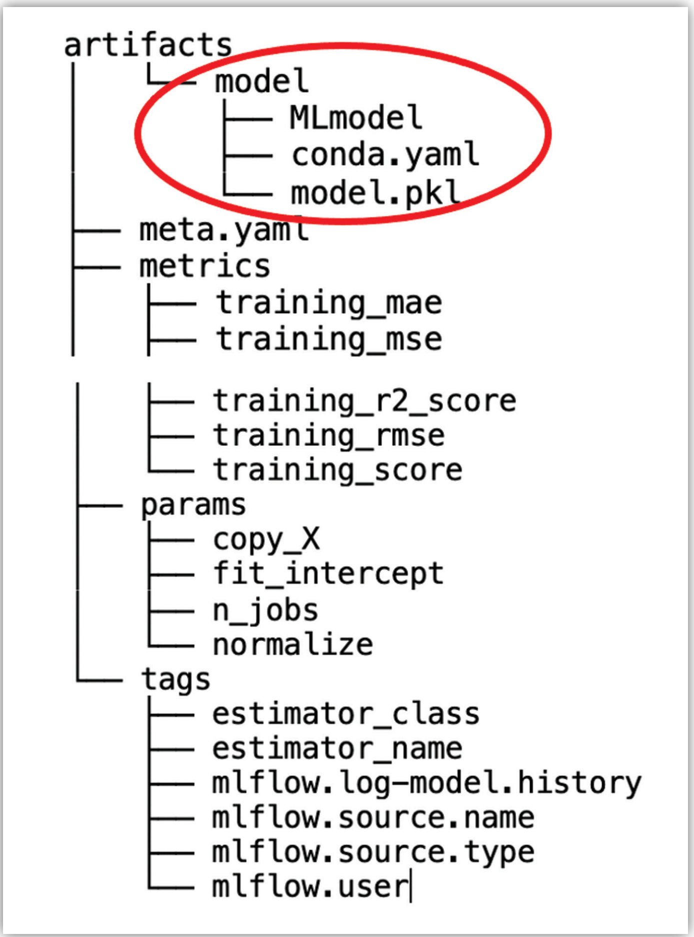
- yield_artifacts is a helper function that yields all the artifacts from a specified run.
- fetch_logged_data fetches params, metrics, tags, and artifacts in the specified run.
- We don't want the system tags and hence they have been excluded.
- We call the autolog before the training code to enable automatic logging of sklearn metrics, params, and models. It is because of auto logging that you don't see any explicit logging.

When you run the previous code, you would get the following code as output:

```
#---------- logged params ----------
{'copy_X': 'True',
 'fit_intercept': 'True',
 'n_jobs': 'None',
 'normalize': 'False'}


#---------- logged metrics ----------
{'training_mae': 3.730349362740526e-15,
 'training_mse': 2.4296915880807164e-29,
 'training_r2_score': 1.0,
 'training_rmse': 4.929190185091986e-15,
 'training_score': 1.0}


#---------- logged tags ----------
{'estimator_class': 'sklearn.linear_model.base.LinearRegression',
 'estimator_name': 'LinearRegression'}


#---------- logged artifacts ----------
['model/MLmodel', 'model/conda.yaml', 'model/model.pkl']
```

Code 5.16: *Code 5.15 output*

If you look under the mlruns folder under the appropriate <<run_id>>, (If there are many runs, then delete the mlruns folder and rerun the preceding code), you'll see a tree like *figure 5.16*:



```
artifacts
│      └── model
│          ├── MLmodel
│          ├── conda.yaml
│          └── model.pkl
├── meta.yaml
├── metrics
│      ├── training_mae
│      ├── training_mse

│      ├── training_r2_score
│      ├── training_rmse
│      └── training_score
├── params
│      ├── copy_X
│      ├── fit_intercept
│      ├── n_jobs
│      └── normalize
└── tags
       ├── estimator_class
       ├── estimator_name
       ├── mlflow.log-model.history
       ├── mlflow.source.name
       ├── mlflow.source.type
       └── mlflow.user
```

Figure 5.16: *Code 5.16 output*

For MLmodel discussion, we are interested in the model folder. Had there been no auto logging, this folder would have been written by a call to the mlflow.sklearn.save_model method.

If you open the MLmodel, it will look like the following code:

```
artifact_path: model


flavors:

  python_function:

    env: conda.yaml

    loader_module: mlflow.sklearn

    model_path: model.pkl

    python_version: 3.7.3

  sklearn:

    pickled_model: model.pkl

    serialization_format: cloudpickle

    sklearn_version: 0.21.2


run_id: 645d3a3b55134d28af27d731643cd5a4


signature:

  inputs: '[{"type": "double"}, {"type": "double"}]'

  outputs: '[{"type": "double"}]'


utc_time_created: '2021-01-16 09:33:52.265697'
```

Code 5.17: *MLmodel file*

MLmodel is the configuration file for the model. The purpose of MLmodel is very much like MLproject that you saw earlier. Let us unpack other attributes as follows:

- **`time_created`:** Date and time when the model was created. It is the UTC ISO 8601 format.
- **`Run_id`:** ID of the run that created the model using MLflow tracking.
- **Signature:** Model signature defines the model input and output schema in a

JSON format. The model signature is from code 5.17 and is shown below. As a reminder, this model was saved with code 5.15:

```
signature:
  inputs: '[{"name":"x1", type": "double"}, {"name":"x2", "type": "double"}]'
  outputs: '[{"type": "double"}]'
```

Why are there two inputs and a single output? If you recall, we build an estimator using two columns (feature). "X" and "y" shape and value defined the model signature. The signatures are extremely helpful for parameters validations. Imagine that you are building a REST endpoint and you would like to validate the incoming parameters.

To add the signature along with the model, you need to pass a signature object to the log_model method. The signature could be inferred from the input parameters or explicitly crafted. I recommend using the explicit coding of signature as follows to avoid any hidden bugs later:

```
from mlflow.models.signature import ModelSignature
from mlflow.types.schema import Schema, ColSpec


input_schema = Schema([
  ColSpec("double", "x1"),
  ColSpec("double", "x2")
])
output_schema = Schema([ColSpec("long")])
signature = ModelSignature(inputs=input_schema, outputs=output_
schema)
```

Code 5.18: *Model signature creation and initialization*

ColSpec is an object specifying the name and type of the column in the dataset. Now, when the log_model is called, you pass the signature object. Note that signature is passed along by default in the autolog method:

```
mlflow.sklearn.log_model(..., input_example=input_example, signature = signature)
```

Along with the signature, it would be extremely helpful if we can provide some sample inputs. Examples are a great documentation tool to understand the correct parameters usage.

A model input example provides an instance of a valid model input. This may be a single record or a batch of records that are stored with the model as separate artifacts.

Using the log_model method as an example input can be added along with the model. The following code shows the logging for our model in code 5.15:

```
input_example   = {

"x1" : 5.0,

"x2" : 25.0

}

mlflow.sklearn.log_model(..., input_example=input_example)
```

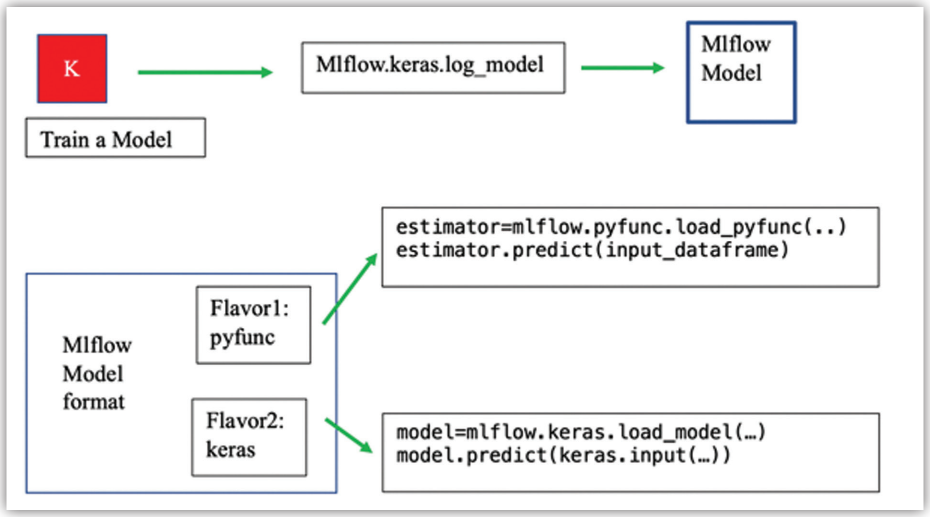
Code 5.19: *Logging an input example*

The sample input could be passed in as Pandas DataFrame, Numpy array, list, or dictionary.

- **Flavors:** A flavor is language and tools specific representation of a ML model. It is a very important concept that makes the MLmodel powerful in MLFlow. Think of it as a convention that helps different serving or production tools understand the model and use it without having to integrate with the underlying model library. There are certain standard flavors that all the built-in tools support. python_function is one such standard flavor that allows the model inference to be treated as a normal Python function. The *figure 5.18* model is offering two flavors – python_function and sklearn:

```
flavors:

  python_function:

    env: conda.yaml

    loader_module: mlflow.sklearn

    model_path: model.pkl

    python_version: 3.7.3

  sklearn:

    pickled_model: model.pkl

    serialization_format: cloudpickle

    sklearn_version: 0.21.2
```

This model is ready to use by a tool supporting sklearn or Python. An example would be helpful here. Let us take a look at the visual first as follows:



```
estimator=mlflow.pyfunc.load_pyfunc(..)
estimator.predict(input_dataframe)
```

```
model=mlflow.keras.load_model(…)
model.predict(keras.input(…))
```

Figure 5.17: *MLflow model workflow*

In *figure 5.18*, you first train the Keras model and then track it using the log_model method. This adds two flavors for the model – the Python function flavor abbreviated as pyfunc and a Keras model.

Using the two flavors is straight-forward. By using the mlflow.pyfunc.load_pyfunc method, the model is treated like a normal Python function that during evaluation, accepts and returns a pandas DataFrame. No knowledge of Keras is required to use the model. It simply abstracts away all the Keras details. The signature of the predict function is shown as follows:

```
predict(model_input: pandas.DataFrame) -> [numpy.ndarray | pandas.(Series
| DataFrame)]
```

In case you need the Keras object, then you load the Keras specific flavor via the load_model method. The two flavors are essentially different level of abstractions that you can use based on your use case.

It is important to be clear on the power of pyfunc abstraction. A model when treated as a Python function can be used by any of the deployment tools supporting Python. This interoperability is very powerful because it allows any Python model to be productionized in a variety of environments.

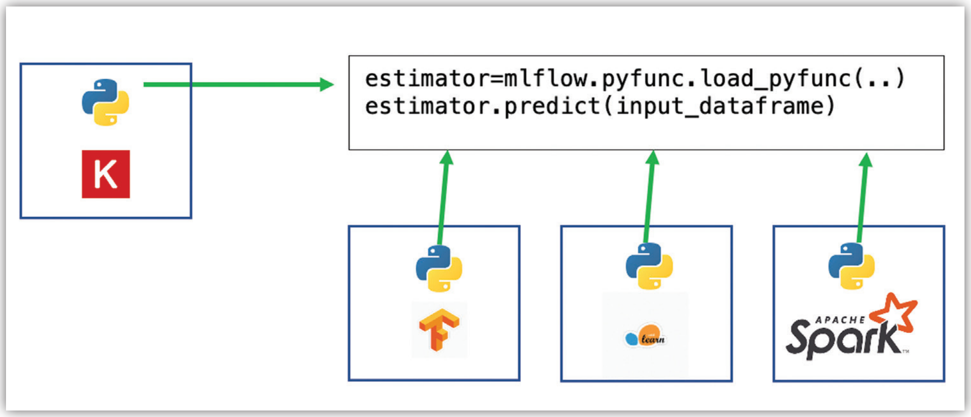
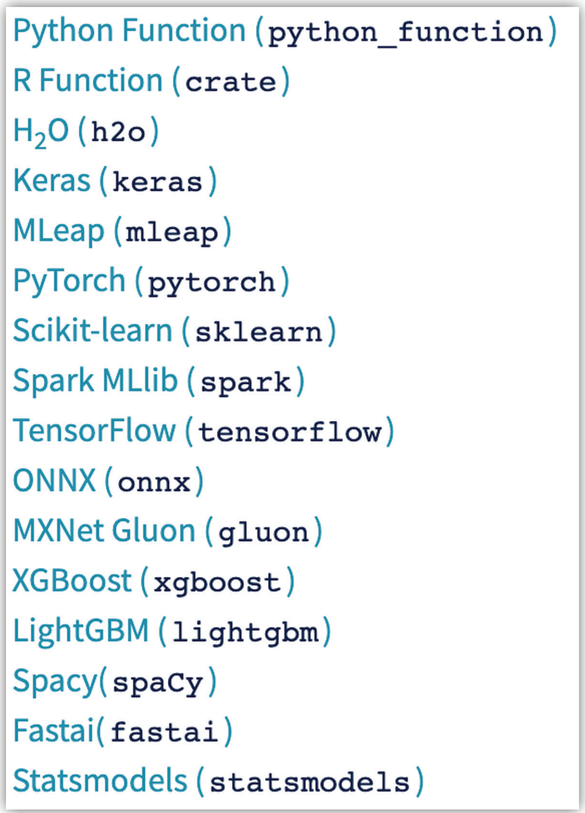

```
estimator=mlflow.pyfunc.load_pyfunc(..)
estimator.predict(input_dataframe)
```

Figure 5.18: *Model interoperability*

Figure 5.19 shows the list of built-in flavors:



Python Function (`python_function`)
R Function (`crate`)
H_2O (`h2o`)
Keras (`keras`)
MLeap (`mleap`)
PyTorch (`pytorch`)
Scikit-learn (`sklearn`)
Spark MLlib (`spark`)
TensorFlow (`tensorflow`)
ONNX (`onnx`)
MXNet Gluon (`gluon`)
XGBoost (`xgboost`)
LightGBM (`lightgbm`)
Spacy(`spaCy`)
Fastai(`fastai`)
Statsmodels (`statsmodels`)

Figure 5.19: *MLflow built-in flavors*

Let us go through other parameters in the flavor section:

- **Loader_module:** This is a required parameter and specifies the Python module that can load the module. For the preceding example, it is mlflow. sklearn.

- **code:** This is an optional parameter and points to the directory containing the code associated with the model. The code must have an implementation of the predict method with the method signature shown in *figure 5.36*.
- **code:** This is an optional parameter again and points to a file or directory containing model data.
- **env:** Again, this is an optional parameter pointing to the exported conda environment. Our example uses this parameter.

Let us now switch gears to the serving model. We will use the MLmodel format for the model in code 5.15 and explained in code 5.33. We will expose a local REST endpoint that can serve predictions. Serving the model is straightforward. The command is shown as follows:

```
mlflow models serve -m <You model folder> -p 1234
```

The location of your model folder can be used in *figure 5.32*. It is important to ensure that the version of Python used to create the model must be the same as the one running mlflow models serve.

When you run the serve command, you would see something like Code 5.20:

```
2021/01/18 21:26:09 INFO mlflow.models.cli: Selected backend for flavor
'python_function'

2021/01/18 21:26:11 INFO mlflow.utils.conda: === Creating conda environment
mlflow-0c1dee08b665ecfe19d88fcce73bca066942b08c ===

Collecting package metadata (repodata.json): done
```

***Code 5.20:** Serving command output*

The selected backend is the python_function and then it is creating a conda environment installing the dependencies. But from where did it find the dependent packages details? The answer is in the MLmodel file python_function entry:

```
flavors:

  python_function:

    env: conda.yaml

    loader_module: mlflow.sklearn

    model_path: model.pkl
    python_version: 3.7.3
```

The env key points to the conda yaml file containing the dependent libraries details. This is how it looks for the code in code 5.21:

```
channels:

- defaults

- conda-forge

dependencies:

- python=3.7.3

- pip

- pip:

  - mlflow

  - scikit-learn==0.21.2

  - cloudpickle==1.2.2

name: mlflow-env
```

Code 5.21: *Conda YAML for model dependencies*

Rather than creating the conda environment locally, it is possible to create a Docker image to serve the model. The default entrypoint serves the specified MLmodel at port 8080 within the container within the python_function flavor. The commands are straightforward. The following command builds a Docker image with the name foo-image and serves the model from the specification run-id folder containing the model (This would be inside your mlruns folder):

```
mlflow models build-docker -m "runs:<Path to your model folder>" -n "foo-image"
```

Serving the model will now be a Docker run command as shown as follows:

```
docker run -p 5001:8080 "foo-image"
```

Note that Docker build is experimental at the time of writing this book. Now that the model is deployed as a REST service, it is time to call it as shown in Code 5.22:

```
curl -X POST -H "Content-Type:application/json; format=pandas-split" --data
'{"columns":["x1", "x2"],"data":[[12.8, 0.029]]}' http://127.0.0.1:1234/
invocations
```

***Code 5.22:** HTTP post call to the REST service*

The input data is a JSON serialized pandas DataFrame. There are a couple of things to know about JSON serialization of pandas DataFrame.

The data format in the Code 5.22 is split oriented – the column and data are separate. If you are passing split oriented data, then make sure that the format attribute is set to pandas-split. The other way is to serialize it in the record orientation format. To indicate this serialization, you need to set the format attribute as pandas-record. An example for pandas-record is shown in the following code:

```
curl -X POST -H "Content-Type:application/json; format=pandas-record"
--data '[

{"x1":12.8, "x2":0.029},

{"x1":7.8, "x2":3.029}

]' http://127.0.0.1:1234/invocations
```

We deployed and served the models locally, but we all know that it is a cloud era. It is no surprise that if you desire to deploy the python_function flavor to the Azure environment, then you should use the mlflow.azureml module (). What about Amazon sagemaker – mlflow.sagemaker (https://aws.amazon.com/blogs/machine-learning/managing-your-machine-learning-lifecycle-with-mlflow-and-amazon-sagemaker/)? You can have custom targets as well. I would recommend reading the documentation for the details (https://mlflow.org/docs/latest/models.html#deployment-to-custom-targets). I am not going into the details of cloud deployment as it requires understanding of cloud environments. However, the following figure illustrates the MLflow architecture with aws sagemaker:

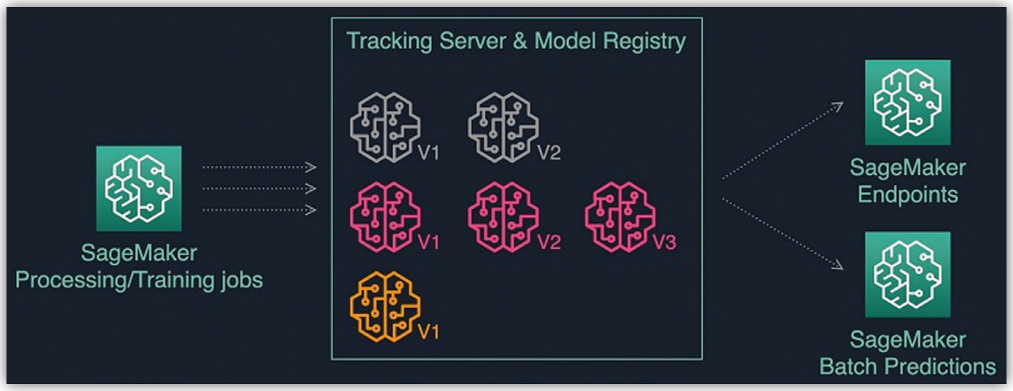

Tracking Server & Model Registry
V1
V2
V1
V2
V3
V1
SageMaker Processing/Training jobs
SageMaker Endpoints
SageMaker Batch Predictions

***Figure 5.20:** MLflow sagemaker architecture*

5.2.5 MLflow Registry

Now, we will take a look at model registry which is a recent entry in the MLflow project. At a high-level, model registry is a centralized store to manage the different stages of a model deployment pipeline. Think of tracking stage transitioning; for example, staging and production. This includes model versioning, annotations, and descriptions. Let us understand the motivation for the registry. Just logging a model doesn't mean it is productionized or production ready. You know that quite a few iterations are required before you decide on a particular model. In a business environment, these experiments could run into 1000. Beyond the experiment stage, a model might have to be tested in a staging and production environment. The production pipeline becomes a challenge when you have multiple ML projects going in parallel. A picture speaks a thousand words so the following is a visualization of where the model registry fits in the architecture:

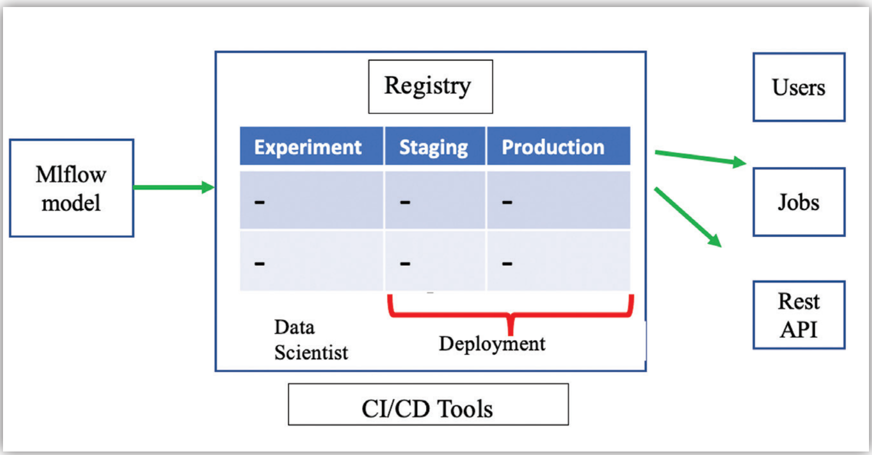


Figure 5.21: MLflow registry motivation

MLmodel is the starting point for registry entries. What registry offers is versioning and tracking of different models. This may feel like classical CI/CD pipeline which it is but it has additional semantics of ML baked in. There are two functionality frames related to registry that are important to understand and remember.

The first is stages. In the real world, no model coming directly from a data scientist experiment would ever get deployed to production. It would go through stages like staging, production, etc. The MLflow UI has features to segregate the model into different stages.

The second is collaboration. In a lot of places, a formal handover is required between the data scientist and ML/MLops engineers. The MLflow registry offers request and approval features to meet these needs. Note that along with UI, there are API available as well so that you can automate the tasks.

5.2.5.1 Registry components

There are a few concepts that we touched upon while explaining registry and it is now time to put the formal definition in place:

- **Model:** This is MLmodel which you already know now. Not that it needs to be logged before getting registered with the registry.
- **Registered model:** Once a model is registered (We will learn the registration process shortly), it has a unique name, versions, associated transitional stages, model lineage, and other metadata.
- **Model version:** When a new model is registered with the registry, it is assigned a value of 1. It is the version of the registered model. Every subsequent addition of the model increments the version number by one.
- **Model stage:** MLflow provides certain pre-defined stages like staging, production, or archived to which you can assign a registered model. Along with assigning a version to a stage, you can also transition a model version from one stage to another.
- **Annotations and description:** Along with the versioning and associating with a stage, it really helps if you add notes and annotations to the top-level model. You can use markdown to add descriptions and other relevant information that you think would be helpful for others to know to use the model effectively and easily. Some details that I have seen commonly provided include algorithm used along with dataset that was used to train the model.

5.2.5.2 Workflow

We will explore the UI and go through the steps for registering the model and transitioning to various stages. It is no surprise that for any model to be registered, it needs to be first logged via the tacking API. We will use the same model explained in code 5.23 for the sake of continuity. Also, I would recommend deleting the mlruns folder (if present). This helps in quickly finding your train model. Recall that logged/tracked model are saved inside the run-id folder:

1. Use this location (https://sqlite.org/index.html) to install SQLite. Creating DB is a straightforward command. On your Command Prompt or Terminal

type, use the command sqlite3 mlflow.db. This would create the SQLite DB for you. While I have given mlflow.db, feel free to give a name that you like but make sure to update the DB name in the server command.

2. While running the server, we also need to provide two values as follows:
   a. **`--backend-store-uri:`** URI where to persist the experiment and data. In our case, it will be aSQLite DB.
   b. **`--default-artifact-root:`** Local URI to store the artifacts. Please create a folder called "store" in the same folder containing Code 5.23.

   The command to launch the server is shown as follows:

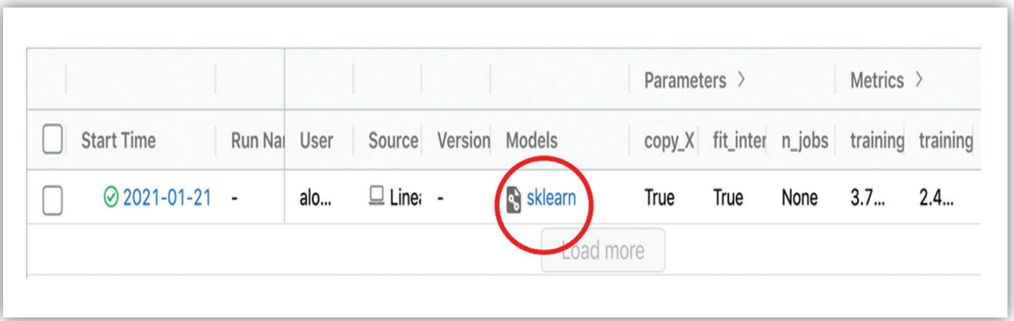
```
mlflow server --backend-store-uri sqlite:////<REPLACE_WITH_YOUR_
PATH_TO_DB>/mlflow.db --default-artifact-root < REPLACE_WITH_YOUR_
PATH_TO_DB >/store
```

Code 5.23: *MLflow server command with the backend and artifact URI*

3. Before we train the model, we need to change the tracking server URI. Until now, by default, it was going to the mlruns folder. We will change it to SQLite DB. This is required because the mlfow registry only works for SQLAlchemy supported database (PostgreSQL, MySQL, SQLite, etc.). In code 5.23, we need to add the tracking location:

```
import mlflow
mlflow.set_tracking_uri("http://localhost:5000")
```

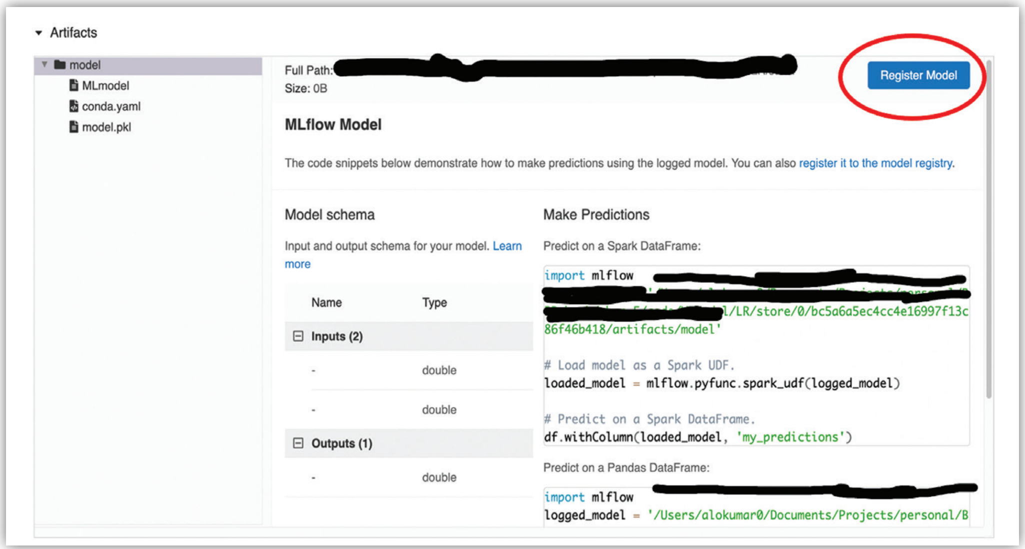
4. You are all set to run your training code. Run code 5.23.
5. If everything went well until step 4, then you would see your experiment on the UI (http:/ /localhost:5000):



							Parameters ›			Metrics ›	
☐	Start Time	Run Na	User	Source	Version	Models	copy_X	fit_inter	n_jobs	training	training
☐	✓ 2021-01-21	-	alo...	Line:	-	sklearn	True	True	None	3.7...	2.4...

Figure 5.22: *Tracking with SQLite backend*

6. Click on the model. Scroll to the **Artifacts** section and register the model by clicking on the **Register Model** button.



Full Path: [redacted]
Size: 0B

MLflow Model

The code snippets below demonstrate how to make predictions using the logged model. You can also register it to the model registry.

Model schema

Input and output schema for your model. Learn more

Name	Type
Inputs (2)	
-	double
-	double
Outputs (1)	
-	double

Make Predictions

Predict on a Spark DataFrame:

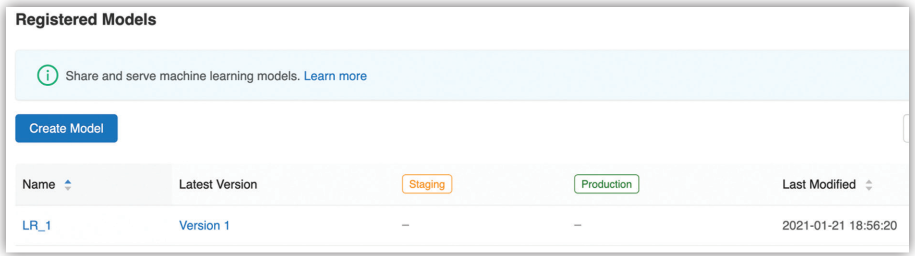
```
import mlflow [redacted]
[redacted]l/LR/store/0/bc5a6a5ec4cc4e16997f13c
86f46b418/artifacts/model'

# Load model as a Spark UDF.
loaded_model = mlflow.pyfunc.spark_udf(logged_model)

# Predict on a Spark DataFrame.
df.withColumn(loaded_model, 'my_predictions')
```

Predict on a Pandas DataFrame:

```
import mlflow [redacted]
logged_model = '/Users/alokumar0/Documents/Projects/personal/B
```

Figure 5.23: Registering a model

7. Click on the model menu option (http://127.0.0.1:5000/#/models) and you should see the registered model as follows:



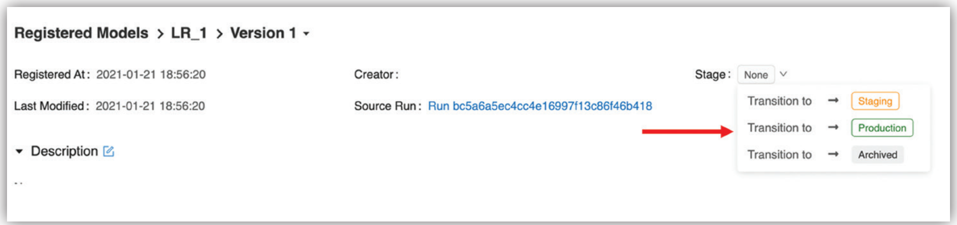
Registered Models

Share and serve machine learning models. Learn more

Name	Latest Version	Staging	Production	Last Modified
LR_1	Version 1	–	–	2021-01-21 18:56:20

Figure 5.24: Registered model view

8. Note that right now, it is not transitioned to staging or production. The staging and production columns are empty. Even though it is not recommended, for learning, let us transition it to production. To do that, click on the version 1 link under the latest Version column. This page is called the model version page.



Registered Models > LR_1 > Version 1

Registered At: 2021-01-21 18:56:20
Last Modified: 2021-01-21 18:56:20
Creator:
Source Run: Run bc5a6a5ec4cc4e16997f13c86f46b418
Stage: None

Figure 5.25: Model stage transition

9. Now that the model has been pushed to production. Let us use the production version for prediction. Code 5.24 is the prediction for code 5.23 (An estimator using linear regression) with backend tracking URI set to SQLite:

```
import mlflow.pyfunc

import mlflow

import pandas as pd  #--1

mlflow.set_tracking_uri("http://localhost:5000")  #--2


model_name = "estimator1" #--3

stage = 'Production' #--4


model = mlflow.pyfunc.load_model(

    model_uri=f"models:/{model_name}/{stage}"  #--5

)

input_sample = {'x1': [3.0, 2.0], 'x2': [2.5,3.8]}  #--6

df = pd.DataFrame.from_dict(input_sample)  #--7


print(model.predict(df))  #--8
```

Code 5.24: *Prediction using productionized model*

Let us unpack the code as follows:

1. The Pandas module had to be imported in order to package the input into a pandas DataFrame. Recall the predict method signature:

   ```
   predict(model_input: pandas.DataFrame) -> [numpy.ndarray | pandas.
   (Series | DataFrame)]
   ```

2. We have set the tracking URI. You don't need to do it if you have set the MLFLOW_TRACKING_URI environment variable.
3. We provide the name of the registered model.
4. The stage is set to production in order to use the production version model.
5. We are using the pyfunc flavor and loading the model.

6. We create a dummy input dictionary. If you recall, mlflow allows you to log signature and input examples.
7. We load the dict.Into a DataFrame. Essentially, we need a DataFrame as the input type to predict method
8. Finally, we call the predict method.

So, what is the benefit here? Well, let us think about it. You have a designated production version model that you would like others to use. But at same time, there may be another version undergoing testing using the staging version. When someone pushes the new version to production, the consumer need not worry. The inference code would pick up the new production version.

It is also possible to fetch a specific version. The only thing that needs to be changed is the URI:

```
model = mlflow.pyfunc.load_model(

    model_uri=f"models:/{model_name}/{model_version}"
```

It is also possible to serve the production model as a REST endpoint. The syntax is no different from what you have seen earlier other than the model location:

mlflow models serve -m "models:/estimator1/Production" -p 1234

To post a request to the end point, you can use the curl command:

```
curl -X POST -H "Content-Type:application/json; format=pandas-
split" --data '{"columns":["x1","x2"],"data":[[8.0, 3.5]]}'
http://127.0.0.1:1234/invocations
```

This completes our exploration of model registry. While we used UI to manage the registration and transitions, it is worthwhile to remember that all those functionalities are available as API as well. The purpose of API is enabled automation. I would encourage you to read the documentation for the details (https://mlflow.org/docs/latest/model-registry.html#serving-an-mlflow-model-from-model-registry).

5.3 Conclusion

Well, we have reached the end of the chapter. Knowing an ML pipeline is an essential skill to be an effective full stack AI developer. In the initial chapter, we covered Sacred and Omniboard as a tracking tool. So, how are they different from MLflow and how do you decide between the two? I prefer to use a simple thumb rule – If your organization has only one or two ML projects with very small teams (2-3 people), consider sacred + omniboard. However, if you are scaling beyond it, then

MLflow would be a good choice.

The classical workflow management solutions are not apt for the iterative nature of ML applications development. MLflow is a workflow management system that offers CI/CD pipeline but with ML semantics baked in. The demand and need for workflow management for ML application is growing as organizations beyond the big 4 are building their own ML powered tools and systems. It is because of these opportunities, that you see MLflow as a cloud-based managed service being offered by lots of vendor. Knowing a framework like MLflow is essential for any ML engineer or data scientist these days. In the next chapter, we will explore the feature stores which emerged as a big trend in 2020. We will explore a new aspect – feature store – which is quite bleeding edge and gaining a lot of attention lately. See you in the next chapter.

5.4 Points to remember

- ML project development is fundamentally different from traditional projects.
- In the classical software, the objective is to meet functional requirements, whereas in ML, the objective is to optimize some metrics.
- In the classical software, the quality is dependent on code, whereas in the ML, the quality is dependent on model architecture and data.
- In the classical software, the outcome is fixed, whereas in the ML, the outcome might change due to data drift.
- The common ML pipeline challenges are as follows:
    - What features were being used?
    - Was there any change in feature engineering?
    - What hyperparameters had been changed?
    - What data was used – same, new, or modified?
    - What architecture was tried?
    - How would you retrain if you have new dataset?
    - How would you debug it?
- MLflow is an open-source platform to simplify the management of machine learning project lifecycle. At a high level, the lifecycle includes the ability of experimentation, reproducibility, and deployment.
- MLFlow groups its features across the following four components:
    - Tracking - Record and query, experiments, code, data, config, results.
    - MLProject - Packaging format for reproducible runs on any platform.

  - o MLModel - General format that standardizes deployment paths.
  - o Registry - Centralized and collaborative model lifecycle management.
- These four components solve the following crucial problems of ML Pipeline:
  - o How to keep track of experiments?
  - o How to reproduce results?
  - o How to package and deploy models in a standard way?
  - o How to centrally manage models?
- Set the environment variable MLFLOW_TRACKING_URI to track remotely.
- Use mlflow.set_tracking_uri() to set the tracking location inside the code.
- The tracking location could be the following:
  - o Local file system
  - o SQLAlchemy supported database
  - o HTTP Server
  - o Managed service from Databricks
- MLFlow UI offers the following features:
  - o Run listing
  - o Searching for runs by parameter or metric value
  - o Downloading run results
  - o Comparison
  - o Visualizing run metrics
- Tracking services have the two following components:
  - o Entity or metadata store - Stores lightweight metadata
  - o Artifact store - Stores heavyweight metadata
- The Artifact store could be any of the following:
  - o Amazon S3
  - o Azure Blob storage
  - o HDFS
  - o Google cloud storage
  - o Databricks' file system
  - o FTP
  - o SFTP
  - o NFS

- MLFlow project spec contains pointers to the following artifacts:
    - o Code
    - o Config
    - o Dependency
    - o Data
- You can have multiple entry-points and combine them to create multistep workflow.
- MLmodels offers abstraction over models.
- By using flavors, MLmodels can be used in different deployment environments.
- The standard flavor is python_function.
- To serve models, use model serve command.
- To manage models centrally, they need to register.
- Registry will only work with SQLAlchemy supported backend storage.
- Registry offers out of the box stages like "Stage," "Production," and "Archived".

5.5 Questions

1. What are the four components of MLflow?
2. What are the different tracking URI supported with MLflow?
3. What are two types of tracking components?
4. Which tracking component would you use for storing lightweight information like parameters and metrics?
5. Which tracking component would you use for storing heavyweight information like models?
6. What is the default flavor offered by MLModels?

CHAPTER 6
Feature Store for ML

Features are the backbone of ML and that is why feature engineering is such an important step and skill in ML. We are not going to discuss feature engineering but how to effectively maintain and reuse those engineered features via feature stores. At a high level, feature store could be imagined as a warehouse of features. It is a central vault for storing documented, access-controlled features. Feature store is an emerging concept with the objective of removing the challenges in taking ML models to production.

The focus of this chapter will be on learning about feature stores via an open-source feature store called feast (https://docs.feast.dev/).

Structure

In this chapter, you will learn the following:

- Introduction to feature store
- Feature store framework
- Feast

Objective

By the end of this chapter, you will know the following:

- Introduction to feature store.
- Transforming raw data into features.
- Storage, curation, and retrieval of features from feature store
- Intro to feast

6.1 Introduction to feature stores

Let us understand the motivation for feature stores. The motivation is based on the simple fact that data scientists are not data engineers. For a data scientist, the objective is to optimize some metrics, or in simple terms, improve the predictions. Their toolset would typically look as follows:

Figure 6.1: *Data scientist tools*

However, in the real world, the features can come from a variety of data sources:

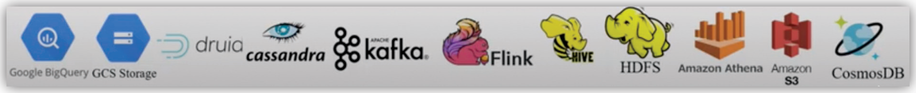
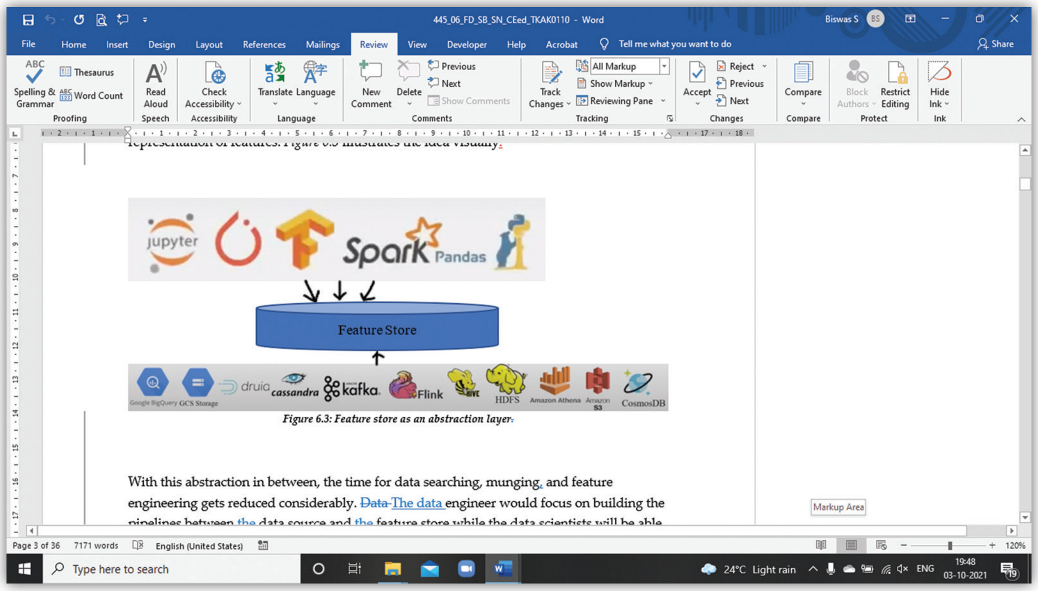


Figure 6.2: *Varied features sources*

There are a few problems that immediately stands out. These are as follows:

1. Getting the data now requires the knowledge of all these platforms.
2. Transforming all the data into a format compatible with the ML/DL framework. TensorFlow expects data in TFRecords, PyTorch in its all format, and then a few in NumPy. You get the idea!
3. The problem is not restricted to access and transformation but also leads to duplication. One feature somehow extracted by one team is not easily sharable with the others.

If you have heard of the cliché that 80% of the time is spent on preparing the data, then you know the reasons for it. So, is there any solution for this? Well, the problems can be solved to a great extent by creating a platform independent representation of features. *Figure 6.3* illustrates the idea visually:



representation of features. Figure 6.3 illustrates the idea visually.

Figure 6.3: Feature store as an abstraction layer.

With this abstraction in between, the time for data searching, munging, and feature engineering gets reduced considerably. ~~Data~~ The data engineer would focus on building the pipelines between the data source and the feature store while the data scientists will be able

Page 3 of 36 7171 words English (United States) 120%

Figure 6.3: *Feature store as an abstraction layer*

With this abstraction in between, the time for data searching, munging, and feature engineering gets reduced considerably. The data engineer would focus on building the pipelines between the data source and the feature store while the data scientists will be able to use the features conveniently. With the feature store as a separate entity, it becomes quite easy to share as well. Don't be surprised to know that large organizations can have thousands of features and can be continuously extended with new features seamlessly. I prefer to think of it as a warehouse of features.

6.1.1 Understanding features

It may feel a bit strange to revisit the question – What is a feature? A feature is generally imagined as a column in a data warehouse, but at the core it is a measurable property of a phenomena under observation. This also means that features can be computed from raw or structured data sources and not necessarily always a column in a table. The following are a few examples of features:

1. A raw word if the source data is a natural language text.
2. A pixel if it is an image.
3. A sensor value when coming from an IOT device.
4. An aggregate like mean, max, etc.

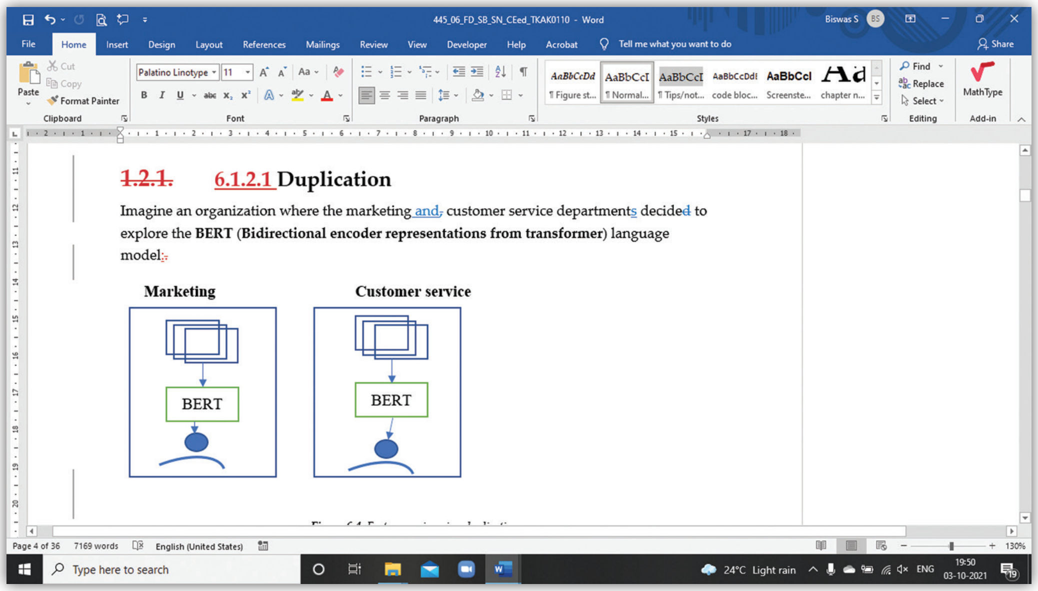
5. A window like `last_hour, last_day`, etc. if it is a time series data.
6. Derived representations like embeddings (think of wordvec, gloVec, etc.)

6.1.2 Feature store motivation

With the refined knowledge of features, let us explore some of the problems associated with the solid feature engineering approach.

6.1.2.1 Duplication

Imagine an organization where the marketing and customer service departments decide to explore the **BERT (Bidirectional encoder representations from transformer)** language model:



~~1.2.1.~~ 6.1.2.1 Duplication

Imagine an organization where the marketing and~~,~~ customer service departments decide~~d~~ to explore the **BERT (Bidirectional encoder representations from transformer)** language model:~~.~~

Figure 6.4: Feature engineering duplication

So, each will take their text, transform it into a format compatible with BERT, and then test it. It may work for one of them, but then customer service too would do the same thing. There is a duplication of feature engineering at work here.

6.1.2.2 Inconsistency

There is another bigger problem of feature inconsistency as shown in *figure 6.5*:

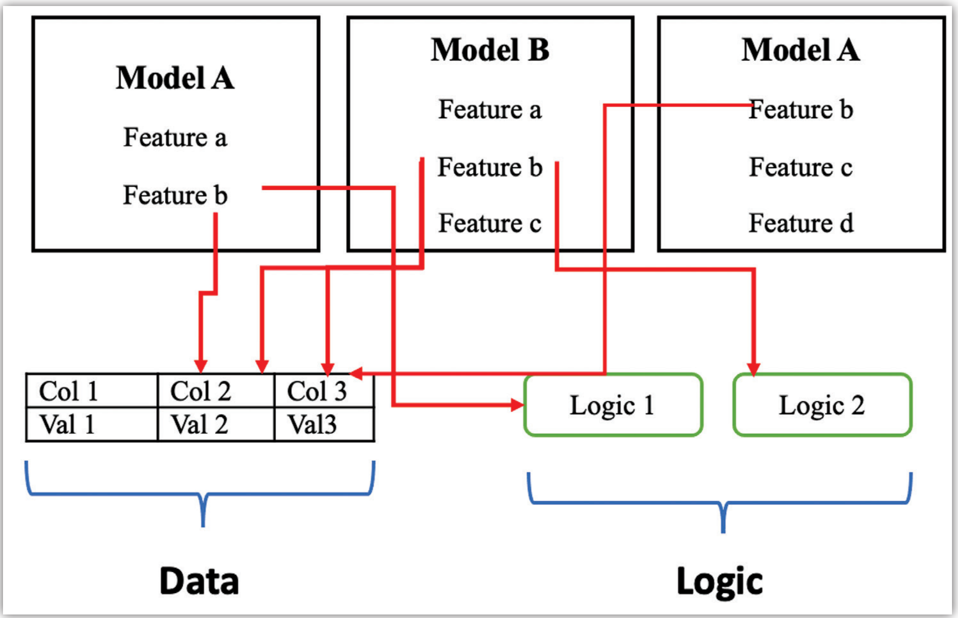


Figure 6.5: *Feature inconsistency*

In *figure 6.5*, the same features are being calculated in different ways across different model trainings. We could imagine these models being trained by different teams in the organization. Feature b in model A is using Col 2 and using some custom logic, that is, logic 1. Whereas in Model B, feature b is using a combination of col 2 and 3 along with custom logic 2. The highlight here is the situation that the same feature has different implementation/interpretation across different systems.

6.1.2.3 Inconsistent access to data

ML systems built on traditional data infrastructures are often tightly coupled to databases, object stores, streams, and files. We all know that tightly coupled systems are extremely brittle and would break even from a small change in the dependent systems. Here, any change in data infrastructure may break the dependent ML systems. Another associated challenge is of double and different (a lot of the times) implementations of data retrieval and serving causing inconsistencies in data, and that in turn can lead to training-serving skew.

6.1.2.4 Difficulty in production deployment

Different teams may have different objectives, leading to frictions causing slow to market. Data engineering teams like classical software team's objective would ensure stability in production environments. Whereas a data scientist's objective would be to achieve high accuracy and expect frequent production deployments.

6.1.2.5 Incorrect training and production data

A lot of times, a model performs with high accuracy during the development time but performs poorly in the production environment. This is majority of the times due to insistent view of data in development and production. ML models in production require a view of data that is consistent with the one on which they are trained; otherwise, the accuracy of these models could be compromised.

6.1.3 Feature store guiding principles

To solve the preceding problems, there are a few guiding principles that are important to know and expect from a feature store:

- The features should be discoverable and reused.
- The features should be versioned, access-controlled, and governed.
- The ability to precompute and automatically backfill features. Aggregates and embeddings are a few examples of it. These kinds of features must be pre-computed. Pre-computation helps in reducing the time to train the models.
- Along with pre-computation, the real time computation of features should be possible.

The guiding principles could be imagined in the form of the following high-level architecture shown in *figure 6.6*:

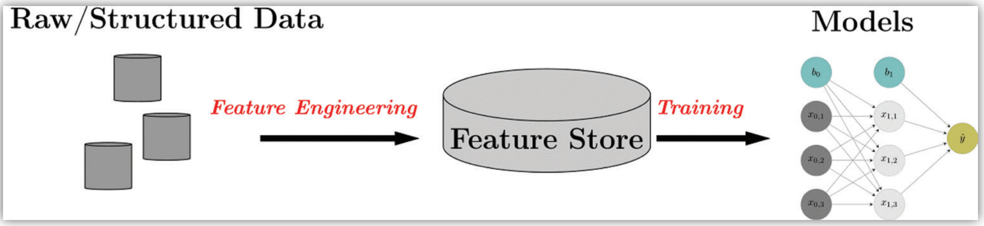


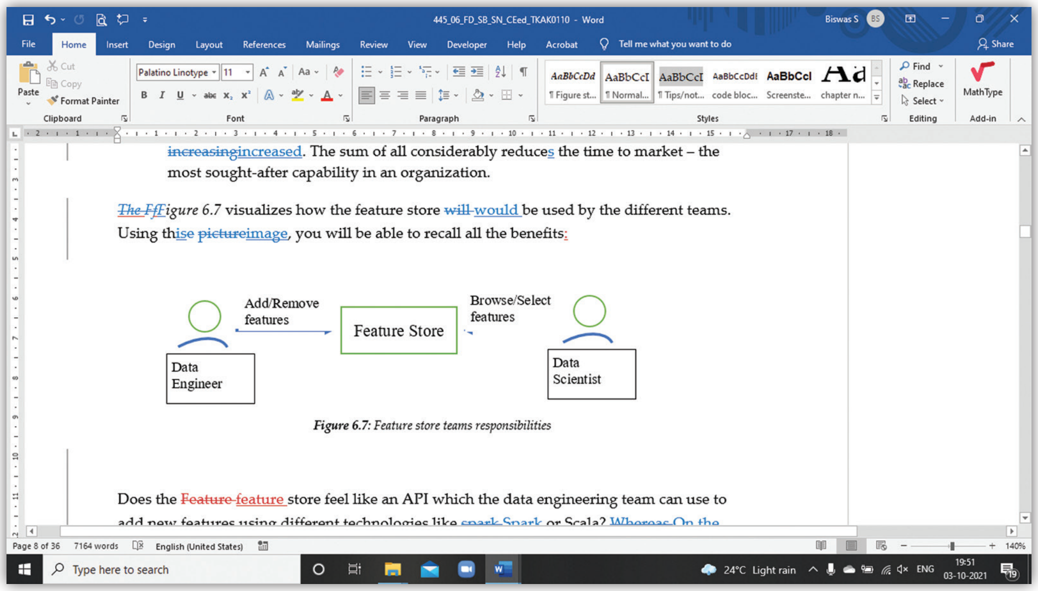
Figure 6.6: *Feature store abstraction/interface*

The data engineer's focus will be on feature engineering and the data scientist would focus on training. Let us list down the benefits of the figure 6.6 architecture as follows:

- The feature store decouples the models from the underlying data infrastructure. The layer abstracts feature storage from feature retrieval, providing a consistent means of referencing feature data for retrieval, and thus ensuring the models' portability between training and serving.

- The feature store offers a centralized repository where new features can be added by the data engineering team and can be reused by different data scientist teams.
- With a centralized system comes the benefit of security and governance.
- By removing the dependency on learning new tools for featuring engineering, the training time will be reduced and the data scientist's productivity will be increased. The sum of all considerably reduces the time to market – the most sought-after capability in an organization.

Figure 6.7 visualizes how the feature store would be used by the different teams. Using this image, you will be able to recall all the benefits:



~~increasing~~increased. The sum of all considerably reduces the time to market – the most sought-after capability in an organization.

~~*The F*~~*fFigure 6.7* visualizes how the feature store ~~will~~ would be used by the different teams. Using thise ~~picture~~image, you will be able to recall all the benefits:

Figure 6.7: Feature store teams responsibilities

Does the ~~Feature~~ feature store feel like an API which the data engineering team can use to add new features using different technologies like ~~spark~~ Spark or Scala? ~~Whereas~~ On the

Figure 6.7: *Feature store teams responsibilities*

Does the feature store feel like an API which the data engineering team can use to add new features using different technologies like Spark or Scala? On the other hand, the data scientists will use the API to train their Python code without worrying how the features were produced.

Let us zoom out a little further to see the overall training/serving process:

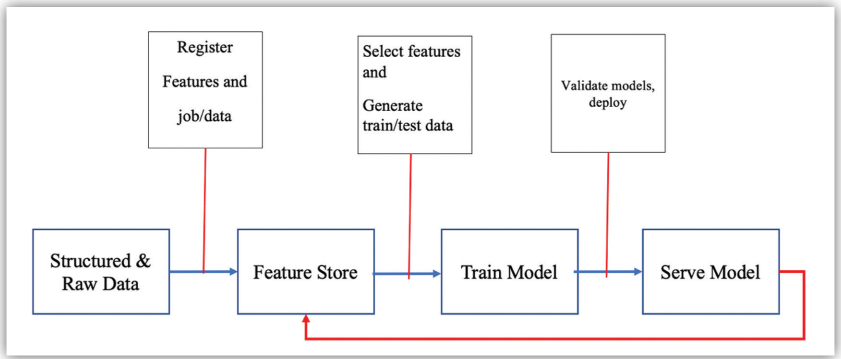


Figure 6.8: *ML pipeline with feature store*

Except the line connecting the model serving with the feature stores, the rest of the stuff should feel familiar now. The model serving is connected to the feature store since while doing the predictions, the model may need features. An example could be a fraud detection where the transaction data may not have all the data. It goes without saying that for this kind of online access, the latency should be low.

Figure 6.9 shows the overall end-to-end ML process with the feature store:

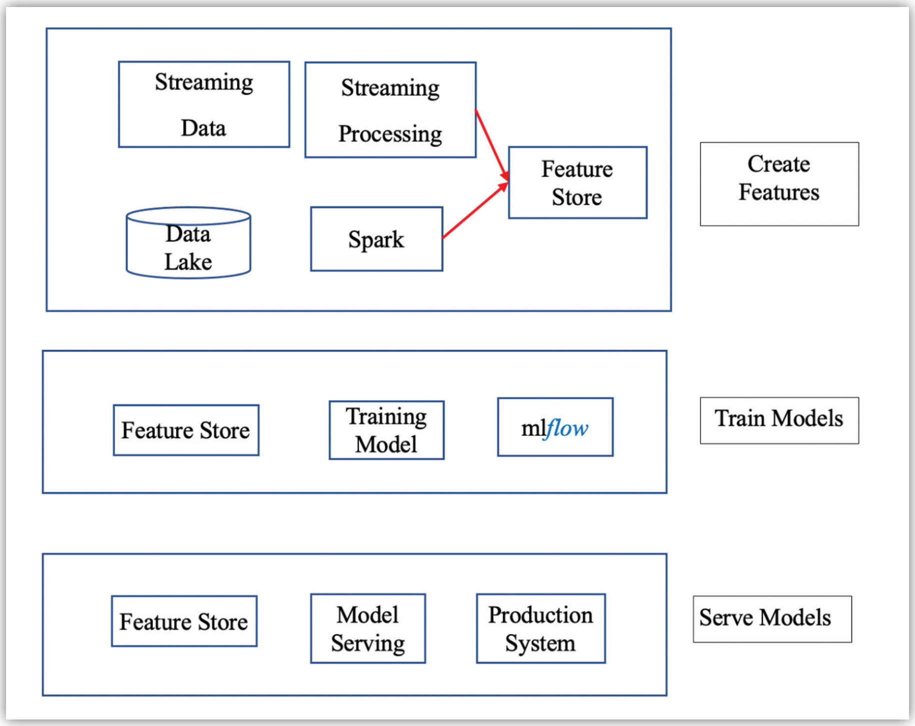


Figure 6.9: *End-to-end ML with feature stores*

Based on the discussion so far, we can split the feature store uses in two types as follows:

- **Offline (batch/streaming) feature store:** This is useful for batch applications or streaming applications. These applications work on a large amount of data. You may think of Spark applications or Flink applications that run large jobs. Imagine a personalized email campaign that you would like to run for all your customers.

Figure 6.10 shows the offline feature store high-level process for the personalized marketing campaign. The picture has been simplified so that you get the idea and not get bogged down with the details:

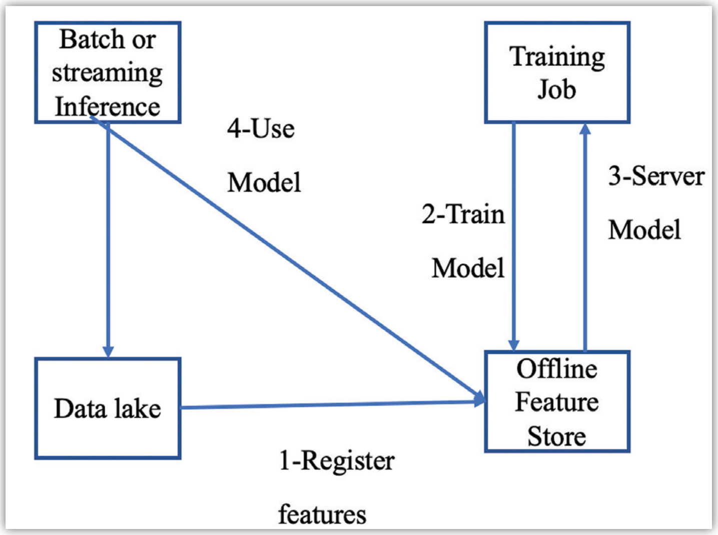


Figure 6.10: *Offline feature store process*

So, you register features that you are going to use to train the model. Then, you create the training data from those features. With the training data in place, you train the model and persist it. Then, you write your streaming app that will use the model and make the content personalization based on the model predictions. To use the model, the job will have to calculate those features and then pass it to the model.

- **Online feature store:** As the name suggests, it is for real time applications. In this case, the application will send some feature data to the feature store, but it will also ask the store to fill the missing feature values. Once all the required feature values have been received, the model is requested to make the prediction. *Figure 6.11* shows the process visually:

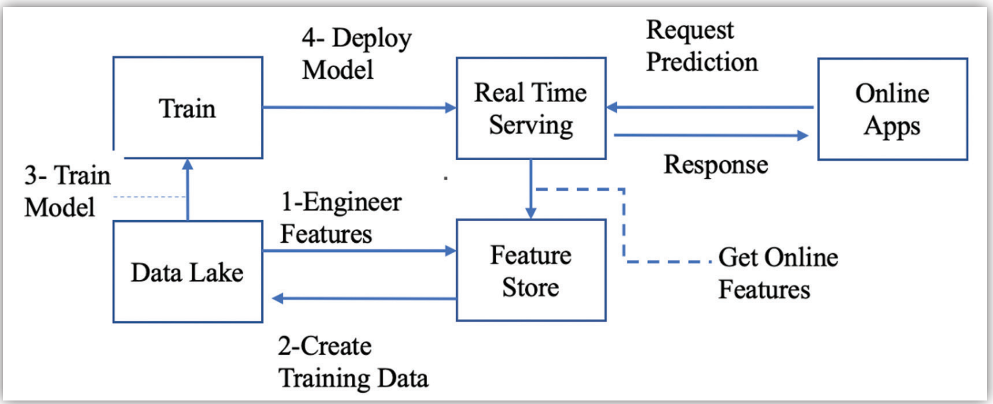


Figure 6.11: *Online Feature Store*

Now, to ensure that we are clear on the purpose of the feature store, let us be clear on what it is not which is as follows:

- It is not the traditional ELT (extract, load and transform).
- It is not a replacement of the data warehouse.
- It is not a general-purpose data catalogue but a catalogue of features.

6.2 Feature store frameworks

While the whole idea of the feature store is powerful and exciting, there are not many projects in the open-source space. It is primarily because the idea is bleeding edge. However, have no doubt of it growing in the coming times. This gets reflected when you look at the list of the feature stores which is as follows:

Platform	Open source	URL	Support platforms
Hopsworks	Yes (AGPL-V3)	https://github.com/logicalclocks/hopsworks	
AWS, GCP, On-Prem			
Michelangelo	N/A		
Feast	Apache V2	https://github.com/gojek/feast	
GCP			
Conde Nast	N/A		Proprietary
Zipline	N/A		Proprietary

Comcast	N/A		Proprietary
Netflix Metaflow	N/A		Proprietary
Twitter	N/A		Proprietary
Facebook FBLearner	N/A		Proprietary
Pinterest			
Galaxy	N/A		Proprietary
Iguazio Feature Store			
N/A		Proprietary	
Amazon sagemaker feature store		https://aws.amazon.com/sagemaker/feature-store/	
AWS			

Table 6.1: *Feature Store Project list (Source - https://www.featurestore.org/)*

Beyond these organizations, I have heard of feature store implementation in a lot of large organizations like Intuit, etc., in podcasts. The underlying expectation that I have for you is to feel excited about feature store. We will next explore Feast (https://github.com/gojek/feast).

6.3 Feast

As per the documentation, Feast is an operational data system for managing and serving machine learning features to models in production. The following figure maps all the concepts that you have learned so far to the feast architecture:

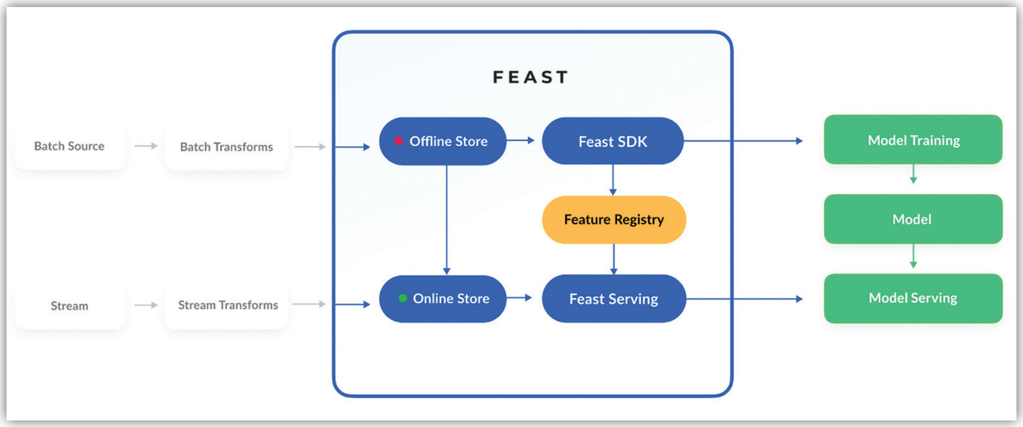


Figure 6.12: *Feast high-level architecture (Source – feast documentation)*

6.3.1 Feast components

At a high level, feast has four components:

- **Register:** Feast provides a central registry through which features are re-used and collaborated. Any new feature needs to be registered with the registry. You already know that the registry is the central interface for all interactions with the feature store.
- **Ingest:** Ingest allows data ingestion from both batch and streaming sources into both online and offline feature stores. Ingested features are available for training and serving.
- **Serve:** A feature retrieval interface that provides a consistent view of the features in stores. Feast provides a point-in-time correct interface for training data and a low-latency API for online serving.
- **Monitor:** Monitor allows the publication of operational metrics, statistics, and logs to their existing production monitoring infrastructure. This enables the monitoring of features in feature store.

6.3.2 Installing Feast

It is time for to us to explore feast but before that, let us go through the installation steps. The setup is straightforward due to the use of Dockers.

```
git clone https://github.com/feast-dev/feast.git

cd feast/infra/docker-compose

cp .env.sample .env

docker-compose pull && docker-compose up -d
```

When you run the docker-compose command, you would find a few containers running. The following are high-level components:

- Feast core with Postgres.
- Feast online serving with Redis.
- Feast job Service.

If the containers are running successfully, then you should be able to connect to the running Jupyter notebook server via the URL – http://localhost:8888/tree.

The following are two common problems in case you face problems with your installations:

- **Ports already in use:** Feast uses the following ports on your host machine. Please ensure that the ports are available. If required, you can change the ports in docker-compose.yml file:
  a. 6565
  b. 6566
  c. 8888
  d. 9094
  e. 5432
- **Containers not running:** Check the Docker logs using the following command:

```
docker-compose logs -f -t
```

6.3.3 Architecture

The following figure shows the detailed architecture along with the technology stacks. The technology stack reinforces how the feature store benefits are implemented:

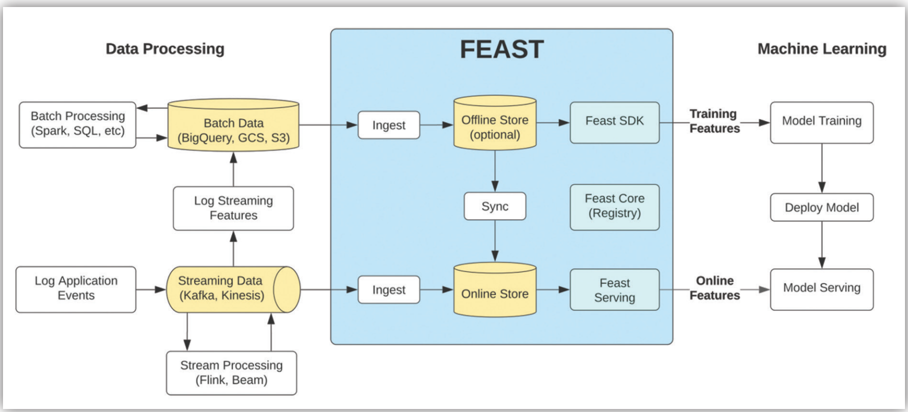


Figure 6.13: *Feast tech architecture (Source: feast documentation)*

Based on the architecture, the following are the components that you would typically find in a feast deployment:

- **Feast core:** Core is the central registry for feature and entity definitions. Think of entities like customer, sales, etc. We will look at entities in detail shortly. The registry is a central interface for user interactions with the feature store. Teams use the registry as a common catalogue to explore, develop, collaborate on, and publish new definitions within and across teams.

  The definitions in the registry configure feature store system behavior.

Automated jobs use the registry to schedule and configure data ingestion and storage. It forms the basis of what data is stored in the feature store and how it is organized. The serving APIs use the registry for a consistent understanding of which feature values should be available, who should be able to access them, and how they should be served.

The registry allows for important metadata to be attached to feature definitions. This provides a route for tracking ownership, project or domain specific information, and a path to easily integrate with the adjacent systems. This includes information about the dependencies and versions which are used for lineage tracking. Not all of this is provided out of the box from feast at the time of writing this book, but going forward, they should be integrated as these are the table stake requirements from the feature store registry.

To help with common debugging, compliance, and auditing workflows, the registry acts as an immutable record of what is available analytically and what is running in production.

- **Feast job service:** Data loading jobs from sources into stores and training datasets jobs are managed by job service.
- **Feast online serving:** Online serving provides low-latency access to feature values via the online store. For online serving, a feature store delivers a single vector of features at a time, made up of the freshest feature values. The responses are served through a high-performance API backed by a low-latency database.
- **Feast Python SDK:** With the SDK, you can do the following:
  - o Manage feature definitions with Feast Core.
  - o Launch jobs through the Feast Job Service.
  - o Retrieve training datasets.
  - o Retrieve online features.
- **Online store:** The online store stores only the latest feature values of an entity in the database. The online store can be populated by either batch ingestion jobs (in case the user has no streaming source) or can be populated by a streaming ingestion job from a streaming source. The feast online serving service looks up feature values from the online store.
- **Offline store:** The offline store persists batch data that has been ingested into feast. This data is used for producing training datasets. When retrieving data offline (for training), feature values are commonly accessed through notebook-friendly feature store SDKs. They provide point-in-time correct views of the state of the world for each example used to train a model (a.k.a. "time-travel").

Storage

Feature stores persist feature data to support retrieval through feature serving layers. They typically contain both an online and offline storage layer to support the requirements of the different feature serving systems.

Offline storage layers are typically used to store months or years' worth of feature data for training purposes. Offline feature store data is often stored in data warehouses or data lakes like S3, BigQuery, Snowflake, and Redshift. Extending an existing data lake or data warehouse for offline feature storage is typically preferred to prevent data silos. For feast, it is only BigQuery now.

Online storage layers are used to persist feature values for low-latency lookup during inference. They typically only store the latest feature values for each entity, essentially modeling the current state of the world. Online stores are usually eventually consistent, and do not have strict consistency requirements for most ML use cases. They are usually implemented with key-value stores like DynamoDB, Redis, or Cassandra. For feast, it is Redis. The following figure summarizes the storage view:

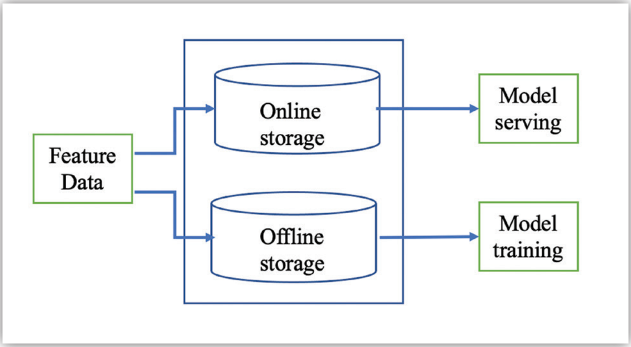


Figure 6.14: *Storage view*

Using the architecture feast can offer the following features:

- Ability to load feature data from both batches (Big query and GCS) and streaming sources like Kafka. Do check out the documentation (https://kafka.apache.org/) to learn about Kafka.
- Ability to serve large volumes of historical features for model training (Python SDK).
- Ability to serve features to online models through a low latency gRPC API (available in java and Go SDK).

- Ensuring feature consistency between training and serving to avoid model-skew.
- Allowing centralized management of features and metadata.

Given the fact that feast is a collaboration between Gojek (https://www.gojek.io/) and Google, you will find more Google supported technologies in the tech stack. I guess this would change over time.

Figure 6.15 shows a deployment view that highlights all the options you have for data:

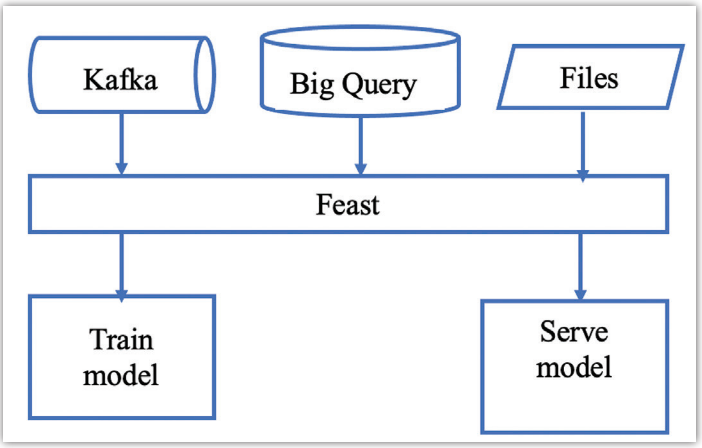


Figure 6.15: *Deployment technology stack*

There are four concepts that you should be aware of to use feast. These are as follows:

- **Projects:** This is the top-level namespace that uniquely identifies a project. Each project maps to a completely independent environment in feast. Users can only work on a single project at a time.
- **Entities:** Entities are the objects in a project that have features. Think of entities as domain objects like customers, products, transactions, etc.
- **Feature tables:** This defines a group of features that occur on a specific entity.
- **Features:** An individual feature within a feature table that highlights the attributes of an entity.

There is a hierarchical relationship, and a visualization helps in remembering the concepts and the relationship as follows:

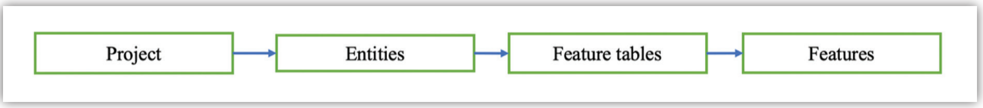


Figure 6.16: *Concepts relationship*

We will next explore the concepts in detail.

6.3.4 Feast concepts

Let us understand the various concepts under this.

6.3.4.1 Entity

An entity is any domain object that the organization is interested in and can be modelled and stored. Entities are usually recognizable concepts, either concrete or abstract, such as persons, places, things, or events. Almost all the feast documentation uses ride hailing data as an example dataset and we will also use it so that you can have a continuation while reading the documentation. Perhaps this is the case because gojek.com is a local delivery platform.

In the world of ride-hailing and food delivery, you can think of customer, order, driver, restaurant, dish, and area as some of the entities. I prefer to think of entities as classes. If entities are classes, then features are the attributes/members of those classes. If you don't like object-oriented metaphors, then think of features as properties of the entity. Another safe way of thinking about features is as columns if it is tabular data and feature values as the column data.

Feature stores use an entity-based data model where each feature value is associated with an entity (for example, a user) and a timestamp. The reason for using an entity-based model is because it provides minimal structure to support standardized feature management, fits naturally with common feature engineering workflows, and allows for simple retrieval queries in production.

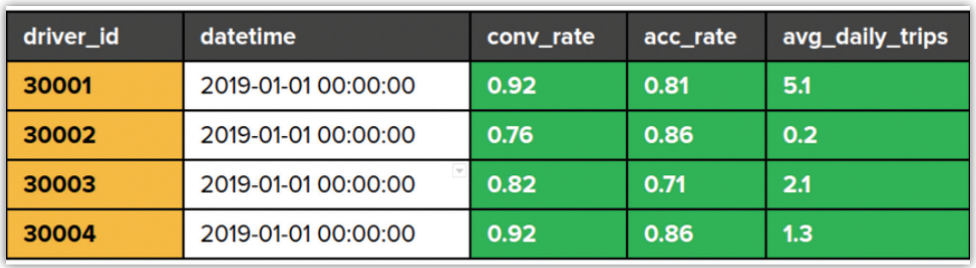
Based on the explanation so far, you can visualize that entities serve two purposes in feast:

- They are used as keys to look up features for producing training datasets and online feature values.
- They offer a grouping of features in a feature table. A feature table must belong to an entity.

Shown below is an example of a customer entity with few attributes. It is not complete but you get the idea.

```
customer = Entity(

name="customer_id",

description="Customer id for ride customer",

value_type=ValueType.INT64,
labels={}
)
```

Before we elaborate on the specification, let us take a look at more evolved examples that map a table to an entity. The following table shows some driver-related information:



driver_id	datetime	conv_rate	acc_rate	avg_daily_trips
30001	2019-01-01 00:00:00	0.92	0.81	5.1
30002	2019-01-01 00:00:00	0.76	0.86	0.2
30003	2019-01-01 00:00:00	0.82	0.71	2.1
30004	2019-01-01 00:00:00	0.92	0.86	1.3

Figure 6.17: *Driver details source*

The specification for *figure 6.17* is mentioned as follows:

```
name: "driver_weekly",

entities:
name: driver_id,
valueType: INT64

features:
name: conv_rate
valueType: FLOAT
name: acc_rate
valueType: FLOAT
name: avg_daily_trips
valueType: FLOAT
```

This allows for the definition of entities, features, and the associated properties. Think of this as a bulk way to define the features.

It is time to go through each of the keys in the specification:

- **Name:** Name of the entity
- **Description:** Description of the entity

- **Value Type:** Value type of the entity. Feast will attempt to coerce entity columns in your data sources into this type.
- **Labels:** Labels are maps that allow users to attach their own metadata to entities.

Now that we know about entity, let us take a look at how to add or update them in the code. This is quite straight-forward:

```
from feast import Client --1

from feast.entity import Entity

from feast.value_type import ValueType


feast_client = Client(core_url="localhost:6565") --2


entity = Entity( --3

     name="driver_entity",

     description="Driver entity for car rides",

     value_type=ValueType.STRING,

     labels={

         "key": "val"

     }

)

feast_client.apply(entity) --4
```

Let us unpack the code as follows:

- We import the necessary libraries. The only new thing is the client import. This contains the functionality for managing features.
- The client object is used for creating, managing, and retrieving features. It needs to be instantiated with a service URL. Here, we have provided core_url which points to the service managing features.
- Entity represents a collection of entities and associated metadata.
- Applying the method of the client object idempotently registers the feature tables and entities with the registry. We will take a look at feature table later.

Note that the parameter could be a single entity, feature table, or a list of entities.

Updating an entity is also straightforward with few API calls though with few restrictions. There are just two steps as follows:

1. Get the entity.
2. Update the entity.

It shows the two steps in code as follows:

```
customer_entity = client.get_entity("customer_id") --1

customer_entity.description = "ID of bike customer" --2

client.apply(customer_entity) --3
```

There are not many steps. The get_entity parameter retrieves the entity to be updated. It's no surprise that we need to specify the name of the entity. If the entity is not found, it raises an exception. You next update the property (metadata) and call the apply method. The version (0.9 at the time of writing this book) only allows labels and description. You are not allowed to change the following:

- Project
- Name of an entity
- Type

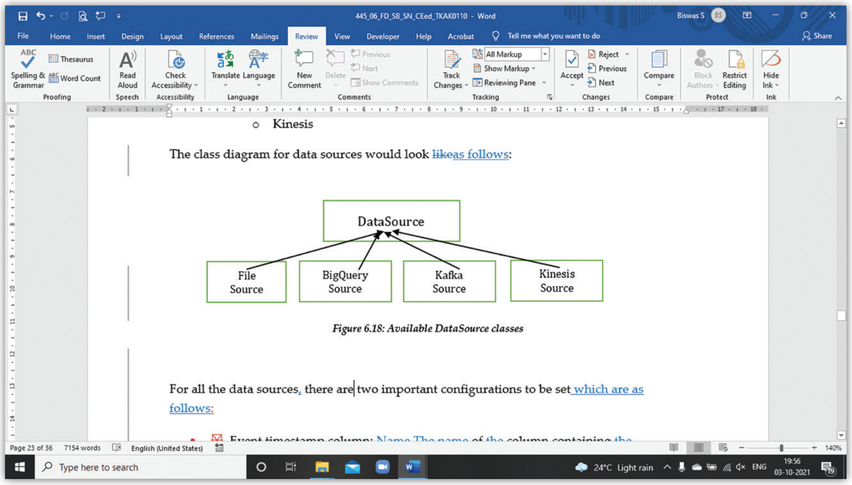
6.3.4.2 Sources

Sources describe the external data that would become the feature data inside the feature store. Sources needs to be registered with feast for the ingestion process **(Source | Feature Store)**.

From the different feast views described earlier, you can easily figure out that there are the following two supported source types:

- Batch source
  - o File (as in Spark): Parquet and CSV files supported.
  - o BigQuery
- Stream source (supports Avro and Protobuf encoding)
  - o Kafka
  - o Kinesis

The class diagram for data sources would look as follows:



○ Kinesis

The class diagram for data sources would look ~~like~~as follows:

Figure 6.18: Available DataSource classes

For all the data sources, there are two important configurations to be set which are as follows:

Figure 6.18: Available DataSource classes

For all the data sources, there are two important configurations to be set which are as follows:

- **Event timestamp column:** The name of the column containing the timestamp when the event data occurred. This is used during the point-in-time join feature values to entity timestamps. Entity timestamps capture the event occurrence, creation, or observation timestamp. Every saved entity in store will have an associated timestamp. In case you don't know about the point-in-time join feature, then you need to understand time-travel. Read the following paragraph to understand time travel.
- **Temporal databases support time-travel:** This is the ability to query data as it was at a given point-in-time or data changes in each time-interval. For a feature store, time-travel has several important uses: when creating train/test data (for example, training data is data from the years 2010-2018, while test data is data from the years 2019-2020). Time-travel is also important while performing point-in-time joins, where we ensure that there is no data leakage from the future when we create train/test datasets from historical data. Note that we rarely require time-travel for features used in serving.
- **Created timestamp column:** The name of the column containing timestamp when data is created. It is used to deduplicate data when multiple copies of the same entity key are ingested. The entity key is the combination of entities that uniquely identify a row. For example, a feature table with the composite entity of (customer, country) might have an entity key of (1001, 5). The key is used during the lookups of feature values and for deduplicating historical rows.

Let us look at the Python code snippet to instantiate a FileSource:

```
from feast import FileSource
from feast.data_format import ParquetFormat


batch_file_source = FileSource(
file_format=ParquetFormat(),
file_url="file:///feast/customer.parquet",
event_timestamp_column="event_timestamp",
created_timestamp_column="created_timestamp",
)
```

Other than the Parquet format, you already know everything else. In case you don't know about Parquet, then Parquet is an open-source file format available to any project in the Hadoop ecosystem. It is designed for efficient as well as performant flat columnar storage format of data compared to row-based files like CSV or TSV files. If you're interested, you can read more about it here: (https://parquet.apache.org/).

What about a BigQuery (https://cloud.google.com/bigquery) source?

```
batch_bigquery_source = BigQuerySource(
table_ref="gcp_project:bq_dataset.bq_table",
event_timestamp_column="event_timestamp",
created_timestamp_column="created_timestamp",
)
```

Feast ensures that the source complies with the schema of the feature table. We will look at the feature table next. These specified data sources can then be included inside a feature table specification and registered to Feast Core.

6.3.4.3 Feature tables

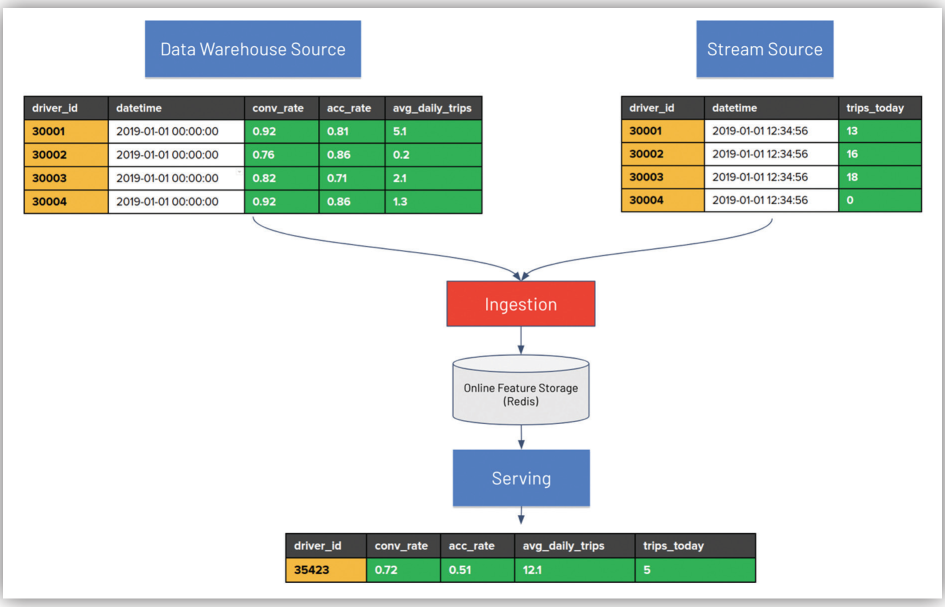
Feature tables represents a schema and a logical means of grouping features, data sources, and other related metadata. It essentially groups the features together and describe how they can be retrieved.

They have the following purposes:

- Feature tables define the location and properties of data sources (You have seen the four sources in the previous section).

- Feature tables creates the database structure to store the feature values.
- The attached data sources enable feast to find and ingest feature data into stores within feast.
- Feature tables ensure that the data is efficiently stored during ingestion by providing a grouping mechanism of feature values that occur on the same event timestamp.

The following figure shows the feature tables along with the ingestion process:



driver_id	datetime	conv_rate	acc_rate	avg_daily_trips
30001	2019-01-01 00:00:00	0.92	0.81	5.1
30002	2019-01-01 00:00:00	0.76	0.86	0.2
30003	2019-01-01 00:00:00	0.82	0.71	2.1
30004	2019-01-01 00:00:00	0.92	0.86	1.3

driver_id	datetime	trips_today
30001	2019-01-01 12:34:56	13
30002	2019-01-01 12:34:56	16
30003	2019-01-01 12:34:56	18
30004	2019-01-01 12:34:56	0

driver_id	conv_rate	acc_rate	avg_daily_trips	trips_today
35423	0.72	0.51	12.1	5

Figure 6.19: *Feature table with ingestion process*

To instantiate a feature table in Python API, we need to be aware of the fields or the structure of feature tables:

- **Name:** Name of feature table. This name must be unique within a project.
- **Entities:** List of entities to associate with the features defined in this feature table. Again, recall that entities are used as lookup keys when retrieving features from a feature table.
- **Features:** List of features within a feature table.
- **Labels:** Labels are arbitrary key-value properties that can be defined by users.
- **Max age:** Max age affects the retrieval of features from a feature table. Age is measured as the duration of time between the event timestamp of a feature and the lookup time on an entity key used to retrieve the feature. Feature

values outside max age will be returned as unset values. Max age allows for the eviction of keys from online stores and limits the amount of historical scanning required for historical feature values during retrieval.

- **Batch source:** The batch data source from which feast will ingest feature values into stores. This can either be used to back-fill stores before switching over to a streaming source, or it can be used as the primary source of data for a feature table.
- **Stream source:** The streaming data source from which you can ingest streaming feature values into feast. Streaming sources must be paired with a batch source containing the same feature values. A streaming source is only used to populate online stores. The batch equivalent source that is paired with a streaming source is used during the generation of historical feature datasets.

Now that we understand the fields, let us build the feature table as follows:

1. The first step is to define the features and the entities. You already know that Entity defines the primary key(s) associated with one or more feature tables. The entity must be registered before declaring the associated feature tables:

```
driver_id = Entity(name="driver_id", description="Driver
identifier", value_type=ValueType.INT64)
```

2. Next comes the feature definition. You already know about features:

```
# Daily updated features

acc_rate = Feature("acc_rate", ValueType.FLOAT)

conv_rate = Feature("conv_rate", ValueType.FLOAT)

avg_daily_trips = Feature("avg_daily_trips", ValueType.INT32)


# Real-time updated features

trips_today = Feature("trips_today", ValueType.INT32)
```

3. It is time to initialize the feature tables as follows:

```
driver_statistics = FeatureTable(

    name = "driver_statistics",

    entities = ["driver_id"],

    features = [

        acc_rate,

        conv_rate,
        avg_daily_trips
    ],
    batch_source=FileSource(
        event_timestamp_column="datetime",
        created_timestamp_column="created",
        file_format=ParquetFormat(),
        file_url=driver_statistics_source_uri,
        date_partition_column="date"
    )
)

driver_trips = FeatureTable(
    name = "driver_trips",
    entities = ["driver_id"],
    features = [
        trips_today
    ],
    batch_source=FileSource(
        event_timestamp_column="datetime",
        created_timestamp_column="created",
        file_format=ParquetFormat(),
        file_url=driver_trips_source_uri,
        date_partition_column="date"
    )
)
```

4. With the entities and the feature table are initialized, we need to register it with the feast core:

```
client.apply(driver_id)
client.apply(driver_statistics)
client.apply(driver_trips)
```

5. To view the feature tables specifications, you can use the get_feature_table function:

```
print(client.get_feature_table("driver_statistics").to_yaml())
print(client.get_feature_table("driver_trips").to_yaml())
```

To ensure that you are not lost by the details, *figure 6.20* gives a visual representation of the relationships between different objects:

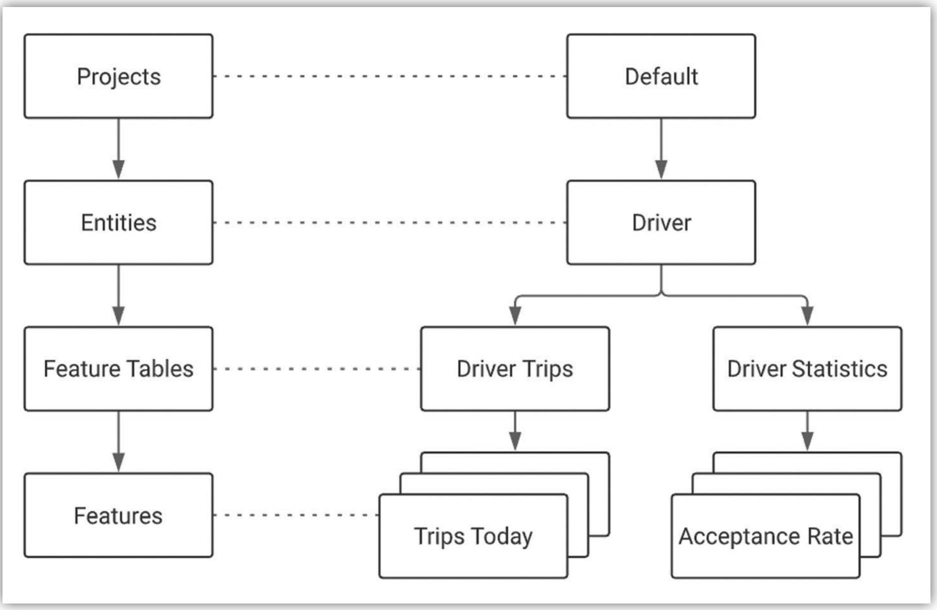


Figure 6.20: *Relationship between different objects*

Now that the feature tables are in place, we will go through the code for ingesting the data. If the feature table specification is complied, feast doesn't care about how the batch source is populated. This means that any existing ETL tools can be used for the purpose of data ingestion. As an alternative, feast offers SDK to ingest a Panda DataFrame to the batch source:

```
import pandas as pd

import numpy as np

from datetime import datetime


--1

def generate_entities():

    return np.random.choice(999999, size=50, replace=False)
```

```
--2

def generate_trips_data(entities):

    df = pd.DataFrame(columns=["driver_id", "trips_today",
    "datetime", "created"])

    df['driver_id'] = entities

    df['trips_today'] = np.random.randint(0, 500, size=50).
    astype(np.int32)

    df['datetime'] = pd.to_datetime(

            np.random.randint(

                datetime(2020, 2, 10).timestamp(),

                datetime(2020, 2, 20).timestamp(),

                size=50),

        unit="s"

    )

    df['created'] = pd.to_datetime(datetime.now())

    return df


--3

def generate_stats(entities):

    df = pd.DataFrame(columns=["driver_id", "conv_rate", "acc_
    rate", "avg_daily_trips", "datetime", "created"])

    df['driver_id'] = entities

    df['conv_rate'] = np.random.random(size=100).astype(np.float32)

    df['acc_rate'] = np.random.random(size=100).astype(np.float32)

    df['avg_daily_trips'] = np.random.randint(0, 500, size=50).
    astype(np.int32)

    df['datetime'] = pd.to_datetime(

            np.random.randint(
                datetime(2020, 2, 10).timestamp(),
                datetime(2020, 2, 20).timestamp(),
                size=50),
        unit="s"
    )
    df['created'] = pd.to_datetime(datetime.now())
    return df


--4
 entities = generate_entities()
stats_df = generate_stats_data(entities)
trips_df = generate_trips_data(entities)
--5
client.ingest(driver_statistics, stats_df)
client.ingest(driver_trips, trips_df)
```

Let us unpack the code as follows:

1. We are going to generate 50 dummy entities. Recall that entities are the keys to lookup features. The function returns an array of numbers.
2. We are initializing a pandas DataFrame to generate the data. The entities are the driver IDs that are key to the features. Note the event timestamp and the created timestamp column (Can you recall what they were?).
3. It is the same as step 2 but data is generated from trips entity.
4. We then initialize the entities which is array here and populate the data fame.
5. Next, we call the ingest method to batch load the feature data into the feature table. The feature tables driver_statistics and driver_trips were initialized. Note that the source is a pandas DataFrame, but it will be a file as well. At the time of writing this book, following file types are supported:

    a. Parquet

    b. csv

    c. Son

Next, we will take a look at another scenario where we will populate the online storage with the batch storage. To populate the online storage, we will start a Spark job via the Feast Python SDK that will extract the feature from the batch source and load it in online storage:

```
job = client.start_offline_to_online_ingestion(

    driver_statistics,

    datetime(2020, 10, 10),

    datetime(2020, 10, 20)

) --1


job.get_status() --2
```

- `start_offline_to_online_ingestion` launches ingestion job from the batch source to online store for a given feature table. The two datetime stamp are the lower and upper datetime boundary.
- The job takes some time to complete, and you can check the status via the get_status method. If the job fails, then it raises an exception of type: `feast.pyspark.abc.SparkJobFailure`.

Once the job is complete, we need to retrieve the results from the online store:

```
entities_sample = np.random.choice(entities, 10, replace=False) --1

entities_sample = [{"driver_id": e} for e in entities_sample] --2


features = client.get_online_features(

    feature_refs=["driver_statistics:avg_daily_trips"],

    entity_rows=entities_sample).to_dict() --3

pd.DataFrame(features) --4
```

Let us unpack the code as follows:

1. We randomly sample 10 entities from the full entities array. The entities was generated via the generate entities method.
2. We build a list of driver ids. Only these entities features will be retrieved. The output of this step look as follows. This is required because the get_

online_feature method expects a list of dictionaries where each key-value is an entity-name and entity-value pair:

```
[{'driver_id': 845144},
 {'driver_id': 987858},
 {'driver_id': 230942},
 {'driver_id': 447572},
 {'driver_id': 223221},
 {'driver_id': 405388},
 {'driver_id': 822999},
 {'driver_id': 30832},
 {'driver_id': 999547},
 {'driver_id': 520871}]
```

3. We call the get_online_features to finally retrieve the features. This method retrieves the latest online feature data from Feast Serving. The feature_refs parameter represents the list of feature references that will be returned for each entity. Do note the format - feature_table:feature where feature_table and feature refer to the feature and feature table names respectively. The final response is converted to the dictionary. But why the dictionary? Well, you can use the dictionary to build a pandas DataFrame.
4. We initialize the pandas DataFrame with the dictionary. It can now be used as an input to the trained model.

We will now take a look at another scenario – How to ingest from a streaming source. To demonstrate the scenario, we will launch a Spark job that will continuously update the online store. While the feast API to ingest from a streaming source is straightforward, there is a little bit of setup required in case you are planning to practice the code on your machine. We will use Kafka as the streaming source. In case you don't have Kafka installed locally, then use the following command to install it:

```
pip install confluent_kafka
```

```
--1

import json

import pytz
import io

import avro.schema

from avro.io import BinaryEncoder, DatumWriter

from confluent_kafka import Producer


--2

KAFKA_BROKER = os.getenv("DEMO_KAFKA_BROKERS", "kafka:9092")


--3

avro_schema_json = json.dumps({

    "type": "record",

    "name": "DriverTrips",

    "fields": [

        {"name": "driver_id", "type": "long"},

        {"name": "trips_today", "type": "int"},

        {

            "name": "datetime",

            "type": {"type": "long", "logicalType": "timestamp-micros"},

        },

    ],

})


--4

driver_trips.stream_source = KafkaSource(

    event_timestamp_column="datetime",

    created_timestamp_column="datetime",

    bootstrap_servers=KAFKA_BROKER,
    topic="driver_trips",
    message_format=AvroFormat(avro_schema_json)
)

client.apply(driver_trips)  --5

job = client.start_stream_to_online_ingestion(
    driver_trips
)  --6

--7
def send_avro_record_to_kafka(topic, record):
    value_schema = avro.schema.parse(avro_schema_json)
    writer = DatumWriter(value_schema)
    bytes_writer = io.BytesIO()
    encoder = BinaryEncoder(bytes_writer)
    writer.write(record, encoder)

    producer = Producer({
        "bootstrap.servers": KAFKA_BROKER,
    })
    producer.produce(topic=topic, value=bytes_writer.getvalue())
    producer.flush()

--8
for record in trips_df.drop(columns=['created']).to_dict('record'):
    record["datetime"] = (
        record["datetime"].to_pydatetime().replace(tzinfo=pytz.utc)
    )


    send_avro_record_to_kafka(topic="driver_trips", record=record)
```

Let us unpack the code as follows. The feast code are a few API calls, but we need to set up a serialization system:

1. **Imported modules:**
   a. The Python pytz module provides the date-time conversion functionalities. The module allows accurate and cross platform time zone calculations in Python. This is required because our feature tables use date-time fields. Remember that we are pushing data between different systems and a consistent and an accurate date time representation is required.
   b. Next is the Avro module. Apache Avro is an open-source Apache data serialization system. It offers a rich data structure and a compact binary format. Avro relies on schemas. When the Avro data is read, the schema used when writing it is always present. This permits each datum to be written with no per-value overheads, making serialization both fast and small. This also facilitates use with dynamic, scripting languages, since data, together with its schema, is fully self-describing. Think of Avro as a language-independent data serialization system that is used by big data systems like Spark and Kafka.
   c. `confluent_kafka` is a Python client for Kafka. In case you are hearing about Kafka for the first time, then Apache Kafka is a distributed streaming platform that can publish, subscribe, store and process messages in real-time. Its pull-based architecture reduces the pressure on the service with a heavy load and makes it easy to scale. It moves a huge amount of data from the source to the destination with low latency.
2. Change this only if your Kafka broker address is different. Don't change this if you are using feast provided Docker containers.
3. We define the Avro schema using JSON. This facilitates implementation in language that already has JSON libraries and for Python, we have already imported the JSON module.
4. We define the KafkaSource. You have already seen some of the other source types such as - file source and BigQuery source. The interesting new parameter is message_format. We have specified the message format as AvroFormat to send avro messages to Kafka.

5. With the Kafka source defined, we need to update the feature table to add the new Kafka source.
6. It is time to kick off the streaming ingestion job.
7. To send the avro message to Kafka, we define a function. This is not feast code and if you have knowledge of Kafka, there is nothing new here:
   a. The DatumWriter class responsible for translating the given data object into an avro record with a given schema.
   b. The next two lines encode an Avro object to a byte array.
8. In the simplest view, there are three players in kaka - producer, broker, and consumers.
   a. Producers - Produce messages to the topic of choice.
   b. Topics are logs that receive data from the producers and store them across their partitions. Producers always write new messages at the end of the log.
9. Next, we change the date time field in the record into UTC format and call the function (step 7) to send to Kafka.

To read the features, you use the get_online_features that you saw in the preceding code.

6.4 Conclusion

Storing precomputed feature values in warehouse is not a new idea, however, they fall short of new expectations around speed, consistency, privacy, and security. With the proliferation of ML projects in an organization, the need for feature governance is rising. Feature store is an important step in that direction. There are not many choices in the open sources space other than hop works and feast. While the beginning is good, the project desires good documentation and developer support. It has high dependency on a lot of other third-party projects, making it a little challenging to effectively maintain and monitor in production. But with the potential and excitement around feature stores, things are bound to improve. We are nearing the end of the book and the last chapter would be focused on learning a tool to expose the model as an API endpoint. For an ML project to be useful and successful, it must be easy to be used by others. See you in the next chapter. Until then, happy learning!

6.5 Points to remember

- The feature store could be imagined as a warehouse of features.

- It is a central vault for storing documented, access-controlled features.
- Feast aims to:
  - o Provide a unified means of managing feature data from a single person to large enterprises.
  - o Provide scalable and performant access to feature data when training and serving models.
  - o Provide consistent and point-in-time correct access to feature data.
  - o Enable discovery, documentation, and insights into your features.
- At a high level, feast has four functional components which are as follows:
  - o **Register –** The central registry for feature re-use and collaboration.
  - o **Ingest –** It enables feature data ingestion from batch and stream source.
  - o **Serve** - A feature retrieval interface from the feature stores.
  - o **Monitor** - Monitoring features in feature stores.
- Feature has architecture components.
- Feast core – This is the central registry for feature and entity definitions.
- Feast job service - Data loading jobs from source into stores.
- Feast online serving - Low latency access to features.
- Feast Python SDK - Python SDK to interact with feast.
- Online store - Online store to store the latest feature values.
- Offline store - Offline store to persist batch data.
- Storage - Backend for storing the features values.
- Load feature data from both batch (Big query and GCS) and streaming sources like Kafka.
- The four concepts of feast are as follows:
  - o **Projects:** This is the top-level namespace that uniquely identifies a project. Each project maps to a completely independent environment in feast. The users can only work in a single project at a time.
  - o **Entities:** Entities are the objects in a project that have features. Think of Entities as domain objects like customers, products, transactions, etc.

  - o **Feature tables:** This defines a group of features that occur on a specific entity.
  - o **Features:** The individual feature within a feature table that highlights the attributes of an entity.
- Sources describe the external data that would become the feature data inside the feature store.
- Feast supports two types of data sources - batch source and stream source.

6.6 Questions

1. What are the four components of feast?
2. What are two types of stores available in feast?
3. What storage is used for model serving?
4. What storage is used for model training?
5. Which two types of data sources are supported by feast?

CHAPTER 7
Serving ML as API

The real value of ML is only realized when the model is used by people beyond the data science team. These days, it could be web or mobile apps. The applied team would want to use the predictions from the ML without getting the hassle of learning ML. API is a wonderful scalable abstraction to provide this feature. For the consumers, the ML model will be an API endpoint that can integrate into their applicable function flow.

The focus of this chapter will be on learning how we can deploy ML model as an API. We will use fastAPI which is a modern, high-performance Python web framework that is perfect for building RESTful APIs. fastAPI can handle both synchronous and asynchronous requests and has built-in support for data validation, JSON serialization, authentication, and authorization.

Structure

In this chapter, you will learn the following:

- Introducing fastAPI
- Deplyoing ML model as fastAPI
- Scaling ML serving fastAPI and Ray Serve

Objective

By the end of this chapter, you will know the following:

- Building API with fastAPI
- Deploying model as fastAPI endpoint
- Scaling fastAPI application with Ray Serve

7.1 Introducing fastAPI

fastAPI (https://fastapi.tiangolo.com/) is a relatively new web framework for building restful API with Python 3.6+. If you are currently using Flask, then you may wonder about the reasons to switch to fastAPI. The following are a few reasons to get you excited:

- **Speed:** fastAPI is one of the fastest Python web frameworks. In fact, the framework author claims the speed to be at par with node.js and GO.
- **Type hint:** Type hints are special syntax that allows declaring the type of a variable. With type hinting, there are free data validations and conversions.
- **Standard based:** fastAPI is compatible with an open specification like openAPI (https://swagger.io/resources/open-api/).

7.1.1 Installation of fastAPI

Installing fastAPI is pretty straightforward. You can install it via pip:

```
pip install fastapi
```

In order to run the API, you also need a server. For fastAPI, it will be the ASGI server with two options – Uvicorn (https://www.uvicorn.org/) or Hypercorn (https://gitlab.com/pgjones/hypercorn). Let us install Uvicorn.

```
pip install uvicorn[standard]
```

In order to verify the installation, we will quickly test a hello world application:

```
from typing import Optional
from fastapi import FastAPI


app = FastAPI()
```

```
@app.get("/")

def home_page():

    return {"Hello": "World"}
```

Since it is a webAPI, we need to launch the webserver. Use the following command to run the Uvicorn server. Make sure that the preceding code is present in the same folder:

```
uvicorn main:app --reload
```

If the server runs successfully, then open your browser and go to http://127.0.0.1:8000/. If you see the hello world message, then congratulations! Let us the unpack the Uvicorn command:

- **`main:`** The file main.py (Python file).
- `app:` The object created inside the main.py (app = FastAPI()). It tells Uvicorn where to find the fastAPI ASGI application.
- `--reload:` Restart the server in case there is a code change. It should only be used in the development settings.
- **`--worker:`** The number of worker process
- **`--host:`** Server address
- **`--port:`** Port on which the server is listening on.

7.1.2 fastAPI concepts

fastAPI features are powered by two open source projects. These are as follows:

- **Starlette (https://www.starlette.io/)-** Starlette is a lightweight ASGI framework to build asyncio services. But wait, what is a ASGI framework?

  ASGI stands for **Asynchronous Server Gateway Interface** and is the successor for **WSGI (Web Server Gateway Interface)**. These are specifications to standardize the communication between the Python web server and web application. WSGI applications are a single, synchronous callable that take a request and return a response; this doesn't allow for long-lived connections like you get with long-poll HTTP or WebSocket connections.

  Flask, being a WSGI server, reflects those limitations and is therefore not a good choice for production.
- **Pydantic (https://pydantic-docs.helpmanual.io/) -** Pydantic enforces type hints at runtime and provides user-friendly errors when the data is invalid. Type hints are special syntax introduced with Python 3.5 that allow type

declarations for a variable. If you already know about type hints, then feel free to skip the Pydantic section. The following code shows an example with type hints:

```
def get_life_details(name:str, age:int) -> str:
    # name_age = f'{name} is {age} old.'
    years_left = name + "has " + str(100-age) + " years left."
    return years_left


print(get_life_details(name="Foo",age=35))
```

`The name:` str syntax indicates that the name argument should be of type str. The age: int indicates that the age argument should be of type int. The -> syntax indicates the get_life_details() function will return a string. Beyond str and int, all the other standard Python types can be used as follows:

- int
- float
- bool
- bytes

It is also possible to provide type hint for generic types – dict, list, set, and tuple:

from typing import Dict, List, Tuple, Set

```
def process_items(items : Dict[str,int]) --1
…………


def process_items(items : List[str]) --2
…………..


def process_items(items : Tuple[str, str, int]) --3
…………..


def process_items(items : Set[bytes]) --4
…………..
```

Let us unpack the function definitions with the type hints as follows:

1. The function accepts a dictionary with the key type as str and value type as int.
2. The function accepts a list of strings.
3. The function accepts a tuple of string, string, and integer type.
4. The function accepts a set of bytes.

It's no surprise that you can declare a class as the type of a variable. So where does Pydantic come into the picture? Well, pydantic enforces type hints at run time and provides user-friendly errors when the data is invalid. The following example would be helpful here:

```
class User(BaseModel): --1
    id: int
    name: str
    timestamp: Optional[datetime] = None
    friends: List[int] = []


external_data = { --2
    "id": "123",
    "name" : "foo"
    "timestamp": "2020-08-03 10:22",
    "friends": [1, "2", b"3"],
}
user = User(**external_data) --3
--4
print(user)
print(user.id)
```

Let us unpack the preceding code as follows:

1. We declare a new class that inherits from **`BaseModel`**. Pydantic uses the built-in type hinting to determine the data type of each variable.
   a. **`id`** – An integer variable representing the ID of the record. This is a required field since no default value has been specified.
   b. **`name`** – The name of the user with string type. It is a required field as no default value has been provided.
   c. **`timestamp`** – A datetime field representing the signup time. It defaults to None and hence is not a required field.

   d. **`friends`** – A list of integers representing the id of other users.
2. We define a dictionary with keys being the user class members.
3. The dictionary is unpacked (** unpacks a dictionary) to initialize a user.
4. When printed, you will see that the ID has been converted to integer even though it was provided as string in the external_data dictionary. Likewise, bytes are automatically converted into integers for the friends list.

```
id=123 name='foo' timestamp=datetime.datetime(2017, 6, 1, 12, 22)
friends=[1, 2, 3]
```

   fastAPI leverages the Pydantic feature for the following:

   - **Type checks**
   - **Define requirements** - from request path parameters, query parameters, headers, bodies, dependencies, etc.
   - **Convert data** – from the request to the required type.
   - **Validate data** – coming from the request.
   - **Document** – The API using OpenAPI.

Let us go ahead and modify the first code block as follows to include a parameter:

```
from typing import Optional
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def home_page():
    return {"Hello": "World Alok"}

@app.get("/product/{product_id}")
def read_product(product_id: int, q: Optional[str] = None):
    return {"product_id": product_id, "q": q}
```

We have added a new path that accepts parameters as follows:

- It supports HTTP method (notice the @app.get decorator)
- The path /product/{product_id} has a path parameter product_id that should be an int.
- The path /product/{product_id} has an optional str query parameter q.

Go to the browser and type http://127.0.0.1:8000/product/5?q=someproduct after you run the Uvicorn server. You would get a JSON response like this:

```
{"product_id":5,"q":"someproduct"}
```

Since the q parameter is optional, you can skip it as well in the URL http://127.0.0.1:8000/product/5. The output would look as follows:

```
{"product_id":5,"q":null}
```

What happens when you pass a non-numeric value in the product_id parameter? Try searching http://127.0.0.1:8000/product/A in your browser. The data type validation is automagically done by the framework:

```
{"detail":[{"loc":["path","product_id"],"msg":"value is not a valid
integer","type":"type_error.integer"}]}
```

This doesn't end here. You also get the interactive documentation of your API based on openAPI specification. Open http://127.0.0.1:8000/docs in your browser to read the documentation:

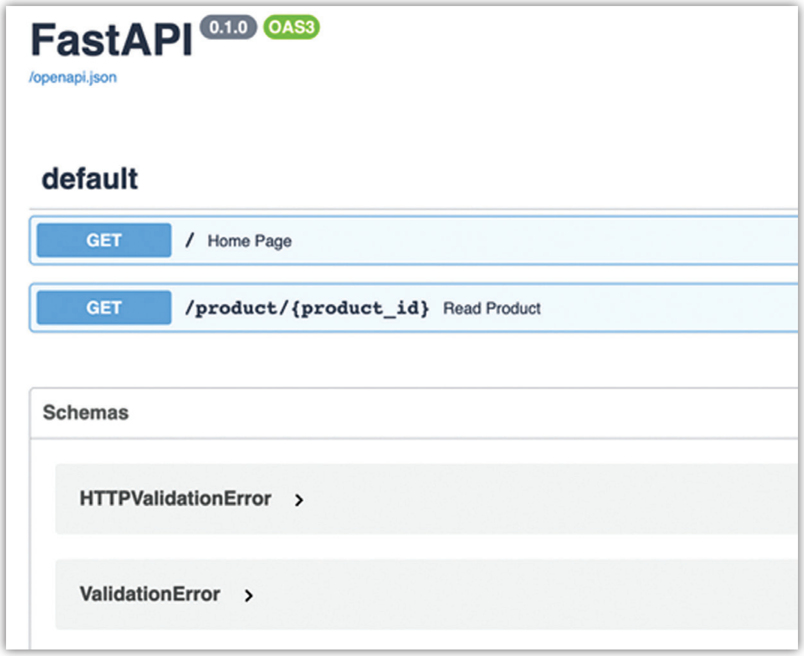
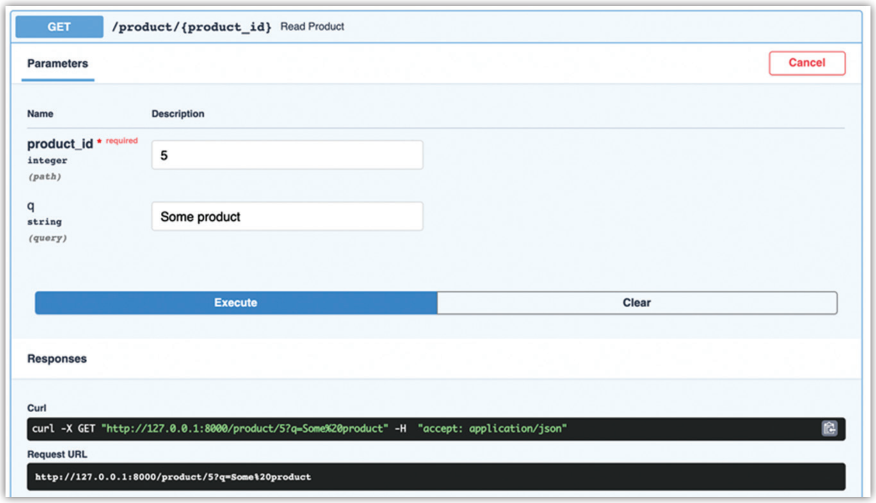


Figure 7.1: *Interactive API documentation based on swagger UI (https://github.com/swagger-api/swagger-ui)*

Since it is interactive, you can try the API. *Figure 7.2* shows the interactive request page for the product path as follows:



GET /product/{product_id} Read Product

Parameters

Name	Description
product_id * required integer *(path)*	5
q string *(query)*	Some product

Responses

Curl

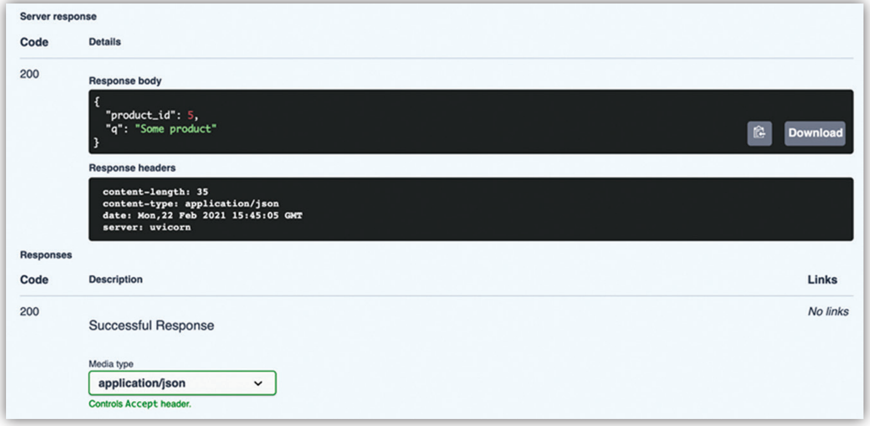
```
curl -X GET "http://127.0.0.1:8000/product/5?q=Some%20product" -H  "accept: application/json"
```

Request URL

```
http://127.0.0.1:8000/product/5?q=Some%20product
```

Figure 7.2: *Request page*

The following image shows the response page:



Server response

Code Details

200

Response body

```
{
  "product_id": 5,
  "q": "Some product"
}
```

Response headers

```
content-length: 35
content-type: application/json
date: Mon,22 Feb 2021 15:45:05 GMT
server: uvicorn
```

Responses

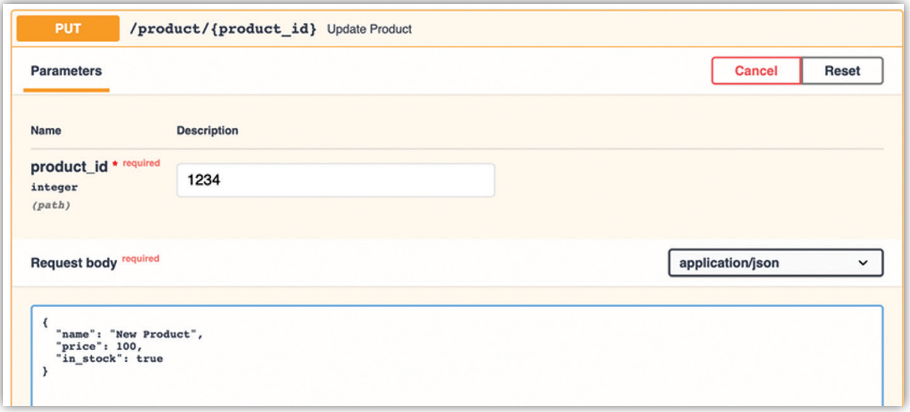
Code	Description	Links
200	Successful Response Media type application/json Controls Accept header.	*No links*

Figure 7.3: *Response page.*

Let us modify the preceding code to add a PUT method. This time, the body will be defined as Pydantic data type. The following code only shows the put method for brevity. In order to try the code, add this method in the main.py file:

```
from pydantic import BaseModel


app = FastAPI()


class Product(BaseModel):
    name: str
    price: float
    in_stock: Optional[bool] = None

@app.put("/product/{product_id}")
def update_product(product_id: int, product: Product):
    return {"product_name": product.name, "product_id": product_id}
```

In order to test the code, go to http://127.0.0.1:8000/docs and open the PUT request section. Provide the necessary values and execute it as follows:



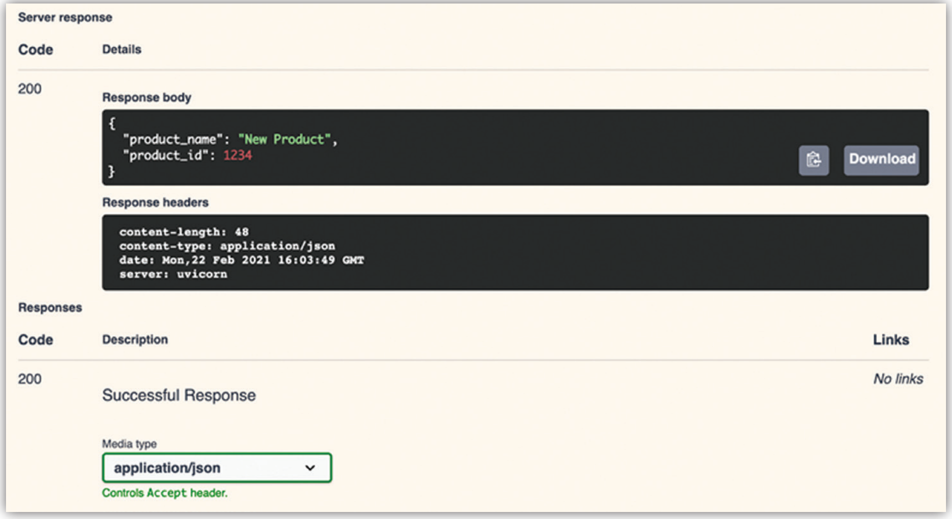
product_id * required
integer
(path)
1234

Request body required
application/json

```
{
  "name": "New Product",
  "price": 100,
  "in_stock": true
}
```

Figure 7.4: Put request

The following image shows the response after the put request:



200

Response body

```
{
  "product_name": "New Product",
  "product_id": 1234
}
```

Response headers

```
content-length: 48
content-type: application/json
date: Mon,22 Feb 2021 16:03:49 GMT
server: uvicorn
```

200 Successful Response No links

Media type: application/json

Controls Accept header.

Figure 7.5: Put request response

This brings an end to a quick tour of fastAPI and I would encourage you to read the documentation to further gain an understanding of the framework. It is time to wrap our trained model in the fastAPI endpoint.

7.2 Deploying ML model as fastAPI

We are all set to expose a trained model as an API endpoint. For the walkthrough, we will build a classifier for the iris dataset (https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html). This data sets consists of three different types of irises' (Setosa, Versicolour, and Virginica) petal and sepal length, stored in a 150×4 numpy.ndarray. The rows being the samples and the columns being: Sepal Length, Sepal Width, Petal Length, and Petal Width features. The following code shows the complete code for it. I have used the same Python file for training and deploying to keep things simple. For production situations, the training will be done in a separate workflow, and in the API, you need to load the trained model as follows:

```
--1
from typing import Optional
from fastapi import FastAPI
from pydantic import BaseModel
--2
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
import numpy as np


--3
# Loading Iris Dataset
iris = load_iris()


--4
# Getting features and targets from the dataset
X = iris.data
y = iris.target
```

```
--5
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.5)

--6
logreg=LogisticRegression(solver='lbfgs', multi_class='auto')

# Fitting our Model on the dataset --7
logreg.fit(X_train,y_train)

app = FastAPI() --8

--9
class request_body(BaseModel):
    sepal_length : float
    sepal_width : float
    petal_length : float
    petal_width : float

--10
@app.post("/predict")
def predict(data : request_body): --11
    test_data = [[ --12
            data.sepal_length,
            data.sepal_width,
            data.petal_length,
            data.petal_width
    ]]
    class_idx = logreg.predict(test_data)[0] --13
    return { 'class' : iris.target_names[class_idx]} --14
```

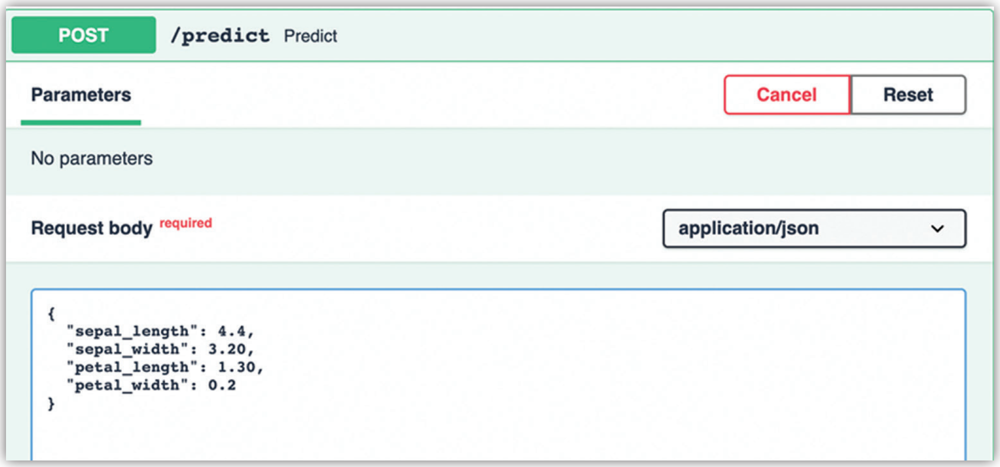
Let us unpack the code as follows. It would feel like revision since you have already seen all the parts:

1. We import the fastAPI modules.

2. We import the sklearn modules to train the classifier.
3. We load the iris dataset. It's because of this utility function that I prefer to leverage datasets available with sklearn.
4. We extract the features and the targets.
5. We split the dataset into train and test.
6. To train the classifier, we are using logisticRegression.
7. Finally, we fit the model on the dataset. These are all standard steps which you would have done lots of times. Again, to re-iterate, the training step will always be done in a separate workflow and in the API, you load the saved model.
8. With the model trained, it is now time to set up the API. We create a new instance of fastAPI.
9. We define the request object schema using Pydantic. With Pydantic, the request payload JSON is converted into the request_body object schema. This means all the payload properties will be available to us with the dot notation.
10. We define the path for the end point. If you are running it locally, then it maps to http://127.0.0.1:8000/predict
11. We define the request handler for the path. I hope you see the beauty of fastAPI. fastAPI, by using Pydantic, maps the request headers to an object model that we can easily use in the code. Since it uses type hints, any mismatch in the data type would get pointed out.
12. Once we have the test data, it is standard sklearn steps. We initialize the test data.
13. Call the predict method.
14. Return the predicted class of the test data.

In order to test the API, you can use the swagger UI. Go to http://127.0.0.1:8000/docs and expand the predict endpoint. Click on the "try it out" button. Update the request body with the following values:

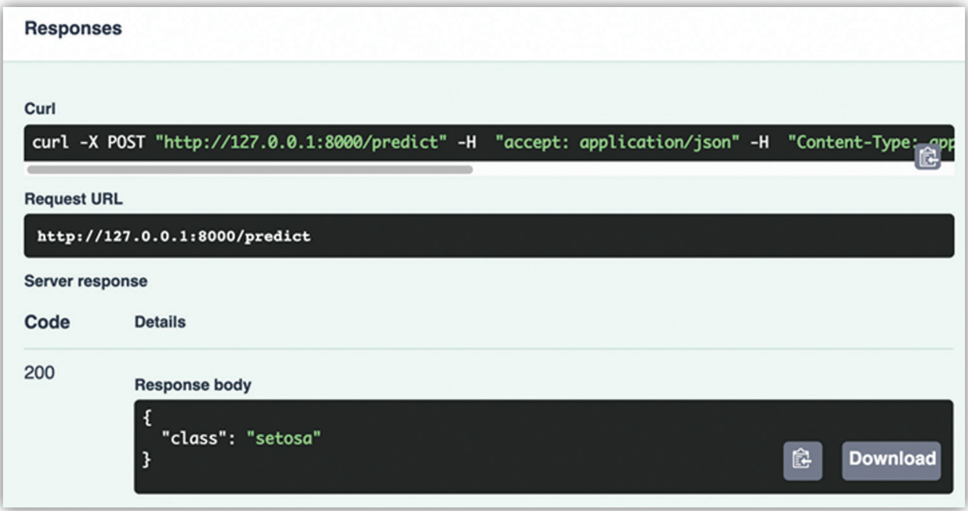
```
{

  "sepal_length": 4.4,

  "sepal_width": 3.20,

  "petal_length": 1.30,

  "petal_width": 0.2

}
```



```
{
  "sepal_length": 4.4,
  "sepal_width": 3.20,
  "petal_length": 1.30,
  "petal_width": 0.2
}
```

Figure 7.6: *Request for predict API*

Once executed, you would get the predicted class in the response as follows:



```
curl -X POST "http://127.0.0.1:8000/predict" -H  "accept: application/json" -H  "Content-Type: app
```

```
http://127.0.0.1:8000/predict
```

```
{
  "class": "setosa"
}
```

Figure 7.7: *Predict API call response*

7.3 Scaling ML serving with fastAPI and Ray Serve

You already know that fastAPI is a high performance Python web framework to serve ML models. With Ray Serve (https://docs.ray.io/en/master/serve/index.html), you can scale it to large machine clusters. Ray Serve a is model serving library built on top of ray (https://ray.io/). At a high level, think of ray as a distributed framework that can scale your ML application from your laptop → multi-core machine → any cloud provide r → a Kubernetes cluster.

Ray serve runs on top of ray and abstracts away the distributed aspect of the system. Essentially, as a user, you don't need to have a distributed system knowledge to scale your application. An architecture diagram would be helpful to understand the relationship between fastAPI, ray serve, and ray:

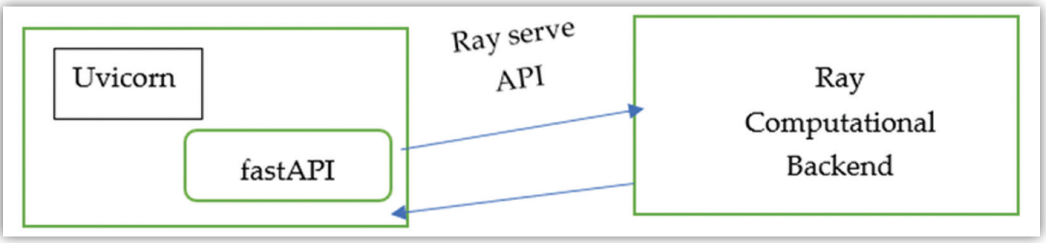


Figure 7.8: *fastAPI with ray serve architecture*

In *figure 7.8*, we have offloaded the inference to a ray cluster and interacted with the cluster via the Ray Serve API. The other visual way of thinking about this off-loading is shown in *figure 7.9*:

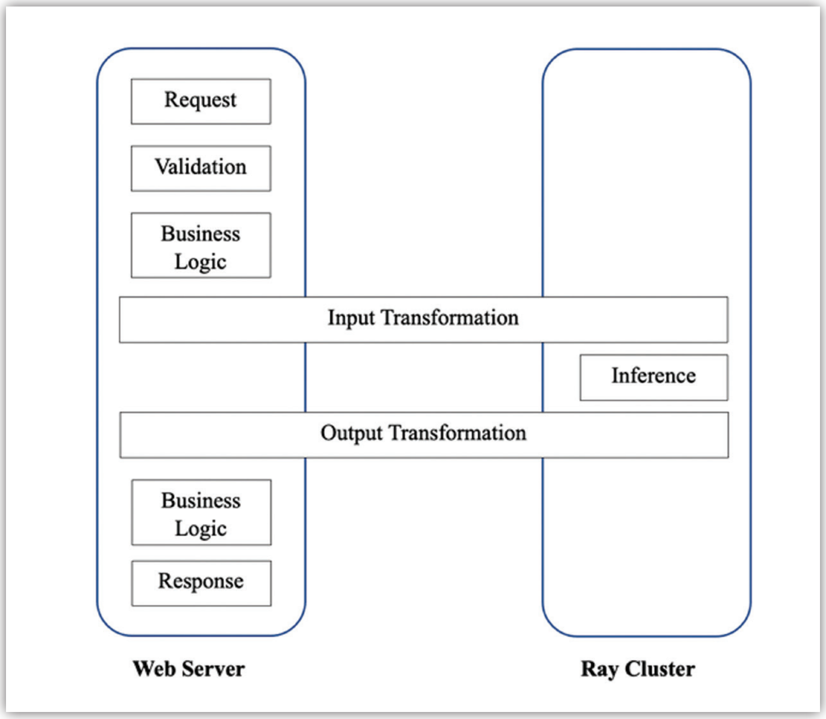


Figure 7.9: *Inference offloading*

In *figure 7.9*, when the request comes to the web server, it parses the input, validates it, and applies some business logic to it. However, for the ML inference, it offloads it to the external service which is the ray cluster here. If required, some transformation can be applied before and after the inference. Finally, the response from the web server is returned to the caller. Some of the benefits of this separation of concern immediately stand out. On the web server side, the inference is reduced to a call. Models can be updated without impacting the web server. It hides the inference backend from the web server developer.

One important takeaway from this architecture is to realize the fact that web serving is not equal to model serving. At the uber level, the web serving and model servings may feel the same. Both of them receive input and produce some output. However, under the hood, they are very different. The concurrency and scalability techniques applicable in the traditional serving will not work. A web server is network-bounded; it can handle thousands of requests at a time. However, an ML model is memory and compute bound. A heavy ML model throughput will be quite less compared to web server throughputs. Due to a single global variable with the loaded model, you can only run one computation at a time. You can think of load balancing in order to solve the problem but that is also quite different between the two.

In the web serving world, all the machines will have the same configuration and they can be replicated easily and any query can be executed on any of the servers. However, you can't imagine doing the same for ML models. Imagine putting a state of art CNN architecture for image processing and transformer for NLP tasks in one process. Both of them are resource intensive. In order to scale models inference, we would need fine-grained replications at the model level. If we were to imagine the model serving requirement as list, then only a few of them would be fulfilled by the web server.

- Deliver prediction – yes
- Framework agnostic – yes
- Business logic – yes
- Scaling – no
- Batching – no
- High concurrency – no
- Low latency – no

While there is value in offloading the inference, it come with complexity. Fortunately, Ray Serve along with ray cluster can manage the offloaded inference service on behalf of our API. You may wonder how ray serving is different from TFX (Tensorflow serving). In other words, why would you consider ray over TFX, Torch serve, etc.? Well, you may want to consider it for the following two reasons:

- **Framework-agnostic:** With a single toolkit, you can a serve different model (deep learning models, classical ML models) built with different tech stacks (TensorFlow, PyTorch, Scikit-learn).
- **Python-first:** All the model serving can be managed via Python. You don't need those clumsy YAML or JSON configs.

Now that we understand the value and purpose of Ray Serve , let us go through an implementation. We will stick to the iris classifier that we exposed as a fastAPI endpoint earlier to keep things simple. In order to run the code, please make sure to install Ray Serve. .

```
pip install "ray[serve]"
import ray
from ray import serve
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
import numpy as np

from fastapi import FastAPI, Request
from pydantic import BaseModel
app = FastAPI()

serve_handle = None

--1
@app.on_event("startup")  # Code to be run when the server starts.
async def startup_event():
    ray.init(address="auto") # Connect to the running Ray cluster. --2
    client = serve.start() # Start the Ray Serve client. --3

    class sklean_LRModel: --4
```

```
        def __init__(self) -> None:
            self.iris = load_iris()
            X = self.iris.data
            y = self.iris.target

            X_train,X_test,y_train,y_test=train_test_split(X,y,test_
            size=0.5)

            self.logreg=LogisticRegression(solver='lbfgs', multi_
            class='auto')

            # Fitting our Model on the dataset
            self.logreg.fit(X_train,y_train)

        async def __call__(self, request):
            payload = request
            input_vector = [[
            payload._data.sepal_length,
            payload._data.sepal_width,
            payload._data.petal_length,
            payload._data.petal_width
            ]]

            class_idx = self.logreg.predict(input_vector)[0]
            return { 'class' : self.iris.target_names[class_idx]}


    client.create_backend("sklean_LRModel:v1", sklean_LRModel) --5
   client.create_endpoint("iris_classifier", backend=" sklean_LRModel:v1")
--6

    global serve_handle
    serve_handle = client.get_handle("iris_classifier") --7
```

```
class request_body(BaseModel):
    sepal_length : float
    sepal_width : float
    petal_length : float
    petal_width : float


--8
@app.post("/predict")
async def predict(data: request_body):
    return await serve_handle.remote(data) --9
```

Before you run the code, you need to run the long running ray cluster. The command for it is straightforward. If you run this on your local machine, then your machine would be the head node. In order to scale, you may run this cluster in cloud, but for the application, it would be transparent:

ray start –head

Since fastAPI is the front end server, we need to run the Uvicorn server as well.

uvicorn main:app --port 8080

Let us unpack the preceding code as follows:

1. There are a bunch of things that happens during the initialization of fastAPI service. You decorate a function with @app.on_event("startup") in fastAPI to designate a function to run before the application starts.
2. Calling the ray_init(address="auto") instantiates the runtime and connects it to the cluster. Since the cluster is running on the same machine. It is able to connect to the cluster. If that is not the case, then you need to specify the cluster address. You get the cluster address when you run the ray start command.
3. We initialize the Ray Serve client.
4. We define a callable class (services) to be used by the ray serve backend. The __init__() should contain the model initialization/load code. If we had picked our model, then it would be loaded here. In our example, I am running the full training as it is quick and keeps things at one place. NEVER DO THIS IN PRODUCTION. The __call__() method contains the code for the request processing. If required, you can do request pre-processing or transformation here. While we used a class, it could have been an arbitrary

function as well since everything is Python. For production code base, this callable class should be maintained in a separate Python file.

5. Next, we need to deploy this model in Ray Serve and that can be done via API. In order to scale out, Ray Serve has a concept called backend. Backend defines the implementation of the model that will handle the incoming request. Backends can have many replicas to help in scaling. To define a backend, we need a handler that you will response with for a request. It needs to accept the starlette request. Earlier, it used to be flask request. It is no surprise that you create the backend with the create_backend function. The first parameter to the function is a tag that identifies the backend. In our case, the backend has a name and a version. The second parameter is called the backend definition and could be a class or a function. As already described, if it is a class, then it needs to implement __init__ and __call__. So, how do you decide between a class and a function? As a thumb rule, you use function when the response is stateless and a class when some sort of state needs to be maintained. In our case, we have used a class. The backend can be configured by passing a config to the create_backend function. The following example would be helpful here:

   ```
   client.create_backend("sklean_LRModel:v1", sklean_LRModel, config =
   {num_replicas: 3})
   ```

   The number of replicas specifies the number of model replica to be set up in order to scale your model serving. Another example of config is to enable "batching" that allows the backend a list of requests. The code snippet is shown as follows:

   ```
   client.create_backend("sklean_LRModel:v1", sklean_LRModel, config =
   {max_batch_size: 32})
   ```

   In case your model needs GPU, then that can be specified too:

   ```
   client.create_backend("sklean_LRModel:v1", sklean_LRModel, config =
   {num_replicas: 10}, ray_actor_options = {"num_cpu":2,"num_gpu":1})
   ```

6. Now that the backend is defined which is essentially the implementation of request handling logic, we need to expose the backend as an endpoint. The two important parameters are the names of the endpoint and the backend name that you specified while calling the create_backend function. The endpoint could be exposed by a http server or can be queried by the ServeHandle interface.

7. In our code, we are using the ServeHandle interface. get_handle method returns a handle to service endpoint. You invoke the endpoint via the remote method.

8. Next, the fastAPI async endpoint is defined that accepts the http request.
9. Finally, we call the endpoint remote method. Recall that this endpoint is using ServeHandle interface.

It is important to reiterate that while I have kept the fastAPI code and ray backend in one Python file, in production code, this won't be the case. Since the API is running as a fastAPI endpoint, you can test it via the interactive documentation. Finally, if you ran the code locally then don't forget to call the ray stop command to shut down the cluster.

This was a quick tour of scaling ML serving with fastAPI and Ray Serve. I would encourage you to read the documentations to gain further understanding of the concepts and the framework.

7.4 Conclusion

The future of the model serving will be framework agnostic. As the ML adoption in the organization increases, so will be the framework used. With the rise in expectations, there is a growing need to build frameworks that allow seamless scale out across many machines, at the same time making the model upgrade transparent. Even though it's a fairly recent project, it is gaining support from a lot of premium organizations indicating the growing influence of the project.

This chapter also marks the end of this book. While I have tried to cover as much as possible, we all know that it is never sufficient. Think of a book as a torch that shines light on an idea. In order to understand the idea and use it effectively, you need to play and stay with the idea.

Hope this book has inspired and guided you to think of ML applications in a new way.

```
Happy learning!
```

7.5 Points to remember

- fastAPI is a new python web framework to build API
- The top 3 USP of fastAPI are
    - o speed
    - o Type hints
    - o Open standards
    - o ASGI server are required to run a fastAPI application

- You can use uvicorn or hypercorn ASGI for fastAPI
- fastAPI features are powered by two important libraries
  - o Starlette - Starlette is a lightweight ASGI framework to build asyncio services.
  - o Pydantic - enforces type hints at run time
  - o fastAPI documentation openAPI specification and uses swagger UI for generating the interactive documentation.
- Ray serve along with fastAPI can be used to scale ML serving
- Ray serve is a model serving library that sits on top of a ray cluster
- Ray is a distributed framework to scale ML applications
- Web serving is not equal to ML serving
- Web serving is homogenous - each server having the same code and environment
- Model serving is heterogenous - each model may have very different resources need
- ML inference offloading opens up the opportunity for fine grained scaling of ML models

7.6 Questions

1. What are the three top features of fastAPI?
2. Which two ASGI servers can you use for fastAPI serving?
3. What is the full form of ASGI?
4. What is Ray Serve?
5. What are the two important features of Ray Serve?

Index