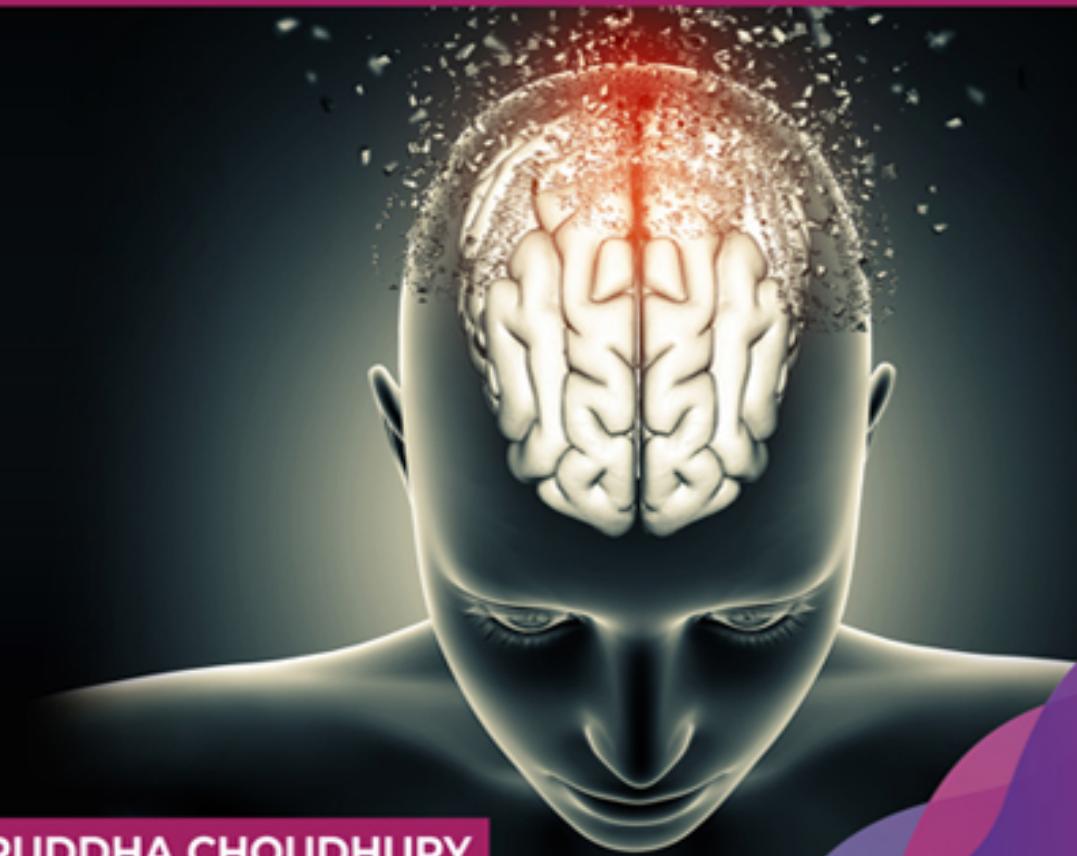


Continuous Machine Learning

with

Kubeflow

Performing Reliable MLOps with Capabilities of TFX, Sagemaker and Kubernetes



ANIRUDDHA CHOWDHURY





Continuous Machine Learning

with

Kubeflow

Performing Reliable MLOps with Capabilities of TFX, Sagemaker and Kubernetes



ANIRUDDHA CHOWDHURY



Continuous Machine Learning with Kubeflow

*Performing Reliable MLOps with Capabilities of
TFX, Sagemaker and Kubernetes*

Aniruddha Choudhury



www.bpbonline.com

FIRST EDITION 2022

Copyright © BPB Publications, India

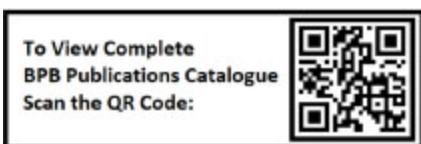
ISBN: 978-93-89898-50-7

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.



www.bpbonline.com

Dedicated to

My beloved parents and family

About the Author

Aniruddha Choudhury has 5 years of IT professional experience in providing Artificial Intelligence development solutions, MLOPS Kubeflow, Multi Cloud GCP, AWS, Azure and is passionate about providing Data Science and Data Engineering complex solutions in machine learning and deep learning. He is always looking for new opportunities for a new dimensional challenge for high impact business problems to become a valuable contributor for his future employers.

Presently he is working in Publicis Sapient as Senior Data scientist (Full stack MLOPS) for the last 2 year. Previously he worked with Incture technology and before that he has worked at Wells Fargo Bank on diverse financial products' AI solutions on various lines of business.

As a tech geek, Aniruddha is always enthusiastic about working in cross-dimensional knowledge. He is working on Kaggle and Google data projects building a statistical and machine learning NLP model with an end-to-end data wrangling and preparation, building model framework with visualization and predictive/text/image analytics and Amazon AWS and Microsoft Azure DataBricks Cloud with Apache scala and spark and finding patterns in all forms of data. He has a passion to break complex problems in data science field and find resolutions with deep learning and machine learning. He is also working on deep learning frameworks like Tensorflow, Keras, and Pytorch. As an individual, Aniruddha always believes in constantly learning new skills and taking the road less travelled. He likes to keep himself updated about the technological world and is always toying with some new innovation.

He has mastered in building Artificial Intelligence solutions and finding complex patterns from research papers to gain optimal solution to current product development and possesses a self-innovative mind alongside.

About the Reviewer

Rajdeep Kumar is a lead data scientist at one of the top consulting companies. He has a Post Graduate Degree in Data Science from BITS Pilani, and is an alumni of IIIT Bangalore. He has more than 11 years of work experience in the IT industry. In his capacity, he drives and contributes to the Data Science activities along with defining the road map, scoping and mentoring the team members. He is an expert in deriving and productionizing end-to-end ML solutions to grow business and have a positive impact on the key business KPIs.

Acknowledgement

There are a few people I want to thank for the continued and ongoing support they have given me during the writing of this book. First and foremost, I would like to thank my parents for continuously encouraging me for writing the book — I could have never completed this book without their support. I would like to thank my friend Dr. Someswar Deb, who is working as MSL in Novartis, for his constant support and motivation.

I am grateful to the journey provided from my companies who gave me support throughout the learning process of my career.

Thank you for all the hidden support provided. I gratefully acknowledge Mr. Abhishek Kumar, Senior Director of Publicis Sapient for his kind technical help for deployment related stuffs which was helpful for this book. I would like to thank Sivaram Annadurai Senior Manager at Publicis Sapient for his constant motivation in technical and business aspects in machine learning real-time projects.

My gratitude also goes to the team at BPB Publications for being supportive enough to provide me quite a long time to finish the first part of the book and also allow me to publish the book in multiple parts. Since image processing being a vast and very active area of research, it was impossible to deep-dive into different classes of problems in a single book, especially by not making it too voluminous.

Preface

This book focuses on the DevOps and MLOps of deploying and productionising machine learning projects with Kubeflow in Google Cloud platform. The authors feel that in this era of machine learning, lot of companies failed to make production of AI/ML projects in real time which was also a study from Forbes. It is compelling and relevant content for today's practicing DevOps/MLOps teams as this sector is still changing. So, many machine learning platforms today take different approaches to the architecture and solution space of managing machine learning workflows. The core concepts of Kubernetes and Kubeflow and its architecture alongside teaches us how to approach and make your AI/ML projects from training to serving with scale in production with Kubeflow.

This book starts by taking you through today's machine learning infrastructure of Kubernetes and Kubeflow architecture. We then go on to outline the core principles of deploying various AI/ML use cases with TensorFlow training serving with Kubeflow and explain how Kubernetes solves some of the issues that arise. We further show how to use TFX with Kubeflow alongside Explainable AI for determining fairness and biasness with What-if Tool. We learn various serving techniques framework for different use cases with Kubeflow KF serving. After that we look at building sample computer vision based UI in streamlit and deploying that in Google cloud platform Kubernetes and Heroku deployment.

This book is divided into 8 chapters. They cover Kubernetes, Kubeflow basics, advance deployment projects with Kubeflow, AWS Sagemaker deployment and explainable AI with real time examples for deployment and container creation with Docker and building pipeline in Kubeflow. More interest will arise among learners in Machine learning deployment with Kubeflow.

The details are listed as follows:

Chapter 1: In this chapter, we will learn about the complete features of Kubeflow, how it works and its need. We will also learn about the

architecture functionality of Kubernetes such as service, pod, Ingress, and so on. We will learn to build the docker image and learn it's working. Here, we will see the components of Kubeflow advantage, which we will be using in the upcoming chapters. Then, we will proceed towards the complete setup of Kubeflow in the Google Cloud platform and Jupyter notebook setup. We have an optional item – how to create the persistent volume claim and attach it to the file store to save your codes and data.

Chapter 2: In this chapter, we will build an end-to-end TensorFlow classification model deployment with Kubeflow orchestration which includes deploying Kubeflow in Kubernetes Cluster in GCP, building the pipeline components for the model with Docker and Kubeflow SDK, and then serving the model with KF serving to have an endpoint for prediction. We will also track the monitoring and performance for our serving traffic endpoint in Grafana dashboard.

Chapter 3: In this chapter, we will build an end-to-end TensorFlow computer vision model with OpenCV operation and deploy that with the Kubeflow orchestration, which includes deploying Kubeflow in Kubernetes cluster in GCP, building the pipeline components for the model with Docker and Kubeflow SDK and then serving the model with KF serving to have an endpoint for prediction. We will then track the monitoring and performance in Grafana dashboard.

Chapter 4: In this chapter, we will build an end-to-end structured data classification model and make it ready for production with the help of TFX, and serve the model outputs with TF serving to get the prediction. We will also be building the TensorFlow ecosystem model and visualizing the evaluation with Tensorboard and Fairness. Then, we will learn about the various TFX components like TFT, TFMA, TFDV, and so on. Later on, we will create a Kubeflow Pipeline on Google Cloud.

Chapter 5: In this chapter, we will work on a classification model with the hotel booking dataset, train the TensorFlow and boosting models, and visualize the advanced explanation of our model results with Tensorboard, Shap, and What-if products.

Chapter 6: In this chapter, we will build an end-to-end Light Model framework and will monitor the model performance in the Weights and

Biases (Wandb) tool. Within Weights and Biases, we will see the live model RMSE graphs and parallel coordinates' hyper parameter performance graphs for each iteration. Next, we will deploy the model with the KF serving in our Kubernetes Cluster inside Google Cloud platform. We will be serving model endpoint which will be used for prediction and monitored in the Grafana Dashboard, such as model rate request with respect to the time and CPU and GPU consumption.

Chapter 7: In this chapter, we will work on the Housing Price Sales dataset project, where we will completely run, evaluate, and deploy the model in the Amazon SageMaker Cloud environment and use S3 for data storage. We will also be using the in-built container algorithm XG-Boost for model building so that we are able to understand the architecture of SageMaker model building framework end to end.

Chapter 8: In this chapter, we will build an end-to-end web application for the computer vision models, and build the UI with Streamlit. We will be learning about many Open CV models for image like cropping, changing pixels, and so on. Next, we will host the web application with the Heroku Container Registry or Kubernetes Cluster as a service application in Google Cloud.

Downloading the code bundle and coloured images:

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

<https://rebrand.ly/bfbaf6>

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, that may have 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.

Did you know that BPB offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at **www.bpbonline.com** and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at **business@bpbonline.com** for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.



BPB is searching for authors like you

If you're interested in becoming an author for BPB, please visit www.bpbonline.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Continuous-Machine-Learning-with-Kubeflow>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/bpbpublications>. Check them out!

PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit www.bpbonline.com.

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase

decisions, we at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit www.bpbonline.com.

Table of Contents

1. Introduction to Kubeflow & Kubernetes Cloud Architecture

Structure

Objectives

1.1 Docker understanding

 1.1.1 Dockerfile

1.2 Kubernetes Architecture

 1.2.1 What is Kubernetes?

 1.2.2 Why do we need Kubernetes?

 1.2.3 What are the Advantages of Kubernetes?

 1.2.4 How do Kubernetes work?

1.3 Kubernetes components

 1.3.1 Types of Services

1.4 Introduction on Kubeflow Orchestration for ML Deployment

1.5 Components of Kubeflow

 1.5.1 Central Dashboard

 1.5.2 Registration Flow

 1.5.3 Metadata

 1.5.4 Jupyter Notebook server

 1.5.5 Katib

1.6 Getting Started in GCP Kubeflow setup

 1.6.1 Install and Set Up kubectl

 1.6.2 Install and Set Up gcloudsdk

 1.6.3 Set Up OAuth from Cloud IAP

 1.6.4 Set Up Docker

 1.6.5 Set Up Kubeflow in Kubernetes Cluster in GCP

 1.6.6 Connect to cluster and Deploy Grafana

 1.6.7 Jupyter Notebook server setup in Kubeflow

1.7 Optional: PVC setup for Jupyter Notebook

1.8 Conclusion

1.9 Reference

2. Developing Kubeflow Pipeline in GCP

Structure

Objectives

2.1 Problem statement

2.2 Getting started in GCP Kubeflow setup

2.3 Breakdown technique to build production pipeline

2.4 Building the Kubeflow Pipeline components for TensorFlow model

 2.3.1 Data Extraction or Ingestion Component

 2.3.2 Data pre-processing component

 2.3.3 Training model component

 2.3.4 Evaluation component

2.5 Serving the Model with KF Serving

2.6 Building the pipeline end to end

2.7 Monitoring the performance with Grafana dashboard

2.8 Conclusion

2.9 Reference

3. Designing Computer Vision Model in Kubeflow

Structure

Objectives

3.1 Problem statement

3.2 Getting started in GCP Kubeflow setup

3.3 Analytics behind the problem statement

3.4. Building the Kubeflow pipeline components for Computer Vision (CNN) TensorFlow model

 3.4.1 Data extraction or Ingestion component

 3.4.2 Data pre-processing component

 3.4.3 Training model component

 3.3.4 Evaluation component

3.5. Serving the Model with KF Serving

3.6 Building the pipeline end to end

3.7. Auto-Scaling of the Serving Endpoint

3.8 Conclusion

3.9 Reference

4. Building TFX Pipeline

Structure

Objective

4.1 Problem statement

4.2 Architecture of TFX components

4.3 TFX environment setup

4.4 TFX pipeline components

 4.4.1 ExampleGen

 4.4.2 StatisticsGen

 4.4.3 SchemaGen

 4.4.4 ExampleValidator

 4.4.5 Transform

 4.4.6 Tuner and Trainer

 4.4.7 Evaluator

 4.4.7.1 Fairness and TFMA Visualization

 4.4.8 Pusher

4.5 Serve the Model with TF Serving

4.6 Building Kubeflow Pipeline Orchestrator

4.7 Conclusion

4.8 Reference

5. ML Model Explainability & Interpretability

Structure

Objectives

5.1 Problem

5.2 General idea and concept behind Shap

5.3 Getting Started with Python library Installation and Data loading in Colab

5.4 Feature transformation for Training Model

5.5 LightGBM Model training

5.6 Model Analysis with advance Visualization along Shap Tool

 5.6.1 Basic decision plot features

 5.6.2 Force Plots Analysis

5.7 TensorFlow Estimator Model Framework Building

 5.7.1 TensorFlow Estimator Model

[5.8 Advance Visualization for TensorFlow Model with Tensorboard & What-If Tool](#)
[5.8.1 Tensorboard](#)
[5.8.2 What-If Tool](#)
[5.9 Conclusion](#)
[5.10 References](#)

6. Building Weights & Biases Pipeline Development

[Structure](#)
[Objectives](#)
[6.1 Problem statement](#)
[6.2 Setup of project requirements in GCP & Wandb](#)
[6.2.1 Kubeflow Cluster in GCP and Docker setup](#)
[6.2.2 Kaggle API setup for downloading data](#)
[6.2.3 Weights & Biases API Key](#)
[6.3 Introduction on how to use Weights & Biases](#)
[6.4 Modeling and training the LightGBM Model for Equity Data](#)
[6.4.1 Get the latest version of Weights & Biases Dependency & Kaggle Setup](#)
[6.4.2 Weights & Biases Dependency & Kaggle API Setup](#)
[6.4.3 Loading and Extracting of Data](#)
[6.4.4 Exploratory Data Analysis](#)
[6.4.4 Utility Metrics Function](#)
[6.4.5 Training model \(using Weights & Biases\) with LightGBM Framework](#)
[6.5 Serving the model with KF Serving](#)
[6.6 Monitoring the performance with Grafana Dashboard](#)
[6.7 Conclusion](#)
[6.8 References](#)

7. Applied ML with AWS SageMaker

[Structure](#)
[Objectives](#)
[7.1 Problem](#)
[7.2 Getting started in AWS SageMaker setup](#)

[7.3 Getting Started with JupyterLab Notebook Instances and SDK & S3 Bucker](#)
[7.3.1 Create an S3 Bucket](#)
[7.3.2 Create an Amazon SageMaker Notebook Instance](#)
[7.4 Getting Started by Launching Notebook and loading data to S3](#)
[7.5 Load, Analyse, and Transform the Training Data](#)
[7.5.1 Data Loading from S3 and Library](#)
[7.5.2 Feature Engineering](#)
[7.5.2.1 Finding Categorical & Numerical Columns](#)
[7.5.2.2 Checking the missing values sum](#)
[7.5.2.3 Log transformation of dependent feature](#)
[7.5.2.4 Correlation & Scatter Plots](#)
[7.5.2.5 Outlier Detection](#)
[7.5.3 Feature Transformation](#)
[7.6 Amazon SageMaker Training Model](#)
[7.6.1 Splitting Data into Train/Validation and push to S3](#)
[7.6.2 Train with SageMaker API XG-Boost which maintains the algorithm container](#)
[7.7 Amazon SageMaker model deployment](#)
[7.8 Conclusion](#)
[7.9 References](#)

8. Web App Development with Streamlit & Heroku

[Structure](#)
[Objectives](#)
[8.1 Problem statement](#)
[8.2 Setup of project requirements in GCP & Heroku](#)
[8.3 Introduction on components of Streamlit](#)
[8.3.1 Main concepts](#)
[8.4 Building the Framework for Streamlit for OpenCV models](#)
[8.5 Creating the components for Heroku Deployment](#)
[8.6 Deploying the Streamlit code by containerizing in Kubernetes cluster](#)
[8.7 Summary](#)
[8.8 References](#)

[Index](#)

CHAPTER 1

Introduction to Kubeflow & Kubernetes Cloud Architecture

In this chapter, we will learn about the complete features of Kubeflow, how it works, and why we need Kubeflow. We will also learn about the architecture functionality of Kubernetes, like service, pod, Ingress, and so on, and how to build the docker image, and how it works. Here, we will see the components of Kubeflow advantage, which we will be using in the upcoming chapters. Then, we will proceed towards the complete setup of Kubeflow in the Google Cloud Platform and Jupyter notebook setup. We have an optional item – how to create the Persistent Volume Claim, and attach it to the File store to save your codes and data.

Structure

In this chapter, we will cover the following topics:

- Docker understanding
- Kubernetes concepts and architecture
- Kubernetes components
- Introduction on Kubeflow Orchestration for ML Deployment
- Components of Kubeflow
- Setting Up for Kubeflow in GCP
- Jupyter Notebook setup
- **Optional:** PVC setup for Jupyter Notebook

Objectives

This chapter will help you learn the following:

- The core understanding of Docker and Kubernetes, and its application to be used in Cloud.

- Kubernetes and Kubeflow Architecture and its functionality and advantages.
- The components of Kubeflow and how to setup Kubeflow IAP Cluster in Google Cloud Platform.
- Docker image of CPU for setting up the Jupyter notebook, alongside the PVC Setup in Cloud.

NOTE	Rest all the imports I have showed in my Colab Notebook, for which the hyperlink of GitHub account of this chapter is given below. Note Colab platform Python 3.x. RUN IN GOOGLE COLAB
CODE	https://github.com/bpbpublications/Continuous-Machine-Learning-with-Kubeflow/tree/main/Chapter1

1.1 Docker understanding

Docker is a platform for the developers and system admins to build, run, and share the applications with the containers. The containers used to deploy the applications is called containerization. To deploy the applications, the containers are making things easier and flexible.

Containerization is increasingly becoming popular because the containers have the following features:

- **Flexible:** We can containerize the most complex applications as well.
- **Scalable:** We can distribute the container replica's process across a data center automatically.
- **Lightweight:** The containers make things more efficient than the Virtual machines by sharing and leveraging the host kernel.
- **Portable:** Due it's portable nature, we can build them locally, deploy to the cloud, and run it from anywhere.
- **Loosely coupled:** They are highly independent and encapsulated, which allows us to replace or upgrade anyone without disrupting others.
- **Secure:** Without any required configuration on the part of the user, it applies aggressive isolations and constraints to the processes.

Images and containers:

Logically, a container is a running process, with some added encapsulated features which applies for the Host to be isolated from it and from the other containers. The most important aspects is that each container interacts with its own private file system, which is called container isolation; the Docker image provides this file system. So, an image includes most of the things which are needed to run an application – the code, runtimes, dependencies, and any other file system objects required.

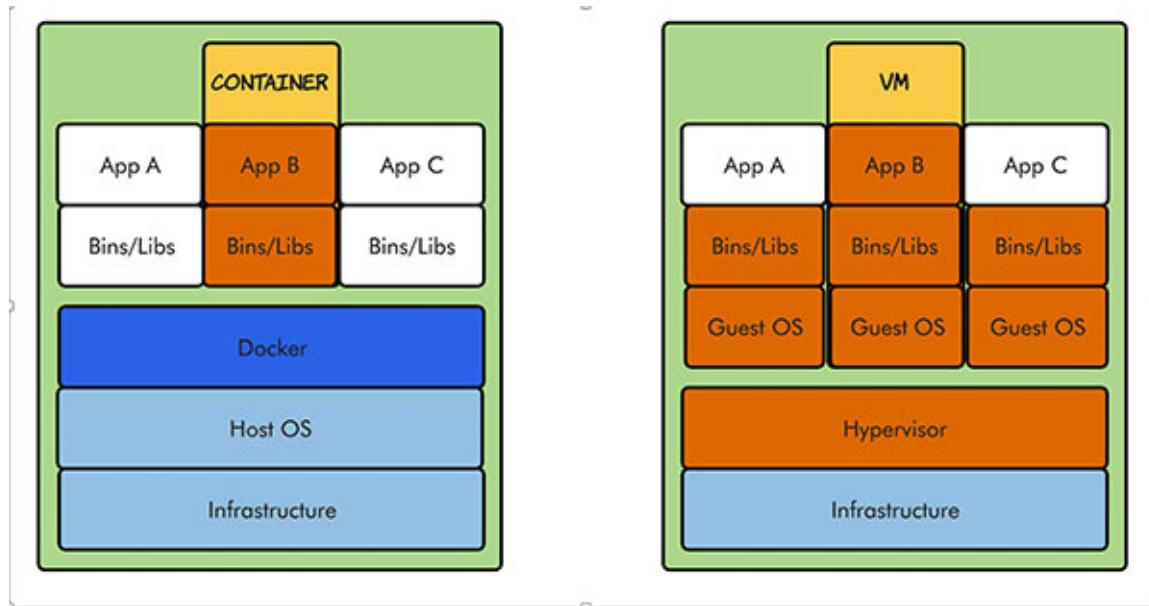


Figure 1.1: Docker Architecture

The Docker has two concepts which are almost the same with its VM containers as the idea of an image and a container. An image, which is the definition of that will be executed, just like an operating system image, and for a given image, the container is the running instance.

1.1.1 Dockerfile

To get our Python or any language code running in a container is to warp it in a package as a Docker image, after that run a container based on it. The steps are sketched as follows:

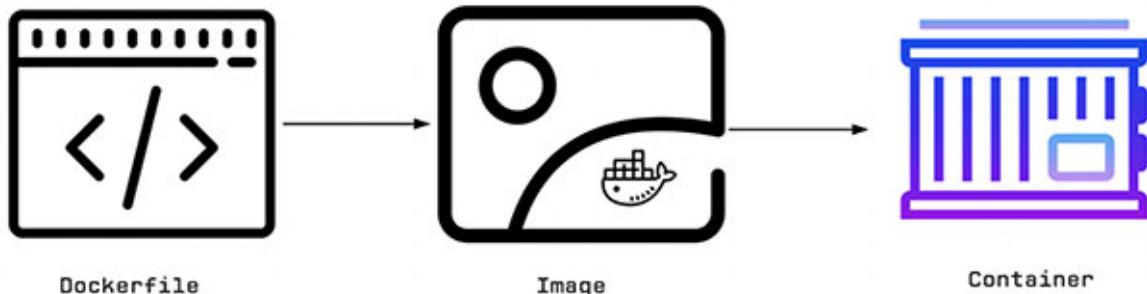


Figure 1.2: Docker file process

Next, for generating a Docker image, we need to create a Dockerfile which contains some set of instructions needed to build the image. The Dockerfile is then processed by the Docker builder which generates the Docker image. At last, with a simple Docker run command, we can create and run a container with the Python service.

```

Dockerfile X
# set base image (host OS)
FROM python:3.8 ← Base Docker Public Python Image
# set the working directory in the container
WORKDIR /app

# copy the dependencies file to the working directory
COPY requirements.txt .

# install dependencies
RUN pip install -r requirements.txt ← Python Libraries Installation

# copy the content of the local src directory to the working directory
COPY src/ .

# command to run on container start
CMD [ "python", "./server.py" ] ← Entrypoint when the image will run the server python file

```

Figure 1.3: Dockerfile code

Let's split this file into the following lines:

- It uses the Python base image with the tag Python:3.8, which is a specific version of Python.
- Then, it creates a working directory, where we will copy our local files to that directory; here we have created an app folder and copied the

requirements file which contains the Python library.

- Then, we have installed all the Python libraries with the pip command.
- Next, we use CMD, which is a command to run the Python file whenever the container gets started.

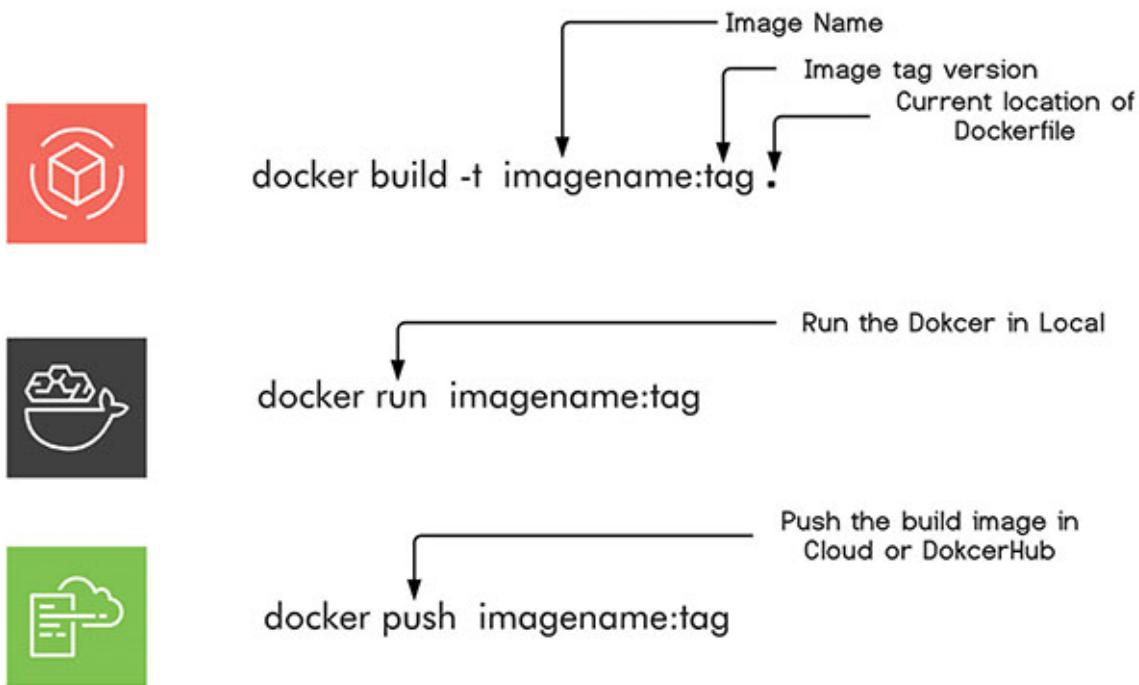


Figure 1.4: Dockerfile command

Now, run the preceding code in the terminal to push the image, which you have built in local to cloud (GCP/AWS/AZURE) and Docker hub.

1.2 Kubernetes Architecture

In this section, we will see how the Kubernetes work, and learn about its architecture.

1.2.1 What is Kubernetes?

Kubernetes is an open-source container management system used in large-scale enterprises in several dynamic industries to perform a mission-critical task or any orchestration task. Some of its capabilities include the following:

- It manages the containers inside cluster.
- It deploys applications to which it provides tools.

- It scales the applications as per requirement.
- It manages the existing containerized application changes.
- It optimizes the use of the underlying hardware complexity beneath our container.
- It enables an application component to restart and move across multiple systems as per need.

1.2.2 Why do we need Kubernetes?

We need Kubernetes to manage the containers when we run our production grade environments using a pattern of microservice with many containers. We need to track features such as health check, version control, scaling, and rollback mechanism among other things. It can be quite challenging and frustrating to make sure that all of these things are running alright. Kubernetes gives us the orchestration and management capabilities required to deploy the containers at scale. To build the application services with the Kubernetes orchestration allows us to span multiple containers and timely schedule those containers across a cluster, scale those containers when it's not in use, and manage the health of those containers from time to time. In a nutshell, Kubernetes is more like a Master manager that has many subordinates (containers). What a manager does is maintain what the subordinates need to do.

1.2.3 What are the Advantages of Kubernetes?

The following are the advantages of Kubernetes:

Portable and Open-Source:

Kubernetes can run the containers on one or more public cloud environments, virtual machines, or bare metal, which means it can be deployed on any infrastructure. Moreover, Kubernetes is compatible across multiple platforms, making a multi-cloud strategy a highly flexible and usable component.

Workload Scalability:

Kubernetes course offers the following useful features for scaling purpose:

- **Horizontal Infrastructure Scaling:** Operates on the individual server level to implement horizontal scaling. New servers can be added or

removed easily.

- **Auto-Scaling:** We can alter the number of containers running, based on the usage of CPU resources or other application-metrics.
- **Manual Scaling:** The number of running containers through a command or the interface can be manually scaled.
- **Replication Controller:** The replication controller makes sure that the cluster has a specified number of equivalent pods in a running condition. If there are too many pods, the replication controller can remove the extra pods or vice-versa.

High Availability:

Kubernetes can handle the availability of both the applications and the infrastructure. It tackles the following:

- **Health Checks:** The application doesn't fail by constantly checking with the health of nodes and containers. Kubernetes offers self-healing and auto replacement if a pod crashes due to an error.
- **Traffic Routing and Load Balancing:** Kubernetes' load balancer distributes the load across multiple nodes, enabling us to balance the resources quickly during incidental traffic or batch processing.

Designed for Deployment:

Containerization has an ability to speed up the process of building, testing, and releasing the software, and the useful features include the following:

- **Automated Rollouts and Rollbacks:** It can handle the new version and update our app without any downtime, while we monitor the health during the roll-out process. If any failure occurs during the process, it can automatically roll back to the previous version.
- **Canary Deployments:** So, the production of the new deployment and the previous version can be tested in parallel, that is, before scaling up the new deployment and parallelly scaling down the previous deployment.
- **Programming Language and Framework Support:** Most of the programming languages and frameworks like Java, Python, and so on, are supported by Kubernetes. If an application has the ability to run in a container, it can run in Kubernetes as well.

Kubernetes and Stateful Containers:

Kubernetes' Stateful Sets provides resources like volumes, stable network ids, and ordinal indexes from 0 to N, and so on, to deal with the stateful containers. Volume is one such key feature that enables us to run the stateful application. The two main types of volume supported are as follows:

- **Ephemeral Storage Volume:** Ephemeral data storage is different from Docker. In Kubernetes, the volume is taken into account in any containers that run within the pod, and the data is stored across the container. But, if the pods get killed, the volume is automatically removed.
- **Persistent Storage:** The data remains for the lifetime. So, when the pod dies or it is moved to another node, that data will still remain until it is deleted by the user. Hence, the data is stored remotely.

1.2.4 How do Kubernetes work?

A cluster is the foundation of **Google Kubernetes Engine (GKE)**; the Kubernetes objects that represent your containerized applications all run on top of a cluster. In GKE, a cluster consists of at least one control plane and multiple worker machines, called nodes. These control plane and node machines run the Kubernetes cluster orchestration system.

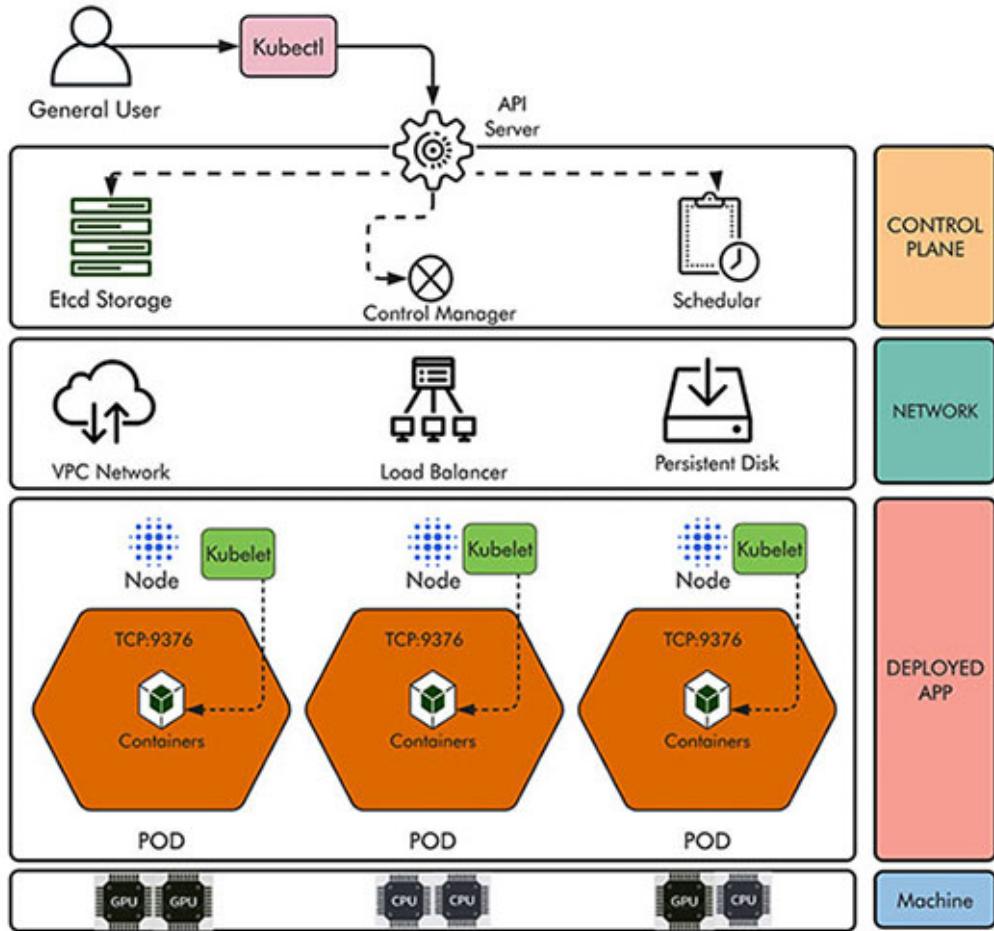


Figure 1.5: Kubernetes Architecture

Master:

The master is the controlling element of the cluster. The master has the following three parts:

- **API Server:** The application that serves Kubernetes' functionality through a RESTful interface and stores the state of the cluster.
- **Scheduler:** The scheduler watches the API server for new Pod requests. It communicates with the Nodes to create new pods and assign work to the nodes while allocating resources or imposing constraints.
- **Controller Manager:** The component on the master runs the controllers. It includes the Node controller, Endpoint Controller, Namespace Controller, and so on.

Slave (Nodes):

These machines perform the requested, assigned tasks. The Kubernetes master controls them. There are the following four components inside the Nodes:

- **Pod:** All containers will run in a pod. Pods abstract the network and storage away from the underlying containers. Your app will run here.
- **Kubelet:** The Kubectl registering the nodes with the cluster, watches for work assignments from the scheduler, instantiates new Pods, and reports back to the master.
- **Container Engine:** It is responsible for managing the containers, image pulling, stopping the container, starting the container, destroying the container, and so on.
- **Kube Proxy:** It is responsible for forwarding the app user requests to the right pod.

1.3 Kubernetes components

In this section, let's understand the deep concept behind the functionality of Kubernetes' each and every important components.

NODE:

A node is the smallest unit of the computing hardware in Kubernetes. It is a representation of a single machine in your cluster. In most production systems, a node will likely be either a physical machine in a datacenter, or a virtual machine hosted on a cloud provider, like the Google Cloud Platform. Don't let conventions limit you; however, in theory, you can make a node out of almost anything.

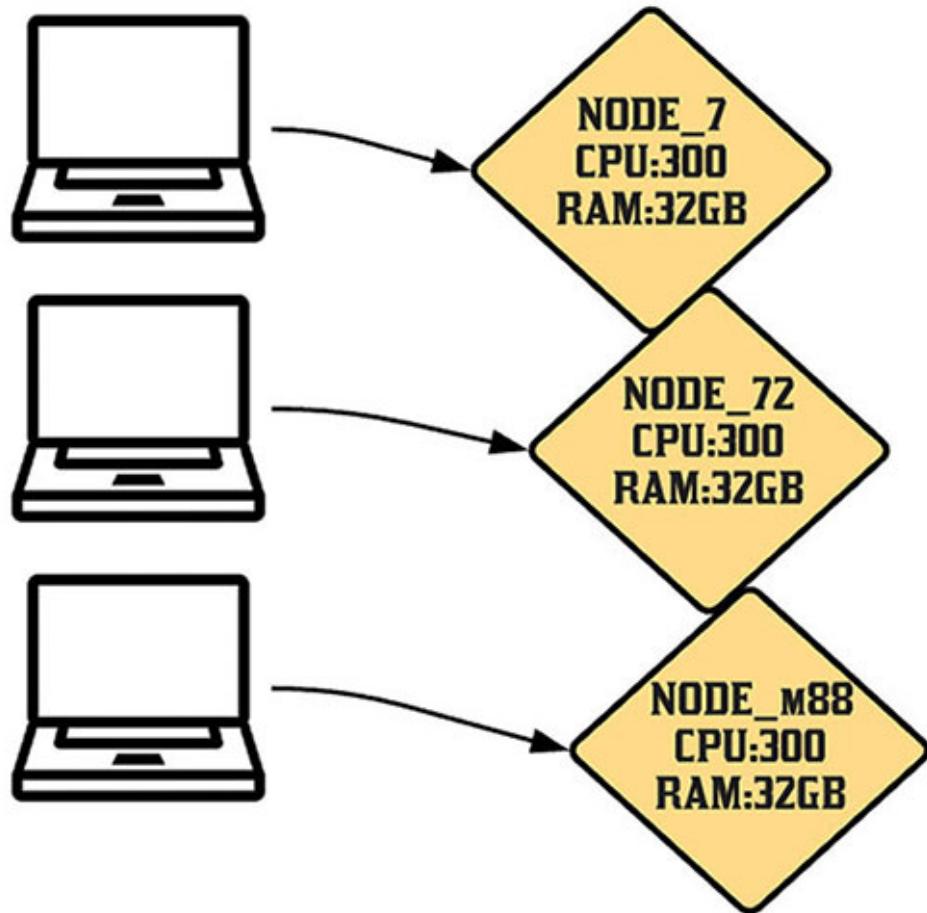


Figure 1.6: Node Concept

Thinking of a machine as a “node” allows us to insert a layer of abstraction. Now, instead of worrying about the unique characteristics of any individual machine, we can instead simply view each machine as a set of CPU and RAM resources that can be utilized. In this way, any machine can substitute any other machine in a Kubernetes cluster.

CLUSTER:

Although working with the individual nodes can be useful, it’s not the Kubernetes way. In general, you should think about the cluster as a whole, instead of worrying about the state of individual nodes.

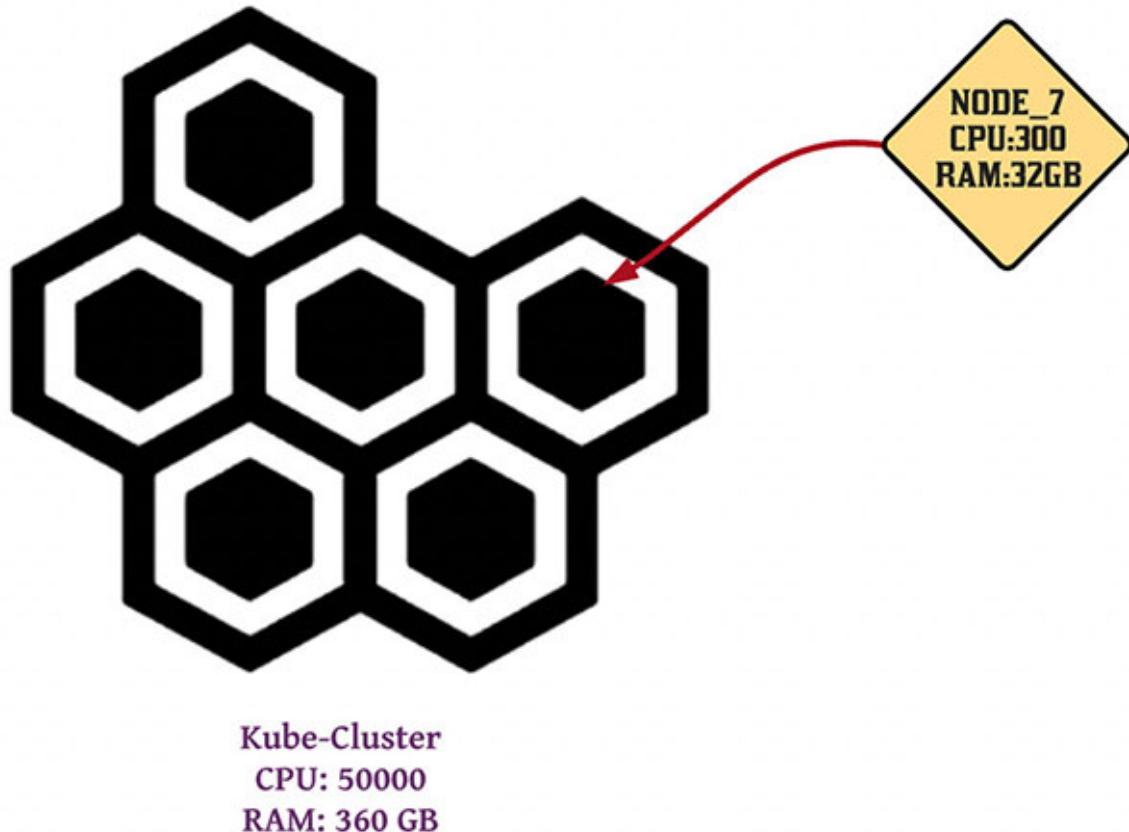


Figure 1.7: Cluster Concept

POD:

Unlike the other systems that you may have used in the past, Kubernetes doesn't run the containers directly; instead it wraps one or more containers into a higher-level structure called a **pod**. Any containers in the same pod will share the same resources and local network. The containers can easily communicate with the other containers in the same pod, as though they were on the same machine while maintaining a degree of isolation from the others.

Pods are used as the unit of replication in Kubernetes. If your application becomes too popular and a single pod instance can't carry the load, Kubernetes can be configured to deploy the new replicas of your pod to the cluster as necessary. Even when not under the heavy load, it is standard to have multiple copies of a pod running at any time in a production system to allow the load balancing and failure resistance.

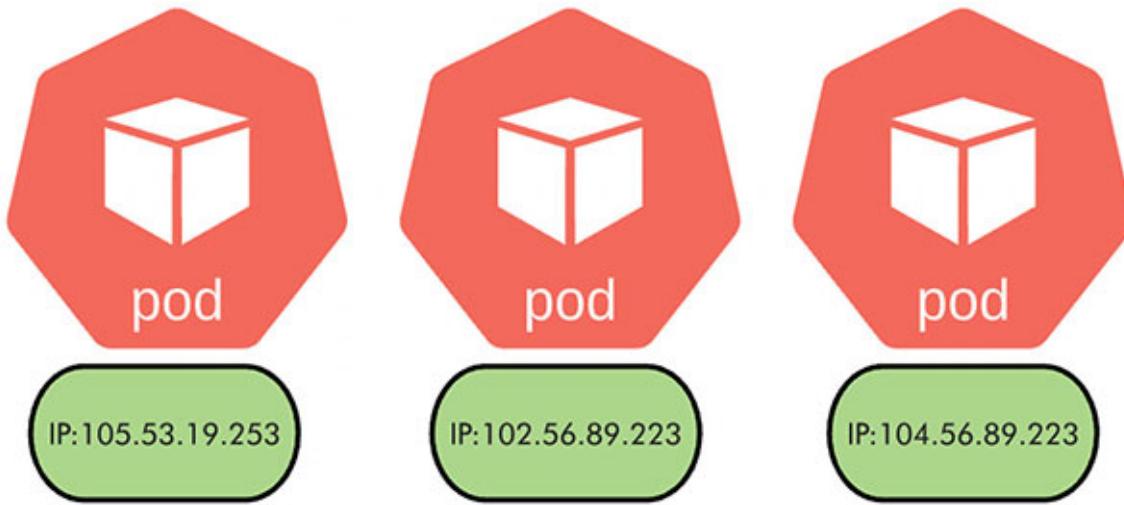


Figure 1.8: Pod Concept

The Pods contain one or more containers, such as the Docker containers. When a Pod runs multiple containers, the containers are managed as a single entity, and share the Pod's resources. Generally, running multiple containers in a single Pod is an advanced use case.

Pods also contain shared networking and storage resources for their containers:

Network: Pods are automatically assigned unique IP addresses. Pod containers share the same network namespace, including the IP address and network ports. Containers in a Pod communicate with each other inside the Pod on localhost.

Storage: Pods can specify a set of shared storage volumes that can be shared among the containers.

```
! pod.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: hello
spec:
  template:
    # This is the pod template
    spec:
      containers:
        - name: hello
          image: busybox:v1
          command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
restartPolicy: OnFailure
```

The diagram illustrates the structure of a Kubernetes pod creation YAML file. It highlights specific fields with arrows pointing to corresponding labels: 'apiVersion' points to 'Pod API version', 'name' points to 'Pod Name', and 'image' points to 'Container Image'.

Figure 1.9: Pod creation yaml

Deployment:

Although the pods are the basic unit of computation in Kubernetes, they are not typically directly launched on a cluster. Instead, the pods are usually managed by one more layer of abstraction – the deployment.

A deployment's primary purpose is to declare how many replicas of a pod should be running at a time. When a deployment is added to the cluster, it will automatically spin up the requested number of pods, and then monitor them. If a pod dies, the deployment will automatically re-create it.

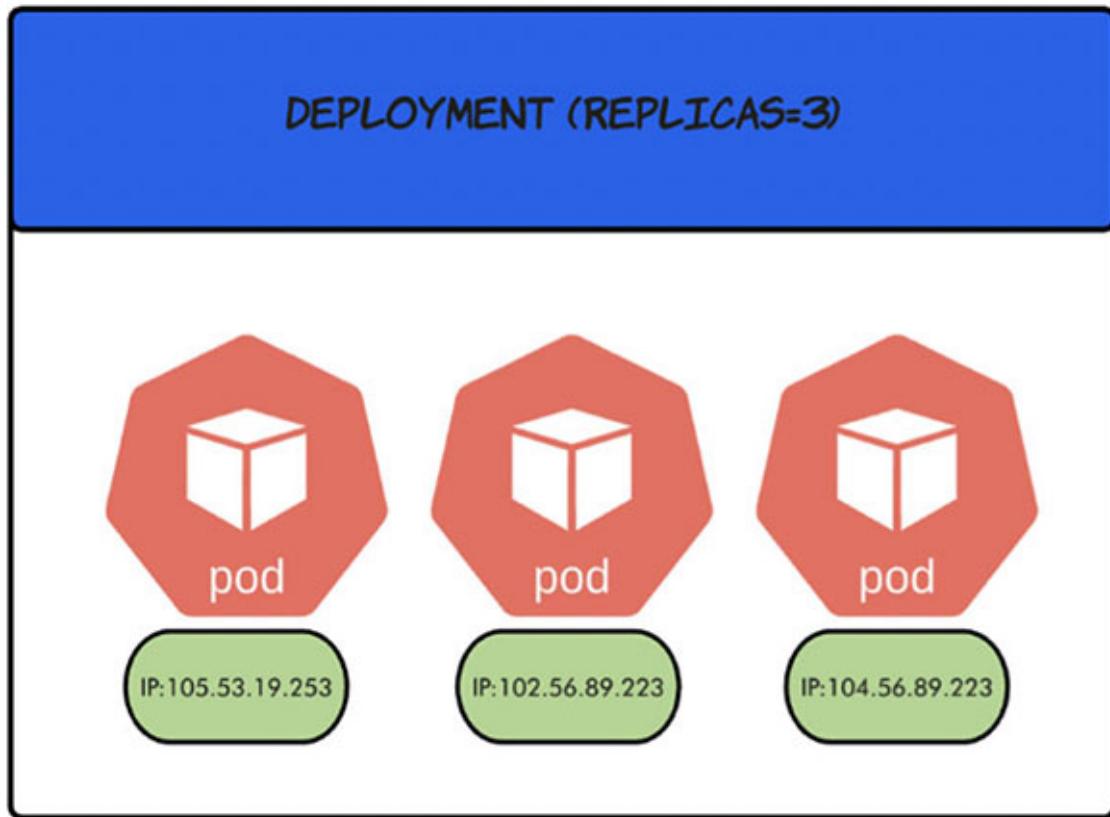


Figure 1.10: Deployment concept

Using a deployment, you don't have to deal with the pods manually. You can just declare the desired state of the system, and it will be managed for you automatically.

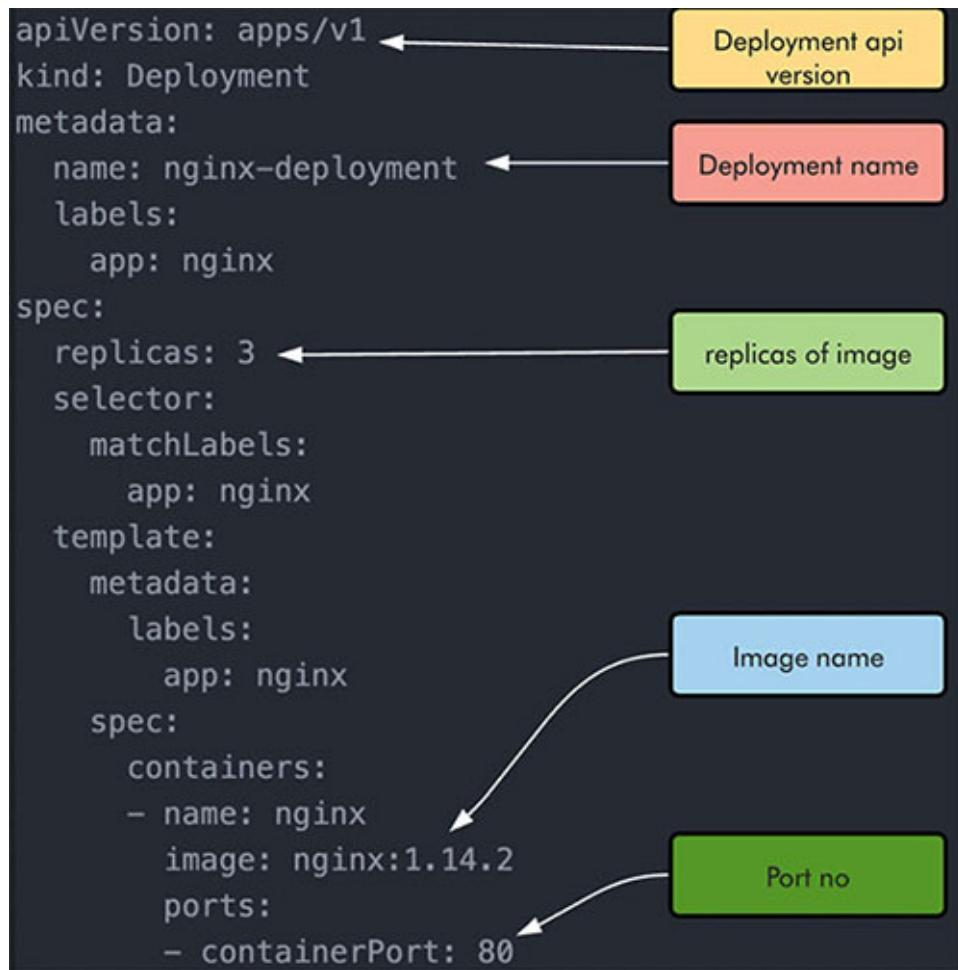


Figure 1.11: Deployment yaml file

In the preceding figure, you can see that it was the template for Deployment and it was the format for each container deployment.

Service:

A service in Kubernetes is a REST object, similar to a Pod. Like all of the REST objects, you can POST a Service definition to the API server to create a new instance. The name of a Service object must be a valid DNS label name.

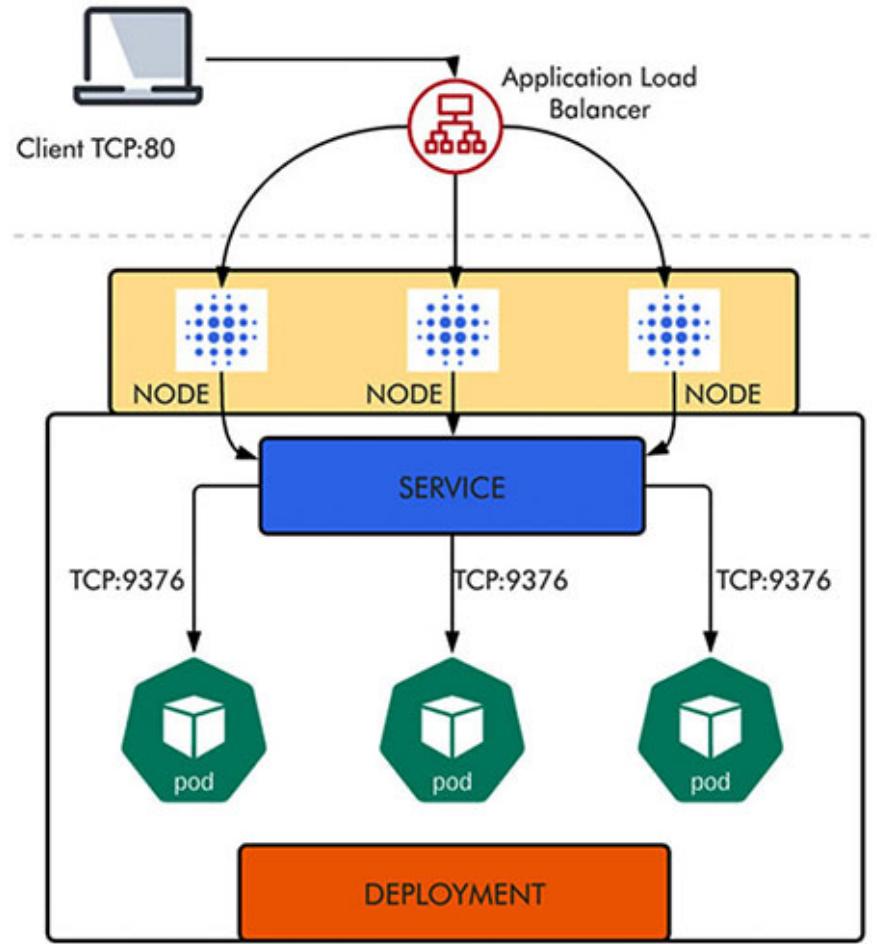


Figure 1.12: Service Concept

For example, suppose you have a set of Pods that each listen on TCP port 9376 and carry a label app=MyApp – this specification creates a new Service object named "my-service", which targets the TCP port 9376 on any Pod with the app=MyApp label.

Kubernetes assigns this Service an IP address (sometimes called the "cluster IP"), which is used by the Service proxies. Port definitions in Pods have names, and you can reference these names in the targetPort attribute of a Service.

Why use a Service?

In a Kubernetes cluster, each Pod has an internal IP address. But the Pods in a Deployment come and go, and their IP addresses change. So, it doesn't make sense to use the Pod IP addresses directly. With a Service, you get a stable IP

address that lasts for the life of the Service, even as the IP addresses of the member Pods change.

A Service also provides load balancing. Clients call a single, stable IP address, and their requests are balanced across the Pods that are members of the Service.

1.3.1 Types of Services

There are the following five types of Services:

- **ClusterIP (default):** Internal clients send requests to a stable internal IP address.
- **NodePort:** Clients send requests to the IP address of a node on one or more nodePort values that are specified by the Service.
- **LoadBalancer:** Clients send requests to the IP address of a network load balancer.
- **ExternalName:** Internal clients use the DNS name of a Service as an alias for an external DNS name.
- **Headless:** You can use a headless service in situations where you want a Pod grouping, but don't need a stable IP address.

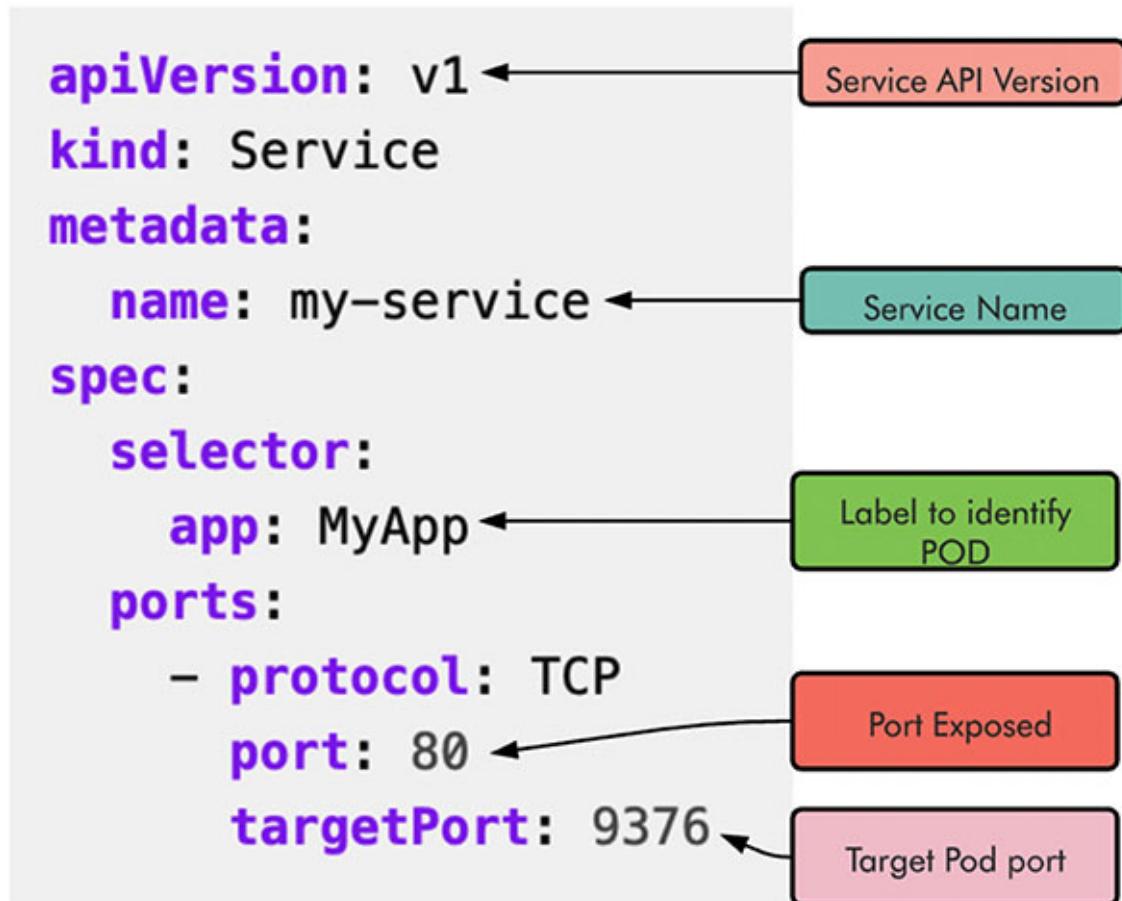


Figure 1.13: Service yaml file

Ingress:

Using the concepts described earlier, you can create a cluster of nodes, and launch deployments of pods onto the cluster. There is one last problem to solve, however – allowing external traffic to your application.

By default, Kubernetes provides isolation between pods and the outside world. If you want to communicate with a service running in a pod, you have to open up a channel for communication. This is referred to as Ingress. There are multiple ways to add ingress to your cluster. The most common ways are by adding either an Ingress controller, or a LoadBalancer. The exact trade-offs between these two options are out of scope for this post, but you must be aware that ingress is something you need to handle before you can experiment with Kubernetes.

What is Ingress?

Ingress exposes HTTP and HTTPS routes from outside the cluster to the services within the cluster. Traffic routing is controlled by the rules defined on the Ingress resource. The following is a simple example where an Ingress sends all its traffic to one Service.

An Ingress may be configured to give the Services externally-reachable URLs, load balance traffic, terminate SSL/TLS, and offer name-based virtual hosting. An Ingress controller is responsible for fulfilling the Ingress, usually with a load balancer, though it may also configure your edge router or additional frontends to help handle the traffic.

An Ingress does not expose the arbitrary ports or protocols. Exposing the services other than the HTTP and HTTPS to the internet, typically uses a service of type **Service.Type=NodePort** or **Service.Type=LoadBalancer**.

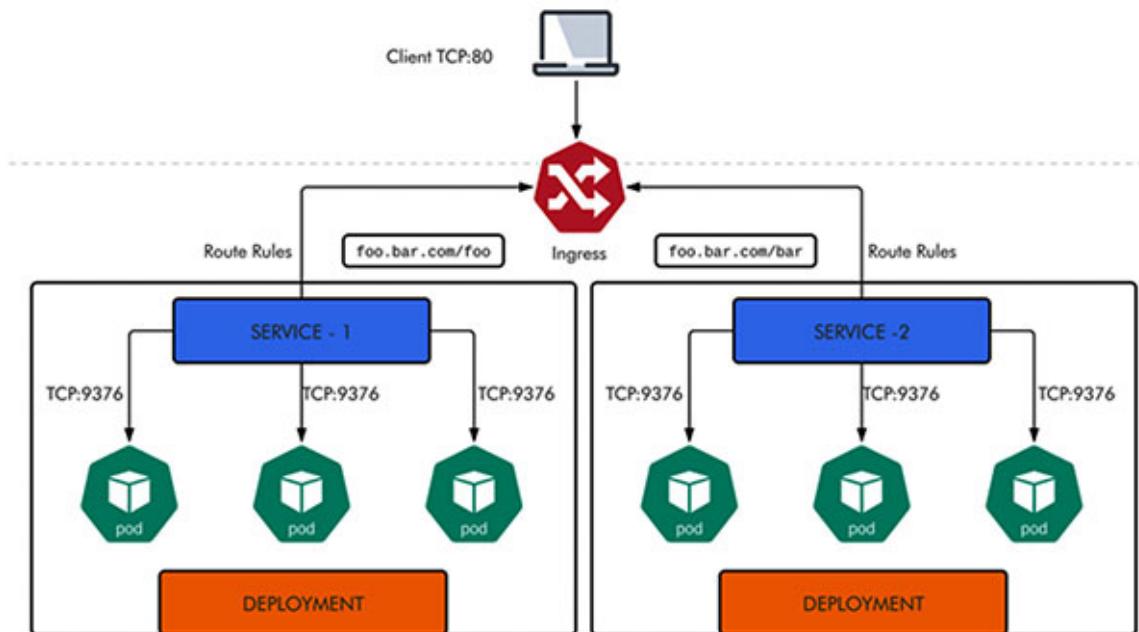


Figure 1.14: Ingress architecture

A fanout configuration routes traffic from a single IP address to more than one Service, based on the HTTP URL being requested. An Ingress allows you to keep the number of load balancers down to a minimum. For example, a setup like the following:

The Ingress controller provisions an implementation-specific load balancer that satisfies the Ingress, as long as the Services (service1, service2) exist. When it

has done so, you can see the address of the load balancer at the Address field.

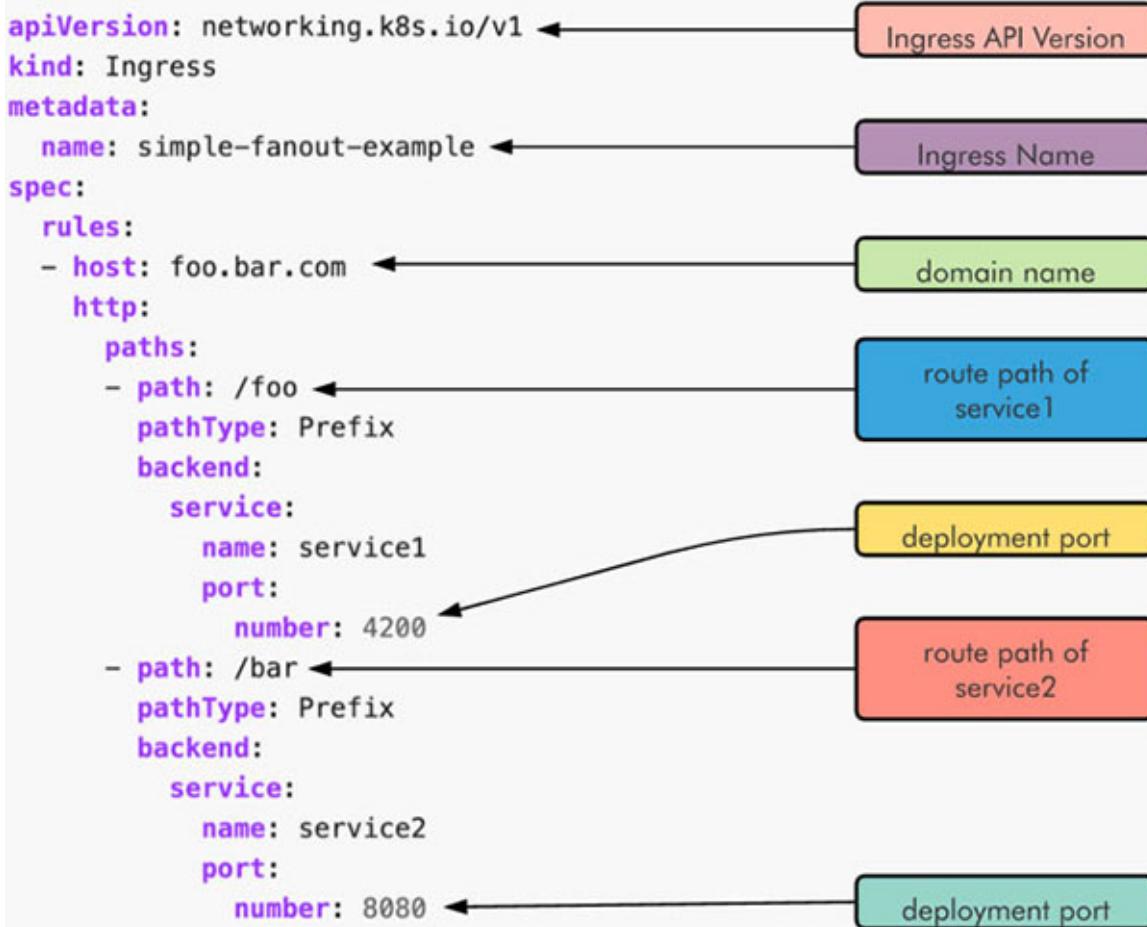


Figure 1.15: Ingress yaml file

Namespace:

When to Use Multiple Namespaces?

Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide. Namespaces provide a scope for the names. The names of resources need to be unique within a namespace, but not across namespaces. Namespaces cannot be nested inside one another and each Kubernetes resource can only be in one namespace.

Kubernetes starts with the four initial namespaces:

- **default:** The default namespace for objects with no other namespace.

- **kube-system**: The namespace for objects created by the Kubernetes system.
- **kube-public**: This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.
- **kube-node-lease**: This namespace for the lease objects associates with each node which improves the performance of the node heartbeats as the cluster scales.

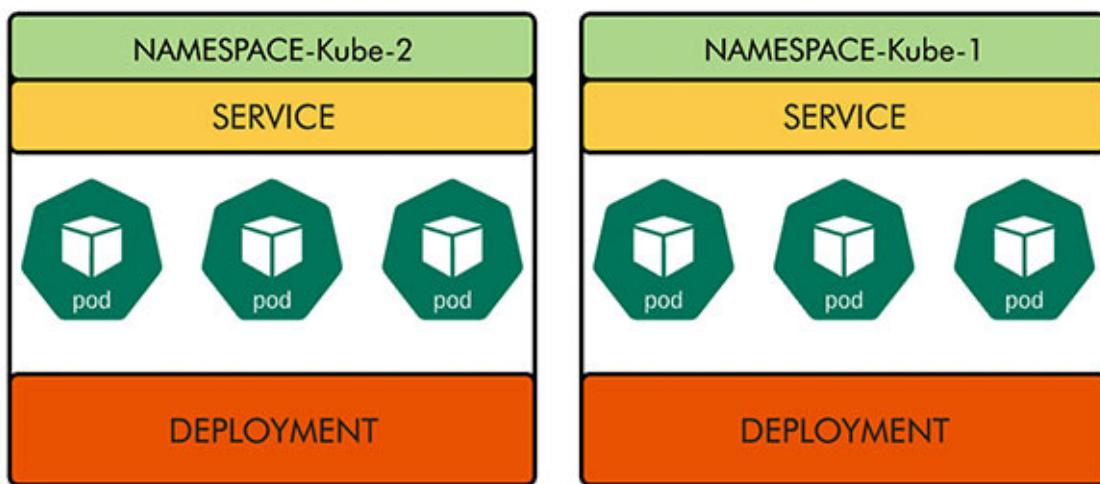


Figure 1.16: Namespace Concept

1.4 Introduction on Kubeflow Orchestration for ML Deployment

Kubeflow is an open source Kubernetes-native platform for developing, orchestrating, deploying, and running scalable and portable ML workloads. It helps support the reproducibility and collaboration in ML workflow lifecycles, allowing you to manage the end-to-end orchestration of ML pipelines, to run your workflow in multiple or hybrid environments, and to help you reuse the building blocks across different workflows.

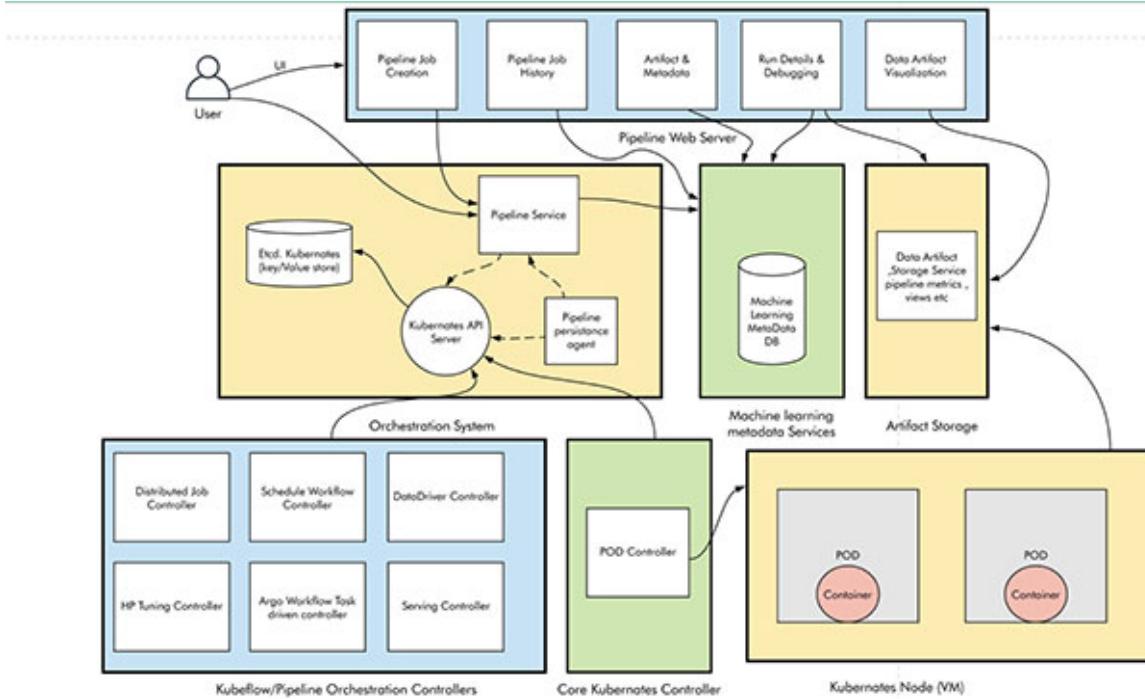


Figure 1.17: Kubeflow Architecture

Kubeflow also provides support for the visualisation and collaboration in your ML workflow.

- 1. Kubernetes resources:** The Pipeline Service calls the Kubernetes API server to create the necessary Kubernetes resources (CRDs) to run the pipeline.
- 2. Python SDK:** You create the components or specify a pipeline using the Kubeflow Pipelines **domain-specific language (DSL)**.
- 3. DSL compiler:** The DSL compiler transforms your pipeline's Python code into a static configuration (YAML).
- 4. Pipeline Service:** You call the Pipeline Service to create a pipeline run from the static configuration.
- 5. Orchestration controllers:** A set of orchestration controllers execute the containers needed to complete the pipeline. The containers execute within the Kubernetes Pods on virtual machines. An example controller is the **Argo Workflow** controller, which orchestrates the task-driven workflows.
- 6. Pipeline web server:** The Pipeline web server gathers data from various services to display relevant views – the list of pipelines currently running,

the history of pipeline execution, the list of data artifacts, debugging information about individual pipeline runs, and execution status about individual pipeline runs.

7. Pipeline Service: You call the Pipeline Service to create a pipeline run from the static configuration.

8. Persistence agent and ML metadata: The Pipeline Persistence Agent watches the Kubernetes resources created by the Pipeline Service and persists the state of these resources in the ML Metadata Service.

9. Artifact storage: The Pods store the following two kinds of data:

- **Metadata:** Experiments, jobs, pipeline runs, and single scalar metrics. Metric data is aggregated for the purpose of sorting and filtering. Kubeflow Pipelines store the metadata in a MySQL database.
- **Artifacts:** Pipeline packages, views, and large-scale metrics (time series). Use large-scale metrics to debug a pipeline run or investigate an individual run’s performance. Kubeflow Pipelines stores the artifacts in an artifact store like Minio server or Cloud Storage.

Conceptual Overview of Kubeflow: You can deploy the workflow to various clouds, local, and on-premises platforms for experimentation and production use.

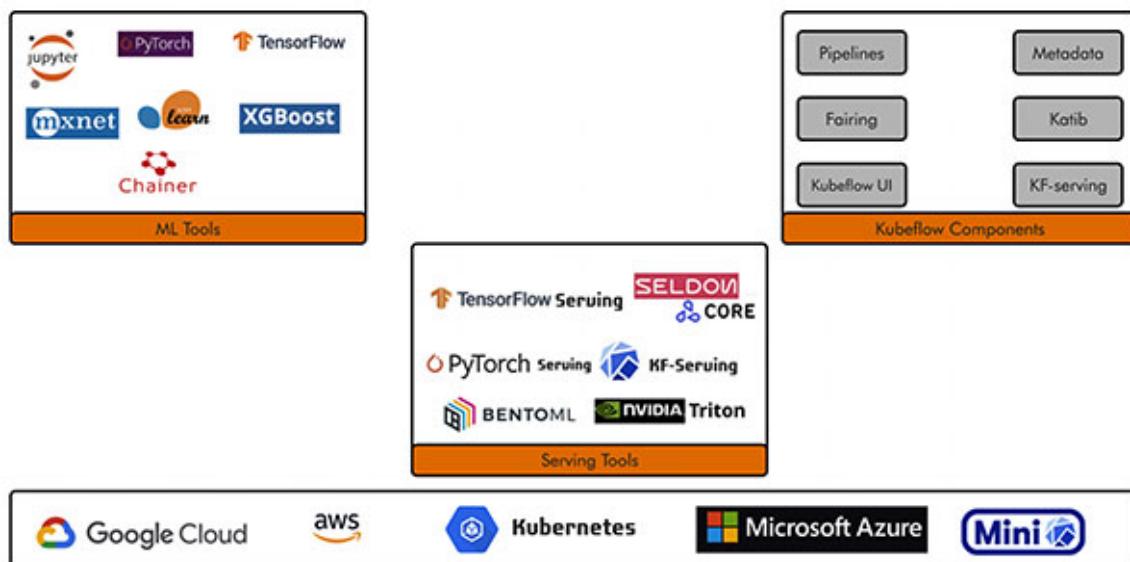


Figure 1.18: Kubeflow Overview features

Kubeflow is the ML toolkit for Kubernetes. The preceding diagram shows Kubeflow as a platform for arranging the components of your ML system on top of Kubernetes.

1.5 Components of Kubeflow

In this section, let's explore some of the components for the introduction of Kubeflow, which we will be using in the upcoming chapters in details.

1.5.1 Central Dashboard

The Kubeflow UIs include the following:

- Home, a central dashboard for navigation between the Kubeflow components.
- Pipelines for a Kubeflow Pipelines dashboard.
- Notebook Servers for Jupyter notebooks.
- Katib for hyperparameter tuning.
- Artifact Store for tracking of artifact metadata.
- Manage Contributors for sharing the user access across namespaces in the Kubeflow deployment.

Kubeflow central UI is accessible at the following URL:

<https://<application-name>.endpoints.<project-id>.cloud.google/>

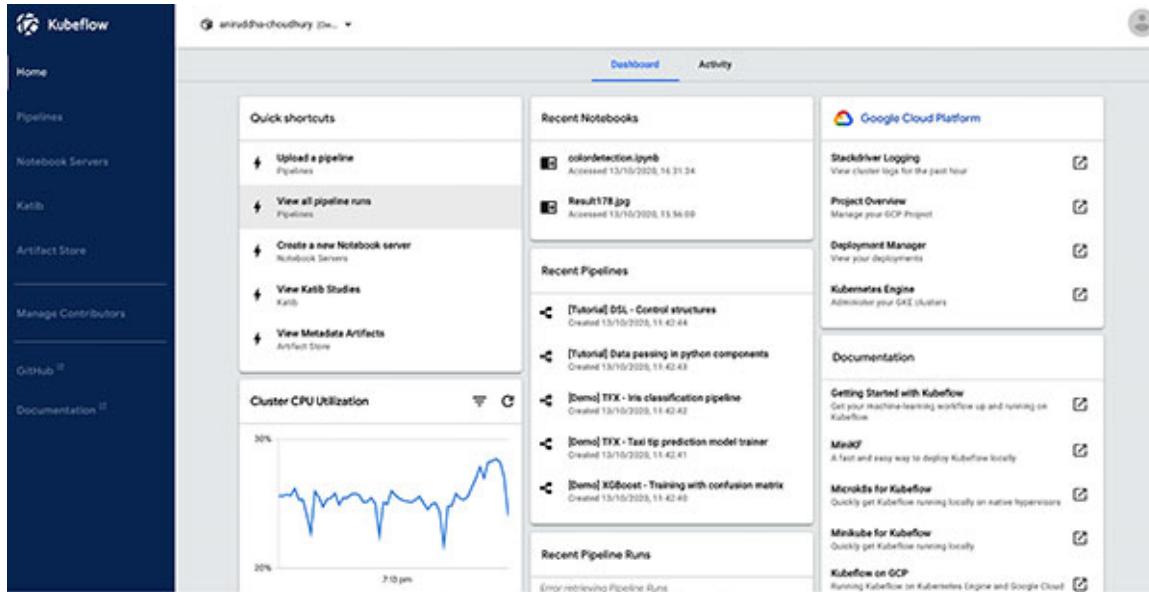


Figure 1.19: Kubeflow Dashboard

1.5.2 Registration Flow

Depending upon the setup of your Kubeflow cluster, you may need to create a namespace when you first log in to Kubeflow. Namespaces are sometimes called profiles or workgroups.

For Kubeflow deployments that support single-user isolation, the Kubeflow cluster has no namespace role bindings.



Welcome

In order to use Kubeflow, a namespace for your account must be created. Follow the steps to get started

Start Setup



Figure 1.20: Kubeflow Profile Setup

Click on the Start Setup button and follow the instructions on the screen to set up your namespace. The default name for your namespace is your username.

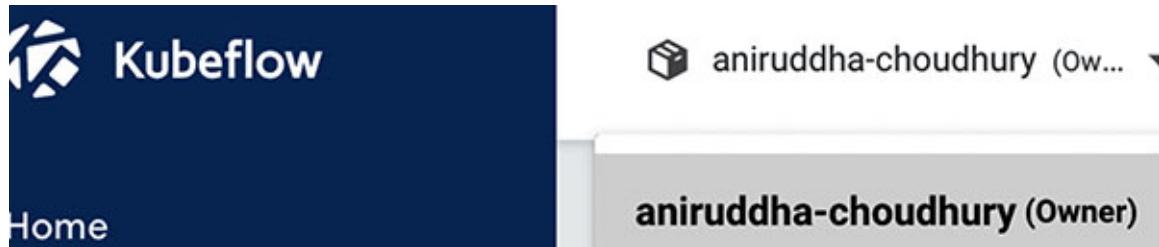


Figure 1.21: Kubeflow Namespace

1.5.3 Metadata

The goal of the Metadata project is to help the Kubeflow users understand and manage their **machine learning (ML)** workflows by tracking and managing

the metadata that the workflows produce. Complete the following steps:

- Go to Kubeflow in your browser.
- Click on Artifact Store on the left-hand navigation panel.
- The Artifacts screen opens and displays a list of items for all the metadata events that your workflows have logged. You can click on the name of each item to view the details.



The screenshot shows the Kubeflow interface with the title 'Kubeflow' and a user profile 'anudatta-choudhury'. On the left, there's a navigation bar with 'Artifacts' selected. The main area is titled 'Artifacts' and contains a search bar labeled 'Filter'. Below the search bar is a table header with columns: 'Name', 'Version', 'Type', 'URI', 'Workspace', and 'Created at'. A message 'No artifacts found.' is displayed below the table.

Figure 1.22: Kubeflow Metadata

1.5.4 Jupyter Notebook server

Click on **Notebook Servers** on the left-hand panel of the Kubeflow UI to access the Jupyter notebook services deployed with Kubeflow:

The screenshot shows the Kubeflow UI dashboard. On the left sidebar, there is a list of options: Home, Pipelines, Notebook Servers (which has a red arrow pointing to it), Katib, Artifact Store, GitHub, Documentation, and Privacy + Usage Reporting. The main area of the dashboard is titled 'default'. It features a 'Quick shortcuts' section with links to upload a pipeline, view pipeline runs, create a new notebook server, view Katib studies, and view metadata artifacts. Below this is a 'Recent Notebooks' section which says 'No Notebooks in namespace default'. To the right of these sections is a 'Google Cloud Platform' sidebar with links to Stackdriver Logging, Project Overview, Deployment Manager, Kubernetes Engine, and MiniKF. At the bottom of the dashboard, there is a 'Cluster CPU Utilization' chart showing 20% usage.

Figure 1.23: Kubeflow Jupyter Notebook

The screenshot shows the 'Notebook Servers' dashboard. At the top, there is a header with 'Notebook Servers' and a '+ NEW SERVER' button. Below the header is a table with columns: Status, Name, Age, Image, GPU, Memory, Volumes, and Actions. There is one entry in the table: 'kf-serving' (Status: green checkmark, Age: 4 days ago, Image: computervision-env:v1.15:v1, GPU: 3, Memory: 4.0Gi, Volumes: 2, Actions: CONNECT and a trash bin icon).

Status	Name	Age	Image	GPU	Memory	Volumes	Actions
✓	kf-serving	4 days ago	computervision-env:v1.15:v1	3	4.0Gi	2	CONNECT █

Figure 1.24: Kubeflow Jupyter Notebook dashboard

1.5.5 Katib

Use Katib for automated tuning of your ML model's hyper parameters and architecture. For more information, go to the following link:
<https://www.Kubeflow.org/docs/components/hyperparameter-tuning/overview/>

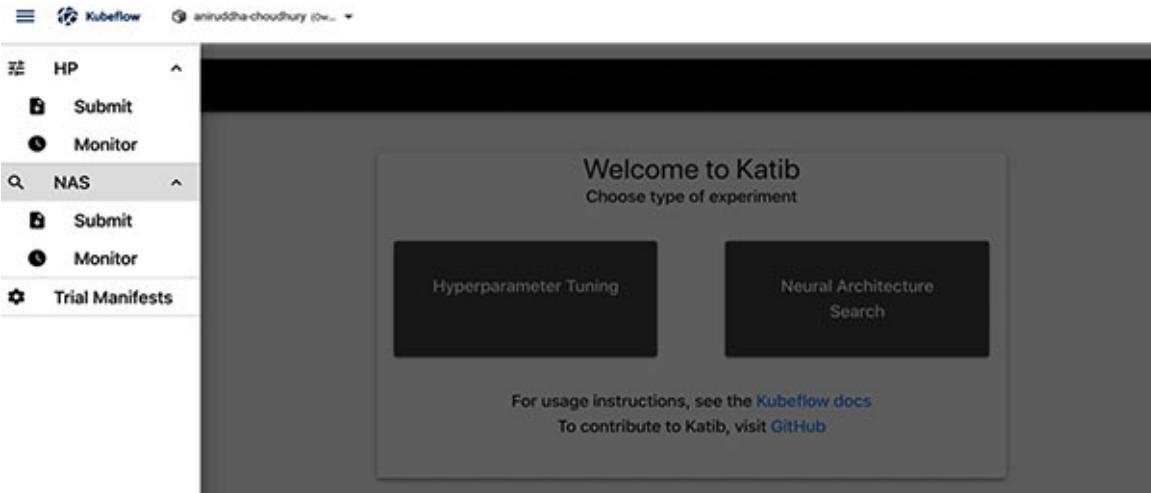


Figure 1.25: Kubeflow katib dashboard

1.6 Getting Started in GCP Kubeflow setup

Prerequisites: You must have an active GCP account and while you practice this chapter, as it might charge for running the Kubernetes cluster.

I assume some basic Kubernetes and Docker knowledge is a must.

If you are using the GCP Free Tier or the 12-month trial period with \$300 credit, note that you can't run the default GCP installation of Kubeflow, because the free tier does not offer enough resources. You need to upgrade to a paid account.

Please follow the following link to set up the GCP project and make sure it will charge you once after the free credits from your credit card, and enable all the API required for our work: <https://v1-0-branch.Kubeflow.org/docs/gke/deploy/project-setup/>

- Creating a **Google Cloud Platform (GCP)** project for your Kubeflow deployment.
- Making sure that you have the owner role for the project.
- Making sure that billing is enabled for your project.
- Going to the GCP Console will ensure specified APIs are enabled:
 - Compute Engine API
 - Kubernetes Engine API
 - Identity and Access Management (IAM) API

- Deployment Manager API
- Cloud Resource Manager API
- Cloud Filestore API
- AI Platform Training and Prediction API
- Cloud Build API



Figure 1.26: GCP API & Service Dashboard

The following are to be installed for Kubeflow on the command line:

- Ensure you have installed the following tools:
 1. kubectl.
 2. gcloud
- If you’re using Cloud Shell, enable boost mode.
- Next, please make sure that your GCP project meets the minimum requirements described in the project setup guide.

- Follow the guide setting up the OAuth credentials. To create the OAuth credentials for **Cloud Identity-Aware Proxy (Cloud IAP)**, click on the following link:

<https://v1-0-branch.Kubeflow.org/docs/gke/deploy/>

1.6.1 Install and Set Up kubectl

Now, let's first install kubectl gcloud sdk, which will be used for connecting Kubernetes.

Install with Homebrew on macOS:

If you are on macOS and using the Homebrew package manager, you can install kubectl with Homebrew:

1. Run the installation command:

```
$ brew install kubectl
```

2. Test to ensure the version you installed is up-to-date:

```
$ kubectl version --client
```

For the other OS, follow the link for Linux or Windows or MacOs:

<https://kubernetes.io/docs/tasks/tools/install-kubectl/>

1.6.2 Install and Set Up gcloudsdk

Check which version (64-bit or 32-bit) your OS is running on.

Linux / macOS: Run `getconf LONG_BIT` from your command line.

Windows: Control Panel | System | System Type

Run the following command:

```
```bash
$ curl -O
https://dl.google.com/dl/cloudsdk/channels/rapid/downloads/google-
cloud-sdk-302.0.0-linux-x86_64.tar.gz
$ tar zxvf google-cloud-sdk-302.0.0-linux-x86_64.tar.gz
$/google-cloud-sdk/install.sh
$ gcloud version
```

```

```
Google Cloud SDK 329.0.0
app-engine-python 1.9.91
beta 2021.02.19
bq 2.0.65
cloud-datastore-emulator 2.1.0
core 2021.02.19
gsutil 4.59
kpt 0.33.0
kubectl 1.17.17
```

Figure 1.27: Gcloud version

Now, to install the google cloud sdk, please click on the following link:

<https://cloud.google.com/sdk/docs/downloads-versioned-archives>

1.6.3 Set Up OAuth from Cloud IAP

In this section, we will set up the OAuth for Cloud IAP to get the secret key. So, the OAuth set up requires a few steps.

First create the OAuth Consent screen from the APIs and Service in GCP, and complete the following steps:

- In the **Application name** box, enter the name of your application. The following example uses the name “Kubeflow”.
- Under **Support email**, select the email address that you want to display as a public contact. You must use either your email address or a Google Group that you own.
- If you see **Authorized domains**, enter the following:

<project>.cloud.goog

Here, **<project>** is your GCP project ID.

If you are using your own domain, such as **acme.com**, you should add that as well.

The Authorized domains option appears only for certain project configurations. If you don’t see the option, then there’s nothing you need to set.

- Click on **Save**.

Please go to the following link and do the same that was provided: <https://v1-0-branch.Kubeflow.org/docs/gke/deploy/oauth-setup/>

Step: Go to **APIs & Service | OAuth consent screen | Credentials**

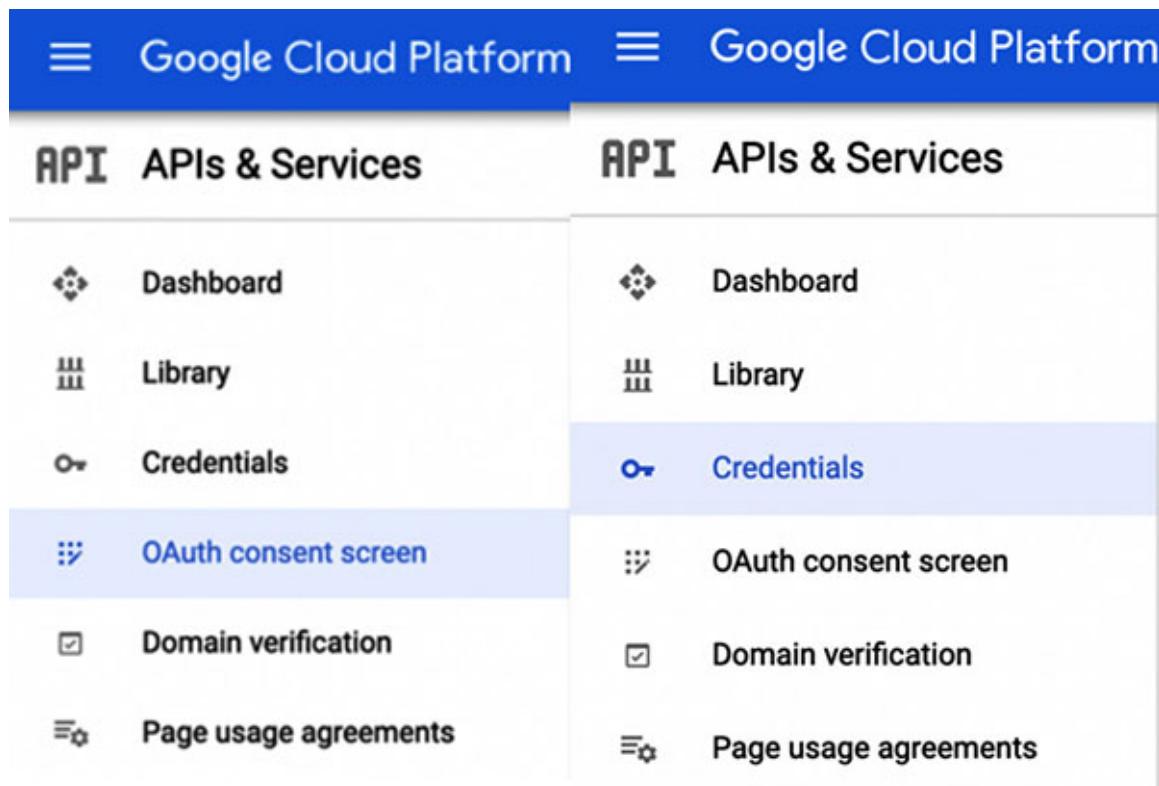


Figure 1.28: GCP APIs & Service

Please note: Copy the few important points after the preceding steps, which will be required for the Kubeflow Deployment later.

To do that, complete the following steps:

1. Copy the **CLIENT_ID** and **CLIENT_SECRET** in a note for future use.

| | |
|---------------|--|
| Client ID | 449053263861-j66phntgagp35hl190bqc38o1rojsk04.apps.googleusercontent.com |
| Client secret | 2y7tNtbbH_SJZ_irWP2_jJPo |

Figure 1.29: OAuth ID & Secret Key

2. Change the Authorized redirect URIs in OAuth consent screen.

https://iap.googleapis.com/v1/oauth/clientIds/<CLIENT_ID>:handleRedirect

3. Paste the CLIENT_ID in this URL and paste in the Authorized redirect URIs in OAuth consent screen.

Please click on the following link for a detailed understanding:

<https://www.Kubeflow.org/docs/gke/deploy/oauth-setup/>

1.6.4 Set Up Docker

Now, we will set up the Docker which is important to build the container Image.

Pre-requisites:

You should have the Docker Hub account <https://hub.Docker.com/>

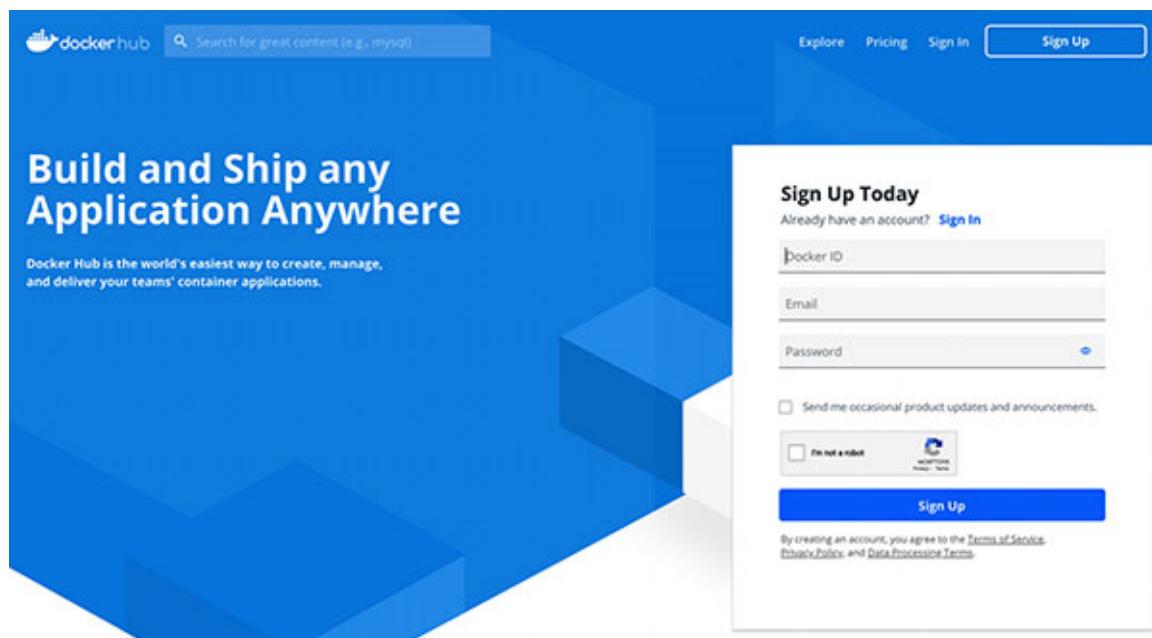


Figure 1.30: Docker Hub Sign hub page

Once you install the Docker for MacOS, Linux, or Windows, you have to login to the application with the same credentials of the Docker hub with the DOCKER_ID and PASSWORD.

Please click on the following link for the installation for Docker:

<https://docs.Docker.com/Docker-for-mac/install/>

1.6.5 Set Up Kubeflow in Kubernetes Cluster in GCP

To setup Kubeflow in Kubernetes cluster, either you use the Google cloud shell or Microsoft Visual Studio Terminal of your system. Here, we will use the Google cloud shell.

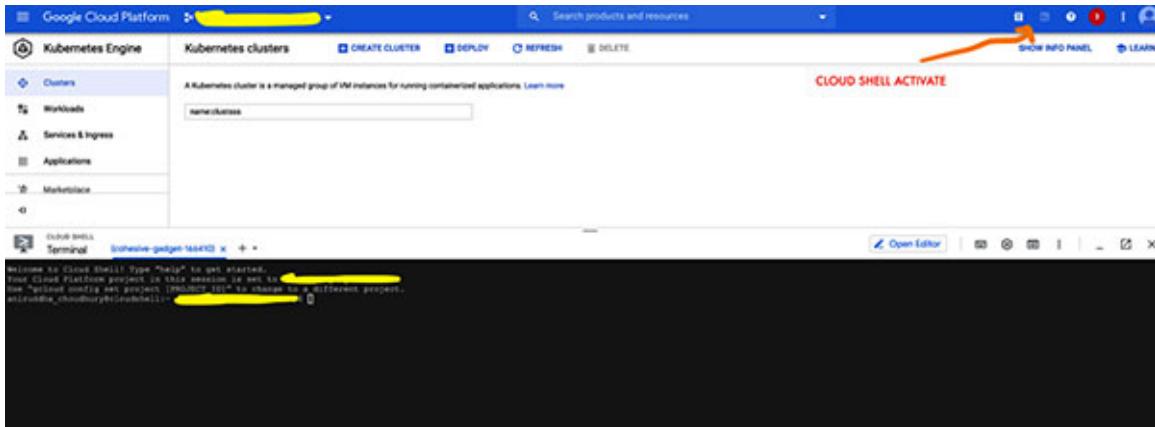


Figure 1.31: Activate cloud shell

Now, we will activate the boost shell.

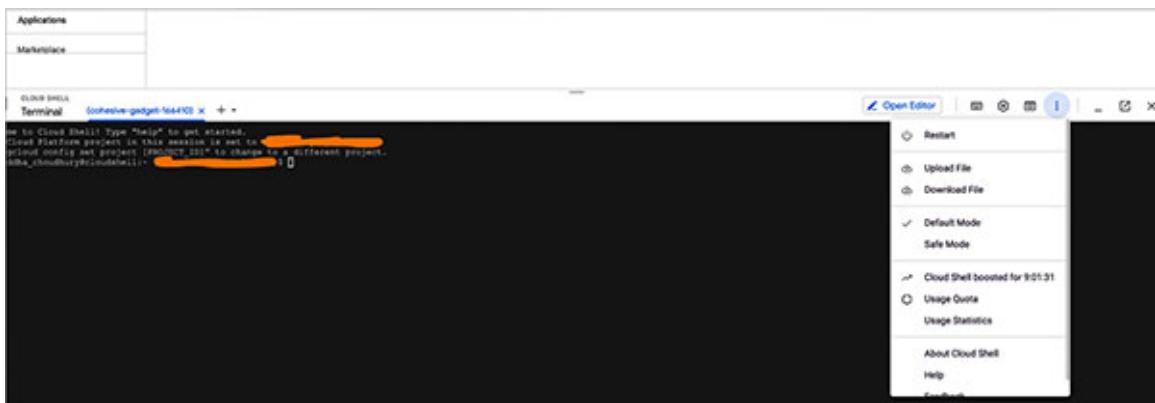


Figure 1.32: Activate cloud shell Boost Mode

Please complete the following steps in cloud shell.

The only thing which we need to give is **CLIENT_ID**, **CLIENT_SECRET**, **PROJECT_ID**, **ZONE**, **CLUSTER_NAME** and change the **CONFIG_URI** according to the version of Kubeflow release yaml file.

And you have this below code yaml file in GitHub. Please run the step 5 after all the activities of the chapter to incur cost in google cloud.

Deploy using CLI: This guide describes how to use the **kubectl command line interface (CLI)** to deploy Kubeflow on GCP. The command line deployment gives you more control over the deployment process.

```
## Setting Up Kubeflow on Google Cloud Platform
Note: Use latest official Kubeflow documentation available at **
https://v1-0-branch.Kubeflow.org/docs/gke/deploy/deploy-cli/** to
install the latest release and config files.
latest release are available at
**https://github.com/Kubeflow/kfctl/releases**

### Step 1 : Setup GCP
```bash
login to gcloud for authentication : done once
gcloud auth login
create application default credentials : done once
gcloud auth application-default login
GCP Project ID
export PROJECT=<PROJECT_ID>
gcloud config set project ${PROJECT}
GCP Zone (use us-east1-c)
export ZONE=<ZONE>
gcloud config set compute/zone ${ZONE}
```

### Step 2 : download the release based on your Operating system
```bash
KFCTL file url : Get the latest file from
https://github.com/Kubeflow/kfctl/releases based on the
operating system
KFCTL_FILE_PATH="https://github.com/Kubeflow/kfctl/releases/download/v1.0.2/kfctl_v1.0.2-0-ga476281_linux.tar.gz"
KFCTL_FILE="kfctl.tar.gz"
download KFCTL compressed file
wget $KFCTL_FILE_PATH -O $KFCTL_FILE
extract KFCTL
tar -xvf${KFCTL_FILE}
#mv kfctl-${PLATFORM} kfctl
add KFCTL to path
PATH=${PATH}:$(pwd)
```

```

Step 3: setup the deployment

Copy and paste the Client ID and the secret in step C (OAuth Setup Cloud).

```bash

```
Deployment Name e.g.
export KF_NAME=<CLUSTER_NAME>
set client ID and client secret
export CLIENT_ID=<CLIENT_ID>
export CLIENT_SECRET=<CLIENT_SECRET>

set the config URI : use the official documentation to use the
latest config file
get latest config URI from official Kubeflow documentation :
https://v1-0-branch.Kubeflow.org/docs/gke/deploy/deploy-cli/
#export
CONFIG_URI=https://raw.githubusercontent.com/Kubeflow/manifests/v1
.0-branch/kfdef/kfctl_gcp_iap.v1.0.2.yaml
```

Copy the preceding URL to the following command in bash:

```

```
export CONFIG_URI="<>CONFIG_URI>"  
# set the base directory  
BASE_DIR=$(pwd)  
# set the directory for deployment  
KF_DIR=${BASE_DIR}/${KF_NAME}  
# create the directory  
mkdir -p ${KF_DIR}  
# navigate to the directory  
cd${KF_DIR}  
# build deployment using the config file. Make changes in your  
configuration if needed  
kfctl build -v -f ${CONFIG_URI}  
```  
Step 4: Deploy
```bash
```

```
# setup the config file
```

Now, that you are in the present folder `${KF_DIR}`, run the following command to check files:

```
```bash
```

```
ls
```

```
```
```

```
gcp_config kfctl_gcp_iap.v1.0.2.yaml kustomize
```

Figure 1.33: Kubeflow installation yaml files

Now, from preceding figure, we can copy the `kfctl_gcp_iap.v1.0.2.yaml` and paste in the following `<CONFIG_FILE_NAME>` to set the environment variables to deploy that yaml.

Next, go to the `gcp_config` folder from earlier, and run vim `cluster-Kubeflow.yaml` and change the cluster-version to “1.19” and save the file, and go back to the root folder of `{KF_DIR}`.

```
export CONFIG_FILE=${KF_DIR}/<CONFIG_FILE_NAME>
# apply changes
kfctl apply -v -f ${CONFIG_FILE}
gcloud container clusters get-credentials ${KF_NAME} --zone
${ZONE} --project ${PROJECT}
```
```

Please use run the step 5 after all the activities to delete everything at the end of the project to incur cost in google cloud.

```
Step 5 : Delete
```

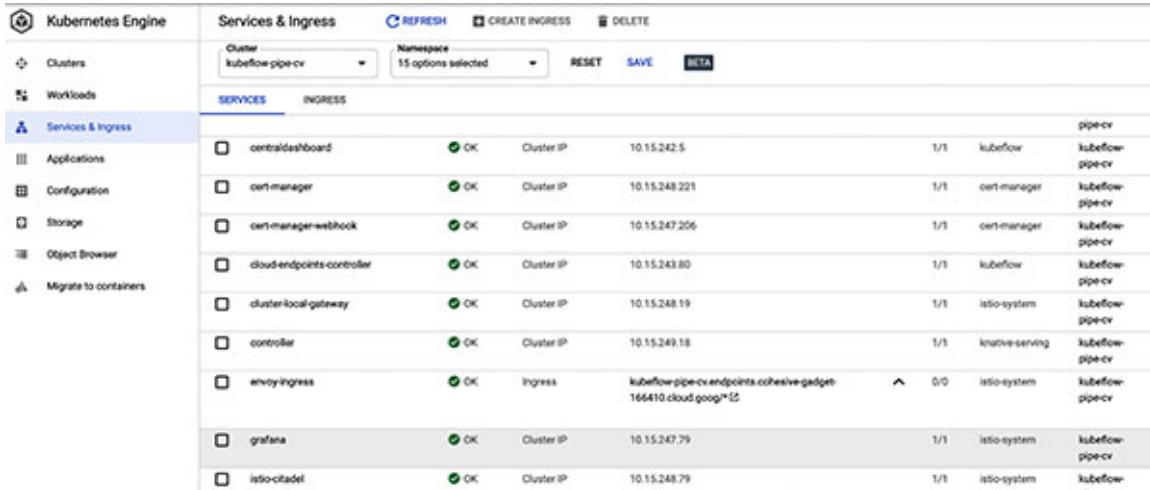
```
```bash
```

```
# If you want to delete all the resources, including storage:
```

```
kfctl delete -f ${CONFIG_FILE} --delete_storage
```
```

Now, wait for 15-20 minutes for the Ingress to be ready. Let’s open the Ingress to check; click on **Kubernetes Engine | Service & Ingress**, and in the name,

find the **envoy-ingress**; see the following example as the URL where the Kubeflow is hosted; click to open the Kubeflow dashboard.



The screenshot shows the Kubernetes Engine interface with the 'Services & Ingress' tab selected. The left sidebar includes 'Clusters', 'Workloads', 'Applications', 'Configuration', 'Storage', 'Object Browser', and 'Migrate to containers'. The main area displays a table of services:

|   | SERVICE                    | STATUS | TYPE       | PORTS                           | PORTS | SELECTOR        | LAST UPDATE |
|---|----------------------------|--------|------------|---------------------------------|-------|-----------------|-------------|
| 1 | centraldashboard           | OK     | Cluster IP | 10.15.242.5                     | 80    | kubeflow        | 10m         |
| 2 | cert-manager               | OK     | Cluster IP | 10.15.248.221                   | 80    | cert-manager    | 10m         |
| 3 | certmanager-webhook        | OK     | Cluster IP | 10.15.247.206                   | 80    | cert-manager    | 10m         |
| 4 | cloud-endpoints-controller | OK     | Cluster IP | 10.15.243.80                    | 80    | kubeflow        | 10m         |
| 5 | cluster-local-gateway      | OK     | Cluster IP | 10.15.248.19                    | 80    | istio-system    | 10m         |
| 6 | controller                 | OK     | Cluster IP | 10.15.249.18                    | 80    | knative-serving | 10m         |
| 7 | envoy-ingress              | OK     | Ingress    | kubeflow-pipe-cv.endpoints.kube | 80    | istio-system    | 10m         |
| 8 | grafana                    | OK     | Cluster IP | 10.15.247.79                    | 80    | istio-system    | 10m         |
| 9 | istiod                     | OK     | Cluster IP | 10.15.248.79                    | 80    | istio-system    | 10m         |

**Figure 1.34: Kubeflow Service & Ingress URL**

Click on the following URL which is shown in the preceding figure:

**<https://<application-name>.endpoints.<project-id>.cloud.google/>**

If you want to run Kubeflow 1.3 please check below link:

**<https://github.com/aniruddhachoudhury/mlopsworld>**

## 1.6.6 Connect to cluster and Deploy Grafana

```
Setup knative-monitoring
```bash
##RUN THE COMMAND IN GOOGLE SHELL
# Connect to cluster
gcloud container clusters get-credentials <$ClusterName> --zone
<$ZONE> --project <$PROJECTID>
#create namespace
kubectl create namespace knative-monitoring
#setup monitoring components
kubectl apply --filename
https://github.com/knative/serving/releases/download/v0.17.2/monitoring-metrics-prometheus.yaml```

```

1.6.7 Jupyter Notebook server setup in Kubeflow

Let's create a custom Jupyter lab or Notebook for all the operations and projects in this book.

```
1  FROM gcr.io/kubeflow-images-public/tensorflow-2.1.0-notebook-cpu:1.0.0
2
3  USER root
4
5  COPY . .
6  RUN pip3 install -r requirements.txt
7
8  tensorflow-datasets==2.1.0
9  jupyterlab==2.1.1
10 google-cloud-storage==1.26.0
11 plotly==4.10.0
12 streamlit==0.66.0
13 matplotlib==3.3.2
14 seaborn==0.11.0
15 altair==4.1.0
16 pandas==1.1.2
17 kf-serving==0.3.0
18 numpy==1.18.1
19 scikit-learn==0.22.2.post1
20 kubeflow-metadata==0.3.1
21 Pillow==7.0.0
```

Figure 1.35: Custom Docker image Jupyter NB

Now, we will build the custom Docker image; for that, make sure that you start the Docker. You will see the Docker is activated on top, as we can see the first icon in the following figure, which indicates that the Docker is started:



Figure 1.36: Docker Activation in your system

```
```bash
PROJECT_ID=$(gcloud config get-value core/project)
IMAGE_NAME=custom-image-1
IMAGE_NAME=gcr.io/$PROJECT_ID/$IMAGE_NAME
IMAGE_TAG=latest

build image
Docker build -t $IMAGE_NAME:$IMAGE_TAG .
run locally to test
Docker run -it --rm -p 8888:8888 -p 6006:6006 -v
$(pwd):/home/jovyan $IMAGE_NAME:$IMAGE_TAG
authorize Docker
gcloud auth configure-Docker --quiet
push image
Docker push $IMAGE_NAME:$IMAGE_TAG
```

```

So, we will use the custom Docker image in the Jupyter Notebook.

`gcr.io/custom-image-1:latest`

Complete the following steps:

1. Click on **Notebook Servers** on the left-hand panel of the Kubeflow UI to access the Jupyter notebook services deployed with Kubeflow:

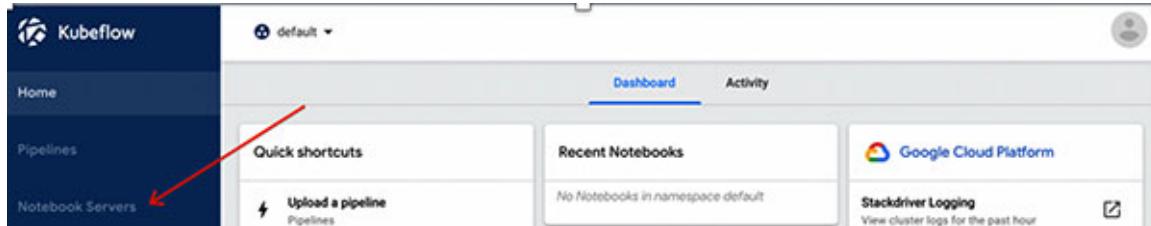


Figure 1.37: Kubeflow NB Dashboard

2. Now click on the **NEW SERVER** to create a new Jupyter Notebook.



Figure 1.38: Click New server

3. Enter a name of your choice for the notebook server. The name can include the letters and numbers, but no spaces. For example, **my-first-notebook**.

Kubeflow automatically updates the value in the namespace field to be the same as the namespace that you selected in a previous step. This ensures that the new notebook server is in a namespace that you can access.

Name

Specify the name of the Notebook Server and the Namespace it will belong to.

| | |
|--------------|--------------------|
| Name | Namespace |
| kubeflownote | kubeflow-anichoud2 |

Image

A starter Jupyter Docker Image with a baseline deployment and typical ML packages.

Custom Image

Custom Image
`gcr.io/cohesive-gadget-166410/custom-image-1:latest`

Figure 1.39: Configuration Name for Notebook

4. Select a Docker image for the baseline deployment of your notebook server. You can specify a custom image or choose from a range of standard images. Here, we will use **Custom Image**. Paste the preceding Docker Image: `gcr.io/custom-image-1:latest`. Specify the total amount of CPU that your notebook server should reserve. The default is 0.5. For the CPU-intensive jobs, you can choose more than one CPU (for example, 1.5). Specify the total amount of memory (RAM) that your notebook server should reserve. The default is 1.0Gi.



Figure 1.40: Configuration image and CPU & Memory

5. Specify a workspace volume to hold your personal workspace for this notebook server. Kubeflow provisions a Kubernetes **persistent volume (PV)** for your workspace volume. The PV ensures that you can retain the data even if you destroy your notebook server.

The default is to create a new volume for your workspace with the following configuration:

Name: The volume name is synced with the name of the notebook server, and has the form workspace-<server-name>. When you start typing the notebook server name, the volume name appears. You can edit the volume name, but if you later edit the notebook server name, the volume name changes to match the notebook server name.

Size: 10Gi

Access mode: ReadWriteOnce. This setting means that the volume can be mounted as read-write by a single node.

Mount point: /home/jovyan

Alternatively, you can point the notebook server to an existing volume by specifying the name of the existing volume.

(Optional) Specify one or more data volumes if you want to store and access data from the notebooks on this notebook server. You can add new volumes or specify the existing volumes. Kubeflow provisions a Kubernetes **persistent volume (PV)** for each of your data volumes.

The screenshot shows two sections of the Kubeflow UI for configuring storage.

Workspace Volume Configuration:

- Type:** New
- Name:** workspace-kubeflownote
- Size:** 10Gi
- Mode:** ReadWriteOnce
- Mount Point:** /home/jovyan

Data Volumes Configuration:

- + ADD VOLUME** button
- Type:** New
- Name:** kubeflownote-vol-1
- Size:** 10Gi
- Mode:** ReadWriteOnce
- Mount Point:** /home/jovyan/data-vol-1
- Delete icon** (for removing the volume)

Figure 1.41: Configuration PVC & Volume

6. Select **add gcp credentials** from the drop down. Similarly, to use the GPU image, we have to give a custom GPU Image which will be there in GitHub. But, as we will work on CPU, as of now, we will not provide any GPU here.

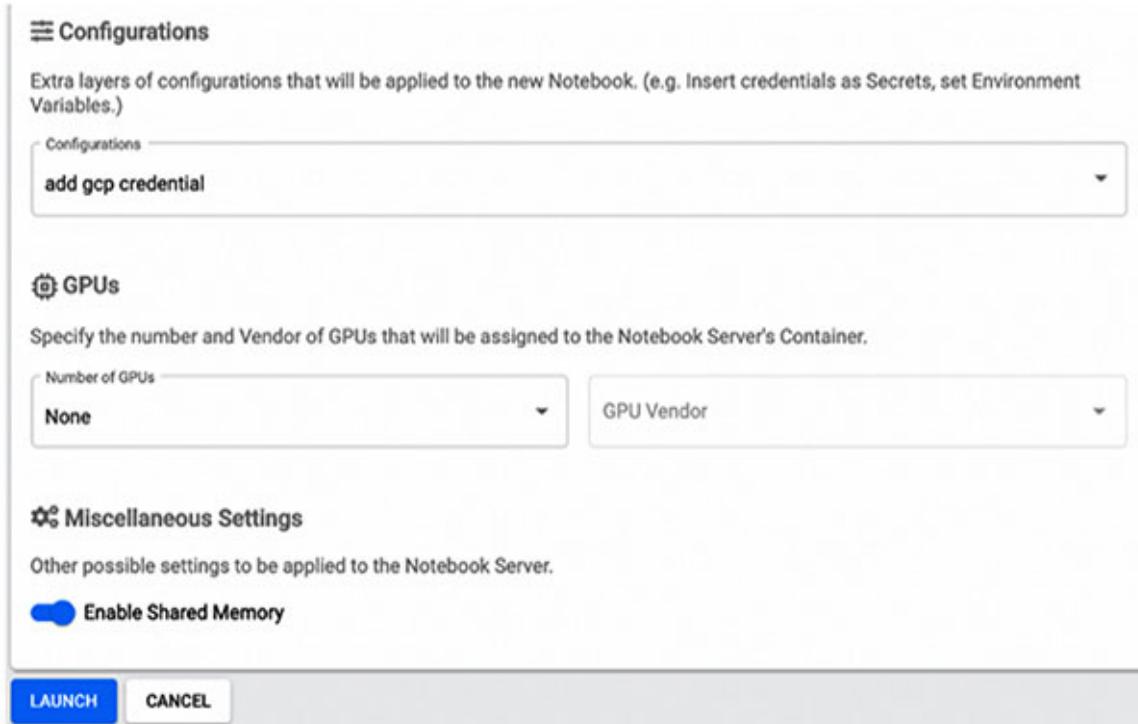


Figure 1.42: Configuration/GPU/Launch

7. Click on **LAUNCH**. You should see an entry for your new notebook server on the Notebook Servers page, with a spinning indicator in the Status column. The following is the sample Notebook which we have created:



Figure 1.43: Notebook ready

8. When the notebook server is running, you should see the Jupyter dashboard interface. If you requested a new workspace, the dashboard should be empty of notebooks.

1.7 Optional: PVC setup for Jupyter Notebook

If you want to attach your PVC to the file store, so that you can save your data or codes in the Jupyter Notebook to external storage, then we can complete the following steps:

Optional: (Cost will be applied for 1TB storage)

Step 1: Create a Filestore instance with 1 TB of storage capacity.

```
```bash
FS=[NAME FOR THE FILESTORE YOU WILL CREATE]
gcloud beta filestore instances create ${FS} \
 --project=${PROJECT} \
 --zone=${ZONE} \
 --tier=STANDARD \
 --file-share=name="volumes",capacity=1TB \
 --network=name="default"
````
```

Step 2: Retrieve the IP address of the Filestore instance.

```
```bash
FSADDR=$(gcloud beta filestore instances describe ${FS} \
 --project=${PROJECT} \
 --zone=${ZONE} \
 --format="value(networks.ipAddresses[0]))")
````
```

Step 3: Connect to the cluster which you have created.

```
```bash
export ZONE=us-east1-c
export CLUSTER_NAME= {Your-cluster name}
gcloud container clusters get-credentials <CLUSTER_NAME>--zone
<ZONE> --project <PROJECT>
````
```

Step 4: Grant yourself the cluster-admin privileges.

```
```bash
ACCOUNT=$(gcloud config get-value core/account)
kubectl create clusterrolebinding core-cluster-admin-binding \
 --user ${ACCOUNT} \
 --clusterrole cluster-admin
````
```

Step 5: Download Helm.

```
```bash
wget https://storage.googleapis.com/kubernetes-helm/helm-v2.11.0-
linux-amd64.tar.gz
tar xf helm-v2.11.0-linux-amd64.tar.gz
sudo ln -s $PWD/linux-amd64/helm /usr/local/bin/helm
```

```

Step 6: Create a file named `rbac-config.yaml` containing the following:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: tiller
  namespace: kube-system
```

Step 7: Create the tiller service account and cluster-admin role binding.

```
```bash
kubectl apply -f rbac-config.yaml
```

```

Step 8: Initialize Helm.

```
```bash
helm init --service-account tiller

```

```

Step 9: Copy the namespace that you can find by either running the following command which will give a list of namespace, or from the UI.

```bash

```
kubectl get namespace
```

```

Otherwise, copy the drop down and paste in the Notepad example here: aniruddha-choudhury.

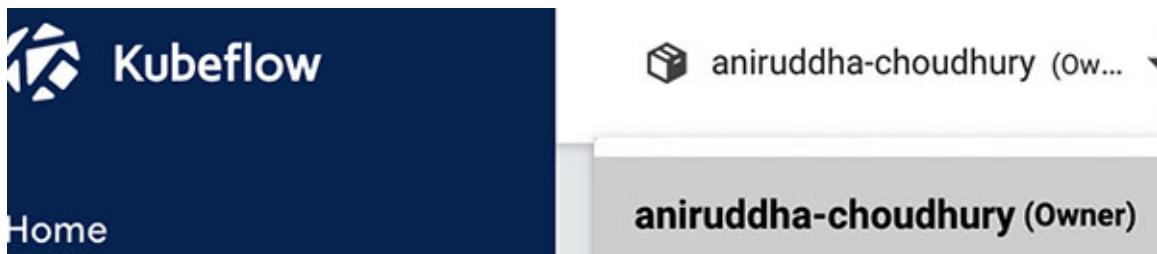


Figure 1.44: Namespace

Step 10: Deploy the NFS-Client Provisioner.

Create an instance of NFS-Client Provisioner connected to the Filestore instance that you created earlier via its IP address (`${FSADDR}`). The NFS-Client Provisioner creates a new storage class: `nfs-client`. Persistent volume claims against that storage class will be fulfilled by creating persistent volumes backed by directories under the `/volumes` directory on the Filestore instance's managed storage.

```bash

```
helm install stable/nfs-client-provisioner --name nfs-cp --set
nfs.server=${FSADDR} --set nfs.path=/volumes
watch kubectl get po -l app=nfs-client-provisioner
```

```

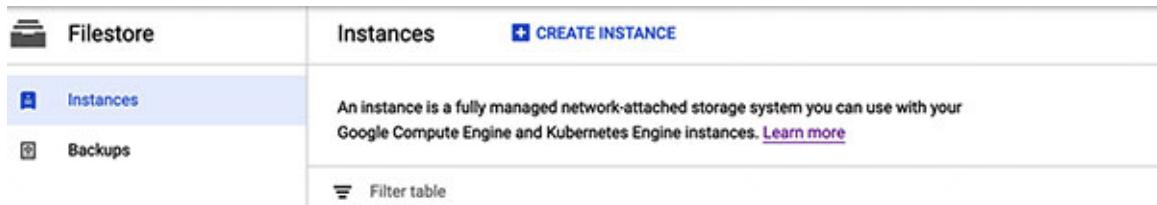


Figure 1.45: FileStore

Step 11: Make a Persistent Volume Claim.

Create a Kubernetes Persistent Volume Claim specification. This is a `.yaml` file that allows a Kubernetes pod to access the storage resources of a Persistent Volume. The specification looks similar to the following example:

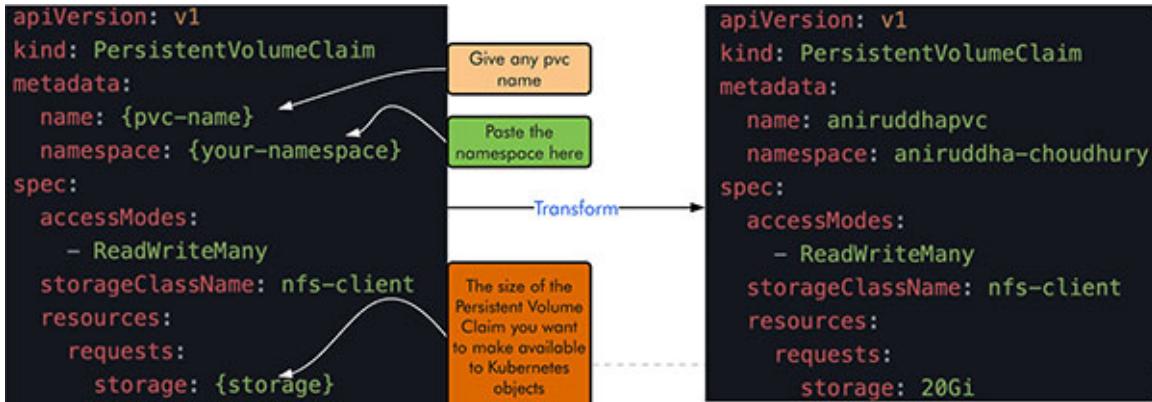


Figure 1.46: PVC yaml

Here, storage is the size of the Persistent Volume Claim that you want to make available to the Kubernetes objects.

You must specify the storage value in one of the supported units described in the Resource quantities. The value you specify must be equal to or less than the storage you specified for the Persistent Volume.

Deploy the Persistent Volume Claim specification:

```
kubectl create -f persistent-volume-claim-file-name.yaml
```

Here, `persistent-volume-file-name` is the name of the Kubernetes Persistent Volume specification file that you created in the previous step.

Step 12: Copy the PVC name from above and paste it in the Jupyter Notebook in Kubeflow Workspace Volume Name.

```
```bash
paste the namespace here to get the name
kubectl get pvc -n {namespace}
```

```

1.8 Conclusion

In this chapter, we learned about the Kubernetes Orchestration concepts and architecture, and how it works in deployment level and scaling our application, alongside exposing to the outer world with the Network & Service.; and how the Docker makes it easy for a developer to containerize applications.

Then, we saw how to set up the Jupyter Notebook with CPU and GPU with the PVC attached and how to setup the PVC to store our required data for lifetime. We also learned more about deployment, service, and the ingress framework structure in Google Cloud. Finally, we learned how to setup Kubeflow in Google Cloud Platform for our following chapters.

In this chapter, we have gained how to leverage the power of Google Cloud Platform, and how to use your Devops knowledge with Machine Learning to become a MLops.

1.9 Reference

- <https://kubernetes.io/docs/tutorials/kubernetes-basics/>
- <https://kubernetes.io/docs/reference/>
- <https://github.com/kubernetes/kubernetes>
- <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>

CHAPTER 2

Developing Kubeflow Pipeline in GCP

In this chapter, we will build an end-to-end TensorFlow classification Model deployment with Kubeflow Orchestration, which includes deploying Kubeflow in Kubernetes Cluster in GCP, building the pipeline components for the model with Docker and Kubeflow SDK, and then serving the Model with KF serving to have an endpoint for prediction. We will also track the monitoring and performance for our serving traffic endpoint in Grafana Dashboard.

Structure

In this chapter, we will cover the following topics:

- Problem statement
- Getting started in GCP Kubeflow and Docker setup
- Breakdown technique to build the production pipeline
- Building the Kubeflow Pipeline components for TensorFlow model
- Serving the Model with KF Serving
- Building the pipeline end to end
- Monitoring the performance with Grafana dashboard

Objectives

In this chapter, we will learn the following:

- How to use Docker and Kubernetes to build the Kubeflow Pipeline.
- How to build the individual pipeline components like the training and model evaluation.
- How to serve the Model with KF serving and predict the model request, and monitor with the Grafana Dashboard.

- How to use Kubernetes, and many Google Cloud Platform to leverage the power of Machine learning with Devops Knowledge.

2.1 Problem statement

In this example, we have a classification dataset of breast cancer, and it will have 30 attributes. We have to classify the Malignant and Benign.

| | |
|-------------|---|
| NOTE | Rest all the imports I have showed in my Colab Notebook, for which the hyperlink of GitHub Account of this chapter is given below. Note Colab platform Python 3.x. RUN IN GOOGLE COLAB. |
| CODE | https://github.com/bpbpublications/Continuous-Machine-Learning-with-Kubeflow/tree/main/Chapter2 |

2.2 Getting started in GCP Kubeflow setup

So, before we begin, we must setup the Kubeflow Cluster in GCP; please refer to [Chapter 1: Introduction to Kubeflow & Kubernetes Cloud Architecture](#), section 1.6 (Getting Started in GCP Kubeflow setup).

2.3 Breakdown technique to build production pipeline

In this section, we will see how to frame your business problem, so that we can break down the Machine learning life cycle components, and build each component, so that we can build the Kubeflow pipeline.

The following is the high level framework from the Jupyter Notebook code to the Pipeline components, and serving and hosting that in Cloud.

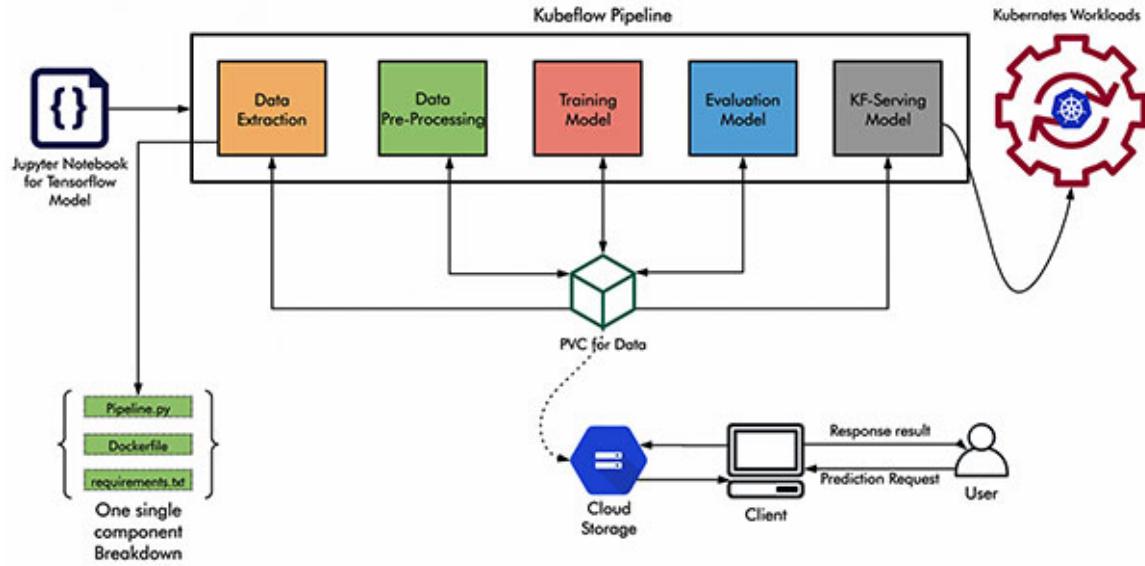


Figure 2.1: Pipeline Components E2E Architecture

We will Dockerize each and every component, and push to the container registry of any Cloud, like for AWS, GCP, Azure, so that we can use that in the Kubeflow pipeline.

Steps to perform:

Data Extraction or Ingestion: Let's say we have our machine learning; the first component is to extract the data from any sources like S3 bucket, Cloud Storage, and so on. Now, we will write the single component of the Input data sources and connection between Kubeflow components to all the data sources in the Python code.

And a next a service account in json format will be used so that we can connect to external source. Then, we will Dockerize the component.

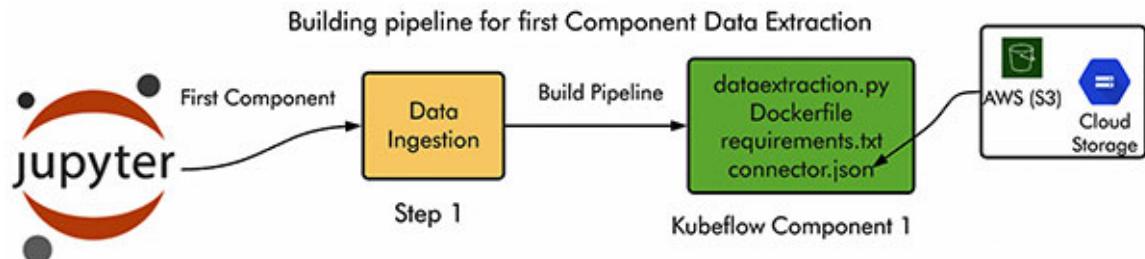


Figure 2.2: Pipeline Component 1: Data Ingestion

Data Pre-processing: The next component in our machine learning project is to pre-process or feature the engineering step. Here, we will build or transform

all the incoming data which will be required for the training, according to all the use case like Computer vision, NLP, Structured data with the respective business pre-processing logic, and we will save the artifacts or the transformed data back to the Storage bucket for future reference.

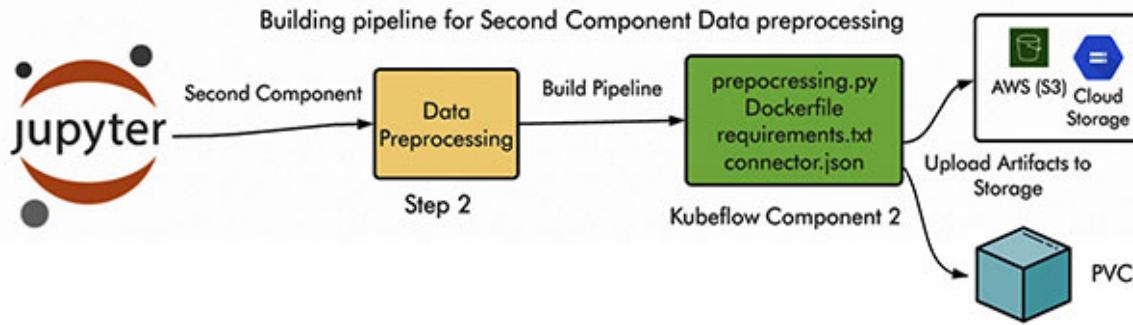


Figure 2.3: Pipeline Component 2: Data Pre-processing

Training: So, here we will train the model it can any TensorFlow or Sklearn, and so on with our pre-process data, and will save the model artifacts and logs in the external storage and internally in the native Kubernetes' **Persistent Volume Claim (PVC)**.

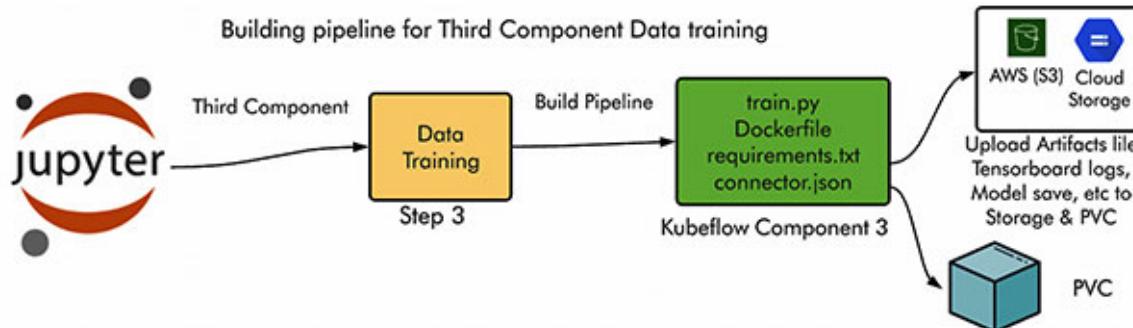


Figure 2.4: Pipeline Component 3: Training

Similarly, like the preceding step, we will build the evaluator component for the Kubeflow pipeline and then we will serve the model by loading the model from any storage like S3, Cloud Bucket, and so on.

Serving: Next, we will keep our pre-process logic inside the serving Python code to the incoming data which will be transformed automatically prior to prediction. Then, we will load our trained model which is saved in S3, Bucket, and so on for prediction. Simultaneously, if we have any other pre-process model, let's say for example, the label-encoder in Scikit-learn, we can load that

when we saved the artifacts in our data pre-processing step 2, so that it can transform the incoming data, and predict with our loaded model.

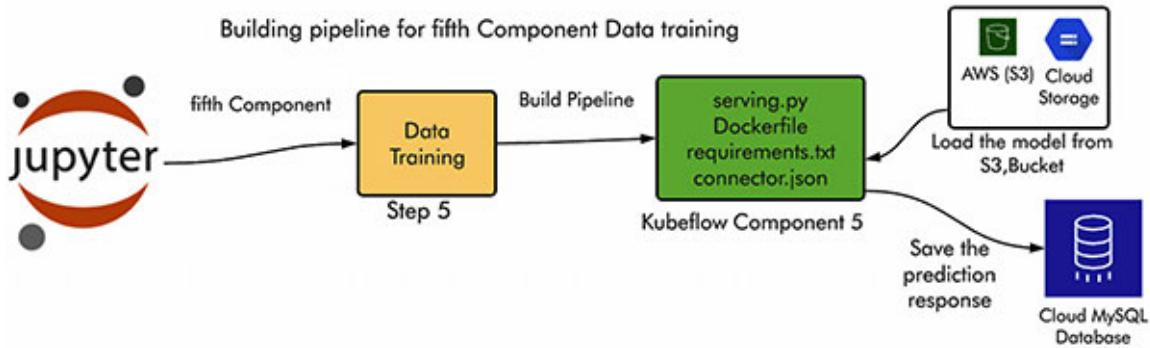


Figure 2.5: Pipeline Component 4:serving

Next, we will start to build a TensorFlow Model pipeline from scratch.

2.4 Building the Kubeflow Pipeline components for TensorFlow model

In this section, we will not show how we will build the training classifier with TensorFlow; the main focus here is how we build the pipeline in Kubeflow to structure that into the pipeline end-to-end. You can use Microsoft Visual Studio and install the Docker and configure in Visual Studio.

So, the folder structure for our pipeline would be like the following format:

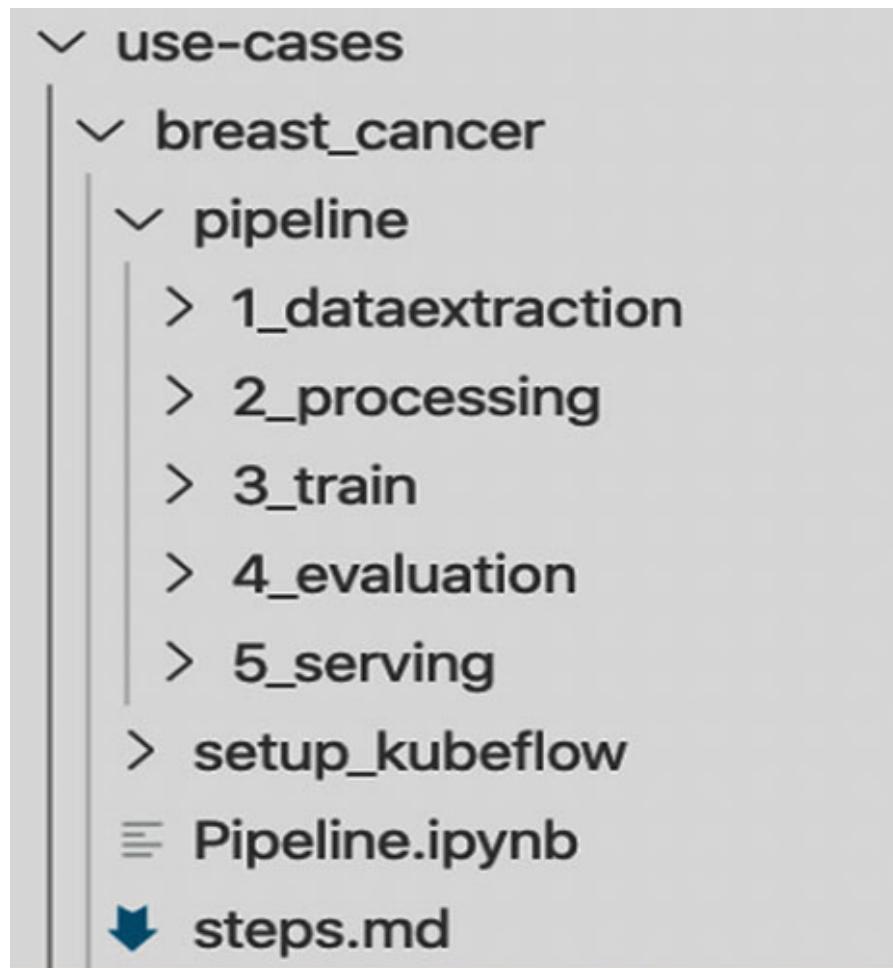


Figure 2.6: Pipeline folder Structure

Below we going to see the Breakdown Architecture for our Pipeline and what each component compromises to build a pipeline end to end.

Here, we have KF-serving which is deployed in Kubernetes and running, and we can hit the endpoint for prediction.

Similar to this step, all of the other steps can be found in the pipeline/folder, and all have the following structure:

- **Pipeline.py** which exposes the functionality through a CLI.
- **requirements.txt** which states the Python dependencies to run.
- Dockerfile which uses the requirements to build the image with one line.

2.3.1 Data Extraction or Ingestion Component

Let's see how we will build the component Data extraction Python file, which is given as follows:

dataextraction.py:

```
from __future__ import absolute_import, division, print_function,
unicode_literals
import click, json, os, dill, argparse
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer

@click.command()
@click.option('--data-file', default="/mnt/breast.data")
def get_data(data_file):
    cancer = load_breast_cancer()
    df_cancer = pd.DataFrame(np.c_[cancer['data'], cancer['target']],
    columns = np.append(cancer['feature_names'], ['target']))
    print(df_cancer.head(3))
    print(df_cancer.describe())
    with open(data_file, "wb") as f:
        dill.dump(df_cancer, f)
    return

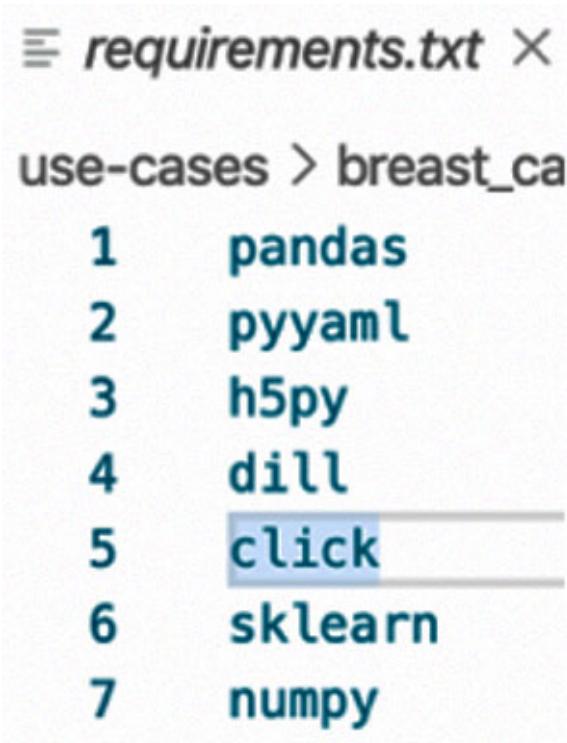
if __name__ == "__main__":
    get_data()
```

Let's breakdown the code as follows:

- First we will load the data from Sklearn datasets. Then, import all the required datasets.
- Then the `@click` command will be passing the argument in the function `get_data()`.
- Then the `dill` command will be helping to save the data in pvc.
- So, as we can see “data-file” is “/mnt”, so mnt is the mount point; we will use mnt every time; after that we can give a file name with `.data` extension; it will save the data in any format like csv, txt, numpy, dataframe, and so on.

- So, whatever data we save with `dill.dump`, it will be used for the next pipeline data pre-processing, where with `@click`, we will input the saved data.

Next, let's see how we will build the component Dockerfile for the data extraction. The following screenshot shows the requirements file where each library will be used inside the `pipeline.py` and we will need to load the `requirements.txt` file as an environment to run the Python function inside Docker:



```

requirements.txt ×

use-cases > breast_ca

1  pandas
2  pyyaml
3  h5py
4  dill
5  click
6  sklearn
7  numpy

```

Figure 2.7: Pipeline Requirements.txt

Docker:

```

FROM python:3.7-slim-stretch
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update&& \
apt-get-y install gcc mono-mcs g++ git curl && \
rm -rf /var/lib/apt/lists/*
RUN mkdir/app
WORKDIR /app

```

```

ADD requirements.txt /app/requirements.txt
RUN pip3 install -r requirements.txt
# copy python
ADD dataextract.py. /app/dataextract.py
RUN chmod +x /app/dataextract.py
ENTRYPOINT ["python"]
CMD["/app/dataextract.py"]

```

So let's break the Docker file code as follows:

- Loading the Python image as base Image to run the pipeline code of 3.7 version.
- **RUN** function will run whenever the Docker image will start and create a /app folder first with **mkdir** command.
- Next, we will redirect our command to **WORKDIR** where all the files and dependencies will be installed.
- Next, we will **ADD** the requirements file in the /app folder, and then run the **pip** command to install the Python libraries.
- Next, we will copy our **pipeline.py** Python code and give an administrative access to run the file whenever the Docker image will run with **chmod** command.
- Next, we will set our entry point as Python and CMD as “/app/dataextract.py” to run this first.

Now, we will build the Docker image; for that, make sure you start the Docker, after which you will see the Docker is activated on the top. As we can see in the following screenshot, the first icon is the Docker, which is started.



Figure 2.8: Docker Activation

Next, we can redirect to our visual studio code, and open the terminal from the visual studio by right clicking on the Docker file inside the **dataextraction** folder:

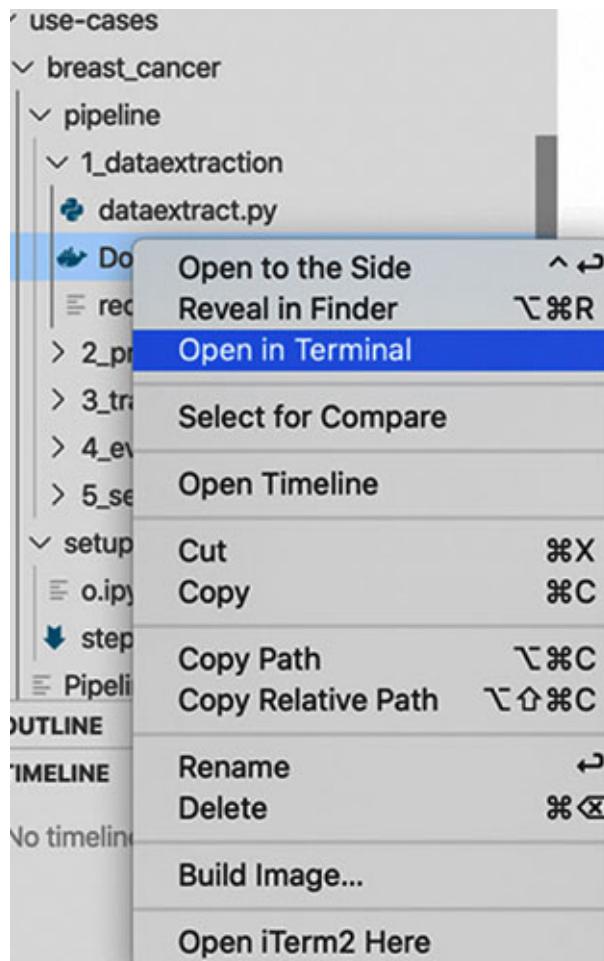


Figure 2.9: Open terminal from Visual Studio

Run all the following commands by connecting to GCP via Local:

```
```bash
gcloud init
#Select the Email/Project associated with GCP
```

```

Build the container for Data Extraction:

```
```bash
cd /pipeline/dataextraction
PROJECT_ID=$(gcloud config get-value core/project)
IMAGE_NAME=breast_cancer/step1_loadingdata
IMAGE_VERSION=v1
IMAGE_NAME=gcr.io/$PROJECT_ID/$IMAGE_NAME
```

```

Building Docker image:

```
Docker build -t $IMAGE_NAME:$IMAGE_VERSION .
```

Push training image to **Google container registry (GCR)**:

```
Docker push $IMAGE_NAME:$IMAGE_VERSION
```

The following is the screenshot where your image will be stored:

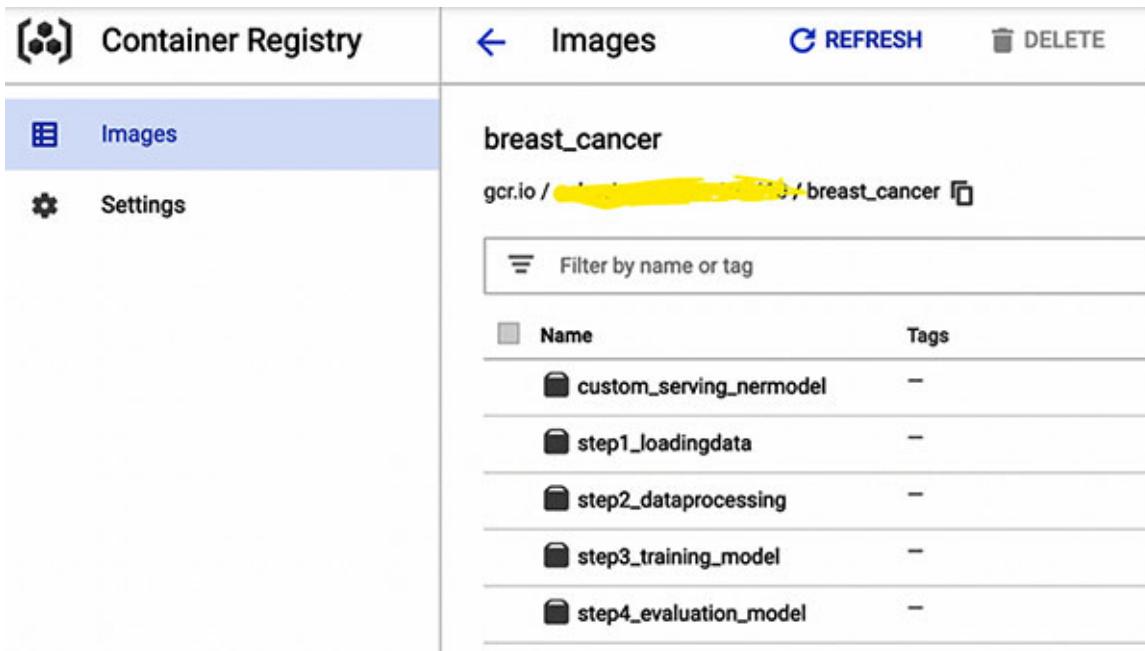


Figure 2.10: Docker Container Registry in GCR

Next, we will see how to build the training component, and as shown earlier, we will build pre-processing.

2.3.2 Data pre-processing component

Now, we will build the pre-processing step, and assume that the input will come from the previous data ingestion component. So, whatever data we have

dumped with `@dill` command, we will load it first and will do the necessary pre-processing step.

In this part of the code, let's import all the dependency, and create the utility correlation plot function. `preprocessing.py`:

```
from __future__ import absolute_import, division, print_function,
unicode_literals
import os, argparse, json, click, dill
from tensorflow.python.lib.io import file_io
import pandas as pd
from sklearn.model_selection import train_test_split
from google.cloud import storage
from plotly.subplots import make_subplots
import plotly.graph_objects as go
import numpy as np
os.environ['GOOGLE_APPLICATION_CREDENTIALS'] =
"service_account_iam.json"

def correlation_plotting(data,s,correlation_plot):
    correlation = data.corr()
    matrix_cols = correlation.columns.tolist()
    corr_array = np.array(correlation)
    trace = go.Heatmap(z = corr_array, x = matrix_cols, y =
matrix_cols, xgap = 2, ygap = 2, colorscale='Viridis', colorbar
= dict())
    layout = go.Layout(dict(title = 'Correlation Matrix' +s,
autosize = False, height = 720, width = 800, margin = dict(r =
0, l = 210, t = 25,b = 210,), yaxis = dict(tickfont = dict(size
= 9)), xaxis = dict(tickfont = dict(size = 9),)))
    fig = go.Figure(data = [trace],layout = layout)
    fig.update_layout(title={'y':1, 'x':0.6, 'xanchor': 'center',
'yanchor': 'top'})
    fig.write_image(correlation_plot)

@click.command()
@click.option('--data-file', default="/mnt/breast.data")
@click.option('--train-file', default="/mnt/training.data")
```

```

@click.option('--test-file', default="/mnt/test.data")
@click.option('--validation-file', default="/mnt/validation.data")
@click.option('--train-target',
default="/mnt/trainingtarget.data")
@click.option('--test-target', default="/mnt/testtarget.data")
@click.option('--validation-target',
default="/mnt/validationtarget.data")
@click.option('--split-size', default=0.1)
@click.option('--bucket-data',
default="gs://kubeflowusecases/breast/data.csv")
@click.option('--bucket-name', default="gs://kubeflowusecases")
@click.option('--commit-sha', default="breast/visualize")
@click.option('--metrics-plot', default="/mnt/correlation.png")
def
training_data_processing(data_file,train_file,test_file,metrics_pl
ot,validation_file,split_size,train_target,test_target,validation_
target,bucket_data,bucket_name,commit_sha):
    with open(data_file, 'rb') as in_f:
        data= dill.load(in_f)

    data=data.fillna(data.mean())
    correlation_plotting(data,"for the Breast Cancer
    Dataset",metrics_plot)

```

In continuation to the previous page of pre-processing code, here we will upload the correlation image which we saved with plotly for Kubeflow Python Visualization, in the following section. Then, we will dump the metadata for the image and table in the run time pipeline to visualize that.

```

image_path = os.path.join(bucket_name, commit_sha,
'correlation.png')
image_url = os.path.join('https://storage.cloud.google.com',
bucket_name.lstrip('gs://'), commit_sha, 'correlation.png?
authuser=0')
html_path = os.path.join(bucket_name,
commit_sha,'correlation.html')
data.to_csv(bucket_data)

```

```

header = data.columns.tolist()
file_io.copy(metrics_plot, image_path)
rendered_template = """
<html>
  <head>
    <title>Correlation</title>
  </head>
  <body>
    <img src={}>
  </body>
</html>""".format(image_url)
file_io.write_string_to_file(html_path, rendered_template)

metadata = {'outputs' : [ {'type': 'table', 'storage': 'gcs',
  'format': 'csv', 'header': header, 'source': bucket_data},
  {'type': 'web-app', 'storage': 'gcs', 'source': html_path, }]}
with open('/mlpipeline-ui-metadata.json', 'w') as f:
  json.dump(metadata, f)
target_name = 'target'
data_target = data[target_name]
data = data.drop([target_name], axis=1)

# %% split training set to validation set
train, test, target, target_test = train_test_split(data,
  data_target, test_size=split_size, random_state=0)
Xtrain, Xval, Ztrain, Zval = train_test_split(train, target,
  test_size=split_size, random_state=0)
with open(train_file,"wb") as f:
  dill.dump(Xtrain,f)
with open(test_file,"wb") as f:
  dill.dump(test,f)
with open(validation_file,"wb") as f:
  dill.dump(Xval,f)
with open(train_target,"wb") as f:
  dill.dump(Ztrain,f)

with open(test_target,"wb") as f:

```

```

dill.dump(target_test,f)
with open(validation_target,"wb") as f:
    dill.dump(Zval,f)
return

```

Let's breakdown the code for pre-processing as follows:

- First, we will import all the libraries, and create the utility `plotly correlation` function, which will save an image of the plot. Then we will save that in the GCS bucket, so that we can visualize that in static HTML format in the Kubeflow Dashboard.
- Then we will add the service account `.json` file in the environment, so that it can give access to the push data or artifacts in the GCP Bucket.
- Next, the `@click` option will input all the required dumps which we have done in the data ingestion step, so that we can use it here, after which we will load those first and use those for the pre-processing steps for training the data.
- Then, we will authorise and push the artifacts image in the GCS bucket by TensorFlow `file.io` function, and after that, we will create a static HTML to publish in the Kubeflow dashboard. It is pulling the image from storage bucket location, which we saved earlier in following path `gs://Kubeflowusecase/breast/visualize/correlation.html` and dumped the metadata as json format. We have split the data and dumped the data for the next training step.

Before creating the Docker file, make sure we have the service `service_account_iam.json` file which will be used for the cloud storage bucket access.

Please click on the following link to download the json key, and paste it in the `preprocessing` folder:

<https://cloud.google.com/iam/docs/creating-managing-service-account-keys>

The JSON File will look like the following:

```
{"type": "service_account",
"project_id": "project-id",
```

```

"private_key_id": "key-id",
"private_key": "-----BEGIN PRIVATE KEY-----\nprivate-key\n-----END
PRIVATE KEY-----\n",
"client_email": "service-account-email",
"client_id": "client-id",
"auth_uri": "https://accounts.google.com/o/oauth2/auth",
"token_uri": "https://accounts.google.com/o/oauth2/token",
"auth_provider_x509_cert_url":
"https://www.googleapis.com/oauth2/v1/certs",
"client_x509_cert_url":
"https://www.googleapis.com/robot/v1/metadata/x509/service-
account-email"
}

```

While creating the Service account, search **Cloud Storage**, and then select **Storage Admin**.

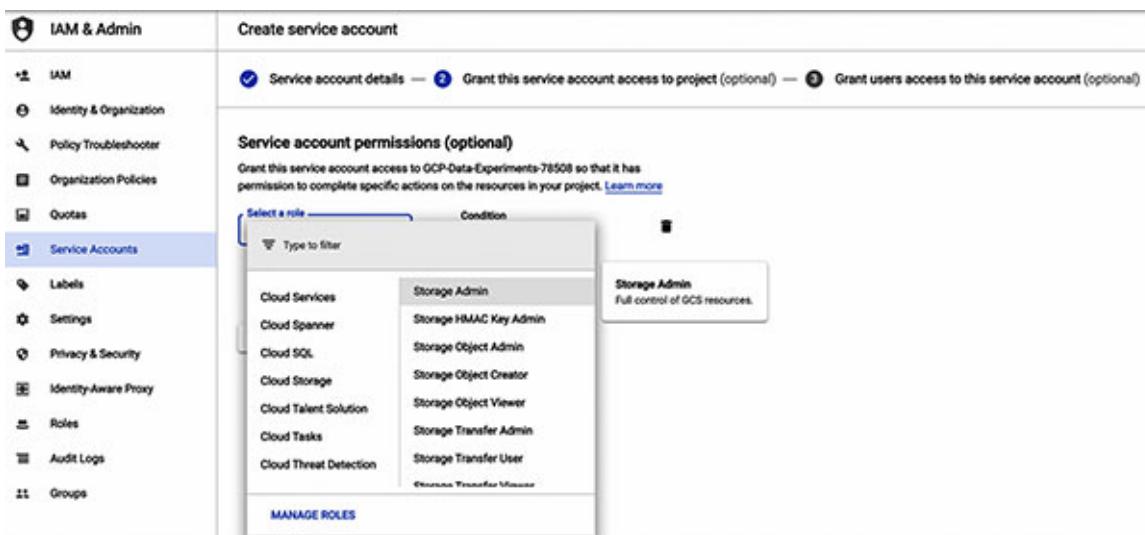


Figure 2.11: Service account cloud storage selection

The preceding screenshot shows the role we will give for the service account. Similarly, we will create the Docker Image, and add the service account json key.

Docker:

```

FROM python:3.7-slim-stretch
ENV DEBIAN_FRONTEND noninteractive

```

```

RUN apt-get update && apt-get -y install gcc mono-mcs g++ git curl
&& \
rm -rf /var/lib/apt/lists/*
RUN mkdir /app
WORKDIR /app
COPY service_account_iam.json
ENV GOOGLE_APPLICATION_CREDENTIALS="service_account_iam.json"
ADD requirements.txt /app/requirements.txt
RUN pip3 install -r requirements.txt
ADD dataextract.py. /app/preprocessing.py
RUN chmod +x /app/preprocessing.py
ENTRYPOINT ["python"]
CMD["/app/processing.py"]

```

Similar to the preceding step, in the data ingestion step, we will build the Docker image.

Now, from the Dockerfile location, we will build the Docker image, and push that in the GCS Container registry. You can get the entire folder in the GCP Bucket, and all you need is to rename the Docker image name, and run those Docker build and push command.

2.3.3 Training model component

In this section, we will build a TensorFlow model, and then we will build a distributed model strategy, save the model in the GCP bucket, and push the Tensorboard logs in the Bucket for the Kubeflow Dashboard Tensorboard Visualization.

Let's see how we will build the component **pipeline.py** file.

train.py:

```

from __future__ import absolute_import, division, print_function,
unicode_literals
import click, dill, json, logging, os
import pandas as pd
import tensorflow as tf
from storage import Storage

```

```

def model_build(Xtrain):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Dense(units=32,
            kernel_initializer='glorot_uniform', activation='relu',
            input_shape=(len(Xtrain.columns),)),
        tf.keras.layers.Dense(units=64,
            kernel_initializer='glorot_uniform', activation='relu'),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(units=64,
            kernel_initializer='glorot_uniform', activation='relu'),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(units=1,
            kernel_initializer='glorot_uniform', activation='sigmoid'))])
    return model

def get_callbacks(path):
    checkpointdir = path
    class customLog(tf.keras.callbacks.Callback):
        def on_epoch_end(self, epoch, logs={}):
            logging.info('epoch: {}'.format(epoch + 1))
            logging.info('loss={}'.format(logs['loss']))
            logging.info('accuracy={}'.format(logs['accuracy']))
            logging.info('val_loss={}'.format(logs['val_loss']))
            logging.info('val_accuracy={}'.format(logs['val_accuracy']))
    callbacks =
        [tf.keras.callbacks.ModelCheckpoint(filepath=checkpointdir), custom
        Log()]
    return callbacks

@click.command()
@click.option('--epochs', default=100)
@click.option('--batch-size', default=4)
@click.option('--learning-rate', default=0.001)
@click.option('--tensorboard-logs', default='/mnt/logs/')
@click.option('--tensorboard-gcs-logs',
    default='gs://kubeflowusecases/breast/logs')

```

```

@click.option('--model-output-base-path',
default="/mnt/saved_model")
@click.option('--gcs-path',
default="gs://kubeflowusecases/breast/model")
@click.option('--mode', default="local")
def
train_model(train_file,test_file,validation_file,train_target,test
_target,validation_target,
batch_size,learning_rate,tensorboard_logs,tensorboard_gcs_logs,mod
el_output_base_path,gcs_path,mode,epochs):
with open(train_file, 'rb') asin_f:
    train= dill.load(in_f)

```

So, this a continuation of the **train.py**, as we declared the utility function for the callback and model building function, alongside the **@click** to input the previous pipeline outputs, which will be used as an input here, alongside the new input for these pipeline parameters like epoch, learning rate, and so on.

```

with open(test_file, 'rb') asin_f:
    test= dill.load(in_f)
with open(validation_file, 'rb') asin_f:
    validation= dill.load(in_f)
with open(train_target, 'rb') asin_f:
    train_tar= dill.load(in_f)
with open(test_target, 'rb') asin_f:
    test_tar= dill.load(in_f)
with open(validation_target, 'rb') asin_f:
    validation_tar= dill.load(in_f)

    strategy =
    tf.distribute.experimental.MultiWorkerMirroredStrategy()
    logging.info("Number of devices:
    {0}".format(strategy.num_replicas_in_sync))
with strategy.scope():
    optimizer = tf.keras.optimizers.Adam(learning_rate)
    model = model_build(train)

```

```

model.compile(optimizer=optimizer,
loss=tf.keras.losses.binary_crossentropy,metrics=['accuracy'])
    TF_STEPS_PER_EPOCHS=5
    BATCH_SIZE = batch_size * strategy.num_replicas_in_sync
tensorboard_callback =
tf.keras.callbacks.TensorBoard(log_dir=tensorboard_logs,
histogram_freq=1)
logging.info("Training starting...")
model.fit(train,train_tar, epochs=epochs,
batch_size=BATCH_SIZE,
validation_data=(validation, validation_tar), callbacks=
[tensorboard_callback])
    logging.info("Training completed.")
model.save(model_output_base_path)
new_model =
    tf.keras.models.load_model(model_output_base_path)
# Check its architecture
print(new_model.summary())
Storage.upload(tensorboard_logs,tensorboard_gcs_logs)
metadata = {
'outputs': [{ 'type': 'tensorboard', 'source':
tensorboard_gcs_logs,}]}
with open("/mlpipeline-ui-metadata.json", 'w') as f:
json.dump(metadata,f)
if mode!= 'local':
print("uploading to {}".format(gcs_path))
Storage.upload(model_output_base_path,gcs_path)
else:
print("Model will not be uploaded")
pass

if __name__ == "__main__":
train_model()

```

So, the preceding is the **train.py** file; let's break the pipeline code as follows:

- First we import all the required datasets.

- Then the `@click` command will be passing the argument in the function `train_model()`.
- Then the `dill` command will help save or load the data from pvc.
- Next, we will pass all the arguments like epochs and batch size, learning rate from the outside and train our model.
- We will push the model train output to we will push Google Storage Bucket. We have imported the `storage.py` file to the Storage class to save our model artifacts.
- Next, we will visualize the Tensorboard and follow the same metadata json format with “`/mpipeline-ui-metadata.json`” and give the GCS bucket path, where you save your model artifacts.

For Python visualization, you can visit the following link:

<https://www.Kubeflow.org/docs/pipelines/sdk/output-viewer/>

Docker:

Before creating the Docker file, make sure you copy the service `service_account_iam.json` file which we created earlier and which will be used for we will push cloud storage bucket access, and the `storage.py` Python file will be there for uploading the data to GCS bucket.

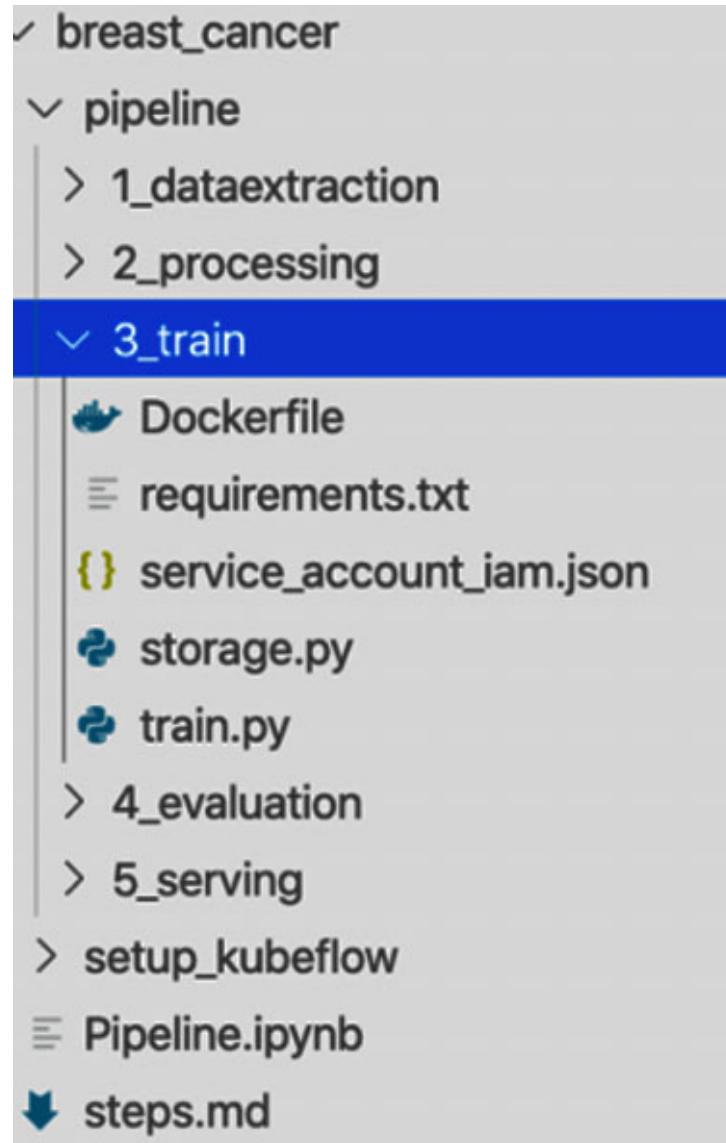


Figure 2.12: Training Folder structure

Similarly, we will create the Docker Image and add the service account json key.

Docker:

```
FROM python:3.7-slim-stretch
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update && \apt-get -y install gcc mono-mcs g++ git curl
&& \
rm -rf /var/lib/apt/lists/*
RUN mkdir/app
```

```
WORKDIR /app
ADD requirements.txt /app/requirements.txt
RUN pip3 install -r requirements.txt
COPY service_account_iam.json service_account_iam.json
ENV GOOGLE_APPLICATION_CREDENTIALS="service_account_iam.json"
ADD train.py /app/train.py
ADD storage.py /app/storage.py
RUN chmod +x /app/train.py
ENTRYPOINT["python"]
CMD["/app/train.py"]
```

Next we will create the Docker image and will push the image similarly what we did above on Data Extraction.

[2.3.4 Evaluation component](#)

Similarly, we will create the evaluation component. Please have a look at the GitHub **steps.md** file.

The following are a few important highlights on what we did in the evaluation component:

We have dumped the confusion Matrix and ROC Curve as a csv in the GCP bucket and gave the path a storage location of those in the metadata, and dumped that as json, so that the Kubeflow pipeline will take that and visualize the ROC & Confusion Matrix.

[2.5 Serving the Model with KF Serving](#)

In this section, we will build our serving for our model. In the following screenshot, we can see the three major components for our model serving Dockerfile is the model folder, Dockerfile, **servebreast.py**.

So, here we kept the saved trained model from the local to the root of the Docker file or the Cloud Storage bucket.

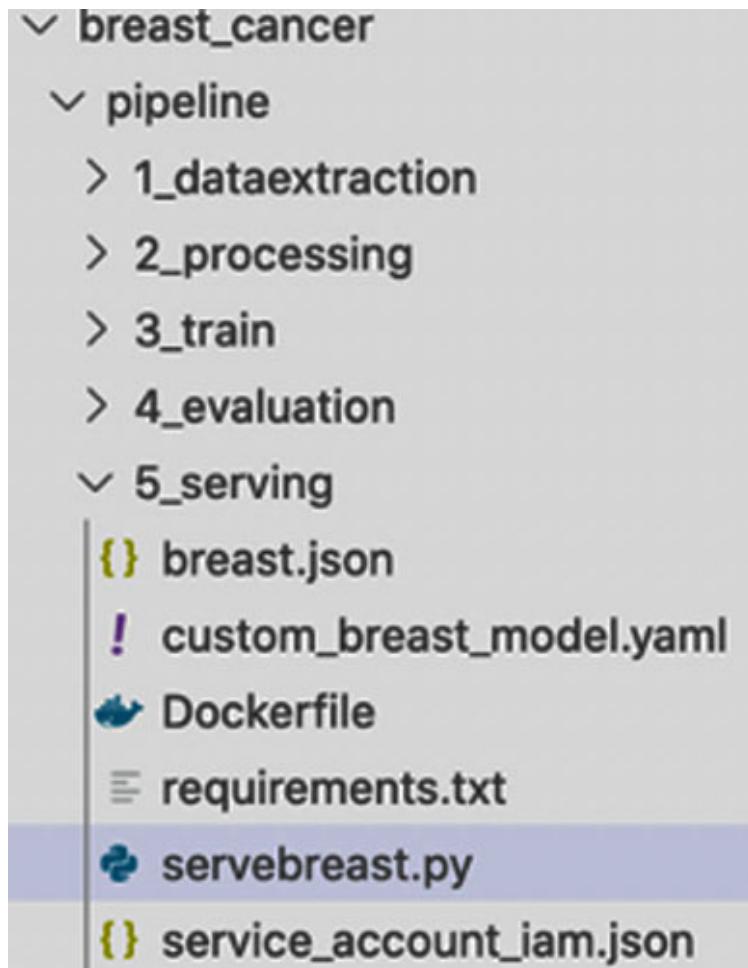


Figure 2.13: KF – Serving Component Folder Structure

Architecture of KF-Serving:

The InferenceService Data Plane architecture consists of a static graph of components which coordinates the requests for a single model.

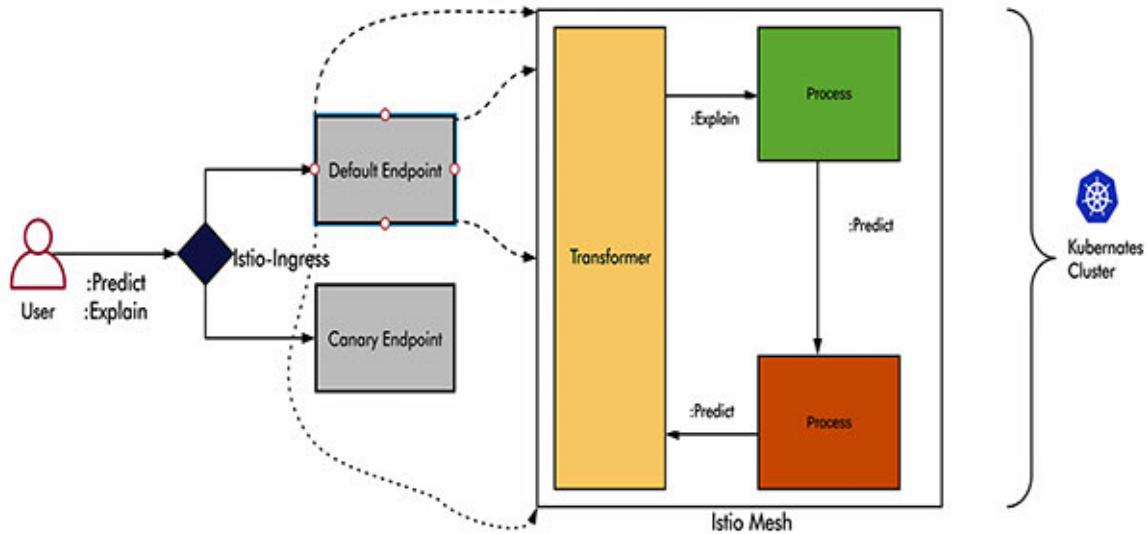


Figure 2.14: KF – Serving Architecture

Endpoint: InferenceServers are divided into two endpoints – “default” and “canary”. The endpoints allow the users to safely make the changes using the Pinned and Canary rollout strategies. Canarying is completely optional, enabling the users to simply deploy with a BlueGreen deployment strategy on the “default” endpoint.

Component: Each endpoint is composed of multiple components – “predictor”, “explainer”, and “transformer”. The only required component is the predictor, which is the core of the system. As KFServing evolves, we plan to increase the number of supported components to enable the use cases like Outlier Detection.

Predictor: The predictor component is the workhorse of the InferenceService. It is simply a model and a model server that makes it available at a network endpoint.

Explainer: It enables an optional alternate data plane that provides our model explanations, in addition to the predictions. So the users may also define their own explanation container, which KFServing will configure with the relevant environment variables like prediction endpoint. For the common scenario, KFServing provides out-of-the-box explainers like Alibi.

Transformer: The transformer enables the users to define a pre and post processing step before the prediction and explanation workflows. Like the explainer, it also configures with the relevant environment variables. For the

common use cases, KFServing provides the out-of-the-box transformers like Feast.

Here, we will use the custom image serving because of the following reason:

The goal of the custom image support is to allow the users to bring their own wrapped model inside a container and serve it with KFServing. Here, in this example, located in the model-server directory extends the kfserving. KFModel uses the tornado web server.

You can use KFServing to do the following:

- Provide a Kubernetes Custom Resource Definition for serving the ML models on the arbitrary frameworks.
- Encapsulate the complexity of autoscaling, networking, health checking, and server configuration to bring cutting edge serving features like GPU autoscaling, scale to zero, and canary rollouts to your ML deployments.
- Enable a simple, pluggable, and complete story for your production ML inference server by providing prediction, pre-processing, post-processing and explainability out of the box.

Let's see how we will build the serving component **Transformer.py** file.

Transformer.py:

```
import tensorflow as tf
import sys,os,json,kfserving
import numpyas np
import kfserving
from typing import List, Dict

os.environ['GOOGLE_APPLICATION_CREDENTIALS'] =
"service_account_iam.json"

class KFServingSampleModel(kfserving.KFModel):

    def __init__(self, name: str):
        super().__init__(name)
        self.name = name
        self.ready = False
    self.model_output_base_path =
"gs://kubeflowusecases/breast/model/"
```

```

def load(self):
    model =
        tf.keras.models.load_model(self.model_output_base_path)
    self.model = model
    self.ready = True

def predict(self, request: Dict) ->Dict:
    inputs = np.array(request["instances"])
    reshaped_to_2d = np.reshape(inputs, (-1, len(inputs)))
    results = self.model.predict(reshaped_to_2d)
    result = (results >0.5)*1
    if result==1:
        result="malignant"
    else:
        result="Benign"
    print("result : {0}".format(result))
    return {"predictions": result}

if __name__ == "__main__":
    model = KFServingSampleModel("kfserving-breast-model")
    model.load()
    kfserving.KFServer(workers=1).start([model])

```

So let's break the transformer predictor code as follows:

- We kept the service account which has the storage bucket in the Docker root and then declared that folder as an environment variable to an object **model_output_base_path** for gcp bucket path.
- Next, in the Load function, we loaded the model in from the TensorFlow library.
- Then, in the predict method, the incoming data will come as a json format which we need to extract as a key-value pair and do the necessary prediction and return as a dictionary.
- So, in the “main” function, the **KFServingSampleModel** class takes the name of that deployment; keep a note of that and apply to the yaml file; here it is ”kfserving-breast-model”

Next, we will build the Docker image for our serving model.

Docker:

```
FROM python:3.7-slim-stretch
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update && \
apt-get-y install gcc mono-mcs g++ git curl bash && \
rm-rf /var/lib/apt/lists/*
RUN mkdir /app
WORKDIR /app
ADD requirements.txt /app/requirements.txt
RUN pip install -r /app/requirements.txt
ADD servebreast.py /app/servebreast.py
COPY service.json
ENV GOOGLE_APPLICATION_CREDENTIALS="service.json"
CMD ["python", "servebreast.py"]
```

Now, in the preceding Docker code, we have copied the service account for the storage bucket access and saved it in `/app`. Then, we copied the Python serving file in the same location and kept the working directory as `/app`. Then, we built the image using the following code. Similarly, we created the Docker image; please have a look at the GitHub `steps.md` file.

To deploy the model server using the `kubectl` command line, or using the KFServing client SDK, complete the following steps:

- Deploy using the command line.
- Deploy using the KFServing client SDK.

Deploy using the command line:

Now, let's deploy it with the command line, but first let's fill the yaml file.

`Custom_KFServing.yaml`:

```
apiVersion: serving.kubeflow.org/v1alpha2
kind: InferenceService
metadata:
  annotations:
    sidecar.istio.io/inject: "false"
  name: kfserving-breast-model
```

```

namespace: kubeflow
spec:
  default:
    predictor:
      custom:
        container:
          image: gcr.io/<PROJECT_ID>/breast_cancer/custom_serving:v1

```

Here, in the preceding yaml file, we will give the same name which we have provided in the Transformer.py file, having the model name (“*Kfserving-breast-model*”), and then we will provide the namespace “Kubeflow”, where it will be deployed. Next, we will give the Docker image a name for our Transformer model, which we have created.

```

servebreast.py X
use-cases > BPP > breast_cancer > pipeline > 5_serving > servebreast.py > ...
6 import kfserving
7 from typing import List, Dict
8
9 os.environ['GOOGLE_APPLICATION_CREDENTIALS'] = "service_account_iam.json"
10
11
12 class KFServingSampleModel(kfserving.KFModel):
13     def __init__(self, name: str):
14         super().__init__(name)
15         self.name = name
16         self.ready = False
17         self.model_output_base_path = "gs://kubeflowusecases/breast/model/"
18
19     def load(self):
20         model = tf.keras.models.load_model(self.model_output_base_path)
21         self.model = model
22         self.ready = True
23
24     def predict(self, request: Dict) -> Dict:
25         inputs = np.array(request["instances"])
26         reshaped_to_2d = np.reshape(inputs, (-1, len(inputs)))
27         results = self.model.predict(reshaped_to_2d)
28         result = (results > 0.5)*1
29         if result==1:
30             result="malignant"
31         else:
32             result="Benign"
33
34         print("result : {}".format(result))
35         return {"predictions": result}
36
37
38 if __name__ == "__main__":
39     model = KFServingSampleModel("kfserving-breast-model")
40     model.load()
41     kfserver.KFServer(workers=1).start([model])
42

```

```

custom_breast_model.yaml X
apiVersion: serving.kubeflow.org/v1alpha2
kind: InferenceService
metadata:
  annotations:
    sidecar.istio.io/inject: "false"
  name: kfserving-breast-model
  namespace: kubeflow
spec:
  default:
    predictor:
      custom:
        container:
          image: gcr.io/<PROJECT_ID>/breast_cancer/custom_serving_breast_ci
          imagePullPolicy: Always
          name: user-container
          imagePullSecrets:
            - name: user-gcp-sa

```

Figure 2.15: KF – Serving Model name match

As we can see in the preceding screenshot, line number 38 from the left image and 6 from the right should be always the same.

Next, run the following command from bash where the files are kept in Visual Studio.

- Connect to the GCP cluster using the following command:

```
gcloud container clusters get-credentials <$ClusterName> --  
zone  
<$ZONE> --project <$PROJECTID>
```

- Create the inference service by deploying it in the cluster:

```
kubectl apply -f custom_breast_model.yaml
```

- Check the inference service. Try it after some interval to check if it has been created:

```
kubectl get inferenceservice -n Kubeflow
```

| NAME | URL | READY | DEFAULT TRAFFIC | CANARY TRAFFIC | AGE |
|------------------------|---|-------|-----------------|----------------|-------|
| kfserving-breast-model | http://kfserving-breast-model.kubeflow.example.com/v1/models/kfserving-breast-model | True | 100 | | 2d18h |

Figure 2.16: KF – Serving Inference Output

Sample Prediction:

- Run the following command in Bash from the serving folder:

```
```bash  
MODEL_NAME=kfserving-braintumor
HOST=$(kubectl get inferenceservice -n Kubeflow$MODEL_NAME -o
jsonpath='{.status.url}' | cut -d "/" -f 3)
INPUT_PATH=@./data.json
CLUSTER_IP=$(kubectl -n istio-system get service kfserving-
ingressgateway -o
jsonpath='{.status.loadBalancer.ingress[0].ip}')
curl -v -H "Host: ${HOST}"
http://${CLUSTER_IP}/v1/models/${MODEL_NAME}:predict -d
$INPUT_PATH
```
```

Now, the following is the Response prediction request:

```

* Trying [REDACTED]...
* TCP_NODELAY set
* Connected to [REDACTED] ([REDACTED]) port 80 (#0)
> POST /v1/models/kfserving-breast-model:predict HTTP/1.1
> Host: kfserving-breast-model.kubeflow.example.com
> User-Agent: curl/7.64.1
> Accept: */*
> Content-Length: 256
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 256 out of 256 bytes
< HTTP/1.1 200 OK
< content-length: 28
< content-type: application/json; charset=UTF-8
< date: Fri, 24 Jul 2020 05:40:42 GMT
< server: istio-envoy
< x-envoy-upstream-service-time: 11796
<
* Connection #0 to host [REDACTED] left intact
{"predictions": "malignant"}* Closing connection 0

```

Figure 2.17: KF – Serving Prediction Output

- Run the following command in Python from the serving folder:

Now, we will create some sample data to predict the results from the preceding URL. The following is the code to create the sample data:

```

import base64, json, requests
from sklearn.datasets import load_breast_cancer
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
cancer = load_breast_cancer()

df_cancer = pd.DataFrame(np.c_[cancer['data'],
cancer['target']], columns =
np.append(cancer['feature_names'], ['target']))
X = df_cancer.drop(['target'],axis=1)
y = df_cancer['target']
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size = 0.1, random_state = 0)

```

```

data = json.dumps({"instances": X_test.iloc[7].to_list()})
data

```

Figure 2.18: Sample Data

```

MODEL_NAME="kfserving-breast-model"
cluster_ip = <COPY YOUR CLUSTER IP HERE>
headers={"Host": "
{0}.Kubeflow.example.com".format(MODEL_NAME)}
response =
requests.post("http://{0}/v1/models/{1}:predict".format(cluster_ip, MODEL_NAME), data = data, headers = headers)
response.json()

```

```
{'predictions': 'malignant'}
```

Figure 2.19: Prediction output

As we can see in the preceding screenshot, our prediction output is malignant.

2.6 Building the pipeline end to end

Now, let's see how we build the Pipeline, and we will run this platform in the Kubeflow Notebook server.

Open the URL: GCP Kubernetes > **Service & ingress** > Click on the URL.

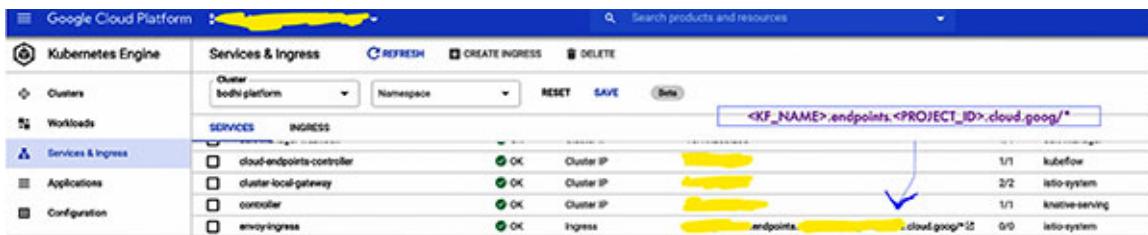


Figure 2.20: Ingress URL

As in [Chapter 1, Introduction to Kubeflow & Kubernetes Cloud Architecture](#) already in section 1.6, we have created a Jupyter notebook that we will use.



Figure 2.21: Notebook Server output

Now, paste the following code and run the pipeline; before that replace the PROJECT_ID and bucket name from the following code, and it will dump a zip file:

```
import kfp.dsl as dsl
import yaml
from kubernetes import client as k8s
import kfp.gcp as gcp
from kfp import components
from string import Template
import json
from kubernetes import client as k8s_client

@dsl.pipeline(
    name='breast cancer pipeline',
    description='End to End pipeline for Tensorflow Breast Cancer')
def breast_cancer_tensorflow_pipeline(
    dataextraction_step_image="gcr.io/<$PROJECT_ID>/breast_cancer/
step1_loadingdata",
    dataprocessing_step_image="gcr.io/<$PROJECT_ID>/breast_cancer/
step2_dataprocessing",
    trainmodel_step_image="gcr.io/<$PROJECT_ID>/breast_cancer/step
3_training_model",
    evaluator_step_image="gcr.io/<$PROJECT_ID>/breast_cancer/step4
_evaluation_model",
    train_file='/mnt/training.data',
    data_file="/mnt/breast.data",
    test_file='/mnt/test.data',
    validation_file="/mnt/validation.data",
    split_size=0.2,
    train_target="/mnt/trainingtarget.data",
```

```
test_target="/mnt/testtarget.data",
validation_target="/mnt/validationtarget.data",
epochs=5,
learning_rate=.001,
batch_size=64,
shuffle_size=1000,
tensorboard_logs="/mnt/logs/",
tensorboard_gcs_logs="gs://<$YOUR_BUCKET>/breast/logs",
model_output_base_path="/mnt/saved_model",
gcs_path="gs://<$YOUR_BUCKET>/breast/model",
gcs_path_confusion="gs://<$YOUR_BUCKET>/breast",
mode="gcs",
probability=0.5,
serving_name="kfserving-breast-model",
serving_namespace="Kubeflow",
image="gcr.io/<$PROJECT_ID>/breast_cancer/custom_serving_nermodel"):

"""
Pipeline
"""

# PVC : PersistentVolumeClaim volume
vop = dsl.VolumeOp(
    name='my-pvc',
    resource_name="my-pvc",
    modes=dsl.VOLUME_MODE_RWO,
    size="1Gi"
)
# data extraction
data_extraction_step = dsl.ContainerOp(
    name='data_extraction',
    image=dataextraction_step_image,
    command="python",
    arguments=[
        "/app/dataextract.py",
        "--data-file", data_file],
```

```
pvolumes={"/mnt": vop.volume})  
  
# processing  
data_processing_step = dsl.ContainerOp(  
    name='data_processing',  
    image=dataprocessing_step_image,  
    command="python",  
    arguments=[  
        "/app/preprocessing.py",  
        "--train-file", train_file,  
        "--test-file", test_file,  
        "--validation-file", validation_file,  
        "--data-file", data_file,  
        "--split-size", split_size,  
        "--train-target", train_target,  
        "--test-target", test_target,  
        "--validation-target", validation_target],  
    pvolumes={"/mnt": data_extraction_step.pvolume})  
#trainmodel  
train_model_step = dsl.ContainerOp(  
    name='train_model',  
    image=trainmodel_step_image,  
    command="python",  
    arguments=[  
        "/app/train.py",  
        "--train-file", train_file,  
        "--test-file", test_file,  
        "--validation-file", validation_file,  
        "--train-target", train_target,  
        "--test-target", test_target,  
        "--validation-target", validation_target,  
        "--epochs", epochs,  
        "--batch-size", batch_size,  
        "--learning-rate", learning_rate,  
        "--tensorboard-logs", tensorboard_logs,  
        "--tensorboard-gcs-logs", tensorboard_gcs_logs,
```



```

    'namespace': str(serving_namespace),
    'image': str(image)})
kfservingdeployment = json.loads(kfservingjson)
serve = dsl.ResourceOp(
    name="serve",
    k8s_resource=kfservingdeployment,
    action="apply",
    success_condition="status.url")
serve.after(evaluation_model_step)

if __name__ == '__main__':
    import kfp.compiler as compiler
    pipeline_func = breast_cancer_tensorflow_pipeline
    pipeline_filename = pipeline_func.__name__ + '.pipeline.zip'
    compiler.Compiler().compile(pipeline_func, pipeline_filename)

```

Now, let's break the pipeline code as follows:

- Pipelines are expected to include a `@dsl.pipeline` decorator to provide the metadata about the pipeline.
- The pipeline is defined in the `breast_cancer_tensorflow_pipeline` function. It includes a number of arguments, which are exposed in the Kubeflow Pipelines UI when creating a new Run. Although passed as strings, these arguments are of type `kfp.dsl.PipelineParam`.
- Each individual block defines one component like ‘train’, ‘evaluation’, and so on. A component is made up of a `kfp.dsl.ContainerOp` object with the container path and a specified name. The container image used is defined as the Docker file which we have created.
- After defining the train component, we also set a number of environment variables for the training script.
- At the bottom of the script is the main function. This is used to compile the pipeline when the script is run; next, the `.after` method will trigger the pipeline one after the other.

When you run the preceding code, it will dump a zip like the following:

```

File Edit View Run Kernel Git Tabs Settings Help
Name Last Modified
breast_cancer_tensorflow_pipeline.pipeline.zip 12 minutes ago
pipeline.ipynb 10 minutes ago

pipeline.ipynb X Code Submit Notebook... Python 3

kfservingjson = kfserving_template.substitute({'name': str(serving_name),
                                              'namespace': str(serving_namespace),
                                              'image': str(image)})

kfservingdeployment = json.loads(kfservingjson)

serve = dsl.ResourceOp(
    name='serve',
    k8s_resource=kfservingdeployment,
    action='apply',
    success_condition='status.url'
)
serve.after(evaluation_model_step)

...
if __name__ == '__main__':
    import kfp.compiler as compiler
    pipeline_func = breast_cancer_tensorflow_pipeline
    pipeline_filename = pipeline_func.__name__ + '.pipeline.zip'
    compiler.Compiler().compile(pipeline_func, pipeline_filename)

[22]: import kfp
from kfp import compiler
import kfp.components as comp
import kfp.dsl as dsl
from kfp import gcp
EXPERIMENT_NAME = 'Breast'
client = kfp.Client()

try:
    experiment = client.get_experiment(experiment_name=EXPERIMENT_NAME)
except:
    experiment = client.create_experiment(EXPERIMENT_NAME)

print(experiment)
{'created_at': datetime.datetime(2028, 10, 20, 17, 40, 56, tzinfo=tzlocal()),
 'description': None,
 'id': '683959317-4c58-4730-bb84-89ec89876dfc',
 'name': 'Breast',
 'resource_references': None,
 'storage_state': None}

```

Figure 2.22: Pipeline zip

Next, we will create an experiment; under that, we can create multiple runs of a pipeline. The following code is for creating the experiment:

```

import kfp
from kfp import compiler
import kfp.components as comp
import kfp.dsl as dsl
from kfp import gcp
EXPERIMENT_NAME = 'Breast'
client = kfp.Client()

try:
    experiment =
        client.get_experiment(experiment_name=EXPERIMENT_NAME)
except:
    experiment = client.create_experiment(EXPERIMENT_NAME)
print(experiment)

```

The following snippet will create a run for the zip that we have dumped in that location:

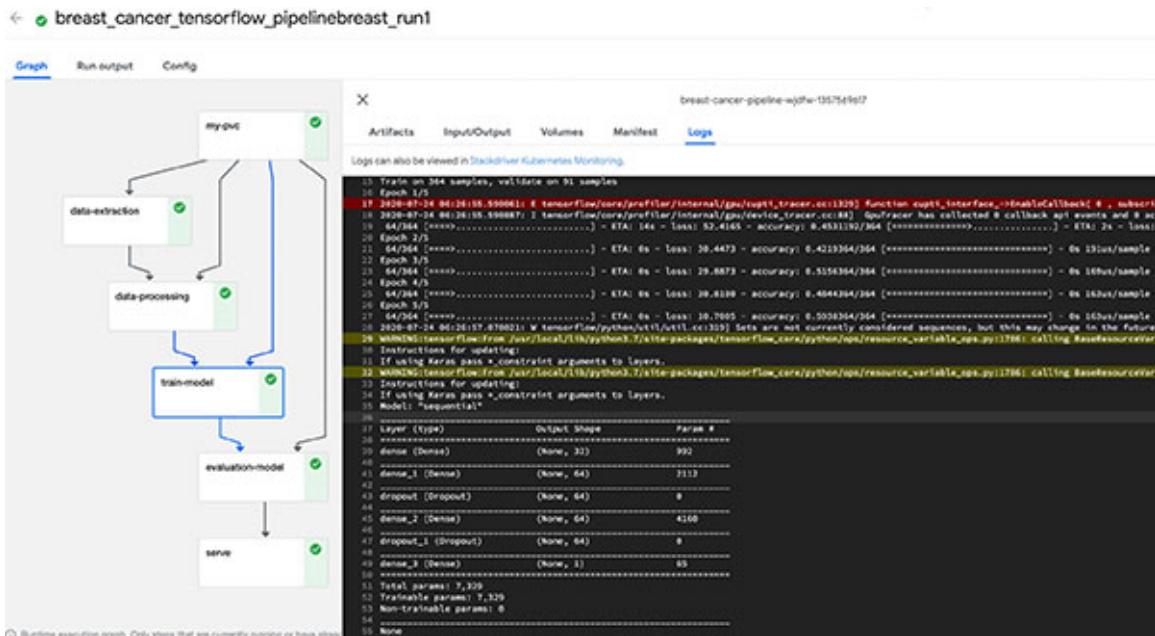
```

arguments = {}
run_name = pipeline_func.__name__ + 'breast_run'
run_result = client.run_pipeline(experiment.id, run_name,
pipeline_filename, arguments)

print(experiment.id)
print(run_name)
print(pipeline_filename)
print(arguments)

```

Click on the run link, once the pipeline is ready. The following is how our pipeline training looks:



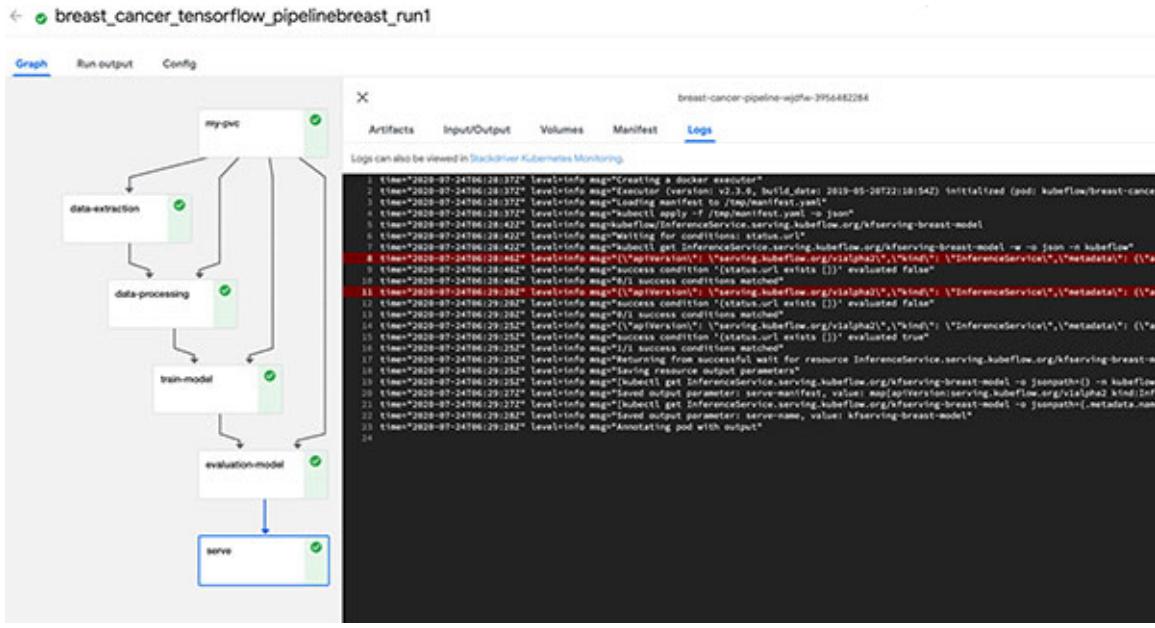


Figure 2.24: Serving pipeline output

The following screenshot is of the pipeline which we have created and the Python visualizations, tables and static HTML plots:



Figure 2.25: Serving pipeline output

Next, let's find out how to Monitor an endpoint in Grafana.

2.7 Monitoring the performance with Grafana dashboard

Run the command in the Google Cloud shell after connecting the Cluster and then Create Namespace.

```
```bash
gcloud container clusters get-credentials <$ClusterName> --zone
<$ZONE> --project <$PROJECTID>
#create namespace
kubectl create namespace knative-monitoring
#setup monitoring components
kubectl apply --filename
https://github.com/knative/serving/releases/download/v0.177.20/mon
```

```
itoring-metrics-prometheus.yaml
```

```
```  
configmap/scaling-config created  
configmap/grafana-dashboard-definition-knative created  
configmap/grafana-datasources created  
configmap/grafana-dashboards created  
service/grafana created  
deployment.apps/grafana created  
configmap/prometheus-scrape-config created  
service/kube-controller-manager created  
service/prometheus-system-discovery created  
serviceaccount/prometheus-system created  
role.rbac.authorization.k8s.io/prometheus-system created  
role.rbac.authorization.k8s.io/prometheus-system created  
role.rbac.authorization.k8s.io/prometheus-system created  
role.rbac.authorization.k8s.io/prometheus-system created  
clusterrole.rbac.authorization.k8s.io/prometheus-system created  
rolebinding.rbac.authorization.k8s.io/prometheus-system created  
rolebinding.rbac.authorization.k8s.io/prometheus-system created  
rolebinding.rbac.authorization.k8s.io/prometheus-system created  
rolebinding.rbac.authorization.k8s.io/prometheus-system created  
clusterrolebinding.rbac.authorization.k8s.io/prometheus-system created  
service/prometheus-system-np created  
statefulset.apps/prometheus-system created
```

Figure 2.26: Prometheus deploye

Next, run from the Local Terminal and open the `grafana` dashboard by using `localhost:8080` on the browser. Explore the different components of the grafana dashboard.

```
```bash  
kubectl port-forward --namespace knative-monitoring $(kubectl get pod --namespace knative-monitoring --selector="app=grafana" --output jsonpath='{.items[0].metadata.name}') 8080:3000
```
```

Hit the Curl request from bash and see the output in the Dashboard.

```
curl -v -H "Host: ${HOST}"  
http://${CLUSTER_IP}/v1/models/${MODEL_NAME}:predict -d  
$INPUT_PATH
```

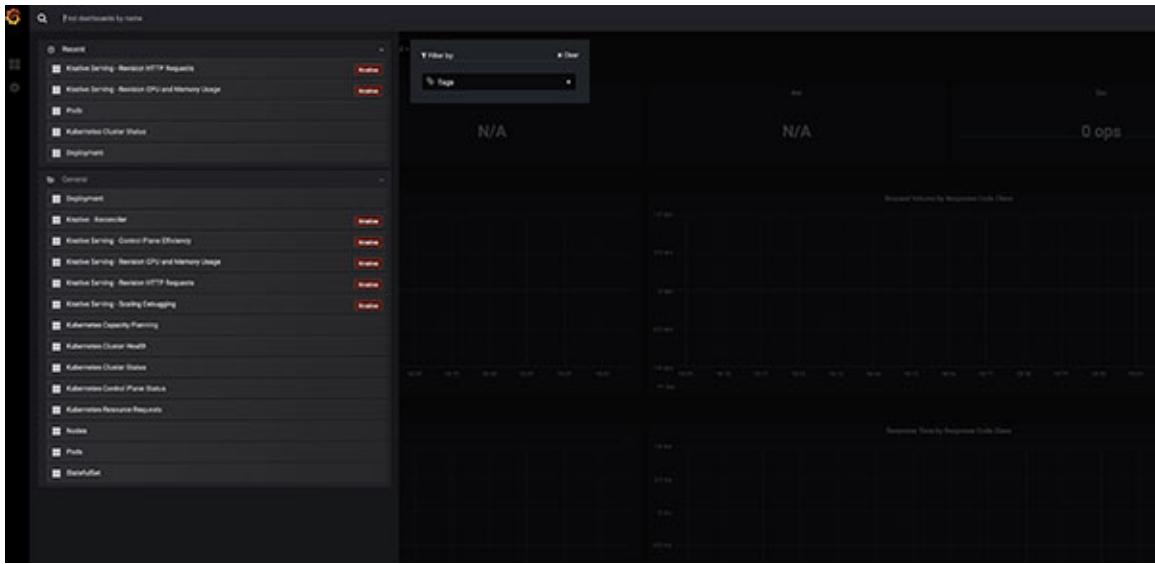


Figure 2.27: Grafana Dashboard Contents

The preceding dashboard is for the contents of Grafana; we can click on the individual section and see the reports there.

The following Dashboard talks about the HTTP requests for Knative Serving Visualization which we serve per/sec request.

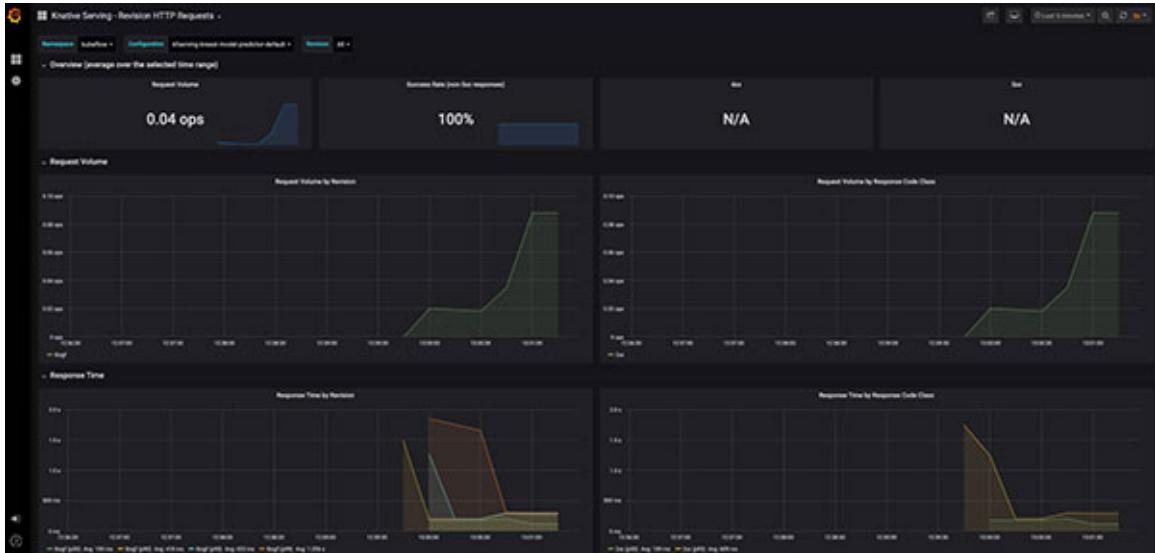


Figure 2.28: Grafana Dashboard HTTP Request

So, the following is the Dashboard for the Control Plane which shows the CPU Usage and memory usage efficiency.



Figure 2.29: Prometheus Dashboard

2.8 Conclusion

In this chapter, we learned how to build end-to-end Kubeflow Orchestrator Pipeline for a TensorFlow Model and how to Dockerize each component and build it in Google Platform.

Then, we learned how to build the pipeline with the kfp library package and triggered the pipeline from the Kubeflow Dashboard. We have now deployed the Kubeflow on the Kubernetes Platform and learned how to trigger the pipeline from the Notebook. We have also deployed the model in the Kubernetes cluster with the KF serving and monitored our prediction results in the Grafana Dashboard.

We have also learned how to leverage the power of Google Cloud Platform, and use our Devops knowledge with Machine Learning to become an Mlops.

2.9 Reference

- <https://knative.dev/docs/serving/autoscaling/autoscaling-concepts/>
- <https://v1-1-branch.Kubeflow.org/docs/>
- <https://v1-1-branch.Kubeflow.org/docs/gke/>
- <https://v1-1-branch.Kubeflow.org/docs/gke/monitoring/>

CHAPTER 3

Designing Computer Vision Model in Kubeflow

In this chapter, we will build an end-to-end TensorFlow Computer Vision Model with OpenCV operation and deploy that with the Kubeflow Orchestration, which includes deploying Kubeflow in Kubernetes Cluster in GCP, building the pipeline components for the model with Docker and Kubeflow SDK, and then serving the Model with KF serving to have an endpoint for prediction. We will then perform the monitor and performance in Grafana Dashboard.

Structure

In this chapter, we will cover the following topics:

- Problem statement
- Getting started in GCP Kubeflow and Docker setup
- Analytics behind the problem statement
- Building the Kubeflow Pipeline components for Computer Vision (CNN) TensorFlow model
- Serving the Model with KF Serving
- Building the pipeline end to end
- Auto-Scaling of the serving endpoint

Objectives

In this chapter, we will learn the following:

- How to set Docker and Kubernetes to build the Kubeflow Pipeline.
- How to pre-process Image with OpenCV library, and build our data for the training model, and how it will be used for Kubeflow pipeline.
- How to do Ingestion of data from an external source like Kaggle and how to do Batch-Prediction.

- How to build the individual pipeline components like training and model evaluation.
- How to serve the Model with KF serving, and predict the model request and monitor with the Grafana Dashboard.
- How to use Kubernetes and many Google Cloud Platform to leverage the power of Machine learning with Devops Knowledge.
- How to use pre-trained model weights of TensorFlow and use that for Model building.
- How to autoscale a kf-serving inference service with concurrent request and target concurrency.

3.1 Problem statement

Here, we have a classification dataset of the brain cancer X-ray images, and it has a two-class folder, one having Brain Tumor and the other not; we will publish the data from Kaggle and build an end to end classification model; after that we have to deploy the same.

- **NO:** No tumor, encoded as 0
- **YES:** Tumor, encoded as 1

| | |
|-------------|---|
| NOTE | Rest all the imports I have showed in my Notebook, which we gave hyperlink of my Github Account of this chapter. |
| CODE | https://github.com/bpbpublications/Continuous-Machine-Learning-with-Kubeflow/tree/main/Chapter3 |

3.2 Getting started in GCP Kubeflow setup

Before we start with this chapter, we must set up the Kubeflow Cluster in GCP; please refer to [Chapter 1, Introduction to Kubeflow & Kubernetes Cloud Architecture](#), Section 1.6 (Getting Started in GCP Kubeflow setup).

3.3 Analytics behind the problem statement

The main purpose of this project was to build a CNN model that would classify if the subject has a tumor or not, based on an MRI scan. We used the **VGG-16** model architecture and weights to train the model for this binary problem. We

used accuracy as a metric to justify the model performance which can be defined as follows:

What is brain tumor?

A cancerous or non-cancerous mass or growth of abnormal cells in the brain. Tumors can start in the brain, or the cancer from elsewhere in the body can spread to the brain. These symptoms may include headaches, seizures, problems with vision, vomiting, and mental changes. The headache is classically worse in the morning and goes away with vomiting. Other symptoms may include difficulty in walking, speaking or difficulty with other sensations. As the disease progresses, unconsciousness may occur.

Here, our goal is to compute the extreme points along the contour of the brain scan in the image; find the extreme north, south, east, and west (x, y)-coordinates along a given contour. This method can be used on both raw contours and rotated bounding boxes.

Let's import the required libraries.

```
from IPython.display import clear_output
!pip install imutils
clear_output()
import numpy as np
from tqdm import tqdm
import cv2,os, shutil, itertools, imutils
import plotly.express as px
from skimage import io
import plotly.graph_objs as go
from plotly.offline import init_notebook_mode, iplot
from plotly import tools
from plotly.subplots import make_subplots
```

The following function performs thresholding the image and a series of erosions and dilations to remove any small regions of noise; next it finds contours in the threshold image, then grabs the largest one. At last, it finds extreme points to crop the image, and return all the steps array, so that we can plot them.

```
def analytics(filepath):
```

```

IMG_SIZE = (224,224)
img = cv2.imread(filepath)
img = cv2.resize(img, dsize=IMG_SIZE,
interpolation=cv2.INTER_CUBIC)
gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
gray = cv2.GaussianBlur(gray, (5, 5), 0)
# threshold the image, then perform a series of erosions +
# dilations to remove any small regions of noise
thresh = cv2.threshold(gray, 45, 255, cv2.THRESH_BINARY)[1]
thresh = cv2.erode(thresh, None, iterations=2)
thresh = cv2.dilate(thresh, None, iterations=2)
# find contours in thresholded image, then grab the largest
one
cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
cnts = imutils.grab_contours(cnts)
c = max(cnts, key=cv2.contourArea)
# find the extreme points
extLeft = tuple(c[c[:, :, 0].argmin()][0])
extRight = tuple(c[c[:, :, 0].argmax()][0])
extTop = tuple(c[c[:, :, 1].argmin()][0])
extBot = tuple(c[c[:, :, 1].argmax()][0])
# add contour on the image
img_cnt = cv2.drawContours(img.copy(), [c], -1, (0, 255, 255),
4)
# add extreme points
img_pnt = cv2.circle(img_cnt.copy(), extLeft, 8, (0, 0, 255),
-1)
img_pnt = cv2.circle(img_pnt, extRight, 8, (0, 255, 0), -1)
img_pnt = cv2.circle(img_pnt, extTop, 8, (255, 0, 0), -1)
img_pnt = cv2.circle(img_pnt, extBot, 8, (255, 255, 0), -1)
# crop
ADD_PIXELS = 0
new_img = img[extTop[1]-ADD_PIXELS:extBot[1]+ADD_PIXELS,
extLeft[0]-ADD_PIXELS:extRight[0]+ADD_PIXELS].copy()

```

```
return img,img_cnt,img_pnt,new_img
```

Here, we can call the function by downloading the sample data folder from the GitHub link.

```
img,img_cnt,img_pnt,new_img=analytics('~/<YOUR_LOCATION>/Sample_Tum  
or_Data/Y2.jpg')
```

Next, we will create a utility plotly function to plot the extreme points of the original and cropped image with the help of plotly.

```
def make_subplots_image(img,img_cnt,img_pnt,new_img):  
    fig = make_subplots(1, 4, horizontal_spacing=0.08)  
    fig.add_trace(go.Image(z=img), 1, 1)  
    fig.add_trace(go.Image(z=img_cnt), 1, 2)  
    fig.add_trace(go.Image(z=img_pnt), 1, 3)  
    fig.add_trace(go.Image(z=new_img), 1, 4)  
    fig['layout']['xaxis1'].update(showgrid=False, title='Original  
image')  
    fig['layout']['xaxis2'].update(showgrid=False, title='Find the  
biggest contour')  
    fig['layout']['xaxis3'].update(showgrid=False, title='Find the  
extreme points')  
    fig['layout']['xaxis4'].update(showgrid=False, title='Crop the  
image')  
    fig.update_layout(height=400, width=1500)  
    fig.show()
```

Call the preceding function:

```
make_subplots_image(img,img_cnt,img_pnt,new_img)
```

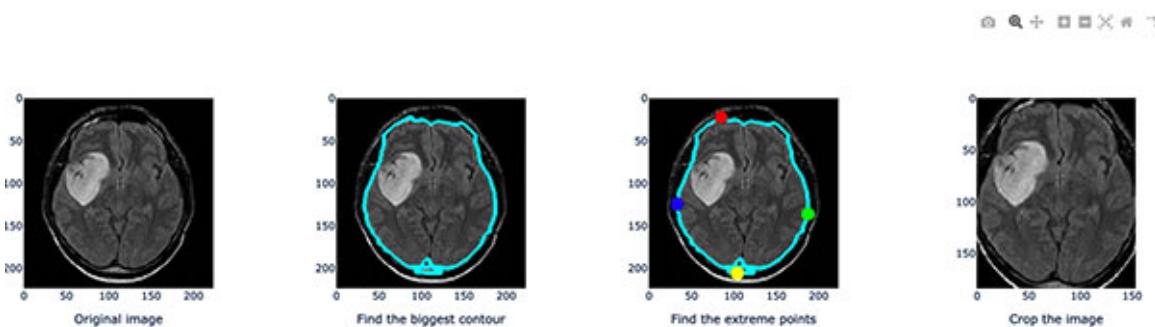


Figure 3.1: Extreme Points of Brain Scan

Now, we will apply the transformation for all the images for our training data during our Pipeline building phase.

3.4. Building the Kubeflow pipeline components for Computer Vision (CNN) TensorFlow model

In this section, we will see the complete Architecture that we will be using to build the individual components.

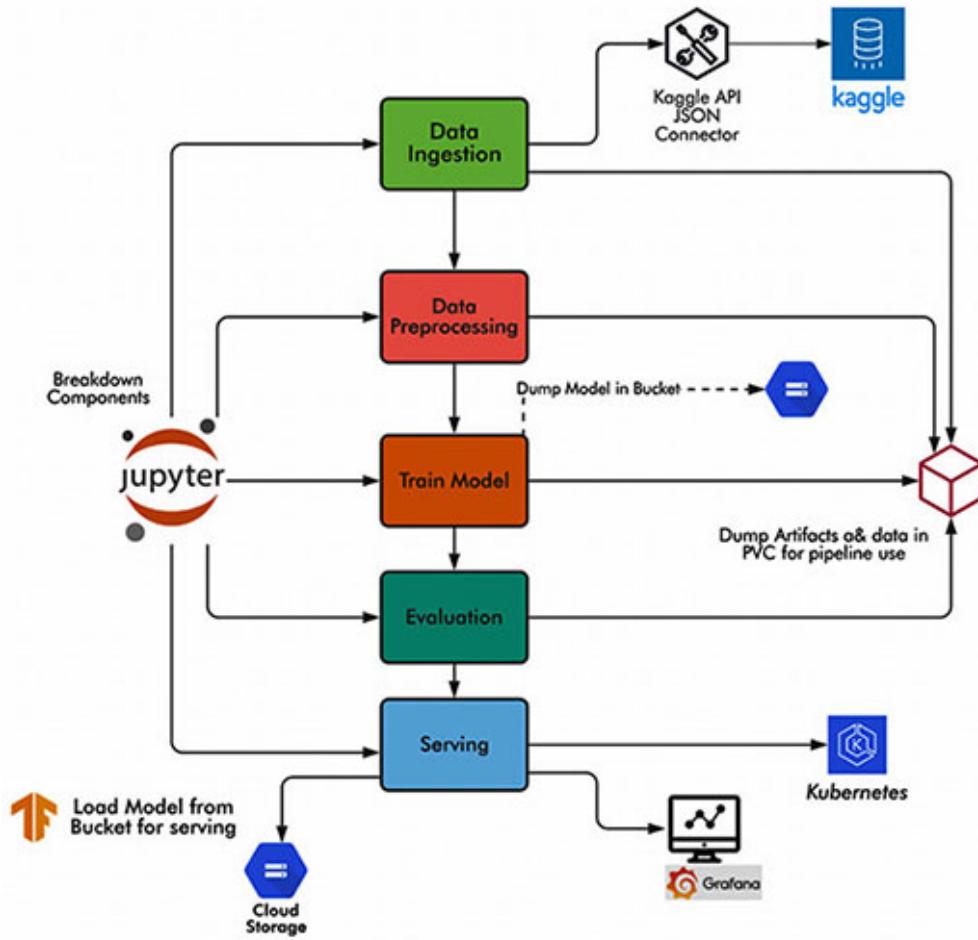


Figure 3.2: Architecture of CNN Kubeflow Pipeline

From the preceding diagram, let's breakdown the component's overview as follows:

- The data ingestion will be provided by Kaggle connector API, and we will download that inside PVC. It will be used for further pipeline

components.

- Next, we will train our CNN TensorFlow model, and save that in the GCP bucket.
- After that, we will evaluate our model performance and visualize that in the Kubeflow Dashboard with the ROC and Confusion Matrix for that experiment Run.
- At last, we will deploy the model for serving and loading that model from the bucket which we saved during the training phase, and then check the model endpoint performance traffic in the Grafana dashboard.
- We will see how to autoscale a kf-serving InferenceService with the concurrent request and target concurrency, and here we will build a Batch-Prediction of KF serving.

3.4.1 Data extraction or Ingestion component

Now, let's build the Data ingestion component; here we will get the data from Kaggle, and we will extract that with the help of Kaggle API.

dataextraction.py:

```
from __future__ import absolute_import, division, print_function,
unicode_literals
import click,json,os,dill,argparse,logging,shutil,itertools,Kaggle

@click.command()
@click.option('--data-file', default="/mnt/BrainScan_Data/")
@click.option('--root',default="/mnt/")
@click.option('--kaggle-api-data',default="navoneel/brain-mri-
images-for-brain-tumor-detection")
def download_data(root,data_file,kaggle_api_data):
    logging.info(kaggle.api.authenticate())
    kaggle.api.dataset_download_files(kaggle_api_data,
path=data_file, unzip=True)
    logging.info("Downloaded Data")
    print(len(os.listdir(data_file +"brain_tumor_dataset/no")))
    print(len(os.listdir(data_file +"brain_tumor_dataset/yes")))
```

```

directory=["TRAIN", "TEST", "VAL", "TRAIN/YES", "TRAIN/NO",
"TEST/YES", "TEST/NO", "VAL/YES", "VAL/NO"]
for i in directory:
    path = os.path.join(root, i)
    try:
        os.mkdir(path)
    except OSError as error:
        print(error)

for CLASS in os.listdir(data_file):
    logging.info(CLASS)
    print(CLASS)
    if not CLASS.startswith('.'):
        IMG_NUM = len(os.listdir(data_file + CLASS))
        logging.info(IMG_NUM)
        print(IMG_NUM)
        for (n, FILE_NAME) in enumerate(os.listdir(data_file +
CLASS)):
            img = data_file + CLASS + '/' + FILE_NAME
            if n < 5:
                try:
                    shutil.copy(img, '/mnt/TEST/' + CLASS.upper() + '/' +
FILE_NAME)
                except OSError as error:
                    print(error)
            elif n < 0.8*IMG_NUM:
                try:
                    shutil.copy(img, '/mnt/TRAIN/' + CLASS.upper() + '/' +
FILE_NAME)
                except OSError as error:
                    print(error)
            else:
                try:
                    shutil.copy(img, '/mnt/VAL/' + CLASS.upper() + '/' +
FILE_NAME)
                except OSError as error:

```

```

        print(error)
    return
if __name__ == "__main__":
    download_data()

```

Let's breakdown the code as follows:

- Load the data from Kaggle DB with Kaggle API, and give it a particular project name.
- Import all the required datasets.
- The `@click` command is a passing argument in the function `get_data()`.
- Next, we will create individual folders for Train, Test, Validation, and we will shuffle the raw data to fill all the folders inside Tumour Yes or No Tumour sub-folders.
- All the folders saved and stored inside the PVC which will be used after pipeline.

Next, see how we will build the component Dockerfile for the data ingestion and we will keep the `requirements.txt` file in the root of the dockerfile, which we will be needing as an environment to run the Python function inside Docker. Then, we will place the kaggle api json file in the root of Docker for authentication to Kaggle to download the data from the Kaggle projects.

To get the Kaggle API json file, please look into [Chapter 6, Building Weights & Biases Pipeline Development](#), Section 6.2.2 (Kaggle API Setup).



Figure 3.3: dataextraction folder

Now, we will build the Docker image; for that make sure you start the Docker, and you will see the Docker is activated on the top. As we can see, the first icon Docker is started.

Docker:

```
FROM python:3.7-slim-stretch
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update && \
    apt-get -y install gcc mono-mcs g++ git curl && \
    rm -rf /var/lib/apt/lists/*
RUN mkdir /app
RUN mkdir ~/.kaggle
WORKDIR /app
ADD requirements.txt /app/requirements.txt
RUN pip3 install -r requirements.txt
ADD dataextract.py /app/dataextract.py
ADD kaggle.json /app/kaggle.json
RUN chmod +x /app/dataextract.py
RUN cp /app/kaggle.json ~/.kaggle
RUN chmod 600 ~/.kaggle/kaggle.json
ENTRYPOINT ["python"]
CMD ["/app/dataextract.py"]
```

So let's break the Docker file code as follows:

- Load the Python image as the base Image to run the pipeline code of 3.7 version.
- The **RUN** function will run whenever the Docker image will start and create a **/app** folder first with the **mkdir** command, and another one for the **./kaggle** folder for storing kaggle API json file.
- Next, we will redirect our command to **WORKDIR** where all the files and dependencies will be installed.
- Next, we will **ADD** the requirements file in the **/app** folder, and run the **pip** command to install the Python libraries.
- Then we will copy our **pipeline.py** Python code and **kaggle.json** giving an administrative access to run the file whenever the Docker

image will run with the **chmod** command.

- Next, we will set our entry point as Python and CMD as “**/app/dataextract.py**” to run this first.

Now, we will build the Docker image; for that, make sure you start the Docker, and you will see the Docker is activated on top. As we can see, the first icon Docker is started.

Run all the following commands by connecting to GCP via Local:

```
```bash
gcloud init
#Select the Email/Project associated with GCP
```

```

Build the container for Data Extraction.

```
```bash
cd $WORKDIR/pipeline/dataextraction
PROJECT_ID=$(gcloud config get-value core/project)
IMAGE_NAME=brain_tumor_scan/step1_download_data
IMAGE_VERSION=v1
IMAGE_NAME=gcr.io/$PROJECT_ID/$IMAGE_NAME
```

```

Building Docker image:

```
```
docker build -t $IMAGE_NAME:$IMAGE_VERSION .
```

```

Push training image to **Google container registry (GCR)**:

```
```
docker push $IMAGE_NAME:$IMAGE_VERSION
```

```

Next, we will see how to build the training component and similarly as the above, after that we going to build the pre-processing step for our Kubeflow pipeline.

3.4.2 Data pre-processing component

Now, we will build the pre-processing step, assuming the input will come from the previous data ingestion component. So, whatever data we have dumped with the `@dill` command, we will load first and will do the necessary pre-processing step.

In this part of the code, let's import all the dependency and create the utility correlation plot function.

Preprocessing.py

```
from __future__ import absolute_import, division, print_function,
unicode_literals
import click,json,os,argparse,dill,cv2,imutils
from tqdm import tqdm
import numpy as np
import tensorflow as tf
def load_data_array(dir_path, img_size=(100,100)):
    X = []
    y = []
    i = 0
    labels = dict()
    for path in tqdm(sorted(os.listdir(dir_path))):
        if not path.startswith('.'):
            labels[i] = path
            for file in os.listdir(dir_path + path):
                if not file.startswith('.'):
                    img = cv2.imread(dir_path + path + '/' + file)
                    X.append(img)
                    y.append(i)
            i += 1
    X = np.array(X)
    y = np.array(y)
    print(f'{len(X)} images loaded from {dir_path} directory.')
    return X, y, labels

def crop_imgs(set_name, add_pixels_value=0):
    set_new = []
```

```

for img in set_name:
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    gray = cv2.GaussianBlur(gray, (5, 5), 0)
    thresh = cv2.threshold(gray, 45, 255, cv2.THRESH_BINARY)[1]
    thresh = cv2.erode(thresh, None, iterations=2)
    thresh = cv2.dilate(thresh, None, iterations=2)
    cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
                           cv2.CHAIN_APPROX_SIMPLE)
    cnts = imutils.grab_contours(cnts)
    c = max(cnts, key=cv2.contourArea)
    extLeft = tuple(c[c[:, :, 0].argmin()][0])
    extRight = tuple(c[c[:, :, 0].argmax()][0])
    extTop = tuple(c[c[:, :, 1].argmin()][0])
    extBot = tuple(c[c[:, :, 1].argmax()][0])
    ADD_PIXELS = add_pixels_value
    new_img = img[extTop[1]-ADD_PIXELS:extBot[1]+ADD_PIXELS,
                  extLeft[0]-ADD_PIXELS:extRight[0]+ADD_PIXELS].copy()
    set_new.append(new_img)
return np.array(set_new)

```

The code is continued to next page here; in the preceding code, we have imported the required libraries and built some utility functions like **crop_img()** and **load_data_array()**.

In continuation, the pre-processing code is as follows:

```

def save_new_images(x_set, y_set, folder_name):
    i = 0
    for (img, imclass) in zip(x_set, y_set):
        if imclass == 0:
            cv2.imwrite(folder_name+'NO/'+str(i)+'.jpg', img)
        else:
            cv2.imwrite(folder_name+'YES/'+str(i)+'.jpg', img)
        i += 1

def preprocess_images(set_name, img_size):
    set_new = []

```

```

for img in set_name:
    img = cv2.resize(img, dsize=img_size,
                    interpolation=cv2.INTER_CUBIC)
    set_new.append(tf.keras.applications.vgg16.preprocess_input(im
g))
return np.array(set_new)

@click.command()
@click.option('--root', default="/mnt/")
@click.option('--train-file', default="/mnt/training.data")
@click.option('--test-file', default="/mnt/test.data")
@click.option('--validation-file', default="/mnt/validation.data")
@click.option('--train-target',
              default="/mnt/trainingtarget.data")
@click.option('--test-target', default="/mnt/testtarget.data")
@click.option('--validation-target',
              default="/mnt/validationtarget.data")
@click.option('--label', default="/mnt/labels.data")
@click.option('--image-size', default=224)
def training_data_processing(root, train_file, label, test_file, valida
tion_file, image_size, train_target, test_target, validation_target):
    TRAIN_DIR = root + 'TRAIN/'
    TEST_DIR = root + 'TEST/'
    VAL_DIR = root + 'VAL/'
    IMG_SIZE = (image_size, image_size)
    X_train, y_train, labels = load_data_array(TRAIN_DIR, IMG_SIZE)
    X_test, y_test, _ = load_data_array(TEST_DIR, IMG_SIZE)
    X_val, y_val, _ = load_data_array(VAL_DIR, IMG_SIZE)

    X_train_crop = crop_imgs(set_name=X_train)
    X_val_crop = crop_imgs(set_name=X_val)
    X_test_crop = crop_imgs(set_name=X_test)
    directory=[["TRAIN_CROP", "TEST_CROP", "VAL_CROP",
                "TRAIN_CROP/YES", "TRAIN_CROP/NO", "TEST_CROP/YES",
                "TEST_CROP/NO", "VAL_CROP/YES", "VAL_CROP/NO"]]
    for i in directory:
        path = os.path.join(root, i)

```

```

try:
    os.mkdir(path)
except OSError as error:
    print(error)
    save_new_images(X_train_crop, y_train,
                    folder_name='/mnt/TRAIN_CROP/')

    save_new_images(X_val_crop, y_val,
                    folder_name='/mnt/VAL_CROP/')
    save_new_images(X_test_crop, y_test,
                    folder_name='/mnt/TEST_CROP/')

```

The code is continued to the next page; in the preceding section of the code, we are inputting the files from the previous pipeline component, after which we will transform the images into an array, and crop those images; after that we will be creating some new folders for those cropped images and save those images.

```

with open(label,"wb") as f:
    dill.dump(labels,f)
with open(train_file,"wb") as f:
    dill.dump(X_train_prep,f)
with open(test_file,"wb") as f:
    dill.dump(X_test_prep,f)
with open(validation_file,"wb") as f:
    dill.dump(X_val_prep,f)
with open(train_target,"wb") as f:
    dill.dump(y_train,f)
with open(test_target,"wb") as f:
    dill.dump(y_test,f)
with open(validation_target,"wb") as f:
    dill.dump(y_val,f)
return
if __name__ == "__main__":
    training_data_processing()

```

Let's breakdown the code for pre-processing as follows:

- First of all, we will import all the libraries and create the utility function like `crop_img()` which will crop an original image, and `load_data_array()` will help to transform the images into arrays X, y and labels.
- Next, the `save_new_images()` will save those cropped images into new folders and the `pre_process_images()` function will pre-process those cropped images with the help of the vgg16 applications of keras and save those in the new array sets.
- Next, the `@click.option` will input all the required dumps which we have done in the data ingestion step, so that we can use here, and load those first and use those for the pre-processing steps for training the data.
- Then, the `@dill.dump` will dump the pre-process arrays for training for the next pipeline components.

Docker:

```
FROM python:3.7-slim-stretch
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update && \
    apt-get -y install gcc mono-mcs g++ git curl && \
    rm -rf /var/lib/apt/lists/*
RUN mkdir /app
WORKDIR /app
RUN curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
RUN python get-pip.py
RUN rm get-pip.py
RUN pip install --upgrade pip
RUN apt-get update && yes | apt-get upgrade
RUN apt-get install -y libsm6 libxext6 libxrender-dev
RUN apt-get install -y protobuf-compiler python-pil python-lxml
python-pip python-dev git
RUN apt-get update && apt-get install -y protobuf-compiler python-
pil python-lxml
ADD requirements.txt /app/requirements.txt
RUN pip3 install -r requirements.txt
ADD preprocessing.py /app/preprocessing.py
```

```
RUN chmod +x /app/preprocessing.py  
ENTRYPOINT ["python"]  
CMD ["/app/preprocessing.py"]
```

Similarly like the preceding data ingestion step, we will build the Docker image. As we can see in the preceding Docker images, we have installed the required libraries to run any computer vision Docker image.

```
```bash  
PROJECT_ID=$(gcloud config get-value core/project)
IMAGE_NAME=brain_tumor_scan/step2_dataprocessing
IMAGE_VERSION=v1
IMAGE_NAME=gcr.io/$PROJECT_ID/$IMAGE_NAME
```
```

Building Docker image.

```
```  
docker build -t $IMAGE_NAME:$IMAGE_VERSION .
```
```

Push training image to GCR.

```
```  
docker push $IMAGE_NAME:$IMAGE_VERSION
```
```

3.4.3 Training model component

In this section, we will build a TensorFlow CNN model, and download the pre-trained VGG-16 weights and keep that in the path of the train folder. So, go to the following link and download the file:

[https://www.kaggle.com/gaborfodor/keras-pretrained-models?
select=vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5](https://www.kaggle.com/gaborfodor/keras-pretrained-models?select=vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5)

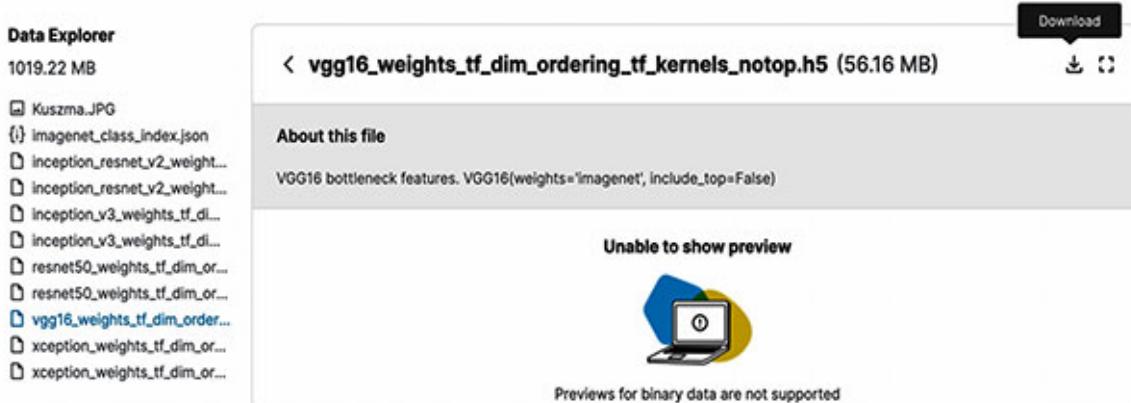


Figure 3.4: VGG16 download location

This is how our train folder looks, and we will keep the storage bucket service account and `Storage.py` to upload to the bucket.

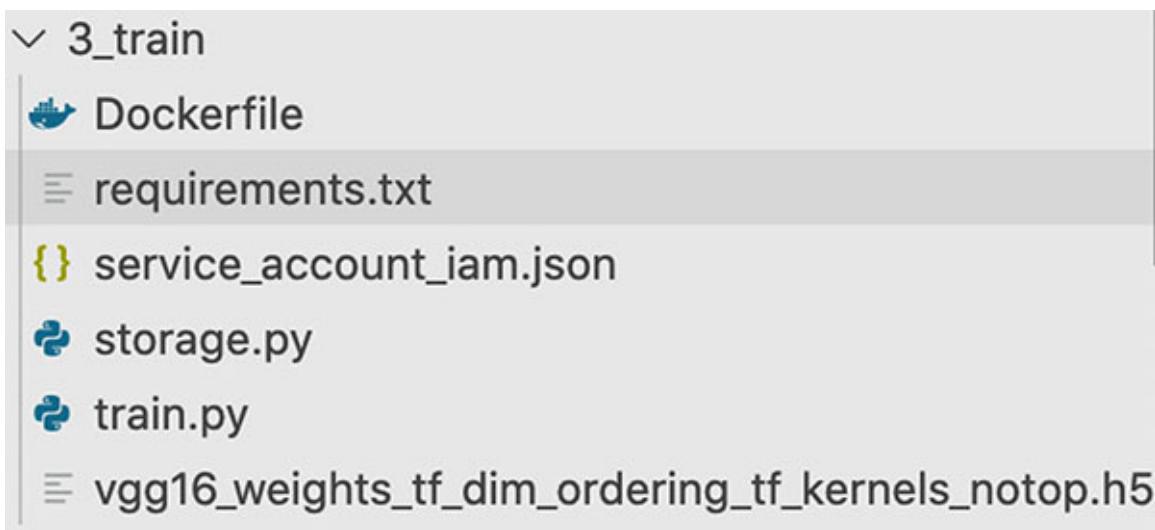


Figure 3.5: train folder

Next, we will find out how we built the Train Component.

Train.py

```
from __future__ import absolute_import, division, print_function,  
unicode_literals  
import click,dill,json,logging,os,PIL  
import pandas as pd  
import tensorflow as tf  
from storage import Storage  
from sklearn.metrics import accuracy_score
```

```

def model_build(base_model,NUM_CLASSES,activation):
    model =
        tf.keras.models.Sequential([base_model,tf.keras.layers.Flatten(),
        ),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(NUM_CLASSES,activation=activation)])
    model.layers[0].trainable = False
    return model

@click.command()
@click.option('--train-file', default="/mnt/training.data")
@click.option('--test-file', default="/mnt/test.data")
@click.option('--validation-file', default="/mnt/validation.data")
@click.option('--train-target',
default="/mnt/trainingtarget.data")
@click.option('--test-target', default="/mnt/testtarget.data")
@click.option('--validation-target',
default="/mnt/validationtarget.data")
@click.option('--epochs', default=100)
@click.option('--activation', default="sigmoid")
@click.option('--learning-rate', default=0.001)
@click.option('--tensorboard-logs', default='/mnt/logs/')
@click.option('--tensorboard-gcs-logs',
default='gs://kubeflowusecases/brain/logs')
@click.option('--model-output-base-path',
default="/mnt/saved_model")
@click.option('--gcs-path',
default="gs://kubeflowusecases/brain/model")
@click.option('--mode', default="local")
@click.option('--image-size', default=224)
@click.option('--label', default="/mnt/labels.data")
def train_model(train_file,test_file,validation_file,train_target,test_target,validation_target,
label,epochs,activation,image_size,learning_rate,tensorboard_logs,
tensorboard_gcs_logs,model_output_base_path,gcs_path,mode):
    with open(label, 'rb') as in_f:

```

```

    labels= dill.load(in_f)
with open(train_file, 'rb') as in_f:
    train= dill.load(in_f)
with open(test_file, 'rb') as in_f:
    test= dill.load(in_f)
with open(validation_file, 'rb') as in_f:
    validation= dill.load(in_f)
with open(train_target, 'rb') as in_f:
    train_tar= dill.load(in_f)
with open(test_target, 'rb') as in_f:
    test_tar= dill.load(in_f)
with open(validation_target, 'rb') as in_f:
    validation_tar= dill.load(in_f)

```

The code is continued to the next page, here in preceding section, we have imported the required libraries and we have all the input from the previous pipeline; we have imported them with @dill load. And we have created one utility function **model_build()** for building layers for TensorFlow model.

In continuation to the preceding snippet code, the **train.py** Python file contains the following section:

```

IMG_SIZE = (image_size,image_size)
RANDOM_SEED = 123
TRAIN_DIR = '/mnt/TRAIN_CROP/'
VAL_DIR = '/mnt/VAL_CROP/'

train_datagen =
tf.keras.preprocessing.image.ImageDataGenerator(
    rotation_range=15,width_shift_range=0.1,height_shift_range=
0.1,shear_range=0.1,brightness_range=[0.5, 1.5],
horizontal_flip=True,vertical_flip=True,
preprocessing_function=tf.keras.applications.vgg16.preproce
ss_input)
train_generator =
train_datagen.flow_from_directory(TRAIN_DIR,color_mode='rgb',
target_size=IMG_SIZE,batch_size=32,class_mode='binary',
seed=RANDOM_SEED)

```

```
val_datagen = tf.keras.preprocessing.image.ImageDataGenerator(  
    preprocessing_function=tf.keras.applications.vgg16.preprocess_  
    input)  
validation_generator = val_datagen.flow_from_directory(  
    VAL_DIR,color_mode='rgb',target_size=IMG_SIZE,  
    batch_size=16,class_mode='binary',seed=RANDOM_SEED)  
vgg16_weight_path="/app/vgg16_weights_tf_dim_ordering_tf_kernels_n  
otop.h5"  
base_model=tf.keras.applications.VGG16(  
    include_top=False, weights=vgg16_weight_path,  
    input_shape=IMG_SIZE + (3,))  
NUM_CLASSES=1  
model=model_build(base_model,NUM_CLASSES,activation)  
optimizer=tf.keras.optimizers.RMSprop(learning_rate=learning_r  
ate)  
model.compile(loss=tf.keras.losses.binary_crossentropy,optimiz  
er=optimizer, metrics=['accuracy'])  
tensorboard_callback =  
tf.keras.callbacks.TensorBoard(log_dir=tensorboard_logs,  
histogram_freq=1)  
earlystopping =  
tf.keras.callbacks.EarlyStopping(monitor='val_loss',  
mode='max',patience=6)  
logging.info("Training starting...")  
model.fit_generator(train_generator,epochs=epochs,  
    validation_data=validation_generator,validation_steps=25,  
    callbacks=[earlystopping,tensorboard_callback])  
logging.info("Training completed.")  
model.save(model_output_base_path)  
new_model = tf.keras.models.load_model(model_output_base_path)  
print(new_model.summary())  
predictions = new_model.predict(validation)  
predictions = [1 if x>0.5 else 0 for x in predictions]  
accuracy = accuracy_score(validation_tar, predictions)  
print('Val Accuracy = %.2f' % accuracy)
```

```

logging.info('Val Accuracy = %.2f' % accuracy))
Storage.upload(tensorboard_logs,tensorboard_gcs_logs)
metadata = {'outputs': [{'type': 'tensorboard',
    'source': tensorboard_gcs_logs,}]}

```

So, the preceding code contains the training and pre-processing with Image Data Generator after that we trained the model with preprocessed images in pipeline and the trained artifacts are saved as tensorflow models in GCP Bucket.

```

with open("/mlpipeline-ui-metadata.json", 'w') as f:
    json.dump(metadata,f)
if mode!= 'local':
    print("uploading to {}".format(gcs_path))
    Storage.upload(model_output_base_path,gcs_path)
else:
    print("Model will not be uploaded")
    pass

if __name__ == "__main__":
    train_model()

```

So, the preceding code is the **train.py** file; lets break the pipeline code as follows:

- Import all the required datasets.
- The **@click** command is passing the argument in the function **train_model()**.
- Here, we load the VGG16 model from the Docker root in the train Python code, so that we can import that with TensorFlow for our model training, Then, the **dill** command will be helping to save or load the data from pvc. Next, we will pass all the arguments like epochs and batch size, learning rate from the outside and will train our model.
- We will then push the model train output to the Google Storage Bucket. Then, we will import the **Storage.py** file to the imported storage class to save our model artifacts.

- And to visualize, the Tensorboard will follow the same metadata json format with “`/mpipeline-ui-metadata.json`” and will give the gcs bucket path where you saved your model artifacts.

Similarly, we will create the Docker Image, and add the service account json key. Please have a look at the GitHub `steps.md` file.

3.3.4 Evaluation component

Similarly, we will create the evaluation component. Please have a look at the GitHub `steps.md` file. We have dumped the confusion Matrix and ROC Curve as a csv in GCP bucket and gave the path a storage location of those in the metadata and dumped that as json.

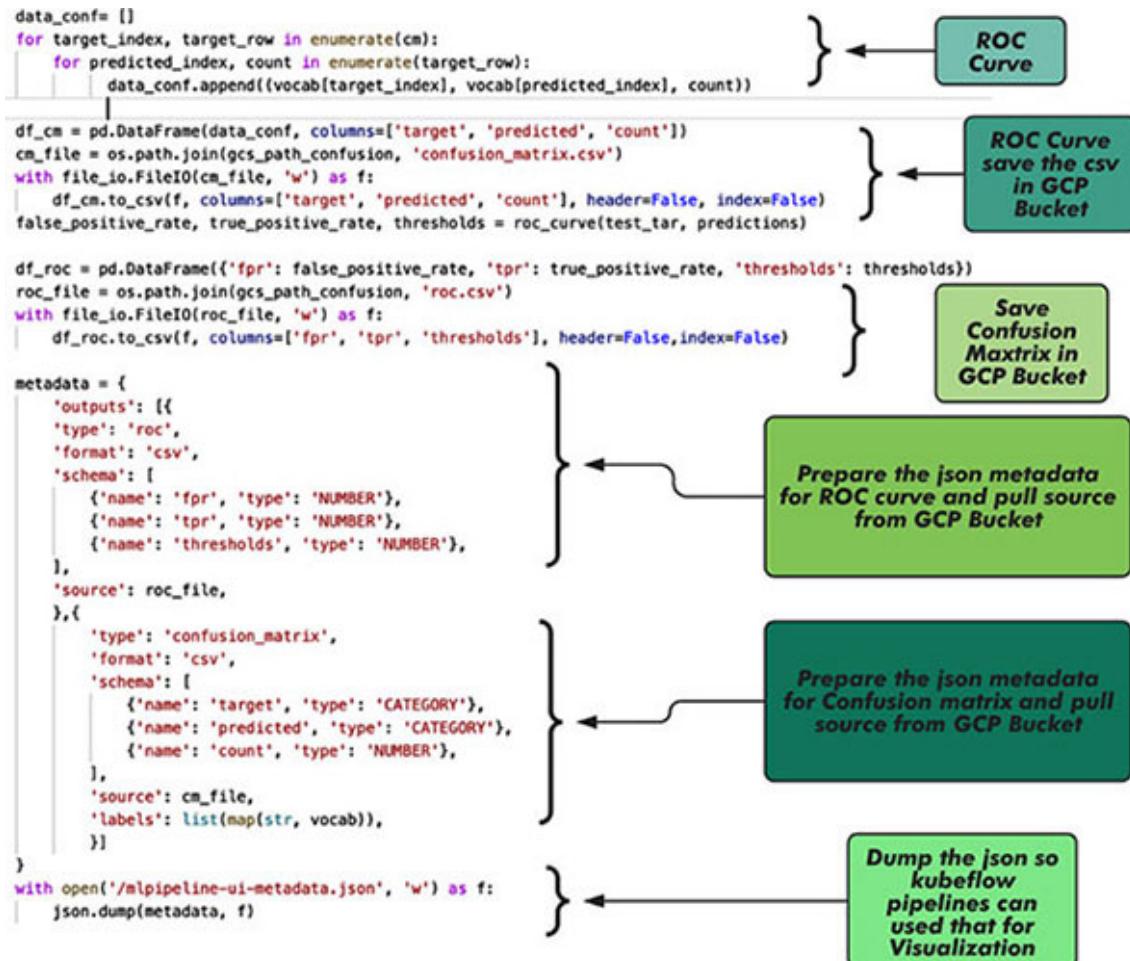


Figure 3.6: Evaluator Component

3.5. Serving the Model with KF Serving

In this section, we will build our serving model. As shown in the following section, there are three major components for the service account with the storage bucket access json file, **requirements.txt**, Dockerfile, **brainserving.py**.

We will build a custom endpoint serving, and we will see how we can do a Batch prediction of multiple test input images.

brainserving.py

```
import kfserving, argparse, json, cv2,
logging,os,base64,io,imutils
from typing import List, Dict
import numpy as np
from PIL import Image
import tensorflow as tf
os.environ['GOOGLE_APPLICATION_CREDENTIALS'] =
"service_account_iam.json"

def crop_imgs(set_name, add_pixels_value=0):
    set_new = []
    for img in set_name:
        gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
        gray = cv2.GaussianBlur(gray, (5, 5), 0)
        thresh = cv2.threshold(gray, 45, 255, cv2.THRESH_BINARY)[1]
        thresh = cv2.erode(thresh, None, iterations=2)
        thresh = cv2.dilate(thresh, None, iterations=2)
        cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
                               cv2.CHAIN_APPROX_SIMPLE)
        cnts = imutils.grab_contours(cnts)
        c = max(cnts, key=cv2.contourArea)
        extLeft = tuple(c[c[:, :, 0].argmin()][0])
        extRight = tuple(c[c[:, :, 0].argmax()][0])
        extTop = tuple(c[c[:, :, 1].argmin()][0])
        extBot = tuple(c[c[:, :, 1].argmax()][0])
        ADD_PIXELS = add_pixels_value
        new_img = img[extTop[1]-ADD_PIXELS:extBot[1]+ADD_PIXELS,
                     extLeft[0]-ADD_PIXELS:extRight[0]+ADD_PIXELS].copy()
```

```

        set_new.append(new_img)
    return np.array(set_new)

def preprocess_imgs(set_name, img_size):
    set_new = []
    for img in set_name:
        img =
            cv2.resize(img, dsize=img_size, interpolation=cv2.INTER_CUBIC)
        set_new.append(tf.keras.applications.vgg16.preprocess_input(im
g))
    return np.array(set_new)

def image_transform(instance):
    logging.info("Inside Image Transform")
    originalimage = base64.b64decode(instance)
    jpg_as_np = np.frombuffer(originalimage, dtype=np.uint8)
    img = cv2.imdecode(jpg_as_np, flags=1)
    image_expanded = np.expand_dims(img, axis=0)
    crop_image = crop_imgs(set_name=image_expanded)
    IMG_SIZE=(224,224)
    prep_image = preprocess_imgs(set_name=crop_image,
    img_size=IMG_SIZE)
    return prep_image

class Transformer(kfserving.KFModel):
    def __init__(self, name: str):
        super().__init__(name)
        self.name = name
        self.ready = False
        self.model_output_base_path='gs://kubeflowusecases/brain/model
/'

    def load(self):
        self.model =
            tf.keras.models.load_model(self.model_output_base_path)
        self.ready = True
    def predict(self, request: Dict) -> Dict:

```

```

data={'instances': [image_transform(request['instances'][i])
for i in range(len(request['instances']))]}
transformdata=[]
for i in data['instances']:
    logging.info("Inside transform data")
    arraydata=self.model.predict(i)
    logging.info(self.model.predict(i))
    transformdata.append(arraydata)
result=[]
Predict=0
predictions = [1 if x>0.5 else 0 for x in transformdata]
for i in predictions:
    if i == Predict:
        result.append("No tumor inside Brain")
    else:
        result.append("Tumor inside Brain")
return json.dumps({"predictions" : result})

if __name__ == "__main__":
    model = Transformer("kfserving-braintumor")
    model.load()
    kfserving.KFServer(workers=1).start([model])

```

So, let's break the transformer predictor code as follows:

- We kept the service account, which has the storage bucket in the Docker root, and here we have declared that folder as an environment variable to an object **model_output_base_path** for the gcp bucket path.
- Next, in the load function, we loaded the model in from the TensorFlow library.
- Here, the incoming data is an encoded string of Image; then we transformed the image with the following functions **image_transform()**, which will change the encoded strings to the decoded one; next we will crop the image with the **crop_imgs()** function, after which, we will preprocess the array of the image with the **preprocess_image()** and return that array.

- Then, in the predict method, the incoming data will come as a json format which we need to extract as a key-value pair and do the necessary prediction and return as a dictionary.
- So, in the “main” function, the **KFServingSampleModel** Class will take the name of that deployment; keep a note of that and apply to the yaml file; here it is ”kfserving-breast-model”.

Docker:

```
FROM python:3.7-slim-stretch
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update && apt-get -y install gcc mono-mcs g++ git curl
bash && \
    rm -rf /var/lib/apt/lists/*
RUN mkdir /app
WORKDIR /app
RUN curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
RUN python get-pip.py && rm get-pip.py && pip install --upgrade
pip
RUN apt-get update && yes | apt-get upgrade
RUN apt-get install -y libsm6 libxext6 libxrender-dev protobuf-
compiler python-pil python-lxml python-pip python-dev git
protobuf-compiler python-pil python-lxml
ADD requirements.txt /app/requirements.txt
RUN pip3 install -r requirements.txt
ADD brainserving.py /app/brainserving.py
COPY service_account_iam.json service_account_iam.json
ENV GOOGLE_APPLICATION_CREDENTIALS="service_account_iam.json"
CMD ["python", "brainserving.py"]
```

Now, in the preceding Docker code, we have copied the service account for the storage bucket access, and saved it in **/app**. Then, we copied the Python serving file in the same location and kept the working directory as **/app**. Then, we build the image by using the above code. Similarly, we created the Docker image; please have a look at the GitHub **steps.md** file.

To deploy the model server using the **kubectl** command line, or using the KFServing client SDK, you can do either of the following:

- Deploy using the command line
- Deploy using the KFServing client SDK

Deploy using the command line:

Now, let's deploy it with the command line, and first let's fill the yaml file:

Custom_KFServing.yaml:

```
apiVersion: serving.kubeflow.org/v1alpha2
kind: InferenceService
metadata:
  annotations:
    sidecar.istio.io/inject: "false"
  name: kfserving-braintumor
  namespace: kubeflow
spec:
  default:
    predictor:
      custom:
        container:
          image: gcr.io/<PROJECT_ID>/brain_tumor_scan/
          kf_serving_braintest:v1
```

Here, in the preceding yaml file, we will give the same name which we have provided in the **serving.py** file having the model name (“kfserving-braintumor”), and then we will provide the namespace “kubeflow” where it will be deployed. Next, we will give the Docker image name.

```
brainerving.py
...
113     if i == Predict:
114         result.append("No tumor Inside Brain")
115     else:
116         result.append("Tumor Inside Brain")
117
118     return json.dumps({"predictions": result})
119
120 if __name__ == "__main__":
121     model = Transformer("kfserving-braintumor")
122     model.load()
123     kfserving.KFServer(workers=1).start([model])
124
125
126
custom_brain_model.yaml
...
1  apiVersion: serving.kubeflow.org/v1alpha2
2  kind: InferenceService
3  metadata:
4    labels:
5      controller-tools.k8s.io: "1.0"
6    name: kfserving-braintumor
7    namespaces: kubeflow
8  spec:
9    default:
10       predictor:
11         custom:
12           container:
13             image: gcr.io/<PROJECT_ID>/brain_tumor_scan/kf_serving_braintest:v1
14             imagePullPolicy: Always
15             name: user-container
16             imagePullSecrets:
17               - name: user-gcp-sa
```

Line 121th Model
name
[kfserving-braintumor]
should be same on
right side line number
6th

Figure 3.7: KF – Serving Model name match

As we can see, the 121st line number from the left image and the 6th from the right should always be the same.

Next, run the following command from the bash where the files are kept in the Visual Studio:

- Connect to the GCP cluster by using the following command:

```
gcloud container clusters get-credentials <$ClusterName> --  
zone  
<$ZONE> --project <$PROJECTID>
```

- Create the inference service by deploying it in the cluster:

```
kubectl apply -f custom_brain_model.yaml
```

- Check the inference service. Try it after some interval to check if it has been created:

```
kubectl get inferenceservice -n kubeflow
```

```
kfserving-braintumor http://kfserving-braintumor.default.example.com/v1/models/kfserving-braintumor True 100
```

Figure 3.8: KF – Serving Inference service ready

Sample Prediction:

- Run the following command in Bash from the serving folder:

```
```bash  
MODEL_NAME=kfserving-breast-model
HOST=$(kubectl get inferenceservice -n kubeflow$MODEL_NAME -o
jsonpath='{.status.url}' | cut -d "/" -f 3)
INPUT_PATH=@./breast.json
CLUSTER_IP=$(kubectl -n istio-system get service kfserving-
ingressgateway -o
jsonpath='{.status.loadBalancer.ingress[0].ip}')
curl -v -H "Host: ${HOST}"
http://${CLUSTER_IP}/v1/models/${MODEL_NAME}:predict -d
$INPUT_PATH
```
```

Now, the following is the response prediction request:

```

* Trying 35.231.208.41...
* TCP_NODELAY set
* Connected to 35.231.208.41 (35.231.208.41) port 80 (#0)
> POST /v1/models/kfserving-brainumor:predict HTTP/1.1
> Host: kfserving-brainumor.kubeflow.example.com
> User-Agent: curl/7.64.1
> Accept: */*
> Content-Length: 194683
> Content-Type: application/x-www-form-urlencoded
> Expect: 100-continue
>
< HTTP/1.1 100 Continue
* We are completely uploaded and fine
< HTTP/1.1 200 OK
< content-length: 108
< content-type: text/html; charset=UTF-8
< date: Thu, 22 Oct 2020 13:11:09 GMT
< server: istio-envoy
< x-envoy-upstream-service-time: 2920
<
* Connection #0 to host 35.231.208.41 left intact
{"predictions": ["No tumor inside Brain", "Tumor inside Brain", "Tumor inside Brain", "Tumor inside Brain"]}* Closing connection 0

```

Figure 3.9: KF – Serving Prediction Output

- Run the following command in Python from the serving folder.

Now we create some sample data to predict the results from the preceding URL. To create the sample data, the code is as follows:

```

import json
import base64
import requests
samples=[]
NUM_SAMPLES=4
for index in range(NUM_SAMPLES):
    with open("Brain_{0}.jpg".format(index + 1), "rb") as
        image_file:
        encoded_bytes = base64.b64encode(image_file.read())
        # result: string (in utf-8)
        encoded_string = encoded_bytes.decode('utf-8')
        samples.append(encoded_string)

    # prepare test data
data = json.dumps({"instances": samples})
data_read = json.loads(data)
with open('data.json', 'w') as out:
    json.dump(data_read, out)

%%bash
gcloud container clusters get-credentials <CLUSTER_NAME> --
zone us-east1-d --project <PROJECT_ID>

```

```

CLUSTER_IP=$(kubectl -n istio-system get service kferving-
ingressgateway -o
jsonpath='{.status.loadBalancer.ingress[0].ip}')
echo $CLUSTER_IP
MODEL_NAME="kferving-braintumor"
#Replace the cluster IP
cluster_ip = "COPY YOUR IP FROM ABOVE"
headers={"Host": "
{0}.kubeflow.example.com".format(MODEL_NAME),"Content-Type":"
application/json"}
response =
requests.post("http://{0}/v1/models/{1}:predict".format(cluster_ip, MODEL_NAME), data = data,headers = headers)
print(response.json())

{&#39;predictions&#39;: [&#39;No tumor inside Brain&#39;, &#39;Tumor inside Brain&#39;, &#39;
Tumor inside Brain&#39;, &#39;Tumor inside Brain&#39;]}

```

Figure 3.10: Prediction output

In the preceding screenshot, we can see our prediction output of Batch predictions.

[3.6 Building the pipeline end to end](#)

Now, let's see how we will build the Pipeline, and run this platform in Kubeflow Notebook server.

Open the URL: GCP Kubernetes > Service & ingress > Click the URL

As shown in [Chapter 1, Introduction to Kubeflow & Kubernetes Cloud Architecture](#), already in *section 1.6*, we have created a Jupyter notebook that we will be using.

Now, paste the following code and run the pipeline; before that, replace the PROJECT_ID and bucket name from the following code and it will dump a zip file:

```

import kfp.dsl as dsl
import yaml
from kubernetes import client as k8s

```

```
import kfp.gcp as gcp
from kfp import components
from string import Template
import json
from kubernetes import client as k8s_client

@dsl.pipeline(
    name='',
    description='End to End pipeline for Tensorflow Brain MRI '
)

def brain_tensorflow_pipeline(
    dataextraction_step_image="gcr.io/<PROJECT_ID>/brain_tumor_scan1/step1_download_data:v1",
    dataprocessing_step_image="gcr.io/<PROJECT_ID>/brain_tumor_scan4/step2_dataprocessing:v1",
    trainmodel_step_image="gcr.io/<PROJECT_ID>/brain_tumor_scan1/step3_training_model:v1",
    evaluator_step_image="gcr.io/<PROJECT_ID>/brain_tumor_scan1/step4_evaluation_model:v1",
    root="/mnt/", data_file="/mnt/BrainScan_Data/",
    kaggle_api_data="navoneel/brain-mri-images-for-brain-tumor-detection",
    train_file='/mnt/training.data', test_file='/mnt/test.data',
    validation_file="/mnt/validation.data", label="/mnt/labels.data"
    ,
    activation="sigmoid", image_size=224, train_target="/mnt/training_target.data",
    test_target="/mnt/testtarget.data", validation_target="/mnt/validationtarget.data",
    epochs=10, learning_rate=.001, shuffle_size=1000, tensorboard_logs="/mnt/logs/",
    tensorboard_gcs_logs="gs://<BUCKET_NAME>/brain/logs",
    model_output_base_path="/mnt/saved_model", gcs_path="gs://<BUCKET_NAME>/brain/model",
```

```

gcs_path_confusion="gs://<BUCKET_NAME>/brain",
mode="gcs", probability=0.5,
serving_name="kfserving-
braintumor", serving_namespace="kubeflow",
image="gcr.io/<PROJECT_ID>/brain_tumor_scan/kf_serving_braintes
t:v1"):

"""
Pipeline
"""

# PVC : PersistentVolumeClaim volume
vop = dsl.VolumeOp(
    name='my-pvc', resource_name="my-
    pvc", modes=dsl.VOLUME_MODE_RWO, size="1Gi")

# data extraction
data_extraction_step = dsl.ContainerOp(
    name='data_extraction', image=dataextraction_step_image, command
    ="python",
    arguments=[
        "/app/dataextract.py",
        "--root", root,
        "--data-file", data_file,
        "--kaggle-api-data", kaggle_api_data,
    ], pvolumes={"/mnt": vop.volume}).apply(gcp.use_gcp_secret("user-gcp-sa"))

# processing
data_processing_step = dsl.ContainerOp(
    name='data_processing', image=dataprocessing_step_image, command
    ="python",
    arguments=[
        "/app/preprocessing.py",
        "--train-file", train_file,
        "--test-file", test_file,
        "--validation-file", validation_file,
        "--root", root,
        "--image-size", image_size,
    ]
)

```

```

"--train-target", train_target,
"--test-target", test_target,
"--validation-target", validation_target,
"--label", label], pvcolumes={"/mnt":  

    data_extraction_step.pvolume}  

).apply(gcp.use_gcp_secret("user-gcp-sa"))  

#trainmodel  

train_model_step = dsl.ContainerOp(  

    name='train_model', image=trainmodel_step_image,  

    command="python",  

    arguments=[  

        "/app/train.py",  

        "--train-file", train_file,  

        "--test-file", test_file,  

        "--label", label,  

        "--activation", activation,  

        "--validation-file", validation_file,  

        "--train-target", train_target,  

        "--test-target", test_target,  

        "--validation-target", validation_target,  

        "--epochs", epochs,  

        "--image-size", image_size,  

        "--learning-rate", learning_rate,  

        "--tensorboard-logs", tensorboard_logs,  

        "--tensorboard-gcs-logs", tensorboard_gcs_logs,  

        "--model-output-base-path", model_output_base_path,  

        "--gcs-path", gcs_path,  

        "--mode", mode,  

    ], file_outputs={"mlpipeline-ui-metadata": "/mlpipeline-ui-  

    metadata.json"},  

    pvcolumes={"/mnt":  

    data_processing_step.pvolume}).apply(gcp.use_gcp_secret("user-  

    gcp-sa"))  

#evaluation  

evaluation_model_step = dsl.ContainerOp(  


```

```
name='evaluation_model',image=evaluator_step_image,command="python",
arguments=[  
    "/app/evaluator.py",  
    "--test-file", test_file,  
    "--test-target", test_target,  
    "--probability", probability,  
    "--model-output-base-path", model_output_base_path,  
    "--gcs-path", gcs_path,  
    "--label", label,  
    "--gcs-path-confusion", gcs_path_confusion,  
],file_outputs={"mlpipeline-metrics":"/mlpipeline-  
metrics.json","mlpipeline-ui-metadata": "/mlpipeline-ui-  
metadata.json"},  
pvolumes={"/mnt":  
train_model_step.pvolume}).apply(gcp.use_gcp_secret("user-gcp-  
sa"))  
kfserving_template = Template("""{  
    "apiVersion": "serving.kubeflow.org/v1alpha2",  
    "kind": "InferenceService",  
    "metadata": {  
        "labels": {  
            "controller-tools.k8s.io": "1.0"  
        },  
        "name": "$name",  
        "namespace": "$namespace"  
    },  
    "spec": {  
        "default": {  
            "predictor": {  
                "custom": {  
                    "container": {  
                        "image": "$image"  
                    }  
                }  
            }  
        }  
    }  
}"""
```

```

        }
    }
}
}""")

kfservingjson = kfserving_template.substitute({'name':
str(serving_name),
'namespace': str(serving_namespace),
'image': str(image)})

kfservingdeployment = json.loads(kfservingjson)

serve = dsl.ResourceOp(
    name="serve", k8s_resource=kfservingdeployment,
    action="apply", success_condition="status.url")
serve.after(evaluation_model_step)

if __name__ == '__main__':
    import kfp.compiler as compiler
    pipeline_func = brain_tensorflow_pipeline
    pipeline_filename = pipeline_func.__name__ + '.pipeline.yaml'
    compiler.Compiler().compile(pipeline_func, pipeline_filename)

```

So, let's break the pipeline code as follows:

- Pipelines are expected to include a `@dsl.pipeline` decorator to provide metadata about the pipeline.
- The pipeline is defined in the `brain_tensorflow_pipeline` function. It includes a number of arguments, which are exposed in the Kubeflow Pipelines UI when creating a new Run. Although passed as strings, these arguments are of type `kfp.dsl.PipelineParam`.
- Each individual block defines one component like ‘train’, ‘evaluation’, etc. A component is made up of a `kfp.dsl.ContainerOp` object with the container path and a name specified. The container image used is defined as Dockerfile which we have created.
- After defining the train component, we also set a number of environment variables for the training script.
- At the bottom of the script is the main function. This is used to compile the pipeline when the script is run; then the `.after` method will trigger

the pipeline one after the other.

Next, we will create an experiment; under that, we can create multiple runs of a pipeline. The following code is for creating the experiment:

```
EXPERIMENT_NAME = 'Brain_experiment'
client = kfp.Client()
try:
    experiment =
        client.get_experiment(experiment_name=EXPERIMENT_NAME)
except:
    experiment = client.create_experiment(EXPERIMENT_NAME)
print(experiment)
```

This following snippet will create a run for the zip that we had dumped in that location:

```
arguments = []
run_name = pipeline_func.__name__ + 'heart_run'
run_result = client.run_pipeline(experiment.id, run_name,
pipeline_filename, arguments)
print(experiment.id)
print(run_name)
print(pipeline_filename)
print(arguments)
```

Click on the run link once the pipeline is ready. The following is how our pipeline training looks:

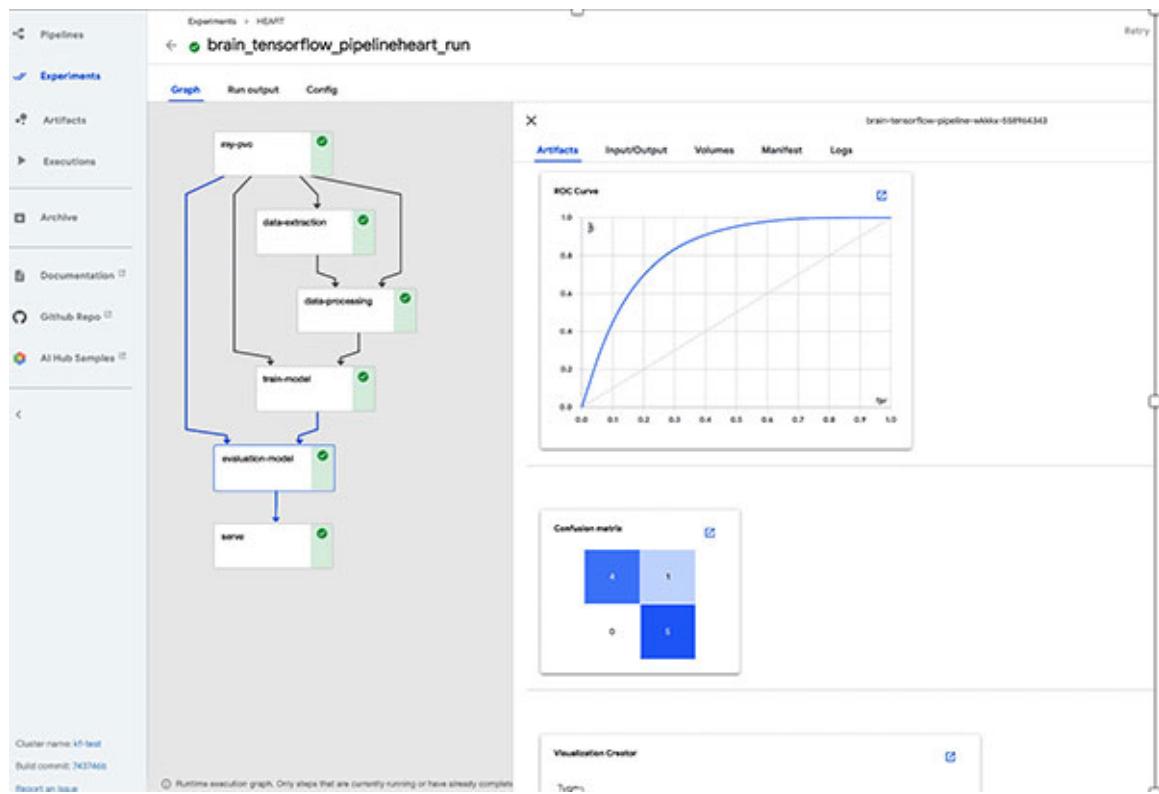


Figure 3.11: Pipeline Kubeflow e2e

The following image is the pipeline which we have created, and the Python visualizations, ROC Curve & Confusion Matrix & Tensorboard:

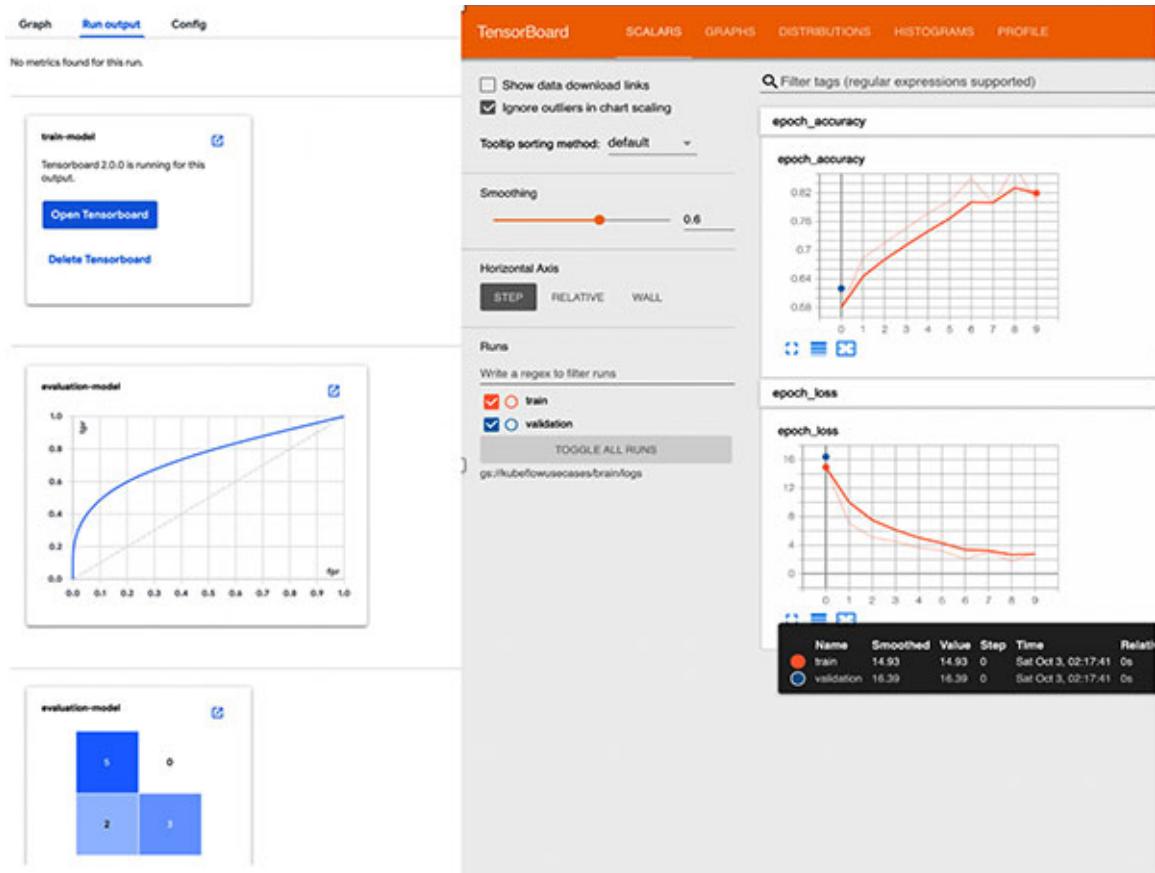


Figure 3.12: Pipeline Visualization

Another way to run the pipeline from UI is as follows:

- Create a new experiment button after clicking on that it will redirect to a new screen for experiment creation page.

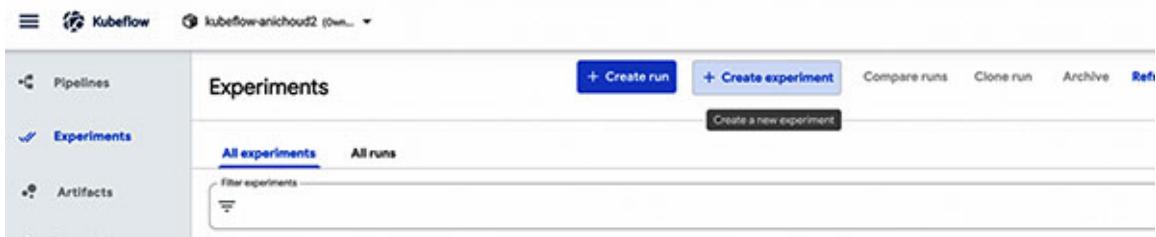


Figure 3.13: Create experiment

- Provide an experiment name.



Figure 3.14: Create experiment and provide name

- Click on **Skip this step**.

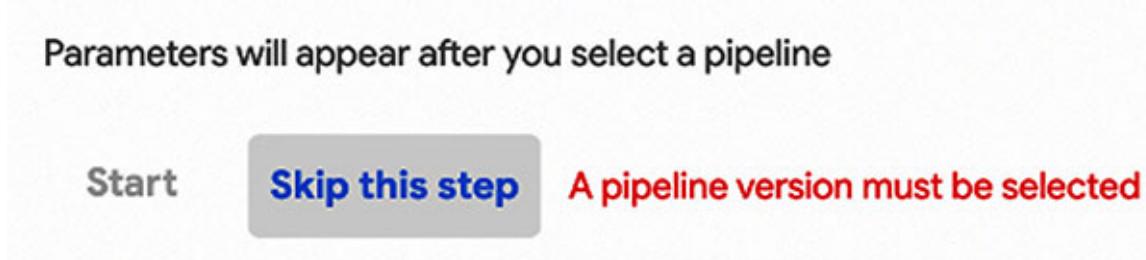


Figure 3.15: Create experiment skip step

- Now, the experiment is ready; now click on **Run**.

| Experiments | | + Create run | + Create experiment | Compare runs | Clone run | Archive | Refin | | |
|------------------------------------|-------------|--------------------------|---------------------|----------------------------------|------------|---------|-------|--|--|
| All experiments | | All runs | | Create a new run | | | | | |
| Filter experiments | | | | | | | | | |
| Experiment name | Description | Last 5 runs | | | | | | | |
| Brain_experiment | | | | | | | | | |
| Run name | Status | Duration | Pipeline Version | Recurring Run | Start time | | | | |

Figure 3.16: Create run

- Next, upload the zip which we have created during the pipeline building earlier.



[←](#) Upload Pipeline or Pipeline version

Create a new pipeline Create a new pipeline version under an existing pipeline

Upload pipeline with the specified package.

Pipeline Name *

Pipeline Description *

Choose a pipeline package file from your computer, and give the pipeline a unique name.
You can also drag and drop the file here.

For expected file format, refer to [Compile Pipeline Documentation](#).

Upload a file

File *

[Choose file](#)

Import by url

Package Url

Code Source (optional)

Create

[Cancel](#)

Must specify either package url or file in .yaml, .zip, or .tar.gz

Figure 3.17: Upload Pipeline tar or yaml file

- Now, create a run of that Pipeline, which you have uploaded, and choose the experiment which we have created earlier.

Experiments

[← Start a run](#)

Run details

Pipeline * [Choose](#)

Pipeline Version * [Choose](#)

Run name * [Choose](#)

Description (optional) [Choose](#)

This run will be associated with the following experiment

Experiment * [Choose](#)

Brain_experiment [Choose](#)

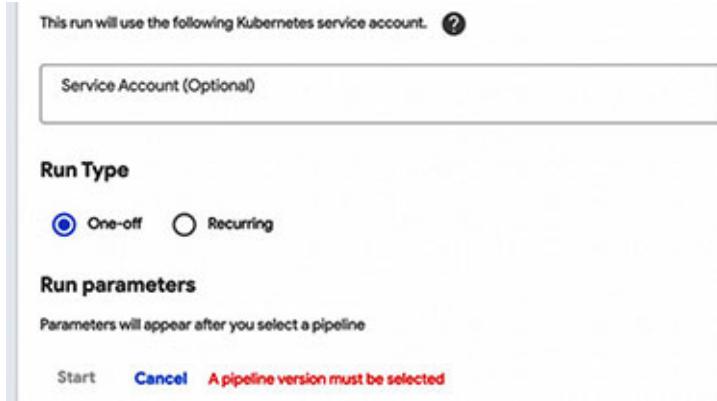


Figure 3.18: Start Run with choosing experiment

Next, let's see how to autoscale the KF-Serving endpoint and monitor that in the Grafana Dashboard.

3.7. Auto-Scaling of the Serving Endpoint

One of the main features of Knative is the automatic scaling of the replicas for an application to closely match the incoming demand, including the scaling applications to zero, if no traffic is being received. Knative Serving enables this by default, using the **Knative Pod Autoscaler (KPA)**. The Autoscaler component watches the traffic flow to the application, and scales the replicas up or down, based on the configured metrics.

Deploy the kf-serving Knative Service:

`custom_auto_scale.yaml:`

```

apiVersion: serving.kubeflow.org/v1alpha2
kind: InferenceService
metadata:
  annotations:
    sidecar.istio.io/inject: "false"
    autoscaling.knative.dev/target: "10"
  labels:
    controller-tools.k8s.io: "1.0"
  name: <SERVING_MODEL_NAME>
  namespace: kubeflow
spec:
  default:
    predictor:
      minReplicas: 1
      custom:
        container:
          image:
            gcr.io/<PROJECT_ID>/brain_tumor_scan/kf_serving_braintest:
            v1
          imagePullPolicy: Always
          name: user-container
        imagePullSecrets:
          - name: user-gcp-sa

```

- Connect to the GCP cluster by using the following command:

```
gcloud container clusters get-credentials <$ClusterName> --zone <$ZONE> --project <$PROJECTID>
```

- Create the inference service by deploying it in the cluster:

```
kubectl apply -f custom_autoscale.yaml
```

- Check the inference service. Try it after some interval to check if it has been created:

```
kubectl get inferenceservice -n kubeflow
```



Figure 3.19: KF – Serving autoscale Inferenceservice ready

We can use any load testing tool to simulate the load. Here, we will be using the Hey (<https://github.com/rakyll/hey>) tool to test the autoscaling:

```
```bash
on macOS
brew install hey
```
```

We have already installed Prometheus in [Chapter 1, Introduction to Kubeflow & Kubernetes Cloud Architecture](#), section 1.6.6; if you haven't, please refer to that chapter.

- Send 30 seconds of traffic, maintaining 50 in-flight requests:

```
```bash
MODEL_NAME=kf-serving-braintumor
HOST=$(kubectl get inferenceservice -n kubeflow $MODEL_NAME -
o jsonpath='{.status.url}' | cut -d "/" -f 3)
INPUT_PATH=@./data.json
CLUSTER_IP=$(kubectl -n istio-system get service kf-serving-
ingressgateway -o
jsonpath='{.status.loadBalancer.ingress[0].ip}')
hey -z 30s -c 100 -m POST -H "Host: ${HOST}" -d $INPUT_PATH
http://${CLUSTER_IP}/v1/models/${MODEL_NAME}:predict
```
```

```

Summary:
  Total:      30.9381 secs
  Slowest:    2.4160 secs
  Fastest:    0.3205 secs
  Average:    0.6304 secs
  Requests/sec: 155.6978

Response time histogram:
  0.321 [1]
  0.530 [928] ━━━━━━
  0.740 [3435] ━━━━━━━━━━
  0.949 [326] ━━
  1.159 [45] ─
  1.368 [25]
  1.578 [0]
  1.787 [30]
  1.997 [21]
  2.206 [4]
  2.416 [2]

Latency distribution:
  10% in 0.4698 secs
  25% in 0.5516 secs
  50% in 0.6244 secs
  75% in 0.6655 secs
  90% in 0.7328 secs
  95% in 0.8477 secs
  99% in 1.6786 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0184 secs, 0.3205 secs, 2.4160 secs
  DNS-lookup:   0.0000 secs, 0.0000 secs, 0.0000 secs
  req write:    0.0000 secs, 0.0000 secs, 0.0004 secs
  resp wait:   0.3716 secs, 0.3189 secs, 2.2550 secs
  resp read:   0.0000 secs, 0.0000 secs, 0.0012 secs

Status code distribution:
  [404] 4817 responses

```

Figure 3.20: Hey Traffic Report

- Check out the pods that it was running.

```

```bash
kubectl get pods -n kubeflow | grep kferving-braintumor
```

```

| | | | | |
|--|-----|---------|---|-----|
| kferving-braintumor-predictor-default-xk9np-deployment-582ct7n | 1/2 | Running | 0 | 53s |
| kferving-braintumor-predictor-default-xk9np-deployment-588lp7k | 1/2 | Running | 0 | 33s |
| kferving-braintumor-predictor-default-xk9np-deployment-589mw7n | 1/2 | Running | 0 | 53s |
| kferving-braintumor-predictor-default-xk9np-deployment-58kc4jv | 1/2 | Running | 0 | 54s |
| kferving-braintumor-predictor-default-xk9np-deployment-58qqt6m | 1/2 | Running | 0 | 31s |
| kferving-braintumor-predictor-default-xk9np-deployment-58rq4qz | 1/2 | Running | 0 | 50s |
| kferving-braintumor-predictor-default-xk9np-deployment-58szgg6 | 1/2 | Running | 0 | 27s |
| kferving-braintumor-predictor-default-xk9np-deployment-58vk4jm | 1/2 | Running | 0 | 50s |
| kferving-braintumor-predictor-default-xk9np-deployment-58vnjb5 | 1/2 | Running | 0 | 28s |
| kferving-braintumor-predictor-default-xk9np-deployment-58wx6cw | 1/2 | Running | 0 | 31s |

Figure 3.21: Pods of autoscale

- Open the Grafana dashboard.

View the Knative Serving Scaling dashboards:

```

```bash
use port-forwarding
kubectl port-forward --namespace knative-monitoring $(kubectl
get pod --namespace knative-monitoring --
selector="app=grafana" --output
jsonpath='{.items[0].metadata.name}') 8080:3000
```

```

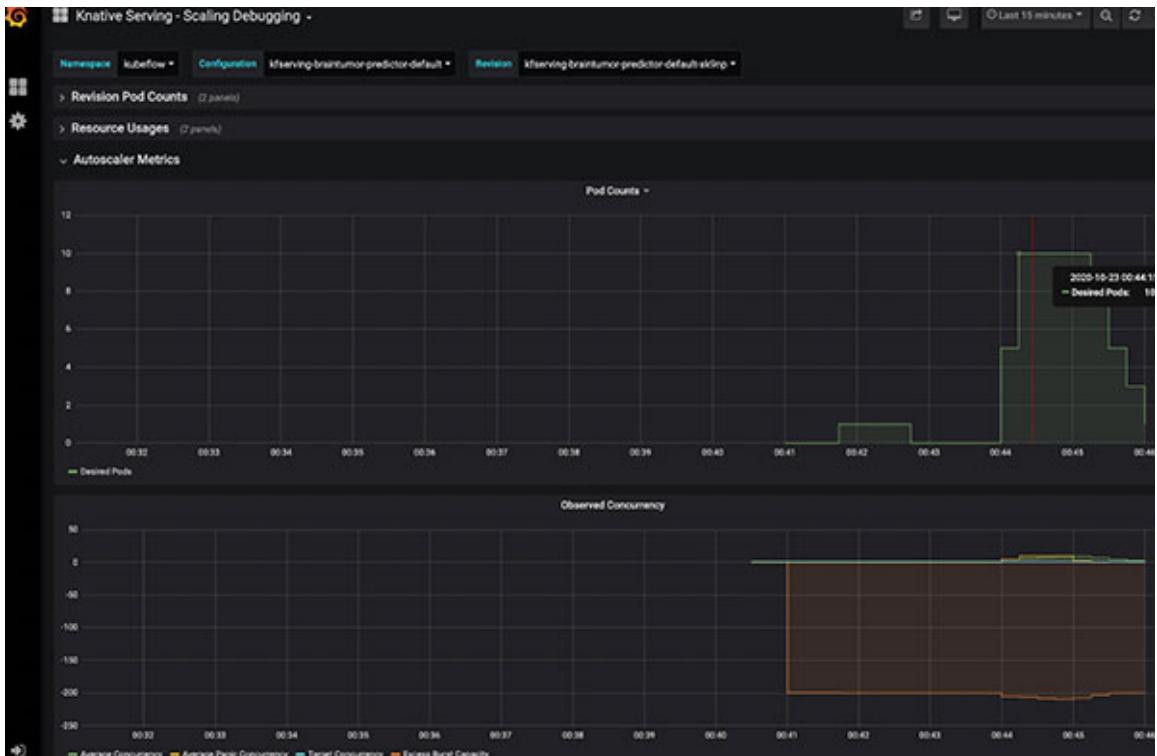


Figure 3.22: Grafana Knative Dashboard autoscale

As we can see, when we sent the traffic of 100 in-flight requests for 30 seconds, it created 10 pods.

Auto-Scaling explanation: how it works?

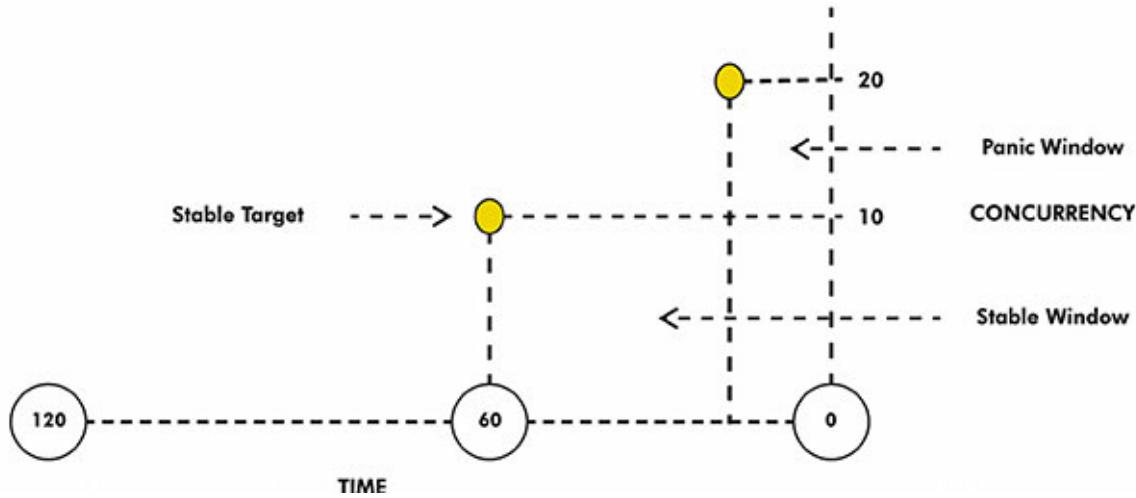


Figure 3.23: Auto-scaling Working Concept

Algorithm

Knative Serving autoscaling is based on the average number of in-flight requests per pod (concurrency). The system has a default target concurrency of 100 (search for the container-concurrency-target-default), but we used 10 for our service.

We loaded the service with 100 concurrent requests, so the autoscaler created 5 pods.

$$(100 \text{ concurrent requests} / \text{target of } 10 = 10 \text{ pods})$$

Panic

The autoscaler calculates the average concurrency over a 60-second window, so it takes a minute for the system to stabilize at the desired level of concurrency. However, the autoscaler also calculates a 6-second panic window and will enter the panic mode if that window reached 2x the target concurrency. In the panic mode, the autoscaler operates on the shorter, more sensitive panic window. Once the panic conditions are no longer met for 60 seconds, the autoscaler will return to the initial 60 second stable window.

3.8 Conclusion

In this chapter, we learned how to build end-to-end Kubeflow Orchestrator Pipeline for a TensorFlow CNN Model and how we Dockerized each component and built it in Google Platform.

Then, we saw how to build the pipeline with the kfp library package and triggered the pipeline from the Kubflow Dashboard and built a batch prediction serving. Now, we have deployed Kubeflow on the Kubernetes Platform and learned how to trigger the pipeline from the Notebook. We have also deployed the model in the Kubernetes cluster with KF serving and Monitored the auto-scaling in Grafana Dashboard.

In this chapter, we have learned how to leverage the power of Google Cloud Platform, and use our Devops knowledge with Machine Learning to become an Mlops.

3.9 Reference

- <https://knative.dev/docs/serving/autoscaling/autoscaling-concepts/>
- <https://v1-1-branch.kubeflow.org/docs/>
- <https://v1-1-branch.kubeflow.org/docs/gke/>
- <https://v1-1-branch.kubeflow.org/docs/gke/monitoring/>
- <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>

CHAPTER 4

Building TFX Pipeline

In this chapter, we will build an end-to-end structured data classification model and make it ready for production with the help of TFX, and serve the model outputs with TF serving to get the prediction. We also be building the TensorFlow ecosystem Modelling, and visualizing the evaluation with Tensorboard and Fairness. Then, we will learn about the various TFX Components like TFT, TFMA, TFDV, and so on. Later on, we will create a Kubeflow Pipeline in Google cloud.

Structure

In this chapter, we will cover the following topics:

- Problem statement
- Architecture of TFX components
- TFX environment setup
- TFX pipeline
- Serve the model with TF serving
- Building Kubeflow Pipeline Orchestrator

Objective

After studying this chapter, we will be able to do the following:

- Understand the complete BERT Architecture Model, and it's tokenization and pre-processing.
- Evaluate the BERT Base Model and creation of Framework for sentiment Analysis.
- Build the TFX Pipeline Components, which provides a configuration framework and shared library to integrate the common components needed to define, launch, and monitor your machine learning system.

- Pre-process the training data for your BERT model training and validation.
- Analyse and review the trained and tuned models, deploying the best model which will be pushed by the pusher component.

4.1 Problem statement

In this chapter, we will be using Taxi Trips dataset. This is a dataset for the binary sentiment classification, containing substantially more data than the previous benchmark datasets. Also, we will be predicting the tips. Here, we will build a classification model and a Kubeflow pipeline in TFX with various components, so that it will be ready for production. Also, we will be using the TF-Serving.

| | |
|-------------|---|
| NOTE | Rest all the imports I have showed in my Google Colab, which I gave hyperlink of Github Account of this chapter. Note: Package Python 3.x |
| CODE | https://github.com/bpbpublications/Continuous-Machine-Learning-with-Kubeflow/tree/main/Chapter4 |

4.2 Architecture of TFX components

The machine learning pipelines can become very complicated and consume a lot of overhead to manage the task dependencies. At the same time, the machine learning pipelines can include a variety of tasks, including the tasks for data validation, pre-processing, model training, and any post-training tasks. The connections between the tasks are often brittle, and can cause the pipelines to fail. Having brittle connections ultimately means that the production models will be updated infrequently; the data scientists or machine learning engineers loathe updating the stale models. Pipelines also require well-managed distributed processing, which is why the TFX leverages Apache Beam. This is especially true for large workloads.

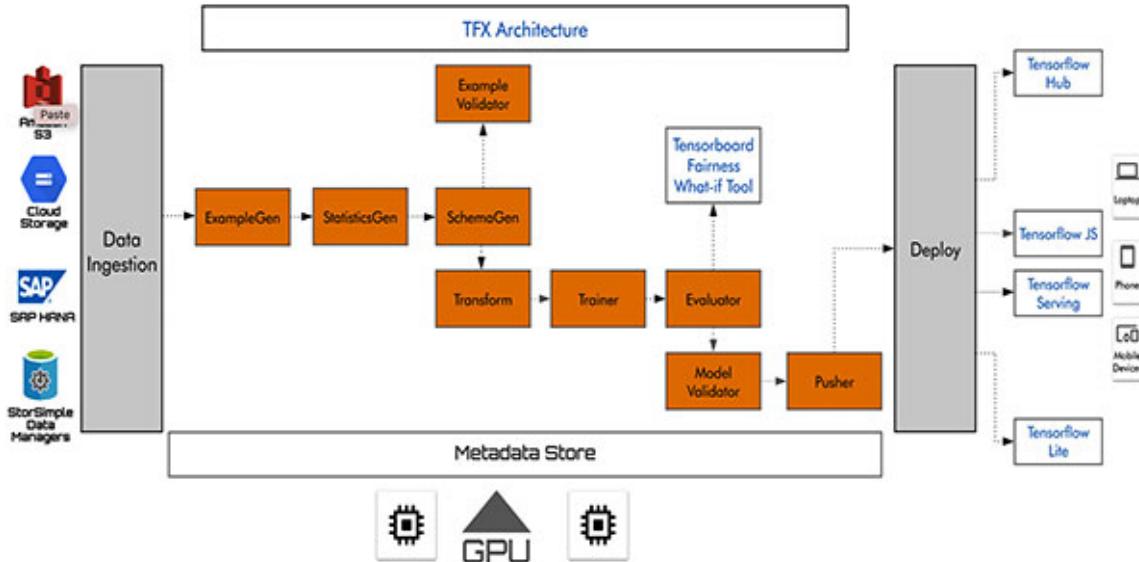


Figure 4.1: TFX Architecture

A TFX pipeline is a sequence of components that implement a machine learning pipeline, which is specifically designed for scalable, high-performance machine learning tasks.

- **ExampleGen** is the initial input component of a pipeline that ingests and optionally splits the input dataset.
- **StatisticsGen** calculates the statistics for the dataset.
- **SchemaGen** examines the statistics and creates a data schema.
- **ExampleValidator** looks for the anomalies and missing values in the dataset.
- **Transform** performs feature engineering on the dataset.
- **Trainer** trains the model.
- **Evaluator** tunes the hyper parameters of the model.
- **Evaluator** performs deep analysis of the training results and helps you validate your exported models, ensuring that they are "*good enough*" to be pushed to production.
- **Pusher** deploys the model on a serving infrastructure.

Brief of TFX Components: A component handles a more complex process than just the execution of a single task. All machine learning pipeline components read from a Channel to get the input artifacts from the metadata

store. The data is then loaded from the path provided by the metadata store and processed. The output of the component, the processed data, is then provided to the next pipeline components. The generic internals of a component are always as follows:

- Receive some inputs
- Perform an action
- Store the final result

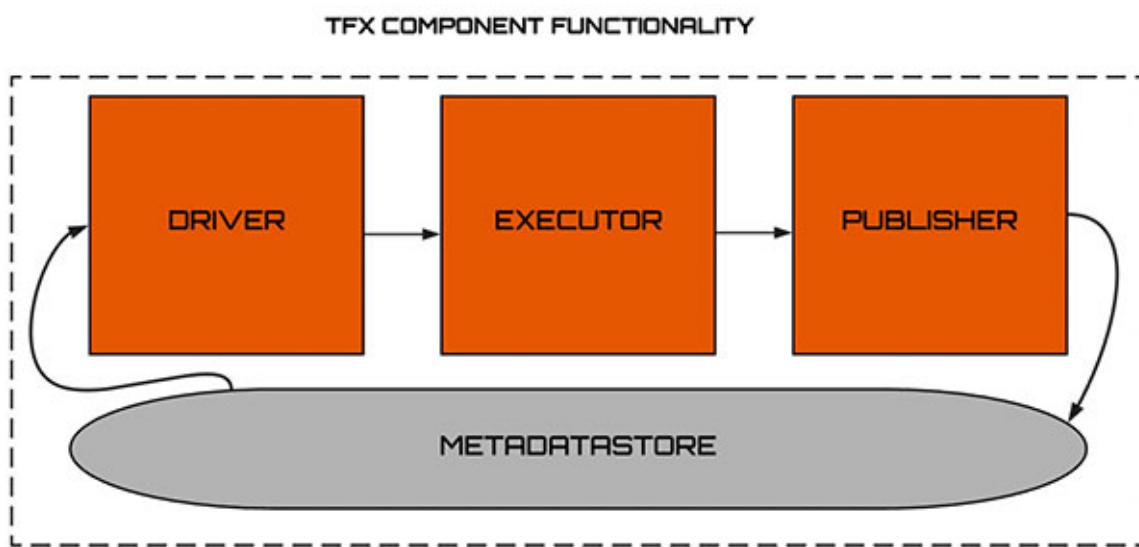


Figure 4.2: TFX Component Functionality

In TFX terms, the three internal parts of the component are called Driver, Executor, and Publisher. The driver handles the querying of the metadata store; the executor performs the actions of the components; and the publisher manages the saving of the output metadata in the MetadataStore. The driver and the publisher aren't moving any data, but instead, they read and write the references from the MetadataStore.

4.3 TFX environment setup

Now, we will install each and every dependency with respect to our projects.

```
try:  
    import colab  
    !pip install --upgrade pip
```

```
except:  
    pass  
!pip install -q -U --use-feature=2020-resolver tfx
```

Next, we have to restart the kernel and we will import each and every dependency.

```
from tfx.components.base import executor_spec  
from tfx.components.trainer.executor import GenericExecutor  
from tfx.dsl.experimental import latest_blessed_model_resolver  
from tfx.proto import evaluator_pb2, example_gen_pb2, pusher_pb2,  
    trainer_pb2  
from tfx.types import Channel  
from tfx.types.standard_artifacts import Model, ModelBlessing  
from tfx.utils.dsl_utils import external_input  
from tfx.components import (Evaluator, ExampleValidator,  
    ImportExampleGen, ModelValidator, Pusher, ResolverNode,  
    SchemaGen, StatisticsGen, Trainer, Transform, Tuner)  
from tfx.orchestration import (metadata, pipeline)  
from  
tfx.orchestration.experimental.interactive.interactive_context  
import InteractiveContext  
from tfx.proto import (pusher_pb2, trainer_pb2)  
from tfx.proto.evaluator_pb2 import SingleSlicingSpec  
from tfx.utils.dsl_utils import external_input  
from tfx.types.standard_artifacts import (Model, ModelBlessing)  
tf.get_logger().propagate = False  
pp = pprint.PrettyPrinter()  
%load_ext  
tfx.orchestration.experimental.interactive.notebook_extensions.ski  
p
```

Setting Directory and Download Data

Steps for Setting the Root directory are as follows:

Step 1: This is the root directory for your TFX pip package installation.

Step 2: This is the directory containing the TFX Chicago Taxi Pipeline example.

Step 3: This is the path where your model will be pushed for serving.

Step 4: Set up logging.

```
def setting_directory():
    #step1
    _tfx_root = tfx.__path__[0]
    print(_tfx_root)
    #step2
    _taxi_root = os.path.join(_tfx_root,
    'examples/chicago_taxi_pipeline')
    print(_taxi_root)
    #step3
    _serving_model_dir = os.path.join(tempfile.mkdtemp(),
    'serving_model/taxi_simple')
    print(_serving_model_dir)
    #step4
    absl.logging.set_verbosity(absl.logging.INFO)
    return _tfx_root,_taxi_root,_serving_model_dir
```

Now, we call the preceding function to set the directory:

```
_tfx_root,_taxi_root,_serving_model_dir=setting_directory()

def download_data():
    _data_root = tempfile.mkdtemp(prefix='tfx-data')
    DATA_PATH =
    'https://raw.githubusercontent.com/tensorflow/tfx/master/tfx/ex
    amples/chicago_taxi_pipeline/data/simple/data.csv'
    _data_filepath = os.path.join(_data_root, "data.csv")
    urllib.request.urlretrieve(DATA_PATH, _data_filepath)
    return _data_root,_data_filepath
```

Now let's call the preceding function to download the data in the directory:

```
_data_root,_data_filepath = download_data()
```

Let's check a glimpse on our dataset:

| | pickup_community_area | fare | trip_start_month | trip_start_hour | trip_start_day | trip_start_timestamp | pickup_latitude | pickup_longitude | dropoff_latitude | dropoff_longitude | trip_miles | pickup_census tract | drop |
|---|-----------------------|-------|------------------|-----------------|----------------|----------------------|-----------------|------------------|------------------|-------------------|------------|---------------------|------|
| 0 | NaN | 12.45 | 5 | 19 | 6 | 1400269500 | NaN | NaN | NaN | NaN | 0.0 | NaN | |
| 1 | NaN | 0.00 | 3 | 19 | 5 | 1362083700 | NaN | NaN | NaN | NaN | 0.0 | NaN | |
| 2 | 80.0 | 27.06 | 10 | 2 | 3 | 13605083700 | 41.894180 | -87.646788 | NaN | NaN | 12.4 | NaN | |
| 3 | 10.0 | 5.85 | 10 | 1 | 2 | 1362219000 | 41.985015 | -87.604532 | NaN | NaN | 0.0 | NaN | |
| 4 | 14.0 | 16.66 | 8 | 7 | 5 | 13608097200 | 41.948069 | -87.721558 | NaN | NaN | 0.0 | NaN | |

Figure 4.3: Chicago Dataset

In the preceding screenshot, we can see the features in our dataset; we will be using that for the classification model building with TensorFlow.

4.4 TFX pipeline components

TFX provides several Python package libraries which will be used here to create the pipeline components.

In this section, we have created the Interactive Context using the default parameters, which will help to create the ephemeral ML Metadata database instance.

```
context = InteractiveContext()
```

```
[+] WARNING:absl:InteractiveContext pipeline_root argument not provided: using temporary directory /tmp/tfx-interactive-20
[+] WARNING:absl:InteractiveContext metadata_connection_config not provided: using SQLite ML Metadata database at /tmp/tf>
```

Figure 4.4: Initializing interactive Context

Next, we will build the pipeline.

4.4.1 ExampleGen

The ExampleGen component is usually at the start of a TFX pipeline. TFX examplegen will customize the train/eval split ratio, which the ExampleGen will output, set the **output_config**; for example, Gen component. Each Version within a Span can further be subdivided into multiple Splits. The most common use-case for splitting a Span is to split it into the training and eval data.

The **hash_buckets** were set in this example.

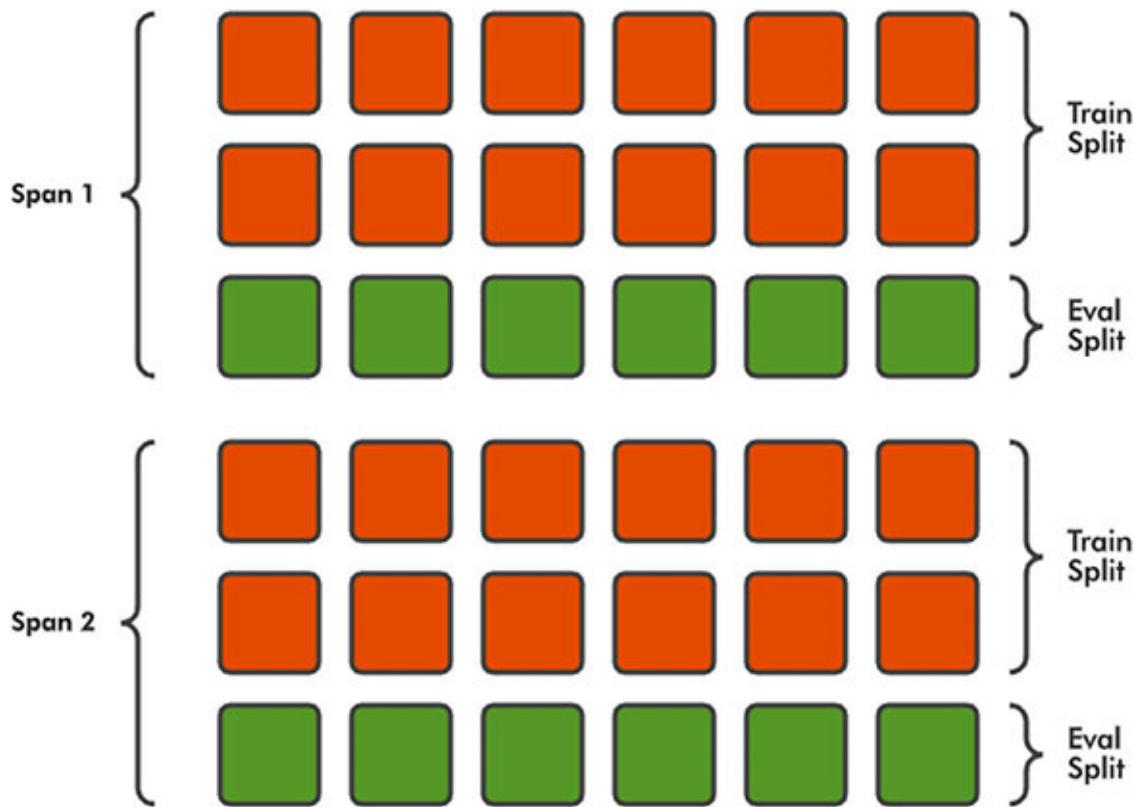


Figure 4.5: Example Gen

The preceding diagram is the example gen component, which will split the dataset in that manner. So, this is the architecture of the ExampleGen.

Component: ExampleGen

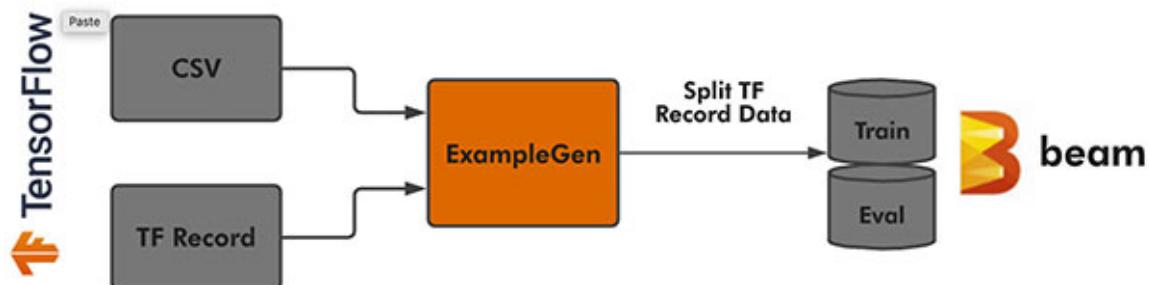


Figure 4.6: Example Gen Architecture

The ExampleGen TFX Pipeline component ingests the data into TFX pipelines.

```
def examplegen(_data_root):
```

```

example_gen = CsvExampleGen(input=external_input(_data_root))
return example_gen

```

Define the path where we downloaded the data:

```

example_gen= examplegen(_data_root)
context.run(example_gen)

```

```

▼ ExecutionResult at 0x7f11c58e5978
  .execution_id      1
  .component         ► CsvExampleGen at 0x7f11c58e5780
  .component.inputs  0
  .component.outputs ["examples"]
    ▼ Channel of type 'Examples' (1 artifact) at 0x7f11c58e5320
      .type_name Examples
      .artifacts [0] ▼ Artifact of type 'Examples' (uri: /tmp/tfx-interactive-2020-10-30T18_40_25.173616-vnjm_j02/CsvExampleGen/examples/1)
        .type          <class 'tfx.types.standard_artifacts.Examples'>
        .url           /tmp/tfx-interactive-2020-10-30T18_40_25.173616-vnjm_j02/CsvExampleGen/examples/1
        .span          0
        .split_names  ["train", "eval"]
        .version       0

```

Figure 4.7: Example Gen Output

The preceding screenshot is the output for the ExampleGen where we split the data into train and evaluation and stored the path as shown.

4.4.2 StatisticsGen

The StatisticsGen component computes the statistics over your dataset for the data analysis, as well as for use in the downstream components. It uses the **TensorFlow Data Validation** library.

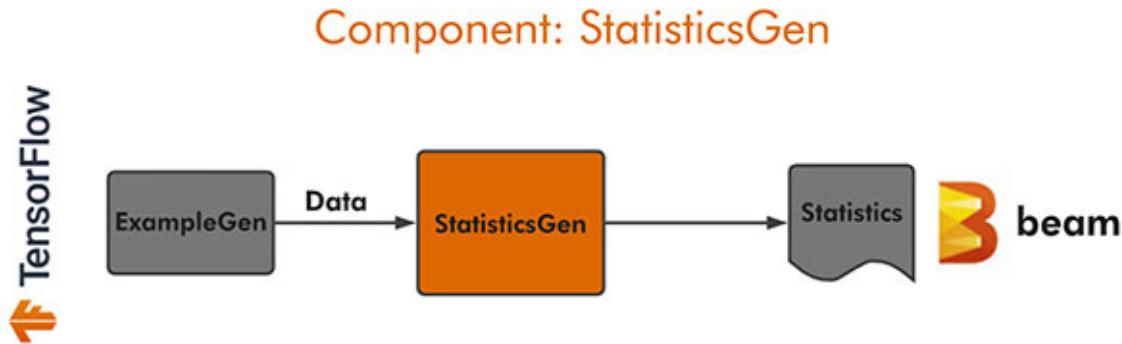


Figure 4.8: Statistics Gen Architecture

StatisticsGen takes as input, the dataset that we just ingested using ExampleGen:

```

def statisticsgen(example_gen):

```

```

statistics_gen =
StatisticsGen(examples=example_gen.outputs['examples'])
return statistics_gen

statistics_gen=statisticsgen(example_gen)
context.run(statistics_gen)

```

ExecutionResult at 0x7efbb2652748

- .execution_id 2
- .component ►StatisticsGen at 0x7efc29aebac8
- .component.inputs ['examples'] ►Channel of type 'Examples' (1 artifact) at 0x7efc0014cc50
- .component.outputs ['statistics'] ►Channel of type 'ExampleStatistics' (1 artifact) at 0x7efc13ee98d0

Figure 4.9: Statistics Gen Output

Visualize the outputted statistics of our training data and evaluation data:

```

%%skip_for_export
context.show(statistics_gen.outputs['statistics'])

```

The StatisticsGen will give us the distribution of all the features and stats of the categorical and numerical columns, from which we can analyse the skewness and missing value analysis.

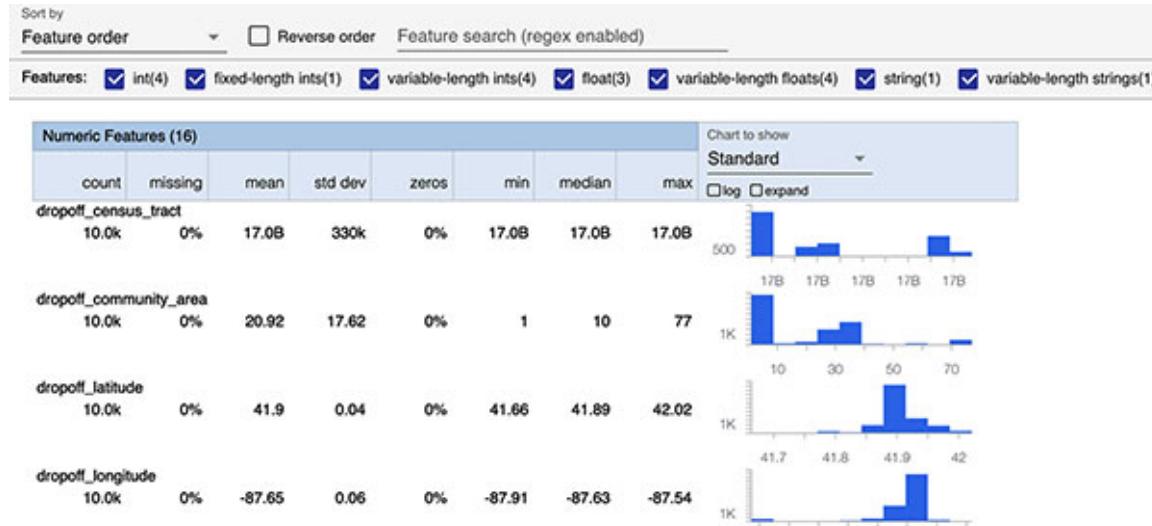


Figure 4.10: Statistics Gen Output Visualization

The StatisticsGen TFX pipeline component generates the feature statistics over both the training and the serving data, which can be used by the other pipeline

components. StatisticsGen uses Beam to scale to large datasets.

4.4.3 SchemaGen

The SchemaGen component generates a schema based on your data statistics. (A schema defines the expected bounds, types, and properties of the features in your dataset.) It also uses the TensorFlow Data Validation library.

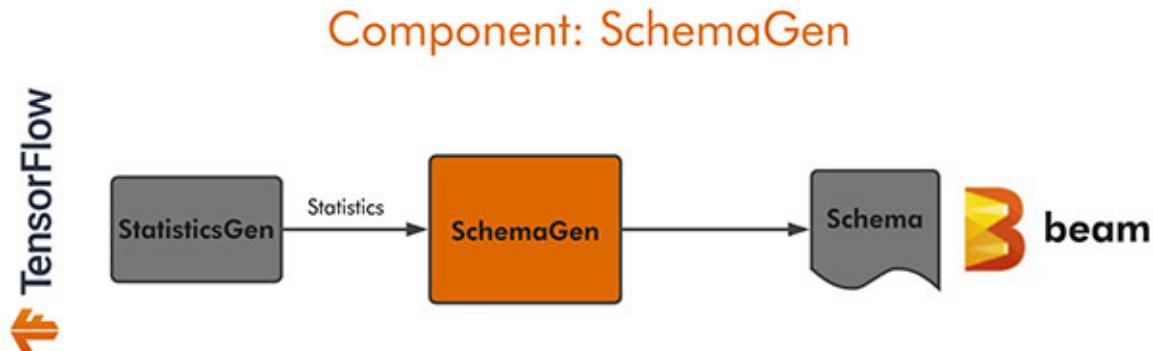


Figure 4.11: Schema Gen Architecture

SchemaGen will take as input, the statistics that we generated with StatisticsGen, looking at the training split by default.

```
def schemagen(statistics_gen):
    schema_gen =
        SchemaGen(statistics=statistics_gen.outputs['statistics'], infer
        _feature_shape=False)
    return schema_gen
```

Let's generate the schema table:

```
schema_gen=schemagen(statistics_gen)
%%skip_for_export
context.show(schema_gen.outputs['schema'])
```

Data schema for the type of input tensors is as follows:

| Feature name | Type | Presence | Valency | Domain |
|--------------------------|--------|----------|---------|----------------|
| 'company' | STRING | required | | 'company' |
| 'dropoff_census_tract' | INT | required | | - |
| 'dropoff_community_area' | INT | required | | - |
| 'dropoff_latitude' | FLOAT | required | | - |
| 'dropoff_longitude' | FLOAT | required | | - |
| 'fare' | FLOAT | required | single | - |
| 'payment_type' | STRING | required | single | 'payment_type' |
| 'pickup_census_tract' | INT | required | | - |
| 'pickup_community_area' | INT | required | | - |
| 'pickup_latitude' | FLOAT | required | | - |
| 'pickup_longitude' | FLOAT | required | | - |
| 'tips' | FLOAT | required | single | - |
| 'trip_miles' | FLOAT | required | single | - |
| 'trip_seconds' | INT | required | | - |
| 'trip_start_day' | INT | required | single | - |
| 'trip_start_hour' | INT | required | single | - |
| 'trip_start_month' | INT | required | single | - |
| 'trip_start_timestamp' | INT | required | single | - |

Figure 4.12: Schema Gen Output Visualization

So, schema is a part of **schema.proto**. It will specify the data types for the feature values, whether that feature has to be present in all the examples, value ranges, and other properties. The SchemaGen component will generate that schema by inferring the types, categories, and ranges from the training data automatically.

4.4.4 ExampleValidator

The **ExampleValidator** component detects the anomalies in your data, based on the expectations defined by the schema. It also uses the **TensorFlow Data Validation** library.

Component: ExampleValidator

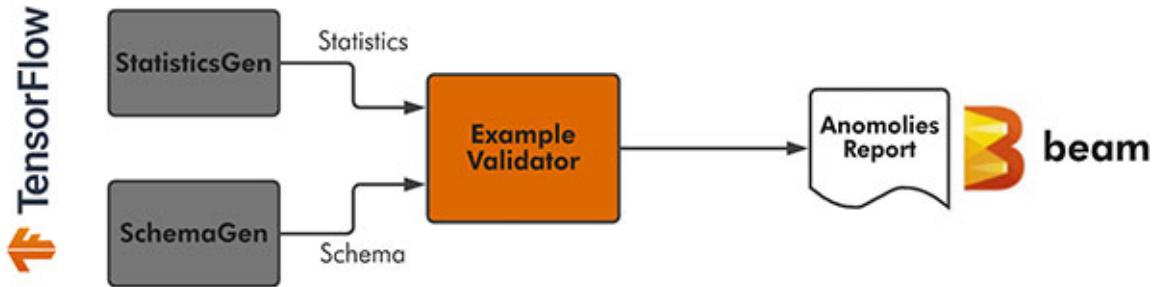


Figure 4.13: ExampleValidator Architecture

ExampleValidator will take as input, the statistics from StatisticsGen, and the schema from SchemaGen. By default, it compares the statistics from the evaluation split to the schema from the training split.

```
def examplevalidator(statistics_gen, schema_gen):
    example_validator =
        ExampleValidator(statistics=statistics_gen.outputs['statistics'],
                        schema=schema_gen.outputs['schema'])
    return example_validator
```

Let's call the function to check the anomaly report.

```
example_validator = examplevalidator(statistics_gen, schema_gen)
context.run(example_validator)
```

Visualize the anomalies table:

```
%%skip_for_export
context.show(example_validator.outputs['anomalies'])
```

Artifact at /tmp/tfx-interactive-2020-10-30T18_40_25.173616-vnjm_j02/ExampleValidator/anomalies/4

'train' split:

No anomalies found.

'eval' split:

No anomalies found.

This cell will be skipped during export to pipeline.

Figure 4.14: ExampleValidator Anomaly Table

So, the ExampleValidator pipeline component finds the anomalies in the training and serving data. It helps to detect the different anomaly classes in our data. Take a look at the following:

- It does a validity check that compares the data statistics against a schema, which will codify the expectations of a user.
- Next, it helps to detect a training-serving skew by checking the comparison between the training and serving data.
- It helps to detect the data drift by looking at a series of data.

4.4.5 Transform

The Transform component performs the feature engineering for both the training and the serving. The Transform component processes the data that we ingested into our pipeline, together with the earlier generated data set schema, and it outputs the following two artifacts:

- Pre-processed training and evaluation datasets in the TFRecords format. The produced datasets can be consumed downstream in a Trainer component of our pipeline.
- Exported preprocessing graph (with assets) which will be used when we'll export our machine learning model.

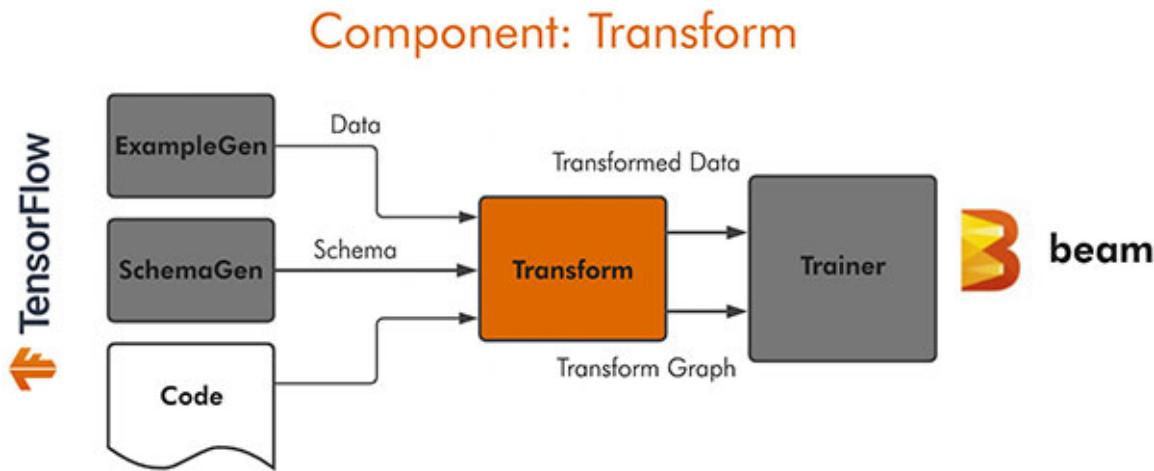


Figure 4.15: Transform Architecture

Steps for Transformation: The Transform is used to transform the numerical and categorical one-hot encoded features, bucketized features, and raw string representation data by completing the following steps:

Step 1: Categorical features are assumed to each have a maximum value in the dataset and dense float features, and the categorical is distributed in a list.

Step 2: Number of buckets used by `tf.transform` for encoding each feature.

Step 3: Number of vocabulary terms used for encoding `VOCAB_FEATURES` by `tf.transform`.

Step 4: Count of out-of-vocab buckets in which the unrecognized `VOCAB_FEATURES` are hashed.

Step 5: Designating the target Feature columns.

Step 6: It is a good practice to rename the features by appending a suffix to the feature name (for example, `_xf`). The suffix will help to distinguish whether the errors are originating from the input or output features and it prevents us from accidentally using a non-transformed feature in our actual model.

```
%%skip_for_export
%%writefile {_taxi_constants_module_file}

# Step1
MAX_CATEGORICAL FEATURE_VALUES = [24, 31, 12]
CATEGORICAL FEATURE_KEYS = ['trip_start_hour', 'trip_start_day',
'trip_start_month',
'pickup_census_tract', 'dropoff_census_tract',
'pickup_community_area', 'dropoff_community_area']
DENSE_FLOAT FEATURE_KEYS = ['trip_miles', 'fare', 'trip_seconds']

# Step2
FEATURE_BUCKET_COUNT = 10
BUCKET_FEATURE_KEYS = ['pickup_latitude', 'pickup_longitude',
'dropoff_latitude', 'dropoff_longitude']

# Step3
VOCAB_SIZE = 1000

# Step4
OOV_SIZE = 10
VOCAB_FEATURE_KEYS = ['payment_type', 'company']

# Step5
```

```

LABEL_KEY = 'tips'
FARE_KEY = 'fare'

#Step6
def transformed_name(key):
    return key + '_xf'

```

The Transform component from TFX in our pipeline, expects the transformation code to be provided in a separate Python file. The name of the module file can be set by the user (for example, in our case `taxi_transform.py`), but the entry point `preprocessing_fn()` needs to be contained in the module file and the function can't be renamed. Import the preceding Python code in the following transform file:

The Transformation code for the pre-processing steps and filling missing values are as follows:

Step 1: Importing the Features from the Python file.

Step 2: `tf.transform`'s callback function for preprocessing inputs of the artifacts data for training.

Step 3: TensorFlow Transform expects the transformation outputs to be dense, therefore we are using the following helper function to convert the sparse to dense features:

```

%%skip_for_export
%%writefile {_taxi_transform_module_file}

#Step1
import tensorflow as tf
import tensorflow_transform as tft
import taxi_constants

_DENSE_FLOAT_FEATURE_KEYS =
taxi_constants.DENSE_FLOAT_FEATURE_KEYS
_VOCAB_FEATURE_KEYS = taxi_constants.VOCAB_FEATURE_KEYS
_VOCAB_SIZE = taxi_constants.VOCAB_SIZE
_OOV_SIZE = taxi_constants.OOV_SIZE
 FEATURE_BUCKET_COUNT = taxi_constants.FEATURE_BUCKET_COUNT
_BUCKET_FEATURE_KEYS = taxi_constants.BUCKET_FEATURE_KEYS

```

```

_CATEGORICAL_FEATURE_KEYS =
taxi_constants.CATEGORICAL_FEATURE_KEYS
_FARE_KEY = taxi_constants.FARE_KEY
_LABEL_KEY = taxi_constants.LABEL_KEY
_transformed_name = taxi_constants.transformed_name

#Step2
def preprocessing_fn(inputs):
    outputs = {}
    for key in _DENSE_FLOAT FEATURE KEYS:
        # Preserve this feature as a dense float, setting nan's to the
        mean.
        outputs[_transformed_name(key)] =
            tft.scale_to_z_score(_fill_in_missing(inputs[key]))

    for key in _VOCAB_FEATURE_KEYS:
        # Build a vocabulary for this feature.
        outputs[_transformed_name(key)] =
            tft.compute_and_apply_vocabulary(
                _fill_in_missing(inputs[key]), top_k=_VOCAB_SIZE, num_oov_bucket
                s=_OOV_SIZE)

    for key in _BUCKET_FEATURE_KEYS:
        outputs[_transformed_name(key)] =
            tft.bucketize(_fill_in_missing(inputs[key]),
            _FEATURE_BUCKET_COUNT, always_return_num_quantiles=False)
        for key in _CATEGORICAL_FEATURE_KEYS:
            outputs[_transformed_name(key)] = _fill_in_missing(inputs[key])

    # Was this passenger a big tipper?
    taxi_fare = _fill_in_missing(inputs[_FARE_KEY])
    tips = _fill_in_missing(inputs[_LABEL_KEY])
    outputs[_transformed_name(_LABEL_KEY)] =
        tf.where(tf.math.is_nan(taxi_fare),
            tf.cast(tf.zeros_like(taxi_fare), tf.int64),
            # Test if the tip was > 20% of the fare.

```

```

    tf.cast(tf.greater(tips, tf.multiply(taxi_fare,
        tf.constant(0.2))), tf.int64))
return outputs

#step3
def _fill_in_missing(x):
    default_value = '' if x.dtype == tf.string else 0
    return tf.squeeze(tf.sparse.to_dense(tf.SparseTensor(x.indices,
        x.values, [x.dense_shape[0], 1]), default_value), axis=1)

```

The following function is the transformation function for our Transform function for all the features:

```

def transformation(example_gen, schema_gen):
    transform =
        Transform(examples=example_gen.outputs['examples'], schema=schem
a_gen.outputs['schema'], module_file=os.path.abspath(_taxi_trans
form_module_file))
    return transform

```

When we execute the transform component, TensorFlow Extended will apply the transformations, defined in our taxi_transform.py module file, and apply those to the loaded input data, loaded to TFRecords during the data ingestion step. The component will then output our transformed data, a transform graph, and the required metadata.

```

transform=transformation(example_gen, schema_gen)
context.run(transform)

```

```

▼ExecutionResult at 0x7f11683ece80
.execution_id      5
.component          ►Transform at 0x7f11683d92e8
.component.inputs   ['examples'] ►Channel of type 'Examples' (1 artifact) at 0x7f11c58e5320
                      ['schema']  ►Channel of type 'Schema' (1 artifact) at 0x7f11c3ac9f60
.component.outputs  ['transform_graph']      ►Channel of type 'TransformGraph' (1 artifact) at 0x7f11683a5e80
                      ['transformed_examples']  ►Channel of type 'Examples' (1 artifact) at 0x7f11683a58d0
                      ['updated_analyzer_cache']►Channel of type 'TransformCache' (1 artifact) at 0x7f11683a5a58

```

Figure 4.16: Transform Output

The output data structure for the transformation structure is shown in the preceding screenshot.

```
        }
      feature {
        key: "pickup_longitude_xf"
        value {
          int64_list {
            value: 9
          }
        }
      }
      feature {
        key: "tips_xf"
        value {
          int64_list {
            value: 0
          }
        }
      }
      feature {
        key: "trip_miles_xf"
        value {
          float_list {
            value: -0.15886740386486053
          }
        }
      }
    }
```

Figure 4.17: Transform Serialized json output

preprocessing_fn function, as shown in the preceding function defines all the transformations that we want to apply to the raw data. When we execute the transform component, the **preprocessing_fn** function will receive the raw data, applying the transformation and returning the processed data.

4.4.6 Tuner and Trainer

The Trainer component will train a model that you define in TensorFlow (either using the Estimator API or the Keras API with **model_to_estimator**).

Now, first the Model will be trained with the hyperparameter, and then it will train the model with the Trainer component which takes as input the schema

from **SchemaGen**, the transformed data and graph from **Transform**, training parameters, as well as a module that contains the user-defined model code.

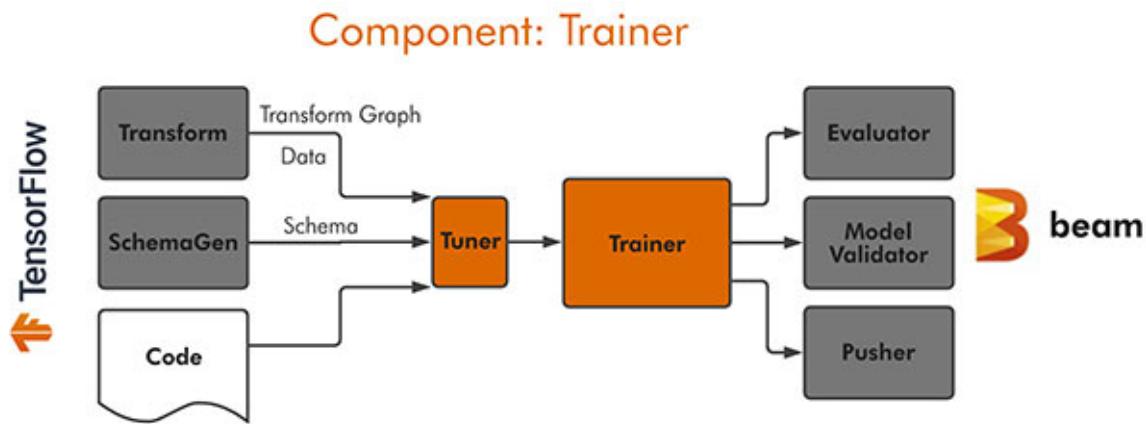


Figure 4.18: Trainer Architecture

The Trainer component requires the following inputs:

- The previously generated data schema, generated by the data validation.
- The transformed data and its preprocessing graph.
- The training parameters (for example, number of training steps).
- A module file containing a `run_fn()` function which defines the training process.

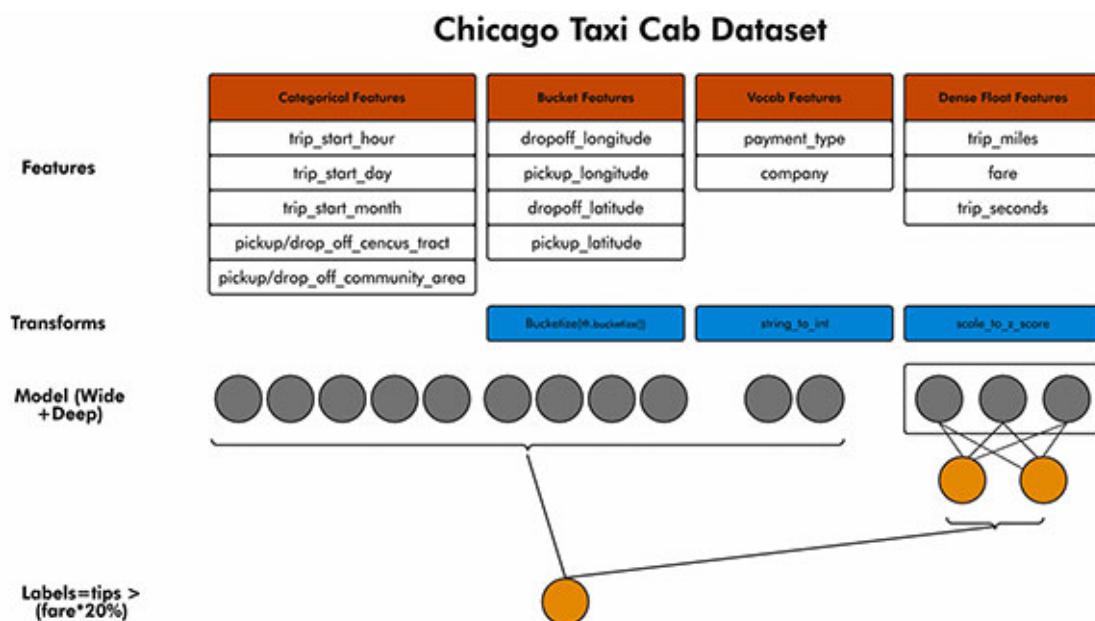


Figure 4.19: Transform Features

PRE-PROCESSING FUNCTION()

1: **tft.scale_to_z_score()**: If you want to normalize a feature with a mean of 0 and standard deviation of 1, you can use this useful TFT function.

2: **tft.bucketize()**: This useful function lets us bucketize a feature into bins. It returns the bin or bucket index. You can specify the argument **num_buckets** to set the number of buckets. TFT will then divide the equal sized buckets.

3: **tft.compute_and_apply_vocabulary()**: This is one of the most amazing TensorFlow Transform functions. It computes all the unique values of a feature column and then maps the most frequent values to an index. This index mapping is then used to convert the feature to a numerical representation. The function generates all the assets for your graph behind the scenes.

The Trainer file will train an estimator model by completing the following steps:

Step 1: Import the features in a list.

Step 2: Suffix will help to distinguish whether the errors are originating from the input or the output features and it prevent us from accidentally using a non-transformed feature in our actual model.

Step 3: Tf.Transform considers these features as "raw".

Step 4: Small utility returning a record reader that can read gzip'ed files.

Step 5: _build_estimator() function will create Wide and Deep model architecture with the estimator.

Step 6: Build the serving in inputs.

Step 7: Build the tf-model-analysis to run the model.

Step 8: Generate the features and labels for training or evaluation by the data generators.

Step 9: TFX will call this function as the main function to execute all the utility function.

```
%%skip_for_export
%%writefile {_taxi_trainer_module_file}
#Step1
import tensorflow as tf
import tensorflow_model_analysis as tfma
```

```

import tensorflow_transform as tft
from tensorflow_transform.tf_metadata import schema_utils
import taxi_constants

_DENSE_FLOAT_FEATURE_KEYS =
taxi_constants.DENSE_FLOAT_FEATURE_KEYS
_VOCAB_FEATURE_KEYS = taxi_constants.VOCAB_FEATURE_KEYS
_VOCAB_SIZE = taxi_constants.VOCAB_SIZE
_OOV_SIZE = taxi_constants.OOV_SIZE
 FEATURE_BUCKET_COUNT = taxi_constants.FEATURE_BUCKET_COUNT
_BUCKET_FEATURE_KEYS = taxi_constants.BUCKET_FEATURE_KEYS
_CATEGORICAL_FEATURE_KEYS =
taxi_constants.CATEGORICAL_FEATURE_KEYS
_MAX_CATEGORICAL_FEATURE_VALUES =
taxi_constants.MAX_CATEGORICAL_FEATURE_VALUES
_LABEL_KEY = taxi_constants.LABEL_KEY
_transformed_name = taxi_constants.transformed_name

#Step2
def _transformed_names(keys):
    return [_transformed_name(key) for key in keys]

#Step3
def _get_raw_feature_spec(schema):
    return schema_utils.schema_as_feature_spec(schema).feature_spec

#Step4
def _gzip_reader_fn(filenames):
    return
    tf.data.TFRecordDataset(filenames, compression_type='GZIP')

#Step5
def _build_estimator(config, hidden_units=None,
warm_start_from=None):
    real_valued_columns = [tf.feature_column.numeric_column(key,
shape=())]
    for key in _transformed_names(_DENSE_FLOAT_FEATURE_KEYS)]

```

```

categorical_columns =
[tf.feature_column.categorical_column_with_identity(
key, num_buckets=_VOCAB_SIZE + _OOV_SIZE, default_value=0)
for key in _transformed_names(_VOCAB_FEATURE_KEYS)]
categorical_columns +=

[tf.feature_column.categorical_column_with_identity(
    key, num_buckets=_FEATURE_BUCKET_COUNT, default_value=0)
     for key in _transformed_names(_BUCKET_FEATURE_KEYS)]
categorical_columns +=

[tf.feature_column.categorical_column_with_identity(key,
num_buckets=num_buckets, default_value=0) for key, num_buckets
in zip(
    _transformed_names(_CATEGORICAL_FEATURE_KEYS),_MAX_CATEGORICAL_
FEATURE_VALUES)]
return tf.estimator.DNNLinearCombinedClassifier(config=config,
    linear_feature_columns=categorical_columns,dnn_feature_columns
    =real_valued_columns,
    dnn_hidden_units=hidden_units or [100, 70, 50,
    25],warm_start_from=warm_start_from)

#Step6
def _example_serving_receiver_fn(tf_transform_graph, schema):
    raw_feature_spec = _get_raw_feature_spec(schema)
    raw_feature_spec.pop(_LABEL_KEY)
    raw_input_fn =
        tf.estimator.export.build_parsing_serving_input_receiver_fn(
            raw_feature_spec,
            default_batch_size=None)serving_input_receiver =
        raw_input_fn()
    transformed_features =
        tf_transform_graph.transform_raw_features(
            serving_input_receiver.features)
    return tf.estimator.export.ServingInputReceiver(
        transformed_features,
        serving_input_receiver.receiver_tensors)

```

```

#Step7
def _eval_input_receiver_fn(tf_transform_graph, schema):
    # Notice that the inputs are raw features, not transformed
    # features here.
    raw_feature_spec = _get_raw_feature_spec(schema)
    serialized_tf_example = tf.compat.v1.placeholder(
        dtype=tf.string, shape=[None], name='input_example_tensor')
    # Add a parse_example operator to the tensorflow graph, which
    # will parse raw, untransformed, tf examples.
    features = tf.io.parse_example(serialized_tf_example,
                                    raw_feature_spec)
    # Now that we have our raw examples, process them through the
    # tf-transform function computed during the preprocessing step.
    transformed_features =
        tf_transform_graph.transform_raw_features(
            features)
    # The key name MUST be 'examples'.
    receiver_tensors = {'examples': serialized_tf_example}
    # NOTE: Model is driven by transformed features (since training
    # works on the materialized output of TFT, but slicing will
    # happen on raw features.
    features.update(transformed_features)
    return
    tfma.export.EvalInputReceiver(features=features, receiver_tensor=
        s=receiver_tensors,
        labels=transformed_features[_transformed_name(_LABEL_KEY)])

```

```

#Step8
def _input_fn(filenames, tf_transform_graph, batch_size=200):
    transformed_feature_spec =
        (tf_transform_graph.transformed_feature_spec().copy())
    dataset = tf.data.experimental.make_batched_features_dataset(
        filenames, batch_size, transformed_feature_spec,
        reader=_gzip_reader_fn)
    transformed_features =
        (tf.compat.v1.data.make_one_shot_iterator(dataset).get_next())

```

```

# We pop the label because we do not want to use it as a
feature while we're training.
return transformed_features,
transformed_features.pop(_transformed_name(_LABEL_KEY))

#Step9
def trainer_fn(trainer_fn_args, schema):
    # Number of nodes in the first layer of the DNN
    first_dnn_layer_size = 100
    num_dnn_layers = 4
    dnn_decay_factor = 0.7
    train_batch_size = 40
    eval_batch_size = 40
    tf_transform_graph =
        tft.TFTransformOutput(trainer_fn_args.transform_output)
    train_input_fn = lambda:
        _input_fn(trainer_fn_args.train_files, tf_transform_graph,
                  batch_size=train_batch_size)
    eval_input_fn = lambda:
        _input_fn(trainer_fn_args.eval_files, tf_transform_graph,
                  batch_size=eval_batch_size)
    train_spec =
        tf.estimator.TrainSpec(train_input_fn, max_steps=trainer_fn_args
                               .train_steps)
    serving_receiver_fn = lambda:
        _example_serving_receiver_fn(tf_transform_graph, schema)
    exporter = tf.estimator.FinalExporter('chicago-taxi',
                                          serving_receiver_fn)
    eval_spec =
        tf.estimator.EvalSpec(eval_input_fn, steps=trainer_fn_args.eval_
                             steps,
                             exporters=[exporter], name='chicago-taxi-eval')
    run_config = tf.estimator.RunConfig(save_checkpoints_steps=999,
                                         keep_checkpoint_max=1)
    run_config =
        run_config.replace(model_dir=trainer_fn_args.serving_model_dir)

```

```

estimator = _build_estimator(hidden_units=[max(2,
int(first_dnn_layer_size * dnn_decay_factor**i))for i in
range(num_dnn_layers)],config=run_config,
warm_start_from=trainer_fn_args.base_model)
# Create an input receiver for TFMA processing
receiver_fn = lambda: _eval_input_receiver_fn(# pylint:
disable=g-long-lambda
    tf_transform_graph, schema)
return {'estimator': estimator, 'train_spec':
train_spec, 'eval_spec': eval_spec,
'eval_input_receiver_fn': receiver_fn}

```

Let's build the Tuner function with the train and eval parameters steps:

```

def tuner_model(transform):
    tuner =
        Tuner(module_file=os.path.abspath(_taxi_trainer_module_file),
examples=transform.outputs['transformed_examples'],
transform_graph=transform.outputs['transform_graph'],
train_args=trainer_pb2.TrainArgs(num_steps=20),
eval_args=trainer_pb2.EvalArgs(num_steps=50))
    return tuner

```

Next, we will create a Trainer function which will take the Tuner, Transform, Schema as the input for the model training:

```

def trainer_model(transform, schema_gen, tuner):
    trainer =
        Trainer(module_file=os.path.abspath(_taxi_trainer_module_file),
transformed_examples=transform.outputs['transformed_examples'],
schema=schema_gen.outputs['schema'],
transform_graph=transform.outputs['transform_graph'],
hyperparameters=tuner.outputs['best_hyperparameters'],
train_args=trainer_pb2.TrainArgs(num_steps=10000),
eval_args=trainer_pb2.EvalArgs(num_steps=5000))
    return trainer

```

Let's call the Tuner component and pass it to the trainer function:

```
tuner=tuner_model(transform)
trainer=trainer_model(transform,schema_gen)
context.run(trainer)
```

```
▼ExecutionResult at 0x7f1128089080
.execution_id      6
.component          ►Trainer at 0x7f1128089828
.component.inputs   ['examples']    ►Channel of type 'Examples' (1 artifact) at 0x7f11683a58d0
                           ['transform_graph'] ►Channel of type 'TransformGraph' (1 artifact) at 0x7f11683a5e80
                           ['schema']        ►Channel of type 'Schema' (1 artifact) at 0x7f11c3ac9f60
                           ['hyperparameters'] ►Channel of type 'HyperParameters' (0 artifacts) at 0x7f1128089b70
.component.outputs  ['model']       ►Channel of type 'Model' (1 artifact) at 0x7f1128089940
                           ['model_run']     ►Channel of type 'ModelRun' (1 artifact) at 0x7f11280890f0
```

Figure 4.20: Trainer Output

Analyze the Training with TensorBoard

Optionally, we can connect the TensorBoard to the Trainer to analyze our model's training curves. Get the URI of the output artifact representing the training logs, which is a directory:

```
model_dir = trainer.outputs['model'].get()[0].uri
%load_ext tensorboard
%tensorboard --logdir {model_dir}
```

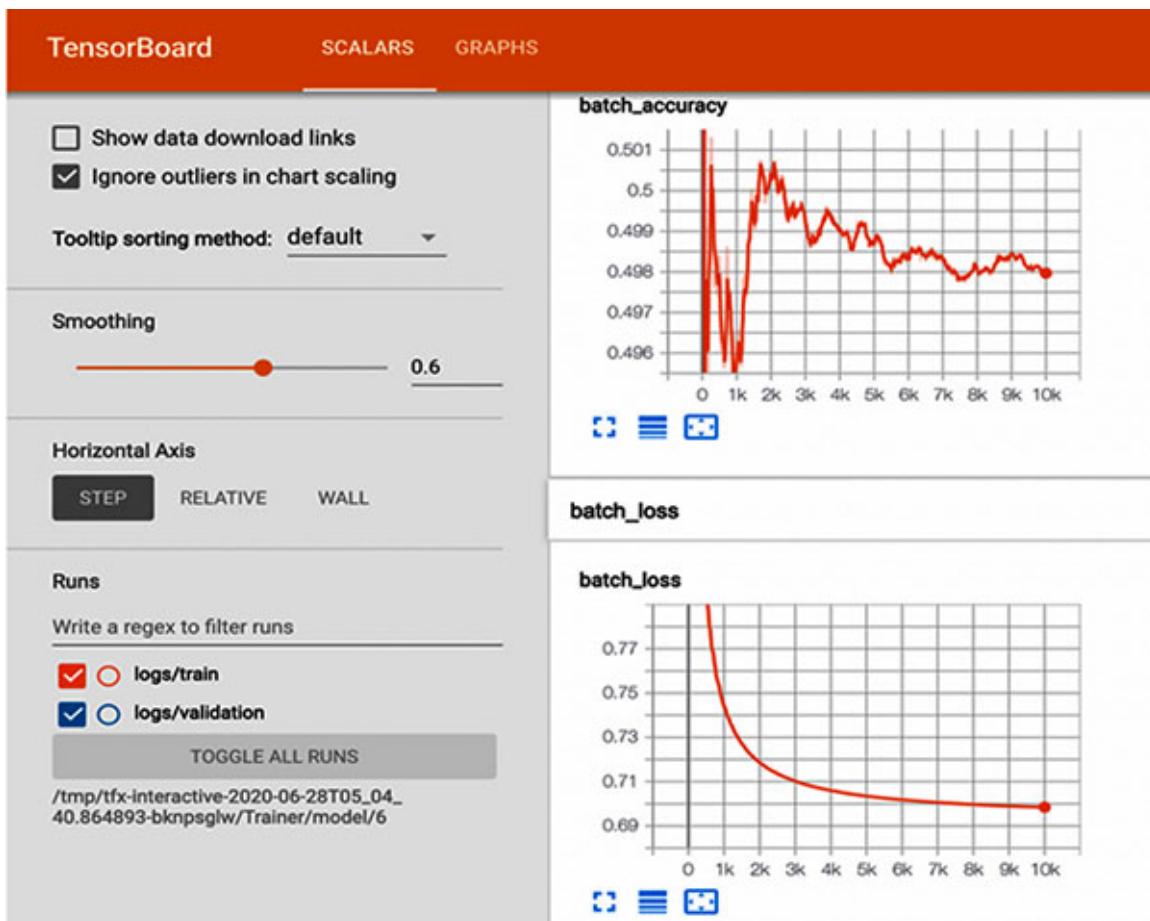


Figure 4.21: Tensorboard Output

Here, we will visualize the Batch Accuracy and Batch loss and Epoch Loss and epoch accuracy in our TensorBoard.

4.4.7 Evaluator

The Evaluator component computes the model performance metrics over the evaluation set. So, the Evaluator component automatically evaluates the sentiment as the probability.

Component: Evaluator

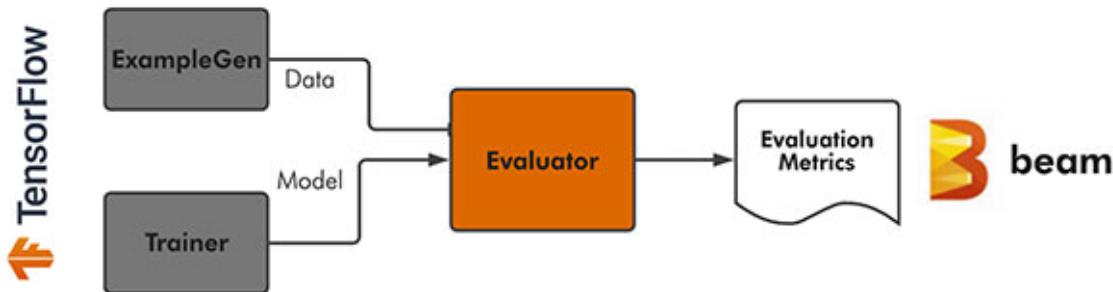


Figure 4.22: Evaluator Architecture

Let's build the Evaluation Component which will evaluate the Model performance:

```
def evaluation_configuration():
    eval_config = tfma.EvalConfig(
        model_specs=[tfma.ModelSpec(signature_name='eval')],
        metrics_specs=tfma.MetricsSpec(metrics=
            [tfma.MetricConfig(class_name='ExampleCount')]),
        thresholds = {'accuracy': tfma.MetricThreshold(
            value_threshold=tfma.GenericValueThreshold(lower_bound=
                {'value': 0.5}),
            change_threshold=tfma.GenericChangeThreshold(
                direction=tfma.MetricDirection.HIGHER_IS_BETTER, absolute=
                {'value': -1e-10}))},
        slicing_specs=
            [tfma.SlicingSpec(), tfma.SlicingSpec(feature_keys=
                ['trip_start_hour'])])
    return eval_config
```

Here, we will return the metrics which we will define for our evaluator model:

```
eval_config = evaluation_configuration()
```

The Resolver component is required if we want to compare a new model against a previous model. It checks for the last blessed model and returns this as a baseline, so it can be passed on to the Evaluator component with the new candidate model.

```
model_resolver =  
ResolverNode(instance_name='latest_blessed_model_resolver',  
resolver_class=latest_blessed_model_resolver.LatestBlessedModelRes  
olver,  
    model=Channel(type=Model),model_blessing=Channel(type=ModelBles  
sing))
```

The Evaluator component uses the TFMA library to evaluate a model's predictions on a validation dataset. It takes as input the data from the ExampleGen component, the trained model from the Trainer component, and an EvalConfig for TFMA.

```
def evaluator_component(model_resolver):  
    evaluator =  
        Evaluator(examples=example_gen.outputs['examples'],model=traine  
r.outputs['model'],  
            baseline_model=model_resolver.outputs['model'],eval_config=  
            eval_config)  
    return evaluator  
evaluator=evaluator_component(model_resolver)
```

The Evaluator helps to validate our exported models, confirming that they are "*good enough*" to be pushed for production.

4.4.7.1 Fairness and TFMA Visualization

TensorFlow Model Analysis (TFMA) helps to get more detailed metrics than just those used during the model training. TFMA visualizes the metrics as the time series across the model versions, and it gives us the ability to view the metrics on the slices of a dataset. It also scales easily to large evaluation sets, thanks to the Apache Beam.

```
%%skip_for_export  
context.show(evaluator.outputs['evaluation'])
```

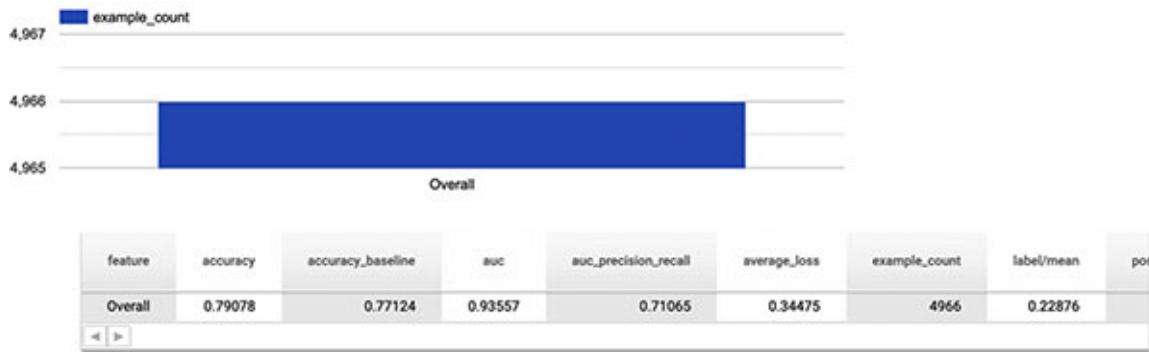


Figure 4.23: TFMA Metric Visualization

TFMA helps to visualize them in the `metrics_specs` argument to the `EvalConfig`.

```
evaluation_uri = evaluator.outputs['output'].get()[0].uri
eval_result = tfma.load_eval_result(evaluation_uri)
tfma.addons.fairness.view.widget_view.render_fairness_indicator(ev
al_result)
```

The Fairness Indicators is a library that enables the easy computation of the commonly-identified fairness metrics for the binary and multiclass classifiers. With the Fairness Indicators tool suite, you can do the following:

- Compute commonly-identified fairness metrics for the classification models.
- Compare the model performance across the subgroups to a baseline, or to the other models.
- Use the confidence intervals to surface the statistically significant disparities.
- Perform evaluation over multiple thresholds.

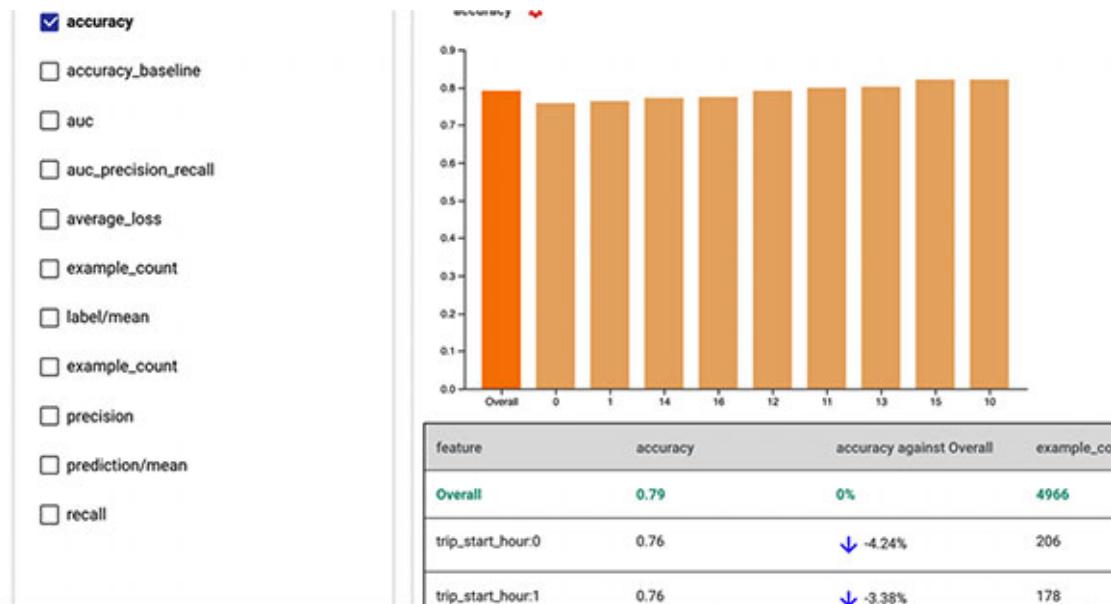


Figure 4.24: Fairness Metric Visualization

The Fairness Indicators is extremely useful tool for model analysis. It helps to some overlapping capabilities with TFMA, but one particularly useful feature of it is the ability to view metrics sliced on features at a variety of decision thresholds.

4.4.8 Pusher

The Pusher component is usually at the end of a TFX pipeline. It checks whether a model has passed the validation, and if so, exports the model to `_serving_model_dir`.

It takes as input a saved model, the output of the Evaluator component and a file path for the location of our models will be stored for serving.

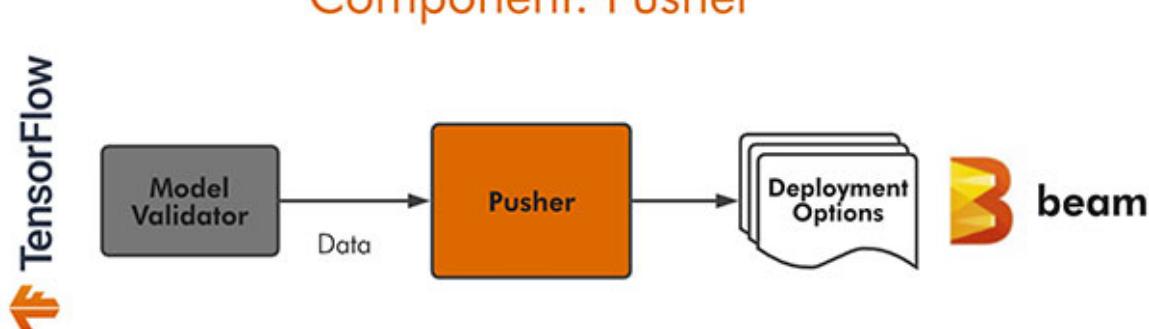


Figure 4.25: Pusher Architecture

Let's build the Pusher component, where the model will be pushed automatically for serving.

```
def pusher_model(trainer, evaluator):
    pusher = Pusher(model=trainer.outputs['model'],
                    model_blessing=evaluator.outputs['blessing'],
                    push_destination=pusher_pb2.PushDestination(
                        filesystem=pusher_pb2.PushDestination.Filesystem(base_directory=_serving_model_dir)))
    return pusher
```

Next, call the pusher Model for pushing the model for serve ready.

```
pusher=pusher_model(trainer, evaluator)
context.run(pusher)
```

```
▼ExecutionResult at 0x7f1cbbcf940
  .execution_id      9
  .component        ►Pusher at 0x7f1d500856d8
  .component.inputs  ['model']      ►Channel of type 'Model' (1 artifact) at 0x7f1cbb996668
                           ['model_blessing'] ►Channel of type 'ModelBlessing' (1 artifact) at 0x7f1d500b124
  .component.outputs ['pushed_model'] ►Channel of type 'PushedModel' (1 artifact) at 0x7f1d50085c88
```

Figure 4.26: Pusher Output

The Pusher component is provided with the model evaluator outputs and the serving destination. It depends on one or more blessing model from the other validation components to decide whether to push that model or not. The Evaluator blesses that model if the new trained model is fairly "*good enough*" to be pushed in production.

4.5 Serve the Model with TF Serving

Now that we have a trained model that has been blessed by ModelValidator, and pushed to our deployment target by Pusher, we can load it into the TensorFlow Serving and start serving the inference requests.

Installation

We're preparing to install the TensorFlow Serving using Aptitude since this Colab runs in a Debian environment. We'll add the tensorflow-model-server

package to the list of packages that Aptitude knows about. Note that we're running as root. This example is running the TensorFlow Serving natively, but you can also run it in a Docker container, which is one of the easiest ways to get started using TensorFlow Serving.

Step 1:

```
!echo "deb http://storage.googleapis.com/tensorflow-serving-apt
stable tensorflow-model-server tensorflow-model-server-universal"
| tee /etc/apt/sources.list.d/tensorflow-serving.list && \
curl https://storage.googleapis.com/tensorflow-serving-
apt/tensorflow-serving.release.pub.gpg | apt-key add -
!apt update
```

Step 2:

Running TensorFlow Serving in a Docker Container:

```
!apt-get install tensorflow-model-server
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  tensorflow-model-server
0 upgraded, 1 newly installed, 0 to remove and 13 not upgraded.
Need to get 210 MB of archives.
After this operation, 0 B of additional disk space will be used.
Get:1 http://storage.googleapis.com/tensorflow-serving-apt stable/tensorflow-model-server amd64 tensorflow-mode
Fetched 210 MB in 3s (73.7 MB/s)
Selecting previously unselected package tensorflow-model-server.
(Reading database ... 144628 files and directories currently installed.)
Preparing to unpack .../tensorflow-model-server_2.3.0_all.deb ...
Unpacking tensorflow-model-server (2.3.0) ...
Setting up tensorflow-model-server (2.3.0) ...
```

Figure 4.27: Installation Output

The next step is to load the Pusher and it will export your model in the SavedModel format and load the path.

```
latest_pushed_model = os.path.join(_serving_model_dir,
max(os.listdir(_serving_model_dir)))
!saved_model_cli show --dir {latest_pushed_model} --all
```

```

MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

signature_def['classification']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['inputs'] tensor_info:
      dtype: DT_STRING
      shape: (-1)

```

Figure 4.28: Show the graphs

Start running the Tensorflow Serving

This is where we start running the TensorFlow Serving and load our model. Once it loads, we can start making the inference requests using REST. There are some important parameters, which are as follows:

- **rest_api_port**: The port that you'll use for the REST requests.
- **model_name**: You'll use this in the URL of the REST requests. It can be anything.
- **model_base_path**: This is the path to the directory where you've saved your model. Note that this **base_path** should not include the model version directory, which is why we split it off as follows:

```

os.environ["MODEL_DIR"] = os.path.split(latest_pushed_model)
[0]
%%bash --bg
nohup tensorflow_model_server \
--rest_api_port=8501 \
--model_name=online_news_simple \
--model_base_path="${MODEL_DIR}" >server.log 2>&1
!tail server.log

```

Perform Inference on the example data

Let's load some examples from the eval dataset, remove their labels (as the serving model does not expect labels), and send them to the Tensorflow Serving through a single REST API call. Note that this will include the labels, but the server will ignore them.

```
eval_uri = example_gen.outputs['examples'].get()[0].uri
```

```

eval_tfrecord_paths = [os.path.join(eval_uri, name)for name in
os.listdir(eval_uri)]

def strip_label(serialized_example):
    example =
        tf.train.Example.FromString(serialized_example.numpy())
    return example.SerializeToString()

dataset =
    tf.data.TFRecordDataset(eval_tfrecord_paths1,compression_type="GZI
P")
    serialized_examples = [strip_label(serialized_example)for
    serialized_example in dataset.take(3)]

```

Here, we serialized the transformed data for evaluation and made a dataset TFRecord.

Send requests to the model, and print the results as follows:

- **server_addr**: Network address of the model server in "host:port" format.
- **model_name**: Name of the model as understood by the model server.
- **serialized_examples**: Serialized examples of the data to do the inference on.

```

def do_inference(server_addr, model_name,
serialized_examples):
    parsed_server_addr = server_addr.split(':')
    host=parsed_server_addr[0]
    port=parsed_server_addr[1]
    json_examples = []
    for serialized_example in serialized_examples:
        example_bytes =
            base64.b64encode(serialized_example).decode('utf-8')
        predict_request = '{"b64": "%s"}' % example_bytes
        json_examples.append(predict_request)

    json_request = '{"instances": [' + ', '.join(map(str,
    json_examples)) + ']}'
```

```

server_url = 'http://' + host + ':' + port + '/v1/models/' +
+ model_name + ':predict'
response = requests.post(server_url, data=json_request,
timeout=5.0)
response.raise_for_status()
prediction = response.json()
print(json.dumps(prediction, indent=4))

```

Let's call the inference function for our prediction of our sample dataset:

```

do_inference(server_addr='127.0.0.1:8501',
model_name='online_news_simple',
serialized_examples=serialized_examples)

{
    "predictions": [
        {
            "scores": [
                0.775838792,
                0.224161178
            ],
            "classes": [
                "0",
                "1"
            ]
        },
        {
            "scores": [
                0.775838792,
                0.224161178
            ],
            "classes": [
                "0",
                "1"
            ]
        },
        {
            "scores": [
                0.775838792,
                0.224161178
            ],
            "classes": [
                "0",
                "1"
            ]
        }
    ]
}

```

Figure 4.29: Test Model Prediction

In the preceding screenshot, we can see the classification output for the batch of 3 records and the probability score for both the Tips greater than Fare by 20% or not.

4.6 Building Kubeflow Pipeline Orchestrator

Prerequisites: So, before starting this chapter, we must setup the Kubeflow Cluster in GCP, we have already created a Jupyter notebook; we will use that.

Open the terminal once after you connect to the Jupyter Notebook from Kubeflow Dashboard and install the TFX SDK.

```
```bash
!pip install tfx==0.22.0
```

```

Optional: If you get a following error run the following commands:

```
/usr/local/lib/python3.6/dist-packages/apache_beam/portability/api/endpoints_pb2.py in <module>
    19     syntax='proto3',
    20     serialized_options=b'\n!org.apache.beam.model.pipeline.v1B\tEndpointsZ@013pipeline_v1',
--> 21     create_key=_descriptor._internal_create_key,
    22     serialized_pb=b'\n\x0f\x65ndpoints.proto\x12!org.apache.beam.model.pipeline.v1"\r\n\x14\x0e
\x03url\x18\x01 \x01(\t\x12M\x0e\x61uthentication\x18\x02 \x01(\x0b\x32\x35.org.apache.beam.mod
pec"\x2\x12\x41uthenticationSpec\x12\x0b\x03urn\x18\x01 \x01(\t\x12\x0f\x07payload\x18\x02 \x0
model.pipeline.v1B\tEndpointsZ\x0bpipeline_v1b\x06proto3'
    23 )
AttributeError: module 'google.protobuf.descriptor' has no attribute '_internal_create_key'
```

Figure 4.30: Error

Run the following commands:

```
```bash
PROTOC_ZIP=protoc-3.7.1-osx-x86_64.zip
curl -OL
https://github.com/protocolbuffers/protobuf/releases/download/v3.7
.1/$PROTOC_ZIP
sudo unzip -o $PROTOC_ZIP -d /usr/local/bin/protoc
sudo unzip -o $PROTOC_ZIP -d /usr/local/include/*
rm -f $PROTOC_ZIP
pip install --upgrade protobuf
```

```

Now, please go to the GitHub of this project and download those folders and save those files in your GCP Bucket, as shown in the following screenshot:



Figure 4.31: Kubeflow Pipeline Folders data & file

Now, let's see the pipeline code and how we can generate the constructive pipeline for TFX.

Complete the following steps:

1. Define the pipeline parameters used for the pipeline execution. The path to the module file should be a GCS path, or a module file baked in the Docker image used by the pipeline.
2. The path to the CSV data file, under which there should be a **data.csv** file.
3. The path of the pipeline root should be a GCS path.
4. Create a simple Kubeflow-based Chicago Taxi TFX pipeline.

pipeline_root: The root of the pipeline output.

csv_input_location: The location of the input data directory.

taxi_module_file: The location of the module file for Transform/Trainer.

enable_cache: Whether to enable cache or not.

5. A pipeline is a **directed acyclic graph (DAG)** with a containerized process on each node, which runs on the top of argo.

A logical TFX pipeline object. After that we going to build KubeflowDagrunner which will dump a yaml file and it will be used to deploy that in Kubeflow to run the pipeline.

```
import os
from typing import Text
import kfp
import tensorflow_model_analysis as tfma
from tfx.components.evaluator.component import Evaluator
from tfx.components.example_gen.csv_example_gen.component import CsvExampleGen
from tfx.components.example_validator.component import ExampleValidator
from tfx.components.pusher.component import Pusher
from tfx.components.schema_gen.component import SchemaGen
from tfx.components.statistics_gen.component import StatisticsGen
from tfx.components.trainer.component import Trainer
from tfx.components.transform.component import Transform
from tfx.orchestration import data_types
from tfx.orchestration import pipeline
from tfx.orchestration.kubeflow import kubeflow_dag_runner
from tfx.utils.dsl_utils import external_input
from tfx.proto import pusher_pb2, trainer_pb2
```

STEP 1:

```
_taxi_module_file_param =
data_types.RuntimeParameter(name='module-file',
    default='gs://<BUCKET_NAME>/tfx_taxi_simple/modules/taxi_utils.
    py', ptype=Text)
```

STEP 2:

```
_data_root_param = data_types.RuntimeParameter(name='data-root',
ptype=Text
    default='gs://<BUCKET_NAME>//tfx_taxi_simple/data')
```

STEP 3:

```
pipeline_root = os.path.join('gs://{{kfp-default-bucket}}',
'tfx_taxi_simple', kfp.dsl.RUN_ID_PLACEHOLDER)
```

STEP 4:

```
def _create_pipeline(pipeline_root: Text, csv_input_location:  
    data_types.RuntimeParameter,  
    taxi_module_file: data_types.RuntimeParameter, enable_cache:  
    bool):  
    examples = external_input(csv_input_location)  
    example_gen = CsvExampleGen(input=examples)  
    statistics_gen =  
        StatisticsGen(examples=example_gen.outputs['examples'])  
    infer_schema =  
        SchemaGen(statistics=statistics_gen.outputs['statistics'],  
        infer_feature_shape=False)  
    validate_stats =  
        ExampleValidator(statistics=statistics_gen.outputs['statistics'],  
        schema=infer_schema.outputs['schema'],)  
    transform = Transform(examples=example_gen.outputs['examples'],  
        schema=infer_schema.outputs['schema'], module_file=taxi_modul  
        e_file)  
    trainer = Trainer(module_file=taxi_module_file,  
        transformed_examples=transform.outputs['transformed_examples'],  
        schema=infer_schema.outputs['schema'],  
        transform_graph=transform.outputs['transform_graph'],  
        train_args=trainer_pb2.TrainArgs(num_steps=10),  
        eval_args=trainer_pb2.EvalArgs(num_steps=5))  
    eval_config = tfma.EvalConfig(model_specs=[tfma.ModelSpec(signature_name='eval')], metrics_specs=[tfma.MetricsSpec(metrics=[tfma.MetricConfig(class_name='ExampleCount')]), thresholds={'binary_accuracy':tfma.MetricThreshold(value_threshold=tfma.GenericValueThreshold(lower_bound={'value': 0.5}), change_threshold=tfma.GenericChangeThreshold(direction=tfma.MetricDirection.HIGHER_IS_BETTER, absolute={'value': -1e-10}))}], slicing_specs=
```

```

[tfma.SlicingSpec(), tfma.SlicingSpec(feature_keys=
['trip_start_hour'])])
model_analyzer =
Evaluator(examples=example_gen.outputs['examples'],
    model=trainer.outputs['model'], eval_config=eval_config)
pusher = Pusher(model=trainer.outputs['model'],
    model_blessing=model_analyzer.outputs['blessing'],
    push_destination=pusher_pb2.PushDestination(
        filesystem=pusher_pb2.PushDestination.Filesystem(
            base_directory=os.path.join(str(pipeline.ROOT_PARAMETER),
                'model_serving'))))
return
pipeline.Pipeline(pipeline_name='parameterized_tfx_oss', pipeline_root=pipeline_root, components=[example_gen, statistics_gen, infer_schema, validate_stats, transform, trainer, model_analyzer, pusher], enable_cache=enable_cache)
if __name__ == '__main__':
    enable_cache = True
    pipeline =
        _create_pipeline(pipeline_root,_data_root_param,_taxi_module_file_param, enable_cache=enable_cache)
    metadata_config =
        kubeflow_dag_runner.get_default_kubeflow_metadata_config()
    config = kubeflow_dag_runner.KubeflowDagRunnerConfig(
        kubeflow_metadata_config=metadata_config, tfx_image='gcr.io/tfx-oss-public/tfx:0.22.0')
    kfp_runner =
        kubeflow_dag_runner.KubeflowDagRunner(output_filename='pipe9' +
        '.yaml', config=config)
    kfp_runner.run(pipeline)

```

After running the preceding code, it will dump a yaml, as shown in the following screenshot. Before running the file, replace the bucket name here in step1/2.

The screenshot shows a Jupyter Notebook interface with two tabs: 'Pipeline.ipynb' and 'Terminal 1'. The 'Pipeline.ipynb' tab contains Python code for defining a Kubeflow pipeline. The code uses the 'kubeflow_dag_runner' module to set up a runner configuration and run the pipeline.

```

config = kubeflow_dag_runner.KubeflowDagRunnerConfig(
    kubeflow_metadata_config=metadata_config,
    tfx_image='gcr.io/tfx-oss-public/tfx:0.22.0',
)
kfp_runner = kubeflow_dag_runner.KubeflowDagRunner(
    output_filename='pipeline' + '.yaml', config=config
)
kfp_runner.run(pipeline)

```

Figure 4.32: Kubeflow Pipeline yaml

Once we dump the yaml, complete the following steps to run the pipeline in Kubeflow:

1. Upload the `pipeline.yaml` file in the Kubeflow pipeline, and click on **Create**.

The screenshot shows the Kubeflow UI for uploading a pipeline. The left sidebar has navigation links: Pipelines, Experiments, Artifacts, Executions, Archive, Documentation, Github Repo, and AI Hub Samples. The main area is titled 'Upload Pipeline or Pipeline Version'. It has two radio button options: 'Create a new pipeline' (selected) and 'Create a new pipeline version under an existing pipeline'. Below this is a 'Pipeline Name' field containing 'pipeline'. A 'Pipeline Description' field is also present. A note says: 'Choose a pipeline package file from your computer, and give the pipeline a unique name. You can also drag and drop the file here.' Below this is a 'File' input field with 'pipeline.yaml' selected, and a 'Choose file' button. There is also an 'Import by url' option with a 'Package Url' input field. At the bottom are 'Create' and 'Cancel' buttons.

Figure 4.33: Upload pipeline.yaml

2. Next, create an experiment if you haven't yet; we have done so already in our previous chapters. After that, click on **Run** and choose that pipeline.



Figure 4.34: Create Run

3. Before starting the pipeline, run 'choose an experiment' and then change the following parameters, like replacing your Bucket name; for example: tfx-pipeline is my bucket name and tfx is my experiment name.
4. Click on **Start**.

This run will be associated with the following experiment

Experiment* [Choose](#)

Run parameters

Specify parameters required by the pipeline

pipeline-root

data-root

module-file

Start **Cancel**

The form shows the 'Experiment' field set to 'TFX' with a 'Choose' button. Under 'Run parameters', the 'pipeline-root' field contains the value 'gs://tfx-pipeline/tfx_taxi_simple/{{workflow.uid}}'. The 'data-root' and 'module-file' fields also contain their respective paths. At the bottom, there are 'Start' and 'Cancel' buttons.

Figure 4.35: Pipeline parameters

After that, the pipeline will be ready, as shown in the following screenshot:

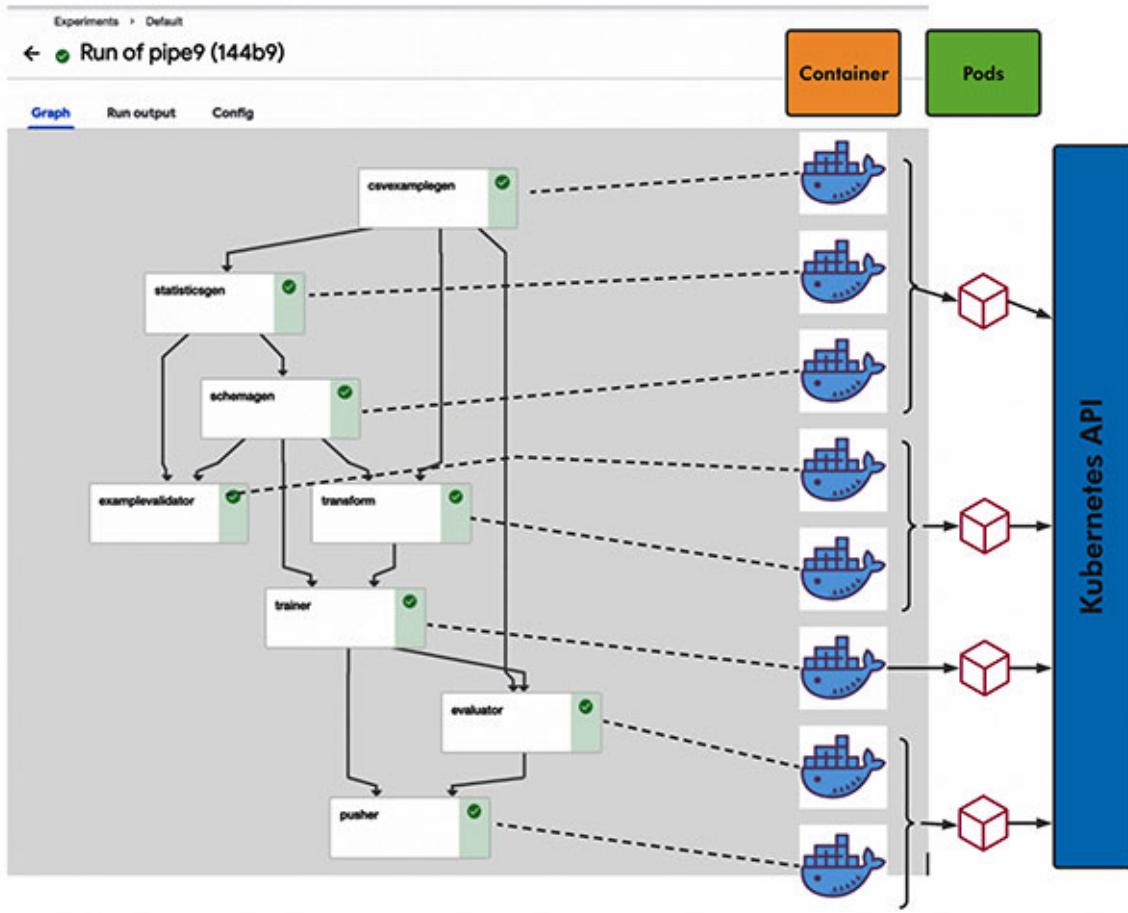


Figure 4.36: Pipeline of TFX

Each pipeline component, represented as a block, is a self-contained piece of code, packaged as a Docker image. It contains the inputs (arguments) and outputs and performs one step in the pipeline. In the example pipeline, shown earlier, the `transform_data` step requires the arguments that are produced as an output of the `extract_data` and of the `generate_schema` steps, and its outputs are the dependencies for `train_model`.

Your ML code is wrapped into components, where you can do the following:

- Specify **parameters** – which become available to edit in the dashboard and configurable for every run.
- Attach **persistent volumes** – without adding persistent volumes, we would lose all the data if our notebook was terminated for any reason.
- Specify **artifacts** to be generated – graphs, tables, selected images, models – which end up conveniently stored on the Artifact Store, inside

the Kubeflow dashboard.

Finally, when you run the pipeline, each container will now be executed throughout the cluster, according to the Kubernetes scheduling, taking the dependencies into consideration. This containerized architecture makes it simple to reuse, share, or swap out the components as your workflow changes, which tends to happen.

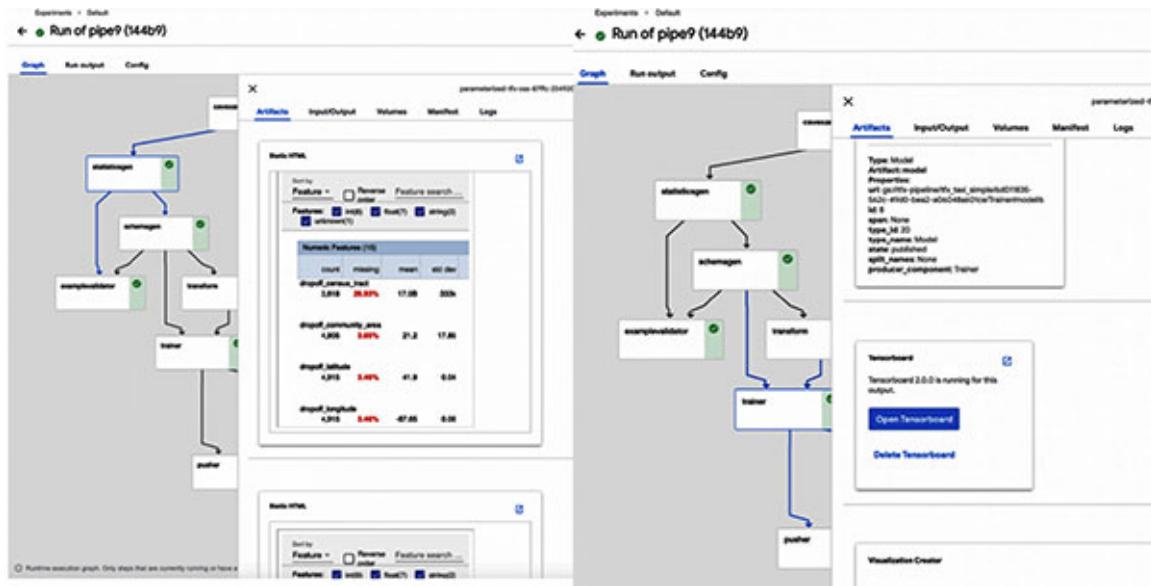


Figure 4.37: Kubeflow pipeline visualization

After running the pipeline, you will be able to explore the results on the **pipelines interfaces**, debug, tweak parameters, and run experiments by executing the pipeline with the different parameters or data sources.

Now, after the completion of the pipeline, run the model artifacts dumped inside the GCS bucket. In the following screenshot, we can see the serving model directory:

| Bucket details | | | | | | | |
|---|---|-------------------------------|------------------------------|---|---------------|---------------|--|
| tfx-pipeline | | | | | | | |
| OBJECTS | CONFIGURATION | PERMISSIONS | RETENTION | LIFECYCLE | | | |
| Buckets > [REDACTED] tfx_taxi_simple > 6d011835-562c-49d0-bea2-a06048a601ce [REDACTED] | | | | | | | |
| UPLOAD FILES | UPLOAD FOLDER | CREATE FOLDER | MANAGE HOLDS | DELETE | | | |
| Filter Filter by object or folder name prefix | | | | | | | |
| <input type="checkbox"/> | Name | Size | Type | Created time [REDACTED] | Storage class | Last modified | Public access [REDACTED] |
| <input type="checkbox"/> | [REDACTED] CsvExampleG | — | Folder | — | — | — | — |
| <input type="checkbox"/> | [REDACTED] Evaluator/ | — | Folder | — | — | — | — |
| <input type="checkbox"/> | [REDACTED] ExampleValidi | — | Folder | — | — | — | — |
| <input type="checkbox"/> | [REDACTED] Pusher/ | — | Folder | — | — | — | — |
| <input type="checkbox"/> | [REDACTED] SchemaGen/ | — | Folder | — | — | — | — |
| <input type="checkbox"/> | [REDACTED] StatisticsGen/ | — | Folder | — | — | — | — |
| <input type="checkbox"/> | [REDACTED] Trainer/ | — | Folder | — | — | — | — |
| <input type="checkbox"/> | [REDACTED] Transform/ | — | Folder | — | — | — | — |
| <input type="checkbox"/> | [REDACTED] model_servin | — | Folder | — | — | — | — |

Figure 4.38: Kubeflow pipeline Artifacts & Model in GCS

4.7 Conclusion

In this chapter, we learned how to build the TFX pipeline for the BERT Model for production ready with the integration of Visualization Tools like Fairness, Tensorboard, TFMA.

We also learned how to load the BERT Model from TF-HUB and how to use it. Then, we learned about the transformation pipeline prior to training the Model. We have also learned how to train our BERT Model with the distributed GPU Node strategy. Then, we created a pipeline with the TFX Components and visualized and evaluated the blessed model with the Fairness Indicators and metrics. Then, we loaded the server model and tested with a sample movie review and checked its sentiment probability.

4.8 Reference

- <https://www.tensorflow.org/tfx/tutorials>

- https://www.tensorflow.org/responsible_ai/fairness_indicators/tutorials/Fairness_Indicators_Example_Colab

CHAPTER 5

ML Model Explainability & Interpretability

In this chapter, we will work for a classification model with the hotel booking dataset, train the TensorFlow and boosting models, and visualize the advanced explanation of our model results with Tensorboard, Shap, and What-if products.

Structure

In this chapter, we will cover the following topics:

- Problem statement
- Getting started with Python library installation and data loading in Colab
- Feature transformation for Training Model
- LightGBM Model training
- Model Analysis with advance visualization along Shap tool
- TensorFlow Estimator Model Building Framework
- Advance Visualization for TensorFlow Model with Tensorboard, What-IF Tool and Fairness

Objectives

After studying this chapter, we will be able to understand the following:

- The implementation of Shapely Additive explanations and how to use it's different approach on model evaluation.
- How to build LightGBM model building from scratch.
- The different Advance Shap plots for model analysis.
- How to build the TensorFlow Estimator Framework end to end.
- How to evaluate the Model performance with the What-if Tool alongside the Fairness indicator.

5.1 Problem

The problem statement for this data set contains the booking information for a city hotel and a resort hotel, which includes some information, such as when the booking was made, length of stay, and the number of adults, children, or babies, and the number of available parking spaces, among other things.

Have you ever wondered when the best time is in a year to book a hotel room? Or wondered about the optimal length of stay in order to get the best daily rate? Or, what if you wanted to predict whether or not a hotel is familiar to receive a disproportionately high amount of special requests?

| | |
|------|---|
| NOTE | Rest all the imports I have showed in my Colab Notebook, which I gave hyperlink of the GitHub Account of this chapter. Note: Colab platform Python 3.x. RUN IN GOOGLE COLAB |
| CODE | https://github.com/bpbpublications/Continuous-Machine-Learning-with-Kubeflow/tree/main/Chapter5 |

5.2 General idea and concept behind Shap

The goal of SHAP is to explain the prediction of an instance x by computing the contribution of each feature to the prediction. From the coalitional game theory, the shapely values were computed from the SHAP explanation methods. Each feature value instance acts as players in a coalition. How to fairly distribute the prediction among the features, Shapley values tells us that. A player is an individual feature value, for example, from a tabular data. A group of feature values can also be a player.

For example, the prediction distribution among the image pixels can also be grouped to super pixels. One of the innovation that SHAP brought to the table is an additive feature attribution method, a linear model that explain the Shapley value.

SHAP specifies the explanation as follows:

$$g(z') = \phi_o + \sum_{j=0}^M \phi_j z'_j$$

SHAP describes the following three desirable properties:

- Local accuracy

$$f(x) = g(x') = \phi_o + \sum_{j=1}^M \phi_j X'_j$$

For & set all to 1, it is the efficiency property of Shapley; only with a different name and by using coalition vector.

$$f(x) = \phi_o + \sum_{j=1}^M \phi_j X'_j = E_x(\hat{f}(x)) + \sum_{j=1}^M \phi_j$$

- **Missingness**

$$X'_j = 0 \Rightarrow \phi_j = 0$$

It tells that a missing feature gets an attribution of zero. Note that x'_j refers to the coalitions, where a value of 0 represents the absence of a feature value. In the coalition notation, all the feature values x'_j of the instance to be explained should be '1'. The presence of a 0 means that the feature value is missing.

- **Consistency**

Let $f_x(z') = f(h_x(z'))$ and $z(j')$ indicate that $z_{j'} = 0$. For any two models, f and f' that satisfy the following:

$$f'_x(z') - f'_x(z|_{j'}) \geq f_x(z') - f_x(z|_{j'})$$

For all inputs $z' \in \{0,1\}^M$, then:

$$\phi_j(f',x) \geq \phi_j(f,x)$$

The consistency property tells that if a model changes so that the marginal contribution of a feature value increases or stays the same irrespective of other features, then the Shapley value also increases or stays the same.

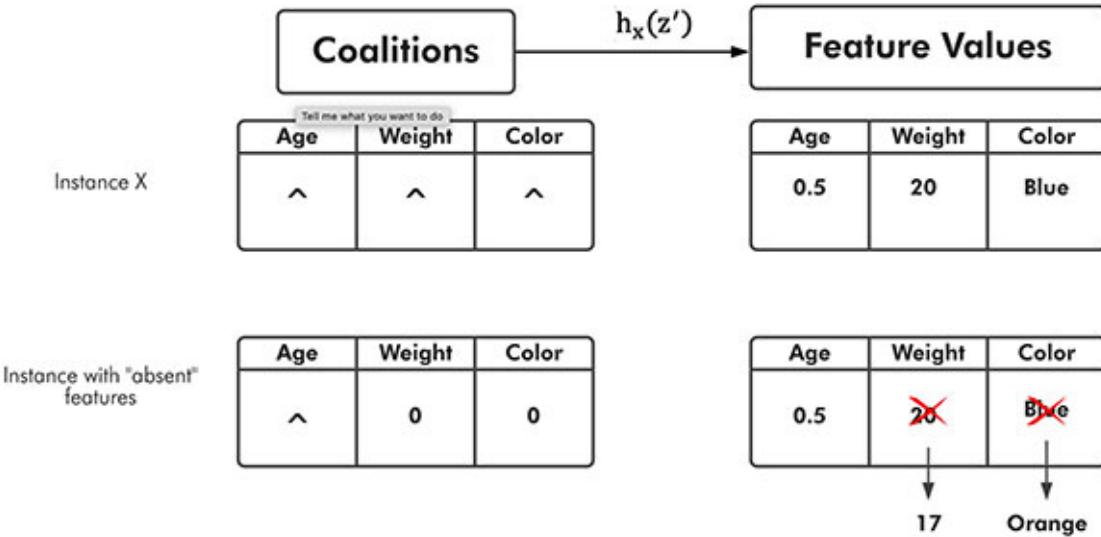


Figure 5.1: SHAP explanation example

For example, for a vector of $(1,0,1,0)$, it represents that the first and third features have coalitions. The dataset for the regression model became K sampled coalitions. So, for the regression model, the target became the prediction for a coalition.

We need a function $h_x(z') = z$ where $h_x: \{0,1\}^M \rightarrow \mathbb{R}^P$ to get the data instances which are valid from the coalitions of the features values. So, maps to 1's respective value to the instance x which will we explain here. Now, for a tabular form of data, it maps 0's to another instance values that we got from the sample data. This says that next we will equate the "absent feature value" with the "feature value" and will replace that by the random set of feature values from data.

Now, from [figure 5.1](#), functions maps a valid instance to coalition. Next, the maps feature values (1) of x to the present features.

For the absent features (0), maps the values of a randomly sampled data instance. for tabular data treats and as independent and integrates over the marginal distribution:

$$f(h_x(z')) = E_{XC}[f(x)]$$

In the game theory, SHAP has a **solid theoretical foundation**. The fairly distributed predictions among the feature values compare the prediction with the average prediction to get the **contrastive explanations**.

5.3 Getting Started with Python library Installation and Data loading in Colab

Now, we will start with the installation of the Library of Shap and Witwidget.

```
!pip install shap
```

```
Successfully built shap
Installing collected packages: shap
Successfully installed shap-0.35.0
```

Figure 5.2: Installation of Shap

```
try:
    import google.colab
    !pip install --upgrade witwidget
except:
    pass

Requirement already satisfied, skipping upgrade: pyparsing>=2.0.2 in
Installing collected packages: witwidget
Successfully installed witwidget-1.6.0
```

Figure 5.3: Installation of wit-widget

We have successfully installed both the wit-widget and the Shap visualization tools.

Now we will be importing the required libraries.

```
import numpy as np
import tensorflow as tf
from pprint import pprint
import lightgbm as lgb
import shap, warnings, functools
from sklearn.model_selection import train_test_split,
StratifiedKFold
from sklearn.preprocessing import LabelEncoder
data=pd.read_csv("hotel_bookings.csv")
data.head()
```

| | hotel | is_canceled | lead_time | arrival_date_year | arrival_date_month | arrival_date_week_number | arrival_date_day_of_month | stays_in_weekend_nights | stays_in_week |
|---|--------------|-------------|-----------|-------------------|--------------------|--------------------------|---------------------------|-------------------------|---------------|
| 0 | Resort Hotel | 0 | 342 | 2015 | July | 27 | 1 | 0 | |
| 1 | Resort Hotel | 0 | 737 | 2015 | July | 27 | 1 | 0 | |
| 2 | Resort Hotel | 0 | 7 | 2015 | July | 27 | 1 | 0 | |
| 3 | Resort Hotel | 0 | 13 | 2015 | July | 27 | 1 | 0 | |
| 4 | Resort Hotel | 0 | 14 | 2015 | July | 27 | 1 | 0 | |

Figure 5.4: Hotel Booking Dataset Table

```
data.is_canceled.value_counts()
```

```
0    75166
1    44224
Name: is_canceled, dtype: int64
```

Figure 5.5: Target column distribution

In the preceding screenshots, we can see the value count distribution for the target predictor column.

Now, according to our domain knowledge and understanding, we choose some columns manually.

```
features =
["lead_time", "arrival_date_week_number", "arrival_date_day_of_month",
",
"stays_in_weekend_nights", "stays_in_week_nights", "adults", "children",
"babies", "is_repeated_guest", "previous_cancellations",
"previous_bookings_not_canceled", "agent", "company",
"required_car_parking_spaces", "total_of_special_requests",
"adr", "hotel", "arrival_date_month", "meal", "market_segment",
"distribution_channel", "reserved_room_type", "deposit_type", "customer_type", 'is_canceled']
Raw_Data=data[features]
```

5.4 Feature transformation for Training Model

Now we will be creating some copy for our dataframe and not change the original dataframe, so that we can use that for further analysis without intervening any changes to that.

```
X = Raw_Data.drop(["is_canceled"], axis=1)
y = Raw_Data["is_canceled"]
X_display = X.copy()
y_display = y.copy()
X_new=X.copy()
```

Next, we will check the data types for our dataset and will do necessary transformation for our features which is categorical.

X.dtypes

| | | |
|---|--------------------------------|---------|
| ↳ | lead_time | int64 |
| | arrival_date_week_number | int64 |
| | arrival_date_day_of_month | int64 |
| | stays_in_weekend_nights | int64 |
| | stays_in_week_nights | int64 |
| | adults | int64 |
| | children | float64 |
| | babies | int64 |
| | is_repeated_guest | int64 |
| | previous_cancellations | int64 |
| | previous_bookings_not_canceled | int64 |
| | agent | float64 |
| | company | float64 |
| | required_car_parking_spaces | int64 |
| | total_of_special_requests | int64 |
| | adr | float64 |
| | hotel | object |
| | arrival_date_month | object |
| | meal | object |
| | market_segment | object |
| | distribution_channel | object |
| | reserved_room_type | object |
| | deposit_type | object |
| | customer_type | object |
| | dtype: | object |

Figure 5.6: Data Types for features

In the preceding screenshot, we can see the datatypes object that we need to transform, as we don't have any missing values in this dataset. So, we will deal with all the categorical columns like deposit_type, meal, customer_type, etc.

Here is the function for transformation as we will use label encoder in object data types columns and will iterate through those and replace the dataset columns.

```
def transform_categorical():
    s = (X.dtypes == 'object')
    object_cols = list(s[s].index)
    print(object_cols)
    for i in object_cols:
        lb_make = LabelEncoder()
        X[i] = lb_make.fit_transform(X[i])
    #for i in object_cols:
    #    X[i] = X[i].astype(float)
    return X
X=transform_categorical()
X.head()
```

| arking_spaces | total_of_special_requests | adr | hotel | arrival_date_month | meal | market_segment | distribution_channel | reserved_room_type | deposit_type | customer_type |
|---------------|---------------------------|-----|-------|--------------------|------|----------------|----------------------|--------------------|--------------|---------------|
| 0 | 0 0.0 | 1 | | 5 | 0 | 3 | | 1 | 2 | 0 |
| 0 | 0 0.0 | 1 | | 5 | 0 | 3 | | 1 | 2 | 0 |
| 0 | 0 75.0 | 1 | | 5 | 0 | 3 | | 1 | 0 | 0 |
| 0 | 0 75.0 | 1 | | 5 | 0 | 2 | | 0 | 0 | 0 |
| 0 | 1 98.0 | 1 | | 5 | 0 | 6 | | 3 | 0 | 0 |

Figure 5.7: Transform Table

Now, we have transformed our dataset; next, we will split our dataset for training and will train our model.

```
def data_split():
    random_state = 7
    X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=random_state)
    return X_train, X_test, y_train, y_test
X_train, X_test, y_train, y_test=data_split()
```

Now, we are ready for training our Model.

5.5 LightGBM Model training

LightGBM uses the histogram-based algorithm which bucket the continuous feature (attribute) values into discrete bins. This speeds up the training and reduces the memory usage. For further reading, please go to the following link:

<https://lightgbm.readthedocs.io/en/latest/Features.html>

Now, we will be creating the data preparation for LightGBM model.

```
d_train = lgb.Dataset(X_train, label=y_train)
d_test = lgb.Dataset(X_test, label=y_test)

random_state=7
params = {
    "max_bin": 512, "learning_rate": 0.05, "boosting_type": "gbdt",
    "objective": "binary", "metric": "binary_logloss",
    "num_leaves": 10,
    "verbose": -1, "min_data": 100, "boost_from_average": True,
    "random_state": random_state
}
```

Let's give a brief summary for the Model Parameters:

- **max_bin default = 255, type = int, constraints: max_bin > 1** so, the maximum number of bins that feature values will bucket in, and the small number of bins will reduce the training accuracy but it may increase the general power (deal with over-fitting).
- **boosting_type** is choosing the algorithm technique traditional Gradient Boosting Decision Tree.
- **Num_leaves** it is the maximum number of leaves in one tree.

So, there are many parameters which we can learn in detail from the following link:

<https://lightgbm.readthedocs.io/en/latest/Parameters.html>

Next, we will train our model.

```
model = lgb.train(params, d_train, 10000, valid_sets=[d_test],
early_stopping_rounds=50, verbose_eval=1000)
```

```

↳ Training until validation scores don't improve for 50 rounds.
[1000]  valid_0's binary_logloss: 0.344903
[2000]  valid_0's binary_logloss: 0.334606
[3000]  valid_0's binary_logloss: 0.32872
[4000]  valid_0's binary_logloss: 0.325002
[5000]  valid_0's binary_logloss: 0.322382
Early stopping, best iteration is:
[5889]  valid_0's binary_logloss: 0.320531

```

Figure 5.8: LightGBM model Output

Next, we will visualise our results with the advanced AI technique with the help of Shap to have a better visibility.

5.6 Model Analysis with advance Visualization along Shap Tool

In this section, we will visualize the different patterns and analysis of our model results. Now, we will compute the SHAP values and the SHAP interaction values for the first 20 test observations.

```

explainer = shap.TreeExplainer(model)
expected_value = explainer.expected_value
if isinstance(expected_value, list):
    expected_value = expected_value[1]
print(f"Explainer expected value: {expected_value}")

select = range(20)
features = X_test.iloc[select]
features_display = X_display.loc[features.index]

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    shap_values = explainer.shap_values(features)[1]
    shap_interaction_values =
        explainer.shap_interaction_values(features)
if isinstance(shap_interaction_values, list):
    shap_interaction_values = shap_interaction_values[1]

↳ Setting feature_perturbation = "tree_path_dependent" because no background data was given.
Explainer expected value: [-0.22869578]

```

Figure 5.9: Shap calculation Values

So, now the explainer value is -0.22; next, we see the patterns and relation analysis.

5.6.1 Basic decision plot features

Refer to the following decision plot of the 20 test observations; note that this plot isn't informative by itself; we use it only to illustrate the primary concepts:

- The x-axis represents the model's output. In this case, the units are log odds.
- The plot is centered on the x-axis at `explainer.expected_value`. All the SHAP values are relative to the model's expected value, like a linear model's effects are relative to the intercept.
- The y-axis lists the model's features. By default, the features are ordered by its descending importance. The importance is calculated over the observations plotted. This is usually different from the importance ordering for the entire dataset. In addition to the feature importance ordering, the decision plot also supports the hierarchical cluster feature ordering and the user-defined feature ordering.
- Each observation's prediction is represented by a colored line. At the top of the plot, each line strikes the x-axis at its corresponding observation's predicted value. This value determines the color of the line on a spectrum.
- Moving from the bottom of the plot to the top, the SHAP values for each feature are added to the model's base value. This shows how each feature contributes to the overall prediction.
- At the bottom of the plot, the observations converge at `explainer.expected_value`.

```
shap.decision_plot(expected_value, shap_values,  
features_display)
```

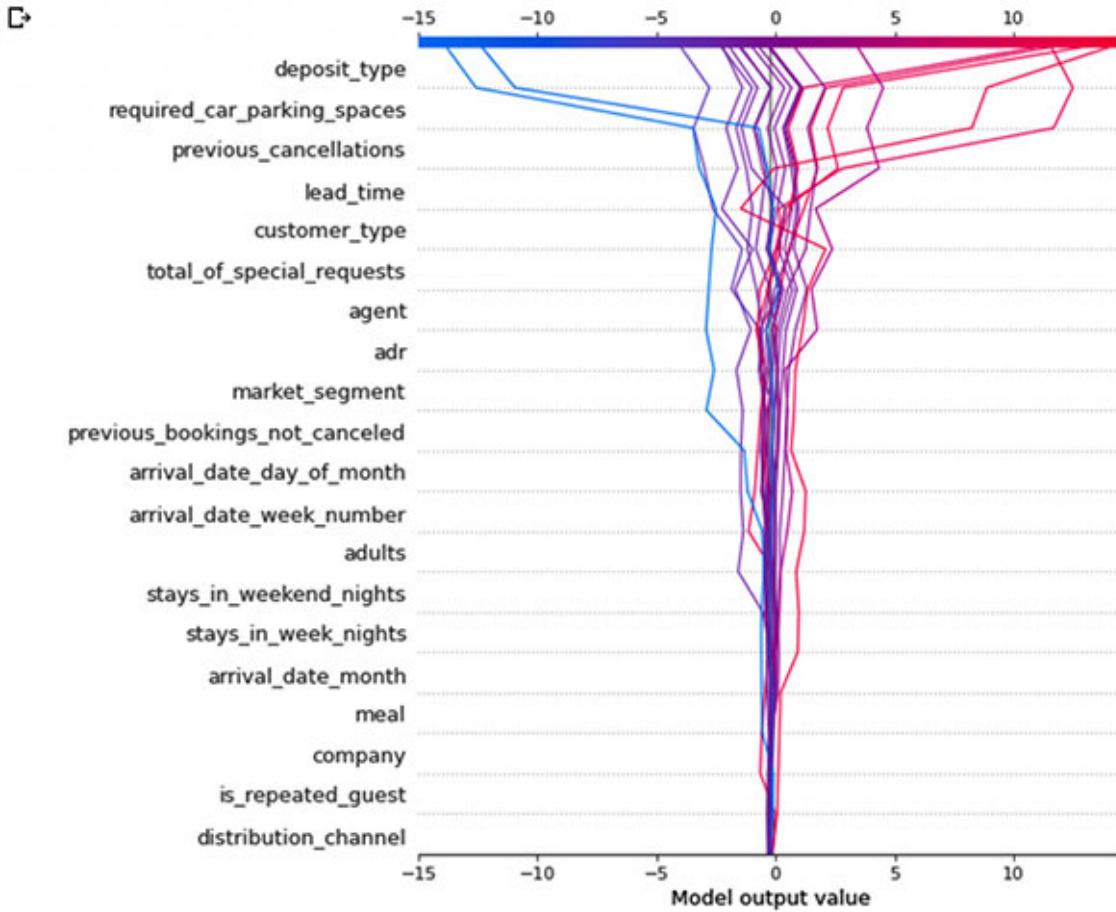


Figure 5.10: SHAP Model Output Values

Now, we can see from the preceding screenshot that the expected value is at -0.22, and at the bottom of the plot, the observations converges. Like the force plot, the decision plot supports `link='logit'` to transform the log odds to probabilities.

```
shap.decision_plot(expected_value, shap_values, features_display,
link='logit')
```

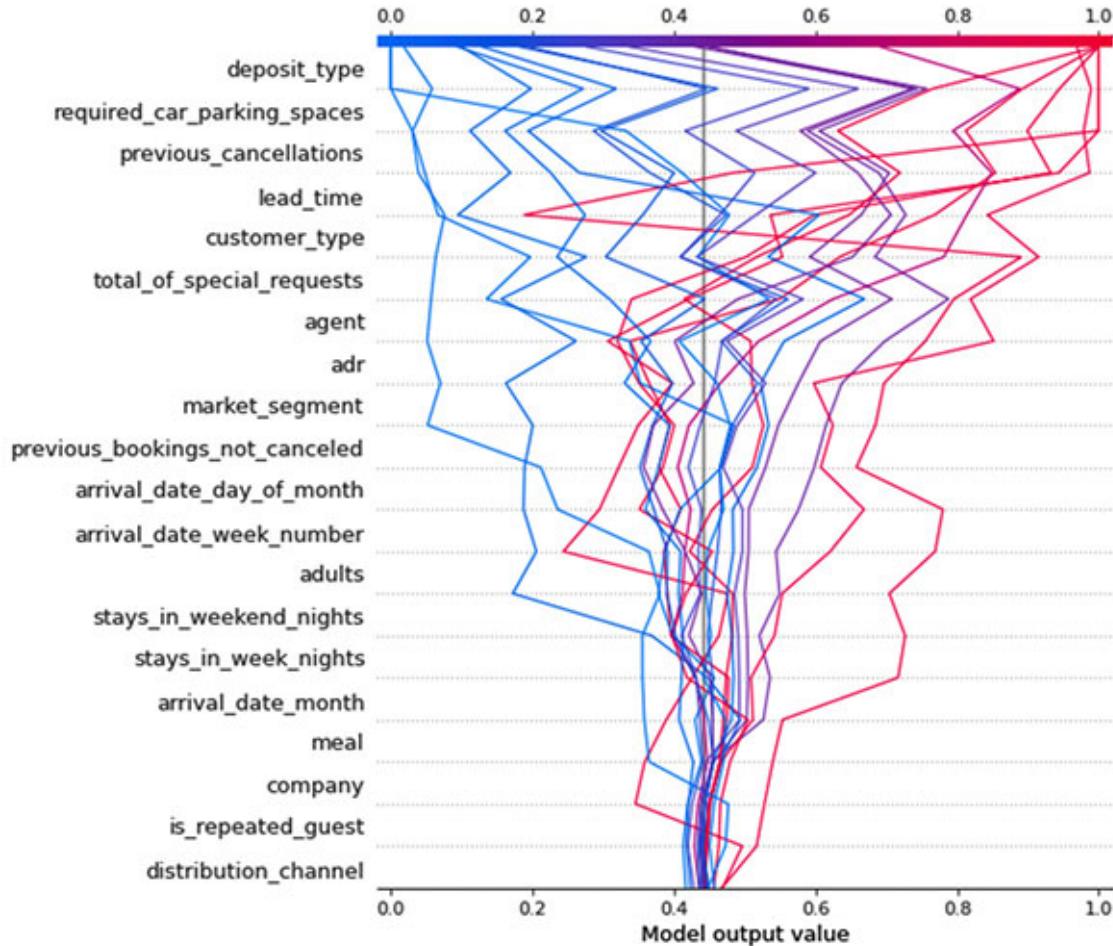


Figure 5.11: SHAP Model Log transform output

The cumulative effect of interactions:

The decision plots support the SHAP interaction values. So, from the tree-based models, the first-order interactions were estimated. The SHAP dependence plots provides the interactions of the individual's visualize, which plot a decision plot to show the cumulative effect of the main effects and interactions with one or another observation among the data.

5.6.2 Force Plots Analysis

The observations can be highlighted using a dotted line style. Here, we highlighted a misclassified observation. Our naive cutoff point is zero log odds (probability 0.5).

```
y_pred = (shap_values.sum(1) + expected_value) > 0
```

```

misclassified = y_pred != y_test[:20]
shap.decision_plot(expected_value, shap_values, features_display,
link='logit', highlight=misclassified)

```

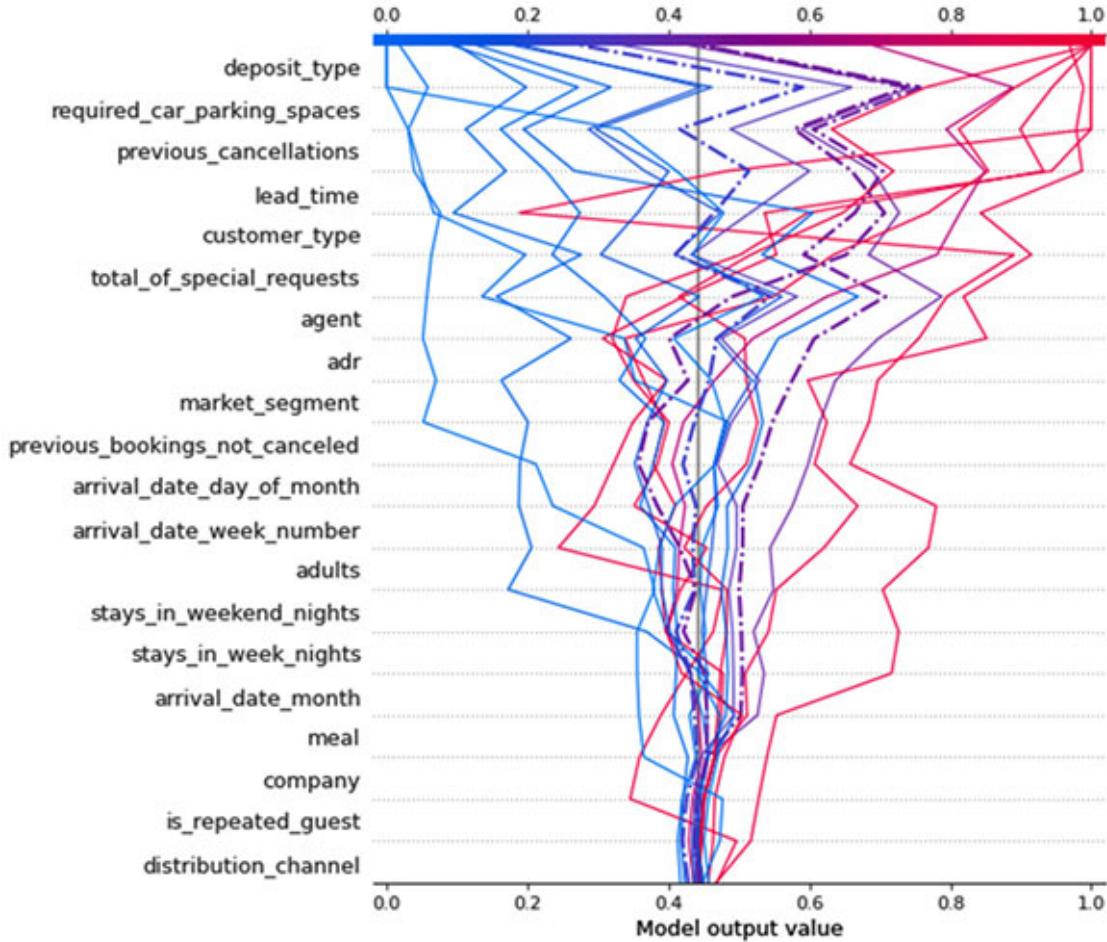


Figure 5.12: SHAP Model Naïve cut threshold graph

Let's inspect the misclassified observation by plotting it alone. When a single observation is plotted, its corresponding feature values are displayed. Notice that the shape of the line has changed. Why? The feature order has changed on the y-axis, based on the feature importance for this line observation. The section on "*Preserving order and scale between plots*" shows how to use the same feature order for multiple plots.

```

shap.decision_plot(expected_value, shap_values[misclassified],
features_display[misclassified], link='logit', highlight=0)

```

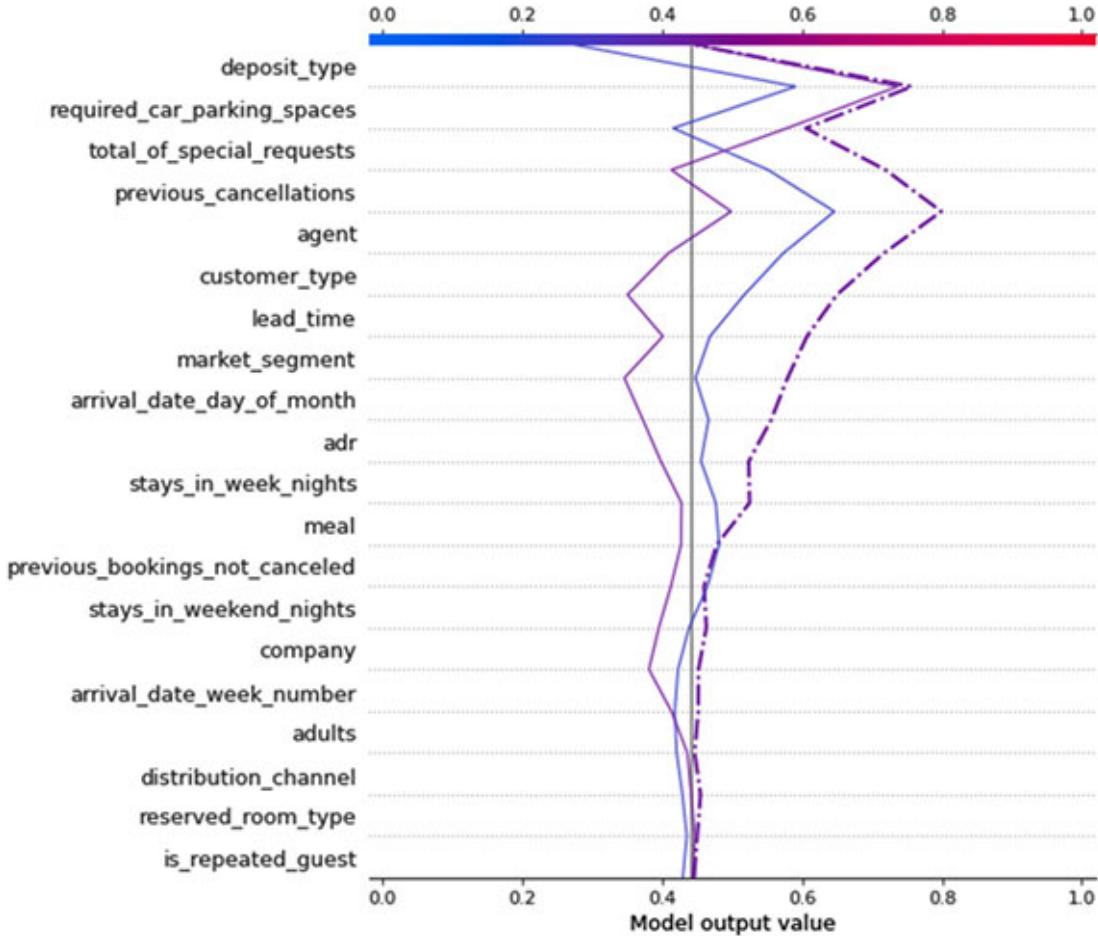


Figure 5.13: SHAP Misclassified output

A force plot for the misclassified observation is shown as follows. In this case, the decision plot and the force plot are both effective at showing how the model arrived at its decision.

```
%matplotlib inline
shap.initjs()
shap.force_plot(expected_value, shap_values[misclassified],
features_display[misclassified],
link='logit', matplotlib=False, show=True)
```

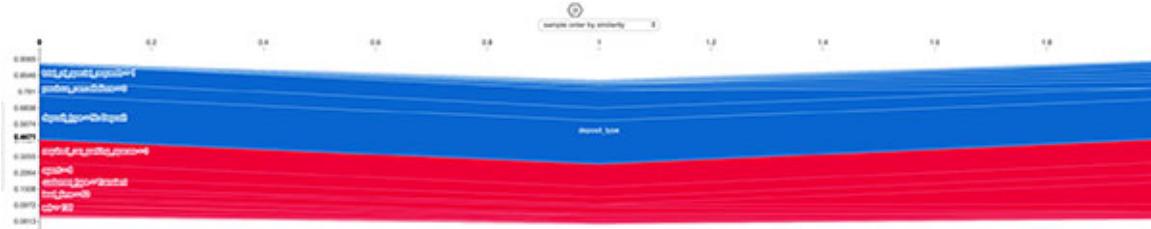


Figure 5.14: SHAP Force plot decision misclassified

From the preceding JavaScript dynamic visualization for the misclassified analysis, we can figure out the manual interpretation by seeing those results. The focus of this section is to build the architecture and how to play with our dataset.

5.7 TensorFlow Estimator Model Framework Building

In this section, we will work on building the TensorFlow Estimator model and then we will analyse the Model analysis with the Tensorboard and what-if tool. Next, we will build the dataset for training by splitting the datasets.

```
X = Raw_Data.drop(["is_canceled"], axis=1)
y = Raw_Data["is_canceled"]
from sklearn.model_selection import train_test_split
xTrain, xTest, yTrain, yTest = train_test_split(X, y, test_size =
0.2, random_state = 0)
xTrain['is_canceled']=yTrain
xTrain.reset_index(inplace=True, drop=True)
xTrain.head()
```

| Lead_time | Arrival_date_week_number | Arrival_date_day_of_month | Stayed_in_weekend_nights | Stayed_in_week_nights | Adults | Children | Babies | Is_repeated_guest | Previous_cancellations | Previous_bookings_not_canceled | Agent | Company | Required_car_parking |
|-----------|--------------------------|---------------------------|--------------------------|-----------------------|--------|----------|--------|-------------------|------------------------|--------------------------------|-------|---------|----------------------|
| 0 | 66 | 32 | 6 | 2 | 2 | 3 | 0.0 | 0 | 0 | 0 | 0.0 | 0.0 | 0.0 |
| 1 | 51 | 36 | 21 | 1 | 4 | 2 | 0.0 | 0 | 0 | 0 | 0.0 | 0.0 | 0.0 |
| 2 | 22 | 38 | 15 | 0 | 1 | 1 | 0.0 | 0 | 0 | 0 | 0.0 | 0.0 | 0.0 |
| 3 | 69 | 16 | 17 | 3 | 3 | 2 | 2.0 | 0 | 0 | 0 | 0.0 | 0.0 | 0.0 |
| 4 | 14 | 17 | 25 | 0 | 4 | 2 | 2.0 | 0 | 0 | 0 | 0 | 14.0 | 0.0 |

Figure 5.15: Table of House Booking Dataset

Next, we will create some Utility Function which we will use for the TensorFlow Estimator Model Building.

Here, it will create a **tf** feature spec from the dataframe and columns specified.

```

def create_feature_spec(df, columns=None):
    feature_spec = {}
    if columns == None:
        columns = df.columns.values.tolist()
    for f in columns:
        if df[f].dtype is np.dtype(np.int64):
            feature_spec[f] = tf.io.FixedLenFeature(shape=(),
                dtype=tf.int64)
        elif df[f].dtype is np.dtype(np.float64):
            feature_spec[f] = tf.io.FixedLenFeature(shape=(),
                dtype=tf.float32)
        else:
            feature_spec[f] = tf.io.FixedLenFeature(shape=(),
                dtype=tf.string)
    return feature_spec

```

Create simple numeric and categorical feature columns from a feature spec and a list of columns from that spec to use. The models might perform better with some feature engineering such as the bucketed numeric columns and hash-bucket/embedding columns for the categorical features.

```

def create_feature_columns(columns, feature_spec):
    ret = []
    for col in columns:
        if feature_spec[col].dtype is tf.int64 or
        feature_spec[col].dtype is tf.float32:
            ret.append(tf.feature_column.numeric_column(col))
        else:
            ret.append(tf.feature_column.indicator_column(
                tf.feature_column.categorical_column_with_vocabulary_list(c
                ol, list(xTrain[col].unique()))))
    return ret

```

The following is an input function for providing the input to a model from **tf.Examples**.

```

def tfexamples_input_fn(examples, feature_spec, label,
mode=tf.estimator.ModeKeys.EVAL,
    num_epochs=None, batch_size=64):
    def ex_generator():
        for i in range(len(examples)):
            yield examples[i].SerializeToString()
    dataset = tf.data.Dataset.from_generator(
        ex_generator, tf.dtypes.string, tf.TensorShape([]))
    if mode == tf.estimator.ModeKeys.TRAIN:
        dataset = dataset.shuffle(buffer_size=2 * batch_size + 1)
    dataset = dataset.batch(batch_size)
    dataset = dataset.map(lambda tf_example:
        parse_tf_example(tf_example, label, feature_spec))
    dataset = dataset.repeat(num_epochs)
    return dataset

```

Next, we will parse `Tf.Example` protos into the features for the input function.

```

def parse_tf_example(example_proto, label, feature_spec):
    parsed_features = tf.io.parse_example(serialized=example_proto,
    features=feature_spec)
    target = parsed_features.pop(label)
    return parsed_features, target

```

Here, it will convert a dataframe into a list of `tf.Example` protos.

```

def df_to_examples(df, columns=None):
    examples = []
    if columns == None:
        columns = df.columns.values.tolist()
    for index, row in df.iterrows():
        example = tf.train.Example()
        for col in columns:
            if df[col].dtype is np.dtype(np.int64):
                example.features.feature[col].int64_list.value.append(int(r
ow[col]))

```

```

    elif df[col].dtype is np.dtype(np.float64):
        example.features.feature[col].float_list.value.append(row[col])
    elif row[col] == row[col]:
        example.features.feature[col].bytes_list.value.append(row[col].encode('utf-8'))
    examples.append(example)
return examples

```

Next, this function converts a dataframe column into a column of 0's and 1's based on the provided test. It is used to force the label columns to be numeric for the binary classification using a TF estimator.

```

def make_label_column_numeric(df, label_column, test):
    df[label_column] = np.where(test(df[label_column]), 1, 0)

```

The following are the steps which we will perform prior to the training:

- Specify the input columns and target predictor and set the column in the dataset that you wish for the model to predict:

```

label_column = 'is_canceled'
input_features = ['lead_time', 'arrival_date_week_number',
'arrival_date_day_of_month',
'stays_in_weekend_nights', 'stays_in_week_nights',
'adults', 'children', 'babies', 'is_repeated_guest',
'previous_cancellations', 'previous_bookings_not_canceled',
'agent', 'company', 'required_car_parking_spaces',
'total_of_special_requests', 'adr', 'hotel',
'arrival_date_month', 'meal',
'market_segment', 'distribution_channel',
'reserved_room_type', 'deposit_type', 'customer_type']

# Create a list containing all input features and the label column
features_and_labels = input_features + [label_column]

```

- Convert the dataset to **tf.Example** protos:

```

examples = df_to_examples(xTrain)

• Create and train the linear classifier and a feature spec for the classifier:

num_steps = 2000
feature_spec = create_feature_spec(xTrain,
features_and_labels)
feature_spec

{
    'adr': FixedLenFeature(shape=(), dtype=tf.float32, default_value=None),
    'adults': FixedLenFeature(shape=(), dtype=tf.int64, default_value=None),
    'agent': FixedLenFeature(shape=(), dtype=tf.float32, default_value=None),
    'arrival_date_day_of_month': FixedLenFeature(shape=(), dtype=tf.int64, default_value=None),
    'arrival_date_month': FixedLenFeature(shape=(), dtype=tf.string, default_value=None),
    'arrival_date_week_number': FixedLenFeature(shape=(), dtype=tf.int64, default_value=None),
    'babies': FixedLenFeature(shape=(), dtype=tf.int64, default_value=None),
    'children': FixedLenFeature(shape=(), dtype=tf.float32, default_value=None),
    'company': FixedLenFeature(shape=(), dtype=tf.float32, default_value=None),
    'customer_type': FixedLenFeature(shape=(), dtype=tf.string, default_value=None),
    'deposit_type': FixedLenFeature(shape=(), dtype=tf.string, default_value=None),
    'distribution_channel': FixedLenFeature(shape=(), dtype=tf.string, default_value=None),
    'hotel': FixedLenFeature(shape=(), dtype=tf.string, default_value=None),
    'is_canceled': FixedLenFeature(shape=(), dtype=tf.int64, default_value=None),
    'is_repeated_guest': FixedLenFeature(shape=(), dtype=tf.int64, default_value=None),
    'lead_time': FixedLenFeature(shape=(), dtype=tf.int64, default_value=None),
    'market_segment': FixedLenFeature(shape=(), dtype=tf.string, default_value=None),
    'meal': FixedLenFeature(shape=(), dtype=tf.string, default_value=None),
    'previous_bookings_not_canceled': FixedLenFeature(shape=(), dtype=tf.int64, default_value=None),
    'previous_cancellations': FixedLenFeature(shape=(), dtype=tf.int64, default_value=None),
    'required_car_parking_spaces': FixedLenFeature(shape=(), dtype=tf.int64, default_value=None),
    'reserved_room_type': FixedLenFeature(shape=(), dtype=tf.string, default_value=None),
    'stays_in_week_nights': FixedLenFeature(shape=(), dtype=tf.int64, default_value=None),
    'stays_in_weekend_nights': FixedLenFeature(shape=(), dtype=tf.int64, default_value=None),
    'total_of_special_requests': FixedLenFeature(shape=(), dtype=tf.int64, default_value=None)
}

```

Figure 5.16: Feature Columns Tensors and datatypes

```

train_inpf = functools.partial(tfexamples_input_fn, examples,
feature_spec, label_column)

```

5.7.1 TensorFlow Estimator Model

Here, we will build the Model with the two estimators, Model Linear Classifier and DNN Classifier. For further reading, check out the following link:

https://www.tensorflow.org/api_docs/python/tf/estimator

Now, we trained the Model first with the Linear Model Classifier estimator.

```

classifier = tf.estimator.LinearClassifier(
    feature_columns=create_feature_columns(input_features,
    feature_spec))
classifier.train(train_inpf, steps=num_steps)

```

```

INFO:tensorflow:global_step/sec: 42.7867
INFO:tensorflow:loss = 0.4609873, step = 1400 (2.337 sec)
INFO:tensorflow:global_step/sec: 44.1391
INFO:tensorflow:loss = 0.3516879, step = 1500 (2.269 sec)
INFO:tensorflow:global_step/sec: 43.212
INFO:tensorflow:loss = 0.5797381, step = 1600 (2.315 sec)
INFO:tensorflow:global_step/sec: 43.3536
INFO:tensorflow:loss = 0.6372526, step = 1700 (2.305 sec)
INFO:tensorflow:global_step/sec: 42.7713
INFO:tensorflow:loss = 0.42908752, step = 1800 (2.337 sec)
INFO:tensorflow:global_step/sec: 44.9273
INFO:tensorflow:loss = 0.5124675, step = 1900 (2.227 sec)
INFO:tensorflow:Calling checkpoint listeners before saving checkpoint 2000...
INFO:tensorflow:Saving checkpoints for 2000 into /tmp/tmplmg3f5o4/model.ckpt.
INFO:tensorflow:Calling checkpoint listeners after saving checkpoint 2000...
INFO:tensorflow:Loss for final step: 0.5286524.
<tensorflow.estimator.python.estimator.canned.linear.LinearClassifierV2 at 0x7fb06287ff60>

```

Figure 5.17: Estimator Output Result for Linear Classifier

Next, we will train our second model DNN Classifier as follows:

```

#title Create and train the DNN classifier {display-mode: "form"}
num_steps_2 = 2000
classifier1 = tf.estimator.DNNClassifier(
    feature_columns=create_feature_columns(input_features,
    feature_spec),
    hidden_units=[128, 64, 32])
classifier1.train(train_inpf, steps=num_steps_2)

INFO:tensorflow:global_step/sec: 44.8722
INFO:tensorflow:loss = 0.6890626, step = 1600 (2.227 sec)
INFO:tensorflow:global_step/sec: 45.8162
INFO:tensorflow:loss = 0.49594885, step = 1700 (2.184 sec)
INFO:tensorflow:global_step/sec: 45.3199
INFO:tensorflow:loss = 0.52540064, step = 1800 (2.206 sec)
INFO:tensorflow:global_step/sec: 43.919
INFO:tensorflow:loss = 0.5785385, step = 1900 (2.280 sec)
INFO:tensorflow:Calling checkpoint listeners before saving checkpoint 2000...
INFO:tensorflow:Saving checkpoints for 2000 into /tmp/tmpdsixjqdc/model.ckpt.
INFO:tensorflow:Calling checkpoint listeners after saving checkpoint 2000...
INFO:tensorflow:Loss for final step: 0.6581625.
<tensorflow.estimator.python.estimator.canned.dnn.DNNClassifierV2 at 0x7fb057f2fef0>

```

Figure 5.18: Estimator Output Result for DNN Classifier

In the preceding example, we trained both our TensorFlow Estimator Model Linear Classifier and DNN classifier; next, we will check out some cool Model evaluation products to analyse our results.

5.8 Advance Visualization for TensorFlow Model with Tensorboard & What-IF Tool

In this section, we will conduct the TensorFlow Model evaluation with what-if and tensorboard tools.

5.8.1 Tensorboard

Now, we will visualize our Model results and logs in Tensorboard for the DNN classifier.

```
%load_ext tensorboard  
%tensorboard --logdir=/tmp/
```

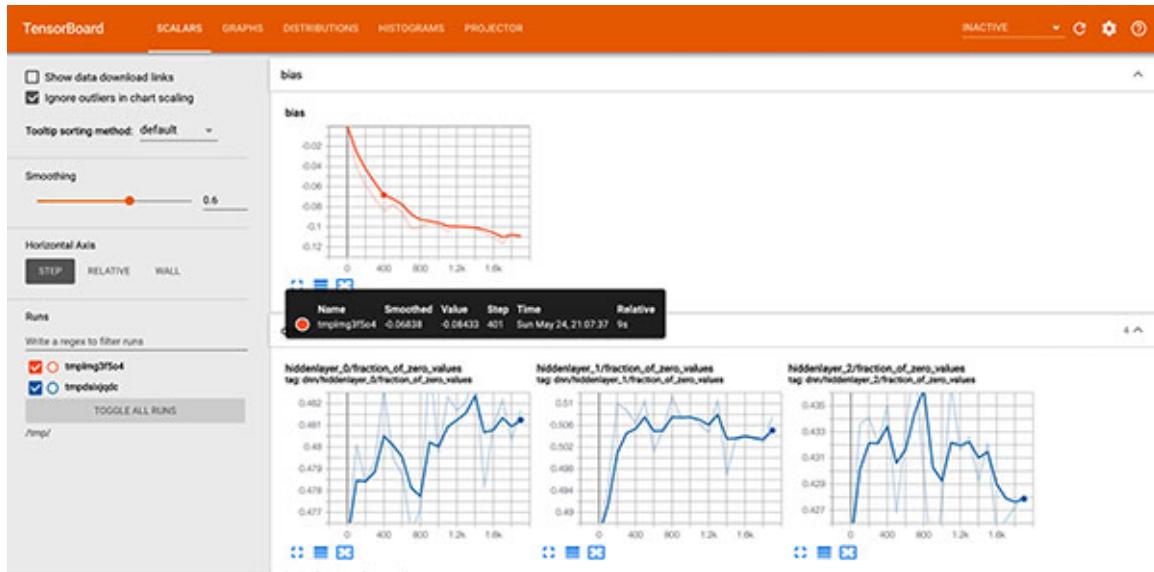


Figure 5.19: DNN Estimator Model Evaluation scalar results

Here, this Scaler Tab will tell us the model evaluation results like the accuracy, bias, Global step, Loss and so on.

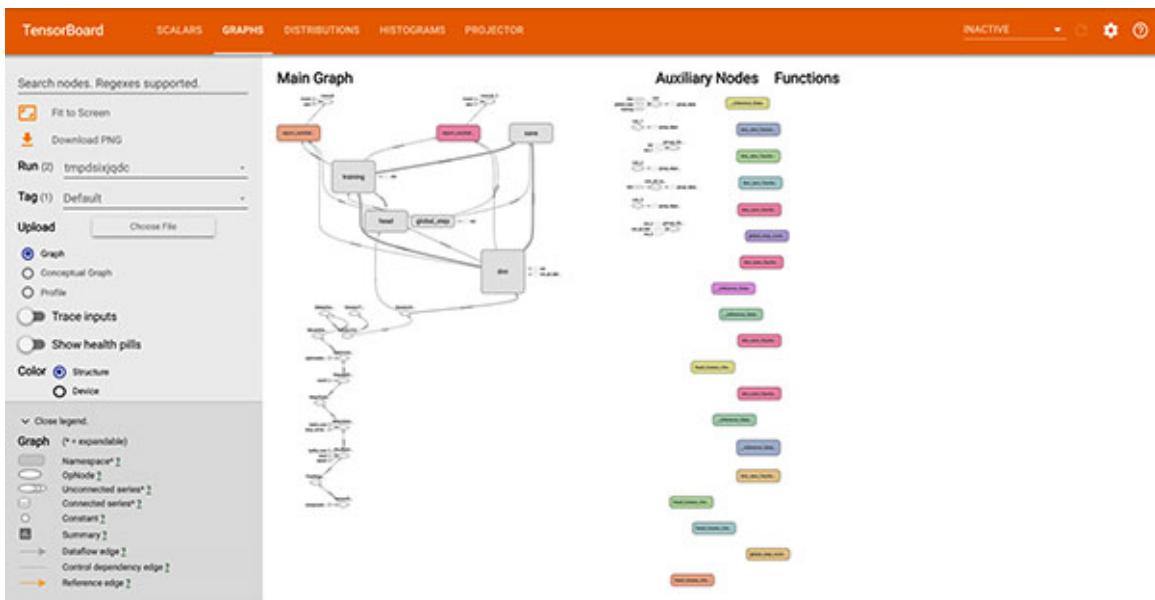


Figure 5.20: DNN Estimator Model Evaluation graphs

The preceding screenshot shows us the model graphs and computation of matrix multiplication in tensors in a graphical visualisation.

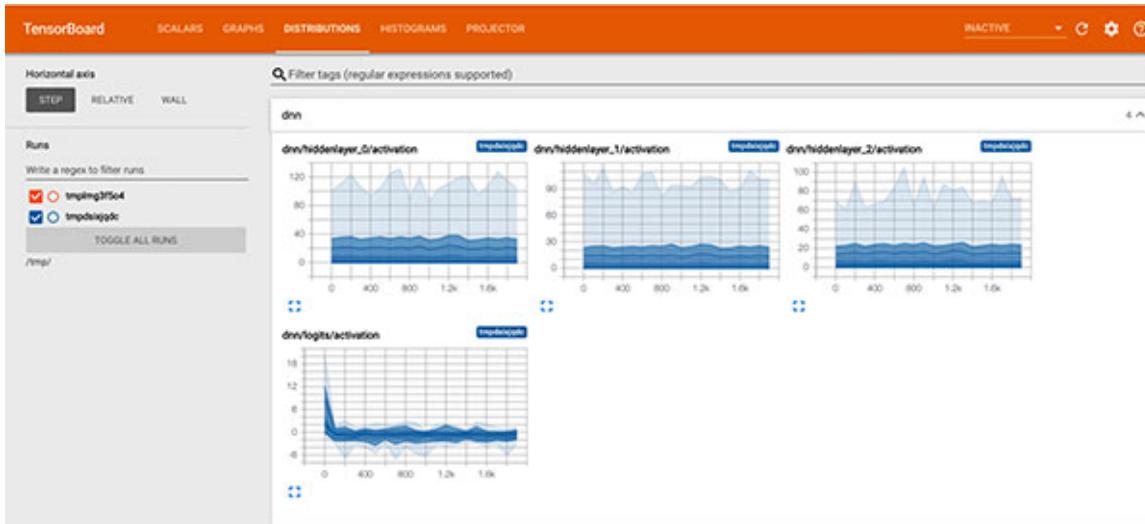


Figure 5.21: DNN Estimator Model Layer weights

In this preceding model, the layer activation function and weights are projected in dense plots in the distribution tab.

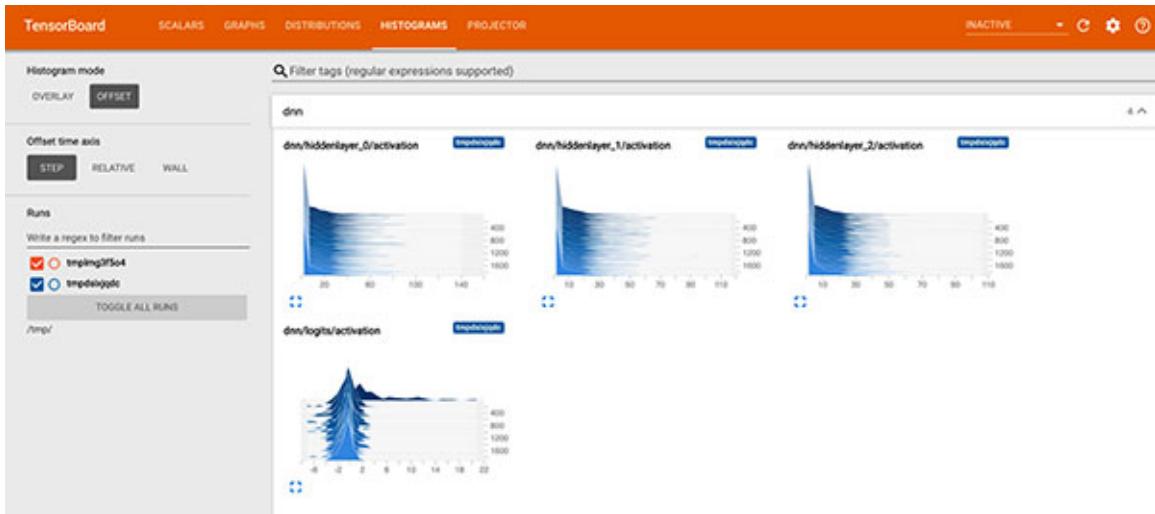


Figure 5.22: DNN Estimator Model Histograms

Here, the frequency in epochs at which the activation will compute the histograms for the layers of the model is shown in the **HISTOGRAMS** tab.

5.8.2 What-If Tool

Now, we will prepare the test dataset with the Model evaluation before the deployment of how the Model is performing.

```
xTest['is_canceled']=yTest
xTest.reset_index(inplace=True, drop=True)
```

Next, we will import our library and transform or pre-process our test dataset with the help of our util function **df_to_examples**, and set the pixel and height of dynamic Visualization.

```
# title Invoke What-If Tool for test data and the trained models
{display-mode: "form"}
num_datapoints = 2000
tool_height_in_px = 1000

from witwidget.notebook.visualization import WitConfigBuilder
from witwidget.notebook.visualization import WitWidget

# Load up the test dataset
test_examples = df_to_examples(xTest[:2000])
```

Next, set up the tool with the test examples and the trained classifier for the what-if visualization.

```
config_builder =
WitConfigBuilder(test_examples).set_estimator_and_feature_spec(
    classifier,
    feature_spec).set_compare_estimator_and_feature_spec(
    classifier1, feature_spec).set_label_vocab(['Check-
Out', 'Canceled'])
a = WitWidget(config_builder, height=tool_height_in_px)
```

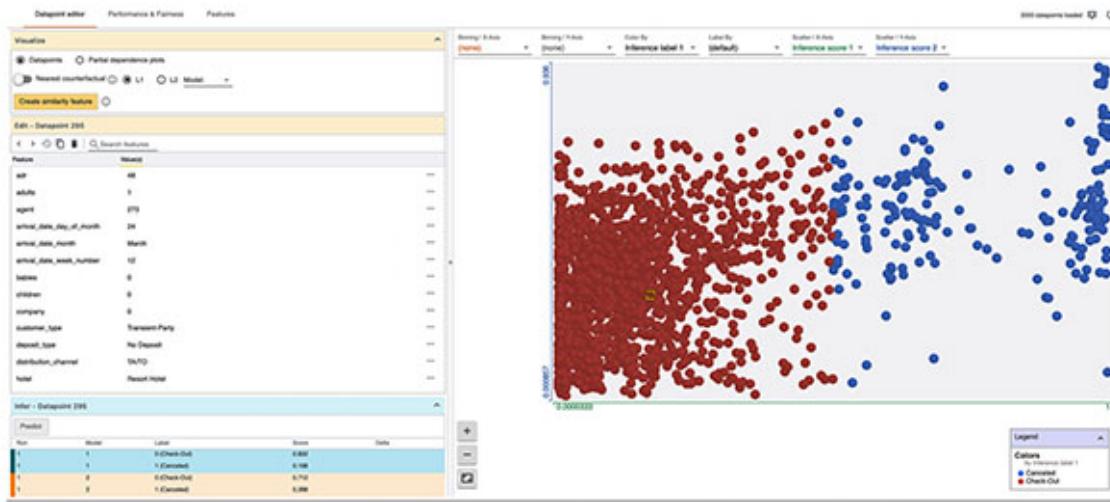


Figure 5.23: Dynamic Datapoint editor for prediction dataset

Now, the preceding screenshot shows the dynamic datapoint editor and all its information and we can play around to check our data analysis.

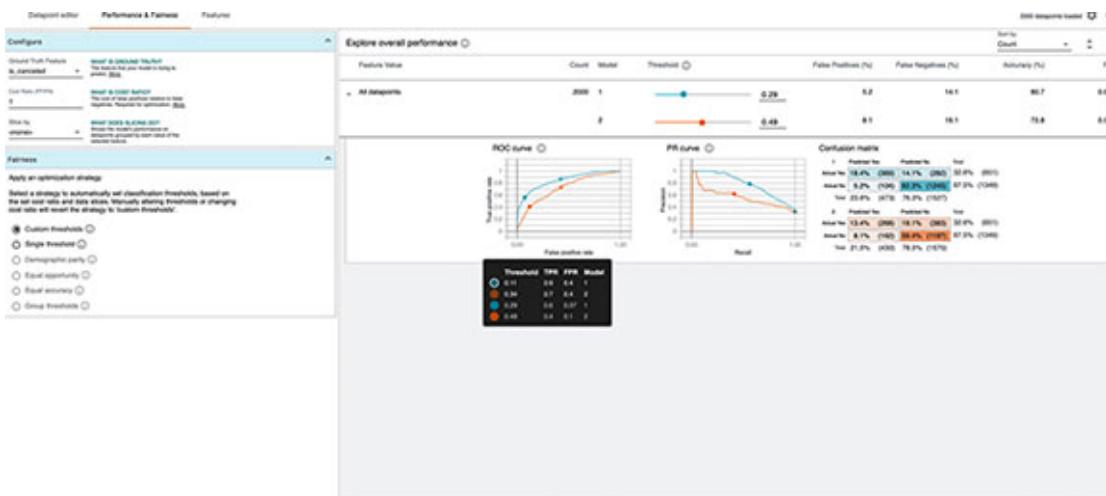


Figure 5.24: Performance and fairness for all features

Next, from the tab in the Performance and Fairness, we can have a look at the model evaluation and compare the study between the two classifier models and its PR curve and ROC with the Confusion Matrix.

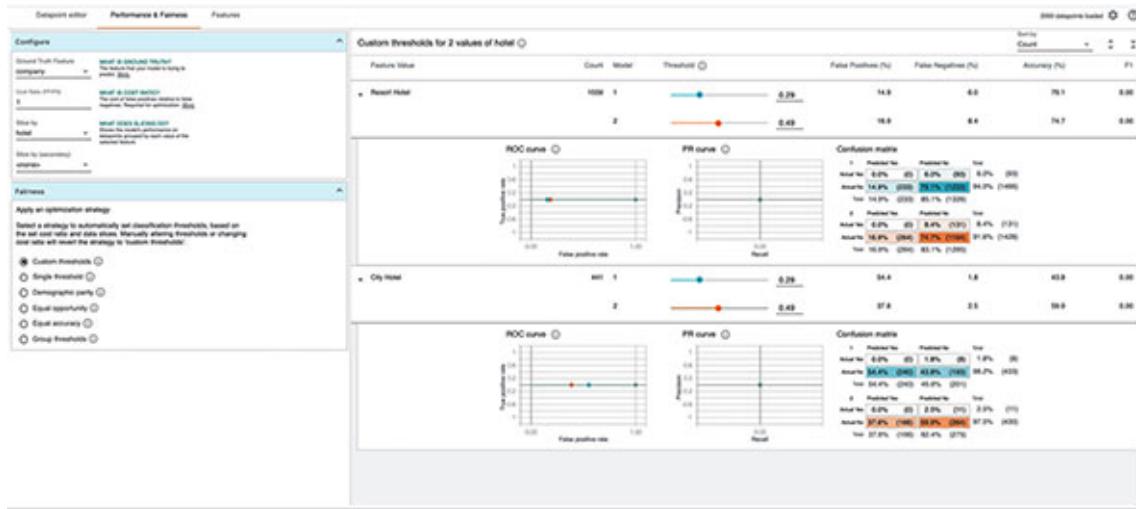


Figure 5.25: Multi-slicing ROC/PR curve for two features

We can slice the datapoints and create some buckets to see the interdependent feature's evaluation. In the preceding screenshot, we used the multi-slicing strategy, its analysis on the two features, and its definition on the Confusion matrix and Fairness stud.

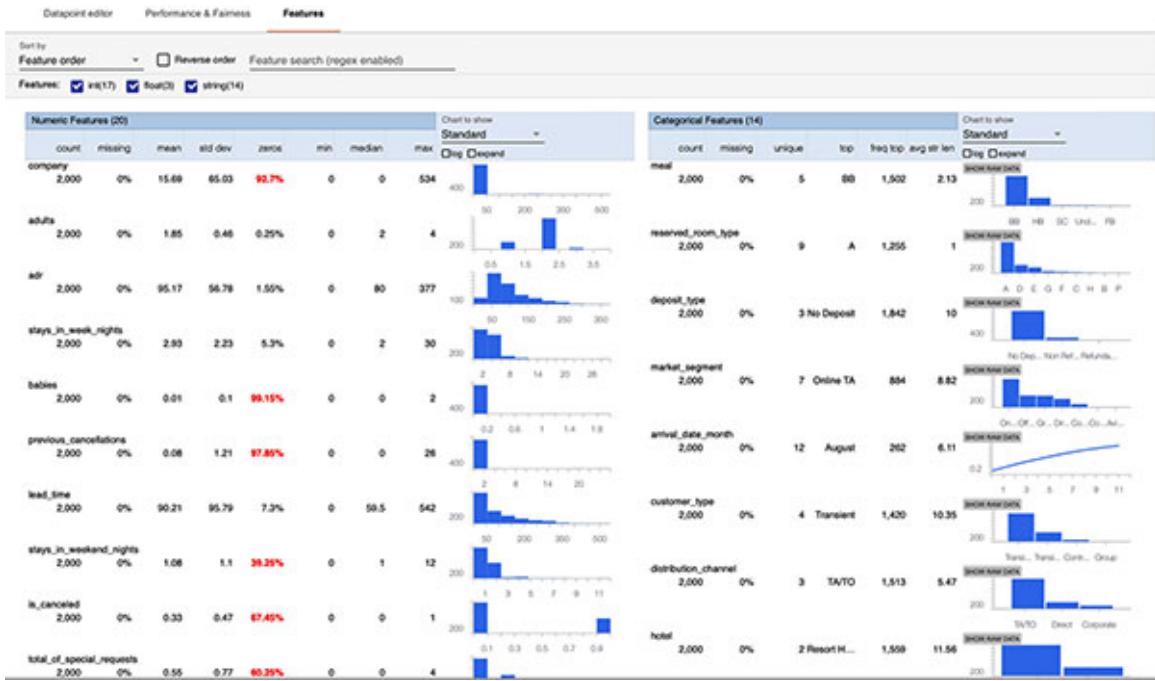


Figure 5.26: Features Statistics of Categorical and Numerical

At last, in the preceding screenshot, we can see the feature distribution for both the numerical and the categorical values and its statistical nature and behaviour.

5.9 Conclusion

In this chapter, we learned about the end-to-end advance visualization for our TensorFlow model evaluation with the various tools like Tensorboard and what-if.

We also learned how to build the advance Boosting algorithm LightGBM and use the model endpoint to work on the feature analysis with the advance Visualization technique to explain the decision plots for our Model, like Feature plot, summary plots, Expected Shap plots and Misclassified plots, and so on. Furthermore, we have built a TensorFlow estimator models, like the DNN Classifier and visualized our Dense layers and evaluated the histogram and Logs with Tensorboard. We also learned how to use the what-if tool to visualise the dynamic comparative study between the two TensorFlow models prior to the deployment on the test data, which includes the Datapoint editor, Performance with Fairness (ROC Curve, Confusion matrix, and so on), and finally the feature distribution analysis tab.

5.10 References

- https://shap.readthedocs.io/en/latest/example_notebooks/plots/decision_plot.html
- <https://christophm.github.io/interpretable-ml-book/preface-by-the-author.html>
- <https://www.tensorflow.org/tutorials/estimator/premade>
- <https://pair-code.github.io/what-if-tool/>
- https://www.tensorflow.org/tensorboard/get_started
- <https://lightgbm.readthedocs.io/en/latest/Features.html>

CHAPTER 6

Building Weights & Biases Pipeline Development

In this chapter, we will build an end-to-end LightGBM Model framework, and will monitor the model performance in the Weight & Biases (Wandb) tool. Inside Weights & Biases, we will see the live model RMSE graphs and parallel coordinates' hyper parameter performance graphs for each iteration. Next, we will deploy the model with the KF serving in our Kubernetes Cluster inside Google Cloud Platform. Then, we will be serving model endpoint which will be used for prediction and monitored in the Grafana Dashboard, such as Model Rate request with respect to the time and CPU and GPU consumption.

Structure

In this chapter, we will cover the following topics:

- Problem statement
- Setup of project requirements in GCP & Wandb
- Introduction on the Weight & Biases usage
- Modelling and training the LightGBM Model for Equity Stocks Data
- Serving the Model with KF Serving in Kubernetes Cluster
- Monitoring the Performance with Grafana Dashboard

Objectives

After studying this chapter, we will be able to understand the following:

- How to set use Docker and Kubernetes.
- How to build the individual pipeline components like training and model evaluation with Hyperparameter in the Weights & Biases tool.
- How to serve the Model with KF serving and predict the model request and monitor with the Grafana Dashboard.
- How to use Kubernetes and many other Google Cloud Platform to leverage the power of Machine learning with DevOps Knowledge.

6.1 Problem statement

The data is cleaned, regularized, and encrypted in global equity data. The first 21 columns (feature1 - feature21) are features, and target is the binary class you're trying to predict.

In the provided **training_data**, each id corresponds to a stock with a set of obfuscated features. The target represents future performance. The rows are grouped into eras that represent different points in time. Your goal is to train a machine learning model to predict the target, given new features, and we will tell whether the equity is good or bad. Let's take a glimpse at the dataset:

| | feature1 | feature2 | feature3 | feature4 | feature5 | feature6 | feature7 | feature8 | feature9 | feature10 | feature11 | feature12 | feature13 | feature14 | feature15 | feature16 |
|---|----------|----------|----------|----------|----------|----------|----------|----------|----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| 0 | 0.137662 | 0.445825 | 0.471079 | 0.279196 | 0.275892 | 0.861967 | 0.305975 | 0.222414 | 0.893704 | 0.475781 | 0.071068 | 0.640546 | 0.024044 | 0.703299 | 0.128572 | 0.0634 |
| 1 | 0.651766 | 0.242053 | 0.720764 | 0.784000 | 0.685828 | 0.345841 | 0.038447 | 0.326108 | 0.760536 | 0.741738 | 0.741064 | 0.607355 | 0.544590 | 0.449178 | 0.547613 | 0.8631 |

Figure 6.1: Stocks feature data

| | |
|-------------|---|
| NOTE | Rest all the imports I have showed in my Colab Notebook, for which the hyperlink of GitHub Account of this chapter is given below. Note Colab platform Python 3.x. RUN IN GOOGLE COLAB |
| CODE | https://github.com/bpbpublications/Continuous-Machine-Learning-with-Kubeflow/tree/main/Chapter6 |

6.2 Setup of project requirements in GCP & Wandb

Prerequisites: You must have an active GCP account, and while you practice this chapter, it might charge for running the Kubernetes cluster. I am running all the codes in MacOS.

I think some basic Kubernetes and Docker knowledge is a must.

6.2.1 Kubeflow Cluster in GCP and Docker setup

We have already deployed Kubeflow in the Kubernetes Cluster inside the Google Cloud Platform and installed the Docker in Local in previous chapters.

6.2.2 Kaggle API setup for downloading data

In this section, we will see how to get the Kaggle API, and use that to download the data directly from the Kaggle website. So, a user must have a Kaggle account; if not, please create an account by clicking on the following link:

<https://www.kaggle.com/>

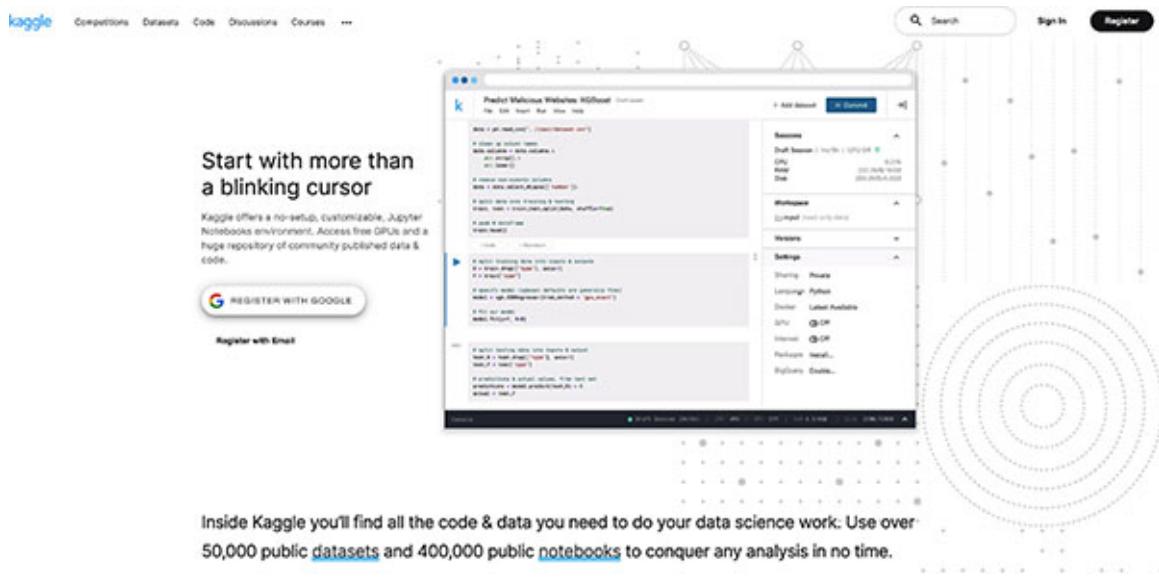


Figure 6.2: Kaggle website

- **Installing Kaggle API:**

You can run `pip install Kaggle` to install the API. You might need to run `pip install --user Kaggle` on Linux or Mac if you are encountering issues with the installation.

<https://github.com/Kaggle/kaggle-api>

- **Setting up API Key:**

Now, go to your Kaggle account tab <https://www.kaggle.com/<username>/account> and click on '**Create API Token**'. A file named `kaggle.json` will be downloaded.

Next, move the file into the `~/.kaggle/` folder in Mac or Linux and to `C:\Users\.kaggle\` for Windows.

Alternatively, you can populate the `KAGGLE_USERNAME` and `KAGGLE_KEY` environment variables with the values from `kaggle.json` to get the API to authenticate. Here, we will be using the Google Colab for training the model; now we will upload the `Kaggle.json` there.

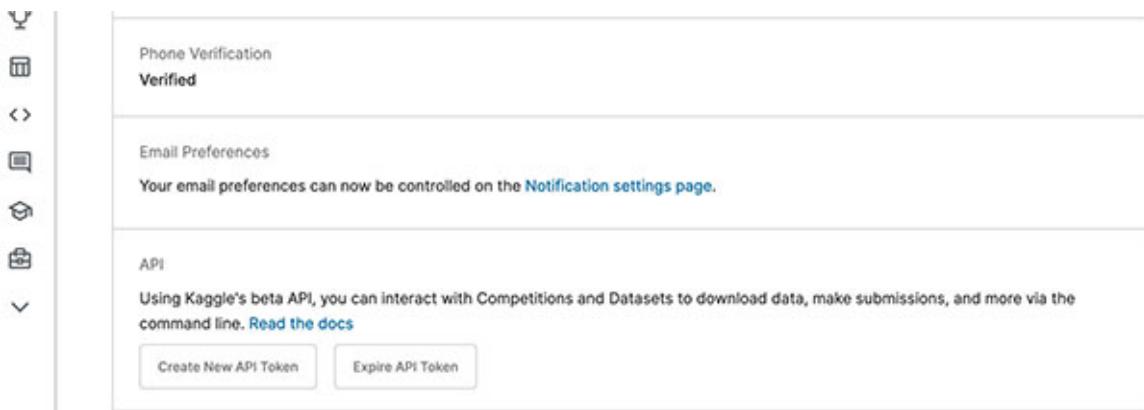


Figure 6.3: Kaggle Account API

In this preceding screenshot, you can see that after login, you can click on **Create New API Token** and it will download a **Kaggle.json**, as shown in the following screenshot:

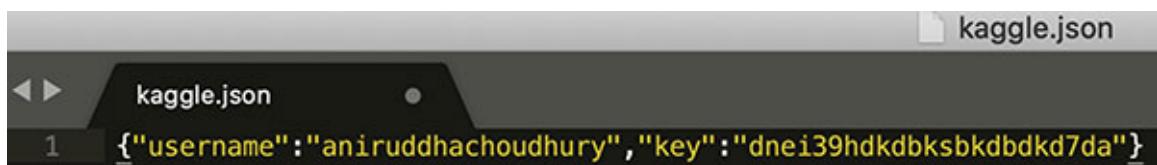


Figure 6.4: Kaggle Json API key

Keep this json file, which we will be using later.

6.2.3 Weights & Biases API Key

Weights & Biases will help you keep track of your machine learning projects. You can use this tool to log the hyperparameters and output metrics from your runs, then visualize and compare the results, and quickly share the findings with your colleagues. The following is the website for Weights & Biases:

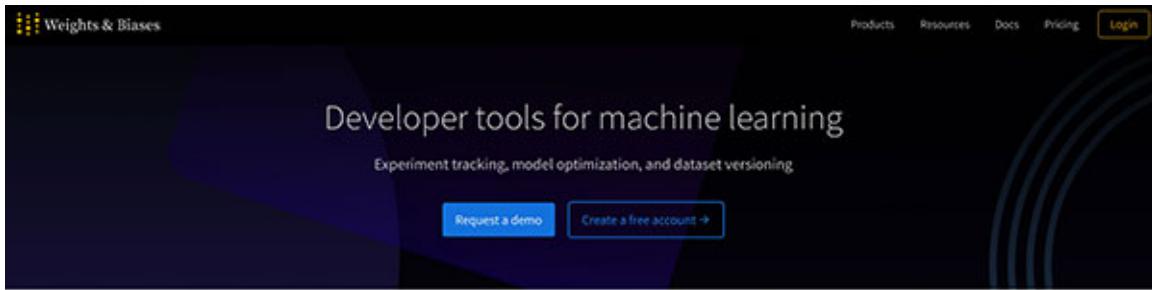


Figure 6.5: Weights & Biases Website

It offers the following Framework and Integration in the On-premise and Cloud Setup:

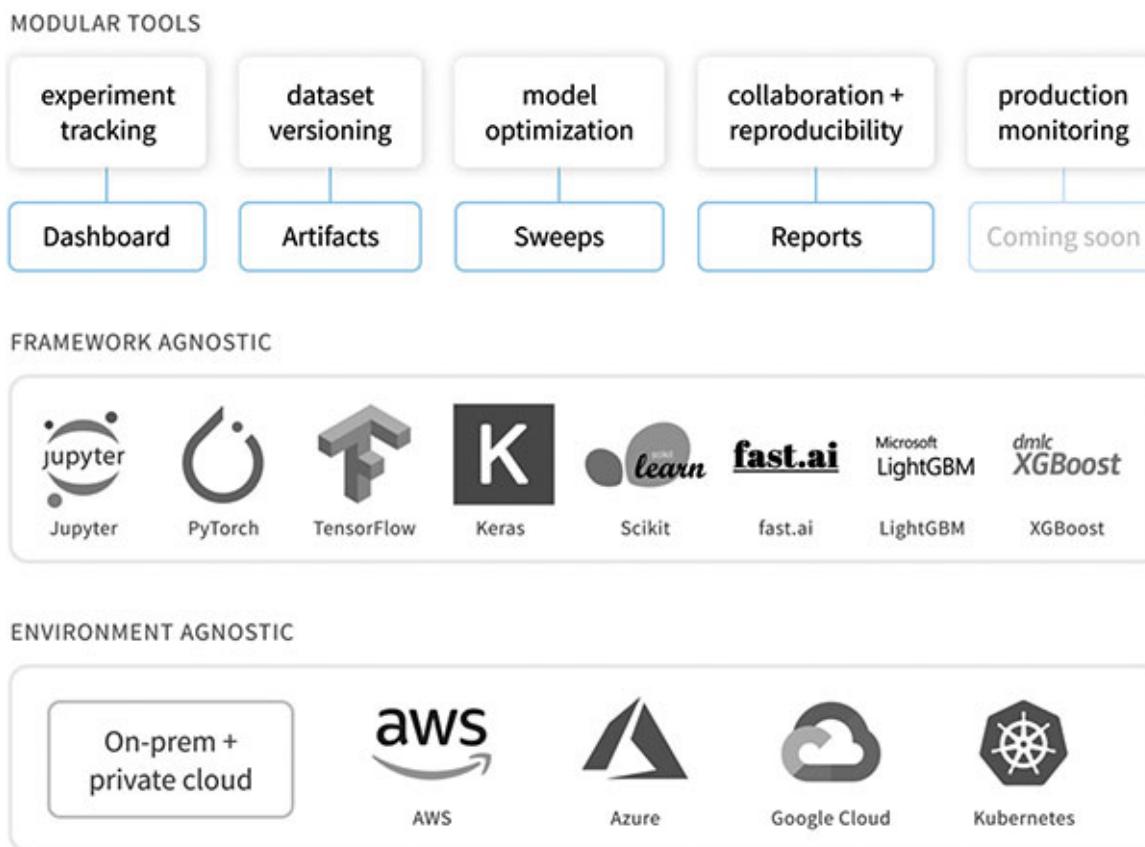


Figure 6.6: Framework and Cloud Support

Next, let's see how to create an API; so a user must have an account or they can create an account by clicking on the following link:
<https://docs.wandb.com/>

Next, go to the user settings and create an API and copy and paste it somewhere in the notes.

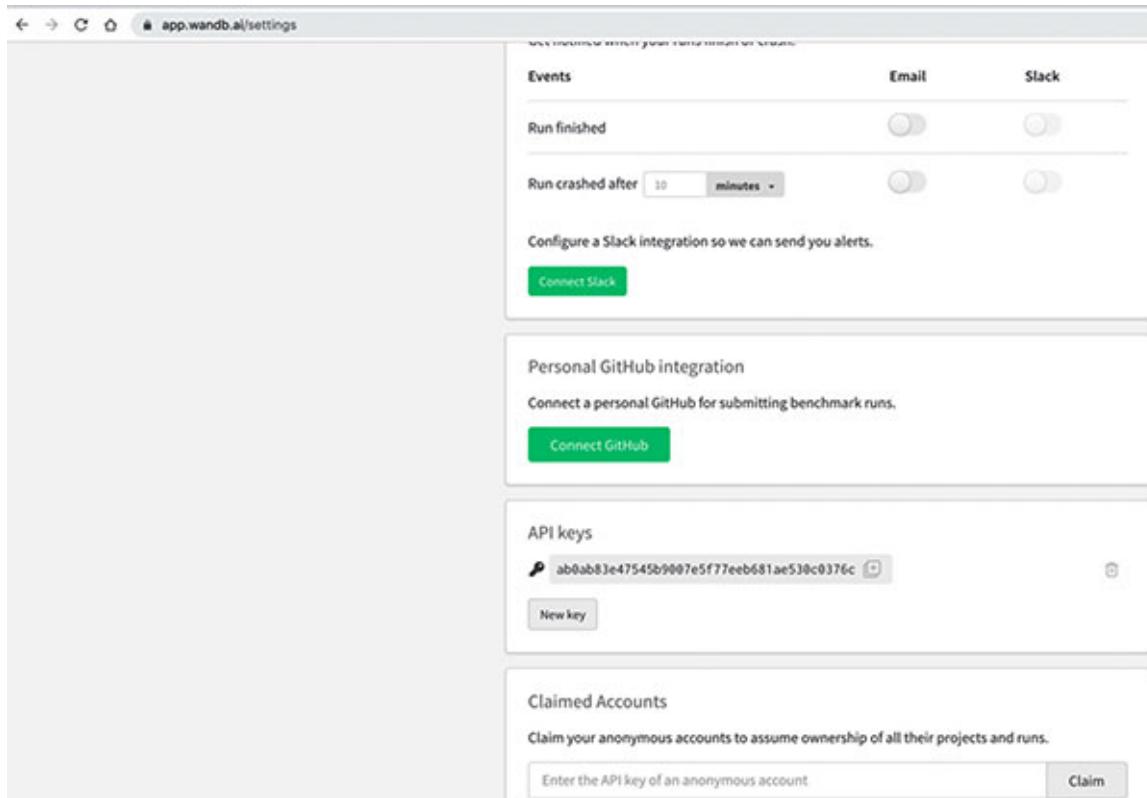


Figure 6.7: Wandb User settings for API

The preceding screenshot shows the API Key, which we will use later for the Integration of the Machine Learning Training Logs and Evaluation graphs.

6.3 Introduction on how to use Weights & Biases

So, What-if offers some features, which are as follows:

1. **Dashboard:** It helps to track the experiments and visualize our training results.
2. **Reports:** It saves and shares reproducible findings.
3. **Sweeps:** It optimizes the models with the hyperparameter tuning.
4. **Artifacts:** Dataset and model versioning, pipeline tracking.

Now, install our library in an environment using Python 3, run the following command `pip install wandb`, and then run the command `wandb login`.

```
wandb: You can find your API key in your browser here: https://app.wandb.ai/authorize
wandb: Paste an API key from your profile and hit enter: 
```

Figure 6.8: Wandb Login from CLI

Modify your training script: We can add a few lines to our script to log the hyperparameters and evaluation metrics.

- **Initialize Wandb:** Next, Initialize Wandb at the beginning of your script, right after the imports.

```
# Inside my model training code
import wandb
wandb.init(project="my-project")
```

The preceding command will automatically create a project for you, if it doesn't exist in the Home. It will capture each run of the preceding training script, and sync to that project, named "**my-project**".

- **Declare Hyperparameters:**

Here, it's easy to save the hyperparameters with the `wandb.config` object.

```
wandb.config.learningrate = 0.2
wandb.config.hidden_layer_size = 64
```

- **Log Metrics:**

Wandb offers to log the metrics for our training loss or accuracy as your model trains (in many cases, we provide the framework-specific defaults). It logs more complicated output or results like histograms, graphs, or images with `wandb.log`.

```
def train_logs():
    for epoch in range(20):
        loss = 0
        wandb.log({'epoch': epoch, 'loss': loss})
```

- **Save Files:**

Anything which was saved in the `wandb.run.dir` directory, will be uploaded to Weight & Biases and saved along with our run when it

completes training, which is convenient for saving the literal Weights & Biases for our trained model. By default, this will save to a new subfolder for the files associated with your run, created in `wandb.run.dir` (which is `./wandb` by default).

```
wandb.save("mymodel.h5")
```

We can pass the full path to the Keras or Tensorflow model API.

```
model.save(os.path.join(wandb.run.dir, "mymodel.h5"))
```

Awesome! Next, run your script normally and it will sync the logs in a backend process. This will be your terminal output, metrics, logs and files, and will be synced to the cloud, alongside the record of our git state if we are running from a git repo.

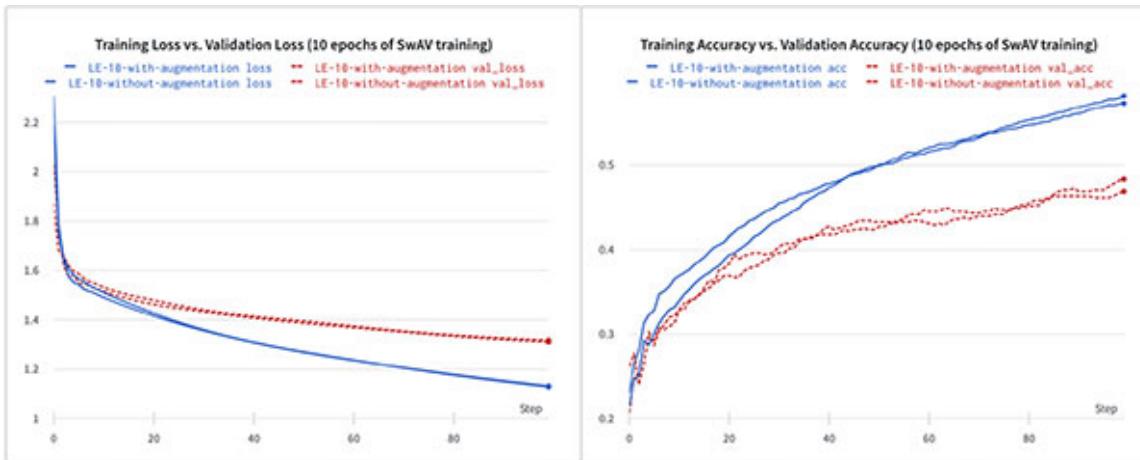


Figure 6.9: Sample Capture of Training Logs

In the preceding screenshot, you can see in one sample Graph of training and Validation accuracy.

6.4 Modeling and training the LightGBM Model for Equity Data

The dataset for equity has 21 features which are time-related. We have 96320 rows of data structure which will be used by your model evaluation. Your goal is to predict the test and live data. Here, the features in the dataset are regularized; there are no categorical features.

The features and the target variable talks about the encrypted stock market data and the target variable is converted into a positive trend of hedge fund versus negative. No domain finance knowledge is required.

6.4.1 Get the latest version of Weights & Biases Dependency & Kaggle Setup

Here, we will run the entire platform in Colab.



Figure 6.10: Colab Logo

Now, run the following command in Colab:

```
from tqdm import tqdm
for i in tqdm(range(2)):
    !pip install -q -r requirements.txt --upgrade
```

```
⌚ 100% |██████████| 2/2 [00:05<00:00, 2.94s/it]
```

Figure 6.11: Library Installation Output

The following is to run the **requirements.txt** which contains the library:



Figure 6.12: Python Library Name

Next, import the following library for the Notebook:

```
import wandb
import logging
```

```
import kaggle
import os
import numpy as np
import random as rn
import pandas as pd
import seaborn as sns
import lightgbm as lgb
import matplotlib.pyplot as plt
from scipy.stats import spearmanr
from sklearn.metrics import mean_absolute_error,
mean_squared_error
from wandb.lightgbm import wandb_callback
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings("ignore");
import plotly.offline as py
py.init_notebook_mode(connected=True)
import plotly.graph_objs as go
import plotly.tools as tls
import plotly.figure_factory as ff
# Set seed for reproducability
seed = 1234
rn.seed(seed)
np.random.seed(seed)
os.environ['PYTHONHASHSEED'] = str(seed)
# Suppress Pandas warnings
pd.set_option('chained_assignment', None)
```

Now, we have imported the required library; next, let's see how to setup the Wandb API, which we created earlier, so that we can save our runs in the Wandb project.

6.4.2 Weights & Biases Dependency & Kaggle API Setup

Copy the key and paste it in the Notebook and store the object in **WANDB_KEY** as a string and run the Wandb login with API.

Weights and Biases:

```
# Obfuscated WANDB API Key  
WANDB_KEY = "ab0ab83e47545b9007e5f77eeb681ae530c0376c"  
!wandb login ab0ab83e47545b9007e5f77eeb681ae530c0376c  
  
↳ wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc  
Successfully logged in to Weights & Biases!
```

Figure 6.13: Wandb Configuration Message

The following is the successful message for the Wandb, and it configured our connection.

Kaggle API Setup:

Now, upload the **Kaggle.json** file in the Colab, and it will be saved, by default, in the contents folder in Colab. The following is the screenshot for the folder in Colab.

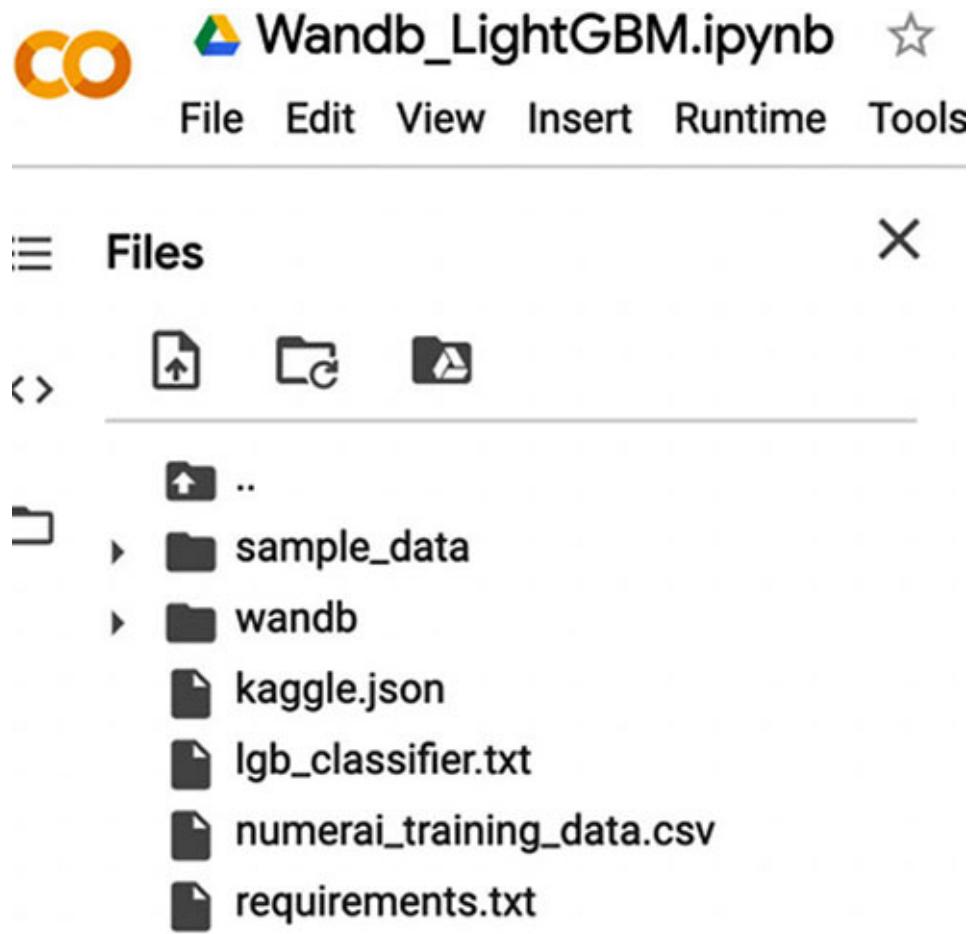


Figure 6.14: Folder structure in Colab

Move this file into `~/.kaggle/`, which is inside the root folder in the environment. This is required for authentication and `chmod` will give the required permission to have that key to be used everywhere in the Notebook.

```
!mkdir ~/.kaggle
!touch ~/.kaggle/kaggle.json
import json
api=json.load(open("/content/kaggle.json"))
with open('/root/.kaggle/kaggle.json', 'w') as file:
    json.dump(api, file)
!chmod 600 ~/.kaggle/kaggle.json
```

In the preceding command, we created an empty json file `Kaggle.json` inside `root./kaggle` and copied the content from `/content/kaggle.json` and gave

the required access with **chmod** and changed the access permissions of the file system objects.

Next, we will download the data from Kaggle. The following is the screenshot of the data which we will use:

The screenshot shows a web browser window for the Kaggle dataset URL kaggle.com/numerai/encrypted-stock-market-data-from-numerai. The page displays the following information:

- Dataset:** Encrypted Stock Market Data from Numerai
- Description:** ~100,000 rows of cleaned, regularized and encrypted equities data.
- Owner:** Numerai • updated 4 years ago (Version 1)
- Category:** Data
- Download:** Download (35 MB)
- Actions:** New Notebook, Copy API command, New Starter Notebook, Social share..., Report issue...
- Usability:** 71
- License:** CC0: Public Domain
- Tags:** business, computer science
- Description:** This is a sample of the training data used in the Numerai machine learning competition. <https://numerai.ai/about>
- Context:** The data is cleaned, regularized and encrypted global equity data. The first 21 columns (feature1 - feature21) are features, and target is the binary class you're trying to predict.
- Content:** We want to see what the Kaggle community will produce with this dataset using Kernels.
- Goal:**

Figure 6.15: Kaggle Dataset URL

Click on the right side, copy the API command and paste it in a cell copy **numerai/encrypted-stock-market-data-from-numerai**

```
kaggle datasets download -d numerai/encrypted-stock-market-data-from-numerai
```

Figure 6.16: API of Data

The following is the format for Kaggle API; run the following commands which unzip that file as the **numerai_training.csv** file:

```
Signature: filename: [owner]/[dataset-name]
dataset_download_file(dataset, file_name, path=None, force=False,
quiet=True)

logging.info(kaggle.api.authenticate())
kaggle.api.dataset_download_files('numerai/encrypted-stock-market-
data-from-numerai', path='/content', unzip=True)
logging.info("Downloaded Data")
```

6.4.3 Loading and Extracting of Data

Now, we will load the data with the Pandas library.

```
data=pd.read_csv("/content/numerai_training_data.csv")
import numpy as np
data['target']=np.random.uniform(0,1,size=data.shape[0])
data.head(5)
```

| feature7 | feature8 | feature9 | feature10 | feature11 | feature12 | feature13 | feature14 | feature15 | feature16 | feature17 | feature18 | feature19 | feature20 | feature21 | target |
|----------|----------|----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|----------|
| 305975 | 0.222414 | 0.893704 | 0.475781 | 0.071068 | 0.640566 | 0.024044 | 0.703299 | 0.128572 | 0.083492 | 0.639390 | 0.299443 | 0.055273 | 0.000308 | 0.784610 | 0.191519 |
| 108447 | 0.326108 | 0.790536 | 0.741738 | 0.741064 | 0.607355 | 0.544590 | 0.449178 | 0.547613 | 0.863165 | 0.832569 | 0.682060 | 0.009634 | 0.829331 | 0.969233 | 0.622109 |
| 192084 | 0.439066 | 0.442903 | 0.404538 | 0.368161 | 0.472579 | 0.796002 | 0.361473 | 0.395270 | 0.390462 | 0.086636 | 0.578854 | 0.603345 | 0.265151 | 0.052699 | 0.437728 |
| 143542 | 0.728365 | 0.573129 | 0.603860 | 0.543855 | 0.679266 | 0.190868 | 0.5533522 | 0.403684 | 0.827155 | 0.102296 | 0.323720 | 0.656344 | 0.704652 | 0.152971 | 0.785359 |
| 147788 | 0.975318 | 0.338439 | 0.406945 | 0.552693 | 0.436036 | 0.440494 | 0.141096 | 0.613245 | 0.508625 | 0.315753 | 0.397510 | 0.111777 | 0.473443 | 0.259582 | 0.779976 |

Figure 6.17: Features Table

Next, we will prepare the data for training; let's split the data with 60/20/20 percentage for training, validation, and test respectively.

```
train, test = train_test_split(data, test_size=0.2)
train, val = train_test_split(train, test_size=0.2)
print(len(train), 'train examples')
print(len(val), 'validation examples')
print(len(test), 'test examples')
```

```
61644 train examples
15412 validation examples
19264 test examples
```

Figure 6.18: Train-Test split size

Now that we have spit our data, let's do some EDA.

6.4.4 Exploratory Data Analysis

Let's see the correlation matrix among the encrypted data with respect to the target columns. The term positive correlation depicts that both the features will move in the same direction, and negative correlation depicts that when one of the variable's value increases, the other variable's values decrease. So, correlation with neural/zero depicts that the variables are not related.

```
def correlation_plot(data):
```

```

correlation = data.corr()
matrix_cols = correlation.columns.tolist()
corr_array = np.array(correlation)
trace = go.Heatmap(z = corr_array,x = matrix_cols,y =
matrix_cols,xgap = 2,ygap = 2, colorscale='Plasma',colorbar =
dict())
layout = go.Layout(dict(title = 'Correlation Matrix for
variables', autosize = False, height = 900,width = 1000,margin
= dict(r = 0, l = 210, t = 25,b = 210), yaxis = dict(tickfont =
dict(size = 9)),xaxis = dict(tickfont = dict(size = 9))))
fig = go.Figure(data = [trace],layout = layout)
py.iplot(fig)

```

Now, let's check the plot on the following page.

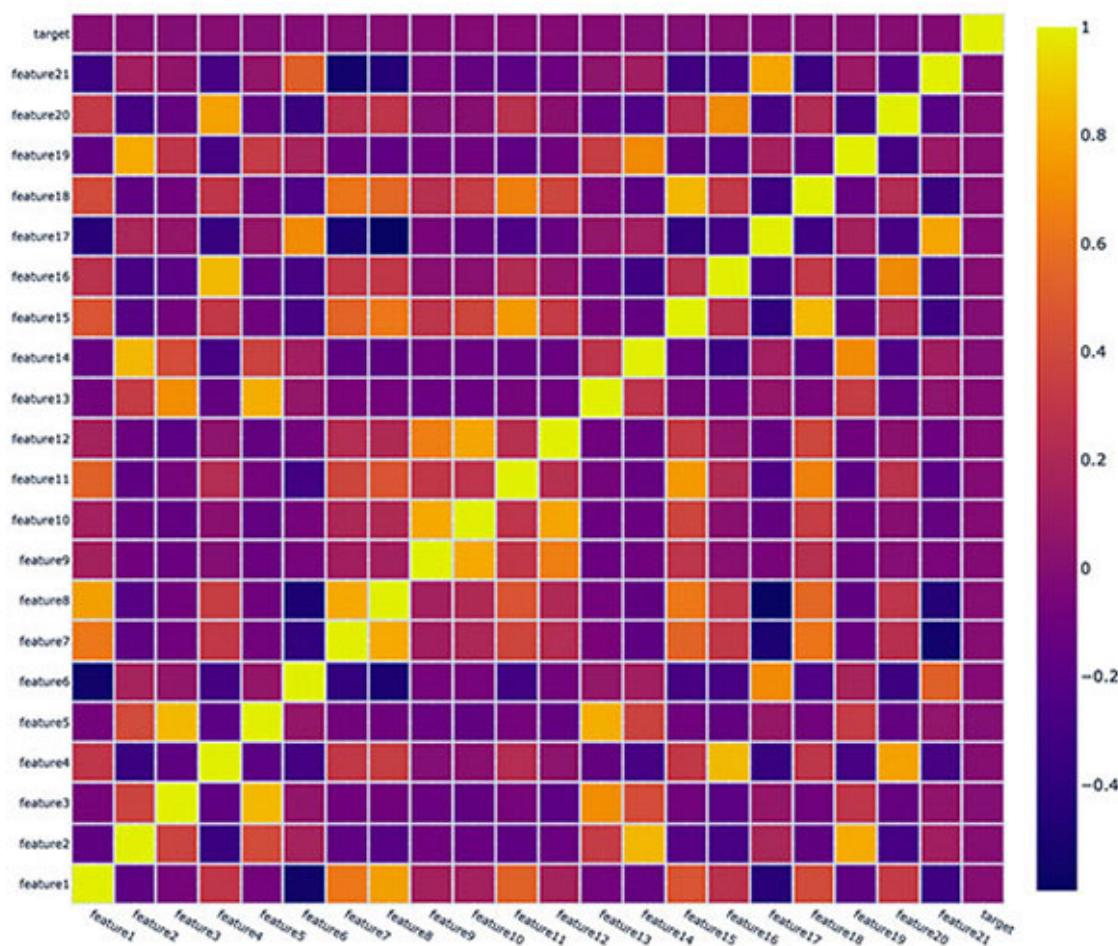


Figure 6.19: Correlation Plot

So, here the features are having a remarkably low correlation to the target variable. Even the most correlated features only have around 1.5% correlation with the target variable.

Next, we have noticed that the importance of features may change over time. So, by selecting a minimum number of features, we risk having a high "*feature exposure*." So, the feature exposure can be quantified as a standard deviation of all your predictions' correlations with respect to each and every feature. We can mitigate the risk by using the dimensionality reduction techniques, such as **Principal Component Analysis (PCA)** to integrate almost all the features into your model.

Next, let's see the Distribution of the Train, Test, and Validation Data.

```
feats = [f for f in train.columns if "feature" in f]
plt.figure(figsize=(16, 5))
sns.distplot(pd.DataFrame(train[feats].std()), bins=50)
sns.distplot(pd.DataFrame(val[feats].std()), bins=50)
sns.distplot(pd.DataFrame(test[feats].std()), bins=50)
plt.legend(["Train", "Val", "Test"], fontsize=20)
plt.title("Standard deviations over all features in the data",
          fontsize=20);
```



Figure 6.20: Distribution Plot

Now, we can see that all the train, validation, and test dataset's standard deviation among the features are widely spread in the breadth in its normal distribution, between ranges 0.275 and 0.295.

```
feature_list = list(train.columns)
feature_list.remove("target")
```

Now, graph the feature column names inside a list `feature_list` for further operations.

6.4.4 Utility Metrics Function

We will evaluate our training LightGBM Regressor with the following three metrics:

- **Spearman Correlation:** Statistically, this method will quantify the degree to rank the variables which are associated by a monotonic function, that represents as an increasing or decreasing relationship behaviour. The method does a hypothesis test, and tells that the samples are uncorrelated (fail to reject H0). The function next takes the two real-valued data as args and returns both the correlation coefficient in the range between -1 and 1, and also the p-value which will interpret the coefficient significance.

For a sample of size n , the n raw scores X_i, Y_i are converted to ranks $\text{rg}_{X_i}, \text{rg}_{Y_i}$, and r_s is computed as

$$r_s = \rho_{\text{rg}_X, \text{rg}_Y} = \frac{\text{cov}(\text{rg}_X, \text{rg}_Y)}{\sigma_{\text{rg}_X} \sigma_{\text{rg}_Y}},$$

where

ρ denotes the usual Pearson correlation coefficient, but applied to the rank variables,

$\text{cov}(\text{rg}_X, \text{rg}_Y)$ is the covariance of the rank variables,

σ_{rg_X} and σ_{rg_Y} are the standard deviations of the rank variables.

Only if all n ranks are *distinct integers*, it can be computed using the popular formula

$$r_s = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)},$$

where

$d_i = \text{rg}(X_i) - \text{rg}(Y_i)$ is the difference between the two ranks of each observation,

n is the number of observations.

Figure 6.21: Spearman Correlation (Source Credit: Wiki)

- **Mean Absolute Error:** The average of the absolute differences between the actual and predictions values. It tells how wrong the predictions were. It also measures the magnitude error, and gives an idea direction, so a value of 0 indicates no error or perfect predictions.
- **Root Mean Square Error:** The R² (or R Squared) metric talks about an indication of the goodness of fit for our predictions with respect to the

original values. In statistical terminology, it is also known as the coefficient of determination. Its value lies in between 0 and 1 for no-fit and perfect fit respectively.

```
def evaluate(df: pd.DataFrame) -> tuple:
    def _score(sub_df: pd.DataFrame) -> np.float32:
        """Calculates Spearman correlation"""
        return spearmanr(sub_df["target"], sub_df["prediction"])[0]

    mae = mean_absolute_error(df["target"],
                               df["prediction"]).round(4)
    RMSE=mean_squared_error(df["target"], df["prediction"])**0.5
    spearman=spearmanr(df["target"], df["prediction"])[0]
    # Display metrics
    print(f"Spearman Correlation: {spearman}")
    print(f"RMSE Score: {RMSE}")
    print(f"Mean Absolute Error (MAE): {mae}")
    return spearman,RMSE, mae
```

So, we will use the preceding function in our Model of LightGBM.

[6.4.5 Training model \(using Weights & Biases\) with LightGBM Framework](#)

In this section, let's see how we will build our LightGBM framework with Weights & Biases.

How does LightGBM work?

LightGBM uses the histogram-based algorithm, which buckets the continuous feature (attribute) values into discrete bins. This speeds up the training and reduces the memory usage.

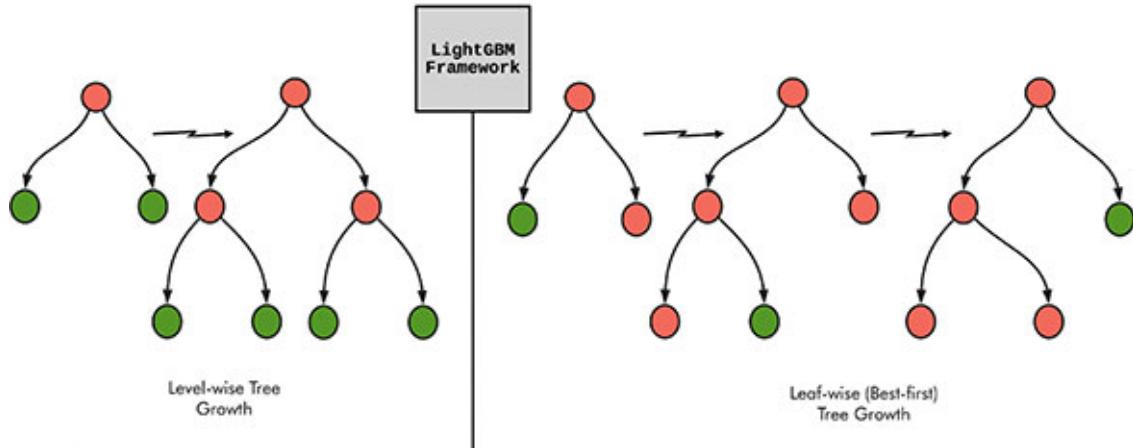


Figure 6.22: LightGBM Architecture

LightGBM grows its trees in a leaf-wise manner for the best-first. Next, it takes the leaf with the maximum delta loss for growing. The leaf-wise algorithms tend to achieve lower loss than the level-wise algorithms by holding the leaf fixed.

The leaf-wise algorithm may cause over-fitting with a small set of data, so LightGBM includes a parameter to limit the tree depth which is `max_depth`. However, the leaf-wise trees still grow even when the `max_depth` is specified.

The advantages of the histogram-based algorithms include the following:

- Reduced cost of calculating the gain for each split.
- Uses histogram subtraction for further speedup.
- Reduces memory usage.
- Reduces communication cost for parallel learning.

For further reading, please go to the following link:

<https://lightgbm.readthedocs.io/en/latest/Features.html>

Sweeps:

We will train the LightGBM model to get a first good model, and then we will use Weights & Biases to do the hyperparameter sweep. Here, in this example, we will do a grid search for the LightGBM Framework algorithm over some of the most important hyperparameters. First, we will define the configuration of the sweep.

The common use cases are as follows:

- 1. Explore:** It efficiently discovers the promising region's sample space of hyperparameter combinations; next it builds an intuition about your model.
- 2. Optimize:** It helps to find the set of hyperparameter's optimal performance for our model.
- 3. K-fold cross validation:** Here's a brief example below of k-fold cross validation with W&B Sweeps.

Approach:

- 1. Add Wandb:** Write a couple of lines of code to log the hyperparameters and evaluation output metrics in our Python script.
- 2. Write config:** Next, for the sweep over, we will define the variables and ranges as well. We will choose a search strategy — it supports random, grid, and Bayesian search, alongside early stopping.
- 3. Initialize sweep:** Now, start the sweep server. It hosts the central controller and will coordinate between the multiple agents which will execute the sweep.

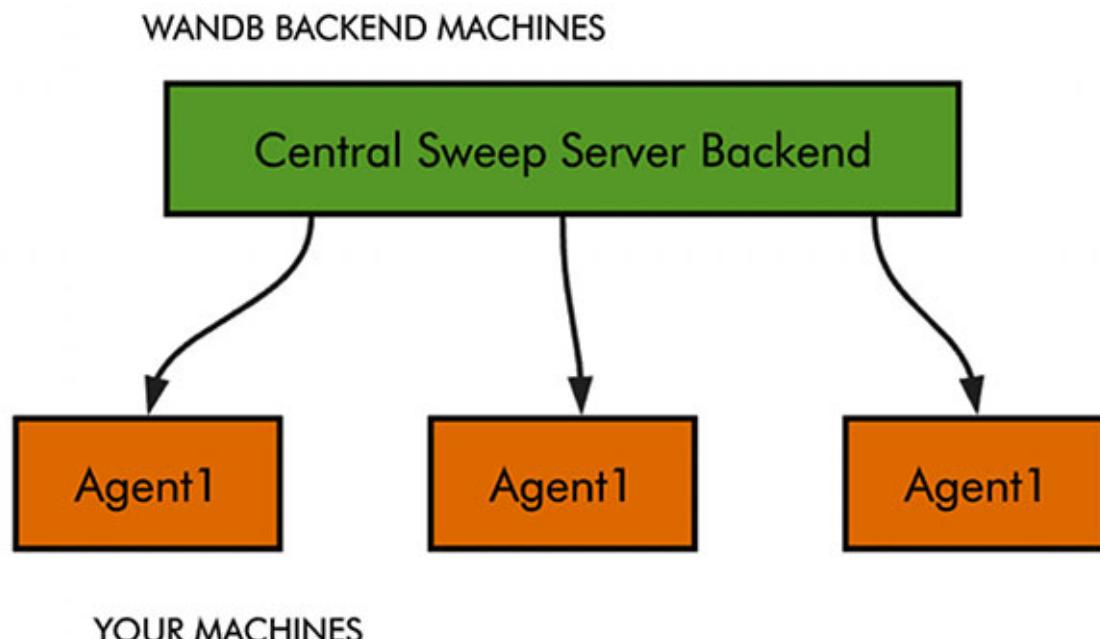


Figure 6.23: Wandb Sweep Architecture

- 4. Launch agent:** To use the train models in the sweep, run the command on each machine, then the agents ask the backend central sweep server

about what hyperparameters to try next, and it will execute the runs.

5. **Visualize results:** Click on the link, which will be generated and open our live dashboard to see all our results in one central place. Check the following link for configuration:
- <https://docs.wandb.com/sweeps/configuration>

Run the following configuration in our notebook:

```
# Configuration for hyperparameter sweep
sweep_config = {
    'method': 'grid',
    'metric': {
        'name': 'mse',
        'goal': 'minimize'},
    'parameters': {
        "num_leaves": {'values': [30, 40, 50]},
        "max_depth": {'values': [4, 5, 6, 7]},
        "learning_rate": {'values': [0.1, 0.05, 0.01]},
        "bagging_freq": {'values': [7]},
        "bagging_fraction": {'values': [0.6, 0.7, 0.8]},
        "feature_fraction": {'values': [0.85, 0.75, 0.65]},})
sweep_id = wandb.sweep(sweep_config, project="simpletransformers")
```

➡ Create sweep with ID: d3b061m9
Sweep URL: <https://app.wandb.ai/aniruddha/simpletransformers/sweeps/d3b061m9>

Figure 6.24: Sweep Configuration Link

Let's break the preceding configuration for the sweep metric method and also we will talk about LightGBM parameters.

Metric:

Here, specify the metric to optimize. This metric will log explicitly to Wandb while in our training script. In this example, we want to minimize the validation loss for our LightGBM model:

name: Name of the metric to optimize goal: minimize or maximize (Default is minimize)

Method:

Next, we will specify the search strategy from the grid, random search, and bayes:

- **Grid**: It iterates over all the possible combinations to find the parameter values.
- **Random**: It chooses the random sets of values.
- **Bayes**: It uses a gaussian process to model our function and then chooses the parameters which will optimize the probability of improvement into our model. It is also called *Bayesian Optimization*.

Parameters:

Next, check out the following links for the parameter's description:

<https://lightgbm.readthedocs.io/en/latest/Parameters.html>

The last code line from the preceding `sweep_id` will store the project name and configuration setup which we will be providing during LightGBM Model Training.

- Prepare Data for LightGBM:

Now, we will be creating the data preparation for the LightGBM model.

```
dtrain = lgb.Dataset(train[feature_list],  
label=train["target"])  
dvalid = lgb.Dataset(val[feature_list], label=val["target"])  
watchlist = [dtrain, dvalid]
```

- Train the Model Integration Wandb:

Next, let's see how to build the LightGBM regressor and build a `train()` method as a utility function:

```
def _train():  
    # Configure and train model  
    wandb.init(name="LightGBM_sweep")  
    lgbm_config = {"num_leaves": wandb.config.num_leaves,  
                  "max_depth": wandb.config.max_depth, "learning_rate":  
                  wandb.config.learning_rate, "bagging_freq":  
                  wandb.config.bagging_freq, "bagging_fraction":  
                  wandb.config.bagging_fraction, "feature_fraction":
```

```

wandb.config.feature_fraction, "metric": 'mse',
"random_state": seed}
lgbm_model = lgb.train(lgbm_config, train_set=dtrain,
num_boost_round=750, valid_sets=watchlist, callbacks=
[wandb_callback()], verbose_eval=100,
early_stopping_rounds=50)

# Create predictions for evaluation
val_preds = lgbm_model.predict(val[feature_list],
num_iteration=lgbm_model.best_iteration)
print(val_preds)
print(type(val_preds))
val.loc[:, "prediction"] = val_preds
# W&B log metrics
spearman, RMSE, mae = evaluate(val)
wandb.log({"Spearman": spearman, "RMSE": RMSE, "Mean
Absolute Error": mae})
#lgb_path = '/content/'
save_to = "/content/lgb_classifier.txt"
lgbm_model.save_model(save_to)

```

Now, let's break down the preceding `train()` method:

- We will be initiating the training of our model in Wandb with the name **LightGBM _sweep** which will save all the runs in that name.

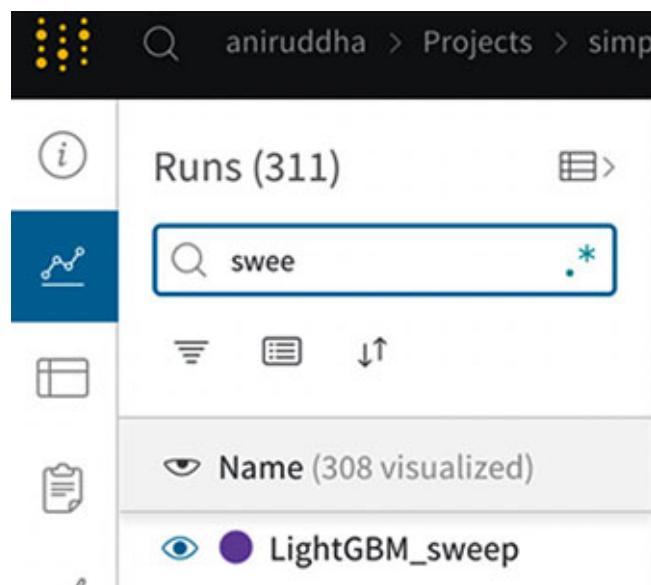




Figure 6.25: Run Name in Wandb

- So, here we will set `wandb.config` in our script to save our training config, hyperparameters, and input the settings like the dataset name or model type, and any other independent variables or parameters, such as the learning rate, max depth, and so on, for the LightGBM model experiments. It is very useful for analyzing our experiments and reproducing our work for the future. We will be able to group by the configuration values in our web interface and compare the different runs and see how these affect the output. Let's set the object in `lgbm_config`.

Format: `wandb.config.[PARAMETER]`

- Next, we will train our LightGBM model with `early_stopping_rounds` which will stop the training if one metric of one validation data doesn't improve in the end. Then both, the train and the validation data which we created, will pass `lgbm_config`, which we created earlier.

- Then, we will evaluate our model with the test dataset and evaluate the prediction with the original dataset to calculate MAE, RMSE, and Correlation.
- The output metrics or the dependent variables RMSE, MAE, and Spearman should be saved with `wandb.log` instead. We have saved the LightGBM model with the best iteration in the `/content/lgb_classifier.txt` folder which we will be using for the KF Serving production in the Kubeflow Kubernetes cluster in Goggle Cloud Platform.

Now, run the following command with the `sweep_id` configuration which we have done earlier, and pass the preceding `train()` function.

```
# Run hyperparameter sweep (grid search)
wandb.agent(sweep_id, function=_train)

INFO:wandb.wandb_agent:Running runs: ['u5n9ep55']
INFO:wandb.wandb_agent:Cleaning up finished run: u5n9ep55
INFO:wandb.wandb_agent:Agent received command: run
INFO:wandb.wandb_agent:Agent starting run with config:
    bagging_fraction: 0.6
    bagging_freq: 7
    feature_fraction: 0.85
    learning_rate: 0.1
    max_depth: 4
    num_leaves: 40
wandb: Agent Starting Run: b9ml3fz3 with config:
    bagging_fraction: 0.6
    bagging_freq: 7
    feature_fraction: 0.85
    learning_rate: 0.1
    max_depth: 4
    num_leaves: 40
wandb: Agent Started Run: b9ml3fz3
Logging results to Weights & Biases \(Documentation\).
Project page: https://app.wandb.ai/aniruddha/simpletransformers
Sweep page: https://app.wandb.ai/aniruddha/simpletransformers/sweeps/03edx2p8
Run page: https://app.wandb.ai/aniruddha/simpletransformers/runs/b9ml3fz3
Training until validation scores don't improve for 50 rounds.
Early stopping, best iteration is:
[3]      training's 12: 0.0835953          valid_1's 12: 0.0827124
[0.5026042 0.5016406 0.5016406 ... 0.5016406 0.5026042 0.5016406]
<class 'numpy.ndarray'>
Spearman Correlation: 0.004787618906806593
RMSE Score: 0.2875976805060863
Mean Absolute Error (MAE): 0.2481
wandb: Agent Finished Run: b9ml3fz3
```

Figure 6.26: Output of Each Run

The preceding example is one run output logs in Notebook; it talks about the model performance and each run is saving in the Blue Hyperlink URL where we can navigate and see the performance and the Model Hyperparameters' Parallel Coordinates. For example, check the following link:

<http://app.wandb.ai/aniruddha/simpletransformers/sweeps/03edx2p8>

- Wandb Dashboard

In the preceding training logs, we got 3 URLs. Let's navigate to the PROJECT URL and check one sample Run performance.



Figure 6.27: Dashboard of one Run

Now, in the preceding screenshot, we can see that there are a total of 311 Runs, as it was mentioned on the top left, next, the Dashboard consists of RMSE, Spearman, Training Loss, and Validation Loss, and the Mean Absolute Error for each Run in a single axis, and it tells which run is the best by comparing each other; let's take one example Spearman and train/validation loss.

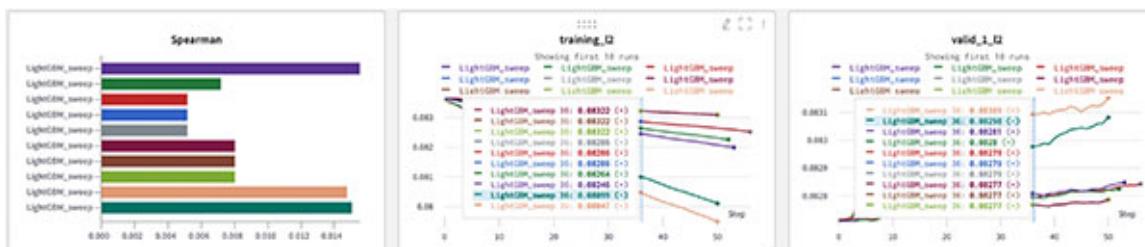


Figure 6.28: Multiple Run Comparison

Now, in the preceding screenshot, in the first Bar plot, we can see the 3 top most correlation run purple, green, and light pink bars. Similarly, we can analyse the multiple scores for each run in train and test, which it trained in the number steps or epochs, and we can take the best run for the Model Deployment. Next, click on **RUN URL**; on the individual run, you will get the following report charts:

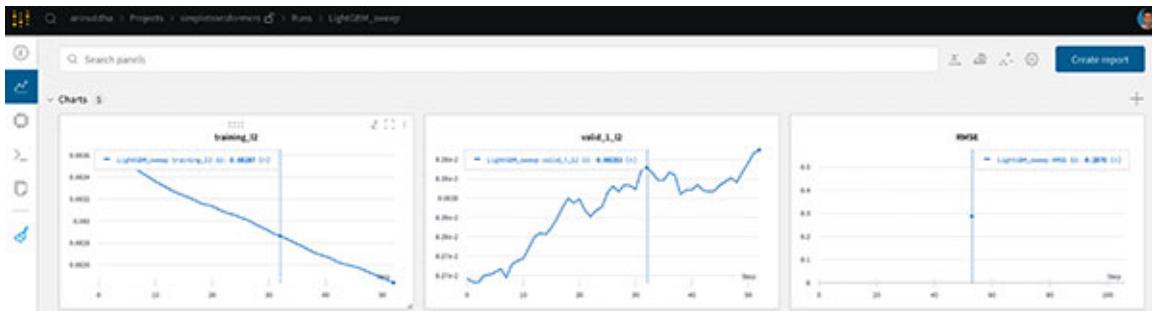


Figure 6.29: Multiple Run Comparison

So, the preceding screenshot talks about the individual run, as we can see the validation loss is low with respect to the train loss.

Next, we can see the GPU or CPU consumption with respect to the various parameters:

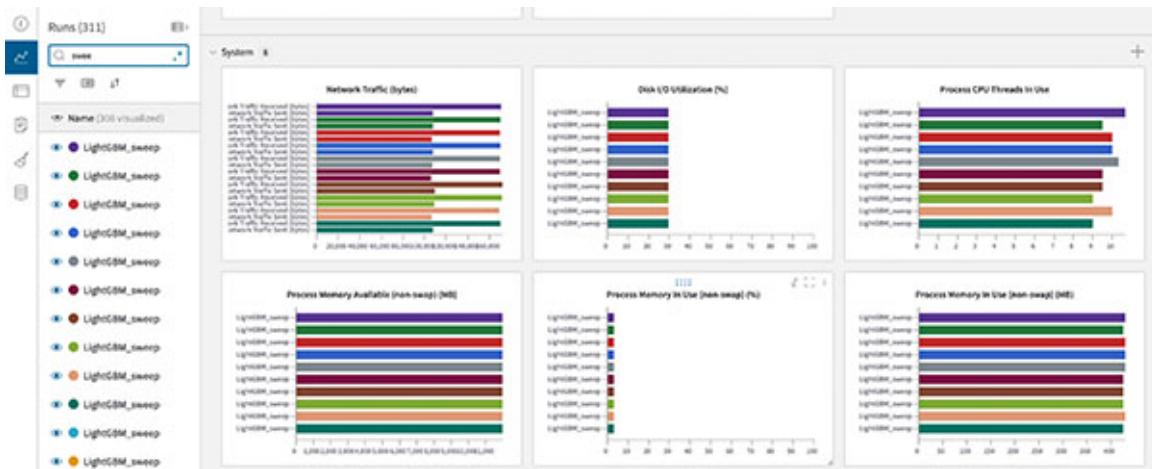


Figure 6.30: Model Machine performance

The preceding screenshot talks about the Memory Utilization for each run while training the model and how much network traffic in bytes are consumed for pushing the logs, and what's the CPU consumption for the various runs.

Let's navigate to the SWEEP URL to visualize the sweep results:

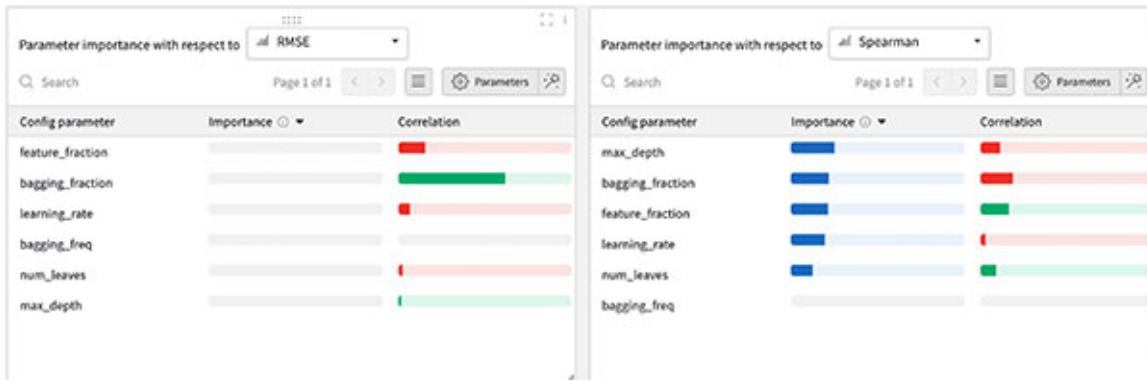


Figure 6.31: Hyperparameter Importance Plot

So, the hyperparameter importance plot surfaces tell which hyperparameters are the best predictors and highly correlated to that desirable values for your dataframe.

Here, the correlation represents a linear correlation between the hyperparameter and the chosen metric, in this case, the Validation Loss. So, here the high correlation tells that the hyperparameter has a higher value, and the metric also has higher values and vice versa. Next, the correlation is also an awesome metric to look at, but it can't capture the second order interactions between the inputs, and it can get messy to compare the inputs with the different range metrics.

Let's go down a bit to check the parallel coordinate for hyperparameters to find out the metric's performance. It plots the map hyperparameter values to model the metrics which is useful for honing in on combinations of hyperparameters that led to the best model performance.

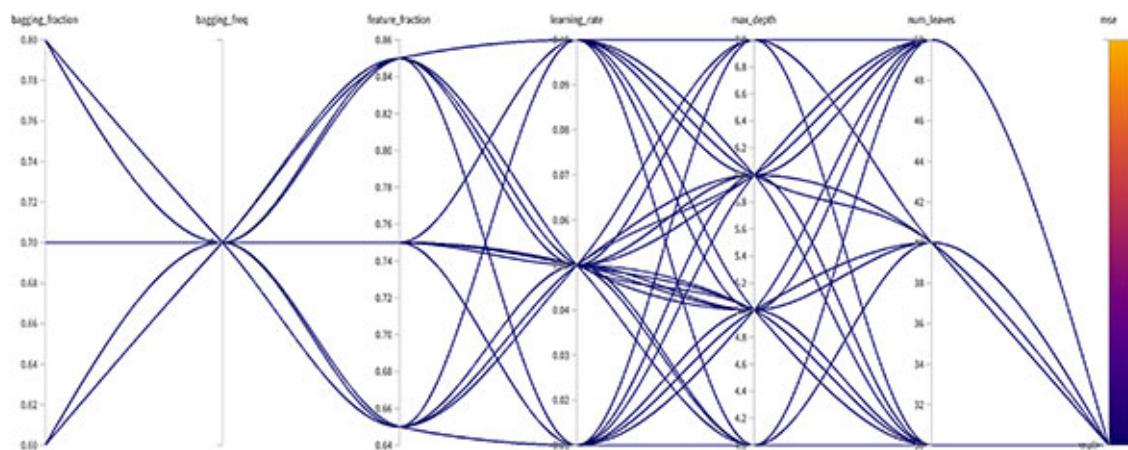


Figure 6.32: Hyperparameter Parallel Coordinate

In the preceding screenshot, each axis represents a different parameter range values, in this case, we have 7 vertical axes such as `bagging_fraction`, `learning_rate`, `max_depth`, `_num_leaves`, and so on. Next, visualize the relationship between the different hyperparameters and the final mean square error of my model, which we will be minimizing.

- **Axes:** It represents the different hyperparameters from `wandb.config`, which I have mentioned earlier and all the metrics from `wandb.log()`.
- **Lines:** Here, a single line represents a single run. Hover the mouse over a line to see a tooltip with all the details about each run.

6.5 Serving the model with KF Serving

In this section, we will build our serving model. In the following screenshot, we can see the three major components for our model serving below screenshot.

So, here we kept the saved trained model from the local to the root of the Docker file, or we can call it from the Cloud Storage bucket.

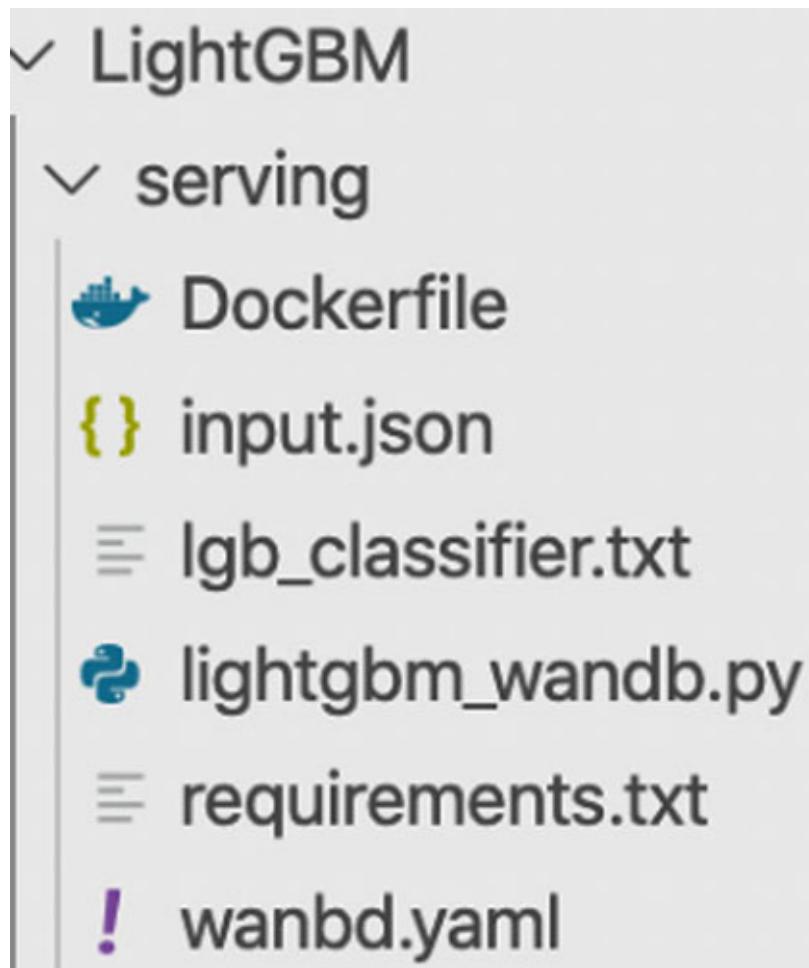


Figure 6.33: Folder Structure

In previous chapters, we have covered the KF Serving Architecture; so you can refer

Let's see how we will build the serving component `lightgbm_wandb.py` file:

```
import os
import sys
import json
import numpy as np
import kfserving
import lightgbm as lgb
from typing import List, Dict
class KFServingSampleModel(kfserving.KFModel):
    def __init__(self, name: str):
        super().__init__(name)
```

```

self.name = name
self.ready = False
self.model_output_base_path = "lgb_classifier.txt"
def load(self):
    model = lgb.Booster(model_file=self.model_output_base_path)
    self.model = model
    self.ready = True
def predict(self, request: Dict) -> Dict:
    inputs = np.array(request["instances"])
    reshaped_to_2d = np.reshape(inputs, (-1, len(inputs)))
    results = self.model.predict(reshaped_to_2d)
    result = (results > 0.5)*1
    if result==1:
        result="Positive Equity"
    else:
        result="Negative Equity"
    print("result : {0}".format(result))
    return {"predictions": result}

if __name__ == "__main__":
    model = KFServingSampleModel("kfserving-wandb-lightgbm-model")
    model.load()
    kfserving.KFServer(workers=1).start([model])

```

So, let's break the transformer predictor code as follows:

- We kept the save model file **lgb_classifier.txt** in the Docker root, and here we declared that file as an environment variable to an object **model_output_base_path**.
- Next, in the Load function, we loaded the model in from the LightGBM library.
- Then, in the predict method, the incoming data came as a json format, which we needed to extract as a key-value pair and do the necessary prediction and return as a dictionary.
- So, in the “main” function, the **KFServingSampleModel** Class took the name of that deployment; keep a note of that and apply to the yaml file;

here it is "kfserving-wandb-lightgbm-model".

```

1  import os
2  import sys
3  import json
4  import numpy as np
5  import kfserving
6  import lightgbm as lgb
7  from typing import List, Dict
8
9
10
11 class KFServingSampleModel(kfserving.KFModel):
12     def __init__(self, name: str):
13         super().__init__(name)
14         self.name = name
15         self.ready = False
16         self.model_output_base_path = "lgb_classifier.txt"
17
18     def load(self):
19         model = lgb.Booster(model_file=self.model_output_base_path)
20         self.model = model
21         self.ready = True
22
23     def predict(self, request: Dict) -> Dict:
24         inputs = np.array(request["instances"])
25         reshaped_to_2d = np.reshape(inputs, (-1, len(inputs)))
26         results = self.model.predict(reshaped_to_2d)
27         result = (results > 0.5)*1
28         if result==1:
29             result="Positive Equity"
30         else:
31             result="Negative Equity"
32
33         print("result : (%)" .format(result))
34         return {"predictions": result}
35
36
37 if __name__ == "__main__":
38     model = KFServingSampleModel("kfserving-wandb-lightgbm-model")
39     model.load()
40     kfserving.KFServer(workers=1).start([model])

```

wandb.yaml

```

use-cases > LightGBM > serving > wandb.yaml > () spec > () default >
1  apiVersion: serving.kubeflow.org/v1alpha2
2  kind: InferenceService
3  metadata:
4    labels:
5      controller-tools.k8s.io: "1.0"
6    name: kfserving-wandb-lightgbm-model
7    namespace: kubeflow
8  spec:
9    default:
10      predictors:
11        custom:
12          container:
13            image: gcr.io/<PROJECT_ID>/<IMAGE_NAME>:<TAG>
14            imagePullPolicy: Always
15        name: user-container
16

```

Make Sure the KF Serving inference name in line 38th what we provided in **lightgbm_wandb.py** here it is **kfserving-wandb-lightgbm-model** it should be place to left side **wandb.yaml** file 6th line as beside name

Figure 6.34: kf Serving python code left & Deployment yaml right

Let's say after building the image, the name is **gcr.io/<PROJECT_ID>/<IMAGE_NAME>:<TAG>**

Now, let's understand the KF serving transformer code here. From above screenshot on the left side of the Python file, the **KFServingSampleModel** class will always have one **Load()** method where we can load our trained model from the docker path or from the Google Bucket; then we can load there for the prediction and **predict()** method which will be used to take the incoming json request as an input. Then, we can do our necessary operations prior to the model prediction of either the text or the image data case.

Let's see how we will build the serving component Dockerfile:

```

FROM python:3.7-slim-stretch
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update && \
    apt-get -y install gcc mono-mcs g++ git curl bash && \
    rm -rf /var/lib/apt/lists/*

```

```

ADD requirements.txt /app/requirements.txt
RUN pip install -r /app/requirements.txt
ADD lightgbm_wandb.py /app/lightgbm_wandb.py
ADD lgb_classifier.txt /app/lgb_classifier.txt
# COPY . /app
WORKDIR /app
CMD ["python", "lightgbm_wandb.py"]

```

So from the above dockerfile we have the following steps:

- The first line of our Dockerfile begins with FROM. This is where we import our OS or programming language.
- The next two lines involve setting up the environment and executing it on the server. The ADD line makes the local file, `requirements.txt`, available in the Docker container. The RUN command can be followed with any bash code that you would like being executed.
- We use RUN to install our dependencies. Then ENV sets our environment variable.
- The WORKDIR line sets our working directory to the app. Then, the ADD line makes the remaining local files available in the Docker container. Next, CMD will run the command when the Docker file will execute each time.

Now, let's deploy it with the command line; and first let's fill the yaml file:

```

apiVersion: serving.kubeflow.org/v1alpha2
kind: InferenceService
metadata:
  labels:
    controller-tools.k8s.io: "1.0"
  name: kf-serving-wandb-lightgbm-model
  namespace: kubeflow
spec:
  default:
    predictor:
      custom:
        container:

```

```

image: gcr.io/<PROJECT_ID>/<IMAGE_NAME:<TAG>
imagePullPolicy: Always
name: user-container

```

Here, in the preceding yaml file, we gave the same name which we had provided in the lightgbm_wandb.py file; it will be (“**kfserving-wandb-lightgbm-model**”), and then we provided the namespace “kubeflow” where it is deployed. Next, we gave the Docker image a name for our Transformer model which we had created.

Next, run the following command from bash where the files are kept in the Visual Studio:

1. Connect to the GCP cluster by using the following command:

```
gcloud container clusters get-credentials <$ClusterName> --zone <$ZONE> --project <$PROJECTID>
```

2. Create the inference service by deploying it in the cluster:

```
kubectl apply -f wandb.yaml
```

3. Check the inference service. Try it after some interval to check if it has been created:

```
kubectl get inferenceservice -n kubeflow
```

| NAME | URL | READY | DEFAULT TRAFFIC | CANARY TRAFFIC | AGE |
|--------------------------------|--|-------|-----------------|----------------|-------|
| kfserving-wandb-lightgbm-model | http://kfserving-wandb-lightgbm-model.default.example.com/v1/models/kfserving-wandb-lightgbm-model | True | 100 | | 2h53m |

Figure 6.35: KF serving Inference

Sample Prediction:

- Run the following command in Bash from the serving folder:

```

MODEL_NAME=kfserving-wandb-lightgbm-model
HOST=$(kubectl get inferenceservice -n kubeflow$MODEL_NAME -o
jsonpath='{.status.url}' | cut -d "/" -f 3)
INPUT_PATH=@./input.json
CLUSTER_IP=$(kubectl -n istio-system get service kfserving-
ingressgateway -o
jsonpath='{.status.loadBalancer.ingress[0].ip}')

```

```

curl -v -H "Host: ${HOST}"
http://${CLUSTER_IP}/v1/models/${MODEL_NAME}:predict -d
$INPUT_PATH

* trying [REDACTED]...
* TCP_NODELAY set
* Connected to [REDACTED] ([REDACTED]) port 80 (#0)
> POST /v1/models/kfserving-wandb-lightgbm-model:predict HTTP/1.1
> Host: kfserving-wandb-lightgbm-model.default.example.com
> User-Agent: curl/7.64.1
> Accept: */*
> Cookie: authservice_session=MTU50Tkx0TQ0NXx0d3dBTkZrM1Z6UTB0a3RCV2taYVdTTFSa3BTVHpKU1dGUklUME;
> Content-Length: 490
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 490 out of 490 bytes
< HTTP/1.1 200 OK
< content-length: 34
< content-type: application/json; charset=UTF-8
< date: Sat, 12 Sep 2020 14:09:49 GMT
< server: istio-envoy
< x-envoy-upstream-service-time: 22
<
* Connection #0 to host [REDACTED] left intact
{"predictions": "Negative Equity"}* Closing connection 0

```

Figure 6.36: KF serving Prediction Output

Run the following below command in Python from the Colab notebook:

Now, we will create some sample data to predict the results from the preceding below URL. To create the sample data, the code is as follows:

```

import requests
MODEL_NAME="kfserving-wandb-lightgbm-model"
cluster_ip = <COPY YOUR CLUSTER IP HERE>
headers={"Host": "{0}.kubeflow.example.com".format(MODEL_NAME)}
response =
requests.post("http://{0}/v1/models/{1}:predict".format(cluster_ip
, MODEL_NAME), data = data1,headers = headers)
response.json()

↳ {"predictions": 'Negative Equity'}

```

Figure 6.37: KF serving Prediction O/P Python

6.6 Monitoring the performance with Grafana Dashboard

Grafana includes the built-in support for Prometheus. This topic explains the options, variables, querying, and the other options specific to the Prometheus

data source. Launch the Grafana dashboard. As shown in [Chapter 4, Building TFX Pipeline](#) in section 4.7, we have already installed Grafana.

Next, run from the Local Terminal and open the `grafana` dashboard by using `localhost:8080` on the browser. Explore the different components of the Grafana dashboard.

```
```bash
kubectl port-forward --namespace knative-monitoring $(kubectl get pod --namespace knative-monitoring --selector="app=grafana" --output jsonpath='{.items[0].metadata.name}') 8080:3000
```

```

The following Dashboard talks about the HTTP requests for the Knative Serving Visualization which we will be serving per/sec request.

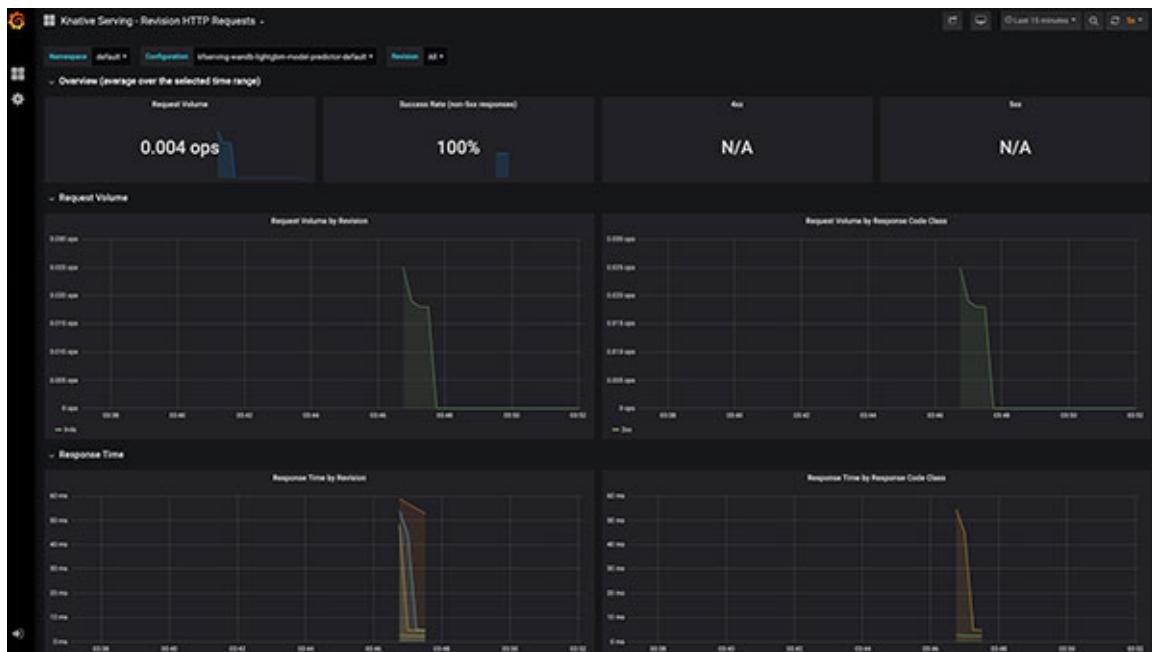


Figure 6.38: Grafana Dashboard HTTP Request

So, the following is the Dashboard for the Control Plane which shows the CPU Consumption and the memory usage efficiency.

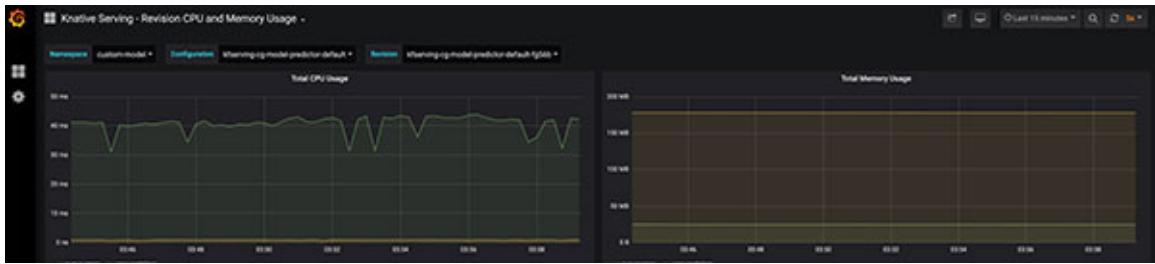


Figure 6.39: Grafana CPU/Memory Dashboard

6.7 Conclusion

In this chapter, we learned how to build an end-to-end LightGBM framework in Weights & Biases which captures our model in each run and output metrics. We built the sweep and Wandb config for the LightGBM algorithm parameters alongside the parallel coordinates plot for the Hyperparameter search.

We also deployed Kubeflow on the Kubernetes Platform and the model in Kubernetes cluster with KF serving and monitored our prediction results in the Grafana Dashboard.

In this chapter, we also gained knowledge on how to leverage the power of Google Cloud Platform, and use our DevOps knowledge with Machine Learning to become an MLops.

6.8 References

- <https://plotly.com/python/builtin-colorscales/>
- <https://technowhisp.com/kaggle-api-python-documentation/>
- <https://www.kaggle.com/numerai/encrypted-stock-market-data-from-numerai>
- <https://github.com/Kaggle/kaggle-api>
- <https://lightgbm.readthedocs.io/en/latest/Parameters.html>
- <https://www.wandb.com/>
- <https://github.com/kubeflow/kfserving>

CHAPTER 7

Applied ML with AWS SageMaker

In this chapter, we will work on the Housing Price Sales Dataset project, where we will completely run, evaluate, and deploy the model in the Amazon SageMaker Cloud environment and use S3 for Data Storage. We will also be using the in-built container algorithm XG-Boost for Model Building, so that we are able to understand the architecture of SageMaker Model Building framework end to end.

Structure

In this chapter, we will cover the following topics:

- Problem statement
- Getting started in AWS SageMaker setup
- Getting started with JupyterLab Notebook instances and SDK and S3 Bucket
- Getting started with launching notebook and loading data to S3
- Load, analyse, and transform the training data
- Amazon SageMaker training model
- Amazon SageMaker model deployment

Objectives

After studying this chapter, you will be able to understand the following:

- How to load and push data in Amazon S3 which is used for data storage.
- Outlier analysis, feature transformation, and imputation of categorical and numerical columns.
- How to create notebook in Amazon SageMaker and build a model and deploy it. Here, we will use XG-Boost in-built algorithm.
- The metrics and performance of our deployed model in Amazon CloudWatch.

7.1 Problem

Here, a home buyer described their dream house, and they probably won't begin with the height of the basement ceiling or the proximity to an east-west railroad.

So, we have 79 variables describing what covers almost each and every aspect of the residential homes in Ames, Iowa; this challenges to predict the final price of each home.

| | |
|------|---|
| NOTE | Rest all the imports I have showed in my Jupyter Notebook, for which the hyperlink of GitHub Account of this chapter is given. Note to use Anaconda Package Python 3.x. |
| CODE | https://github.com/bpbpublications/Continuous-Machine-Learning-with-Kubeflow/tree/main/Chapter7 |

7.2 Getting started in AWS SageMaker setup

So, we will start our journey with the AWS Account setup. To complete the setup, complete the following steps:

1. Create an AWS account email address and password to sign in as the **AWS account root user** and navigate to the IAM console at the following link:
<https://console.aws.amazon.com/iam/>.

The screenshot shows the AWS Sign In interface. On the left, there are two radio button options: 'Root user' (selected) and 'IAM user'. Below each option is a brief description. A text input field for 'Root user email address' contains 'username@example.com'. A large blue 'Next' button is centered below the input field. At the bottom, there are links for 'New to AWS?' and 'Create a new AWS account'. On the right, a dark sidebar features the text 'Build highly accurate training datasets' and 'And reduce data labeling costs by up to 70% with Amazon SageMaker Ground Truth'. It includes a 'Learn more »' button and the 'aws machine learning' logo. A stylized graphic of a brain connected to chemical structures is also present.

About Amazon.com Sign In
Amazon Web Services uses information from your Amazon.com account to identify you and allow access to Amazon Web Services. Your use of this site is governed by our Terms of Use and Privacy Policy linked below. Your use of Amazon Web Services products and services is governed by the AWS Customer Agreement linked below unless you have entered into a separate agreement with Amazon Web Services or an AWS Value Added Reseller to purchase these products and services. The AWS Customer Agreement was updated on March 31, 2017. For more information about these updates, see Recent Changes.

Figure 7.1: AWS Sign-in Console

2. Next, enable the access of the billing data for the IAM admin user that you will create:

- Choose **My Account** on the navigation bar of your account name.
- Choose **Edit** Next to **IAM User and Role Access to Billing Information**.
- Next, select the check box to **Activate IAM Access** and choose the **Update**.

▼ IAM User and Role Access to Billing Information

You can give IAM users and federated users with roles permissions to access billing information. This includes access to Account Settings, Payment Methods, and Report pages. You control which users and roles can see billing information by creating IAM policies. For more information, see [Controlling Access to Your Billing Information](#).



Figure 7.2: AWS My account. Dashboard

3. Choose **Services** from the navigation bar, and then **IAM** to go to the IAM dashboard.

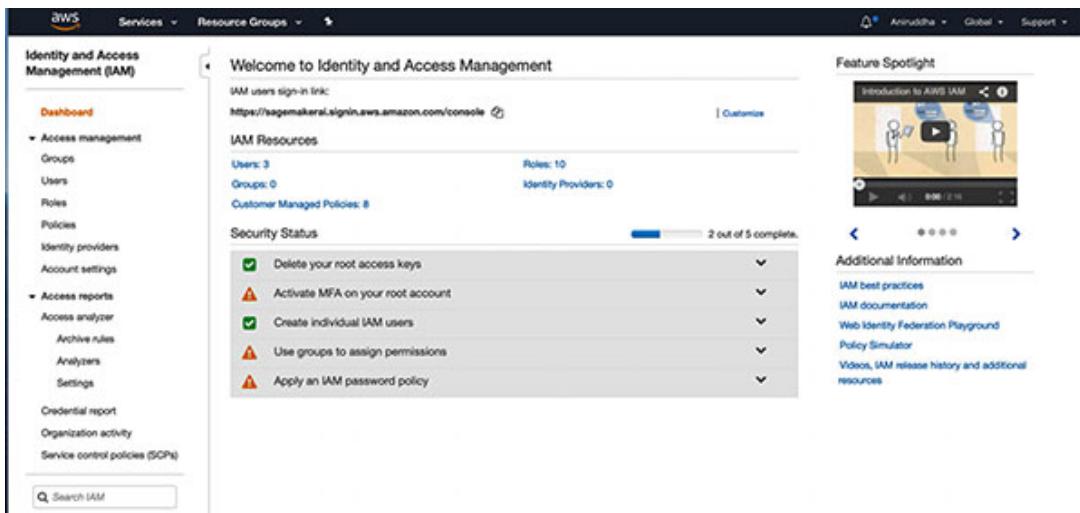


Figure 7.3: AWS IAM Dashboard

Optional: You can customise the Console Login Link given below, from the Access management console in IAM, where there is a customise option to change <Alias>.

4. Click on **Users** and add new users:

- Type **myrole**.
- Now, select the check box next to the AWS Management Console access; then select custom password; next, type your own new password in the text box. So by default, AWS will force the new user to create a new password when you first log into AWS. So, we can optionally uncheck the box next to the User, as the user must create a new password at the next sign-in to allow the new user to reset their password after they sign in.

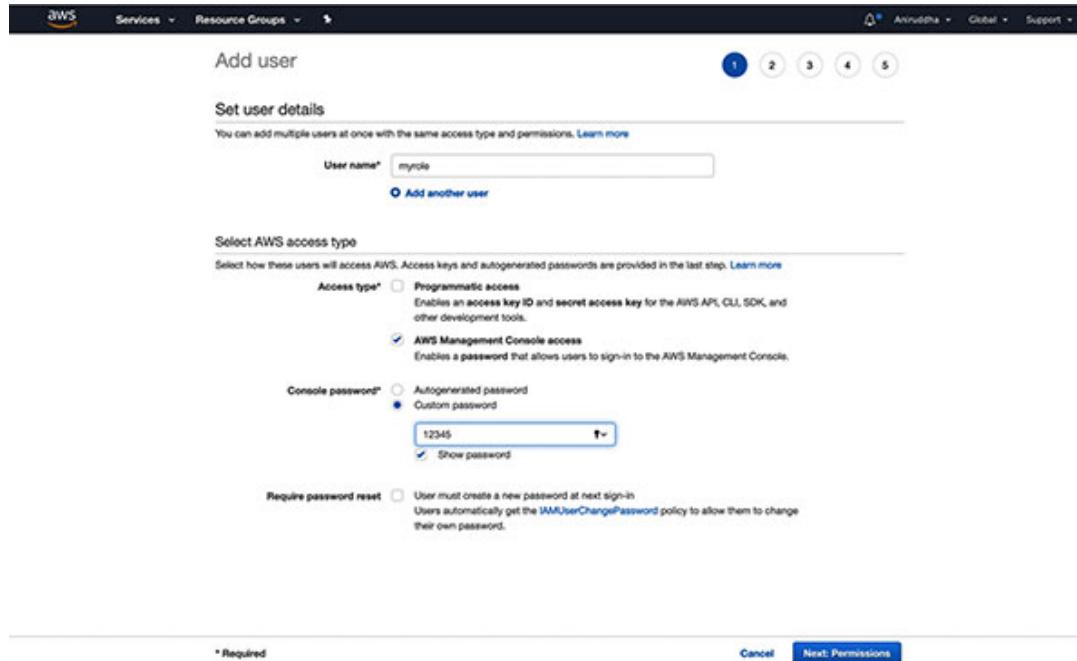


Figure 7.4: AWS Add User Screen

- Choose **Next: Permissions**.
- On the Set permissions page, choose Attach existing policies directly.
- Click on **Next: Review**.
- Click on **Create user**.
- Download the csv file.

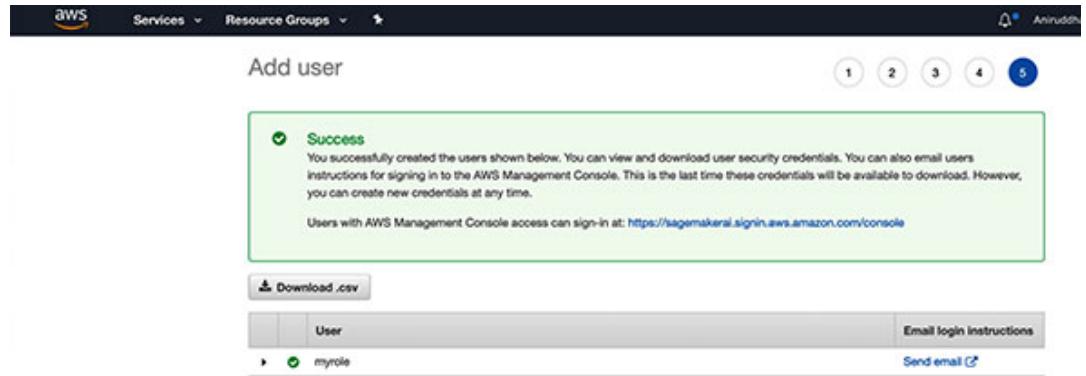


Figure 7.5: AWS User Creation Last Step Screen

Next, open the csv file and copy the link to your browser to login as myrole Username and your Custom Password. Optional: You can customise the Console Login Link below, from Access management console in IAM where there is a customise option to change <Alias>.

The following is the screenshot of the csv file which we have downloaded:

| | A | B | C | D | E | F | G | H | I |
|---|-----------|----------|---------------|-------------------|--|---|---|---|---|
| 1 | User name | Password | Access key ID | Secret access key | Console login link | | | | |
| 2 | myrole | | | | https://<AccountId>/Alias>.signin.aws.amazon.com/console | | | | |

Figure 7.6: AWS IAM Role Creation csv file credentials

5. Next, login as an IAM User and search for **sagemaker**.

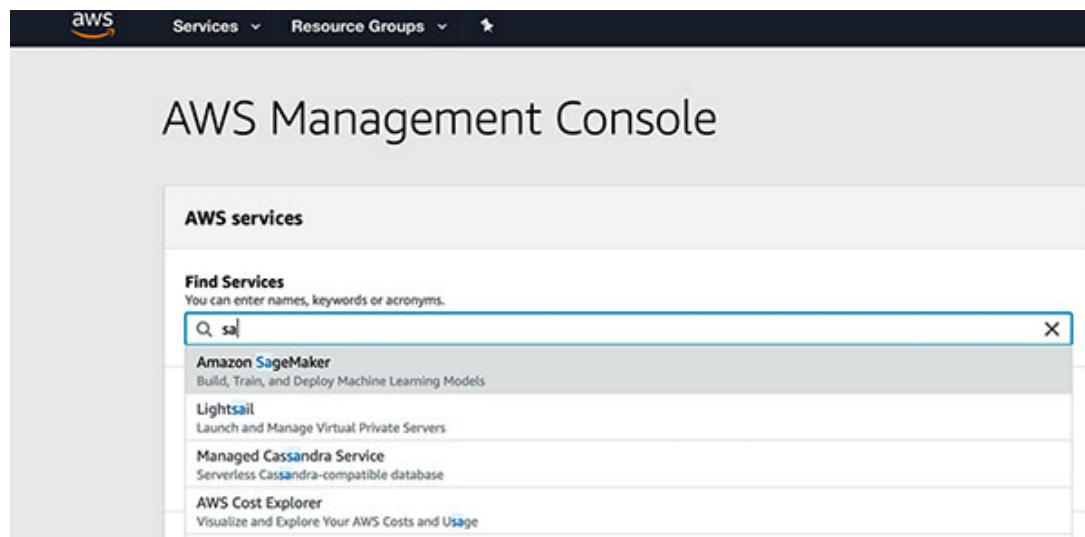


Figure 7.7: AWS Service search dashboard

Here, we will search for the **AmazonSagemaker**

7.3 Getting Started with JupyterLab Notebook Instances and SDK & S3 Bucket

7.3.1 Create an S3 Bucket

Now, the training of a model will produce the following:

- The model training data which is transformed is stored in S3.
- Model artifacts, which the Amazon SageMaker generates during the model training.

We can save all these things in the **Amazon Simple Storage Service (Amazon S3)** bucket; we can store the datasets that we have used during our training data and the model artifacts that are the output of a training job in a single bucket or in two separate buckets.

The following is the screenshot of how the screen will look, once we log in:

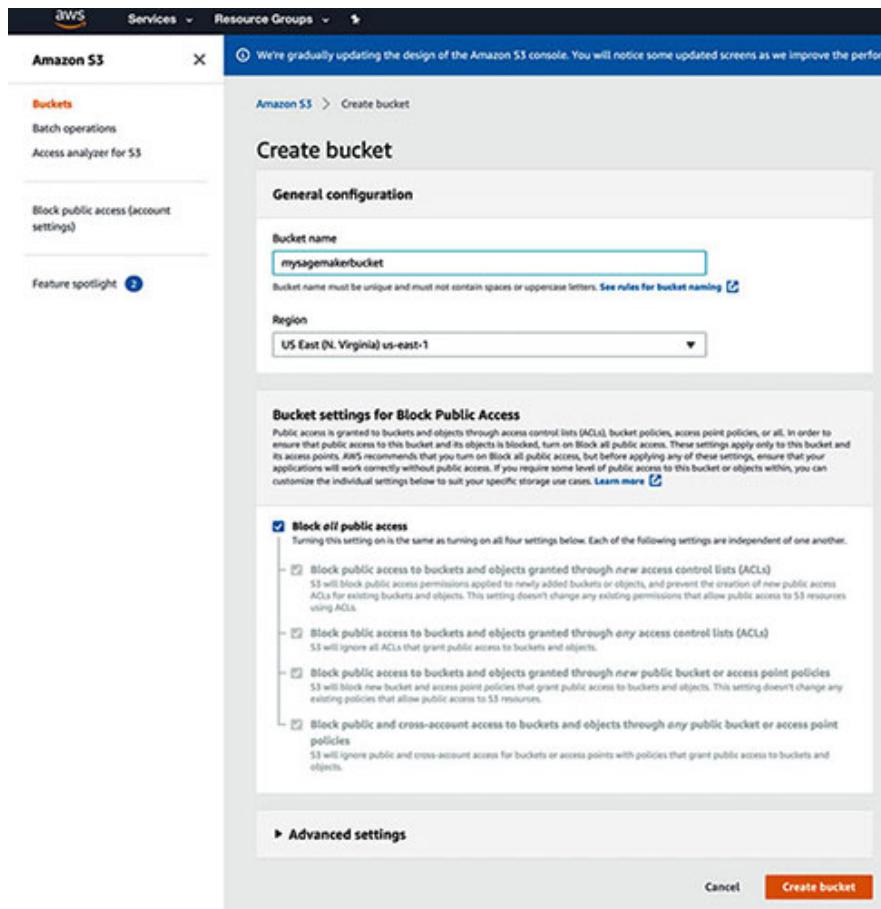


Figure 7.8: AWS S3 Create Bucket Screen

Now, your Bucket (**mysagemakerbucket**) has been created.

| Buckets (3) | | | |
|-------------------|---------------------------------|------------|--------------------------|
| Name | Region | Access | Bucket created |
| mysagemakerbucket | US East (N. Virginia) us-east-1 | Not Public | 2020-04-05T08:26:56.000Z |

Figure 7.9: AWS S3 Bucket Dashboard

7.3.2 Create an Amazon SageMaker Notebook Instance

Navigate to the Amazon SageMaker console > Notebook instances > create notebook instance. The steps are as follows:

1. On the Create notebook instance page, provide the following information (if a field is not mentioned, leave the default values):
 - In the Notebook instance name section, type a name for your notebook instance.
 - Then, for the Instance type, choose `ml.t2.medium`. This is the least expensive instance type and is the recommended type for the notebook instances support.

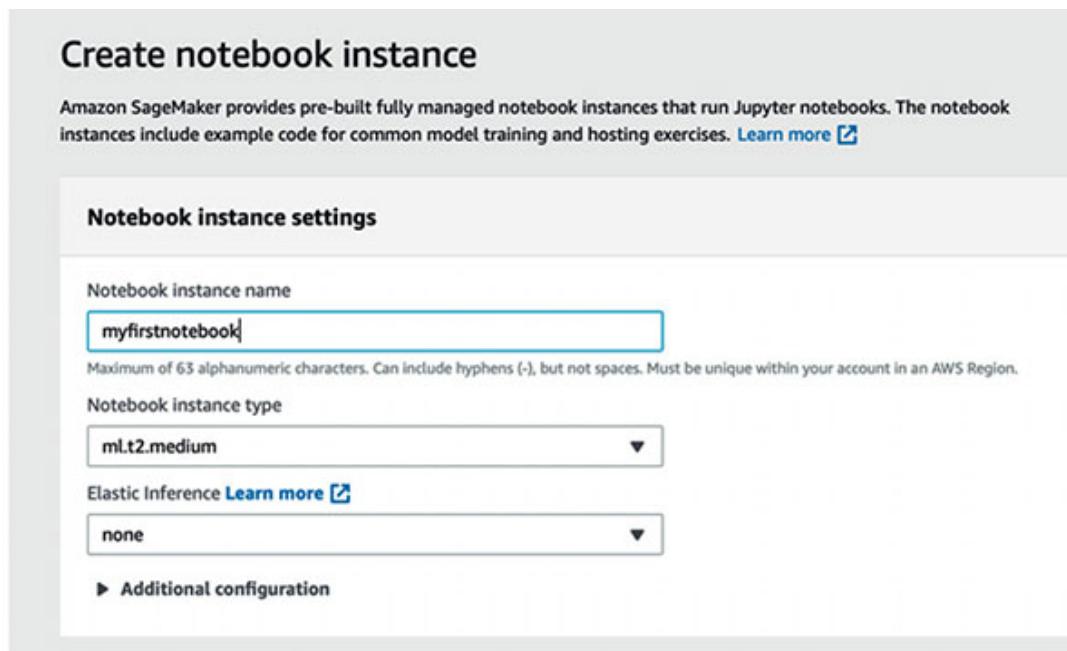


Figure 7.10: AWS SageMaker Create Notebook Screen

- For the IAM role, choose Create a new role, then choose Create role.
- Choose Create notebook instance.

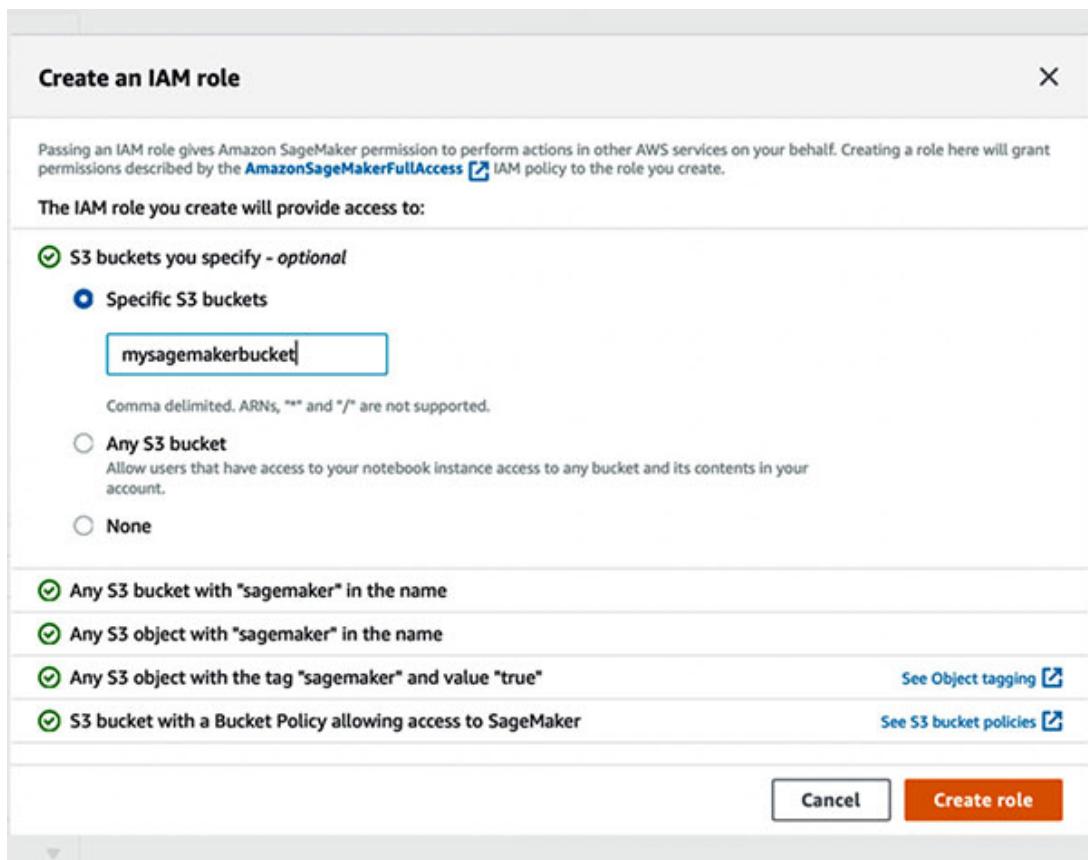


Figure 7.11: Create IAM Role screen

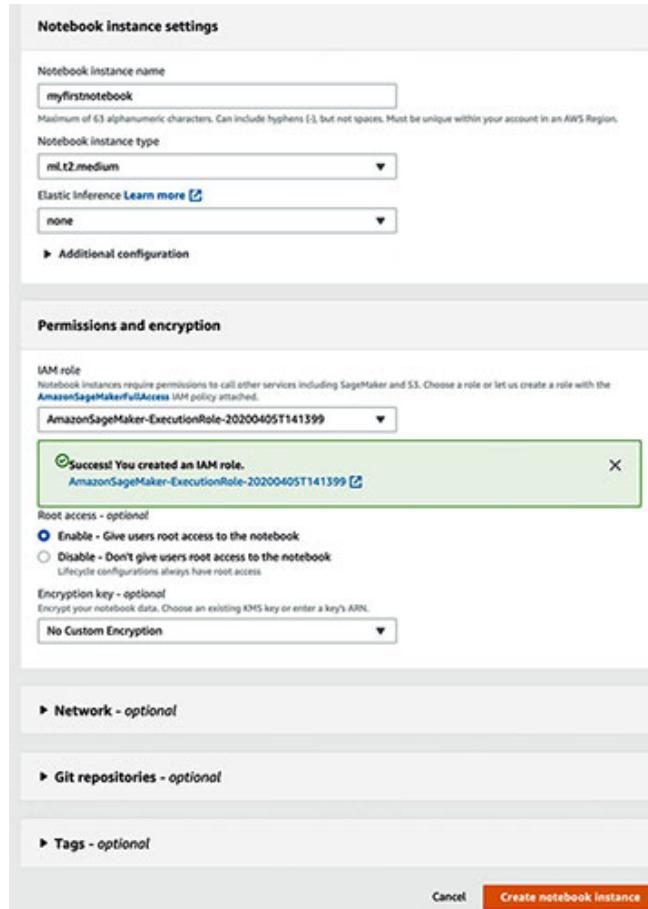


Figure 7.12: Create Notebook Screen

Wait for a few minutes, the Amazon SageMaker will launch an ML compute instance — in this case, it will be the notebook instance — and it attaches an ML storage volume to it. The notebook instance has a preconfigured Jupyter notebook server /JupyterLab and a set of Anaconda libraries.

2. Launch the JupyterLab. The following screenshot shows how the deck will look like:

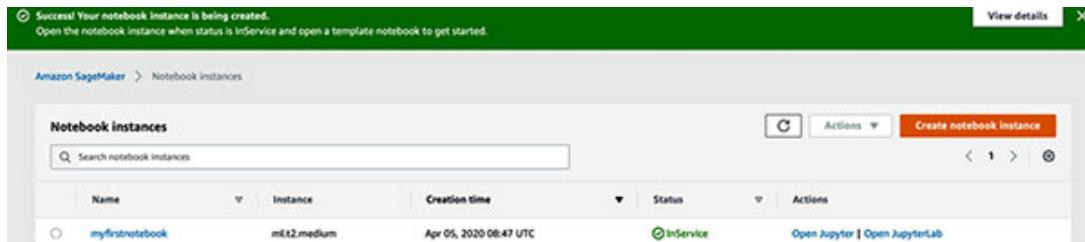


Figure 7.13: AWS Notebook Instances Dashboard

3. Next, we have to add a few IAM polices, so click on the Notebook instance name **myfirstnotebook**, as shown in the following screenshot:

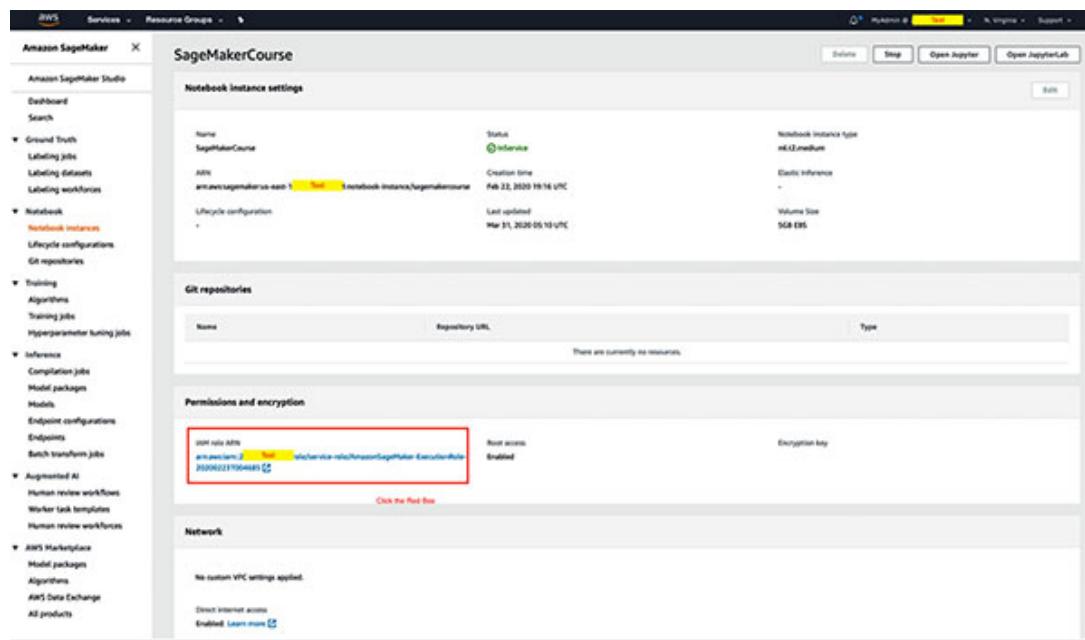


Figure 7.14: AWS Notebook Dashboard which we created

4. And then click on the IAM role with ARN; it will redirect to a new page. Now, if the following policy is not there, then click on attach policy, and search those policy names and click on the check box.

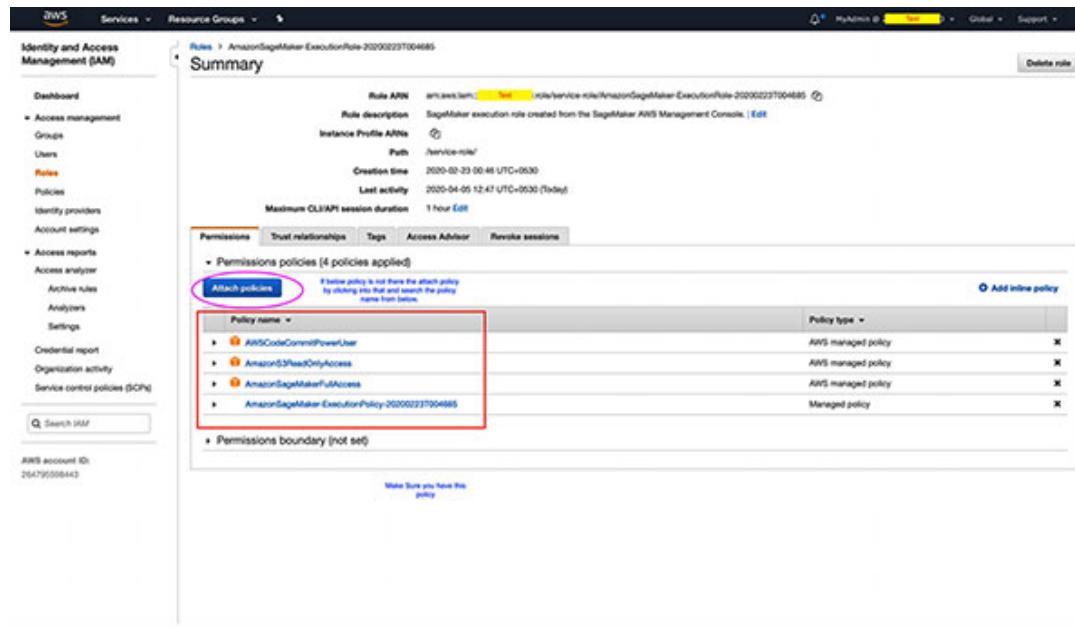


Figure 7.15: AWS Notebook IAM Role Policy Dashboard

Make sure all the red-boxed policies are attached in our IAM or else search and attach those policies.

7.4 Getting Started by Launching Notebook and loading data to S3

To get started for launching the Jupyter Notebook server, complete the following steps:

1. Now, we will create a notebook.

- When we open the notebook in the JupyterLab classic view, on the **Files** tab, choose **New**, and **conda_python3**. Here, the preinstalled environment includes the default Anaconda installation and Python 3.

2. In the Jupyter notebook, choose **File** and **Save as**, and name the notebook.

Next, we will have a look at the Notebook options which will come across once we click on that.

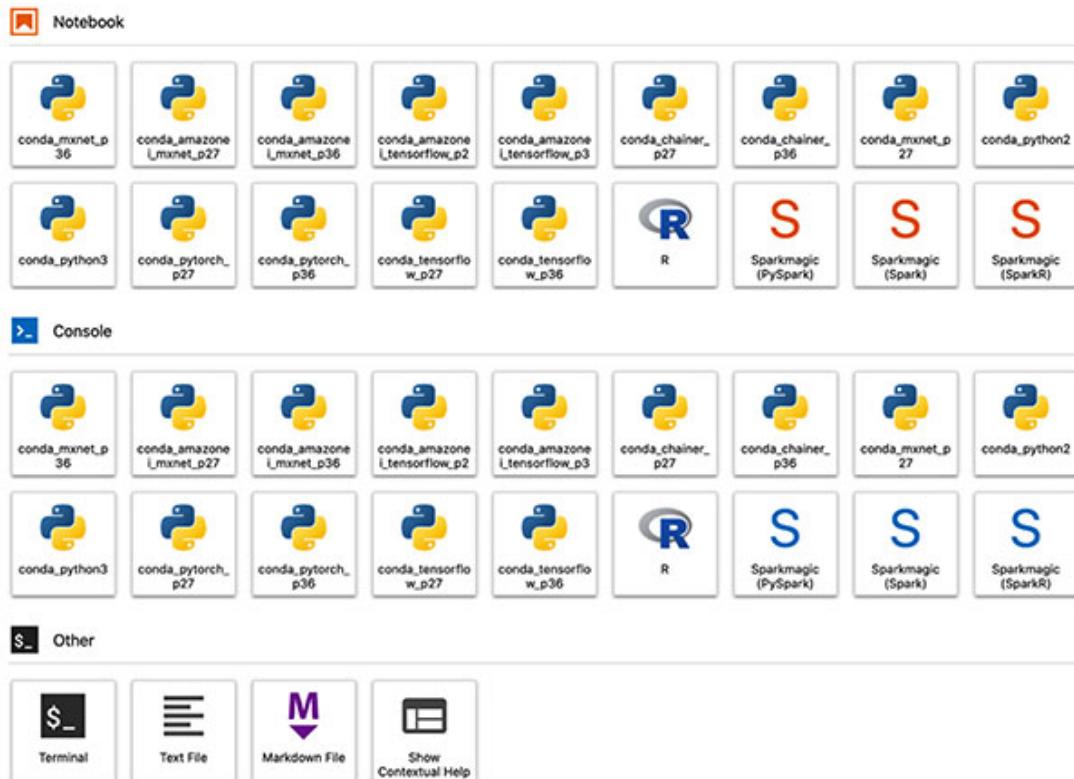


Figure 7.16: JupyterLab default Screen

3. Load the data into the S3 Bucket (**mysagemaker**) by creating a folder (**houseprice**), then in another folder (**rawdata**), upload the **train.csv** file.

Amazon S3> /mysagemakerbucket >/houseprice >/rawdata >train.csv



Figure 7.17: Amazon S3 Bucket file path

So, the preceding example is the S3 bucket location and file.

7.5 Load, Analyse, and Transform the Training Data

Here we will complete some of the following steps:

- Load the data from S3 and Python library.
- Feature Engineering of the raw data like Imputation and Outlier Detection.
- Then, log Transform our Target feature and check the correlation and Normal distribution plots.
- Transform the categorical features with the Label Encoding and Dummies.

7.5.1 Data Loading from S3 and Library

Now we will load all the required Python Library for our model building and deployment.

```
import numpy as np
import pandas as pd
import boto3
import re
import sagemaker
from sagemaker import get_execution_role
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.base import TransformerMixin
from warnings import filterwarnings
filterwarnings('ignore')
from sklearn.base import BaseEstimator, TransformerMixin,
RegressorMixin, clone
from sklearn.neighbors import LocalOutlierFactor
```

Load the IAM Role which has been created during the notebook instance.

```
role = get_execution_role()
role
```

```
[2]: 'arn:aws:iam::Account ID :role/service-role/AmazonSageMaker-ExecutionRole-20200223T004685'
```

Figure 7.18: AWS-ARN-IAM path

Now, load the raw data from the S3 bucket, and give the address where it's located in the **houseprice** folder.

```
bucket_name = 'mysagemakerbucket'. # Your Bucket name
raw_folder=r'houseprice/rawdata/train.csv'
s3_raw_file_location =r's3://{}{}'.format(bucket_name, raw_folder)
```

Now, load the file as a pandas dataframe; it will access the S3 location.

```
raw_data=pd.read_csv(s3_raw_file_location)
raw_data.head()
```

```
[5]:
```

| | Id | MSSubClass | MSZoning | LotFrontage | LotArea | Street | Alley | LotShape | LandContour | Utilities | ... | PoolArea | PoolQC | Fence | MiscFeature | MiscVal |
|---|----|------------|----------|-------------|---------|--------|-------|----------|-------------|-----------|-----|----------|--------|-------|-------------|---------|
| 0 | 1 | 60 | RL | 65.0 | 8450 | Pave | NaN | Reg | Lvl | AllPub | ... | 0 | NaN | NaN | NaN | 0 |
| 1 | 2 | 20 | RL | 80.0 | 9600 | Pave | NaN | Reg | Lvl | AllPub | ... | 0 | NaN | NaN | NaN | 0 |
| 2 | 3 | 60 | RL | 68.0 | 11250 | Pave | NaN | IR1 | Lvl | AllPub | ... | 0 | NaN | NaN | NaN | 0 |
| 3 | 4 | 70 | RL | 60.0 | 9550 | Pave | NaN | IR1 | Lvl | AllPub | ... | 0 | NaN | NaN | NaN | 0 |
| 4 | 5 | 60 | RL | 84.0 | 14260 | Pave | NaN | IR1 | Lvl | AllPub | ... | 0 | NaN | NaN | NaN | 0 |

5 rows × 81 columns

Figure 7.19: Raw DataFrame

7.5.2 Feature Engineering

Now, we will complete the feature engineering steps, prior to our model training process.

7.5.2.1 Finding Categorical & Numerical Columns

Here, we will find out the numerical and categorical columns by Object type including and excluding all the features.

```
num_cols = raw_data.select_dtypes(exclude='object').columns
print('{} Numeric columns \n{}'.format(len(num_cols), num_cols))
categ_cols = raw_data.select_dtypes(include='object').columns
print('\n{} Categorical columns \n{}'.format(len(categ_cols),
categ_cols))
```

```

38 Numeric columns
Index(['Id', 'MSSubClass', 'LotFrontage', 'LotArea', 'OverallQual',
       'OverallCond', 'YearBuilt', 'YearRemodAdd', 'MasVnrArea', 'BsmtFinSF1',
       'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF',
       'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath',
       'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'TotRmsAbvGrd',
       'Fireplaces', 'GarageYrBlt', 'GarageCars', 'GarageArea', 'WoodDeckSF',
       'OpenPorchSF', 'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea',
       'MiscVal', 'MoSold', 'YrSold', 'SalePrice'],
      dtype='object')

43 Categorical columns
Index(['MSZoning', 'Street', 'Alley', 'LotShape', 'LandContour', 'Utilities',
       'LotConfig', 'LandSlope', 'Neighborhood', 'Condition1', 'Condition2',
       'BldgType', 'HouseStyle', 'RoofStyle', 'RoofMatl', 'Exterior1st',
       'Exterior2nd', 'MasVnrType', 'ExterQual', 'ExterCond', 'Foundation',
       'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2',
       'Heating', 'HeatingQC', 'CentralAir', 'Electrical', 'KitchenQual',
       'Functional', 'FireplaceQu', 'GarageType', 'GarageFinish', 'GarageQual',
       'GarageCond', 'PavedDrive', 'PoolQC', 'Fence', 'MiscFeature',
       'SaleType', 'SaleCondition'],
      dtype='object')

```

Figure 7.20: Output Numerical & Categorical Features

Now, we have a total of 38 Numeric Columns and 43 Categorical Columns out of the 81 columns.

7.5.2.2 Checking the missing values sum

Now, we will check the missing values from the table and then we will prepare for the different imputation technique.

```
df_na = (raw_data.isnull().sum()) / len(raw_data) * 100
df_na = df_na.drop(df_na[df_na==0].index).sort_values(ascending=False)
```

We will plot the missing values count or sum in a bar plot.

```
with plt.rc_context(rc={'font.size':14}):
    fig, ax = plt.subplots(figsize=(16, 6))
    sns.barplot(df_na.index, df_na, palette="pastel", ax=ax)
    ax.set(xlabel='Features', ylabel='Missing values percentages')
    ax.tick_params(axis='x', rotation=55)
```

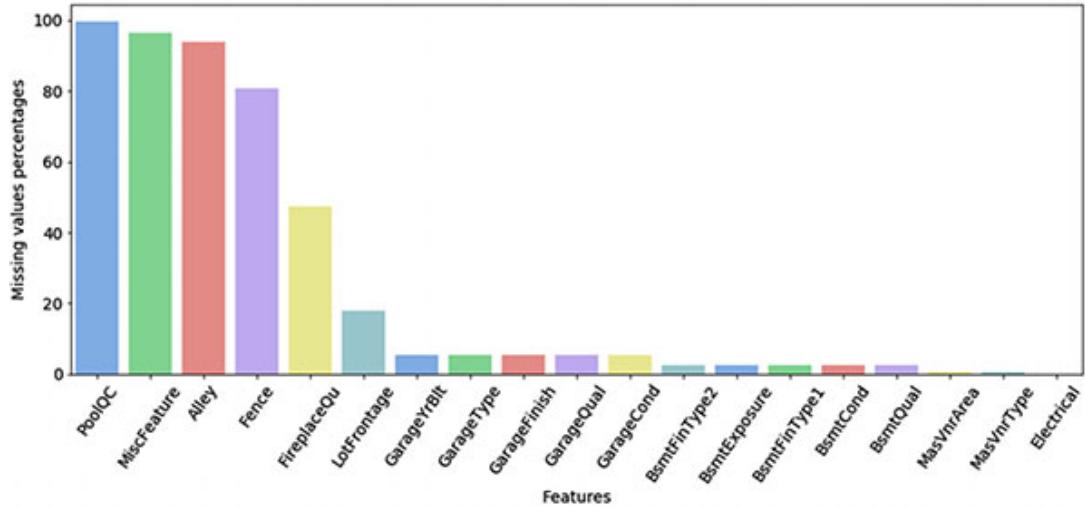


Figure 7.21: Missing Values Bar plot of Features

In the preceding screenshot, we can see the order from the most number of Missing values with PoolQC, Alley, and so on; now we have to fill those columns for our model building.

7.5.2.3 Log transformation of dependent feature

Now, we will transform our model dependent column to reduce the skewness and maintain normal distribution across the mean.

```
def skew_distribution(data, col='SalePrice'):
    fig, ax1 = plt.subplots()
    sns.distplot(data[col], ax=ax1, fit=stats.norm)
    (mu, sigma) = stats.norm.fit(data[col])
    ax1.set(title='Normal distribution ($\mu=$ {:.2f} and $\sigma=$ {:.2f})'.format(mu, sigma))
    fig, ax2 = plt.subplots()
    stats.probplot(data[col], plot=plt)
    print('The {} skewness is {:.2f}'.format(col, stats.skew(data[col])))
```

The distribution of the price and fit of normal distribution:

```
skew_distribution(raw_data, 'SalePrice')
```

So, the distribution plots will look like the following:

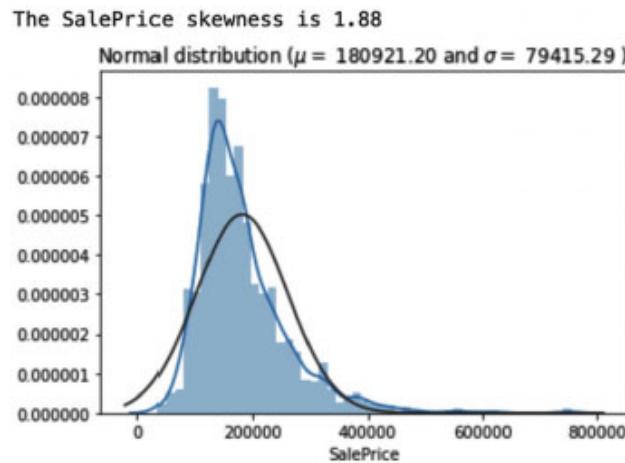


Figure 7.22: Normal Distribution Plot

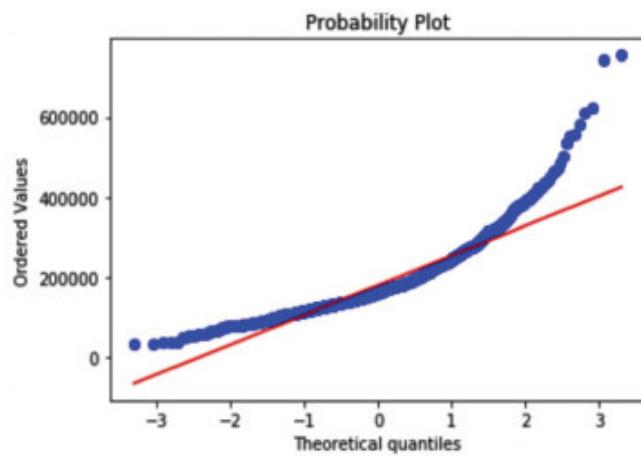


Figure 7.23: Probability plot

Therefore, we need to transform it into a more normal distribution, since the linear models will perform better. So, after the Log Transformation, the Normal distribution and P-P plot is as follows:

```
raw_data['SalePrice']=np.log1p(raw_data['SalePrice'])
skew_distribution(raw_data, 'SalePrice')
```

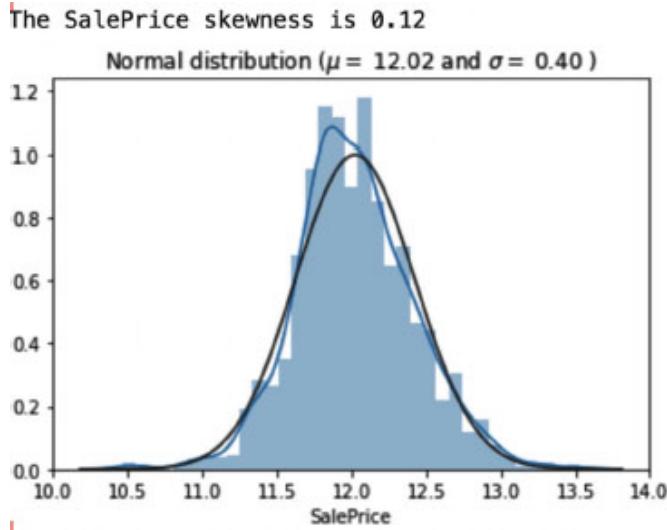


Figure 7.24: Normal Distribution Plot

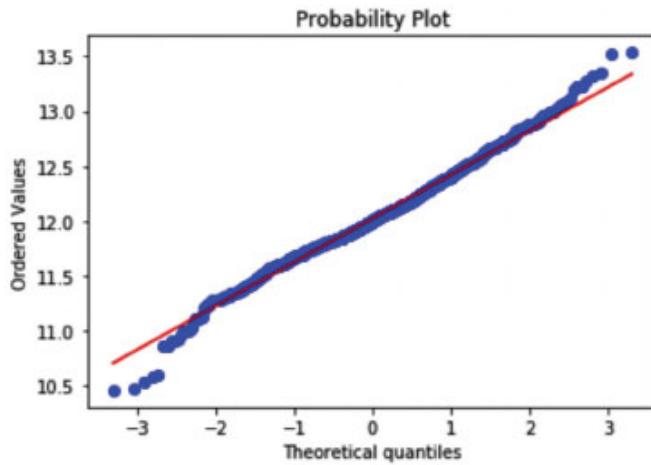


Figure 7.25: Probability plot

SalePrice is now more Gaussian, and the second plot which represents the probability plot shows that the distribution follows almost a normal distribution.

7.5.2.4 Correlation & Scatter Plots

Here, we will plot the correlation to check the relation between the variables and scatter the plots to check the outlier analysis with respect to the dependent feature House price.

```
corr = raw_data.corr()
top_correlation = corr['SalePrice'].sort_values(ascending=False)[:25]
threshold = 0.51
```

```

top_corr = corr.index[np.abs(corr["SalePrice"]) > threshold]

plt.figure(figsize=(10,8))
sns.heatmap(raw_data[top_corr].corr(), annot=True, cmap="RdBu_r")

```

So, the correlation plot will look like the following:

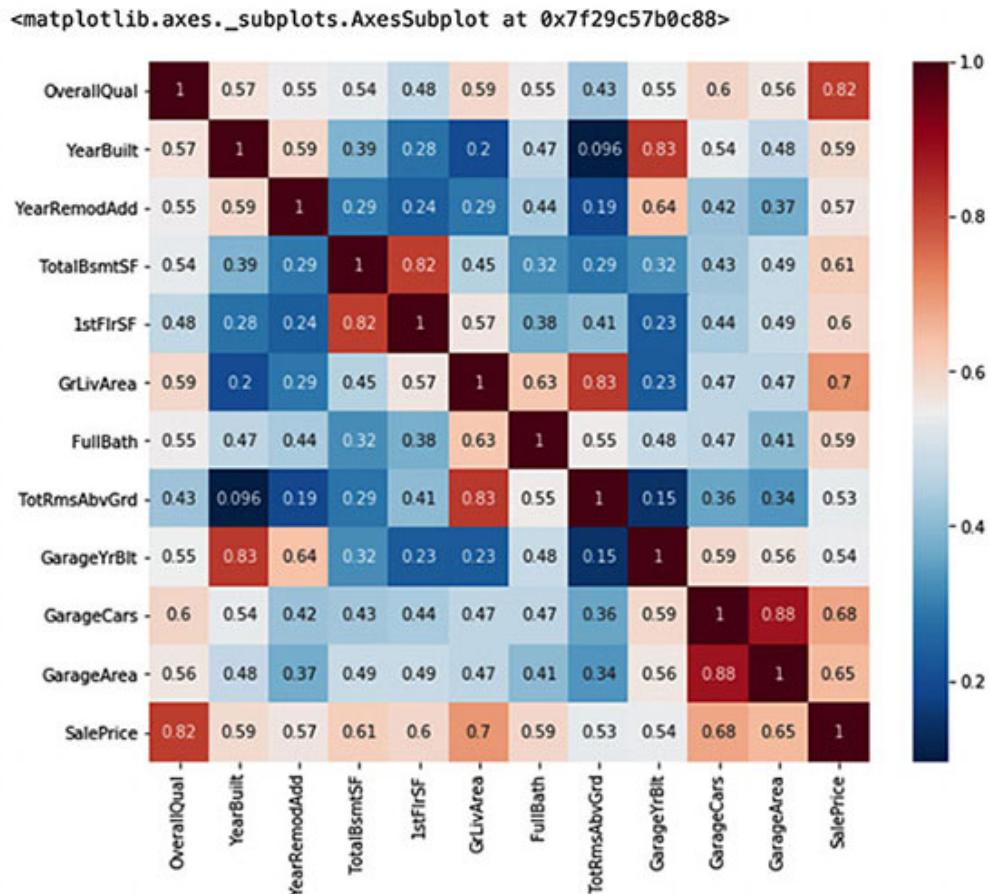


Figure 7.26: Top correlation Plot

Now, in the preceding screenshot, we can see that we plotted the top correlated features with the 0.51 threshold value and figured out a few columns which are very important with respect to the dependent columns.

We will find the mean correlation of the top numeric correlated features.

```

for col in top_correlation.index[:15]:
    print('{} - unique values: {} - mean: {:.2f}'.format(col,
        raw_data[col].unique()[:5], np.mean(raw_data[col])))

```

```

SalePrice - unique values: [12.24769912 12.10901644 12.31717117 11.84940484 12.4292202 ] - mean: 12.02
OverallQual - unique values: [7 6 8 5 9] - mean: 6.10
GrLivArea - unique values: [1710 1262 1786 1717 2198] - mean: 1515.46
GarageCars - unique values: [2 3 1 0 4] - mean: 1.77
GarageArea - unique values: [548 460 608 642 836] - mean: 472.98
TotalBsmtSF - unique values: [ 856 1262 920 756 1145] - mean: 1057.43
1stFlrSF - unique values: [ 856 1262 920 961 1145] - mean: 1162.63
FullBath - unique values: [2 1 3 0] - mean: 1.57
YearBuilt - unique values: [2003 1976 2001 1915 2000] - mean: 1971.27
YearRemodAdd - unique values: [2003 1976 2002 1970 2000] - mean: 1984.87
GarageYrBlt - unique values: [2003. 1976. 2001. 1998. 2000.] - mean: 1978.51
TotRmsAbvGrd - unique values: [8 6 7 9 5] - mean: 6.52
Fireplaces - unique values: [0 1 2 3] - mean: 0.61
MasVnrArea - unique values: [196. 0. 162. 350. 186.] - mean: 103.69
BsmtFinSF1 - unique values: [706 978 486 216 655] - mean: 443.64

```

Figure 7.27: Correlated Mean wrt. Target feature

So, the following columns are the important columns which we figured out for further analysis through the scatter plot.

```

cols = 'SalePrice GrLivArea GarageArea TotalBsmtSF YearBuilt 1stFlrSF
MasVnrArea TotRmsAbvGrd'.split()
with plt.rc_context(rc={'font.size':14}):
    fig, ax = plt.subplots(figsize=(16,13), tight_layout=True)
    pd.plotting.scatter_matrix(raw_data[cols], ax=ax, diagonal='kde',
    alpha=0.8)

```

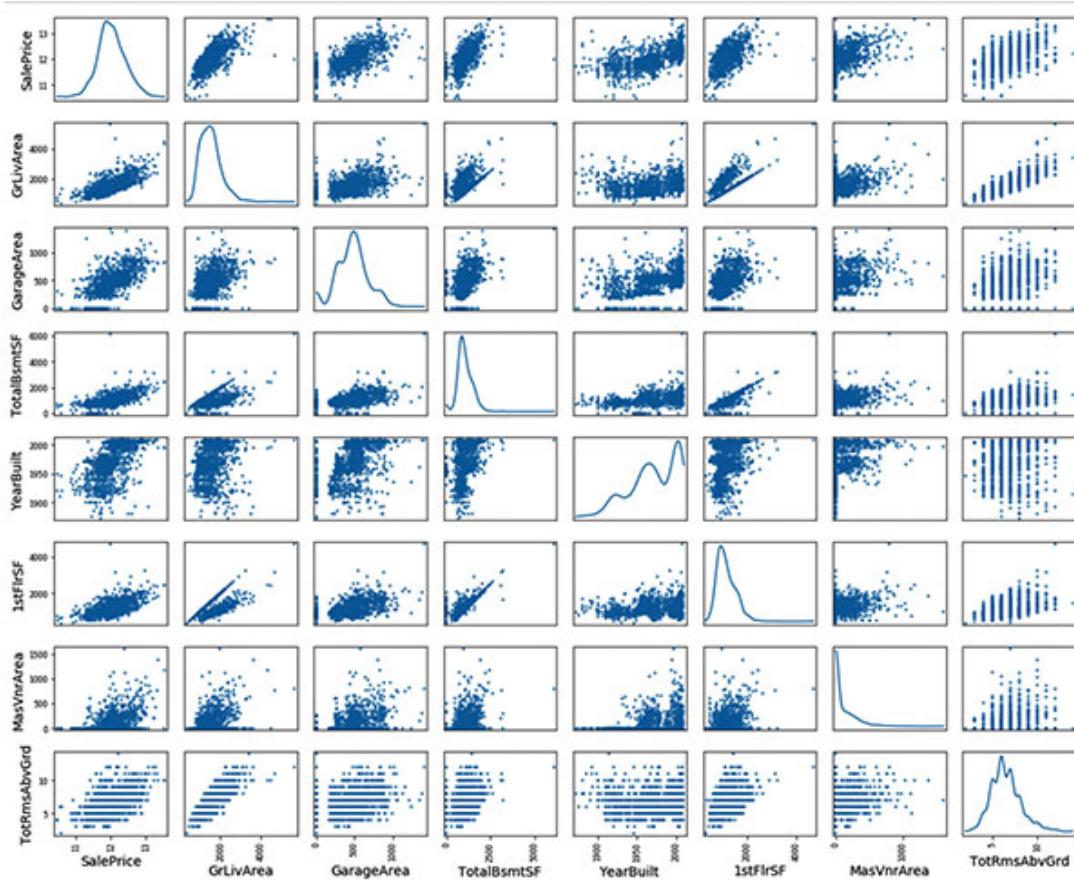


Figure 7.28: Scatter Plot

These scatter plots will give us some insight into a few outliers, that is, the values which resemble some incoherent/huge values, which will now impact the models. Thus, we will only remove a few of them since these ones are really important. We will remove the principal outliers in the scatter plots of (GrLivRea - GarageArea - TotalBsmtSF - 1stFlrSF - MasVnrArea - TotRmsAbvGrd) vs SalePrice).

7.5.2.5 Outlier Detection

Now, we will remove the outliers from our Dataset with Scikit-Learn LocalOutlierFactor Method.

```
def detect_outliers_plots(x, y, name, top=5, plot=True):
    lof = LocalOutlierFactor(n_neighbors=40, contamination=0.1)
    x_ = np.array(x).reshape(-1,1)
    preds = lof.fit_predict(x_)
    lof_scr = lof.negative_outlier_factor_
    out_idx = pd.Series(lof_scr).sort_values()[:top].index
```

```

if plot:
    f, ax = plt.subplots(figsize=(9, 6))
    plt.scatter(x=x, y=y, c=np.exp(lof_scr), cmap='RdBu')
    ax.set(ylabel='SalePrice', xlabel=name)
return out_idx

```

Extract the 8 columns in a separate table and list and drop the null values.

```

outlier_ana_data=raw_data[['SalePrice', 'GarageArea', 'TotalBsmtSF',
'YearBuilt', '1stFlrSF', 'MasVnrArea', 'TotRmsAbvGrd', 'GrLivArea']]
outlier_ana_data.dropna(inplace=True)
cols_out = ['GarageArea', 'TotalBsmtSF',
'YearBuilt', '1stFlrSF', 'MasVnrArea', 'TotRmsAbvGrd', 'GrLivArea']
for i in cols_out:
    outs = detect_outliers_plots(outlier_ana_data[i],
outlier_ana_data['SalePrice'], i, top=5) #got 1298,523
    print(outs)

Int64Index([540, 515, 1364, 318, 1158], dtype='int64')
Int64Index([1290, 331, 495, 522, 439], dtype='int64')
Int64Index([1429, 802, 800, 282, 762], dtype='int64')
Int64Index([1290, 495, 522, 1018, 1365], dtype='int64')
Int64Index([1017, 1353, 976, 378, 384], dtype='int64')
Int64Index([531, 633, 203, 1094, 612], dtype='int64')
Int64Index([1290, 522, 1176, 688, 531], dtype='int64')

```

Figure 7.29: Output of Index Outlier

So, it will return the index of all the outliers in those 7 columns.

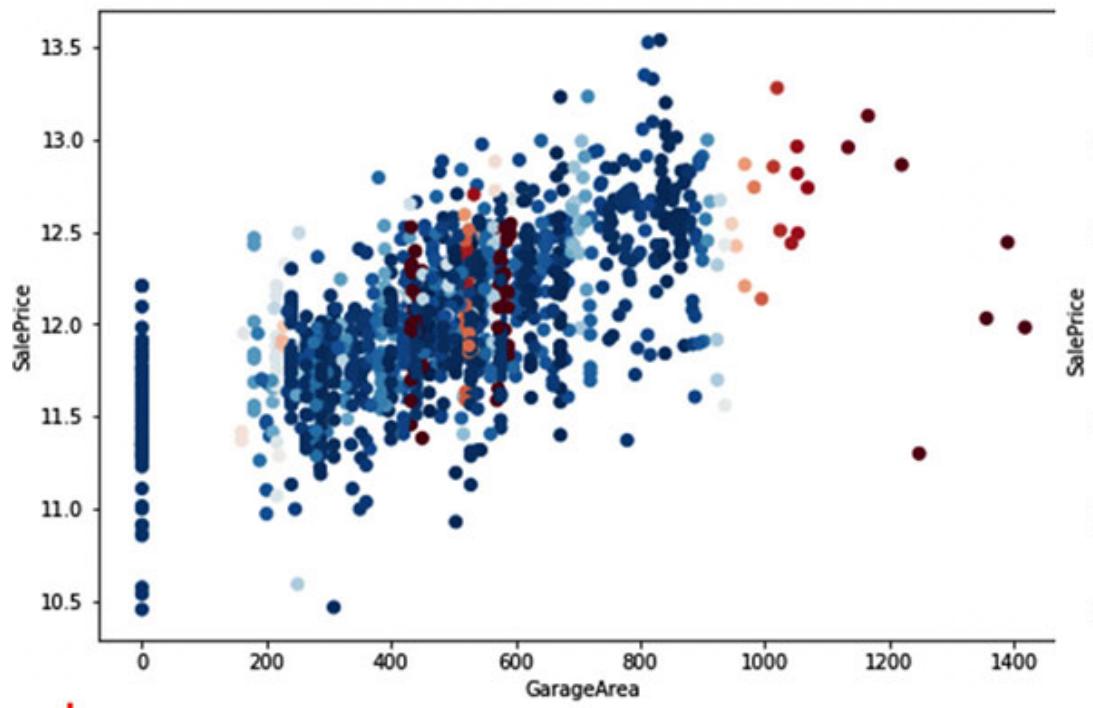


Figure 7.30: Normal Distribution Plot

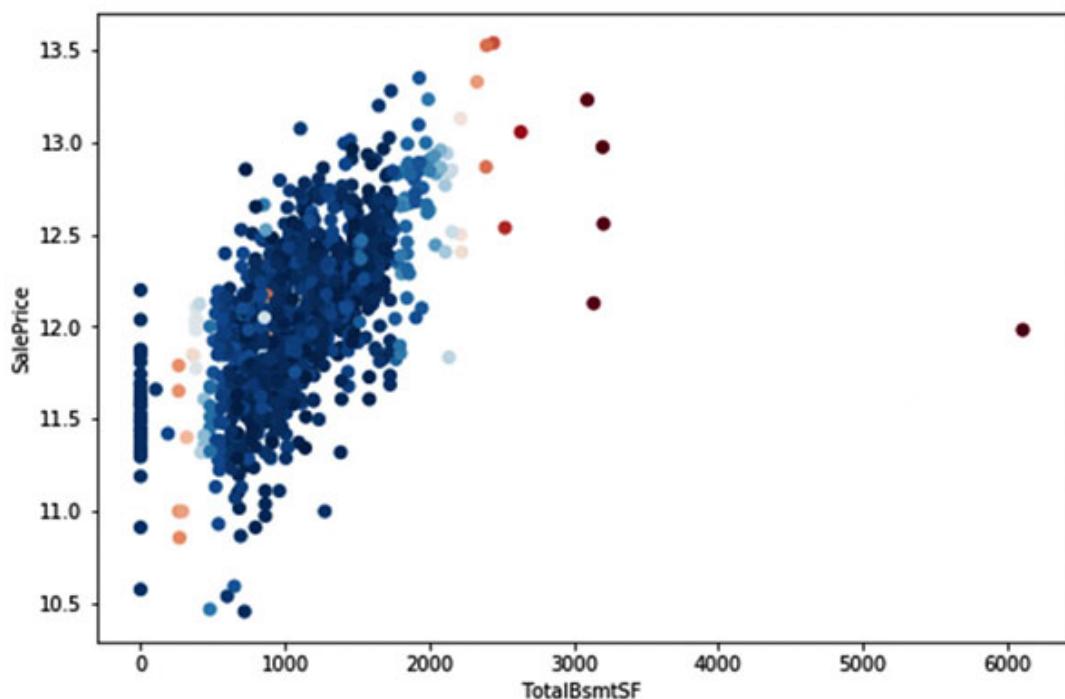


Figure 7.31: Probability plot

We have created a function and called it here to detect the outliers; you can find the code in GitHub, so don't get confused if you see this method in the following

section.

We extracted the most common outliers from the dataset and found out the outliers.

```
outliers = [30, 88, 462, 523, 632, 1298, 1324]
from collections import Counter
all_outliers=[]
numeric_features = raw_data.dtypes[raw_data.dtypes != 'object'].index
for feature in numeric_features:
    try:
        outs = detect_outliers(raw_data[feature],
                               raw_data['SalePrice'], top=5, plot=False)
    except:
        continue
    all_outliers.extend(outs)

print(Counter(all_outliers).most_common())
for i in outliers:
    if i in all_outliers:
        print(i)

[(0, 4), (533, 4), (1298, 4), (1270, 3), (375, 3), (523, 3), (635, 3), (634, 3), (976, 3), (975, 3), (978, 3), (977, 3), (313, 2), (335, 2), (916, 2), (1213, 2), (812, 2), (77, 2), (7, 2), (953, 2), (496, 2), (1182, 2), (954, 2), (597, 2), (1163, 2), (1350, 2), (1328, 2), (495, 2), (1459, 1), (1, 1), (1458, 1), (1457, 1), (164, 1), (873, 1), (589, 1), (555, 1), (249, 1), (706, 1), (451, 1), (636, 1), (1100, 1), (304, 1), (508, 1), (218, 1), (1442, 1), (1058, 1), (240, 1), (1166, 1), (591, 1), (277, 1), (771, 1), (1148, 1), (1223, 1), (699, 1), (219, 1), (229, 1), (790, 1), (930, 1), (1028, 1), (695, 1), (645, 1), (1149, 1), (125, 1), (599, 1), (574, 1), (332, 1), (448, 1), (1024, 1), (1373, 1), (431, 1), (1400, 1), (185, 1), (170, 1), (1089, 1), (88, 1), (691, 1), (738, 1), (188, 1), (326, 1), (624, 1), (298, 1), (1283, 1), (53, 1), (189, 1), (809, 1), (48, 1), (203, 1), (434, 1), (1218, 1), (642, 1), (166, 1), (309, 1), (605, 1), (1198, 1), (747, 1), (420, 1), (1340, 1), (542, 1), (1372, 1), (516, 1), (351, 1), (1068, 1), (499, 1), (63, 1), (1397, 1), (131, 1), (874, 1), (155, 1), (9, 1), (297, 1), (1859, 1), (341, 1), (1349, 1), (198, 1), (935, 1), (205, 1), (55, 1), (1437, 1), (1346, 1), (1012, 1), (1443, 1), (588, 1), (995, 1), (1423, 1), (810, 1), (1170, 1), (1386, 1), (867, 1), (51, 1), (968, 1), (30, 1)]
30
88
523
1298
```

Figure 7.32: Outlier Index Count Dictionary

```
Outliers_table=raw_data.loc[outliers]
Outliers_table=Outliers_table[cols_out]
Outliers_table
```

| | GarageArea | TotalBsmtSF | YearBuilt | 1stFlrSF | MasVnrArea | TotRmsAbvGrd | GrLivArea |
|------|------------|-------------|-----------|----------|------------|--------------|-----------|
| 30 | 250 | 649 | 1920 | 649 | 0.0 | 6 | 1317 |
| 88 | 0 | 1013 | 1915 | 1013 | 0.0 | 6 | 1526 |
| 462 | 360 | 864 | 1965 | 864 | 0.0 | 5 | 864 |
| 523 | 884 | 3138 | 2007 | 3138 | 762.0 | 11 | 4676 |
| 632 | 544 | 1386 | 1977 | 1411 | 209.0 | 6 | 1411 |
| 1298 | 1418 | 6110 | 2008 | 4692 | 796.0 | 12 | 5642 |
| 1324 | 895 | 1795 | 2006 | 1795 | 428.0 | 7 | 1795 |

Figure 7.33: Outlier Table

Now we will delete the outliers via the index lists.

```
raw_data = raw_data.drop(raw_data.index[outliers])
raw_data.reset_index(drop=True, inplace=True)
raw_data.shape()
```

[23]: (1453, 81)

Figure 7.34: New Data shape

So, we have removed 7 rows of outliers which were most common.

7.5.3 Feature Transformation

Here, group the columns into numerical, categorical, Label Encoder columns, and changing datatype of columns.

```
N= ['GarageYrBlt', 'MasVnrArea', 'GarageArea',
'GarageCars', 'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF',
'BsmtFullBath', 'BsmtHalfBath', 'MasVnrArea']
M= ['MSZoning', 'Electrical', 'KitchenQual', 'Exterior1st',
'Exterior2nd', 'SaleType']
S=['OverallCond', 'YrSold', 'MoSold', 'MSSubClass', 'GarageCars',
'Fireplaces', 'HalfBath', 'OverallQual']
L= ['PoolQC', 'MSSubClass', 'MasVnrType', 'Alley', 'MiscFeature',
'Fence', 'FireplaceQu', 'BsmtQual', 'BsmtCond',
'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2', 'GarageType',
'GarageFinish', 'GarageQual', 'GarageCond']
cols= ['BsmtExposure', 'BsmtFinType1', 'BsmtFinType2', 'BsmtQual',
'BsmtCond', 'GarageQual', 'GarageCond', 'GarageFinish', 'GarageType',
```

```

'FireplaceQu', 'ExterQual', 'ExterCond',
'HeatingQC', 'PoolQC', 'KitchenQual',
'Functional', 'Fence', 'LandSlope',
'LotShape', 'PavedDrive', 'Street', 'Alley', 'CentralAir',
'MSSubClass', 'OverallCond', 'GarageCars', 'YrSold', 'MoSold',
'Fireplaces', 'HalfBath']

```

So, the following **transformation()** function is used for the imputation of the numerical columns:

```

def transformation(data) -> pd.DataFrame:
    for i in N:
        data[i] = data[i].fillna(0)
    for i in M:
        data[i] = data[i].fillna(data[i].mode()[0])
    for i in L:
        data[i] = data[i].fillna("None")
    for i in S:
        data[i] = data[i].astype(str)
    data["LotFrontage"] = data.groupby("Neighborhood")
    ["LotFrontage"].transform(lambda x: x.fillna(x.median()))
    data["Functional"] = data["Functional"].fillna("Typ")
    data = data.drop(['Utilities'], axis=1)
    return data

```

Here **cat_transform()** is used for the categorical label encoding and dummies transformation:

```

def cat_transform(data)-> pd.DataFrame():
    le = LabelEncoder()
    for col in cols:
        data[col] = le.fit_transform(data[col])
    data = pd.get_dummies(data)
    return data

```

Here, the **add_features()** will add some important features which will help with our prediction:

```

def add_features(data)-> pd.DataFrame():
    data['TotalSF'] = data['TotalBsmtSF'] + data['1stFlrSF'] +
    data['2ndFlrSF'] + data['GrLivArea'] + data['GarageArea']

```

```

# Combine the bathrooms
data['Bathrooms'] = data['FullBath'] + data['HalfBath']* 0.5
# Combine Year built, Garage Year Built and Year Remod
# (with a coeff 0.5 since it's less correlated to Year Built than
the Garage year built).
data['YearMean'] = data['YearBuilt'] + data['YearRemodAdd'] * 0.5 +
data['GarageYrBlt']
return data

```

Now, we have loaded the test dataset without the target columns and here we have pre-processed together with the train data.

```

# We don't need the Id column so we save it
df_train_id = raw_data['Id']
df_test_id = test_data['Id']
raw_data.drop("Id", axis=1, inplace=True)
test_data.drop("Id", axis=1, inplace=True)

# same transformation to the train / test datasets to avoid
irregularities
size_train = len(raw_data.index)
size_test = len(test_data.index)
print((size_train),(size_test))

```

(1453, 81) (1459, 80)

Figure 7.35: Raw Data /Test Data shape

Now, we merged the Raw data and Test data and prepared for the data transformation.

```

df_tot = pd.concat([raw_data, test_data],
sort=False).reset_index(drop=True)
df_tot.drop(['SalePrice'], axis=1, inplace=True)
print(df_tot.shape)
df_tot.head()

```

| | (2912, 79) | | | | | | | | | | | | |
|-------|------------|----------|-------------|---------|--------|-------|----------|-------------|-----------|-----------|-----|-------------|--|
| [82]: | MSSubClass | MSZoning | LotFrontage | LotArea | Street | Alley | LotShape | LandContour | Utilities | LotConfig | ... | ScreenPorch | |
| 0 | 60 | RL | 65.0 | 8450 | Pave | NaN | Reg | Lvl | AllPub | Inside | ... | 0 | |
| 1 | 20 | RL | 80.0 | 9600 | Pave | NaN | Reg | Lvl | AllPub | FR2 | ... | 0 | |
| 2 | 60 | RL | 68.0 | 11250 | Pave | NaN | IR1 | Lvl | AllPub | Inside | ... | 0 | |
| 3 | 70 | RL | 60.0 | 9550 | Pave | NaN | IR1 | Lvl | AllPub | Corner | ... | 0 | |
| 4 | 60 | RL | 84.0 | 14260 | Pave | NaN | IR1 | Lvl | AllPub | FR2 | ... | 0 | |

5 rows × 79 columns

Figure 7.36: Merged Table

Let's print the shape of the table.

```
print (df_tot.shape)
df_tot = transformation(df_tot)
print(df_tot. shape)
df_tot = cat_transform(df_tot)
print(df_tot.shape)
```

Now, we will transform the categorical and impute the missing values, both the train and the test dataset; following which, we will just separate out our train data from the merged data and then add features. The following is the process for the train data:

```
x = df_tot[:size_train]
x=add_features(x)
print (x.shape)
x.head()
```

| | (2912, 79) | | | | | | | | | | | | |
|-------|------------|-------------|---------|--------|-------|----------|-----------|-------------|-----------|--------------|-----|-------------|----------------------|
| [84]: | MSSubClass | LotFrontage | LotArea | Street | Alley | LotShape | LandSlope | OverallCond | YearBuilt | YearRemodAdd | ... | SaleType_WD | SaleCondition_Abnorm |
| 0 | 10 | 65.0 | 8450 | 1 | 1 | 3 | 0 | 4 | 2003 | 2003 | ... | 1 | |
| 1 | 5 | 80.0 | 9600 | 1 | 1 | 3 | 0 | 7 | 1976 | 1976 | ... | 1 | |
| 2 | 10 | 68.0 | 11250 | 1 | 1 | 0 | 0 | 4 | 2001 | 2002 | ... | 1 | |
| 3 | 11 | 60.0 | 9550 | 1 | 1 | 0 | 0 | 4 | 1915 | 1970 | ... | 1 | |
| 4 | 10 | 84.0 | 14260 | 1 | 1 | 0 | 0 | 4 | 2000 | 2000 | ... | 1 | |

5 rows × 225 columns

Figure 7.37: Raw Train Data after transformation then separating from merge data

In the preceding screenshot, you can see the shape of the train data changed to (1453,225) after transformation. After that, we checked the Null values in our data train which is zero for all columns.

```
logSalesPrice=raw_data['SalePrice']
x.reset_index(drop=True, inplace=True)
x.insert(loc=0, column='SalePrice', value=logSalesPrice)
```

```
x.head()
```

| [48]: | SalePrice | MSSubClass | LotFrontage | LotArea | Street | Alley | LotShape | LandSlope | OverallCond | YearBuilt | ... | SaleType_WD | SaleCondition_Abnormal | S4 |
|-------|-----------|------------|-------------|---------|--------|-------|----------|-----------|-------------|-----------|-----|-------------|------------------------|----|
| 0 | 12.247699 | 10 | 65.0 | 8450 | 1 | 1 | 3 | 0 | 4 | 2003 | ... | 1 | 0 | 0 |
| 1 | 12.109016 | 5 | 80.0 | 9600 | 1 | 1 | 3 | 0 | 7 | 1976 | ... | 1 | 0 | 0 |
| 2 | 12.317171 | 10 | 68.0 | 11250 | 1 | 1 | 0 | 0 | 4 | 2001 | ... | 1 | 0 | 0 |
| 3 | 11.849405 | 11 | 60.0 | 9550 | 1 | 1 | 0 | 0 | 4 | 1915 | ... | 1 | 1 | 0 |
| 4 | 12.429220 | 10 | 84.0 | 14260 | 1 | 1 | 0 | 0 | 4 | 2000 | ... | 1 | 0 | 0 |

5 rows x 226 columns

Figure 7.38: Now the Final table with target column in first

Now, we will add our independent feature in the first column because we will use the in-built SageMaker algorithm XG-boost, and it will automatically take the first column as the target column and rest as the independent features.

7.6 Amazon SageMaker Training Model

Now, for our consideration, our transformed raw data into **Dataframe (x)**/**train.csv** file is our master table and we will split this into Train/Validation file. We will evaluate in the Validation dataset, and the test data which we will predict the Target feature sales price.

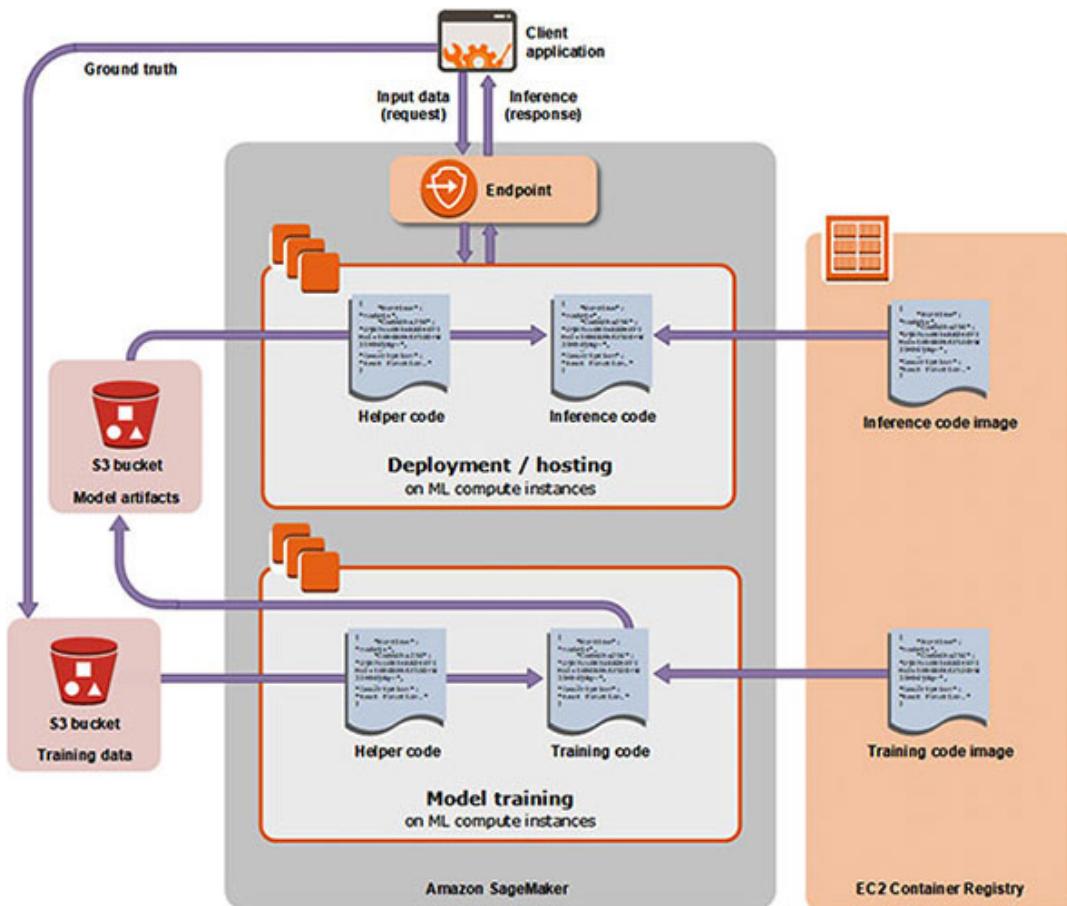


Figure 7.39: Amazon SageMaker Train & Deploy (Credit: AWS Docs)

7.6.1 Splitting Data into Train/Validation and push to S3

We will specify your bucket name and then give the path and folder a name, which we will create in S3.

```
bucket_name = 'mysagemakerbucket' # your bucket name

training_folder = r'houseprice/training/'
validation_folder = r'houseprice/validation/'
test_folder = r'houseprice/test/'

s3_model_output_location =
r's3://{0}/houseprice/model'.format(bucket_name)
s3_training_file_location =
r's3://{0}/{1}'.format(bucket_name, training_folder)
s3_validation_file_location =
r's3://{0}/{1}'.format(bucket_name, validation_folder)
s3_test_file_location =
r's3://{0}/{1}'.format(bucket_name, test_folder)
```

So, the model output and artifacts will store in the model output location, and the training of the model will store in the training location, the evaluation of model will be done from the validation data and we will predict our test data from the test file location.

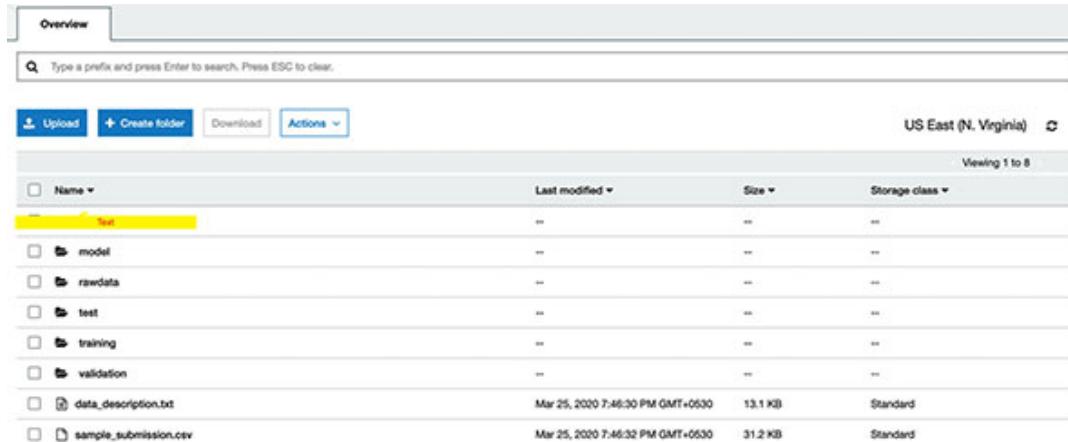


Figure 7.40: Amazon S3 Folder Structure

The structure will look like the preceding screenshot in your S3 bucket. The following function will be used to push the data to S3 by converting to binary:

```

def write_to_s3(filename, bucket, key):
    with open(filename, 'rb') as f: # Read in binary mode
        return
    boto3.Session().resource('s3').Bucket(bucket).Object(key).upload
    _fileobj(f)

```

We will split our train data into 80-20 percentage into the train set and validation set randomly.

```

df=x.copy()
np.random.seed(5)
l = list(df.index)
np.random.shuffle(l)
df = df.loc[l]
rows = df.shape[0]
train = int(.20 * rows)
test = rows-train

```

Saving the train/validation/test dataset local path:

```

# Write Training Set
df.iloc[:train].to_csv('house_train_final.csv',index=False,header=False)

# Write Validation Set
df.iloc[train: ].to_csv('house_validation_final.csv',index=False,header=False)

# Write test Set
test_csv.to_csv('house_test_final.csv')

```

Here, we will be pushing out the train/validation/test dataset into S3 by calling the **write_to_s3** function:

```

write_to_s3('house_train_final.csv', bucket_name, training_folder +
'train.csv')
write_to_s3('house_validation_final.csv',bucket_name,
validation_folder + 'validation.csv')

write_to_s3('house_test_final.csv',bucket_name, test_folder +
'test_tran.csv')

```

So, we have pushed our data to S3.

7.6.2 Train with SageMaker API XG-Boost which maintains the algorithm container

Now, we will establish a session with AWS by calling the session from SageMaker SDK.

```
sess = sagemaker.Session()
```

Here, we will import the Amazon SageMaker Python SDK and get the XGBoost container default version. Now, SageMaker API maintains the algorithm container mapping for us specifying the region, algorithm, and version.

```
container = sagemaker.amazon.amazon_estimator.get_image_uri(  
    sess.boto_region_name,  
    "xgboost")  
print('Using SageMaker XGBoost container:\n{} ({} {})'.format(container,  
    sess.boto_region_name))
```

```
Using SageMaker XGBoost container:  
683313688378.dkr.ecr.us-east-1.amazonaws.com/sagemaker-xgboost:0.90-2-cpu-py3 (us-east-1)
```

Figure 7.41: XG-Boost Container Location in AWS Cloud and Version

Next, we will configure the training job, and then we will specify the type and number of instances to use; then we will specify the S3 location where the final artifacts need to be stored. We will create an instance of the **sagemaker.estimator.Estimator** class.

```
# Reference: http://sagemaker.readthedocs.io/en/latest/estimators.html  
estimator = sagemaker.estimator.Estimator(  
    container,  
    role,  
    train_instance_count=1,  
    train_instance_type='ml.m4.xlarge',  
    output_path=s3_model_output_location,  
    sagemaker_session=sess,  
    base_job_name ='xgboost-Model')
```

In the constructor, you can specify the following parameters:

- **role:** The AWS Identity and Access Management (IAM) role that Amazon SageMaker can assume to perform tasks on our behalf (for example: it can

read the training results, it can call the model artifacts from our S3 bucket, and it can write the training results to Amazon S3). This is the role that we got in during the creation of Notebook instance.

- **train_instance_count** and **train_instance_type**: So, think of the type and number of ML compute instances that we use for the model training; here, we have to use only a single training instance.
- **train_volume_size**: This is the size, in GB, of the **Amazon Elastic Block Store (Amazon EBS)** storage volume which will attach to the training instance. It must be large enough to store our training data if you use the file mode, which is default.
- **output_path**: This is the path to the S3 bucket where the Amazon SageMaker stores our training results as a default extension file.
- **sagemaker_session**: This is the session object that will manage the interactions with SageMaker APIs and for any other AWS service that our training job will use.

So, we will set the hyperparameter values for our XGBoost training job by calling the **SDK** method **set_hyperparametersmethod** of the estimator class.

```
estimator.set_hyperparameters(base_score=0.5,
colsample_bylevel=1, num_round=150,
    colsample_bynode=1, colsample_bytree=0.5,
    learning_rate=0.02, gamma=0.025,
    max_depth=4, n_estimators=1500, min_child_weight=2,
    nthread=1, reg_alpha=0., reg_lambda=1, subsample=0.5,
    objective='reg:linear', random_state=28)
```

Here, we have to create the training channels which will be used for the training job. We use both the train and the validation channels.

```
# content type can be libsvm or csv for XGBoost
training_input_config = sagemaker.session.s3_input(
    s3_data=s3_training_file_location,
    content_type='csv',
    s3_data_type='S3Prefix')
validation_input_config = sagemaker.session.s3_input(
    s3_data=s3_validation_file_location,
    content_type='csv',
    s3_data_type='S3Prefix'
```

```
)
data_channels = {'train': training_input_config, 'validation':
validation_input_config}
```

Now, to start the model training, we will call the estimator's fit method. XGBoost supports "train", "validation" channels.

```
#Reference: Supported channels by algorithm
# https://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-algo-
docker-registry-paths.html
estimator.fit(data_channels)
```

```
2020-04-05 16:42:08 Starting - Starting the training job...
2020-04-05 16:42:09 Starting - Launching requested ML instances.....
2020-04-05 16:43:12 Starting - Preparing the instances for training...
2020-04-05 16:44:06 Downloading - Downloading input data...
2020-04-05 16:44:37 Training - Downloading the training image...
2020-04-05 16:45:09 Uploading - Uploading generated training model
2020-04-05 16:45:09 Completed - Training job completed

Arguments: train
[2020-04-05:16:44:58:INFO] Running standalone xgboost training.
[2020-04-05:16:44:58:INFO] File size need to be processed in the node: 0.74mb. Available memory size in the node: 8508.11mb
[2020-04-05:16:44:58:INFO] Determined delimiter of CSV input is ','
[16:44:58] S3DistributionType set as FullyReplicated
[16:44:58] 1235x225 matrix with 277875 entries loaded from /opt/ml/input/data/train?format=csv&label_column=0&delimiter=,
[2020-04-05:16:44:58:INFO] Determined delimiter of CSV input is ','
[16:44:58] S3DistributionType set as FullyReplicated
[16:44:58] 218x225 matrix with 49950 entries loaded from /opt/ml/input/data/validation?format=csv&label_column=0&delimiter=,
[16:44:58] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 18 extra nodes, 0 pruned nodes, max_depth=4
[0]#011train-rmse:5.41071#011validation-rmse:5.58529
[16:44:58] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 22 extra nodes, 0 pruned nodes, max_depth=4
[1]#011train-rmse:4.03268#011validation-rmse:4.23648
[16:44:58] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 22 extra nodes, 0 pruned nodes, max_depth=4
[2]#011train-rmse:2.98134#011validation-rmse:3.15959
[16:44:58] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 18 extra nodes, 2 pruned nodes, max_depth=4
[3]#011train-rmse:2.26828#011validation-rmse:2.4225
[16:44:58] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 22 extra nodes, 0 pruned nodes, max_depth=4
[4]#011train-rmse:1.67801#011validation-rmse:1.83374
[16:44:58] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 20 extra nodes, 0 pruned nodes, max_depth=4
[5]#011train-rmse:1.33205#011validation-rmse:1.48437
[16:44:58] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 12 extra nodes, 0 pruned nodes, max_depth=4
[6]#011train-rmse:1.10017#011validation-rmse:1.23461
[16:44:58] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 24 extra nodes, 0 pruned nodes, max_depth=4
[16:44:59] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 22 extra nodes, 6 pruned nodes, max_depth=4
[148]#011train-rmse:0.095235#011validation-rmse:0.693852
[16:44:59] src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 18 extra nodes, 4 pruned nodes, max_depth=4
[149]#011train-rmse:0.094575#011validation-rmse:0.694274

Training seconds: 63
Billable seconds: 63
```

Figure 7.42: XG-Boost Model Fit Logs of RMSE of 0.6942 which is minimum

It is a synchronous operation. The method will display the progress logs and it waits until the training completes before returning.

Go to the training Jobs and Model in Amazon SageMaker service and you can check all the details of the model hyperparameter and the output artifacts location.

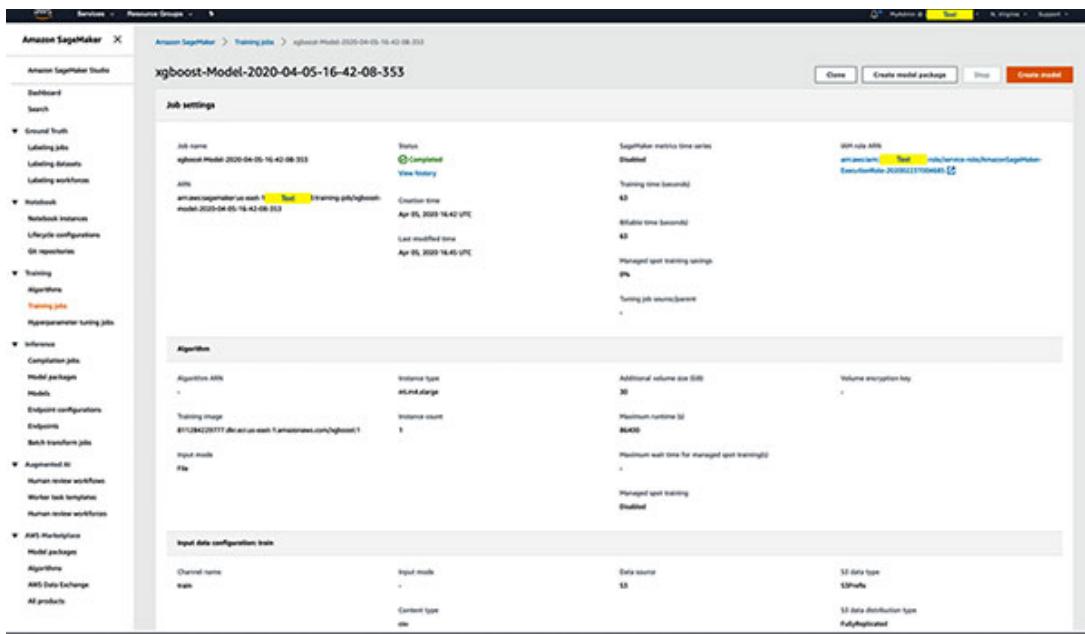


Figure 7.43: XG-Boost Model Training Job Dashboard

Now, the preceding screenshot is of the dashboard of the model which we have fitted; we will get all the output location and instance type and required information in the dashboard.

7.7 Amazon SageMaker model deployment

Now, we will deploy from the training job which was fitted earlier and give an endpoint name and instance type.

```
# Ref: http://sagemaker.readthedocs.io/en/latest/estimators.html
predictor = estimator.deploy(initial_instance_count=1,
    instance_type='ml.m4.xlarge', endpoint_name = 'xgboost-
    house-v1')
```

Here, we will deploy the model that we have trained in create and run a Training Job by calling the `deploy` method of the `sagemaker.estimator.Estimator` object from SDK. So, now it is the same object which we have used to train our model. When we call the `deploy` method, it will specify the number and type of the ML instances that we want to use to host the endpoint for deployment.

Note: If an error comes, please use `instance_type='ml.t2.medium'`.

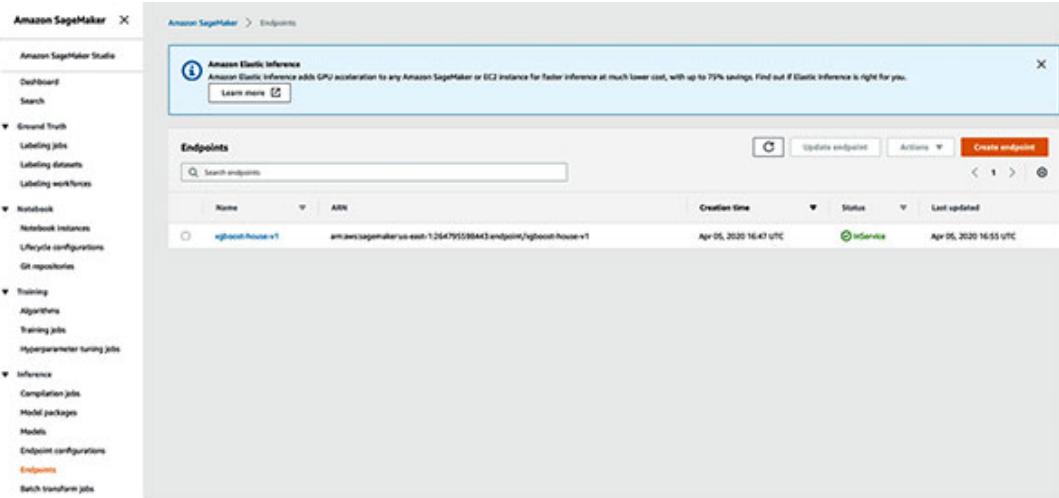


Figure 7.44: XG-Boost Model Deployment Endpoint

In the preceding screenshot, we can see the endpoint deployment name and below that, we can call the endpoint, with **RealTimePredictor** API by passing the name.

```
endpoint_name = 'xgboost-house-v1'
predictor =
sagemaker.predictor.RealTimePredictor(endpoint=endpoint_name)
```

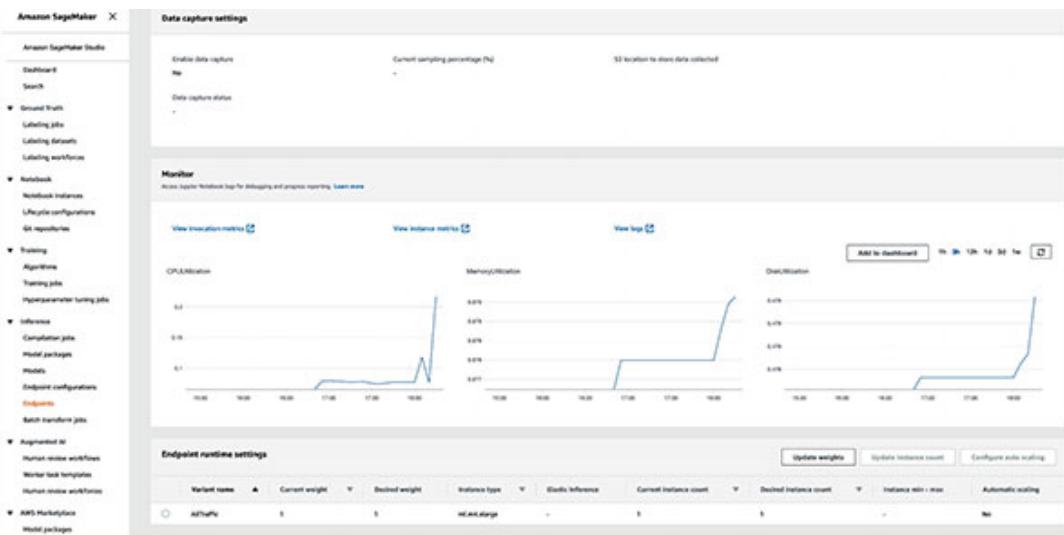


Figure 7.45: XG-Boost Model End-Point Cloud-Watch Metrics

Now, the preceding screenshot shows the model CPU Utilization and its performance with respect to time and we can see those in CloudWatch and Cloud logs to track all the model performance.

```

class sagemaker.predictor.RealTimePredictor(endpoint, sagemaker_session=None, serializer=None,
deserializer=None, content_type=None, accept=None)

Bases: object

Make prediction requests to an Amazon SageMaker endpoint.

Initialize a RealTimePredictor.

Behavior for serialization of input data and deserialization of result data can be configured through initializer arguments. If not specified, a sequence of bytes is expected and the API sends it in the request body without modifications. In response, the API returns the sequence of bytes from the prediction result without any modifications.

Parameters:
    • endpoint (str) – Name of the Amazon SageMaker endpoint to which requests are sent.
    • sagemaker_session (sagemaker.session.Session) – A SageMaker Session object, used for SageMaker interactions (default: None). If not specified, one is created using the default AWS configuration chain.
    • serializer (callable) – Accepts a single argument, the input data, and returns a sequence of bytes. It may provide a content_type attribute that defines the endpoint request content type. If not specified, a sequence of bytes is expected for the data.
    • deserializer (callable) – Accepts two arguments, the result data and the response content type, and returns a sequence of bytes. It may provide a content_type attribute that defines the endpoint response's "Accept" content type. If not specified, a sequence of bytes is expected for the data.
    • content_type (str) – The invocation's "ContentType", overriding any content_type from the serializer (default: None).
    • accept (str) – The invocation's "Accept", overriding any accept from the deserializer (default: None).

predict(data, initial_args=None, target_model=None)

```

Figure 7.46: SageMaker API Notes (Credit: AWS SageMaker SDK)

Now we will set the environment variables.

```

from sagemaker.predictor import csv_serializer, json_deserializer
predictor.content_type = 'text/csv'
predictor.serializer = csv_serializer
predictor.deserializer = None

```

We need to pass an array; so the prediction can pass a numpy array or a list of values [[19,1],[20,1]]. Here, we will load the test data and transform the `add_features` function, and then we will predict.

```

t=pd.read_csv("house_test_final.csv")
t=add_features(t)
t.shape
arr_test = t[t.columns[1:]].values

```

For a large number of predictions, we can split the input data and query the prediction service. `array_split` is convenient to specify how many splits are

needed.

```
predictions = []
for arr in np.array_split(arr_test,50):
    result = predictor.predict(arr)
    result = result.decode("utf-8")
    result = result.split(',')
    print (arr.shape)
    predictions += [float(r) for r in result]
```

Now, we will reverse and transform our log values. We can evaluate our validation dataset as well.

```
pred=np.expm1(predictions)
sub=pd.DataFrame()
sub['id']=df_test_id.values
sub['SalePrice']=pred
sub.head()
```

| [130]: | id | SalePrice |
|--------|-----------|------------------|
| 0 | 1461 | 10.459867 |
| 1 | 1462 | 7.723757 |
| 2 | 1463 | 51.396668 |
| 3 | 1464 | 68.385400 |
| 4 | 1465 | 133.739232 |

Figure 7.47: Prediction results

So, the preceding table is the prediction output which we have predicted from the deployed endpoint.

The following is the deployed endpoint:

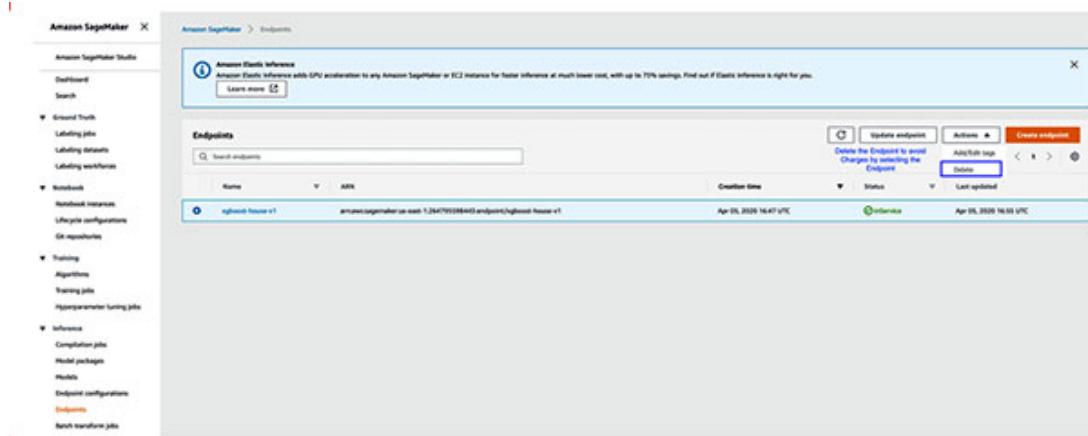


Figure 7.48: Model Endpoint Dashboard, Delete the endpoint

To avoid incurring unnecessary charges, use the AWS Management Console, as shown in the preceding screenshot, to delete the resources, or delete from the Notebook as follows:

```
sagemaker.Session().delete_endpoint(predictor.endpoint)
```

7.8 Conclusion

We have come toward the end of this chapter, and you can run and deploy your own model in cloud with comfort, and use the power of Cloud.

In this chapter, we learned about outlier analysis, feature transformation, and the imputation of categorical and numerical columns. Then, we learned how to create notebook in Amazon SageMaker and build a model and deploy it. We also learned how to use XG-Boost in-built algorithm. Then, we checked the metrics and performance of our deployed model in Amazon CloudWatch.

7.9 References

- <https://docs.aws.amazon.com/sagemaker/latest/dg/studio-jumpstart.html>
- <https://docs.aws.amazon.com/sagemaker/index.html>
- <https://github.com/awslabs/sagemaker-churn-prediction-text>

CHAPTER 8

Web App Development with Streamlit & Heroku

In this chapter, we will build an end-to-end web application for the computer vision models, and build that UI with Streamlit. We will be learning about the many Open CV models for Image like cropping, changing pixels, and so on. Next, we will host the Web application with the Heroku Container Registry or Kubernetes Cluster as a service application in Google Cloud.

Structure

In this chapter, we will cover the following topics:

- Problem statement
- Setup of project requirements in GCP & Heroku
- Introduction on components of Streamlit
- Building the Framework for Streamlit for OpenCV models
- Creating the components for Heroku Deployment
- Deploying the Streamlit code by containerizing in Kubernetes cluster

Objectives

After studying this chapter, you will be able to understand the following:

- How to use Docker and Kubernetes.
- How to build the web application in Python Streamlit without any JavaScript knowledge.
- The various Computer Vision OpenCV models.
- How to construct the framework for the Streamlit and host it to Heroku.
- How to use Kubernetes and many Google Cloud Platform to leverage the power of that to deploy and application to host.

8.1 Problem statement

In this chapter, we will be using OpenCV model's various image processing technique applications, for which we will build the framework for UI with Streamlit Library. So, we will be building the OpenCV models like pencil sketch, cropping image, sharpening image, and color changing image, and then we will add a comic reader. Then, we will host the application either in Heroku or in Kubernetes cluster in GCP.

| | |
|-------------|---|
| NOTE | Rest all the imports I have showed in my Colab Notebook, for which the hyperlink of the GitHub Account of this chapter is given. Note Colab platform Python 3.x. RUN IN GOOGLE COLAB |
| CODE | https://github.com/bpbpublications/Continuous-Machine-Learning-with-Kubeflow/tree/main/Chapter8 |

8.2 Setup of project requirements in GCP & Heroku

Google Cloud Platform: Create an account with your email id and you will be redirecting to the home page of the cloud account.

You must have an active GCP account, and while you practice this chapter, it might charge for running the Kubernetes cluster, as I am running all the codes on MacOS. I recommend some basic Kubernetes and Docker knowledge is a must.

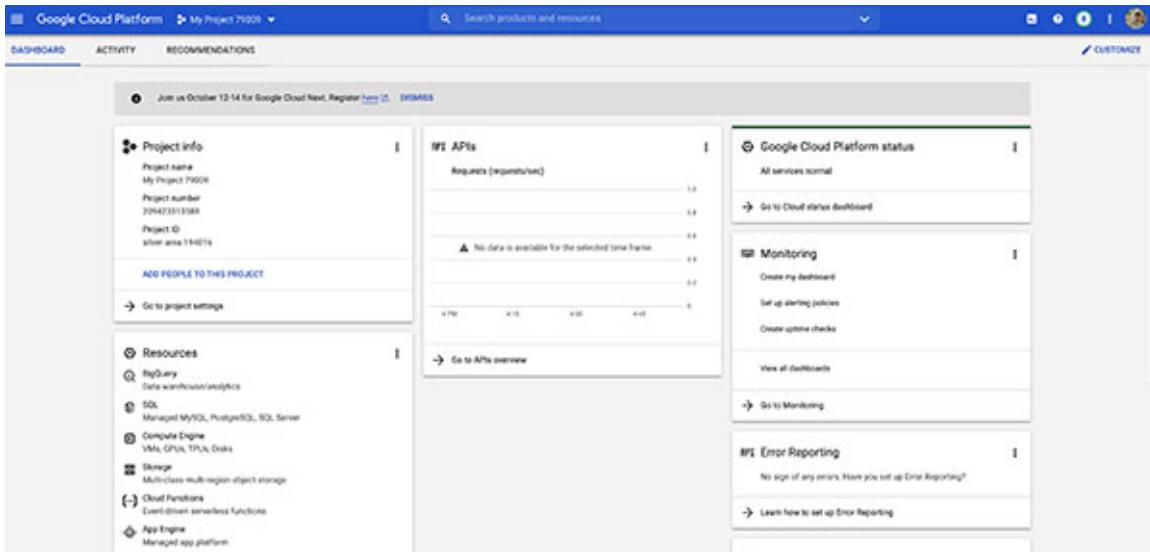


Figure 8.1: Google Cloud Platform

Heroku: Create an account with your email id.

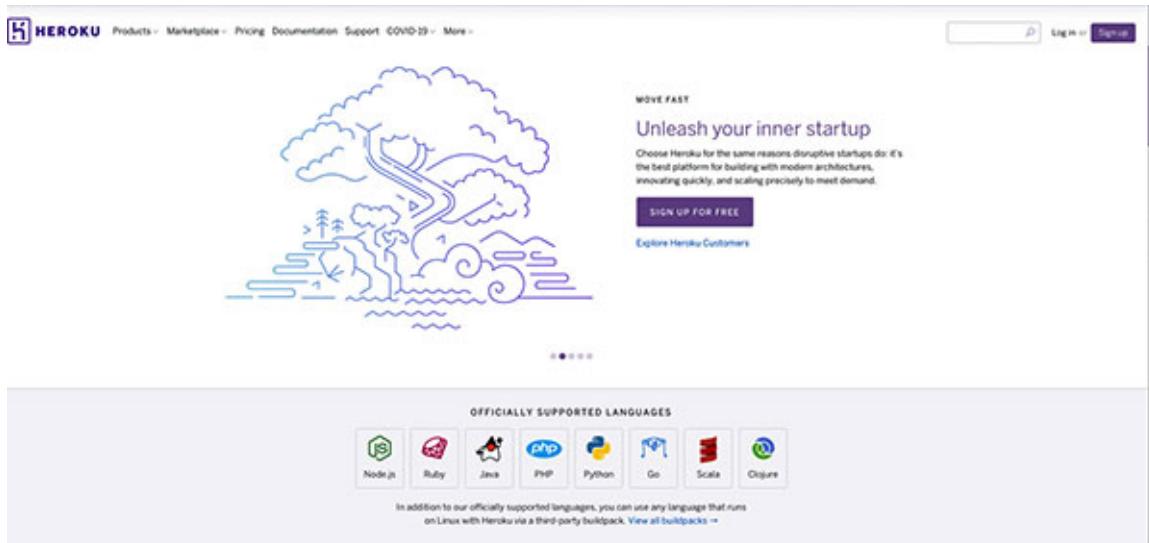


Figure 8.2: Heroku Platform

Let's build the Streamlit component in the following section.

8.3 Introduction on components of Streamlit

Streamlit is an open-source app framework and is the easiest way for the data scientists and machine learning engineers to create beautiful, performant apps in a few hours. The goal of Streamlit is to create an interactive app for our data or model and, along the way, to use Streamlit to check, debug, perfect, and share our code.

Set up your virtual environment: `pip install streamlit==0.71.0`

Features:

- We can build an app with a few lines of code and leverage the Streamlit simple API, which we will automatically update as we can save the source Python file.

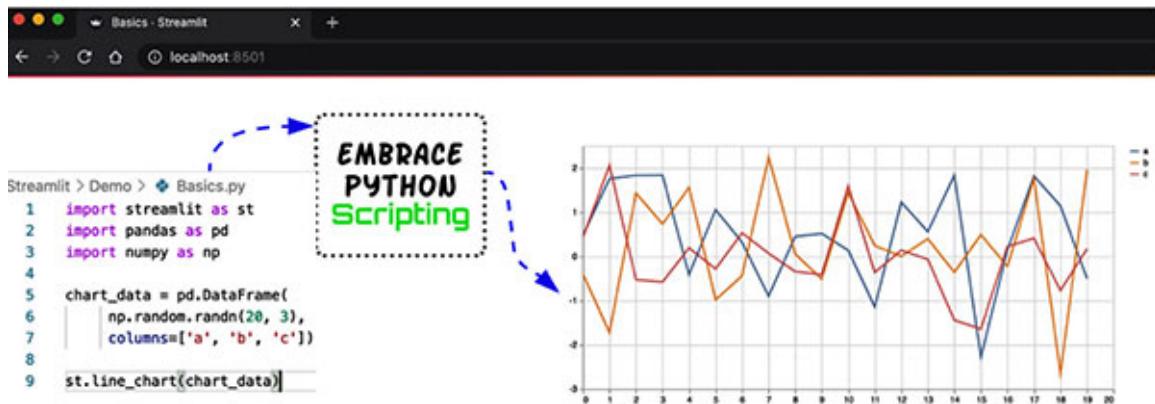


Figure 8.3: Streamlit Features 1

- We can add the various widgets for the variable declaration in Python like checkbox, slider, text box, and so on. No need for the hard coding of the variables from the backend and define routes, which will handle the HTTP requests and so on.

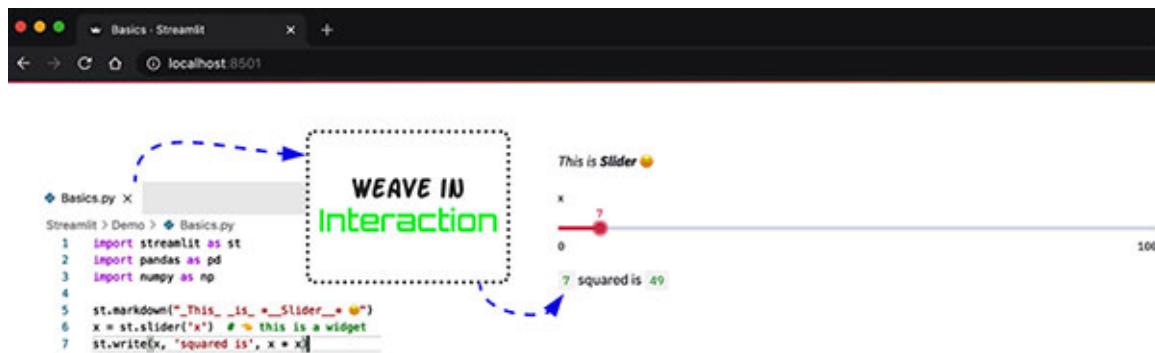


Figure 8.4: Streamlit Features 2

- Instantly host your application on prem or cloud without much effort and maintenance.



Figure 8.5: Streamlit Features 3

So, these are the most import features for Streamlit.

8.3.1 Main concepts

First, let's write a few Streamlit commands into a Python script, then we will run it with the Streamlit run. A new tab will open in your default browser. It'll be blank for now. That's OK.

```
```bash
streamlit run your_script.py
```
(base) WKMN9818087:Demo anichoud2$ streamlit run Basics.py

You can now view your Streamlit app in your browser.

Local URL: http://localhost:8501
Network URL: http://192.168.29.199:8501
```

Figure 8.6: Streamlit Host URL on-prem

Working with Text:

```
streamlit.write(*args, **kwargs)
```

“Swiss Army knife”. You can pass almost anything to **st.write()**: text, data, Matplotlib figures, Altair charts, and more. Use specific text functions to add content to your app.

```
streamlit.title(body)
```

It displays the text in the title format.

```
streamlit.subheader(body)
```

This API displays the text in the sub-header format.

```
streamlit.code(body, language='python')
```

It displays a code block with the optional syntax highlighting.

```
streamlit.latex(body)
```

This one displays the mathematical expressions formatted as LaTeX.

```

import streamlit as st
import pandas as pd
import numpy as np

st.title('My first Web Application')
st.subheader("This is Subheader for Maths Equation")

st.markdown('Streamlit is **_really_ cool**.')
st.latex(r'''f(x)=a_0 +\sum_{n=1}^{\infty}(a_n + b_n)\left(\frac{1-r^n}{1-r}\right) \cos\left(\frac{n\pi x}{L}\right) \sin\left(\frac{n\pi x}{L}\right)''')

st.write('Hello, *World!* :sunglasses:')
code = '''def python():
    print("Hello, Streamlit!")'''
st.code(code)

```

Now, open Bash or Terminal, wherever the code is, then run **streamlit run Basics.py**:

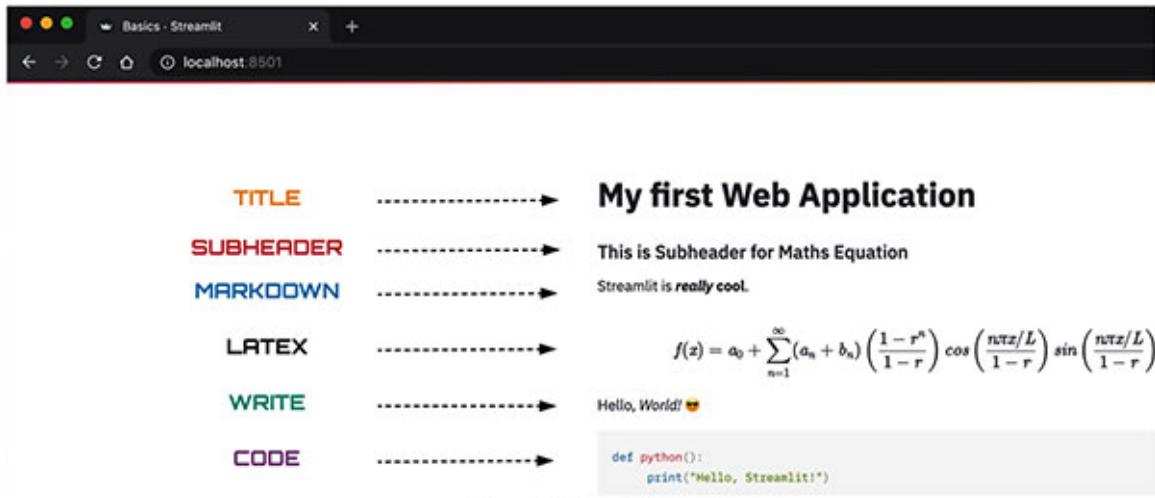


Figure 8.7: Streamlit Text Components

So, the preceding screenshot shows the output for the Text components for the Streamlit tool.

Working with Media:

```
streamlit.audio(data, format='audio/wav', start_time=0)
```

This API displays an audio player.

```
streamlit.image(image, caption=None, width=None,  
use_column_width=False, clamp=False, channels='RGB',  
output_format='auto', **kwargs)
```

It's easy to embed images, videos, and audio files directly into your Streamlit apps.

```
streamlit.video(data, format='video/mp4', start_time=0)
```

Here, this API displays a video player.

Now, open Bash or Terminal, wherever the code is, then run **Streamlit run Basics.py**.

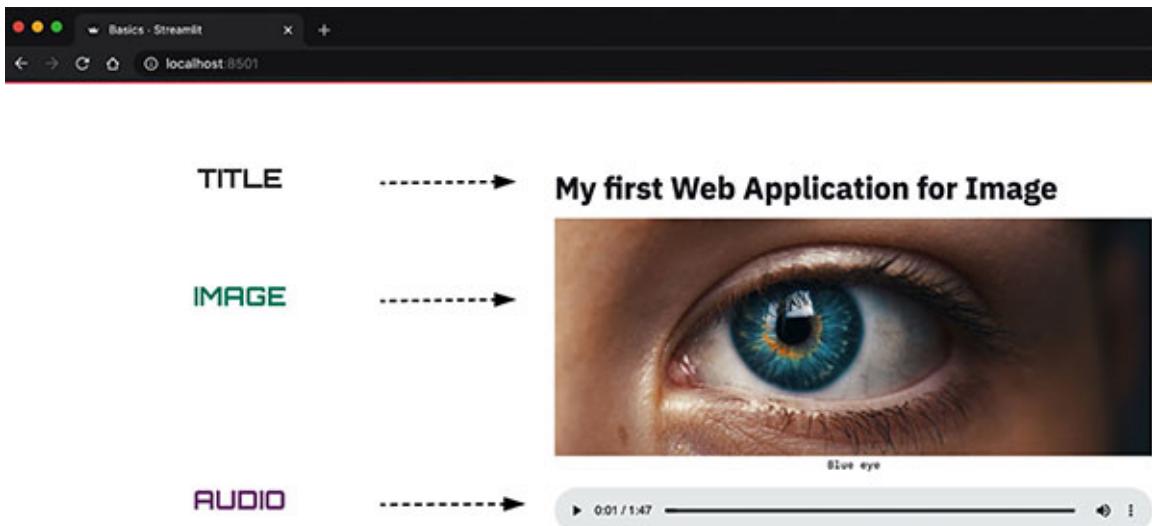


Figure 8.8: Streamlit Media Components

So, the preceding is the output for the media components for Streamlit.

Working with Media :

```
streamlit.button(label, key=None)
```

It is used to display a button widget.

```
streamlit.checkbox(label, value=False, key=None)
```

This API displays a checkbox widget.

```
streamlit.selectbox(label, options, index=0, format_func=<class  
'str'>, key=None)
```

It displays a select widget.

```
streamlit.radio(label, options, index=0, format_func=<class  
'str'>, key=None)
```

It is used to display a radio button widget.

```
streamlit.multiselect(label, options, default=None, format_func=  
<class 'str'>, key=None)
```

This API displays a multiselect widget. The multiselect widget starts as empty.

```
streamlit.slider(label, min_value=None, max_value=None,  
value=None, step=None, format=None)
```

It displays a slider widget. This supports int, float, date, time, and datetime types.

```
streamlit.text_input(label, value='', max_chars=None, key=None,  
type='default')
```

This API displays a single-line text input widget.

```
streamlit.number_input(label, min_value=None, max_value=None,  
value=<streamlit.elements.utils.NoValue object>, step=None,  
format=None, key=None)
```

Display a numeric input widget.

Let's build the preceding API in our Python script:

```
import streamlit as st  
import pandas as pd  
import numpy as np  
from PIL import Image  
import datetime  
  
st.title('My first Web Application for Widget')  
# Button  
if st.button('HIT ME'):
```

```
    st.write('Hello welcome to my world :sunglasses:')
else:
    st.write("It's time for GoodBye")
# Checkbox
condition = st.checkbox('Do you want?')
if condition:
    st.write('Welcome to my checkbox world!')

# RadioButton
sport = st.radio(
    "What's your favorite sport?",
    ('Cricket', 'Football', 'Hockey'))
if sport == 'Cricket':
    st.write('You selected Cricket.')
elif sport == 'Football':
    st.write('You selected Football.')
elif sport == 'Hockey':
    st.write('You selected Hockey.')
else:
    st.write("You didn't select anything.")

# SelectBox
option = st.selectbox(
    'Which movie type you want to see?',
    ('Romantic', 'Action', 'Horror'))
st.write('You selected:', option)

# Multiselect
options = st.multiselect(
    'What are your favorite colors',
    ['Green', 'Yellow', 'Red', 'Blue'],
    ['Yellow', 'Red'])
st.write('You selected:', options)

# Slider
age = st.slider('How old are you?', 0, 130, 25)
st.write("I'm ", age, 'years old')
```

```

# Text Input
title = st.text_input('Movie title', 'Dark night Rises')
st.write('The current movie title is', title)

# Number Input
number = st.number_input('Insert a number')
st.write('The current number is ', number)

#File Upload
uploaded_file = st.file_uploader("Choose a CSV file", type="csv")
st.set_option('deprecation.showfileUploaderEncoding', False)
if uploaded_file:
    text_io = io.TextIOWrapper(uploaded_file)
    if text_io is not None:
        data = pd.read_csv(text_io)
        st.write(data)

# Input Date
d = st.date_input(
    "When's your birthday",
    datetime.date(2019, 7, 6))
st.write('Your birthday is:', d)

```

Now, open Bash or Terminal, wherever the code is, then there run **Streamlit run Basics.py**. And open the link, **http://localhost:8051**, in your browser.

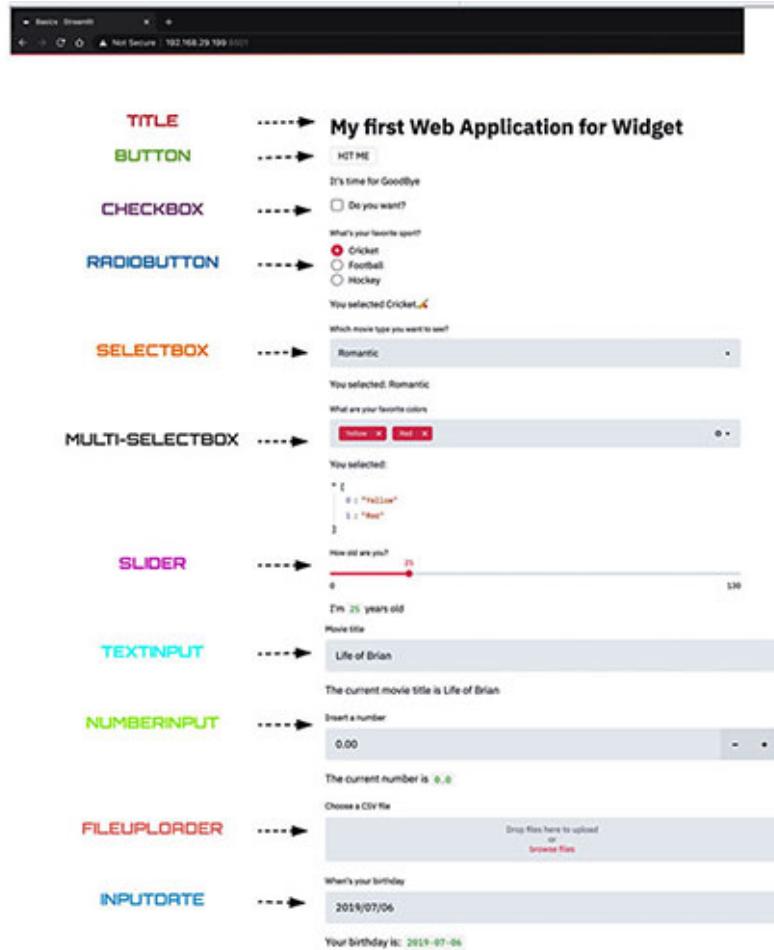


Figure 8.9: Streamlit Media Components

So, the preceding is the output for the media components for Streamlit.

Working with Data & Chart:

```
streamlit.dataframe(data=None, width=None, height=None)
```

It displays a dataframe as an interactive table.

```
streamlit.line_chart(data=None, width=0, height=0,
use_container_width=True)
```

It displays a line chart. Similarly, we can plot Bar plot, and Area Chart.

```
streamlit.plotly_chart(figure_or_data, width=0, height=0,
use_container_width=False, sharing='streamlit', **kwargs)
```

It displays an interactive Plotly chart. Plotly is a charting library for Python.

```

import streamlit as st
import pandas as pd
import numpy as np

st.title('My first Web Application for Data & Chart')
# Data Table
df = pd.DataFrame(np.random.randn(4, 2),columns=(('col %d' % i for
i in range(2))))
st.dataframe(df) # Same as st.write(df)

# Line Chart
chart_data = pd.DataFrame(np.random.randn(20, 3),columns=['a',
'b', 'c'])
st.line_chart(chart_data)

# plotly
import plotly.express as px
df = px.data.gapminder().query("year == 2007")
fig = px.sunburst(df, path=['continent', 'country'], values='pop',
color='lifeExp', hover_data=
['iso_alpha'],color_continuous_scale='RdBu',
color_continuous_midpoint=np.average(df['lifeExp'],
weights=df['pop']))
st.plotly_chart(fig, use_container_width=True)

```

Now, open Bash or Terminal, wherever the code is, then run **Streamlit run Basics.py**. Then, open the following link in your browser:
http://localhost:8051

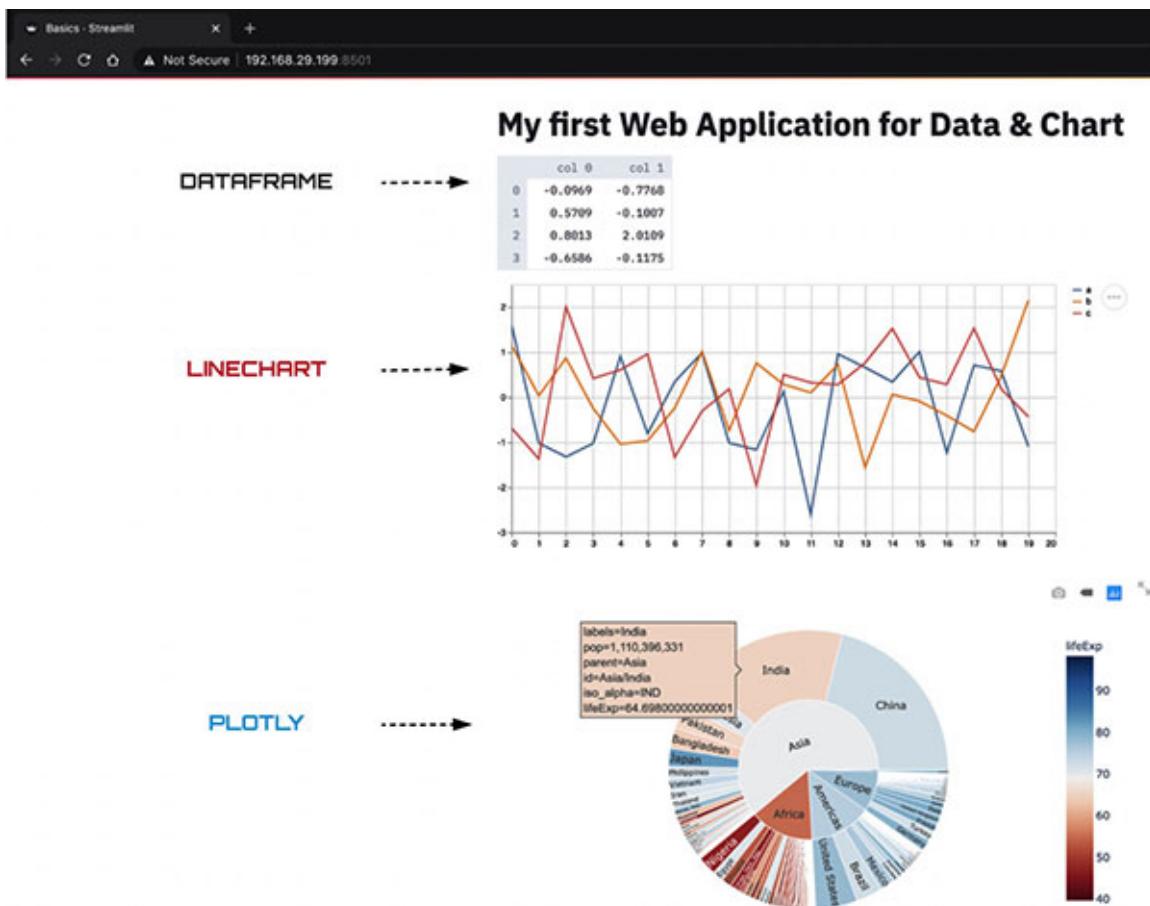


Figure 8.10: Streamlit Data & Charts Components

For further API study, please visit the following link:
<https://docs.streamlit.io/en/stable/api.html>.

8.4 Building the Framework for Streamlit for OpenCV models

To build a framework for Streamlit for OpenCV models, complete the following steps:

STEP 1: Load all the Dependency for the Computer Vision Model and Streamlit.

```
import streamlit as st
import cv2
import numpy as np
from PIL import Image
```

```
from matplotlib import pyplot as plt
import io
import cv2 as cv
import os
import glob
import requests
from bs4 import BeautifulSoup
import urllib.request
import random
import pandas as pd
from io import BytesIO
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
```

STEP 2: Now we will set some Logo and Title for our application. Then, we can select the OpenCV model from the dropdown. Then, `st.sidebar.selectbox()` will add the widget to the right side of the UI.

```
st.markdown("![Alt Text]  
(https://raw.githubusercontent.com/aniruddhachoudhury/AR-RL-/master/COMPUTER%20VISION.gif)")  
  
st.title("Computer Vision Use Case")  
st.sidebar.subheader("Choose Computer Vision Model")  
model = st.sidebar.selectbox("Model", ("Pencil Sketch", "Crop Image", "Sharp Image", "Color Spacer", "Comic Reader"))
```

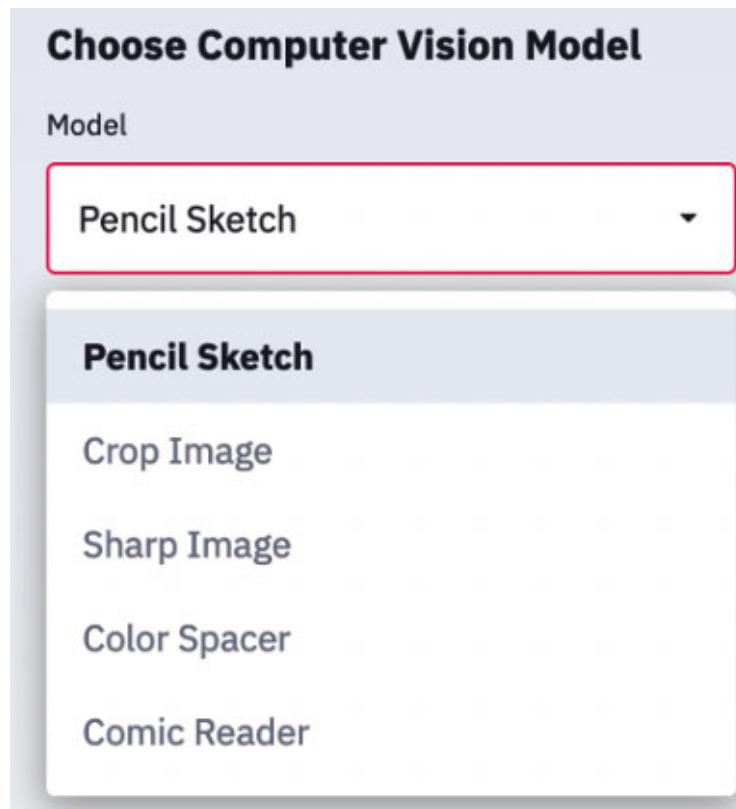


Figure 8.11: Streamlit dropdown of models

STEP 3: In this step, we will write one heading with the `st.write()` function, and we will set some depreciation option. Next, we will upload the image in any format of jpg, png, and so on, so the `st.file_uploader()` object will be stored inside `file_image`.

```
st.write("This Web App is to help convert your photos to  
realistic images")  
st.set_option('deprecation.showfileUploaderEncoding', False)  
file_image = st.sidebar.file_uploader("Upload your Photos",  
type=['jpeg', 'jpg', 'png'])  
st.set_option('deprecation.showfileUploaderEncoding', False)
```

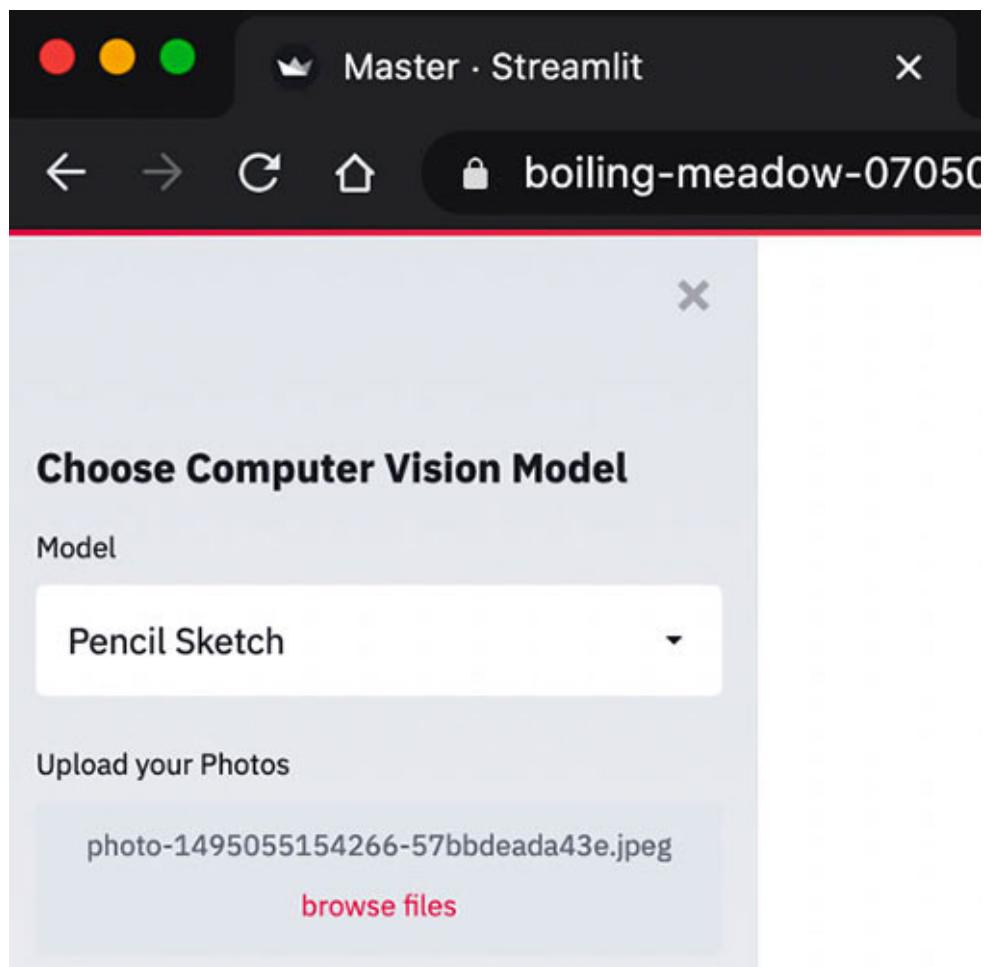


Figure 8.12: Streamlit st.file_uploader()

STEP 4: Now, let's build the first model for OpenCV Pencil Sketch. If we gave the condition that if you want to select the model from Step 2 here, it will be the pencil sketch, and the `dodgeV2()` and `pencilsketch()` function will do the necessary pre-processing like transforming and scaling the image, and returning the output. The `st.image()` function will show the input and output image in the browser. The `st.write()` function shows the text you want in your browser. The `st.file_uploader()` function will return a BytesIO object. However, `Image.open()` accepts a string to read an image.

```
def dodgeV2(x, y):  
    return cv2.divide(x, 255 - y, scale=256)  
def pencilsketch(inp_img):  
    img_gray = cv2.cvtColor(inp_img, cv2.COLOR_BGR2GRAY)  
    img_invert = cv2.bitwise_not(img_gray)
```

```
    img_smoothing = cv2.GaussianBlur(img_invert, (21, 21), sigmaX=0, sigmaY=0)
    final_img = dodgeV2(img_gray, img_smoothing)
    return(final_img)

if model == "Pencil Sketch":
    st.subheader("PencilSketcher app to Cartoon Image")
if file_image is None:
    st.write("You haven't uploaded any image file")
else:
    input_img = Image.open(file_image)
    final_sketch = pencilsketch(np.array(input_img))
    st.write("'''Input Photo'''")
    st.image(input_img, use_column_width=True)
    st.write("'''Output Pencil Sketch'''")
    st.image(final_sketch, use_column_width=True)
```

Now, open Bash or Terminal, wherever the code is, and then run **streamlit run Master.py**. Then, open the following link in your browser: **http://localhost:8051**.

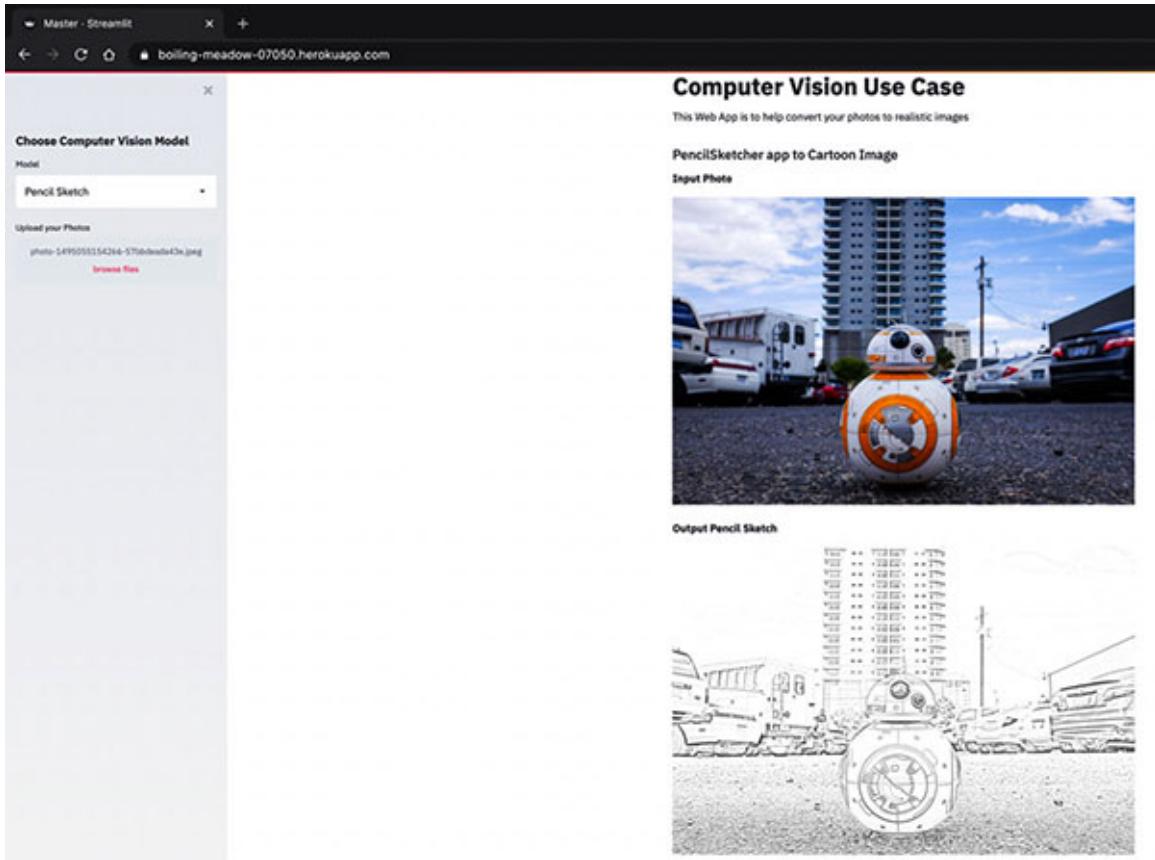


Figure 8.13: OpenCV Pencil Sketch Output

STEP 5: Now, let's build the second model for the OpenCV cropping and changing pixel. If we gave the condition that if you want to select the model from Step 2 here, then it will be the Cropping Image. The `st.file_uploader()` function will return a BytesIO object. However, `Image.open()` accepts a string to read an image. Next, we will use the `st.sidebar.slider()` function for selecting the pixel coordinates from the slider for all the coordinates. We have added one if condition – either you want to crop or not. After that, we will do the necessary OpenCV image cropping operations and pixel change. The `st.image()` will dump the image and show the input and output image in the browser.

```
if model == "Crop Image":
    st.subheader("Crop your Image app to your size")
    if file_image is None:
        st.write("You haven't uploaded any image file")
    else:
```

```

input_img = Image.open(file_image)
image = np.array(input_img)
x, y = image.shape[:2]
height, width = image.shape[:2]
print(height,width)
st.sidebar.subheader("Choose Pixel for image")
# Let's get the starting pixel coordiantes (top left of
cropping rectangle)
startrowper=st.sidebar.slider("Start Row", min_value=0.,
max_value=1.0)
startcolper=st.sidebar.slider("Start Column",
min_value=0., max_value=1.0)
endrowper=st.sidebar.slider("End Row", min_value=0.,
max_value=1.0)
endcolper=st.sidebar.slider("End Column", min_value=0.,
max_value=1.0)
Crop = st.sidebar.selectbox("You want to Crop", ("Yes",
"No"))
if Crop=='Yes':
    start_row, start_col = int(height * startrowper),
    int(width * startcolper)
    # Let's get the ending pixel coordinates (bottom right)
    end_row, end_col = int(height * endrowper), int(width *
    endcolper)

# Simply use indexing to crop out the rectangle we desire
cropped = image[start_row:end_row, start_col:end_col]
row, col = 1, 2
fig, axs = plt.subplots(row, col, figsize=(15, 10))
fig.tight_layout()
axs[0].imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
axs[0].set_title('Original Image')
cv2.imwrite('original_image.png', image)
st.image('original_image.png', use_column_width=True)
axs[1].imshow(cv2.cvtColor(cropped, cv2.COLOR_BGR2RGB))
axs[1].set_title('Cropped Image')

```

```

cv2.imwrite('cropped_image.png', cropped)
st.image('cropped_image.png', use_column_width=True)
else:
    st.write("You don't want to crop")

```

Now, open Bash or Terminal, wherever the code is, and then run **Streamlit run Master.py**. Then, open the following link in your browser: **http://localhost:8051**.

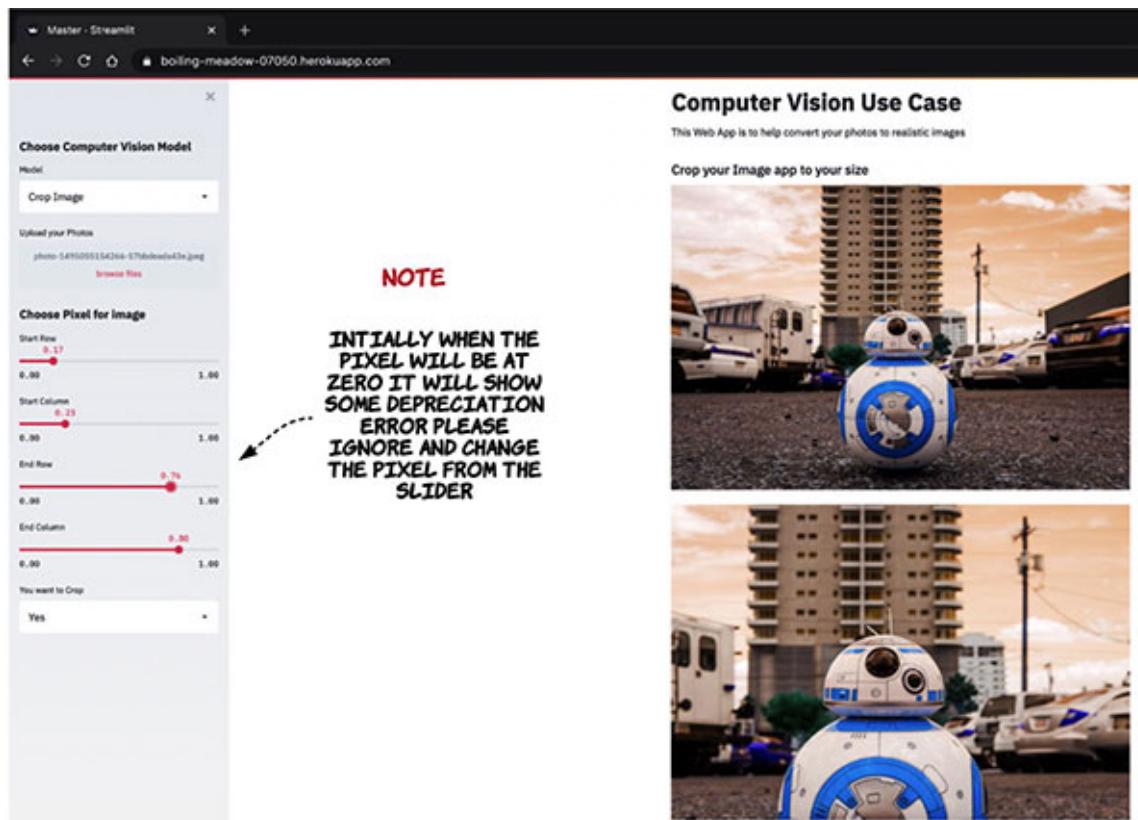


Figure 8.14: OpenCV Pixel change Output

So, the preceding is the output for the OpenCV pixel changer model in the Streamlit app.

STEP 6: Now let's build the third model for OpenCV sharpening our image. If we gave the condition that if you want to select the model from Step 2 here, it will be the Cropping Image. The **st.file_uploader()** function will return a BytesIO object. However, **Image.open()** accepts a string to read an image.

Next, we will use the `st.sidebar.slider()` function for selecting the pixel coordinates from the slider for all the coordinates. After that, we will do the necessary OpenCV image sharpening with the `cv2.filter2d` kernel and normalization application. `st.image()` will dump the image and show the input and output image in the browser.

```
if model == "Sharp Image":  
    st.subheader("Sharpen your Image")  
    if st.sidebar.button('Changer'):  
        showpred = 1  
        if file_image is None:  
            st.write("You haven't uploaded any image file")  
        else:  
            input_img = Image.open(file_image)  
            image = np.array(input_img)  
            row, col = 1, 2  
            fig, axs = plt.subplots(row, col, figsize=(15, 10))  
            #fig.tight_layout()  
            axs[0].imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))  
            axs[0].set_title('Original Image')  
            # Create our sharpening kernel, we don't normalize since  
            # the values in the matrix sum to 1  
            kernel_sharpening = np.array([[-1, -1, -1], [-1, 9, -1],  
                [-1, -1, -1]])  
            # applying different kernels to the input image  
            sharpened = cv2.filter2D(image, -1, kernel_sharpening)  
            axs[1].imshow(cv2.cvtColor(shARPened,  
                cv2.COLOR_BGR2RGB))  
            axs[1].set_title('Image Sharpening')  
            st.image(input_img, use_column_width=True)  
            cv2.imwrite('shARPEN_image.jpg', sharpened)  
            st.image('shARPEN_image.jpg', use_column_width=True)
```

Now, open Bash or Terminal, wherever the code is, and then run `streamlit run Master.py`. Then, open the following link in your browser:

<http://localhost:8051>.

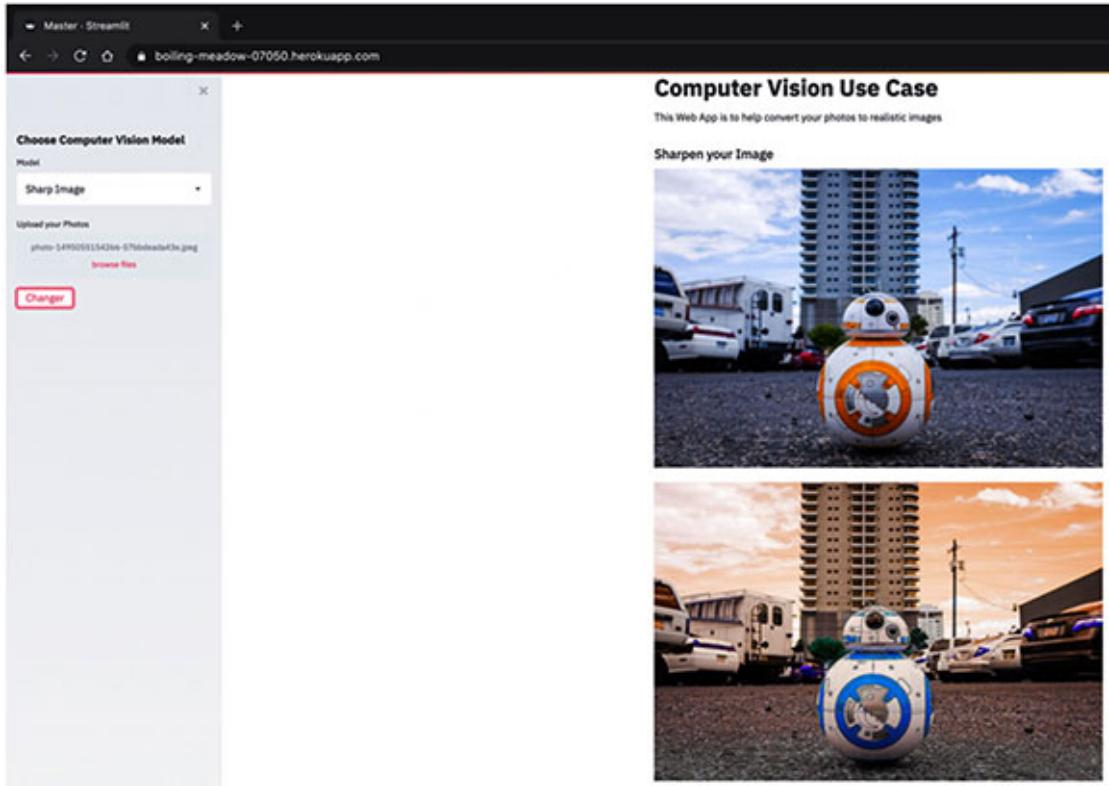


Figure 8.15: OpenCV Sharpening Output

STEP 7: Now, let's build the fourth model for the OpenCV colour spacer change. If we gave the condition that if you want to select the model from Step 2 here, it will be the Cropping Image. The `st.file_uploader()` function will return a BytesIO object. However, `Image.open()` accepts a string to read an image. Next, we have added the `st.sidebar.selectbox()` function on the sidebar for UI; here we have different OpenCV colour changing options like "bw", "hsv", and so on. Then, if the condition matches the option which we chose, it will do the necessary operations, and at last `st.image()` will show the image inside the browser.

```
if model == "Color Spacer":  
    st.subheader("Sharpen your Image")  
    cs = ["bw", "hsv", "yuv", "lab"]  
    color_space = st.sidebar.selectbox("Pick a space.", cs)  
    if st.sidebar.button('Changer'):  
        showpred = 1
```

```
if file_image is None:  
    st.write("You haven't uploaded any image file")  
else:  
    input_img = Image.open(file_image)  
    st.write("***Input Photo**")  
    st.image(input_img, use_column_width=True)  
    src = np.array(input_img)  
    if color_space == "bw":  
        image = cv.cvtColor(src, cv.COLOR_BGR2GRAY)  
    if color_space == "hsv":  
        image = cv.cvtColor(src, cv.COLOR_BGR2HSV)  
    if color_space == "yuv":  
        image = cv.cvtColor(src, cv.COLOR_BGR2YUV)  
    if color_space == "lab":  
        image = cv.cvtColor(src, cv.COLOR_BGR2LAB)  
    st.write("***Output Pencil Sketch**")  
    st.image(image,use_column_width=True)
```

Now, open Bash or Terminal, wherever the code is, and then run **streamlit run Master.py**. Then, open the following link in your browser: **http://localhost:8051**.

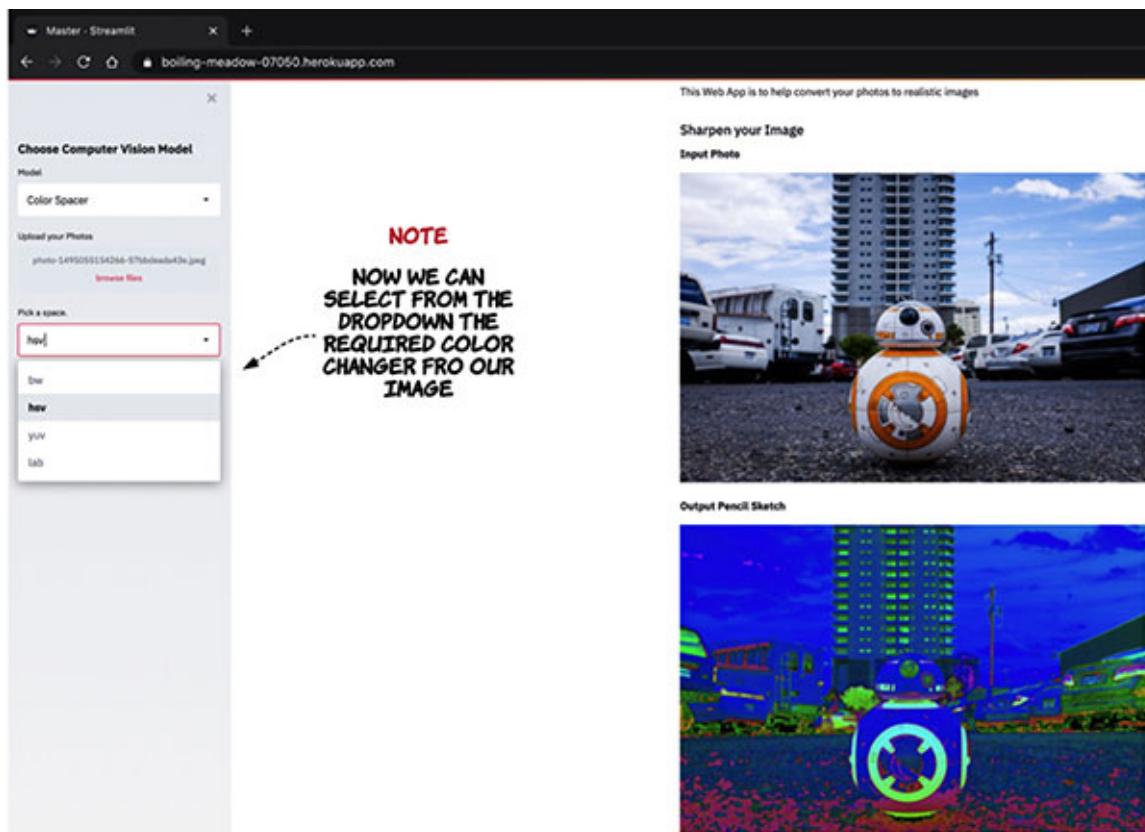


Figure 8.16: OpenCV Colour Changer Output

So, the preceding is the output for the OpenCV model components in Streamlit.

We composed all the models under a function like the following:

```
def main():
{
    CODE
}

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

To locally debug the test, we run the command, Streamlit run **Basics.py**.

You can now view your Streamlit app in your browser.

Local URL: <http://localhost:8501>

Network URL: <http://192.168.29.199:8501>

Figure 8.17: Local Host URL for App

8.5 Creating the components for Heroku Deployment

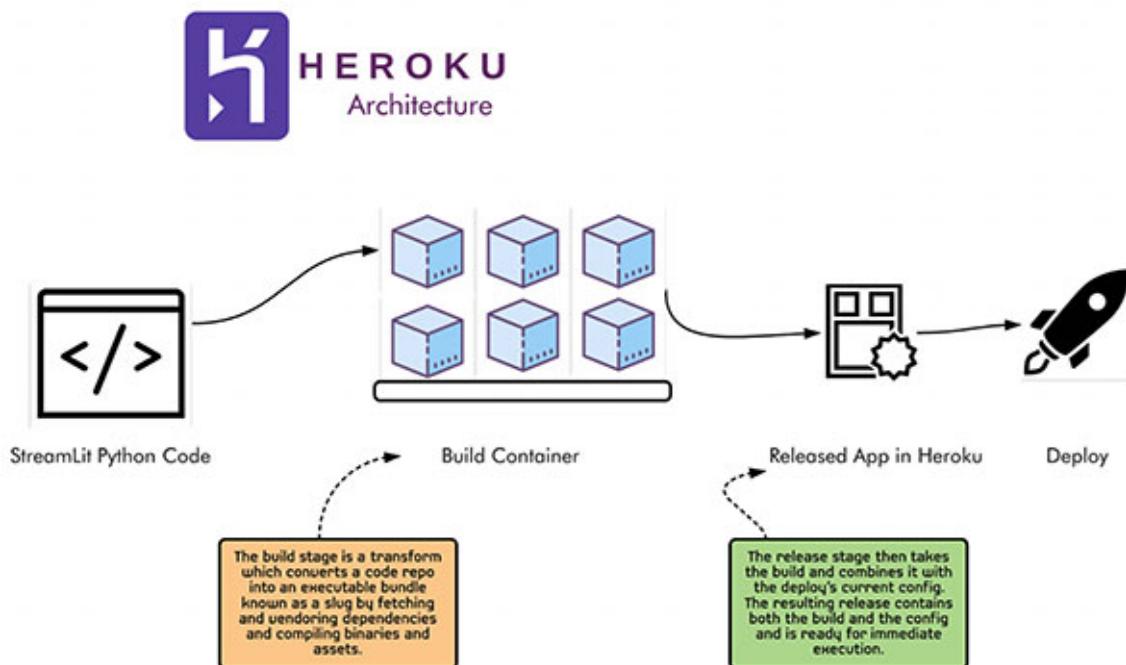


Figure 8.18: Heroku Architecture

Now, create the Python Script for your application, containerize your whole Framework with Docker, release the build to App inside Heroku, and then host the application inside and deploy it there, which will generate a host URL.

Now, we will build the Docker image; for that, make sure you start the Docker, after which you will see the Docker icon activated on the top. In the following screenshot, we can see that the first icon Docker is started. You can stop or restart from that icon.



Figure 8.19: Docker Activation

Heroku provides three ways to deploy your Dockerized app. We will be using the Container Registry.

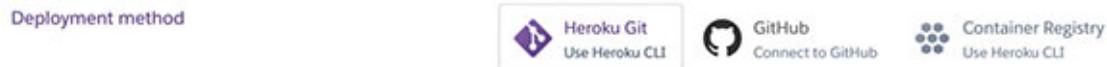


Figure 8.20: Heroku Deployment Method

The Heroku Deployment folder consists of the following files:



Figure 8.21: Heroku Deployment Folder

Now, let's create the Docker file for the Heroku Deployment.

Dockerfile:

```
FROM ubuntu:18.04
# streamlit-specific commands for config
ENV LC_ALL=C.UTF-8
ENV LANG=C.UTF-8
RUN mkdir -p /root/.streamlit

RUN bash -c 'echo -e "\n[general]\n\"
```

```
email = '\"\"\n\" > /root/.streamlit/credentials.toml'\nRUN bash -c 'echo -e \"\n[server]\nenableCORS = false\n\" > /root/.streamlit/config.toml'\n\nRUN apt-get update && \
    apt-get install -y \
    python3.7 python3-pip \
    libsm6 libxext6 libxrender-dev
EXPOSE 8501
# make app directory
WORKDIR /streamlit-docker\n\n
1 streamlit
2 black
3 altair
4 pandas
5 pydeck
6 matplotlib
7 numpy
8 Pillow
9 bs4
10 pipenv
11 xlrd
12 opencv-python==4.2.0.34
13 gdown
```

Figure 8.22: Heroku Docker requirements File

```
COPY requirements.txt ./requirements.txt  
RUN pip3 install -r requirements.txt  
COPY . .
```

Figure 8.22: Heroku Docker requirements File

```
RUN chmod +x ./heroku_startup.sh  
ENTRYPOINT "./heroku_startup.sh"
```

Let's breakdown the Docker as follows:

- **FROM:** To get a base image. Just like you need an OS as a basis of your application.
- **MAINTAINER:** To show the message about the author of this image.
- **ENV:** To set an environment variable to a certain value.
- **RUN:** To execute the command requirements file which contains the Python libraries.
- **EXPOSE:** To let your container listen on the specific port while running.
- **WORKDIR/ COPY:** To set the working directory, copy new files/directories.
- **CMD:** To provide default actions when executing the container.
- Change the permission of **heroku_startup.sh** or you will receive the permission denied error while executing **heroku_startup.sh**.
- **ENTRYPOINT "./heroku_startup.sh":** To tell the Docker to execute heroku_startup.sh when starting the container.
- The purpose of the second and the third RUN Command in the preceding line is to let OpenCV run normally in Docker to mitigate an issue that if we don't install libSM.so.6 and import OpenCV when building the Docker Image, our container will silently crash with the following message: segmentation fault (core dumped) when executing.



Heroku.yml:

```
build:  
  docker:  
    web: heroku.Dockerfile
```

run:

```
web: ./heroku_startup.sh
```

Heroku needs **heroku.yml** to build and deploy the Docker images, so we will create one for the usage. Obviously, the build and run sections indicate what we wanted to do in build and run in the stage on Heroku, respectively.



Heroku_startup.sh:

```
echo PORT $PORT  
streamlit run --server.port $PORT Master.py
```

Heroku uses a \$PORT environment variable for the port exposure. We have to set the \$PORT variable to Streamlit or your app will not appear. In this case, we will expose at 8501.

Installation:

Now download Heroku CLI : <https://devcenter.heroku.com/articles/heroku-cli#download-and-install>

Install Brew: <https://brew.sh/>



```
brew tap heroku/brew && brew install heroku
```



```
sudo snap install --classic heroku
```

The Component Steps are as follows:

- First log in to your Heroku account and follow the prompts to create a new SSH public key.

```
$ heroku login
```

- Sign in to the Container Registry:

```
$ heroku container:login
```

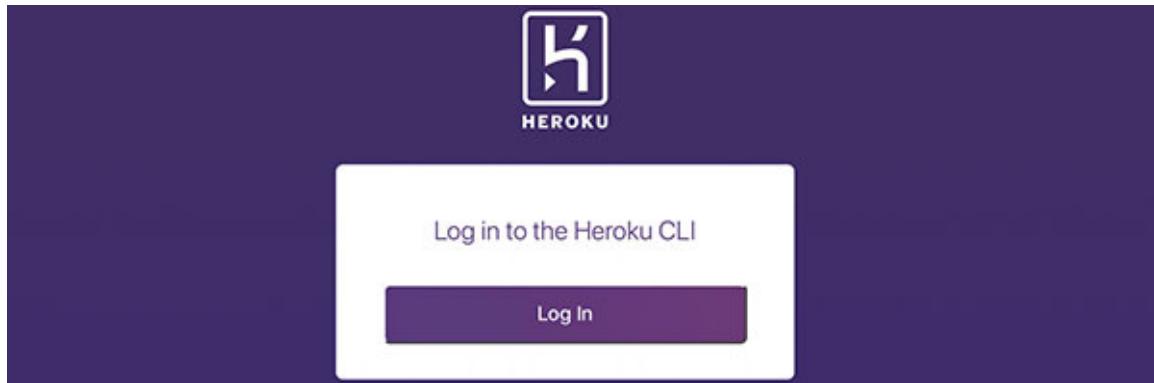


Figure 8.23: Heroku Login for CLI

- Navigate to the app's directory and create a Heroku app:

```
$ heroku create
```

```
Creating app... done, ⬤ hidden-river-78993
https://hidden-river-78993.herokuapp.com/ | https://git.heroku.com/hidden-river-78993.git
```

Figure 8.24: Heroku App creation

- Build the image and push to the Container Registry in one command. Make sure you started the Docker. And make sure you change the app name below for your case which will be created above.

```
$ heroku container:push web -a hidden-river-78993
```

```
(base) [REDACTED].Heroku[REDACTED]$ heroku container:push web -a hidden-river-78993
== Building web (/Users/[REDACTED]/Downloads/Streamlit/ComputerVision/Heroku/Dockerfile)
Sending build context to Docker daemon 16.38kB
Step 1/14 : FROM ubuntu:18.04
18.04: Pulling from library/ubuntu
5d9821c94847: Downloading 25.57MB/26.7MB
a610eae58dfc: Download complete
a40e0eb9f140: Download complete
Successfully built 68a10f4d1819
Successfully tagged registry.heroku.com/hidden-river-78993/web:latest
```

Figure 8.25: Heroku Container Built and Pushed message

- Then release the image to our app which we have created; here it will be hidden-river-78993.

```
$ heroku container:release web -a hidden-river-78993
```

```
Releasing images web to hidden-river-78993... done
```

Figure 8.26: Heroku Container released to App

Now, open the app in your browser or we can navigate to the website and click on **Open app** at the right corner.

```
$ heroku open-a hidden-river-78993
```

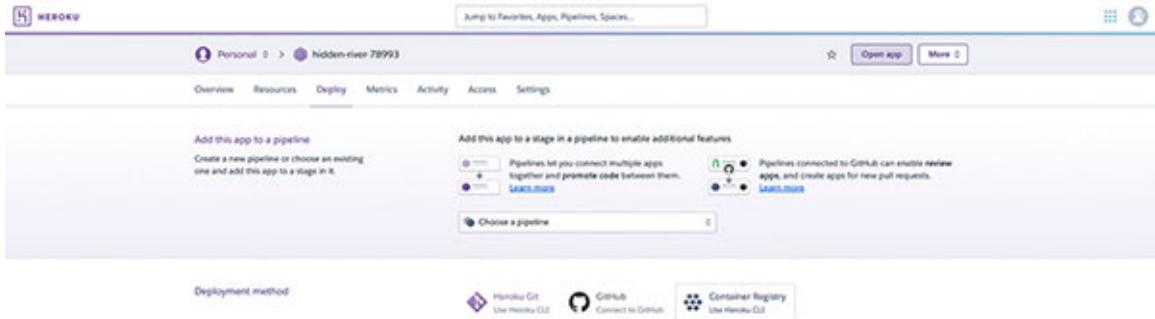


Figure 8.27: Heroku Website Click Open App

Now, we can see our web Application hosted in Heroku.

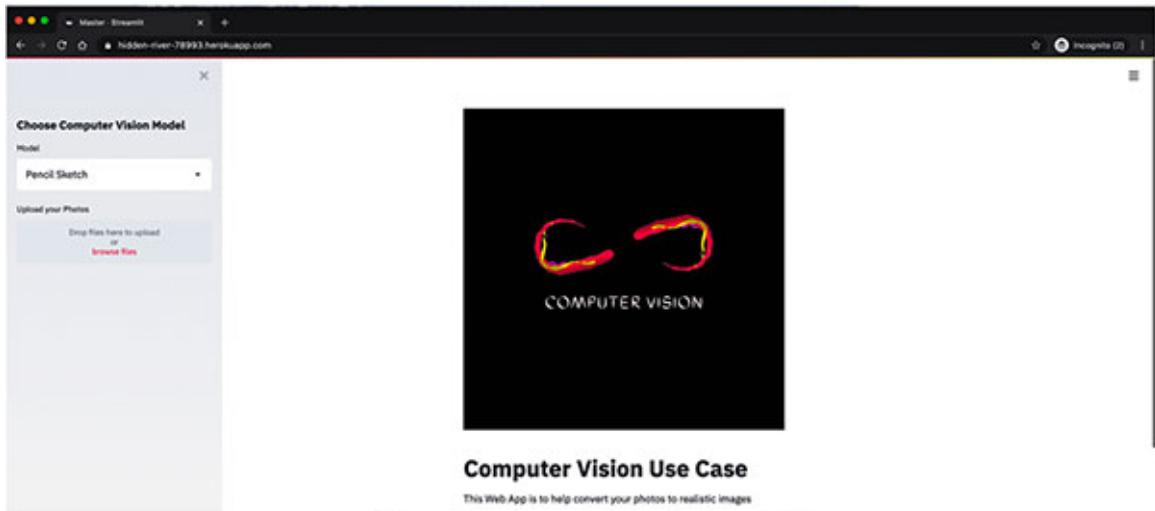


Figure 8.28: Computer Vision App hosted in Heroku

8.6 Deploying the Streamlit code by containerizing in Kubernetes cluster

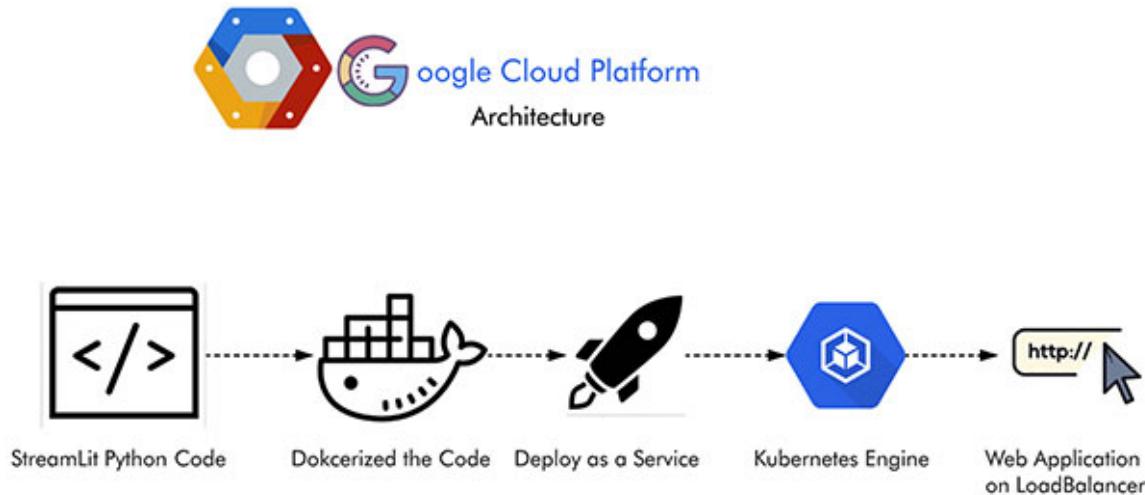


Figure 8.29: Architecture for GCP Kubernetes Deployment

Now, create the Python script for your application and containerize your whole Framework with Docker and deploy the application inside Kubernetes cluster as a deployment service and the Docker image will be exposed as a Load Balancer deploy, which will generate a host URL on port 8501:

- A Container is a type of software that packages up an application and all its dependencies, so the application runs reliably from one computing environment to another.
- Docker is a software used for building and managing the containers.
- Kubernetes is an open-source system for managing the containerized applications in a clustered environment.
- Google Kubernetes Engine is an implementation of the open source Kubernetes framework on the Google Cloud Platform.

Let's build the Docker image with everything being the same from earlier, just remove the last two lines from the earlier Heroku Dockerfile and add one line of CMD to run the Streamlit Python script:

```
FROM ubuntu:18.04
ENV LC_ALL=C.UTF-8
ENV LANG=C.UTF-8
RUN mkdir -p /root/.streamlit
RUN bash -c 'echo -e "\n[general]\n\n" > /root/.streamlit/config.json'
```

```

email = '\"\"\n\
" > /root/.streamlit/credentials.toml'

RUN bash -c 'echo -e \"\
[server]\n\
enableCORS = false\n\
" > /root/.streamlit/config.toml'

RUN apt-get update && \
    apt-get install -y \
    python3.7 python3-pip \
    libsm6 libxext6 libxrender-dev

EXPOSE 8501

WORKDIR /streamlit-docker
COPY requirements.txt ./requirements.txt
RUN pip3 install -r requirements.txt
COPY . .
CMD streamlit run Master.py

```

Now, run the following commands in Bash to build the Docker image for your GCP deployment, assuming you have the GCP project as a pre-requisite:

```

``` bash
gcloud init
gcloud auth configure-docker
PROJECT_ID=$(gcloud config get-value core/project)
IMAGE_NAME=opencv-streamlit
IMAGE_NAME=gcr.io/$PROJECT_ID/$IMAGE_NAME
IMAGE_TAG=v1

build image

docker build --no-cache -t $IMAGE_NAME:$IMAGE_TAG .
docker push $IMAGE_NAME:$IMAGE_TAG
```

```

Let's say our Docker image will be like the following:
gcr.io/<PROJECT_ID>/opencv-streamlit:v1

Make sure you install kubectl which we have installed in [Chapter 1, Introduction to Kubeflow & Kubernetes Cloud Architecture](#). Also, go to the following link: <https://kubernetes.io/docs/tasks/tools/install-kubectl/>

Create Cluster:

To setup the Kubernetes cluster, either you use Google Cloud Shell or Microsoft Visual Studio Terminal of your system. Navigate to **Google Cloud Platform > Kubernetes > Cluster**.

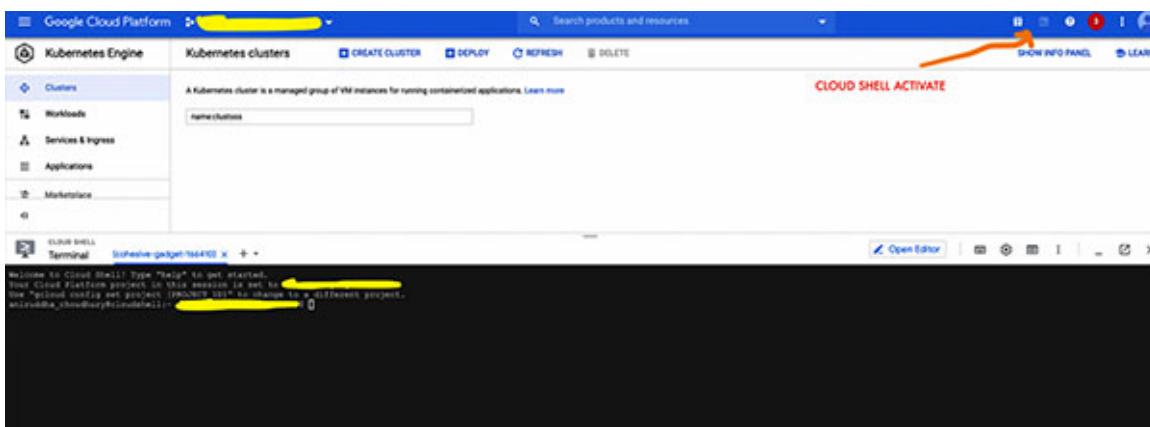


Figure 8.30: GCP Cloud Shell

Now, complete the following steps:

1. Set your project ID and Compute Engine zone options for the gcloud tool.

```
PROJECT_ID=$(gcloud config get-value core/project)
gcloud config set project $PROJECT_ID
gcloud config set compute/zone us-central1
```

2. Create a cluster by executing the following code:

```
gcloud container clusters create streamlit-computer-vision --
num-nodes=2
```

```
kubeconfig entry generated for streamlit-computer-vision.
NAME          LOCATION      MASTER_VERSION  MASTER_IP    MACHINE_TYPE   NODE_VERSION    NUM_NODES  STATUS
streamlit-computer-vision  us-central1  1.15.12-gke.20  34.68.60.2  n1-standard-1  1.15.12-gke.20  6        RUNNING
```

Figure 8.31: Cluster Creation

3. Connect to cluster by clicking on the **Connect** button.

```
gcloud container clusters get-credentials streamlit-computer-vision --region us-central1 --project <PROJECT_ID>
```

The screenshot shows the Google Cloud Platform Cluster Connect interface. On the left, there's a sidebar with options: Services & Ingress, Applications, Configuration, and Storage. The main area is a table titled 'Name' with a single row: 'streamlit-computer-vision'. The table includes columns for Location (us-central1), Cluster size (6), Total cores (6 vCPUs), Total memory (22.50 GB), Notifications, and Labels. There are also 'Connect', 'Edit', and 'Delete' buttons at the top right of the table.

Figure 8.32: Cluster Connect

4. To deploy and manage the applications on a GKE cluster, you must communicate with the Kubernetes cluster management system. Execute the following command to deploy the application:

```
kubectl create deployment computervision-streamlit --image=gcr.io/<PROJECT_ID>/opencv-streamlit:v1
```

```
deployment.apps/computervision-streamlit created
```

Figure 8.33: Successful Deployment Message

5. The containers you run on GKE are not accessible from the internet because they do not have external IP addresses. Execute the following code to expose the application to the internet:

```
kubectl expose deployment computervision-streamlit --type=LoadBalancer --port 80 --target-port 8501
```

```
service/computervision-streamlit exposed
```

Figure 8.34: Successful LoadBalancer Service Message

6. Execute the following code to get the status of the service. EXTERNAL-IP is the web address you can use in the browser to view the published app.

```
kubectl get service
```

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
|--------------------------|--------------|--------------|-------------|--------------|-------|
| computervision-streamlit | LoadBalancer | 10.11.241.25 | 34.71.219.0 | 80:30883/TCP | 5m24s |
| kubernetes | ClusterIP | 10.11.240.1 | <none> | 443/TCP | 22m |

Figure 8.35: Successful Service Message

Copy the External IP and paste it in a browser. We will get our application **<http://34.71.219.0>**

Now, we can see our web application hosted in Google Cloud Platform:



Figure 8.36: Computer Vision App hosted in Google Kubernetes Engine

8.7 Summary

In this chapter, we learned how to build a web application with the Streamlit tool in Python without having prior knowledge to any other language. Then, we hosted the application in the Heroku Platform and Google Kubernetes Engine.

Next, we learned about the various Streamlit components for how to display the text, image, charts, and so on. Also, we built the framework for our web application for the various computer vision OpenCV models like Sharpening, Cropping Image, and so on. After that, we saw the architecture to build the Heroku deployment which is a common skeleton to be used for all Skelton Machine Learning web application deployment alongside Google Kubernetes Engine.

In this chapter, we have gained knowledge on how to leverage the power of Google Cloud Platform and how to use your Devops knowledge with Machine Learning to become an MLops. Also, we also learned how to use Streamlit to build Skelton Web application in just a few hours in Heroku and Cloud.

8.8 References

- <https://devcenter.heroku.com/categories/reference><https://technowhis.p.com/kaggle-api-python-documentation/>
- <https://docs.streamlit.io/en/stable/api.html>
- <https://kubernetes.io/docs/tutorials/kubernetes-basics/>
- <https://cloud.google.com/kubernetes-engine/docs/quickstart>
- <https://www.pyimagesearch.com/start-here/>
- <https://docs.docker.com/get-started/overview/>
- <https://devcenter.heroku.com/articles/container-registry-and-runtime#unsupported-dockerfile-commands>

Index

A

Amazon SageMaker Notebook Instance
 creating [242-245](#)
Amazon SageMaker Training model [261](#)
 data, splitting into train/validation [262](#)
 deploying [267-271](#)
 SageMaker API XG-Boost, using [264-267](#)
 train data, pushing to S3 [263](#)
 validation data, pushing to S3 [263](#)
auto-scaling
 algorithm [129](#)
 panic [129](#)
 working concept [129](#)
AWS S3 Bucket
 creating [241](#)
AWS SageMaker
 problem statement [236](#)
 setup [236-240](#)

B

brain tumor [87](#)
breakdown technique
 for building production pipeline [46](#)

C

Cloud IAP
 OAuth, setting up from [27](#)
cluster [10](#)
CNN Kubeflow Pipeline
 architecture [90, 91](#)
 building [90](#)
 data extraction or ingestion component, building [91-95](#)
 data pre-processing component, building [95-100](#)
 evaluation component, building [106](#)

GCP Kubeflow setup [86](#)
model, serving with KF Serving [106-113](#)
problem statement [86](#)
problem statement analytics [87](#)
training model component, building [101-105](#)
components, KF Serving
 endpoint [66](#)
 explainer [66](#)
 predictor [66](#)
 transformer [66](#)
components, Kubeflow [20](#)
 central UI [21](#)
 dashboard [24](#)
 Jupyter Notebook server [23](#)
 Katib [24](#)
 Metadata [22, 23](#)
 registration flow [21, 22](#)
container [3](#)
 features [2, 3](#)
container isolation [3](#)
contrastive explanations [178](#)
cumulative effect of interactions [186](#)
custom Docker image
 building [35](#)
 using, in Jupyter Notebook [36](#)

D

decision plot
 features [184-186](#)
default namespace [18](#)
deployment [12](#)
 using [13](#)
desirable properties, SHAP
 consistency [177, 178](#)
 local accuracy [177](#)
 missingness [177](#)
Docker [2](#)
 reference link, for installation [29](#)
 setting up [29](#)
Docker architecture

- image [3](#)
- VM containers [3](#)
- Dockerfile [4, 5](#)
- Docker Hub
 - URL [29](#)
- Docker image [3](#)

F

- Fairness Indicators [159](#)
- Fairness metric visualization [159](#)
- feature exposure [215](#)
- force plots analysis [186-189](#)

G

- gcloudsdk
 - installing [26](#)
 - setting up [26, 27](#)
- GCP Kubeflow setup
 - cluster, connecting to [34](#)
 - Docker, setting up [29](#)
 - gcloudsdk, installing [26](#)
 - gcloudsdk, setting up [26, 27](#)
 - Grafana, deploying [34](#)
 - kubectl, installing [26](#)
 - kubectl, setting up [26](#)
 - Kuberflow, setting up [30-34](#)
 - OAuth, setting up from Cloud IAP [27, 28](#)
 - performing [25, 26](#)
 - prerequisites [24](#)
 - reference link [24](#)
- Google Cloud Platform (GCP) [24](#)
- Google Kubernetes Engine (GKE) [8](#)
- Grafana dashboard
 - serving traffic endpoint performance, monitoring with [81-84](#)

H

- Heroku Deployment
 - components, creating for [296-301](#)
- high availability

health checks [7](#)
load balancing [7](#)
traffic routing [7](#)
histogram-based algorithms
advantages [218](#)

I

Ingress [15](#)
architecture [16, 17](#)

J

Jupyter Notebook
correlation and scatter plots [251-254](#)
custom Docker image, using [36-39](#)
data, loading from Python library [247](#)
data, loading from S3 [247, 248](#)
feature engineering [248](#)
feature transformation [257-261](#)
launching [245, 246](#)
missing values sum, checking [249, 250](#)
model dependent column transformation [250, 251](#)
numerical and categorical columns, finding [248, 249](#)
outlier detection [254-257](#)
PVC setup [39-43](#)
server [23](#)
server, setting up in Kuberflow [35](#)

K

Kaggle API
API Key, setting up [203, 204](#)
installing [203](#)
URL [203](#)
Katib
reference link [24](#)
kfctl command line interface (CLI)
using [31](#)
KF Serving
architecture [66](#)
components [66](#)

LightGBM model, serving with [226-231](#)
used, for serving TensorFlow model [65](#)
using [67](#)

Knative [124](#)

Knative Pod Autoscaler (KPA) [125](#)

Knative Serving [124](#)

kubectl

installing [26](#)

setting up [26](#)

Kubeflow [18](#)

architecture [19](#)

components [20](#)

features [20](#)

Jupyter Notebook server, setting up [35](#)

setting up, in Kubernetes cluster [30-34](#)

Kubeflow central UI

URL [21](#)

Kubeflow orchestration, for ML deployment [18](#)

Kubeflow pipeline

building [47](#)

building, for TensorFlow model [49](#)

data extraction [47](#)

data ingestion [47](#)

data pre-processing [48](#)

GCP Kuberflow setup [46](#)

problem statement [46](#)

serving [48](#)

training [48](#)

Kubeflow Pipeline Orchestrator

building [164-173](#)

kube-node-lease namespace [18](#)

kube-public namespace [18](#)

Kubernetes

advantages [6](#)

capabilities [5, 6](#)

cluster orchestration system [8](#)

components [9](#)

features [6](#)

master [8](#)

Nodes [9](#)

purpose [6](#)

slave [9](#)

working [8](#)

Kubernetes components

cluster [10](#)

deployment [12](#)

network [11](#)

node [9, 10](#)

pod [11](#)

service [14](#)

storage [12](#)

Kubernetes, features

automated rollouts and rollbacks [7](#)

canary deployments [7](#)

designed for deployment [7](#)

high availability [7](#)

open-source [6](#)

portable [6](#)

programming languages and framework support [7](#)

Stateful containers [7](#)

workload scalability [6](#)

kube-system namespace [18](#)

L

LightGBM [182](#)

architecture [217](#)

working [217](#)

LightGBM model

data extracting [213](#)

data loading [213](#)

exploratory data analysis [213-215](#)

Kaggle API key setup [211, 212](#)

Kaggle setup, obtaining [209](#)

library, installing [209](#)

modeling, for equity data [208](#)

performance, monitoring with Grafana dashboard [231, 232](#)

serving, with KF Serving [226-231](#)

training [182, 183](#)

training, for equity data [208](#)

training, with Weights & Biases [217-226](#)

utility metrics function [215](#)

Wandb API key setup [210](#)
Wandb dependency, obtaining [209](#)
LightGBM parameters
reference link [220](#)

M

machine learning (ML) [22](#)
master, Kubernetes
 API server [8](#)
 Control Manager [9](#)
 scheduler [9](#)
mean absolute error [216](#)
metadata, Kuberflow
 managing [22, 23](#)
model analysis
 with advance visualization, along SHAP tool [183, 184](#)

N

namespaces
 default [18](#)
 kube-node-lease [18](#)
 kube-public [18](#)
 kube-system [18](#)
 multiple namespaces, using [17, 18](#)
network [11](#)
node [9, 10](#)
Nodes
 Container Engine [9](#)
 Kubelet [9](#)
 Kube Proxy [9](#)
 Pod [9](#)

O

OAuth
 setting up, from Cloud IAP [27, 28](#)

P

persistent volume (PV) [37, 38](#)

pipeline, CNN TensorFlow model
building [114-124](#)
pipeline, TensorFlow model
building [73-81](#)
pod [11](#)
Principal Component Analysis (PCA) [215](#)
profile setup, Kubeflow [21, 22](#)
project requirements, GCP & Heroku
setup [274, 275](#)
project requirements, GCP & Wandb
Kaggle API setup [203, 204](#)
Kubeflow Cluster [203](#)
setting up [202](#)
Weights & Biases API Key [204-206](#)
PVC
setting up, for Jupyter Notebook [39-43](#)

R

root mean square error [216](#)

S

Service [14](#)
purpose [14](#)
types [15](#)
serving component, TensorFlow model pipeline
building [67-72](#)
serving endpoint, CNN TensorFlow model
auto-scaling, with Knative [124-128](#)
serving traffic endpoint, TensorFlow model
performance, monitoring with Grafana [81-83](#)
SHAP
desirable properties [177](#)
explanation example [178](#)
goal [176](#)
installation [179](#)
Python libraries, installing [179](#)
solid theoretical foundation [178](#)
Spearman correlation [216](#)
Stateful containers

ephemeral storage volume [7](#)
persistent storage [8](#)
volume [7](#)
storage [12](#)
Streamlit [275](#)
code deployment, by containerizing in Kubernetes cluster [301-305](#)
commands, writing into Python script [277](#)
data & chart, working with [284](#), [285](#)
features [276](#), [277](#)
framework, building for OpenCV models [285-295](#)
media, working with [279-282](#)
text, working with [277](#), [278](#)
Streamlit component
building [275](#), [276](#)

T

Tensorboard
advanced visualization, for TensorFlow model [195-197](#)
TensorFlow Estimator model
building [189-193](#)
building, with DNN Classifier [194](#)
building, with Linear Model Classifier estimator [194](#)
building, with two estimators [193](#)
TensorFlow model
evaluating, with Tensorboard [195-197](#)
evaluating, with What-If tool [197-199](#)
TensorFlow Model Analysis (TFMA) [158](#)
TensorFlow model pipeline
building [49](#), [50](#)
data extraction or ingestion component, building [50-54](#)
data pre-processing component, building [54-59](#)
Docker image, building for data extraction [53](#)
evaluation component, building [65](#)
training model component, building [60-65](#)
TensorFlow Serving
inference, performing on example data [162-164](#)
installing [161](#)
model, serving with [161](#)
running [162](#)
TFMA metric visualization [158](#)

TFX components
Evaluator [133](#)
ExampleGen [133](#)
ExampleValidator [133](#)
functionality [134](#)
Pusher [133](#)
SchemaGen [133](#)
StatisticsGen [133](#)
Trainer [133](#)
Transform [133](#)
TFX environment setup [134](#)
root directory, setting [135](#), [136](#)
TFX Pipeline
architecture [132](#)
problem statement [132](#)
TFX pipeline components [136](#), [137](#)
Evaluator, building [157](#), [158](#)
ExampleGen, building [137](#), [138](#)
ExampleValidator, building [142](#), [143](#)
Pusher, building [159](#), [160](#)
SchemaGen, building [140-142](#)
StatisticsGen, building [139](#), [140](#)
Trainer, building [149-155](#)
training, analyzing with TensorBoard [156](#)
Transform, building [143-148](#)
Tuner, building [156](#)
training model
feature transformation [180-182](#)
types, Services
ClusterIP (default) [15](#)
ExternalName [15](#)
Headless [15](#)
LoadBalancer [15](#)
NodePort [15](#)

U

utility metrics function
mean absolute error [216](#)
root mean square error [216](#)
Spearman correlation [216](#)

W

Wandb sweep architecture [219](#)

Weights & Biases

 API, creating [206](#)

 API Key [204](#)

 features [206](#)

 Framework and Cloud support [205](#)

 using [207, 208](#)

What-If tool

 advanced visualization, for TensorFlow model [197-199](#)

wit-widget

 installation [179](#)

 Python libraries, installing [179, 180](#)

workload scalability

 auto-scaling [6](#)

 horizontal infrastructure scaling [6](#)

 manual scaling [6](#)

 replication controller [7](#)