Applied Machine Learning Solutions with Python

Production-ready ML Projects Using Cutting-edge Libraries and Powerful Statistical Techniques

SIDDHANTA BHATTA

Applied Machine Learning Solutions with Python

Production-ready ML Projects Using Cutting-edge Libraries and Powerful Statistical Techniques

Siddhanta Bhatta

www.bpbonline.com

FIRST EDITION 2022

Copyright © BPB Publications, India

ISBN: 978-93-91030-438

Dedicated to

My Family and Friends

About the Author

Siddhanta Bhatta works as a Senior Software Engineer at Dell with over 6 years of experience in the field of Software Engineering, Machine Learning. He has experience using statistical modeling, natural language processing, deep learning algorithms to solve challenging business problems. Passionate about building full-stack machine learning solutions, deep reinforcement learning, and dimensionality reduction techniques. He is motivated towards building data literacy and making machine learning accessible to everyone.

Outside work, Siddhanta volunteers his spare time as a Machine Learning mentor, helping and mentoring young data scientists in taking up careers in technology.

About the Reviewers

- **Nishant Kumar** is a machine learning engineer currently working at Branch International (a microloans fintech startup). He has over six years of experience and has successfully designed and developed machine learning and deep learning systems for multiple startups. He has implemented machine learning algorithms ranging from RandomForest, XGBoost, CNNs, LSTMs, Transformers, etc., across domains like healthcare, e-commerce, education, and finance. In addition, he co-founded a startup Dijkstra Labs, which helped traditional organizations leverage machine learning. He graduated from IIT Delhi in 2015 with a B.Tech. in chemical engineering and has since then been working on artificial intelligence and data products.

- **Sandeep Kosanam** is a senior data scientist with work experience in ML/DL life cycle. He has good experience in working on huge data by leveraging GPU infrastructure and scaling out the applications using Dask.

  Sandeep has good hands-on experience on NLP use cases, which can extract intents and entities. Major experience is in the Banking and Pharma sector.

  He has worked on highly challenging projects that added good value along with good customer satisfaction.

- **Sonam Chawla Bhatia** has ten years of experience in software research and development. She has worked in proof-of-concept development of various projects based on machine learning, artificial intelligence, the Internet of Things, and blockchain. Sonam pursued M.E in software engineering from the department of computer science, Thapar University, Patiala. She has worked with Samsung, Bangalore and Noida for nine years. She is currently working as a senior software engineer at Microsoft, Noida.

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 best friend and lifelong support, Soumya, who supported me over the entire period of this book writing—I could have never completed this book without her support.

This book would not have happened without the support from my parents. They always encouraged me to pursue writing. I would like to thank Sandeep, one of my colleagues, who is like a brother to me. He has given me a lot of insights to write this book and is one of the technical reviewers also.

Finally, I would like to thank BPB Publications for giving me this opportunity to write my first book for them.

Preface

The first computer program that can be categorized as a program that can learn, first came into picture in 1952, It was a game that played checkers, created by Arthur Samuel. He also coined the term "Machine Learning". I wonder if he realized how popular this word will become. But the progress in the field of Artificial Intelligence halted till 1980s. With Hopfield's recurrent neural networks, the neural networks started gaining researchers' interest again. Then a groundbreaking idea of "Back propagation" was invented by Carnegie Mellon professor and computer scientist Geoffrey Hinton. With that, rapid advancements came into the field of machine learning and in 2006, he published a paper showing how to train a deep neural network capable of recognizing handwritten digits with state-of-the-art precision (98%). He also coins the term "Deep Learning" and a revolution began. Fast forward to 14 years, and unless you are living under a rock, you have at least heard of machine learning. It now auto completes your sentences while you write your email, gives you movie suggestions, ranks your search queries, defeats professionals in their own game(OpenAI), raises security issues etc. It’s everywhere now. And it’s a highly attractive skill to know machine learning.

As Arthur C. Clark, rightly said, "A sufficiently advanced technology is indistinguishable from magic", Machine Learning embodies this statement better than any other technology in the current time. But the objective of this book is to break this spell. To bring machine learning into the stage and look deeper into its working. And by the time you finish reading this book, you will understand machine learning is not magic. It’s just sufficiently advanced technology with rapid development in computation hardware (cheap GPU, TPU, and so on.) and the availability of big data. These two are the cornerstones of modern deep learning. So, the objective of this book is to make machine learning more accessible and weaken the idea that a machine learning engineer needs to first develop statistics, linear algebra, machine learning concepts to become a machine learning practitioner. You need less of these and more experimentation and coding knowledge to do machine learning in the Industry. Research, however, is a field where a deeper understanding of underlying math and theory is necessary. But in my opinion, the programming aspect of machine learning is much more relevant than mere theoretical learning.

Machine Learning is a new field and with any new field, you will get a lot of opinionated ideas. Things are not completely standardized in machine learning yet. Although some practices are common and provide good results almost all the time. In this book, you will find a lot of practices that are not common. These practices will provide you with tools to tackle any machine learning problems in Industry and provide you with techniques by which you can create your practices.

This book also focuses on the practical implementation of machine learning ideas and play with real-world data and real-world problems. We will go through various data sources and understand the importance of data pipelines. We will go through a lot of examples and case studies that will give you an ample idea of how machine learning works in the Industry.

We will follow a top-down approach to machine learning, where we will go through the high-level idea and implementation (through working code!) first. then we will drill down into the details and finetune our models. We will go through some machine learning concepts in depth too wherever necessary but the overall focus of this book is to provide a practitioner's approach. What skills do you need to create an industry-standard machine learning project on messy data? We will detour to a few programming concepts and ideas in statistics and linear algebra. But we will keep that minimum. Over the 15 chapters in this book, you will learn the following:

Chapter 1 introduces the machine learning process, and a plethora of examples and use cases show how machine learning can be leveraged to make awesome products. It also introduces a lot of machine learning jargon that makes following later chapters easier.

Chapter 2 discusses the first step in building any data product, that is, problem formulation. This is one of the most overlooked areas in machine learning but carries huge importance in giving value to the industry. This chapter encourages the reader to ask few questions to evaluate the feasibility of a machine learning problem.

Chapter 3 gives tips and tricks for data acquisition and cleaning, the second step in a machine learning process. We will go through various tools to ensure data sufficiency and augmentation. We will also understand the need for automation in acquiring data. Cleaning data is a tricky concept and often misunderstood and this chapter ensures what questions to ask for when cleaning a dataset.

Chapter 4 discusses the third step in a machine learning process which is data exploration. We will understand the need for it and this chapter shows the disadvantage of extensive EDA in the beginning and how it can introduce bias into the machine learning problem. We will go through the initial exploration along with visualization tips. At each step, the chapter will explain why it's needed.

Chapter 5 has the model selection and tuning techniques. This chapter will introduce a rather trivial way of choosing models than searching over a large space of models. We will go through few initial models that generally perform well and then take our model selection from there. We will also learn about hyperparameter tuning techniques. We will learn about tips and tricks to ease this time-intensive process.

Chapter 6 introduces model deployment techniques. It goes through a formal process of providing users an interface to use our model. We will go through few REST API frameworks with python and understand the need for model interpretation techniques. The chapter also introduces tools like MLFlow which will help readers to maintain a machine learning project.

Chapter 7 explores the data analytics world. From this chapter onwards we will enter the second section of the book that goes through a lot of industry use cases. We will learn best practices in data analytics and go through a practical example. We will go through all the processes mentioned in section 1(chapter 1- 6) for tabular data.

Chapter 8 describes how to build custom image classifiers from scratch. It will introduce transfer learning and how it's a revolutionary idea. It will be about deep learning and provide a quick intro and talk about ways to tune a deep learning model.

Chapter 9 explores an NLP use case. In this chapter, we will build a cutting-edge news summarization application from scratch. We will use web scrapping to get the data, train our model with transfer learning and learn about data cleaning, feature extraction techniques. We will go through a quick guide on RNNs, and then explore transformers concept and how it has changed the world of NLP forever.

Chapter 10 explores multiple input and multiple output models, we will learn how to look beyond traditional deep learning architectures and build complex networks using the functional API from TensorFlow, we will also learn about model subclassing and build a visual question answering model that can ask questions to an image.

Chapter 11 marks the beginning of section 3 of the book, where we will explore building a machine learning profile. In this chapter, we will explore contributing to the community. Explore areas of research for machine learning and learn how to efficiently read a machine learning paper. We will also explore the avenue of contributing to open-source projects. Finally, we will talk about writing blogs and building Kaggle kernels.

Chapter 12 explores building our projects for machine learning. We will go through a plethora of projects and define goals for each. We will then explore some useful and underused tools that can vastly improve your machine learning experience.

Chapter 13 – 15 has section 4 of the book, which is crash courses for few important libraries such as Numpy, Matplotlib, and Pandas(Chapter 13). There will be crash courses on linear algebra and statistics(Chapter 14), followed by the last chapter that goes through the FastAPI crash course (Chapter 15).

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/b45284

Errata

Table of Contents

Chapter 1
Introduction to Machine Learning

Machine learning is not just part of sci-fi movies anymore. It is already here. Not the kind that is shown in movies, though; our models will not plot against humanity because they found humans are destroying themselves, like in the movie iRobot. At least not yet. *But what exactly is machine learning? How does it work? How can a piece of code learn? Where does it end? What are its capabilities, and how can we use it?* We will go through these questions and why you want to use them too.

This chapter will also introduce many machine learning jargons and give a bird's eye view of machine learning types and landscape through an example. We will learn about supervised learning, unsupervised machine learning, and semi-supervised machine learning. We will go through few modern examples and learn how those work to understand how important machine learning is and its capabilities. We will go through Gmail Smart compose feature and Netflix recommendation. We will go through some of the key ideas and not the exact implementation to give you a feel and admiration of machine learning. This chapter will not contain any code, only basic motivation, and ideas.

In the end, we will go through the skill set required to do machine learning in the industry effectively. We will go through some of the Python programming concepts quickly as this book assumes you have working Python knowledge. So, many of you might feel it's inadequate, and there is a lot to Python programming than what is listed here. And that is true. So, I would recommend going through the prerequisite section in the book's preface to get a better Python programming understanding. With that in mind, let us get started!

Structure

In this chapter, we will cover the following topics:

- What is machine learning?
- Some machine learning jargons
- Machine learning definitions
- Why should we use it?
- Types of machine learning
- How much do I need to know to do machine learning?

Objective

After studying this chapter, you should be able to:

- Understand what machine learning is
- Get familiar with some machine learning jargons
- Understand the importance of machine learning in the current time

1.1 What is machine learning?

The way we approach problems in traditional software engineering is as follows:

- We are given a problem. We want to output and are given some constraints.
- Then, we think about taking the input and writing rules to get the output (keeping the constraints satisfied).
- Then, we test our code, find errors, modify our rules, or update the understanding of the problem. In some problems, we change the problem a bit or add new inputs. Thus, it is a flexible approach.

The following is a short diagram of traditional programming:

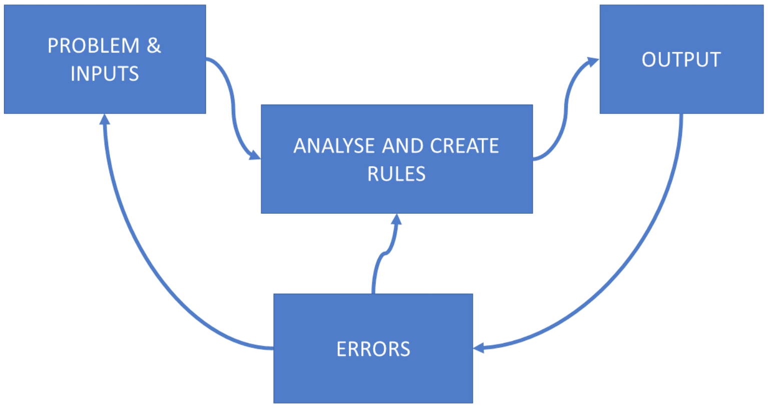


***Figure 1.1**: Traditional programming*

This works out in most cases, and saying this works out is an understatement. We have been solving problems using this approach long before machine learning even came into existence. And it's hardly representative of what programming is by putting a simple diagram of input and output. But for the discussion of how machine learning is different from traditional programming, this definition will suffice. We wanted to emphasize this since many people (I included sometime back) think machine learning is a swiss army knife, and I find people trying to solve problems that are not fit for machine learning. This way of approaching problems is dangerous. On the other hand, traditional programming, done right, can also feel like magic. The beauty is in the flexibility.

Keeping all these in mind, let's understand what machine learning is in the context of traditional programming. Let us think of this with an example. Let's take one of the most used examples in machine learning: a spam filter or classifier. A spam filter should filter out those pesky spam emails from ham (we call the non-spam emails ham). Now, let's do it traditionally. We can analyze the problem and think of inputs first.

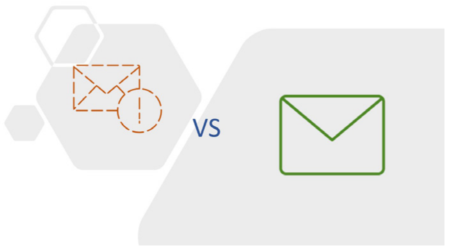


***Figure 1.2**: Spam vs. ham classifier*

How do we categorize a spam email? We can start with a few spam and ham emails. What immediately comes to mind is product promotion emails. And let us think about what input we have; we have emails. They have a lot of information like from, subject, body, and so on. So, we can take those as input; let's say we know all the emails are spam when it comes from a certain email id (to as input). Then we can write a rule, reading the email, parsing To, and then check who sent that email. If it matches with the spammer, then mark it as spam else ham. Voila!

But wait, what if one more email address is also a spammer. What about emails where spammer sometimes sends hams. And we can know that only by reading the content, or, like in the following image, an email from a good domain contains spam in the body. Again, we can create some complex logic to handle it (check for certain keywords, and if those are present, then spam else ham). But when shall we stop? We don't know all the emails. We can't possibly know how all the spam or ham emails look like. We only have a dataset containing some spams and some hams.

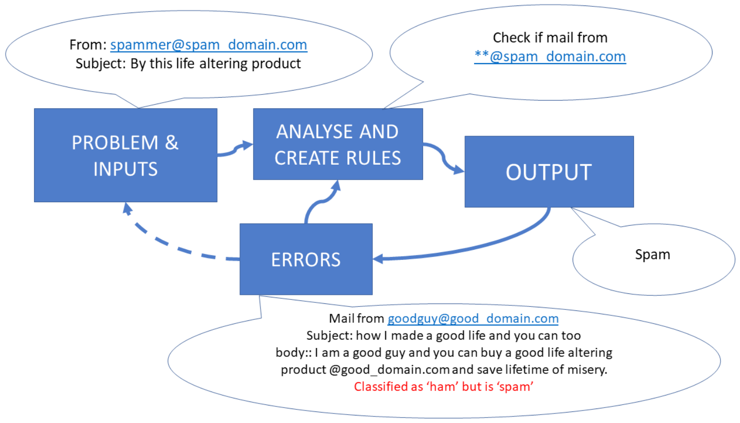

From: spammer@spam_domain.com
Subject: By this life altering product
Check if mail from
**@spam_domain.com
PROBLEM & INPUTS
ANALYSE AND CREATE RULES
OUTPUT
ERRORS
Spam
Mail from goodguy@good_domain.com
Subject: how I made a good life and you can too
body:: I am a good guy and you can buy a good life altering
product @good_domain.com and save lifetime of misery.
Classified as 'ham' but is 'spam'

***Figure 1.3**: Limitation of traditional programming in data-driven problem*

What we can do is, mimic what we do through a **model**. And a lot of you might think that's not possible. A model like this will try to mimic human intelligence, which is so hard to comprehend. But wait, we don't want to do that to solve the problem of spam and ham. And we don't even want to be 100% correct. Let's say, out of 100 spam emails, our model can filter 80 spam, and 20 are marked as ham even if they are spam. We call those **false negatives****. They are falsely identified as ham (negative) even if they are spam (positive). And it's better to have 20 spam emails to delete than 100. That's where **metrics** come in; we set expectations in the user's mind, and we try to optimize that. Optimization is a huge part of machine learning.

With all this, do you understand the difference in the approach? I never said what the model is. In machine learning, well define a **crude**** model and feed a bunch of data. Then we try and minimize the metric, and the crude model **auto-adjusts/learns**** to do that. This is a beginner example of machine learning. There are many other types. But it's a start.

The following is a diagram of how machine learning works in the crudest sense (we will update this diagram eventually):

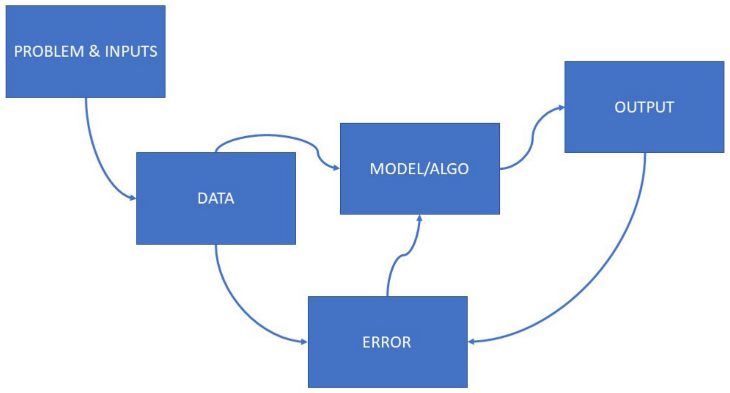


***Figure 1.4**: A simple machine learning process*

Note: ** It's not illegal to call it false positives. We can take ham (positive) and spam (negatives). But in general, we take data of interest as positive, which in our case is spam emails. There are other reasons too, but we will discuss more on it later.

****We will deep dive into what crude means here, but crude means general here. A model that can be used for a lot of similar problems.**

****Some machine learning jargons use 'learn' instead of 'auto adjusts'; I find auto adjust closer to what we do. Although it's close to how learning works, I find it ludicrous to compare it to human learning. But we will use learn from now on since standards are important.**

1.2 Some machine learning jargons

The preceding spam ham problem gave us few machine learning jargon; throughout this book, we will go through many jargons, which will make your life easier in studying machine learning. These jargons are something every data scientist/ machine learning engineer should know.

Remember, we learned machine learning model auto-adjusts to input data for your specific output. We call that input data a "*training set*." This is the data that our model or algorithm uses to tweak the knobs and dials in the model to get the desired output. We call those knobs and dials as "*parameters*" of a model. And such a model is aptly called the "*parametric model*." And there are models where there are no knobs or dials. We call those "*non-parametric models*." We will go through some non-parametric models in the upcoming chapters.

Since we need to convince the user (in our problem context, the user is the email user) that our model is useful, we need to give them a "*metric*" on which they can judge.

In our case, we talked about false negatives. Maybe the user is also interested in how many of our models got right and wrong. We can use a metric called "*accuracy*," defined by how many data points our model correctly identified (spam as spam, ham as ham) by the real examples.

But wait, if we show this accuracy for the training set, then our model can cheat to memorize. There is nothing wrong if you know all your future emails will match exactly with your dataset, which is impossible. So, we might keep some data on which we don't tweak the parameters or train the model. We call that part of data a "*test set*." Now additionally, it's called a "*validation set*" or "*hold-out set*," too. But there is a subtle difference between a test set and a validation set. We will go through that when we learn about hyperparameter tuning. All of these may sound like alien words for a beginner in machine learning. But trust me, they are simple ideas that work.

Now in this current example, we need to tweak the model to give us the output we want. And since we are not relying on rules, we need to tell our model how it performs when tweaking the parameters. That's where "*loss function*" comes into the picture. We start with an initial set of parameters and output; then, we compare it with our desired output through a loss function. Then, we will try and minimize that loss function by changing the parameters.

What is the desired output in our example? How to get that? Our example calling our desired output for spam emails is 'spam' and 'ham' vice versa. So, someone needs to collect some historical emails and tag them manually for desired output to learn the machine learning model. This process is called "*annotation*." This type of machine learning is called "*supervised machine learning*," where you need to tell the model what you want apart from the input. Many of you might think this is useless as you need to tell the model earlier, but it is not; the way machine learning works is called "representation learning," the model understands the hidden rules/representations that cause the input to produce the input-output. So even if new emails come, which our model is not trained with, it will classify the email.

The actual output is dirty (because of noise), and the good model will be robust against noise. For example, our spam filter problem can be noisy during the annotation process; the annotator reads a spam email and marks it as ham due to human error. There are many ways noise can occur in the dataset. We will learn about those in detail.

1.3 Machine learning definition

I love to provide definitions after I explain a concept through examples, which I find is easier. As we have already gone through one example, it is time for some definitions of machine learning:

"Machine learning is the field of study that gives computers the ability to learn without explicitly programmed."

-- *Arthur Samuel, 1959*

This is self-explanatory. Unlike traditional programming, where we think about rules and code it, we make the computers "learn" by examples with machine learning.

Another more engineered definition,

*"A computer program is said to learn from experience **E** concerning some task **T** and some performance measure **P**, if its performance on **T**, as measured by **P**, improves with experience **E**."*

-- *Tom Mitchell*, 1997

This one is more specific and introduces a performance measure P, which is the loss function (not the metric), and experience E is a fancy way of saying learning from data. For example, t in our case was creating a spam ham filter. So, machine learning is a computer program that uses *training data E* to auto-adjust *parameters* to a task T, which we can measure by loss *function P*.

With definition and one example, this concludes a brief introduction to what machine learning is and how it's useful, but there is a lot more to machine learning than to create a simple spam-ham filter. In the next section, we will learn how machine learning helps solve complex problems in the industry and why we should use it.

1.4 How and why machine learning helps?

Why use machine learning when traditional programming has solved so many business problems? We will discuss how it helps and why machine learning is thriving below:

- There are specific use cases like the spam filter, where doing traditional programming is hard. Also, the real use of machine learning, that is, cognitive problems, such as image recognition, speech processing, **Natural Language Processing** (**NLP**), and so on. These tasks are extremely data-driven and complex, and solving them using rules would be a nightmare. So, an increase in complexity and data-driven problems are the key areas where machine learning can thrive. For example, we have NLP models that can write entire movie scripts, image processing models that can colorize old black and white images, and so on.
- Another driving factor of machine learning is the boom in data. The generation of data is exponential. As per an estimate by Statista[1], around 41 zettabytes of data are created in 2019 itself. To put that into perspective, if you watch a full HD movie (1024p, 2 hours approx.) with Netflix, it takes about 8GB of

data. So, in 2018, the data created is equivalent to ~5000 billion such movies. That's a lot of data. And as I mentioned earlier, machine learning problems are data-driven problems. So, it helps *generalize* the models a lot better. *Generalization* in machine learning signifies how the model performs to new unseen data; that is how general the model is to perform even with various examples ridden with noise. And this also gives another key advantage for machine learning over human learning; we can't comprehend data at this scale, even in GBs, let alone zettabytes. So, machine learning in certain use cases of big data helps humans learn or infer. For example, machine learning can let us see hidden *dependencies/correlations* in seemingly unrelated data.

One example I can think of is the beer and diaper correlation story/urban legend. As per the story, Wal-Mart, the world's leading retail chain, supposedly found a correlation between beer and diaper sales on Friday evenings using their transactional data. This kind of learning of association among products from transaction data is called association rule learning/mining. The story suggests that young men take the last dash to take beers on Friday evening and their wives ask them to buy diapers for their child. According to the story, Wal-Mart exploited this association and placed two of these products together. This created a funny meme of kids holding beer bottles. Although this story is said to be fake, association rule mining is true. You can see that during Amazon's recommendations on products bought together. And there are use cases in genetic engineering where scientists use machine learning to identify genes associated with dominant disorders. You can read more on this in the paper titled "*DOMINO: Using Machine Learning to Predict Genes Associated with Dominant Disorders*" by *Mathieu Quinodoz et al.* [2].

- Improvement and accessibility in computation is another driving factor of machine learning. We have a lot of computation power nowadays. Also, they are cheaper. We can find a beefy GPU nowadays cheap. And machine learning code has the potential for parallel processing and taking advantage of a high number of cores present in GPUs. Even you can get a shared GPU for free (even TPUs) using Google Colab. So, without having a lot of high computation infrastructure settings, you can still do machine learning.

Now let's go through a few of the use cases that we use in our day-to-day life and understand how they work. First, we will go through Google Smart Compose that auto-completes the emails were written by a user, and then Netflix recommendations.

1.4.1 How Gmail autocompletes your emails? – Smart Compose

Gmail Smart Compose is a feature that provides sentence completion suggestions when writing an email. It increases user experience as the suggestions are rich, personalized, context-dependent, and even understand holidays and location details. It came after Google launched the Smart Reply feature, which was an overall email response prediction. But smart compose is way more powerful, and it suggests predictions/completions in real-time. With each keystroke, the machine learning model provides you with a suggestion, which is useful.

For example, Thank you for ___. The next set of sentences can be "*your email.*"

Now we will not go through the complete details of how it works, not at this moment, but we will go through some of the key concepts. So how did they do it? If you have used it, then you must appreciate the power it possesses.

The way such a model works is by predicting the next word, given a word sequence. Such a model in NLP is called the **language model**.

Then using this language model output, we create a layer that can generate a sequence of words. (In reality, we don't take the exact output, but something called an **embedding** which we will learn more in detail). With this given set of words, we would be predicting the next set of words. We call the set of words a sequence, and the model is also called sequence-to-sequence modeling.

In the previous example we had of spam-ham classifier, the user needs to provide the actual output, a.k.a. the ground truth. But in this task, if we have several emails (Gmail has a lot of user email data), we don't need to annotate since the next word is already there in the model. This type of machine learning is called a self-supervised machine learning problem since the ground truth is automatically identified from input data. This is a great thing. Since annotation is the actual bottleneck in any machine learning problem. Manual annotation is tedious and requires a whole lot of time and manpower to do. But for a language model with no need for manual annotation and with a company like Google, having nearly unlimited computation power and big data, they can achieve utterly amazing results.

Few things to keep in mind is that training a language model is not that straightforward task. And we do more stuff after training the language model to create a sequence-to-sequence model too. We will go through it in detail in section 2 of this book, where we will go through an industry use case on NLP. So, we will be stopping here, but there is a lot of depth to how the smart compose model actually works; a curious reader with prior knowledge can read a paper titled "*Gmail Smart Compose: Real-Time Assisted Writing*" by *MiaXuChen et al. [3].*

We will understand three key things required for any deep learning model (not just NLP, any deep learning model) with this example.

- **Data**

  Data is the first thing that comes when you start any machine learning problem. But, first, we need to figure out where to get it. What will be the input and what will be our required output, and so on? The following is how we approach the data in the preceding problem:

- o In Gmail Smart Compose, data is user emails. But they also include previous emails, subjects so that the model will be more context-aware. In addition, they add date and time (which provides suggestions like good morning, happy new year, and so on, and they also add the locale of the user. The machine learning practitioner adds all these new inputs/features.
- o This step of enriching the existing data is called **feature engineering**, and with carefully added features, a machine learning model can work wonders. Unfortunately, this is also the most non-automatable part of machine learning, although few AutoML libraries can create features for you. But they can't create such rich and domain-specific features yet. So even though it sounds like a problem, this is the step where your domain knowledge which comes from years of work in the field, can help you provide great features.

- **Model Architecture**
  - o For this example, a language model is created, as previously discussed. The final model generates a sequence of words and takes a sequence of words as input, thereby named **sequence-to-sequence modeling**.
- **Loss Function**
  - o We will talk more about this later since the loss function is complex and needs many vocabularies to explain. But this is the function our model minimizes while training.

1.4.2 Do not overuse machine learning

With all these functionalities and machine learning magic, we need to be extra careful when applying machine learning in the industry. This is because machine learning problems are different from traditional problems, and machine learning is time-consuming too.

- Given a problem, try not to think if it can be solved using machine learning first. Instead, try thinking if I have enough data and if the problem can be solved using few business rules.
- If not, and there is scope for machine learning, try and check for AI services available with **Amazon Web Services** (**AWS**), **Google Cloud Platform** (**GCP**), Microsoft Azure, and so on. But, again, there are a lot of these readymade services available which take care of the infrastructure and scaling; all you need to do is API calls.

- Sometimes you train a custom model with these service providers too. It's easier to maintain, reliable, and scales, which is important for industry problems.
- Even after this, if you feel your problem is specific and no service providers provide readymade solutions, try AutoML first. Then, we will go through few AutoML examples. AutoML lets you create ML models easily and quickly and mostly do the tuning themselves.
- Finally, after exhausting all these options, if you still feel you need your models, try and do it over a cloud VM or container. Again, those are easier to maintain, they are reliable, and they scale.
- But some companies don't trust cloud providers (their data needs to be secure and in-house) and do all of this in-house. In that case, be aware of the CI/CD requirement and production needs. Allocate time and resource for that also and not just model building.

The following is an image of a process that I recommend when you come across any business problem:

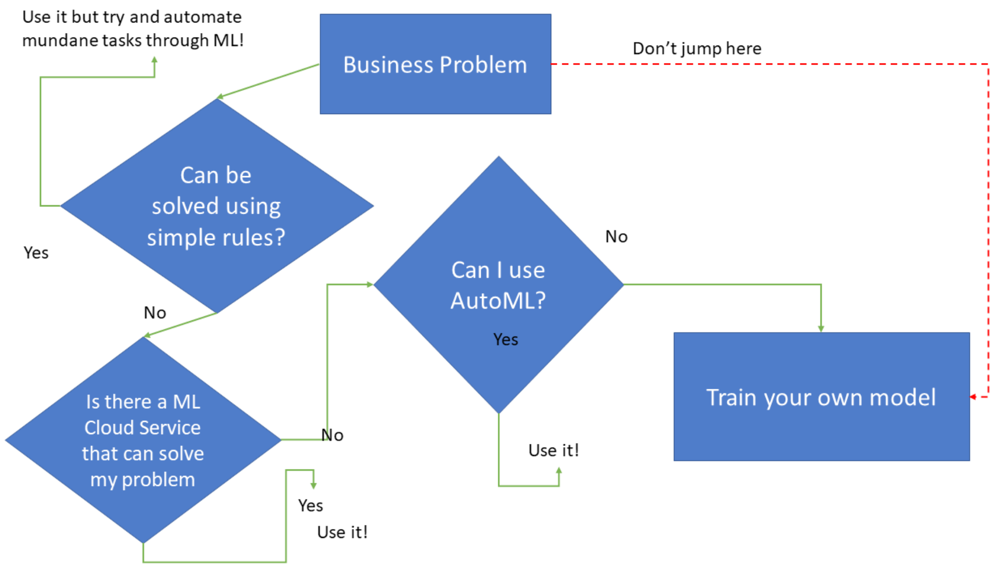

Use it but try and automate mundane tasks through ML!
Business Problem
Don't jump here
Can be solved using simple rules?
Yes
No
Can I use AutoML?
No
Yes
Train your own model
Is there a ML Cloud Service that can solve my problem
No
Use it!
Yes
Use it!

***Figure 1.5**: Approaching an industry problem*

1.5 How much do I need to know to do machine learning?

As mentioned in the prerequisite section, programming knowledge is required. And by that, We mean working knowledge. Apart from that, some knowledge in statistics, linear algebra is needed, and we will learn it throughout the book. So, you don't need to have any prior knowledge of that. Also, for interested readers, the Appendix contains crash courses in both statistics and linear algebra. I will also list a few tips on experimentation and research, as much of machine learning is experimentation and requires some discipline not to be lost.

1.5.1 Short introduction to Python programming

Here is a short introduction to key Python ideas that we will frequently use. This is not a Python tutorial, just a few key concepts you need to brush up on that we will use a lot in this book.

- **Functions**

  Functions are important for our use. We will write a lot of functions throughout our machine learning journey. This allows our code to be more reusable. The following is an example of a simple Python function:

```
def print_dict(somedict: dict) -> None:
  """Prints key -> value of dictionary"""
  for key, value in somedict.items():
       print(f"{key} -> {value}")


   print_dict({a: 1, b: 2, c: 3})
   a -> 1
   b -> 2
   c -> 3
```

  Few things to consider when writing functions:

- Try and make them short. Functions are supposed to be simple and do one task. Try not to write a lot of code in one function. Also, no one writes their code in a single go. Try not to overthink. Write your function and refactor it later when possible.
- Add documentation, write function annotations, and doc string. Don't write huge comments explaining exactly how the code runs. Some coders think it helps in the maintenance, but it just makes the code less readable. Your code should be readable; you don't need external comments explaining everything in the code.
- Use args and kwargs wisely. Heavy use might make your life easier when coding it for the first time, but it takes away the effectiveness during debugging.

- **List comprehension**

  It makes the code more readable and pythonic. Don't overuse. Overusing raises complexity a lot. Example code:

  ```
  [i for i in range(20) if i%2==0] # this gives all even numbers
  from 0-19 in a single line of code.
  ```
- **Dictionaries**

  Dictionaries are handy and constantly used in the machine learning field. Go through the following methods and familiarize yourself:
- **items()**
- **values()**
- **keys()**
- **del**
- **for key in dict**
- **Defaultdict**
  - o Do not raise KeyError if the key not present. Useful when you are appending values to a key that does not exist inside a loop.
- **Classes**
  - o Classes will help us write object-oriented code. First, familiarize yourself with dunder methods, static methods, and class methods.
- **Context managers**
  - o Context managers are an especially useful feature; these let you allocate and release resources when you want.
  - o For example, when we want to read something from a file, we first need to open it, and after reading, we need to close it. Similarly, we need to connect with the database, and then we need to close it.
  - o For the file open, we use a keyword that allows us to do all these with the following code:

    ```
    with open('some_file', 'w') as opened_file:
       opened_file.write('Hello World!')
    ```
  - o Using context managers allows you to reduce errors pertaining to the allocation and release of resources. Also, they give you control to execute a piece of code upon creation and deletion of an object which is very handy.
- **Debugging**
  - o Debugging, in my view, is the most important skill a software engineer has. And you should be comfortable in debugging code. Python has a **`pdb`** package. Notebooks have a **`%debug`** magic function. And if you use

a text editor like *VSCode*, *Sublime*, or *Atom*, you can use the debugging feature that comes with each of these text editors. We will have more on debugging, notebooks, and text editors in the Appendix.

Apart from all these, there is an appendix on using notebooks and creating environments for machine learning that you can check.

1.5.2 Tips and tricks for experimentation

Machine learning requires a lot of experimentation, and with every experimentation, discipline and standards are necessary. Unfortunately, since this is a new field, there are no agreed-upon standards, at least not everywhere. So, for now, the following are few things that are a must for any standard machine learning experimentation process. This will make your life way easier.

- **Standard folder structure**

  Do you have a standard folder structure? You can follow a few packages to do that, for example - The Cookiecutter project (**https://drivendata.github.io/cookiecutter-data-science/**). You can have your structure. But make sure you have a similar structure to Cookiecutter. This is barebones, your project must-have. We will also go through some version management tools and production deployment in the upcoming chapters.

- **Iterative**

  Like every experiment, machine learning is also very iterative. So don't try to make the best possible model/architecture/data on the first try. Better to create something that works quickly. And then play around with it. Since in machine learning, there are no agreed-upon processes.

- **Logging**

  Logging is important and often overlooked. We need to keep track of many things like model hyperparameters, inference, performance, dataset, and features. And we do need a standard approach to log all these so that we can track things easily. We will go through some of the advanced tools during the production deployment chapter.

- **Visualization**

  Here are few key visualization libraries that are needed for this. Matplotlib (subplots), Plotly (really powerful and interactive), and Seaborn. These three are more than enough. We have an appendix on all these libraries for you to explore.

Conclusion

With this, we conclude the first chapter on machine learning. To summarize, we came across many machine learning jargons. That is to familiarize you with machine learning vocabulary. We learned how the process works as a bird's eye view using a spam ham filter on a supervised machine learning problem. Similarly, we came across a self-supervised machine learning problem through Gmail Smart Compose. In the next chapter, we will go through one of the important and underappreciated aspects of machine learning: problem formulation. Stay tuned.

Points to remember

The following are a few points to remember for machine learning problems:

- Machine learning versus traditional programming, unlike traditional programming, we let the model learn how to come up with the output
- Most cognitive problems can be solved with machine learning rather than traditional programming since the problem is complex
- Machine learning has a lot of experimentation, logging, and project structure is important for lower maintenance
- Try not to apply machine learning blindly; most problems can be solved using some carefully written rules

MCQ

1. **Pick the supervised machine learning problem**
   a) Sentiment analysis
   b) Clustering
   c) AlphaGo
   d) Path finding

2. **Which one is a metric in classification?**
   a) F1 score
   b) RMSE
   c) Mean
   d) P-value

3. **What is correlation?**
    a) How different two features are
    b) Mean difference between two features
    c) A measure of dependency between two features
    d) Metric

4. **Mention the non-parametric model**
    a) K Nearest Neighbour
    b) Convolutional Neural Network
    c) Long Short-Term Memory
    d) Linear Regression

Answers to MCQ

1. Answer a; Sentiment Analysis
2. Answer a; F1 score
3. Answer c; A measure of dependency between two features
4. Answer a; KNN model is a non-parametric model; it uses neighborhood information to classify data rather than creating a model and tweaking its parameters

Questions

1. Identify few supervised machine learning problems in your field.
2. Write few metrics in classification
3. Create a program to classify spam versus ham email using just rules. Write down its limitations.

Key Terms

1. Supervised machine learning
2. Representation learning
3. Feature
4. Metric
5. Desired output

Chapter 2 Problem Formulation in Machine Learning

If someone asks about the most difficult step in any machine learning process, we will say problem formulation. This step not only requires an analytical mindset but also requires experience in solving machine learning problems. It's one thing to work on readymade problems in *Kaggle*, *AnalyticsVidya,* and so on, with curated datasets and well-defined problem descriptions, to formulate an actual machine learning problem with messy real-world data and vague expectations. This step also requires business acumen, but it's there in any problem solving, so this is not a blocker to formulate a problem. And when you work in a specific field for a long, you kind of develop that intuition in the first place. Since this book is more towards machine learning practice in the industry, we will go through a few tips and tricks to approach a machine learning problem. We will go through few checklists you should go through when approaching a problem with a machine learning solution mindset.

In this chapter, we will discuss if a problem warrants machine learning in the first place, what are key things to check to answer this question. We will understand the complexities of data acquisition even before starting to collect the data. We will understand how metrics translate to business values. I have seen many machine learning projects suffer only because they didn't spend much time creating a proper metric.

This chapter will provide an initial set of tricks that can speed up this process by standardizing. Keeping these in mind, let's start understanding the first step to any machine learning solution, the problem formulation.

Structure

In this chapter, we will cover the following topics:

- Checking if a problem needs machine learning to solve
- Understanding data requirements
- Can we allow error?
- Validation metrics versus expected outcome versus business value addition

Objective

After studying this chapter, you should be able to:

- Understand where not to apply machine learning
- Understand the need for new and clean data
- Understand fuzziness in machine learning
- Understand how to design a proper validation metric for business value addition

2.1 Checking if a problem needs machine learning to solve

In *Chapter 1, Introduction to Machine Learning*, *figure 1.5*, we discussed, given a business problem, we should first look if the problem can be solved using rules in the first place. This gives us a clue about what problems are good for machine learning. Machine learning, especially deep learning problems, performs wonders when a problem is complicated enough. For example, we went through our spam-ham filter in the first chapter. It's really difficult to categorize an email as spam or ham based on rules, as we discussed earlier, if we go through some simple rules like checking certain domains or looking for keywords (product placement and so on). If we are lenient in our rules, we might miss some spam emails that we called false negatives, and if our rules are extremely stringent, we might miss good emails, causing **false positives**. So, this problem requires a lot of complicated rules to design and warrants a machine learning solution.

One more field where machine learning is heavily used is cognitive services, vision, speech, and natural language processing. The reason is that these problems are complicated for a computer program to solve. Let's go through a few use cases in each of these fields to understand the complexity and requirement of the machine learning solution.

- **Computer Vision**

  Computer Vision deals with problems pertaining to vision-related tasks, for example, classifying images, identifying objects in an image, understanding video, and so on. The following are a few specific examples where machine learning can be applied.

  o **Face detection**

  If you have not updated Facebook's privacy settings, your image is used to train the face detection model. The requirement is simple, given a set of example photos containing people's faces, the algorithm should tag who is who. And Facebook uses this model in its Tag Suggestions feature. This feature auto tags your friends in a newly uploaded picture. And asks you if you want to edit, keep or remove the tag. The algorithm uses facial recognition software to do this.

  Now imagine trying to do this through a traditional programming approach. Where to even start? We can create some landmark points** for each face. And then identify the Euclidian distance between them and a new image. The idea is that the distance would be small for the same person than for a different person. So, we will set up a threshold for the distance and check if the new image crosses that distance.

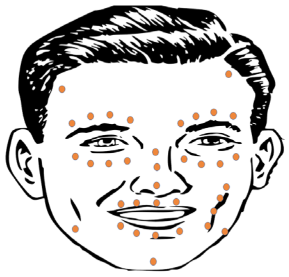


Figure 2.1: *Face Recognition using Landmarks*

  But this approach is not that robust (remember the rules we created for our spam-ham filter). There might be two different people having similar landmark points, and our threshold might fail. And the scenario where tagging your friend is one thing (you may choose to edit/ remove the suggestion here), but imagine a facial recognition software like Windows Hello that uses your face to open your desktop; there, we can't afford such problems. So, this warrants a machine learning solution. And since you are already aware, we are using this in multiple applications (opening your phone with your face, Facebook tagging, and so on.) that machine learning works in this scenario.

- **Object detection**

  Object detection is identifying multiple objects in one image. It's different from simple image classification, where the whole image is characterized as a single class. One example is the revolutionary employee-less shop called *Amazon Go*. It's a shop where a set of carefully placed cameras replace the need for human shopkeepers (well, for the context of object detection, I am hugely oversimplifying). So, when a buyer picks up an item from the shelf, these cameras understand which item he/she has picked.

  As the shelf contains multiple such items in a single frame, object detection is necessary.

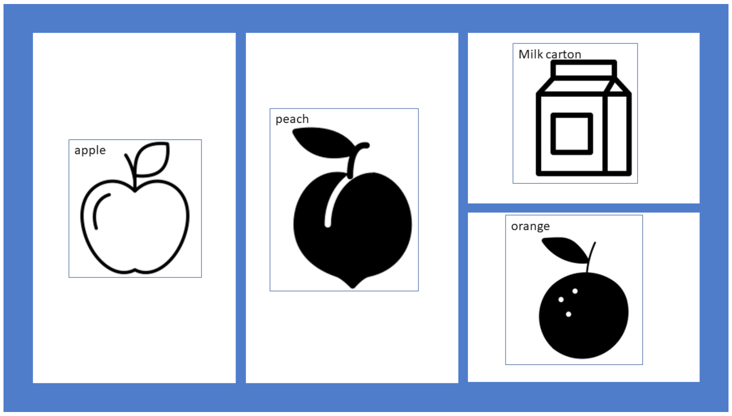


Figure 2.2*: Object detection*

  Now imagine doing it through traditional means. We need first to find a way to *localize* the items on the shelf. Then each such item needs to be classified, taking some features. This is so difficult to do with traditional programming. Hence it warrants machine learning.

- **Medical image processing**

  Medical image processing is another field where there is a lot of potential for machine learning. We have X-ray images, MRI images, and so on. Again, we need to check if there is complexity—for example, X-ray images of lung cancer patients. We need to classify it as cancer or not. Doing this through traditional programming would be difficult. Tumors look quite different from case to case. You could not write a single all-encompassing rule that can classify all tumor images. So, we use image classification here to identify.

As we went through all these examples, we could see that machine learning has many potentials for image processing since the problem is so complex. But why so? The reason is extremely high dimensional data (images contain many pixels) and fuzziness around data. Look at these when you are trying to hunt for machine learning problems.

- **Natural Language Processing (NLP)**

  Natural Language Processing is the field of artificial intelligence that deals with any form of language processing, for example, sentiment analysis for movie reviews (written in text), translation from one language to another, summarizing news, and so on. The following are a few examples of NLP:

  - **Machine translation**

    Machine translation is one of the hardest natural language processing problems. Languages are difficult. Some are written right to left and some left to right. Some contain many characters, start differently, end differently, have context-dependent words, and so on. So, imagine identifying a language and converting one language to another. This makes it an ideal candidate for machine learning.

    One way you can do it traditionally is by memorizing word to word, and memorizing some phrases, and so on. This works to some extent but doesn't provide quality as Google Translate does.

  - **Summarization**

    Suppose you have used Inshorts, an app that gives you news in 60 words. Imagine what a huge customer experience improvement it is, the ability to consume news in only 60 words or less. This introduces a widely popular and equally complex problem of text summarization.

    Creating a bad summary is easy; you can look for few noun phrases or important keywords and exclude all other sentences. But that is not a good summarization. A good summarization is context-aware, co-reference aware.

    Co-reference is where you refer to a person or object indirectly; for example, *Samir is a criminal. His criminal activities involve an organized crime in India, USA*. Here "*His*" refers to Samir's).

    Creating a summarization algorithm that understands all these to extract valuable sentences and exclude useless ones is difficult. This kind of summarization where we extract good sentences is called **extractive summarization,** and that's not how we even summarize. Humans read a particular text and understand it and then summarize based on prior knowledge and context. This kind of summary is called

abstractive summarization. So, summarization is an area where machine learning can be applied, creating a lot of value.

There are other examples of *question answering, image captioning, intent-entity identification, and sentiment analysis* in natural language processing that are not included in this discussion, but NLP is a vast field with many use cases.

With the examples on Computer Vision and Natural Language processing, one key takeaway is that for cognitive problems, enough complexity is there to warrant machine learning because cognitive problems are complex.

The analytics field is a bit tricky to identify for machine learning; there is a lot of problem in analytics that looks complicated outside but runs on a bunch of excel rules. But one key thing to look for is fuzziness. Suppose there is noise involved, data coming from some sensors, data containing human decisions. These are clues for machine learning problems.

Also, we have not even touched the surface of possible use cases in machine learning. There are a lot of use cases. And I would suggest going through as much as you can. This will build an intuition for applying machine learning. You would be able to identify problems much easier if you already went through many industry use cases.

2.2 Understanding data requirements

Even if we convinced ourselves that a problem is complex enough to warrant machine learning, we need to be extra careful with understanding data requirements. To reiterate, machine learning problems are data-dependent problems. The following are a few tips for identifying if sufficient data is there for a machine learning problem:

- **Look for clean data**

  I have seen business problems with vast amounts of data (big data), but they are dirty with a lot of noise and disjoint tables, different formats, and so on. Although it's not a showstopper for machine learning, machine learning models are data-dependent. And no matter how much data we feed to a machine learning model, if the data is bad, the model will be bad. We have a saying "*garbage in, garbage out*" in machine learning. But the real difficulty is how to know if the data is garbage? We have business use cases where data looks bad with 1000 dimensions but provides great insights. There is no single answer to this. Machine learning also has a trial-and-error aspect. But we can be structured when it comes to doing trial and error. One suggestion to decide if data is good is to look for correlations, visualizing, etc. The idea is bad data is an uncorrelated mess, where we can't derive any meaningful insight, and the best way to do that is through visualizing. But there is a risk to that too. We are pattern-finding machines. We can introduce bias by looking very closely into the data. That's the reason we have a test set. But some visualizations don't hurt much. We can also look for recent data. Recent data is good for business value. Trends change, so training a model on 10-year-old data seldom helps.

- **Self-supervised**

  When we went through the example of face detection in our Facebook Tag Suggestions feature, the data availability is huge. Facebook has many images, and the cherry on the cake is that labels are also there. For example, the Facebook database contains my username and pictures already, so there is no need to tag them manually. Also, we gave an example of language models, where we are trying to predict the next word, thereby automatically creating labels. We called this kind of problem self-supervised, where we can create labels automatically.

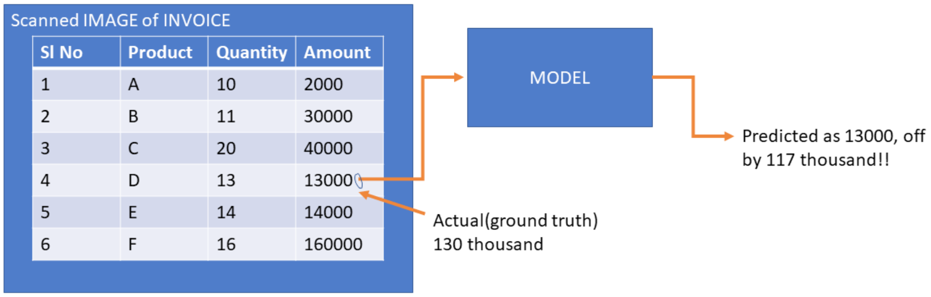
  And it need not be that straightforward too. Imagine Google search rank results; Google can measure the search quality simply by calculating click-through rates. So, look for these if you find a lot of data having potential for auto-tagging; it's extremely good for training models.

2.3 Can we allow error?

I have worked on an automated invoice processing machine learning solution during my early days of machine learning. The problem description is simple; we have many different types of scanned invoices (images in pdf) present in purchase orders. We needed to provide how many items each product is made and the total order amount. Some contain bordered tables; some contain borderless ones; for some, there are merged columns in tables. There are invoices in different languages, invoices of multiple pages where a single table is there in multiple pages, and so on. The problem was complex enough for machine learning. We had a lot of historical data as well as recent data for testing. It was a good problem to solve.

But we made a mistake; we were so focused on applying machine learning that we misunderstood one of its key limitations—the fuzziness and error allowance. Machine learning solutions are inherently probabilistic. We will learn more about this when we will build some in the upcoming chapters. But what we mean is, we define a validation metric like accuracy or mean square error, and so on.

Machine learning models make mistakes, and we also need to understand their impact on them. For example, in our problem, as our input is scanned images and sometimes, they are messy, our model sometimes makes an error by missing a '0' (it made other problems too) in the final count. Imagine the impact of that. Suppose your purchase order has a 1 million total amount, and you missed a zero. It will be 100 thousand, which is ten times less. The following image shows how much impactful a single mistake on an invoice sheet is:



Scanned IMAGE of INVOICE

Sl No	Product	Quantity	Amount
1	A	10	2000
2	B	11	30000
3	C	20	40000
4	D	13	13000
5	E	14	14000
6	F	16	160000

MODEL
Predicted as 13000, off by 117 thousand!!
Actual(ground truth) 130 thousand

Figure 2.3: *Impact of an error*

Can we allow such an error, then? No. So, be careful of problems where you can't allow an error. It's not an ideal problem for machine learning if you can't allow error.

So, shall we abandon such problems? The answer is 'No' still. What we did in our case is changed the expectation. We understood that once a cashier receives these scanned pdfs, he does the painful task of creating an excel sheet. And we simplified it by automatically filling it and then asking the cashier to check if it's correct. It doesn't automate the process fully as we expected, but it reduced the human effort. And we also introduced confidence by color-coding cells. We marked cells green where our model is most confident and red where it's least. We introduced thresholds for confidence, and if it doesn't cross that, we didn't fill anything and leave that to the cashier.

So, if possible, modify your expectation to fit machine learning if it's complex enough and data sufficiency is there. But don't forget to take error impact into account. Ask the business heads if they can allow error and how much they can allow before building your model.

2.4 Validation metrics versus expected outcome versus business value addition

We came across one of the validation metrics in chapter one called accuracy for supervised machine learning problem (classification of spam-ham emails). The following is a more formal description of accuracy:

		Actual	
		SPAM	HAM
Predicted	SPAM	15	3
	HAM	10	100

Figure 2.4: *Confusion matrix*

The above is called a confusion matrix for a supervised classification problem. Let me break it down for you:

- **True Positives (TP)**: The top-right cell specifies actual or ground truth, these are the numbers that are there, and the bottom left cell specifies predicted, and these are what your model predicts. So, the first value, 15, means that these 15 are spam emails and our model also correctly identifies them as spam emails.
- **True Negatives (TN)**: Similarly, 100 is the value of ham emails or negatives, and our model is also predicted as negatives, thereby called **True Negatives**.
- **False Positives (FP)**: The second row in the first column, that is, 3, are ham emails incorrectly identified as spams(positives), so these are called false positives. These are also called type-1 errors.
- **False Negatives (FN)**: Finally, the value 10 is the spam emails but is marked as ham by our model. These are false negatives, and we came across them earlier in chapter Introduction of Machine learning. These are also called **type 2 errors**.

So total ground truth positives or spam emails in our case is 25, which is TP + FN. And total ground truth negatives are 103, which is TN+FP. Now think about the performance of this model in terms of identifying spams versus hams. Yes, it made little mistakes pertaining to identifying ham emails, only 3 out of 103. But it made a lot of mistakes when identifying spam emails. 15 out of 25! The question is, does accuracy give us this insight?

- **Accuracy**: Accuracy is a simple metric; as discussed before, how many of our models correctly identified out of N examples? Our model identified 115 correctly *(TN+TP).* Out of 128 examples *(TP+FP+TN+FN)*. So, it's around ~90% accurate. But as discussed, our model did bad identifying spam emails; only 15 out of 25 are correct! Which is only 60% correct. This also makes sense in real life; we will have many good emails and fewer spam emails (or vice versa if you select a newsletter for every website). Such a dataset is aptly called an imbalanced dataset. So, with imbalanced datasets, accuracy is not a good metric. You can perform well in the class with many examples, and your accuracy will be great even if you perform way worse in minority class examples.
- **Precision**: It is *TP/(TP+FP)*; optimizing this metric will optimize reducing False Positives and increasing. Another way of saying precision is to try to optimize type 1 error.
- **Recall or Sensitivity**: It is *TP/(TP+FN)*; optimizing this metric will optimize reducing False Negative. Another way of saying is Recall tries to optimize type 2 error.

Sometimes our problem requires reducing both type 1 and type 2 errors. And we can do that by taking a harmonic mean of precision and recall. Since harmonic mean has a cool property that, if penalizes lower values heavily. If one of the precision or recall has low values, the harmonic mean will be lower. And this has a name too; it's called F1 score.

The reason for me to introduce all these metrics is to select a metric that translates to your business needs. For example, it's really bad for medical image classification to skip a positive patient than mark a negative patient positive. Because later case, a doctor can check the X-ray image and confirm if it's truly positive. So, for medical, image Recall is a metric often used or F1 score. Similarly, based on your needs, you can choose a better metric. In section 2 of this book, our business use cases, we will go through various metrics and their effectiveness and limitations for each.

2.5 Next steps

Once you are convinced that the problem has enough complexity for machine learning and has enough data to train a model, the following is a checklist that you can prepare to proceed further:

- Validation metric that translates to business value. Initial success range for POC. For example, set up a range for success, 60-70% recall or >85% accuracy. Have a lenient success range for POC but not too lenient.
- Write down all the data sources; how much unstructured and structured data present; how much we need to annotate if any. Write down all the different types of augmentations you can do.
- Get yourself familiar with some domain knowledge without seeing the data. Ask your clients to keep a separate data set for testing (test set) if they can. Make sure that the test set has similar but not the same data.
- Keep a proper structure to data and notebooks for data create folders for external, internal, raw, and processed folders. The proper notebook folder structure will be discussed when we will go through a real-world example.

Conclusion

With this, we conclude the chapter on formulation. To summarize, not every problem can be solved using machine learning, and machine learning problems require a certain level of error tolerance. We should think about data sufficiency and different ways of acquisition even before collecting any data. Logging is just as important as structuring the machine learning project. In the next chapter, we will go through data, the fuel on which the machine learning model runs. We will understand how to acquire data, check its sufficiency, clean, and much more. So, stay tuned.

Points to remember

The following are a few key points when it comes to data in machine learning:

- For ideal machine learning problems, look for cognitive problems where a lot of data is available
- If a problem does not allow any mistakes, it is probably not a good idea to apply machine learning
- Find a metric that gives you good business value

MCQ

1. **Classifying dog and cat images comes under which category?**
   a) Computer Vision
   b) Natural Language Processing
   c) Reinforcement learning
   d) Tabular data analysis

2. **Which of the below formulas define precision?**
   a) TP/(TP+FP)
   b) TP/(TP+FN)
   c) FP/(TP+FP)
   d) TP/(TN+FP)

3. **Which metric is also called sensitivity?**
   a) Recall
   b) Precision
   c) F1 score
   d) AUC ROC

Answers to MCQ

1. a
2. a
3. a

Questions

1. Identify where you can apply machine learning in a human resources department—hints: resume processing, skill clustering, and so on.
2. Define the AUC ROC metric; how is it useful?

Key Terms

1. Computer Vision
2. Sentiment Analysis
3. Text Summarization
4. Machine Translation
5. Precision/Recall/F1 score

Chapter 3
Data Acquisition and Cleaning

In this chapter, we will learn about different techniques to acquire data. We will go through various sources of data. We will also understand how important it is to determine if the data is sufficient for a machine learning project. We will go through various guidelines to follow when looking for data. Many data science projects get stalled due to insufficient data requirements and sufficiency analysis during this stage. Even seasoned data scientists often misjudge the amount of data that is required. Although there is no clear answer to how much data is required and how we will acquire data, it will make your life easier if you follow few guidelines during this phase. We will also understand automation needs during this phase.

In the later section of this chapter, we will go through data augmentation techniques and understand where we can apply and where we should not. We will also go through various data cleaning techniques, but they will not be exhaustive in nature. Finally, we will extend our understanding of the test set and how important it is to create a good test set for any machine learning project. With this in mind, let us begin understanding data, the fuel for any machine learning model.

Structure

In this chapter, we will cover the following topics:

- Data sufficiency and where to get data

- Data augmentation
- The need for automation
- Why is cleaning data necessary?
- Need and importance for a test set

Objective

After studying this chapter, you should be able to:

- How much data is sufficient
- What all augmentation we can do
- Sources of data to look for
- How we can clean data
- How we can create a good test set

3.1 Data sufficiency and where to get data?

The first question you will come across when you start doing machine learning (unless it is a *Kaggle* competition where a dataset is given already) is "*How much data do I need?*".

Let us take an example. Suppose you want to automate a support ticket process. When people use it, you have a product X; they will have a few issues/concerns/queries. You track it using applications such as *JIRA* or *SNOW*. And you want specific tickets to come under a specific bucket. For example, tickets concerning accounts should go to accountants, tickets concerning bugs go to developers, and so on.

When you are hired to automate this, you will ask if I can get some historical data with labels. Since there is a labeling or manual annotation aspect, the client will ask you, "*How much do you need?*" An experienced practitioner will immediately think, "*Well, that depends!*" since there is no single answer. This section will give you some idea of how to negotiate an answer.

The following is a process of finding an answer to this question,

3.1.1 The initial answers

The initial answers are broad and vague. The first answer you can give is, "*That depends; it's always better to have as much data as possible. Also, it is not necessary to provide all the labeled data to create a model. You can share an initial set of data and can share more data iteratively*." This works in many cases. Looking for a perfect dataset is futile. Start with whatever you have or get within a short period. Build some baseline model. Then, iteratively make the algorithm better (not necessarily on the same model, you can change the model too). If "*as much as possible*" does not work, we need to be prepared for the question, "But how much data do *you need to start?*"

Another way to give an initial answer is to create a graph between a number of samples vs. the model's performance (called a learning curve) and then extrapolate. The shape of this curve, ideally, is logistic, which is a fancy way of saying the rate of performance improvement slows down after adding data to a certain extent. I don't recommend this method because you first need a decent amount of data to create an interesting plot that we don't have initially. The next step is to come up with a general estimation of a number of records, which will be discussed next.

3.1.2 The naive "in general"

The follow-up question can be, "*Ok, how many categories you have, how many features or inputs you have.*" This gives you some idea about how much complexity there is. And the more complex the problem is, generally, the more data points you need. Also, if you can use unstructured data in any way, ask for it. For example, we discussed language models and self-supervised machine learning. For these problems, you can ask for all the data to get started.

In general, we can decide the number by three details, the first two are for traditional algorithms, and the third one is for deep learning:

- **X per label**: For example, for traditional machine learning, 2-5k samples for a label, deep learning models, 5-10k samples per label.
- **X per feature**: Like labels, 100-400 samples per feature or input.
- **X per parameters**: For deep learning models, around 10-15 samples per parameter in a model. So, a deep learning model with 10k parameters an estimated 100k-150k records you can ask.

The problem with this is that not all machine learning models or problems require the same number of samples. Then we can start thinking about sample size for specific problems/algorithms.

3.1.3 The specific "in general"

This part is tricky, as this requires some experience in work on some problems. But the way you can approach it is by looking into others' work. For example, if you are working on an image classification problem on X-Ray images, look for prior work. Many machine learning problems are either the same or similar to some earlier work. There you can check their sample size to get a guestimate.

This can also provide you with some useful external data, dramatically reducing the need for data. Just make sure your client allows that, and you can follow license agreements to use such data. We will go through a variety of data sources in this chapter where you can look. We are also going to talk about automating this process.

3.1.4 Role of augmentation and analysis

Since annotation is a huge investment and time-consuming process, generating synthetic data is another aspect of understanding data requirements. We can use specific data augmentation techniques to create more data. During data sufficiency analysis, keep this in mind; understand what augmentation you can use and what you can't. We will go into detail about data augmentation in this chapter too.

Another aspect is analysis; this is where the iterative prospect of machine learning comes into the picture. This part is a bit subtle. For example, if you create a model for image classification and have 'x' data points with 'y' labels. After creating a baseline model, let's say you identify your model performing badly with one of the labels. Then you can do some error analysis (which we will cover in detail during the industry use cases part) to understand why. The analysis often provides valuable insight for what to ask for in the next iteration in terms of data. Error analysis has saved many machine learning projects. But the error analysis comes under the validation part of this book, so we won't dive deeper into that here.

Let's go through some external data sources that could help you.

3.1.5 Data Sources

Now let us go into external sources that can help you with data (for **Proof of Concept** (**POC**) projects where no data is present, these are gold). These sources contain a different type of datasets for problems involving data analytics (tabular data, for example, loan credit score prediction, sale price prediction, and so on, image data for image classification, segmentation, captioning, and text data such as sentiment analysis, named entity detection, and so on.)

- Kaggle (**https://www.kaggle.com/datasets**)

  Kaggle has a huge repository of datasets in every type of machine learning problem. So anytime you are creating a POC where you don't have access to any dataset, this is the first place you look. In addition, many industries post competitions on their dataset here, so this contains a rich ecosystem of data.

- Google Datasets (**https://datasetsearch.research.google.com/**)

  This is an alternative search engine for people who are looking for data. This website provides you with different places to look for a specific dataset. For example, you can search for the keyword in dataset search if you need weather data, and relevant searches will come for that dataset.

- UCI Machine Learning Repository (**https://archive.ics.uci.edu/ml/index.php**)

  This also contains a huge repository of data for various machine learning problems in analytics, image, and text. Many popular datasets, for example, iris, wine, and so on came from here. This repository is also actively updated, and you can find new datasets to play with.

- Microsoft Datasets (**https://msropendata.com/**)

  This is a newer repository but contains rich datasets used by Microsoft research and distributed in the public domain.

  Awesome Public Datasets Collection

  (**https://github.com/awesomedata/awesome-public-datasets**)

  This repository contains a huge archive of a different publicly available dataset for analytics, text and speech analysis, and image data.

- Government Datasets

  If you are working on government dataset problems, such as analytics on coronavirus data, look for the following websites; countries post their data on various domains here to look for.

  a) EU Open Data Portal: European Government Datasets (**https://data.europa.eu/euodp/data/dataset**)

  b) US Gov Data (**https://www.data.gov/**)

  c) New Zealand's Government Dataset (**https://catalogue.data.govt.nz/dataset**)

  d) Indian Government Dataset (**https://data.gov.in/**)

  e) Northern Ireland Public Dataset (**https://www.opendatani.gov.uk/**)

- Amazon datasets (**https://registry.opendata.aws/**)

Furthermore, there are various datasets available as default in packages like Scikit-learn, Tensorflow, Fastai, Pytorch, and so on for learning purposes. These are many more use case-specific which we will go through later in this book.

3.2 Data augmentation

Data augmentation is a process of adding synthetic data to the existing dataset to create a more general dataset. This gives the model more data to work with, and it can learn general representations. To give you some intuition for this, let's take an example of buses and cars. In your lifetime you have seen a lot of buses and cars. So even if it's dark outside and you see a bus from the corner of your eye from a long distance, you can still identify a bus from a car with high accuracy. Why is it so?

Because as I said you had seen a lot of buses and cars in your lifetime. And not only that, you have seen it from different angles, different distances, in different lighting conditions, and so on.

Now to do the same with an image classification model that identifies buses from cars, you can collect a lot of such images of cars and buses from various angles, distances, sometimes you see part of a bus, so you need to have cropped images, in different lighting conditions, and so on. This can be a huge task to collect each image for all of these situations. This is where augmentation comes into the picture. All these different conditions can be modeled using some transformations to the original image. For example, to get an upside-down image of a car, you need to flip all y-values in the image, which can be done using a simple equation. And if we add some randomness to this process, we can create a synthetic dataset that contains all these situations. And you don't need to label or take more pictures. Just apply some transformations to the image, and you are good to go.

Two more things make data augmentation special, especially in Computer Vision. First, you can combine a lot of different transformations. For example, rotate 10 degrees, then crop randomly. This gives you a lot of different options to create data. The second is parallelization. You don't need to do anything sequentially here. You can just process a batch of images simultaneously. This makes data augmentation a powerful tool at your disposal to create more general datasets. This is why most libraries like TensorFlow, fastai, and so on have inbuilt functionalities for data augmentation, especially for Computer Vision. For NLP, it's not that common, but we will still go through some ways to create synthetic data.

3.2.1 Image data

Augmentation in image data is quite straightforward. But there are a lot of options, a few of which we will go through. For all these augmentations, you are required to write 1-2 lines of code (both Tensorflow and fastai), and we are not going to go through the detailed code walkthrough at this moment. This is to show you what all augmentations are available for use, and you can play around with these transformations using the following code, give different options, check how your final synthetic images look, and so on. Experiment as much as you can. We will not go through all of the transformations and go through the code for a single transformation: rotation. Rest all code is already available at **https://fastai1.fast.ai/vision.transform.html**. You can refer to that and play around. The following code is just to get you started!

The examples can run in *Google Colab*; you can check the appendix to get a quick guide on using *Google Colab*; it's easy to use.

Loading a sample image:

You can load any sample image; for the following example; we will use this image: **https://www.pexels.com/photo/brown-leaf-3081752/**; after downloading, I resized it into 300x450 in MS. Paint.

To load any file in **`Google Colab`** run following code:

```
from google.colab import files
uploaded = files.upload()
```

This opens a window where you can upload the file you want. Then you can check the file you have uploaded.

```
! ls
```

The output for preceding code, **`pexels-cole-keister-3081752.jpg sample_data`**

Now, let's load the Fastai library to use:

```
from fastai.vision import *
```

The following function opens an image using fastai utility function **`open_image`**:

```
def get_ex(): return open_image('pexels-cole-keister-3081752.jpg')
```

To show the image we uploaded, we can run the following code:

```
get_ex().show()
```

`The` output for this code:

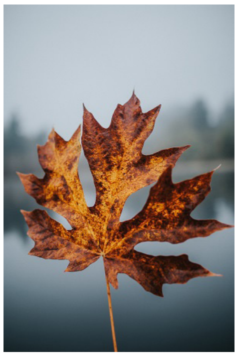


***Figure 3.1**: Showing image in notebook*

We will go through the following transformation:

- Random rotation: The following is the code for random rotation:

  ```
  fig, axs = plt.subplots(1,5,figsize=(12,4))
  rotations = np.linspace(-60,60,5)
  ```

```
random.shuffle(rotations)
for deg, ax in zip(rotations, axs):
    get_ex().rotate(degrees=deg).show(ax=ax,
title=f'degrees={deg}')
```

The output for the preceding code is as follows:

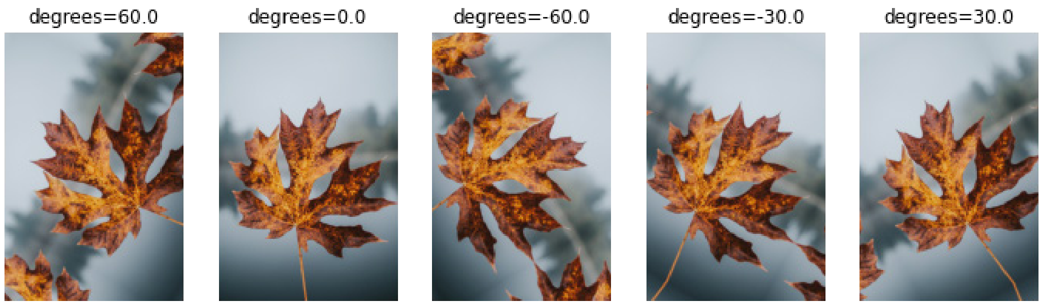


***Figure 3.2**: Visualizing random rotation*

How does this code work?

Now we will go line by line how the code works:

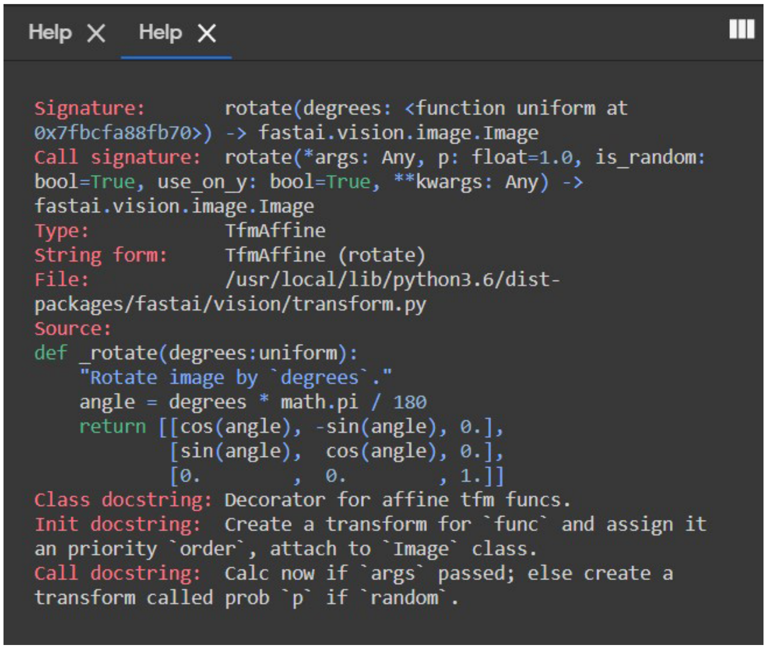
1. First, we create a figure using the matplotlib subplots function to create a single row with five columns.
2. Then we create five different rotations from -60 to 60 using **`numpy linspace`**. Then, we use **`random.shuffle`** to shuffle these values randomly.
3. Finally, using. rotate (**`fastai`** function), we rotate the image and display it.

Here you can see we randomly rotate the original image from -60 degrees to +60 degrees. A rotated image class label does not change (even if you rotate a cat, it remains a cat). But be careful in applying this when your task is localizing an object in an image. That is, given an image, you need to give the coordinates of the object. If you use a rotation to the image in that case, then your coordinate for the object will change. You can handle it by applying the same transformation to the labels also.

One key tip in notebooks is to use '??' after the function like the following code:

```
get_ex().rotate??
```

The output for this code:

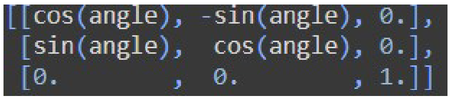


```
Signature:      rotate(degrees: <function uniform at
0x7fbcfa88fb70>) -> fastai.vision.image.Image
Call signature:  rotate(*args: Any, p: float=1.0, is_random:
bool=True, use_on_y: bool=True, **kwargs: Any) ->
fastai.vision.image.Image
Type:           TfmAffine
String form:    TfmAffine (rotate)
File:           /usr/local/lib/python3.6/dist-
packages/fastai/vision/transform.py
Source:
def _rotate(degrees:uniform):
    "Rotate image by `degrees`."
    angle = degrees * math.pi / 180
    return [[cos(angle), -sin(angle), 0.],
            [sin(angle),  cos(angle), 0.],
            [0.        ,  0.        , 1.]]
Class docstring: Decorator for affine tfm funcs.
Init docstring:  Create a transform for `func` and assign it
an priority `order`, attach to `Image` class.
Call docstring:  Calc now if `args` passed; else create a
transform called prob `p` if `random`.
```

Figure 3.3: *Looking into function signature*

This will open documentation right next to the Notebook to check the function's source so that you can quickly understand what the function is and how you can use it with more flexibility. This will also help you debug your code. A full walkthrough of using jupyter notebooks and debugging is there in the appendix for beginner programmers. So for the example of image localization (where we predict the location or coordinates of the object), we can use the parameter '*use_on_y*' to achieve that.

You can also see how the rotation works. If you know matrix multiplications and linear transformations, you can see that it uses the following matrix and multiplies with our original image to rotate it:



```
[[cos(angle), -sin(angle), 0.],
 [sin(angle),  cos(angle), 0.],
 [0.        ,  0.        , 1.]]
```

Figure 3.4: *Rotation matrix*

But it's not necessary to understand all these to experiment, so if you don't get it, it's all right. Just experiment with the functionalities.

Now you can go through the FastAI documentation on image data transformations (**https://fastai1.fast.ai/vision.transform.html**) and play around with the following features:

- Random crop
- Padding - zero padding and reflection padding

- Flip
- Zoom
- Lighting
- Perspective warping
- Resize and crop

As discussed, image augmentation is useful to create synthetic data but be careful in applying any augmentations to the data. Look at all the transformations you can apply and what you cannot apply if you need to add the labels to the labels. All of these you need to keep in mind when doing data augmentation.

3.2.3 Text data

Augmentation in text data is not that common as in images. The reasons are quite straightforward. If you tilt your face 10 degrees, a dog in front of you does not become a cat. But if you jumble up your words or replace a word with something else, you may not sound sane (unless you are Yoda from Star Wars). I know it's an oversimplification at best to compare tilting of a face with image data augmentation, but this is to make a point that text data augmentation is not that trivial. But that doesn't mean we don't have any augmentation for text data; contrary to that, we have interesting solutions for text data augmentation. And with handy libraries like **`nlpaug`**, we can create a lot of good text data augmentation.

In this section, we will use the **`nlpaug`** package to create more text data. Since these techniques require some prior knowledge in natural language processing, such as a bag of word models, TFIDF, word embedding, language models, etc., will give some overall intuition behind these ideas. I won't go into the details. We will go into details of Natural Language Processing when we go through the industry use case on NLP.

Some techniques to create new examples from existing text data are either replacing some characters (to introduce noise), or by replacing some words, or replacing some sentences, and so on. The full set of augmentations can be accessed here (**https://github.com/makcedward/nlpaug/blob/master/example/textual_augmenter.ipynb**); similar to image augmentation, We would suggest going through each augmentation done using the **`nlpaug`** library and play with it.

The following are a few text data augmentation techniques in NLP:

- TFIDF based

  This method is used to preserve keywords when we replace some words in a document or text. The key idea is that we should not replace any keywords. So, first, we need to identify it.

The way important words are identified using the intuition that keywords will be the words that are common in a document. This occurs a lot. But in English, words like is, are, go, and so on (these are called **stopwords**) also occur a lot, but they are not that important. This can be rectified by penalizing words that are in almost all documents.

So, we need a metric that captures frequent words in a document, gives it a higher value, and penalizes if a word is present in many other documents. The metric that does this is a TFIDF metric, Term Frequency Inverse Document Frequency. The formula and how it works is quite straightforward, and we will go through it in the NLP use case later in this book.

- Synonym based

  It's quite straightforward, replace words with synonyms to create more documents, for example:

  ```
  Original:
  ```

  `The` **`quick`** `brown fox jumps` **`over`** `the lazy dog.`

  ```
  Augmented Text:
  ```

  `The` **`speedy`** `brown fox jumps` **`complete`** `the lazy dog.`

- Word-Embedding based

  This type of augmentation is where words are replaced with similar words; way similarity is measured by first finding a numerical vector for the word, then finding similar vectors by finding distance between two vectors. The process to create a numerical vector introduces some contextual information in the vector itself. We will go through word embedding models in detail in the later chapters. The following is an example of augmentation using a word embedding:

  ```
  Original:

  The quick brown fox jumps over the lazy dog

  Augmented Text:

  The quick brown fox jumps Alzeari over the lazy Superintendents dog
  ```

- Back translation

  The back translation method is done in two steps. First, the document is translated into another language and then translated back to English. You might think this will make your document/text the same as the original, but translation is not one-to-one. A single sentence can be translated in different ways, thereby creating different examples. For example,

  ```
  Original Text: 'The quick brown fox jumped over the lazy dog'

  Augmented Text: 'The speedy brown fox jumped over the lazy dog
  ```

This is not an exhaustive list of text augmentation techniques; there are many different methods we did not discuss; I would suggest similar to the Computer Vision task, the reader should experiment with text data augmentation.

Task-specific augmentation

Like augmentation in image data, we need to perform task-specific augmentations. Since it might be detrimental, too, if applied blindly.

For example, for an Optical Character Recognition task, you can use **`nlpaug`** to create an OCR augmentation. For example,

```
Original:

The quick brown fox jumps over the lazy dog.

Augmented Texts:

['The quick bkown fox jumps ovek the lazy dog.', 'The quick 6rown fox
jumps ovek the lazy dog.', 'The quick brown f0x jomps over the la2y dog.']
```

The preceding example shows common OCR mistakes like brown -bkown,/6rown, and so on. This can help create a lot of examples specific to how the OCR data works.

So, another important intuition about any type of augmentation. i.e., blind augmentations will make your data garbage. And as we say in machine learning, garbage in, garbage out. So be careful using augmentation; it's a great tool, but when misused, it can lead to disastrous results.

Next, we will go through a few areas that we can automate in the whole process of data acquisition, synthetic data creation, etc.

3.3 The need for automation

Machine learning is repetitive. We also follow a specific sequence of steps to do any machine learning project. Formulate the problem first, then collect the data and perform pre-processing, cleaning, augmentation, feature engineering; after that, choose a specific model (through trying many models and logging performance), train and validate, deploy, and test. All these processes, although done in a sequence are dependent. For example, if your model is not performing well and you identified some other data sources that can be better as a root cause, you then go back to acquire that data and do the pre-processing, cleaning, and so on before training. Most of these processes are cyclical.

So, scripting is an extremely valuable skill set as a machine learning engineer. It will just make your life way easier to automate certain tasks. Automation will also make your process less error-prone. But ask yourself the following questions whenever you are creating any data acquisition process.

What can we automate?

Let us go through a normal data acquisition process and see what we can automate.

1. Write a script to check your required data present in any of the preceding sources.
2. Write a script to handle data acquisition from any database (connection, fetching all these you can automate).

How easy it is to modify the acquisition process if sources change/pre-processing changes, and so on.

Ask yourself if some source changes, will the same automation work. If not, how you can make your code modular, so, changes will be easy to accommodate.

Now, scripting is a huge topic and beyond the scope of this book, but few libraries will help you a lot, For example, scrapping(requests, bs4, selenium, scrapy), file manipulation (os, pathlib (i find this package really useful), shutil), argument parsing(fire, plac), and linux tools (grep, wget, curl). We will use these tools throughout this book.

3.4 Why is cleaning the data necessary?

In industry or any practical setup, data is collected by humans or using some process/sensors, and so on. Since it's a physical process, data is often riddled with errors and inconsistencies. This is probably one of the reasons for machine learning chosen over traditional coding because this adds complexity and machine learning is also robust to noise (remember in *Chapter 1: Introduction to machine learning*, we discussed representation learning).

But machine learning is not magic; if your data has 70% missing data because of some human error or sensor fault, then it's not going to work, at least not without some cleaning. This is where data cleaning comes into the picture. This is such a big area that discussing all cleaning data is not feasible in a single chapter; rather, we will go through specific cleaning when we come across the use cases. For example, predictive analytics on tabular data has a different set of cleaning techniques than natural language.

In the *Introduction to Machine Learning* chapter, we discussed a spam-ham email classifier. So, are email signatures, salutations (for example, Hi Mr.X or Regards, and so on) required to decide if an email is a spam or ham? The answer is no. But that may confuse the model in focusing on these unnecessary words. So there cleaning is required to remove the salutations in data.

So how do we remove errors? We will follow the following process:

1. First, follow best practices in cleaning data, e.g., missing value handling, text cleaning, and so on. This part is well researched and standardized; packages allow you to do all of the cleanings with a single line of code. And some algorithms even have inbuilt mechanisms to handle this.
2. After that, we should do domain-specific cleaning. A big part of machine learning is communication, communicating with the domain experts. Let's say you created an anomaly model for a wind energy company. You take their daily power consumption, data related to propellers (yaw angle, bearing oil temperature), and so on and predict if the current data is an anomaly or not. And when you use inference on why your model predicted such, you realized power consumption dipped to a real low value for the anomalies. And hence you asked the domain experts.

   But they might tell you power consumption periodically dips anyway, and that's no anomaly. So, you need to do some specific cleaning on power consumption data to remove this periodic trend. So, just applying few well-known cleanings might not be sufficient. And sometimes you need to be careful in cleaning the data too.

3.5 Need and importance for a test set

When we create a machine learning model, our end goal is to predict something in the future using that model. And we have not seen the future data nor the model. So, when we train our model, we need part of the data aside from training because human beings are pattern-finding machines and can introduce their own bias.

So, never skip a test set. A test set is particularly important. We divide our data into three different sets; first, the training set is the data on which the model parameters are learned. The second is a validation set; this is used to tweak additional parameters a model has, which cannot be learned from data. These are called *hyperparameters*. We will go through hyperparameter tuning in the model training chapter. Also, training and validation sets are sometimes intertwined; we divide the training and validation data into n parts and train on the n-1 part, and keep 1st part of validation. Then we change the validation to 2nd part and so on. This is called k-fold *cross validation*. All these tactics will be learned during the model training part. But we should always keep a dataset that is not part of training and should also not be part of hyperparameter tuning.

One way to do this is by asking the client to keep some recent data for testing. We need to specify that the data should not be completely different; it should be somewhat similar.

Conclusion

With this, we conclude the third chapter on machine learning. To summarize, we came across data acquisition techniques and sources. We learned how to negotiate a sufficiency for machine learning data. We understood how to add synthetic data with careful analysis as blind augmentation can ruin your model. We discussed the automation need and see why cleaning is important. In the next chapter, we will go through how we do the initial round of analysis on data. Stay tuned.

Points to remember

The following are a few points to remember:

- There is no way to know how much data is required for you to create a model exactly. All we can do is to come up with an overall answer with few educated guesses.
- Data augmentation works like a charm in many machine learning problems with less data.
- Automate whatever you can; it helps a lot in speeding up the machine learning process.
- Keep aside a test set at the beginning; don't touch it or see it.

MCQ

1. **The parameters of a model that is not learned from data.**
   a) Hyperparameters
   b) Internal variables
   c) External variables
   d) Environment variables

2. **Which metric is used to capture important words?**
   a) F1 score
   b) TFIDF
   c) RMSE
   d) P-value

3. **Creating synthetic data is also called?**
   a) Data processing
   b) Feature engineering
   c) Data augmentation
   d) Exploratory data analysis

4. **Which code prints 1-5**
   a) print(list(range(5))
   b) print(np.linspace(1, 5))
   c) print(list(range(1, 6))
      a. both a and c
      b. b only

Answers to MCQ

1. a
2. a
3. c
4. c

Questions

1. Write a script to check if data for a specified keyword is present in the sources mentioned previously.
2. Note the difference between a training set, validation set, and test set.
3. Write a function that takes input and performs rotation, zoom, and crop randomly and generates ten images.

Key Terms

1. Data augmentation
2. Test set
3. Term Frequence- Inverse Document Frequency

Chapter 4
Exploratory Data Analysis

In this chapter, we will learn about exploring data, asking initial questions, and understanding various data visualization techniques that will help us answer questions. We call this data exploration and the analysis part Exploratory Data Analysis, and from now on, we will call it EDA for short. We will start this chapter with a disclaimer; EDA can cause more harm than helping us. Yes, that is correct, and we will learn why we should not look too deep into the data in the beginning. That is why we call it initial analysis (and not EDA, because we won't explore too deep), and we will learn what to ask and what not to ask during this stage. We will learn about human biases and data leaking issues in machine learning.

This section will focus more on the "*How EDA helps to achieve the goal?*". I have seen many tutorials on EDA, which goes through extensive EDA techniques but fails to provide enough reasoning and experimentation/ablation studies on the effectiveness of EDA. Furthermore, many tutorials on EDA fail to answer, "*Why should we do this analysis in the first place?*" So, we will dig deep into this question. Also, we will focus mostly on tabular data rather than image/text data since EDA is done more extensively in tabular data. This section may seem a bit different from the general approach of EDA, but we will understand why it works better. With that in mind, let's start with the first look into the data.

Structure

In this chapter, we will cover the following topics:

- Initial analysis
- Key advice to follow during the initial analysis
- Which visualization to choose?

Objective

After studying this chapter, you should be able to:

- Why I should not go for an in-depth analysis at the beginning
- Exploratory data analysis versus initial analysis
- The initial analysis makes the machine learning experience better
- The visualization help conveying a specific idea

4.1 Initial analysis

The reason for an initial analysis is first to make the data that we collected during the acquisition phase consumable for the machine. Machine learning models are nothing but mathematical equations; it is performing some calculations on numbers. But real-world data is not just numbers. We have categories; we have text; we have missing values and incorrect values due to sensor or human error. All these need to be handled so that a machine learning model can start learning. Now, doing this requires us first to see these details of data. This is where initial analysis comes in. But human beings are also pattern-finding machines. We even found patterns in seemingly random star constellations. This may be dangerous if we do it at the start. There are two reasons for that. We will go through those reasons next.

First, we are doing machine learning so we can learn about hidden patterns in huge data. So, if we start with our own biases about data, we may exclude certain variables which do not conform to our biases, and hence we might miss them. Second, machine learning is not the same as human learning; machines might use some variables and come up with patterns we may even think ridiculous. But if we dig deeper, we might understand why after analyzing it with a domain expert because they are aware of the subtleties.

That is when we will do true Exploratory Data Analysis after we make a model using initial analysis and feature engineering. Next, we will use the model to tell us interactions and inferences about the data, and we will dig deeper. Then we will take those insights to a domain expert and confirm.

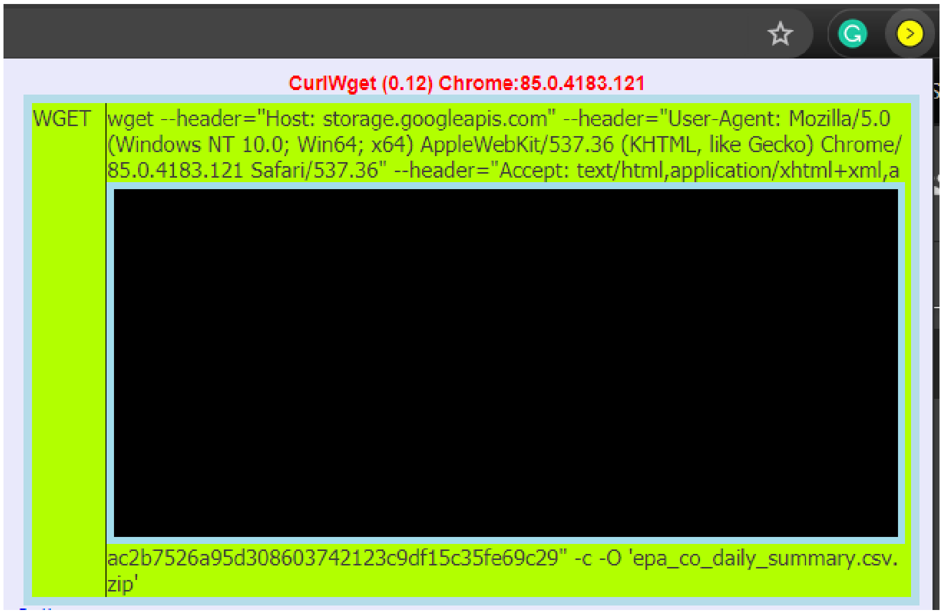
Now let's go through the following few initial analysis techniques:

4.1.1 See if you can load the data

First, let's take a dataset; we can choose a huge dataset for this discussion from *Kaggle*, *Carbon Monoxide Daily Summary*. This is a dataset for sales forecasting. You can go through this dataset and problem description by following this link: **https://www.kaggle.com/epa/carbon-monoxide**. Do register to *Kaggle* to download; also, competition datasets are only available if you participate, so first, register if it's a competition.

Quick Tip: We are going to use *Google Colab* again for loading this data. And since machine learning is often done on remote systems with generally Linux operating system, which you access via your day-to-day windows system, here is how you can get the data.

1. First, install a chrome extension called **`CurlWget`** from the Chrome web store.
2. Now click on the **`Download all`** button for this dataset (for any other dataset, you can also click on the download button).
3. Cancel the download. Now, click on the **`CurlWget`** icon present on the right side, where all extension icons are present. Next, you can see the **`wget`** command for the download with all the cookies required to download. The following is the screenshot of the **`Wget`** command in **`CurlWget`** (the credentials are masked so that the big black box won't be there for you).

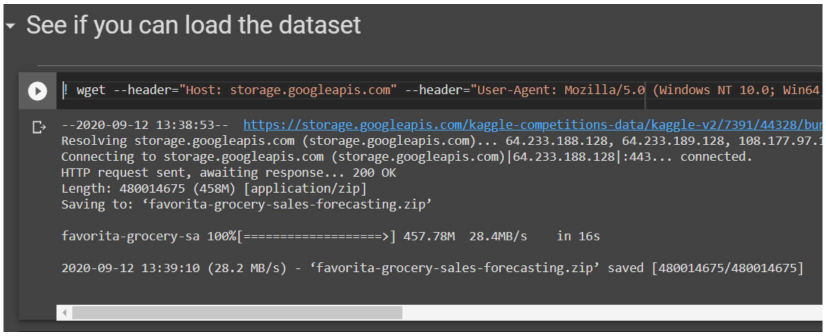

CurlWget (0.12) Chrome:85.0.4183.121

WGET

```
wget --header="Host: storage.googleapis.com" --header="User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4183.121 Safari/537.36" --header="Accept: text/html,application/xhtml+xml,a
[redacted]
ac2b7526a95d308603742123c9df15c35fe69c29" -c -O 'epa_co_daily_summary.csv.zip'
```

Figure 4.1: *CurlWget in action*

4. From the notebook, you can download the files simply by adding a '! 'in the beginning. For example, the following is the screenshot of the **`wget`** command in Google Colab:



```
! wget --header="Host: storage.googleapis.com" --header="User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64
```

```
--2020-09-12 13:38:53--  https://storage.googleapis.com/kaggle-competitions-data/kaggle-v2/7391/44328/bu
Resolving storage.googleapis.com (storage.googleapis.com)... 64.233.188.128, 64.233.189.128, 108.177.97.
Connecting to storage.googleapis.com (storage.googleapis.com)|64.233.188.128|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 480014675 (458M) [application/zip]
Saving to: ‘favorita-grocery-sales-forecasting.zip’

favorita-grocery-sa 100%[==================>] 457.78M  28.4MB/s    in 16s

2020-09-12 13:39:10 (28.2 MB/s) - ‘favorita-grocery-sales-forecasting.zip’ saved [480014675/480014675]
```

Figure 4.2: *Using the CurlWget command in Google Colab notebook*

The **`wget`** command for your download will be different than mine. So just copy and paste the exact command that the **`curlWget`** generates. After downloading the data, we need to unzip it and unarchive the train.csv.7z file to get the training set. We can use the following command:

```
! unzip 1505_2690_compressed_epa_co_daily_summary.csv.zip
```

Sometimes the file will be archived using tar or 7z, and so on; you can use specific Linux tools (tar, 7z) to unzip them. Don't worry if you don't know how to do it; someone may have already faced this issue, and you can search for it online.

Now, we will go through the following checklist to load data (this is mostly part of data engineering and not analysis, but the checklist will be helpful for the reader first to load some data to analyze); let's go through them below:

1. Check where you are loading the data from; the following are few sources and how you can load them.

    a) *SQL* – you can use **`pandas.read_sql`** and use appropriate connection parameters to connect to SQL.

    b) *CSV files* – you can use **`pandas.read_csv`** to load the CSV file; also, for TSV files, you can use the delimiter parameter.

    c) *Excel files* – you can use **`pandas.read_excel`** to load excel files; it will require an additional package(**`xlsxwriter`**).

    We will use **`pandas.read_csv`** for our case since our file is a CSV file. But pandas do support a variety of different data types.

2. See if you can load the data into the primary memory (RAM), taking care of datatypes.

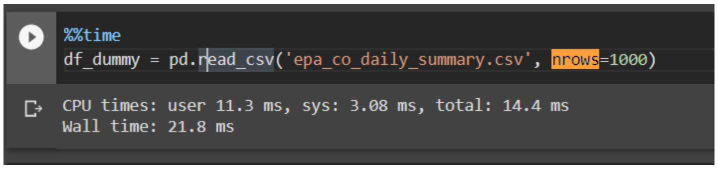
Often the dataset will load into primary memory, but a simple way you can check is simply by seeing the size of the dataset. That doesn't mean we should load an 8GB dataset if we have 8GB RAM. Because doing so won't allow anything to load. One simple rule of thumb I use in my experiments is to keep 40% of my RAM free after loading data. Since we are going to need more RAM during processing and other stuff (creating new features and so on)

One parameter in pandas you can use is **`chunk_size`**; this will load mentioned size into primary memory in batch.

Also, when I start loading a dataset, I use the **`nrows`** parameter to load around 1000 first rows to load the data and quickly visualize it; the following is the code for that:

```
%%time

import pandas as pd

df_dummy = pd.read_csv('epa_co_daily_summary.csv', nrows=1000)
```

Let's run the above piece of code; output is shown below:



```
%%time
df_dummy = pd.read_csv('epa_co_daily_summary.csv', nrows=1000)

CPU times: user 11.3 ms, sys: 3.08 ms, total: 14.4 ms
Wall time: 21.8 ms
```

***Figure 4.3**: Loading a dummy data with 1000 rows*

Quick Tip1: %%time is a Jupyter notebook magic function to show run duration. We should use it as much as possible to find out how long a certain code takes to run. There are other useful magic functions we will go through. For example, our code took around 25ms for 1000 rows.

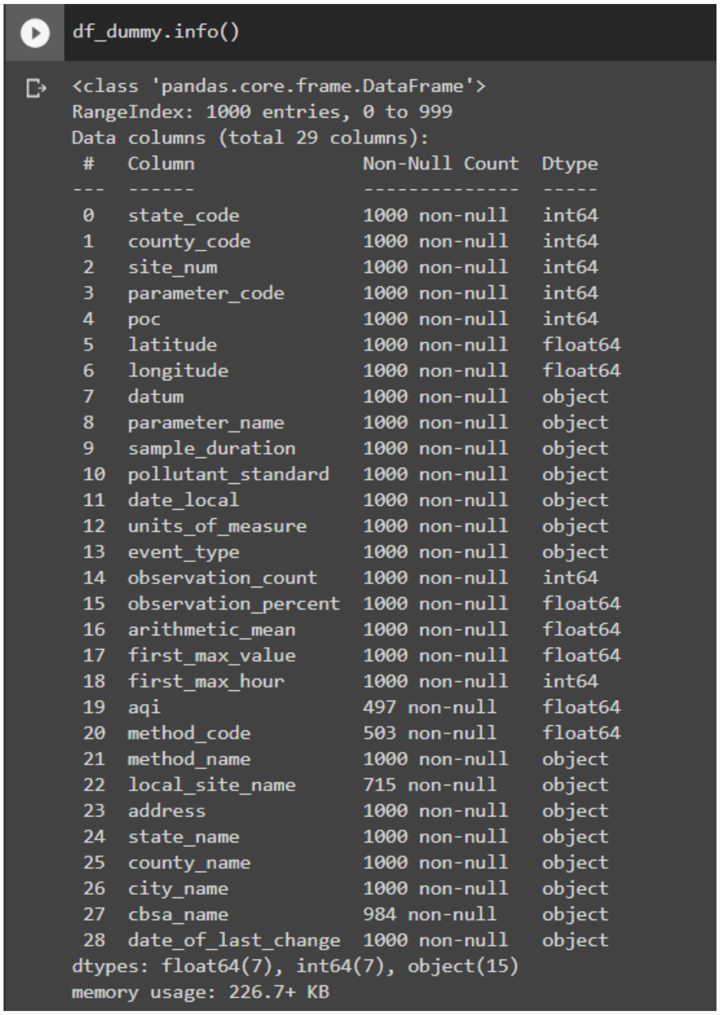
Quick Tip2: The dataset we are using here has a download size of 574MB and an unzipped size of 2.14GB. Sometimes you may get memory errors as we are working on a memory-constrained environment. In case that happens, try clearing the downloaded files or start a fresh session.

After this, we can get some quick info using the **`.info()`** method on dataframes. And display the first few rows using the **`.head()`** method.

The following is the code to run info and head information:

```
df_dummy.info()
```

The output of the preceding code:



```
df_dummy.info()
```

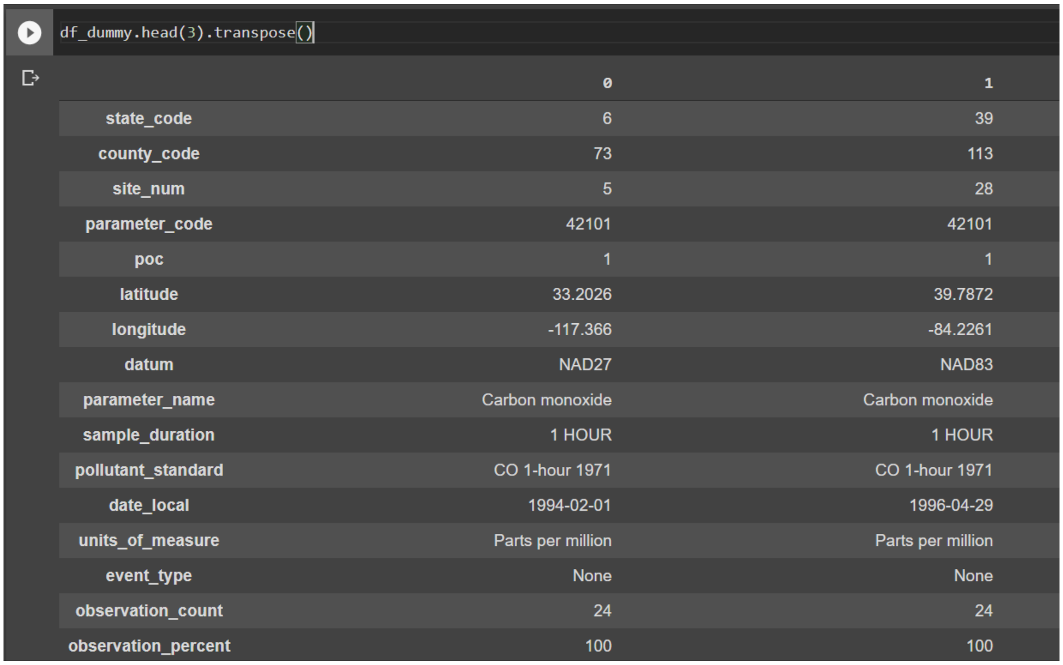
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 29 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   state_code           1000 non-null   int64  
 1   county_code          1000 non-null   int64  
 2   site_num             1000 non-null   int64  
 3   parameter_code       1000 non-null   int64  
 4   poc                  1000 non-null   int64  
 5   latitude             1000 non-null   float64
 6   longitude            1000 non-null   float64
 7   datum                1000 non-null   object 
 8   parameter_name       1000 non-null   object 
 9   sample_duration      1000 non-null   object 
 10  pollutant_standard   1000 non-null   object 
 11  date_local           1000 non-null   object 
 12  units_of_measure     1000 non-null   object 
 13  event_type           1000 non-null   object 
 14  observation_count    1000 non-null   int64  
 15  observation_percent  1000 non-null   float64
 16  arithmetic_mean      1000 non-null   float64
 17  first_max_value      1000 non-null   float64
 18  first_max_hour       1000 non-null   int64  
 19  aqi                  497 non-null    float64
 20  method_code          503 non-null    float64
 21  method_name          1000 non-null   object 
 22  local_site_name      715 non-null    object 
 23  address              1000 non-null   object 
 24  state_name           1000 non-null   object 
 25  county_name          1000 non-null   object 
 26  city_name            1000 non-null   object 
 27  cbsa_name            984 non-null    object 
 28  date_of_last_change  1000 non-null   object 
dtypes: float64(7), int64(7), object(15)
memory usage: 226.7+ KB
```

***Figure 4.4**: Checking info for loaded dummy dataset*

Let's check the first few rows. Here I will transpose the first three rows using .head(3) followed by **`.transpose()`** for better display.

```
df_dummy.head(3).transpose()
```

The output of the following code:



```
df_dummy.head(3).transpose()
```

	0	1
state_code	6	39
county_code	73	113
site_num	5	28
parameter_code	42101	42101
poc	1	1
latitude	33.2026	39.7872
longitude	-117.366	-84.2261
datum	NAD27	NAD83
parameter_name	Carbon monoxide	Carbon monoxide
sample_duration	1 HOUR	1 HOUR
pollutant_standard	CO 1-hour 1971	CO 1-hour 1971
date_local	1994-02-01	1996-04-29
units_of_measure	Parts per million	Parts per million
event_type	None	None
observation_count	24	24
observation_percent	100	100

***Figure 4.5**: Looking into some values*

Here are the things we can get:

a) We can check if there are any date fields; later, we can use this info to parse the date values differently. For example, we can see **`date_local`**, **`date_of_last_change`** are date fields.

b) We can get some idea about column names; using it, we can check on the Kaggle documentation or client about its maximum and minimum values. This will help us determining data types for the column. Data type determination is a must step and often overlooked. For e.g., **`country_code`** is an int64 data type (64 bits to store), but we don't require 64 bits to store it. This is because we don't have that many countries anyways.

c) Similarly, longitude and latitude are float64; we don't need that many bits. Since longitude and latitude also contain at max five significant values after a floating point. So we can define the data types with much less precision. This allows us to store the data with much less space and increases the processing time significantly.

Do not skip this part in the beginning; many people skip the dtypes and date fields part at the beginning, which makes every processing much longer. So, spend some time in the beginning to determine data types. Then, it hugely speeds up the rest of the machine learning process.

With this analysis we can now load our data into memory, sometimes with huge data we may get memory error and we can use the **`chunk_size`** parameter to load less data. But for our case we can now load the data, with proper date parsing and datatypes with the following code:

```
%%time
df_raw = pd.read_csv('epa_co_daily_summary.csv',
                      low_memory=False,
                     # dtypes=dtypes,
                     parse_dates=['date_of_last_change', 'date_
local'],
                     infer_datetime_format=True)
```

Here we are parsing the dates for the date columns identified; also, we can give a data dictionary dtypes, which we will figure out using the initial look into the data. As an exercise, try to identify the various datatypes and pass them in the variable dtypes (uncomment line 4). **`infer_datetime_format`** just automatically parses the format of the date string. So if your date format in data is some weird string, you can also pass that.

4.1.2 Size of dataset

Size matters a lot when creating any model. It directly affects the training time. Some datasets are so huge in size that we cannot even load them into memory. We can use **`chunk_size`** to load it in batch. But even for our dataset, we should start with a sample. Test your code on that sample first because it's very frustrating if you identify an error in code and rerun it after rectifying it on 1 million rows which will take so much time.

The basic idea is to ensure your code runs fine on a sample dataset first (use nrows, or after loading, randomly sample your dataset). First, test your code, then load the whole dataset. Now there are certain errors that you might only get once you load the whole dataset. e.g., your sample might not contain any null values; in that case, you might get the error when you load the whole data. One way to create a good sample is (which I follow) is to get the first and last 300 rows and sample 600 rows randomly in between. But you can create the sample on your own. Just think for some time before you create a sample. It will help you minimize errors in the future.

Another aspect of dataset size is visualization. Unfortunately, visualizing huge data is not that simple. It may just take a long time to render or may create an extremely cluttered chart that you can't comprehend. We will follow some guidelines for that too in the future.

4.1.3 Different feature types

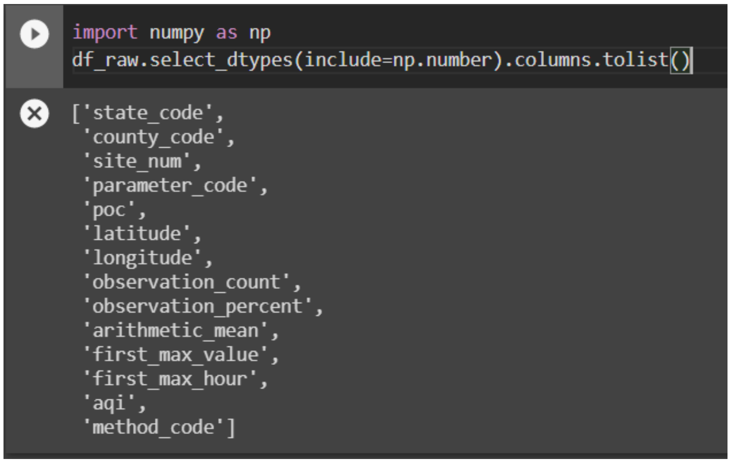
As discussed before, machine learning models are nothing but mathematical functions that take some numbers and then spit out an output. So, we first need to look at all the feature types that are not numbers. So, let's divide our feature into different types as follows:

- Numerical features:

  These features are numerical already, so they don't require converting. The following is the code snippet to identify numerical features:

  ```
  import numpy as np
  df_raw.select_dtypes(include=np.number).columns.tolist()
  ```

  The output of this code:



```
import numpy as np
df_raw.select_dtypes(include=np.number).columns.tolist()
```

```
['state_code',
 'county_code',
 'site_num',
 'parameter_code',
 'poc',
 'latitude',
 'longitude',
 'observation_count',
 'observation_percent',
 'arithmetic_mean',
 'first_max_value',
 'first_max_hour',
 'aqi',
 'method_code']
```

***Figure 4.6**: Selecting numerical columns*

We use the select_dtypes method of Pandas dataframe and only include numbers using **`np.number`** and then just taking the column names.

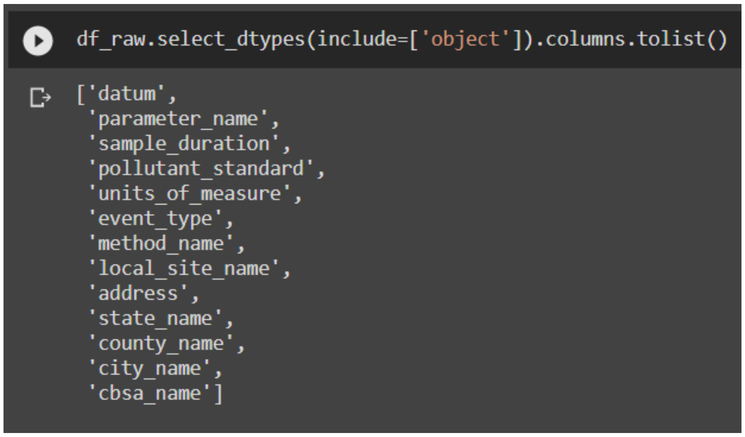
Caution: Numbers may also be deceiving; for example, parameters such as **`state_code`**, **`county_code`**, **`site_num`**, **`parameter_code`**, **`poc`**, **`method_code`** are representing some code. They are categories and not numbers. We need to handle these carefully because numbers have a sense of distance and order, e.g., two is greater than one. But it makes no sense in saying country code 32 is greater than country code 30. Since these are just codes representing. So, we need to tell the model that these are not normal numbers but categories. So have some general understanding of what features mean and ask the client about it. Machine learning at the initial data phase requires a lot of communication between client and engineer.

- Categorical features

  These are categories; for example, **county_name** is a category. We can select categorical variables using the following code. But this code will also include text features, so keep an eye on that too. In our example, there are no text features. Here is the code for selecting categorical column names:

```
df_raw.select_dtypes(include=['object']).columns.tolist()
```

  The output to this code:



```
df_raw.select_dtypes(include=['object']).columns.tolist()
```

```
['datum',
 'parameter_name',
 'sample_duration',
 'pollutant_standard',
 'units_of_measure',
 'event_type',
 'method_name',
 'local_site_name',
 'address',
 'state_name',
 'county_name',
 'city_name',
 'cbsa_name']
```

***Figure 4.7**: Selecting categorical features*

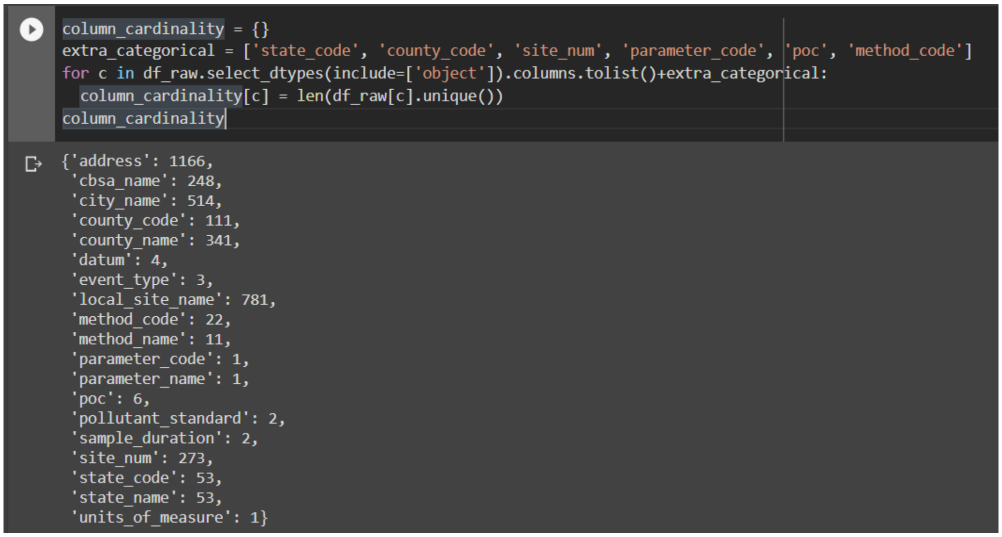
Now before converting these into numbers let's first understand a few things about cardinality. Cardinality of a categorical variable is unique values present in that column. Based on that, its divided into two types mentioned:

- **High cardinality**: If there are a lot of unique values e.g. address will have a lot of different values
- **Low cardinality**: Less unique values e.g., **event_type** only has three unique values

  The code for finding out cardinality:

```
column_cardinality = {}
extra_categorical = ['state_code', 'county_code', 'site_num',
'parameter_code', 'poc', 'method_code']
for c in df_raw.select_dtypes(include=['object']).columns.
tolist()+extra_categorical:
  column_cardinality[c] = len(df_raw[c].unique())
column_cardinality
```

  The output for this code:



```
column_cardinality = {}
extra_categorical = ['state_code', 'county_code', 'site_num', 'parameter_code', 'poc', 'method_code']
for c in df_raw.select_dtypes(include=['object']).columns.tolist()+extra_categorical:
  column_cardinality[c] = len(df_raw[c].unique())
column_cardinality
```

```
{'address': 1166,
 'cbsa_name': 248,
 'city_name': 514,
 'county_code': 111,
 'county_name': 341,
 'datum': 4,
 'event_type': 3,
 'local_site_name': 781,
 'method_code': 22,
 'method_name': 11,
 'parameter_code': 1,
 'parameter_name': 1,
 'poc': 6,
 'pollutant_standard': 2,
 'sample_duration': 2,
 'site_num': 273,
 'state_code': 53,
 'state_name': 53,
 'units_of_measure': 1}
```

***Figure 4.8**: Finding cardinality of features*

Here we are treating **`state_code`**, **`county_code`**, **`site_num`**, **`parameter_code`**, **`poc`**, **`method_code`** as categorical, too, as discussed. One rule of thumb is to consider any column with cardinality > 20 as high cardinality.

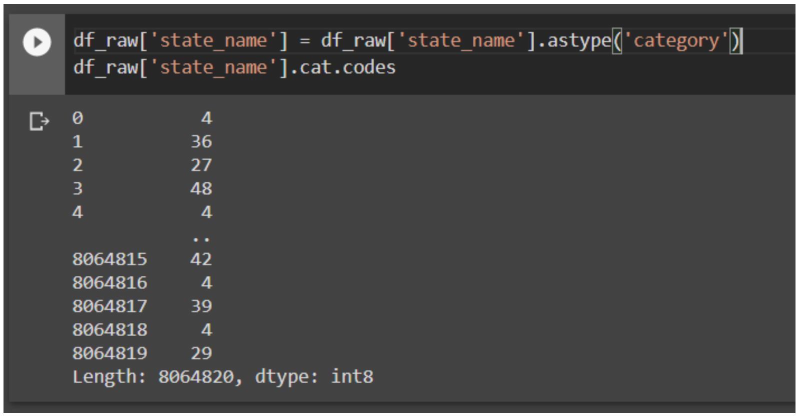
Once we determine the cardinality, we need to now convert these into numbers. The following is how we can convert:

- Label encoding for high cardinality:

  `df_raw['state_name'] = df_raw['state_name'].astype('category')`

  `df_raw['state_name'].cat.codes`

  The output for this code:



```
df_raw['state_name'] = df_raw['state_name'].astype('category')
df_raw['state_name'].cat.codes
```

```
0           4
1          36
2          27
3          48
4           4
           ..
8064815    42
8064816     4
8064817    39
8064818     4
8064819    29
Length: 8064820, dtype: int8
```

***Figure 4.9**: Converting a categorical variable*

So now the values are changed into codes between 0-52, as there are 53 values for **`state_name`**. The **`state_name`** actual values are also present. You can check it by printing. The codes are stored in **`.cat.codes`**.

- One hot encoding for low cardinality: For low cardinality variables, e.g., **`method_name,`** which has 11 values only, we can create 11 columns with these values and assign 1 to the value for that row and zero for others. This is called one hot encoding. Let's understand this with an example.

  Let's say the column has four values A, B, B, C. There are three unique values. We can create three columns A, B, C, and for these four values, we can assign one hot encoding as follows:

  `A - [1, 0, 0]`

  `B - [0, 1, 0]`

  `B - [0, 1, 0]`

  `C - [0, 0, 1]`

  To do this in code for our **`method_name`** feature, the following is the code:

  ```
  df_raw = pd.get_dummies(df_raw, columns=["method_name"])
  df_raw[df_raw.columns[pd.Series(df_raw.columns).str.
  startswith('method_name')]]
  ```

  `pd.get_dummies` are a powerful function that can create these dummy variables in a single line of code. It has many options; you can add a prefix to these columns, provide which columns you want to create dummies for, etc. By default, it will create dummy columns with column names as a prefix. The second line of code selects these columns using **`method_name`** as a prefix. This code will print out all the dummy variable columns.

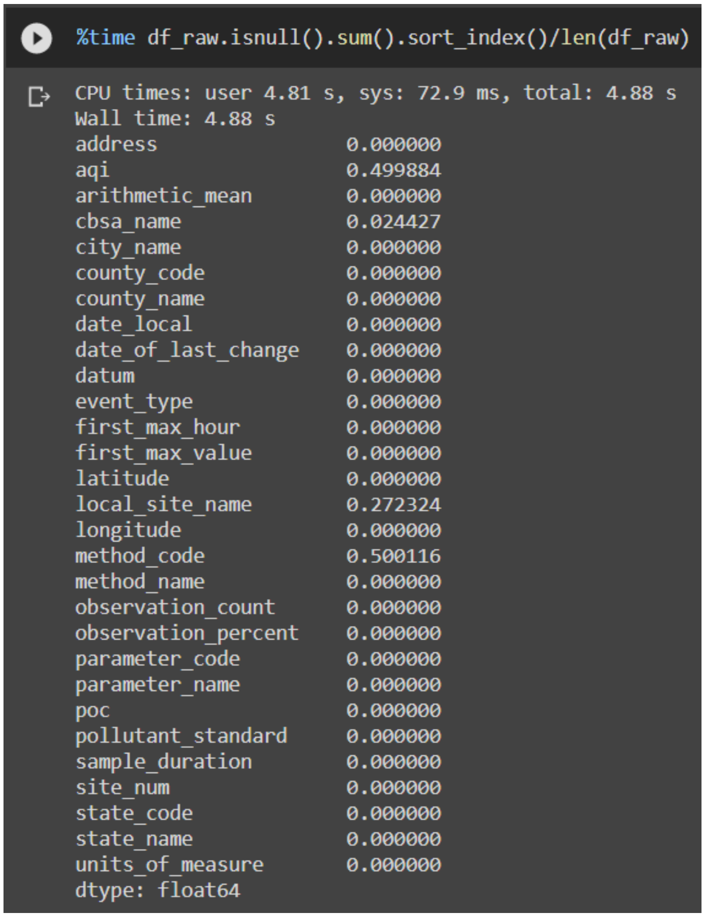
  Caution: One hot encoding or label encoding has few problems also; one of them is a mismatch between test and train set. For example, if your test set has some categories that are not there in the training set, you will face an issue in converting these values into one hot encoding. These problems will be addressed when we will go through building models.

4.1.4 Missing values

Our data may contain missing values; this can happen for many reasons; you can miss certain values due to human error. There might be some sensor error that introduces missing values. There might be some periodic missing values due to environmental issues, and so on. There are many ways to treat missing values, but first, we need to identify them; here is the code for that:

```
%time df_raw.isnull().sum().sort_index()/len(df_raw)
```

Output:



```
%time df_raw.isnull().sum().sort_index()/len(df_raw)
```

```
CPU times: user 4.81 s, sys: 72.9 ms, total: 4.88 s
Wall time: 4.88 s
address                  0.000000
aqi                      0.499884
arithmetic_mean          0.000000
cbsa_name                0.024427
city_name                0.000000
county_code              0.000000
county_name              0.000000
date_local               0.000000
date_of_last_change      0.000000
datum                    0.000000
event_type               0.000000
first_max_hour           0.000000
first_max_value          0.000000
latitude                 0.000000
local_site_name          0.272324
longitude                0.000000
method_code              0.500116
method_name              0.000000
observation_count        0.000000
observation_percent      0.000000
parameter_code           0.000000
parameter_name           0.000000
poc                      0.000000
pollutant_standard       0.000000
sample_duration          0.000000
site_num                 0.000000
state_code               0.000000
state_name               0.000000
units_of_measure         0.000000
dtype: float64
```

Figure 4.10*: Checking null value percentage*

This code gives you a percentage of missing values as raw numbers might not be that intuitive. So we can see **air quality index** (**aqi**) as around 50% missing values, **`cbsa_name`** around 2.5%, and so on. This is also another time you should go to the client and ask about these missing values, especially columns which has more than 50% missing values. There might be some systemic issue that is causing this.

Ways to deal with missing values: The following are a few strategies to handle missing values:

1. **Just skip them**: Using the **`.dropna`** method, we can skip missing rows. This is used if a lot of missing value present, and it does not add anything to the analysis.
2. **Replace with some central tendency**: Replace these missing values with mean, median, mode using **`.fillna`**. Using median over mean works better

for columns with extreme values (outliers) since the median is less susceptible to these.

3. **Using other variables to predict the missing values**: We can use related variables to predict the missing values, e.g., let's say we have height and weight as two features. As height and weight are highly correlated, we can use one of them to predict the missing values of the other.
4. Some algorithms like random forest deal with missing values uniquely using proximity, so we need not worry about them.

Also, even after filling in the missing values, we should keep another variable containing the information if that column is missing. This gives us extra flexibility. Of course, missing value treatment is a huge topic and requires experimentation with modeling to go through it in detail. But as a rule of thumb, any central tendency with the missing information works fine in most cases.

Sometimes missing values are deceptive; someone may insert NA, 'not present,' Empty, and so on. in a column if they don't know the value. Be aware of this. We need to convert it into proper missing value.

4.1.5 Outliers

Outliers are extreme values for a feature/column. For example, let's say we take the average income in a country like India and then compare it to *Mukesh Ambani's* income. There will be so big a difference. That is an outlier. Most of the time, outliers are removed from the dataset. But they can be useful for some inferential analysis. So, don't blindly remove them. We can detect them using box plots for numerical variables, which we will see in the visualization section of this chapter. One way to do a quick analysis of data is to see some descriptive statistics like mean, median, quantiles, max, min values, and so on using the **`.describe()`** method in a Pandas dataframe.

4.1.6 Nonusable features

Features like **`id`**, **`item_no`**, and so on are generally not useful for predicting anything. And we generally skip these. Also, columns with cardinality one, which means there is only a single unique value, will not add anything to the table. These are non-usable features. Sometimes features contain information that is not present during the prediction time. For example, if you are trying to predict a student performance index based on historical data. One of the columns is filled by a teacher; let's call it teacher feedback. The teacher gives high value (in that column, teacher feedback) for students with good performance. This information should not be present during the prediction time, but it's present in historical data. This will give an extra advantage to the model since one column already contains information regarding what the model is trying to predict. This is called **data leakage**. This is dangerous too. Another example for data leakage I came across is a competition to predict sale price, and **`item_no`** is aligned according to the sale price. Using **`item_no`** as a feature, we can readily predict the sale price. But this is cheating. This information is not present during prediction, and hence it's also part of data leakage. So, communicate if you find some absurd variable that should not be a good predictor but highly correlated with prediction. Chances are it is due to data leakage. We will learn about how to get good predictors in our modeling chapter.

4.2 Key advice to follow during the initial analysis

The following are some advice to follow when working with real world data:

4.2.1 Do not do correlation analysis at the start

If you already know some correlated features to the output, we may unknowingly bias the system and miss other hidden meanings. That is the reason we should not do correlation analysis at the start. Let the model decide first. But this doesn't mean we should not do any correlation analysis. The idea is to do it at a later stage.

4.2.2 Look into the target variable

The target variable is important. Let's take an example for "*Corporacion Favorita Grocery Sales Forecasting*" (**https://www.kaggle.com/c/favorita-grocery-sales-forecasting**) dataset from Kaggle. If you go to the evaluation section of the competition, you can find out the metric they are judging is **Normalized Root Mean Squared Logarithmic Error** (**NWRMSLE**). Don't be afraid of big names; in machine learning, you will often find big names for small things. The following is the formula:

$$NWRMSLE = \sqrt{\frac{\sum i = 1^n wi\,(\ln(\hat{y}i + 1) - \ln(yi + 1))^2}{\sum i = 1^n wi}}$$

So, what does this metric mean?

Generally, in problems where we predict a continuous number (Regression Problem), **Mean Squared Error** (**MSE**) or **Root Mean Square Error** (**RMSE**) is taken as a metric. MSE takes the difference between what you have predicted and the ground truth, squares it, and sums it up the overall test set. For RMSE, you take the root of that number (MSE). But what the problem is asking for is first to take a log of all the predictions and log of all the ground truth, add 1 to both of them (because the log of 0 is a disaster), then take the difference and square it. Then multiply it with a weight given in a separate file and take the square root. It is similar to RMSE. So, for this problem, the good idea is to convert the target variable that is **`unit_sales`** by adding one to it and taking the logarithm of that, then multiply by the weight derived from another table.

I am giving this example because we need to look deeply into the target variable; we may need to convert it, we may need to use some additional data to make it inline with our business ask. So, spend some time understanding the target variable.

4.2.3 Focus on speeding things up

The following are a few guidelines for speeding things up:

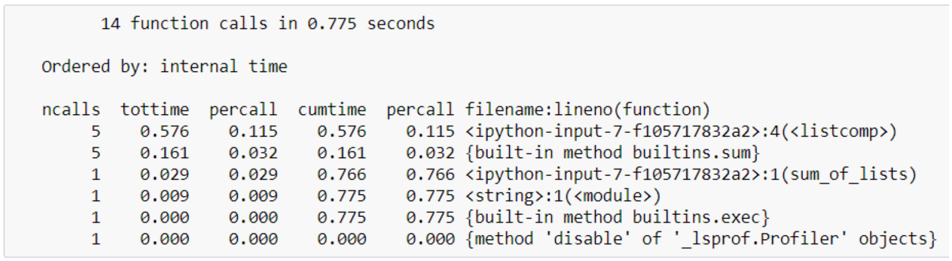
- **Work with samples first**: Before working on the full dataset, work with small samples; the idea is to complete the pipeline first with small data, where if you face any issue or code errors, it takes less time to fix. Of course, this does not mean your code will run fine once you work with complete data. As discussed earlier, some issues may only come when you put all the data. But it will isolate coding errors.
- **Run profile on code**: Also, use the magic function **`%prun`** in notebooks to profile your long running code. This will tell you which lines of code runs longer in a function. Let's take an example as follows:

```
def sum_of_lists(N):
    total = 0
    for i in range(5):
        L = [j ^ (j >> i) for j in range(N)]
        total += sum(L)
    return total
```

  It's a simple program that uses a list comprehension in the middle, which should take the highest time. Let's run a profile in that:

```
%prun sum_of_lists(1000000)
```

  The output of this code:



```
         14 function calls in 0.775 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        5    0.576    0.115    0.576    0.115 <ipython-input-7-f105717832a2>:4(<listcomp>)
        5    0.161    0.032    0.161    0.032 {built-in method builtins.sum}
        1    0.029    0.029    0.766    0.766 <ipython-input-7-f105717832a2>:1(sum_of_lists)
        1    0.009    0.009    0.775    0.775 <string>:1(<module>)
        1    0.000    0.000    0.775    0.775 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Figure 4.11: *Output of %prun for profiling*

This output shows the code ran in 0.775 seconds, and out of that, 0.576 seconds is taken by line 4, which is the list comprehension. There are other more sophisticated profilers like **`line_profiler`**. For example, for memory profiling, you can use **`memory_profiler`**. To use these, just **`run pip install line_profiler memory_profiler`**. And use **`%lprun`**, **`%mprun`**. Of course, you can use any other profiler you used in your line of work too.

But what will we do with this information? This comes in handy when you call a function repeatedly, and some lines of code take a long time to run; you can think of ways to optimize it. For example, you may take the code outside so that it will be run only once, and so on. This will make your code significantly faster.

- Standardize everything, First code. Refactoring helps. Try to look for repetitive calls; make them a function. Use classes. Do not treat machine learning separately from software engineering. Use proper folder structure. Log your progress. All of these help a lot in the future.

4.2.4 Keep visualization to minimal at this stage

As discussed, we don't want to overdo exploration, so we should not create a lot of visualizations at this stage. But bar charts and histograms are a yes. For extreme values/outliers, we can plot box plots. Not only tabular data for images, but we should also learn how to show images efficiently. But avoid using heatmaps at this stage as we need to avoid correlation analysis in the beginning. But if the client asks for some inference at the beginning or your project is not prediction rather exploration, you can go for many visualizations. The appendix of this book contains how to plot using matplotlib, plotly, and seaborn. But in the next section, we will go through a specific analysis, which visualization we should use.

4.3 Which visualization to choose?

We need to convey the story as efficiently as possible with visualization. Sometimes we need to show distribution, and sometimes we need to show relation among different variables to infer. The following are a few guidelines to follow when choosing a visualization:

1. For distribution, use bar charts, histograms, and scatter plots
2. For showing relationships or dependency among different variables, use line plots, correlation heatmaps, bar charts with two variables (stacked side by side), and so on
3. For showing extreme values, use box plots, whisker plots, and so on
4. For showing composition, use pie charts, stacked bar charts, and so on

We will learn about this in the upcoming chapters.

Conclusion

With this, we conclude the fourth chapter on machine learning. To summarize, we learned about the initial set of analyses to perform on data. We understood why we do those analyses and how that helps us in a later stage of machine learning. We learned about the dangers of overdoing exploration. We understood various feature types and understood how to handle each. We learned about speeding things up and using profiling on our code. Finally, we learned where to use visualization to convey our story most efficiently. In the next chapter, we will learn about model building and tuning. Stay tuned.

Points to remember

The following are a few points to remember:

- Machine learning models are mathematical functions, so we need to convert our features into numbers.
- Memory is limited for huge datasets load data in batch using the **`batch_size`** parameter of Pandas.
- Datatype identification and loading with mentioning data types is a must. On the other hand, it saves memory and time for free.
- Look out for weird values; sometimes, a missing value is a text such as '*empty,*' '*missing*,' '*na*,' and so on. Also, some columns may contain the same category but in different text. Look for these weird values and standardize.
- Don't overdo exploration at the initial stage; correlation analysis may put bias into your mind and model.
- Speed up things. Use profiling. Minimize visualization if possible. Use crisp, clean visualization to convey ideas.

MCQ

1. **What to use for high cardinality categorical features**
   a) Label Encoding
   b) One hot encoding
   c) Feature scaling
   d) Drop them

2. **Which method fills NA values in pandas**
   a) .complete()
   b) .addNA()
   c) .remove()
   d) .fillna()
3. **Which chart will be useful for showing the composition**
   a) Area chart
   b) Box plot
   c) Pie chart
4. **Which magic function does memory profiling**
   a) %prun
   b) %load_ext
   c) %mrun
   d) None of the above

Answers to MCQ

1. a
2. d
3. c
4. c

Questions

1. Use profiling (both memory and time) on the function you wrote in *Chapter 3: Data Acquisition and Cleaning*, which performs rotation, zoom, and crop randomly.
2. Create a dataframe with missing values and fill it using median, mean, drop the missing. Then, create another variable that says if the value originally was missing.
3. Learn about wget, curl, grep, unzip, zip, tar utilities of Linux.

Key Terms

- Exploratory Data Analysis
- Continuous features, categorical features
- Cardinality

Chapter 5
Model Building and Tuning

In this chapter, we will investigate the model-building process. I mention investigating because the model selection and tuning (part of the model building process) is detective work. It's not an exact science, and many heuristics are involved. The way I approach any investigation process is simple, pros and cons. We will talk about the pros and cons of many approaches in this chapter. We will also talk about trade-offs. All of these may sound unscientific, and this is partially true also. We are going more into engineering in this chapter. Many readers may find this chapter highly opinionated since the process we will discuss here is experience-oriented and not exhaustive at all. By this, I mean there is more than one correct way to do this task of model building and tuning.

We will start the chapter with bias and variance, understand how the model selection is a way to minimize both. We will talk about model end goals and generalization. We will explore various validation frameworks, and a few other considerations often overlooked during model building but incredibly important when it comes to industry projects. Then, we will start exploring few good initial models. This will cut the time in choosing a model from a huge list of possible models. Finally, we will discuss various automation ideas for model selection and briefly discuss hyperparameter optimization techniques. So, let us dive in.

Structure

In this chapter, we will cover the following topics:

- Bias and variance
- Model selection
- Automating model selection
- Hyperparameters and how to tune them

Objective

After studying this chapter, you should be able to:

- Understand the importance of bias and variance in machine learning
- Understand and apply validation frameworks
- Build models to start with various problems
- Understand and apply hyperparameter tuning

5.1 Bias and variance

To understand bias and variance, let us take an example. The following data shows a distribution between weight and height:

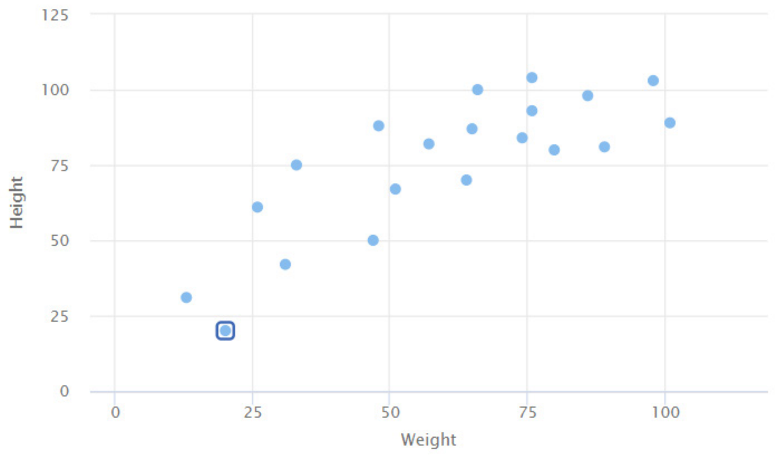


Figure 5.1*: Sample data distribution b/w weight and height*

The data shows that an increase in weight height usually increases (positive correlation). But after a certain weight, the increase in weight does not increase the height much. This portion of people *may* be considered obese (This does not guarantee these people are obese; they might have a lot of muscle mass (bodybuilders, for example)). Now, if we want to model this, then ideally, the curve that represents the distribution will be shown as follows:

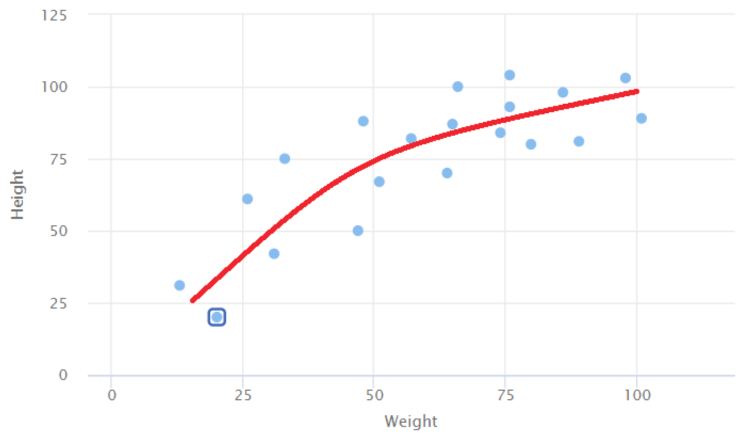


***Figure 5.2**: Ideal curve that represents the distribution*

The curve has an increasing trend initially, and after a certain weight, it does not increase much that captures the people falling under the red zone(obese).

This curve is ideal, but it doesn't know the data when we train a machine learning model. All it sees the data, and from that data, we want to create a model (in this case, a curve) that fits the data well. The following are two different models:

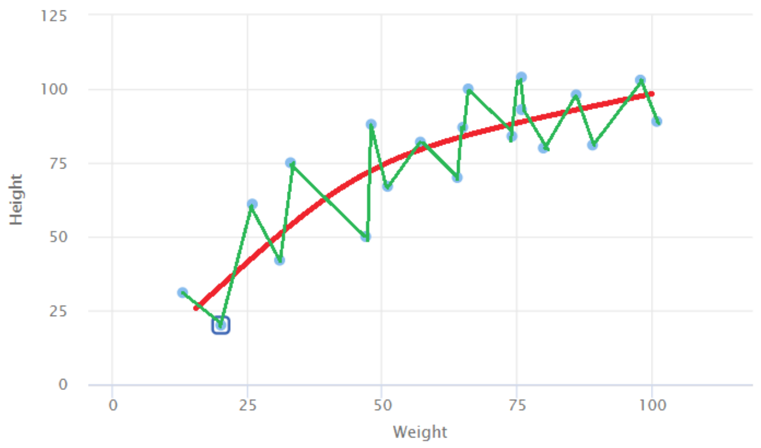


***Figure 5.3**: High bias scenario*

Suppose we use simple linear regression, which fits a line ($y=mx+c$) to the existing data using least square error. The blue line shows a line fitted with the data (not the least squares line). If we do this, then the error between our training data and the fitted line will be much worse than the ideal red line shown in the figure. So, in a way, we can say that our model *underfits* the data. This is a *high bias* scenario. So, one way to identify high bias or underfitting is to check the training error. If we have a high training error, then the model is underfitting the data.

Now consider the following example:

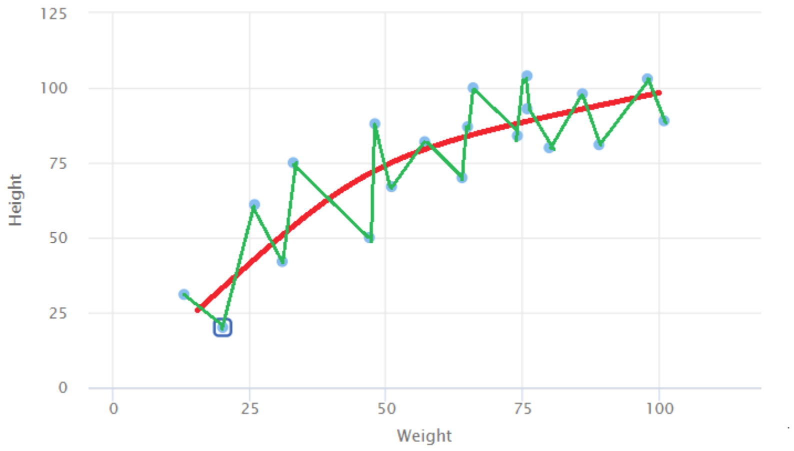


***Figure 5.4**: High variance scenario*

The green line here is another model that exactly fits the data. So, the training error here will be zero as the fitted curve passes through every point. But what's the problem here? Don't we want our model to fit the data well? The problem arises when unseen data appears (validation and test set). Since we are creating machine learning models to help us predict things in the future and not on the data it has already seen (training set), when the new data comes as the model is *overfitted* with the training data, the errors will be huge. Also, we won't be confident with our predictions as the model is overly sensitive to the input. This is undesirable and does not generalize well. So, one way to identify *high variance* or overfitting is to look for the error in validation and test data. Also, we can look for validation and test accuracy scores. If they are low, then it suggests overfitting.

One misconception is that if the validation accuracy (using accuracy here as an example, this is valid for any metric) is lower than training accuracy, then the model is overfitting. This is false. Training accuracy will be higher than validation accuracy as the model learns on the training data. Overfitting is there in the model when validation accuracy is much less and does not improve with more *epochs*.

5.1.1 Visualizing bias and variance in a machine learning model

Now let's take an actual model to see bias and variance in action. We will use decision trees for this example. Details of how the model works are not important for the discussion of bias and variance. A decision tree model is just a tree like the following:

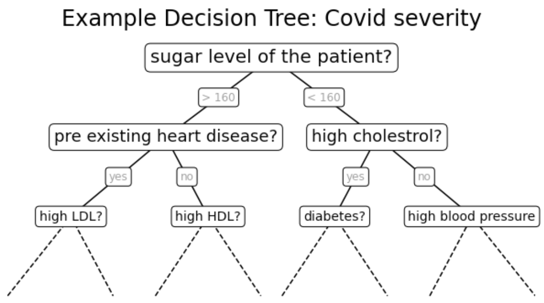

Example Decision Tree: Covid severity
sugar level of the patient?
> 160
< 160
pre existing heart disease?
high cholestrol?
yes
no
yes
no
high LDL?
high HDL?
diabetes?
high blood pressure

***Figure 5.5**: Example of a decision tree*

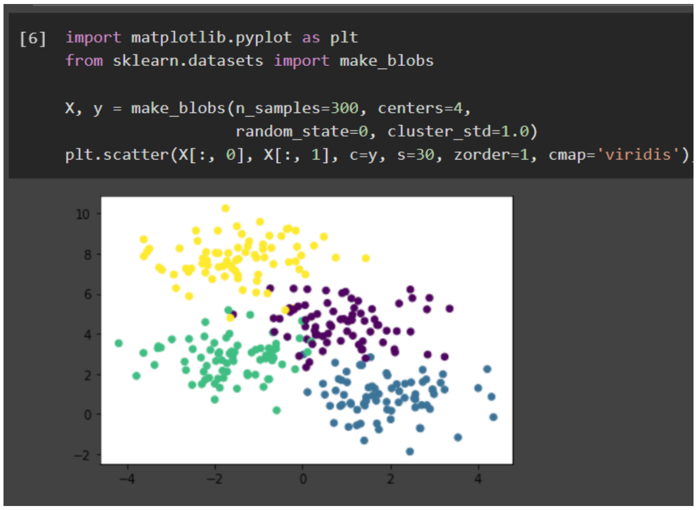
The preceding example is a decision tree; the image is self-explanatory; with input data, we can go along from top to bottom until we reach a leaf node (node with no children) to get a prediction. In this image, the last level is in dotted lines to indicate that the tree is continued. The details of how this tree will be created from existing data are unimportant and discussed in much detail during the second section of this book when we go through industry use cases.

But one hyperparameter to the model is depth; depth is the number of levels the decision tree has. We call this a hyperparameter because depth in a decision tree model is not learned with data. The user needs to specify it. And decision trees with different depths can be considered as separate models too. Unlike parameters, hyperparameters to a model are something that is not learned during training. These are often identified through some sort of search. We will talk about hyperparameter tuning in detail at the end of this chapter.

Now let's create some data to fit into a decision tree. The following is the code for creating some classification dataset:

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=300, centers=4,
                  random_state=0, cluster_std=1.0)
plt.scatter(X[:, 0], X[:, 1], c=y, s=30, zorder=3, cmap='viridis');
```

The output to the preceding code:



```
[6] import matplotlib.pyplot as plt
    from sklearn.datasets import make_blobs

    X, y = make_blobs(n_samples=300, centers=4,
                      random_state=0, cluster_std=1.0)
    plt.scatter(X[:, 0], X[:, 1], c=y, s=30, zorder=1, cmap='viridis')
```

Figure 5.6*: Sample dataset with five classes*

The preceding code is quite simple; we will use the **`make_blobs`** function from **`scikit-learn`** to create data with four different classes clustered together. We are creating 300 data points, and the **`cluster_std`** parameter controls the variance of these clusters. You can play around with these parameters on your own. Just put '??' right next to **`make_blobs`** in the jupyter notebook as discussed in *Chapter 1: Introduction to Machine Learning* to find out the function parameter and uses. Then we create a scatterplot using **`matplotlib`** with X and Y axis as the two features/independent variables. And we use different colors to show different classes or y or dependent variables. You can use different cmap or color maps for various color combinations.

Now fitting a decision tree is quite simple on this data. We need the following two lines of code:

```
from sklearn.tree import DecisionTreeClassifier
model= DecisionTreeClassifier(max_depth=3)
model.fit(X, y)
```

First, we import the required library for the decision tree from scikit-learn. Then specify the depth (here I am creating a depth 3 tree). Finally, invoke **`model.fit`**.

But to visualize the bias and variance, we need to do a little extra. First, we need to figure out a way to show decision boundaries. It is nothing but showing how the model is deciding the classes. One way is to show which areas of the figure belong to a particular class.

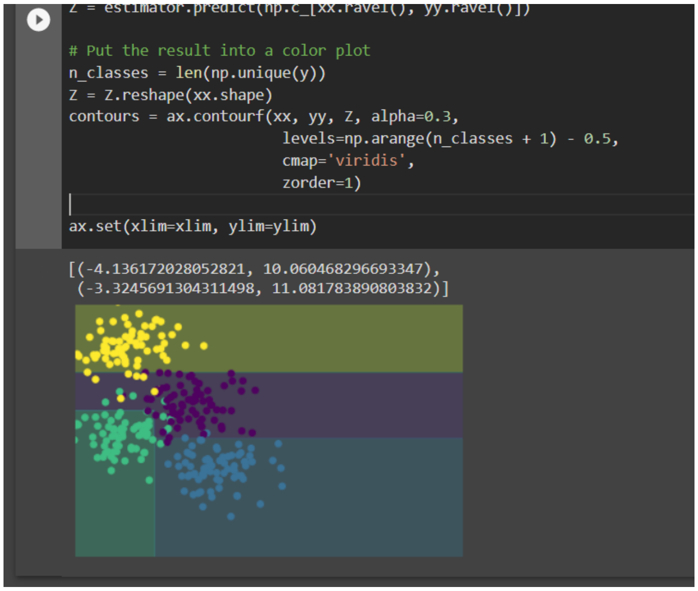
The following is the code to do this:

```
ax = plt.gca()

# Plot the training points
ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap='viridis',
            clim=(y.min(), y.max()), zorder=3)
ax.axis('tight')
ax.axis('off')
if xlim is None:
    xlim = ax.get_xlim()
if ylim is None:
    ylim = ax.get_ylim()

# fit the estimator
model.fit(X, y)
xx, yy = np.meshgrid(np.linspace(*xlim, num=200),
                      np.linspace(*ylim, num=200))
Z = estimator.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
n_classes = len(np.unique(y))
Z = Z.reshape(xx.shape)
contours = ax.contourf(xx, yy, Z, alpha=0.3,
                        levels=np.arange(n_classes + 1) - 0.5,
                        cmap='viridis',
                        zorder=1)

ax.set(xlim=xlim, ylim=ylim)
```

We first create the same scatter plot using the data created earlier. Then we get the ranges for the x and y axis. Then we will create a mesh of 200x200 datapoints in the x and y axis to represent the area. So, we will give 40000 datapoints the chart area for us to predict the class using a decision tree. This is done in lines 15-16 using `the` **`np.linspace`** method from NumPy(which creates 200 points in the range of xlim, i.e., range of min and maximum x and similarly for y axis).

Once we have these points from **`np.meshgrid`**, they will be in a matrix/array of 200, 200 shapes, but we need to make it 40000, 2 shape. This is done in line 17 using the .ravel method of NumPy and then concatenating using np.c_. I would suggest the user run the previous code line by line. And it was just printing out the shape of the variables to understand exactly what is happening here.

Finally, using the .contourf method of matplotlib, we create a filled contour plot to show the decision boundaries. The following is the output of the preceding code:



```
Z = estimator.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
n_classes = len(np.unique(y))
Z = Z.reshape(xx.shape)
contours = ax.contourf(xx, yy, Z, alpha=0.3,
                       levels=np.arange(n_classes + 1) - 0.5,
                       cmap='viridis',
                       zorder=1)

ax.set(xlim=xlim, ylim=ylim)
```

```
[(-4.136172028052821, 10.060468296693347),
 (-3.3245691304311498, 11.081783890803832)]
```

***Figure 5.7**: Decision boundaries*

The preceding figure shows the four areas as there are four classes. And as discussed previously, for an incoming point, the class will be determined by where the point lies. For example, if the point lies in the yellow area, it will be classified as yellow.

Now let's create the same plot for different depths. As discussed in *Chapter 1: Introduction to Machine Learning*, we should create functions multiple times when using a functionality multiple times. So, let's create a function that does the visualization part; the following is the code. I only took the previous code and put it inside a function. Also, I put the axis inside the function as a parameter if we want to create multiple plots and create boundaries using the tree's threshold parameter.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
def visualize_tree(estimator, X, y, boundaries=True,
                   xlim=None, ylim=None, ax=None):
```

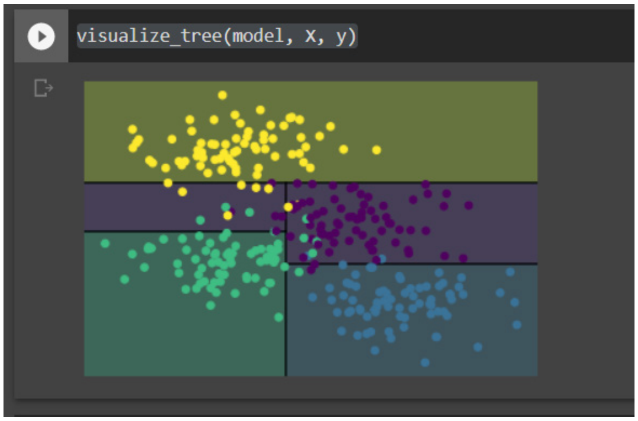
```
ax = ax or plt.gca()

# Plot the training points
ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap='viridis',
           clim=(y.min(), y.max()), zorder=3)
ax.axis('tight')
ax.axis('off')
if xlim is None:
    xlim = ax.get_xlim()
if ylim is None:
    ylim = ax.get_ylim()

# fit the estimator
estimator.fit(X, y)
xx, yy = np.meshgrid(np.linspace(*xlim, num=200),
                     np.linspace(*ylim, num=200))
Z = estimator.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
n_classes = len(np.unique(y))
Z = Z.reshape(xx.shape)
contours = ax.contourf(xx, yy, Z, alpha=0.3,
                       levels=np.arange(n_classes + 1) - 0.5,
                       cmap='viridis',
                       zorder=1)

ax.set(xlim=xlim, ylim=ylim)

# Plot the decision boundaries
def plot_boundaries(i, xlim, ylim):
    if i >= 0:
        tree = estimator.tree_
```

```
            if tree.feature[i] == 0:
                ax.plot([tree.threshold[i], tree.threshold[i]], ylim,
'-k', zorder=2)
                plot_boundaries(tree.children_left[i],
                                [xlim[0], tree.threshold[i]], ylim)
                plot_boundaries(tree.children_right[i],
                                [tree.threshold[i], xlim[1]], ylim)

            elif tree.feature[i] == 1:
                ax.plot(xlim, [tree.threshold[i], tree.threshold[i]],
'-k', zorder=2)
                plot_boundaries(tree.children_left[i], xlim,
                                [ylim[0], tree.threshold[i]])
                plot_boundaries(tree.children_right[i], xlim,
                                [tree.threshold[i], ylim[1]])

    if boundaries:
        plot_boundaries(0, xlim, ylim)
```

Now we can plot the preceding figure with the following code:

```
visualize_tree(model, X, y)
```

The output is as follows:



```
visualize_tree(model, X, y)
```

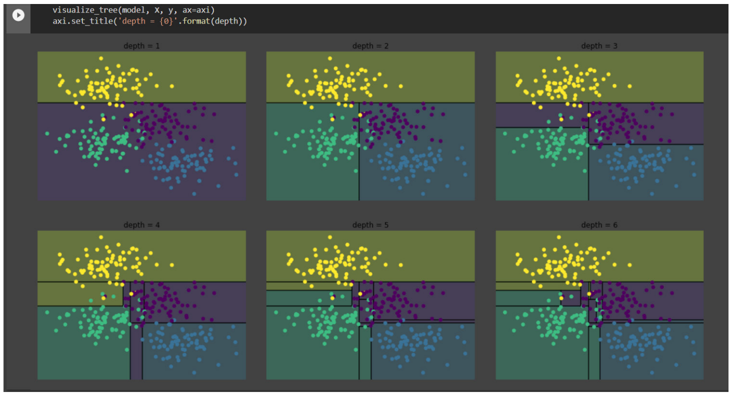
Figure 5.8: Using function and highlighting boundaries

So, let's create for different depths and see how the decision boundaries look with this code:

```
fig, ax = plt.subplots(2, 3, figsize=(16, 10))
fig.subplots_adjust(left=0.02, right=0.98, wspace=0.1)


for axi, depth in zip(ax.ravel(), range(1, 7)):
    model = DecisionTreeClassifier(max_depth=depth)
    visualize_tree(model, X, y, ax=axi)
    axi.set_title('depth = {0}'.format(depth))
```

We are going to use *subplots* to create multiple plots in the same figure. Here we are creating six figures in two rows and three columns. This is a useful method from matplotlib, so I suggest the user go through it in detail as we will be using it often.

Now using this, we will create six different plots with depth 1-6. The following is the output:



```
visualize_tree(model, X, y, ax=axi)
axi.set_title('depth = {0}'.format(depth))
```

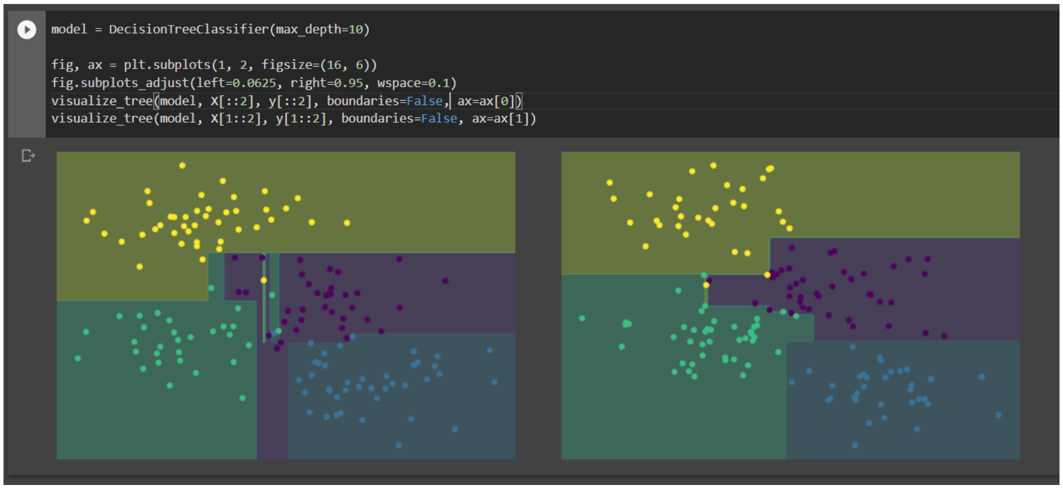
***Figure 5.9**: Visualizing with different depths*

Here we can see, for depth=1, there is only one node in the decision tree, and hence there can only be two classes from the decision tree to identify. So, the decision boundary is a straight line. This is a case of underfitting because the error will be huge here as all the green and blue points are incorrectly identified as purple. So, the training accuracy will be less showing underfitting or high bias.

Similarly, for depth = 6, there are too many boundaries, and the boundaries are overly sensitive to data. Another way to identify overfitting is to show the decision boundaries with a different set of data. Let us train our model with half of the original data with this code:

```
model = DecisionTreeClassifier(max_depth=10)
fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
visualize_tree(model, X[::2], y[::2], boundaries=False, ax=ax[0])
visualize_tree(model, X[1::2], y[1::2], boundaries=False, ax=ax[1])
```

The output to this code:



```
model = DecisionTreeClassifier(max_depth=10)

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
visualize_tree(model, X[::2], y[::2], boundaries=False, ax=ax[0])
visualize_tree(model, X[1::2], y[1::2], boundaries=False, ax=ax[1])
```

***Figure 5.10**: Overfitting*

Here we can see the decision boundaries are different with a different set of data. This shows the problem clearly. When new data comes, as the decision boundaries are overly sensitive to the training data (hence overfitting), the error will be huge for unseen data (validation, test data). This is how overfitting looks like.

Another way the reader can play around with the plots is to use the **`ipython`** interact. This will be an exercise for the user to create an interactive plot. The following is the code for how easily you can create an interactive plot for various depths:

```
from ipywidgets import interact


@interact(depth=(1, 10))
def tree_output(depth=5):
    clf = DecisionTreeClassifier(max_depth=depth, random_state=0)
    visualize_tree(clf, X, y)
```

It's just a single line of code, that is, a decorator @interact. The reader can use '??' right next to interact to identify how it works. This code will create the same graph with a slider for depth between 1-10 and increase the slider to see how the decision boundaries change. This will show how bias and variance come in the model and the sweet spot with less variance and less bias.

I go into so much detail about bias and variance because all of the model selection is a way to try and minimize both. We are now clear that bias and variance are related; if we create an overly complicated model (for example, high depth decision forest), then bias will decrease, but the variance will increase. And if we create an overly simple model (for example, depth = 1 or low depth in decision forest), it will make the model less variance, but it will introduce a lot of bias into the model, and the training error will be huge. Model selection is made to optimize the trade-off between bias and variance. And there are methods to orthogonalize this trade-off, techniques by which we can reduce variance without affecting the bias.

5.2 Model selection

In machine learning, selecting a model starts with a performance metric. We need a metric to evaluate a model. Now, this is not that trivial. Selecting a proper metric is one complex problem that we discussed in earlier chapters. But apart from that, optimizing a performance metric also contains few caveats. We cannot just divide our data into three separate sets (training, test, and validation) and compare the performance. For example, a lot of detail is still missing, how we will create these datasets from original data. What if we select a part of the dataset for validation that is not present during training time? All these questions need to be carefully answered during machine learning.

So, the first problem is how we should divide our dataset for training, test, and validation. There are several ways to do this:

- **Hold out validation**

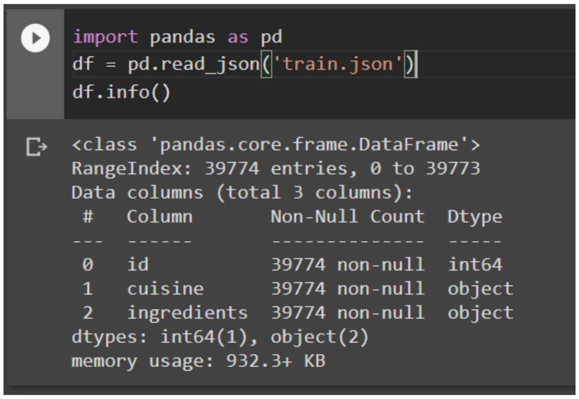
  Hold out validation is what we discussed in earlier chapters; randomly split and then keep a larger part of the dataset for training and a smaller part for validation. Popular split percentages are 80-20, 70-30, and so on. But this should not be done blindly; for time series data, we cannot randomly split as there is a time sequence involved. We may choose the first 80% as training and the last 20% data as validation.

  One more aspect is stratification, e.g., for highly imbalanced datasets where if we randomly split, it may make the split more imbalanced, and there is a chance we miss some of the classes altogether. So, in those types of datasets, be careful to use stratified split. Let's take an example:

  We will use the "*What's cooking?*" dataset from Kaggle. The link for this dataset is **https://www.kaggle.com/c/whats-cooking/data**; let's load the data into our Google Colab instance using the same trick discussed before using **`curlWget`**. Once downloaded, load the following code in a pandas dataframe.

  ```
  import pandas as pd

  df = pd.read_json('train.json')

  df.info()
  ```

The output to the code is as follows:



```
import pandas as pd
df = pd.read_json('train.json')
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 39774 entries, 0 to 39773
Data columns (total 3 columns):
 #   Column       Non-Null Count  Dtype 
---  ------       --------------  ----- 
 0   id           39774 non-null  int64 
 1   cuisine      39774 non-null  object
 2   ingredients  39774 non-null  object
dtypes: int64(1), object(2)
memory usage: 932.3+ KB
```

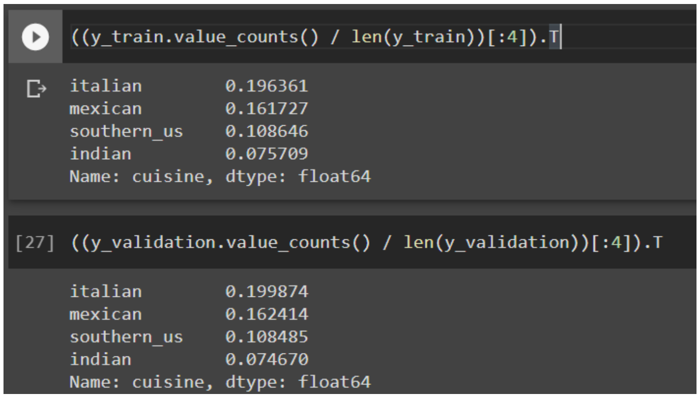
***Figure 5.11**: Loading "What's cooking?" dataset*

The data has 3 columns' id', 'cuisine', and 'ingredients'. Here the task is to identify the cuisine given the ingredients. So, our dependent variable or y is 'cuisine,' and the 'ingredients' is the feature/independent variable/X.

Now, let's split the data into 80-20 training and validation. (I am skipping the test set here as the competition itself provides one test set, so we need not create it ourselves.) Use the following code:

```
import pandas as pd

from sklearn.model_selection import train_test_split

X_train, X_validation, y_train, y_validation = train_test_
split(X, y, test_size=0.2)

y_train = pd.Series(y_train)

y_validation = pd.Series(y_validation)
```

Now, let's check the class percentage in 'cuisine' in both training and test.



```
((y_train.value_counts() / len(y_train))[:4]).T
```

```
italian        0.196361
mexican        0.161727
southern_us    0.108646
indian         0.075709
Name: cuisine, dtype: float64
```

```
[27] ((y_validation.value_counts() / len(y_validation))[:4]).T
```

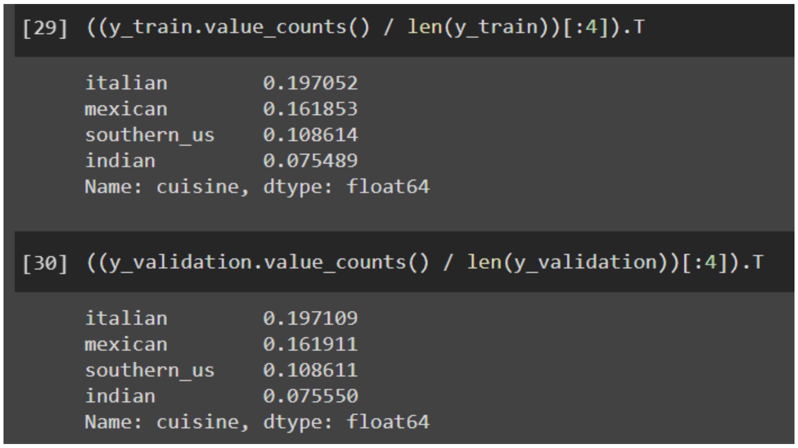
```
italian        0.199874
mexican        0.162414
southern_us    0.108485
indian         0.074670
Name: cuisine, dtype: float64
```

***Figure 5.12**: Class percentage by random split*

Here are the first 4 class percentages, and there is a slight discrepancy. Italian has 19.6% in training, but in test, it's almost 20%. The variation is not that much, but if we take a dataset with an extremely high imbalance, this becomes a huge issue. Now let's take the same percentage with stratified sampling. The following is the code for stratified sampling:

```
from sklearn.model_selection import train_test_split

X_train, X_validation, y_train, y_validation = train_test_
split(X, y, test_size=0.2, stratify=y)
```

We just need to specify the stratify parameter in **`train_test_split`** as y. Let us check the percentages now:



```
[29] ((y_train.value_counts() / len(y_train))[:4]).T

     italian        0.197052
     mexican        0.161853
     southern_us    0.108614
     indian         0.075489
     Name: cuisine, dtype: float64

[30] ((y_validation.value_counts() / len(y_validation))[:4]).T

     italian        0.197109
     mexican        0.161911
     southern_us    0.108611
     indian         0.075550
     Name: cuisine, dtype: float64
```

***Figure 5.13**: Class percentage by stratified split*

Now the class percentages are almost the same, Italian 19.7% in both train and validation and so on.

But what about the features? In hold out validation, since there is a single split, there is a chance that some of the data distribution might be lost in training/validation. We cannot do stratification in all the features as the feature space can be huge. This is one of the issues with hold out validation.

- **K Fold cross validation**

  One way to ensure the proper distribution of data is to use k-fold cross validation. It is done by basically shuffling the dataset randomly and then splitting it into k groups. Parameter 'k' is the user given, which is usually 5/10. So we do 5-fold or 10-fold cross validation. We take that group as validation for each group in this k group as validation and the rest groups as training. Then, evaluate the model performance on the validation set. We repeat this process on all of the k groups. Finally, we have 'k' different scores that we summarize (calculate mean, standard deviation, and so on). The following is a pictorial representation of K-Fold cross validation:

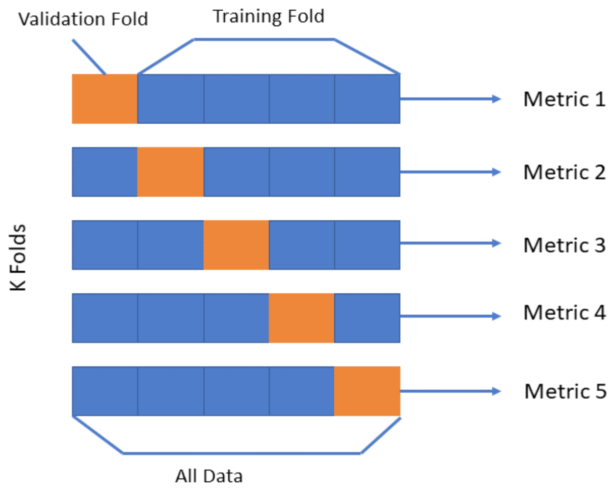


Figure 5.14*: K-Fold cross validation*

Importantly, this means the model is trained on each sample, and that sample is there once in the validation set. This eliminates the issue with the hold out validation. There are some extreme variations of k fold cross validation. For example, leave one out cross validation, where 'k' is the total number of records in the dataset, so one data point is taken as validation and the rest for training for each data point. Also, there is a stratified K-Fold, where we consider the stratification of the dependent variable. There is repeated K fold cross validation and nested K fold cross validation also. The interested reader may explore further these cross-validation techniques. But no matter what variant we use, cross validation also has a huge disadvantage: complexity. Nowadays, some deep learning models train for a week, so cross-validation is not feasible. But cross-validation is widely used for analytics/predictive problems where model complexity is not as much as with deep learning.

5.2.1 Considerations other than performance score during model selection

Apart from the performance score, we also need to consider few important aspects of model selection. These matter a lot when it comes to industry-level machine learning projects. The following are a few other important metrics to consider:

- **Speed**

  Speed is one of the most important metrics to consider. Usually, it is not a problem when the user consumes the model because of its inference/

prediction time. So, it's much faster than training, with huge models; however, this might not be true. Also, models that predict sentences or summarize a text may take a long time to process. In that case, managing latency is important.

Here we will discuss more on the speed at which we can train the model. One way is to look for models that can be easily parallelized. For example, for prediction problems, I generally choose random forests, xgboost. These algorithms can be parallelized easily. For deep learning, many models can be put on GPU. Even sequential tasks in natural language processing also, with the emergence of transformers (which we will learn about in detail and build a use case), can run those on GPU.

For parallel processing of data, libraries like Dask can be explored; in our industry use case, we will explore Dask with xgboost. The idea is to use parallel processing as much as possible to reduce training time.

Logging `time_taken` for each model is also good practice. Later we can refer to the amount of time taken by various models and then decide.

- **Explain-ability**

  Getting a good score out of a model is just the first step; many clients will ask for the reason for the prediction or explain-ability of the model. For many client's black box models (where internal workings are invisible and hard to explain the prediction) is a strict no go especially in industries where the decision is ethical like banking, healthcare, and so on. A bank cannot reject a loan application just because an algorithm said so. It also needs to explain why it was rejected. In all projects, it's good to understand the model predictions. Models like the random forest xgboost provide inbuilt options to do this. Also, model agnostic methods like LIME and SHAP works on any model. We will discuss all these in the upcoming chapters. There are techniques to explain a model for deep learning models, which we will explore in detail.

5.2.2 Good starting models

As we discussed, model selection is an experimental science, and a lot of trial and error is involved. But that does not mean we should explore everything. That would require a lot of time, and with huge datasets and complex deep learning models, sometimes time is a luxury we cannot afford. So, let's get started with few starting models for each category of problems.

5.2.2.1 Analytics problems

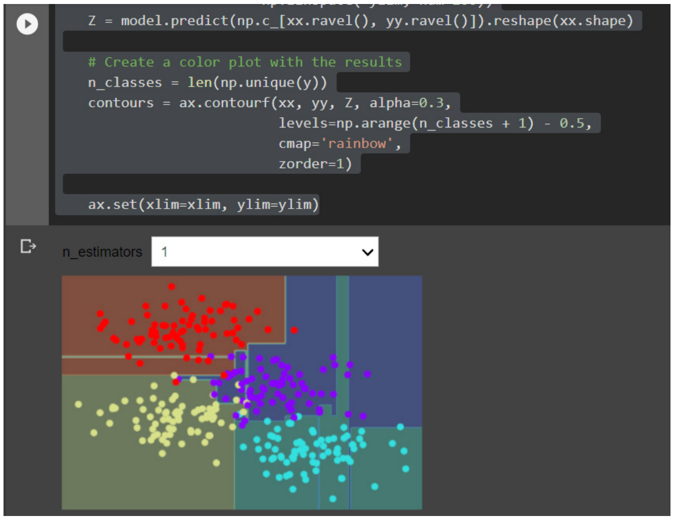
Analytics problems are those where some tabular data is involved, for example, predicting housing prices using the house demographics data. If I come across such a problem, the first thing I try is a random forest. It's easy to implement, can be trained in parallel, and provides an inbuilt mechanism to give feature importance. Then, I start to explore models such as xgboost, lightgbm, and so on. We will not go into the details of the inner working of any of these models in this chapter, but all these models have one thing in common, they are all ensemble models. Ensemble models are nothing but a way to *combine* multiple models.

For example, one simple way to combine multiple mobile predictions is to take a majority vote. This is also called the *wisdom of the crowd*. It's like how democracy works. The implementation is also quite simple; we will take multiple models and train them. Then, we will collect the predictions from each model and take a *mode* of those predictions (here I am talking about classification problem only, for regression problems where there is a continuous dependent variable, we will take median or mean). The *mode* of a set of data values is the value that appears most often, equivalent to majority voting.

Using an ensemble of models reduces overfitting/variance without affecting bias that much. Random forest is an ensemble model that creates multiple decision trees and combines their results. There is much more to the random forest than just combining the prediction results, but we will investigate ensemble working here. A random forest can be easily built using **`RandomForestClassifier`** from the **`sklearn.ensemble`** module. The **`n_estimator`** parameter determines the number of trees to create and use for the ensemble. Let's use interact discussed earlier and create an interactive plot that takes **`n_estimator`** as an input. The following is the code for the same. Much of the visualization code is the same that we used in the bias and variance example.

```
from ipywidgets import interact
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
import numpy as np

@interact(n_estimators=[1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100])
def visualize_bagging(n_estimators=1):
    ax = plt.gca()

    # Plot the training points
    ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap='rainbow', zorder=3)
    ax.axis('tight')
    ax.axis('off')
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # fit the estimator
    model = RandomForestClassifier(n_estimators=n_estimators, random_
state=0)
    model.fit(X, y)
    xx, yy = np.meshgrid(np.linspace(*xlim, num=200), np.linspace(*ylim,
num=200))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    # Create a color plot with the results
    n_classes = len(np.unique(y))
    contours = ax.contourf(xx, yy, Z, alpha=0.3,
                           levels=np.arange(n_classes + 1) - 0.5,
                           cmap='rainbow',
                           zorder=1)

    ax.set(xlim=xlim, ylim=ylim)
```

The output for this program:



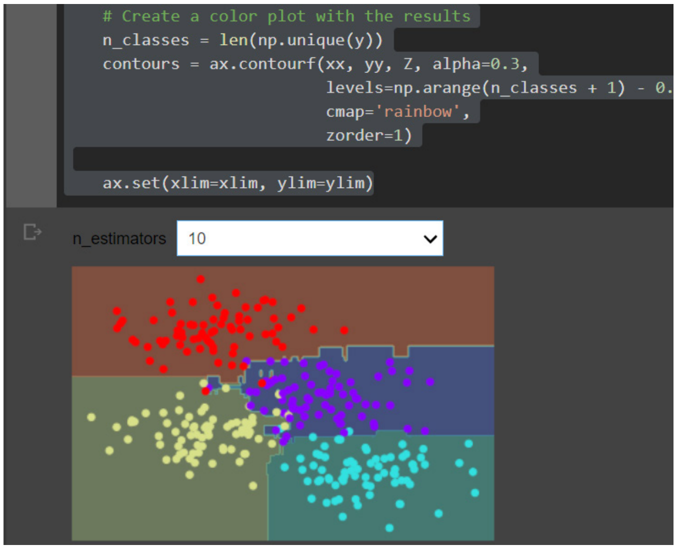
```
Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

# Create a color plot with the results
n_classes = len(np.unique(y))
contours = ax.contourf(xx, yy, Z, alpha=0.3,
                       levels=np.arange(n_classes + 1) - 0.5,
                       cmap='rainbow',
                       zorder=1)

ax.set(xlim=xlim, ylim=ylim)
```

n_estimators 1

***Figure 5.15**: Random forest with one estimator*

Here we can select multiple **`n_estimator`** from the dropdown box. With only one decision tree, you can see a clear case of high variance as discussed earlier in the bias and variance section. Now, let's change it to 10 trees.

The following is the output:



```
# Create a color plot with the results
n_classes = len(np.unique(y))
contours = ax.contourf(xx, yy, Z, alpha=0.3,
                       levels=np.arange(n_classes + 1) - 0.
                       cmap='rainbow',
                       zorder=1)

ax.set(xlim=xlim, ylim=ylim)
```

n_estimators 10

__Figure 5.16__: Random forest with ten estimators

You can immediately see the improvement in the decision boundaries. It's much smoother now, eliminating the variance. Similarly, for more trees, the boundaries will become smoother and smoother, but there won't be much change in the boundary even when we increase **`n_estimators`**. This is because of saturation. After a while, even adding more trees won't change the crowd's opinion, which we call overpopulation. I would suggest the reader play around as much as possible with this and change the limits of **`n_estimators`** or add a slider, and so on.

Similarly, xgboost and lightgbm are libraries that use a different type of ensembling called **boosting**. To put it simply, boosting is creating multiple weak models that sequentially improve the error from the earlier model. Finally, the overall model becomes strong. We will go through much more in detail about boosting and the algorithms like xgboost and lightgbm in the second section of this book.

There are other ensembling methods such as stacking, ranked averaging, blending, and so on. which are mostly used in competitions but not much in practical applications. But we will go through examples of those in the second section too.

Another good starting model for tabular data is using neural networks. We will use the **`fastai`** library to create neural network models that can provide state-of-the-art results when exploring the industry use case. Most people avoid neural networks in tabular data because traditional methods outperform neural network models. Also, explain-ability is much less in neural network models. But we will see how neural networks can be used effectively for tabular problems too.

5.2.2.2 Image and text problems

For cognitive problems like Computer Vision or natural language processing, the first thing to look for is transfer learning. Now, let us understand transfer learning when a model is re-used/re-purposed for a different task than originally trained on. So, it allows a model to be more effective by not starting from scratch. For example, we know how a tv/computer screen looks. It's a rectangular box with a thin border. But we already know what a rectangular box looks like, and we also know what borders are. We have past knowledge about what different shapes look like. We also know what a rectangle is, what edges are, and so on. These are transferred knowledge from the past making our, ability to detect remarkably effective and perform tasks even with little data.

Similarly, transfer learning takes knowledge from past models trained on huge datasets and then somehow trains new models with fewer data. The details of how it works will be discussed in detail in the Computer Vision use case in *Chapter 8: Building a Custom Image Classifier from Scratch*. But the main idea is remarkable. Creating models on top of earlier models gives a huge advantage as the training can now be faster and with fewer data. So, we should always start with transfer learning models. Also, even when we have a lot of data, we still need not start from scratch.

For Computer Vision problems, I generally start with some variant of RESNETs (which is trained on ImageNet and uses the model for transfer learning) for image classification tasks. If it is a segmentation task, I take Unets. For text data, transformers are the first thing that we should try. They are powerful and easy to access with many pre-trained models (we can do our transfer learning). In *Chapter 8: Building a Custom Image Classifier from Scratch* and *Chapter 9: News Summarization App using Transformers*, we will go through all these models in much more detail and understand the inner workings of these models. The following is a small diagram showing a general idea about transfer learning:

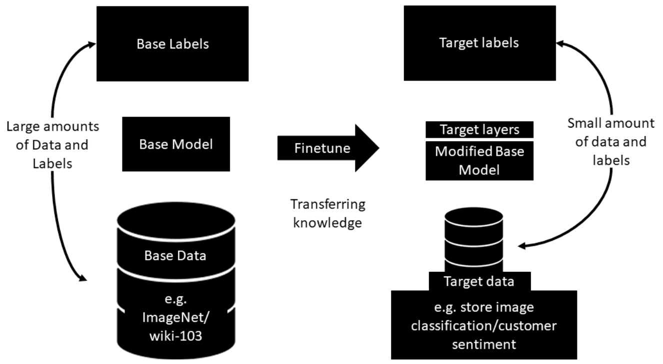


***Figure 5.17**: Transfer learning*

If this does not work or our dataset is completely new and doesn't fit into any existing models that we can use for transfer learning, we should still use well-known architectures in deep learning. We will come across many such architectures later in this book.

5.3 Hyperparameters and how to tune them

Hyperparameters are also like dials and knobs. But unlike model parameters/ weights, they are not learned from data like the depth parameter we discussed for decision trees. It's something the user provides, and we can try various hyperparameter settings for the model and identify the best-performing model by **searching** for the proper hyperparameter setting. Here I used the word searching rather than learning since hyperparameters are searched based on the performance metric and not learned like parameters. So how do we search for hyperparameters? A few approaches, namely grid search and random search are used extensively in the machine learning world. These searches are not intelligent, which means we don't adjust our search space based on the results. Another set of searches include some intelligence but are not that extensively used due to the complexity. So, let us discuss more on these approaches.

- **Grid search**

  Grid search is simple; I think of it as an outer loop for each parameter search space. For example, let us consider a model m1 with two hyperparameters p1 and p2, where we want to search p1 within and and similarly for p2, we want to search within and . Then, using grid search we can achieve it with following pseudo code,

```
for  in range( , ):
      for  in range( , ):
            try model with p1 = , p2 =
            save model result
```

  Select the model with the best result

  And with more hyperparameters, there will be more loops. Let us do this in a code. Here we will go for the iris dataset that comes along with the **`scikit-learn`** package. The following is the code for loading it:

```
from sklearn import datasets
iris = datasets.load_iris()
import pandas as pd
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['target'] = iris.target
```

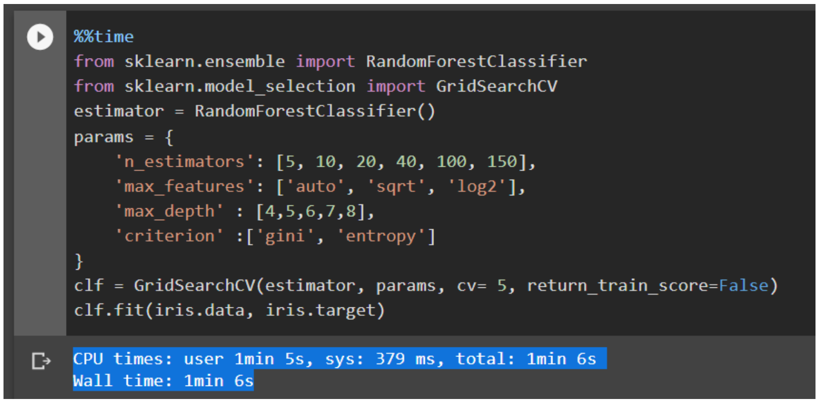
```
    df['target'] = df['target'].apply(lambda x: iris.target_names[x])
```

The code from 1-5 is self-explanatory. In line 6, we map the actual class names since the target contains only the codes. Here I use to apply and a lambda function to do this. This combination of applying with lambda function is used a lot in data munging.

Now, let's perform a grid search on random forest model on various hyperparameters. Ignore what exactly each hyperparameter does at this moment.

```
%%time
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
estimator = RandomForestClassifier()
params = {
    'n_estimators': [5, 10, 20, 40, 100, 150],
    'max_features': ['auto', 'sqrt', 'log2'],
    'max_depth' : [4,5,6,7,8],
    'criterion' :['gini', 'entropy']
}
clf = GridSearchCV(estimator, params, cv= 5, return_train_score=False)
clf.fit(iris.data, iris.target)
```

The output to the code:



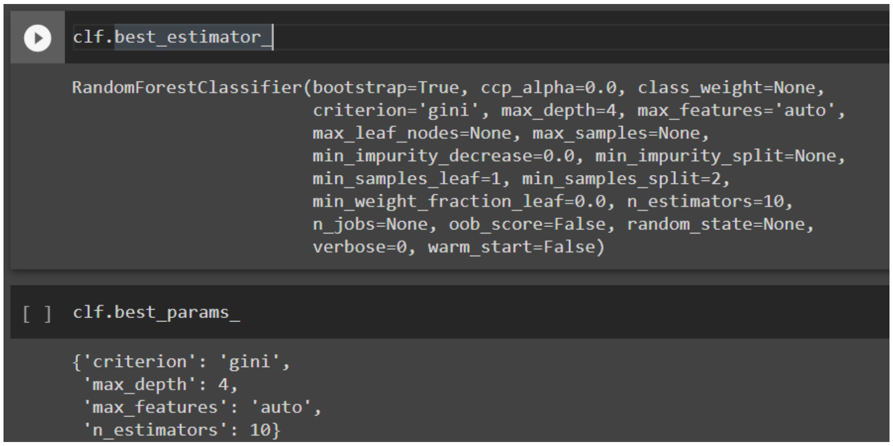
```
%%time
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
estimator = RandomForestClassifier()
params = {
    'n_estimators': [5, 10, 20, 40, 100, 150],
    'max_features': ['auto', 'sqrt', 'log2'],
    'max_depth' : [4,5,6,7,8],
    'criterion' :['gini', 'entropy']
}
clf = GridSearchCV(estimator, params, cv= 5, return_train_score=False)
clf.fit(iris.data, iris.target)
```

```
CPU times: user 1min 5s, sys: 379 ms, total: 1min 6s
Wall time: 1min 6s
```

Figure 5.18*: GridSearchCV implementation*

So, it took 1 min 5 seconds to run. Now the hyperparameters that we want to search here are given in the params dictionary. These are the **`n_estimators`**, **`max_features`**, **`max_depth`**, criterion. All of these are hyperparameters to a random forest. We use scikit-learn GridSearchCV to perform the grid search on these parameters. The run results are stored in the **`clf.cv_results_`** parameter. The reader can convert it into a dataframe and check the results. This is also an exercise for this chapter. I would suggest the reader play around with the code as much as possible.

We can just run clf.best_estimator_, clf.best_params_ to see the best model and parameters.

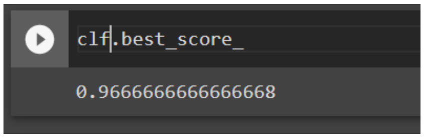


```
clf.best_estimator_

RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=4, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=10,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

```
[ ] clf.best_params_

{'criterion': 'gini',
 'max_depth': 4,
 'max_features': 'auto',
 'n_estimators': 10}
```

***Figure 5.19**: Best estimator and parameters*

The grid search went through all hyperparameter ranges and identified preceding values for the four hyperparameters that we want. **`clf.best_score_`** contains the best score among all these settings. Here is the output for **`clf.best_score_`**:



```
clf.best_score_

0.9666666666666668
```

***Figure 5.20**: Best Score*

But if you run it with a larger grid, it will take a long time to run as it goes through all combinations of hyperparameters. This is a caveat for a grid search. As we are searching over the whole space, it takes a long time to run. Also, we used cross validation 5-fold for the reliability of performance, as discussed on K-Fold cross validation.

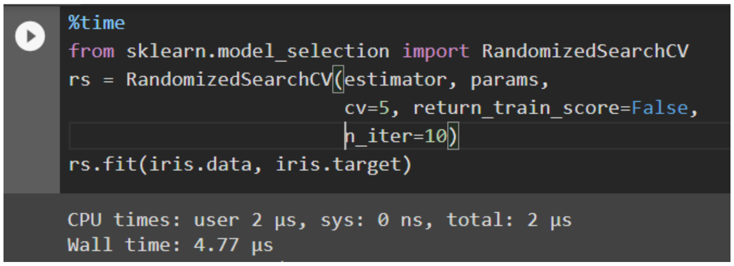
- **Random search**

  Grid search is slow as it explores every possible setting. So other than important search space, it spends a lot of time exploring unimportant search space too. Also, for large datasets and complex models where there are many tuneable parameters using grid search may take days, and we have not talked about a number of hyperparameters. For some algorithms like xgboost, there are many hyperparameters; imagine running so many nested loops.

The grid search is nested loops for each hyperparameter. To solve this, Bergstra and Bengio proposed a simple idea in their paper "*Random Search for Hyperparameter Optimization*." The idea is don't look for all possible space and rather randomly pick. This will reduce the time by avoiding those unimportant reasons. And it works like a charm. Generally, with 20-30 trials, we get a good set of hyperparameter to work with. It is not optimal like grid search, but it's amazingly fast. The following is the code:

```
%time
from sklearn.model_selection import RandomizedSearchCV
rs = RandomizedSearchCV(estimator, params,
                        cv=5, return_train_score=False,
                        n_iter=10)
rs.fit(iris.data, iris.target)
```

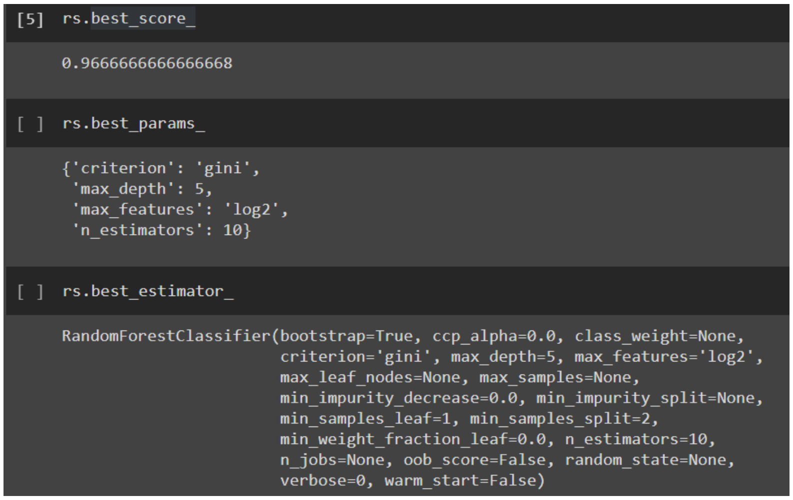
The output for the preceding code:



```
%time
from sklearn.model_selection import RandomizedSearchCV
rs = RandomizedSearchCV(estimator, params,
                        cv=5, return_train_score=False,
                        n_iter=10)
rs.fit(iris.data, iris.target)

CPU times: user 2 µs, sys: 0 ns, total: 2 µs
Wall time: 4.77 µs
```

***Figure 5.21**: Best score*

This only took 2 microseconds compared to 1 min 5 seconds using gridsearch. Now, let's look into the results:



```
[5] rs.best_score_

    0.9666666666666668

[ ] rs.best_params_

    {'criterion': 'gini',
     'max_depth': 5,
     'max_features': 'log2',
     'n_estimators': 10}

[ ] rs.best_estimator_

    RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                           criterion='gini', max_depth=5, max_features='log2',
                           max_leaf_nodes=None, max_samples=None,
                           min_impurity_decrease=0.0, min_impurity_split=None,
                           min_samples_leaf=1, min_samples_split=2,
                           min_weight_fraction_leaf=0.0, n_estimators=10,
                           n_jobs=None, oob_score=False, random_state=None,
                           verbose=0, warm_start=False)
```

***Figure 5.22**: Random search results*

It gives the same accuracy for a different setting as gridsearch with much less time. And believe it or not, it happens often. So always start with a random search as it saves time. But random search might give sub-optimal results, so here is the strategy I follow:

1. Use random search first with a bigger hyperparameter space because it is fast
2. Use the results of a random search to build a focused hyperparameter grid for grid search
3. Do this for all the promising results from random search to zone in into the optimal hyperparameters

It's a coarse to fine approach to finding the hyperparameters. First, use a faster method (Random Search) to find coarse results. Then, use an optimal but slow method (Grid Search) to find finer results.

But we have not discussed the hyperparameter search for deep learning algorithms. And we are going to do it in the Computer Vision use case as it has a lot of details, and without understanding how deep learning works, it might be confusing for the reader.

5.4 Automating model selection

If using the initial set of models does not yield the required results, then we need to try other models, and in machine learning, the number of models you will get is huge. There is no way to select a model by trying it one by one and tuning separately. So, we need to automate the process. Now I use the following approach to automate model selection:

1. **Use AutoML libraries**

   First, I go for AutoML libraries like auto-sklearn, auto-keras, and so on. Less line of code and spend more time on features and problem understanding. I am not adding any code here; these libraries are made so that users can build models with little effort. So, I am putting this part of another exercise for the user.

2. **Do it manually on models**

   For few problems, we need to do the model selection manually. And we can do it in a simple for loop that wraps our earlier hyperparameter training code. The following is the code for the same. Here I am using **`svm`**, **`random_forest`** and logistic regression as candidate models.

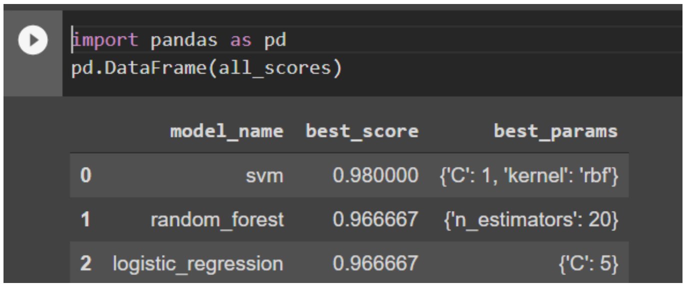
```
from sklearn.linear_model import LogisticRegression
from sklearn import svm
from sklearn.ensemble import RandomForestClassifier
params = {
    'svm': {
        'model': svm.SVC(gamma='auto'),
        'params': {
            'C': [1, 10, 20],
            'kernel': ['rbf', 'linear']
        }
    },
    'random_forest': {
        'model': RandomForestClassifier(),
        'params': {
            'n_estimators': [1, 10, 20]
        }
    },
    'logistic_regression': {
        'model': LogisticRegression(solver='liblinear', multi_
class='auto'),
        'params': {
            'C': [1, 5, 15]
        }
    }
}
all_scores = {'model_name': [], 'best_score': [], 'best_params':
[]}
for model_name, model_data in params.items():
  clf = GridSearchCV(model_data['model'], model_data['params'],
cv=5, return_train_score=False)
  clf.fit(iris.data, iris.target)
  all_scores['model_name'].append(model_name)
  all_scores['best_score'].append(clf.best_score_)
  all_scores['best_params'].append(clf.best_params_)
```

Here I am using the **`params`** dictionary to save the three models and associated hyperparameter search. After the grid search, I am saving all the best scores and parameters for each model in the **`all_scores`** dictionary. We can then create a pandas dataframe using this variable. This is another good practice. Whenever you make a model selection, save the results in a dataframe and then use libraries like qgrid and pivotable to interact with them. Also, summarizing and grouping in a dataframe is much easier.

The following is the code to create the dataframe and visualize it:

```
import pandas as pd

pd.DataFrame(all_scores)
```

The output for the preceding code:



```
import pandas as pd
pd.DataFrame(all_scores)
```

	model_name	best_score	best_params
0	svm	0.980000	{'C': 1, 'kernel': 'rbf'}
1	random_forest	0.966667	{'n_estimators': 20}
2	logistic_regression	0.966667	{'C': 5}

Figure 5.23: *Model selection manually*

Here SVM performs better with the settings in **`best_params`**. You can use many more models and many different parameters. And then just run it overnight to get results. But remember to log everything, and also, it's a good practice to save these results periodically as a safeguard when a system crashes.

3. **Kitchen sink ensemble approach**

   If you are going for a Kaggle competition or any other machine learning hackathon where the sole purpose is to get a better performance metric, I would have enough models and their output after trying the initial models and AutoML and manual search ensemble them. This is what we call the kitchen-sink approach. Just dump a lot of models. But this is neither ideal nor recommended during an actual project. Hence, I am not going into details of it here. Interested readers can read about this approach in "*strategies for hackathon*" in the appendix.

We optimize more than just a model. If a particular set of models does not work, then we also need to change our data. We also need to change our problem statement sometimes. So, a good machine learning practitioner keeps track of all these too.

Conclusion

With this, we conclude *chapter 5*. To summarize, we started with understanding bias and variance and how it is the key trade-off to build good machine learning models. Then we learned about the model selection process by understanding performance metrics and validation frameworks. Then we discussed the various starting models to work with to reduce the selection process and learned about kitchen sink and auto ml approaches if we fail to select an initial model. Finally, we glanced into what hyperparameters are and different ways to tune them. In the next chapter, we will learn about the often overlooked aspect of machine learning: taking our model into production and various guidelines to smooth the process.

Points to remember

The following are a few points to remember:

- Decreasing bias generally increases variance and vice versa. Hence there is a trade-off.
- Model selection starts with validation performance, but speed and explainability are also important.
- Ensemble models are a good start with tabular data as they reduce overfitting without affecting bias much, can be trained in parallel.
- Transfer learning is a good start for cognitive problems as they allow knowledge transfer and give good results even with fewer data.
- If these do not work, then one idea is to use AutoML to try many models or write a kitchen sink script to select a model. Finally, for competitions where validation metric is all we need to optimize, ensembling many non-correlated models may do the trick.
- Grid search is slow; random search works fine. But if we can afford grid search, go for it.

MCQ

1. **High training set error refers to**
    a) High Variance
    b) High Accuracy
    c) Low Bias
    d) High Bias

2. **Low validation accuracy refers to**
   a) High Variance
   b) High Accuracy
   c) Low Bias
   d) High Bias

3. **Which parameters handle class imbalance during splitting dataset in the train_test_split method?**
   a) Random_state
   b) stratify
   c) skiprows

4. **The mechanism where we create k groups in data and use each group as validation and the rest k-1 group as training in repeat is**
   a) Hold one out validation
   b) Human in the loop validation
   c) Stratified hold out validation
   d) K Fold cross validation

Answers to MCQ

1. d
2. a
3. b
4. d

Questions

1. Write an interactive visualization code for Random Forest, where both **`n_estimators`** and dataset size are user input.
2. Create a class-based code for hold out and K-Fold cross validation; use the random forest model.
3. Explore the **`lightgbm`** package; create a classification model on the Titanic dataset.

4. Save the output into a dataframe and explore qgrid, pivotable.js to summarize the results. The starter code can be found in *section 5.3 Hyperparameters, and how to tune them* in the 'grid search' topic.
5. Explore the auto-sklearn package; try it for the iris dataset. Readers who have prior knowledge in deep learning can also explore autokeras.

Key Terms

1. Bias and variance trade-off
2. Ensemble models
3. Hyperparameters
4. Transfer learning
5. AutoML
6. Grid search and random search

CHAPTER 6
Taking Models into Production

In this chapter, we are going to explore model deployment. Until now, we were only working on our models offline; that is, we were training, testing our model/ data on our environment. No external user has interacted with the model yet. In this chapter, this is going to change. We are going to introduce outside users. Now, we cannot expect a user to create Jupyter notebooks to use our model. Also, the user is not concerned with the overall metric we have. They will expect an accurate result that can add value to their project. Now, machine learning imposes some unique challenges with this. First is technology; since Python is relatively slow, many modern-day apps do not use it in production. Second is we might not require a powerful system to deploy apps. Still, some machine learning models are so computation-heavy that it requires a lot of processing power to predict (although this is rarely a problem as the most computation-heavy part of the machine learning model is the training part, which is done before taking the model into production). In this chapter, we are going to explore tools and techniques to handle this.

Another key idea is to understand the model results and show them to the user. For example, it's one thing to provide a credit score using a bank user's historical data and a whole other thing to explain why the credit score is low, exactly what aspects (features) of the user data influence the score more. These inferences/interpretations are key to adding business value. We are going to study these interpretation tips.

Finally, we will make our model iteratively better by doing proper analysis on model errors. And we are going to understand why we should treat our model as a piece of code that requires versioning and how tools like MLFlow and Streamlit are going to help us in that and other aspects of the machine learning process.

Structure

In this chapter, we will cover the following topics:

- How and where to deploy a ML model?
- Model interpretation tips
- Managing model versions

Objective

After studying this chapter, you should be able to:

- Understand how we can take our model into production, various technologies, and tools that streamline this process
- Understand how to interpret model results and properly present this to the user
- Understanding how we can test our model online/after putting it in production
- Managing machine learning process, versioning, retraining, and logging

6.1 How and where to deploy a ML model?

In *Chapter 5: Model Building and Tuning*, we learned about training and fine tuning a model. Although we merely touched upon the surface of model selection and hyperparameter tuning, but believe it or not, we are now ready to put our model into production, where real people can use it. We can either have a web application like a website. The user can input the relevant values to get an appropriate output or make a desktop application ship executable. In this section, we will talk more about the former approach, and we will create a website that can serve the model. And model deployment has more to do with software engineering than machine learning. So, in this chapter, we will learn more about software engineering than machine learning. Also, a real website has a lot of complexity. We need to understand security, frontend, backend, DevOps (scaling, redundancy, availability, and so on). Here, we will not learn about it all because that will be beyond the scope of this book.

First, let us talk about technology; until now, all our code is written in Python. And much of the machine learning code is written in Python. R is another preferred language for data science. But much of the production machine learning code is written using Python. And with a well-established Python community, it is a great option too for finding relevant code, debugging, and so on. So, Python is our go-to choice for the implementation of the machine learning code for production.

Next is the framework; we need a backend framework to host the model. We are going to use FastAPI for this. Other options include Flask, Django, and so on. We go for FastAPI because it's based on Starlette, which supports asynchronous calls, and it's faster than Flask or Django. Interested readers may research more into all these backend frameworks but going through all these in detail is beyond the scope of this book. All we need is a faster and modern framework that can serve our model.

Here we are not interested in the front end. Our task is to run model prediction and send it to be used later to create meaningful UI. It does not mean we cannot create the front end easily. All you need to make a good-looking website is to know where to look what. For example, you can easily make a website using bootstrap, simple JQuery (just for making AJAX calls to the API). But all of this is at a beginner level only. For complex UI with multiple features and dynamic components, we will require some framework. But generally, using output from a machine learning model and showing it in a better-looking way at a beginner level is easy to do. We will create a UI for our machine learning model in *Chapter 9: News Summarization App using Transformers*. Here we will talk more about making a backend that will take the user input, process it, run model prediction, and send the output back to the user.

FastAPI is a fast web framework for building Application Programming Interfaces or APIs. In a nutshell, API is an interface between user interactions and a piece of software. This is a way of abstracting the underlying implementation and only exposing objects or actions the user needs. Using an API is simple; we don't want the users to worry about how and where our models work. The '*how*' part hides the implementation of our model, which is important but not as important as the "*where*" part. The main reason for making our machine learning code a separate API is to create independent environments/machines. As we discussed earlier, machine learning is mostly done (and in this book entirely done) using Python. But Python is slow and often not preferred in production for making large-scale applications. Also, we need JavaScript to create web apps anyway. And any UI-related tasks are difficult and inefficient to achieve with Python. Another reason is machine learning has its difficulties, best practices, and guidelines. So, it's better to keep it separate. Now, let's build an example application with FastAPI and try to deploy it on the web.

6.1.1 Building and deploying a machine learning model

Let's take a classification example; the data we will use comes from Bank Note Authentication UCI ML Repository Dataset in Kaggle, **https://www.kaggle.com/ritesaluja/bank-note-authentication-uci-data**. This data contains four feature columns: variance, skewness, kurtosis, entropy, and the target variable '*class.*' These features are derived from wavelet transformation of banknote images. The task is to identify if it's a genuine or a fake currency note. So, our end goal is to create an

API that takes these feature values as input and uses the model to predict the class, whether it's a genuine or a fake note. We will follow the steps to create a model that can predict user data and create an API using a FastAPI package and deploy it in Heroku.

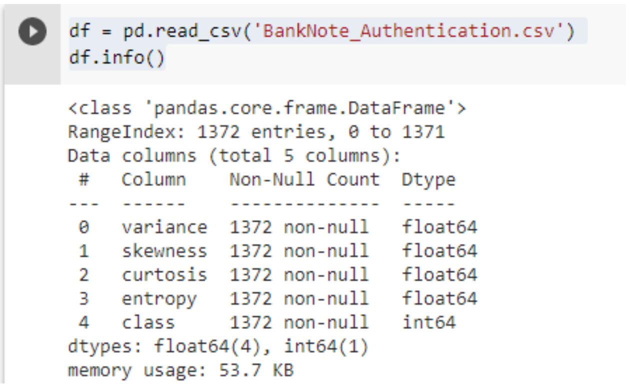
- **Loading data, importing packages, and pre-processing data**

  First, let's load the data using the CurlWget trick we learned in *Chapter 3, Data Acquisition and Cleaning*. Once we get that data, we need to unzip it to get the CSV file **`BankNote_Authentication.csv`**.

  Now, let's import the relevant packages:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
Now, let's load the data and check information using pandas:

df = pd.read_csv('BankNote_Authentication.csv')
df.info()
```

  The output of the code:



```
df = pd.read_csv('BankNote_Authentication.csv')
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1372 entries, 0 to 1371
Data columns (total 5 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   variance  1372 non-null   float64
 1   skewness  1372 non-null   float64
 2   curtosis  1372 non-null   float64
 3   entropy   1372 non-null   float64
 4   class     1372 non-null   int64
dtypes: float64(4), int64(1)
memory usage: 53.7 KB
```

Figure 6.1: *Banknote authentication dataset info*

  Here we have four features as discussed and a total of 1372 data points. It's a small dataset just for showing the model deployment process.

The next few lines of code just split the data into training and testing datasets.

```
X, y = df.iloc[:, :-1], df.iloc[:, -1]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_
size=0.25)
```

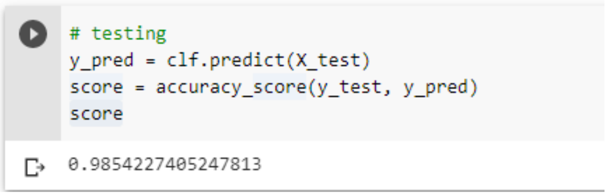
There is no more pre-processing required for this data as all the features are numerical in nature.

1. **Training model and validation (offline)**

Now we can use the **`X_train`**, **`y_train`** to train our model and validate using **`X_test`**, **`y_test`**. We use the hold out validation here just for simplicity.

```
# training
clf = RandomForestClassifier(n_estimators=20)
clf.fit(X_train, y_train)


# testing
y_pred = clf.predict(X_test)
score = accuracy_score(y_test, y_pred)
score
```

The output of the code:



```
# testing
y_pred = clf.predict(X_test)
score = accuracy_score(y_test, y_pred)
score
```

```
0.9854227405247813
```

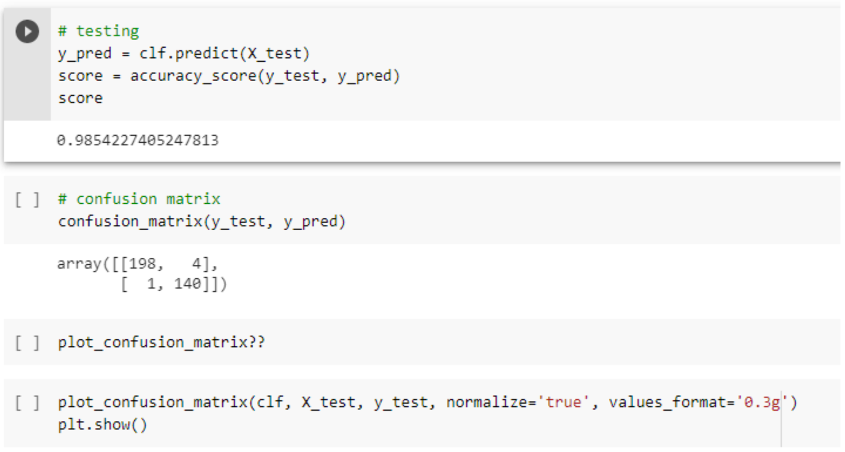
***Figure 6.2**: Score of the random forest model*

This model gives a score of around 0.98, which is good enough for serving a model. We can also check the confusion matrix to see how many errors our model made and plot it using **`plot_confusion_matrix`**.

The following is the code for the same:

```
# confusion matrix
confusion_matrix(y_test, y_pred)


plot_confusion_matrix(clf, X_test, y_test, normalize='true',
values_format='0.3g')
plt.show()
```

The output to the code:



```
# testing
y_pred = clf.predict(X_test)
score = accuracy_score(y_test, y_pred)
score

0.9854227405247813
```

```
[ ]  # confusion matrix
     confusion_matrix(y_test, y_pred)

     array([[198,   4],
            [  1, 140]])
```

```
[ ]  plot_confusion_matrix??
```

```
[ ]  plot_confusion_matrix(clf, X_test, y_test, normalize='true', values_format='0.3g')
     plt.show()
```

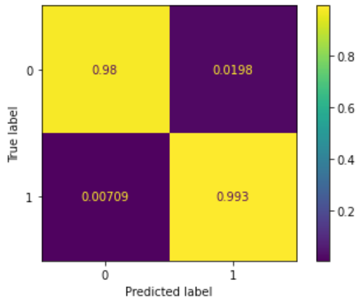


***Figure 6.3**: Confusion matrix*

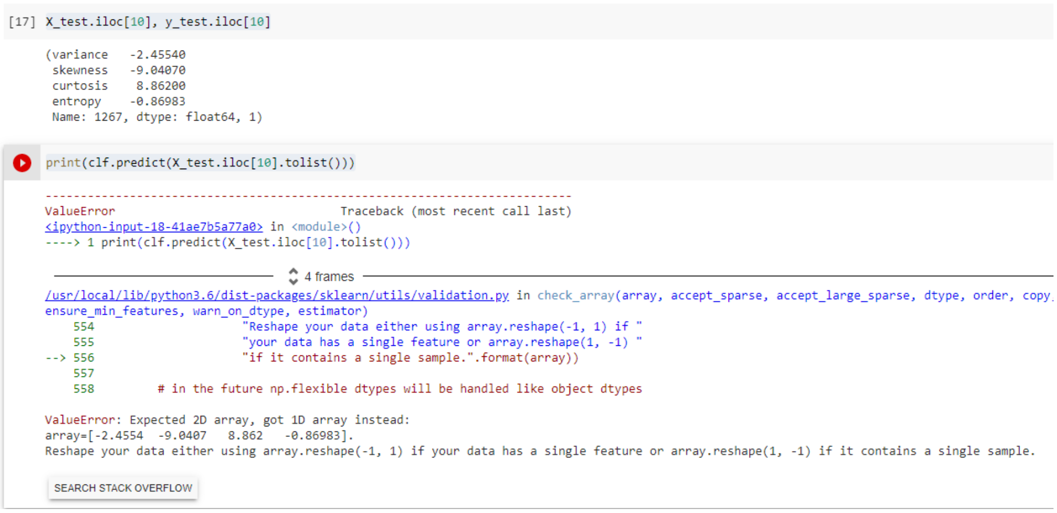
Here, seeing the confusion matrix, around 98% of class 0 is correctly identified, and 99.3% of class 1 is correctly identified. The reason for me showing this graph is to show a way to visualize the confusion matrix. Here I use the **`plot_confusion_matrix`** function and used normalize='true' parameters to calculate the percentages. You can also use "?? Trick" (In Jupyter Notebook, putting a double question mark after any function/object/method to check source code) to see the inner workings of the function. One key takeaway is that the machine learning field is so much developed that there is no single way to do a task. You can have multiple packages to do some tasks in a better way. We could have only used a confusion matrix to see the numbers, but a better way is to plot it in a heatmap with percentages for better understanding. Always look for these ways. As the field is changing and new ways are getting discovered, you can always do a bit of research and find a better way to do things.

2. **Model prediction/inference**

Now let's take a datapoint from the test set and try to predict it with the following code:

```
X_test.iloc[10], y_test.iloc[10]
print(clf.predict(X_test.iloc[10].tolist()))
```

The output of the code:



```
[17] X_test.iloc[10], y_test.iloc[10]

     (variance   -2.45540
      skewness   -9.04070
      curtosis    8.86200
      entropy    -0.86983
      Name: 1267, dtype: float64, 1)
```

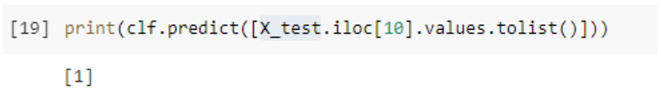
```
print(clf.predict(X_test.iloc[10].tolist()))

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-18-41ae7b5a77a0> in <module>()
----> 1 print(clf.predict(X_test.iloc[10].tolist()))

 4 frames
/usr/local/lib/python3.6/dist-packages/sklearn/utils/validation.py in check_array(array, accept_sparse, accept_large_sparse, dtype, order, copy,
ensure_min_features, warn_on_dtype, estimator)
    554                     "Reshape your data either using array.reshape(-1, 1) if "
    555                     "your data has a single feature or array.reshape(1, -1) "
--> 556                     "if it contains a single sample.".format(array))
    557 
    558         # in the future np.flexible dtypes will be handled like object dtypes

ValueError: Expected 2D array, got 1D array instead:
array=[-2.4554  -9.0407   8.862   -0.86983].
Reshape your data either using array.reshape(-1, 1) if your data has a single feature or array.reshape(1, -1) if it contains a single sample.
```

Figure 6.4: *Dimensionality error due to prediction*

Here we select the 10th instance from the test set to predict with the model. The expected output is class 1. This is how we expect the user to give data too. But if we directly use it to predict the model, we get an error with dimension. This is because the model is trained with multiple inputs as a matrix or a 2D array. And we are passing a 1D array of values to the model to predict. So, we can easily solve this by adding another dimension to our data. The following is the code for the same:

```
print(clf.predict([X_test.iloc[10].values.tolist()]))
```

The output to this code:



```
[19] print(clf.predict([X_test.iloc[10].values.tolist()]))

     [1]
```

Figure 6.4 (a): *Adding another dimension to our data to predict*

So, the model is now able to give the correct prediction as 1. But notice the model gives the data in an array, too, because in our training example, the output comes in the form of an array and not a single value (scalar).

3. **Saving model**

We can save our model because we don't want to retrain it every time a user asks for a prediction. And saving the model allows us to move the serialized model to a different location too. The following is the code for saving the model:

```
import pickle
with open("model.pkl", "wb") as wf:
  pickle.dump(clf, wf)
```

Here we use a pickle module to save the model object. Later we can load the model with the following code:

```
model_path = "model.pkl"
with open(model_path, 'rb') as rf:
  model = pickle.load(rf)
```

Now the problem with saving the model this way is that it carries no metadata information. For example, what is the version on which this model is trained? That is particularly important, because we should not train a model with a different version of scikit-learn and then use another version to predict. It might cause errors. What about old models? What if we want our model, not more than a few years old? Also, what if we only want this model to run with Python 3.6 version? These are called metadata, and when we pickle the model without it, we may run into problems. The following is a function I wrote to solve some of these problems:

```
import datetime
from pathlib import Path
import json
import uuid

def save_mode_with_metadata(model, path=None):
  model_id = str(uuid.uuid4())
  timestamp_format = "%m/%d/%Y, %H:%M:%S"
  timestamp = datetime.datetime.utcnow().strftime(timestamp_
format)
  model_metadata = {
      "sklearn_version": sklearn.__version__,
      "python_version": platform.python_version(),
      "timestamp_format": timestamp_format,
      "timestamp": timestamp,
      "model_id": model_id
  }

  if not path:
    path = Path().resolve()

  (path/model_id).mkdir()

  with open(path/model_id/f"{model_id}.pkl", 'wb') as wf:
    pickle.dump(model, wf)

  with open(path/model_id/f"metadata_{model_id}.json", 'w') as
wf:
    json.dump(model_metadata, wf, indent=4)

  print(f"Saved model in {path/model_id} folder")
```

Here I save the scikit-learn version, Python version, timestamp, **`model_id`**, and so on, creating the model id automatically using UUID4. This model metadata I am saving in a JSON file next to the model. So, we can write another function that loads the model only if this metadata match with the environment it runs in.

This does not solve all issues with serialization. But at least this is a start. We will also use MLFlow later in this chapter to save these metadata and parameters to the model.

4. **Creating API**

We are going to use FastAPI as discussed. First, we need to install FastAPI and Uvicorn using pip. You can follow the tutorial (**https://fastapi.tiangolo.com/tutorial/**) for creating a simple FastAPI application; the code is simple, only five lines:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello World"}
```

We can save it in **`main.py`**, and now we can simply run this code using a uvicorn server by running uvicorn main:app –reload. I am not going into the details about the code as you can always refer to the documentation of FastAPI in case you face issues, and that is the best practice. Software engineering is an ever-changing field. Innovations happen every hour, so the best way to get up to date is simply by doing.

Now, you can run your first API with FastAPI by opening the browser and just typing **http://127.0.0.1:8000**. It will return you a JSON response as **`{"message": "Hello World"}`**. Also, it creates an OpenAPI documentation page if you select **`http://127.0.0.1:8000/docs`**. The following is a screenshot of the page:

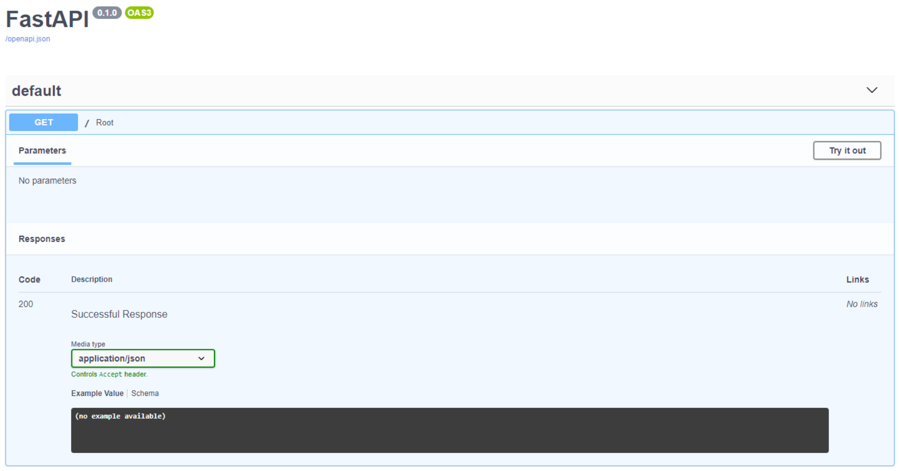
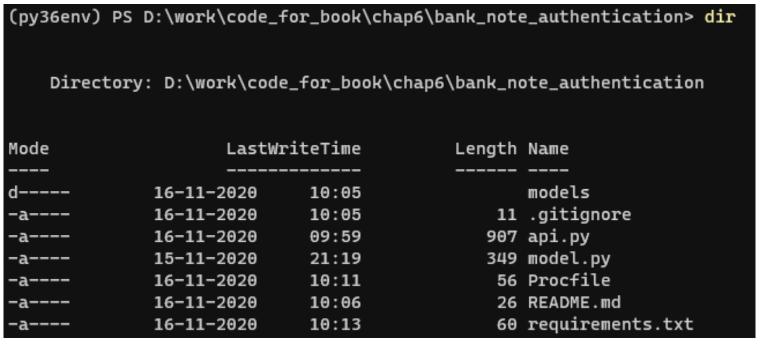


***Figure 6.5**: FastAPI documentation*

Isn't it great? We didn't write any code to create this documentation; we can use it to try the API endpoints we created. This also helps cross-team development. Since the documentation is very readable, someone using our API would find this page extremely helpful. Now, let us create the code for our model. The following is a screenshot of the directory structure I created:

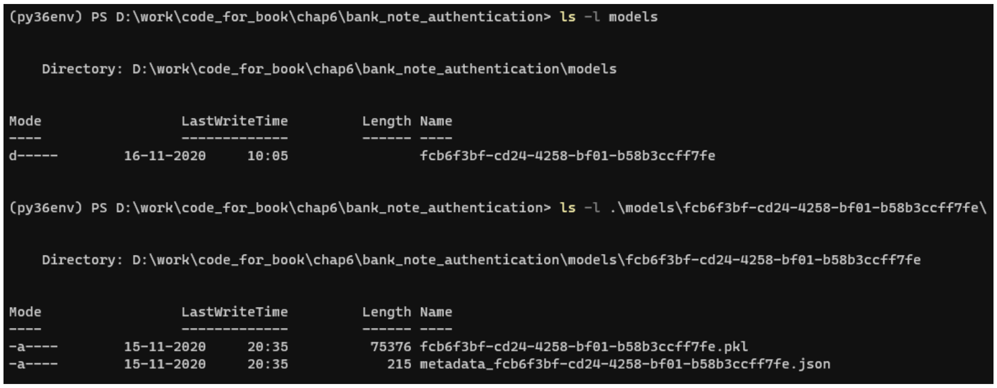


```
(py36env) PS D:\work\code_for_book\chap6\bank_note_authentication> dir


    Directory: D:\work\code_for_book\chap6\bank_note_authentication


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        16-11-2020     10:05                models
-a----        16-11-2020     10:05             11 .gitignore
-a----        16-11-2020     09:59            907 api.py
-a----        15-11-2020     21:19            349 model.py
-a----        16-11-2020     10:11             56 Procfile
-a----        16-11-2020     10:06             26 README.md
-a----        16-11-2020     10:13             60 requirements.txt
```

***Figure 6.6**: Directory structure of our project*

I didn't spend much time creating the structure since this is not a very complicated project. Later in *Chapter 8, Building a Custom Image Classifier from Scratch*, we will go through unit testing. The structure of the project will be there be a lot more standardized. Let us go through each of these folders/ files and understand what it is trying to do.

5. '**models' folder**: This folder contains our model file; here is a screenshot of the contents of this folder:



```
(py36env) PS D:\work\code_for_book\chap6\bank_note_authentication> ls -l models


    Directory: D:\work\code_for_book\chap6\bank_note_authentication\models


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        16-11-2020     10:05                fcb6f3bf-cd24-4258-bf01-b58b3ccff7fe


(py36env) PS D:\work\code_for_book\chap6\bank_note_authentication> ls -l .\models\fcb6f3bf-cd24-4258-bf01-b58b3ccff7fe\


    Directory: D:\work\code_for_book\chap6\bank_note_authentication\models\fcb6f3bf-cd24-4258-bf01-b58b3ccff7fe


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        15-11-2020     20:35          75376 fcb6f3bf-cd24-4258-bf01-b58b3ccff7fe.pkl
-a----        15-11-2020     20:35            215 metadata_fcb6f3bf-cd24-4258-bf01-b58b3ccff7fe.json
```

Figure 6.7*: Contents of the models folder*

We have our model folder that contains both the model pickle file and the metadata.

6. '**`api.py`**' file: This file contains the code for our API; the following is the code:

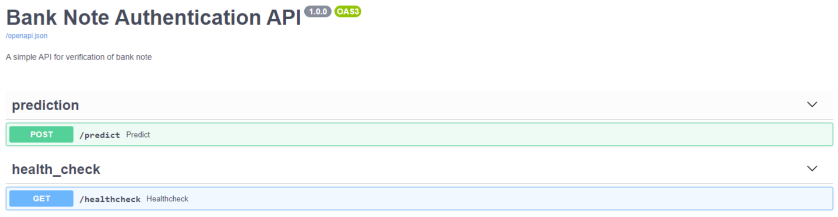
```
from typing import List
import numpy as np
from fastapi import FastAPI
from pydantic import BaseModel
from model import load_model

app = FastAPI(
    title="Bank Note Authentication API",
    description="A simple API for verification of bank note",
    version="1.0.0",
)
model = load_model()

class PredictAuthenticationRequest(BaseModel):
    data: List[List[float]]
```

```
class PredictAuthenticationResponse(BaseModel):
    data: List[float]

@app.post("/predict", response_
model=PredictAuthenticationResponse, tags=['prediction'])
async def predict(input: PredictAuthenticationRequest):
    X = np.array(input.data)
    y_pred = model.predict(X)
    result = PredictAuthenticationResponse(data=y_pred.tolist())
    return result

@app.get('/healthcheck', status_code=200, tags=['health_check'])
async def healthcheck():
    return 'Bank Note Authentication Classifier is ready!'
```

Here the main code for prediction uses the function predict; this takes input in a List[List[float]], this is in line with our model input. Remember earlier in the chapter; we discussed adding another dimension to our 1D array. Hence, we need a list of lists as an input to the model. The **`PredictAuthenticationRequest`** class just ensures the input is a list of lists of floats. If you pass anything other than this format, you will get an error from the API server.

So, once we have our data, we can convert it into a NumPy array and then use our model to predict (the model is loaded using the **`load_model`** function, which is there in the model.py file). Then the prediction result is given using the class that also ensures the data is in the List[float] format, as our model output format is a list of floats.

Apart from this, we code for giving a title and description to the API. This is just for documentation in OpenAPI. Also, the tags in **`@app.post/@app.get`** are for documentation only. If your run uvicorn –api:app –reload to run the server, you will get the following output:


Bank Note Authentication API 1.0.0 OAS3
/openapi.json
A simple API for verification of bank note

prediction

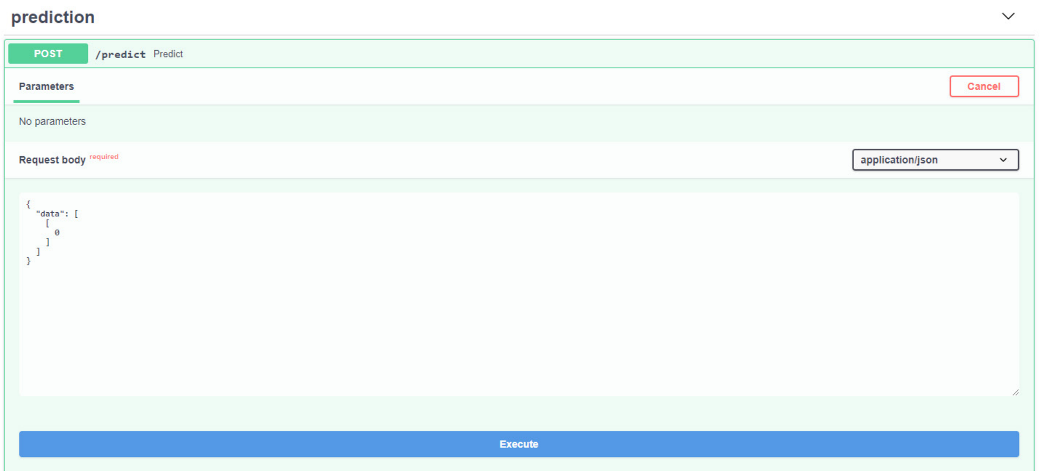
POST /predict Predict

health_check

GET /healthcheck Healthcheck

Figure 6.8: Documentation page for our API

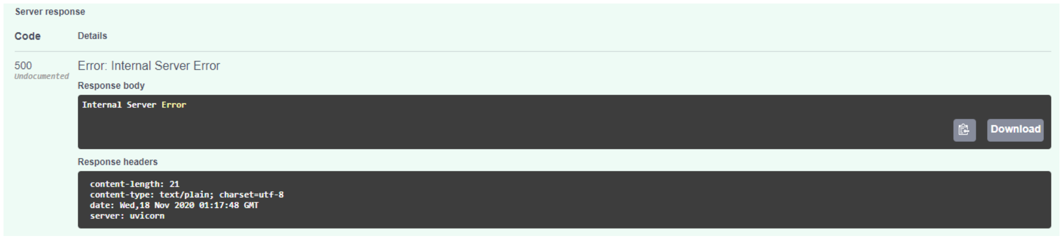
The **`health_check`** endpoint is just for checking if the server is live or not. The code for it is straightforward. Just returning a text with status code 200. If you now open up the /predict POST call, you can give some example value. The following is the screenshot:



```
{
  "data": [
    [
      0
    ]
  ]
}
```

Figure 6.9*: Using Prediction API*

Try passing various values and see if the API works, and check what happens if we pass a list of strings or any different format other than List [List[float]]. You will see an output error code 422, Unprocessable Entity. The reason for that is we are using pydantic to do some data validation for us. So, if the input format is anything other than mentioned, it will give us a 422 error code. But it won't break the code. These validations are important because we can never trust the user. Anything can be thrown at the API, and it should be robust enough to handle it. For example, our code will break if we pass a lower dimension data because our model expects four values (variance, skewness, kurtosis, entropy). See what happens if we only pass three values. The following is the output you will get:



```
Internal Server Error
```

```
content-length: 21
content-type: text/plain; charset=utf-8
date: Wed,18 Nov 2020 01:17:48 GMT
server: uvicorn
```

Figure 6.10*: Passing input with the wrong dimension*

This is bad; this means our code is unable to handle the incorrect input. You can see where it fails if you open up the prompt where you ran the uvicorn command. It will give you an error like this, "*ValueError: Number of features* *of the model must match the input. Model n_features is 4 and input n_features is 3"*. But this is a server-side error, and the user doesn't know if he/she/it (the API call from UI code) made this error. It will just receive a 500-error code which means Internal Server Error. So, we need to handle such situations too.

This is where FastAPI shines. Pydantic offers an easy and powerful way to handle any validations needed. Here I am not going to do the validation for dimensionality and encourage the reader to do it. It will give some exposure to FastAPI. If you could not do it, then you can find dimensionality validation in *Chapter 8: Building a Custom Image Classifier from Scratch* and *Chapter 9: News Summarization App using Transformers*.

7. **'model.py' file**: This file contains the code for loading the model; the following is the code:

```
import pickle

import numpy as np

from pathlib import Path


from sklearn.ensemble import RandomForestRegressor


MODEL_ID = "fcb6f3bf-cd24-4258-bf01-b58b3ccff7fe"

MODEL_PATH = Path().resolve()/"models"/f"{MODEL_ID}/{MODEL_ID}.
pkl"


def load_model():

    with open(MODEL_PATH, 'rb') as rf:

        model = pickle.load(rf)

    return model
```

We are using the pickled model file to load the model. Here we are not using the metadata to do further validation about the environment. But we can write a simple function to check the metadata. This **`load_model`** is used by the api.py file to load the model.

I am not going into the details of the other files. These are just extra files for different purposes; for example, '.gitignore' specifies which files to ignore when we push it to git, 'ProcFile' is for Heroku deployment and contains the UVICORN command, i.e., *web: uvicorn api:app --host=0.0.0.0 --port=${PORT:-5000}*, 'README.md' contains some info about our API, 'requirements.txt' contains the python package requirements.

8. **Deploying to Heroku**

   We are all set to deploy our website to Heroku; first, push the preceding code into a GitHub repository. Then create a Heroku account. After you are done with the account creation and activation (through email), you can go to the Heroku dashboard at **https://dashboard.heroku.com/apps**.

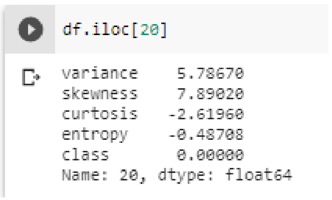
   Here if you previously used Heroku, you can find your apps listed. For our project, click on "**`New`**" and select "**`create a new app`**," give an app name, I gave mine "**`banknoteauthenticationapi`**."

   You can select the deployment method as GitHub. Connect to GitHub for the first time (it will ask for a GitHub login). Then you can select the **`repo_name`** that you created for the preceding project. Click on connect.

   We can skip automatic deployment for the first time, manual deploy, and click on deploy branch.

   After some time (installing dependencies, creating VM, and so on.), your site will be ready at <given_name>.herokuapp.com/docs. See how easy that was? And I intentionally didn't add any screenshots because the UI of Heroku might change, but it is intuitive, and you can easily deploy it. My site is now live at **https://banknoteauthenticationapi.herokuapp.com/docs/**

   Now let's use one example from the dataset, that is, the data at index 20:

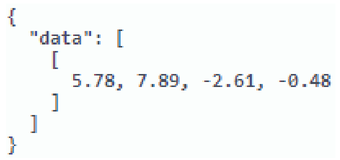


```
df.iloc[20]
```

```
variance    5.78670
skewness    7.89020
curtosis   -2.61960
entropy    -0.48708
class       0.00000
Name: 20, dtype: float64
```

***Figure 6.11**: An instance from dataset to test our API*

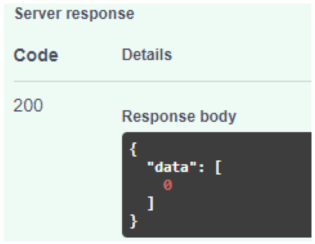
   Here is the request body screenshot:



```
{
  "data": [
    [
      5.78, 7.89, -2.61, -0.48
    ]
  ]
}
```

***Figure 6.12**: Request body for testing*

The output we get from this:


Server response

Code Details

200

Response body

```
{
  "data": [
    0
  ]
}
```

Figure 6.13: *Output from the server*

This is what we expect as the class for it was 0 in the dataset.

With this, we are now done with the deployment of the model in production. There is a lot of improvement to be made to the API code, which we will go through in detail in the upcoming chapters when we are going through a business use case.

6.2 Model interpretation tips

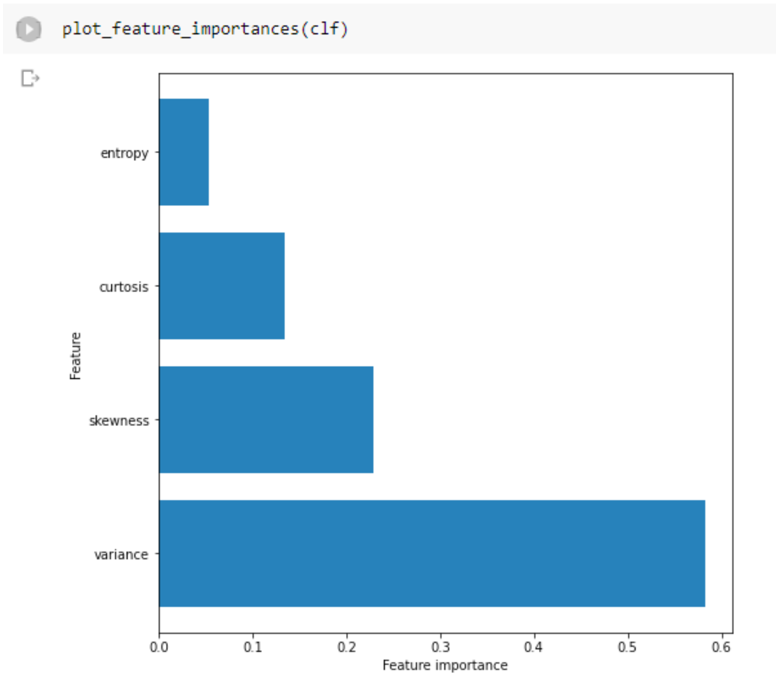
In our preceding API, we are giving an output to the user based on the input provided. But that is not enough. The user may expect *how confident our model is with this prediction. What features are important to this prediction? Why did our model predict this output?* All of these come under model interpretation. This is one of the most overlooked areas of machine learning. We cannot just expect a user to be happy with the prediction we gave. We also need to explain why it is. There are use cases where this is not needed, for example, for hackathons. The sole purpose is to make the model perform better and not worry about interpretation. Still, even then, if we understand why the model predicts an output, we can get valuable information about the data we have.

So, let us go through the questions one by one. *What about the confidence in the prediction?* One way to do that is to include a probability score for the prediction and the prediction; in our random forest example, we can use the **`predict_proba`** method to get the probability score for each class prediction. This way, we can say to the user that the prediction is X with probability p. This will give additional information to the user about how strongly a model feels about its prediction. What features are important to this prediction?

Next comes *What features are important to this prediction?* This question can also be answered in various ways, the reason for using tree-based approaches like random forest is we can get feature importance score for each feature. We will go through the inner workings of a random forest model in *Chapter 7: Data Analytics*, and there we will understand how random forest identifies the feature importance. But for now, let us just look into the code to plot feature importance; it's simple:

```
def plot_feature_importances(model):
    n_features = X.shape[1]
    plt.figure(figsize=(8,8))
    plt.barh(range(n_features), model.feature_importances_,
align='center')
    plt.yticks(np.arange(n_features), X.columns.values)
    plt.xlabel('Feature importance')
    plt.ylabel('Feature')

plot_feature_importances(clf)
```

The **`model.feature_importances_`** contains the feature importance of each of the features. We use a bar plot to show that value for each feature. The output of the code:



```
plot_feature_importances(clf)
```

***Figure 6.14**: Feature importance plot*

This plot is quite easy to understand; we can see that variance is the most important, followed by skewness, curtosis, and entropy.

We can use different techniques to identify the feature importance for models other than tree or linear regression where in-built feature importance is not there. They are quite straightforward too. One way is to remove the feature completely and then validate the model. Now add it back to see how much the model improves. Another way is to shuffle the feature randomly and to the same process to see what the gain in model performance is.

Now, let us come to the third question, *why our model predicted this output?* This is where we will use some model prediction interpretation techniques like *SHapley* Additive *exPlanations* or SHAP or Local Interpretable Model-agnostic Explanations or LIME. Both are model agnostic methods, which means we need not understand or care about the model's inner workings to interpret the results. Let's first start with SHAP. SHAP comes from a game theory concept. We explain the model output based on Shapley values.

The intuition for Shapley values is simple; let us take an example. Three friends, Manish, Vinod, and Shashi, are going to the same office. They plan to share a cab. And the total cost came out to be X. Now how would they split the charge? We can think of one simple way based on the distances we can split, but let's say their houses are not on the same road or need to make several turns to reach Vinod's place, which is not required if only Manish and Shashi as the two travelers. Or maybe the road between Shashi and the office is bad, and hence if we remove Shashi, the cost would be less for Manish and Vinod. The idea here is that, let's say the interactions are so complicated, we would require a lot of understanding of road, map, etc., to identify the exact split. With SHAP, the intuition is to let us identify all the possibilities for each of the friends. For example, For Manish, one possibility is going alone, one is to go only with Vinod, and so on. Then, the cost for Manish is the average of all those possibilities/interactions. In our case, the features are the friends.

It's okay if it's difficult to understand what SHAP means; an interested reader can explore more on this idea. We are interested in implementing SHAP, which is quite easy; we will use a third-party library that already implemented SHAP and use the library for our mode. We need first to install the shap package using pip (**`run ! pip install shap`** inside notebook). Then, run the following code for implementing SHAP for our model:

```
import shap

explainer = shap.TreeExplainer(clf)

X_test.iloc[10], y_test.iloc[10]

shap_values = explainer.shap_values(X_test.iloc[10])

shap.initjs()

shap.force_plot(explainer.expected_value[1], shap_values[1], X_test.
iloc[10])
```

Here we try to explain the example **`X_test.iloc[10]`** or the 10th instance for the **`test_set`**. This might be different for you as we are randomly splitting. In line 4, we circulate **`shap_values`** and then plot using **`force_plot`**. The following is the output of the code:

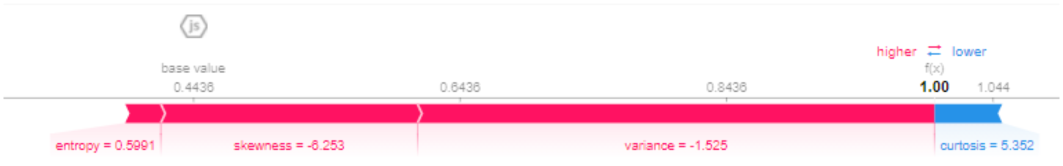


***Figure 6.15**: Shap output*

The way to interpret the result is as follows; there is a **`base_value`** for each prediction. This is just an average score. If none of the feature influences, then this would be the score. Then each feature pushes the model prediction in two different directions, red one towards 1 (positive) and blue towards zero(negative). Here we can see a variance of -1.525 has the maximum effect on pushing the prediction towards 1. Then skewness of -6.253. Curtosis = 5.352 tries to push it towards zero, but since all the others are pushing it towards 1, the result is class '1'. So, we can now answer why the model output is 1? Because of the value of skewness, variance, entropy, it is like that. I would encourage you to play around with multiple instances of data and try to interpret. Also, you can explore **`decision_plot`** and other plots too. We will go through all the other plots in *Chapter 7: Data Analytics* too.

For LIME, the interpretation and implementation are straightforward too. It assumes the data to be locally linear and fits a linear regression model. Then, identify the weights. Interested readers can go through the code by exploring **https://github.com/marcotcr/lime**. This would be one of the exercises for the reader. If you face issues, we will use it again in *Chapter 7: Data Analytics*.

For deep learning models also, we can use SHAP, and we have examples for SHAP in the GitHub repository (**https://github.com/slundberg/shap**). There are other techniques such as deconvolution for CNN-based models. We will go through those in detail in *Chapter 8: Building a Custom Image Classifier from Scratch*.

6.3 Managing model versions

Managing model versions is an important aspect of machine learning, and managing models also includes managing the features. Because few models might be trained with a particular set of features, and in the second version, we may find more features through feature engineering, and hence the input feature also changes from version to version. Therefore, keeping a log of all these things is important for experimentation. Finally, we can refer to this log and may be able to make decisions about what features to use or what model to choose by analyzing.

The way I started this logging is through Excel. It can be done with Excel. The only thing is that with a big project with enough experimentation, things start to become a mess. For example, let's say we only log the model and its performance in the beginning. Then we can also introduce the feature set, model hyperparameters, and so on. We can store the model name, model feature set, model hyperparameters, and the metric with Excel. But what if we want to compare models visually? We need to write some code, then read the Excel and do some visualization. All of this is possible if we enter every data point into Excel correctly, which sounds simple but with a huge problem and limited time, people seldom are able to do this properly. Here, we will explore two new packages to manage, visualize and even create fully interactive applications. We will start with MLflow and then learn about Streamlit.

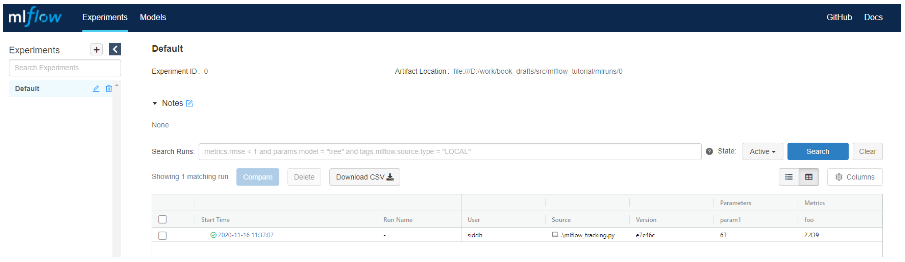
- **MLFlow**

  MLFlow is an open source software to manage the ML lifecycle. It is a great software that has tracking, reproducibility, deployment, and a registry. We are only going to talk about the tracking aspect here. To install MLflow, all you need to do is **`pip install mlflow`**. Now, let's talk about a basic tracking example. You can also find this example on the MLFlow documentation page. Here we will not use MLFlow to track our project as we will be doing it in upcoming chapters when we deal with business use cases. Here we are going to see the features and familiarize ourselves with the UI MLflow provides. So, let's start with a small example, and the following is the code:

```
import os
from random import random, randint
from mlflow import log_metric, log_param, log_artifacts

if __name__ == "__main__":
    # Log a parameter (key-value pair)
    log_param("param1", randint(0, 100))

    # Log a metric; metrics can be updated throughout the run
    log_metric("foo", random())
    log_metric("foo", random() + 1)
    log_metric("foo", random() + 2)

    # Log an artifact (output file)
    if not os.path.exists("outputs"):
        os.makedirs("outputs")
    with open("outputs/test.txt", "w") as f:
        f.write("hello world!")
    log_artifacts("outputs")
```

In the preceding example, you can store it in a file with any name; I named it **`model.py`** because this will be our model training file in the future. Because we are going to track the model building process. So, three things we need to understand here with tracking, parameters, metrics, and artifacts. You can store anything you want here, as you can see in the preceding code. Still, in general, we store model hyperparameters as parameters, any model metric (that we are checking if it's increasing throughout the run, for example, training and testing accuracy with epoch) we store as metric, and any visualizations for a presentation are stored in artifacts.

Now, let's run this file with python **`model.py`** in command prompt, creating the folders and storing the parameters, metrics, and artifacts. Now we can enter **`mlflow ui`** to start the UI server. Now you can type localhost:5000 in the browser to start the MLFlow UI. Here is how it looks:



Default

Experiment ID: 0 Artifact Location: file:///D:/work/book_drafts/src/mlflow_tutorial/mlruns/0

Notes

None

Search Runs: metrics.rmse < 1 and params.model = "tree" and tags.mlflow.source.type = "LOCAL" State: Active Search Clear

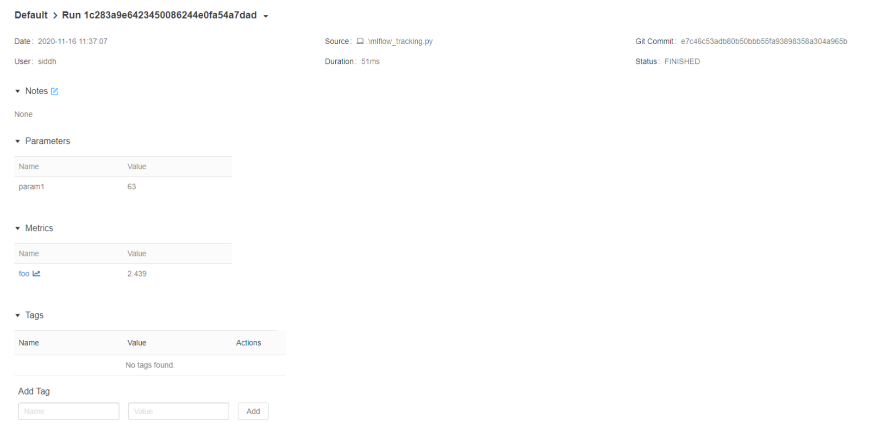
Showing 1 matching run Compare Delete Download CSV Columns

						Parameters	Metrics
	Start Time	Run Name	User	Source	Version	param1	foo
	2020-11-16 11:37:07	-	siddh	.\mlflow_tracking.py	e7c46c	63	2.439

Figure 6.16: *MLFlow UI*

You can see the experiment run; there is a search runs option, where we can search any parameter and metric, which will help us search for models based on their performance/parameters. It has a start time, all parameter details, and so on automatically. We need not create anything.

Let's click on the experiment we created; the following is the screen that comes up:



Default > Run 1c283a9e6423450086244e0fa54a7dad

Date: 2020-11-16 11:37:07 Source: .\mlflow_tracking.py Git Commit: e7c46c53adb80b50bbb55fa93898358a304a965b

User: siddh Duration: 51ms Status: FINISHED

Notes

None

Parameters

Name	Value
param1	63

Metrics

Name	Value
foo	2.439

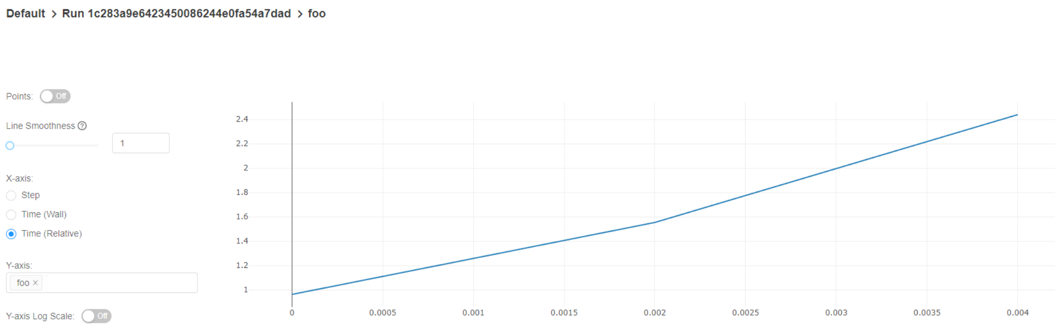
Tags

Name	Value	Actions
	No tags found.	

Add Tag

Figure 6.17: *MLFlow UI experiment*

We have the run ID, an option to add notes and add tags to the experiment. Also, look at the metrics parameter. We randomly added few points in an increasing order, and it created a metric graph. Let us explore that by clicking on the foo metric:


Default > Run 1c283a9e6423450086244e0fa54a7dad > foo

Points: Off

Line Smoothness: 1

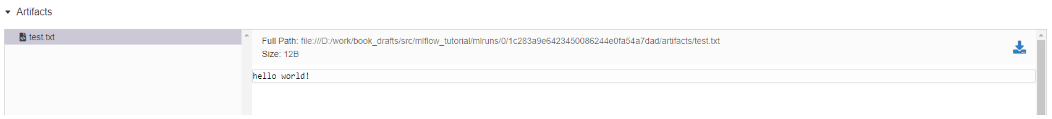
X-axis: Step, Time (Wall), [✓ Time (Relative)]

Y-axis: foo

Y-axis Log Scale: Off

Figure 6.18: *MLFlow UI metrics*

Here we can see the progression of the machine learning metric; we have many options to modify the graph. Next, we can see the model artifacts on the same experiment page as follows:


Artifacts

test.txt

Full Path: file:///D:/work/book_drafts/src/mlflow_tutorial/mlruns/0/1c283a9e6423450086244e0fa54a7dad/artifacts/test.txt
Size: 12B

```
hello world!
```

Figure 6.19*: MLFlow UI artifacts*

Here we can see our text file; in a real scenario, it will contain our visualization graphs to see and download whenever we want.

The Machine Learning lifecycle does not stop after putting the model into production. Data from the real world may change over time, and our model requires retraining. So, what we do is we collect data even after putting the model into production and then retrain the model after a particular amount of data is collected. Some models support incremental training; that is, we only train with new data, but some do not. So, we need to add the data to the original data and retrain the model. Also, in extreme cases, we can incrementally train the model with a single row of data, often called online training. All these model retraining is dependent upon the problem you are trying to solve, and there is no single way to do it. MLFlow also helps nicely with model performance analysis over time.

We are just offline validation of models, i.e., we are taking historical datasets, splitting them into train, validation, and test. Use train validation to train model and fine tune hyperparameters. Finally, we use the test set to validate our model. But even that test set is historical and not live. In real-world systems, on the other hand, data changes every day. We can still keep our models with new data, but we need to make sure our model catches up with this data ***drift***. One simple way is to train our model with the new data (either in batch or incrementally or online (with every new data point)), then check the validation metric and compare it with the old model. This is what we call online validation, validation with live data. But we can compare the performance of the models with MLFlow and decide if we want to change the model.

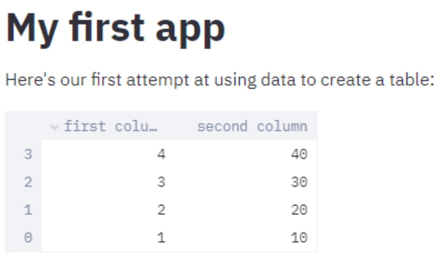
This is just the tip of the iceberg of what MLFlow is capable of. We can also name experiments and then search for them. We can log models with signatures, and it will solve the metadata and environment we discussed earlier in the chapter. All these features we will explore much in more detail in the upcoming chapters.

- **Streamlit**

  Streamlit is another software that is used to create data apps. You can create fully operational, fast web apps in seconds. Its usage is also very intuitive. Although you can create complete data apps using this software, we will mostly use it for model inference, interactive data visualizations, metric visualizations, and so on. And this software has great documentation for any user. And the following example is taken directly from the documentation. Here is the example to see basic usage of Streamlit:

```
! pip install streamlit
import streamlit as st
import time
import numpy as np
import pandas as pd


st.title('My first app')


st.write("Here's our first attempt at using data to create a
table:")
st.write(pd.DataFrame({
    'first column': [1, 2, 3, 4],
    'second column': [10, 20, 30, 40]
}))
```

The preceding code is just used to create an interactive pandas data frame; we can run it by **`streamlit run file_name.py`** in prompt; it will create a web app at **`http://localhost:8501`**. The following is the output:



My first app

Here's our first attempt at using data to create a table:

	first colu…	second column
3	4	40
2	3	30
1	2	20
0	1	10

***Figure 6.20**: App using streamlit: Dataframes*

With this, we now have a web app, and we see the data frame we created; we can sort the values for each column. It is a quick way to interact with any data frame for analysis.

We can also create interactive line charts simply using **`st.line_chart`**; the following is the code:

```
chart_data = pd.DataFrame(
     np.random.randn(20, 3),
     columns=['a', 'b', 'c'])

st.line_chart(chart_data)
```

This will create an interactive line chart:

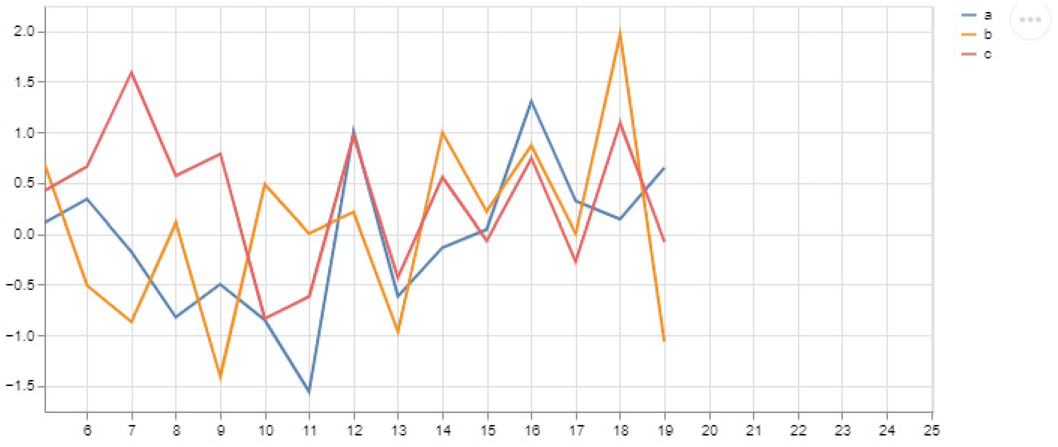


***Figure 6.21**: App using streamlit: Charts*

Other options include creating progress bars and maps, and so on, which we are not going into details about here. But an interested reader can explore streamlit documentation. It is a cool piece of software and can help create interesting data apps in very little time.

Conclusion

With this, we conclude the chapter on model deployment in production. Much of this chapter is dedicated to creating a model serving application and producing it for public use that we first did with FastAPI and Heroku. We learned about best practices of saving models and creating faster APIs. Then, we learned about one of the most overlooked aspects of machine learning which is model interpretation. We learned about in-built options with tree-based models and model agnostic ones with SHAP and LIME. Then we learned about the importance of model testing even after putting it into production. And finally, we learned two useful software, i.e., MLFlow, Streamlit for model version management, and interactive visualizations.

In the next chapter, we will discuss section two of this book and start building industry use cases for a data analytics problem. It would be code-heavy and interesting, so stay tuned.

Points to remember

The following are a few points to remember:

- The best way to deploy our model is to serve it through an API; this separates the software and model concerns.
- Apart from prediction, we also need to provide a confidence score to the user for better clarity.
- Model interpretation is often required to understand the predictions of the model.
- Model improvement doesn't stop after putting the model in production. We need to retrain the model with data changes.
- Model versioning is important, and keeping track of versions and features on which model is training valuable in creating top-notch models.
- The machine learning ecosystem is rich with different software and different options. Always keep an eye out for better technologies.

MCQ

1. **Which one below is a backend framework?**
   a) Django
   b) Flask
   c) FastAPI
   d) All of the above

2. **Which parameter contains the feature importances for a random forest?**
   a) best_params
   b) feature_importance
   c) feature_importances
   d) feature_importances_

3. **Which is the model agnostic model interpretation?**
   a) LIME
   b) Feature_importance_
   c) SHAP
   d) Both a and c
   e) Both b and c

4. **We usually save model hyperparameters in MLFlow as,**
   a) artifacts
   b) params
   c) tags
   d) model

Answers to MCQ

1. d
2. d
3. c
4. b

Questions

1. Implement LIME for the example given in this chapter (banknote authentication)
2. Include the probability score as output to the server we created with Heroku.
3. Use MLFlow to train the model in Bank Note Authentication, save params, artifacts and log the metric.
4. Create an application with Streamlit taking our banknote authentication data, take the input from the user and provide output, also include the shap interpretation.

Key Terms

- Application Programming Interface (API)
- Model Agnostic Interpretation
- Feature Importance
- Model Versioning
- Model Deployment
- Incremental retraining

Chapter 7 Data Analytics Use Case

Now we have approached section two of this book, which has more practical examples and use cases. In this chapter, we are going to explore a housing price prediction use case. Although such examples are common, and the reader may not feel this example as a practical use case, we will understand many implementation details. And *chapter 7* is not about one example; there will be multiple examples. Like real-life data, we will explore dirty data and how to preprocess it. We will also explore huge datasets (>20GB) and small datasets too (<1000 examples). Such scenarios are common in real life; you either have datasets split across multiple tables in multiple databases with hundreds of millions of rows or for a POC work, you may find very little data. In the worst case, you may be required to create data. So, we will explore these cases too. Then we will discuss what to do after a model is built.

We are going to dive deep into the question that is far more important than any machine learning concept, "*How is a user going to benefit from this?*" But first, we will start with a few in-depth understanding of the models like the random forest, XGBOOST, neural networks, and so on. Machine learning has a plethora of models and understanding each one of them is not required. Here we will appreciate a few key concepts and models, and we will discuss how to understand and implement a different model. So, let us start with building our first use case in machine learning.

Structure

In this chapter, we will cover the following topics:

- Use case and practical tips
- Understanding the random forest model
- Other Models – XGBoost and Neural Networks
- The idea for a web application

Objective

After studying this chapter, you should be able to:

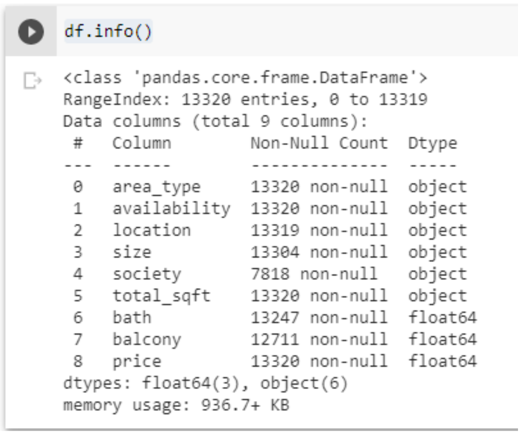
- Implement a machine learning model from scratch and go through the machine learning lifecycle
- A deeper understanding of some of the commonly used models
- Understanding machine learning at scale
- Few practical tips for real-life datasets

7.1 Use case and practical tips

Let us start with an example; here, we will take the dataset for Bangalore housing prices. We will use the *curlWget* chrome plugin, as usual, to download the data into Google Colab. Let us load the data into a Pandas DataFrame.

```
import pandas as pd

import numpy as np


df_raw = pd.read_csv("Bengaluru_House_Data.csv")

df = df_raw.copy()


# first look into data

df.info()
```

The output of the preceding code:

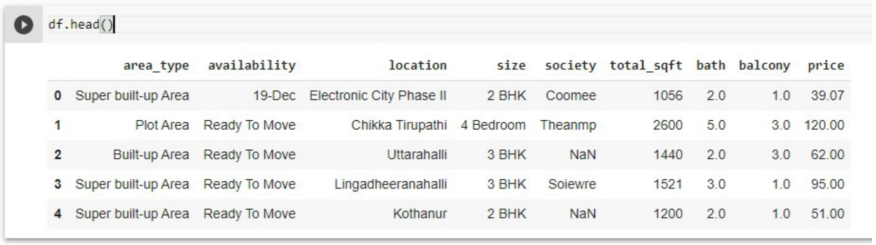


```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13320 entries, 0 to 13319
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   area_type     13320 non-null  object
 1   availability  13320 non-null  object
 2   location      13319 non-null  object
 3   size          13304 non-null  object
 4   society       7818 non-null   object
 5   total_sqft    13320 non-null  object
 6   bath          13247 non-null  float64
 7   balcony       12711 non-null  float64
 8   price         13320 non-null  float64
dtypes: float64(3), object(6)
memory usage: 936.7+ KB
```

***Figure 7.1**: Bangalore housing dataset column information*

As we can see, there are nine columns and 13320 rows; we have six categorical values '**`area_type`**,' **`'availability`**,' **`'location`**,' **`'size`**,' **`'society`**,' **`'total_sqft`**,' and three numerical columns **`'bath`**,' **`'balcony`**,' and **`'price`**.' At first glance, it seems there is some issue with this dataset, as **`total_sqft`** and size seem to be numerical columns. But let's see what kind of data it contains by checking its head.

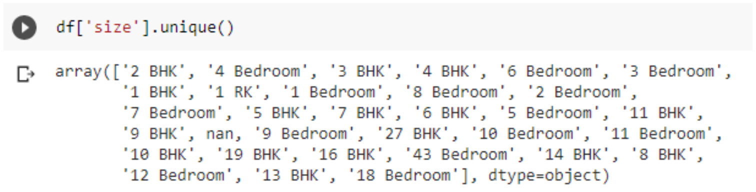


```
df.head()
```

	area_type	availability	location	size	society	total_sqft	bath	balcony	price
0	Super built-up Area	19-Dec	Electronic City Phase II	2 BHK	Coomee	1056	2.0	1.0	39.07
1	Plot Area	Ready To Move	Chikka Tirupathi	4 Bedroom	Theanmp	2600	5.0	3.0	120.00
2	Built-up Area	Ready To Move	Uttarahalli	3 BHK	NaN	1440	2.0	3.0	62.00
3	Super built-up Area	Ready To Move	Lingadheeranahalli	3 BHK	Soiewre	1521	3.0	1.0	95.00
4	Super built-up Area	Ready To Move	Kothanur	2 BHK	NaN	1200	2.0	1.0	51.00

***Figure 7.2**: Bangalore housing dataset head*

Here we can see the size is an object datatype because it contains suffixes like BHK, Bedroom. But **`total_sqft`** still seems to be numbered. Let us dig a bit deeper into these two columns. We can check the distinct columns using the **`.unique()`** method.

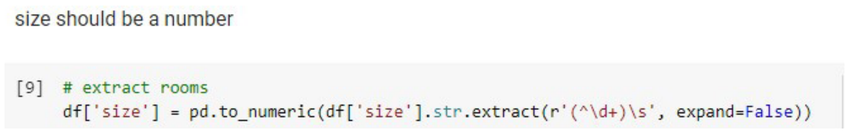


```
df['size'].unique()
```

```
array(['2 BHK', '4 Bedroom', '3 BHK', '4 BHK', '6 Bedroom', '3 Bedroom',
       '1 BHK', '1 RK', '1 Bedroom', '8 Bedroom', '2 Bedroom',
       '7 Bedroom', '5 BHK', '7 BHK', '6 BHK', '5 Bedroom', '11 BHK',
       '9 BHK', nan, '9 Bedroom', '27 BHK', '10 Bedroom', '11 Bedroom',
       '10 BHK', '19 BHK', '16 BHK', '43 Bedroom', '14 BHK', '8 BHK',
       '12 Bedroom', '13 BHK', '18 Bedroom'], dtype=object)
```

***Figure 7.3**: Unique 'size' values*

We can see the first number is the actual size numerical value followed by an inconsistent suffix. Here we can extract the value using the **`.str.extract`** method:



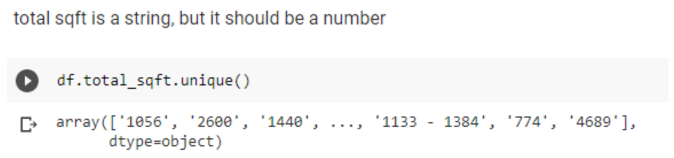
size should be a number

```
[9] # extract rooms
    df['size'] = pd.to_numeric(df['size'].str.extract(r'(^\d+)\s', expand=False))
```

Figure 7.4: *Extracting rooms*

Here we first extract the required string using regex r'(^\d+)\s,' giving any digit that the string starts with. Then, we change it to numerical value using **`pd.to_numeric`**.

Now let's come to '**`total_sqft`**.' This should also be a number. Let us check its distinct values.

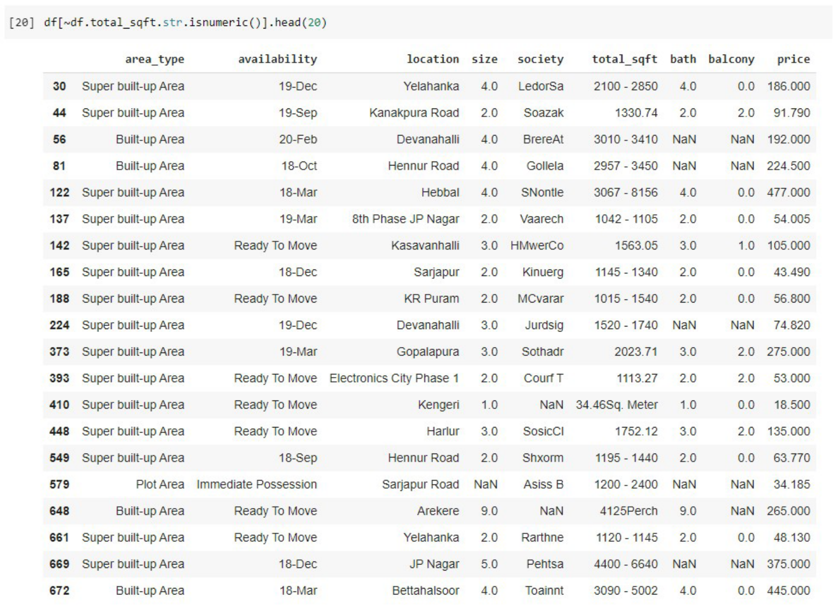


total sqft is a string, but it should be a number

```
df.total_sqft.unique()

array(['1056', '2600', '1440', ..., '1133 - 1384', '774', '4689'],
      dtype=object)
```

Figure 7.5*: total_sqft distinct values*

We can see some values containing '-'; let us explore more by filtering only non-digit numbers.



```
[20] df[~df.total_sqft.str.isnumeric()].head(20)
```

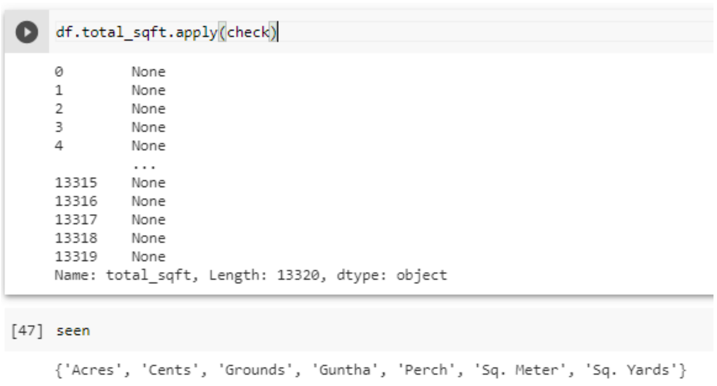
	area_type	availability	location	size	society	total_sqft	bath	balcony	price
30	Super built-up Area	19-Dec	Yelahanka	4.0	LedorSa	2100 - 2850	4.0	0.0	186.000
44	Super built-up Area	19-Sep	Kanakpura Road	2.0	Soazak	1330.74	2.0	2.0	91.790
56	Built-up Area	20-Feb	Devanahalli	4.0	BrereAt	3010 - 3410	NaN	NaN	192.000
81	Built-up Area	18-Oct	Hennur Road	4.0	Gollela	2957 - 3450	NaN	NaN	224.500
122	Super built-up Area	18-Mar	Hebbal	4.0	SNontle	3067 - 8156	4.0	0.0	477.000
137	Super built-up Area	19-Mar	8th Phase JP Nagar	2.0	Vaarech	1042 - 1105	2.0	0.0	54.005
142	Super built-up Area	Ready To Move	Kasavanhalli	3.0	HMwerCo	1563.05	3.0	1.0	105.000
165	Super built-up Area	18-Dec	Sarjapur	2.0	Kinuerg	1145 - 1340	2.0	0.0	43.490
188	Super built-up Area	Ready To Move	KR Puram	2.0	MCvarar	1015 - 1540	2.0	0.0	56.800
224	Super built-up Area	19-Dec	Devanahalli	3.0	Jurdsig	1520 - 1740	NaN	NaN	74.820
373	Super built-up Area	19-Mar	Gopalapura	3.0	Sothadr	2023.71	3.0	2.0	275.000
393	Super built-up Area	Ready To Move	Electronics City Phase 1	2.0	Courf T	1113.27	2.0	2.0	53.000
410	Super built-up Area	Ready To Move	Kengeri	1.0	NaN	34.46Sq. Meter	1.0	0.0	18.500
448	Super built-up Area	Ready To Move	Harlur	3.0	SosicCl	1752.12	3.0	2.0	135.000
549	Super built-up Area	18-Sep	Hennur Road	2.0	Shxorm	1195 - 1440	2.0	0.0	63.770
579	Plot Area	Immediate Possession	Sarjapur Road	NaN	Asiss B	1200 - 2400	NaN	NaN	34.185
648	Built-up Area	Ready To Move	Arekere	9.0	NaN	4125Perch	9.0	NaN	265.000
661	Super built-up Area	Ready To Move	Yelahanka	2.0	Rarthne	1120 - 1145	2.0	0.0	48.130
669	Super built-up Area	18-Dec	JP Nagar	5.0	Pehtsa	4400 - 6640	NaN	NaN	375.000
672	Built-up Area	18-Mar	Bettahalsoor	4.0	Toainnt	3090 - 5002	4.0	0.0	445.000

Figure 7.6*: Non digit values*

We can see there are other metrics present that is, Sq. Meter, Perch , and so on. Let us write a function to check all these values:

```
import re
seen = set()
def check(s):
  if s.find('-') != -1: return
  if not s: return
  try:
    float(s.strip())
  except Exception as err:
    s = re.sub(r'^\d+', '', s)
    s = re.sub(r'^.\d+', '', s)
    seen.add(s)
```

Here we check if the string has '-'or a None value and just returning nothing. If we try to change the string to float and any error occurs, it contains some text. And then, we are just removing the numbers from the string to get all the different metrics. Finally, we add all these metrics to a set to avoid repetition. Now let us apply it to the dataframe.



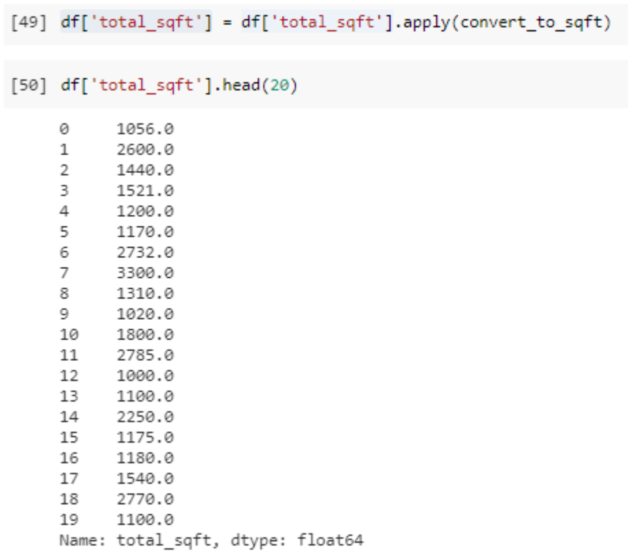
```
df.total_sqft.apply(check)

0        None
1        None
2        None
3        None
4        None
         ...
13315    None
13316    None
13317    None
13318    None
13319    None
Name: total_sqft, Length: 13320, dtype: object

[47] seen

{'Acres', 'Cents', 'Grounds', 'Guntha', 'Perch', 'Sq. Meter', 'Sq. Yards'}
```

***Figure 7.7**: Different metrics for total_sqft*

As we apply, the column and function return nothing (only tracking seen); we get the preceding output when we apply check. But now seen contains all those metrics. Upon quick googling, we get the conversion rates. Let us keep track of these conversion rates in a dictionary and apply another function to clean the **`total_sqft`** column.

```
import re
CONVERSION_DICT = {
    'Sq. Meter': 10.7639,
    'Sq. Yards': 9,
    'Perch': 272.25,
    'Acres': 4046.86,
    'Cent': 435.6,
    'Guntha': 1089,
    'Ground': 2400
}

def convert_to_sqft(s):
    if '-' in s:
        t = s.split(' - ')
        return float(t[0])+float(t[1])//2 if len(t) == 2 else float(t[0])
    for area_type, conversion_multiplier in CONVERSION_DICT.items():
      if s.find(area_type) != -1:
        t = s.split(area_type)
        return float(t[0]) * conversion_multiplier if len(t) == 2 else
float(t[0])
    return float(s)
```

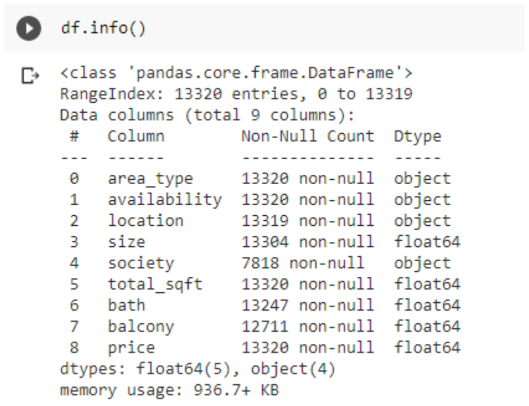
Here we simply split the string using '-' or any of the metric types, and if it is '-, 'we know it is a range, so we take mean. If it contains a specific metric type specified in our conversion dictionary, we simply multiply with **`conversion_multiplier`** (tracked in the dictionary) in line 19. Now let us apply it to our column.



```
[49] df['total_sqft'] = df['total_sqft'].apply(convert_to_sqft)

[50] df['total_sqft'].head(20)

0     1056.0
1     2600.0
2     1440.0
3     1521.0
4     1200.0
5     1170.0
6     2732.0
7     3300.0
8     1310.0
9     1020.0
10    1800.0
11    2785.0
12    1000.0
13    1100.0
14    2250.0
15    1175.0
16    1180.0
17    1540.0
18    2770.0
19    1100.0
Name: total_sqft, dtype: float64
```

Figure 7.8: *Cleaned total_sqft column*

We can see **`total_sqft`** is converted to sqft value for all rows; let us check all the data types once again.



```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13320 entries, 0 to 13319
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   area_type     13320 non-null  object
 1   availability  13320 non-null  object
 2   location      13319 non-null  object
 3   size          13304 non-null  float64
 4   society       7818 non-null   object
 5   total_sqft    13320 non-null  float64
 6   bath          13247 non-null  float64
 7   balcony       12711 non-null  float64
 8   price         13320 non-null  float64
dtypes: float64(5), object(4)
memory usage: 936.7+ KB
```

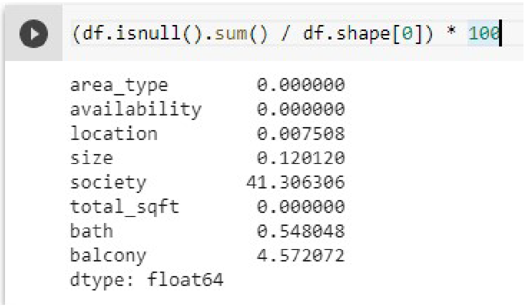
***Figure 7.9**: Datatypes after conversion*

Now, as desired, size and **`total_sqft`** are converted into floats. We can reduce the size and save it in a feather formatted file for quicker access in the future, but we can discuss this in the upcoming practical tips section as this dataset is very small and takes seconds to process. But as per best practice, this is the time to set aside some time and save the intermediate results because cleaning is often computation heavy and takes a lot of time, so it is advised to save the dataset. Also, I advise the reader to use MLFlow to log this file (after saving) to track it. We won't be using MLFlow in this use case as this example is pretty small, and we need not require MLFlow.

Now let us work on pre-processing; first we need to drop the target column as we don't want any pre-processing to be done to the target. The following is the code:

```
y_field = 'price'
y = df[y_field].values
df.drop([y_field], axis=1, inplace=True)
```

First, let us do some missing value treatment; let us check how much missing value is present in each column.



```
(df.isnull().sum() / df.shape[0]) * 100
```

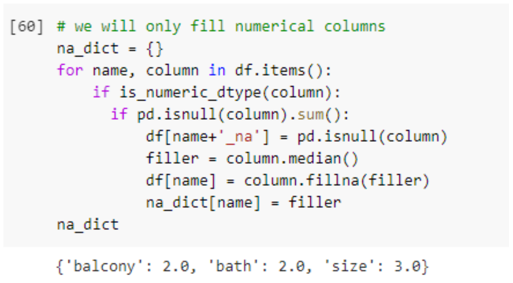
```
area_type        0.000000
availability     0.000000
location         0.007508
size             0.120120
society         41.306306
total_sqft       0.000000
bath             0.548048
balcony          4.572072
dtype: float64
```

***Figure 7.10**: Missing values*

Here we can see society has a lot of missing values; we may consider dropping it but for now, let us not do that as our dataset is quite small. The following is the code for handling missing values:

```
from pandas.api.types import is_numeric_dtype
# we will only fill numerical columns
na_dict = {}
for name, column in df.items():
    if is_numeric_dtype(column):
      if pd.isnull(column).sum():
          df[name+'_na'] = pd.isnull(column)
          filler = column.median()
          df[name] = column.fillna(filler)
          na_dict[name] = filler
na_dict
```

Here we fill only numeric data with median, creating another column to track the missing values. Remember missing values can also be an indicator of something interesting to track—the output of the preceding code.



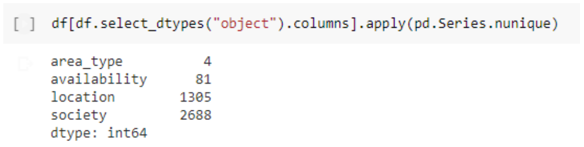
```
[60] # we will only fill numerical columns
     na_dict = {}
     for name, column in df.items():
         if is_numeric_dtype(column):
           if pd.isnull(column).sum():
               df[name+'_na'] = pd.isnull(column)
               filler = column.median()
               df[name] = column.fillna(filler)
               na_dict[name] = filler
     na_dict

     {'balcony': 2.0, 'bath': 2.0, 'size': 3.0}
```

***Figure 7.11**: Missing value treatment*

We track all the imputed values during testing; we will use these values to fill the missing values (as training and test set data may contain different statistics).

Now let us change the categorical columns to numerical; we will convert high cardinality columns (columns having a lot of unique values) into simple numbers, for example, if a column contains 'a', 'b,' 'c,' we will change it to 1, 2, 3. For columns with low cardinality, we are going to create dummy variables.

Checking cardinality code:



```
[ ] df[df.select_dtypes("object").columns].apply(pd.Series.nunique)

    area_type          4
    availability      81
    location        1305
    society         2688
    dtype: int64
```

***Figure 7.12**: Cardinality check*

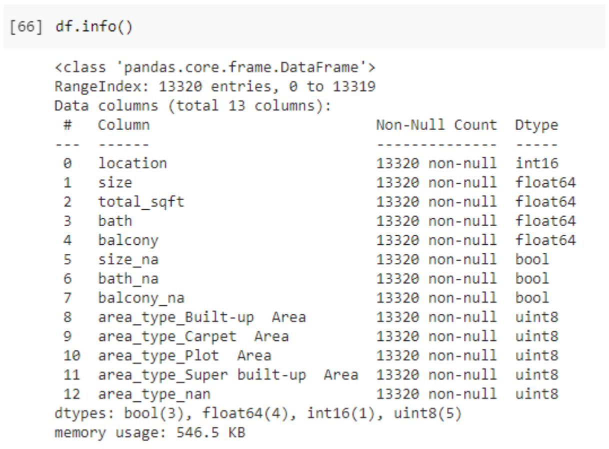
Here we can see society and location have many unique values. Still, as shown in missing value treatment, society has many missing values (around 40%), so we will drop it anyway.

Let us drop society and availability (as this column does not seem to affect the price intuitively, the reader may choose to keep this and remove it later in another experiment if it doesn't add value. That would be the right way to do it).

```
# society has a lot of missing values, so we will drop it
# availability is a column that should not matter for our analysis too
skip_fields = ['society,' 'availability']

df.drop(skip_fields, axis=1, inplace=True)
```

```
Now let us set a threshold for high cardinality values and convert.
MAX_CAT = 20 # THRESHOLD FOR CARDINALITY
for name, column in df.items():
  if not is_numeric_dtype(column) and (column.nunique()>MAX_CAT):
      cat_col = pd.Categorical(column)
      df[name] = cat_col.codes+1

# getting dummy variables for the rest
df = pd.get_dummies(df, dummy_na=True)
```

Now let us check the dataset types.



```
[66] df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13320 entries, 0 to 13319
Data columns (total 13 columns):
 #   Column                        Non-Null Count  Dtype
---  ------                        --------------  -----
 0   location                      13320 non-null  int16
 1   size                          13320 non-null  float64
 2   total_sqft                    13320 non-null  float64
 3   bath                          13320 non-null  float64
 4   balcony                       13320 non-null  float64
 5   size_na                       13320 non-null  bool
 6   bath_na                       13320 non-null  bool
 7   balcony_na                    13320 non-null  bool
 8   area_type_Built-up  Area      13320 non-null  uint8
 9   area_type_Carpet  Area        13320 non-null  uint8
 10  area_type_Plot  Area          13320 non-null  uint8
 11  area_type_Super built-up  Area  13320 non-null  uint8
 12  area_type_nan                 13320 non-null  uint8
dtypes: bool(3), float64(4), int16(1), uint8(5)
memory usage: 546.5 KB
```

Figure 7.13*: After treatment*

After treatment, we have no null values in any columns, and everything is changed to a number. Now let us start building our model. (Here is another chance to save this data into a preprocessed folder and track the median values in MLFlow and keep track of column names in MLFlow).

Let us create a baseline model:

```
X = df.to_numpy()

# Training and Testing Sets
from sklearn.model_selection import train_test_split

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

from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(n_jobs=-1, n_estimators=5)
model.fit(X_train, y_train)
```

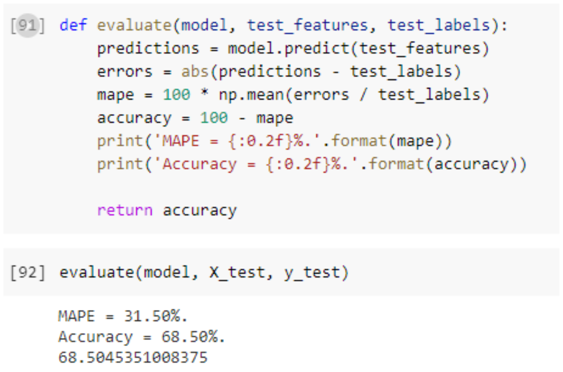
We first split the dataset into training and testing with 75% to train and 25% test. Then we use five trees in Random Forest to create a model. Scikit-learn has a user-friendly API to create and train models. So not much explanation is needed; all the code is self-explanatory.

Now comes the metric; since this is not a competition, we don't have a specific metric to look for; I checked few example notebooks in Kaggle that worked on this dataset and seen people mostly using RMSE or MAPE to evaluate. When you go to your domain expert or client to understand what a good metric could track. As discussed in *Chapter 1: Introduction to Machine Learning*, machine learning requires a lot of back-and-forth communication.

Let us take **MAPE** as a metric here, it stands for **Mean Absolute Percentage Error**, which is simple, we just take the absolute error (not squared as in RMSE) and divide it to the samples and take a mean:

```
 def evaluate(model, test_features, test_labels):
    predictions = model.predict(test_features)
    errors = abs(predictions - test_labels)
    mape = 100 * np.mean(errors / test_labels)
    accuracy = 100 - mape
    print('MAPE = {:0.2f}%.'.format(mape))
    print('Accuracy = {:0.2f}%.'.format(accuracy))
        return accuracy
```

Here we print accuracy as 100-mape too. Let us evaluate our baseline model.



```
[91] def evaluate(model, test_features, test_labels):
         predictions = model.predict(test_features)
         errors = abs(predictions - test_labels)
         mape = 100 * np.mean(errors / test_labels)
         accuracy = 100 - mape
         print('MAPE = {:0.2f}%.'.format(mape))
         print('Accuracy = {:0.2f}%.'.format(accuracy))

         return accuracy

[92] evaluate(model, X_test, y_test)

     MAPE = 31.50%.
     Accuracy = 68.50%.
     68.5045351008375
```

***Figure 7.14**: Evaluate baseline model*

We get a mape of 31.05% and an accuracy of 68.05%. Not bad for a baseline model.

Now let us start fine tuning the results; note the data is very less and simple, so we may not find a large improvement even after fine-tuning here, but we will learn the techniques to fine tune properly.

We will use the coarse to fine approach as discussed in *Chapter 5: Model Building and Tuning,* First. We will use random search which is faster to identify some hyperparamters.

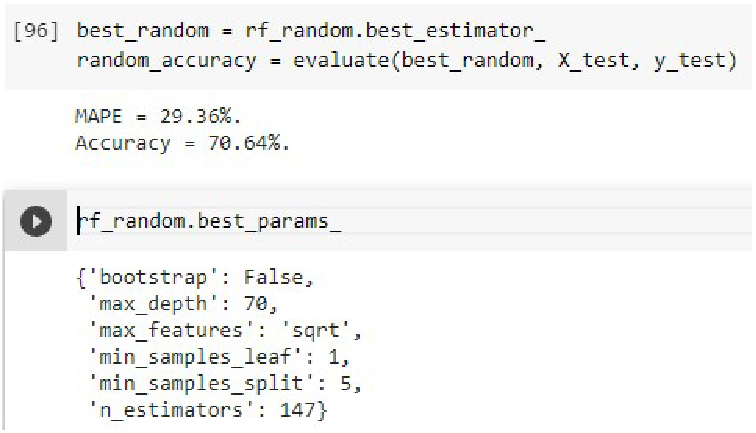
```
from sklearn.model_selection import RandomizedSearchCV

n_estimators = [int(x) for x in np.linspace(start = 100, stop = 400, num
= 20)]

max_features = ['auto', 'sqrt']

max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]

max_depth.append(None)

min_samples_split = [2, 5, 10]

min_samples_leaf = [1, 2, 4]

bootstrap = [True, False]

# Create the random grid

random_grid = {'n_estimators': n_estimators,

               'max_features': max_features,

               'max_depth': max_depth,

               'min_samples_split': min_samples_split,

               'min_samples_leaf': min_samples_leaf,

               'bootstrap': bootstrap}
```

We created a random grid containing a range for each hyperparameter; understanding each hyperparameter is necessary. We will discuss this in the upcoming model section of this book.

Let us now run the random search:

```
rf = RandomForestRegressor()

rf_random = RandomizedSearchCV(estimator = rf, param_distributions =
random_grid, n_iter = 100, cv = 3, verbose=2, random_state=42, n_jobs =
-1)

rf_random.fit(X_train, y_train)
```

We are also using cross-validation of 3, three-fold cross-validation (as discussed in *Chapter 5: Model Building and Tuning*). This may take some time to run as there are 300 different combinations to check. After it runs, now let us check the best parameters and score.



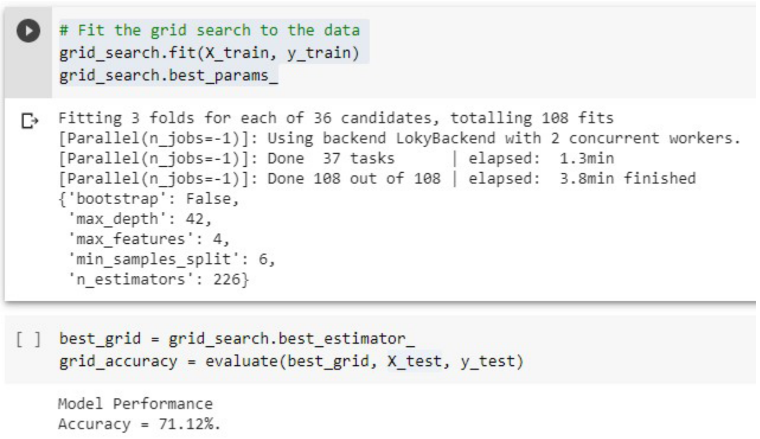
```
[96] best_random = rf_random.best_estimator_
     random_accuracy = evaluate(best_random, X_test, y_test)

     MAPE = 29.36%.
     Accuracy = 70.64%.

     rf_random.best_params_

     {'bootstrap': False,
      'max_depth': 70,
      'max_features': 'sqrt',
      'min_samples_leaf': 1,
      'min_samples_split': 5,
      'n_estimators': 147}
```

***Figure 7.15**: Random search results*

As expected, this is not going to give a drastic improvement, but we still got a 2.14 reduction in MAPE which is good. Now, let us go finer using grid search we will use these results from random search.

```
from sklearn.model_selection import GridSearchCV

# Create the parameter grid based on the results of random search

param_grid = {

    'bootstrap': [False],

    'max_depth': [70, 71, 72],

    'max_features': [3, 4],

    'min_samples_split': [5, 6],

    'n_estimators': [147, 150, 155]
}
# Create a based model
rf = RandomForestRegressor()
# Instantiate the grid search model
grid_search = GridSearchCV(estimator = rf, param_grid = param_grid,
                          cv = 3, n_jobs = -1, verbose = 2)
```

Here we gave the ranges close to the output from the random search (the coarse to fine approach). Let us see the results:



```
# Fit the grid search to the data
grid_search.fit(X_train, y_train)
grid_search.best_params_
```

```
Fitting 3 folds for each of 36 candidates, totalling 108 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done  37 tasks      | elapsed:  1.3min
[Parallel(n_jobs=-1)]: Done 108 out of 108 | elapsed:  3.8min finished
{'bootstrap': False,
 'max_depth': 42,
 'max_features': 4,
 'min_samples_split': 6,
 'n_estimators': 226}
```

```
best_grid = grid_search.best_estimator_
grid_accuracy = evaluate(best_grid, X_test, y_test)
```

```
Model Performance
Accuracy = 71.12%.
```

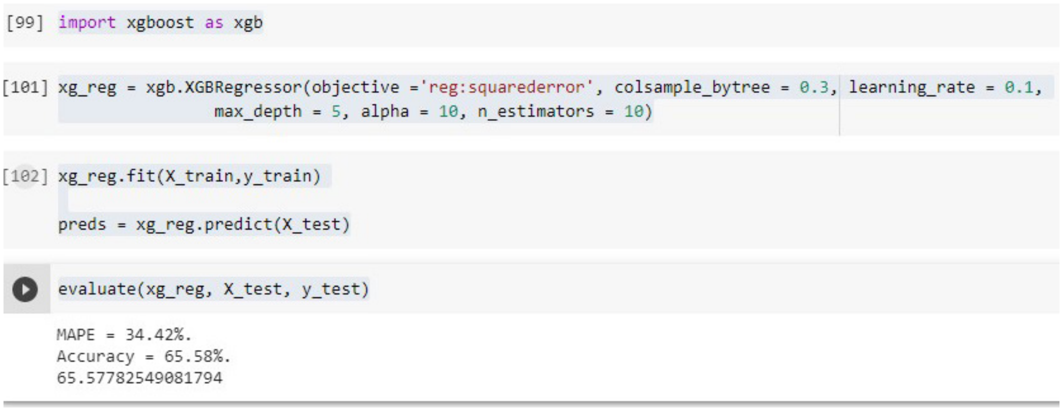
***Figure 7.16**: Grid search results*

We now got another improvement over 0.48% MAPE. Now the reader may repeat this grid search to finer ranges and see if things improve. And we will get a diminished return, but since we are not searching over a large space, it is fine. Now with such less data, we may not even require random search or this coarse to fine approach in the first place. But this shines when our models are complex, and the dataset is really large.

We can use a more complex model such as XGBOOST:

```
import xgboost as xgb

xg_reg = xgb.XGBRegressor(objective ='reg:squarederror', colsample_
bytree = 0.3, learning_rate = 0.1,
                max_depth = 5, alpha = 10, n_estimators = 10)

xg_reg.fit(X_train,y_train)
preds = xg_reg.predict(X_test)
evaluate(xg_reg, X_test, y_test)
```

The output of this code:



```
[99] import xgboost as xgb

[101] xg_reg = xgb.XGBRegressor(objective ='reg:squarederror', colsample_bytree = 0.3, learning_rate = 0.1,
                 max_depth = 5, alpha = 10, n_estimators = 10)

[102] xg_reg.fit(X_train,y_train)

      preds = xg_reg.predict(X_test)

      evaluate(xg_reg, X_test, y_test)

      MAPE = 34.42%.
      Accuracy = 65.58%.
      65.57782549081794
```

Figure 7.17: *The XGBOOST model*

34.42% without fine tuning is a good model to start with. We are not going to fine tune XGBOOST here. As an exercise, you can fine tune XGBOOST.

After this, we can create a FASTAPI application and deploy it in Heroku, as discussed in the last chapter, taking our model to production. But this data is quite small and does not cover many variations that a real-world problem has. So, we will discuss it further and provide practical tips.

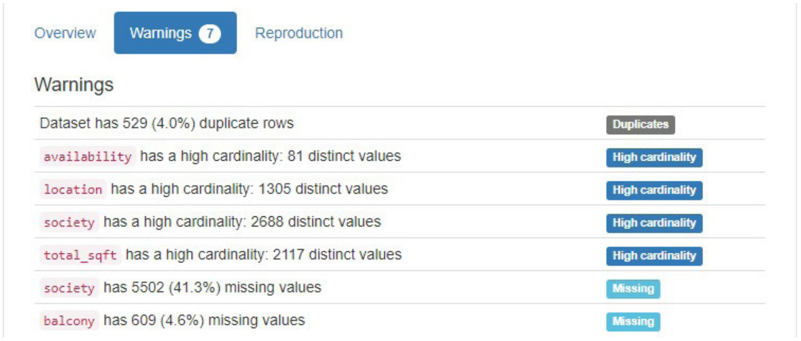
7.1.1 Practical tips: handling a lot of columns

If your dataset has many columns, we should try and reduce the number of columns. What I look for is highly correlated features, zero variance features (feature having only one unique value), high missing values (more than 70%), unimportant variables (variables that have zero importance to the prediction use the feature-selector package), and using all this, we can try to remove some of the features.

The first step that I do when I receive a lot of columns is to run a pandas-profile report using the pandas-profiling library. This gives you all those statistics mentioned; here is a pandas profile report using our dataset (**https://www.kaggle.com/amitabhajoy/bengaluru-house-price-data**) (I am just using it as an example, our data does not have a lot of columns).

```
# ! pip install pandas-profiling
import pandas as pd
from pandas_profiling import ProfileReport
df = pd.read_csv('Bengaluru_House_Data.csv')
profile = ProfileReport(df, title='Bangalore Housing Data')
profile
```

The following is the output for the code; it generates a comprehensive report containing missing values, high cardinality features, correlation among features, missing values, and many other descriptive statistics.



Warnings

Dataset has 529 (4.0%) duplicate rows	Duplicates
`availability` has a high cardinality: 81 distinct values	High cardinality
`location` has a high cardinality: 1305 distinct values	High cardinality
`society` has a high cardinality: 2688 distinct values	High cardinality
`total_sqft` has a high cardinality: 2117 distinct values	High cardinality
`society` has 5502 (41.3%) missing values	Missing
`balcony` has 609 (4.6%) missing values	Missing

***Figure 7.18**: Partial screenshot of pandas-profiling*

The preceding is a partial screenshot of pandas profile created; it shows how many duplicates the dataset has, which columns have how many unique values, and so on. This dataset is small, having fewer columns, but imagine having a dataset with 200 columns. Pandas profiling will ease up your task of initial analysis. You may find many rows that can be removed based on missing values, zero variance, and so on.

Another package I would like to mention is the feature-selector. But use this one carefully, as I always focus more on machine learning-driven EDA than EDA from the beginning. This package can help you identify low importance variables. I will not add any examples of it here, but the interested reader can follow the documentation.

7.1.2 Practical tips: faster loading and reducing size for faster exploration

Machine learning is an iterative process. We go through feature selection and engineering multiple times based on our problem understanding. So as much as possible, we should optimize steps that repeat. One of such steps is reducing the size of the dataset. This helps a lot. A smaller data frame equals faster loading and faster manipulation. This saves a lot of time. Although this step takes some time in the beginning to perform, it is worth it. So always spend some time reducing dataset size. It doesn't make much difference when the dataset is quite small. So, this is not a mandatory step. But with medium to large datasets, it is a good practice to reduce the size of the dataset first. There are many ways we can reduce the size of the dataset. Here are a few common tips:

1. **Using appropriate types**: If your dataset is stored as int64 but only contains values ranging from -500 to + 500, it's a waste of space

2. **Changing categorical columns that are numerical**: In our use case, size is a numerical variable containing no rooms, so we should change it

3. **Using columns that are important and dropping rest**: Columns that only contain ID information, or columns with zero variance (only one unique row), high missing values, and so on (discussed in detail in Practical Tips 1)

We can use data formats like a feather after preprocessing for faster loading, using the `df.to_feather` method.

7.1.3 Practical tips: merge, join, and concat

As discussed earlier, real-life data is very messy and often present in multiple tables. So, an intuitive understanding of how to join things is a key tool to have. We often use merge, join, and concat functionalities that come with pandas. Here we are not going into the details of merge, join and concat as there is a lot of online material available that readers can go through with Pandas. It is a well-documented library. The idea here is to familiarize the concepts that we mostly use with Pandas. This will give you a head start. Merge and join are used to solve similar problems of merging two tables based on condition, but they have differences. Join uses merge under the hood, but it provides a more efficient way than merge. Concat is combining more data to the current dataframe across rows/columns.

7.1.4 Practical tips: handling dirty data

The following are few common noises/inconsistencies presents in data and how to deal with them:

- **Extraction**:

  Often, the data will be present along with some sort of inconsistent suffix/prefix. For example, the Bangalore housing data size column contains <no_of_rooms> followed by either BHK/Bedrooms and so on. If we just want to extract the room number, we can do that using this code:

  ```
  size = df['size'].str.extract('(^\d+)\s', expand=False)

  df['size'] = pd.to_numeric(size)
  ```

  Here we extract the group starting numbers using regex '(^\d+)\s.' Readers not conversant with regular expressions can take a few crash courses available on YouTube; regexes are important for any data scientist.

  Using this method, you can pretty much extract as much you want.

- **Modification:**

  Modification of any column in a pandas dataset can be done using apply. Apply takes a function that you can write and perform on the entire column.

It is also better to use vectorization (discussed in NumPy appendix) as much as possible because it is the order of magnitude faster than traditional loops/apply. Use `np.where` as much as possible.

- **Header and rename**:

  Often, the header row is not present initially; we can use the header parameter of the `pd.read_csv` method to specify where our header column is present. Many real-world datasets come with inappropriate columns or no columns at all. So, using the rename option is handy with pandas. Examples of these scenarios are present in the appendix for Pandas.

7.1.5 Practical tips: handling huge data

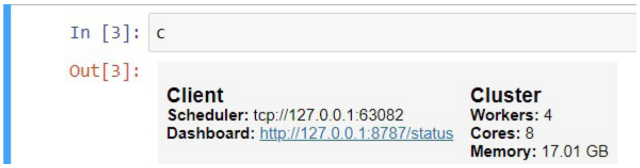
When the dataset is exceptionally large, then we may come across a few issues. Until now, we always used local machines or Google Colab for our tasks. And believe it or not, you can do a lot with this. You may not even require anything more than a setup with few cores (4-8), 16-32 GB of RAM, and a GPU for learning machine learning. And Google Colab is an ideal setup for this. It comes with a lot of libraries installed already too. But in the industry, you may find a large dataset that may not fit your primary memory. So, we cannot process them all at once. So, a general understanding of how to work in such a scenario is needed. Let us consider the following problems, and one by one, we will talk about best practices. So to simulate a low memory scenario, we will run the following code in our local computer and not on Google Colab.

- **When data is larger than primary memory**: For such scenarios, we can use a library called **Dask**. Let us see this in action with code.

  Step 1: Create a Dask client in our local machine. Make sure you install Dask properly on your local PC (follow: **https://docs.dask.org/en/latest/install.html**)

```
import joblib

import dask.distributed

c = dask.distributed.Client()
```

The output of this code:



```
In [3]: c
Out[3]:
Client                                   Cluster
Scheduler: tcp://127.0.0.1:63082         Workers: 4
Dashboard: http://127.0.0.1:8787/status  Cores: 8
                                         Memory: 17.01 GB
```

***Figure 7.19**: Dask client*

This just creates a cluster with the number of cores available in my local computer.

Step 2: Create a dataset that is not going to fit in primary memory. We will use the **`scikit_learn`** datasets to create a small dataset:

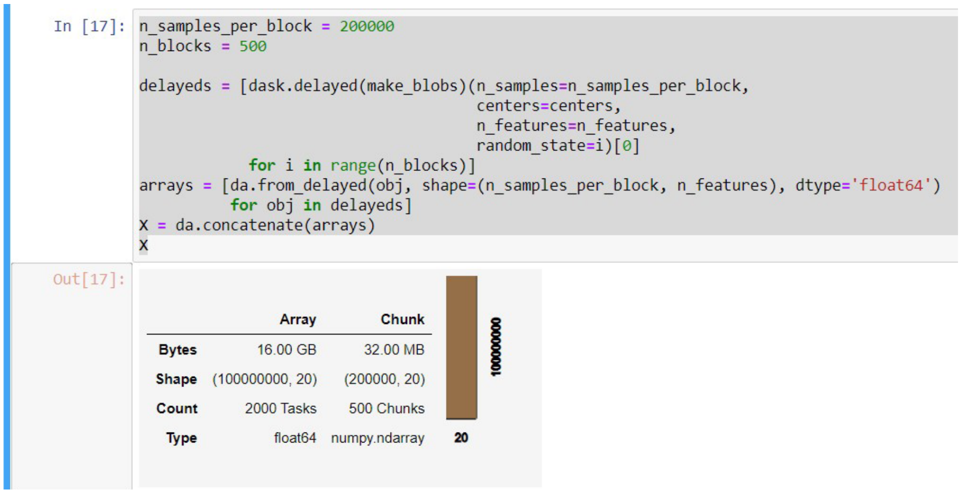
```
import dask.array as da

import dask.delayed

from sklearn.datasets import make_blobs

import numpy as np


n_centers = 12

n_features = 20


X_small, y_small = make_blobs(n_samples=1000, centers=n_centers,
n_features=n_features, random_state=0)


centers = np.zeros((n_centers, n_features))


for i in range(n_centers):

    centers[i] = X_small[y_small == i].mean(0)


centers[:4]
```

We use the **`make_blobs`** function from **`sklearn.datasets`** to create a clustering dataset (later, we will use this dataset to train distributed KMeans). Now, to create a bigger dataset than our primary memory (RAM size), we will use a function called **`dask.delayed`**. This takes a function and makes it delayed; that is, it won't be generated at once and only generated during runtime in the workers.

```
n_samples_per_block = 200000

n_blocks = 500


delayeds = [dask.delayed(make_blobs)(n_samples=n_samples_per_
block,

                                     centers=centers,

                                     n_features=n_features,

                                     random_state=i)[0]
```

```
             for i in range(n_blocks)]
arrays = [da.from_delayed(obj, shape=(n_samples_per_block, n_
features), dtype='float64')
            for obj in delayeds]
X = da.concatenate(arrays)
X
```

We use the same function to create a huge dataset, 200000 * 500 rows (200000 samples per block, for 500 blocks). Each row has 20 features. The following is the output of this code:



```
In [17]: n_samples_per_block = 200000
         n_blocks = 500

         delayeds = [dask.delayed(make_blobs)(n_samples=n_samples_per_block,
                                              centers=centers,
                                              n_features=n_features,
                                              random_state=i)[0]
                     for i in range(n_blocks)]
         arrays = [da.from_delayed(obj, shape=(n_samples_per_block, n_features), dtype='float64')
                   for obj in delayeds]
         X = da.concatenate(arrays)
         X
Out[17]:
```

	Array	Chunk
Bytes	16.00 GB	32.00 MB
Shape	(100000000, 20)	(200000, 20)
Count	2000 Tasks	500 Chunks
Type	float64	numpy.ndarray

Figure 7.20: Large dataset with dask.delayed

This dataset is 16 GB, but dask doesn't load everything into memory (that won't be possible since we don't have that much RAM). Still, you can load all of the data if the dask client is a distributed cluster of machines using **`df.persist()`**. But don't do this on your local machine if you don't have enough RAM.

We need to use the dask_ml models that inherently support RAM computations through tasks to run a model. Let us fit a KMeans model using dask. For this, install Dask ML, which comes up with distributed machine learning models (follow **https://ml.dask.org/install.html**).

```
from dask_ml.cluster import KMeans
clf = KMeans(init_max_iter=3, oversampling_factor=10)
clf.fit(X)
```

Now, this will run; if we use normal pandas and scikit-learn, we will get out of memory error as the dataset is too big. To see how Dask does this, let's head over to the dash dashboard. Remember when we created the client, the dashboard location is created at **http://127.0.0.1:8787/status**. Since it is on a local PC, the IP is localhost. The following is the screenshot of the dashboard when we run the preceding code:

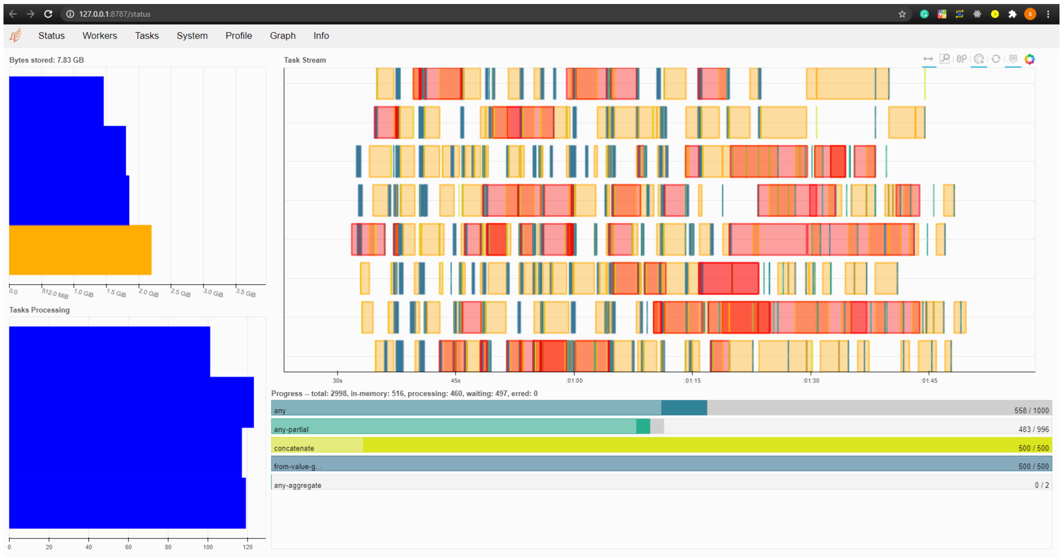
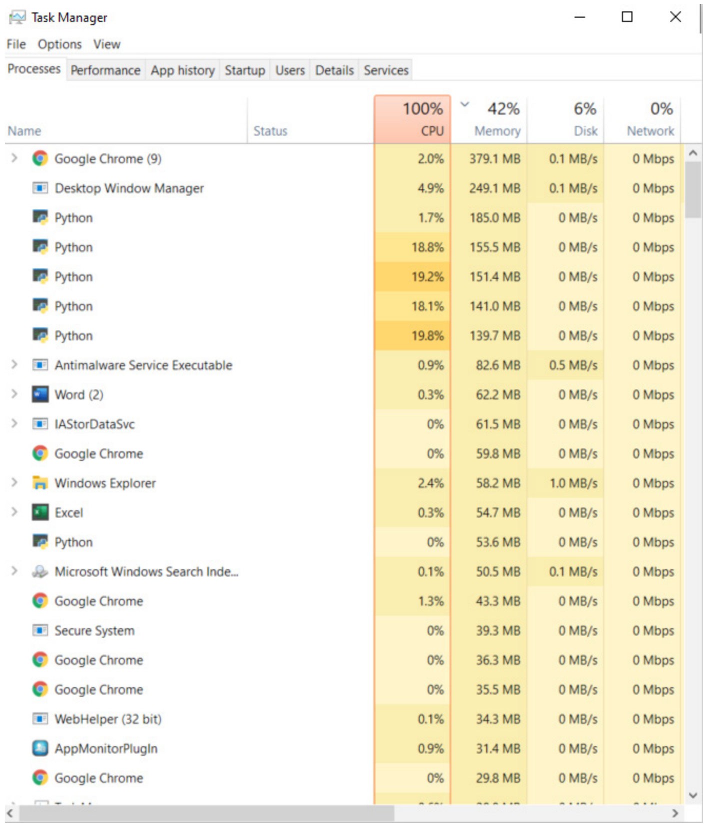


***Figure 7.21**: Dash dashboard*

You may find it intimidating at first, as many things are going on, and I would suggest the reader run this code and see the magic of dask running this huge dataset. So, what dask does is load as much data into primary memory as required for a particular task and combine the result in the end. There are eight horizontal bars in the top right figure, which shows all eight workers the dask client has. It uses all cores (4 cores – 8 logical in my laptop) to run everything in parallel. This will not reduce training time, as we can do the same thing with scikit-learn with n_jobs = -1, but since the data cannot be loaded into primary memory, we cannot just use scikit-learn in this case. Let us now head over to the task manager to check what is happening.



Task Manager

File Options View

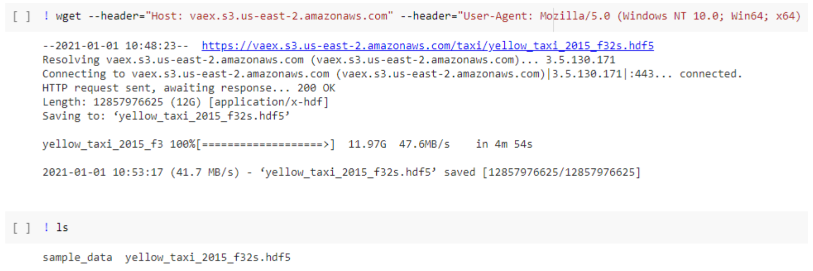
Processes | Performance | App history | Startup | Users | Details | Services

Name	Status	100% CPU	42% Memory	6% Disk	0% Network
Google Chrome (9)		2.0%	379.1 MB	0.1 MB/s	0 Mbps
Desktop Window Manager		4.9%	249.1 MB	0.1 MB/s	0 Mbps
Python		1.7%	185.0 MB	0 MB/s	0 Mbps
Python		18.8%	155.5 MB	0 MB/s	0 Mbps
Python		19.2%	151.4 MB	0 MB/s	0 Mbps
Python		18.1%	141.0 MB	0 MB/s	0 Mbps
Python		19.8%	139.7 MB	0 MB/s	0 Mbps
Antimalware Service Executable		0.9%	82.6 MB	0.5 MB/s	0 Mbps
Word (2)		0.3%	62.2 MB	0 MB/s	0 Mbps
IAStorDataSvc		0%	61.5 MB	0 MB/s	0 Mbps
Google Chrome		0%	59.8 MB	0 MB/s	0 Mbps
Windows Explorer		2.4%	58.2 MB	1.0 MB/s	0 Mbps
Excel		0.3%	54.7 MB	0 MB/s	0 Mbps
Python		0%	53.6 MB	0 MB/s	0 Mbps
Microsoft Windows Search Inde...		0.1%	50.5 MB	0.1 MB/s	0 Mbps
Google Chrome		1.3%	43.3 MB	0 MB/s	0 Mbps
Secure System		0%	39.3 MB	0 MB/s	0 Mbps
Google Chrome		0%	36.3 MB	0 MB/s	0 Mbps
Google Chrome		0%	35.5 MB	0 MB/s	0 Mbps
WebHelper (32 bit)		0.1%	34.3 MB	0 MB/s	0 Mbps
AppMonitorPlugIn		0.9%	31.4 MB	0 MB/s	0 Mbps
Google Chrome		0%	29.8 MB	0 MB/s	0 Mbps

Figure 7.22: *Task manager during dask*

The CPU usage is 100%; the Python processes dask uses all available CPUs has created.

- **Huge data, but I want to visualize and do EDA**: Here, another library called vaex is useful; first, let's download a huge dataset of 146 million rows; this is a New York taxi dataset and can be found here - **https://vaex.readthedocs.io/en/latest/datasets.html**. The following is the screenshot of downloading it into Colab. Follow the curlWget trick discussed in *Chapter 1: Introduction to Machine Learning*)



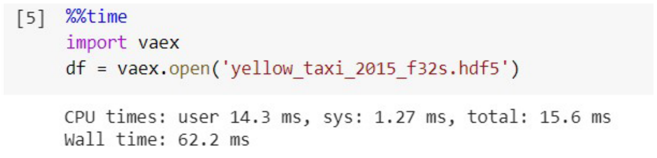
```
[ ]  ! wget --header="Host: vaex.s3.us-east-2.amazonaws.com" --header="User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)

     --2021-01-01 10:48:23--  https://vaex.s3.us-east-2.amazonaws.com/taxi/yellow_taxi_2015_f32s.hdf5
     Resolving vaex.s3.us-east-2.amazonaws.com (vaex.s3.us-east-2.amazonaws.com)... 3.5.130.171
     Connecting to vaex.s3.us-east-2.amazonaws.com (vaex.s3.us-east-2.amazonaws.com)|3.5.130.171|:443... connected.
     HTTP request sent, awaiting response... 200 OK
     Length: 12857976625 (12G) [application/x-hdf]
     Saving to: ‘yellow_taxi_2015_f32s.hdf5’

     yellow_taxi_2015_f3 100%[===================>]  11.97G  47.6MB/s    in 4m 54s

     2021-01-01 10:53:17 (41.7 MB/s) - ‘yellow_taxi_2015_f32s.hdf5’ saved [12857976625/12857976625]


[ ]  ! ls

     sample_data  yellow_taxi_2015_f32s.hdf5
```

Figure 7.23: *Downloading dataset for visualization*

Let us try to open it using vaex.

```
%%time
! pip install vaex
import vaex
df = vaex.open('yellow_taxi_2015_f32s.hdf5')
```

The output of the preceding code:

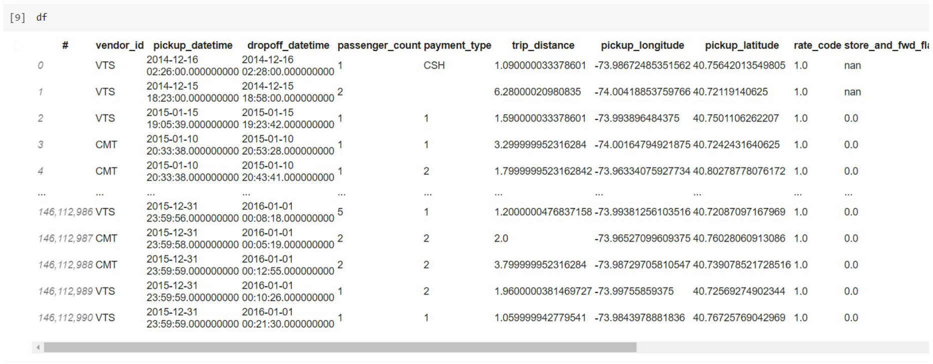


```
[5] %%time
    import vaex
    df = vaex.open('yellow_taxi_2015_f32s.hdf5')

    CPU times: user 14.3 ms, sys: 1.27 ms, total: 15.6 ms
    Wall time: 62.2 ms
```

Figure 7.24: *Opening huge dataset with Vaex*

Surprisingly, this only took 14.3 milliseconds to open a 146 million row dataset.

The way Vaex does it is it only loads stuff that is needed. Unlike pandas (without chunksize), it doesn't load everything to primary memory unless required. It's kind of loads only the schema. You can check the dataset entries (first few and last few) by simply typing **`df`**. The following is the output:



```
[9] df
```

#	vendor_id	pickup_datetime	dropoff_datetime	passenger_count	payment_type	trip_distance	pickup_longitude	pickup_latitude	rate_code	store_and_fwd_fla
0	VTS	2014-12-16 02:26:00.000000000	2014-12-16 02:28:00.000000000	1	CSH	1.090000033378601	-73.98672485351562	40.75642013549805	1.0	nan
1	VTS	2014-12-15 18:23:00.000000000	2014-12-15 18:58:00.000000000	2		6.28000020980835	-74.00418853759766	40.72119140625	1.0	nan
2	VTS	2015-01-15 19:05:39.000000000	2015-01-15 19:23:42.000000000	1	1	1.590000033378601	-73.993896484375	40.7501106262207	1.0	0.0
3	CMT	2015-01-10 20:33:38.000000000	2015-01-10 20:53:28.000000000	1	1	3.299999952316284	-74.00164794921875	40.7242431640625	1.0	0.0
4	CMT	2015-01-10 20:33:38.000000000	2015-01-10 20:43:41.000000000	1	2	1.7999999523162842	-73.96334075927734	40.80278778076172	1.0	0.0
...	...	...	...	...	...	...	...	...	...	...
146,112,986	VTS	2015-12-31 23:59:56.000000000	2016-01-01 00:08:18.000000000	5	1	1.2000000476837158	-73.99381256103516	40.72087097167969	1.0	0.0
146,112,987	CMT	2015-12-31 23:59:58.000000000	2016-01-01 00:05:19.000000000	2	2	2.0	-73.96527099609375	40.76028060913086	1.0	0.0
146,112,988	CMT	2015-12-31 23:59:59.000000000	2016-01-01 00:12:55.000000000	2	2	3.799999952316284	-73.98729705810547	40.739078521728516	1.0	0.0
146,112,989	VTS	2015-12-31 23:59:59.000000000	2016-01-01 00:10:26.000000000	1	2	1.9600000381469727	-73.99755859375	40.72569274902344	1.0	0.0
146,112,990	VTS	2015-12-31 23:59:59.000000000	2016-01-01 00:21:30.000000000	1	1	1.059999942779541	-73.9843978881836	40.76725769042969	1.0	0.0

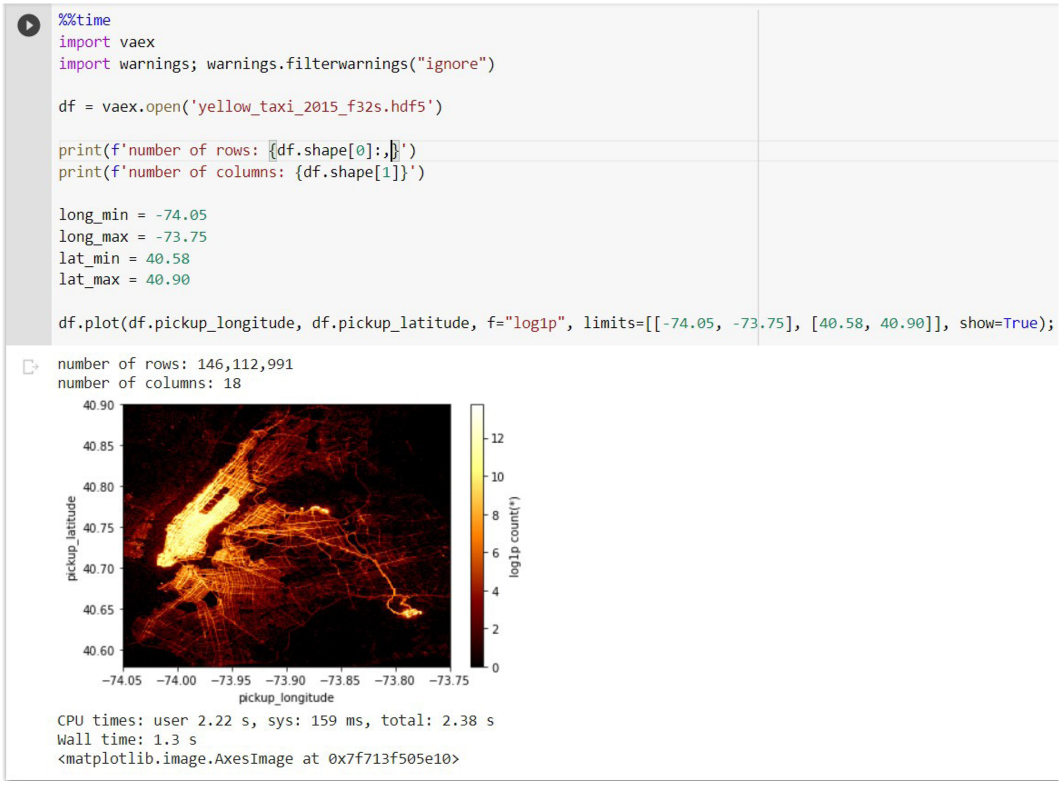
Figure 7.25: *Sneak peek into the huge dataset*

Let us do some visualizations; let's try to plot which locations have the highest pickups.

```
%%time

df = vaex.open('yellow_taxi_2015_f32s.hdf5')

print(f'number of rows: {df.shape[0]:,}')
print(f'number of columns: {df.shape[1]}')
```

```
long_min = -74.05
long_max = -73.75
lat_min = 40.58
lat_max = 40.90

df.plot(df.pickup_longitude, df.pickup_latitude, f="log1p",
limits=[[-74.05, -73.75], [40.58, 40.90]], show=True);
```

We are trying to plot over a limited location (lat-40.58, long—74.05) to (lat-40.90, long—73.75) because the entire data will be too crowded. The following is the output of this code:



```
%%time
import vaex
import warnings; warnings.filterwarnings("ignore")

df = vaex.open('yellow_taxi_2015_f32s.hdf5')

print(f'number of rows: {df.shape[0]:,}')
print(f'number of columns: {df.shape[1]}')

long_min = -74.05
long_max = -73.75
lat_min = 40.58
lat_max = 40.90

df.plot(df.pickup_longitude, df.pickup_latitude, f="log1p", limits=[[-74.05, -73.75], [40.58, 40.90]], show=True);
```

```
number of rows: 146,112,991
number of columns: 18
```

```
CPU times: user 2.22 s, sys: 159 ms, total: 2.38 s
Wall time: 1.3 s
<matplotlib.image.AxesImage at 0x7f713f505e10>
```

***Figure 7.26**: Plotting count of pickup locations*

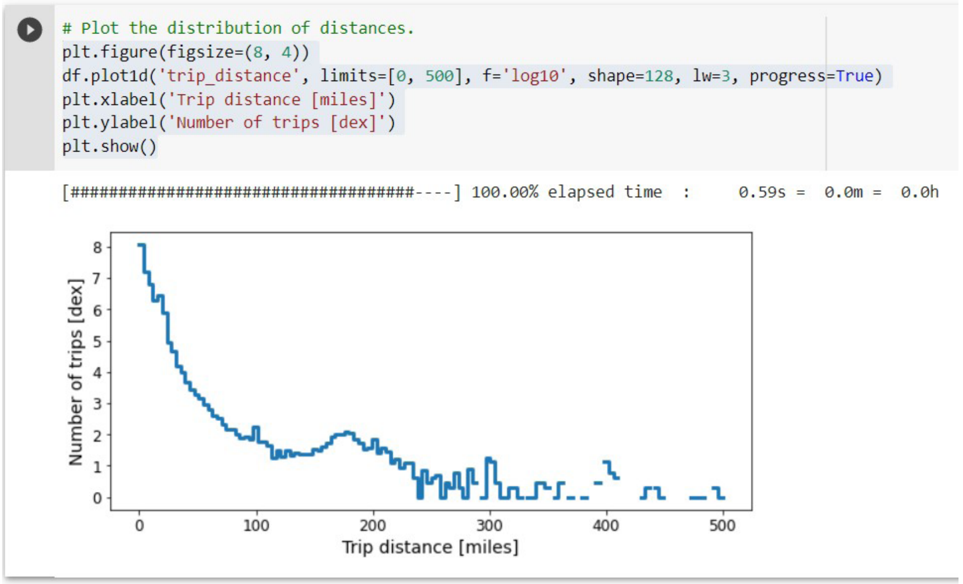
So, it took only 2.38 seconds to plot this huge graph, and we can see where the most pickups happen.

Now let us try to see the distance vs. no of trips. As per intuition, many short trips should happen so the graph should be skewed to left. The following is the code:

```
import matplotlib.pyplot as plt
```

```
plt.figure(figsize=(8, 4))
df.plot1d('trip_distance', limits=[0, 500], f='log10', shape=128, lw=3,
progress=True)
plt.xlabel('Trip distance [miles]')
plt.ylabel('Number of trips [dex]')
plt.show()
```

As you can see, we use a matplotlib plot to control the plot (e.g., title, color, figure size, and so on). The following is the output for this plot:



```
# Plot the distribution of distances.
plt.figure(figsize=(8, 4))
df.plot1d('trip_distance', limits=[0, 500], f='log10', shape=128, lw=3, progress=True)
plt.xlabel('Trip distance [miles]')
plt.ylabel('Number of trips [dex]')
plt.show()
```

```
[####################################----] 100.00% elapsed time  :     0.59s =  0.0m =  0.0h
```

Figure 7.27: No of trips vs. trip distance

As per our intuition, the plot also shows many trips that happened with a small trip distance. Let us now try to identify the most expensive areas by plotting the mean fare amount w.r.t. location. Before that, let us take a small subset of data for a particular location; this will make the visualizations better. Also, when we analyze data, we don't generally look into everything; we generally filter stuff. Filtering data with vaex is also very simple.

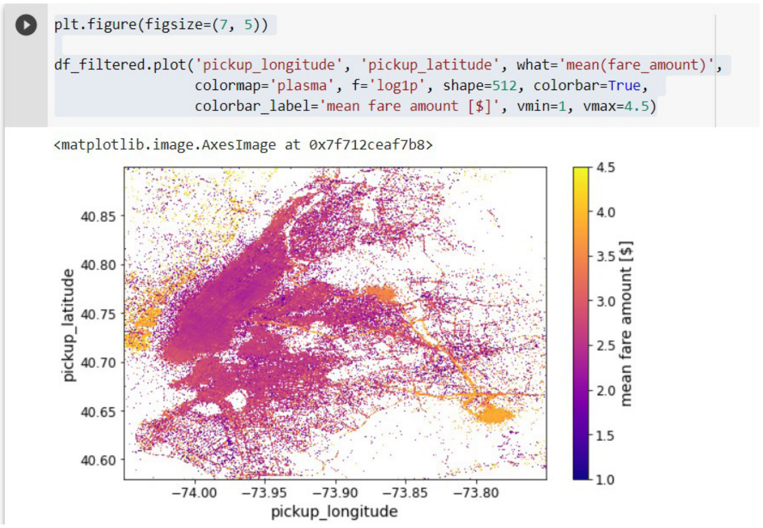
```
long_min = -74.05
long_max = -73.75
lat_min = 40.58
lat_max = 40.90


# Make a selection based on the boundaries
```

```
df_filtered = df[(df.pickup_longitude > long_min)  & (df.pickup_longitude
< long_max) & \
                 (df.pickup_latitude > lat_min)    & (df.pickup_latitude
< lat_max) & \
                 (df.dropoff_longitude > long_min) & (df.dropoff_
longitude < long_max) & \
                 (df.dropoff_latitude > lat_min)   & (df.dropoff_
latitude < lat_max)]
```

It is exactly like pandas. Now, let us plot the desired location vs. mean fare. The following is the code:

```
plt.figure(figsize=(7, 5))
df_filtered.plot('pickup_longitude', 'pickup_latitude', what='mean(fare_
amount)',
                 colormap='plasma', f='log1p', shape=512, colorbar=True,
                 colorbar_label='mean fare amount [$]', vmin=1,
vmax=4.5)
```

Here you can see we are using the plot method of the vaex dataset and passing a parameter called what; this parameter controls what exactly you want to plot; in the preceding cases, we didn't pass anything for what, so by default, it takes count. Here we are passing mean (**fare_amount**), so it will plot the mean fare amount. The following is the output of the code:



```
plt.figure(figsize=(7, 5))

df_filtered.plot('pickup_longitude', 'pickup_latitude', what='mean(fare_amount)',
                 colormap='plasma', f='log1p', shape=512, colorbar=True,
                 colorbar_label='mean fare amount [$]', vmin=1, vmax=4.5)

<matplotlib.image.AxesImage at 0x7f712ceaf7b8>
```

Figure 7.28: Mean amount for different distances

Using what parameter you can also create multiple graphs, I would recommend the interested reader go through the vaex documentation for more options for the API.

Note: The installation of vaex in Google Colab using pip may cause some issues due to outdated ipython install; for now, it is solved by upgrading python and ipython and then installing vaex. In the future, this may get resolved. Still, such issues related to environment creation will come in machine learning, so the reader may need to investigate StackOverflow and other resources to resolve such issues.

- **Tuning Hyperparameters over a lot of settings**: If we have many parameters to tune during Hyperparameter Tuning, we can use Dask to scale with multiple machines in a cluster
- **Prediction at scale**: Use dask to predict at scale. It's the same as "*.predict*, " but even if the prediction set is large, dask handles it automatically(that is, predicts it in parallel)

To summarize, the key takeaway is that there are different libraries/tools available even with huge data and a low-cost machine. Again, the libraries' capabilities discussed are not exhaustive, and when you dig deeper into the documentation, you may find more interesting capabilities. So, the following is a quick summary of what to investigate when we come across infrastructure limitations:

- **A huge dataset that cannot load into memory** = Use Dask, Dask-ML.
- **A lot of hyperparameters to tune** = Use joblib.backend('dask') if you have a distributed system, scikit-learn n_jobs=-1 for parallel if you have a single machine.
- **Visualization of huge datasets, Exploratory Data Analysis** = Vaex
- **Prediction at scale** = Dask

7.1.6 Practical tips 6: pipelines

Use pipelines; they make code much more elegant and fault-tolerant. Also, pipelines are useful during the serialization of a model. You do not need to store every preprocessing object separately. For example, suppose you first make a dimensionality reduction followed by a classifier. In that case, you need to save both the model separately, and during prediction, you need to load them both. If you use a pipeline, you don't need to do any of that. You can simply store the pipeline object. Also, pipelines make the code more readable. Let us go through a simple example.

```
import numpy as np

import matplotlib.pyplot as plt

import pandas as pd


from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

# we will take iris from load_dataset as an example
X, y = datasets.load_iris(return_X_y=True)

# dimentionality reduction
pca = PCA(n_components=2)
# Logistic Regression Model
logistic = LogisticRegression(max_iter=10000, tol=0.1)

# creating pipeline
pipe = Pipeline(steps=[('pca', pca), ('logistic', logistic)])

# training pipeline
pipe.fit(X, y)

# score
y_pred = pipe.predict(X)

# accuracy
sum(y_pred == y) / len(y)
```

Here we take a simple example with the iris dataset. First, we make a dimensionality reduction (reducing the columns) on the iris dataset to two columns. Then we use a classification model with logistic regression. Finally, in line 19, we create a pipeline by doing PCA first, followed by logistic regression. As you can see, the model pipe is the only instance we need to save. Again, this is just scratching the surface of pipelines; we have many features for this. Follow the scikit-learn documentation for the same.

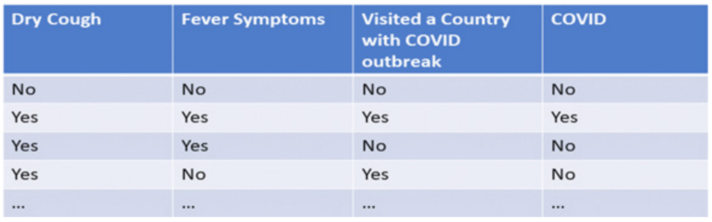
7.2 Understanding random forest model

This section will deep dive into some of the common models used in machine learning with tabular data. The list of models that we will explore is exhaustive, and there are many models present, as discussed earlier in this chapter. The goal is to deep dive into some of the models that we commonly use so that readers may appreciate the inner workings of the models. Although this is not entirely necessary nowadays, I would recommend understanding at least a few basic models completely. The motivation behind this is, some of the concepts are invaluable and can be used in different places if we go through the details of some models. For example., bootstrapping, ensembling and feature importance are a few of the key ideas behind the random forest, and it is not limited to this model only. These concepts appear in many models in machine learning. Another motivation behind this is research. If you want to do research, only practical implementation won't work. You will require skills to read math and understand key concepts. So, this is a nice exercise for people who want to delve deeper into the world of machine learning. Finally, this is another way to understand the strengths and weaknesses of models.

So, the main purpose of this book is to give readers a lot of implementation skills, along with few deeper understandings of some of the models. This is enough to start a career in machine learning. But this is in no way exhaustive. With that in mind, let's understand how a random forest works first.

7.2.1 Random forest

To understand how random forest works, first, we need to understand how a decision tree works. Let us take a small example.



Dry Cough	Fever Symptoms	Visited a Country with COVID outbreak	COVID
No	No	No	No
Yes	Yes	Yes	Yes
Yes	Yes	No	No
Yes	No	Yes	No
...	...	...	...

***Figure 7.29**: Sample data for decision tree*

The preceding table shows the data for people having COVID. I have taken three simple features: if the person has a dry cough, has fever symptoms, or recently visited a country with COVID outbreak. The last column is our target variable, which is if the person has coronavirus or not. This is just a toy example for understanding decision trees. The data is irrelevant. Also, I have only shown the first four rows to simplify; we can assume the data contains many rows. The data may also contain some missing values. We can either use the missing value treatment or skip it. I will be skipping it, but when we learn about the random forest, we will understand how random forest interestingly handles missing values.

So, a decision tree is simple intuitively. For example, if someone asks you how you will identify a person with COVID, then the first thing you may ask is, "*Has the person been to any country with Covid outbreak?*" followed by "*Does the person have fever symptoms?*" and so on like the following tree:

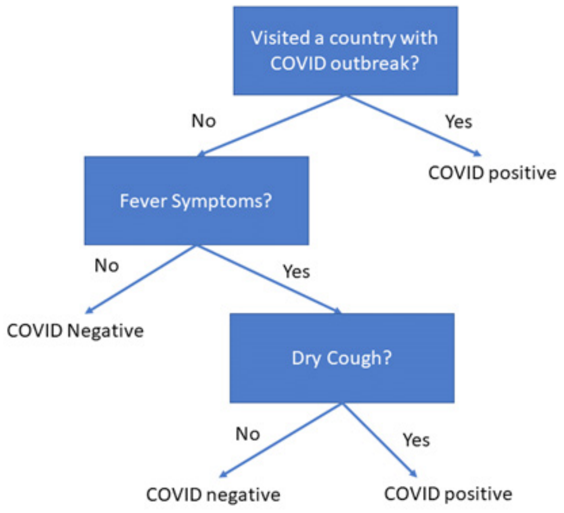


Figure 7.30: *Simple decision tree*

We may come up with the preceding tree (this is just for demonstration purposes only). So as per the tree, if a person visited China, he is COVID positive; if he has a fever and dry cough, he has COVID, otherwise negative. Intuitively, you are creating a decision tree also. We require two things to construct it; *which question should we ask first, second, and so on? Based on the answer, what decision will we take?*

To answer the first question, a simple intuition is to select the best predictor first; in the preceding tree, the best predictor selected is the third feature, if the person visited a country with COVID outbreak or not. But how to do it programmatically? The idea is to identify the purest split point. For example, if there are 100 examples and out of that, 50 visited a country with COVID outbreak, and out of that 50, 30 had a fever, and 20 did not (skipping dry cough as a feature for now). The rest 50, who has not visited a country with COVID outbreak, say 40 got fever and 10 did not. And that first 50, who all visited China, all 50 got COVID. And rest all did not. Then, we can select visited China as our split point. Because '50 'yes (COVID +ve that is, if visited China) in the first split, 50 'No' (COVID -ve that is, if not visited a country with COVID outbreak) in the second split. Both sets contain either 'yes' (COVID +ve) or 'no' (COVID -ve). That is what we call pure sets.

In a real-life scenario, we won't find a feature that is 100% pure like this example. The data would be something like this:

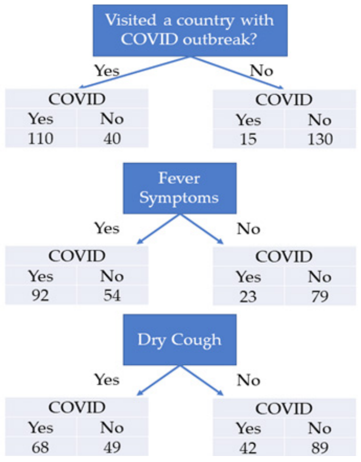


***Figure 7.31**: Impurity*

As you can see in the image, none of the features confirms COVID. All of them are **impure**. So, to select one of the three splits, we need to identify a metric that measures this impurity to choose the split with the least impurity. And the metric we generally use is the 'Gini' impurity. But you can have any impurity measure if it follows the following three rules:

1. If one of the categories (which in our case is yes/no for COVID +ve/-ve, but it can be more than two also for multiclass classifications) is higher, this means the set is pure, so impurity is less = the metric should give a small number.
2. If all categories are similar in number, it's impure = the metric should give a large number.
3. If we add another category to the mix (for example, if we predict the likelihood of COVID as "high," "medium," and "low," there are three categories. And if we add "very high" as another category), this means the set just became more impure = the metric should give a large number.

The idea behind metric is that it should give a single number for us to compare. The Gini index solves this measuring impurity problem for us; it is given by:

$$I_G(p) = 1 - \sum_{i=1}^{N} p_i^2$$

It is quite simple; N is the number of categories. In our case, it is 2. So, the formula reduces to:

$$1 - p(Yes)^2 - p(No)^2$$

Let us check how it follows the three rules. The formula can be written as follows:

p(A)(1-p(A)) + p(B)(1-p(B)), where A and B for our case is Yes, No.

For the first rule, if one of the categories is high (that is, p(A) ~ 1) (say A), 1 – p(A) will be around zero. And p(B) will also be around zero. As p(A) + p(B) = 1(either 'Yes' or 'No'). So, impurity will be very less.

Coming to the second rule, if both are of ½ probability, that is, p(A) = p(B) = 0.5. This means the set is most impure. Both categories are present equally. Then, the Gini impurity will be around 0.5, which is high but not 1 because if we add another category (C) and the split is 1/3, 1/3, and 1/3. Then this set is more impure than before. And plugging in the numbers, you will get 0.6667. This satisfies 3rd rule (adding more categories should increase impurity). Hence, the Gini impurity captures all three rules and can be used as an impurity measure.

We calculate the Gini impurity at each split as follows:

Gini Impurity (for 'visited a country with COVID outbreak' as a split point) = Gini Impurity (visited a country with COVID outbreak = Yes) + Gini Impurity (visited a country with COVID outbreak = No)

$$= \left\{\frac{110}{110+40}\right\}^2 - \left\{\frac{40}{110+40}\right\}^2 + 1 - \left\{\frac{15}{15+130}\right\}^2 - \left\{\frac{130}{15+130}\right\}^2$$

= 0.4 + 0.189 = 0.589**

(** Actually, we take a weighted average since the numbers are not equal for each set. There are 150 examples in the first set and 145 examples in the second, so we multiply 150/ (150+145) to 0.4 and 145/ (150+145) to 0.189. which makes the Gini impurity for '*visited China*' split to 0.29)

Similarly, we can calculate for the other two trees and choose the least impurity split. So, based on our data, let us say, "*visited a country with COVID outbreak*" has the least Gini impurity. But this is just the first split. We then repeat the same(recursively) for the other two features ("*Dry Cough*," "*Has Fever*") in the next level. Calculate the Gini impurity for both splits and then choose the least. But if the Gini impurity does not decrease with any of the split or separating data further, then we simply stop. e.g., in *figure 7.2*, the node to the right of the root node, that is, visited China, is empty. This means that dividing the data based on either "*Dry Cough*" or "*Has fever*" did not reduce the Gini impurity (which is 0.189*weight = 0.09). So, we stopped there. So, to summarize, we follow three steps to create a decision tree:

1. Calculate the Gini impurity for all the remaining features at each node.
2. If the node already has the lowest Gini score, then no need to split. The node becomes a *leaf*.

3. If separating the data improves impurity, then split based on the best Gini impurity.

Now, this data was all categorical, but we may have numerical data. But this still works; we formulate the nodes differently, for example, if we have another feature, temperature, continuous.

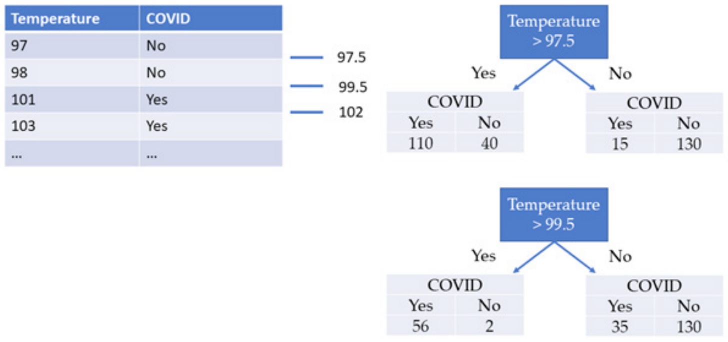


***Figure 7.32**: Continuous features*

Here we take the average for consecutive entries and create the node as shown, and then calculate the Gini Entropy at each of these splits and choose the one with the least impurity as we have done before.

One disadvantage with decision trees is that they tend to overfit. Because as discussed in *Chapter 5: Model Building and Tuning*, they tend to overfit if you increase the depth. Also, trees can be built specifically and may not generalize (as we are taking all input and all features into consideration). Random forest solves this problem by introducing bagging.

7.2.1.1 Bagging

Bagging stands for bootstrap aggregation. It is a combination of two things. We first bootstrap the data. This means we take the original input data with, say, M data points and then create N different folds (like K-Fold cross validation), but with one caveat. We allow repetition. So, all N folds have M data points equal to the size of the original data, but some of the data points are repeated. Then, we train different models (we can use any model since bagging is an ensemble method, but we use decision trees as base models). Finally, we aggregate the output of all these different models/estimators. Aggregation can be based on majority voting in the case of categorical output or classification. In case of a regression problem, we can take an average. This is where the **`n_estimator`** parameter comes into the picture with scikit-learn.

The use of bootstrapping data has two advantages. The first is specialization. We allow some trees to focus on the part of data. The second is validation; this gives rise to an interesting way of validating a model called out of bag error, which is quite simple. As we are bootstrapping data, some trees will have examples that it has never seen during training as we are allowing replacement in data when we create bootstrapped dataset. During training, we can keep track of these examples, which we call out-of-bag samples. We can use this to validate the set of trees during validation on which that bag of examples is not trained.

The following is a pictorial representation of bootstrap aggregation/bagging.

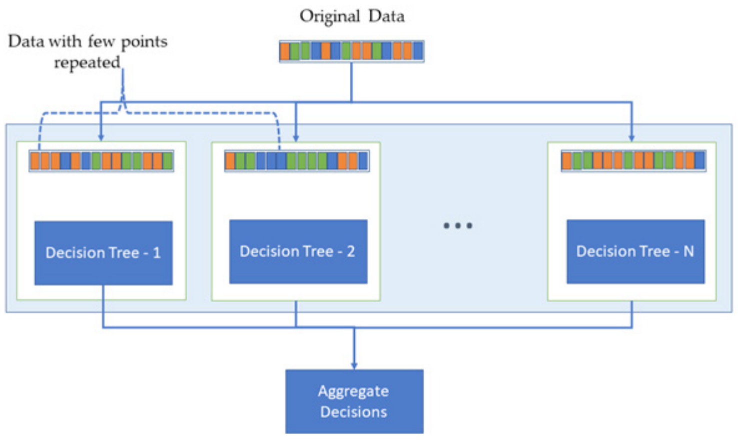


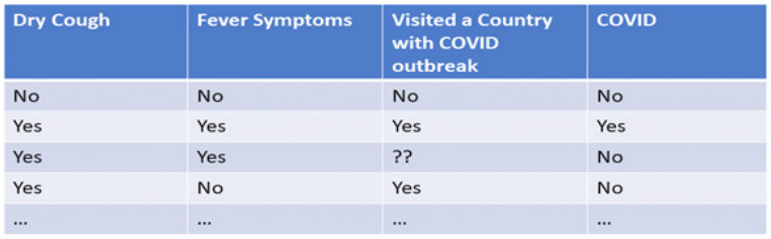
***Figure 7.33**: Bootstrap aggregation*

As you can see in this example, the data is split into N folds with repetition.

The random forest also does not take all columns for creating a tree. It also takes a subset of the columns randomly. This also reduces overfitting further as each tree learns on a different subset(subsampling) of features.

7.2.1.2 Missing value treatment

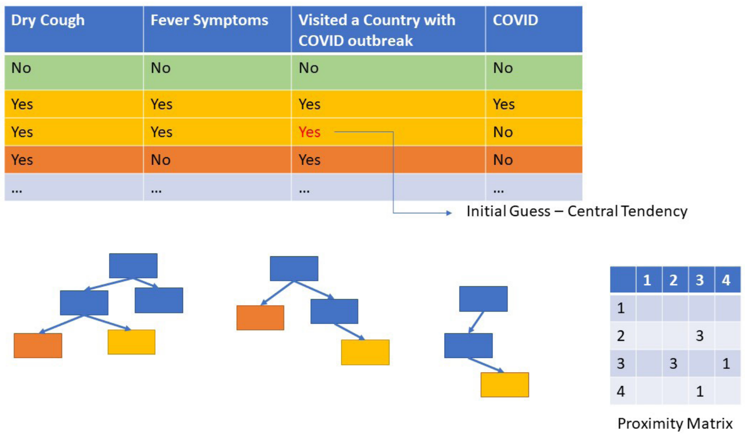
The random forest also interestingly handles missing values. Let us take the following example:



Dry Cough	Fever Symptoms	Visited a Country with COVID outbreak	COVID
No	No	No	No
Yes	Yes	Yes	Yes
Yes	Yes	??	No
Yes	No	Yes	No
...	...	...	...

***Figure 7.34**: Missing values*

Here "*visited a country with COVID outbreak*" has a missing value in row 3. The idea is to fill that row with the value from similar rows. The first random forest model fills that missing value through a central tendency as we generally do, so here we can use mode as the data is categorical. The following is an image explaining this:



Dry Cough	Fever Symptoms	Visited a Country with COVID outbreak	COVID
No	No	No	No
Yes	Yes	Yes	Yes
Yes	Yes	Yes	No
Yes	No	Yes	No
...	...	...	...

	1	2	3	4
1				
2			3	
3		3		1
4			1	

***Figure 7.35**: Missing value treatment in random forest*

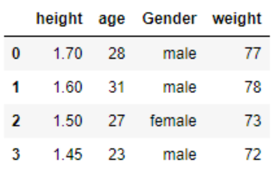
So, with this data, it goes ahead and builds the tree, passing these values and seeing which rows end together and creates a proximity matrix. For example, rows 2 and 3 landed on the same leaf node in 3 decision trees. Hence you can see in the proximity matrix, three is written for row 2, column 3. We create this table for all the rows and then fill the missing value using this matrix. We simply identify the probability of option for the missing value as the frequency of option multiplied by the weight of option from the proximity matrix. For row 3, p(yes) = frequency of yes (in visited a country with COVID outbreak) * weight of yes. Weight is calculated once the full proximity matrix is made and normalized. We won't go into the details of that, and as an exercise, try to create a proximity matrix for all 4 data points and identify the weight. But the intuition is using the proximity matrix; we fill the missing value by considering the similar rows.

7.3 Other models – XGBoost, Neural Networks

The next model we are going to explore is the XGBoost model. XGBoost is based on an ensembling technique called **boosting**. We will first explore the idea of boosting weak models. Then we will explore some of the added features of **eXtreme Gradient Boosting** (**XGBoost**).

7.3.1 Gradient boosting

Let us understand what boosting is with an example. Consider the following dataset:



	height	age	Gender	weight
0	1.70	28	male	77
1	1.60	31	male	78
2	1.50	27	female	73
3	1.45	23	male	72

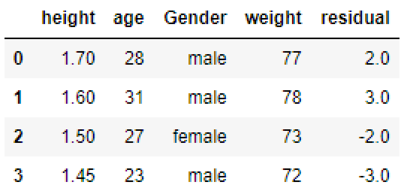
***Figure 7.36**: A sample dataset*

The code to create the preceding dataset:

```
data = {"height": [1.7, 1.6, 1.5, 1.45], "age": [
    28, 31, 27, 23], "Gender": ['male', 'male', 'female', 'male'],
"weight": [77, 78, 73, 72]}

import pandas as pd
df = pd.DataFrame(data)

df
```

The data contains three features (height, age, and gender) and a target variable: weight. We will run our gradient boosting algorithm on this dataset to predict the weight. We will start with the best we can do if we have no access to any feature variables. So, what is a good estimate for weight if you are only provided with the weight column? The best we can do is take the central tendency of the data. We can start with the mean. Here, the mean is 75.

But this is a bad model. Isn't it? But what if we use this model first and then create another model that tries to fit the residuals or errors. For example, for the first row, the residual is 77-75 = 2. Similarly, we can calculate the residual from all of the rows and try fitting another model (in general, you can use any model next, but ideally, we use a tree-based model). The following is the dataset that the new model will use:



	height	age	Gender	weight	residual
0	1.70	28	male	77	2.0
1	1.60	31	male	78	3.0
2	1.50	27	female	73	-2.0
3	1.45	23	male	72	-3.0

***Figure 7.37**: Residuals*

The code to create residuals:

```
mean = df.weight.mean()
df['residual'] = df['weight'] - mean

df
```

If you build a decision tree with the preceding data, it will look as follows:

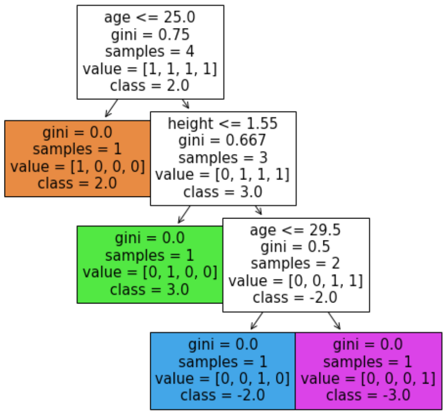


Figure 7.38: *Decision tree*

The code to create the preceding output:

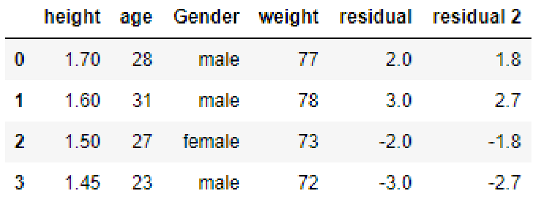
```
# creating dummy variables for categorical data
X = pd.get_dummies(df[['height', 'age', 'Gender']]).values
y = df['residual'].values

# fitting model
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier()
model.fit(X, y)

#plotting the tree
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 8))
plot_tree(model, feature_names=pd.get_dummies(
    df[['height', 'age', 'Gender']]).columns, filled=True, class_
names=[str(i) for i in y])
plt.show()
```

In the model tree figure, ignore the splits here as the data is a dummy one; focus on the leaf nodes (the ones which are colored), you can see the class (since it's a regression problem, the class is the target values) which are residuals. And the decision tree is exactly predicting the residuals.

Now we can combine the two models (the average model + the decision tree model) in sequence to get our prediction. So, for each data point in our dataset, we will pass it through the two models, that is, 75 + the prediction from the second model.

If you pass the first row, you will get *75 + 2 = 77*, which is exactly the weight/target for that value. This is again bad, as we are overfitting training data. So, we won't take the entire prediction rather a fraction of the prediction, *75 + alpha * 2*. This alpha decides how many fractions of the model we should take.

Since we take a partial prediction from the second model, we will again get some error between actual residuals and predicted residuals. The following is the second residual if we take alpha as 0.1:



	height	age	Gender	weight	residual	residual 2
0	1.70	28	male	77	2.0	1.8
1	1.60	31	male	78	3.0	2.7
2	1.50	27	female	73	-2.0	-1.8
3	1.45	23	male	72	-3.0	-2.7

Figure 7.39: *Second residual*

We will repeat this process, adding more and more trees until we reach a maximum number of models or not improving the residual (error). This is boosting, creating weak learners in sequence to create a strong learner. This is also the basis of XGBoost (although XGBoost uses a different tree and many more features).

7.3.2 Features of XGBoost

The key algorithm of an XGBoost model is boosting trees like what we discussed, but the major difference is the vast number of optimizations and features that XGBoost comes up with. Let us discuss a few of the features as follows:

- Approximate algorithms

  One of the memory-intensive areas of boosting is the split finding (when building a tree). To find a split, we need to find some impurity score (earlier, we discussed Gini Impurity, but XGBoost does not use that), and we need to put the entire data into memory to do that. This will be a big problem for huge datasets where we cannot put the entire dataset into memory.

  XGBoost uses an approximate algorithm based on candidate split points using percentiles of feature distribution to deal with this problem. To propose candidate split points, a weighted quantile sketch algorithm is used.

- Parallel learning

  Boosting is a sequential process, where we make the weak learn in sequence to make it better. But that does not mean we can build a tree using all the cores in parallel, and XGBoost does that. This makes the algorithm extremely faster.

- Sparsity aware

  XGBoost uses specialized algorithms that visit only non-missing entries in each node. This is called **sparsity awareness**. Sparsity in data may come from one-hot encoding, missing value, or zero entries.

- Cache aware

  XGBoost is also cache-aware and optimally uses cache to ensure parallelization.

- Out of core

  For data that does not fit into the main memory, XGBoost divides it into multiple blocks and store each block on disk. It also compresses the blocks for faster retrieval and decompresses on the fly (in an independent thread while disk reading).

Other than these, XGBoost has a lot of regularization parameters that you can tune. Creating an XGBoost model is rather easy with the XGBOOST package. But the key is to tune it. In this chapter, we will not go into the details of tuning as it is beyond the scope of this book. There is a lot to parameter tuning. One of the ideas I follow to tune an XGBoost model is discussed in the following blog, **https://towardsdatascience.com/fine-tuning-xgboost-in-python-like-a-boss-b4543ed8b1e** by Felix Revert. Also, you can follow *Jeff Heaton's* GitHub code to tune XGBoost models, **https://github.com/jeffheaton/jh-kaggle-util**. Other methods, such as Bayesian methods (using the *bayesian-optimization* package) to tune XGBoost models. Also, you can explore the *hyperopt package* to tune it.

We just scratched the surface of what XGBoost is all about; try to create multiple models and tune hyperparameters to appreciate how powerful XGBoost is. Now let us have a glance over neural networks for tabular data.

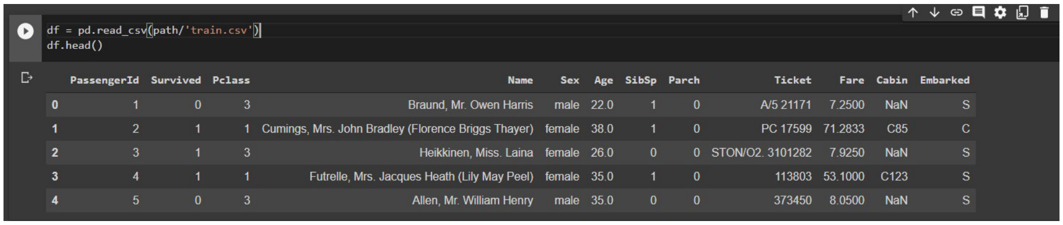
7.3.4 Neural networks for tabular data

Generally, we use traditional models such as SVM, tree-based, ensemble models (bagging, boosting models) to solve tabular data. But neural networks are also gaining popularity over tabular data. The key advantage of using neural networks even for tabular data is to remove the manual feature engineering. But neural networks tend to work better for tabular data with large sizes. For the smaller size, the traditional methods generally outperform neural networks. But even with that, the reader should know about neural networks models because they can be used to the ensemble with traditional models. Since they are a different type of model altogether from traditional models, they often ensemble well.

In this section, we will go through neural networks for tabular data using FastAI.

```
# ! pip install fastbook
from fastai.tabular.all import *
path = Path()
df = pd.read_csv(path/'train.csv')
df.head()
```

Here we will use the Titanic training dataset from Kaggle (**https://www.kaggle.com/c/titanic/data?select=train.csv**). The output of the preceding code:



```
df = pd.read_csv(path/'train.csv')
df.head()
```

```
   PassengerId  Survived  Pclass                                               Name     Sex   Age  SibSp  Parch            Ticket     Fare  Cabin  Embarked
0            1         0       3                            Braund, Mr. Owen Harris    male  22.0      1      0         A/5 21171   7.2500    NaN         S
1            2         1       1  Cumings, Mrs. John Bradley (Florence Briggs Thayer)  female  38.0      1      0          PC 17599  71.2833    C85         C
2            3         1       3                             Heikkinen, Miss. Laina  female  26.0      0      0  STON/O2. 3101282   7.9250    NaN         S
3            4         1       1       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1      0            113803  53.1000   C123         S
4            5         0       3                           Allen, Mr. William Henry    male  35.0      0      0            373450   8.0500    NaN         S
```

***Figure 7.40**: Titanic dataset from Kaggle*

Here the target variable is Survived. There are some categorical variables such as Sex, Cabit, Embarked, and so on, and some continuous variables. First, we need to process the categorical variables; FastAI tabular has a handy class called Categorify. Then, we need to fill in the missing values using FillMissing. Also, for neural networks, the input has to be scaled (remember gradients), so we will normalize using Normalize class from FastAI Tabular. But before all these, let's drop some of the columns; we will drop passenger id, name, and ticket columns. We can do some feature engineering using these columns, but since this is a dummy example to get you started with the neural networks tabular data, we will remove them.

The following is the code for the same:

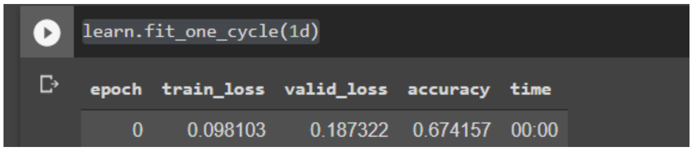
```
drop_names = ['PassengerId', 'Ticket', 'Name']
df.drop(drop_names, axis=1, inplace=True)

cat_names = ['Sex', 'Cabin', 'Embarked']
cont_names = ['Pclass', 'Age', 'Fare', 'SibSp']
dls = TabularDataLoaders.from_csv(path/'train.csv', path=path, y_
names="Survived",
    cat_names = cat_names,
    cont_names = cont_names,
    procs = [Categorify, FillMissing, Normalize])
```

The preceding code takes the data from a csv, then uses the Categorify, FillMissing, and Normalize on it.

Now, we can run the model on it:

```
learn = tabular_learner(dls, metrics=accuracy)
learn.fit_one_cycle(1)
```

The output of the preceding code:



```
learn.fit_one_cycle(1d)

epoch  train_loss  valid_loss  accuracy  time
    0    0.098103    0.187322  0.674157  00:00
```

***Figure 7.41**: Training on tabular data*

So, we get around 67% validation accuracy on the dataset. FastAI makes training models on tabular data much easier.

This concludes our discussion on different models. Many more models can be used for analysis and training on tabular data, but we discussed a few general ones that perform well on a wide range of datasets and can be used as an ensemble.

7.4 Idea for a web application

Even after the model creation, we need to think about how a user would benefit from our solution. Just showing housing prices may be useful, but some more user-specific features can make our product more useful. The following are a few feature ideas that interested readers can implement.

- Creating a Bangalore map and putting cost-specific coloring: You can use streamlit and altair to create a Bangalore map and show prediction prices within regions by coloring it (using latitude and longitude)
- Putting tooltip that shows average cost based on user input (1BHK, 2BHK, and so on): To make it more interactive, it may show the average cost of 1BHK, 2BHK, and so on when a user hovers his mouse over a particular location

Conclusion

With this, we conclude *chapter 7* on the data analytics use case. We created a model using the Bangalore housing prices dataset and learned how to clean, preprocess, and fine tune models. We also discussed several practical tips on handling many columns using pandas profiling and feature selector package; we talked about reducing the size by type conversion for faster exploration. We learned how to handle dirty data, and we investigated huge datasets and models.

Finally, we briefly touched upon pipelines using scikit-learn. Then, we dug deeper into the random forest to get few core concepts. Finally, we discussed some ideas for creating an application out of our model and adding user-specific features. The next chapter will learn about a computer vision use case and build custom image classifiers.

Points to remember

- Fine tuning a model involves a coarse to fine approach for an efficient search. First, do a random search on a broad set of parameters and drill down to minute details.
- Parallelization is an important aspect of machine learning.
- During missing value treatment, keep track of the places where value is missing as that can be an important feature.
- Even after a good performance on the created model, we need to think about the usability aspect of our model too.

MCQ

1. **Gini impurity is used for**
   a) Identifying purity among sets
   b) Identifying accuracy of tree models
   c) Identifying impurity among sets
   d) All of the above

2. **What does bootstrapping data mean?**
   a) Create N folds of the original dataset
   b) Create N folds of the original dataset with repetition
   c) Copy original dataset N times
   d) None of the above

3. **How does random forest handle missing values?**
   a) Using proximity matrix
   b) median
   c) mean

   d) Both a and c
   e) Both b and c

4. **Which library is preferred for large dataset visualization?**
   a) Vaex
   b) matplotlib
   c) numpy
   d) pandas
   e) All of the above

Answers to MCQ

1. c
2. b
3. a
4. a

Questions

1. Add MLFlow tracking to the use case, save params, artifacts and log the metric.
2. Create a web app using FASTAPI and deploy it in Heroku. Follow *Chapter 6: Taking Models into Production to deploy*.
3. Create the proximity matrix for the example discussed in this book's missing value treatment section for a random forest.
4. Fine tune the XGBOOST model in the use case.

Key Terms

- Cardinality
- Bootstrapping
- Proximity matrix
- Parallel prediction

CHAPTER 8
Building a Custom Image Classifier from Scratch

In this chapter, we will build a custom image classifier that can distinguish between healthy vs. unhealthy food. We will create a dataset from scratch, then train it using transfer learning. We will then deploy our application so others can access it, use it to upload an image containing food from the browser, and see if the food is healthy or not. But before that, we will learn about deep learning. Why has it become so effective? How does a neural network learn? What is 'deep' in deep neural networks? What is transfer learning? And much more. We will also go through another Computer Vision problem like a reverse search engine. There, we will create a solution like how Google reverse image search works. Many other use cases around computer vision, such as image segmentation, generation, etc. Still, we will not go into the details of that as Computer Vision is a vast topic and deserves its book(s). So, with that in mind, let us jump straight into neural networks.

Structure

In this chapter, we will cover the following topics:

- Introduction to deep learning
- Regularization
- Optimizers
- The learning rate

- Transfer learning
- Building a custom image classifier
- Reverse search engine
- How to approach a Computer Vision problem?

Objective

After studying this chapter, you should be able to:

- Create custom image classifiers using Computer Vision libraries
- A deeper understanding of how deep learning works
- Understand different deep learning jargons
- Practical tips for approaching a Computer Vision problem

8.1 Introduction to deep learning

We do not understand how a human mind works (yet), so in this book, we will not compare neural networks to how a human mind works. But we will take intuition from biological systems like we always do. We will try a problem solution approach to deep learning; we will figure out how deep learning solves problems so elegantly that it has become such a popular topic.

8.1.1 Artificial neural networks

Neural networks are a naïve representation of how our brains might work. The understanding goes like this; our brain has many interconnected neurons or a *network* of layers. The connections can be weak or strong. These connections are learned based on experience. Take an example (remember, it's a naïve explanation of how our brain learns, which is at its best an over-simplification), moving our hand to catch a ball coming from our right side and moving fast. So as input, we see the ball and some neural pathways carry this data through electrical signals, and a response is made based on the input. So, the pathways/connections that are responsible for the hand movement have strong connections. So, this simplified example may look as follows:

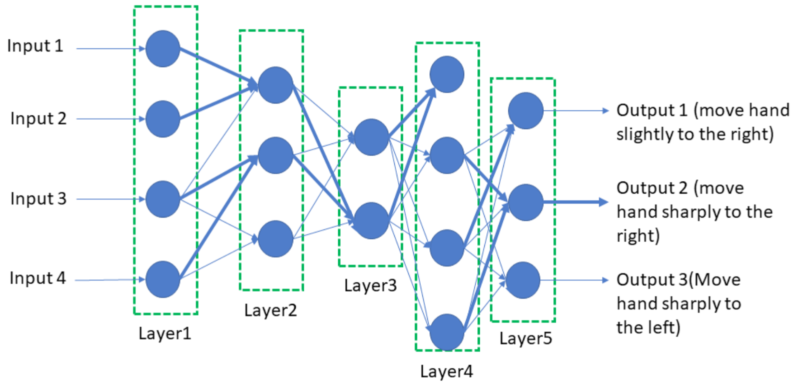

Input 1
Input 2
Input 3
Input 4
Layer1
Layer2
Layer3
Layer4
Layer5
Output 1 (move hand slightly to the right)
Output 2 (move hand sharply to the right)
Output 3(Move hand sharply to the left)

***Figure 8.1**: Simple neural network*

As we can see in the preceding image, there will be some inputs and connected neurons/nodes and some outputs. The inputs for our example can be our depth perception of the ball (giving how far back it is), our estimation of speed, the direction it is coming from (right or left); depending on this, our brain processes this data in a different set of neural connections. For example, let's say the first set of 4 neurons are connected to 3 more neurons, and there will be 12 connections between them, but it doesn't necessarily mean every other neuron is connected. So, I skipped a few connections. Now, the three neurons in the second set or *layer* may have different responsibilities; maybe the first neuron identifies the speed and right direction of the ball. These connections will be strong as in our example; the ball is coming from the right, so the neuron will light up or be *activated*. Now layer three can again focus on more abstract reasonings such as a connection of depth, direction, speed, how big the ball is, and so on. This we do not know because we don't understand these connections fully. The reason for that is complexity. I have shown five layers with 4, 3, 2, 4, 3 neurons in different layers sequentially. But our brain has 100 billion neurons approximately (recent findings may reduce the number to 86 billion, which is still a huge number!) and trillions of connections. But the idea/intuition still works—a set of interconnected neurons processing data. Now, the final layer is what we call an output layer, and it's trivially named. In our example, the final output layer may contain output like moving the arm slightly to the right, moving it sharply to the right, or moving it to the left. Since the ball is coming from the right and coming fast, the second output should come. In a statistical context, we may say the probability of the second output will be high, the first output to be low, and the third the lowest (we don't want to move our hand to the left if the ball is coming from right).

The first layer is called the input layer, which takes input and *forwards* it to the next layer. The last layer is called an *output layer*. This gives the output, and what output it gives is based on what we want. Whatever layers are in between are called *hidden layers*.

Now when I say the layer forwards the input to the next layer, what does that mean? These are analog electrical signals in our brain, but we will again simplify for machine learning. We will process the input by multiplying the inputs with some *weights*; this will signify the strong connections w.r.t some inputs and add some *bias* terms. We will pass it through an activation function to get the output/activations from that layer. We need an activation function to solve two things. First, without *activation function*, there is no bound in output values. With networks containing many layers, the output may get very large and uncontrollable if the input gets multiplied with weights multiple times. Activation functions keep the outputs in check. One example of an activation function is the sigmoid function, which is given by:

```
sigmoidX = 1/(1+np.exp(-X))
```

This looks like the following:

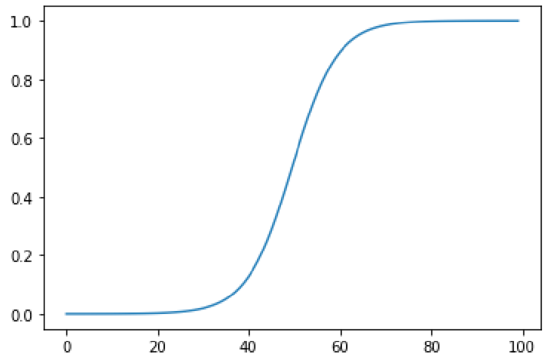
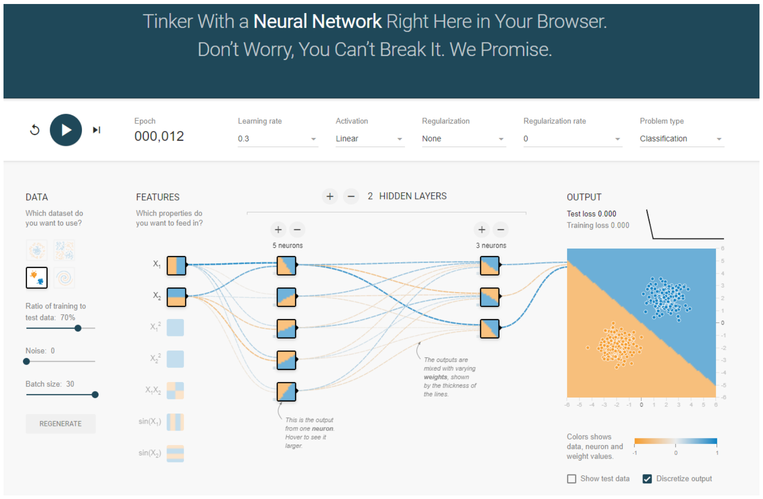


***Figure 8.2**: The sigmoid function*

As you can see, the function takes the function X and squishes it to values between 0-1. Hence it keeps the output in check. The second problem it solves is the non-linearity. This one is a bit subtle. You can think of an output of a layer without activation as a line (output = W.X + b). And even if we add multiple such layers (output = W2(W1. X1 + b1) + b2 = (W2W1).X1 +(W2b1 + b2) which is also linear) it is also linear. Hence a neural network without an activation function cannot learn nonlinear functions. And real-world examples are full of nonlinear problems. Let's take an example of linearly separable data and try fitting a neural network without activation; we will use **http://playground.tensorflow.org/**, by *Daniel Smilkov* and *Shan Carter*, to do this without a single line of code; it is a great resource for understanding and playing around with neural networks.


Tinker With a **Neural Network** Right Here in Your Browser.
Don't Worry, You Can't Break It. We Promise.

Epoch 000,012
Learning rate 0.3
Activation Linear
Regularization None
Regularization rate 0
Problem type Classification

DATA
Which dataset do you want to use?
Ratio of training to test data: 70%
Noise: 0
Batch size: 30
REGENERATE

FEATURES
Which properties do you want to feed in?

2 HIDDEN LAYERS
5 neurons
3 neurons

This is the output from one **neuron**. Hover to see it larger.

The outputs are mixed with varying **weights**, shown by the thickness of the lines.

OUTPUT
Test loss 0.000
Training loss 0.000

Colors shows data, neuron and weight values.

Show test data
Discretize output

Figure 8.3: Screenshots from Tensorflow playground by Daniel Smilkov and Shan Carter

Here we have taken a data example that is linearly separable. Our network contains two hidden layers. We use linear activation here, which simply means there is no non-linearity (data is just scaled linearly to keep output in check). This learns the data effortlessly. But suppose we try a dataset that is not linearly separable, like the following. In that case, i6 won't solve the problem with linear activation no matter how many hidden layers you add (keep in mind, we are only allowed to give X1, X2 as features; if we take other inputs, the network will learn without a nonlinear activation too.)

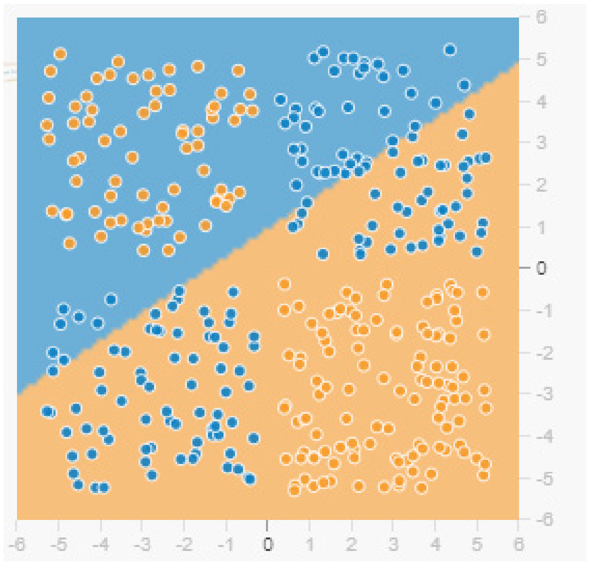


Figure 8.4: Linearly inseparable data with linear activation (screenshots from: **http://playground.tensorflow.org/** *)*

But if we try a **`sigmoid/relu/tanh`** activation, then the network successfully learns:

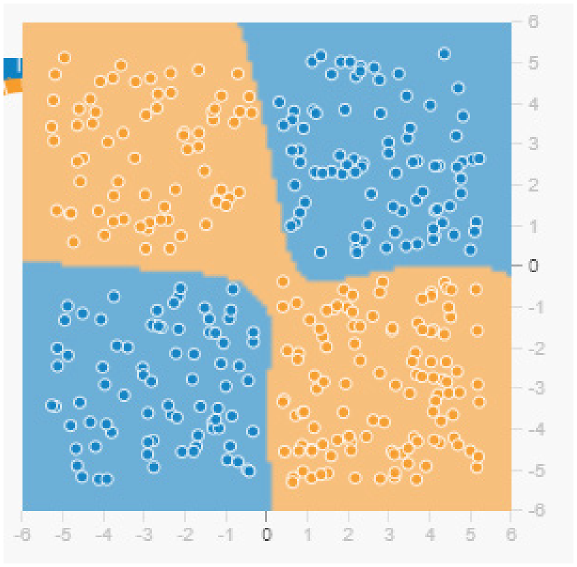


***Figure 8.5**: Linearly inseparable data with the sigmoid activation (screenshots from: **http://playground.tensorflow.org/**)*

I would suggest the reader go to this awesome site and play around with different networks. The site is user-friendly and builds up an intuition about neural networks.

There are many different activation functions, and sigmoid is practically not used unless it's the output layer—for example, ReLU, Leaky-ReLU, Tanh, and so on.

Next comes **bias**, why do we need a bias term? It is to provide another layer of flexibility because of the activation. Let's understand with the code; the following is the code for an output with a sigmoid activation without bias:

```
import numpy as np
import matplotlib.pyplot as plt
ws = [0.5, 1, 2]
X = np.linspace(-10, 10, 100)
for w in ws:
   sigX = 1/(1+np.exp(-w*X))
   plt.plot(X, sigX)
plt.xlabel('input')
plt.ylabel('output')
plt.legend([f'sig({w}X)' for w in ws])
plt.title("Sigmoid without bias")
```

You can see weights 0.5, 1, 2; I am plotting x between -10, 10 having 100 steps. Then, use matplotlib to plot the curve. The following is the output of the code:

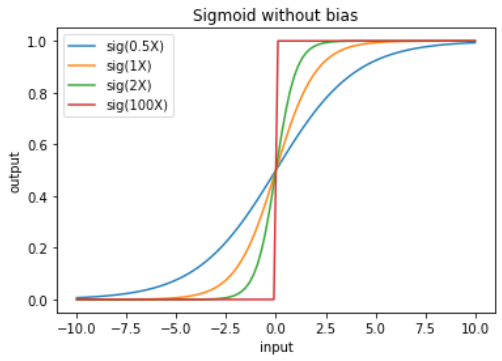


***Figure 8.6**: Activations without bias*

This graph will always have a value of 0.5 for x=0 because the exp(-w*X) term will always be zero, so increasing the value of w will just shrink the graph, and decreasing will expand it but never *translate* it. This is another degree of freedom bias gives you. Let's now try to plot it with bias; the following is the code for the same.

```
biases = np.ones(len(X))
wb = [-6, -4, 0, 4, 6]
for w in wb:
   sigX = 1/(1+np.exp(-(X+w*biases)))
   plt.plot(X, sigX)
plt.xlabel('input')
plt.ylabel('output')
plt.legend([f'bias weight = {w}' for w in wb])
plt.title("Sigmoid with bias")
```

The output to the preceding code:

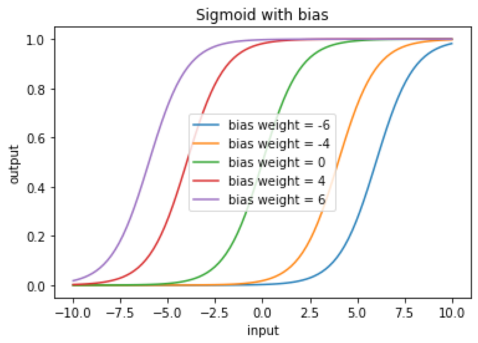


***Figure 8.7**: Activations with bias*

With the bias term, we can now translate the output right or left, and it gives us an additional degree of freedom.

8.1.2 Learning and gradient descent

We now have a simple neural network understanding that comes with a weight and bias and an activation function. Now, how do we adjust these weights and biases? To answer that, let's start with a simple one-layered neural network (no hidden layer). The following is the image of the same:

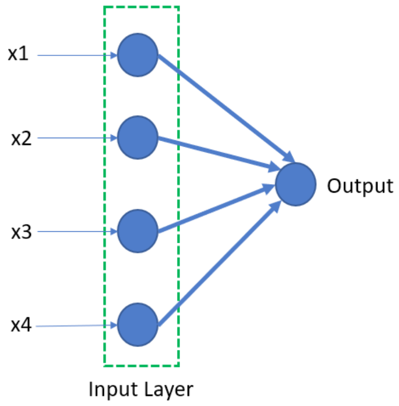


***Figure 8.8**: Simple perceptron*

Now what is the output here with respect to inputs, since the network has weights and biases (let us skip the activation for the moment):

$$y_{pred} = w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + w_4 * x_4 + w_0$$

Also, I have simplified the bias into a single term. Technically, there will be four biases as there are four neurons/nodes. Now we need to tweak our weights in such a way that becomes close to target. To do this, we have a lot of examples, something like this:

$$\begin{aligned} y_{target1} &= w_1 * x_{11} + w_2 * x_{21} + w_3 * x_{31} + w_4 * x_{41} + w_0 \\ y_{target2} &= w_1 * x_{12} + w_2 * x_{22} + w_3 * x_{32} + w_4 * x_{42} + w_0 \\ &\dots \\ y_{targetn} &= w_1 * x_{1n} + w_2 * x_{2n} + w_3 * x_{3n} + w_4 * x_{4n} + w_0 \end{aligned}$$

You may have taken high school maths; if we have N equations with N unknowns, we can solve it (we may get infinite solutions or get no solutions and won't go into the details). But the catch here is noise. Our data is not exactly pure. If it is like that, we need not require machine learning to solve it. It comes from the real world, and it is riddled with noise. So how can we do our best? We will define a cost function (like we did before with other machine learning models) and then try to minimize that. We can start with **Mean Squared Error** (**MSE**), which is given by:

$$MSE(X, h_w) = \frac{1}{n}\sum_{i=n}^{n}(w_i.x_i - y_i^{target})$$

There is a mathematical equation that gives this result directly, which is called the **Normal Equation**. This minimizes the mean squared error. Normal Equation is given by:

$$\widehat{W} = (X^T.X)^{-1}.X^T.y$$

Where minimizes the cost function, and y is the target variable.

This way, we can identify the weights, but the problem is that we need to calculate the inverse of , now imagine having a lot of features which is usually true for deep learning. An image with 64x64 pixels will have 64*64 features, and the complexity of calculating normal function for such a problem will be enormous. It is linear with no examples, so that is a plus point, but the data needs to fit inside the memory. Finally, neural networks are not that simple and single-layered, there may be multiple layers, and the complexity again rises. So normal equation may be used for small feature datasets with linear output, but it is impractical with complex networks.

This brings us to gradient descent; it is a *generic* algorithm that finds solutions to the preceding problem and is an iterative algorithm. The idea behind gradient descent is simple. Imagine being stranded in a mountain range at night with a single light source (maybe a hand torch). And you want to go down to your house. You know how far you are from your house too. You cannot see the whole terrain as it is night. What can you do? The analogy here is that the mountain range is the loss surface (basic values of loss function for different weights, it's not a surface traditionally as more than two weights the loss surface won't be 2D anymore). The house is our minimized loss function value for a particular weight which is our location in a mountain range. The best we could do here is go downwards, i.e., towards a negative slope since we know the house is somewhere where the area is flat, and it's at the bottom of the mountain.

Changing it into mathematics, we begin changing our weights in the direction of the downward slope or change in error surface or gradient of the cost function.

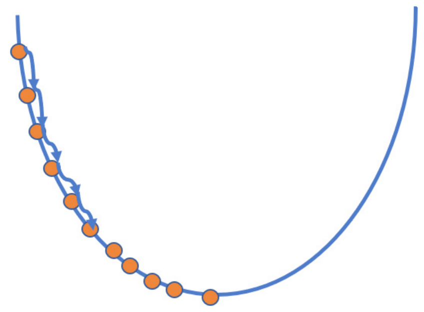


***Figure 8.9**: Gradient descent*

Here the blue curve is the cost function. We gradually go towards the minimum by taking small steps in the direction of the negative gradient. A learning rate parameter determines this small step. The more the learning rate, the faster it goes downhill, but we may miss a problem because we can’t take huge steps as it may make us miss the minimum. And if we take a ridiculously small step, then it takes a long time to reach home. So, it is a parameter that is needed to be optimized.

There is a lot of theory with gradient descent and local minimum and feature scaling, but we won’t go into the details as we want to keep the theory to a minimum.

Let us find the gradient of MSE with respect to weights W.

$$\frac{\partial}{\partial W} MSE(X, h_w) = \frac{\partial}{\partial W}\left(\frac{1}{n}\sum_{i=n}^{n}\left(w_i . x_i - y_i^{target}\right)^2\right)$$

$$= \frac{\partial}{\partial W} MSE(X, h_w) = \frac{2}{n}\sum_{i=1}^{n}\left(W^T . x_i - y_i^{target}\right) x_i$$

Now let us try to solve a linear problem with gradient descent with code.

We will take a simple linear problem and add some noise to it.

```
import numpy as np

import matplotlib.pyplot as plt


X = 3 * np.random.rand(50, 1)

y = 5 + 4 * X + np.random.rand(50, 1)



plt.scatter(X, y)

plt.title("Linear Problem")

plt.xlabel("x")

plt.ylabel("y")
```

The output of the preceding code:

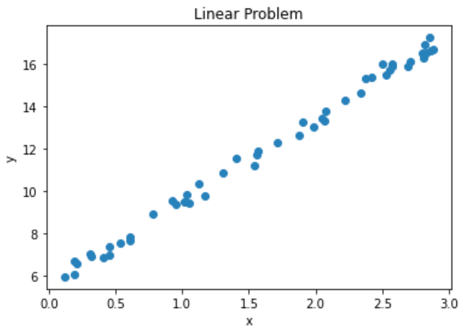


Figure 8.10: *Line with added noise*

Here, the actual weight is 4, and the bias is 5 as per the equation. Now, let us write the gradient descent part.

```
X_b = np.c_[np.ones((50, 1)), X]


eta = 0.3 # learning rate
n_iterations = 1000
m = 100
W= np.random.randn(2,1) # random initialization
for iteration in range(n_iterations):
 gradients = 2/m * X_b.T.dot(X_b.dot(W) - y)
 W= W- eta * gradients


W
```

With 1000 iterations and a learning rate of 0.3, the weight becomes 5.5, 3.95. which is close to what we want, that is, 5, 4.

Again, a problem with gradient descent is that we need to identify the gradient of the entire cost function with respect to each model parameter. So, it is also not practical to use if the training size is large and **`X_b.T.dot(X_b.dot(W))`** part of the gradient blows in computation with an increase in model parameters and an increase the size of the dataset. We will revisit these problems and solutions to them in the optimizers section of this chapter.

We still have not addressed the hidden layer part of a neural network. This is what makes a neural network deep. We can make gradient descent to optimize the layer weights and biases, but what about hidden layers? We know the output at a hidden layer, but we need a desired output that we don't have to write a cost function to minimize.

The way this problem is solved is called *backpropagation* of error. So, we feed the data from input to output with each consecutive layer (forward pass), just like when we make predictions and then measure the network's output error. We check the contribution of error from the previous layer, and we calculate this contribution using the weight of the hidden layer.

8.1.3 Convolutional neural networks

We learned about an artificial neural network, but when we identify an image, do we look at the entire image as a single input? No, we look into some particular features. For example, if we see an orange, we see its colors, its round shape and determine what it is. So, we apply some *preconceived notions* of what a round shape is and what colors are. This is how we filter our input image and identify these features. A convolutional neural network tries to do something like this.

The following is an image of how convolution works:

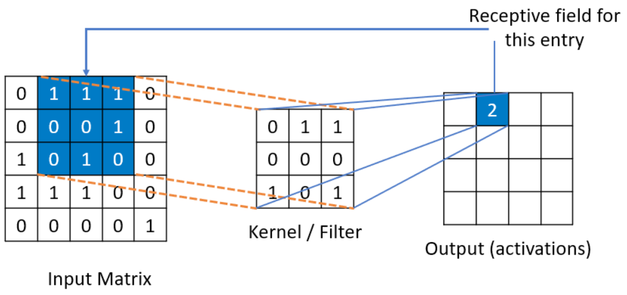


***Figure 8.11**: Applying convolutions*

Our kernel slides around the image and creates output by adding the dot product like shown in the preceding image. If the input size is *(n x n)* and the filter size is *(f x f)*, then the output size will be *(n-f +1 x n-f + 1)*. For example, for 5 x 5 image shown with 3 x 3 filters, the output is *5-3+1 = 4 x 4* output or *activations*.

Let us understand it by taking an example image; we will identify edges present in the image.

First, let us import some of the key packages we will use throughout this chapter.

```
%matplotlib inline
import matplotlib.pyplot as plt

from pathlib import Path
import uuid

import numpy as np
import torch
import torch.nn as nn
from torchvision import datasets, transforms
from torch.autograd import Variable
import torchvision.models as models
from PIL import Image
import torch.nn.functional as F
```

We will load an image of a dog first. The following is the code:

```
imsize = (256, 256)
```

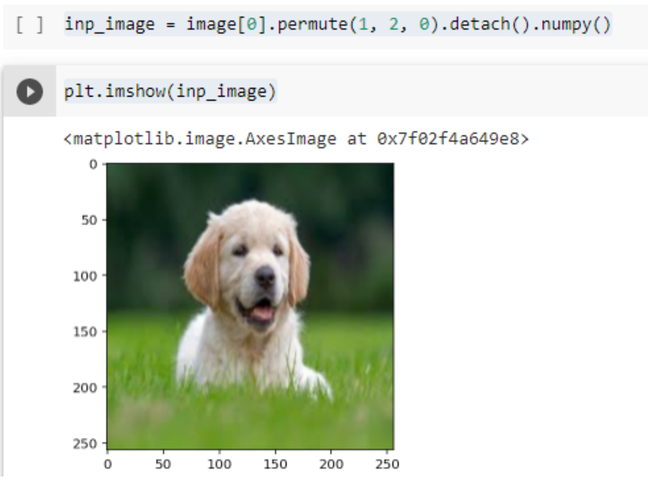
```
loader = transforms.Compose([transforms.Resize(imsize), transforms.
ToTensor()])

def image_loader(image_name):
    """load image, returns cuda tensor"""
    image = Image.open(image_name)
    image = loader(image).float()
    image = Variable(image, requires_grad=True)
    image = image.unsqueeze(0)  # adds another batch dimension
    return image

image = image_loader("dog.jpg")
```

Here we use a pillow to open an image and then use *transforms* to convert it into a fixed size tensor. Transforms are functions that are like pre-processing. Here we are resizing the image to 256, 256 images and then converting it into tensor in line 2.

Then we change the image to float in line 7 and change it to a variable. Then, in line 9, we add another dimension to create a batch. Let us visualize the input image.

```
inp_image = image[0].permute(1, 2, 0).detach().numpy()
plt.imshow(inp_image)
```

We are reading the image using the code, which gives a batch. We removed the batch using index 0 (as only one image in the batch), Then we permuted the dimensions (1, 2, 0) means 1st dimension (index – 0) changed to 2nd (index - 1) and second to 3rd (index - 2) and 3rd to 1st. Then, we changed the tensor to a NumPy array. The output to this code:



```
[ ] inp_image = image[0].permute(1, 2, 0).detach().numpy()

plt.imshow(inp_image)

<matplotlib.image.AxesImage at 0x7f02f4a649e8>
```

Figure 8.12: Input image for convolution

Let us create an edge detector *kernel or filter*. These are the matrices that an image will be convolved with, which means extracting filters from the image. The following is the code for the edge filter:

```
edge_detector = torch.tensor([
                    [0.  , -5/3, 1],
                    [-5/3, -5/3, 1],
                    [1., 1., 1.]
                ])
```

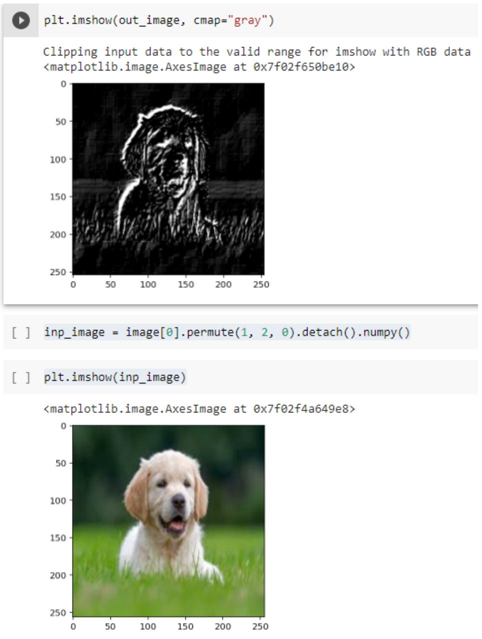
Here we can see the right and bottom numbers in the kernel are high, so an edge detector shows the right edges. Now let's first expand to 3 channels as our input image is 3 channel (RGB) image. We will then pass it to a convolutional layer using pytorch functional API.

```
edge_detector_batch = edge_detector.expand(1, 3, 3, 3)/6


out = F.conv2d(image, edge_detector_batch)
```

Now let us display the out image, the following is the code for the same:

```
out_image = out[0].expand(3, 254, 254).permute(1, 2, 0).detach().numpy()
plt.imshow(out_image, cmap="gray")
```

The output for the preceding code:



```
plt.imshow(out_image, cmap="gray")

Clipping input data to the valid range for imshow with RGB data
<matplotlib.image.AxesImage at 0x7f02f650be10>
```

```
[ ] inp_image = image[0].permute(1, 2, 0).detach().numpy()
```

```
[ ] plt.imshow(inp_image)

<matplotlib.image.AxesImage at 0x7f02f4a649e8>
```

Figure 8.13: Filter output

Here we can see the edges present in the dog image. This is how a convolutional filter works by working as a feature detector. Here we slide the kernel one by one pixel, so the output size is less than the input. But if we need to have the same output size, we can pad the input by zeros outside. So, the input becomes $n + 2p$ so that the output will be according to the formula, $n + 2p - f + 1$. If an image is 5 x 5 and the filter is 3 x 3, and we pad the input p = 1, then the input size becomes 7 x 7. That means the output will be 7 – 3 + 1 = 5, which is the same size as the input. This is called **padding**. Also, rather than moving 1 pixel, we can move the kernel, skipping some pixels. This is called strides. This reduces the output activation size. For example, if we have an 8 x 8 image with a filter size of 3 and with padding 1. That means we will have 8 x 8 with stride 1. But let us say we do stride 2 convolutions. Then the output will be 4 x 4 because we are skipping one pixel. So, the formula changes if we add padding and stride to the convolution.

$$\textit{Activation shape} = (n - f + 2p / s) + 1 \; x \; (n - f + 2p / s) + 1$$

Apart from convolution, we also add some non-linearity as discussed in the ANN example; we generally use ReLU or a variant of ReLU. Convolutional neural networks also have a maxpool layer. The following is an image explaining max pooling:

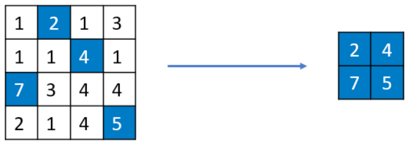


***Figure 8.14**: Max pooling*

This further reduces the size and also adds some invariance to the model.

Now let us see a convolutional neural network architecture.

There is a lot more to convolutional neural networks than what we discussed here. The key takeaway is to understand how shape changes with convolution. Now let us check one of the key innovations in CNN architecture called **Residual Networks** or **RESNETs**.

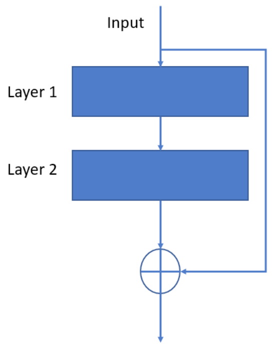


***Figure 8.15**: Key idea – identity connection*

The key idea behind a residual network is the identity connection. This means after some layer; we pass a copy of the input. This way network will also have a copy of the input, and it can learn even deeper networks. In our use case, we will use a RESNET model which has these identity connections. This idea comes from a ground-breaking paper named "*Deep Residual Learning for Image Recognition*" by *Kaiming He et al*. Interested readers may look into the whole architecture.

8.2 Regularization

Regularization is a method of reducing variance from the model. There are many types of regularizations used in deep learning. Regularization helps generalization without affecting the bias a lot. The following are a few that are popular:

1. **Data augmentation**

   In *Chapter 3, Data Acquisition and Cleaning*, we talked about augmentation in Computer Vision. This is also a way of creating more examples, and hence the model becomes more general.

2. **Dropout**

   Dropout is a simple way of reducing the variance by simply switching off some of the weights. This way, the network layers just after dropout don't have access to every detail; hence, the network is forced to learn more general features.

3. **Batch normalization**

   Batch normalization is scaling the output of a layer to have zero mean and variance as one and then giving two learnable parameters, alpha, and beta that control the scale and translation. This provides more freedom for the model to learn and makes the error surface less bumpy. This results in faster convergence and reduces generalization error.

4. **Weight decay**

   Weight decay is just an L2 regularization. When we create the loss function, we can add an L2 norm of weights (sum of weights squared) to penalize the weights. This provides a regularization effect and reduces variance. Another way of thinking about this L2 regularization is when we take the gradient of the loss with the L2 norm with respect to the weights. We are essentially subtracting the weight times a constant (hyperparameter called **Wd** that controls the **weight decay**) from the actual step. Hence, it is named weight decay as we are reducing the weights in sequence.

8.3 Optimizers

Our first neural network talked about gradient descent that gradually changes the weights to minimize the loss function. The gradient descent algorithm is an optimizer. It is the first optimizer we learned about, but it's not a good one; one issue with vanilla gradient descent is that to find each update in parameters, we must calculate the gradient of the error function with respect to all weights over the entire training set, which is computationally not feasible to do and slow also. Now, let us deep dive, into different types of optimizers.

8.3.1 Stochastic gradient descent

Stochastic gradient descent does a small modification to the gradient descent algorithm. Let us see the steps for the gradient descent algorithm:

1. Pick a random initial value for the parameters. Also, consider the objective function = taking all of the data to find the sum of squared residuals or errors.
2. Find the slope of the objective function/gradient with respect to each parameter/feature.
3. Update the parameters as: **`new_parameter = old_parameter - gradient * learning rate`**.
4. Repeat steps 1-2 until the gradient is an extremely small number.

SGD Algorithm – At step 1, take one data point from the training set. This reduces computation and makes the algorithm faster. Yes, the SGD makes the convergence bumpy (as we only have a single image to learn the step from), and rather than taking 1 data point, we can take a **mini batch** containing a small subset of data. This is also called **mini batch gradient descent**. Here's the pseudo code for SGD:

```
mini_batches = get_minibatch(X_train, y_train, batch_size)
Iterate over n_iter and follow below steps
   idxs = np.random.randint(0, len(minibatches))
   X_mb, y_mb = mini_batches[idxs] # shuffle
   grad = calculate_gradient(model, X_mb, y_mb)
UPDATE PARAMETERS as, parameters += alpha * gradient
```

Step 1, we get the minibatch (for SGD case **`batch_size`** is 1), then we iterate over **`n_iter`** times and get a minibatch from data (line – 4). Then identify the gradient over the minibatch (line – 5), and finally update the parameters.

8.3.2 Momentum

Even with the changes mentioned in the SGD algorithm, the convergence is still slow because if we start from a location (during initialization), that is far from the minimum; taking small steps takes a lot of time to converge.

To solve this, we can use momentum; the intuition is simple; rather than going at a fixed step size, we better go in the gradient direction. Don't take all the gradients and just take a percentage of it and rest from the previous gradient. This way, we just go faster in the previous gradient direction and make the convergence faster.

```
velocity= gamma * velocity+ (1 – gamma) * gradients

parameters += alpha * velocity
```

Gamma is usually 0.9, so we take 90% of the step in the direction of the previous output. This drastically improves the convergence.

8.3.3 RMSProp

Another interesting approach to improve the convergence is using an RMSProp optimizer. The intuition behind it is when we are going slow in the input direction or the gradients are small. We can go faster, but when we are making large steps or the gradients are very volatile, we have a problem of overstepping, so we should take smaller steps.

So, we calculate the exponentially weighted sum of the squared average of the gradients for each iteration and store it. Then divide the **`step_size`** by the squared root of the calculated exponentially weighted sum. This may sound complicated, but let us look at the pseudo code.

```
 weighted_gradients = gamma * weighted_gradients + (1 - gamma) * (grad
** 2)

 parameters += alpha * grad / (np.sqrt(weighted_gradients ) + eps)
```

The only difference between RMSProp and momentum is that in the former, there is a squared term. We divide, which normalizes the step size (if the step size is small, the squared gradient is small, we will take longer steps and vice versa). The latter, we use an exponentially weighted average of gradients as velocity and go in that direction.

8.3.4 Adam

Finally, Adam just takes combination of two and it is one of the most used and widely successful optimizers, the following is the pseudo code for Adam:

```
M= beta1 * M+ (1. - beta1) * gradients

R = beta2 * R + (1. - beta2) * gradients**2

parameters+= alpha * M / (np.sqrt(R) + eps)
```

Here in line one, we identify velocity or momentum, and in the next, the RMSProp squared gradients. Finally, by combining it, Adam also allows high learning rates.

We are not going into the code to implement these optimizers. These are implemented already in deep learning libraries; we just went through the pseudo code to understand the intuition. But the interested reader may add these tweaks to vanilla SGD to make these optimizers.

8.4 The learning rate

Learning rate is a crucial parameter and determines convergence. A high learning rate causes the model to diverge meaning rather than improving the error gradually. With a low learning rate, the optimizer algorithm becomes slow to converge. We will discuss a few techniques implemented in the FastAI library (using them in our use case).

8.4.1 Learning rate scheduling

What happens at the beginning of a model training phase? When we reach closer to the minimum, then we should take smaller steps. Optimizers also ensure this as when we reach the desired result, our gradients will become small (an error will be small as we are close). But still, it is not enough, and we can do better. So, it is essential to have a different learning rate at the start of the learning than at the end. In the end, we will require a lower learning rate.

You may think we should have a higher learning rate initially, but it is not quite true. If we take a large learning rate, it may throw us off and diverge as we are nowhere close to our minimum. The error surface is rough at the start. So, we should gradually start increasing the learning rate initially, and as discussed, begin reducing when we reach the end. This way of changing the learning rate during training is called **Learning Rate Scheduling,** and the way we discussed to increase in the start and then reduce in the end gradually is called **One Cycle Learning**.

On the other hand, when we start with small steps and go in the same direction, we better go faster in that direction to use a large momentum. And we will take small steps in the same direction when the steps are large—so small momentum. So as a complement to learning rate scheduling, momentum scheduling starts big, gradually goes down, and increases. This idea came from a paper called "*A Disciplined Approach to Neural Network Hyper-Parameters*" by *Leslie Smith*.

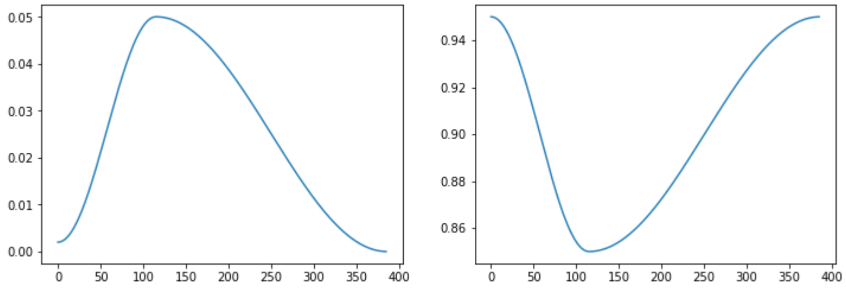


***Figure 8.16**: One cycle for learning rate (left), momentum (right)*

The preceding approach is used in FastAI when we train using **`fit_one_cycle`**.

8.5 Transfer learning

Transfer learning is one of the key innovations in machine learning. This is what changed the game. We can now create models with less data. Training takes less time. We discussed this in earlier chapters; now let us see how it works for a CNN.

For a CNN model, what we can do is, we can train the base model with some generic tasks. For example, RESNET18 that we used in the upcoming section to create a reverse image search is trained on the imagenet dataset, a classification dataset with 1000 classes. Then we will save the model parameters/weights. And when we try some other tasks, we can just remove the topmost layer (as our classes might be different). But why does it work?

For this, we will take the help of visualization. Now CNN visualization is a vast topic with many key ideas. Still, we will go through the results of one of the interesting papers, i.e., "*Visualizing and Understanding Convolutional Networks*" by *Matthew D Zeiler* and *Rob Fergus*. In this paper, the authors showed that initial layers of the neural networks learn very generic features like edges, lines, etc. With an increase in more layers, it will gradually learn more and more complex papers. I would recommend the reader to go through this paper in detail as it is a great paper to understand convolutional networks through visualization.

So, the idea for a different dataset (one on which the original model is not trained), we don't need to learn edges and lines. Because these features are so basic that they would be present for any dataset and we need not train on these. So, we *freeze* the layers of the base model. Freezing means not updating the weights. Because we need not change those basic feature detectors.

We will just train the layers that we add to the base model (after removing the final useless layer). But then, to create even better models, we can unfreeze the whole model and train. If we give a single learning rate to the whole model, it will be bad. Because earlier layers are pretty basic layers, they will become worse if we take large steps. Also, if we take a very small learning rate in the final layers, it will slow and take a lot of time. So, we divide the whole model into different sections carrying a set of layers, e.g., if our model has nine layers, we will divide it into three sections. 1,2,3 – section 1, 4,5,6 – section 2, 7,8,9 – section 3. Then we will assign different learning rates in these. For example, low learning rate in the first section as we don't want to change a lot in this section. And the highest learning rate in the final layer as we want to change a lot in the last. Different learning rates in different sections of a pretrained model during fine tuning are called discriminative learning rates.

8.6 Building a custom image classifier

Now we will train a custom image classifier from scratch. Like any other machine learning problem, we will start with data. We will use the bing-image-downloader package to download images directly from the Bing search. First, let us create a folder to store the images; the following is the code for the same:

```
from pathlib import Path

dataset_path = Path().resolve()/'dataset'/'food'

dataset_path.mkdir(exist_ok=True, parents=True)
```

The following is the code for downloading the dataset into the folder we just created:

```
! pip install bing-image-downloader

from bing_image_downloader import downloader

query_strings = ["healthy", "unhealthy"]

TOTAL_DOWNLOADS = 100

for query_string in query_strings:

  print(f"[ INFO ] Downloading images for query string {query_string}")

  downloader.download(query_string,

                      limit=TOTAL_DOWNLOADS,

                      output_dir=str(dataset_path),

                      adult_filter_off=True,

                      force_replace=False,

                      timeout=60)
```

We are using the '*healthy*' vs. '*unhealthy*' dataset. Currently, I am downloading 100 images each, but we can download more images for each class. This package directly downloads the images, but sometimes we need not download the actual images but just the links to the images. For example, if we want to download 1000 images, we want to download them in batches of 100 and save them. It may be due to download restrictions or space restrictions. We can override the download function in the library and just save the links. The following is the code for the same:

```
from bing_image_downloader.bing import *


class BingURLOnly(Bing):

      def __init__(self, query, limit, output_dir, adult, timeout,
filters=''):

        super().__init__(query, limit, output_dir, adult, timeout,
filters='')

      def download_image(self, link):
        self.download_count += 1

        # Get the image link
        try:
            path = urllib.parse.urlsplit(link).path
            filename = posixpath.basename(path).split('?')[0]
            file_type = filename.split(".")[-1]
            if file_type.lower() not in ["jpe", "jpeg", "jfif", "exif", 
"tiff", "gif", "bmp", "png", "webp", "jpg"]:
                file_type = "jpg"

            # Download the image
            print("[%] Listing Image url #{} from {}".format(self.
download_count, link))

            # save link
            with open(os.path.join(os.getcwd(), self.output_dir, 'links.
txt'), 'a') as wf:
                wf.write(str(link) + '\n')


            # self.save_image(link, "{}/{}/{}/".format(os.getcwd(), 
self.output_dir, self.query) + "Image_{}.{}".format(
            #     str(self.download_count), file_type))
            # print("[%] File Downloaded !\n")


        except Exception as e:
            self.download_count -= 1
            print("[!] Issue getting: {}\n[!] Error:: {}".format(link, 
e))
```

Upon quick look into the library, I saw the code to download the images inside lines 25-27 **`bing_image_downloader.bing.Bing`** class. I just inherited the class to use its methods and created **`BingURLOnly`** class that just changes lines 24-28 (originally to download the image directly into storage. And from lines 22-23, I am just getting the links and saving them inside the links.txt file.

Now I can call the run method inside the base class **Bing**.

```
bu = BingURLOnly(query_string, 10, "dataset", adult='off', timeout=60)
bu.run()
```

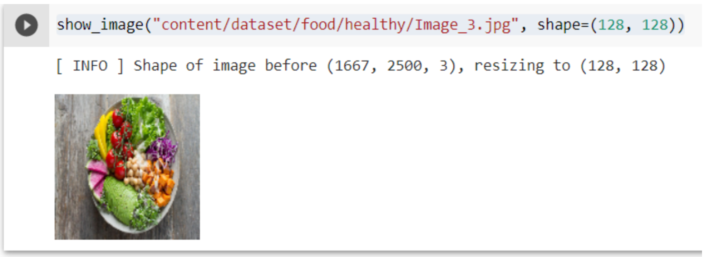
This just records the strings into the **link.txt** file. Which we can use later to download the images as we want. The reason for showing this is to show that it's an essential skill to quickly read into libraries and modify its functionality for our use. Real-life machine learning is like this. We may have problems that some libraries cannot solve directly, but it doesn't mean we need to start from scratch.

Now, as our dataset is ready, we can start exploring the data; let us write a helper function to visualize the image:

```
import cv2
from google.colab.patches import cv2_imshow
import uuid

def show_image(file_name, shape=(64, 64)):
  img = cv2.imread(file_name)
  resized_img = cv2.resize(img, shape)
  print(f"[ INFO ] Shape of image before {img.shape}, resizing to
{shape} \n")
  cv2_imshow(resized_img)

show_image("content/dataset/food/healthy/Image_3.jpg", shape=(128, 128))
```

I am using OpenCV to load the image using the **.imread** method and then resize it to 64, 64(by default) and use a patch function of google colab **cv2_imshow** to show the image. As **cv2.imshow** causes kernel error in Google Colab.

The following is the output of the code:



```
show_image("content/dataset/food/healthy/Image_3.jpg", shape=(128, 128))

[ INFO ] Shape of image before (1667, 2500, 3), resizing to (128, 128)
```

***Figure 8.17**: Looking at the data*

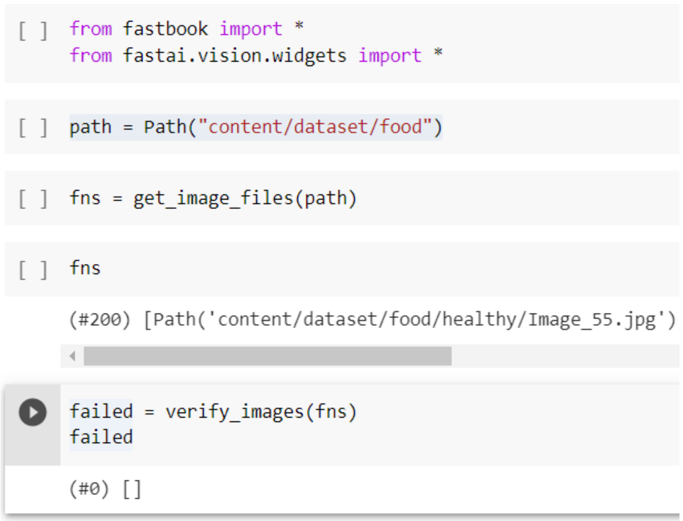
Now let us start training the model with the FastAI library. First, we need to load the data; the following steps are for the same:

First, we need to check if all the images are okay to download. We downloaded the images from the internet, and some images may be corrupt; the FastAI library has a nice function called **`verify_images`** that checks if it can load the image.

The following is the code for the same:

```
! pip install fastbook
from fastbook import *
from fastai.vision.widgets import *

path = Path("content/dataset/food")

fns = get_image_files(path)

failed = verify_images(fns)
failed
```

The output to the preceding code:



```
[ ] from fastbook import *
    from fastai.vision.widgets import *

[ ] path = Path("content/dataset/food")

[ ] fns = get_image_files(path)

[ ] fns

    (#200) [Path('content/dataset/food/healthy/Image_55.jpg')

    failed = verify_images(fns)
    failed

    (#0) []
```

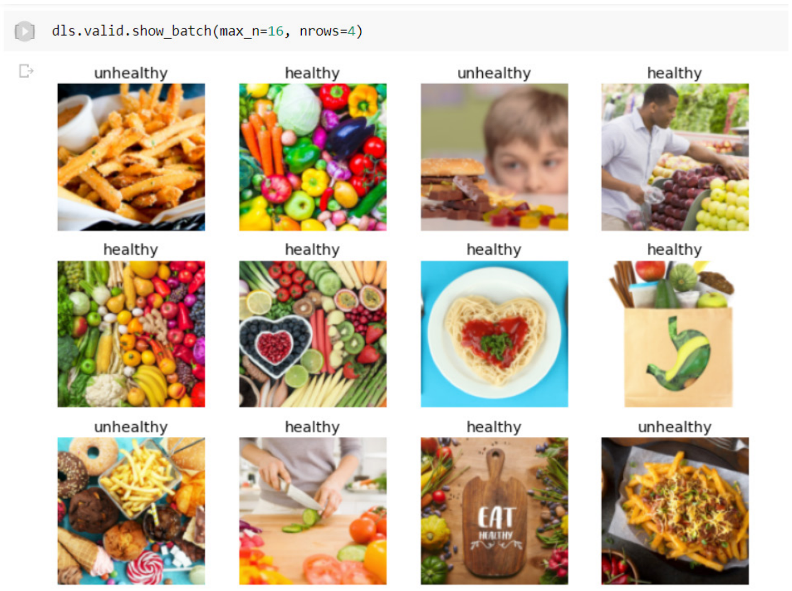
***Figure 8.18**: Verify the images*

Now in our data, it loaded all 200 images, and none of them are invalid. Now let us load the data using the datablock API of FastAI. FastAI has one of the richest ecosystems for data loaders, and I highly encourage going through the documentation to check those out. The library is very well documented, and each function has documentation that can be run as a notebook. The following is the code to load the images into datablock:

```
food = DataBlock(
    blocks=(ImageBlock, CategoryBlock),
    get_items=get_image_files,
    splitter=RandomSplitter(valid_pct=0.2, seed=42),
    get_y=parent_label,
    item_tfms=Resize(128))

dls = food.dataloaders(path)
dls.valid.show_batch(max_n=16, nrows=4)
```

Also, like discussed in the preceding regularization chapters, the FASTAI library has a great implementation of data augmentation. I am only using resize as a transformation, but the reader has many options to try, as discussed earlier.

The following is the output to the code; we are showing a batch of images:



```
dls.valid.show_batch(max_n=16, nrows=4)
```

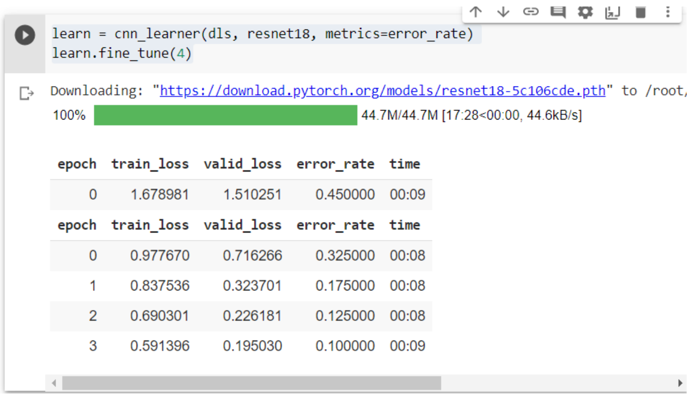
Figure 8.19: Data loading with FASTAI

Now let's apply more data augmentations and train the model.

```
food = food.new(
    item_tfms=RandomResizedCrop(224, min_scale=0.5),
    batch_tfms=aug_transforms())
dls = food.dataloaders(path)
```

```
learn = cnn_learner(dls, resnet18, metrics=error_rate)
learn.fine_tune(4)
```

Here, we use a resnet18 pretrained model with transfer learning and go through one cycle of learning; the following is the output of the code:



```
learn = cnn_learner(dls, resnet18, metrics=error_rate)
learn.fine_tune(4)
```

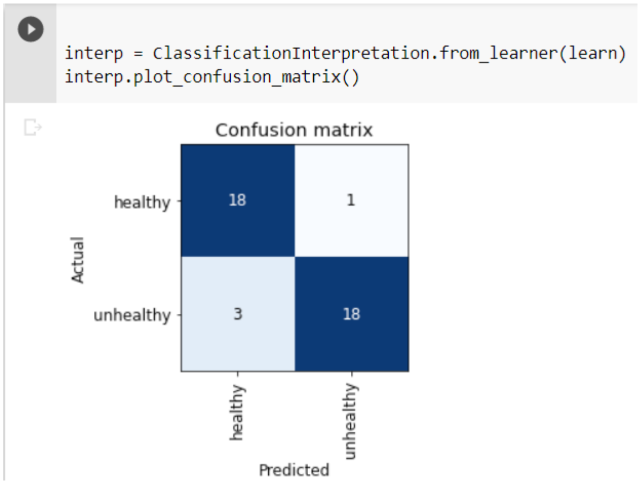
```
Downloading: "https://download.pytorch.org/models/resnet18-5c106cde.pth" to /root,
100% 44.7M/44.7M [17:28<00:00, 44.6kB/s]

epoch  train_loss  valid_loss  error_rate  time
    0    1.678981    1.510251    0.450000  00:09
epoch  train_loss  valid_loss  error_rate  time
    0    0.977670    0.716266    0.325000  00:08
    1    0.837536    0.323701    0.175000  00:08
    2    0.690301    0.226181    0.125000  00:08
    3    0.591396    0.195030    0.100000  00:09
```

***Figure 8.20**: Fine tuning RESNET18 Model*

With 4 iterations, we get around 0.10 error rate. Let us plot the classification matrix to check the output:

```
interp = ClassificationInterpretation.from_learner(learn)
interp.plot_confusion_matrix()
```

The output to the preceding code:



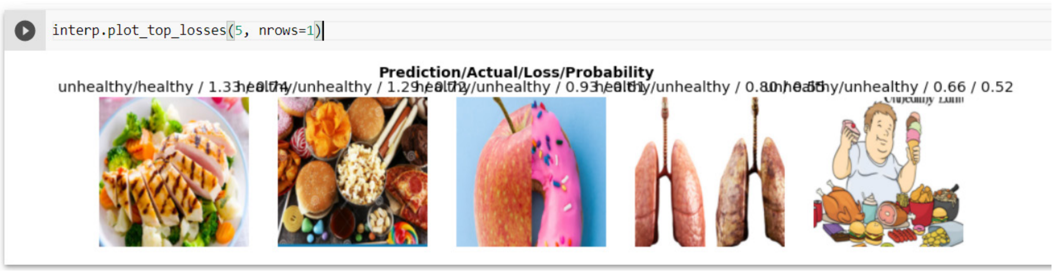
```
interp = ClassificationInterpretation.from_learner(learn)
interp.plot_confusion_matrix()
```

***Figure 8.21**: Confusion matrix*

We have four missed classifications out of 40 validation examples. That is 90% accuracy. Not bad, but we can do better if we collect more images, and also, as we have downloaded the images from the web, there might be some invalid images present. So, the first thing we do after we train a model is error analysis. With FastAI, we can do it pretty conveniently. Let us plot for top losses or where the model is most confused.

```
interp.plot_top_losses(5, nrows=1)
```

The output of the preceding code:



```
interp.plot_top_losses(5, nrows=1)
```

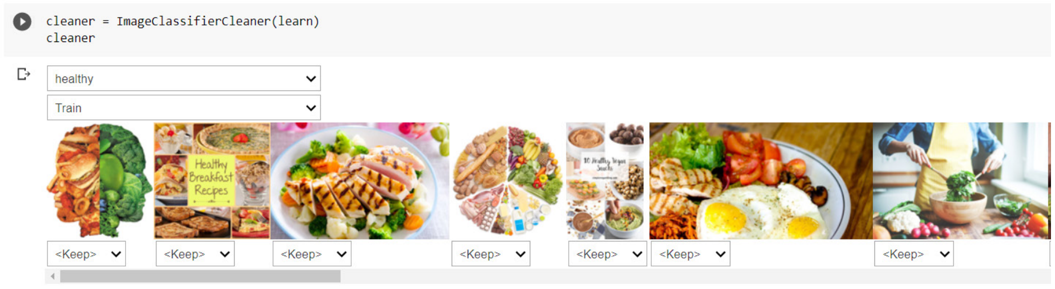
Prediction/Actual/Loss/Probability
unhealthy/healthy / 1.33 / 0.74
healthy/unhealthy / 1.29 / 0.72
healthy/unhealthy / 0.93 / 0.61
healthy/unhealthy / 0.80 / 0.55
unhealthy/unhealthy / 0.66 / 0.52

***Figure 8.22**: Most confusing predictions*

Here we can see three that the five images are invalid. So, we need to remove these. FastAI again comes up with many widgets that allow doing these things inside the notebook. The following is the code for the same:

```
cleaner = ImageClassifierCleaner(learn)
cleaner
```

This will not work in Google Colab (the preceding code will run fine, but you won't delete any of the images). Though widgets are not supported there, you can do this in any VM (refer to the appendix for a Google Cloud Platform VM). The following is the output of the code:



```
cleaner = ImageClassifierCleaner(learn)
cleaner
```

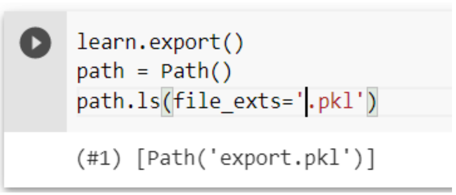
***Figure 8.23**: Cleaning images using ImageClassifierCleaner*

We can now clean the images; we will delete the irrelevant images or change class if required. Also, this shows the most confused or high loss images first, so it shows the most likely incorrect images first.

Now it's time to export the model so that we can have a web application. The following is the code for the same:

```
learn.export()
path = Path()
path.ls(file_exts='.pkl')
```

The output of the code:



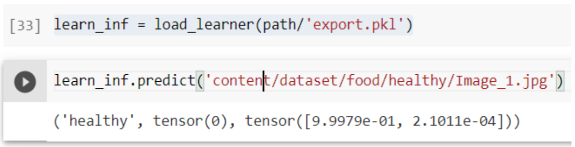
```
learn.export()
path = Path()
path.ls(file_exts='.pkl')

(#1) [Path('export.pkl')]
```

***Figure 8.24**: Serializing the model*

Now we can use this pickle file to load the model and start making our inference; the following is the code for the same:

```
learn_inf = load_learner(path/'export.pkl')
learn_inf.predict('content/dataset/food/healthy/Image_1.jpg')
```

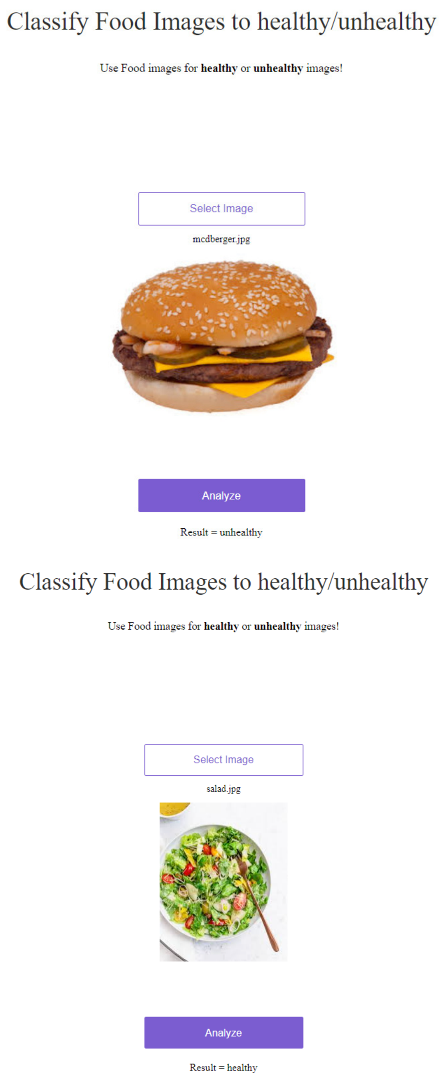
The output of the preceding code:



```
[33] learn_inf = load_learner(path/'export.pkl')

learn_inf.predict('content/dataset/food/healthy/Image_1.jpg')

('healthy', tensor(0), tensor([9.9979e-01, 2.1011e-04]))
```

***Figure 8.25**: Using the saved model.*

So, it successfully identified the healthy image. We can now build our web service as we did for the random forest example in the previous chapter. And I did it using render with a template already provided by the FastAI team. The following is how it looks.



Classify Food Images to healthy/unhealthy

Use Food images for **healthy** or **unhealthy** images!

mcdberger.jpg

Result = unhealthy

Classify Food Images to healthy/unhealthy

Use Food images for **healthy** or **unhealthy** images!

salad.jpg

Result = healthy

Figure 8.26(a) and (b): *Using the application created with FastAI and Render*

So, the model successfully classified images containing healthy vs. unhealthy food (first image I used a burger and second one a salad). I used to render (**https://render.com/**) to deploy the application. It is a simple application using the starlette framework, and we won't go into the details of the implementation as we have already deployed models and I would highly encourage the reader to fine tune the model more as discussed in previous sections of this chapter and with more images and then deploy the model using FastAPI.

8.7 Reverse search engine

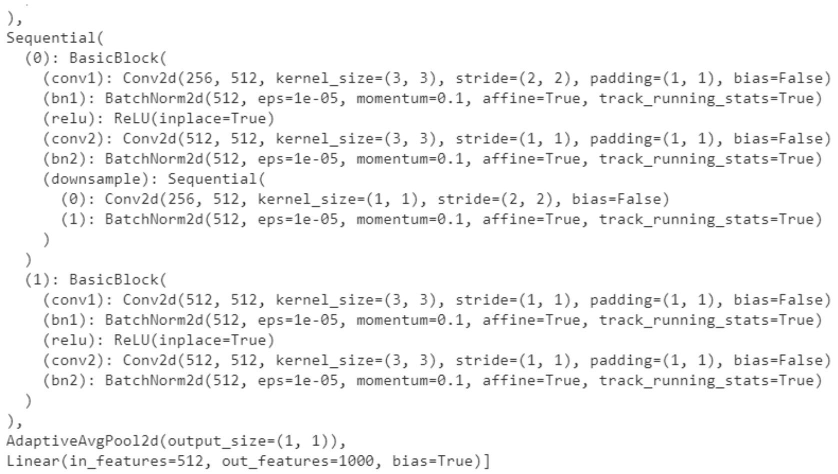
Apart from classification, there is a lot of different use cases with Computer Vision. Let us take one more example of a reverse search engine. The requirement is to build a solution that takes query images and provides an image like the query image. It is like Google Reverse Image search.

The key idea here is to create some sort of embedding for the images, taking our image from pixel space to feature space. Feature space is a set of numbers that *represents* the image features. Once we have a feature representation of the image, we can then run it over our dataset of images and store the features as NumPy arrays. Then when a query image comes, we will first extract the feature and then take the image with the lowest distance in feature space. Let us begin.

When we train a CNN, the layers are learning some features of the image. Initially crude features like edges, color, and so on. In the later layers, it combines those features into more complicated features. The final layer is generally useless because we are not interested in the classes. We are interested in the features. So, we will remove the last layer and take the features just before that. Let us use a pre-trained model to do so. The following is the code:

```
import torchvision.models as models


resnet18 = models.resnet18(pretrained=True)

list(resnet18.children())
```

Here we are loading a resnet18 model with pretrained weights using pytorch model zoo. This will download a resnet18 pretrained model. The output of this code shows the layers:



```
),
Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
),
AdaptiveAvgPool2d(output_size=(1, 1)),
Linear(in_features=512, out_features=1000, bias=True)]
```

***Figure 8.27**: Final Layers of RESNET18 Model*

As you can see, the last layer contains a linear layer that outputs 1000 values. This is because the model is trained on the **ImageNet** dataset (**http://www.image-net.org/**), which has 1000 classes. And we need to remove this layer as we are interested in the 512-dimensional output before this layer. And we can use other models too, **pytorch** has a huge amount of pre-trained models. I would encourage the reader to play around with different models. Now let's write our **FeatureExtractor** class that will extract this final layer of features. The following is the code for the same:

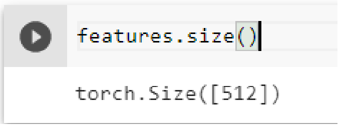
```
class FeatureExtractor(nn.Module):
    def __init__(self, pretrained_model):
        super(FeatureExtractor, self).__init__()
        self.features = nn.Sequential(*list(pretrained_model.children())
[:-1])


    def forward(self, x):
        x = self.features(x)
        o = x.data.squeeze()
        return o


fe = FeatureExtractor(resnet18)
```

The preceding code is simple; we are taking all the children of the pre-trained model and removing the list layer (which is the linear layer we don't want) and since we are subclassing **nn.Module** we can override the forward method that will take our image tensor and give us the features (squeezing the image as a batch dimension we don't want).

Now let us load an image and check the shape; it should be 512 dimensional.

```
image = image_loader("dog.jpg")
features = fe(image)
features.size()
```

We use the **image_loader** (page 13, line 4 of the code snippet) function that we created in the understanding convolution section to load the image to a tensor. The output of the preceding code:



```
features.size()
```

```
torch.Size([512])
```

Figure 8.28: Feature shape

Now, let us get some data using **`bing-image-downloader`** that we used in custom image classifier use case. The following is the code:

```
import random
from pathlib import Path
from bing_image_downloader import downloader
query_strings = ["dog", "cat", "boat", "human", "food", "fish", "house"]
image_dir = Path('images')
for query_string in query_strings:
  print(f"[ INFO ] Downloading images for query string {query_string}")
  downloader.download(query_string,
                      limit=random.choice([1, 2, 3]),
                      output_dir=str(image_dir),
                      adult_filter_off=True,
                      force_replace=False,
                      timeout=60)
```

Here we are randomly generating 1-3 images for classes mentioned in **`query_strings`**. The reader can create more images to make a larger database of images.

Then we will just rename these images as some image names have the same name, and we don't want that as we will query the image. We need unique names for each image to create a feature file for each image. Here is the code to rename.

```
for image_path in image_dir.glob('**/*.jpg'):
    name = f'{str(uuid.uuid4())}.jpg'
    print(image_path)
    image_path.rename(image_dir/name)
```

Now, let us create features and save it as **`numpy`** arrays, the following is the code:

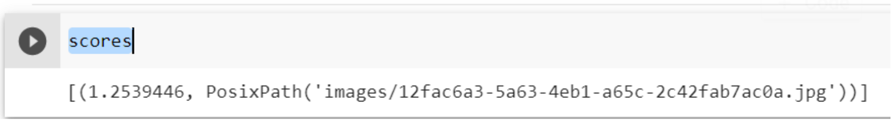
```
db_features = []
feature_dir = Path('features')
feature_dir.mkdir(exist_ok=True, parents=True)
for image_path in image_dir.glob('*.jpg'):
    image = image_loader(image_path)
    features = fe(image).detach().numpy()
    db_features.append(features)
    feature_path = feature_dir / f'{image_path.stem}.npy'
    np.save(feature_path, features)
db_features = np.array(db_features)
```

Here we go through each image in the image_dir and get features using our fe(FeatureExtractor) object and then save the **numpy** array inside the feature path.

Now, as we have the features for all the images, let us query a different image and check how it identifies the closest image.

```
# loading query image
query_image_path = 'dog.jpg'
img = image_loader(query_image_path)
query_image = fe(img).detach().numpy()

# getting distances from each of these feature in db
dists = np.linalg.norm(db_features-query_image, axis=1)
ids = np.argsort(dists)[:1] # here we are taking the top 1 result, for
top n results you can change 1 to n

# closest image
scores = [(dists[id], image_paths[id]) for id in ids]
scores
```

The output of the preceding code:



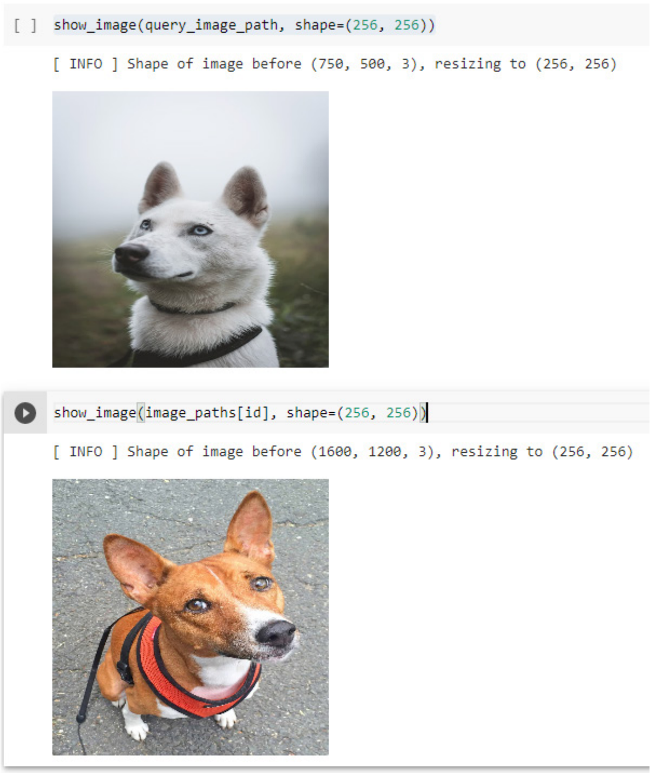
```
scores
[(1.2539446, PosixPath('images/12fac6a3-5a63-4eb1-a65c-2c42fab7ac0a.jpg'))]
```

Figure 8.29: Closest image

Now let us display the query image and the output image (from reverse search).

```
import cv2
from google.colab.patches import cv2_imshow
import uuid

def show_image(file_name, shape=(64, 64)):
  img = cv2.imread(file_name)
  resized_img = cv2.resize(img, shape)
  print(f"[ INFO ] Shape of image before {img.shape}, resizing to
{shape} \n")
  cv2_imshow(resized_img)


show_image(query_image_path, shape=(256, 256))


show_image(image_paths[id], shape=(256, 256))
```

The output of the preceding code:



```
[ ] show_image(query_image_path, shape=(256, 256))

[ INFO ] Shape of image before (750, 500, 3), resizing to (256, 256)
```

```
show_image(image_paths[id], shape=(256, 256))

[ INFO ] Shape of image before (1600, 1200, 3), resizing to (256, 256)
```

***Figure 8.30**: Reverse search engine results*

So, the model understands correctly that the image is of a dog and gives a dog image as a similar image. Now, we can deploy this solution using the techniques discussed in the first use case. And as an exercise, the reader may choose to do that.

8.8 How to approach a Computer Vision problem?

Computer Vision is such a vast area that we still need to work on more use cases after visiting these examples. And I strongly encourage the reader to go through these problems. But the following is the process you can follow when you approach a Computer Vision problem:

1. **Data**

   First is data; see if you can get cleaned images for your problem (for example, Kaggle or Google Datasets), which will reduce the acquisition time. But key skills to have are scrapping, using API to get data—database skills. And most importantly, going through library code and documentation.

2. **Architecture**

   Then comes architecture; if you can use transfer learning, use it. It is always better to start with transfer learning. If you can't use transfer learning (your dataset is completely different), choose well-known architectures like RESNETs.

3. **Loss function**

   Then decide on a loss function. Mostly we will use MSE, cross entropy, and so on. But for tasks like image generation and image label maker, we may need different loss functions. And some business problems may require their loss function.

4. **Error Analysis**

   Error analysis is a must; check which images the model is most confused about and see invalid images. See which class the model is most confused about.

5. **Evaluation**

   With the preceding three ingredients, you can successfully make a model and fit it. But you would require an evaluation metric to judge your success. Success metrics are often tricky and difficult to think about. It generally takes practice.

   Then to fine tune a model, you can start with a fine tuning training error. Low training error means high bias so that you can do the following:

   - Increase epoch / more cycles
   - Use a different architecture
   - More images are always good

   Then once you are okay with the training error or low bias scenario now, we can start focusing on the high variance. We can,

   - Use more regularization – more augmentation, more dropout, and increase dataset size
   - Different Architecture also helps

Finally, go through few papers on Computer Vision and try implementing it without using a library that already implemented it. You can use your favorite library for deep learning (for example, TensorFlow or PyTorch). This gives a lot of insight into the approach defined in the paper and quickly improves your coding for machine learning.

Conclusion

With this, we conclude *chapter 8* of the book. In this chapter, we learned how deep learning works. We came across many jargons such as layers, activations, parameters, etc. We learned convolutional neural networks and explored RESNETs. We then learned various regularization methods, optimizers, and learning rate scheduling. We also learned the importance of transfer learning and learned about discriminative learning rates. Finally, we built two use cases in Computer Vision—one a custom image classifier using web images that classify healthy vs. unhealthy images and deploy them. Then we also built a reverse image search engine; although it's a primitive solution, we learn about how the solution works. In the next chapter, we will explore a natural language processing use case.

Points to remember

The following are a few points to remember:

- The understanding shape of input and intermediate outputs in a CNN is key
- Residual networks allow deeper networks because we pass a copy of the input
- Learning rate scheduling is needed because different layers should be trained with different step sizes
- Convolution layers are feature extractors

MCQ

1. **A _ is a group of neurons/**
    a. Layer
    b. Input
    c. Output
    d. Weight

2. **For a convolution filter, input size = 5, padding = 1, filter size = 3. The output will be.**
    a. 3
    b. 4
    c. 5
    d. Invalid input
3. **Dropout is a type of _**
    a. Regularization
    b. Learning
    c. layer
    d. Both a and c
    e. Both b and c
4. **Weight decay is another way of thinking _ Regularization?**
    a. Data Augmentation
    b. L2 Regularization
    c. Dropout
    d. L1 Regularization

Answers to MCQ

1. a
2. c
3. d
4. b

Questions

1. Create a website for the reverse search engine example; try scaling the system to 10000 images. Use nmslib to scale.
2. Create a segmentation model.
3. Visualize the different layers using hooks in pytorch.

4. Fine tune the custom classifier model by unfreezing the full network and cleaning the images.

Key Terms

- Transfer learning
- Residual connections
- Normal equation
- Learning rate scheduling
- Discriminative learning rates
- One cycle learning
- Exponentially weighted moving average

Chapter 9
Building a News Summarization App Using Transformers

In this chapter, we will build a news summarization application using transformers. But before that, we will go through a bit of theory in transformers. We will learn the internal workings of transformers and some key innovations that make transformers one of the turning points in Natural Language Processing history. But before that, we will go through some data acquisition and cleaning techniques. We will learn some web scrapping using *Selenium* and *BeautifulSoup*. Then we will learn about some feature extraction techniques that are not relevant to our use case but will nevertheless learn about it. We will start our summarization use case with unsupervised techniques. Then, we will proceed with pretrained transformer models such as BART for summarization. Then, we will start fine tuning a transformer model, and for that, we will learn about annotation tools. Let us start building a news summarization application.

Structure

In this chapter, we will cover the following topics:

- Data collection and cleaning
- Word embeddings
- A really quick guide to RNNs
- Transformers

- Unsupervised summarization methods
- Fine tuning a model for summarization

Objective

After studying this chapter, you should be able to:

- Create summarization applications
- A deeper understanding of how transfer learning works
- Exploding and vanishing gradients
- Transfer learning and fine tuning in Natural Language Processing

9.1 Data collection and cleaning

We will start with data first, as with any machine learning problem. In our final application, we will use a Kaggle dataset available for news summarization. Still, a lot of Natural Language Processing is very domain-specific, so it is always great to have data skills. Preparing data for a machine learning project is the most time-consuming part, and especially if we do any supervised learning, annotation is the most manual part. Hence, we will use the Kaggle Dataset (which is already annotated) for our final model. But we will go through data acquisition and cleaning in this chapter as this is an essential skill for any machine learning practitioner. So, let us start with acquiring data using web scrapping.

9.1.1 Web scrapping for data acquisition

The internet is full of data, but it is often unstructured and messy. So, we need to clean it. Unless there is a well-documented API, direct data acquisition from the web is kind of a pain. But the pain part is exactly not acquisition rather maintenance. To acquire the data from a website, we need familiarity with web scraping tools such as Selenium, webdriver, beautifulsoup4, and requests. And these packages are not that difficult to use. Selenium is a huge library used for UI automation, but the usage is not that complicated for our purpose. But the problem with the web scrapping approach is maintenance. Let us understand web scrapping and its limitations with an example.

Let us try to get the data from **https://inshorts.com/en/read** to create a summarization dataset. We will build a web scrapper application using beautifulsoup and Selenium. We will also use webdriver-manager to manage the installations for webdriver. We are not going into the details of how webdriver works or how beautifulsoup works. The interested reader may investigate the documentation of each. We will use our local computer for the following scrapping code as we need to run a webdriver instance to scrape a website. Now, let us investigate the code:

First, let us import all the necessary packages:

```
# ! pip install selenium
# ! pip install webdriver-manager
from typing import List
import uuid
import pandas as pd
import bs4
from bs4 import BeautifulSoup
from selenium import webdriver
from selenium.common.exceptions import NoSuchElementException
from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.chrome.options import Options as ChromeOptions
```

The reason for using Selenium is it works for dynamic websites also. The websites where the html is generated with JavaScript. For this project, we could have only used the requests package. But I also want to automate the process of clicking different pages on the Inshorts website.

Now, let us start writing a class that will do our web scrapping.

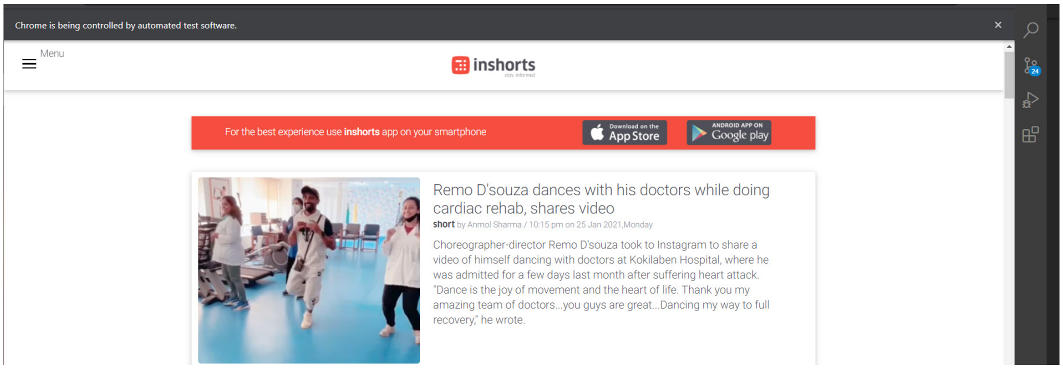
```
class Scrapper:
    def __init__(self, url: str) -> None:
        options = ChromeOptions()
        options.add_argument('--start-maximized')
        options.add_argument('--ignore-ssl-errors=yes')
        options.add_argument('--ignore-certificate-errors')
        self.driver = webdriver.Chrome(ChromeDriverManager().install(),
                                                options=options)
        self.driver.get(url)
        self.driver.implicitly_wait(10)
```

The options are used to ignore some SSL warnings and errors that arise when we try to open https sites. And we will use Chrome as our webdriver, so we used **`ChromeDriverManager().install()`** to install the webdriver for this. The constructer takes the URL as input. We also added an implicit wait of 10 seconds. This is the time Selenium will wait for locating a web element.

Now you can run this much code and open a web browser with the URL opened. Now let us check how the website looks and get some locators to select from. The following is the code to run the code in an ipython window in the same directory (I named the script as **`main.py`**).

```
from main import *
crawler = Scrapper("https://inshorts.com/en/read")
```

The output of the preceding code:



Chrome is being controlled by automated test software.
Menu
inshorts
Download on the App Store
ANDROID APP ON Google play

For the best experience use **inshorts** app on your smartphone

Remo D'souza dances with his doctors while doing cardiac rehab, shares video

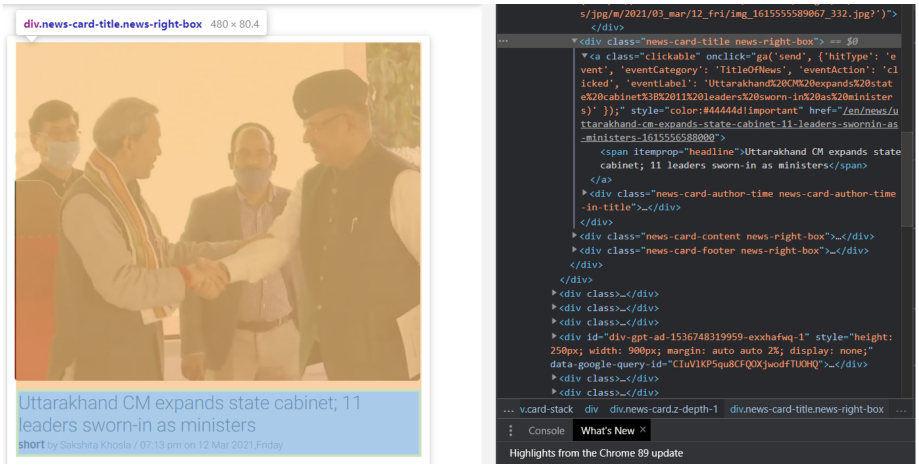
short by Anmol Sharma / 10:15 pm on 25 Jan 2021,Monday

Choreographer-director Remo D'souza took to Instagram to share a video of himself dancing with doctors at Kokilaben Hospital, where he was admitted for a few days last month after suffering heart attack. "Dance is the joy of movement and the heart of life. Thank you my amazing team of doctors...you guys are great...Dancing my way to full recovery," he wrote.

Figure 9.1*: Inshorts landing page*

We have a menu button containing all the different news categories: All News, India, Business, Sports, World, Politics, Technology, and so on. We can get these topics and build our *xpaths* to click on these categories one by one and get the articles and headlines from the page. For our example, we can imagine the article as the main text and the headline as the summary of the article. We need not annotate (remember to use as much trickery as possible to avoid or ease out annotation).

We can identify the locators upon checking the Document Object Model (DOM) in Chrome Developer tools. Locators are something that we will use to get the data.



Uttarakhand CM expands state cabinet; 11 leaders sworn-in as ministers

short by Sakshita Khosla / 07:13 pm on 12 Mar 2021,Friday

```
s/jpg/m/2021/03_mar/12_fri/img_1615555589067_332.jpg?')">
  </div>
▼<div class="news-card-title news-right-box"> == $0
  ▼<a class="clickable" onclick="ga('send', {'hitType': 'e
  vent', 'eventCategory': 'TitleOfNews', 'eventAction': 'cl
  icked', 'eventLabel': 'Uttarakhand%20CM%20expands%20stat
  e%20cabinet%3B%2011%20leaders%20sworn-in%20as%20minister
  s)' });" style="color:#44444d!important" href="/en/news/u
  ttarakhand-cm-expands-state-cabinet-11-leaders-swornin-as
  -ministers-1615556588000">
      <span itemprop="headline">Uttarakhand CM expands state
      cabinet; 11 leaders sworn-in as ministers</span>
    </a>
  ▶<div class="news-card-author-time news-card-author-time
  -in-title">…</div>
  </div>
▶<div class="news-card-content news-right-box">…</div>
▶<div class="news-card-footer news-right-box">…</div>
</div>
</div>
▶<div class>…</div>
▶<div class>…</div>
▶<div class>…</div>
▶<div id="div-gpt-ad-1536748319959-exxhafwq-1" style="height:
250px; width: 900px; margin: auto auto 2%; display: none;"
data-google-query-id="CIuVlKP5qu8CFQOXjwodfTUOHQ">…</div>
▶<div class>…</div>
▶<div class>…</div>
```

... v.card-stack div div.news-card.z-depth-1 div.news-card-title.news-right-box ...
Console
What's New
Highlights from the Chrome 89 update

Figure 9.2*: Getting the locators*

As you can see, the locator for the news headline is on the right. We can use the element type (div) and its class name (news-card-title news-right-box) to get this locator. Did you identify the limitations of our method? Yes, it is always possible that the developers of the Inshorts application decide that they need to change the css classes or the look and feel of the website. So, our code may not work in the future. This is what I mean by high maintenance in using web scrapping. Hence, it is always advised to use well-documented APIs. But for now, let us go ahead with what we have.

First, let us get the soup using beautifulsoup4; the following is the code:

```
    def get_soup(self):

        return BeautifulSoup(self.driver.page_source, 'html.parser')
```

This uses html parser to create a beautifulsoup object (from page source using **`driver.page_source`**) that will help us extract all the data from web elements.

Now, let us write the method that will give us the headlines:

```
    def get_all_headlines(self, soup: BeautifulSoup) -> bs4.element.
ResultSet:

        return soup.find_all('div', class_=["news-card-title news-right-
box"])
```

We use beautifulsoup4 to parse the soup that we will get from the page source using Selenium; we use the **`find_all`** method to get all the news headline divs from the page. Then let us write another method to get the headline string out of these soup.

```
    def get_headline(self, headline_soup: BeautifulSoup) -> str:

        return headline_soup.find('span', attrs={"itemprop":
"headline"}).string
```

The actual text is present inside the anchor tag, inside the span with a **`itemprop`** attribute. In **`Beautifulsoup`**, you can use **`attrs`** to narrow down the search further with attributes of that specific element.

Similarly, for the article, the two methods to get the string are given as follows:

```
     def get_article(self, article_soup: BeautifulSoup) -> str:

        return article_soup.find('div', attrs={"itemprop":
"articleBody"}).string


    def get_all_articles(self, soup: BeautifulSoup) -> bs4.element.
ResultSet:

        return soup.find_all('div', class_=["news-card-content news-
right-box"])
```

The preceding code is self-explanatory, and like what we did for headline.

Now, let us write our final function to get the data from the page.

```
    def get_data_from_page(self, category: str) -> dict:
        soup = self.get_soup()
        headlines, articles = self.get_all_headlines(soup), self.get_
all_articles(soup)

        resp = [
            {
                "category": category,
                "summary": self.get_headline(hsoup),
                "article": self.get_article(asoup)
            }
                for hsoup, asoup in zip(headlines, articles)
        ]
        return resp
```

Let us go through this code one by one; in line 2, we get the soup, line 3, we get all the headlines and articles on that page by the methods written previously. Finally, we use list comprehension to get all the summaries and articles in that category from lines 5-12.

We are complete with getting data from one news category but let us enable Selenium to go through all the categories automatically. We can do that by clicking on the expand button (hamburger icon on the left in image 9.1); here is the method:

```
    def expand(self) -> bool:
        try:
            expand = self.driver.find_element_by_xpath("/html/body/
div[1]/div/button")
            expand.click()
            return True
        except Exception as err:
            print(err)
            return False
```

Now, this is a quick and dirty way of clicking the button using xpath. To get the xpath, you can simply right-click on the element and inspect, then in the DOM, you can. Again, right-click and select xpath of the element. As an alternative, you can use this amazing plugin called *Ruto* to get the xpath easily. In an actual Selenium project, we would wait for the element, and we will handle the exceptions such as **`NoSuchElementException`**, and so on. But for our purpose, we will create a quick and dirty solution.

Let us now get all the categories using the same beautifulsoup trick earlier, the following is the function:

```
def get_categories(self, soup: BeautifulSoup) -> List[str]:
    try:
        categories = [   c.text.strip() for c in
            soup.find('ul', class_=["category-list"]).find_all('a')
        ]
        return categories[1:] # excluding All News
    except Exception as err:
        return []
```

The categories are inside the **`ul`** with class category-list, and all of them are anchor(**`a`**) tags. Also, I am excluding the **`All-News`** category to avoid duplication.

Now let us combine all our methods to build the summarization dataset.

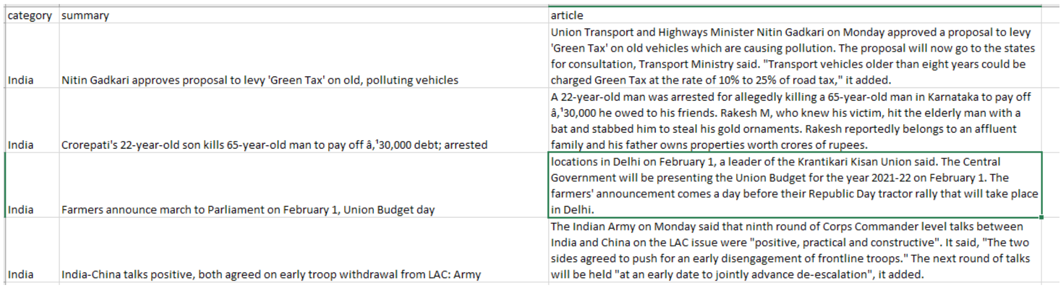
```
def build_summarization_dataset(self) -> List[dict]:
    soup = self.get_soup()
    summary_dataset = []
    if self.expand():
        for c in self.get_categories(soup):
            category_xpath = f"//li[text()='{c}']"
            self.driver.find_element_by_xpath(category_xpath).click()
            resp = self.get_data_from_page(c)
            summary_dataset.extend(resp)
            if not self.expand(): break
    return summary_dataset
```

First, we get the soup (line 2), then we click on the expand button (line 4); if it successfully opens then, we will go through each category (line 6), create the xpath for that category (line 6), then click on that category item (line 7), get the page data (line 8), and add to our summary dataset. Then we expand again to click on the next category.

Now, let us complete our script (**`main.py`**).

```
if __name__ == "__main__":
    url = "https://inshorts.com/en/read"
    crawler = Scrapper(url)
    sd = crawler.build_summarization_dataset()
    df = pd.DataFrame(sd)
    name = str(uuid.uuid4())
    df.to_csv(f"scrapped_inshorts_{name}.csv", index=False)
    crawler.driver.quit()
```

We create a crawler, build the summarization dataset, save it into csv using pandas, and quit the driver. This will create our summarization dataset; let us see how the data looks.



category	summary	article
India	Nitin Gadkari approves proposal to levy 'Green Tax' on old, polluting vehicles	Union Transport and Highways Minister Nitin Gadkari on Monday approved a proposal to levy 'Green Tax' on old vehicles which are causing pollution. The proposal will now go to the states for consultation, Transport Ministry said. "Transport vehicles older than eight years could be charged Green Tax at the rate of 10% to 25% of road tax," it added.
India	Crorepati's 22-year-old son kills 65-year-old man to pay off â,¹30,000 debt; arrested	A 22-year-old man was arrested for allegedly killing a 65-year-old man in Karnataka to pay off â,¹30,000 he owed to his friends. Rakesh M, who knew his victim, hit the elderly man with a bat and stabbed him to steal his gold ornaments. Rakesh reportedly belongs to an affluent family and his father owns properties worth crores of rupees.
India	Farmers announce march to Parliament on February 1, Union Budget day	locations in Delhi on February 1, a leader of the Krantikari Kisan Union said. The Central Government will be presenting the Union Budget for the year 2021-22 on February 1. The farmers' announcement comes a day before their Republic Day tractor rally that will take place in Delhi.
India	India-China talks positive, both agreed on early troop withdrawal from LAC: Army	The Indian Army on Monday said that ninth round of Corps Commander level talks between India and China on the LAC issue were "positive, practical and constructive". It said, "The two sides agreed to push for an early disengagement of frontline troops." The next round of talks will be held "at an early date to jointly advance de-escalation", it added.

***Figure 9.3**: First look into the summarization dataset*

Also, few more improvements to our crawler can be to load more articles by clicking on the load more button using Selenium, here is the code:

```
    def load_more(self):
        try:
            self.driver.find_element_by_id("load_more_btn").click()
        except NoSuchElementException:
            print("[ INFO ] No more load more!")
            return
```

Also, another improvement can be to get the whole text from the actual article by clicking on the read more button. This one is tricky as it will take us to a new page in a new window, and since the website will be different, we need different locators for each page. The first problem is easy to solve with Selenium; you can switch windows easily with the **`switch_to`** method. For the second one, you can use general paragraph texts. As a fun side project, try to implement it.

9.1.2 Data cleaning

Let us deep dive into some of the generic data cleaning techniques, we do not use many of them, but it is an essential skill for NLP. We will go through the code example for each, and as an exercise, the reader may pick one NLP dataset from Kaggle and try all these data cleanings.

9.1.2.1 Contractions

Contractions are words such as I'm, you're, and so on. We need to expand these as our model will think "I'm" and "I am" different; also, domain-specific abbreviations needed to be expanded. For example, IP and Internet Protocol are the same thing. We can use a dictionary-based approach here. The idea is to maintain a dictionary of all the contractions and parse the text using regex. Whenever you get contractions inside a text, replace it with the dictionary, you can use the contractions (**`pip install contractions`**) package to do it.

```
import contractions

contractions.fix("you're happy now")
```

This will return "*you are happy now*." You can go through some more examples at **https://github.com/kootenpv/contractions**. For your domain-specific abbreviations, use **`contractions.add`**. For example,

```
[contractions.add(key, value) for k, v in MY_CONTRACTIONS_DICT.items()]
```

Here, **`MY_CONTRACTIONS_DICT`** contains all the domain specific contractions.

9.1.2.2 Special characters

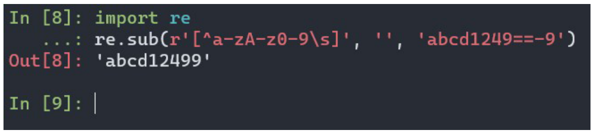
Special characters are often un-necessary to the machine learning problem; we can remove it also.

```
import re

re.sub(r'[^a-zA-Z0-9\s]', '', 'any string')
```

Here we use the regex package to delete non alphabet and numbers. Let us see it with an example.

```
re.sub(r'[^a-zA-Z0-9\s]', '', 'abcd1249==-9]')
```

The output of the preceding code:



```
In [8]: import re
   ...: re.sub(r'[^a-zA-z0-9\s]', '', 'abcd1249==-9')
Out[8]: 'abcd12499'

In [9]: |
```

***Figure 9.4**: Removing special characters*

If you don't want the numbers or keep some special characters, you can modify the regex pattern.

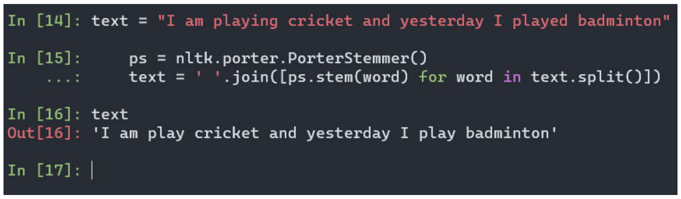
9.1.2.3 Stemming and lemmatization

Stemming and lemmatization are two techniques to reduce the words to their base form. For example, *'play'* and *'playing'* has a similar meaning, and when reduced to the base form, i.e., *'play,'* the model will take both as the same.

The difference between stemming and lemmatization is stemming is a faster approach as it is rule-based. Still, lemmatization gives proper base form because it is a dictionary-based approach, but it's slow. So, use your judgment and problem understanding when using either of the approaches. For stemming, we can use the nltk implementation of the Porter Stemmer algorithm.

```
import nltk


text = "I am playing cricket and yesterday I played badminton"
ps = nltk.porter.PorterStemmer()
text = ' '.join([ps.stem(word) for word in text.split()])
print(text)
```

The output of the preceding code:



```
In [14]: text = "I am playing cricket and yesterday I played badminton"

In [15]:     ps = nltk.porter.PorterStemmer()
    ...:     text = ' '.join([ps.stem(word) for word in text.split()])

In [16]: text
Out[16]: 'I am play cricket and yesterday I play badminton'

In [17]: |
```

***Figure 9.5**: Stemming*

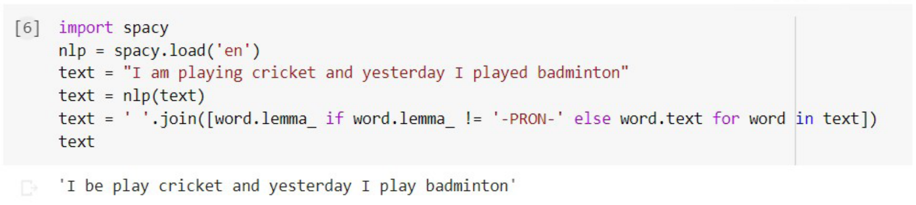
Here we can see the played and playing are reduced to its base form play. Now let us look at lemmatization.

We will use spacy to do this; the following is the code:

```
import spacy
```

```
nlp = spacy.load('en')
text = "I am playing cricket and yesterday I played badminton"
text = nlp(text)
text = ' '.join([word.lemma_ if word.lemma_ != '-PRON-' else word.text
for word in text])
text
```

The output of the preceding code:



```
[6] import spacy
    nlp = spacy.load('en')
    text = "I am playing cricket and yesterday I played badminton"
    text = nlp(text)
    text = ' '.join([word.lemma_ if word.lemma_ != '-PRON-' else word.text for word in text])
    text

    'I be play cricket and yesterday I play badminton'
```

***Figure 9.6**: Lemmatization*

As you can see here, the '*playing*' and '*played*' reduced to '*play*' and also '*am*' reduced to '*be.*'

9.1.2.4 Stop words

Now comes the stop words. These are words such as '*is,*' '*am,*' and so on. These are the most occurring words but carry little meaning when solving problems such as topic modeling or text classification. But stop words are not necessarily bad for the model. So be careful in removing the stop words blindly. We can use nltk to create a stop words list and then remove the words from text using this list.

```
stopwords = nltk.corpus.stopwords.words('english')
```

For this to work, you need to download the stop words file using **`nltk.download`**. Explore the documentation more to download more nltk related data.

There are other text cleaning techniques such as email removal, Unicode normalization, and so on. Text cleaning is extremely domain-specific. You may require some additional text cleaning to normalize your text. This is the reason I am not doing it on my data. Just be aware of packages such as nltk, spacy, Unicode, and so on. to clean your data. Also, few libraries do text cleaning, but I generally don't prefer them myself as text cleaning is not a global technique used for all the NLP problems. It is domain and problem-specific. But the interested reader may use some of these libraries, for example, clean-text.

9.2 Word embeddings

Word embeddings are a process of changing words into numbers/vectors with an added property that those vectors capture the semantic meaning of the word. Now how to do this? Unlike a bag of words method, where each word is represented as one hot encoded value, word embeddings are dense. Because if we have a 50,000-word vocabulary in a text corpus and one of our sentences/documents has only 15 words in it. Then the whole vector will have 50000 dimensions, and there will be only 15 places where the value will be 1's and the rest all will be 0's. This is called highly sparse data. Word embeddings solve this issue by creating a dense representation of a word. The intuition behind this is *a word's meaning is derived from its surrounding words*. So to start creating word embedding, we decide on an embedding dimension. This will be the length of the vector in which the word will be represented. It is generally taken as 300. Initially, these will be random values. Then there are two ways we can learn these vectors. The following is the image showing both the ways:

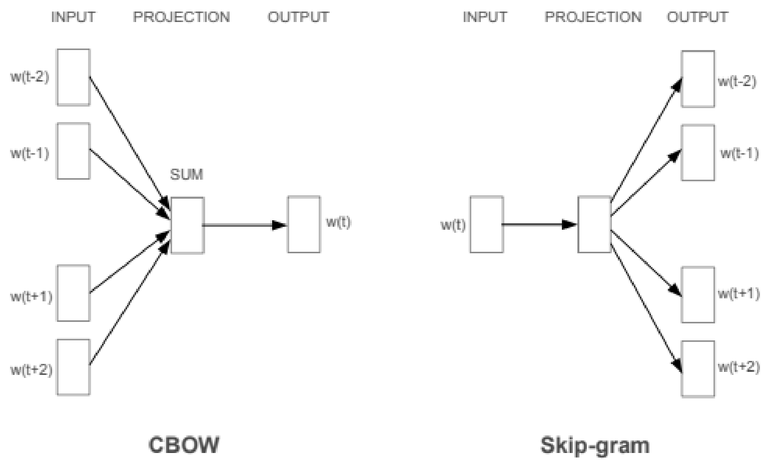


Figure 9.7: *CBOW and Skip-Gram models from "Efficient Estimation of Word Representations in Vector Space" by Mikolov et al.*

Let us take an example sentence, "*I am eating a red apple*," In the first method, we will take '*eating*' as an example word. So, we will look into 2 (this is called window-size) words before and after. i.e., "*I*," "*am*," "*a*," "*red*," And using these four words, we will try to predict the word '*eating*.' This is called **Continuous Bag of Words** (**CBOW**). We used to predict word '*eating*' is called context words, and the word that we are predicting is called the **target word**. And we took the example for a single word in the sentence, but this process is repeated with a sliding window. And the words that we choose before and after were 2 + 2 = 4 words. This is called **window size,** and it is one of the hyperparameters. In our case, the window size is 2.

In the second method, we do the opposite; we take a single word and try to predict the 4 surrounding it. Not at once, one by one. For example, 'eating' we will have four rows, i.e., *'eating – I,' 'eating – am,' 'eating – a', 'eating – red.'*

So, in actual training, we will multiply the embeddings (remember the 300-dimensional vectors we have for each, which are randomly initialized) of the context and the target word to get a prediction. Then, we need to use a softmax to identify each word's probability. Because we need to output a single word, and here if the vocab size is really big (which its often is), calculating the final softmax will be computationally infeasible (imagine creating softmax output for 50000 words and in the denominator, we will need to sum of 50000 words too, and in skip gram, the size is even more). So, we use something called **negative sampling**. So rather than predicting the softmax for all the words, we simply change the problem. We will take the target word and call it an output word because we won't predict the softmax for the output. Rather we will take both input and output words and try to predict a 0 or 1 target. This means if the input and target are neighbors, then predict 1, else 0. This can now be trained faster. We will also add some examples (not all) of non-neighbors that are randomly chosen.

Now, let us see how this **`word2vec`** creates the embeddings that capture the meaning of the words; first, we will download a pre-trained word2vec model, a 300-dimensional embedding vector.

The following are the commands (run it in Google Colab):

```
! wget -c "https://s3.amazonaws.com/dl4j-distribution/GoogleNews-
vectors-negative300.bin.gz"

! gunzip GoogleNews-vectors-negative300.bin.gz
```

Now let us import the genism package to load in the model.

```
import gensim

model = gensim.models.KeyedVectors.load_word2vec_format('GoogleNews-
vectors-negative300.bin', binary=True)
```

Here we will take few words to see the clusters the embeddings make, and if the word2vec vectors carry some semantic meaning, then the similar words should be closer to the keywords mentioned as follows:

```
keys = ['London', 'Python', 'Galileo', 'Monday', 'Twitter', 'android']


embedding_clusters = []

word_clusters = []

for word in keys:

    embeddings = []
    words = []
    for similar_word, _ in model.most_similar(word, topn=5):
        words.append(similar_word)
        embeddings.append(model[similar_word])
    embedding_clusters.append(embeddings)
    word_clusters.append(words)
```

In the preceding code, we take the top 5 similar words (this is calculated by cosine distance from the word in the embedding space). We are creating the embedding clusters by taking the embeddings of these five words.

To see these 300-dimensional clusters, we will use a dimensionality reduction technique to bring the dimension to 2. We will use TSNE as it preserves the local structure.

```
from sklearn.manifold import TSNE
import numpy as np


embedding_clusters = np.array(embedding_clusters)
n, m, k = embedding_clusters.shape
tsne_model = TSNE(perplexity=5, n_components=2, init='pca', n_iter=3500,
random_state=32)
embeddings_en_2d = np.array(tsne_model.fit_transform(embedding_clusters.
reshape(n * m, k))).reshape(n, m, 2)
```

Line-6 the T-SNE model is created with *n-components* = 2. And *line – 7* we are just finding the embeddings in 2D. Now, let us plot these embeddings in a scatter plot with different colors for different word clusters.

```
import matplotlib.pyplot as plt
import matplotlib.cm as colormap
% matplotlib inline


plt.figure(figsize=(16, 9))
colors = colormap.rainbow(np.linspace(0, 1, len(keys)))
for label, embeddings, words, color in zip(keys, embeddings_en_2d, word_
clusters, colors):
    x = embeddings[:, 0]
    y = embeddings[:, 1]
    plt.scatter(x, y, color=color, alpha=0.6, label=label)
    for i, word in enumerate(words):
        plt.annotate(word, alpha=0.5, xy=(x[i], y[i]), xytext=(5, 2),
                                textcoords='offset points', ha='right',
va='bottom', size=8)
plt.legend()
plt.title('Similar words from Google News Word2Vec Model')
plt.grid(True)
plt.show()
```

The output of the preceding code:

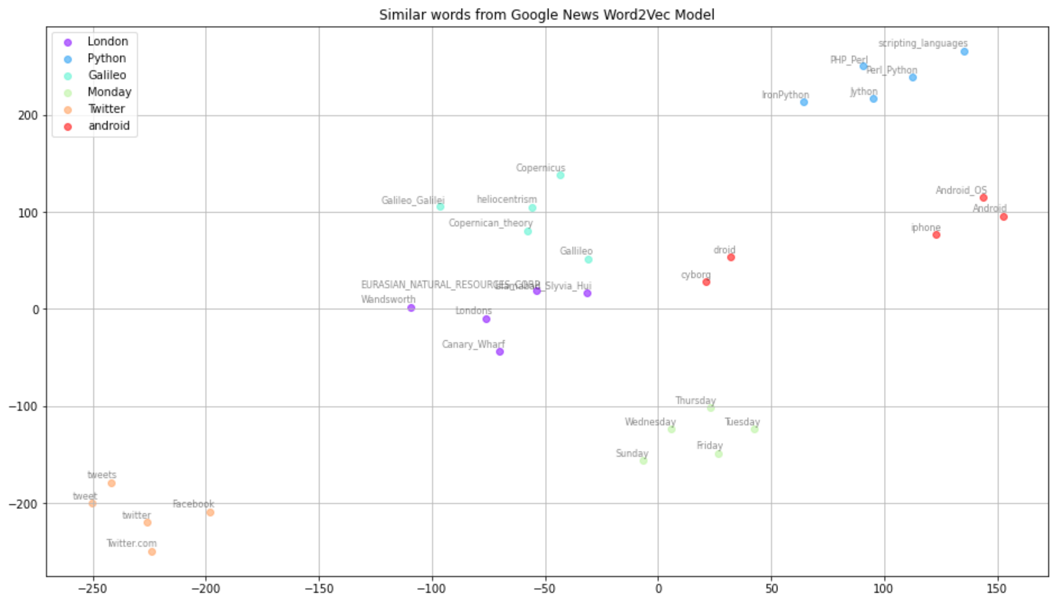


Figure 9.8: Word vector clustering

As you can see, the clusters are meaningful; weeks are clustered to the bottom right, languages (python, PHP, and so on.) are clustered in the top right, and so on. So, these vectors contain some semantic meaning.

9.3 A quick guide to RNNs

This will be a quick guide to Recurrent Neural Networks, and we will just touch the surface of these models. Before transformers, these models reign over the NLP space. But with the emergence of transformers, RNNs are seldom used nowadays. We had a bag of words models without any recurrent connections in the past. For example, we can take one hot encoded input word and build a classification model called email spam or ham. The problem with that approach was there was no sense of position. But language has. We cannot jumble up words in a sentence because that would mean something entirely different. So, the sequence is important. That is where Recurrent Neural Networks come into the picture.

9.3.1 Simple Recurrent Neural Network

Unlike Convolutional Neural Networks or Feed Forward Networks (Multi Layered Perceptron), Recurrent Neural Networks take past input into the network too; the following is an image of a recurrent neural network:

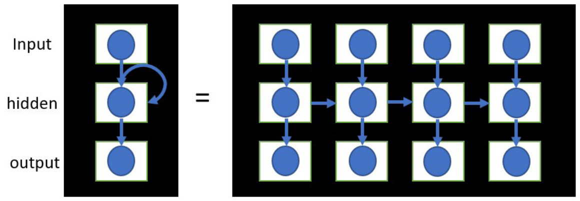


***Figure 9.9**: A simple RNN*

In the figure to the right, the network has a hidden layer where the input is connected; we can unroll the network to a better view, i.e., to the network's right. The network is a Multi Layered Perceptron, but the idea is that the inputs are in sequence of words. Now there is more to RNNs than this, but the main intuition is that we accommodate sequence in the network to remember the past. This is useful for sequential tasks. For example, in translation, the 4th output word may depend on the 1st input word. This network can achieve this.

The major problem with this is this sequential nature; Recurrent Neural Networks are hard to train as running this model in parallel is difficult. Another problem is called exploding/vanishing gradients. Let us explore what those are.

Because the network becomes deep when we try to add more and more past input into the network, there are only 4-time steps added in the above example. As we know, the hidden layer weights get updated via gradient descent or some form of gradient descent. Suppose the network becomes too deep and initially(in the first few layers). In that case, the gradient is low (this may happen because, say in a translation task, the first few words are not important to the output, so the gradient is low), then moving that gradient deeper into the network will die down significantly(as in backpropagation we propagate the gradients in backward direction), and hence for longer sequences the network won't learn or in a way for the network has extremely short term memory.

The other way around where the gradients start big and get bigger and bigger when they reach deeper layers is also true. Gradient explosion is relatively easy to solve; we can clip the gradient values to a maximum limit. But vanishing gradient is a problem we cannot easily solve. Here a concept called gates comes into the picture. This vanishing gradient problem is somewhat resolved by introducing **Long Short-Term Memories** (**LSTMs**) / **Gated Recurrent Units** (**GRUs**). There is a small difference between these two; LSTMs have one more gate than GRU. Let us see the structure of these modified RNN architectures:

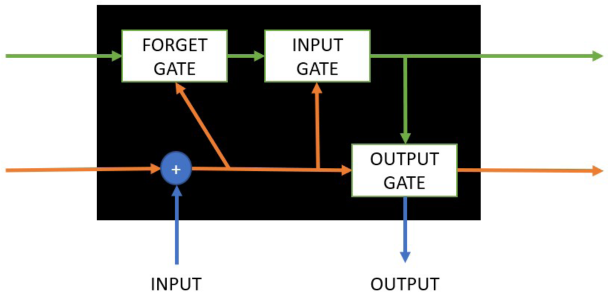


***Figure 9.10**: Gates to the rescue*

These gates are responsible for solving the vanishing gradient problem; now, we have the flexibility of forgetting unimportant words through the forget gates.

Even with the introduction of gates, the vanishing gradient problems still prevalent in LSTMs/GRUs. Also, since they are a modified version of RNNs, these are still a pain to train. So, a ground-breaking approach to NLP comes in the form of transformers, which solved these issues with RNNs. Please note that RNNs are a big topic, and simplification is made in the above discussion to get the gist. RNNs solved many problems, so we cannot dilute these, but detailed discussion on RNNs is beyond the scope for this book, and interested readers can read *Andrej Karpathy's* blog (**http://karpathy.github.io/2015/05/21/rnn-effectiveness/**) for more information on the RNNs.

9.4 Transformers

Transformers removed the recurrent relationship that was present in Recurrent Neural Network architectures. This revolutionized natural language processing. This is often called the imagenet moment for NLP. The idea behind transformers is simple and innovative yet immensely powerful. Let us check the transformer architecture from the original paper, "*Attention is all you need*" by *Vaswani et al.*, 2017.

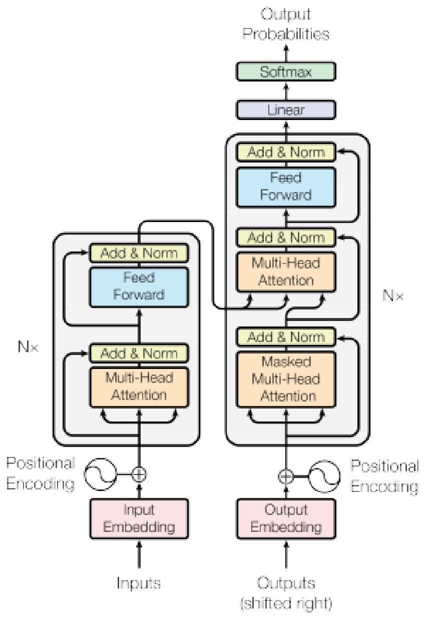


Figure 9.11*: Transformers architecture from "Attention is all you need" by Vaswani et al., 2017.*

The left side block is called an **encoder,** and a right-side block is called a **decoder**. Now the encoder/decoder block can be repeated n times to create a much deeper network; this is why they are grouped. But encoder and decoder blocks are not the innovation as RNN sequence to sequence networks also used such encoder-decoder structure to solve seq-to-seq tasks such as translation. The key idea is to use self-attention. The idea is simple.

If we have a sentence, "*She is eating a green apple*." In this sentence, the words green and apple are highly related; similarly eating, and apple is also highly related. But not eating and green. So, our naïve implementation of word2vec, which draws meaning to a word by its surrounding words, won't capture this intuition. The idea is to give a way to introduce self-context or self-attention. We will go through this in detail in the next section.

Then comes a residual connection and layer normalization; the idea is to pass the input and attention input (inputs passed through the attention layer). Because we discussed the vanishing gradient problem and with deeper networks, this residual connection helps to learn.

Another notable innovation is the position encoding; since we no longer allow any recurrent connections, the network should be aware of some sort of position of words. This information is given by adding a positional encoding to the input embeddings (input embeddings are nothing but the embedding discussed in word2vec). Or another way is to add positional embedding to the embeddings. The difference between the two approaches is that the positional embedding is learned, but the positional encoding is not. It is a fixed vector(non-learnable) applied to the embedding to give some positional information. We will learn about it in detail in upcoming sections.

Finally, in the decoder block, output encoding (shifted right because we don't know what is right, we want to predict that). And masked attention (as we should not see the future to attend to will be cheating to have the information already present that we want to predict). Then, encoder-decoder attention is not a new concept as this kind of connection is present in the RNN based encoder-decoder blocks. This allows the output to attend to different parts of the input. Finally, there are also residual connections followed by a layer norm. Then a feed-forward network. After this, based on the requirement, there is a linear and a softmax layer, as we want to predict words.

These may sound complicated but let's look into the pseudo code for this, explaining all of these.

```
def forward(self, x):
    ax = self.attention(x)
    x = self.norm_layer1(ax+ x)
    x = self.dropout(x)
    fedforward = self.feedforward(x)
    x = self.norm_layer2(fedforward + x)
    x = self.dropout(x)
    return x
```

So, the input (in our case is embedding + positional encoding) to the attention layer is followed by a layer norm with residual connection and a dropout. Then, a feedforward layer with another layer norm and residual connection and a dropout. For the decoder layer, there will be masked attention and encoder-decoder attention. This code is an oversimplification and does not dive into the details of attention, normalization layer, and so on. This is to give an easily consumable simplified version of the transformer model.

9.4.1 Attention is all you need

The idea behind attention is to get an association between input words; another way of thinking about it is matrix products. The following is a diagram, and the equation explains dot product attention. (**Note**: There are different types of attention mechanisms, and learning about all of them is beyond the scope of this book. We will only go through the scaled dot product attention mentioned in the paper "*Attention is all you need*" (Vaswani *et al*., 2017).

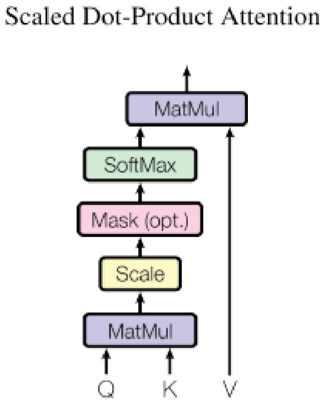


Figure 9.12: *Single attention head*

To intuitively understand it, let us consider how we can get a process to understand the importance of words in the same sentence. We will require some sort of *score*. To calculate a score for a word in a sentence with respect to every other word in the sentence is the basis of the attention mechanism. First, we derive three vectors, query, key, and value, from the same input vector (in our case, it is the embedding vector for input) by multiplying it with three different matrices that we train during the training process.

Imagine the sentence "*She is eating a green apple*," there will be three different vectors for each of the words in the sentence. Btw these vectors are smaller than the embedding dimension because we don't use single attention; we use multi-headed attention (this means we will have more than one attention head). And to reduce the computation, we are reducing the size; this is not a necessity. It is a design choice; why we require multi-headed attention will be discussed next.

For each of these queries and key vectors for each word (there will be six sets of 3 vectors (query, key, and value) each as we have six words in our sentence). We will take the dot product. So, there will be six scores for position 1 (6 queries (dot product) 6 key). And we will scale it through the square root of dimension (keeps the gradients stable); after that, we will take a softmax (this will give the contribution of each word to position 1). And we will multiply this with the value vector to get higher values for higher softmax scores and lower values for lower softmax scores (drown out the irrelevant words). Finally, we sum things up (because there are six things for the first position, remember!). This will contain the information about all the attention to the first word. We do this to every word in the sentence to get a matrix.

9.4.2 Multi-headed attention

Multi-headed attention is one of the key innovations. We create multiple attention heads rather than allow the model to learn different things in the sentence. Or giving the ability to attend to different things in the sentence. For example, one head may learn about grammar. One head may learn about structure. These are called representation subspaces. The following is the picture from the paper about multi-headed attention:

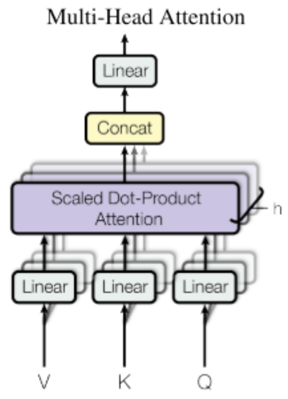


***Figure 9.13**: Multi-headed attention*

This is also the reason to choose the dimension key, query, and value vector sizes to be lesser than the embedding dimension to preserve computation time. Finally, the output of all these attention heads is concatenated and passed through a linear layer.

9.4.3 Positional encoding

Another innovation is the positional encoding part in the paper. As there are no recurrent connections, we need to add position information to the model. This is done through positional encoding in the paper. This information needs to:

1. Be unique (for each time step). To represent the word's position in the sentence.
2. Represent same time step distance across the whole sentence. Meaning the time step distance between 1st word and 2nd word is 1. Similarly, for 2nd word and the 3rd word is also 1.

Also, the encoding should generalize to longer sequences. It needs to be deterministic as we are not learning anything in positional encoding (unlike positional embedding, where we will learn embedding).

The idea proposed in the paper was to use sinusoidal functions. The intuition is that sine and cos will have complimentary values in subsequent steps.

$$\vec{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k . t), & \text{if } i = 2k \\ \cos(\omega_k . t), & \text{if } i = 2k+1 \end{cases}$$

$$\text{Where } \omega_k = \frac{1}{10000^{2k/d}}$$

This creates the positional encoding to each of the word dimensions. Also, the authors chose this function because it allows the model to learn relative positions easily, that is if a word is at position 10. Then it is easier to learn that from word 1.

9.4.4 Pre-training tasks

The main magic comes from the self-supervised pre-training tasks on the transformer models. This is where the models such as BERT, GPT, BART, and so on comes into the picture. They use transformer architecture trained, not some self-supervised pretraining tasks. In the original paper, it was for a translation model, which is a sequence-to-sequence task. Let us now go through a few of the pretrained tasks that these language models generally use.

- **Next word prediction**

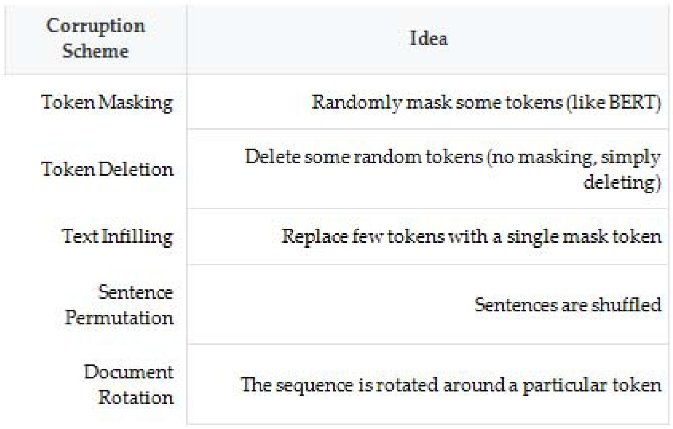
  This task is for autoregression models such as **Generative Pre-Training** (**GPT**) (*Improving Language Understanding by Generative Pre-Training* paper by *Mike Lewis et al*.). This works like a language model which tries to predict the next word in the sequence. This can be self-supervised because we can just set the next word as label/target in a particular sequence of words. The limitation with GPT is that it learns only in a single direction, meaning the words do not have a context for future words. This is essential because it is a next word prediction model, and we cannot have future words as input. GPT essentially takes the transformer decoder part (because it has the masked self-attention to prevent the future word input).

- **Blank word prediction**

  Blank word prediction is used by **Bidirectional Encoding From Transformers** (**BERT**) model from *Pre-training of Deep Bidirectional Transformers for Language Understanding* paper by *A.Radford et al*.. This provides BERT with a bidirectional context. The idea is that rather than predicting the next word in the sequence, we randomly mask few words between the sentences and try to predict that. Since we are not predicting the next word here, we can give future words as input too. But the problem is that BERT now becomes an autoencoding model and not a regressive auto model. So, it outperforms GPT in a variety of tasks but not in text generation. Because text generation is essentially an autoregressive task. BERT takes the encoder part of the transformer model because we don't have a masking concept there.

- **Reconstructing corrupt text**

  Finally, with the issues of GPT (being non-bidirectional but autoregressive) and BERT (being bidirectional but not autoregressive), we have BART (*Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*). This combines the decoder part of GPT and encoder part of the BERT, and the task it tries to do is corrupting the text with different approaches; the following is a table of such corruptions:



Corruption Scheme	Idea
Token Masking	Randomly mask some tokens (like BERT)
Token Deletion	Delete some random tokens (no masking, simply deleting)
Text Infilling	Replace few tokens with a single mask token
Sentence Permutation	Sentences are shuffled
Document Rotation	The sequence is rotated around a particular token

***Table 9.1**: Different Corruption Tasks*

  Using this corruption BART input, the model tries to generate the output. This gives BART greater performance over language generation tasks than BERT. This is an extremely oversimplified version of BART to provide the intuition, and the interested reader may choose to go through the actual BART paper (**https://arxiv.org/abs/1910.13461**).

There is a lot more to BERT, GPT, BART models than just this pretraining task, but going through all these architectures is beyond the book's scope, and the overall idea remains the same, a transformer model.

With this, we finish the theory on transformer models; now, we will jump straight into some of the unsupervised methods to summarize.

9.5 Unsupervised summarization methods

Let us start with some unsupervised summarization methods; the first model we will try is a **text rank model. Then we will use a pretrained transformer model; now, the latter option is not an unsupervised method as the pre-trained model is trained on supervised data.** Still**,** I am calling it unsupervised as I will only be using the model without any annotation data.

9.5.1 Text rank

Text rank is a clustering algorithm; this is an extractive summarization algorithm. The summary will be extracted from the model sentences only; the roots of this algorithm are from Google's PageRank (by Larry Page). So, the idea is to derive a rank of a webpage by determining how important it is. And the importance is defined as how many pages are linking the webpage. Let us try to understand this image.

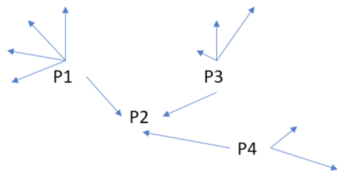


***Figure 9.14**: Text rank graph*

In the preceding figure, there are four pages, and page p2 is linked from p1, p3, and p4. P1 also has four unique links to some other sites; similarly P3, P4 have 3, 2 other unique links to other sites. So, the page rank of P2 is given by.

*Page Rank(p2) = d + d * (Page Rank(p1) /5 + Page Rank(p3) / 4 + Page Rank(p4) / 3)*

So, the contribution of p1 to the page rank of p2 is given by its page rank divided by the number of unique links. This makes sense too. There is a multiplier 'd' called the **damping constant**; it solves two purposes:

- If a page has no links leading to it, the page rank without the 'd' term will be zero, 'd' ensures this does not happen
- It dampens the no of links if there is a lot of links

In another way, the page rank algorithm contribution is a kind of similarity between two pages. In our context, we need some similarity measure between our sentences to create a page rank metric for each sentence and pick those with the highest page rank value, which means selecting the most important sentences.

We first need to change our sentences to vector representation to take the cosine distance between them. This will give us the similarity measure. After that, we can apply the page rank algorithm to calculate the page rank values. To calculate the **`vector/embeddings`** for a sentence, we can use the word embeddings we learned earlier.

So, let us start with the code (run the following code in Google Colab):

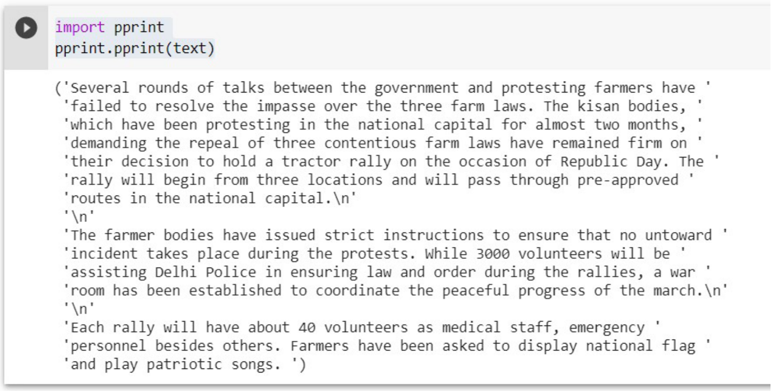
```
import numpy as np

import pandas as pd
import nltk
import re
from nltk.tokenize import sent_tokenize
from nltk.corpus import stopwords
from gensim.models import Word2Vec
from scipy import spatial
import networkx as nx
nltk.download('punkt')
```

We will use **`sent_tokenize`** to tokenize the sentences from *nltk*; we will use stop words from removing some stop words to decrease the performance. For word embeddings, we will use the *genism* package. Finally, for the page rank algorithm, we will use the **`networkx`** package.

Let us start with an example text from the inshorts app.

```
text='''Several rounds of talks between the government and protesting
farmers have failed to resolve the impasse over the three farm laws.
The kisan bodies, which have been protesting in the national capital
for almost two months, demanding the repeal of three contentious farm
laws have remained firm on their decision to hold a tractor rally on the
occasion of Republic Day. The rally will begin from three locations and
will pass through pre-approved routes in the national capital.

The farmer bodies have issued strict instructions to ensure that no
untoward incident takes place during the protests. While 3000 volunteers
will be assisting Delhi Police in ensuring law and order during the
rallies, a war room has been established to coordinate the peaceful
progress of the march.

Each rally will have about 40 volunteers as medical staff, emergency
personnel besides others. Farmers have been asked to display national
flag and play patriotic songs. '''

import pprint
pprint.pprint(text)
```

The output of this code:



```
import pprint
pprint.pprint(text)
```

```
('Several rounds of talks between the government and protesting farmers have '
 'failed to resolve the impasse over the three farm laws. The kisan bodies, '
 'which have been protesting in the national capital for almost two months, '
 'demanding the repeal of three contentious farm laws have remained firm on '
 'their decision to hold a tractor rally on the occasion of Republic Day. The '
 'rally will begin from three locations and will pass through pre-approved '
 'routes in the national capital.\n'
 '\n'
 'The farmer bodies have issued strict instructions to ensure that no untoward '
 'incident takes place during the protests. While 3000 volunteers will be '
 'assisting Delhi Police in ensuring law and order during the rallies, a war '
 'room has been established to coordinate the peaceful progress of the march.\n'
 '\n'
 'Each rally will have about 40 volunteers as medical staff, emergency '
 'personnel besides others. Farmers have been asked to display national flag '
 'and play patriotic songs. ')
```

***Figure 9.15**: Sample Article*

Now let us create the sentence embeddings:

```
sentences=sent_tokenize(text)
sentences_clean=[re.sub(r'[^\w\s]','',sentence.lower()) for sentence in sentences]
stop_words = stopwords.words('english')
sentence_tokens=[[words for words in sentence.split(' ') if words not in stop_words] for sentence in sentences_clean]

w2v=Word2Vec(sentence_tokens,size=1,min_count=1,iter=1000)
sentence_embeddings=[[w2v[word][0] for word in words] for words in sentence_tokens]
max_len=max([len(tokens) for tokens in sentence_tokens])
sentence_embeddings=[np.pad(embedding,(0,max_len-len(embedding)),'constant') for embedding in sentence_embeddings]
```

Explanation of the code:

- Line 1: Tokenize the sentences
- Line 2: Clean up the sentences and remove the punctuations (we don't need these as the sentence similarity only depends on the words here)
- Line 3-4: Remove the stop words
- Line 6-7: Find the word2vec vectors (size = 1 as I am concatenating the word embeddings to create the sentence embedding, but in case you choose to use different size embeddings, make sure to average or sum it, you can use concatenation but the dimension will be huge for huge sentences)

- Line 8-9: Finding maximum length tokens and padding the embeddings to make them the same size for each

We have a vector representation of the sentence we can go and build the similarity matrix to create the page rank. We use a simple sentence vectorization scheme, and the reader may choose to play with different vectorization schemes. Now, let us check the code for creating the similarity matrix.

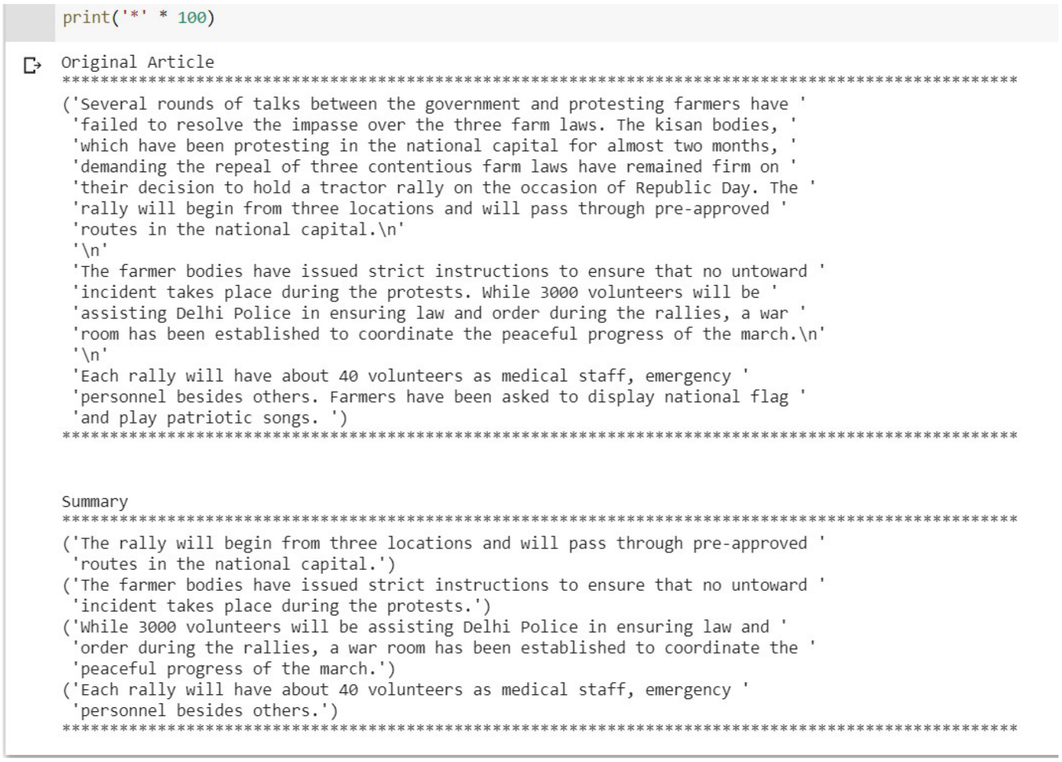
```
similarity_matrix = np.zeros([len(sentence_tokens), len(sentence_
tokens)])
for i,row_embedding in enumerate(sentence_embeddings):
    for j,column_embedding in enumerate(sentence_embeddings):
        similarity_matrix[i][j]=1-spatial.distance.cosine(row_
embedding,column_embedding)
```

Here we go through each embedding and calculate the cosine similarity between the sentences. Now, using this we can create the page rank model and create summarization:

```
nx_graph = nx.from_numpy_array(similarity_matrix)
scores = nx.pagerank(nx_graph)


top_sentence={sentence:scores[index] for index,sentence in
enumerate(sentences)}
top=dict(sorted(top_sentence.items(), key=lambda x: x[1], reverse=True)
[:4])


print("Original Article")
print('*' * 100)
pprint.pprint(text)
print('*' * 100)
print("\n")
print("Summary")
print('*' * 100)
for sent in sentences:
    if sent in top.keys():
        pprint.pprint(sent)
print('*' * 100)
```

Line 1-2, we create the network graph and build the page rank model. Then in lines 4-5, we used the scores to find the top 4 sentences in the whole text; from lines 7-8, we are just printing the results.

The output of the preceding code:



```
print('*' * 100)
```

```
Original Article
****************************************************************************************************
('Several rounds of talks between the government and protesting farmers have '
 'failed to resolve the impasse over the three farm laws. The kisan bodies, '
 'which have been protesting in the national capital for almost two months, '
 'demanding the repeal of three contentious farm laws have remained firm on '
 'their decision to hold a tractor rally on the occasion of Republic Day. The '
 'rally will begin from three locations and will pass through pre-approved '
 'routes in the national capital.\n'
 '\n'
 'The farmer bodies have issued strict instructions to ensure that no untoward '
 'incident takes place during the protests. While 3000 volunteers will be '
 'assisting Delhi Police in ensuring law and order during the rallies, a war '
 'room has been established to coordinate the peaceful progress of the march.\n'
 '\n'
 'Each rally will have about 40 volunteers as medical staff, emergency '
 'personnel besides others. Farmers have been asked to display national flag '
 'and play patriotic songs. ')
****************************************************************************************************


Summary
****************************************************************************************************
('The rally will begin from three locations and will pass through pre-approved '
 'routes in the national capital.')
('The farmer bodies have issued strict instructions to ensure that no untoward '
 'incident takes place during the protests.')
('While 3000 volunteers will be assisting Delhi Police in ensuring law and '
 'order during the rallies, a war room has been established to coordinate the '
 'peaceful progress of the march.')
('Each rally will have about 40 volunteers as medical staff, emergency '
 'personnel besides others.')
****************************************************************************************************
```

***Figure 9.16**: Summarization from text rank*

As you can see, the unsupervised model also gives pretty good results. Since this is unsupervised, we have not done any annotation.

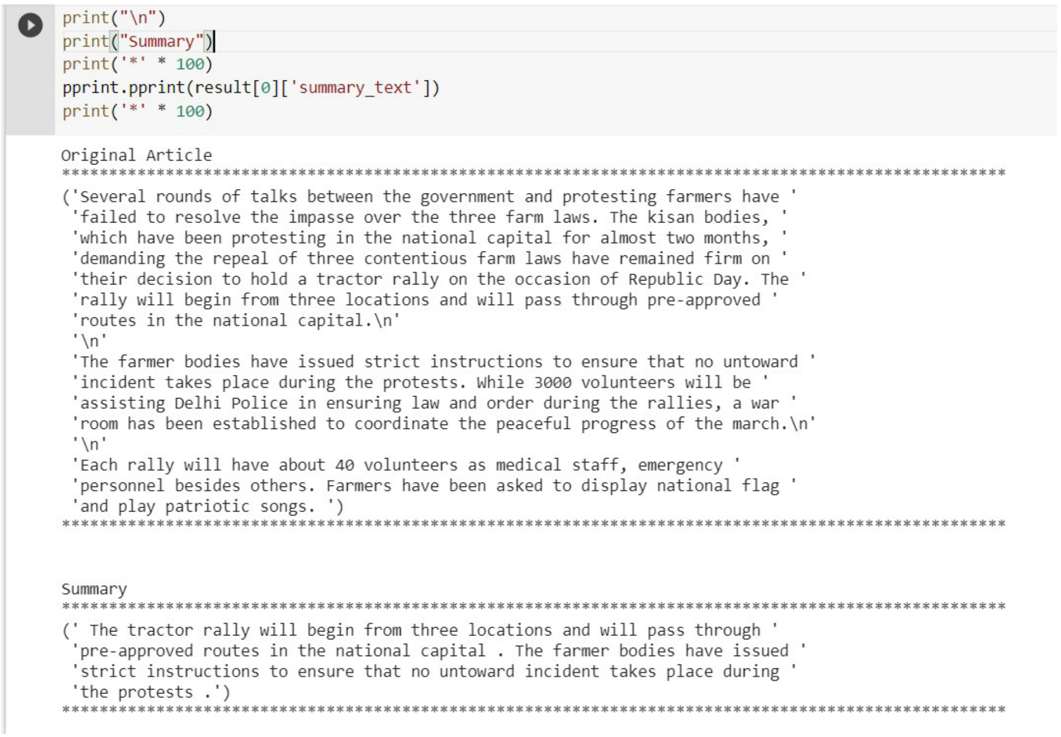
9.5.2 Using a pretrained transformer model

Now we will use a pretrained model to find the summarization. It is very simple to do with the transformers package from huggingface. The following is the code:

```
! pip install transformers

from transformers import pipeline


summarizer = pipeline("summarization")

result = summarizer(text, max_length=130, min_length=30, do_
sample=False)


print("Original Article")
print('*' * 100)
pprint.pprint(text)
print('*' * 100)
print("\n")
print("Summary")
print('*' * 100)
pprint.pprint(result[0]['summary_text'])
print('*' * 100)
```

We are using a **HuggingFace** pipeline object to create a summarization pipeline in line 3. Then we are just calling this pretrained model on our text. This uses a pretrained model that is trained on **CNN Dailymail** and **XSum** datasets. In our final example, we are going to fine tune this model further. This is a BART model.

Now let us look at the results.



```
print("\n")
print("Summary")
print('*' * 100)
pprint.pprint(result[0]['summary_text'])
print('*' * 100)
```

```
Original Article
****************************************************************************************************
('Several rounds of talks between the government and protesting farmers have '
 'failed to resolve the impasse over the three farm laws. The kisan bodies, '
 'which have been protesting in the national capital for almost two months, '
 'demanding the repeal of three contentious farm laws have remained firm on '
 'their decision to hold a tractor rally on the occasion of Republic Day. The '
 'rally will begin from three locations and will pass through pre-approved '
 'routes in the national capital.\n'
 '\n'
 'The farmer bodies have issued strict instructions to ensure that no untoward '
 'incident takes place during the protests. While 3000 volunteers will be '
 'assisting Delhi Police in ensuring law and order during the rallies, a war '
 'room has been established to coordinate the peaceful progress of the march.\n'
 '\n'
 'Each rally will have about 40 volunteers as medical staff, emergency '
 'personnel besides others. Farmers have been asked to display national flag '
 'and play patriotic songs. ')
****************************************************************************************************


Summary
****************************************************************************************************
(' The tractor rally will begin from three locations and will pass through '
 'pre-approved routes in the national capital . The farmer bodies have issued '
 'strict instructions to ensure that no untoward incident takes place during '
 'the protests .')
****************************************************************************************************
```

***Figure 9.17**: Summarization from pretrained transformer model (BART)*

Without any training, this also gives good summarization. As an additional task, the reader may use the csv we created and find the summarization for each example to calculate the rouge and bert score. We will calculate these metrics in our later fine tuning model, so it is good to compare models.

Now there are a lot more options to choose from for using the pretrained model. Can just put '??' right next to the pipeline function to check the model's parameters and try with different settings.

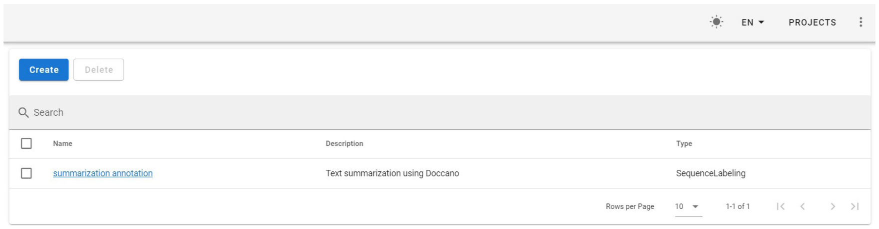
9.6 Fine tuning a model for summarization

Now let us fine tune the transformer model using the blurr package, here we will use the dataset we created using the web. The reader can also use some of the summarization datasets available with the datasets package from HuggingFace. Because the earlier scrapping we did doesn't have much data and headline is not a good summarization. To get good results, we require lot more data. So, you can also do the load more option and some manual annotation using Doccano (on the actual articles). So before fine tuning the model on the scrapped data, let us explore how to annotate data using Doccano; it is a tool that gives us a handy UI to annotate. This will give the reader idea of how to annotate and create datasets, and as an exercise, you can train the same model on the data you annotated.

9.6.1 Annotating data using Doccano

Doccano is one of the annotation tools that I like to use; it is user-friendly and gives an intuitive UI to annotate. Also, it provides an interface for shared annotation, meaning multiple people can collaborate on the same annotation project. So, you can set up a dataset with Doccano and involve your whole team to annotate. This way, the annotation will be faster. Also, with a huge dataset, even with tools like Doccano, it is impractical to annotate everything on your own. So, you can take the help of Dataturks from Amazon to annotate with a cost. We do this when the dataset size is so big it is impossible to annotate on our own.

We will use the doccano tool on our local computer. Now, using doccano is simple; install the package using pip, then type doccano in the project directory. It will create a server at localhost:8000 open the link and click on the getting started button; by default, there will be a user with *'admin,' 'password'* credentials, but you can create your superuser (Doccano is essentially a Django app, people familiar with Django can easily create the superuser but if not familiar with Django just read the documentation on Doccano). After login, you will land on a page that looks like the following:



Name	Description	Type
summarization annotation	Text summarization using Doccano	SequenceLabeling

Rows per Page 10 1-1 of 1

Figure 9.18*: Doccano: creating a project*

There won't be any projects for the first-time user, so you can click on create button, give the problem type as '*sequence labeling*.' Add your labels first; for our case, it will be the '*highlight*.' Part of the text we will highlight as the class.

Then click on the **`dataset`** option to load in a dataset by clicking actions/import dataset; now three ways you can do it. These are the options:

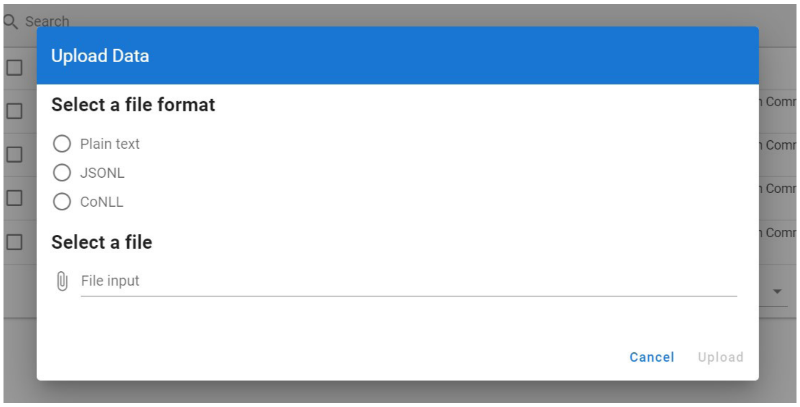
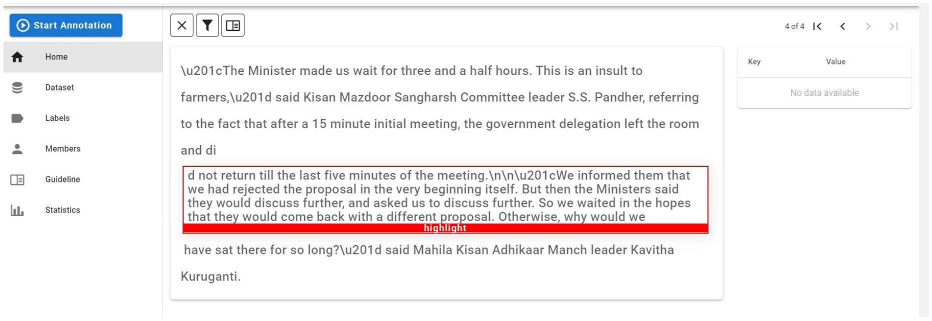


***Figure 9.19**: Importing data*

You can either load a plain text document containing all the news articles to be annotated separated by a new line or load in JSONL, CoNLL formats. I generally prefer JSONL formal because it looks like JSON, and here you can give pre trained label too. This is how I do annotation using a pretrained model:

1. First, I use any pretrained model (like we used) to create the extractive summarization.
2. Then I create the JSONL file by giving the labels that I got from the summarization.

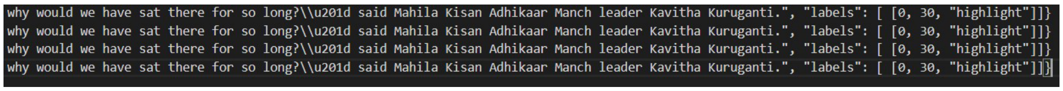
When I load in the dataset, it already contains some summarization that makes the annotator's life easier. For example, when you start annotating the data with docanno, it will look like the following:



\u201cThe Minister made us wait for three and a half hours. This is an insult to farmers,\u201d said Kisan Mazdoor Sangharsh Committee leader S.S. Pandher, referring to the fact that after a 15 minute initial meeting, the government delegation left the room and di d not return till the last five minutes of the meeting.\n\n\u201cWe informed them that we had rejected the proposal in the very beginning itself. But then the Ministers said they would discuss further, and asked us to discuss further. So we waited in the hopes that they would come back with a different proposal. Otherwise, why would we have sat there for so long?\u201d said Mahila Kisan Adhikaar Manch leader Kavitha Kuruganti.

***Figure 9.20**: An annotation example*

As you can see, it already comes with a highlight derived using the pre-trained model. After we annotate, we can export the dataset, and the data looks like this when we export.



```
why would we have sat there for so long?\\u201d said Mahila Kisan Adhikaar Manch leader Kavitha Kuruganti.", "labels": [ [0, 30, "highlight"]]}
why would we have sat there for so long?\\u201d said Mahila Kisan Adhikaar Manch leader Kavitha Kuruganti.", "labels": [ [0, 30, "highlight"]]}
why would we have sat there for so long?\\u201d said Mahila Kisan Adhikaar Manch leader Kavitha Kuruganti.", "labels": [ [0, 30, "highlight"]]}
why would we have sat there for so long?\\u201d said Mahila Kisan Adhikaar Manch leader Kavitha Kuruganti.", "labels": [ [0, 30, "highlight"]]}
```

***Figure 9.21**: The output format*

Now you can write a small function that takes this data and extracts the highlight (using the start and end position of the highlight), then creating a csv out of it in the desired format.

Doccano is a great package to explore for text annotation, and you can annotate classification, sequence labeling, and sequence to sequence tasks with it. We won't go into all the capabilities of Doccano here, but you can try it out for any task.

9.6.2 Using blurr to use transformer models in FastAI

Let us start building our model; we will use the Blurr package (!pip install ohmeow-blurr -q) for this task as blurr is a great wrapper around the HuggingFace transformer package to train with FastAI. As we used FastAI earlier, we know it is a great package for the training model with its differential learning rates, one cycle training, and so on.

Let us start with importing the packages required.

```
import pandas as pd
from fastai.text.all import *
from transformers import *


from blurr.data.all import *
from blurr.modeling.all import *
```

First, let us get our pretrained model; we will not start training from scratch; we will fine tune the BART model.

```
pretrained_model_name = "facebook/bart-large-cnn"
hf_arch, hf_config, hf_tokenizer, hf_model = BLURR_MODEL_HELPER.get_hf_
objects(pretrained_model_name,

model_cls=BartForConditionalGeneration)
```

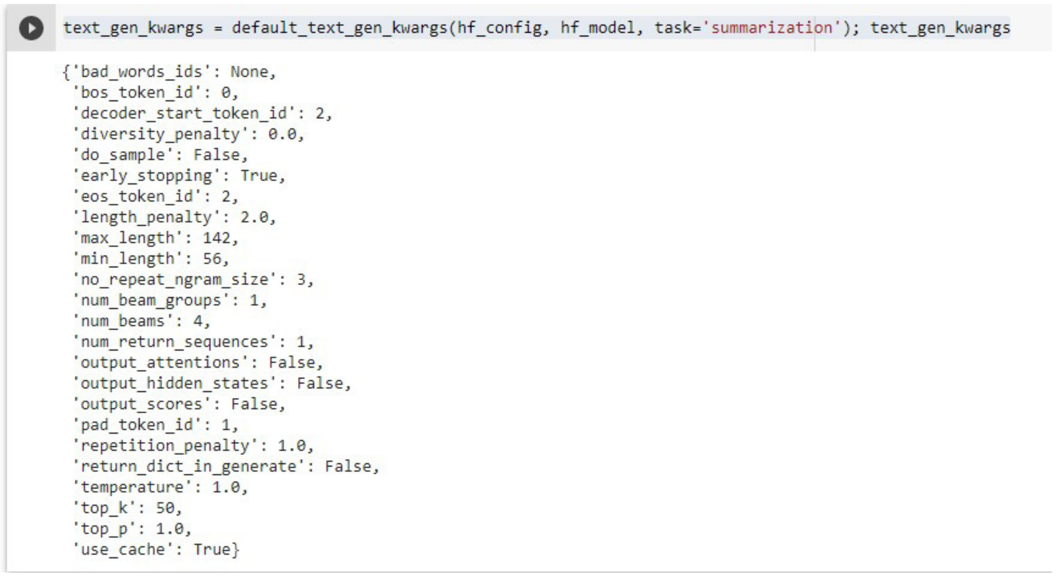
```
hf_arch, type(hf_config), type(hf_tokenizer), type(hf_model)
```

Here we use **`BLURR_MODEL_HELPER.get_hf_objects`** to get the architecture, configuration, tokenizer and model that we will use to train with FastAI.

Now let us get some arguments required for training.

```
text_gen_kwargs = default_text_gen_kwargs(hf_config, hf_model,
task='summarization'); text_gen_kwargs
```

The output to the preceding code:



```
text_gen_kwargs = default_text_gen_kwargs(hf_config, hf_model, task='summarization'); text_gen_kwargs

{'bad_words_ids': None,
 'bos_token_id': 0,
 'decoder_start_token_id': 2,
 'diversity_penalty': 0.0,
 'do_sample': False,
 'early_stopping': True,
 'eos_token_id': 2,
 'length_penalty': 2.0,
 'max_length': 142,
 'min_length': 56,
 'no_repeat_ngram_size': 3,
 'num_beam_groups': 1,
 'num_beams': 4,
 'num_return_sequences': 1,
 'output_attentions': False,
 'output_hidden_states': False,
 'output_scores': False,
 'pad_token_id': 1,
 'repetition_penalty': 1.0,
 'return_dict_in_generate': False,
 'temperature': 1.0,
 'top_k': 50,
 'top_p': 1.0,
 'use_cache': True}
```

***Figure 9.22**: Arguments of the BART model*

Now let us start loading our data.

We can simply upload it in Google Colab by clicking on files and then clicking upload.

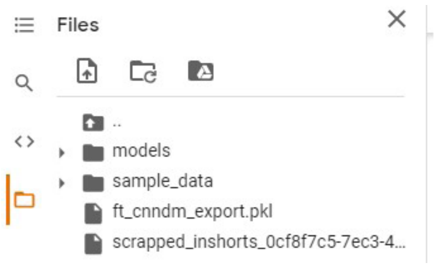


***Figure 9.23**: Importing scrapped data*

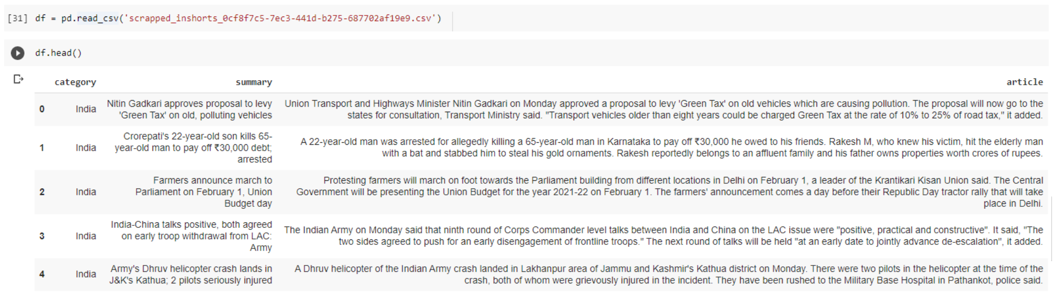
Once the file is uploaded to Colab, we can start reading it into a pandas dataframe.

The following is the code for the same:

```
df = pd.read_csv('scrapped_inshorts_0cf8f7c5-7ec3-441d-b275-
687702af19e9.csv')
```

```
df.head()
```

The output of the preceding code:



```
[31] df = pd.read_csv('scrapped_inshorts_0cf8f7c5-7ec3-441d-b275-687702af19e9.csv')
```

```
df.head()
```

	category	summary	article
0	India	Nitin Gadkari approves proposal to levy 'Green Tax' on old, polluting vehicles	Union Transport and Highways Minister Nitin Gadkari on Monday approved a proposal to levy 'Green Tax' on old vehicles which are causing pollution. The proposal will now go to the states for consultation, Transport Ministry said. "Transport vehicles older than eight years could be charged Green Tax at the rate of 10% to 25% of road tax," it added.
1	India	Crorepati's 22-year-old son kills 65-year-old man to pay off ₹30,000 debt; arrested	A 22-year-old man was arrested for allegedly killing a 65-year-old man in Karnataka to pay off ₹30,000 he owed to his friends. Rakesh M, who knew his victim, hit the elderly man with a bat and stabbed him to steal his gold ornaments. Rakesh reportedly belongs to an affluent family and his father owns properties worth crores of rupees.
2	India	Farmers announce march to Parliament on February 1, Union Budget day	Protesting farmers will march on foot towards the Parliament building from different locations in Delhi on February 1, a leader of the Krantikari Kisan Union said. The Central Government will be presenting the Union Budget for the year 2021-22 on February 1. The farmers' announcement comes a day before their Republic Day tractor rally that will take place in Delhi.
3	India	India-China talks positive, both agreed on early troop withdrawal from LAC: Army	The Indian Army on Monday said that ninth round of Corps Commander level talks between India and China on the LAC issue were "positive, practical and constructive". It said, "The two sides agreed to push for an early disengagement of frontline troops." The next round of talks will be held "at an early date to jointly advance de-escalation", it added.
4	India	Army's Dhruv helicopter crash lands in J&K's Kathua; 2 pilots seriously injured	A Dhruv helicopter of the Indian Army crash landed in Lakhanpur area of Jammu and Kashmir's Kathua district on Monday. There were two pilots in the helicopter at the time of the crash, both of whom were grievously injured in the incident. They have been rushed to the Military Base Hospital in Pathankot, police said.

Figure 9.24: *First look into data*

We have a summary column that contains the headlines of the article from inshorts that we collected using web scrapping. Now we will create the datablock for this; datablock is the blueprint of the data and covers how the data will be model consumable.

```
hf_batch_tfm = HF_Seq2SeqBeforeBatchTransform(hf_arch, hf_config, hf_
tokenizer, hf_model,

                                              max_length=256, max_tgt_
length=130, text_gen_kwargs=text_gen_kwargs)


blocks = (HF_Seq2SeqBlock(before_batch_tfm=hf_batch_tfm), noop)
```

```
dblock = DataBlock(blocks=blocks, get_x=ColReader('article'), get_
y=ColReader('summary'), splitter=RandomSplitter())
```

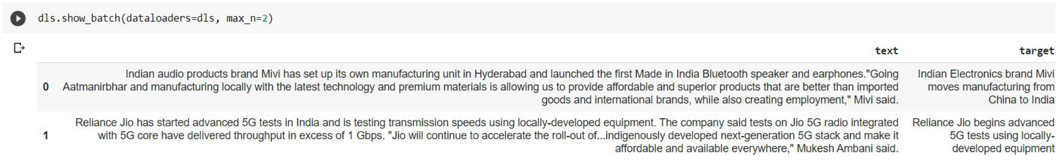
This is a sequence-to-sequence task, so we used **HF_Seq2SeqBeforeBatchTransform** from Blurr, then using **HF_Seq2SeqBlock**, finally we need to specify the x and y column, x is our feature column that is, article and y is the summary column. We are randomly splitting to create the datablock.

Now let us look into one of the batches.

```
dls = dblock.dataloaders(df, bs=2)
```

```
dls.show_batch(dataloaders=dls, max_n=2)
```

The output of the preceding code:



```
dls.show_batch(dataloaders=dls, max_n=2)
```

	text	target
0	Indian audio products brand Mivi has set up its own manufacturing unit in Hyderabad and launched the first Made in India Bluetooth speaker and earphones."Going Aatmanirbhar and manufacturing locally with the latest technology and premium materials is allowing us to provide affordable and superior products that are better than imported goods and international brands, while also creating employment," Mivi said.	Indian Electronics brand Mivi moves manufacturing from China to India
1	Reliance Jio has started advanced 5G tests in India and is testing transmission speeds using locally-developed equipment. The company said tests on Jio 5G radio integrated with 5G core have delivered throughput in excess of 1 Gbps. "Jio will continue to accelerate the roll-out of...indigenously developed next-generation 5G stack and make it affordable and available everywhere," Mukesh Ambani said.	Reliance Jio begins advanced 5G tests using locally-developed equipment

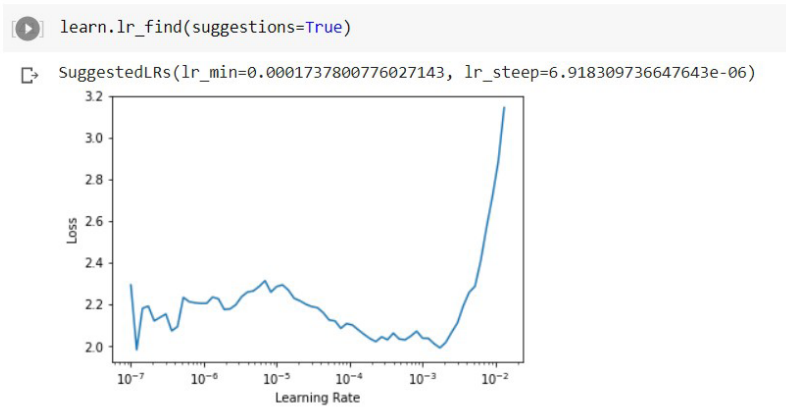
***Figure 9.25**: Data loader batch*

Now we can start training our model, and for metrics, we will use rouge score and bert score. And before training, we will find the learning rate using the LRFinder in FastAI; the following is the code:

```
seq2seq_metrics = {
        'rouge': {
            'compute_kwargs': { 'rouge_types': ["rouge1", "rouge2", 
"rougeL"], 'use_stemmer': True },
            'returns': ["rouge1", "rouge2", "rougeL"]
        },
        'bertscore': {
            'compute_kwargs': { 'lang': 'en' },
            'returns': ["precision", "recall", "f1"]
        }
    }


model = HF_BaseModelWrapper(hf_model)
learn_cbs = [HF_BaseModelCallback]
fit_cbs = [HF_Seq2SeqMetricsCallback(custom_metrics=seq2seq_metrics)]

learn = Learner(dls,
                model,
                opt_func=ranger,
                loss_func=CrossEntropyLossFlat(),
                cbs=learn_cbs,
                splitter=partial(seq2seq_splitter, arch=hf_arch)).to_
fp16()

learn.create_opt()
learn.freeze()
```

```
learn.lr_find(suggestions=True)
```

The output of the preceding code:



```
learn.lr_find(suggestions=True)
```

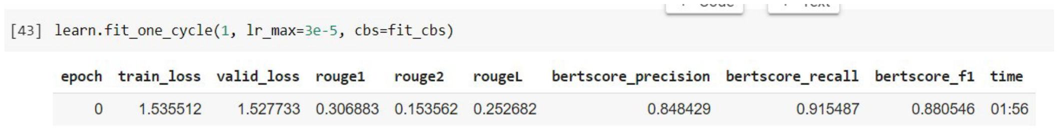
```
SuggestedLRs(lr_min=0.0001737800776027143, lr_steep=6.918309736647643e-06)
```

***Figure 9.26**: Learning Rate Finder*

The learning rate finder from FastAI works by gradually increasing the learning rate till the function diverges. The idea is to choose a learning rate that gives the maximum downward slope. Using this LR finder, we can see around 3e-5, the loss has the steepest slope. Now, let us start training our model.

```
learn.fit_one_cycle(1, lr_max=3e-5, cbs=fit_cbs)
```

The output of the preceding code:



```
[43] learn.fit_one_cycle(1, lr_max=3e-5, cbs=fit_cbs)
```

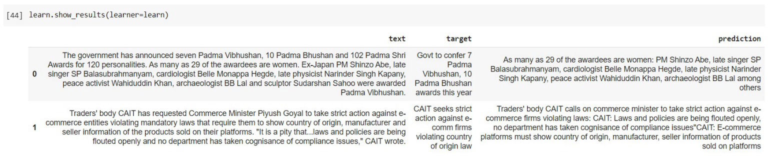
epoch	train_loss	valid_loss	rouge1	rouge2	rougeL	bertscore_precision	bertscore_recall	bertscore_f1	time
0	1.535512	1.527733	0.306883	0.153562	0.252682	0.848429	0.915487	0.880546	01:56

***Figure 9.27**: Training summarization model*

We can see the score as 88% F1, which is not bad for a single running cycle. Let us see some outputs.

```
learn.show_results(learner=learn)
```

The output of the preceding code:



```
[44] learn.show_results(learner=learn)
```

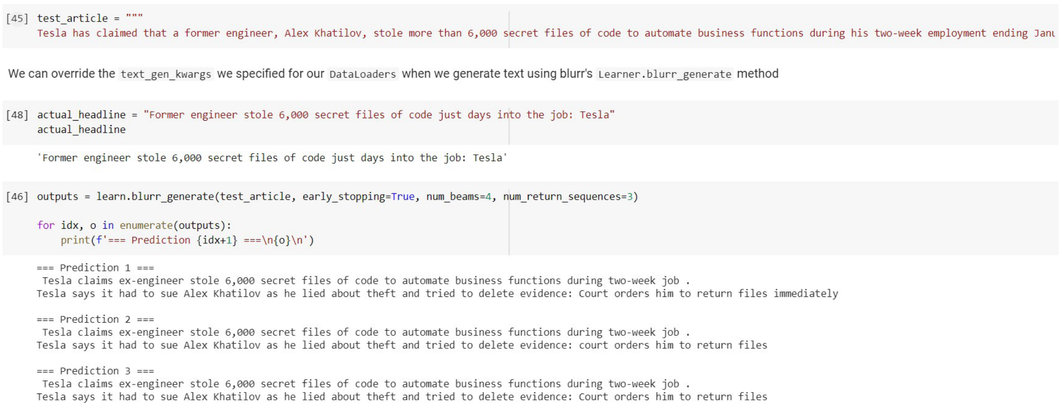
	text	target	prediction
0	The government has announced seven Padma Vibhushan, 10 Padma Bhushan and 102 Padma Shri Awards for 120 personalities. As many as 29 of the awardees are women. Ex-Japan PM Shinzo Abe, late singer SP Balasubrahmanyam, cardiologist Belle Monappa Hegde, late physicist Narinder Singh Kapany, peace activist Wahiduddin Khan, archaeologist BB Lal and sculptor Sudarshan Sahoo were awarded Padma Vibhushan.	Govt to confer 7 Padma Vibhushan, 10 Padma Bhushan awards this year	As many as 29 of the awardees are women: PM Shinzo Abe, late singer SP Balasubrahmanyam, cardiologist Belle Monappa Hegde, late physicist Narinder Singh Kapany, peace activist Wahiduddin Khan, archaeologist BB Lal among others
1	Traders' body CAIT has requested Commerce Minister Piyush Goyal to take strict action against e-commerce entities violating mandatory laws that require them to show country of origin, manufacturer and seller information of the products sold on their platforms. "It is a pity that...laws and policies are being flouted openly and no department has taken cognisance of compliance issues," CAIT wrote.	CAIT seeks strict action against e-comm firms violating country of origin law	Traders' body CAIT calls on commerce minister to take strict action against e-commerce firms violating laws: CAIT: Laws and policies are being flouted openly, no department has taken cognisance of compliance issues"CAIT: E-commerce platforms must show country of origin, manufacturer, seller information of products sold on platforms

***Figure 9.28**: Predictions from the model*

The summarizations look okay at this point; let us try with an example from the inshorts site on which our model is not trained.

```
test_article = """

Tesla has claimed that a former engineer, Alex Khatilov, stole more than 6,000 secret files of code to automate business functions during his two-week employment ending January 6. Tesla says it had to sue Khatilov as he lied about his theft and tried to delete its evidence. A US court has ordered him to immediately return all files to Tesla."""


actual_headline = "Former engineer stole 6,000 secret files of code just days into the job: Tesla"

actual_headline


outputs = learn.blurr_generate(test_article, early_stopping=True, num_beams=4, num_return_sequences=3)


for idx, o in enumerate(outputs):

    print(f'=== Prediction {idx+1} ===\n{o}\n')
```

The output of the preceding code:



```
[45] test_article = """
     Tesla has claimed that a former engineer, Alex Khatilov, stole more than 6,000 secret files of code to automate business functions during his two-week employment ending Janu
```

We can override the `text_gen_kwargs` we specified for our `DataLoaders` when we generate text using blurr's `Learner.blurr_generate` method

```
[48] actual_headline = "Former engineer stole 6,000 secret files of code just days into the job: Tesla"
     actual_headline

     'Former engineer stole 6,000 secret files of code just days into the job: Tesla'

[46] outputs = learn.blurr_generate(test_article, early_stopping=True, num_beams=4, num_return_sequences=3)

     for idx, o in enumerate(outputs):
         print(f'=== Prediction {idx+1} ===\n{o}\n')

     === Prediction 1 ===
      Tesla claims ex-engineer stole 6,000 secret files of code to automate business functions during two-week job .
     Tesla says it had to sue Alex Khatilov as he lied about theft and tried to delete evidence: Court orders him to return files immediately

     === Prediction 2 ===
      Tesla claims ex-engineer stole 6,000 secret files of code to automate business functions during two-week job .
     Tesla says it had to sue Alex Khatilov as he lied about theft and tried to delete evidence: court orders him to return files

     === Prediction 3 ===
      Tesla claims ex-engineer stole 6,000 secret files of code to automate business functions during two-week job .
     Tesla says it had to sue Alex Khatilov as he lied about theft and tried to delete evidence: Court orders him to return files
```

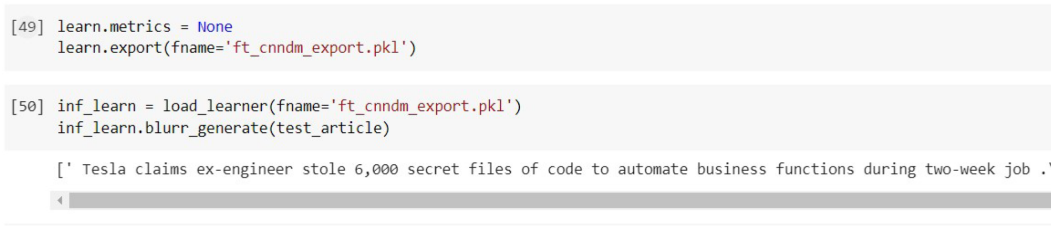
***Figure 9.29**: Predicted summaries*

The model learned well as it is giving good summarization even with unseen data. Now we can export our model as usual if we want to go for an inference API.

The following is the code:

```
learn.metrics = None

learn.export(fname='ft_cnndm_export.pkl')
```

The output of the preceding code:



```
[49] learn.metrics = None
     learn.export(fname='ft_cnndm_export.pkl')

[50] inf_learn = load_learner(fname='ft_cnndm_export.pkl')
     inf_learn.blurr_generate(test_article)

     [' Tesla claims ex-engineer stole 6,000 secret files of code to automate business functions during two-week job .'
```

***Figure 9.30**: Exporting and loading the model*

As an exercise, the reader can use the same tactics used in *chapters 7-8* to create a website that takes news articles and generates one-line summarization.

Conclusion

With this, we conclude *chapter 9* of the book. In this chapter, we learned about how to acquire and clean an NLP dataset. We then started learning some theory around word vector models, and we glanced through the Recurrent Neural Networks. We understood the vanishing and exploding gradient problems and gates that solve these for short sequences. We also understood the difficulty of training these models as they are sequential. We then learned the ground-breaking ideas of transformer architecture. We learned attention and multiheaded attention. We also learned how the transformer architecture accommodates the sequence information through positional encoding using sinusoids. After that, we learned the self-supervised pre-training tasks that make the models such as BERT, GPT, BART, etc., awesome. Finally, we started developing various summarization models for our news summary application. We tried unsupervised methods such as TextRank, and we use the pretrained BART model. In our last exercise, we fine-tuned a BART model using the BLURR library in FastAI. In our next chapter, we will learn about multi-input output models and the functional API.

Points to remember

The following are a few points to remember:

- Data cleaning is domain-specific and should be carefully done
- Word embeddings change the words into numbers such that they contain semantic information about the word
- Exploding gradient problems can be easily solved by gradient clipping, but vanishing gradient is a hard problem to solve
- Transformers retain positional information through positional encoding

MCQ

1. **Gradient clipping is a way to compensate _**
    a. Vanishing Gradient
    b. Overfitting
    c. Underfitting
    d. Exploding Gradient

2. **How does LSTMs/GRUs solve the vanishing gradient problem to an extent?**
    a. Introducing gates
    b. regularization
    c. LSTMs and GRUs reduce the no of layers
    d. All of the above

3. **Do transformers add positional information?**
    a. Adding positional encoding to the input embeddings
    b. Using a sequential model like RNN
    c. Adding positional embedding to input embeddings
    d. Both a and b
    e. Both a and c

4. **How to derive the base form of a word?**
    a. Stemming
    b. Deleting word
    c. Lemmatization
    d. Both a and c

Answers to MCQ

1. d
2. a
3. d
4. d

Questions

1. Create the inference engine for all the models created (Text Rank and trained BART model).
2. How to clean emails so that the final text does not contain salutations and signatures?
3. Visualize the attention layer outputs.
4. Write a model that can do sentiment analysis on the IMDB dataset.

Key Terms

- Self-Supervised pretraining
- Text rank
- Attention head
- Positional encoding

Chapter 10
Multiple Inputs and Multiple Output Models

In this chapter, we will learn about multiple input and multiple output models. When we build models in real life, we generally have mixed data types. For example, one of the projects that I worked on was making a classifier that decides if a customer request should be approved. The data also contained some initial reviewer comments along with customer financial demographics data. So, this was an example of mixed data type, where we have tabular data on one side (containing some continuous data such as salary, plan value, credit score, and categorical variables such as hometown, gender, and so on) and requester comments which is text data in another. And the decision will be based on both data combined. Also, some examples may contain data containing text and image both, and we will go through such an example. In some use cases, we will also have multiple outputs from a single model. We will understand how to walk through these scenarios in this chapter. We will use Tensorflow functional API to create **Multi Input and Multi Output** (**MIMO**) models (We will skip PyTorch in this discussion, but such capability is also present with PyTorch, and as an exercise, the reader may choose to go through PyTorch too.) We will also learn more about the flexibility of functional API and the features it provides, such as sharing layers. Then, we will investigate the mixed data type use case. We will learn to make a visual question answering model where we will create a model to ask questions to an image. Let us start this exciting chapter.

Structure

In this chapter, we will cover the following topics:

- Functional API for multiple inputs and multiple outputs
- Asking questions to an image: visual question answering
- Quick intro to model subclassing

Objective

After studying this chapter, you should be able to:

- Learn how to create models with multiple input and multiple output with Tensorflow Functional API.
- Understand how to build a model for mixed data type use cases such as image + text, image + tabular, and so on.
- How to write create custom layers using model subclassing.

10.1 Functional API for multiple inputs and multiple outputs

Functional API in TensorFlow can handle nonlinear topology, unlike **`tf.keras.Sequential`** API. And sometimes, we need nonlinear topology to solve our problem. As discussed in the introduction to this chapter, there are situations where we require multiple inputs and multiple outputs. The use cases that we will solve in this chapter also have a MIMO structure. The following is an image of linear versus nonlinear topology:

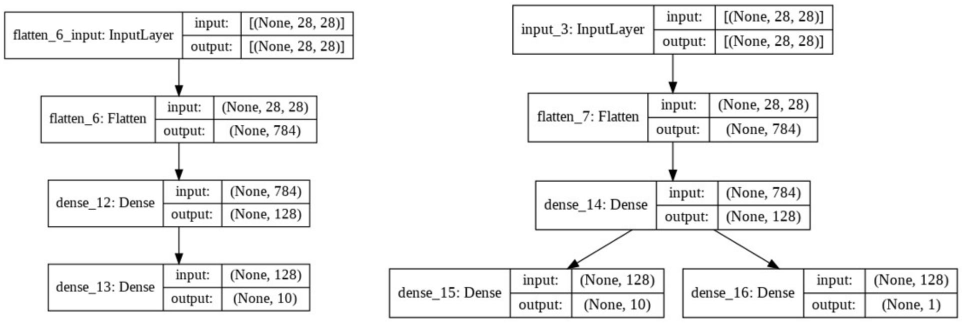


***Figure 10.1**: Linear vs. nonlinear topology*

As you can see, on the right side, we have a sequential model, a linear graph, but on the right side, we have two outputs. Also, with functional API, we can create much more complicated graphs such as given as follows:

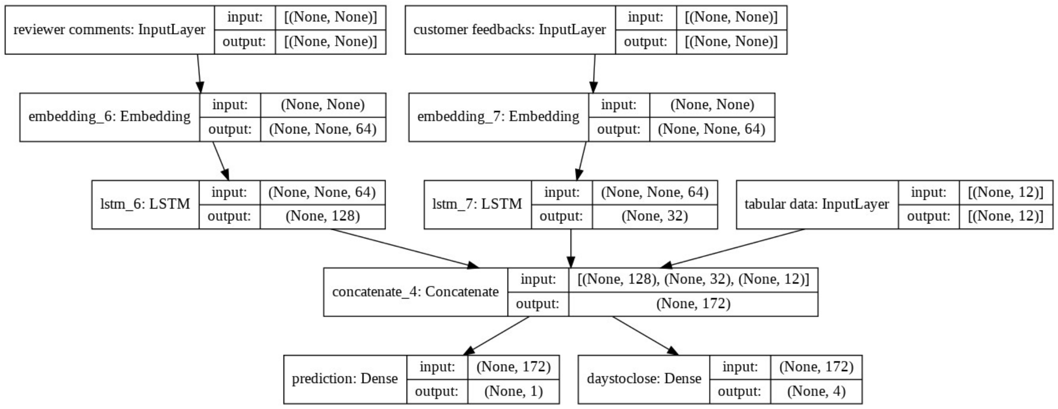

reviewer comments: InputLayer | input: [(None, None)] | output: [(None, None)]
customer feedbacks: InputLayer | input: [(None, None)] | output: [(None, None)]
embedding_6: Embedding | input: (None, None) | output: (None, None, 64)
embedding_7: Embedding | input: (None, None) | output: (None, None, 64)
lstm_6: LSTM | input: (None, None, 64) | output: (None, 128)
lstm_7: LSTM | input: (None, None, 64) | output: (None, 32)
tabular data: InputLayer | input: [(None, 12)] | output: [(None, 12)]
concatenate_4: Concatenate | input: [(None, 128), (None, 32), (None, 12)] | output: (None, 172)
prediction: Dense | input: (None, 172) | output: (None, 1)
daystoclose: Dense | input: (None, 172) | output: (None, 4)

***Figure 10.2**: Multiple input, multiple output model graph*

There are three inputs at different stages of the model in the preceding example and two outputs. These complicated topologies can be also created using functional API. Let us start with a simple example of MNIST and see the differences between functional and sequential API.

10.1.2 MNIST using functional API

We will train a sequential as well as a functional API model using MNIST data. MNIST is the dataset that contains 28 x 28 input images of handwritten images of numbers between 0-9. This dataset is often the starting point of any computer vision problem, so let us start building our model.

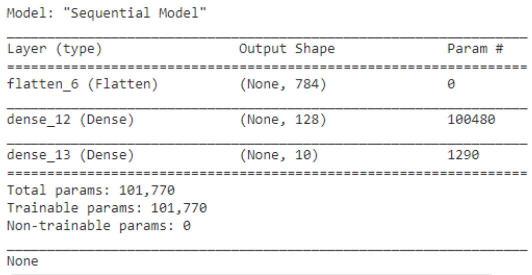
Importing necessary packages:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

Now let us write the sequential model:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10)
], name="Sequential_Model")

print(model.summary())
keras.utils.plot_model(model, "seq_model.png", show_shapes=True)
```

We are using the **`keras.models.Sequential`** API to create a sequential graph in line 1; we first flatten the input from (28, 28) 2D array to a 784-dimensional 1D array. Then we are using two dense layers to take the 784 values into 128 values in the first layer and to 10 values in the last layer. Because with MNIST, we have one of 10 numbers to predict, so the last layer is just the one hot encoded output. Let us check the output of this code.



```
Model: "Sequential Model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_6 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_12 (Dense)             (None, 128)               100480
_________________________________________________________________
dense_13 (Dense)             (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
None
```

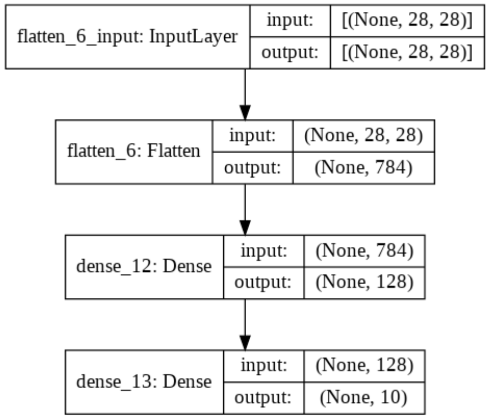


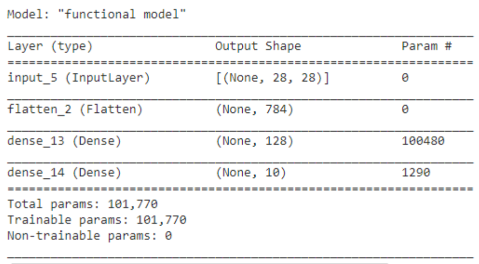
***Figure 10.3**: MNIST using sequential API*

The first image is the model summary that shows the layers and number of parameters in each layer. We don't have any parameters in the first layer as the flatten layer converts the 2D array into a 1D array. In the second layer (which is the hidden layer as per nomenclature), we have a weight matrix whose input is a 784-dimensional vector and output is 128 dimensional. So, there are *784 * 128 = 100352* weight parameters to train. Also, there are 128 bias terms, so *100352 + 128 = 100480* total parameters to train. Similarly, for the last layer, the parameters are mentioned. It is always a good practice to calculate the number of parameters by hand and cross-check with the parameters mentioned in the summary when you start with deep learning; this will create a mental model of shapes and flow.

We also used **`keras.utils.plot_model`** to create the graph of the model; this way of visualizing is much more intuitive. Often bugs present with the model creation can be caught during this visualization because when we create a model in code for a huge model, it is difficult to see how the overall model looks. Now let us do the same but with the functional API.

```
inputs = keras.Input(shape=(28, 28))
flatten = keras.layers.Flatten()
dense1 = keras.layers.Dense(128, activation='relu')
dense2 = keras.layers.Dense(10)

x = flatten(inputs)
x = dense1(x)
outputs = dense2(x)

model = keras.Model(inputs=inputs, outputs=outputs, name="functional_
model")

model.summary()
keras.utils.plot_model(model, "functional_model.png", show_shapes=True)
```

Functional API looks a bit more verbose than the sequential API; we must mention the input as **`keras.Input`** to the model with proper shape. The rest of the layers are the same as the preceding examples, but these are called separately. Once we have these layers defined, we can call them as if they are functions. From lines 7-10, we are simply passing our inputs through flatten, dense1, dense2 layers. And finally, in line 10, we can call **`keras.Model`** to create a model with the inputs and outputs given. Let us check the output of this code:



```
Model: "functional model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         [(None, 28, 28)]          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_13 (Dense)             (None, 128)               100480
_________________________________________________________________
dense_14 (Dense)             (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

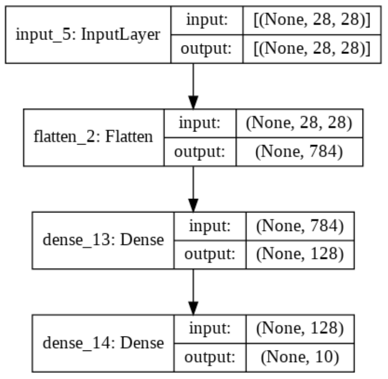


***Figure 10.4**: MNIST using functional API*

It looks the same; the only difference is between a summary of the two models; in the sequential model, the input layer is not mentioned but mentioned in the functional model.

Now let us start the training for the functional API; it is the same as a sequential API model.

```
 (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()


x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255


model.compile(
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=keras.optimizers.RMSprop(),
    metrics=["accuracy"],
)


history = model.fit(x_train, y_train, batch_size=64, epochs=2,
validation_split=0.2)


scores = model.evaluate(x_test, y_test, verbose=2)
print("Test loss:",  scores[0])
print("Test accuracy:", scores[1])
```

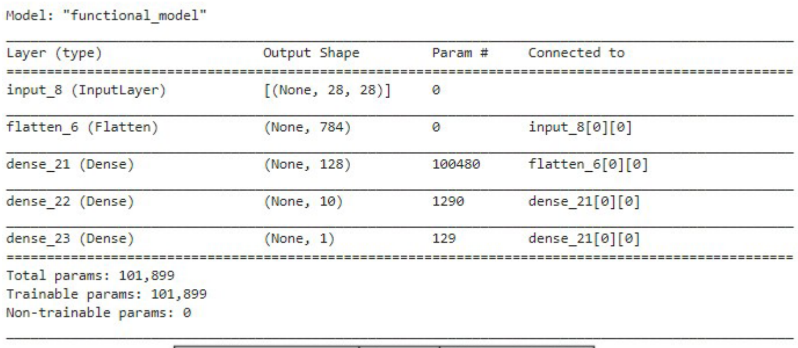
In line – 1, we load the MNIST dataset using **`keras.dataset`** and then divide the pixels by 255 to normalize it in lines 3-4. Then, we compile the model with a **`CategoricalCrossentropy`** and **`RMSProp`** optimizer, and the metric we want to see is the accuracy. And now we can simply do **`model.fit`** to train the model. We can treat the functional model exactly as a sequential model.

Now, MNIST might not be an example that shows the usability of the functional model but imagine we have two outputs from MNIST. The first one is as usual, which is the one-hot encoded output, and the second one something else. Then, in the functional model to incorporate this, we can simply add a new dense layer for output and just provide a list of outputs in the code.

```
inputs = keras.Input(shape=(28, 28))
flatten = keras.layers.Flatten()
dense1 = keras.layers.Dense(128, activation='relu')
```

```
dense2 = keras.layers.Dense(10)
dense3 = keras.layers.Dense(1)

x = flatten(inputs)
x = dense1(x)
outputs = dense2(x)
outputs_1 = dense3(x)

model = keras.Model(inputs=inputs, outputs=[outputs, outputs_1],
name="functional_model")

model.summary()
keras.utils.plot_model(model, "functional_model.png", show_shapes=True)
```

As you can see, the changes are in line 5 we added one more dense layer; then in line 10, we passed the x through that layer we created in line 5, then in line 12, we passed both the outputs as a list. That is it! We now created our first multi-output model. Let see the model summary and graph:



```
Model: "functional_model"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_8 (InputLayer)            [(None, 28, 28)]     0
__________________________________________________________________________________________________
flatten_6 (Flatten)             (None, 784)          0           input_8[0][0]
__________________________________________________________________________________________________
dense_21 (Dense)                (None, 128)          100480      flatten_6[0][0]
__________________________________________________________________________________________________
dense_22 (Dense)                (None, 10)           1290        dense_21[0][0]
__________________________________________________________________________________________________
dense_23 (Dense)                (None, 1)            129         dense_21[0][0]
==================================================================================================
Total params: 101,899
Trainable params: 101,899
Non-trainable params: 0
__________________________________________________________________________________________________
```

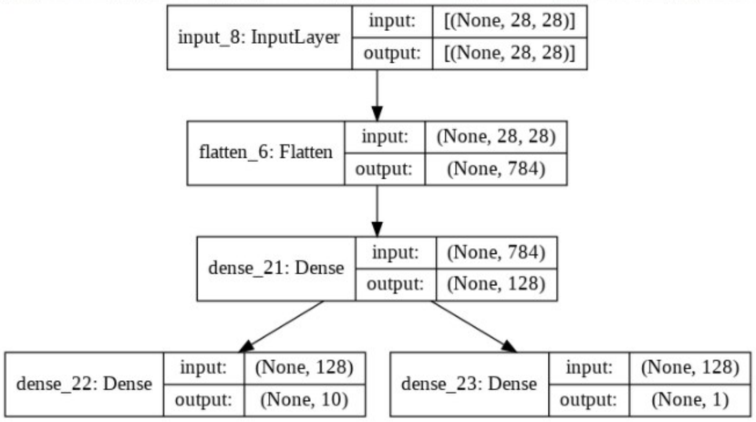
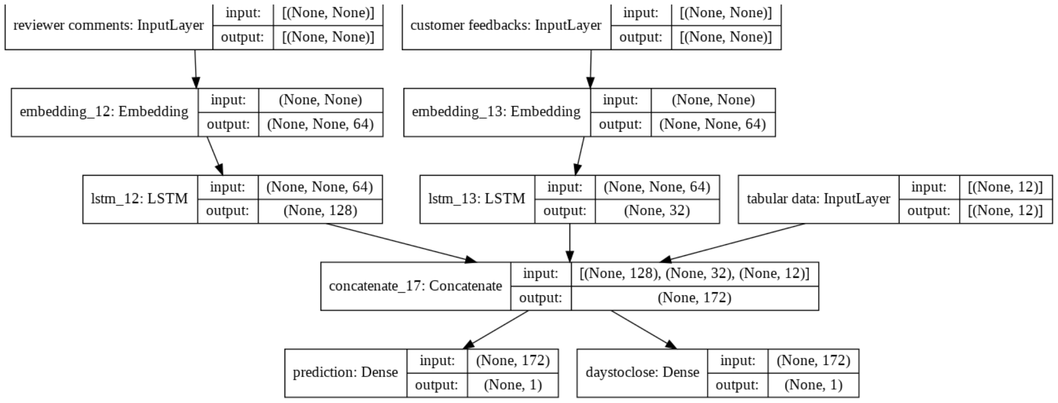


***Figure 10.5**: Adding another output to the MNIST functional model*

As you can see, we now have one more output. This is the advantage of functional API; it's very easy to define such complex architecture. Adding more than one input is also quite simple with functional API; the one I showed in *figure 10.2* is a model architecture I took from the Tensorflow documentation for multiple inputs and multiple output models and modified it to give different names per our requirement; here is the code:

```
n_columns = 12  # Number of tabular data columns
num_words = 10000  # Size of vocabulary

rc_input = keras.Input(
    shape=(None,), name="reviewer comments"
)
cf_input = keras.Input(shape=(None,), name="customer feedbacks")
tabular_input = keras.Input(
    shape=(n_columns,), name="tabular data"
)

rc_features = layers.Embedding(num_words, 64)(rc_input)
cf_features = layers.Embedding(num_words, 64)(cf_input)

rc_features = layers.LSTM(128)(rc_features)
cf_features = layers.LSTM(32)(cf_features)
x = layers.concatenate([rc_features, cf_features, tabular_input])

prediction = layers.Dense(1, name="prediction")(x)
daystoclose_pred = layers.Dense(1, name="daystoclose")(x)

model = keras.Model(
    inputs=[rc_input, cf_input, tabular_input],
    outputs=[prediction, daystoclose_pred],
)


tf.keras.utils.plot_model(model, "MIMO.png", show_shapes=True)
```

Here we have three inputs: one for reviewer comments, one for customer feedback, and a tabular input with 12 columns. In lines 12-13, we are creating 64-dimensional embeddings for both reviewer comments and customer feedback. In lines 5-16, we create two LSTM layers separately for these features, and finally, at line 17, we are concatenating all the inputs, that is, **`rc_features`**, **`cf_features,`** and **`tabular_input`**. Then we find prediction days to close in two different dense layers for regression. Now let us check the output of the code.



reviewer comments: InputLayer | input: [(None, None)] | output: [(None, None)]
customer feedbacks: InputLayer | input: [(None, None)] | output: [(None, None)]
embedding_12: Embedding | input: (None, None) | output: (None, None, 64)
embedding_13: Embedding | input: (None, None) | output: (None, None, 64)
lstm_12: LSTM | input: (None, None, 64) | output: (None, 128)
lstm_13: LSTM | input: (None, None, 64) | output: (None, 32)
tabular data: InputLayer | input: [(None, 12)] | output: [(None, 12)]
concatenate_17: Concatenate | input: [(None, 128), (None, 32), (None, 12)] | output: (None, 172)
prediction: Dense | input: (None, 172) | output: (None, 1)
daystoclose: Dense | input: (None, 172) | output: (None, 1)

***Figure 10.6**: MIMO model with functional API*

10.1.3 Shared layers

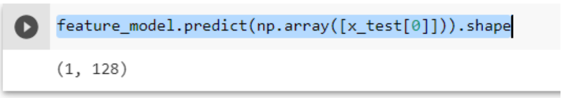
There are other features with a functional model, and one of the key features is called **shared layers**. We used shared layers to create a functional model in *Chapter 8: Creating a custom image classifier from scratch*, which was the reverse image search engine. We used the VGG16 model FC1 layer to get the output from that layer. For example, in the model we created for MNIST using the functional model, if we want to get the model outputs at the first dense layer (which takes data from 784 dimensions to 128 dimensions), we can also do the code.

```
feature_model = keras.Model(inputs=model.inputs, outputs=model.get_
layer('dense_62').output) # here the dense_62 name will be different for
you, you can find the name(auto-assigned, but you can also name it) in
model.summary().
```

Here I use the name **`dense_62`** layer from our model in output. The layer name is generated from TensorFlow, but if we want to use shared layers, it's a good practice to name the layers according to our need so that the names will be meaningful (in the case of VGG16, we used the fc1 layer), now if we predict the input using this **`feature_model,`** we can get the 128-dimensional feature output.

```
feature_model.predict(np.array([x_test[0]])).shape
```

The output of the code is as follows:



```
feature_model.predict(np.array([x_test[0]])).shape

(1, 128)
```

***Figure 10.7**: Getting feature output*

As we can see, the shape of this prediction is 128 dimensional, which is what we want; in our reverse image search problem, we had the 4096-dimensional output from the fc1 layer. Having a way to get the intermediate feature output is very handy.

We take two images, one content and another one a style image, and apply that style to the content image for neural style transfer. And in such problems, we require layer features. As an exercise, try to create a neural style transfer model using shared layers.

Another way to create nonlinear and complex models is through sub-classing, which we will learn in the book's final section. Still, whenever possible, we should go for functional API rather than model subclassing because it is less verbose hence introduces fewer errors. We will investigate model subclassing in the final section of this chapter.

10.2 Asking questions to an image: visual question answering

With the functional API at our disposal, let us now start exploring the mixed data type example. We will train a model which can answer questions about an image. It is a hard task as we have both image and text data. We won't use any pre-trained model or shared layers because we want to create a model from scratch with functional API. So, we will take an easy dataset for VQA that is an easy-VQA dataset which can be found at **https://github.com/vzhou842/easy-VQA**; this contains 4000 train images and 1000 test images with 38575 train questions and 9673 test questions. There are 13 possible answers. The images are simple geometric shapes, and the questions are not that complicated, but it is a good starting point. After this use case, the reader can try to fit the model to a much complex dataset, the actual VQA dataset (**https://visualqa.org/**), and it is also advised to create a much better model with pretrained weights, as we did with the reverse image search engine. Now let us start looking into the data.

10.2.1 Quick look into the dataset

This data contains geometrical shapes with different colors, and questions are about color or shape only, let us start with loading few packages.

```
import numpy as np

import tensorflow

from tensorflow import keras

import matplotlib.pyplot as plt

import matplotlib.image as mpimg
```

Now let us load the images from an easy-vqa dataset; the package comes with handy functions to do this.

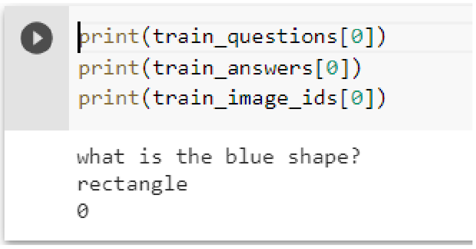
```
! pip install easy_vqa
from easy_vqa import get_train_questions, get_test_questions, get_train_
image_paths, get_test_image_paths


train_questions, train_answers, train_image_ids = get_train_questions()
test_questions, test_answers, test_image_ids = get_test_questions()
```

Line 3-4, we load the train/test questions, answers, and respective images.

Now let us investigate one of the data points.

```
print(train_questions[0])
print(train_answers[0])
print(train_image_ids[0])
```

The output of this code:



```
print(train_questions[0])
print(train_answers[0])
print(train_image_ids[0])
```

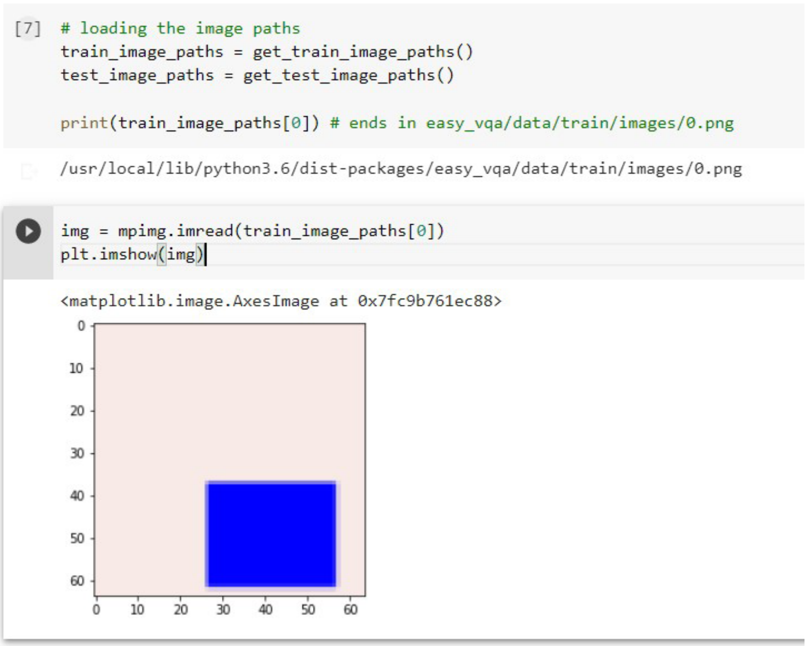
```
what is the blue shape?
rectangle
0
```

***Figure 10.8**: First look into the data*

So, the first question is "*What is the blue shape?*" and the answer is "*rectangle,*" Let us plot the image to confirm.

```
# loading the image paths
train_image_paths = get_train_image_paths()
test_image_paths = get_test_image_paths()


print(train_image_paths[0])


img = mpimg.imread(train_image_paths[0])
plt.imshow(img)
```

To load the image paths, we are calling the **get_train_image_paths** function given by the library. Then using **matplotlib,** we are reading and displaying the image output to this code:



```
[7] # loading the image paths
    train_image_paths = get_train_image_paths()
    test_image_paths = get_test_image_paths()

    print(train_image_paths[0]) # ends in easy_vqa/data/train/images/0.png

    /usr/local/lib/python3.6/dist-packages/easy_vqa/data/train/images/0.png

    img = mpimg.imread(train_image_paths[0])
    plt.imshow(img)

    <matplotlib.image.AxesImage at 0x7fc9b761ec88>
```

Figure 10.9: Plotting the image data in easy-VQA

And as expected, the image is a rectangular blue shape. Let us see what other kinds of images are present in the library; we can create a function that randomly plots a batch of images; let us check the code:

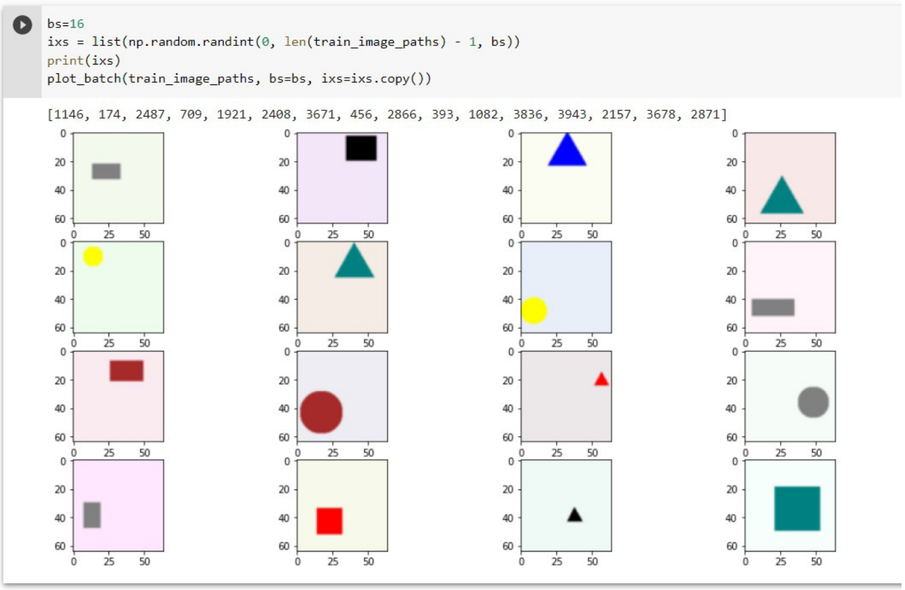
```
%pylab inline
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.image as mpimg
np.random.seed(4)

def plot_batch(from_paths, size=(16, 8), bs=16, ixs=[]):
    assert int(np.sqrt(bs)) ** 2 == bs, "Batch size must be a perfect 
square"
    assert bs <= len(from_paths), "Batch size is greater than total 
images"
    fig, ax = plt.subplots(int(np.sqrt(bs)), int(np.sqrt(bs)), 
figsize=size)
```

```
    if not ixs: ixs = list(np.random.randint(0, len(from_paths) - 1,
bs))

    for i in range(int(np.sqrt(bs))):

        for j in range(int(np.sqrt(bs))):

            ip = from_paths[ixs.pop(0)]

            img = mpimg.imread(ip)

            ax[i][j].imshow(img)
```

We will create a square grid, so the batch size must be a perfect square. That is what line 8 is assuring; similarly, in line 9, we check if the batch size is less than the total images or not because it is impossible to create a batch size greater than the number of images. Such checks are crucial in a good maintainable codebase. In line 10, we will use subplots of matplotlib to create a grid whose rows and columns are the equal and square root of batch size (because we are creating a square grid). Then in lines 11-16, we are randomly creating few indexes if not provided by the user already, and using imread, imshow to load and show the images in the proper axis location (ax[i][j]). Let us try to create a 4x4 grid of images with this code.

```
bs=16

ixs = list(np.random.randint(0, len(train_image_paths) - 1, bs))

print(ixs)

plot_batch(train_image_paths, bs=bs, ixs=ixs.copy())
```

The output of the preceding code:



```
bs=16
ixs = list(np.random.randint(0, len(train_image_paths) - 1, bs))
print(ixs)
plot_batch(train_image_paths, bs=bs, ixs=ixs.copy())
```

```
[1146, 174, 2487, 709, 1921, 2408, 3671, 456, 2866, 393, 1082, 3836, 3943, 2157, 3678, 2871]
```

Figure 10.10: Plotting a batch of data

This shows some of the images in our data; we can run this code multiple times, and each time the images will be different as the indexes are randomly generated.

Now let us pre-process the images; only pre-processing required will be to divide each pixel by 255. We will also scale the images between -0.5 to 0.5; here are the helper functions.

```
## processing images
from easy_vqa import get_train_image_paths, get_test_image_paths
from tensorflow.keras.preprocessing.image import load_img, img_to_array


def load_and_proccess_image(image_path):
  im = img_to_array(load_img(image_path))
  return im / 255 - 0.5


def read_images(paths):
  ims = {}
  for image_id, image_path in paths.items():
    ims[image_id] = load_and_proccess_image(image_path)
  return ims
```

The code is pretty much self-explanatory; we are just processing the images by dividing each pixel by 255 and shifting the values between -0.5 to 0.5.

Now let us pre-process the training and test image data.

```
train_images = read_images(get_train_image_paths())
test_images = read_images(get_test_image_paths())


train_X_imgs = np.array([train_images[id] for id in train_image_ids])
test_X_imgs = np.array([test_images[id] for id in test_image_ids])
```

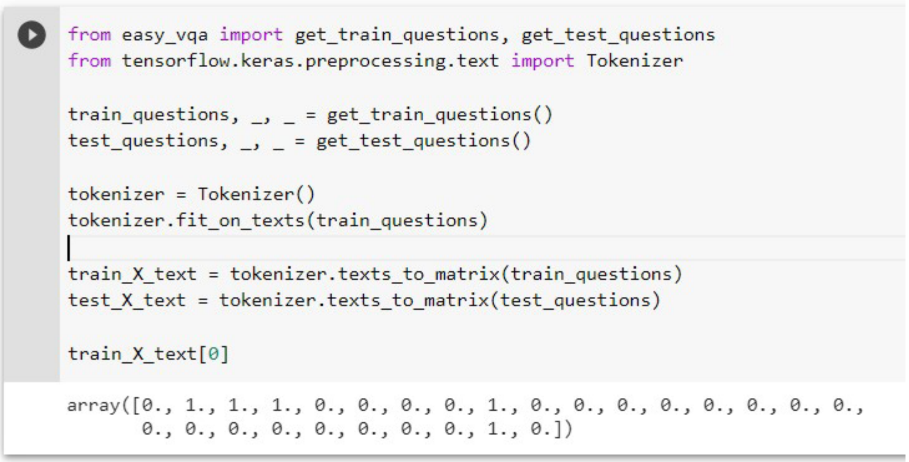
With this, we are done with the image part; now, let us explore the text part of the data.

10.2.2 Text data manipulation

We have already loaded the question data, now; let us process this data; we will create a bag of words representing the text data. This means for every vocabulary word; we will create one hot encoded vector in the question. The following is the code for the same:

```
from easy_vqa import get_train_questions, get_test_questions
from tensorflow.keras.preprocessing.text import Tokenizer

train_questions, _, _ = get_train_questions()
test_questions, _, _ = get_test_questions()

tokenizer = Tokenizer()
tokenizer.fit_on_texts(train_questions)

train_X_text = tokenizer.texts_to_matrix(train_questions)
test_X_text = tokenizer.texts_to_matrix(test_questions)

train_X_text[0]
```

The output of this code:



```
from easy_vqa import get_train_questions, get_test_questions
from tensorflow.keras.preprocessing.text import Tokenizer

train_questions, _, _ = get_train_questions()
test_questions, _, _ = get_test_questions()

tokenizer = Tokenizer()
tokenizer.fit_on_texts(train_questions)

train_X_text = tokenizer.texts_to_matrix(train_questions)
test_X_text = tokenizer.texts_to_matrix(test_questions)

train_X_text[0]
```

```
array([0., 1., 1., 1., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 1., 0.])
```

***Figure 10.11**: One hot encoded answer data*

Here the bag of words model is fine as the vocabulary size is quite less, i.e., 27, but for a better model, this bag of words model is a bad idea. But for now, we will use it, but I would encourage the reader to create a better embedding for the same. We can also use Word2Vec embeddings for the words and create an average vector for each question with 300 dimensions. These are the experiments that you will do when you create real world models. There are a lot of different things we can try.

Now coming to the answers data, these should be one hot encoded; let us do that.

```
from easy_vqa import get_answers
all_answers = get_answers()
```

```
print(all_answers)

from tensorflow.keras.utils import to_categorical
train_answer_indices = [all_answers.index(a) for a in train_answers]
test_answer_indices = [all_answers.index(a) for a in test_answers]
train_Y = to_categorical(train_answer_indices)
test_Y = to_categorical(test_answer_indices)
```

We use **`to_categorical`** from keras to create categorical values for the answers.

With this, we are done with preparing the data; now, let us define and train the data.

10.2.3 Training and prediction

First, let us build the image and text models separately; for the image model, we will use a simple CNN network, again the model architecture is not optimal at this point. This is a starter example so that we will be creating this model; a reader should try different models for complex data; the reader can also use pre-trained layer output as we did in *Chapter 8: Creating a Custom Image Classifier from Scratch* reverse image search engine. There we used VGG16 fc1 output data, but for now, let us build a model from scratch.

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Conv2D, MaxPooling2D,
Flatten, Multiply
from tensorflow.keras.optimizers import Adam

def build_image_model(input_shape):
    # building the CNN here
    image_input = Input(shape=input_shape, name="input")

    # Convolution Blocks (Conv2D --> MaxPool)
    x = Conv2D(8, 3, padding='same')(image_input)
    x = MaxPooling2D()(x)
    x = Conv2D(16, 3, padding='same')(x)
    x = MaxPooling2D()(x)
    x = Conv2D(32, 3, padding='same')(x)
    x = MaxPooling2D()(x)
        # Flatten and Activate
    x = Flatten()(x)
    x = Dense(32, activation='tanh')(x)

    cnn = Model(image_input, x)
    return cnn
```

The model has three convolutions followed by maxpool, and finally, a flattened and dense layer to get a 32-dimensional representation of the image, nothing fancy here.

Let us now build the text model.

```
def build_text_model(vocab_size):
    question_input = Input(shape=(vocab_size,))
    x = Dense(32, activation='tanh')(question_input)
    x = Dense(32, activation='tanh')(x)
    question_model = Model(question_input, x)
    return question_model
```

It is also a simple model that takes the question input (size is the same as **`vocab_size,`** which is the length of the total vocabulary, that is, 27); we then pass it through two fully connected layers with **`tanh`** activations to get our model.

With this, we can now combine these two functional models easily.

```
def build_vqa_model(image_shape, vocab_size, num_answers):
    text = build_text_model(vocab_size)
    image = build_image_model(image_shape)

    out = Concatenate()([text.output, image.output])
    out = Dense(32, activation='tanh')(out)
    out = Dense(num_answers, activation='softmax')(out)

    model = Model(inputs=[image.input, text.input], outputs=out)
    model.compile(Adam(lr=5e-4), loss='categorical_crossentropy',
metrics=['accuracy'])

    return model
```

We get the text and image output from the models defined through **`.output`**, because these are functional models, then we combine it using **`Concatenate()`**, we could use any other merging methods such as **`Multiply()`**, **`Add()`**, **`Subtract()`**, and **`Average()`**. These are the layers that can combine the activations.

Now let us build our model and look into the graph.

```
height = 64
width = 64
channels = 3
IMAGE_SHAPE = (height, width, channels)
VOCAB_SIZE = len(tokenizer.word_index) + 1
NUM_ANSWERS = len(all_answers)
print(IMAGE_SHAPE, VOCAB_SIZE, NUM_ANSWERS)

model = build_vqa_model(IMAGE_SHAPE, VOCAB_SIZE, NUM_ANSWERS)

keras.utils.plot_model(model, "VQA_Model.png", show_shapes=True)
```

The output of this code.

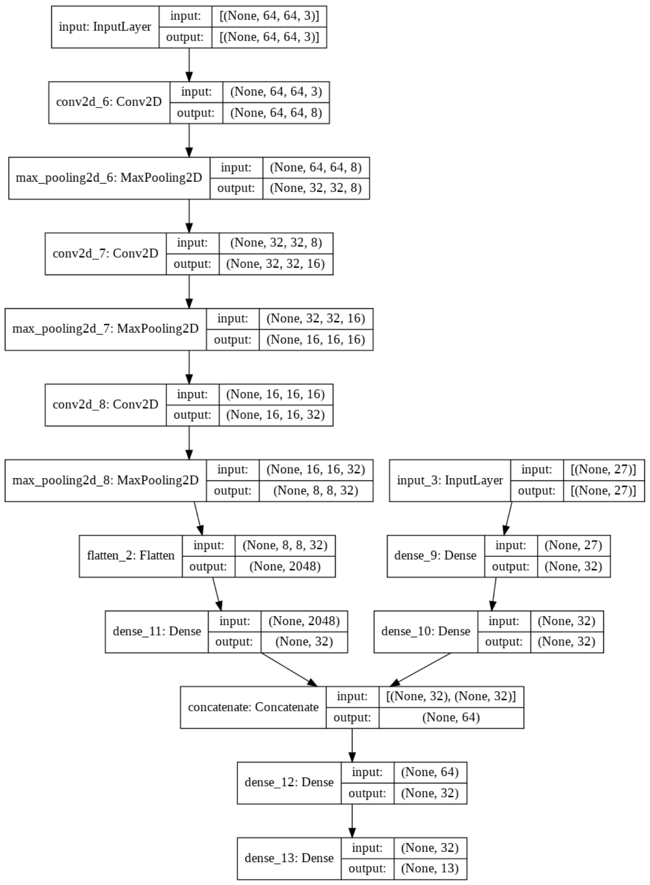

input: InputLayer | input: [(None, 64, 64, 3)] | output: [(None, 64, 64, 3)]
conv2d_6: Conv2D | input: (None, 64, 64, 3) | output: (None, 64, 64, 8)
max_pooling2d_6: MaxPooling2D | input: (None, 64, 64, 8) | output: (None, 32, 32, 8)
conv2d_7: Conv2D | input: (None, 32, 32, 8) | output: (None, 32, 32, 16)
max_pooling2d_7: MaxPooling2D | input: (None, 32, 32, 16) | output: (None, 16, 16, 16)
conv2d_8: Conv2D | input: (None, 16, 16, 16) | output: (None, 16, 16, 32)
max_pooling2d_8: MaxPooling2D | input: (None, 16, 16, 32) | output: (None, 8, 8, 32)
input_3: InputLayer | input: [(None, 27)] | output: [(None, 27)]
flatten_2: Flatten | input: (None, 8, 8, 32) | output: (None, 2048)
dense_9: Dense | input: (None, 27) | output: (None, 32)
dense_11: Dense | input: (None, 2048) | output: (None, 32)
dense_10: Dense | input: (None, 32) | output: (None, 32)
concatenate: Concatenate | input: [(None, 32), (None, 32)] | output: (None, 64)
dense_12: Dense | input: (None, 64) | output: (None, 32)
dense_13: Dense | input: (None, 32) | output: (None, 13)

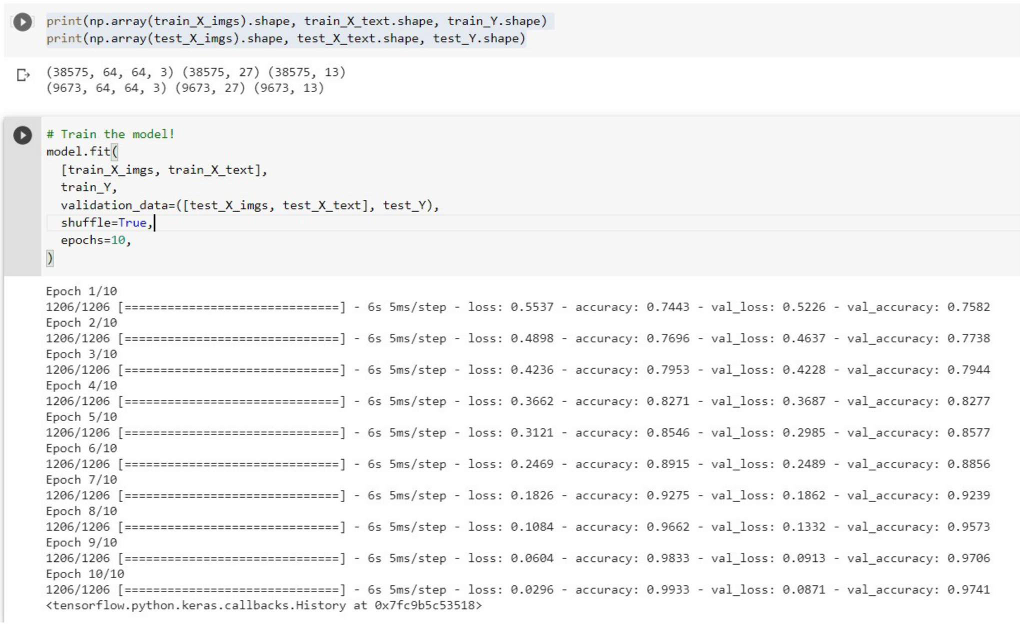
***Figure 10.12**: Final VQA model*

The model may look huge, but it is quite simple; we have an image input and a text input; we run CNN on image and simple multi layered perceptron on text; finally, we concatenate and run it through two more fully connected layers (dense) to get our desired output.

Now let us train the model, as discussed we can train a model using the **`.fit`** method:

```
print(np.array(train_X_imgs).shape, train_X_text.shape, train_Y.shape)
print(np.array(test_X_imgs).shape, test_X_text.shape, test_Y.shape)


# Train the model!
model.fit(
  [train_X_imgs, train_X_text],
  train_Y,
  validation_data=([test_X_imgs, test_X_text], test_Y),
  shuffle=True,
  epochs=10,
)
```

It is always good to check the input shapes before running the model, so we do that in lines 1-2, and in lines 5-10, we are training the model using the preceding data we created. The following is the output of the code:



```
print(np.array(train_X_imgs).shape, train_X_text.shape, train_Y.shape)
print(np.array(test_X_imgs).shape, test_X_text.shape, test_Y.shape)

(38575, 64, 64, 3) (38575, 27) (38575, 13)
(9673, 64, 64, 3) (9673, 27) (9673, 13)

# Train the model!
model.fit(
  [train_X_imgs, train_X_text],
  train_Y,
  validation_data=([test_X_imgs, test_X_text], test_Y),
  shuffle=True,
  epochs=10,
)

Epoch 1/10
1206/1206 [==============================] - 6s 5ms/step - loss: 0.5537 - accuracy: 0.7443 - val_loss: 0.5226 - val_accuracy: 0.7582
Epoch 2/10
1206/1206 [==============================] - 6s 5ms/step - loss: 0.4898 - accuracy: 0.7696 - val_loss: 0.4637 - val_accuracy: 0.7738
Epoch 3/10
1206/1206 [==============================] - 6s 5ms/step - loss: 0.4236 - accuracy: 0.7953 - val_loss: 0.4228 - val_accuracy: 0.7944
Epoch 4/10
1206/1206 [==============================] - 6s 5ms/step - loss: 0.3662 - accuracy: 0.8271 - val_loss: 0.3687 - val_accuracy: 0.8277
Epoch 5/10
1206/1206 [==============================] - 6s 5ms/step - loss: 0.3121 - accuracy: 0.8546 - val_loss: 0.2985 - val_accuracy: 0.8577
Epoch 6/10
1206/1206 [==============================] - 6s 5ms/step - loss: 0.2469 - accuracy: 0.8915 - val_loss: 0.2489 - val_accuracy: 0.8856
Epoch 7/10
1206/1206 [==============================] - 6s 5ms/step - loss: 0.1826 - accuracy: 0.9275 - val_loss: 0.1862 - val_accuracy: 0.9239
Epoch 8/10
1206/1206 [==============================] - 6s 5ms/step - loss: 0.1084 - accuracy: 0.9662 - val_loss: 0.1332 - val_accuracy: 0.9573
Epoch 9/10
1206/1206 [==============================] - 6s 5ms/step - loss: 0.0604 - accuracy: 0.9833 - val_loss: 0.0913 - val_accuracy: 0.9706
Epoch 10/10
1206/1206 [==============================] - 6s 5ms/step - loss: 0.0296 - accuracy: 0.9933 - val_loss: 0.0871 - val_accuracy: 0.9741
<tensorflow.python.keras.callbacks.History at 0x7fc9b5c53518>
```

***Figure 10.13**: Training the VQA model*

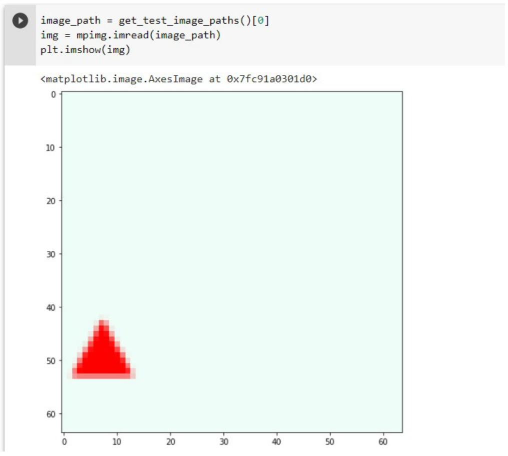
We are getting a good accuracy of 97% without trying any regularization tricks. We have not used any learning rate scheduling or differential learning rates, which the reader should try. The toy models are good to train as they are faster.

Now let us write some code to ask questions to an image.

```
image_path = get_test_image_paths()[0]
img = mpimg.imread(image_path)
plt.imshow(img)

x_text = tokenizer.texts_to_matrix([test_questions[0]])

def process_input(image_path, question):
    x_img = np.array([load_and_proccess_image(image_path)])
    x_text = tokenizer.texts_to_matrix([question])

    return [x_img, x_text]

inf_input = process_input(image_path, "is this a circle?")
pred = model.predict(inf_input)
all_answers[np.argmax(pred)]
```

Here I am using one of the test images to ask a question. We need to use the same tokenizer to create the text embeddings; we also need to use the same pre-process image. After that, we can predict. And using **`np.argmax,`** we can get the index of the highest probability output.

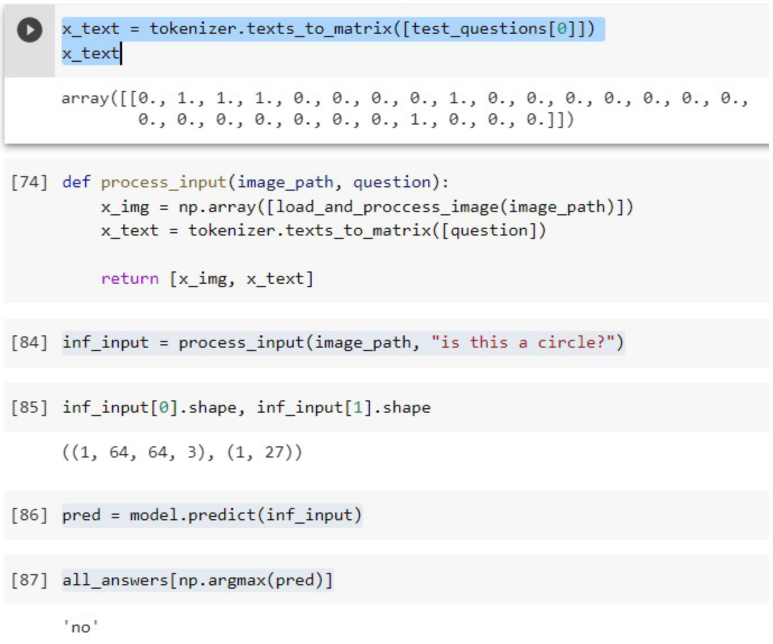
First, let us check the output of the plotting.



```
image_path = get_test_image_paths()[0]
img = mpimg.imread(image_path)
plt.imshow(img)

<matplotlib.image.AxesImage at 0x7fc91a0301d0>
```

Figure 10.14: Prediction input

It is a red triangle; now, let us see the output of our prediction.



```
x_text = tokenizer.texts_to_matrix([test_questions[0]])
x_text

array([[0., 1., 1., 1., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.]])

[74] def process_input(image_path, question):
         x_img = np.array([load_and_proccess_image(image_path)])
         x_text = tokenizer.texts_to_matrix([question])

         return [x_img, x_text]

[84] inf_input = process_input(image_path, "is this a circle?")

[85] inf_input[0].shape, inf_input[1].shape

     ((1, 64, 64, 3), (1, 27))

[86] pred = model.predict(inf_input)

[87] all_answers[np.argmax(pred)]

     'no'
```

***Figure 10.15**: Prediction output*

We asked the model if this is a circle, and the model successfully predicted the answer to be no; in the preceding screenshot, I also checked the input shape. The predict method takes a batch, so we need to have a 4D input vector and always make sure of that.

With this, we conclude the functional API part. We can now appreciate how easy it is to build complex models with the help of the functional API. But even with its flexibility, there are certain things we cannot do with functional API; let us discuss that and explore another way to create nonlinear graphs, which is the most flexible way, that is, model subclassing.

10.3 Model subclassing

Till now, we have learned two ways to create a TensorFlow model, using sequential API and functional API. The third way we can create a model is through model subclassing. Model subclassing is exactly what it sounds like; we inherit the **`tf.keras.layers.Layer`** class to create our layer. To create a model, we can inherit the **`tf.keras.Model`** class. This is the most flexible way to create a model but a bit more verbose than functional API.

We need to define **`__call__`** to create a layer/model, the following is a convolution plus batch norm layer using model subclassing:

```
class ConvBNLayer(tf.keras.layers.Layer):
 def __init__(self, nf, m, s, padding='same'):
  super(ConvBNLayer, self).__init__()
        # conv layer
  self.conv = tf.keras.layers.Conv2D(nf,
                      kernel_size=m,
                      strides=s, padding=padding)

        # batch norm layer
  self.bn   = tf.keras.layers.BatchNormalization()


 def call(self, input_tensor, training=False):
  x = self.conv(input_tensor)
  x = self.bn(x, training=training)
  x = tf.nn.relu(x)


        return x
```

We define the layers in **`__init__`**, and in __call__, we create the forward pass. This way of defining a model also feels natural, because in the functional API, we create layers line by line, which feels a bit unnatural; layers are not independent blocks. However, it gives us the flexibility to do so. This is the reason we create functions to create models with functional API. But model subclassing feels much more readable and intuitive. But the problem with model subclassing may introduce some logical errors that will be difficult to solve because of the flexibility. So, whenever possible, we may choose to use functional API. Since we have complete control over the layers and the forward function, we can write custom layers.

Conclusion

With this, we conclude *chapter 10* on multiple input and output models. This was a short chapter on creating models with complicated nonlinear topology. The key takeaway is the functional API, using which we can create nonlinear models having multiple input and multiple outputs. We can also share layers among models. We build a visual question answering use case, where we built a model to ask questions to an image. Finally, we learned about model subclassing, the third way to create a most flexible model but a bit more verbose. There are certain models, such as tree RNN models (recursive tree models), which cannot be created even with functional API, there you may need to create using model subclassing. We have not gone into the details of model subclassing because we seldom use it for business use cases.

This is the final chapter in business use cases. In our next chapter, we will come across a few tips and tricks on contributing to the machine learning community.

Points to remember

The following are a few points to remember:

- With sequential API, we can only create linear topology models; we need functional API/model subclassing API to build nonlinear models
- Merging layers is the key to building multiple input models; there are **`Add()`**, **`Subtract()`**, **`Multiply()`**, **`Average()`** and **`Concatenate()`** layers that merges two or more activations
- Model subclassing is the most flexible out of the three APIs to create a model
- Always check shapes when building a convolutional model to avoid later shape-related errors

MCQ

1. **How can we create non-linear topology models?**
    a. Using Functional API
    b. Using Sequential API
    c. Using Model Subclassing API
    d. Both a and c

2. **How to read an image from a file?**
    a. list
    b. mpimg.imread
    c. Image
    d. np.array

3. **Which is the most flexible way to create models in TensorFlow**
    a. Sequential API
    b. Functional API
    c. Model Subclassing API
    d. Both a and c
    e. Both b and c

4. **Which function is used for plotting the model graph?**
    a. plot_graph from keras.utils
    b. plot_model from keras.utils
    c. read_graph from matplotlib
    d. imshow from matplotlib.pyplot

Answers to MCQ

1. d
2. b
3. c
4. b

Questions

1. Create a RESNET model using functional API and train the VQA data.
2. Take the Word2Vec embeddings for the VQA data in question 1.
3. Rewrite the VQA model with Model Subclassing API.

Key Terms

- Non linear topology
- MIMO models
- Visual question answering
- Model subclassing API
- Sequential API
- Functional API

CHAPTER 11
Contributing to the Community

In this chapter, we will learn about building a profile for machine learning by contributing to the machine learning community. We will discuss some practical and effective tips on reading and implementing machine learning papers to leverage this to write our blogs and market the skills. We will learn some resources where we can get good papers to implement. Nowadays, having a good GitHub profile is almost mandatory to get a good job. So, we will discuss how to contribute to the open-source community with ease. We will also discuss tips and tricks on building Kaggle kernels. This will give you additional stars on your profile as a machine learning engineer. Finally, we will discuss creating a blog with ease using FastPages. We will discuss some tips to find topics for your blogs quickly. This will be a short chapter, and we will be discussing a few practical tips to get you started with building a great machine learning profile by contributing to the community. So, we will essentially be exploring different options to contribute to the community: writing code implementations of papers, blogging, building kernels, and datasets with Kaggle. Let's get started.

Structure

In this chapter, we will cover the following topics:

- Tips for reading and implementing a machine learning paper
- Contributing to open-source projects

- Building Kaggle kernels and creating datasets
- Writing about your work

Objective

After studying this chapter, you should be able to:

- Learn how to read and implement a machine learning paper through popular frameworks such as Tensorflow/Pytorch
- How to start contributing to open-source projects
- Learn how to build Kaggle kernels and create datasets for building a good Kaggle profile
- Learn about blogging and how to use tools like FastPages to create blogs with ease

11.1 Tips for reading and implementing a machine learning paper

If you want to become good at machine learning, one important skill is implementing machine learning research papers. Although it is not a mandatory skill to build state-of-the-art models for business tasks, it is a skill that will be required in the long run. But if you have never read or implemented any research paper before or are new to this, you will find it extremely difficult. You will find it mathematical, and a lot of the stuff will be derived from other papers. In addition, you will only be able to understand the abstract. It will be like hitting a brick wall once we go through the implementation.

In this chapter, we will discuss how we can break that brick wall. But remember, it won't be an easy task. You will require a lot of practice before you become comfortable in implementing a research paper. And remember, this is not because you want to go into machine learning research; even if you are only interested in building a machine learning engineer career, it is a good skill to have. And the side effects are extremely valuable too. You will get good coding skills if you can implement a paper. You will get an internal understanding of machine learning ideas and algorithms and get good document reading skills. We will discuss where to start, how to choose a paper to implement. Now let us discuss the tips to choose and implement research papers.

Where to choose papers?

With all the advancement in machine learning in this decade, one of the painful aspects of such rapid growth is research papers. If you go to *arxiv.org* and search for machine learning papers, you will find around 100 papers published every day.

So, one of the resources a lot of people who are into research follow is **http://www.arxiv-sanity.com/**. It has a better view of recent papers, ranked differently; you can also find similar papers and add papers to your library. You can also investigate **https://paperswithcode.com/**. Here you will find machine learning papers that come with code implementation. A few GitHub repositories are dedicated to keeping track of good papers in a particular domain such as NLP, Computer Vision, etc. You can also bookmark one of those repositories to find few good papers to implement first.

Determining if a paper is good or not is a subjective idea, and it depends solely on the reader and their interest. Still, if you are new to this, I would suggest selecting a paper with an official implementation because some code is already present to refer to if you are stuck.

Skim through it first

First, skim through the paper, do not read the entire paper in detail. This is one reason many people find reading papers difficult initially because people who read a good amount of paper each day or researchers seldom go through the entire paper in the first pass.

The reason is simple; it's a good approach to follow a coarse to fine approach during research. We need first to have an overall idea about the paper. This comes from reading the abstract, part of the introduction, dataset, how well it performed (results section), overall wording of the algorithm and implementation details, and the conclusion. Once you have an overall idea of what the paper is all about, you can decide if you want to invest your time in implementing the paper. Research papers are also like shopping; you may reserve your judgment till you try a few other options. And to try more than one research paper is a tedious task if you go through the entire paper in detail. So, start with skimming the paper. Try highlighting the parts of text you understand and parts that you don't understand in different colors. Then, try writing a summary of what exactly the paper is trying to do. If you can do that, you got the gist of the paper. And if you like the paper and decided to give it a go, you can continue; otherwise, you can skip it.

Read some blogs to understand if you find it hard

Once you decide, you will invest some time in the paper to go on a second pass. This time try changing the red colored lines into green (those you didn't understand); if you find the papers too complicated, try following some YouTube videos to explain the paper. Or go through some blogs for the paper if available. Therefore, you should start with some well-known papers first. Skip the math at this point. Well-known papers generally have good explanation blogs/videos, which are good to understand rather than the paper. But even if you read the blog, still try to read through the paper after it. This is because our overall goal is to try and implement research papers, including the skill to read research papers. And reading a well-explained blog is not the same as reading the paper yourself. Once you understand the paper, you are finished with the second pass. Now comes the third pass, which is understanding the math. This is the step where many people find reading research papers too difficult.

Three things you can do to tackle it. First, familiarize yourself with Greek symbols. You will often find it difficult to read mathematical symbols because you cannot read most of the symbols aloud. Second, you take a short linear algebra course; we have a short course to linear algebra in the appendix, you can follow that. That will teach you mostly matrix calculus, which is what you mostly need with machine learning. The third is to read the code. If you follow a blog where the author already implemented the math into code, or you selected a paper that comes with an official code, then use it to understand the equation. It is often simpler than math.

Try implementing with your favorite library

Once you have understood the paper and most of its math, you must have already implemented some of its parts in code. Now you start implementing the code completely. This will take time, but eventually, you will get somewhere. Try selecting a deep learning framework. It does not matter much which one. You can choose Tensorflow/Pytorch/FastAI. There are preferences, though, but it is up to you. Whichever framework you are comfortable with, choose it. But some essentially packages you must know are NumPy, matplotlib or any visualization, pandas, any argument parser (most people use **`argparse,`** so learn that), and so on. Some other packages also you will eventually learn like **`tqdm`**, **`pathlib`**, **`os`**, and so on. But the most important package, in my opinion, is **`numpy`**. Not because you will write your entire code in it, but it will give one of the most important skills to have in machine learning: understanding shapes.

If you can do it, then it's great, and now you can try writing a blog on it. Create a public repository of your implementation on GitHub or even create a video explaining your work in a video and post it to YouTube. But if you are still unable to make much progress, follow the blog discussed earlier to understand the code or the official implementation of the code. Don't copy code because it won't help to learn to implement the paper. If you are still stuck, spend some more time with the paper. Try re-reading it. But remember, you will fail to implement some of the papers. And initially, you will fail a lot. And that is expected, and you can select a new paper and try working on it.

Avoid training on huge data or huge architectures at first

Finally, avoid training on huge data or huge architectures at first. Because they take a long time to train. Initially, you will find many bugs in your code, and it will be difficult to implement with large data. So always start small and then iteratively do complex things.

11.2 Contributing to open source projects

Another way you can contribute to the community is by contributing to open source projects. Open source projects are great. It is community-driven, so you will learn to standardize your code. You will learn how to create pull requests and face code scrutiny. Look into the expert's code. All of these will make you a good programmer. And software engineering is a vital part of machine learning. So, start with small **Pull Requests** (**PRs**). Suggest a bug fix or small enhancement. Even some libraries miss proper documentation on their code; you can always write documentation. Writing documentation is not that trivial as it sounds. As a side effect, you will understand code too.

Once you familiarize yourself with open-source framework, you can now create your projects. We will go into some more tips on creating your projects, but a good starting point is creating user-friendly wrappers. For example, if you want to give a more user-friendly library to fine tune your model. Create some hobby projects too. You can build some projects around NLP, Computer Vision, and so on and post them. You may also create a project that implements a research paper that we discussed in the preceding section. And don't limit yourself to machine learning projects only; you can create other projects too. The idea is to learn good software engineering by contributing to open source projects. Some tools you may explore is NBDEV from FastAI. It lets you fast track your project development by automating many things. Go through the official documentation of NBDEV to understand its powers.

So, to summarize, here is how you can contribute to open source:

- Search for good projects, for example, Pytorch, Tensorflow, FastAI, and so on for machine learning.
- Read the documentation first to understand what the project is all about. Look for contributing to documentation first to understand the contribution guidelines and open source contribution process.
- Start with using the application/project. Then, find bugs and raise issues.
- Pick an issue you feel you can solve, comment on it that you are working on, and ask the developers for clarification.

- Once you decide to contribute to the project, fork, and clone the project to your local.
- Follow contribution guidelines to set up your environment. Then, do the changes and push to your local branch (make sure you take a pull before that to be in sync with the actual project).
- Create a PR request and add a precise title and a proper description (often, these are mentioned in the contribution guidelines document).

11.3 Building Kaggle kernels and creating datasets

Kaggle is a great platform to showcase your machine learning knowledge; you can also find great datasets to play with, many interesting competitions to participate in, and build your machine learning skills. Kaggle also has a rank-based system, and you can create good Kaggle kernels (these are notebooks only). Besides participating in a competition and creating our kernels on various datasets provided by Kaggle, you can also create some datasets and post them in Kaggle. This would be a great way to give back to the community. Data acquisition and cleaning are a tedious part of machine learning, so it would be very helpful to build relevant datasets. Before you start building a Kaggle kernel, it is good to go through some of the most voted kernels to understand how to write a good kernel.

11.4 Writing about your work

Blogs are another area of contributing to the community. It's often the easiest one to do. Platforms like Medium make it easier to write blogs on machine learning. You can also write blogs with FastPages by FastAI, enabling you to convert your Jupyter Notebooks/Markdown/Word documents directly into a blog post. You can follow its documentation to use it, and I highly suggest using FastPages to start.

Writing blogs gives you many features like code highlight, interactive graphs with Altair, collapsible codes, to name a few. It automatically creates the blog site using GitHub pages and uses GitHub actions to automate the deployment process. So the only thing you need to worry about is the content. But using a tool is always secondary. You may find other tools like WordPress or Medium more effective for writing blogs. But the main idea is to have something to write about. So the first thing you can do before writing blogs is to read blogs. It will give ideas on writing a blog, articulating your ideas and having a structure, and much more. Then follow some dos and don'ts when writing a blog; the following are few:

- Make it specific and structured. Don't create a long article touching a lot of ideas.

- You can write comparison blogs, for example, comparing the performance of deep learning algorithms on a particular task and summarization blogs that focus on summarizing many different models on a specific task.
- Always start with value addition. The reader needs to get some value out of your blog. Value requires context and application. Your blog needs to give context on how it can be applied to a real-world problem.
- Use images and graphs. Here the FastPages will help you a lot; you can easily add images and interactive graphs. You will also have a good-looking code.

Few don'ts when writing blogs involve writing a dull blog where only text is present without much motivation. Some blogs don't have a specific context, and they stand out as vague and too general. For example, don't write too many math explanations if your blog says, "building a classifier and deploying it on Heroku." People will expect tips on building a classifier and deploying it to Heroku and not explaining how to batch normalization works. Imagine if someone wants information on batch normalization. Will he click on a blog that says, "*Building a classifier and deploy it on Heroku*"? The answer is no. He will search for the specific thing.

Also, start writing a blog on any work you do on machine learning. For example, you are building a project on semantic segmentation, it is always good to write a blog on it. And you know how easy it is to do it with Fast pages; it automatically converts your Jupiter notebook to a blog. And it will also be very helpful to you later if you want to go through the work once again. If you implement a paper, you can also write a blog on that too. Remember you also needed some blogs to understand some paper, so it's a great way of giving back to the community.

Conclusion

With this, we conclude the chapter on contributing to the community. First, we learned about tips and tricks for reading research papers. Then, we learned about few libraries that are necessary to learn to implement research papers. We also learned about sites such as **https://www.arxiv-sanity.com** and **https://www.paperswithcode.com** that are useful in machine learning research. We then learned about contributing to open-source projects, how we can start with finding issues, resolving them, writing test cases and documentation for the open-source projects as a contribution before we can create our open-source projects. Then we learned some tips on blogging, starting a blog, coming up with topics for a blog, and so on. In the next chapter, we will learn more about building a machine learning profile by building interesting projects.

Points to remember

The following are few points to remember:

- It is advised to skim through the paper on the first try. Then gradually build intuition and go into details for implementation.
- Get yourself comfortable with Greek symbols.
- Good documentation is key to writing good code. So always spend some time documenting your code.
- Blogging has more to do with perseverance than writing skills. However, you will eventually learn good writing skills as you write more and more blog posts.
- Getting started is always a difficult step in any project. This is true for open-source contribution, blogging too. FastAI has awesome tools such as NBDev, FastPages to get you started with these.

MCQ

1. **Which is the library from FASTAI that makes it easier to blog?**
   a. NBDEV
   b. FastPages
   c. FastCore
   d. Jekyll

2. **Which is the visualization library used by FastPages to create interactive visualizations in the notebook?**
   a. plotly
   b. matplotlib
   c. altair
   d. None of the above

3. **Which website gives recent papers, search for papers, sort papers by similarity to any paper, see recent popular papers, unlike arxiv?**
   a. Arxiv-sanity
   b. Deep paper
   c. FastAI
   d. None of the above

4. **Which site has code implementation of machine learning papers?**
    a. Papers-with-code
    b. Arxiv-sanity
    c. Google
    d. GitHub

Answers to MCQ

1. b
2. c
3. a
4. a

Questions

1. Create a blog post about implementing the VQA model with Model Subclassing API (or any other work you did throughout this book).
2. Write a small package that gives a short greeting message once installed every day when imported using NBDEV.
3. Implement the VGG16 paper, write a blog on it.

Key Terms

- FastPages
- NBDEV
- Arxiv-Sanity
- Open source

Chapter 12
Creating Your Own Project

In this chapter, we will learn about how to do a machine learning project. We have done few use cases already throughout this book, and now the reader has ample idea on how to create and maintain a machine learning project. In this chapter, we are going to explore how to select a specific project by defining goals. We will also go through some less-known tools to make your machine learning project experience much better. The defining goal is a bit tricky for machine learning, as there are many different skills needed for end-to-end machine learning. In real life, machine learning code in a project almost always contains the least amount of code. Most of the code revolves around user handling, UI, backend, consumption, scaling, and so on. This is what makes machine learning projects a bit unapproachable to beginners. We already deployed a few models using tools like FastAPI and Streamlit. But in this chapter, we will come across few more tools that are not popular but much useful. But tools aside, we will focus more on the way we should approach a machine learning problem. Let us start this chapter on creating your project.

Structure

In this chapter, we will cover the following topics:

- Defining goals for the project
- The project list for various machine learning domains
- Useful tools

Objective

After studying this chapter, you should be able to:

- Learn to define goals for machine learning projects and start with a small project and build upon it
- Understand many project lists with the goals for each level include various domains such as image, text, analytics, and so on
- Various underrated tools for machine learning and how they will make your lives easier

12.1 Defining goals for the project

Machine learning is so vast that beginners often get confused with where to start and what all projects to implement. The way it needs to be handled is by setting goals. And the initial goals should just be getting started with some of the algorithms using Tensorflow/Pytorch for deep learning models and using scikit-learn, XGBoost for the others. But when you have some familiarity with machine learning (you have done few projects, used the mentioned libraries few times), where to go next? Let us define some goals. These will be a bit general, but you may choose to add/remove few goals based on your need.

12.1.1 General goals

These goals are related to general skills that a machine learning engineer needs to have. The following are few learning goals I have listed:

- **Shape manipulation**: This is mostly related to deep learning; shape manipulation is essential. Get yourself comfortable manipulating shapes using reshape, swapaxes, and so on. A lot of deep learning-specific errors come from incorrect shapes of tensors.
- **Data pipelines**: This one is a straightforward goal. To practice data pipelines, always keep one notebook or Python file dedicated to setting up the dataset. Most experienced machine learning engineers create a dataset class for this. Data pipeline includes acquisition, cleaning, creating batches, and so on.
- **Creating custom models**: Rather than using some pre-defined architectures, e.g., AlexNet, BART, BERT, and so on, we will create our custom model.
- **Deep learning framework**: Working knowledge of at least one of the deep learning frameworks, for example, Tensorflow, Pytorch, FastAI, and so on.
- **Deployment and inference**: Deploying a machine learning model so that others can use it.

- **Visualization**: Visualizing results, explain results, and creating a story from the dataset.
- **Splitting data**: Dataset split ratio for training and testing, creating cross validation, stratified split, and so on.
- **Excel**: Excel is another useful skill for a data scientist, although advanced excel usage is not a must-have.
- **SQL**: Most business data will be present inside some database, so familiarize yourself with SQL.
- **Mixed datatype modeling**: Need to have skill for handling mixed data type (Image + tabular, Tabular + text, Image + text), and so on.
- **Training and tuning**: Training models and fine-tuning them using various regularization techniques.
- **Distributed machine learning**: Working with more than one GPU for huge datasets and models.
- **Explainability**: Explaining the prediction.

12.1.2 Computer Vision goals

These goals are related to Computer Vision skills that a machine learning engineer needs to have. The following are a few learning goals I have listed:

- **OpenCV**: A must-know package for Computer Vision contains a rich set of algorithms pertaining to Computer Vision.
- **Model architecture**: Learning various model architectures, for example, ResNets, AlexNet, and so on.
- **Model explanation**: Explaining the predictions using heatmaps, saliency maps, and so on.
- **CNN**: Convolutional neural networks are the backbones of various Computer Vision tasks; understanding them fully is a necessary skill. We should also focus on different layers, fine-tuning, and so on.
- **UNets**: UNets are used for segmentation problems as the output of the model is also an image.

12.1.3 NLP goals

These goals are related to natural language processing skills that a machine learning engineer needs to have. The following are a few learning goals I have listed:

- **Transformers**: Learning transformers has become an essential skill because of the ground-breaking results they provide. Learn to use and fine-tune these models.
- **Regex**: Often overlooked but a great skill to have. Once you master this, you can clean your NLP dataset with ease.
- **Feature extraction**: Feature extraction is a broad term, basically means getting some rich features from text which can be dense—for example, topics, NER, noun phrases, NGRAMS, and so on.
- **Model architecture**: Similar to Computer Vision models, NLP also has some well-defined architectures such as BART, BERT, and so on.

12.1.4 Analytics goals

These goals are related to analytics skills that a machine learning engineer needs to have. The following are a few learning goals I have listed:

- **XGBoost/LightGBM**: This is a library that will give you an edge over the competition, faster to train, has a wide range of features.
- **Random Forest and SGD**: Good initial models for an analytics model. The random forest provides inbuilt missing value handling and feature importance.
- **Time Series**: Learn at least a one-time series model. We can use Prophet library, ARMA, ARIMA models, and so on for time series.

With these goals, we can now discuss some project lists.

12.2 Project list for various machine learning domains

Now let us discuss few projects that you can do in machine learning to accomplish the goals we mentioned. The way we will approach a project is to start small and then gradually do more complicated projects. So, we are going to divide each project into 2-3 levels of implementation. We may use some of the pre-trained models in the first level and create an inference website. In level 2, we can start creating our model or use a big dataset, and so on. The idea here is to accomplish some goals even when we start small. This way of learning ensures you get ample time playing around with different libraries and concepts.

Now let us discuss a few of the projects in each domain and list different actions. We won't list this for all the projects and only list it down for a few. Also, you need to list down the goals you will accomplish doing the project too. This will keep track of progress. As an exercise, the reader should list down these for all the projects you choose to do.

Defining a good project is often tricky, but a good place to start is either A. Adds some impact, has some sensible way of data collection, and most importantly allows error, or B—some project you want to do. Be a bit careful when choosing the latter option. The idea is to start small with projects where you need not think about every aspect of the machine learning lifecycle, create scripts, and automate mundane tasks. Scripting is an essential skill for a machine learning engineer. Always think of a way to present your project to someone else. With this in mind, let us explore some of the machine learning projects to start with.

12.2.1 Computer Vision projects

Let us now discuss some Computer Vision projects with specific goals that we need to complete.

- **Image classification** – Image classification is a must-do project in Computer Vision; this is usually the starting project. Now let us discuss how we can divide this project into different levels.
    - Level 1 – Library based
        - **Image classification from well-known datasets**: MNIST, CIFAR-10, and so on. Summarize results of different model architectures and hyperparameter settings. Try techniques such as one cycle learning, differential learning rates, and so on.
        - **Learnings**: *Training and Tuning, Deployment and Inference, visualization*. Initial usability of deep learning libraries such as Tensorflow/Pytorch/FastAI.
    - Level 2 – Custom Data
        - Image classification from custom images (using BING API) builds a website that serves your model. Add the heatmap of useful features in the image that contributed highly to the prediction.
        - **Learning:** *Data Pipelines, Deployment and Inference, Explainability*
    - Level 3 – Huge Data and Model
        - Train a model with huge data such as Imagenet on multiple GPU setups.
        - Learning – *Distributed Machine Learning*.

- **Face Detection and Recognition**: Face detection is also an often-used project in Computer Vision. It involves recognition (whose face is it?), detection (identifying faces)
    - Level1 - Library based
        - Use Python face detection libraries and build a web application around it
        - Learnings: How to work with a Python package and build websites (using Flask, Streamlit, Gradia, and so on.), *Deployment and Inference*
    - Level2 - OpenCV based
        - Use OpenCV algorithms to detect a face
        - Compare different methods
        - Learning - Learn ***OpenCV*** and apply different computer vision filters and algorithms using it
    - Level3 - Deep learning-based
        - Implement a paper/model that will detect faces
        - Compare different methods
        - Learning: *Deep Learning Framework* such as Tensorflow, Pytorch, neural network architecture, *Tuning and Training*, and so on

The following are a few examples that the reader needs to do the same and define few levels of implementation for each of these projects:

- QR code scanner
- Detect colors in images
- Detect geometric shapes in images.
- Neural style transfer
- Image colorization
- Segmentation
- Localization
- Object detection
- Image captioning
- Pose detection
- Build a people counting solution
- Count vehicles in images and videos

- Object tracking

12.2.2 NLP projects

Let us discuss few NLP projects and how we can divide them into different levels of implementation.

- **Sentiment analysis:** Like the classification model in Computer Vision, we often start an NLP project from sentence classification. Sentiment analysis is also a good starting point for an NLP project.
  - Level1 – Use Libraries
    - Use Huggingface Transformers Pretrained Models, Use Vadersentiment, Textblob, create a Bag of words model with scikit-learn (choose some classifiers and tune them), deploy it using FastAPI and a Bootstrap/Jquery based frontend
    - Learning: Initial Understanding of using *Transformers, Deployment and Inference, Finetuning*
  - Level2 – Fine-tune Transformers
    - Fine tune Huggingface Transformer model
    - Learning: *Transformers, Transfer Learning*

Now you can choose one of the following projects and define the goals and levels of implementation.

- Fake news detection
- Tagging stack overflow questions
- Finding similar sentences
- Question answering
- One shot extraction
- Chatbot
- Summarization

12.2.3 Analytics projects

Let us discuss few analytics projects and how we can divide them into different levels of implementation.

- **Covid19 analytics and visualization no. of cases**: Covid19 data is largely available, hence makes it a good starting project. Also, you can add immediate impact.

- Level1 – Dashboard to Visualize
    - Create a dashboard of the number of cases in each state, add coloring to show the most impacted states; you can take some inspiration from *covid19india.org*
    - Learning: *Visualization, Data Pipelines*, Tools such as Streamlit, Flask, and so on
- Level2 – Machine Learning Forecasting Model
    - Create a time series model for each state to forecast the cases, take variables such as lockdown dates, duration into account
    - Learning – *Time Series Forecasting*

The following are a few example analytics projects you may choose and implement; also, Kaggle is a great place to start looking for analytics projects.

- Time series forecasting of rainfall
- Movie recommendation

12.3 Useful tools

Other than the tools already discussed in this book (MLFlow, Streamlit, ML Frameworks – scikit-learn, deep learning frameworks – Tensorflow, Pytorch, and so on.)

We will discuss few useful tools that are not already discussed in this book but are useful.

- **`tqdm`**: Tool that creates progress bars, useful when training huge models that take a long time to finish
- **`Pathlib`**: Although we discussed this earlier throughout this book, I mentioned it again because using pathlib rather than a library is a great investment
- **Gradio**: Create a shareable application inside notebooks, even with Google Colab
- **Binder**: Deploy any notebook in GitHub so that you can quickly interact with it
- **Colorama**: It makes CLI tools a bit better
- **Fire**: A great argument parsing tool, learn at least one command-line argument parsing tool, this I am suggesting for its easy usage

Conclusion

With this, we conclude the chapter. We learned about how to start a project by defining goals and starting small by defining various levels of implementation of a particular project. Then we discussed a few projects in each domain and did the same. We categorized projects into different levels of implementation based on the skills that we want to learn. Finally, we discussed some useful tools that are not that popular but can improve your experience. The upcoming chapter is a crash course in various useful packages such as Numpy, Pandas, Matplotlib. We will go through some essential functions and methods in these libraries in the next chapter.

Points to remember

The following are few points to remember:

- Define at least two levels for your project to make it easy to implement at first. Then add more features.
- Try adding goals to your project that are mentioned in this book.
- Refactor your code as you go along in your project; use the tools mentioned previously to make it a good experience.
- Try doing at least one project in each of the domains to have coverage. But try achieving all three levels for a project and add more features to it. This ensures a coarse understanding of various domains and a specialized niche for a specific domain.

MCQ

1. **Which library is an argument parsing library?**
   a. argparse
   b. fire
   c. both A and B
   d. None of the above

2. **To create progress bars, which library can you use?**
   a. plotly
   b. matplotlib
   c. tqdm
   d. All of the above

3. **Which package is used for time series modeling?**
   a. Prophet
   b. xgboost
   c. both A and B
   d. None of the above

4. **Which one is a transformer model?**
   a. BERT
   b. BART
   c. XLNet
   d. All of the above

Answers to MCQ

1. c
2. c
3. c
4. d

Questions

1. Choose at least one of the projects mentioned in this chapter and add one new feature to it.
2. Create a simple deep learning library with functionality like optimizers, loss functions, training loop, etc.

Key Terms

1. Transformers
2. TQDM
3. Gradio
4. Prophet

Chapter 13
Crash Course in Numpy, Matplotlib, and Pandas

In this chapter, we will go through a crash course in important packages for machine learning, that is, Numpy, Matplotlib, and Pandas. This is a crash course since we will learn the minimum required to understand the code present in this book. These packages are big and contain much functionality, but we won't be exploring everything in this chapter as we don't need to. It is always better to learn by doing some project rather than mindlessly learning all the features with these packages. So let us start with some of the essential concepts in NumPy, Matplotlib, and Pandas.

Structure

In this chapter, we will cover the following topics:

- Crash course in Numpy
- Crash course in Matplotlib
- Crash course in Pandas

Objective

After studying this chapter, you should be able to:

- Learn essential functions and methods in NumPy for this book code. These include reshaping, indexing, and so on.

- Learn essential functions and methods in Matplotlib for this book code. These include subplots, managing spacing, and various generic plots.
- Learn essential functions and methods in Pandas for this book code. We will learn the basics of selecting columns, reading data, and so on.

13.1 Crash course in Numpy

We will learn the minimum concepts required to get started with machine learning. We will learn most of the concepts here are limited to what we have used throughout this book. We can go through Numpy in detail, but I usually suggest refraining from going into detail about a library at first. In my view, we should learn by doing. Eventually, with more and more examples, we will learn advanced concepts of Numpy. But certain Numpy skills are required even to start deep learning. For example, effortlessly creating different types of numpy arrays is an essential skill. This is something that we always use in machine learning. We create a small example and then develop a model. Now, let us jump straight into the code for the Numpy crash course.

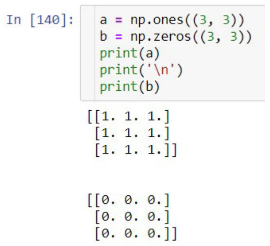
13.1.1 Creating arrays

Creating arrays quickly with NumPy using some of its inbuilt methods. For example, we can create a Numpy array traditionally by defining a Python list and then using **`np.array(some_python_list)`**. But I generally don't prefer this way, as if we want to create a multidimensional array (which we often do in machine learning), we need to enter so many numbers. But with NumPy methods, we can do it quickly; let us check how we can create different types of arrays with ease.

Creating a Numpy array with only 1's/0's:

```
import numpy as np
import matplotlib.pyplot as plt
a = np.ones((3, 3))
b = np.zeros((3, 3))
print(a)
print('\n')
print(b)
```

The output of this code:



```
In [140]:  a = np.ones((3, 3))
           b = np.zeros((3, 3))
           print(a)
           print('\n')
           print(b)

           [[1. 1. 1.]
            [1. 1. 1.]
            [1. 1. 1.]]


           [[0. 0. 0.]
            [0. 0. 0.]
            [0. 0. 0.]]
```

***Figure 13.1**: Creating a numpy 2D array with ones and zeros*

This generates two 3x3 arrays containing ones and zeros only.

Creating a Numpy array with integer range:

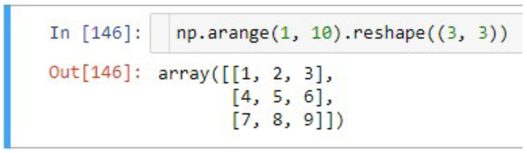
If we want to create a 1D array with an integer range, we can do it as follows:

```
np.arange(1, 10)
```

This will create integers ranging between 1-10. And if we want to create a 2D array containing these numbers, we can simply reshape them.

```
np.arange(1, 10).reshape((3, 3))
```

The output of the preceding code:



```
In [146]:  np.arange(1, 10).reshape((3, 3))

Out[146]: array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])
```

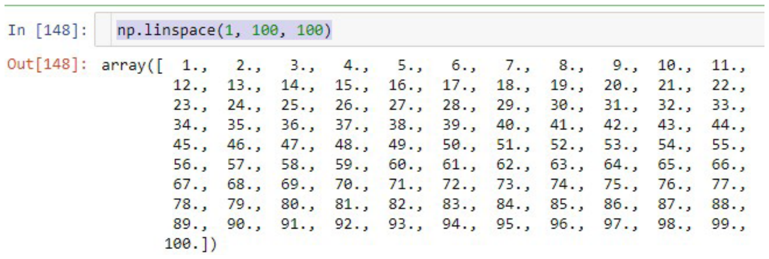
***Figure 13.2**: Creating an array filled with an integer range*

We can also create an array of equally spaced numbers using **`np.linspace`**.

Here is the code:

```
np.linspace(1, 100, 100)
```

The output of this code:



```
In [148]:  np.linspace(1, 100, 100)

Out[148]: array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.,  11.,
                  12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.,  20.,  21.,  22.,
                  23.,  24.,  25.,  26.,  27.,  28.,  29.,  30.,  31.,  32.,  33.,
                  34.,  35.,  36.,  37.,  38.,  39.,  40.,  41.,  42.,  43.,  44.,
                  45.,  46.,  47.,  48.,  49.,  50.,  51.,  52.,  53.,  54.,  55.,
                  56.,  57.,  58.,  59.,  60.,  61.,  62.,  63.,  64.,  65.,  66.,
                  67.,  68.,  69.,  70.,  71.,  72.,  73.,  74.,  75.,  76.,  77.,
                  78.,  79.,  80.,  81.,  82.,  83.,  84.,  85.,  86.,  87.,  88.,
                  89.,  90.,  91.,  92.,  93.,  94.,  95.,  96.,  97.,  98.,  99.,
                 100.])
```

***Figure 13.3:** Creating an array with np.linspace*

This creates 100 numbers between 1 to 100. The first argument is start, the second is stop, and the third is how many you want.

Creating random arrays:

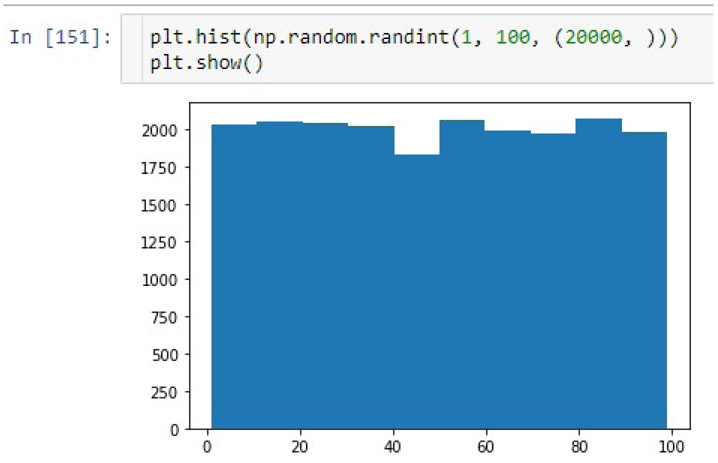
Let us create a few random arrays; for example, if we want to create a random integer array between 1-10 with 4 rows and 5 columns.

```
min = 1
max = 10
shape = (4, 5)
x = np.random.randint(min, max, shape)
x, x.shape
```

`np.random.randint` methods provides a simple way of doing this. This random numbers are pulled from a uniform distribution, we can visualize that as follows:

```
plt.hist(np.random.randint(1, 100, (20000, )))
plt.show()
```

The output of the preceding code:

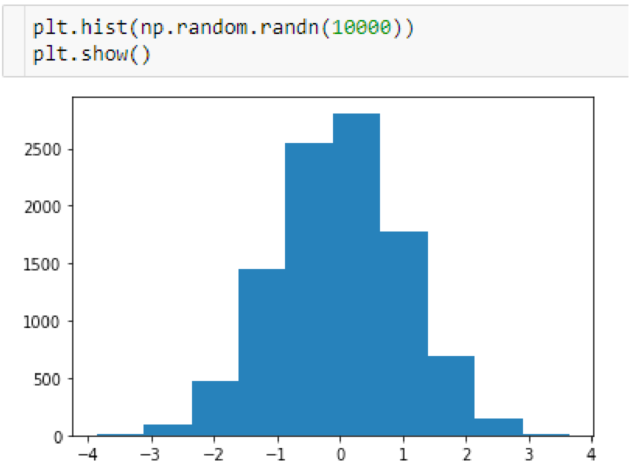


```
In [151]: plt.hist(np.random.randint(1, 100, (20000, )))
          plt.show()
```

Figure 13.4: Plotting randint to check uniform distribution

As per this histogram, we can see almost similar counts for the integers between 1-100 (we took 20000 examples here; the more examples you take, the closer the distribution takes the form of uniform).

Similarly, if we want to create an array from random numbers between zero to one, then we can use **`np.random.rand`**. This will also create a uniform random distribution. For a normal distribution, we have to use **`np.random.randn`**. Let us see this in action.



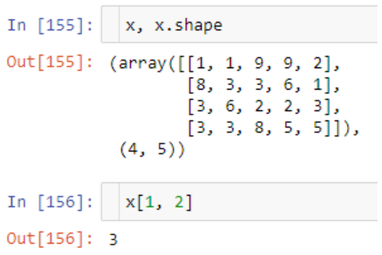
```
plt.hist(np.random.randn(10000))
plt.show()
```

Figure 13.5: Plotting randn for normal distribution

13.1.2 Indexing and reversing arrays

Next, we will learn about indexing into Numpy multidimensional arrays. First, let us take a 2D array that we generated using the **`np.random.randint`** function.

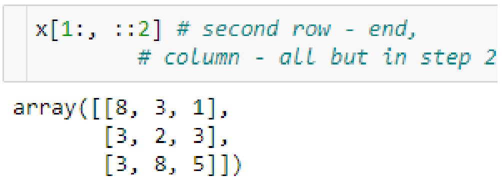
To access the 1 row and second element, we can simply do x[1, 2].



```
In [155]:  x, x.shape

Out[155]: (array([[1, 1, 9, 9, 2],
                  [8, 3, 3, 6, 1],
                  [3, 6, 2, 2, 3],
                  [3, 3, 8, 5, 5]]),
           (4, 5))

In [156]:  x[1, 2]

Out[156]: 3
```

Figure 13.6: *Single indexing 2D array*

Easy right? But what about getting a slice of the array, e.g., if we want the second row to end and all alternative columns starting from the first column. We can use multi-indexing; that is, we can say **`x[start:end:step, start:end:step]`**. Let us see this in code.

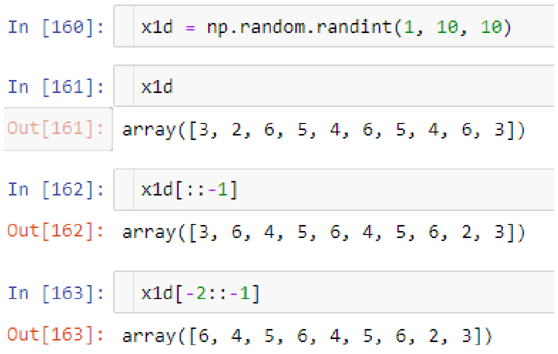


```
x[1:, ::2] # second row - end,
           # column - all but in step 2

array([[8, 3, 1],
       [3, 2, 3],
       [3, 8, 5]])
```

Figure 13.7: *Multi-indexing*

'1:' in the first dimension means we start from the second row onwards for row (as the first row is index 0, and '::2' means all columns, but we skip every other row (take two steps).

To reverse a Numpy array, we simply take the step size as negative, that is:



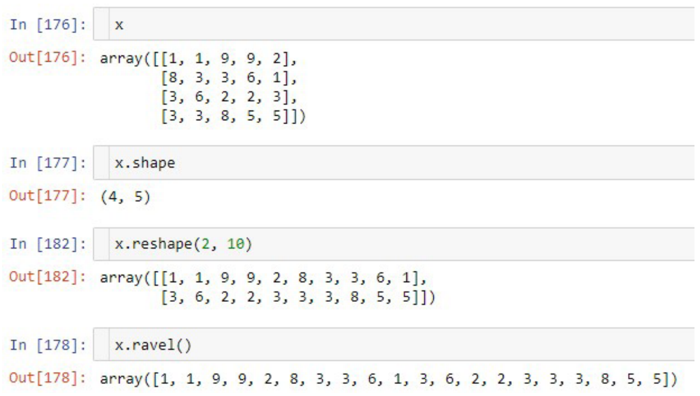
```
In [160]:  x1d = np.random.randint(1, 10, 10)

In [161]:  x1d

Out[161]: array([3, 2, 6, 5, 4, 6, 5, 4, 6, 3])

In [162]:  x1d[::-1]

Out[162]: array([3, 6, 4, 5, 6, 4, 5, 6, 2, 3])

In [163]:  x1d[-2::-1]

Out[163]: array([6, 4, 5, 6, 4, 5, 6, 2, 3])
```

Figure 13.8: *Reversing arrays*

Here we are taking a NumPy array of length 10 (1D), and reversing it is simply doing '::-1'. Similarly, the first argument mentions where to start (-2 means starting from the second last and going in reverse). Reversing arrays might be confusing at first, but play around to understand how the syntax works.

13.1.3 Reshaping

Now let us start some array reshaping; the rule for reshaping is that the total elements (size) should remain the same. Let us see this in code.



```
In [176]:  x

Out[176]: array([[1, 1, 9, 9, 2],
                 [8, 3, 3, 6, 1],
                 [3, 6, 2, 2, 3],
                 [3, 3, 8, 5, 5]])

In [177]:  x.shape

Out[177]: (4, 5)

In [182]:  x.reshape(2, 10)

Out[182]: array([[1, 1, 9, 9, 2, 8, 3, 3, 6, 1],
                 [3, 6, 2, 2, 3, 3, 3, 8, 5, 5]])

In [178]:  x.ravel()

Out[178]: array([1, 1, 9, 9, 2, 8, 3, 3, 6, 1, 3, 6, 2, 2, 3, 3, 3, 8, 5, 5])
```

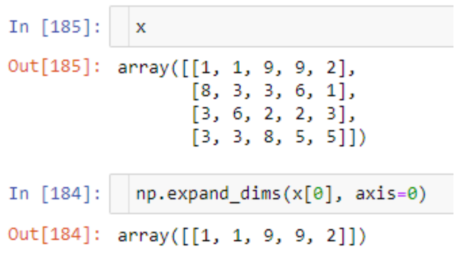
***Figure 13.9**: Reshaping arrays*

We changed our 4 x 5 array to a 2 x 10 array keeping the total number of elements constant. We can also change the 2D (or any D) array into a 1D array using **`ravel()`**. And remember, the reshaping in Numpy does not change the ravel order. This means if you go row by row, the ravel should be the same as that. There is a lot more to reshape (ordering – this is row-wise, but we can reshape column-wise) than this, which you will learn eventually through examples.

13.1.4 Expanding dimensions of batch and grid plot

Finally, before going into matplotlib basics, we will learn expanding batches. This is a simple code but quite useful for prediction because, in deep learning frameworks like Tensorflow or Pytorch, we always take a batch of inputs during prediction.

We can add a batch many ways in NumPy, but I prefer **`expand_dims`**; here is the code:



```
In [185]:  x

Out[185]: array([[1, 1, 9, 9, 2],
                 [8, 3, 3, 6, 1],
                 [3, 6, 2, 2, 3],
                 [3, 3, 8, 5, 5]])

In [184]:  np.expand_dims(x[0], axis=0)

Out[184]: array([[1, 1, 9, 9, 2]])
```

***Figure 13.10**: Adding a batch dimension to the array*

I am taking the second row of our 2D array and adding another dimension along a row in the preceding example.

Using all the skills learned previously, let us write some grid code; grid code is a good test for your Numpy skills. We will try to plot a set of images, but we won't plot them separately in a batch; we will combine the arrays in such a way that we will create a single image. The following is what we want; we will use the easy VQA images (from *Chapter 10: Multiple Input Multiple Output Models*)

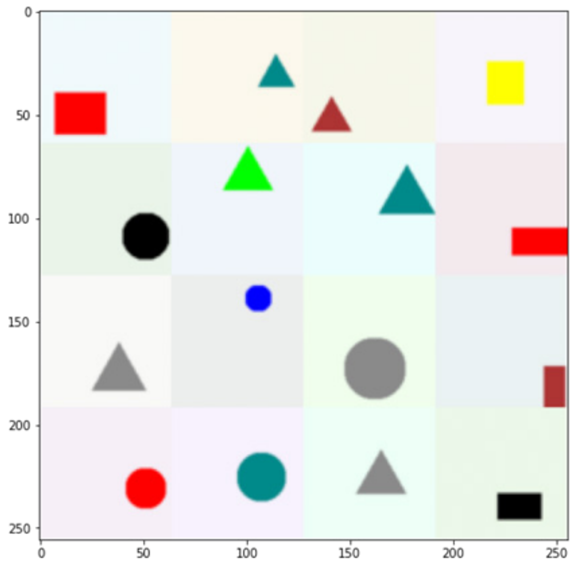


***Figure 13.11**: Plotting an image grid*

Here 16 images are placed in a grid, and a single image is created out of that. To do this, we need to follow steps:

1. Start with 16 images in an array so the shape will be (**`no_images`**, **`height`**, **`width`**, **`channels`**) beginning.

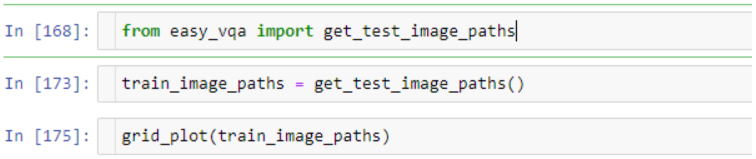
2. We want the final output to be of a shape (**`height*nrows`**, **`width*ncols`**, **`channels`**). We cannot just reshape it to what we want (remember, the reshape works in a ravel direction only).

3. So, we need first to change **`no_images`** to **`ncols*nrows`**. Here we have a squared grid, so **`nrows = ncols = sqrt(no_images)`**. And we will first reshape the 16 images in an array to (**`nrows`**, **`ncols`**, **`height`**, **`width`**, **`channels`**).
4. Then we will swap the axis between **`ncols`** and height to make it (**`nrows`**, **`height`**, **`ncols`**, **`width`**, **`channels`**). We need this because we want the rows and height, cols, and width to lie side by side.
5. Finally, we can reshape it to (**`height*nrows`**, **`width*ncols`**, **`channels`**) to get your grid image.

Here is the code:

```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
def grid_plot(from_paths, size=(16,8), bs=16, ixs=[]):
    if not ixs: ixs = list(np.random.randint(0, len(from_paths) - 1, 
bs))
    batch_paths = [from_paths[i] for i in ixs]
    input = np.array([mpimg.imread(p) for p in batch_paths])
    no_images, height, width, channels = input.shape # this is 
nrows*ncols, height, width, channels

    ncols = int(np.sqrt(bs))
    nrows = no_images//ncols
    assert ncols*nrows == bs, "Batch size must be a perfect square"

    # we want nrows*height, ncols*width, channels
    # so,
    result = (input.reshape(nrows, ncols, height, width, channels) # 
changes to nrows, ncols, height, width, channels
              .swapaxes(1, 2) # (nrows(axis=0), ncols(axis=1), 
height(axis=2), width(axis=3), channels) becomes (nrows, height, ncols, 
width, channels)
              .reshape(height*nrows, width*ncols, channels)) # changes to 
the desired output
    plt.rcParams["figure.figsize"] = size
    plt.imshow(result)
```

We followed all the steps discussed in the code. Additionally, we select the paths randomly and use **`imread`** to read the images and **`imshow`** to show them. Let us pull some images and plot a grid.



```
In [168]: from easy_vqa import get_test_image_paths

In [173]: train_image_paths = get_test_image_paths()

In [175]: grid_plot(train_image_paths)
```

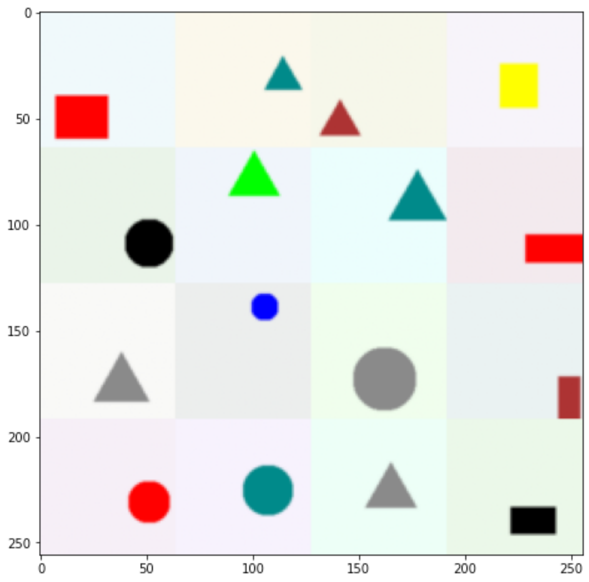


Figure 13.12: *Image grid output with easy VQA*

Every time you run this code, you will get a different set of images as the images are randomly selected.

With this, we conclude the crash course on NumPy; this is scratching the surface of NumPy as the library is extremely feature-rich. But the idea is to learn by doing and learning as much as necessary because learning everything will take a lot of time and shift our focus into many topics. We do not need to be a NumPy expert to start doing machine learning.

13.2 Crash course in Matplotlib

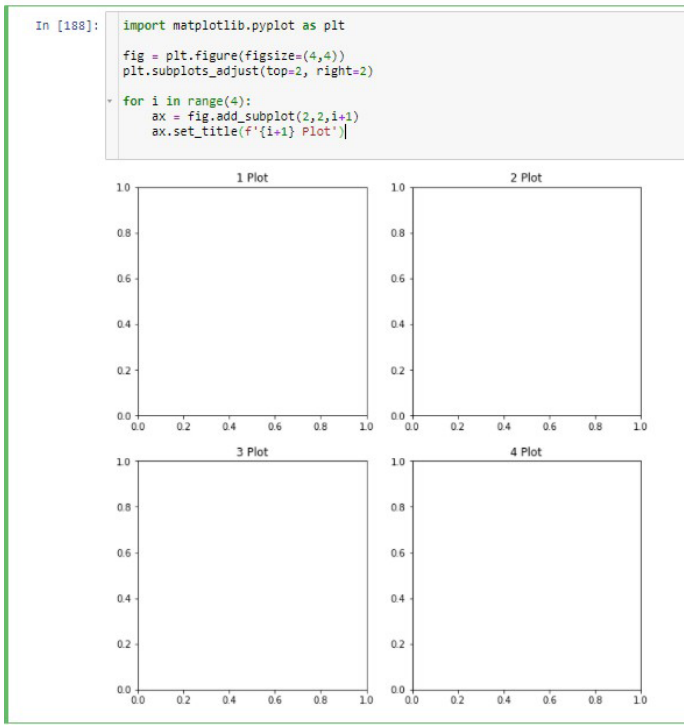
We will go through some of the essentials in matplotlib. We won't go into the details of styling and look and feel that much since styling is pretty specific to the requirement and most often used for presentation. Here we will quickly go through some of the must-known code for matplotlib.

13.2.1 Subplots and spacing

We use **`Matplotlib.pyplot`** to plot various plots with our data. For singular plots, it is done through **`plt.plot`**. But for multiple plots, we can use the object API from matplotlib. Here is how we can create four separate plots:

```
import matplotlib.pyplot as plt


fig = plt.figure(figsize=(4,4))

plt.subplots_adjust(top=2, right=2)


for i in range(4):

    ax = fig.add_subplot(2,2,i+1)

    ax.set_title(f'{i+1} Plot')
```

The output of this code.



```
In [188]: import matplotlib.pyplot as plt

          fig = plt.figure(figsize=(4,4))
          plt.subplots_adjust(top=2, right=2)

          for i in range(4):
              ax = fig.add_subplot(2,2,i+1)
              ax.set_title(f'{i+1} Plot')
```

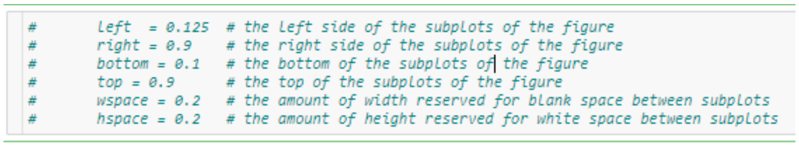
***Figure 13.13**: Subplots with Matplotlib*

We can create a figure in line 3 and add subplots by the add_subplot method giving the number of rows and columns we require as the first two arguments. The third argument specifies the current plot. We are using **`subplots_adjust`** to manage spacing between the plots. But there is another way which I prefer to create multiple charts.

```
rows = 2

cols = 2

fig, axes = plt.subplots(rows, cols, figsize=(4,4))

plt.subplots_adjust(top=2, right=2)

for i in range(rows):

    for j in range(cols):

        axes[i][j].set_title(f"{i+j} location")
```

Here we are doing the same thing but using **`plt.subplots`** and giving rows and cols. We get a figure and axes. Using the axes, we can access each of the subplot locations. This we used in *Chapter 10: Multiple Input Output Models* to create the batch plot. This code outputs the same output as given in the preceding example.

Adjusting spaces between the grid is possible through **`subplots_adjust`** here too. The following are the defaults for these, which you can change as per your requirement:



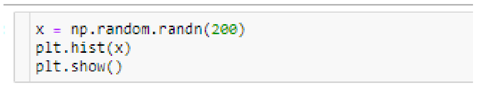
```
#       left  = 0.125  # the left side of the subplots of the figure
#       right = 0.9    # the right side of the subplots of the figure
#       bottom = 0.1   # the bottom of the subplots of the figure
#       top = 0.9      # the top of the subplots of the figure
#       wspace = 0.2   # the amount of width reserved for blank space between subplots
#       hspace = 0.2   # the amount of height reserved for white space between subplots
```

***Figure 13.14**: Default values for subplots spacing*

13.2.2 Bar, line, histogram, and scatter plots

Most often, we use bar, line, histogram, and scatter plots. So, learn to create these plots by heart.

For the histogram chart, here is the code and its output.



```
x = np.random.randn(200)
plt.hist(x)
plt.show()
```

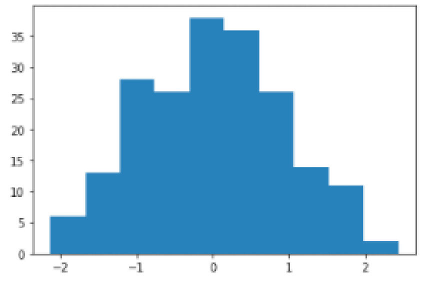


***Figure 13.15**: Histogram plot using matplotlib*

For the bar chart (for categorical data), the following is the chart:

```
departments = ["Science", "Math", "Physics"]
faculties_per_department = [100, 56, 50]

plt.bar(departments, faculties_per_department)
plt.show()
```

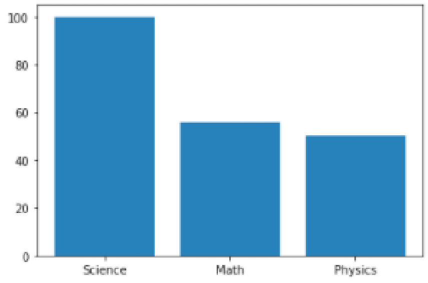


***Figure 13.16**: Bar chart with matplotlib*

For line chart:

```
y = 10*x + 50
plt.plot(x, y)
plt.show()
```

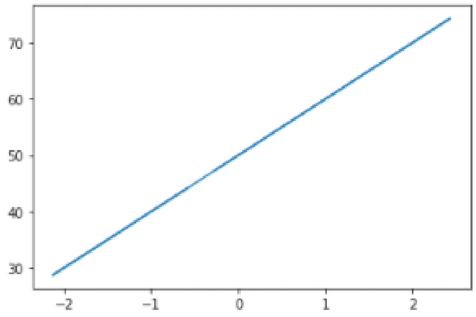


***Figure 13.17**: Line chart with matplotlib*

For scatter plot:

```
plt.scatter(x, y+3*np.random.randn(len(x)))
plt.show()
```

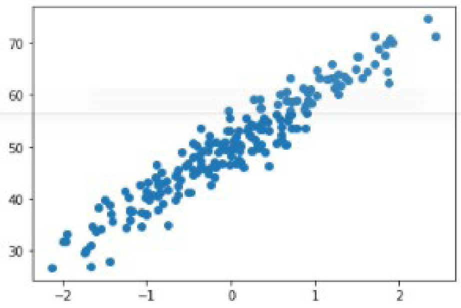


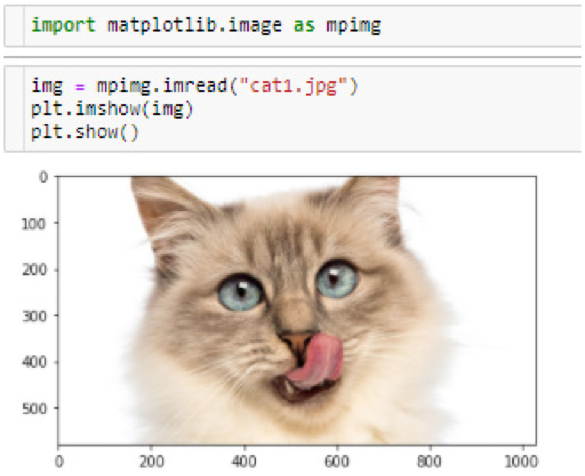
***Figure 13.18**: Scatter plot with matplotlib*

Now, you can do many different stylings with all these, and you will most often do it as the vanilla versions don't look that good.

13.2.3 Handling images

We have already gone through handling images through matplotlib; we can use **`matplotlib.image`** to read the images directly from the path. And then show it using **`matplotlib.pyplot`**.

The following is the code and output:



```
import matplotlib.image as mpimg
```

```
img = mpimg.imread("cat1.jpg")
plt.imshow(img)
plt.show()
```

***Figure 13.19**: Plotting images with matplotlib*

With this, we conclude the crash course on matlplotlib. You can also go through some other visualization libraries such as seaborn, plotly, altair, and so on. Some of these libraries provide interactive plots too.

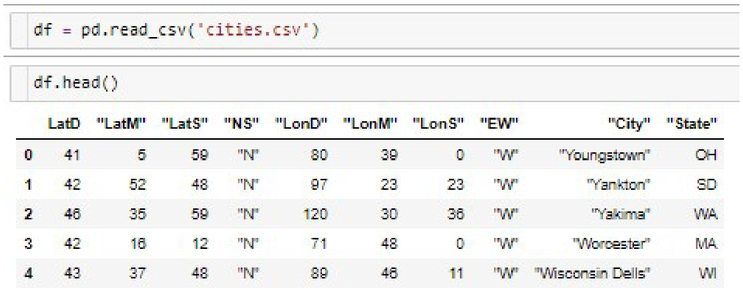
13.3 Crash course in Pandas

The final library that we will go through is Pandas; Pandas is the most used library for data analytics in Python. This is what brought Python into the analytics game. This makes data wrangling way easy and intuitive. So let us start with reading the data with the Pandas library.

The data we are loading is from the following link:

https://people.sc.fsu.edu/~jburkardt/data/csv/cities.csv

This is some random data I selected just for demonstration purposes. To read this data, first, we can download it with **wget** and then load it using **read_csv** (import pandas as pd to load in the Pandas package.)



```
df = pd.read_csv('cities.csv')
```

```
df.head()
```

```
   LatD  "LatM"  "LatS"  "NS"  "LonD"  "LonM"  "LonS"  "EW"            "City"  "State"
0    41       5      59   "N"      80      39       0   "W"       "Youngstown"      OH
1    42      52      48   "N"      97      23      23   "W"          "Yankton"      SD
2    46      35      59   "N"     120      30      36   "W"           "Yakima"      WA
3    42      16      12   "N"      71      48       0   "W"        "Worcester"      MA
4    43      37      48   "N"      89      46      11   "W"  "Wisconsin Dells"      WI
```

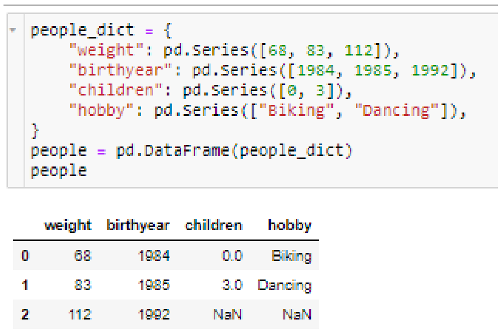
Figure 13.20: *Reading data with Pandas*

There are multiple different formats that we can read our data from. These are json, excel, feather, and so on. Make yourself comfortable with reading data from different sources. To check the columns, we can simply type **df.columns**. This contains all the columns that are present in the dataframe.

We can also create a dataset from dictionaries. This is the way I often create datasets for quickly creating a dataset:

```
people_dict = {
    "weight": pd.Series([68, 83, 112]),
    "birthyear": pd.Series([1984, 1985, 1992]),
    "children": pd.Series([0, 3]),
    "hobby": pd.Series(["Biking", "Dancing"]),
}
people = pd.DataFrame(people_dict)
people
```

The output of this code:

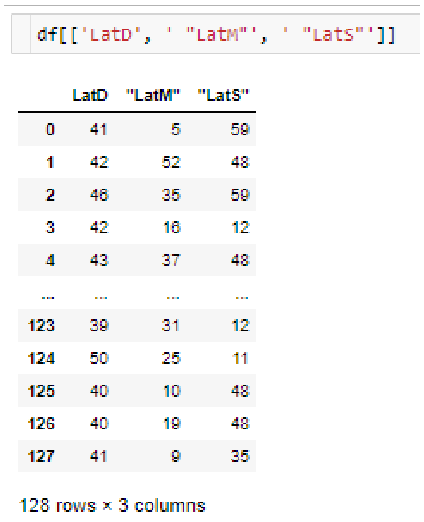


```
people_dict = {
    "weight": pd.Series([68, 83, 112]),
    "birthyear": pd.Series([1984, 1985, 1992]),
    "children": pd.Series([0, 3]),
    "hobby": pd.Series(["Biking", "Dancing"]),
}
people = pd.DataFrame(people_dict)
people
```

```
   weight  birthyear  children    hobby
0      68       1984       0.0   Biking
1      83       1985       3.0  Dancing
2     112       1992       NaN      NaN
```

Figure 13.21: *Creating a dataframe from a dictionary*

Selecting columns is extremely easy with Pandas; we can simply pass the columns we want to extract in a list as an index. Here is the code and output:



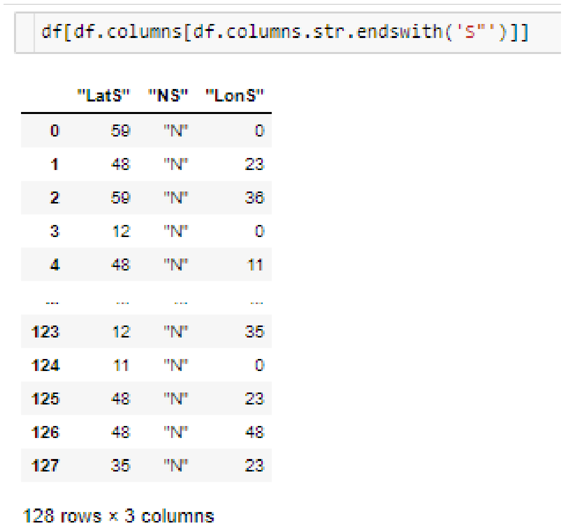
```
df[['LatD', ' "LatM"', ' "LatS"']]
```

	LatD	"LatM"	"LatS"
0	41	5	59
1	42	52	48
2	46	35	59
3	42	16	12
4	43	37	48
...	...	...	...
123	39	31	12
124	50	25	11
125	40	10	48
126	40	19	48
127	41	9	35

128 rows × 3 columns

***Figure 13.22**: Selecting columns*

Suppose we have many columns, and we want to match certain strings with the column name. We can do that with the following code:



```
df[df.columns[df.columns.str.endswith('S"')]]
```

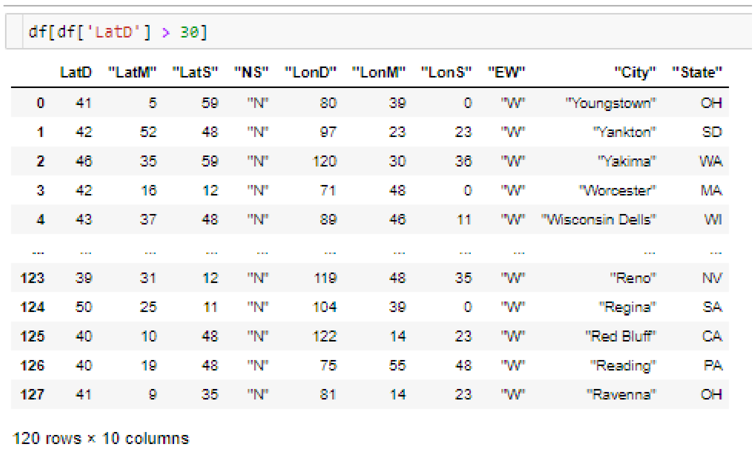
	"LatS"	"NS"	"LonS"
0	59	"N"	0
1	48	"N"	23
2	59	"N"	36
3	12	"N"	0
4	48	"N"	11
...	...	...	...
123	12	"N"	35
124	11	"N"	0
125	48	"N"	23
126	48	"N"	48
127	35	"N"	23

128 rows × 3 columns

***Figure 13.23**: Selecting columns using string pattern*

Here we are selecting all the columns ending with 'S"'. This type of column selection is useful when handling a lot of columns. There is another way we can select both columns and rows from a dataset, too, using **`iloc`** and **`loc`**. But we have already gone through that so that we won't discuss that here.

We can also have filtering with Pandas; let us say we want all latitude values greater than 30, code and output for that:



```
df[df['LatD'] > 30]
```

```
     LatD  "LatM"  "LatS"  "NS"  "LonD"  "LonM"  "LonS"  "EW"            "City"  "State"
  0    41       5      59   "N"      80      39       0   "W"     "Youngstown"       OH
  1    42      52      48   "N"      97      23      23   "W"        "Yankton"       SD
  2    46      35      59   "N"     120      30      36   "W"         "Yakima"       WA
  3    42      16      12   "N"      71      48       0   "W"      "Worcester"       MA
  4    43      37      48   "N"      89      46      11   "W"  "Wisconsin Dells"      WI
...   ...     ...     ...   ...     ...     ...     ...   ...              ...      ...
123    39      31      12   "N"     119      48      35   "W"           "Reno"       NV
124    50      25      11   "N"     104      39       0   "W"         "Regina"       SA
125    40      10      48   "N"     122      14      23   "W"      "Red Bluff"       CA
126    40      19      48   "N"      75      55      48   "W"        "Reading"       PA
127    41       9      35   "N"      81      14      23   "W"        "Ravenna"       OH

120 rows × 10 columns
```

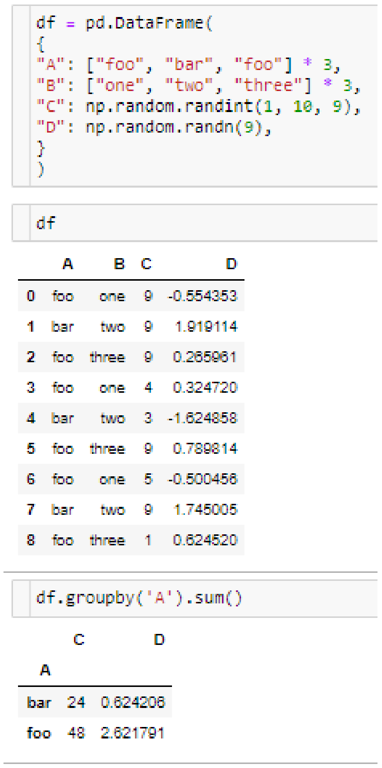
Figure 13.24*: Filtering data with Pandas*

We can combine different filters inside the index too. Another essential thing is to group things in pandas.

Let us create a **`DataFrame`** and group it by certain columns; the code is given as follows:

```
df = pd.DataFrame(
{
"A": ["foo", "bar", "foo"] * 3,
"B": ["one", "two", "three"] * 3,
"C": np.random.randint(1, 10, 9),
"D": np.random.randn(9),
}
)


df.groupby('A').sum()
```

Now this will group by the dataframe with column 'A" and find the sum of all the numerical columns, the output of the preceding code:



```
df = pd.DataFrame(
{
"A": ["foo", "bar", "foo"] * 3,
"B": ["one", "two", "three"] * 3,
"C": np.random.randint(1, 10, 9),
"D": np.random.randn(9),
}
)
```

```
df
```

```
     A      B  C          D
0  foo    one  9  -0.554353
1  bar    two  9   1.919114
2  foo  three  9   0.265961
3  foo    one  4   0.324720
4  bar    two  3  -1.624858
5  foo  three  9   0.789814
6  foo    one  5  -0.500456
7  bar    two  9   1.745005
8  foo  three  1   0.624520
```

```
df.groupby('A').sum()
```

```
      C         D
A
bar  24  0.624206
foo  48  2.621791
```

Figure 13.25: Grouping data and aggregate sum with Pandas

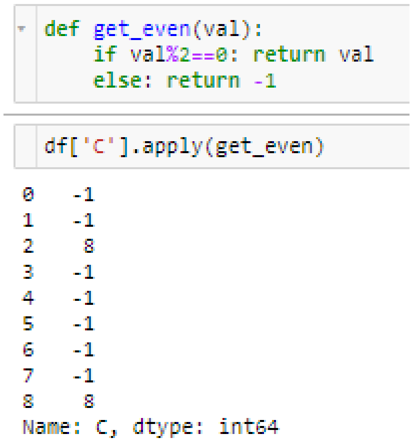
After grouping, we generally perform an aggregate function; we use sum; we have other options, such as count, average, etc.

Other than grouping and indexing, we often use an apply method on dataframe too. The final Pandas skill that we will learn is apply and NumPy where. Often, we require a function to be applied over a complete column; we can do this using one of the two ways.

Using apply:

```
        def get_even(val):
    if val%2==0: return val
    else: return -1


df['C'].apply(get_even)
```

We can write a function taking the column value and process the entire column using apply. Here we are making the odd-numbered values in column 'C' to -1. The output of this code is as follows:



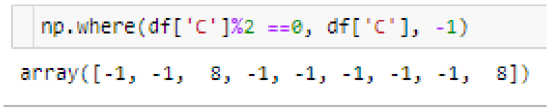
```
def get_even(val):
    if val%2==0: return val
    else: return -1
```

```
df['C'].apply(get_even)
```

```
0   -1
1   -1
2    8
3   -1
4   -1
5   -1
6   -1
7   -1
8    8
Name: C, dtype: int64
```

***Figure 13.26**: Applying a function using apply*

We can do the same thing with **`np.where`**. But we cannot accommodate complicated logic in **`np.where as we cannot give a function, we can only do simpler operations`**.



```
np.where(df['C']%2 ==0, df['C'], -1)
```

```
array([-1, -1,  8, -1, -1, -1, -1, -1,  8])
```

***Figure 13.27**: Using np.where*

Conclusion

With this, we conclude an extremely short introduction to NumPy, pandas, and matplotlib. We barely touched the surface of these libraries. But as discussed, learning a library in detail is not required and may even be detrimental as we won't be requiring every functionality of these libraries. However, engineers use a certain set of methods/functions they like out of these libraries. We learned about some essential functions/concepts that will get us started with machine learning with ease. In our next appendix, we will take a crash course in linear algebra and statistics.

Points to remember

The following are a few points to remember:

- We can perform single and multi-indexing on multidimensional NumPy arrays

- Often deep learning libraries require a batch to predict, using **`numpy.expand_dims`,** we can do this with ease
- Use subplots to create multiple subplots in a single figure with ease
- An aggregate function often follows grouping

MCQ

1. **How to reverse a NumPy 1D array 'a'?**
   a. a.reverse()
   b. reversed(a)
   c. a[:,:,-1]
   d. None of the above
2. **How to swap different axis in NumPy?**
   a. reshape
   b. expand_dims
   c. swapaxes
   d. All of the above
3. **How to add one more dimension using NumPy?**
   a. Np.expand_dims
   b. reshape
   c. Both A and B
   d. None of the above
4. **How to perform a function on all the rows of a particular column in Pandas(dataframe – df, column – A)?**
   a. df[A].applymap(func)
   b. df[A].apply(func)
   c. np.where(df[A], func)
   d. both A and B

Answers to MCQ

1. c

2. c
3. c
4. b

Questions

1. Create a library using Numpy, Pandas, and matplotlib to generate sample datasets for various machine learning tasks like scikit-learn **`make_classification`**.
2. Create an interactive grid of images where you can change the batch size.

Key Terms

- Numpy, Pandas, Matplotlib
- apply
- np.where
- reshaping

CHAPTER 14
Crash Course in Linear Algebra and Statistics

In this chapter, we will learn about linear algebra and statistics. This is a truly short introduction to understand a few of the equations in this book and a basic understanding of math that goes into machine learning. It is not at all extensive. We will start with a small introduction to linear algebra following the paper, *The Matrix Calculus You Need for Deep Learning*, by *Terence Parr and Jeremy Howard*. This paper is a great one to start you with the matrix calculus required for deep learning. The paper is truly concise and gives the structured and minimal math required for deep learning. We will further make it more concise. But I would highly encourage the reader to go through this paper and the blog post on this paper at **https://explained.ai/matrix-calculus/**. We will start with the loss function equation and find the derivative of the loss function with respect to weights. Then, we will go through the chain rule (one of the essentials for deep learning calculus) and a few element-wise operations derivative (also kind of essential for deep learning). Then we will go through a short introduction to statistics. We will learn about p-values and significance tests.

Structure

In this chapter, we will cover the following topics:

- Introduction to linear algebra
- Introduction to statistics

Objective

After studying this chapter, you should be able to:

- Learn how to find the gradient of vectors and find the cost function gradient with respect to weights
- Learn about significance tests and p-values
- Learning basics of z-test for hypothesis testing
- Basic understanding of the chain rule

14.1 Introduction to linear algebra

We will follow the paper to learn linear algebra, even more precisely. In the paper, the author first introduces single variable differentials and then goes through the matrix differentials, elementwise differentials, scalar differentials, and so on. This builds up all the missing pieces to finally get the gradient of a cost function. But we will go in reverse; we will first jump straight into the cost function of gradient descent, read what components we require, and drill down to those components.

Here we follow the top-down approach; we will begin from the overall cost function and go down to each component gradients' specific details. But readers can also follow the bottom-up approach in the paper if they find it more comfortable. This section mostly contains the equations from the paper, with some changes here and there.

Let us first start with defining the input:

$$X = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]^T$$

Our X is a column vector that looks as follows:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

Whenever we talk about vectors, we will think of it as a column. Similarly, for targets, we have the following:

$$\mathbf{y} = [\text{target}\,(\mathbf{x}_1), \text{target}\,(\mathbf{x}_2), \ldots, \text{target}\,(\mathbf{x}_N)]^T$$

Now the loss function of this:

$$C(\mathbf{w}, b, X, \mathbf{y}) = \frac{1}{N}\sum^{N}(y_i - activation(\mathbf{x}_i))^2 = \frac{1}{N}\sum^{N}(y_i - max(0, \mathbf{w}\cdot\mathbf{x}_i + b))^2$$

A loss function contains weights, biases, inputs(X), and targets(y). It is the MSE of the error between the target and activations. We take the dot product of weights with the inputs and add the bias term. Then, add a nonlinear activation which in our case is a ReLU activation.

How can we take the derivative of such a complicated-looking function? The answer is the chain rule. The chain rule is given as follows:

$$derivative\ of\ f(g(x)) = \frac{df(u)}{du}\frac{du}{dx},\ \text{let}\ u = g(x)$$

The idea is to substitute the inner variables of a function. Let us apply this to our loss function and understand how it simplifies the loss function derivative:

$$u(\mathbf{w}, b, X) = max(0, \mathbf{w} \cdot \mathbf{x}_i + b)$$

So, the equation inside the summation is as follows:

$$y_i - max(0, \mathbf{w} \cdot \mathbf{x}_i + b)$$

This later turns into the following:

$$v(y,u) = y - u$$

And finally, the entire cost function is as follows:

$$C(v) = \frac{1}{N}\sum_{i=1}^{N} v^2$$

As per chain rule, we can now find the gradient of the cost function with respect to x as follows:

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial \mathbf{w}}$$

Therefore, to identify the cost function gradient with respect to w, we require these three individual partial derivatives. Let us find the derivative of each of these components one by one:

1. $\frac{\partial}{\partial \mathbf{w}} u(\mathbf{w}, b, \mathbf{x}) = \frac{\partial}{\partial \mathbf{w}}(max(0, \mathbf{w} \cdot \mathbf{x}_i + b))$

For now, let us ignore the max term; we will include it later. And for the 'b' term, with respect to w, the b is a constant (because of partial derivative), so this reduces to the following:

$$\frac{\partial}{\partial \mathbf{w}} \mathbf{w} \cdot \mathbf{x}$$

So, we need the derivative of the dot product between w and x. To calculate this, first, let us understand what a dot product is. It is an elementwise multiplication operation between two vectors followed by a summation.

$$\textit{Dot Product of} \begin{bmatrix}2\\7\\1\end{bmatrix} \textit{ and } \begin{bmatrix}8\\2\\8\end{bmatrix} = \begin{matrix}2\cdot 8 + \\ 7\cdot 2 + \\ 1\cdot 8 = 38\end{matrix}$$

So, the dot product is two operations performed in sequence:

$$\text{sum}\,(\mathbf{w} \otimes \mathbf{x})$$

Where stands for elementwise multiplication, Now, we can assume and perform chain rule to get the result.

$$\mathbf{u} = \mathbf{w} \otimes \mathbf{x}$$
$$y = \text{sum}\,(\mathbf{u})$$

And,

$$\frac{\partial y}{\partial \mathbf{w}} = \frac{\partial y}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{w}}$$

Now, let us identify $\frac{\partial \mathbf{u}}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}}(\mathbf{w} \otimes \mathbf{x})$ term first:

$$\frac{\partial(\mathbf{w} \otimes \mathbf{x})}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial}{\partial w_1}(w_1 * x_1) & \frac{\partial}{\partial w_2}(w_1 * x_1) & \cdots & \frac{\partial}{\partial w_n}(w_1 * x_1) \\ \frac{\partial}{\partial w_1}(w_2 * x_2) & \frac{\partial}{\partial w_2}(w_2 * x_2) & \cdots & \frac{\partial}{\partial w_n}(w_2 * x_2) \\ & \cdots & & \\ \frac{\partial}{\partial w_1}(w_n * x_n) & \frac{\partial}{\partial w_2}(w_n * x_n) & \cdots & \frac{\partial}{\partial w_n}(w_n * x_n) \end{bmatrix}$$

Can you see that all the terms in this matrix except the diagonal elements are zero? If you are unable to see this, then read a bit through partial derivatives. So, this now reduces to the following:

$$\frac{\partial(\mathbf{w} \otimes \mathbf{x})}{\partial \mathbf{w}} = diag(\mathbf{x})$$

In fact, this is true for any elementwise operations. Now, let us try to figure out the second term:

$$\frac{\partial y}{\partial \mathbf{u}} = \frac{\partial \text{sum}\,(\mathbf{u})}{\partial \mathbf{u}}$$

The sum of the vector produces a single output and is hence called a **vector reduction**. If you expand the sum and take the same partial derivative, with respect to , then all the terms in the summation except that index will become zero, and each component will become 1 (dx/dx = 1). As an exercise, try expanding the sum with respect to and do the partial derivatives; you will find:

$$\frac{\partial y}{\partial \mathbf{u}} = \overline{\mathbf{1}}^T$$

Now we can multiply the terms to get the following:

$$\frac{\partial y}{\partial \mathbf{w}} = \frac{\partial y}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial \mathbf{w}} = \vec{1}^T \operatorname{diag}(\mathbf{x}) = \mathbf{x}^T$$

When we accommodate the max as follows:

$$\frac{\partial}{\partial \mathbf{w}} u(\mathbf{w}, b, \mathbf{x}) = \begin{cases} \vec{0}^T & \mathbf{w}\cdot\mathbf{x} + b \le 0 \\ \mathbf{x}^T & \mathbf{w}\cdot\mathbf{x} + b > 0 \end{cases}$$

2. $\frac{\partial v(y,u)}{\partial \mathbf{u}} = \frac{\partial}{\partial \mathbf{u}}(y - u) = \vec{0}^T - \frac{\partial u}{\partial \mathbf{u}} = -\frac{\partial u}{\partial \mathbf{u}} = \overline{-\mathbf{1}}^T$

   This term is straightforward; for the final just expand and take a partial derivative.

3. $\frac{\partial C(v)}{\partial \mathbf{v}} = \frac{\partial}{\partial \mathbf{v}}\left(\frac{1}{N}\sum_{i=1}^{N} v^2\right)$

   In this one, we can move the constant (1/N) and summation outside the derivative (think about it why?)

   So,

$$\frac{\partial}{\partial \mathbf{v}}\left(\frac{1}{N}\sum_{i=1}^{N} v^2\right) = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{\partial}{\partial \mathbf{v}} v^2\right) = \frac{1}{N}\sum_{i=1}^{N} 2v = \frac{1}{N}\sum_{i=1}^{N} 2(y_i - u)$$
$$= \frac{1}{N}\sum_{i=1}^{N} 2(y_i - \max(0, \mathbf{w}\cdot\mathbf{x}_i + b))$$

   Finally, we can multiply all three components:

$$\left(\frac{1}{N}\sum_{i=1}^{N} 2(y_i - \max(0, \mathbf{w}\cdot\mathbf{x}_i + b))\right) * \left(\overline{-\mathbf{1}}^T * \begin{cases} \vec{0}^T & \mathbf{w}\cdot\mathbf{x} + b \le 0 \\ \mathbf{x}^T & \mathbf{w}\cdot\mathbf{x} + b > 0 \end{cases}\right)$$
$$= \tfrac{1}{N}\textstyle\sum_{i=1}^{N} 2(y_i - \max(0, \mathbf{w}\cdot\mathbf{x}_i + b))\,) * \begin{cases} \vec{0}^T & \mathbf{w}\cdot\mathbf{x} + b \le 0 \\ -\mathbf{x}^T & \mathbf{w}\cdot\mathbf{x} + b > 0 \end{cases}$$
$$= \tfrac{1}{N}\textstyle\sum_{i=1}^{N} \begin{cases} \vec{0}^T & \mathbf{w}\cdot\mathbf{x}_i + b \le 0 \\ -2(y_i - \max(0, \mathbf{w}\cdot\mathbf{x}_i + b))\mathbf{x}_i^T & \mathbf{w}\cdot\mathbf{x}_i + b > 0 \end{cases}$$
$$= \tfrac{1}{N}\textstyle\sum_{i=1}^{N} \begin{cases} \vec{0}^T & \mathbf{w}\cdot\mathbf{x}_i + b \le 0 \\ -2(y_i - (\mathbf{w}\cdot\mathbf{x}_i + b))\mathbf{x}_i^T & \mathbf{w}\cdot\mathbf{x}_i + b > 0 \end{cases}$$
$$= \begin{cases} \vec{0}^T & \mathbf{w}\cdot\mathbf{x}_i + b \le 0 \\ \tfrac{2}{N}\textstyle\sum_{i=1}^{N} ((\mathbf{w}\cdot\mathbf{x}_i + b) - y_i)\mathbf{x}_i^T & \mathbf{w}\cdot\mathbf{x}_i + b > 0 \end{cases}$$

This is it; this is the gradient of the cost function with respect to the weight term.

Now how do we interpret it? We can think of the term, as the error term, as is the output and the is the desired output or target, so the cost function reduced to the following:

$$\frac{\partial C}{\partial \mathbf{w}} = \frac{2}{N}\sum_{i=1}^{N} e_i \mathbf{x}_i^T$$

For $\mathbf{w} \cdot \mathbf{x}_i + b > 0$ only; for the zero activations, the cost is simply zero. So, this means the cost function is a weighted average of all the in X, and the weights are the error terms. So, finally, to update the weights, we go in the negative direction of this output.

Similarly, we can calculate the derivative with respect to bias. As an exercise, the reader can follow similar steps to identify the same. In practice, we add a term 1 to the matrix X to accommodate the bias in the same that is, $\hat{\mathbf{w}} = [\mathbf{w}^T, b]^T$.

If you followed until now, you are all set on the linear algebra needed for deep learning. Additionally, you can go through the paper in detail to get some more details on linear algebra.

14.2 Introduction to statistics

We will learn about hypothesis testing in this section. Hypothesis testing is a way to test claims. For example, if you claim that the average height of basketball players is more than the average population height, you need to do hypothesis testing to get to this conclusion. Here, the population refers to all instances. For example, suppose we try to do this experiment in India. In that case, the average population height will take the height of everyone in India and divide it by the number of Indians. This is also a measure of central tendency in data. What do we mean by central tendency? Suppose you pick ten people randomly from different states, called a **sample**, and take their average height; it will most likely be close to the average population height. So, there is a tendency for a sample average to be nearer to the population average. And if you repeat this experiment multiple times (selecting random people each time), then the graph you will get will be something like the following:

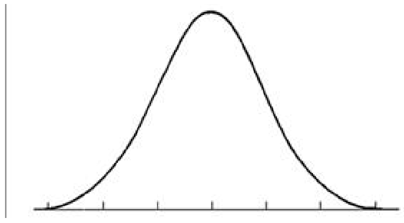


***Figure 14.1**: Sampling distribution*

Here, each point in the graph is an average height of ten random individuals. And the average of that average (mean of the sample means) will be the population mean because when you take more and more such sample averages, most of them will be closer to the mean. This is what we call the sampling distribution. And it can be proven that the mean of the sampling distribution is the population mean.

But this does not cover everything about distribution. For example, if we take 10 Indians, 10 Americans, and 10 Chinese, they take their average height. There is a chance that it will give us a similar average when we take 30 Indians (Here, we are assuming that the Americans have a higher average height than Indians and Indians having a higher average height than Chinese as an example). But the distribution of height of Indian individual samples is different than this case where we take a mixture. And central tendency measures will fail to distinguish that. So, we need a way to measure *variability* in data. This is given by *standard deviation*. The formula for it is given as follows:

$$\mathrm{SD} = \sqrt{\frac{\Sigma|x-\mu|^2}{N}}$$

Where is the population mean and N is the total population size.

We know the mean of the sampling distribution is the population mean. Similarly, it can be proven that the standard deviation of the sampling distribution or **standard error** is population standard deviation, divided by the square root of the number of samples. that is

$$\mathrm{SE} = \frac{\sigma}{\sqrt{n}}$$

The theorem says these two things (mean of the sampling distribution is the population mean, and the standard error is the population standard deviation divided by the square root of the sample size) called the **central limit theorem**. It has a lot of different variations, which we are not going into the details of.

The sampling distribution is a plot of different sample means. But imagine having an experiment where we measure the mean of heights vs. measuring the mean of weights. The sample means will have different scales. So, we normalize it by subtracting the population mean and dividing it by the standard error.

This distribution after the normalization is called a **z distribution**. And the normalized value is called the **z-score**. And this is what we will use to perform the hypothesis testing. The idea is simple if our average height of the basketball players sample is not different from the population mean, then we should get the z-score as close to zero. If the z-score is more, then the **chances** are it is not equal to the population mean.

Since the sampling distribution follows a normal curve, we can calculate exactly the values for the z-score for that **chance**. We have a z-table for that. That will provide us some critical z-values. If our sample z score is more than the z-value, we can say the chances are low. How low? We can decide that too. If the chances are less than 5%, we may consider the basketball players have different heights. These 5%, 10%, or 1% chances (that decides the significance of the hypothesis testing) are called **significance levels** or **p-values**. We won't go into the details of how the critical z-value for these p-values is calculated. But now, we have the tool to test this hypothesis. Here is how we do it:

1. Null Hypothesis = H0 = sample mean = population mean.
2. Alternative Hypothesis = Ha = Sample Mean > Population Mean.
3. Find the sample mean.
4. Calculate the z-score = $Z = \frac{\bar{X}-\mu}{\sigma/\sqrt{n}}$

   Where, is the sample mean, is the population mean, is the population standard deviation.
5. If the z-score exceeds the critical value for that significant level, reject the null hypothesis, which means the alternative hypothesis that basketball players have a higher average height than the population. Otherwise, we will fail to reject the null hypothesis.

The null hypothesis and the alternate hypothesis are just fancy ways of saying if you don't believe or believe the claim. The z-score quantifies that belief. The significance levels or **p-values** define the threshold of that quantification to reject or accept the belief.

One more detail that I want to add is the one-tailed versus two-tailed tests. Our preceding example is of a one-tailed test. Let us see what it means:

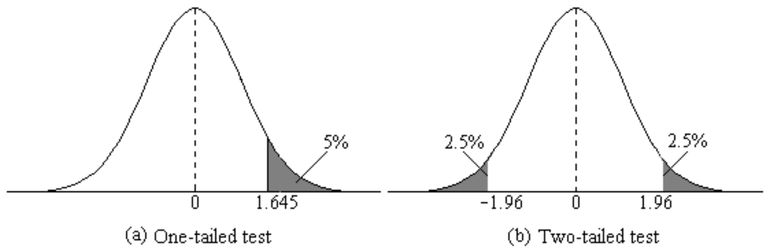


***Figure 14.2**: One-sided vs. two-sided test*

Source: ***http://www.fao.org/3/X6831E/X6831E120.gif***

The preceding image shows the z-curve. And as per our intuition, if the basketball players' height exceeds the population average, our critical z value will have the left figure in the diagram. But in the case of different heights (that means the mean sample height can be more or less than the population average), we will take the two sides of the normalized sampling distribution(z-curve).

That's it; This is what we need to perform hypothesis testing. There is one caveat, though. To calculate the z-score, we require the population mean and standard deviation, which we generally don't have during an experiment (we can't measure each individual's height in India). What we have is again samples. So, the sampling distribution in such a case will have more errors. That distribution becomes t-distribution. And the significance test is called a **t-test**. We won't go into the details of the t-tests here in this chapter. For more details on hypothesis testing, take two free courses on Udacity on descriptive and inferential statistics.

Conclusion

With this, we conclude the chapter on linear algebra and statistics introduction. This is a noticeably short introduction to these vast fields. We are only reading as much we need to understand the topics mentioned in this book. But an interested reader can explore further readings provided in earlier sections of this chapter. In this chapter, we learned to calculate the gradient of a cost function with respect to the weights. We learned how to take the partial derivative of vectors. We also learned about the chain rule and how it helps identify the gradient of complicated functions. Then, we jumped to statistics and learned about hypothesis testing. In the next chapter, we will learn about FastAPI, which is a fast Python backend framework.

Points to remember

The following are few points to remember:

- When you try to find the gradient of a function containing nested functions, use the chain rule
- $\frac{\partial}{\partial \mathbf{w}} \mathbf{w} \cdot \mathbf{x} = \mathbf{x}^T$
- The cost function gradient with respect to weight is a weighted average of the inputs, and the weights are the error terms
- If our sample z-score crosses the critical z-score, then we reject the null hypothesis

MCQ

1. **"Identifying the beak length of a newly discovered bird species to be different than the average beak length" is what type of hypothesis testing?**
   a. One-sided
   b. Two-sided
   c. All of the above
   d. Either A or B

2. **What is the gradient of a sum of two functions?**
   a. Sum of a gradient of individual functions
   b. Multiplication of gradient of individual functions
   c. Chain rule
   d. None of the above

3. **The gradient of the cost function is a weighted average of the inputs. What are the weights?**
    a. target
    b. input
    c. residuals/errors
    d. weights of the neural network

Answers to MCQ

1. b
2. a
3. c

Questions

1. Find the gradient of the cost function with respect to the bias
2. Perform hypothesis testing on real data

Key Terms

- Hypothesis testing
- P-vales
- Z-score

Chapter 15
Crash Course in FastAPI

In this chapter, we will learn about FastAPI that we have used in our earlier chapters to deploy our machine learning models as API. Generally, for machine learning, we use Flask, a lightweight backend framework, for deploying our models. Django is also a popular choice when it comes to building backend in Python. But I prefer to use FastAPI over Flask for several reasons. First, one of the key advantages is speed; FastAPI uses pydantic for data validation and Starlette for tooling. This makes FastAPI comparable performance to Node or Go. Second, it also supports asynchronous mode out-of-the-box. This means you can run coroutines, concurrently which significantly reduces the execution time if run sequentially. Third, FastAPI has a great development experience because it feels like a lightweight library like Flask. Fourth, it comes with out-of-the-box OpenAPI documentation that makes it easier for the frontend team to integrate with the API faster. I have been using FastAPI for a long time. Usually, I get fewer frontend developers than other frameworks as I provide them the docs autogenerated by FastAPI. Finally, it gives you extensive and example-rich documentation. I don't remember the last time I looked up something outside of FastAPI documentation (mainly Stackoverflow) to implement a certain feature. With this in mind, let us write a minimal example with FastAPI and understand how to deploy any machine learning model with FastAPI.

Structure

In this chapter, we will cover the following topics:

- Creating a simple inference API
- Defining input and output data structure and handling data validations

Objective

After studying this chapter, you should be able to:

- Deploy machine learning models with FastAPI
- Learn the basics of using FastAPI
- Easy data validations with pydantic

15.1 Creating a simple inference API

We will start with a simple API for inference and keep our model-related stuff (serialized model file, tests, pre-processing file, and so on) inside a module called **`ml`**. We won't maintain any tests for our simple model, but this is the structure you should follow when creating a machine learning inference API. The following will be the structure of our folder:

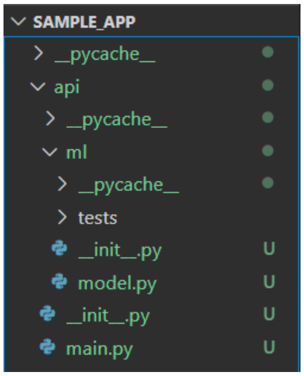


***Figure 15.1**: Inference API scaffolding*

Inside our API module, we have the main file and an **`ml`** folder which contains tests, **`model.py`** which loads the model for the API. Now, let us investigate the **`model.py`** first.

The following is the code inside this file:

```
MALE_WORDS = ['he', 'him', 'his']

FEMALE_WORDS = ['she', 'her']
```

```
min_length = 10

def preprocess(text): return text.lower().split(' ')
def predict(text):
    m = 0
    f = 0
    preprocessed = preprocess(text)
    for word in MALE_WORDS:
        if word.lower() in preprocessed:
            m += 1
    for word in FEMALE_WORDS:
        if word.lower() in preprocessed:
            f += 1

    return {
        "gender_content": {
            "male": m/(m+f) if (m+f) != 0 else 0,
            "female": f/(m+f) if (m+f) != 0 else 0
        },
        "text_processed": preprocessed,
    }
```

It is a simple model that gets the % of male and female words based on certain keywords. I have also mentioned **`min_length`** for text as an input constraint. We will see later how we can enforce this for our input. This is important because when we take input from the user, we have to make sure we get the input in a proper format, and we can enforce this easily with pydantic.

The model is simple. It tokenizes by splitting the text through space and then checks all male/female words present in our simple model. It then returns the male, female content and also the pre-processed text. So, let us now go into the **`main.py`** file and check what the code inside that is:

```
from fastapi import FastAPI

from .ml.model import predict, min_length

@app.get("/")
```

```
def health():
    return {"message": "API is working"}

@app.post("/predict")
def predict_gender_content(text: str):
    resp = predict(text)
    return resp
```

This is how easy it is to create an API with full documentation support (through OpenAPI). We get our predict function and create a **`@app.post`** route to get the text and return the response. To run this, install uvicorn and type "**`uvicorn api.main:app -reload`**." Let us see how the documentation looks for this:

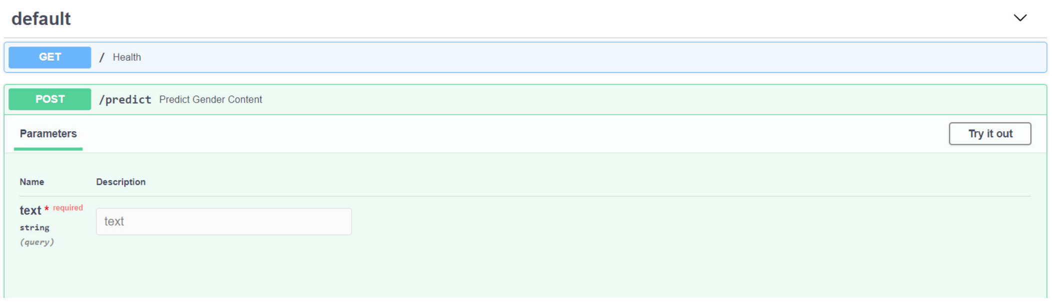


***Figure 15.2**: Automatic documentation from FastAPI*

We can see a text parameter as input, and we can now try out this model using any text. We have done it in *Chapter 7: Data Analytics Use Case* too. But what about the min text length? How can we incorporate this? One way is to check the input length at the predicted endpoint and return an error message when it does not satisfy the minimum length. But we can do better through pydantic validators. We also need to show the user the proper structure for both input and output. Let us see how we can do it with pydantic.

15.2 Defining input and output data structure and handling data validations

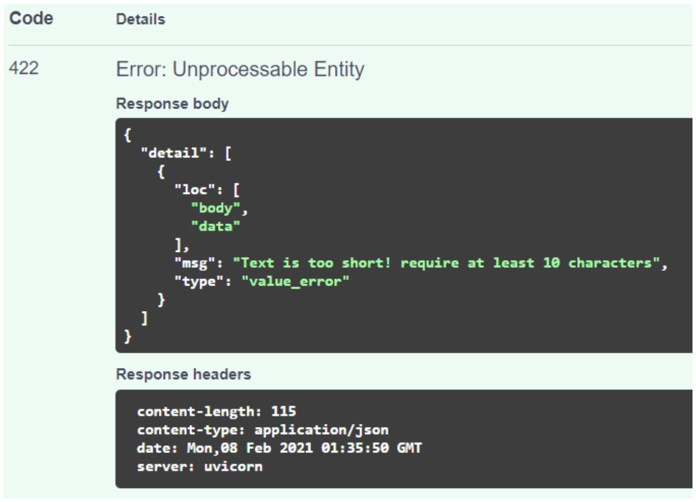
Let us jump straight into the code and see how easy it is to handle this:

```
from typing import Optional, List
from fastapi import FastAPI
from pydantic import BaseModel, validator

from .ml.model import predict, preprocess, min_length
```

```
class PredictRequest(BaseModel):
    data: str

    @validator("data")
    def check_length(cls, t):
        if len(t) < min_length:
            raise ValueError(f"Text is too short! require at least {min_
length} characters")
        return t

class PredictResponse(BaseModel):
    data: dict

app = FastAPI()

@app.get("/")
def health():
    return {"message": "API is working"}

@app.post("/predict", response_model=PredictResponse)
def predict_gender_content(input: PredictRequest):
    text = input.data
    resp = predict(text)
    result = PredictResponse(data=resp)
    return result
```

We define the **`PredictRequest`** class that inherits from **`BaseModel`** from pydantic, and we define it to be a string and a validator that checks if the text has the proper length. This is where we enforce the validations to input data. And generally, in machine learning, we validate the data dimension (let us say we are building an app that predicts housing price and we have ten features; we should check if the data contains ten values. We can do this by following this process of defining a **`PredictRequest`** with proper validations. We also define a **`PredictResponse`** class that defines the structure of the output and the modified predict route.

Now, if we pass a string that is shorter than ten, then we will get a 422 Error with the message we defined in the validator as follows:



Code Details

422 Error: Unprocessable Entity

Response body

```
{
  "detail": [
    {
      "loc": [
        "body",
        "data"
      ],
      "msg": "Text is too short! require at least 10 characters",
      "type": "value_error"
    }
  ]
}
```

Response headers

```
content-length: 115
content-type: application/json
date: Mon,08 Feb 2021 01:35:50 GMT
server: uvicorn
```

***Figure 15.3**: Error response for invalid data*

The reason for defining the request and response format is to make it easy for the users to integrate our API with their code.

We skipped dependency injection in this tutorial as it is beyond the scope but go through the documentation to understand it. Basically, it is required to make sure the model is loaded before making any predictions.

Conclusion

With this, we conclude this short introduction to FastAPI. Now, you can create an inference API for your machine learning model and follow a proper structure. First, we created a simple inference API and then added input and output structures and proper data validations.

Points to remember

The following are few points to remember:

- FastAPI uses starlette for web and pydantic for data validations
- FastAPI has out-of-the-box async and await
- Validate your input data from users always

MCQ

1. **Which is the fastest framework for backend in Python?**
    a. FastAPI
    b. Flask
    c. Django
    d. All are equal in speed
2. **Which error occurs if we violate data validation?**
    a. 422 Unprocessable Entity
    b. matplotlib
    c. altair
    d. None of the above
3. **What is the decorator from pydantic that is used to validate data?**
    a. BaseModel
    b. validator
    c. both A and C
    d. None of the above

Answers to MCQ

1. a
2. a
3. b

Questions

1. Create a model for time series data with Facebook Prophet and deploy it using FastAPI.
2. Add model dependency to the model created in this chapter.

Key Terms

- Data validation
- Asynchronous execution
- Dependency injection

Index