Building explainable machine learning systems

Interpretabl AI

Ajay Thampi

MEAP

MANNING

MEAP Edition
Manning Early Access Program

Interpretable AI

Building explainable machine learning systems

Version 2

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP edition of Interpretable AI.

With breakthroughs in areas such as image recognition, natural language understanding and board games, AI and machine learning are revolutionizing various industries such as healthcare, manufacturing, retail and finance. As complex machine learning models are being deployed into production, the understanding of them is becoming very important. The lack of a deep understanding can result in models propagating bias and we've seen examples of this in criminal justice, politics, retail, facial recognition and language understanding. All of this has a detrimental effect on trust and from my experience, this is one of the main reasons why companies are resisting the deployment of AI across the enterprise.

Explaining or interpreting AI is a hot topic in research and the industry, as modern machine learning algorithms are black boxes and nobody really understands how they work. Moreover, there is EU regulation now to explain AI under the GDPR "right to explanation". Interpretable AI is therefore a very important topic for AI practitioners. There are a few resources available to stay abreast with this active area of research like survey papers, blog posts and a few books but there is no single resource that covers all the important techniques that will be valuable for practitioners. There is also no practical guide on how to implement these cutting-edge techniques.

This book aims to fill that gap by providing a simplified explanation of interpretability techniques and also a practical guide on how to implement them in Python using open, public datasets and libraries. The book will show code snippets and also share the source code for you to follow along and reproduce the graphs and visuals in the book. It is meant to be a hands-on book giving you practical tips to implement and deploy state-of-the-art interpretability techniques. Basic knowledge of probability, statistics, linear algebra, machine learning and Python is assumed.

I really hope you enjoy this book and I highly encourage you to post any questions or comments in the liveBook discussion forum. This feedback will be extremely useful for me to make improvements to the book and increase your understanding of the material.

Happy reading!

— Ajay Thampi

brief contents

PART 1: INTERPRETABILITY BASICS

PART 2: INTERPRETING MODEL PROCESSING

PART 3: INTERPRETING MODEL REPRESENTATIONS

PART 4: FAIRNESS AND BIAS

APPENDIXES

1 Introduction

This chapter covers:

- **Different types of machine learning systems**
- **How machine learning systems are typically built**
- **What is interpretability and why is it important**
- **How interpretable machine learning systems are built**
- **Summary of interpretability techniques covered in this book**

Welcome to this book! I'm really happy that you are embarking on this journey through the world of Interpretable AI and I look forward to being your guide. In the last five years alone, we have seen major breakthroughs in the field of Artificial Intelligence (AI) especially in areas such as image recognition, natural language understanding and board games like Go! As critical decisions are being handed over to AI in industries like healthcare and finance, it is becoming increasingly important that we build robust and unbiased machine learning models that drive these AI systems. In this book, I wish to give you a practical guide on interpretable AI systems and how to build them. Through a concrete example, this chapter will motivate why interpretability is important and will lay the foundations for the rest of the book.

1.1 Diagnostics+ AI – An Example AI System

Let's now look at a concrete example of a healthcare center called Diagnostics+ that provides a service to help diagnose different types of diseases. Doctors who work for Diagnostics+ analyze blood smear samples and provide their diagnosis which can be either positive or negative. This current state of Diagnostics+ is shown in Figure 1.1.

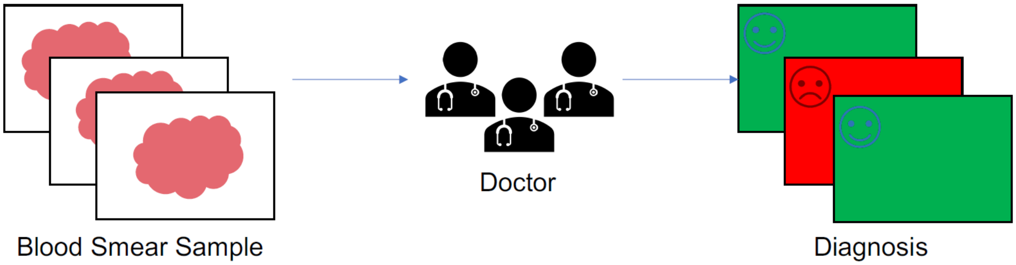


Figure 1.1: Current State of Diagnostics+

The problem with the current state is that the analysis of the blood smear samples is done manually by the doctors. With a finite set of resources, diagnosis therefore takes a considerable amount of time. Diagnostics+ would like to automate this process using AI and diagnose more blood samples so that patients get the right treatment sooner. This future state is shown in Figure 1.2.

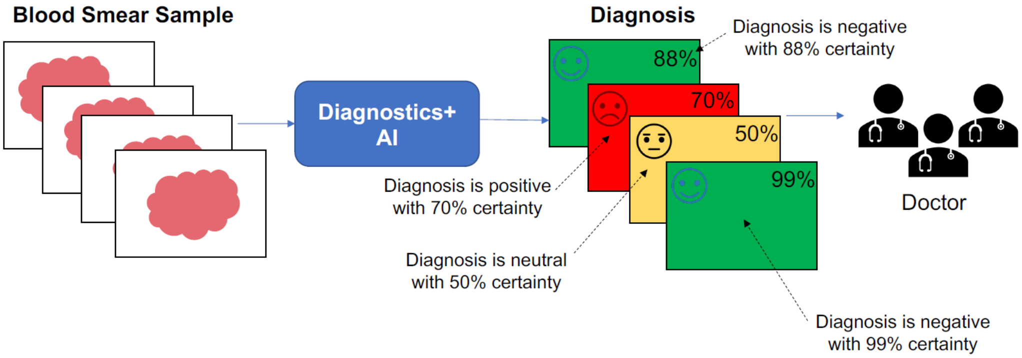

Blood Smear Sample
Diagnostics+ AI
Diagnosis
88%
70%
50%
99%
Diagnosis is negative with 88% certainty
Diagnosis is positive with 70% certainty
Diagnosis is neutral with 50% certainty
Diagnosis is negative with 99% certainty
Doctor

Figure 1.2: Future State of Diagnostics+

The goal for Diagnostics+ AI is to use images of blood smear samples with other patient metadata to provide diagnosis – either positive, negative or neutral – with a confidence measure. Diagnostics+ would also like to have doctors in the loop to review the diagnosis, especially the harder cases, thereby allowing the AI system to learn from mistakes.

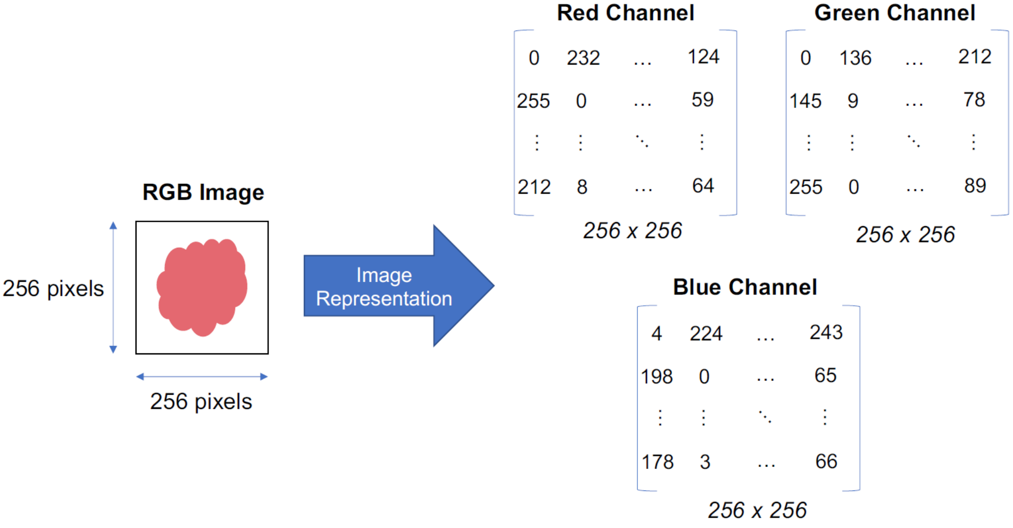
1.2 Types of Machine Learning Systems

There are three broad classes of machine learning systems that can be used to drive Diagnostics+ AI. They are supervised learning, unsupervised learning and reinforcement learning.

1.2.1 Representation of Data

Let's first see how to represent the data that a machine learning system can understand. For Diagnostics+, we know that there's historical data of blood smear samples in the form of images and patient metadata.

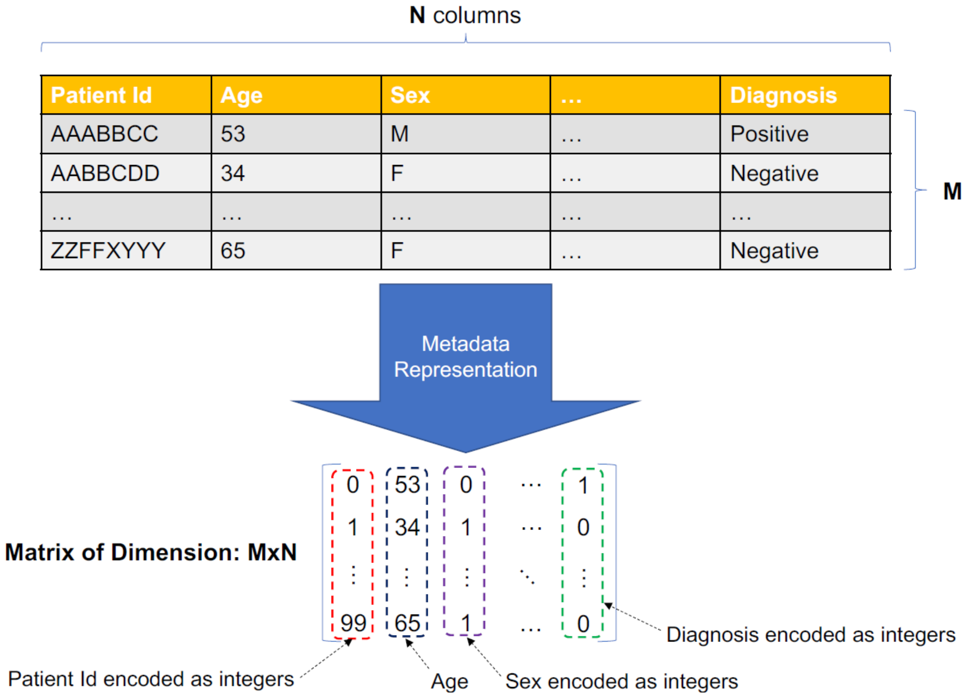
How do we best represent the image data? This is shown in Figure 1.3. Suppose the image of a blood smear sample is a colored image of size 256x256 pixels consisting of three primary channels – red (R), green (G) and blue (B). This RGB image can be represented in mathematical form as three matrices of pixel values, one for each channel and each of size 256x256. The three two-dimensional matrices can be combined into a multi-dimensional matrix of size 256x256x3 to represent the RGB image. In general, the dimension of the matrix representing an image is of the form: *{number of pixels vertically} x {number of pixels horizontally} x {number of channels}*.



$$\begin{bmatrix} 0 & 232 & \cdots & 124 \\ 255 & 0 & \cdots & 59 \\ \vdots & \vdots & \ddots & \vdots \\ 212 & 8 & \cdots & 64 \end{bmatrix} \quad \begin{bmatrix} 0 & 136 & \cdots & 212 \\ 145 & 9 & \cdots & 78 \\ \vdots & \vdots & \ddots & \vdots \\ 255 & 0 & \cdots & 89 \end{bmatrix} \quad \begin{bmatrix} 4 & 224 & \cdots & 243 \\ 198 & 0 & \cdots & 65 \\ \vdots & \vdots & \ddots & \vdots \\ 178 & 3 & \cdots & 66 \end{bmatrix}$$

Figure 1.3: Representation of a Blood Smear Sample Image

Now, how do we best represent the patient metadata? Suppose that the metadata consists of information such as the patient identifier (id), age, sex and the final diagnosis. The metadata can be represented as a structured table shown in Figure 1.4, where there are N column and M rows. This tabular representation of the metadata can be easily converted into a matrix of dimension M x N. In Figure 1.4, you can see that the Patient Id, Sex and Diagnosis columns are categorical and have to be encoded as integers. For instance, the patient id 'AAABBCC' is encoded as integer 0, sex 'M' (for male) is encoded as integer 0 and diagnosis 'Positive' is encoded as integer 1.

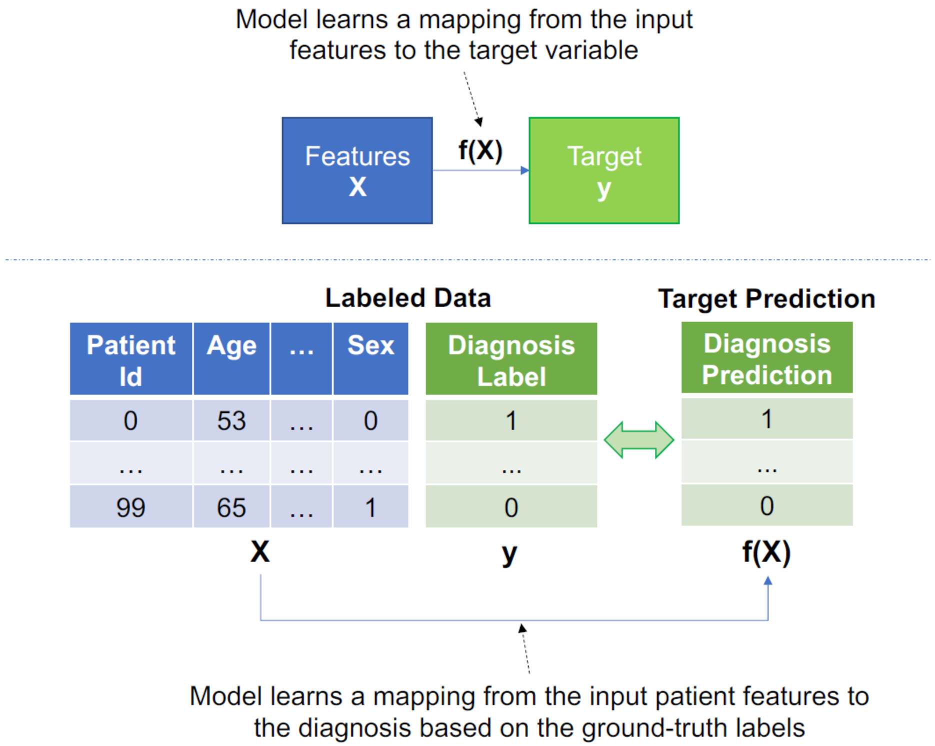


Patient Id	Age	Sex	…	Diagnosis
AAABBCC	53	M	…	Positive
AABBCDD	34	F	…	Negative
…	…	…	…	…
ZZFFXYYY	65	F	…	Negative

Figure 1.4: Representation of Tabular Patient Metadata

1.2.2 Supervised Learning

Supervised Learning is a type of machine learning system where the objective is to learn a mapping from an input to an output based on example input-output pairs. It requires labeled training data where inputs (also known as **features**) have a corresponding label (also known as **target**). Now how is this data represented? The input features are typically represented using a multi-dimensional array data structure or mathematically as a matrix **X**. The output or target is represented as a single-dimensional array data structure or mathematically as a vector **y**. The dimension of matrix **X** is typically **m x n**, where **m** represents the number of examples or labeled data and **n** represents the number of features. The dimension of vector **y** is typically **m x 1** where **m** again represents the number of examples or labels. The objective is to learn a function **f** that maps from input features **X** to the target **y**. This is shown in Figure 1.5.



Model learns a mapping from the input features to the target variable

Patient Id	Age	…	Sex	Diagnosis Label
0	53	…	0	1
…	…	…	…	…
99	65	…	1	0

Diagnosis Prediction
1
…
0

Model learns a mapping from the input patient features to the diagnosis based on the ground-truth labels

Figure 1.5: Illustration of Supervised Learning

In Figure 1.5 you can see how with supervised learning you are learning a function **f** that takes in multiple input features represented as **X** and provides an output that matches known labels or values represented as the target variable **y**. The bottom half of the figure shown an example where a labeled dataset is given and through supervised learning, you are learning how to map the input features to the output. The function **f** is a multivariate function since it maps from multiple input variables or features to a target. There are two broad classes of supervised learning problems:

1. **Regression**: This is a class of problems where the target vector **y** is continuous. For example, predicting the price of a house at a location in U.S. dollars is a regression type of learning problem.
2. **Classification**: This is a class of problems where the target variable **y** is discrete and bounded. For example, predicting if an email is spam or not is a classification type of learning problem.

1.2.3 Unsupervised Learning

Unsupervised learning is a type of machine learning system where the objective is to learn a representation of the data that best describes it. There is no labeled data and the goal is to learn some unknown pattern from the raw data. The input features are represented as a matrix **X** and the system learns a function **f** that maps from **X** to a pattern or representation of the input data. This is depicted in Figure 1.6. An example of unsupervised learning is clustering where the goal is to form groups or clusters of data points with similar properties or characteristics. This is shown in the bottom half of the figure. The unlabeled data consists of two features and the datapoints are shown in 2D space. There are no known labels and the objective of the unsupervised learning system is to learn some pattern from the data. In this illustration, the system learns how to map the raw data points into clusters based on their proximity or similarity with each other. These clusters are not known before hand and hence, the learning is entirely unsupervised.

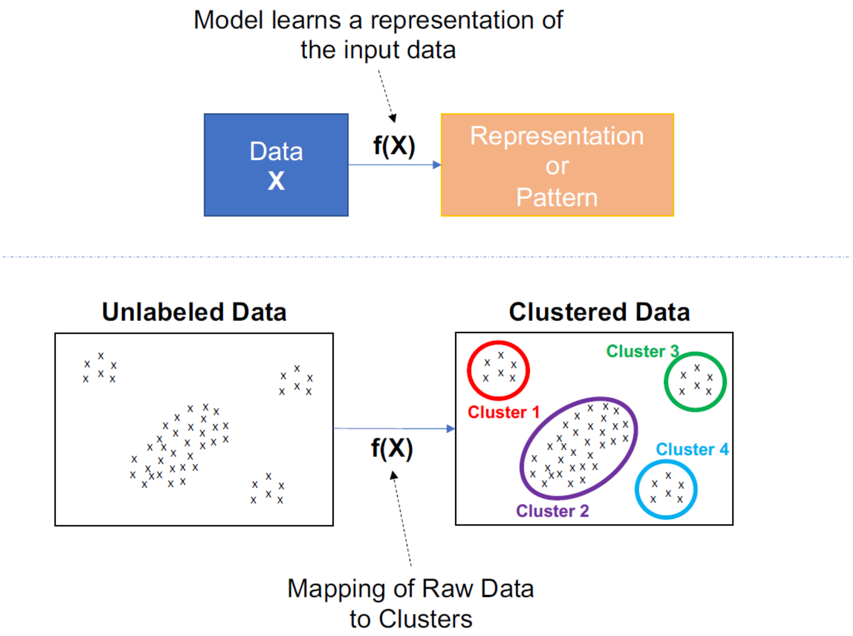

Model learns a representation of the input data
Data
X
f(X)
Representation or Pattern
Unlabeled Data
Clustered Data
Cluster 1
Cluster 2
Cluster 3
Cluster 4
f(X)
Mapping of Raw Data to Clusters

Figure 1.6: Illustration of Unsupervised Learning

1.2.4 Reinforcement Learning

Reinforcement learning is a type of machine learning system that consists of an agent that learns by interacting with an environment. This is shown in Figure 1.7. The learning agent takes an action within the environment and receives a reward or penalty depending on the quality of the action. Based on the action taken, the agent moves from one state to another.

The overall objective of the agent is to maximize the cumulative reward by learning a policy function **f** that maps from an input state to an action. Some examples of reinforcement learning are a robot vacuum cleaner learning the best path to take to clean a home, and an artificial agent learning how to play board games like Chess and Go.

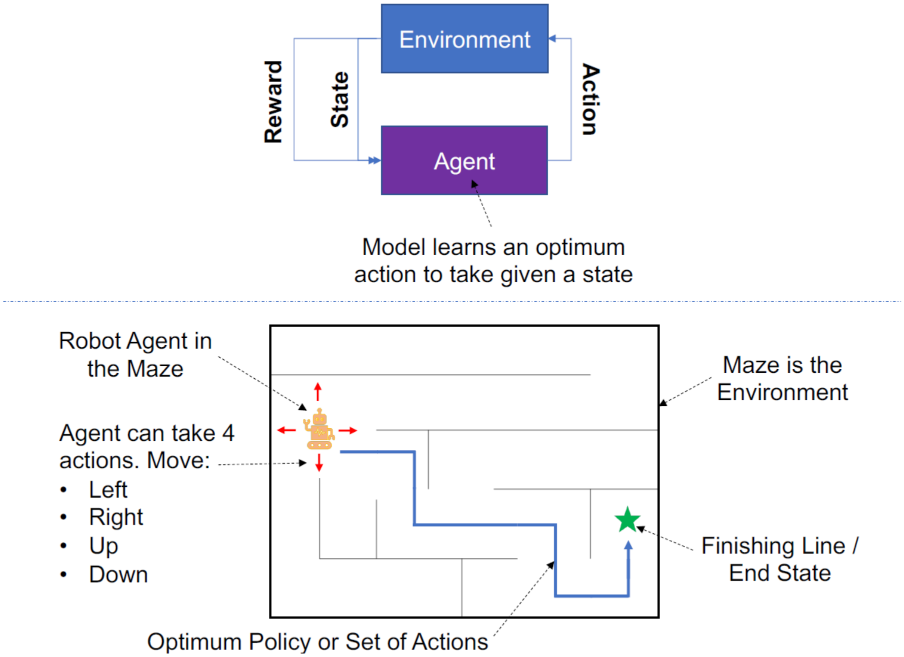

Environment
Reward
State
Action
Agent
Model learns an optimum action to take given a state
Robot Agent in the Maze
Agent can take 4 actions. Move:
• Left
• Right
• Up
• Down
Maze is the Environment
Finishing Line / End State
Optimum Policy or Set of Actions

Figure 1.7: Illustration of Reinforcement Learning

The bottom half of Figure 1.7 gives an illustration of a reinforcement learning system. The system consists of a robot (agent) in a maze (environment). The objective of the learning agent is to determine the optimum set of actions to take so that it can move from its current location to the finishing line (end state) shown as the green star. The agent can take one of four actions – move left, right, up or down.

1.2.5 Machine Learning System for Diagnostics+ AI

Now that you know the three broad types of machine learning systems, which system is most applicable for Diagnostics+ AI? Given that the dataset is labeled where you know from historical data what diagnosis was made for a patient and blood sample, the machine learning system that can be used to drive Diagnostics+ AI is **Supervised Learning**.

What class of supervised learning problem is it? The target for the supervised learning problem is the diagnosis which can be either positive or negative. Since the target is discrete and bounded, it is a **Classification** type of learning problem.

Primary Focus of the Book

This book primarily focuses on **Supervised Learning** systems where there is labeled data. I will teach you how to implement interpretability techniques for both regression and classification types of problems. Although this book does not explicitly cover unsupervised learning or reinforcement learning systems, the techniques learned in this book can be extended to them.

1.3 Building Diagnostics+ AI

Now that you've identified that Diagnostics+ AI is going to be a Supervised Learning system, how do you go about building it? The typical process that is followed consists of three main phases:

1. Learning,
2. Testing, and
3. Deploying

In the Learning phase, illustrated in Figure 1.8, you are in the development environment where you use two subsets of the data called the training set and the dev set. As the name suggests, the training set is used to train a machine learning model to learn the mapping function **f** from the input features **X** (in this case, the image of the blood sample and metadata) to the target **y** (in this case, the diagnosis). Once you've trained the model, you use the dev set for validation purposes and you tune the model based on the performance on that dev set. Tuning the model entails determining the optimum parameters for the model, called **hyperparameters**, that gives the best performance. This is quite an iterative process and you continue doing this until the model reaches an acceptable level of performance.

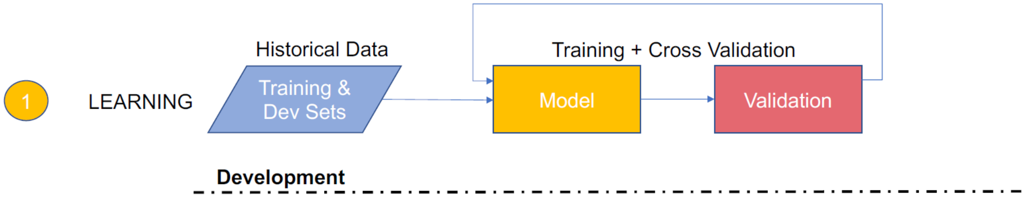


Figure 1.8: Process of Building an AI System – Learning Phase

In the Testing phase, illustrated in Figure 1.9, you now switch over to the test environment where you use a different subset of the data called the test set. You go through a User Acceptance Test (UAT) here where business stakeholders and experts (in this case, doctors) would evaluate the performance of the learned model in phase 1 on the test set. If the performance is not acceptable, then you go back to phase 1 to train a better model. If the performance is acceptable, then you move on to phase 3 which is Deploying.

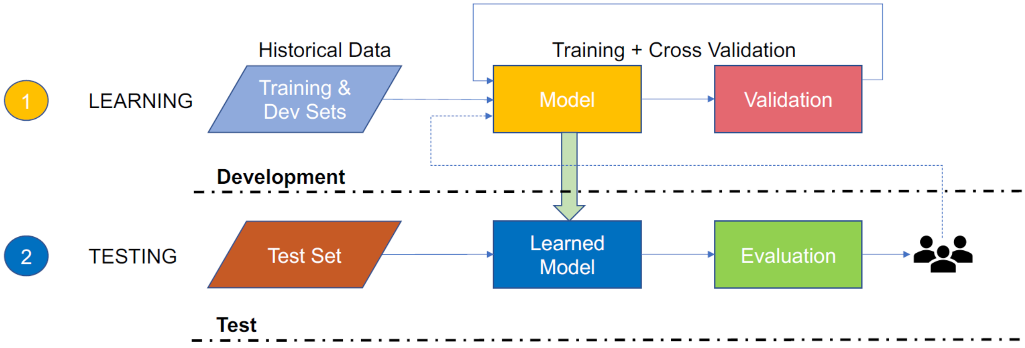


Figure 1.9: Process of Building an AI System –Testing Phase

Finally, in the Deploying phase, you now deploy the learned model into the production system where the model is now exposed to new data that it hasn’t seen before. The complete process is illustrated in Figure 1.10. In the case of Diagnostics+ AI, this data would be new blood samples and patient information using which the model will predict whether the diagnosis is positive or negative with a confidence measure. This information is then consumed by the expert (in this case, the doctor) and in turn the end user (in this case, the patient).

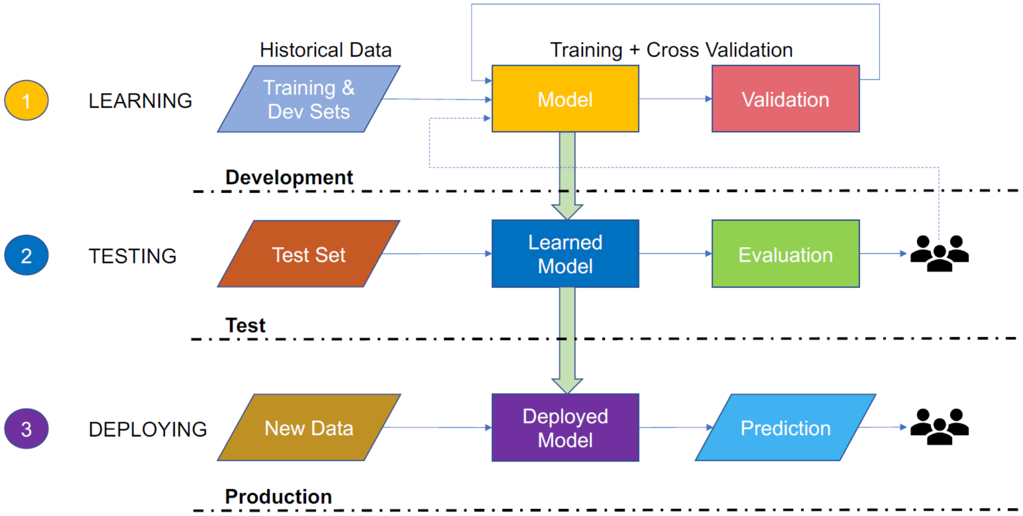


Figure 1.10: Process of Building an AI System – Complete

1.4 Gaps in Diagnostics+ AI

There are major gaps in the Diagnostics+ AI system shown in Figure 1.10. This AI system does not safeguard against some common issues whereby the deployed model does not behave as expected in the production environment. These issues could have a detrimental effect on the business of the diagnostics center. The common issues are:

1. Data Leakage
2. Bias
3. Regulatory Non-Compliance
4. Concept Drift

1.4.1 Data Leakage

Data leakage happens when features in the training, dev and test sets unintentionally leak information that would otherwise not appear in the production environment when the model is scored on new data. For Diagnostics+, suppose you use notes made by the doctor about the diagnosis as a feature or input for your model. While evaluating the model using the test set, you could get inflated performance results thereby tricking yourself into thinking you've built a great model. The notes made by the doctor could contain information about the final diagnosis thereby leaking information about the target variable. This problem, if not detected earlier, could be catastrophic once the model is deployed into production. This is because the model is scored before the doctor has had a chance to review the diagnosis and add their notes. Therefore, the model would either crash in production because the feature is missing or the model would start to make poor diagnoses.

A classic case study of data leakage is the KDD Cup challenge in 2008. This was a machine learning competition based on real data where the objective was to detect if a breast cancer cell was benign or malignant based on X-ray images of the breast. A study showed that teams that scored the most on the test set for this competition used a feature called Patient ID, that was an identifier generated by the hospital for the patient. It turned out that some hospitals used the patient ID to indicate the severity of the condition of the patient when they are admitted at the hospital, thereby leaking information about the target variable.

1.4.2 Bias

Bias is an issue where the machine learning model makes an unfair prediction that favors one person or group over another. This unfair prediction could be caused by the data or the model itself. There may be sampling biases whereby there are systematic differences between the data sample used for training and the population. There may also be systemic social biases inherent in the data which the model picks up on. The trained model could also be flawed where it may have some strong preconceptions despite evidence to the contrary. For the case of Diagnostics+ AI, if there is sampling bias for instance, the model could make more accurate predictions for one group and not generalize well to the whole population. This is far from ideal as the diagnostics center wants the new AI system to be used for every patient regardless of the group they belong to.

A classic case study of machine bias is the COMPAS AI system used by U.S. courts to predict future criminals. A study was conducted by ProPublica where they obtained the COMPAS scores for 7,000 people who had been arrested in a county in Florida in 2013 and 2014. Using the scores, they found out that they could not accurately predict the recidivism rate (i.e. the rate at which convicted person reoffends) where only 20% of the people who were predicted to commit violent crimes actually did so. More importantly though, they uncovered serious racial biases in the model.

1.4.3 Regulatory Non-Compliance

The General Data Protection Regulation (GDPR) is a comprehensive set of regulations adopted by the European Parliament in 2016 that deals with how data is collected, stored and processed by foreign companies. The regulation contains Article 17 which is the "right to be forgotten" where individuals could request a company collecting their data to erase all their personal data. The regulation also contains Article 22 where individuals could challenge decisions made by an algorithm or AI system using their personal data. This regulation presses the need for providing an interpretation or explanation to challenging individual for why the algorithm made that particular decision. The current Diagnostics+ AI system does not comply with both sets of regulations. In this book, we are concerned more with Article 22 as there are a lot of online resources available for how to be compliant with Article 17.

1.4.4 Concept Drift

Concept drift happens when the properties or the distribution of the data in the production environment has changed when compared to the historical data used to train and evaluate the model. For Diagnostics+ AI, this could happen when a new profile of patients or diseases emerge that aren't captured in the historical data. When concept drift happens, you will observe a dip in performance of the machine learning model in production over time. The current Diagnostics+ AI system does not properly deal with concept drift.

1.5 Building a Robust Diagnostics+ AI

Now, how do you address all the gaps highlighted in Section 1.4 and build a robust Diagnostics+ AI system? The process needs some tweaking. The first change is to add a model understanding phase after the testing phase and before deploying. This is shown in Figure 1.11.

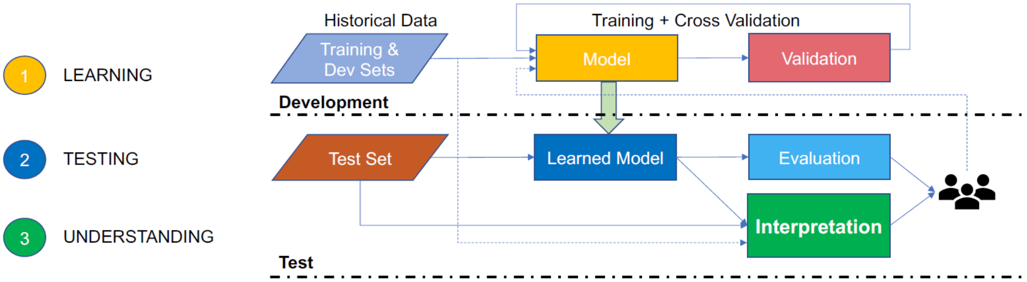


Figure 1.11: Process of Building a Robust AI System – Understanding Phase

The purpose of this new **understanding** phase is to answer the important how-question, i.e. how did the model come up with a positive diagnosis for a given blood sample? This involves interpreting the important features for the model and how they interact with each other, interpreting what patterns the model learned, understanding the blind spots, checking for bias in the data and ensuring those biases are not propagated by the model. This understanding phase should ensure that the AI system is safeguarded against the data leakage and bias issues highlighted in Sections 1.4.1 and 1.4.2 respectively.

The second change is to add an **explaining** phase after deploying, as shown in Figure 1.12. The purpose of the explaining phase is to interpret how the model came up with the prediction on new data in the production environment. By interpreting the prediction on new data, it allows you to expose that information if needed to expert users of the system who challenge the decision made by the deployed model. Another purpose is to come up with a human-readable explanation so that it can be exposed to wider end users of the AI system. By including the interpretation step, you will be able to address the regulatory non-compliance issue highlighted in Section 1.4.3.

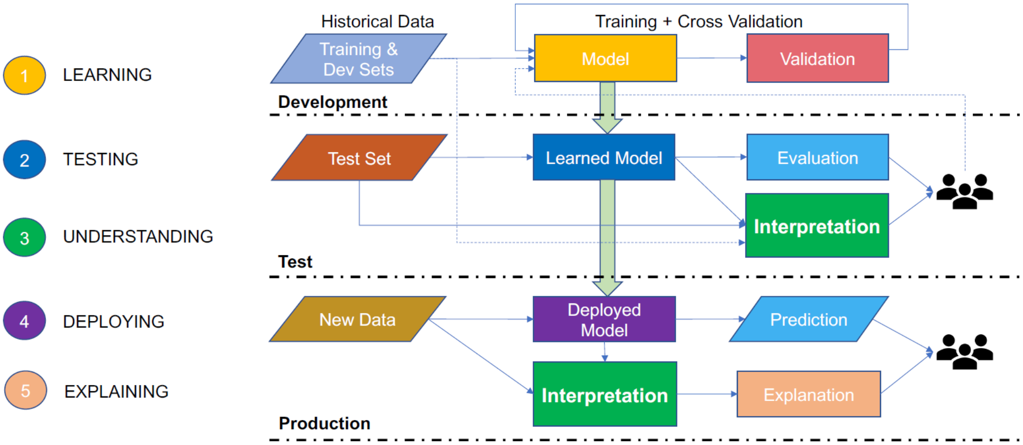


Figure 1.12: Process of Building a Robust AI – Explaining Phase

Finally, to address the concept drift issue highlighted in Section 1.4.4, we need to add a **monitoring** phase in the production environment. This complete process is shown in Figure 1.13. The purpose of the monitoring phase is to keep track of the distribution of the data in the production environment as well as the performance of the deployed model. If there is any change in data distribution or if there is a dip in model performance, then you will need to go back to the learning phase and incorporate the new data from the production environment to retrain the models.

Primary Focus of the Book

This book primarily focuses on the **Interpretation** step in the understanding and explaining phases. I intend to teach you various interpretability techniques that you can apply to answer the important how-question and address the data leakage, bias and regulatory non-compliance issues. Although explainability and monitoring are important steps in the process, it is not the primary focus of this book. It is also important to make a distinction between interpretability and explainability. This is addressed in the following section.

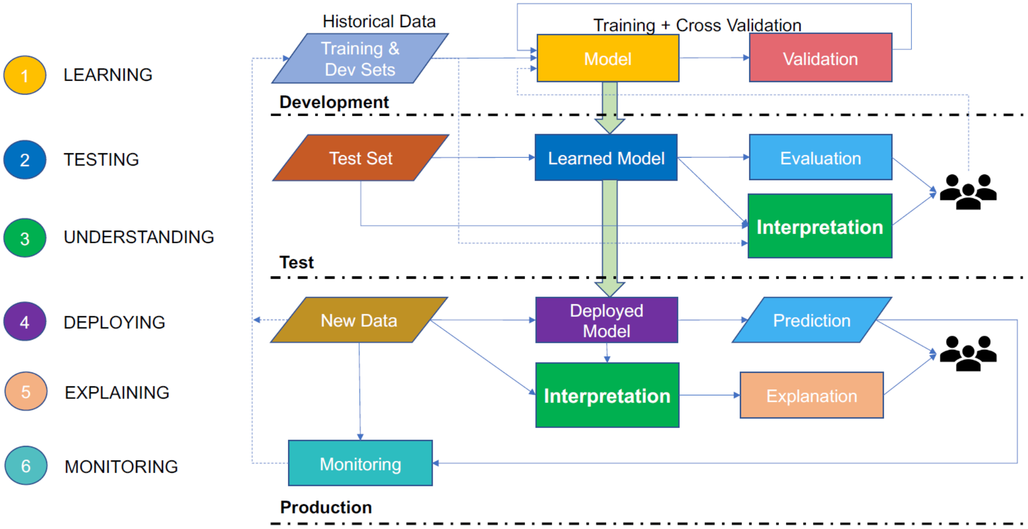


Figure 1.13: Process of Building a Robust AI System – Complete

1.6 Interpretability v/s Explainability

Interpretability and explainability are sometimes used interchangeably but it is important to make a distinction between the two terms.

Interpretability is all about understanding the cause and effect within an AI system. It is the degree to which we can consistently estimate what a model will predict given an input, understand how the model came up with the prediction, understand how the prediction changes with changes in the input or algorithmic parameters and finally understand when the model has made a mistake. Interpretability is mostly discernible by experts who are either building, deploying or using the AI system and these techniques are building blocks that will help you get to explainability.

Explainability, on the other hand, goes beyond interpretability in that it helps us understand in a human-readable form how and why a model came up with a prediction. It explains the internal mechanics of the system in human terms with the intent to reach a much wider audience. Explainability requires interpretability as building blocks and also looks to other fields and areas such as Human-Computer Interaction (HCI), law and ethics. In this book, I wish to focus on interpretability and less so on explainability. There is a lot to cover within interpretability itself, but it should give you a solid foundation to be able to later build an explainable AI system.

There are four different personas that you should be aware of when you consider interpretability. They are the **data scientist** and **engineer** who are building the AI system, the **business stakeholder** who wants to deploy the AI system for their business, the **end**

user of the AI system and finally the **expert** or **regulator** who monitors or audits the health of the AI system. Note that interpretability means different things to these four personas.

- For a **data scientist** or **engineer**, it means to gain a deeper understanding of how the model made a particular prediction, what feature are important and debug issues by analyzing cases where the model did badly. This understanding helps the data scientist build more robust models.
- For a **business stakeholder**, it means to understand how the model madeß a decision so as to ensure fairness and protect the business's users and brand.
- For an **end user**, it means to understand how the model made a decision and to allow for meaningful challenge if the model made a mistake.
- For an **expert** or **regulator**, it means to audit the model and the AI system and follow the decision trail especially when things went wrong.

1.6.1 Types of Interpretability Techniques

There are various types of interpretability techniques and they are summarized in Figure 1.14.

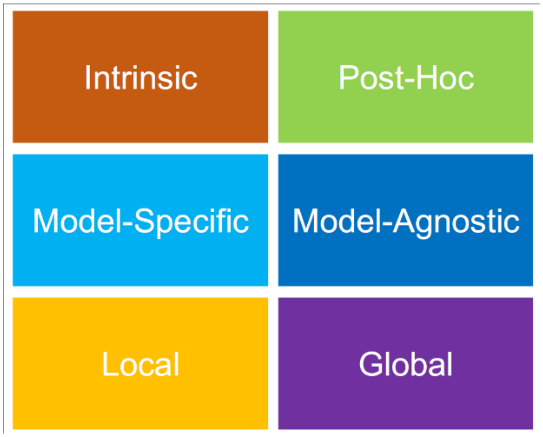


Figure 1.14: Types of Interpretability Techniques

Intrinsic interpretability techniques are related to machine learning models that are have a simple structure, also called white-box models. White-box models are inherently transparent, and it is straightforward to interpret the internals of the model. Interpretability comes right out of the box for such models. **Post-hoc** interpretability techniques are usually applied after model training and are used to interpret and understand the importance of certain inputs for the model prediction. Post-Hoc interpretability techniques are suited for white-box and black-box models, i.e. models that are not inherently transparent.

Interpretability techniques can also be model-specific or model-agnostic. **Model-specific** interpretability techniques, as the name suggests, can only be applied to certain types of models. Intrinsic interpretability techniques are model-specific by nature since the technique is tied to the specific structure of the model being used. **Model-agnostic** interpretability techniques are however not dependent on the specific type of model being used. It can be applied to any model as it is independent of the internal structure of the model. Post-hoc interpretability techniques are mostly model agnostic by nature.

Interpretability techniques can also be local or global in scope. **Local** interpretability techniques aim to give a better understanding of the model prediction for a specific instance or example. **Global** interpretability techniques, on the other hand, aim to give a better understanding of the model as a whole, i.e. the global effects of the input features on the model prediction. We will be covering all of these types of techniques in this book. Now let's take a look at what specifically we will be learning.

1.7 What will I learn in this book?

A map of all the interpretability techniques you will learn in this book is shown in Figure 1.15. When interpreting supervised learning models, it is important to make a distinction between white-box and black-box models. Examples of white-box models are linear regression, logistic regression, decision trees and Generalized Additive Models (GAMs). Examples of black-box models are tree ensembles like random forest and boosted trees, and neural networks. White-box models are much easier to interpret than black-box models. On the other hand, black-box models have much higher predictive power than white-box models. So, there's always a trade-off that you need to make between predictive power and interpretability. It is important to understand the scenarios in which you can apply white-box and black-box models.

In Chapter 2, we will learn about characteristics that make white-box models inherently transparent and black-box models inherently opaque. We will learn how to interpret simple white-box models such as linear regression and decision trees and then switch gears to focus on GAMs. We will learn about the properties that give GAMs high predictive power and also learn how to interpret them. GAMs have quite high predictive power and are highly interpretable too – so you get more bang for your buck by using GAMs. At the time of writing this book, there were not a lot of practical resources on GAMs that give you a good understanding about the internals of the model and also on how to interpret them. To address this gap, a lot of attention will be given to GAMs in Chapter 2. The rest of the chapters will focus on black-box models.

There are two ways of interpreting black-box models. One way is to interpret model processing, i.e. understanding how the model processes the inputs and arrives at the final prediction. Chapters 3 to 5 will be focused on interpreting model processing. The other way is to interpret model representations and is applicable only to deep neural networks. Chapters 6 and 7 will be focused on interpreting model representations with the goal to understand what features or patterns have been learned by the neural network.

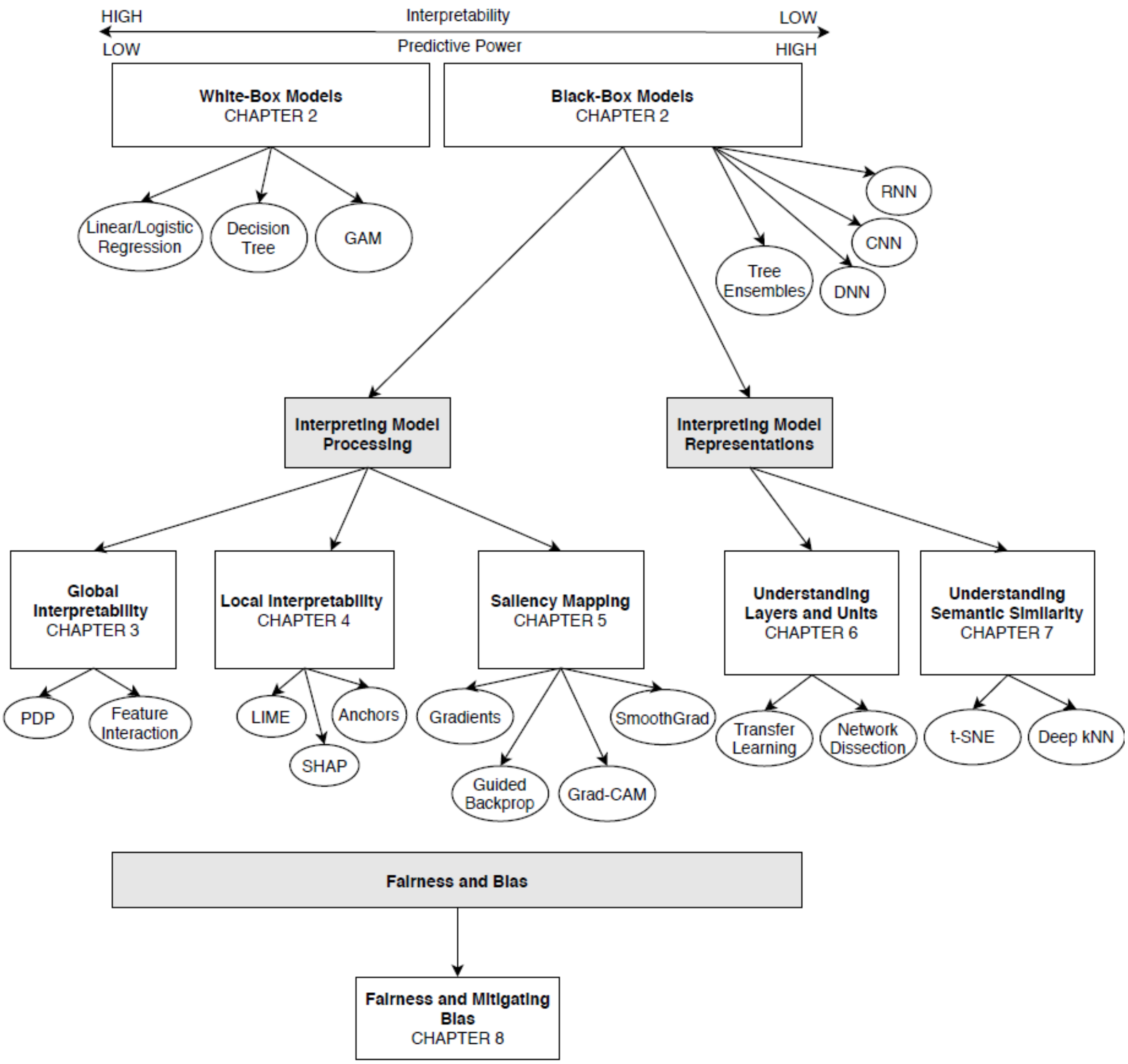


Figure 1.15: Map of interpretability techniques covered in this book

In Chapter 3, we will focus on a class of black-box models called tree ensembles. We will learn about their characteristics and what make them black box. We will also how to interpret them using post-hoc model agnostic methods that are global in scope. We will be specifically focusing on Partial Dependence Plots (PDPs), Individual Conditional Expectation (ICE) plots and feature interaction plots.

In Chapter 4, we will focus on deep neural networks specifically the vanilla fully connected neural networks. We will learn about characteristics that make these models black box and also how to interpret them using post-hoc model agnostic methods that are local in scope. We will specifically learn about techniques such as Local Interpretable Model-agnostic Explanations (LIME), Shapley Additive exPlanations (SHAP) and Anchors.

In Chapter 5, we will focus on convolutional neural networks which is a more advanced form of architecture used mainly for visual tasks such as image classification and object detection. We will learn how to visualize what the model is focusing on using saliency maps. We will learn techniques such as Gradients, Guided Backpropagation (Backprop for short), Gradient-weighted Class Activation Mapping (Grad-CAM), guided Grad-CAM and Smooth Gradients (SmoothGrad).

In Chapters 6 and 7, we will focus on convolutional neural networks and recurrent neural networks, another advanced architecture that is mainly used for language understanding. We will learn how to dissect the neural networks and understand what representations of the data are learned by the intermediate or hidden layers in the neural network. We will also learn how to visualize high-dimensional representations learned by the model using techniques like k-Nearest Neighbors (kNN) and t-distributed Stochastic Neighbor Embedding (t-SNE).

The book ends on the topic of building fair and unbiased models. In Chapter 8, we will learn about various definitions of fairness and how you could check if your model is biased. We will also learn techniques to mitigate bias using a neutralizing technique. We will also learn about a standardizing approach of documenting datasets using datasheets that will help improve transparency and accountability with the stakeholders and users of the system. By the end of this book, you will have various interpretability techniques in your toolkit. When it comes to model understanding, there is unfortunately no silver bullet. Not one interpretability technique is applicable for all scenarios. You will therefore need to look at the model using a few different lenses by applying multiple interpretability techniques. In this book, I will help you in identifying the right tools for the right scenarios.

1.7.1 What tools will I be using in this book?

In this book, we will be implementing the models and the interpretability techniques in the Python programming language. The main reason for choosing Python is because most of the state-of-the-art interpretability techniques are created and actively developed in this language. Figure 1.16 gives an overview of the tools used in this book. For representing data, we will be using Python data structures and common data science libraries such as *pandas* and *numpy*. To implement white-box models, we will be using the *scikit-learn* library for simpler linear regression and decision trees, and *pygam* for GAMs. For black-box models, we will be using *scikit-learn* for tree ensembles and *pytorch* or *tensorflow* for neural networks. For interpretability techniques used to understand model processing, we will be using the *matplotlib* library for visualization and open source libraries that implement techniques such as *PDP*, *LIME*, *SHAP*, *Anchors*, *Gradients, Guided Backprop, Grad-CAM* and *SmoothGrad*. To interpret model representations, we will be using tools that implement *kNN* and *tSNE* and visualize them using the *matplotlib* library. Finally, for mitigating bias, we will be using *pytorch* and *tensorflow* to implement the bias neutralizing technique and GANs for adversarial debiasing.

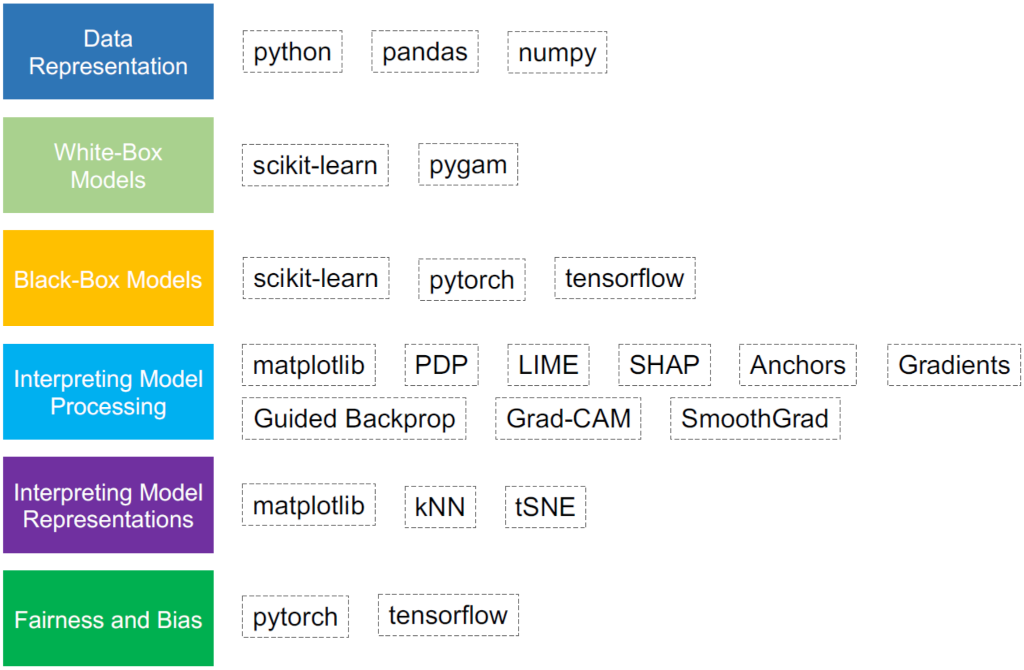


Figure 1.16: An overview of the tools used in this book

1.7.2 What do I need to know before reading this book?

This book is primarily focused on data scientists and engineers with experience programming in Python. A basic knowledge of common Python data science libraries such as numpy, pandas, matplotlib and scikit-learn will help, although this is not necessary. The book will show you how to use these libraries to load and represent data but will not give you an in-depth understanding of them as it is beyond the scope of this book.

The reader must be familiar with linear algebra specifically vectors and matrices, and operations on them such as dot product, matrix multiplication, transpose and inversion. The reader must also have good foundations in probability theory and statistics, specifically on the topics of random variables, basic discrete and continuous probability distributions, conditional probability and Bayes' Theorem. Basic knowledge of calculus is also expected, specifically single-variable and multi-variate functions and specific operations on them such as derivatives (gradients) and partial derivatives. Although this book does not focus too much on the mathematics behind model interpretability, having the above basic mathematical foundation is expected of data scientists and engineers interested in building machine learning models.

Basic knowledge of machine learning or practical experience training machine learning models is a plus, although this is not a hard requirement. This book will not cover machine learning in great depth as there are a lot of online resources and books that do justice to this topic. The book will however give you a basic understanding of the specific machine learning model being used and also show you how to train and evaluate them. The main focus is on the theory related to interpretability and how you can implement techniques to interpret the model after you have trained it.

1.8 Summary

- There are three broad types of machine learning systems – supervised learning, unsupervised learning and reinforcement learning. This book primarily focuses on interpretability techniques for supervised learning systems that include both regression and classification type of problems.
- When building AI systems, it is important to add interpretability, model understanding and monitoring to the process. If not, it could lead to disastrous consequences such as data leakage, bias, concept drift and a general lack of trust. Moreover, with the GDPR, there are legal reasons for including interpretability to your AI process.
- It is important to understand the difference between interpretability and explainability.
- Interpretability is the degree to which we can consistently estimate what a model will predict, understand how the model came up with the prediction and finally understand when the model has made a mistake. Interpretability techniques are building blocks that will help you get to explainability.
- Explainability goes beyond interpretability in that it helps us understand how and why a model came up with a prediction in a human-readable form. It makes use of interpretability techniques and also looks to other fields and areas such as Human-Computer Interaction (HCI), law and ethics.
- You need to be mindful of different personas using or building the AI system, as interpretability means different things to different people.
- Interpretability techniques can be intrinsic or post-hoc, model-specific or model-agnostic, local or global.
- Models that are inherently transparent are called white-box models and models that are inherently opaque are called black-box models. White-box models are much easier to interpret but generally have lower predictive power than black-box models.
- For black-box models, there are two broad classes of interpretability techniques – one that's focused on interpreting the model processing and the other that's focused on interpreting the representation learned by the model.

2
White-Box Models

This chapter covers:

- **Characteristics that make white-box models inherently transparent and interpretable**
- **How to interpret simple white-box models such as linear regression and decision trees**
- **What are Generalized Additive Models (GAMs) and properties that give them high predictive power and high interpretability**
- **How to implement GAMs and how to interpret them**
- **What are black-box models and characteristics that make them inherently opaque**

In order to build an interpretable AI system, it is important to understand different types of models that can be used to drive the AI system and techniques that can be applied to interpret them. In this chapter, I will cover three key white-box models – linear regression, decision trees and Generalized Additive Models (GAMs) – that are inherently transparent. You will learn how they can be implemented, when they can be applied and how they can be interpreted. I will also briefly introduce black-box models. You will learn when they can be applied and characteristics that make them hard to interpret. This chapter is focused on interpreting white-box models and the rest of the book will be dedicated to interpreting complex black-box models.

In Chapter 1, we learned how to build a robust, interpretable AI system. The process is shown again in Figure 2.1. The main focus of Chapter 2 and the rest of the book will be on implementing interpretability techniques to gain a much better understanding of machine learned models that cover both white-box and black-box models. The relevant blocks are highlighted in bold in Figure 2.1. We will apply these interpretability techniques during model development and testing. We will also learn about model training and testing, especially the implementation aspects. Since the model learning, testing and understanding stages are quite iterative, it is important to cover all three stages together. Readers who are already familiar with model training and testing are free to skip those sections and jump straight into interpretability.

When applying interpretability techniques in production, you will also need to consider building an explanation producing system to generate a human-readable explanation for the end users of your system. Explainability is however beyond the scope of this book and the focus will be exclusively on interpretability during model development and testing.

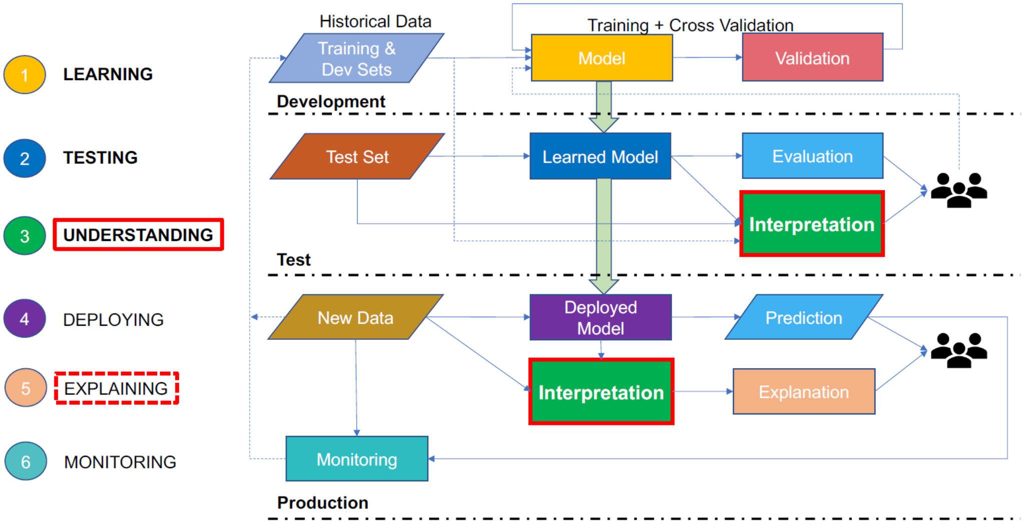


Figure 2.1: Process to build a robust AI system focusing mainly on Interpretation

2.1 White-Box Models

White-box models are inherently transparent and the characteristics that make them transparent are:

- The machine learning process is straightforward to understand, and you can clearly interpret how the input features get transformed into the output or target variable.
- You can identify the most important features to predict the target variable and those features are understandable.

Examples of white-box models are linear regression, logistic regression, decision trees and Generalized Additive Models (GAMs). Table 2.2.1 shows the machine learning tasks for which these models can be applied.

White-Box Model	Machine Learning Task(s)
Linear Regression	Regression

Logistic Regression	Classification
Decision Trees	Regression and Classification
Generalized Additive Models (GAMs)	Regression and Classification

Table 2.2.1: Mapping of White-Box Model to Machine Learning Task

In this chapter, we will be focusing on linear regression, decision trees and GAMs. In Figure 2.2, I have plotted these techniques on a 2D plane with interpretability on the x-axis and predictive power on the y-axis. As you go from left to right on this plane, models go from the low interpretability regime to the high interpretability regime. As you go from bottom to top on this plane, models go from the low predictive power regime to the high predictive power regime. Linear regression and decision trees are highly interpretable but have low to medium predictive power. GAMs on the other hand have quite high predictive power and are highly interpretable as well. The figure also shows black-box models in gray and italics. We will cover those in Section 2.6.

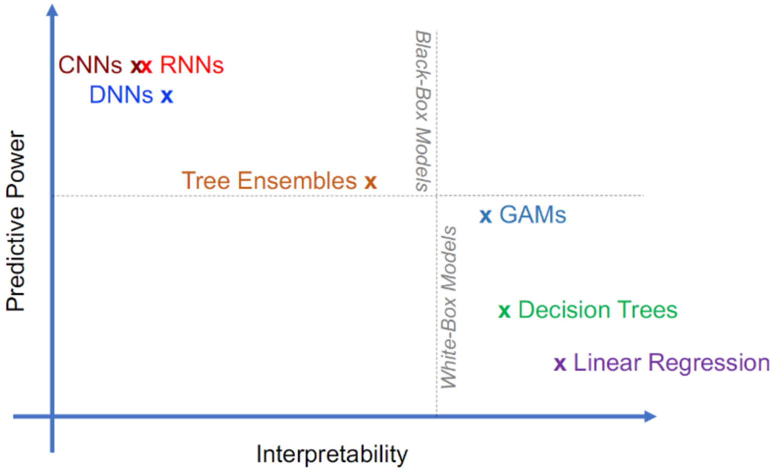


Figure 2.2: White-Box Models on the Interpretability v/s Predictive Power Plane

We will start off with interpreting the simpler linear regression and decision tree models, and then go deep into the world of GAMs. For each of these white-box models, we will learn how the algorithm works and characteristics that make them inherently interpretable. For white-box models, it is important to understand the details of the algorithm as it will help us interpret how the input features are transformed into the final model output or prediction. It will also help us quantify the importance of each input feature. For all models in this chapter and this book, we will learn how to train and evaluate them in Python first, before we dive into interpretability. As mentioned earlier, since the model learning, testing and understanding stages are quite iterative, it is important to cover all three stages together.

The simpler linear regression and decision tree models will be covered in Sections 2.3 and 2.4 respectively. For readers who are already familiar with how to train, evaluate and interpret these models are free to jump to the more advanced GAMs covered in Section 2.5. GAMs will be covered in great depth in this chapter as they provide more bang for our buck, given that it has reasonably high predictive power and is highly interpretable. At the time of writing this book, there were not a lot of practical resources on GAMs that give you a good understanding about the internals of the model and also on how to interpret them. This is another reason why GAMs are covered in great detail, to address this gap.

2.1.1 Diagnostics+ AI – Diabetes Progression

Let's look at white-box models in the context of a concrete example. Please recall the Diagnostics+ AI example from Chapter 1. The Diagnostics+ center would now like to venture into diabetes and determine the progression of the disease for their patients one year after the baseline measurement is taken. This is shown in Figure 2.3. The center has now tasked you as a newly minted data scientist to build a model for Diagnostics+ AI to predict diabetes progression one year out. This prediction will be used by doctors to determine a proper treatment plan for their patients. To gain the doctors' confidence in the model, it is important to not just provide an accurate prediction but also to be able to show how the model arrived at that prediction. So how would you begin with this task?

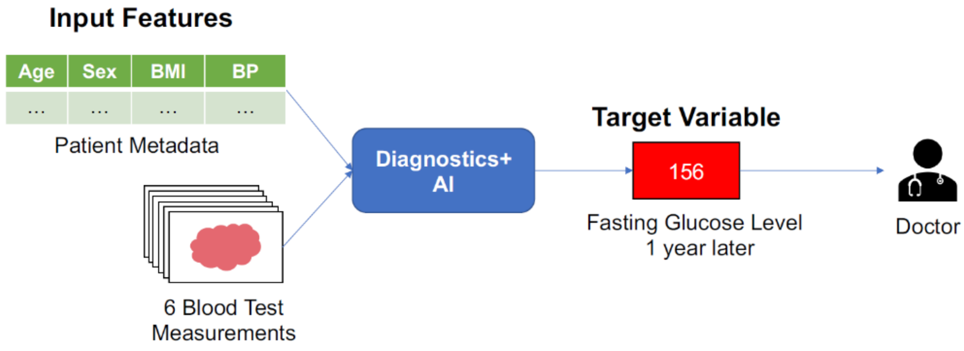


Figure 2.3: Diagnostics+ AI for Diabetes

First, let's look at what data is available. The Diagnostics+ center has collected data from around 440 patients, which consists of patient metadata like their age, sex, body mass index (BMI) and blood pressure (BP). Blood tests were also performed on these patients and the following six measurements were collected:

- LDL (the bad cholesterol)
- HDL (the good cholesterol)
- Total Cholesterol
- Thyroid Stimulating Hormone
- Low Tension Glaucoma
- Fasting Blood Glucose

The data also contains the fasting glucose levels for all patients one year after the baseline measurement was taken. This is the target for the model. Now how would you formulate this as a machine learning problem? Since labeled data is available where you are given 10 input features and one target variable that you have to predict, we can formulate this problem as a supervised learning problem. Since the target variable is real-valued or continuous, it is a regression task. The objective is to learn a function f that will help predict the target variable y given the input features x.

Let’s now load the data in Python and explore how correlated the input features are with each other and the target variable. If the input features are highly correlated with the target variable, then we can use them to train a model to make the prediction. If, however, they are not correlated with the target variable, then we will need to explore further to determine if there is some noise in the data. The data can be loaded in Python as follows.

```
from sklearn.datasets import load_diabetes #A
diabetes = load_diabetes() #B
X, y = diabetes[‘data’], diabetes[‘target’] #C
```

#A Import scikit-learn function to load open diabetes dataset
#B Load the diabetes dataset
#C Extract the features and the target variable

We will now create a pandas DataFrame, which is a two-dimensional data structure, that contains all the features and the target variable. The diabetes dataset provided by scikit-learn comes with feature names that are not easy to understand. The six blood samples measurements are named s1, s2, s3, s4, s5 and s6 and it is hard for us to understand what each feature is measuring. The documentation however provides this mapping and we will use that to rename the columns to something that is more understandable.

```
feature_rename = {'age': 'Age', #A
                  'sex': 'Sex', #A
                  'bmi': 'BMI', #A
                  'bp': 'BP', #A
                  's1': 'Total Cholesterol', #A
                  's2': 'LDL', #A
                  's3': 'HDL', #A
                  's4': 'Thyroid', #A
                  's5': 'Glaucoma', #A
                  's6': 'Glucose'} #A

df_data = pd.DataFrame(X, #B
                       columns=diabetes['feature_names']) #C
df_data.rename(columns=feature_rename, inplace=True) #D
```

```
df_data['target'] = y #E
```

#A: Mapping of feature names provided by scikit-learn to a more readable form
#B: Load all the features (X) into a DataFrame
#C: Use the scikit-learn feature names as column names
#D: Rename the scikit-learn feature names to a more readable form
#E: Include the target variable (y) as a separate column

Now let’s compute the pairwise correlation of columns so that we can determine how correlated each of the input features are with each other and the target variable. This can be done in pandas easily as follows.

```
corr = df_data.corr()
```

By default the `corr()` function in pandas computes the Pearson or standard correlation coefficient. This coefficient measures the linear correlation between two variables and has a value between +1 and -1. If the magnitude of the coefficient is above 0.7, then that means really high correlation. If the magnitude of the coefficient is between 0.5 and 0.7, then that means moderately high correlation. If the magnitude of the coefficient is between 0.3 and 0.5, then that means low correlation and a magnitude is less than 0.3, then that means little to no correlation. We can now plot the correlation matrix in Python as follows.

```
import matplotlib.pyplot as plt #A
import seaborn as sns #A
sns.set(style=’whitegrid’) #A
sns.set_palette(‘bright’) #A

f, ax = plt.subplots(figsize=(10, 10)) #B
sns.heatmap( #C
    corr, #C
    vmin=-1, vmax=1, center=0, #C
    cmap="PiYG", #C
    square=True, #C
    ax=ax #C
) #C
ax.set_xticklabels( #D
    ax.get_xticklabels(), #D
    rotation=90, #D
    horizontalalignment='right' #D
); #D
```

#A Import matplotlib and seaborn to plot the correlation matrix
#B Initialize a matplotlib plot with a predefined size
#C Use seaborn to plot a heatmap of the correlation coefficients
#D Rotate the labels on the x-axis by 90 degrees

The resulting plot is shown in Figure 2.4. Let’s first focus on either the last row or the last column in the figure. This shows us the correlation of each of the inputs with the target variable. We can see that seven features namely, BMI, BP, Total Cholesterol, HDL, Thyroid, Glaucoma and Glucose, have moderately high to high correlation with the target variable. We can also observe that the good cholesterol (HDL) also has a negative correlation with the progression of diabetes. This means that higher the HDL value, then lower the fasting glucose level for the patient one year out. The features therefore seem to have pretty good signal in being able to predict the disease progression and we can go ahead and train a model using them. As an exercise, observe how each of the features are correlated with each other. Total cholesterol for instance seems very highly correlated with the bad cholesterol, LDL. We will come back to this when we start to interpret the linear regression model in Section 2.3.1.

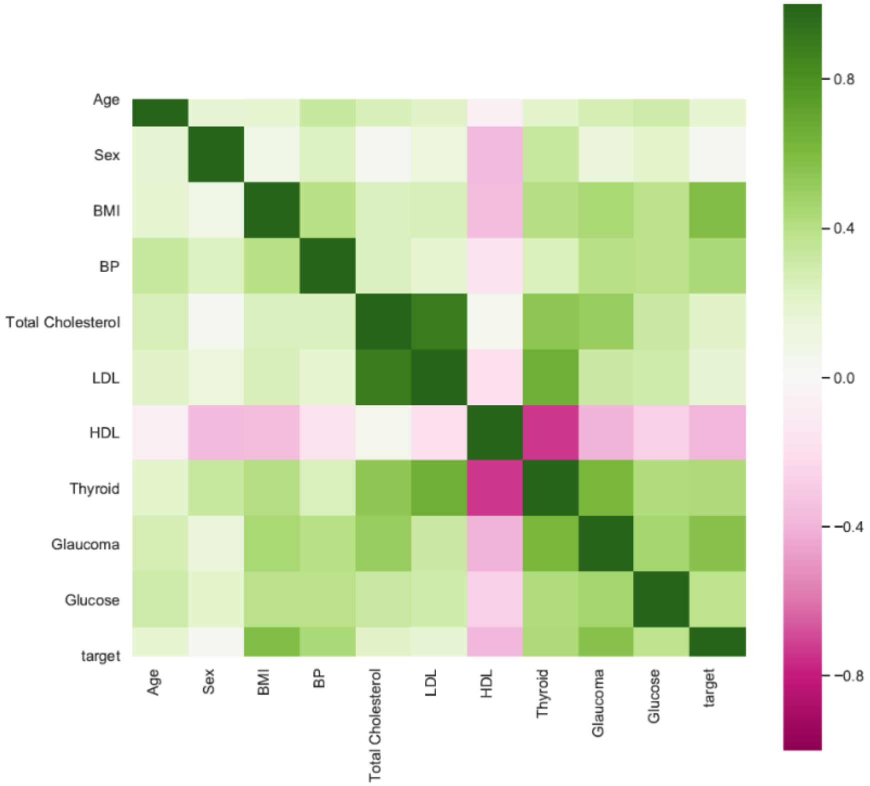


Figure 2.4: Correlation Plot of the Features and the Target Variable for the Diabetes Dataset

2.2 Linear Regression

Linear regression is one of the simplest models you can train for regression tasks. In linear regression, the function f is represented as a linear combination of all the input features. This is depicted in Figure 2.5. The known variables are shown in grey and the idea is to represent the target variable as a linear combination of the inputs. The unknown variables are the weights which must be learned by the learning algorithm.

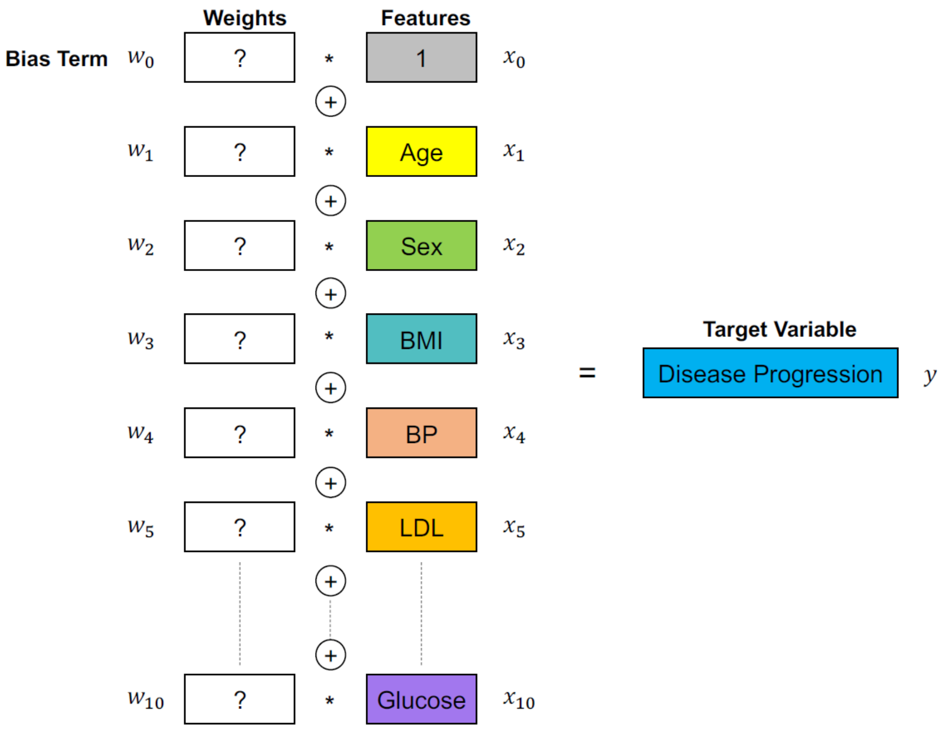


Figure 2.5: Disease progression represented as a linear combination of inputs

In general, the function f for linear regression is shown mathematically as follows, where n is the total number of features.

$$
\begin{aligned}
y &= w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n \\
&= w_0 + \sum_{i=1}^{n} w_i x_i
\end{aligned}
$$

The objective of the linear regression learning algorithm is to determine the weights that accurately predict the target variable for all patients in the training set. There are two techniques that can be applied here:

3. Gradient Descent
4. Closed-form Solution (e.g. Newton Equation)

Gradient descent is the technique that is commonly applied since it scales well to a large number of features and training examples. The general idea is to update the weights such that the squared error of the predicted target variable with respect to the actual target variable is minimized.

The objective of the gradient descent algorithm is to minimize the squared error or squared difference between the predicted target variable and the actual target variable across all the examples in the training set. This algorithm is guaranteed to find the optimum set of weights and since the algorithm minimizes the squared error, it is said to be based on least squares. A linear regression model can be easily trained using the scikit-learn package in Python. The code to train the model is shown below. It is important to note that the open diabetes dataset provided by scikit-learn is used here and this dataset has been standardized having 0 mean and unit variance for all the input features.

```
from sklearn.model_selection import train_test_split #A
from sklearn.linear_model import LinearRegression #B
import numpy as np #C

X_train, X_test, y_train, y_test = train_test_split(X, y, #D
                                                    test_size=0.2, #D
                                                    random_state=42) #D

lr_model = LinearRegression() #E

lr_model.fit(X_train, y_train) #F

y_pred = lr_model.predict(X_test) #G

mae = np.mean(np.abs(y_test - y_pred)) #H
```

#A Import scikit-learn function to split the data into training and test sets
#B Import scikit-learn class for Linear Regression
#C Import numpy to evaluate the performance of model
#D Split the data into training and test sets where 80% of the data is used for training and 20% of the data for testing. Ensure that the seed for the random number generator is set using the random_state parameter to ensure consistent train-test splits.
#E Initialize the linear regression model which is based on least squares
#F Learn the weights for the model by fitting on the training set
#G Using the learned weights predict the disease progression for patients in the test set
#H Evaluate the model performance using the Mean Absolute Error (MAE) metric

The performance of the trained linear regression model can be quantified by comparing the predictions with the actuals on the test set. There are multiple metrics that can be used here such as Root Mean Squared Error (RMSE), Mean Absolute Error (MAE) and Mean Absolute Percentage Error (MAPE). There are pros and cons to each of these metrics and it helps to quantify the performance using multiple metrics to measure the goodness of a model. Both MAE and RMSE are in the same units of the target variable and are easy to understand in that regard. The magnitude of the error, however, cannot be easily understood using these two metrics. For example, an error of 10 may seem small at first but if the actual value you are comparing with is say 100, then that error is not small in relation to that. This is why MAPE is useful, to understand these relative differences as the error is expressed in terms of percentage (%) error. The topic of measuring model goodness is important but is beyond the scope of this book. There are a lot of resources that you can find online. I have written a comprehensive two-part blog post to cover this topic.

The linear regression model trained above was evaluated using the MAE metric and the performance was obtained to be 42.8. But is this performance good? In order to check if the performance of the model is good, we need to compare it with a baseline. For Diagnostics+, the doctors have been using a baseline model that predicts the median diabetes progression across all patients. The MAE of this baseline model was obtained to be 62.2. If we now compare this baseline with the linear regression model, we notice a drop in MAE by 19.4 which is a pretty good improvement. We have now trained a decent model, but it doesn't tell us how the model arrived at the prediction and which input features are most important. I will cover this in the following section.

2.2.1 Interpreting Linear Regression

In the earlier section, we trained a linear regression model during model development and then evaluated the model performance during testing using the MAE metric. As a data scientist building Diagnostics+ AI, you now share these results with the doctors, and they are reasonably happy with the performance. But there is something missing. The doctors don't have a clear understanding of how the model arrived at the final prediction. Explaining the gradient descent algorithm does not help with this understanding as you are dealing with a pretty large feature space in this example, 10 input features in total. It is impossible to visualize how the algorithm converges to the final prediction in a 10-dimensional space. In general, the ability to describe and explain a machine learning algorithm does not guarantee interpretability. So, what is the best way of interpreting a model?

For linear regression, since the final prediction is just a weighted sum of the input features, all we have to look at are the learned weights. This is what makes linear regression a white-box model. What do the weights tell us? If the weight for a feature is positive, a positive change in that input will result in a proportional positive change in the output and a negative change in the input will result in a proportional negative change in the output. Similarly, if the weight is negative, a positive change in the input will result in a proportional negative change in the output and a negative change in the input will result in a proportional positive change in the output. Such a learned function is called a linear, monotonic function and it is shown in Figure 2.6.

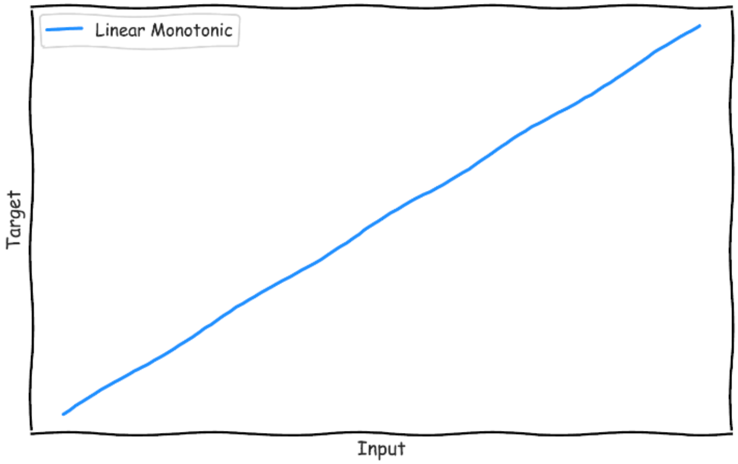


Figure 2.6: Representation of a Linear, Monotonic Function

We can also look at the impact or importance of a feature in predicting the target variable by looking at the absolute value of the corresponding weight. The larger the absolute value of the weight, the greater the importance. The weights for each of the 10 features are shown in descending order of importance in Figure 2.7.

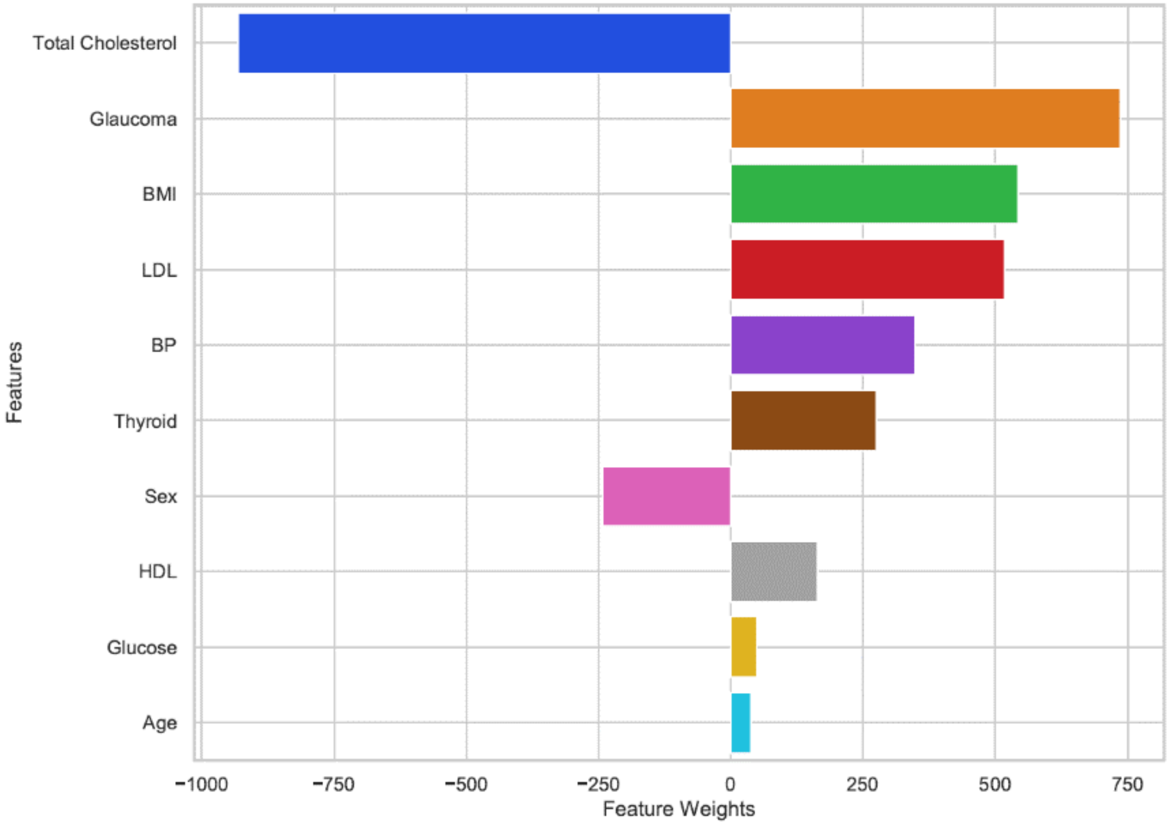


Figure 2.7: Feature importance for the Diabetes Linear Regression Model

The most important feature is the total cholesterol measurement. It has a large negative value for the weight. This means that a positive change in the cholesterol level has a large negative influence on predicting diabetes progression. This could be because total cholesterol also accounts for the good kind of cholesterol.

If we now look at the bad cholesterol or LDL feature, it has a large positive weight and it is also the fourth most important feature in predicting the progression of diabetes. This means that a positive change in LDL cholesterol level, results in a large positive influence in predicting diabetes one year out. The good cholesterol or HDL feature has a small positive weight and is the third least important feature. Why is that? Recall the exploratory analysis that we did in Section 2.2 where we plotted in correlation matrix in Figure 2.4. If we observe the correlation between total cholesterol, LDL and HDL, we see a very high correlation between total cholesterol and LDL and moderately high correlation between total cholesterol and HDL. Because of this correlation, the HDL feature is deemed redundant by the model.

It also looks like the baseline glucose measurement for the patient has very small impact on predicting the progression of diabetes a year out. If we again go back to the correlation plot shown in Figure 2.4, we can see that glucose measurement is very highly correlated with the baseline glaucoma measurement (the second most important feature for the model) and pretty highly correlated with total cholesterol (the most important feature for the model). The model therefore treats glaucoma as a redundant feature since a lot of the signal is obtained from the total cholesterol and glaucoma features.

If an input feature is highly correlated with one or more other features, they are said to be multicollinear. **Multicollinearity** could be detrimental to the performance of a linear regression model based on least squares. Let’s suppose we use two features x1 and x2 to predict the target variable y. In a linear regression model, we are essentially estimating weights for each of the features that will help predict the target variable such that the squared error is minimized. Using least squares, the weight for feature x1 or the effect of x1 on the target variable y is estimated by holding x2 constant. Similarly, the weight for x2 is estimated by holding x1 constant. If x1 and x2 are collinear, then they vary together, and it becomes very difficult to accurately estimate their effects on the target variable. One of the features becomes completely redundant for the model. We have seen the effects of collinearity on our diabetes model above where features such as HDL and Glucose that are pretty highly correlated with the target variable have very low importance in the final model. The problem of multicollinearity can be overcome by removing the redundant features for the model. As an exercise, I highly recommend doing that to see if you can improve the performance of the linear regression model.

In the process of training a machine learning model, it is important to explore the data first and determine how correlated features are with each other and with the target variable. The problem of multicollinearity must be uncovered earlier in the process before training the model but if it has been overlooked, interpreting the model will help expose such issues. The plot in Figure 2.7 can be generated in Python using the following code snippet.

```
import numpy as np #A
import matplotlib.pyplot as plt #B
import seaborn as sns #B
sns.set(style=’whitegrid’) #B
sns.set_palette(‘bright’) #B

weights = lr_model.coef_ #C

feature_importance_idx = np.argsort(np.abs(weights))[::-1] #D
feature_importance = [feature_names[idx].upper() for idx in
                      feature_importance_idx] #E
feature_importance_values = [weights[idx] for idx in
                             feature_importance_idx] #E

f, ax = plt.subplots(figsize=(10, 8)) #F
sns.barplot(x=feature_importance_values, y=feature_importance, ax=ax) #F
ax.grid(True) #F
ax.set_xlabel('Feature Weights') #F
ax.set_ylabel('Features') #F
```

#A Import numpy to perform operation on vectors in an optimized way
#B Import matplotlib and seaborn to plot the feature importance
#C Obtain the weights from the linear regression model trained earlier using the coef_ parameter
#D Sort the weights in descending order of importance and get their indices
#E Using the ordered indices to get the feature names and the corresponding weight values
#F Generate the plot shown in Figure 2.7

2.2.2 Limitations of Linear Regression

In the previous section, we saw how easy it is to interpret a linear regression model. It is highly transparent and easy to understand. It however has poor predictive power especially in cases where the relationship between the input features and target is non-linear. Consider an example as shown in Figure 2.8.

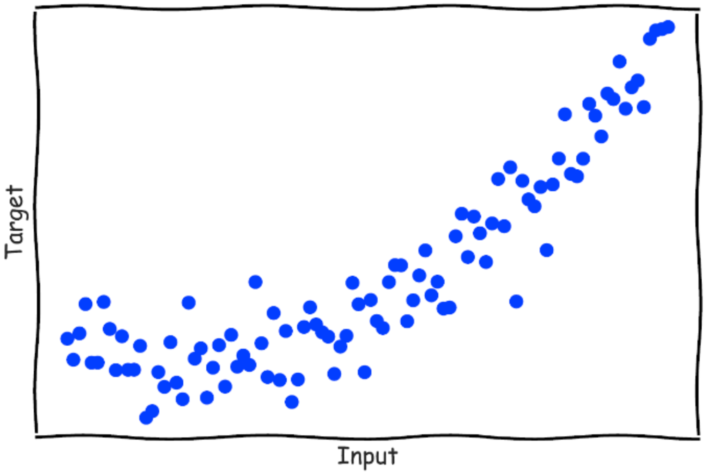


Figure 2.8: Illustration of a Non-Linear Dataset

If we were to fit a linear regression model to this dataset, we will get a straight-line linear fit as shown in Figure 2.9. As you can see the model does not properly fit the data and does not capture the non-linear relationship. This limitation of linear regression is called **underfitting** and the model is said to have **high bias**. In the following sections, we will see how this problem can be overcome by using more complex models with higher predictive power.

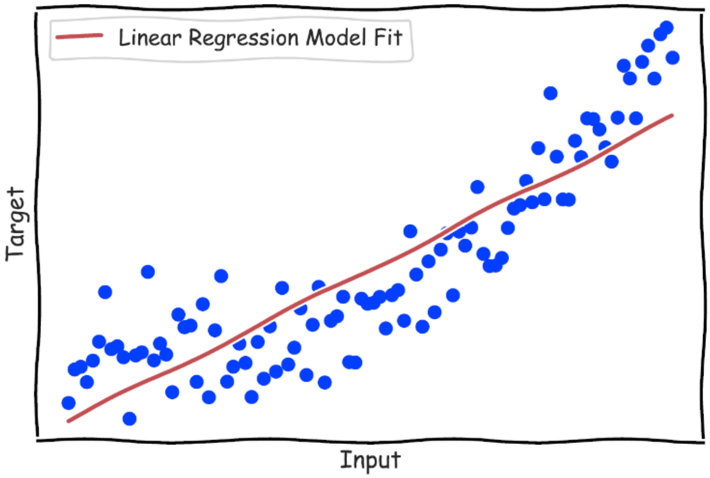


Figure 2.9: Problem of Underfitting (High Bias)

2.3 Decision Trees

Decision trees are a great machine learning algorithm that can be used to model complex non-linear relationships. It can be applied for both regression and classification tasks. It has relatively higher predictive power than linear regression and is highly interpretable too. The basic idea behind a decision tree is to find optimum splits in the data that best predict the output or target variable. In Figure 2.10, I have illustrated this by considering only two features, BMI and Age. The decision tree splits the dataset into five groups in total, three age groups and two BMI groups.

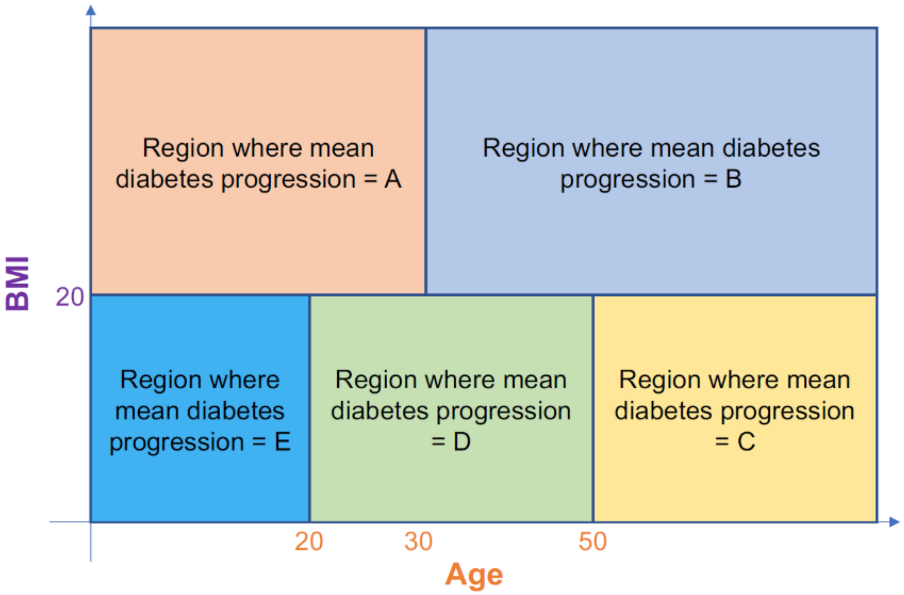

Region where mean diabetes progression = A
Region where mean diabetes progression = B
BMI
20
Region where mean diabetes progression = E
Region where mean diabetes progression = D
Region where mean diabetes progression = C
20
30
50
Age

Figure 2.10: Decision Tree Splitting Strategy

The algorithm that is commonly applied in determining the optimum splits is the Classification and Regression Tree (CART) algorithm. This algorithm first chooses a feature and a threshold for that feature. Based on that feature and threshold, the algorithm splits the dataset into two subsets:

1. Subset 1 where the value of the feature is less than or equal to the threshold, and
2. Subset 2 where the value of the feature is greater than the threshold

The algorithm picks the feature and threshold that minimizes a cost function or impurity measure. For regression tasks, this impurity measure is typically Mean Squared Error (MSE) and for classification tasks, it is typically either Gini Impurity or Entropy. The algorithm then continues to recursively split the data until the impurity measure is reduced further or when a maximum depth is reached. The splitting strategy in Figure 2.10 is shown as a binary tree in Figure 2.11.

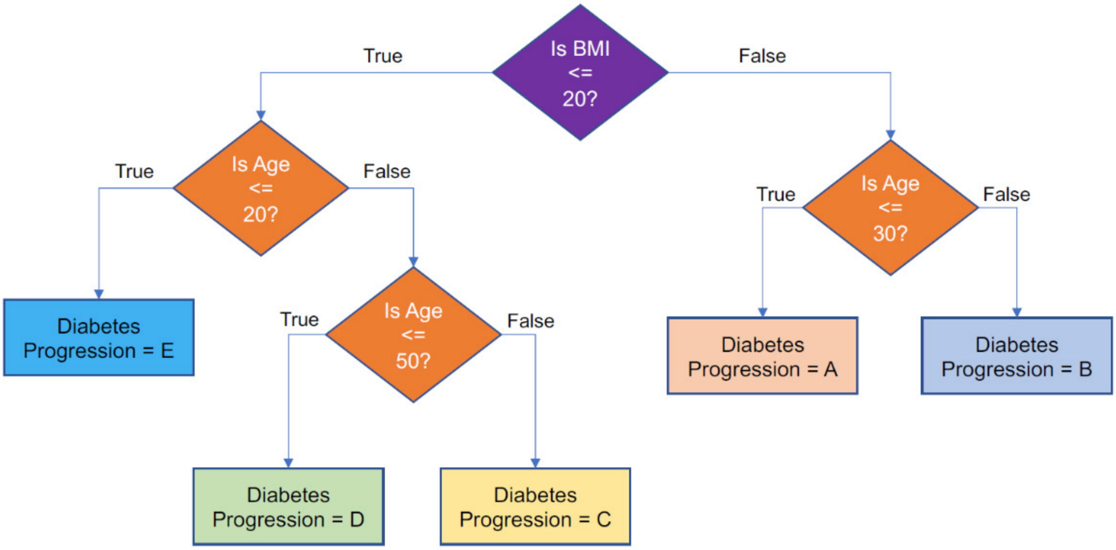


Figure 2.11: Decision Tree data splitting visualized as a Binary Tree

A decision tree model can be trained in Python using the scikit-learn package as follows. The code to learn then open diabetes dataset and to split it into the training and test sets is the same as the one used for linear regression in Section 2.3. So, this code is not repeated below.

```
from sklearn.tree import DecisionTreeRegressor #A

dt_model = DecisionTreeRegressor(max_depth=None, random_state=42) #B

dt_model.fit(X_train, y_train) #C

y_pred = dt_model.predict(X_test) #D

mae = np.mean(np.abs(y_test - y_pred)) #E
```

#A Import the scikit-learn class for decision tree regressor
#B Initialize the decision tree regressor. It is important to set the random_state to ensure that consistent, reproducible results can be obtained.
#C Train the decision tree model
#D Use the trained decision tree model to predict the disease progression for patients in the test set
#E Evaluate the model performance using the Mean Absolute Error (MAE) metric

The decision tree model trained above was evaluated using the MAE metric and the performance was obtained to be 54.7. If we tune the *max_depth* hyperparameter and set it to 3, we can improve the MAE performance further to 48.6. This performance is however poorer than the regression model trained in Section 2.2. I will discuss the reasons for this in Section 2.3.2 but first let’s look at how to interpret a decision tree in the following section.

Decision Tree for Classification Tasks

As mentioned in this section, decision trees can also be used for classification tasks. In the CART algorithm, Gini impurity or entropy is used as the cost function. In scikit-learn, you can easily train a decision tree classifier as follows.

```
from sklearn.tree import DecisionTreeClassifier
dt_model = DecisionTreeClassifier(criterion='gini', max_depth=None)
dt_model.fit(X_train, y_train)
```

The `criterion` parameter in the `DecisionTreeClassifier` can be used to specify the cost function for the CART algorithm. By default, it is set to `gini` but it can be changed to `entropy`.

2.3.1 Interpreting Decision Trees

Decision trees are great at modeling non-linear relationships between the input and the output. By finding splits in the data across features, the model tends to learn a function that is non-linear in nature. The function could be monotonic where a change in the input results in a change in the output in the same direction or non-monotonic where a change in the input could result in a change in the output in any direction and at a varying rate. This is illustrated in Figure 2.12.

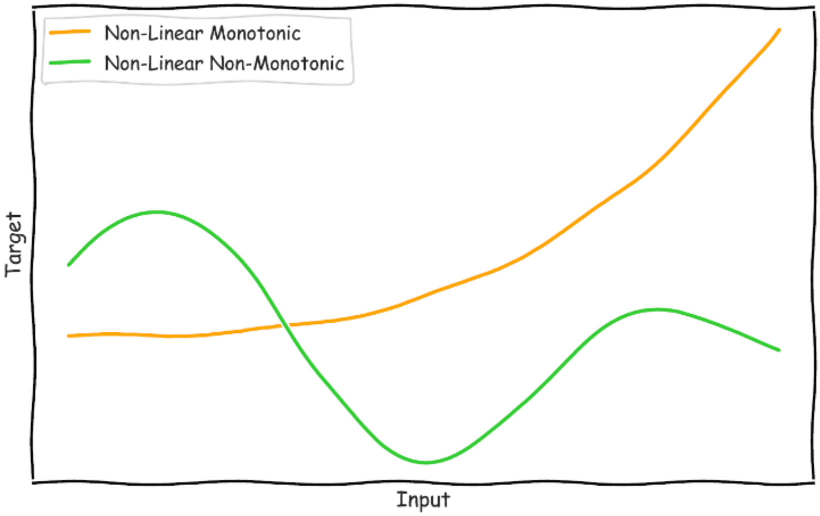


Figure 2.12: Representation of Non-Linear, Monotonic and Non-Monotonic Functions

So how do we interpret such a learned non-linear function? As seen in the previous section, a decision tree can be visualized as a bunch of if-else conditions strung together where each condition splits the data into two. Such a model can be easily visualized as a binary tree as illustrated in Figure 2.11. For the decision tree model trained for diabetes, the visualization of the binary tree is shown in Figure 2.13. The tree can be interpreted as follows.

Starting at the root of the tree, check if the normalized BMI is <= 0. If true, go to the left part of the tree. If false, go to the right part of the tree. Since we are starting at the root of the tree, this node accounts for 100% of the data. This is why *samples* is equal to 100%. Also, if we were to set the *max_depth* to 0 and predict the disease progression then we would use the average value of all the samples in the data which is 153.7, represented as *value* in the tree. By predicting 153.7, we would get an MSE of 6076.4.

If the normalized BMI is <= 0, then we go to the left part of the tree and check if the normalized Glaucoma is <= 0. If BMI is <= 0, we would account for approximately 59% of the data and the MSE would reduce from 6076.4 for the parent node to 3612.7. We can repeat this process until we have reached the leaf nodes in the tree. If we look at say the right-most leaf node, this corresponds to the following condition: If BMI > 0 and BMI > 0.1 and LDL > 0, then predict 225.8 for 2.3% of the data, resulting in an MSE of 2757.9.

Please note that the *max_depth* for the decision tree in Figure 2.13 was set to 3. The complexity of this tree will increase as *max_depth* increases or as the number of input features increases.

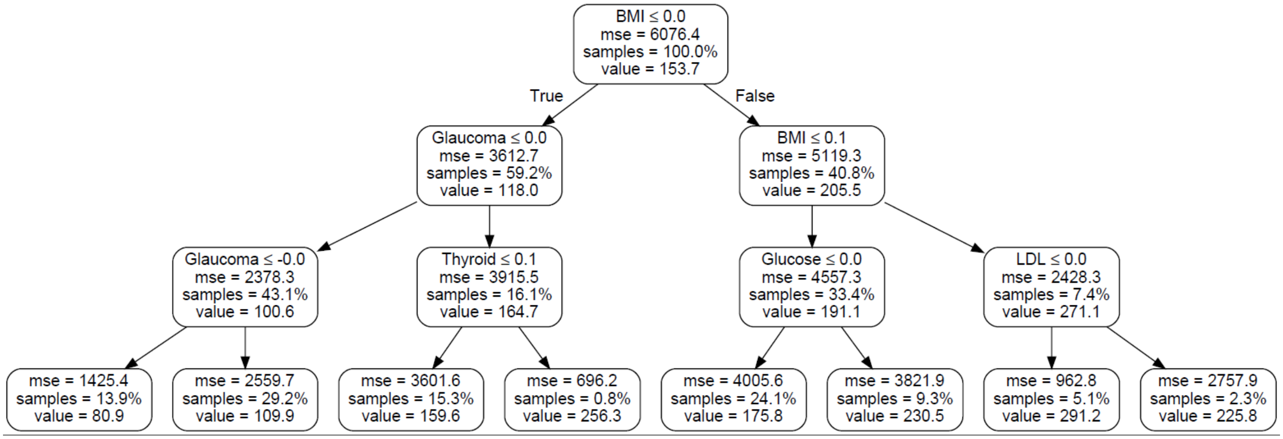


Figure 2.13: Visualization of the Diabetes Decision Tree Model

The visualization in Figure 2.13 can be generated in Python using the following code snippet.

```
from sklearn.externals.six import StringIO #A
from IPython.display import Image #A
from sklearn.tree import export_graphviz #A
import pydotplus #A

diabetes_dt_dot_data = StringIO() #B
export_graphviz(dt_model,
                out_file=diabetes_dt_dot_data,
                filled=False, rounded=True,
                feature_names=feature_names,
                proportion=True,
                precision=1,
                 special_characters=True) #C
dt_graph = pydotplus.graph_from_dot_data(diabetes_dt_dot_data.getvalue()) #D
Image(dt_graph.create_png()) #E
```

#A Import all the necessarily libraries to generate and visualize the binary tree
#B Initialize a string buffer to store the binary tree / graph in DOT format
#C Export the decision tree model as a binary tree in DOT format
#D Generate an image of the binary tree using the DOT format string
#E Visualize the binary tree using the Image class

Since decision trees learn a non-linear relationship between the input features and the target, it is hard to understand what effect changes to each of the inputs has on the output. It is not as straightforward as linear regression. We can however compute the relative importance of each of the features in predicting the target at a global level. In order to compute the feature importance, we first need to compute the importance of a node in the binary tree. The importance of a node is computed as the decrease in the cost function or impurity measure for that node weighted by the probability of reaching that node in the tree. This is shown mathematically below.

$$\underbrace{I_k^{\text{node}}}_{\substack{\text{Importance}\\ \text{of node } k}} = \overbrace{p_k}^{\substack{\text{Proportion}\\ \text{of samples}\\ \text{to reach}\\ \text{node } k}} \cdot \underbrace{m_k}_{\substack{\text{Impurity}\\ \text{measure}\\ \text{of node k}}} - \overbrace{p_k^{(left)}}^{\substack{\text{Proportion}\\ \text{of samples}\\ \text{to reach}\\ \text{left subtree}\\ \text{of node } k}} \cdot \underbrace{m_k^{(left)}}_{\substack{\text{Impurity}\\ \text{measure}\\ \text{of left}\\ \text{subtree of}\\ \text{node } k}} - \overbrace{p_k^{(right)}}^{\substack{\text{Proportion}\\ \text{of samples}\\ \text{to reach}\\ \text{right subtree}\\ \text{of node } k}} \cdot \underbrace{m_k^{(right)}}_{\substack{\text{Impurity}\\ \text{measure}\\ \text{of right}\\ \text{subtree of}\\ \text{node } k}}$$

We can then compute the feature importance by summing up the importance of the nodes that split on that feature normalized by the importance of all the nodes in the tree. This is shown mathematically below. The feature importance for decision tree is there between 0 and 1, where a higher value implies greater importance.

$$\underbrace{I_i^{\text{feature}}}_{\substack{\text{Importance}\\ \text{of feature } i}} = \frac{\overbrace{\sum_{j \epsilon \mathbb{J}} I_j^{\text{node}}}^{\substack{\text{Sum of importance}\\ \text{of all nodes } j\\ \text{that split on feature } i}}}{\underbrace{\sum_{k \epsilon \mathbb{K}} I_k^{\text{node}}}_{\substack{\text{Sum of importance}\\ \text{of all nodes } k\\ \text{in the decision tree}}}}$$

In Python, the feature importance can be obtained from the scikit-learn decision tree model and plotted as follows.

```
weights = dt_model.feature_importances_ #A

feature_importance_idx = np.argsort(np.abs(weights))[::-1] #B
feature_importance = [feature_names[idx].upper() for idx in
                      feature_importance_idx] #C
feature_importance_values = [weights[idx] for idx in
                             feature_importance_idx] #C

f, ax = plt.subplots(figsize=(10, 8)) #D
sns.barplot(x=feature_importance_values, y=feature_importance, ax=ax) #D
ax.grid(True) #D
ax.set_xlabel('Feature Weights') #D
ax.set_ylabel('Features') #D
```

#A Get feature importances from the trained decision tree model
#B Sort indices of feature weights in descending order of importance
#C Get the feature names and feature weights in descending order of importance
#D Generate the plot shown in Figure 2.14

The features ordered in descending order of importance and their corresponding weights are shown in Figure 2.14. As can be seen from the figure, the order of important features is different from linear regression. The most important feature is BMI accounting for roughly 42% of the overall model importance. The Glaucoma measurement is the next most important feature accounting for roughly 15% of the model importance. These importance values are useful to determine what features have the most signal in predicting the target variable. Decision trees are immune to the problem of multicollinearity as the algorithm picks the feature that is highly correlated with the target and that most reduces the cost function or impurity. As a data scientist, it is important to visualize the learned decision tree as shown in Figure 2.13 as this will help you understand how the model arrived at the final prediction. You could reduce the complexity of the tree by setting the *max_depth* hyperparameter or by pruning the number of features you feed into the model. You can determine what features to prune by visualizing the global feature importance as shown in Figure 2.14.

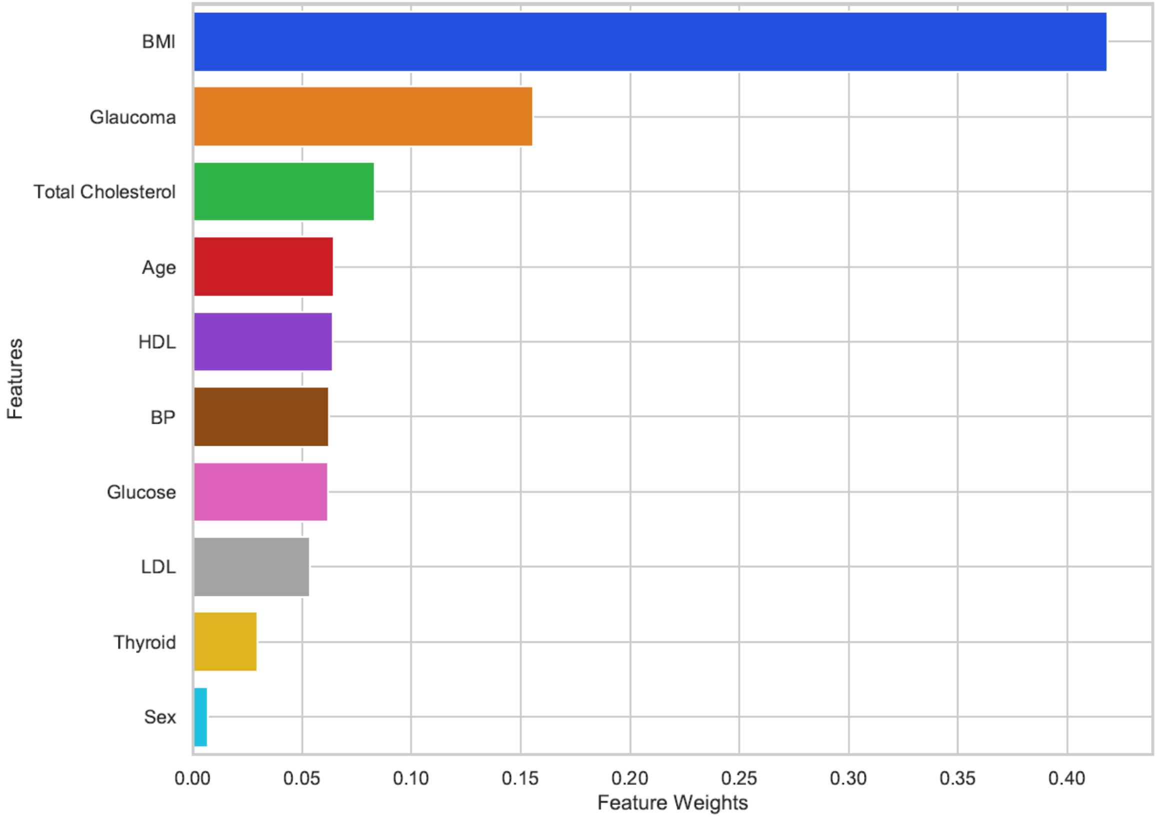


Figure 2.14: Diabetes feature importance for Decision Tree

2.3.2 Limitations of Decision Trees

Decision trees are quite versatile as they can be applied to both regression and classification tasks, and also have the ability to model non-linear relationships. The algorithm however is prone to the problem of **overfitting** and the model is said to have **high variance**.

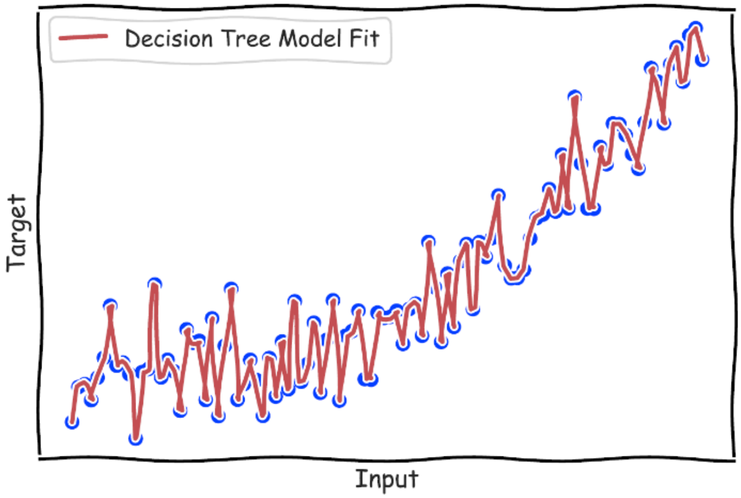


Figure 2.15: Problem of Overfitting (High Variance)

The problem of overfitting occurs when the model fits the training data almost perfectly thereby not generalizing well to data that it hasn't seen before, i.e. the test set. This is illustrated in Figure 2.15. When a model overfits, you will notice really good performance on the training set but poor performance on the test set. This could explain why the decision tree model trained on the diabetes dataset performed poorer than the linear regression model.

The problem of overfitting can be overcome by tuning certain hyperparameters in the decision tree like *max_depth* and the minimum number of samples required for the leaf nodes. By looking at the visualization of the decision tree model in Figure 2.13, there is one leaf node which accounts for only 0.8% of the samples. This means that the prediction for this node is based on roughly only 3 patient data. By increasing the minimum number of samples required to 5 or 10, we could improve the performance of the model on the test set.

2.4 Generalized Additive Models (GAMs)

Diagnostics+ and the doctors are reasonably happy with the two models built so far but the performance is not that good. By interpreting the models, we have also uncovered some shortcomings with these models. The linear regression model does not seem to handle features that are highly correlated with each other such as total cholesterol, LDL and HDL. The decision tree model performs worse than linear regression and it seems to have overfit on the training data.

Let's look at one specific feature from the diabetes data. Figure 2.16 shows a contrived example of a non-linear relationship between age and the target variable, where both variables are normalized. How would you best model this relationship without overfitting? One possible approach is to extend the linear regression model where the target variable is modelled as an n^th degree polynomial of the feature set. This form of regression is called polynomial regression.

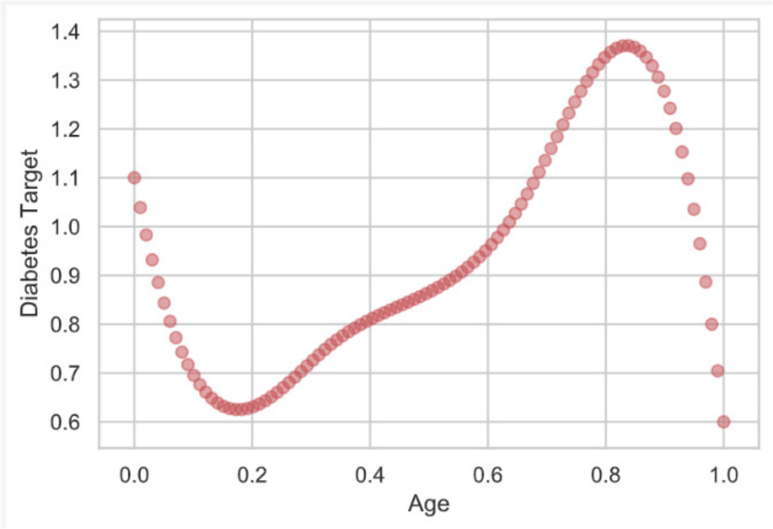


Figure 2.16: Illustration of a Non-Linear Relationship for Diagnostics+ AI

Polynomial regression for various degree polynomials is shown in the equations below. In the equations below, we are considering only one feature x_1 to model the target variable y. The degree 1 polynomial is the same as linear regression. For the degree 2 polynomial, we would add an additional feature which is the squared of x_1. For the degree 3 polynomial, we would add two additional features – one which is the squared of x_1 and the other which is the cubed of x_1.

$$y = w_0 + w_1 x_1 \text{ (Degree 1)}$$
$$y = w_0 + w_1 x_1 + w_2 x_1^2 \text{ (Degree 2)}$$
$$y = w_0 + w_1 x_1 + w_2 x_1^2 + w_3 x_1^3 \text{ (Degree 3)}$$

The weights for the polynomial regression model can be obtained using the same algorithm as linear regression, i.e. the method of least squares using gradient descent. The best fit learned by each of the three polynomials is shown in Figure 2.17. We can see that the degree 3 polynomial fits the raw data better than degrees 2 and 1. We can interpret a polynomial regression model the same way as linear regression since the model is essentially a linear combination of the features including the higher degree features.

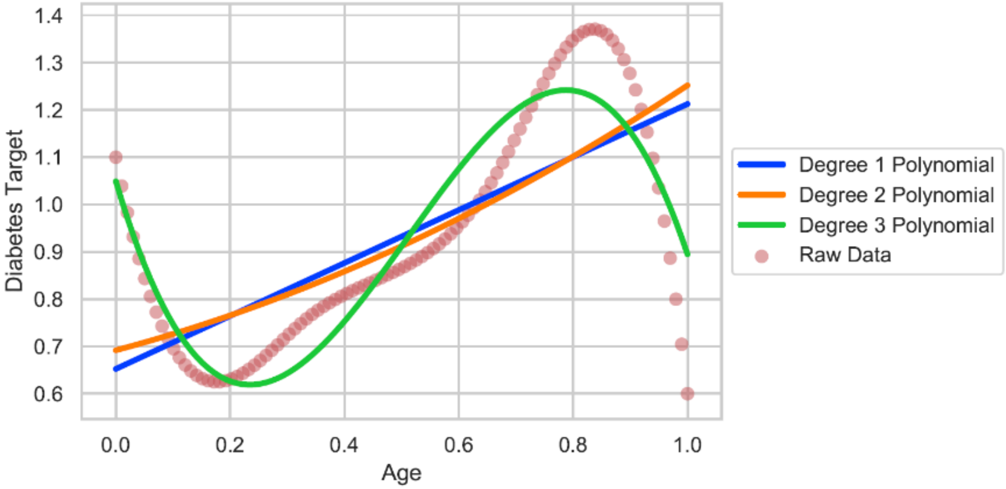


Figure 2.17: Polynomial Regression for Modeling a Non-Linear Relationship

Polynomial regression however has some limitations. The complexity of the model increases as the number of features or the dimension of the feature space increases. It therefore has a tendency to overfit on the data. It is also hard to determine the degree for each feature in the polynomial especially in a higher dimensional feature space.

So, what model can be applied to overcome all these limitations and that's interpretable? Enter, Generalized Additive Models (GAMs)! GAMs are models with medium to high predictive power and that are highly interpretable. Non-linear relationships are modeled using smoothing functions for each feature and by adding all of them. This is shown by the equation below.

$$y = w_0 + \underbrace{f_1(x_1)}_{\substack{\text{Smoothing Function}\\ \text{for Feature } x_1}} + \underbrace{f_2(x_2)}_{\substack{\text{Smoothing Function}\\ \text{for Feature } x_2}} + \ldots + \underbrace{f_n(x_n)}_{\substack{\text{Smoothing Function}\\ \text{for Feature } x_n}}$$

In the equation above, each feature has its own associated smoothing function that best models the relationship between that feature and the target. There are many types of smoothing functions that you can choose from but a widely used smoothing function is called **regression splines** as they are practical and computationally efficient. I will be focusing on regression splines in this book. Let's now go deep into the world of GAMs using regression splines!

2.4.1 Regression Splines

Regression splines are represented as a weighted sum of linear or polynomial functions. These polynomial functions are also known as basis functions. This is shown mathematically below. In the equation, f_j is the function that models the relationship between the feature x_j and the target variable. This function is represented as a weighted sum of basis functions where the weight is represented as w_k and the basis function is represented as b_k. In the context of GAMs, the function f_j is called a smoothing function.

$$f_j(x_j) = \underbrace{\sum_{k=1}^{K} w_k b_k(x_j)}_{\substack{\text{Smoothing Function} \\ \text{represented as a} \\ \text{weighted sum of basis functions}}}$$

Now, what is a basis function? A basis function is a family of transformations that can be used to capture a general shape or non-linear relationship. For regression splines, as the name suggests, splines are used as the basis function. A spline is a polynomial of degree n with $n-1$ continuous derivates. It will be much easier to understand splines using an illustration. Figure 2.18 shows splines of various degrees. The top left graph shows the simplest spline of degree 0, from which higher degree splines can be generated. As you can see from the top left graph, six splines have been placed on a grid. The idea is to split the distribution of the data into portions and fit a spline on each of those portions. So, in this illustration, the data has been split into six portions and we are modeling each portion as a degree 0 spline.

A degree 1 spline, shown in the top right graph, can be generated by convolving a degree 0 spline with itself. Convolution is a mathematical operation that takes in two functions and creates a third function that represents the correlation of the first function and a delayed copy of the second function. When we convolve a function with itself, we are essentially looking at the correlation of the function with a delayed copy of itself. There is a nice blog post by Christopher Olah on convolutions. This animation on Wikipedia will also give you a good intuitive understanding. By convolving a degree 0 spline with itself, we get a degree 1 spline which is triangular shaped, and this has a continuous 0th order derivative.

If we now convolve a degree 1 spline with itself, we will get a degree 2 spline shown in the bottom left graph. This degree 2 spline has a continuous 1st order derivative. Similarly, we can get a degree 3 spline by convolving a degree 2 spline and this has a continuous 2nd order derivative. In general, a degree n spline has a continuous $n-1$ derivative. In the limit as n approaches infinity, we will obtain a spline that has the shape of a Gaussian distribution. In practice, **degree 3** or **cubic splines** are used as it can capture most general shapes.

As mentioned earlier, in Figure 2.18, we have divided the distribution of data into six portions and have placed six splines on the grid. In the mathematical equation earlier, the number of portions or splines is represented as variable K. The idea behind regression splines is to learn the weights for each of the splines so that you can model the distribution of the data in each of the portions. The number of portions or splines in the grid, K, is also called **degrees of freedom**. In general, if we place these K splines on a grid, we will have $K+3$ points of division, also known as **knots**.

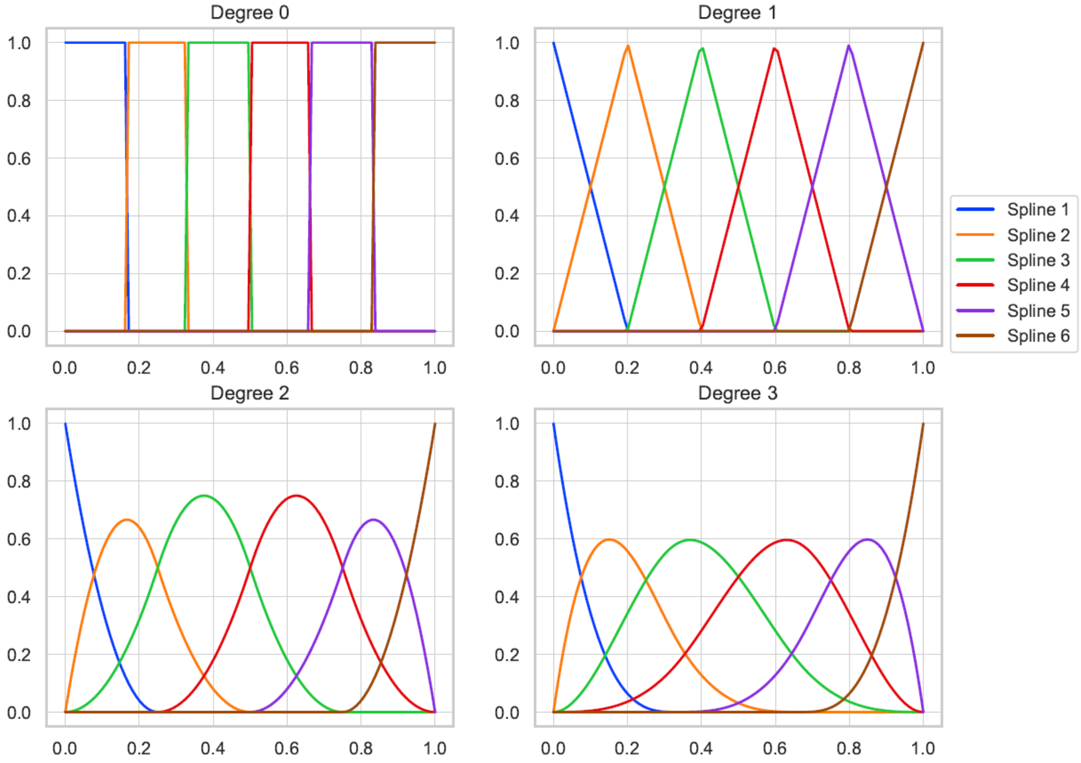


Figure 2.18: Illustration of Degree 0, Degree 1, Degree 2 and Degree 3 Splines

Let’s now zoom into cubic splines as shown in Figure 2.19. We can see that there are 6 splines or 6 degrees of freedom resulting in 9 points of division or knots.

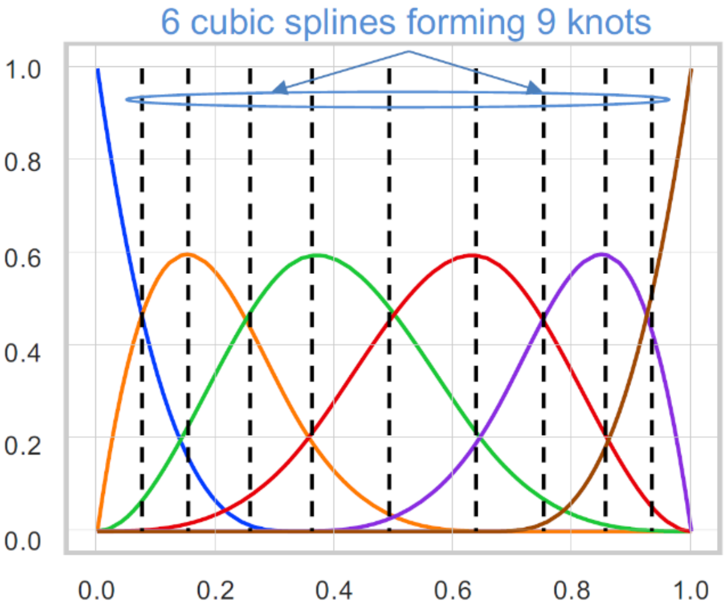


Figure 2.19: Illustration of Splines and Knots

Now to capture a general shape we will need to take a weighted sum of the splines. We will use cubic splines here. In Figure 2.20, we are using the same 6 splines overlaid to create 9 knots. For the graph on the left, I have set the same weights for all 6 splines. As you can imagine, if we take an equally weighted sum of all 6 splines, we will get a a horizontal straight line. This is an illustration of a poor fit to the raw data. For the graph on the right however, I have taken an unequal weighted sum of the 6 splines generating a shape that perfectly fits the raw data. This shows the power of regression splines and GAMs. By increasing the number of splines or by dividing the data into more portions, we will have the ability to model more complex non-linear relationships. In GAMs based on regression splines, we individually model non-linear relationships of each feature with the target variable, and then add them all up to come up with the final prediction.

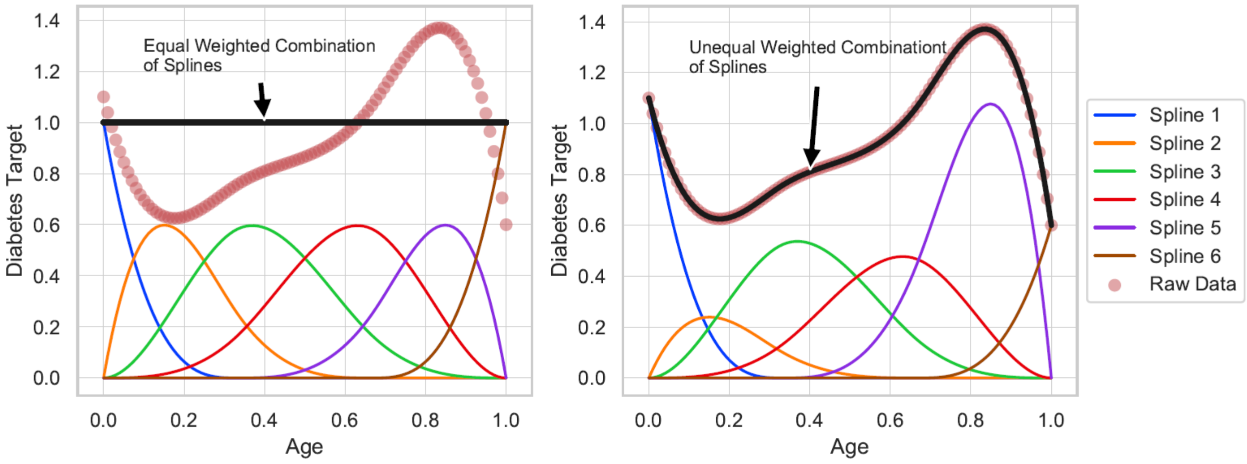


Figure 2.20: Splines for Modeling a Non-Linear Relationship

In Figure 2.20, the weights were determined using trial and error to best describe the raw data. But, how do you algorithmically determine the weights for a regression spline that best captures the relationship between the features and the target? Recall from the start of this section that a regression spline is a weighted sum of basis functions or splines. This is essentially a linear regression problem and you can learn the weights using the method of least squares and gradient descent. We would however need to specify the number of knots or degrees of freedom. We can treat this as a hyperparameter and determine it using a technique called **cross-validation**. Using cross-validation, we would remove a portion of the data and fit a regression spline with a certain number of pre-determined knots on the remaining data. This regression spline is then evaluated on the held-out set. The optimum number of knots is the one that results in the best performance on the held-out set.

In GAMs, you can easily overfit by increasing the number of splines or degrees of freedom. If the number of splines is high, the resulting smoothing function which is a weighted sum of the splines would be quite 'wiggly', i.e. it would start to fit some of the noise in the data. How can we control this wiggliness or prevent overfitting? This can be done through a technique called **regularization**. In regularization, we would add a term to the least squares cost function that quantifies the wiggliness. The wiggliness of a smoothing function can be quantified by taking the integral of the squared of the 2nd order derivative of the function. By then using a hyperparameter (also called regularization parameter) represented by λ, we can adjust the intensity of wiggliness. A high value for λ penalizes wiggliness heavily. We can determine λ the same way we determine other hyperparameters using cross-validation.

Summary of GAMs

A GAM is a powerful model where the target variable is represented as a sum of smoothing functions representing the relationship of each of the features and the target. The smoothing function can be used to capture any non-linear relationship. This is shown mathematically again below.

$$y = w_0 + f_1(x_1) + f_2(x_2) + \dots + f_n(x_n)$$

It is a white-box model as we can easily see how each feature gets transformed to the output using the smoothing function. A common way of representing the smoothing function is using regression splines. A regression spline is represented as a simple weighted sum of basis functions. A basis function that is widely used for GAMs is the cubic spline. By increasing the number of splines or degrees of freedom, we can divide the distribution of data into small portions and model each portion piecewise. This way we can capture quite complex non-linear relationships. The learning algorithm essentially has to determine the weights for the regression spline. We can do this the same way as linear regression using the method of least squares and gradient descent. We can determine the number of splines using the cross-validation technique. As the number of splines increases, GAMs have a tendency to overfit on the data. We can safeguard against this by using the regularization technique. Using a regularization parameter λ, we can control the amount of 'wiggliness'. Higher λ ensures a smoother function. The parameter λ can also be determined using cross-validation.

GAMs can also be used to model interactions between variables. GA2M is a type of GAM that models pairwise interactions. It is shown mathematically below.

$$y = w_0 + f_1(x_1) + f_2(x_2) + \underbrace{f_3(x_1, x_2)}_{\substack{\text{Modeling interaction} \\ \text{between } x_1 \text{ and } x_2}} + f_4(x_4) + \dots + f_n(x_n)$$

With the help of Subject Matter Experts (SMEs), doctors in the Diagnostics+ example, you can determine what feature interactions need to be modeled. You could also look at the correlation between features to understand what features need to be modeled together.

In Python, there is a package called pyGAM that you can use to build and train GAMs. It is inspired by the GAM implementation in the popular mgcv package in R. You can install pyGAM in your Python environment using the pip package as follows.

```
pip install pygam
```

2.4.2 GAM for Diagnostics+ Diabetes

Let's now go back to the Diagnostics+ example to train a GAM to predict diabetes progression using all 10 features. It is important to note that the sex of the patient is a categorical or discrete feature. It does not make sense to model this feature using a smoothing function. We can treat such categorical features in GAMs as factor terms. The GAM can be trained using the pyGAM package as follows. As with decision trees, I'm not going to repeat that code that loads the diabetes dataset and that splits it into the train and test sets. Please refer Section 2.2 for that snippet of code.

```
from pygam import LinearGAM #A
from pygam import s #B
from pygam import f #C

# Load data using the code snippet in Section 2.2

gam = LinearGAM(s(0) + #D
                f(1) + #E
                s(2) + #F
                s(3) + #G
                s(4) + #H
                s(5) + #I
                s(6) + #J
                s(7) + #K
                s(8) + #L
                s(9), #M
                n_splines=35) #N

gam.gridsearch(X_train, y_train) #O

y_pred = gam.predict(X_test) #P

mae = np.mean(np.abs(y_test - y_pred)) #Q
```

#A Import the LinearGAM class from pygam that can be used to train a GAM for regression tasks
#B Import the smoothing term function to be used for numerical features
#C Import the factor term function to be used for categorical features
#D Cubic spline term for the Age feature
#E Factor term for the Sex feature which is categorical
#F Cubic spline term for the BMI feature
#G Cubic spline term for the BP feature
#H Cubic spline term for the Total Cholesterol feature
#I Cubic spline term for the LDL feature
#J Cubic spline term for the HDL feature
#K Cubic spline term for the Thyroid feature
#L Cubic spline term for the Glaucoma feature
#M Cubic spline term for the Glucose feature
#N Maximum number of splines to be used for each feature
#O Using grid search to perform training and cross-validation to determine the number of splines, the regularization parameter lambda and the optimum weights for the regression splines for each feature
#P Use trained GAM model to predict on the test
#Q Evaluate the performance of the model on the test set using the MAE metric

Now for the moment of truth! How did the GAM perform? The MAE performance of the GAM is 41.4, a pretty good improvement when compared to the linear regression and decision tree models. A comparison of the performance of all 3 models is summarized in Table 2.2. I have also included the performance of a baseline model which Diagnostics+ and the doctors have been using where they looked at the median diabetes progression across all patients. All models are compared against the baseline to show how much of an improvement the models give to the doctors. It looks like GAM is the best model across all performance metrics!

	MAE	RMSE	MAPE
Baseline	62.2	74.7	51.6
Linear Regression	42.8 (-19.4)	53.8 (-20.9)	37.5 (-14.1)
Decision Tree	48.6 (-13.6)	60.5 (-14.2)	44.4 (-7.2)
GAM	41.4 (-20.8)	52.2 (-22.5)	35.7 (-15.9)

Table 2.2: Performance comparison of Linear Regression, Decision Tree and GAM against a Baseline for Diagnostics+ AI

We have now seen the predictive power of GAMs. We could potentially get a further improvement in the performance by modeling feature interactions especially the cholesterol features with each other and with other features that are potentially highly correlated like BMI. As an exercise, I would highly encourage you to try modeling feature interactions using GAMs.

GAMs are white-box and can be easily interpreted. In the following section, we will see how GAMs can be interpreted.

GAMs for Classification Tasks

GAMs can also be used to train a binary classifier by using the logistic link function where the response y can be either 0 or 1. In the pyGAM package, you can make use of the logistic GAM for binary classification problems.

```
from pygam import LogisticGAM
gam = LogisticGAM()
gam.gridsearch(X_train, y_train)
```

2.4.3 Interpreting GAMs

Although each smoothing function is obtained as a linear combination of basis functions, the final smoothing function for each feature is non-linear and we therefore cannot interpret the weights the same way as linear regression. We however can easily visualize the effects of each feature on the target using partial dependence or partial effects plots. Partial dependence looks at the effect of each feature by marginalizing on the rest. It is highly interpretable as we can see the average effect of each feature value on the target variable. We can see if the target response to the feature is linear, non-linear, monotonic or non-monotonic. Figure 2.21 shows the effect of each of the patient metadata on the target variable. The 95% confidence interval around them have also been plotted. This will help us determine the sensitivity of the model to data points with low sample size.

Let’s now look at a couple of features in Figure 2.21 namely, BMI and BP. The effect of BMI on the target variable is shown by the bottom left graph. On the x-axis, we see the normalized values of BMI and on the y-axis, we see the effect that BMI has on the progression of diabetes for the patient. We see that as BMI increases, the effect on the progression of diabetes also increases. We see a similar trend for BP shown by the bottom right graph. We see that higher the BP, higher impact on the progression of diabetes. If we look at the 95% confidence internal lines (the dashed lines in Figure 2.21), we see a wider confidence internal around the lower and higher ends of BMI and BP. This is because there are fewer samples of patients at these range of values resulting in higher uncertainty in understanding the effects of these features at those ranges.

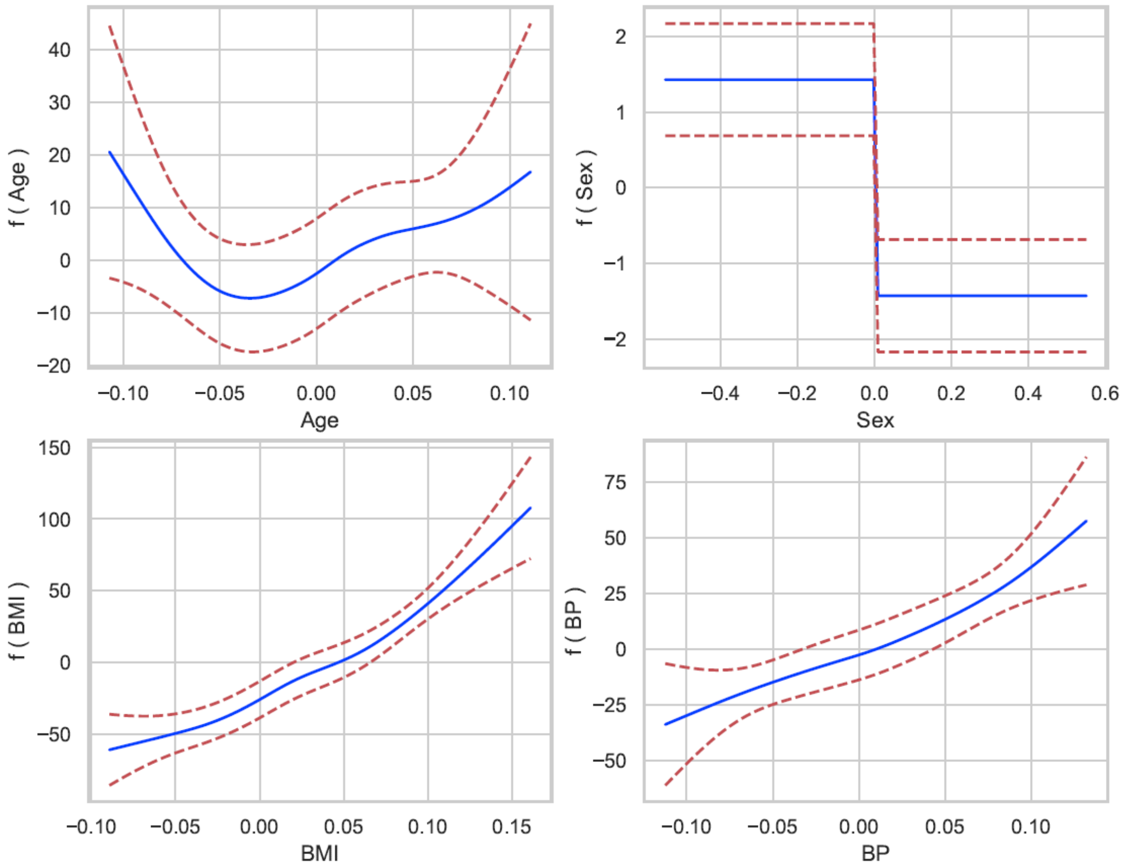


Figure 2.21: Effect of each of the patient metadata on the target variable

The code to generate Figure 2.21 is as follows.

```
grid_locs1 = [(0, 0), (0, 1), #A
              (1, 0), (1, 1)] #A
fig, ax = plt.subplots(2, 2, figsize=(10, 8)) #B
for i, feature in enumerate(feature_names[:4]): #C
    gl = grid_locs1[i] #D
    XX = gam.generate_X_grid(term=i) #E
    ax[gl[0], gl[1]].plot(XX[:, i], gam.partial_dependence(term=i, X=XX)) #F
    ax[gl[0], gl[1]].plot(XX[:, i], gam.partial_dependence(term=i, X=XX, width=.95)[1],
       c='r', ls='--') #G
    ax[gl[0], gl[1]].set_xlabel('%s' % feature) #H
    ax[gl[0], gl[1]].set_ylabel('f ( %s )' % feature) #H
```

#A Locations of the 4 graphs in the 2x2 matplotlib grid
#B Create 2x2 grid of matplotlib graphs
#C Iterate through the 4 patient metadata features
#D Get location of feature in the 2x2 grid
#E Generate the partial dependence of the feature values with the target marginalizing on the other features
#F Plot the partial dependence values as a solid line
#G Plot the 95% confidence interval around the partial dependence values as a dashed line
#H Add labels for the x and y axes

Figure 2.22 shows the effect of each of the 6 blood test measurements on the target. As an exercise, observe the effects that features like total cholesterol, LDL, HDL and glaucoma have on the progression of diabetes. What can you say about the impact of higher LDL values (or bad cholesterol) on the target variable? Why does higher total cholesterol have lower impact on the target variable? In order to answer these questions, let's look at a few patient cases with very high cholesterol values. The code snippet below will help you zoom into those patients.

```
print(df_data[(df_data['Total Cholesterol'] > 0.15) &
              (df_data['LDL'] > 0.19)])
```

If you execute the code above, you will see only 1 patient out of 442 that has a total cholesterol reading greater than 0.15 and an LDL reading greater than 0.19. The fasting glucose level for this patient one year out (the target variable) seems to be 84 which is in the normal range. This could explain why in Figure 2.22 we are seeing a very large negative effect for total cholesterol on the target variable for a range that is greater than 0.15. The negative effect of total cholesterol seems to be greater than the positive effect the bad LDL cholesterol seems to have on the target. The confidence interval seems much wider in these range of values. The model may have overfit on this one outlier patient record and so, we should not read too much into these effects. By observing these effects, we can identify cases or range of values where the model is sure of the prediction and cases where there is high uncertainty. For high uncertainty cases, we can go back to the diagnostics center to collect more patient data so that we have a representative sample.

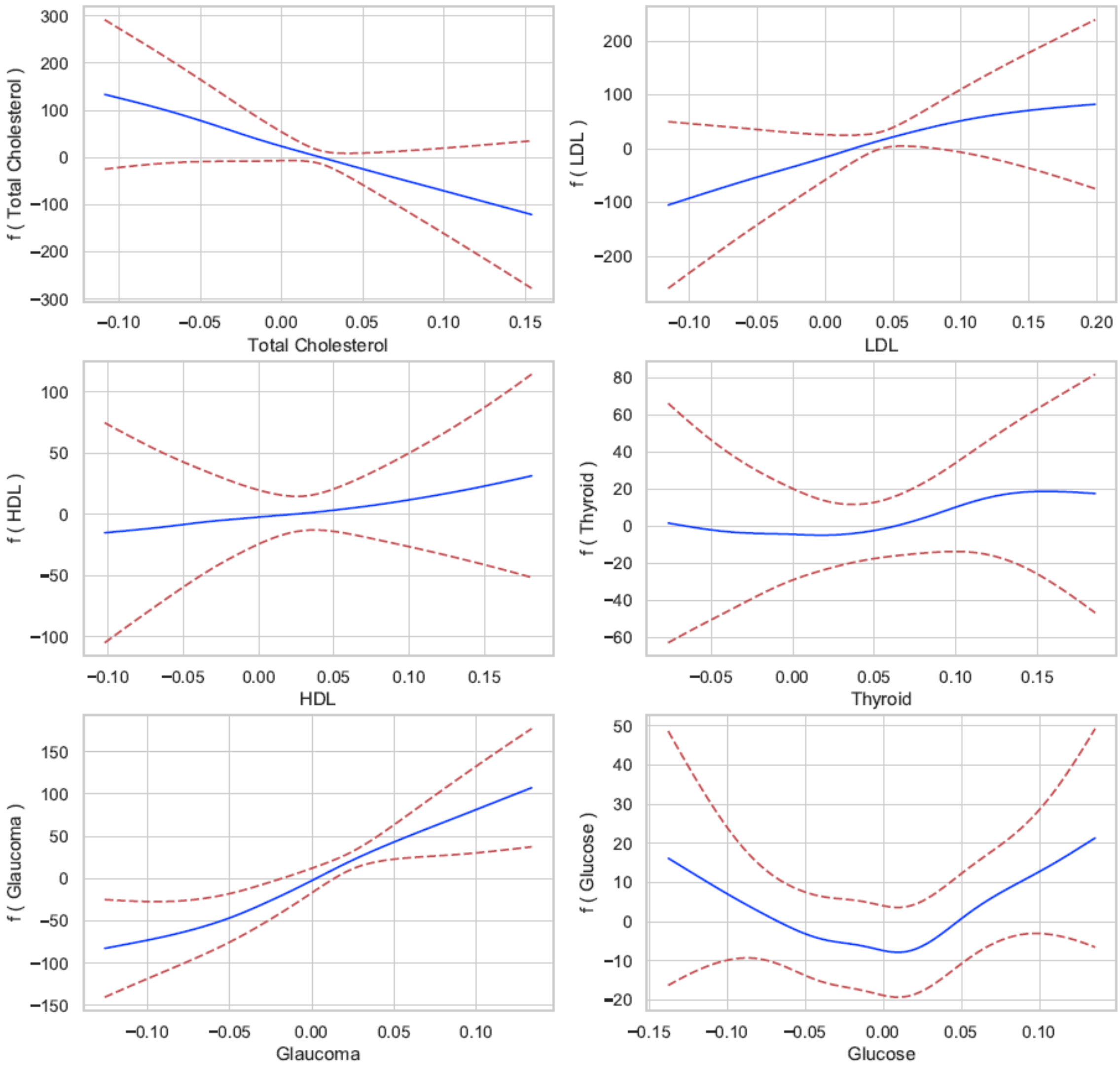


Figure 2.22: Effect of each of the blood test measurements on the target variable

Code to generate Figure 2.22 is as follows.

```
grid_locs2 = [(0, 0), (0, 1), #A
             (1, 0), (1, 1),  #A
             (2, 0), (2, 1)]  #A
fig2, ax2 = plt.subplots(3, 2, figsize=(12, 12)) #B
for i, feature in enumerate(feature_names[4:]): #C
    idx = i + 4 #D
    gl = grid_locs2[i] #D
    XX = gam.generate_X_grid(term=idx) #E
    ax2[gl[0], gl[1]].plot(XX[:, idx], gam.partial_dependence(term=idx, X=XX)) #F
    ax2[gl[0], gl[1]].plot(XX[:, idx], gam.partial_dependence(term=idx, X=XX, width=.95)[1],
       c='r', ls='--') #G
    ax2[gl[0], gl[1]].set_xlabel('%s' % feature) #H
    ax2[gl[0], gl[1]].set_ylabel('f ( %s )' % feature) #H
```

#A Locations of the 6 graphs in the 3x2 matplotlib grid
#B Create 3x2 grid of matplotlib graphs
#C Iterate through the 6 blood test measurement features
#D Get location of feature in the 3x2 grid
#E Generate the partial dependence of the feature values with the target marginalizing on the other features
#F Plot the partial dependence values as a solid line
#G Plot the 95% confidence interval around the partial dependence values as a dashed line
#H Add labels for the x and y axes

Through Figures 2.21 and 2.22, we can gain a much deeper understanding of the marginal effect of each of the feature values on the target. The partial dependence plots are useful to debug any issues with the model. By plotting the 95% confidence interval around the partial dependence values we can also see data points with low sample size. If feature values with low sample size has a dramatic effect on the target, then there could be an overfitting problem. We can also visualize the 'wiggliness' of the smoothing function to determine if the model has fit on the noise in the data. We can fix these overfitting problems by increasing the value of the regularization parameter. These partial dependence plots can also be shared with the SME, doctors in this case, for validation which will help gain their trust.

2.4.4 Limitations of GAMs

We have so far seen the advantages of GAMs in terms of predictive power and interpretability. GAMs have a tendency to overfit although this can be overcome with regularization. There are however some other limitations that you need to be aware of:

- GAMs are sensitive to feature values outside of the range in the training set and tend to lose its predictive power when exposed to outlier values.
- For mission critical tasks, GAMs may sometimes have limited predictive power, in which case you may need to consider more powerful black-box models.

2.5 Looking Ahead to Black-Box Models

Black-box models are models with really high predictive power and are typically applied in tasks for which model performance (such as accuracy) is extremely important. They are however inherently opaque and the characteristics that make them opaque are:

- The machine learning process is complicated, and you can't easily understand how the input features get transformed into the output or target variable.
- You can't easily identify the most important features to predict the target variable.

Examples of black-box models are tree ensembles such as random forest and gradient boosted trees, deep neural networks (DNNs), convolutional neural networks (CNNs) and recurrent neural networks (RNNs). Table 2.3 shows the machine learning tasks for which these models are typically applied.

Black-Box Model	Machine Learning Tasks
Tree Ensembles (Random Forest, Gradient Boosted Trees)	Regression and Classification
Deep Neural Networks (DNNs)	Regression and Classification
Convolutional Neural Networks (CNNs)	Image Classification, Object Detection
Recurrent Neural Networks (RNNs)	Sequence Modeling, Language Understanding

Table 2.3: Mapping of Black-Box Model to Machine Learning Tasks

I have now plotted in the black-box models in the same predictive power v/s interpretability plane as introduced in Section 2.1. This is shown in Figure 2.23.

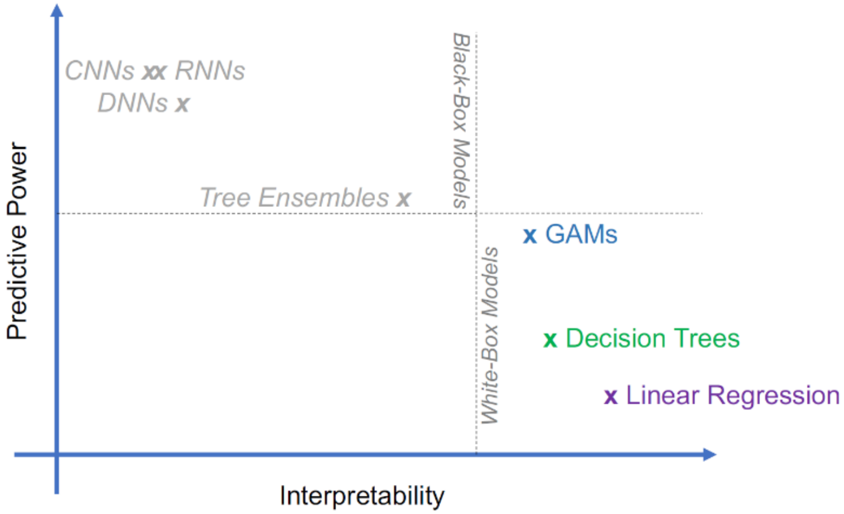


Figure 2.23: Black-Box Models on the Interpretability v/s Predictive Power Plane

The black-box models are clustered in the top left of the plane as they have quite high predictive power but low interpretability. For mission critical tasks, it is important not to trade-off model performance (such as accuracy) for interpretability by applying white-box models. We will need to apply black-box models for such tasks and will need to find ways to interpret them. There are multiple ways of interpreting black-box models and this is the main focus of the remaining chapters in this book.

2.6 Summary

- White-box models are inherently transparent. The machine learning process is straightforward to understand, and you can clearly interpret how the input features get transformed into the output. Using white-box models, you can identify the most important features and those features are understandable.
- Linear regression is one of the simplest white-box models where the target variable is modeled as a linear combination of the input features. You can determine the weights using the method of least squares and gradient descent.
- Linear regression can be implemented in Python using the `LinearRegression` class in the scikit-learn package. You can interpret the model by inspecting the coefficients or learned weights. The weights can also be used to determine the importance of each of the features. Linear regression however suffers from the problems of multicollinearity and underfitting.
- Decision tree is a slightly more advanced white-box model that can be used for both regression and classification tasks. You can predict the target variable by finding splitting the data across all features so as to minimize a cost function. You have learned the CART algorithm to learn the splits.
- A decision tree for regression tasks can be implemented in Python using the `DecisionTreeRegressor` class in scikit-learn. You can implement a decision tree for classification tasks using the `DecisionTreeClassifier` class in scikit-learn. You can interpret a decision tree learned using CART by visualizing it as a binary tree. The scikit-learn implementation also computes the feature importance for you. A decision tree can be used to model non-linear relationships but tends to suffer from the problem of overfitting.
- GAMs are a powerful white-box model where the target variable is represented as a sum of smoothing functions representing the relationship of each of the features and the target. You know that regression splines and cubic splines are widely used to represent the smoothing function.
- Regression splines and GAMs can be implemented using the pyGAM package in Python. The `LinearGAM` class can be used for regression tasks and the `LogisticGAM` class can be used for classification tasks. You can interpret a GAM by plotting the partial dependence of each of the features on the target. GAMs have a tendency to overfit, but this problem can be mitigated through regularization.
- Black-box models are models with really high predictive power and are typically applied

in tasks for which model performance (such as accuracy) is extremely important. They are however inherently opaque. The machine learning process is complicated, and you can't easily understand how the input features get transformed into the output or target variable. As a result of this, you can't easily identify the most important features to predict the target variable.

The focus of the remainder of the book is on interpreting these powerful black-box models such as tree ensembles and neural networks. In the next chapter, we will specifically be focusing on tree ensembles and how to interpret them using global, model-agnostic techniques.

3

Model Agnostic Methods - Global Interpretability

This chapter covers:

- **Characteristics of model agnostic methods and global interpretability**
- **How to implement tree ensembles specifically Random Forest, a black-box model**
- **How to interpret Random Forest models**
- **How to interpret black-box models using a model agnostic method called Partial Dependence Plots (PDPs)**
- **How to uncover bias by looking at feature interactions**

In the previous chapter we saw two different types of machine learning models – white-box and black-box – and focused most of our attention on how to interpret white-box models. Black-box models have very high predictive power and as the name suggests, are black boxes and hard to interpret. In this chapter, we will focus on interpreting black-box models and learn specifically of techniques that are **model agnostic** and **global** in scope. Recall from Chapter 1 that model-agnostic interpretability techniques are not dependent on the specific type of model being used. It can be applied to any model as it is independent of the internal structure of the model. Also, interpretability techniques that are global in scope will help us understand the entire model as a whole. We will also be focusing on tree ensembles, specifically Random Forest. Although the focus is on Random Forest, the model agnostic techniques that you will learn in this chapter can be applied to any model. We will switch our attention to more complex black-box models like neural networks in the following chapter. In Chapter 4, we will also learn about model agnostic techniques that are local in scope such as LIME, SHAP and Anchors.

The structure of Chapter 3 will be similar to Chapter 2. We will start off my looking at a concrete example. In this chapter, we will be taking a break from the Diagnostics+ center and focus on another problem related to education. The reason for choosing this problem is because the dataset has some interesting characteristics and we can expose some issues in this dataset through the interpretability techniques that we will learn in this chapter. As we have done in Chapter 2, the main focus of this chapter will be on implementing interpretability techniques to gain a much better understanding of black-box models (specifically tree ensembles). We will apply these interpretability techniques during model development and testing. We will also learn about model training and testing, especially the implementation aspects. Since the model learning, testing and understanding stages are quite iterative, it is important to cover all three stages together. Readers who are already familiar with training and testing tree ensembles are free to skip those sections and jump straight into interpretability.

3.1 High School Student Performance Predictor

Let's begin by looking at a concrete example. We will switch from the Diagnostic+ and healthcare sector to education. A superintendent of a school district in the United States has approached you to help her with a data science problem. The superintendent would like to understand how students are performing in three key subject areas – math, reading and writing – to determine level of funding required for various schools and also to ensure that every student succeeds as part of the Every Student Succeeds Act (ESSA).

The superintendent is specifically looking for help in predicting the grade of a high-school student in her district in math, reading and writing subjects. The grade can be either A, B, C or F. Given this information, how would you formulate this as a machine learning problem? Since the target of the model is to predict the grade, which can be one of four discrete values, the problem can be formulated as a **classification** problem. In terms of data, the superintendent has collected data of 1000 students in her district representing various schools and backgrounds. There are five data points collected for each student:

- Gender
- Ethnicity
- Parent level of education
- Type of lunch purchased by the student
- Level of preparation for tests

Given this data, you will therefore need to train three separate classifiers, one for each subject area. This is illustrated in Figure 3.1.

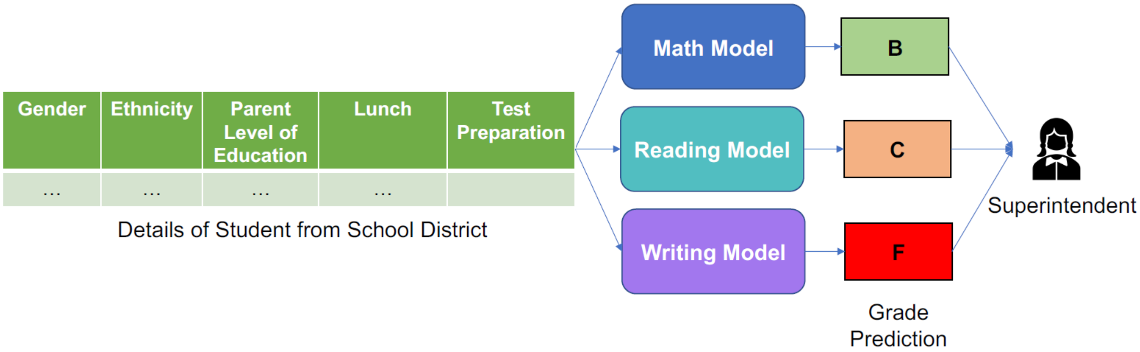


Figure 3.1: Illustration of student performance models required by superintendent of school district

3.1.1 Exploratory Data Analysis

We are dealing with a new dataset here and before we train the model, let's first understand the different features and possible values for them. There are five features in the dataset – the student's gender, ethnicity, their parent level of education, the type of lunch that they purchase and the level of preparation for tests. All of these features are **categorical** features where the possible values are discrete and finite. There are three target variables for each student – math grade, reading grade and writing grade. The grades can be either A, B, C or F.

There are two gender categories – male and female - where female population (52%) of students is slightly higher than the male population (48%). Let's now focus on two other features – the student's ethnicity and their parents' level of education. Figure 3.2 shows the different categories for each of those features and the proportion of students that fall under those categories. There are five groups or ethnicities in the population and groups C and D are the most represented, accounting for about 58% of the student population. There are six different parent levels of education. In ascending order, they are some high school, (recognized) high school, some college, associate's degree, bachelor's degree and master's degree. It looks there are a lot more students in the population whose parents have lower levels of education. Roughly 82% of the students have parents with high school/college level of education or an associate's degree. Only 18% of the students have parents with a bachelor's or master's degree.

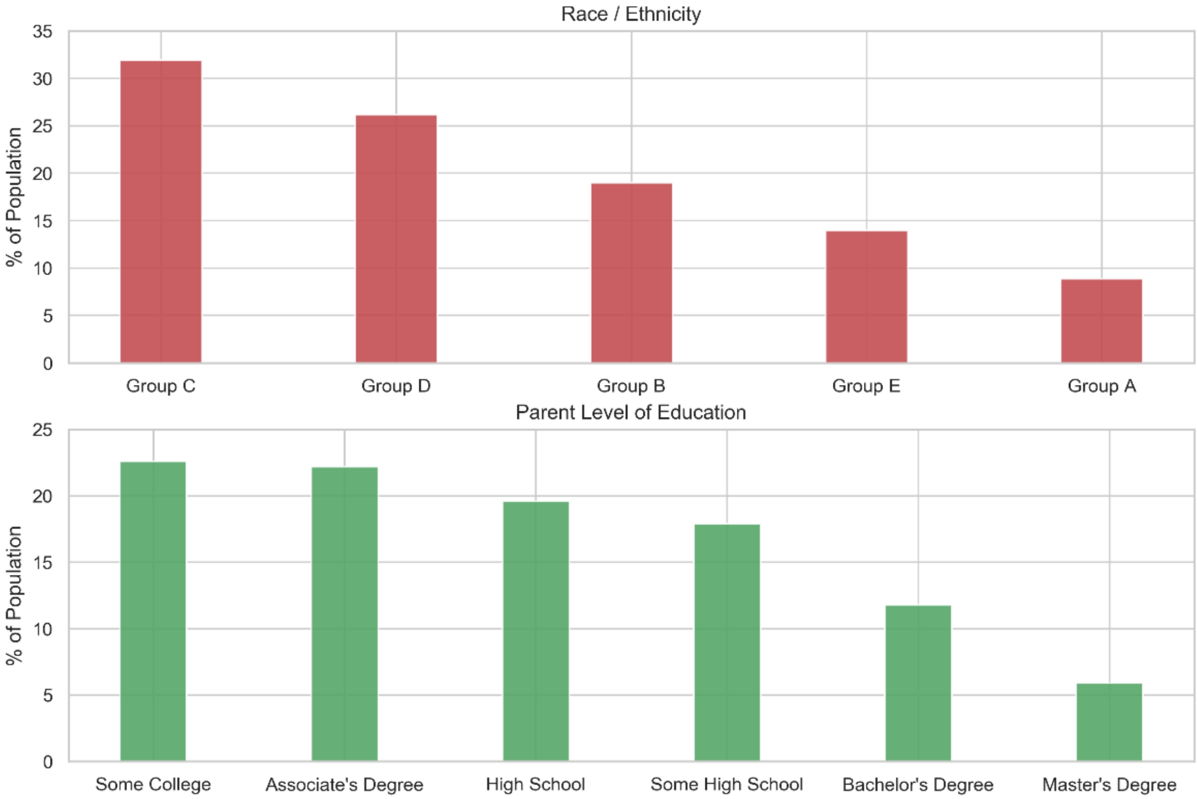


Figure 3.2: Values and proportions for the features – Ethnicity and Parent Level of Education

Now for the remaining two features – type of lunch purchased and the level of preparation for tests. Majority of the students (roughly 65%) purchase a standard lunch and the rest purchase free/reduced lunches. In terms of test preparation, only 36% of the students completed their preparation for the tests whereas for the remaining, it is either not completed or unknown.

Let’s now look at the proportion of students that get grades A, B, C or F for the three subject areas. This is shown in Figure 3.3. We can see that majority of the students (48-50%) get grade B and a very small proportion of the students (3-4%) get grade F. About 18-25% of the students get grade A and 22-28% of the students get grade C across all the three subject areas. It is important to note that the data is quite **imbalanced** before we train our models. Why is it important and how do we deal with imbalanced data? In a classification type of problem, we say that the data is imbalanced when there is a disproportionate number of examples or data points for a given class. It is important to note this because most machine learning algorithms work best when the proportion of samples for each class is roughly the same. This is because most algorithms are designed to minimize error or maximize accuracy and these algorithms tend to naturally bias towards the majority class. There are a few ways to deal with imbalanced classes. The following are a couple of common approaches:

- We will need to use the right performance metrics when we test and evaluate the models.
- Resample the training data such that the majority class is either under sampled or the minority class is over sampled.

We will learn more about them in Section 3.2.

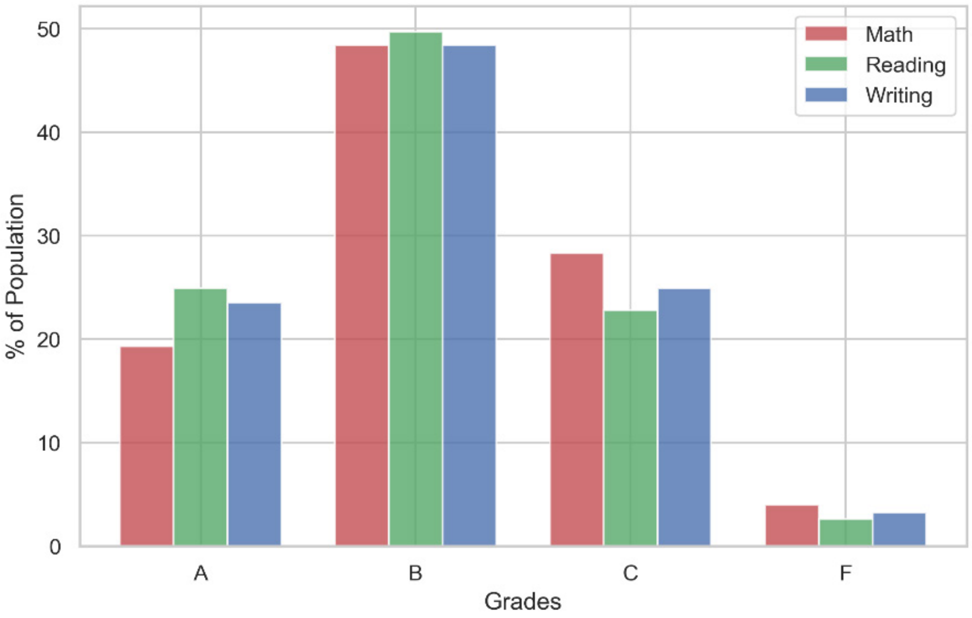


Figure 3.3: Values and proportions of the grade target variable for the three subject areas

Let's dissect the data a bit more. The insights that follow will be useful when we have to interpret and validate what the model has learned in Section 3.4. How do students generally perform when their parents have the lowest and highest levels of education? Let's compare the grade distributions for students whose parents have the lowest level of education (i.e. high school) with students whose parents have the highest level of education (i.e. Master's degree). Figure 3.4 shows this comparison across all three subject areas. The bars in red are students whose parents have high school education and the bars in green are students whose parents have a Master's degree.

Let's focus on the students whose parents have high school education. Across all three subject areas, it looks like in general less students get grade A and more students get grade F than the overall population. For the math subject area for instance, only 10% of the students who parents have high school education get grade A whereas in the overall population (as we’ve seen in Figure 3.3), roughly 20% of the students get grade A. Let’s now focus on students whose parents have a Master’s degree. It looks like in general more students get grade A and zero students get grade F, when compared to the overall population. For the math subject area for instance, roughly 30% of the students whose parents have a Master’s degree get grade A. If we now compare the two bars in Figure 3.4, a lot more students get a higher grade (A or B) when their parents have a higher level of education across all three subject areas.

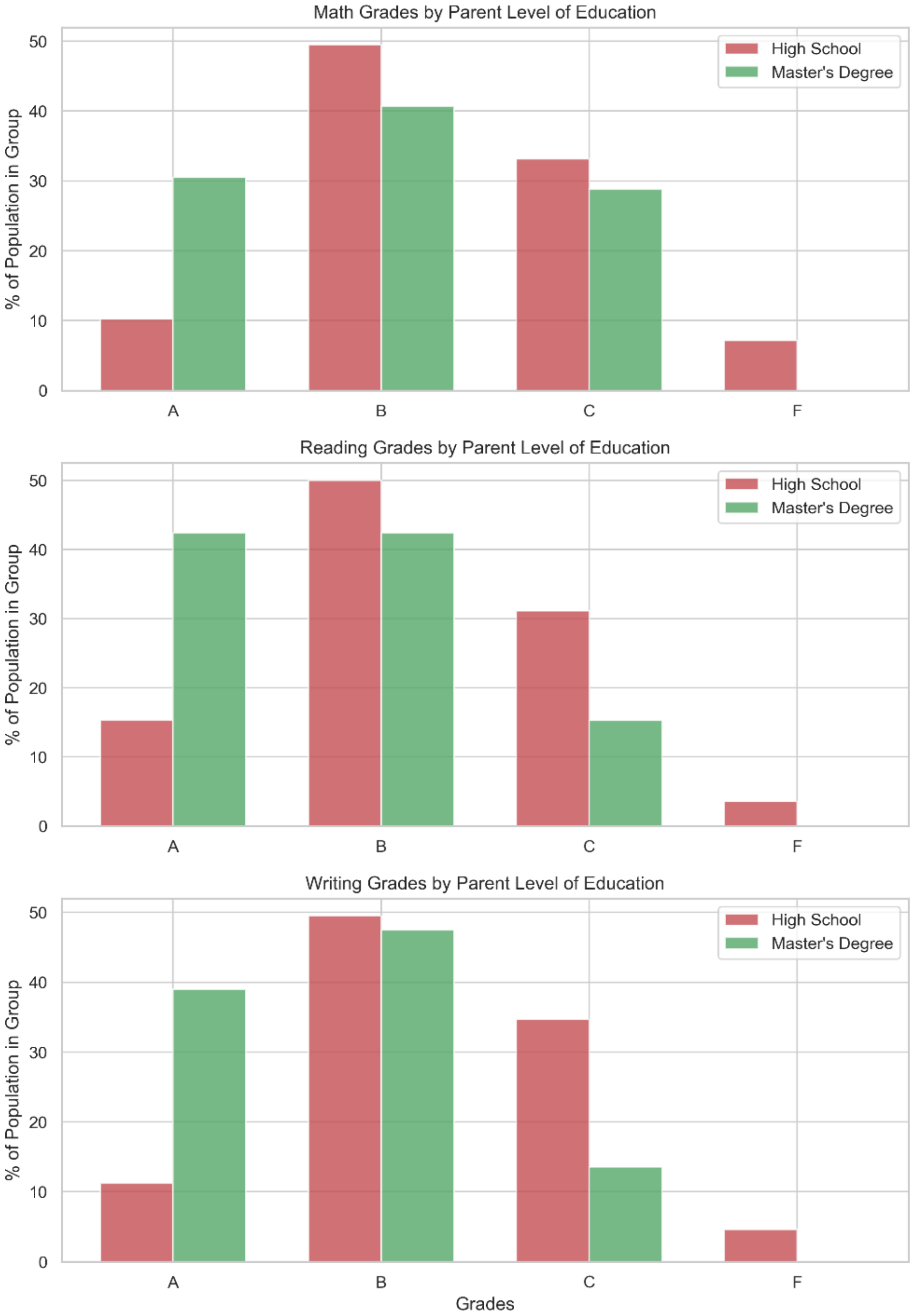

Math Grades by Parent Level of Education
% of Population in Group
0
10
20
30
40
50
A
B
C
F
High School
Master's Degree
Reading Grades by Parent Level of Education
% of Population in Group
0
10
20
30
40
50
A
B
C
F
High School
Master's Degree
Writing Grades by Parent Level of Education
% of Population in Group
0
10
20
30
40
50
A
B
C
F
High School
Master's Degree
Grades

Figure 3.4: Grade distributions comparing % of students whose parents have high school education v/s master's degree

How about ethnicity? How does the performance of a student belonging to the most represented group compare with the least represented group? From Figure 3.2, we know that the most represented group is C and the least represented group is A. Figure 3.5 compares the grade distributions of students belonging to group C with students belong to group A.

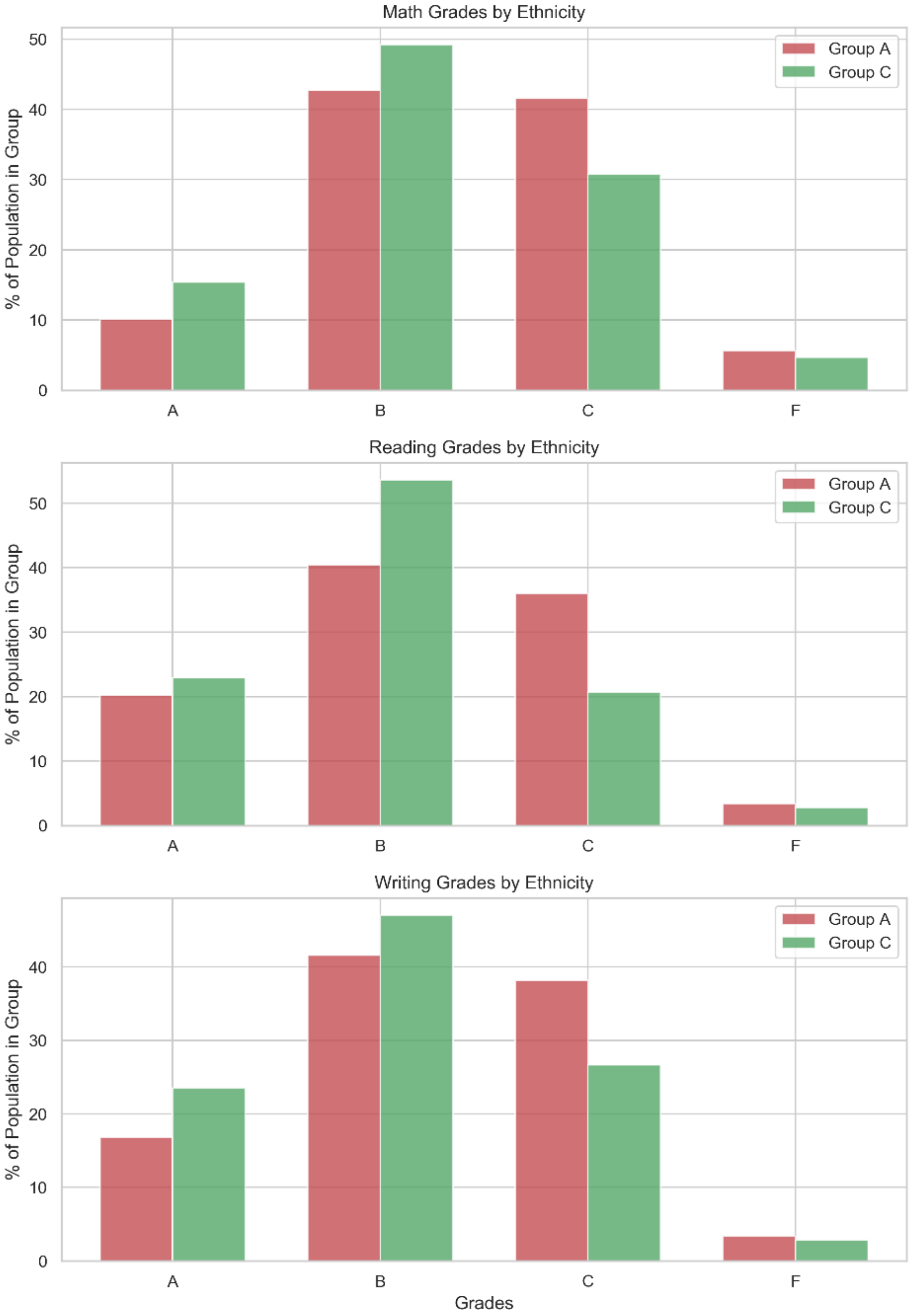

Math Grades by Ethnicity
% of Population in Group
0
10
20
30
40
50
A
B
C
F
Group A
Group C
Reading Grades by Ethnicity
% of Population in Group
0
10
20
30
40
50
A
B
C
F
Group A
Group C
Writing Grades by Ethnicity
% of Population in Group
0
10
20
30
40
A
B
C
F
Grades
Group A
Group C

Figure 3.5: Grade distributions comparing % of students belonging to Ethnicity Group A v/s Group C

It looks like in general, students from group C perform better than group A -- a larger proportion of students seem to get a higher grade (A or B) and a smaller proportion of students get a lower grade (C or F). As mentioned earlier, the insights in this section will come in handy when we interpret and validate what the model has learned in Section 3.4.

3.2 Tree Ensembles

In Chapter 2, we learned about decision trees – a powerful way of modeling non-linear relationships. Decision trees are white-box models and easy to interpret. We however saw that more complex decision trees suffer from the problem of overfitting where the model heavily fits the noise or variance in the data. The problem of overfitting can be overcome by reducing the complexity of decision trees by pruning the tree in terms of depth and the minimum number of samples required for the leaf nodes. This however results in low predictive power.

By combining several decision trees, we can circumvent the overfitting problem without compromising on predictive power. This is the principle behind tree ensembles. There are two broad ways of combining or ensembling decision trees:

1. **Bagging**: Using this technique, multiple decision trees are trained in parallel on separate random subsets of the training data. These individual decision trees are used to make predictions and they are combined by taking an average to come up with the final prediction. Random Forest is a tree ensemble using the bagging technique. In addition to training individual decision trees on random subsets of the data, the random forest algorithm also takes a random subset of the features to split the data on.
2. **Boosting**: Like in bagging, the boosting technique also trains multiple decision trees but in sequence. The first decision tree is typically a shallow tree and is trained on the training set. The objective of the second decision tree is to learn from the errors made by the first tree and to further improve the performance. Using this technique, multiple decision trees are strung together, and they iteratively try to optimize and reduce the errors made by the previous one. Adaptive boosting and gradient boosting are two common boosting algorithms.

In this chapter, we will be focusing on the bagging technique specifically the **random forest** algorithm. This algorithm is illustrated in Figure 3.6. First, random subsets of the training data are taken, and separate decision trees are trained on them. Each decision tree is then split on random subsets of the features. The final prediction is obtained by taking the majority vote across all decision trees. As you can see, a random forest model is much more complex than a decision tree. As the number of trees in the ensemble increases, the complexity increases. Moreover, it is much harder to visualize and interpret how features are split across all decision trees since random subsets of the data and features are taken for each of them. This makes random forest a black-box model and much harder to interpret. Being able to explain the algorithm does not guarantee interpretability in this case.

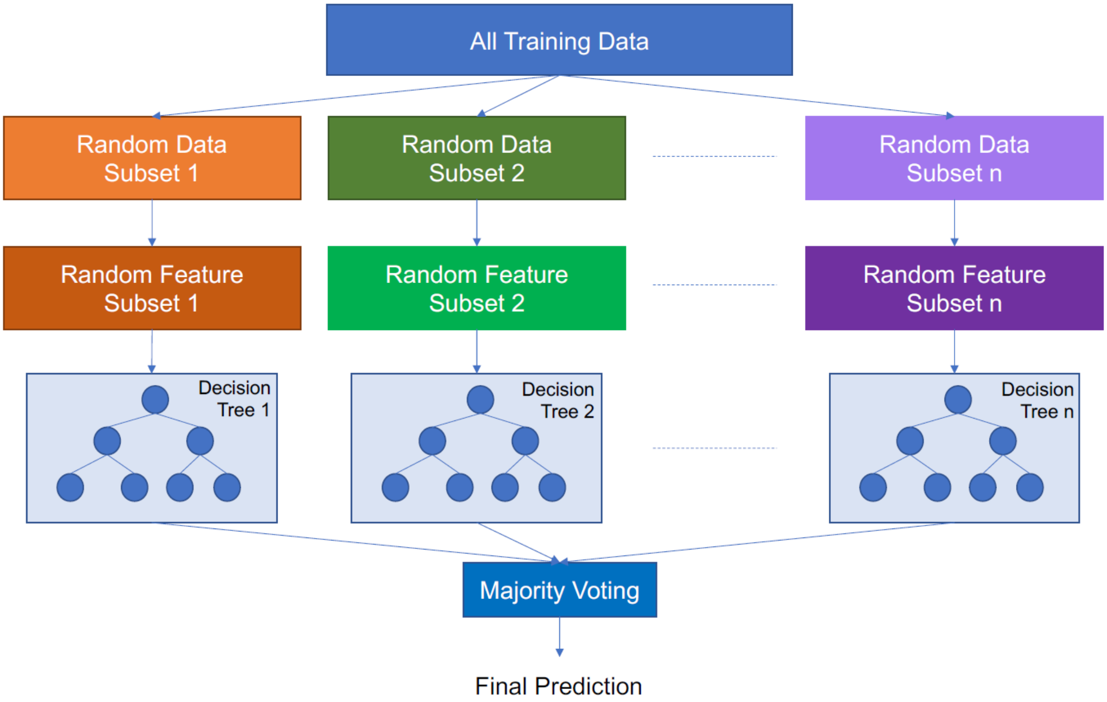


Figure 3.6: Illustration of the Random Forest Algorithm

For completeness, let's also learn how the adaptive boosting and gradient boosting algorithms work. **Adaptive boosting**, usually shortened as **AdaBoost**, is illustrated in Figure 3.7. The algorithm works as follows. The first step is to train a decision tree using all the training data. Each data point is given equal weight for the first decision tree. Once the first decision tree is trained, calculate the error rate of the tree by taking a weighted sum of the error for each data point. This weighted error rate is then used to determine the weight of the decision tree. If the error rate of the tree is high, then a lower weight is given for the tree since its predictive power is low. If the error rate is low, then a higher weight is given for the tree since it has a higher predictive power. The weight of the first decision tree is then used to determine the weights for each data point for the second decision tree. The wrongly classified data points will be given a higher weight so that the second decision tree can try to reduce the error rate. This process is then repeated in sequence until the number of trees we set during training is reached. After all the trees have been trained, then we come up with the final prediction by taking a weighted majority vote. Since a decision tree with a higher weight has higher predictive power, it is given more influence in the final prediction.

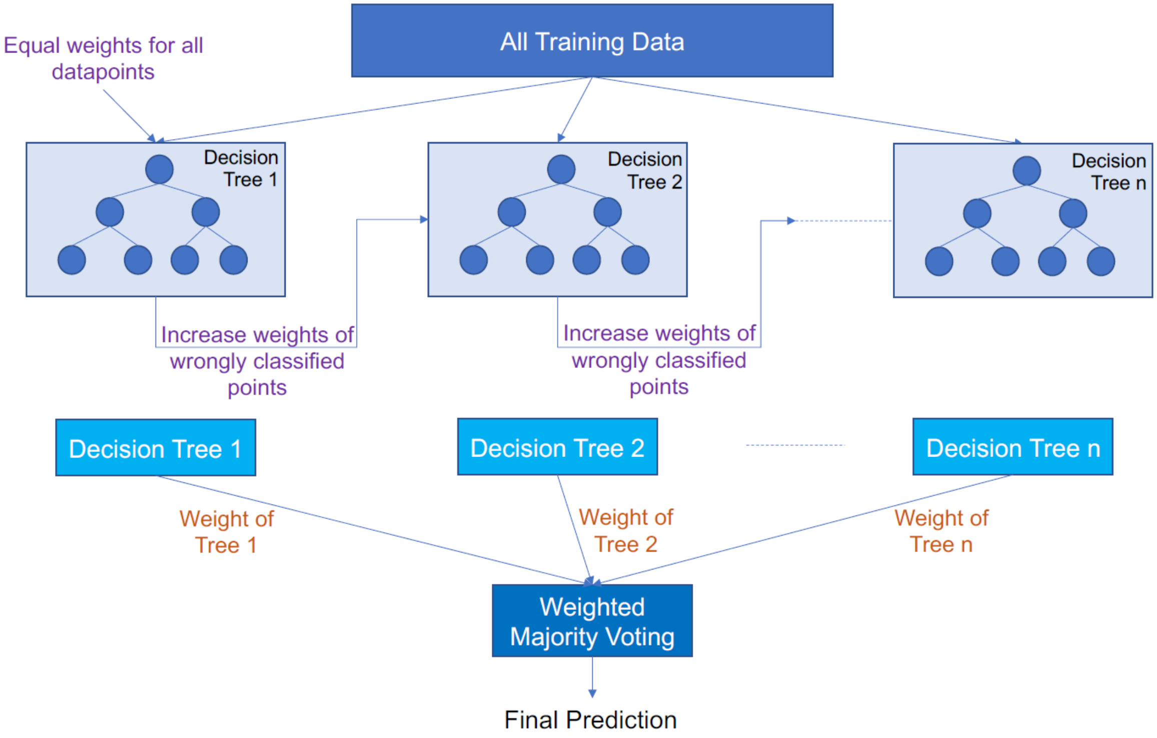


Figure 3.7: Illustration of the AdaBoost Algorithm

The **gradient boosting** algorithm works slightly differently and is illustrated in Figure 3.8. The algorithm works as follows. As with AdaBoost, the first decision tree is trained on all of the training data but unlike AdaBoost, there are no weights associated with the data points. After training the first decision tree, a residual error metric is calculated which is the difference between the actual target and the predicted target. The second decision tree is then trained to predict the residual error made by the first decision tree. So rather than updating the weights for each data point like in AdaBoost, gradient boosting predicts the residual error directly. The objective is for each tree is to fix the errors of the previous tree. This process is repeated in sequence until the number of trees we set during training is reached. After all the trees have been trained, then we come up with the final prediction by summing up the predictions of all the trees.

As mentioned earlier, we will be focusing on the random forest algorithm, but the methods used to train, evaluate and interpret the algorithm can be extended to the boosting techniques as well.

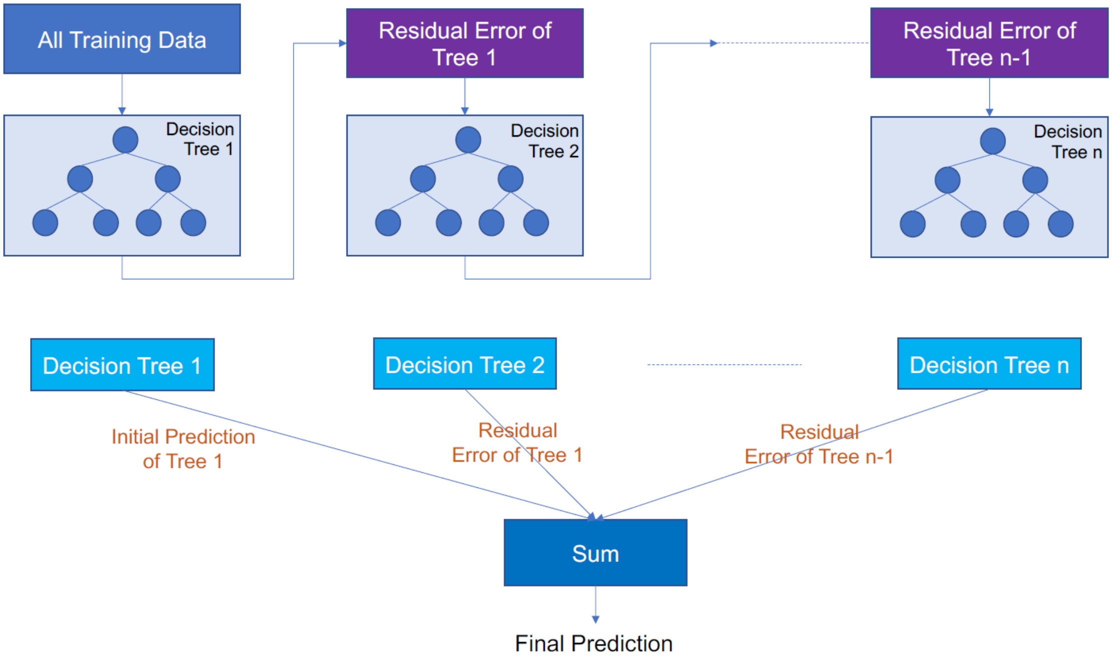


Figure 3.8: Illustration of the Gradient Boosting Algorithm

3.2.1 Training a Random Forest

Let’s now train our random forest model to predict high school student performance. The code snippet below shows how to prepare the data before training the model. Note that when splitting the data into the training and test sets, 20% of the data is used for testing. The rest of the data is used for training and validation. Also, a stratified sample on the math target variable is taken so that the distribution of the grades is the same for both the training and test sets. You can easily create similar splits using the reading and writing grades as well.

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder

# Load the data
df = pd.read_csv('data/StudentsPerformance.csv') #A

# First, encode the input features
gender_le = LabelEncoder() #B
race_le = LabelEncoder()   #B
parent_le = LabelEncoder() #B
lunch_le = LabelEncoder()  #B
test_prep_le = LabelEncoder() #B
df['gender_le'] = gender_le.fit_transform(df['gender']) #C
df['race_le'] = race_le.fit_transform(df['race/ethnicity']) #C
```

```
df['parent_le'] = parent_le.fit_transform(df['parental level of education']) #C
df['lunch_le'] = lunch_le.fit_transform(df['lunch']) #C
df['test_prep_le'] = test_prep_le.fit_transform(df['test preparation course']); #C

# Next, encode the target variables
math_grade_le = LabelEncoder() #D
reading_grade_le = LabelEncoder() #D
writing_grade_le = LabelEncoder() #D
df['math_grade_le'] = math_grade_le.fit_transform(df['math grade']) #E
df['reading_grade_le'] = reading_grade_le.fit_transform(df['reading grade']) #E
df['writing_grade_le'] = writing_grade_le.fit_transform(df['writing grade']) #E

# Creating training/val/test sets
df_train_val, df_test = train_test_split(df, test_size=0.2, #F
                                         stratify=df['math_grade_le'], #F
                                         shuffle=True, random_state=42) #F
feature_cols = ['gender_le', 'race_le',
                'parent_le', 'lunch_le', 'test_prep_le']
X_train_val = df_train_val[feature_cols] #G
X_test = df_test[feature_cols] #G
y_math_train_val = df_train_val['math_grade_le'] #H
y_reading_train_val = df_train_val['reading_grade_le'] #H
y_writing_train_val = df_train_val['writing_grade_le'] #H
y_math_test = df_test['math_grade_le'] #H
y_reading_test = df_test['reading_grade_le'] #H
y_writing_test = df_test['writing_grade_le'] #H
```

#A: Load the data into a pandas dataframe
#B: Since the input features are textual and categorical, we need to encode them into a numerical value
#C: Fit and transform the input features into numerical values
#D: Initialize the LabelEncoders for the target variables as well since letter grades have to be converted to a numerical value
#E: Fit and transform the target variables into numerical values
#F: Split the data into training/val and test sets
#G: Extract the feature matrix for the train/val and test sets
#H: Extract the target vectors for math, reading and writing for both the train/val and test sets

Once you have prepared the data, you are now ready to train the three random forest models for math, reading and writing. The code is shown below. Note that the optimum parameters for the random forest classifier can be determined using cross-validation. Also note that the random forest algorithm first takes random subsets of the training data to train each decision tree on and for each decision tree, the model takes random subsets of the feature to split the data on. For both of these random elements in the algorithm, it is important to set the seed for the random number generator. This is set using the `random_state` parameter. If this seed is not set, you will not be able to get reproducible and consistent results. First, let's use a helper function to create a random forest model with pre-defined parameters.

```
from sklearn.ensemble import RandomForestClassifier

def create_random_forest_model(n_estimators, #A
                               max_depth=10, #B
                               criterion='gini', #C
                               random_state=42, #D
                               n_jobs=4): #E
```

```
    return RandomForestClassifier(n_estimators=n_estimators,
                                  max_depth=max_depth,
                                  criterion=criterion,
                                  random_state=random_state,
                                  n_jobs=n_jobs)
```

#A: This parameter sets the number of decision trees in the random forest
#B: The maximum depth parameter for the decision tree
#C: Gini impurity is used as the cost function to optimize each decision tree on
#D: For reproducibility, set the seed for the random number generator
#E: Set n_jobs to train the individual decision trees in parallel utilizing all available cores in your computer

Now, let's use this helper function to initialize and train the three random forest models for the math, reading and writing subject areas.

```
math_model = create_random_forest_model(50) #A
math_model.fit(X_train_val, y_math_train_val) #B
y_math_model_test = math_model.predict(X_test) #C

reading_model = create_random_forest_model(25) #D reading_model.fit(X_train_val,
       y_reading_train_val) #D
y_reading_model_test = reading_model.predict(X_test) #D

writing_model = create_random_forest_model(40) #E
writing_model.fit(X_train_val, y_writing_train_val) #E
y_writing_model_test = writing_model.predict(X_test) #E
```

#A: Initialize the math model Random Forest classifier with 50 decision trees
#B: Fit the math student performance classifier on the training data using the math grade as the target
#C: Predict the math grade for all the students in the test set using the pre-trained model
#D: Initialize and train the random forest classifier with 25 decision trees to predict the reading grade
#E: Initialize and train the random forest classifier with 40 decision trees to predict the writing grade

Now that we've trained the three random forest models for math, reading and writing, let's evaluate it and compare it with a baseline model that always predicts the majority grade (in this case, B) for all the subjects. A metric that is typically used for classification problems is accuracy. This metric however is not suitable for situations where the classes are imbalanced. In our case, we've seen the student grades are pretty imbalanced as seen earlier in Figure 3.3. If for instance, 98% of the students obtains grade B in math, you can trick yourself into building a highly accurate model with 98% accuracy by always predicting grade B for all students. In order to gauge the performance of the model across all classes, better metrics like precision, recall and F1 are used. Precision is a metric that measures the proportion of predicted classes that are accurate. Recall is a metric that measures the proportion of actual classes that the model predicted accurately. The formulas for precision and recall are shown below.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

The perfect classifier will have a precision score of 1 and a recall score of 1 as the number of false positives and false negative will be 0. But in practice, these two metrics are at odds with each other – there is always a trade-off that you need to make between false positives and false negatives. As you reduce the false positives and increase precision, it will come at a cost of increased false negatives and lower recall. In order to find the right balance between precision and recall, we can combine the two metrics into a score called F1. The F1 score is the harmonic mean of precision and recall, as shown below.

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

The performance of all three models are shown in Table 3.1. They are compared against baselines for each subject to see how much of an improvement in performance the new models provide. A reasonable baseline used by the superintendent is to predict the majority grade (in this case B) for each subject.

	Precision (%)	**Recall (%)**	**F1 Score (%)**
Math Baseline	23	49	32
Math Model	39	41	39
Reading Baseline	24	49	32
Reading Model	39	43	41
Writing Baseline	18	43	25
Writing Model	44	45	41

Table 3.1: Performance of Math, Reading and Writing Models

In terms of performance, we can see that the math and reading random forest models perform better than the baseline in terms of precision and F1. The baseline math and reading models however perform better than the random forest models in terms of recall. Since the baseline models predict the majority class always, it gets all the majority class predictions right. But the precision metric and F1 gives us a better measure of the accuracy of all the predictions. The random forest model for the writing subject area however does better than the baseline for all three metrics. The superintendent is happy with this improvement in performance but would like to now understand how the model came up with the prediction. In Sections 3.3 and 3.4, we will see how to interpret a random forest model.

Training AdaBoost and Gradient Boosting Trees

The AdaBoost classifier can be trained by using the `AdaBoostClassifier` class provided by scikit-learn. An AdaBoost classifier can be initialized in Python as follows.

```
from sklearn.ensemble import AdaBoostClassifier
math_adaboost_model = AdaBoostClassifier(n_estimators=50)
```

The gradient boosting tree classifier can be trained by using the `GradientBoostingClassifier` class provided by scikit-learn.

```
from sklearn.ensemble import GradientBoostingClassifier
math_gbt_model = GradientBoostingClassifier(n_estimators=50)
```

The models can be trained the same way as the random forest classifier. There are variants of gradient boosting trees that are faster and scalable such as CatBoost and XGBoost. As an exercise, please do try training AdaBoost and Gradient Boosting classifiers for all three subject areas and compare it with the random forest models.

3.3 Interpreting a Random Forest

Since random forest is an ensemble of multiple decision trees, we could look at the global relative importance of each feature by averaging the normalized feature importance across all decision trees. In Chapter 2, we saw how to compute the importance of features for a decision tree. This is shown below for a given decision tree 't'.

$$\underbrace{I_{i,t}^{\text{feature}}}_{\substack{\text{Importance}\\ \text{of feature } i\\ \text{in decision tree } t}} = \frac{\overbrace{\sum_{j\epsilon\mathbb{J}} I_{j,t}^{\text{node}}}^{\substack{\text{Sum of importance}\\ \text{of all nodes } j\\ \text{that split on feature } i\\ \text{in decision tree } t}}}{\underbrace{\sum_{k\epsilon\mathbb{K}} I_{k,t}^{\text{node}}}_{\substack{\text{Sum of importance}\\ \text{of all nodes } k\\ \text{in decision tree } t}}}$$

In order to compute the relative importance, we will need to normalize the feature importance shown above by dividing it by the sum of all feature importance values. This is shown below.

$$\underbrace{I_{i,t}^{\text{feature}}}_{\substack{\text{Importance}\\ \text{of feature } i\\ \text{in decision tree } t}} = \frac{\overbrace{\sum_{j\epsilon\mathbb{J}} I_{j,t}^{\text{node}}}^{\substack{\text{Sum of importance}\\ \text{of all nodes } j\\ \text{that split on feature } i\\ \text{in decision tree } t}}}{\underbrace{\sum_{k\epsilon\mathbb{K}} I_{k,t}^{\text{node}}}_{\substack{\text{Sum of importance}\\ \text{of all nodes } k\\ \text{in decision tree } t}}}$$

You can now easily compute the global relative importance of each feature for the random forest by averaging the normalized feature importance for that feature across all decision trees. This is shown below. Note that feature importance is computed the same way for AdaBoost and Gradient Boosting trees.

$$\underbrace{I_i^{\text{feature}}}_{\substack{\text{Relative Importance} \\ \text{of feature } i}} = \frac{\overbrace{\sum_{t \in \text{all trees}} \bar{I}_{i,t}^{\text{feature}}}^{\substack{\text{Sum of normalized importance} \\ \text{of feature } i \\ \text{across all decision trees}}}}{\underbrace{T}_{\text{Total number of trees}}}$$

In Python, the feature importance can be obtained from the scikit-learn random forest model and plotted as follows.

```
math_fi = math_model.feature_importances_ * 100 #A
reading_fi = reading_model.feature_importances_ * 100 #B
writing_fi = writing_model.feature_importances_ * 100 #C

feature_names = ['Gender', 'Ethnicity', 'Parent Level of Education',
                 'Lunch', 'Test Preparation'] #D

# Code below plots the relative feature importance
# of the math, reading and writing random forest models
fig, ax = plt.subplots()
index = np.arange(len(feature_names))
bar_width = 0.2
opacity = 0.9
error_config = {'ecolor': '0.3'}
ax.bar(index, math_fi, bar_width,
       alpha=opacity, color='r',
       label='Math Grade Model')
ax.bar(index + bar_width, reading_fi, bar_width,
       alpha=opacity, color='g',
       label='Reading Grade Model')
ax.bar(index + bar_width * 2, writing_fi, bar_width,
       alpha=opacity, color='b',
       label='Writing Grade Model')
ax.set_xlabel('')
ax.set_ylabel('Feature Importance (%)')
ax.set_xticks(index + bar_width)
ax.set_xticklabels(feature_names)
for tick in ax.get_xticklabels():
    tick.set_rotation(90)
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
ax.grid(True);
```

#A: Get feature importance of math random forest model
#B: Get feature importance of reading random forest model
#C: Get feature importance of writing random forest model

The features and their importance values are shown in Figure 3.9. As can be seen from the figure, the two most important features for the 3 subjects are – parent level of education and the ethnicity of the student. This is useful information, but it does not tell us anything about how the grade is influenced by different levels of education and how race and education interact with each other.

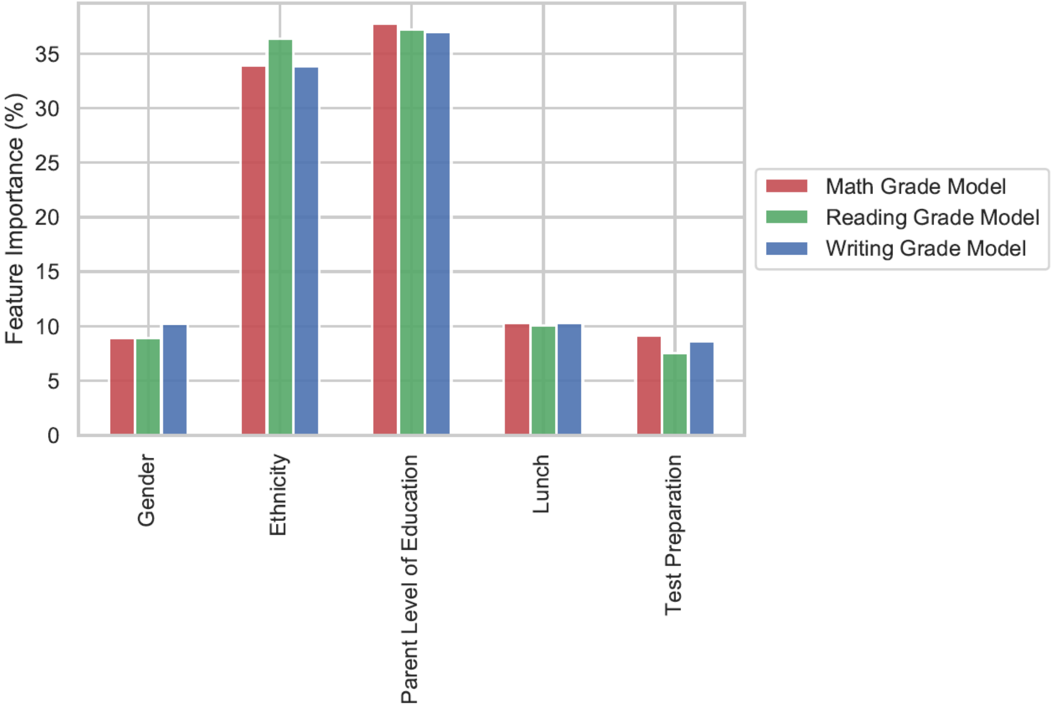


Figure 3.9: Feature Importance of Random Forest Model

Moreover, feature importance can be easily computed and visualized for tree ensembles, but it becomes a lot harder when we look at neural networks and more complex black-box models. This will become more apparent in Chapter 4. We therefore need to look at interpretability techniques that are agnostic of the type of black-box model. These model agnostic methods will be introduced in the following section.

3.4 Model Agnostic Methods – Global Interpretability

So far, we have been looking at interpretability techniques that are model specific or dependent. For white-box models we saw how to interpret linear regression models using the weights learned by the method of least squares. We interpreted decision trees as visualizing them as binary trees where each node splits the data using a feature determined using the CART algorithm. We were also able to visualize the global importance of features, the computation of which was specific to the model. GAMs were interpreted by visualizing the average effect of the basis splines for an individual feature on the target, by marginalizing on the rest of the features. These visualizations were called partial dependence or partial effect plots.

For black-box models like tree ensembles, we can compute the global relative importance of features, but this computation cannot be extended to other black-box models like neural networks. To better interpret black-box models, we will now explore model-agnostic methods that can be applied to any type of model. Like we've seen so far, we will also focus our attention on interpretability techniques that are global in scope in this chapter. Global interpretability techniques aim to give a better understanding of the model as a whole, i.e. the global effects of the features on the target variable. A globally interpretable model agnostic method is Partial Dependency Plots (PDPs). We will see in the following section how to extend PDPs that we learned for GAMs in chapter 2 to black-box models like random forest. We will formalize the definition of PDPs and also see how to extend PDPs to visualize interactions between any two features to validate if the model has picked up on any dependence between them.

Model agnostic interpretability techniques can also be local in scope. These are techniques that are used to interpret a model for a given local instance or prediction. Techniques such as LIME, SHAP and Anchors are model agnostic and local in scope. We will learn more about them in Chapter 4.

3.4.1 Partial Dependence Plots

As we saw with GAMs in Chapter 2, the idea behind Partial Dependence Plots (PDPs) is to show the marginal or average effects of different feature values on the model prediction. Let 'f' be the function learned by the model. For the high school student prediction problem, let f_{math}, $f_{reading}$ and $f_{writing}$ be the functions learned by the random forest models trained for the math, reading and writing subject areas. For each subject, the function returns the probability of receiving a certain grade given the input features. Let's now focus on the math random forest model for ease of understanding. The theory that you will learn now can easily be extended to the other subject areas.

Suppose for the math random forest model, we want to understand what effect different parent levels of education have on predicting a given grade. In order to understand this, we will need to do the following:

- Use the values for the rest of the features as is in the dataset
- Create an artificial dataset by setting the parent level of education to be the value of interest for all data points, i.e. if you are interested in looking at the average effects of high school education on the student's grade, then set the parent level of education to be high school for all data points
- Run through the model and obtain the predictions for all data points in this artificial set
- Take the average of the predictions to determine the overall average effect for that parent level of education

> More formally, if we want to plot the partial dependence of feature S, we marginalize on the rest of the features represented as set C, set feature S to be the value of interest and then look at the average effect of the math model for feature S assuming values of all the features in set C are known. This is shown by the equation below.

$$\hat{f}_{\text{math},\ x_S}(x_s|\mathbf{X}_C) = \frac{1}{n}\sum_{i=1}^{n} f_{\text{math}}\left(x_S, x_C^{(i)}\right)$$

In the above equation, the partial function for feature S is obtained by computing the average of the learned function fmath assuming the values for features in set C are known for all the examples in the training set, represented as 'n'.

It is important to note that the PDP cannot be trusted if feature S is correlated with features in set C. Why is that? In order to determine the average effects of a given value for feature S, we are creating an artificial dataset where we use the actual feature values for all the other features in set C but change the value of feature S to be the one of interest. If feature S is highly correlated with any feature in set C, we could be creating an artificial dataset that is highly unlikely. Let's look at a concrete example. Suppose that we are interested in understanding the average effects of high school level of education for a student's parent on their grade. We will be setting the parent level of education as high school for all the instances in our training set. Now if the parent level of education is highly correlated with ethnicity whereby, we know the parent level of education given the ethnicity, there could be an instance where it is highly unlikely for parents belonging to a certain ethnicity to have just high school education. We are thereby creating an artificial dataset whose distribution does not match the original training data. Since the model has not been exposed to that distribution of the data, the predictions from that model may be way off resulting in untrustworthy PDPs. We will come back to this limitation in Section 3.4.2.

Let's now learn how to implement PDPs. In Python, you can use the implementation provided by scikit-learn but this limits you to gradient boosted regressors or classifiers. A better implementation in Python that is truly model agnostic is PDPBox developed by Jiangchun Lee. You can install this library as follows.

```
pip install pdpbox
```

Now let's see PDPs in action. We will first focus on the most important feature which we learned was the parent level of education in Section 3.3, see Figure 3.9. We can look at the influence of different levels of education on predicting grades A, B, C and F for math as follows.

```
from pdpbox import pdp #A

feature_cols = ['gender_le', 'race_le', 'parent_le', 'lunch_le', 'test_prep_le'] #B

pdp_education = pdp.pdp_isolate(model=math_model, #C
                          dataset=df, #D
                          model_features=feature_cols,
                          feature='parent_le') #E
ple_xticklabels = ['High School', #F
                   'Some High School', #F
                   'Some College', #F
                   "Associate\'s Degree", #F
                   "Bachelor\'s Degree", #F
                   "Master\'s Degree"] #F
# Parameters for the PDP Plot
plot_params = {
    # plot title and subtitle
    'title': 'PDP for Parent Level Educations - Math Grade',
    'subtitle': 'Parent Level Education (Legend): \n%s' % (parent_title),
    'title_fontsize': 15,
    'subtitle_fontsize': 12,
    # color for contour line
    'contour_color':  'white',
    'font_family': 'Arial',
    # matplotlib color map for interact plot
    'cmap': 'viridis',
    # fill alpha for interact plot
    'inter_fill_alpha': 0.8,
    # fontsize for interact plot text
    'inter_fontsize': 9,
}
# Plot PDP of parent level of education in matplotlib
fig, axes = pdp.pdp_plot(pdp_isolate_out=pdp_education,
  feature_name='Parent Level Education',
                         center=True, x_quantile=False, ncols=2,
   plot_lines=False, frac_to_plot=100,
                         plot_params=plot_params, figsize=(18, 25))
axes['pdp_ax'][0].set_xlabel('Parent Level Education')
axes['pdp_ax'][1].set_xlabel('Parent Level Education')
axes['pdp_ax'][2].set_xlabel('Parent Level Education')
axes['pdp_ax'][3].set_xlabel('Parent Level Education')
axes['pdp_ax'][0].set_title('Grade A')
axes['pdp_ax'][1].set_title('Grade B')
axes['pdp_ax'][2].set_title('Grade C')
axes['pdp_ax'][3].set_title('Grade F')
axes['pdp_ax'][0].set_xticks(parent_codes)
axes['pdp_ax'][1].set_xticks(parent_codes)
axes['pdp_ax'][2].set_xticks(parent_codes)
axes['pdp_ax'][3].set_xticks(parent_codes)
axes['pdp_ax'][0].set_xticklabels(ple_xticklabels)
axes['pdp_ax'][1].set_xticklabels(ple_xticklabels)
axes['pdp_ax'][2].set_xticklabels(ple_xticklabels)
axes['pdp_ax'][3].set_xticklabels(ple_xticklabels)
for tick in axes['pdp_ax'][0].get_xticklabels():
    tick.set_rotation(45)
for tick in axes['pdp_ax'][1].get_xticklabels():
    tick.set_rotation(45)
for tick in axes['pdp_ax'][2].get_xticklabels():
    tick.set_rotation(45)
for tick in axes['pdp_ax'][3].get_xticklabels():
    tick.set_rotation(45)
```

#A: Import the pdp function from pdpbox
#B: Extract only the label encoded feature columns

#C: Obtain the partial dependence function for each level of education by passing in the learned math random forest model
#D: Use the preloaded dataset
#E: Marginalize on all the other features except the parent level of education
#F: Initialize the labels for the xticks starting from the lowest level of education till the highest

The plot generated by the code snippet above is shown in Figure 3.10. The partial dependence of parent level of education is shown for each grade A, B, C and F separately. The range of values for the partial dependence function is between 0 and 1 since the learned math model function for this classifier is a probability measure that ranges from 0 to 1. Let’s now zoom into a couple of grades to analyze the impact of parent level of education on the student’s grade.

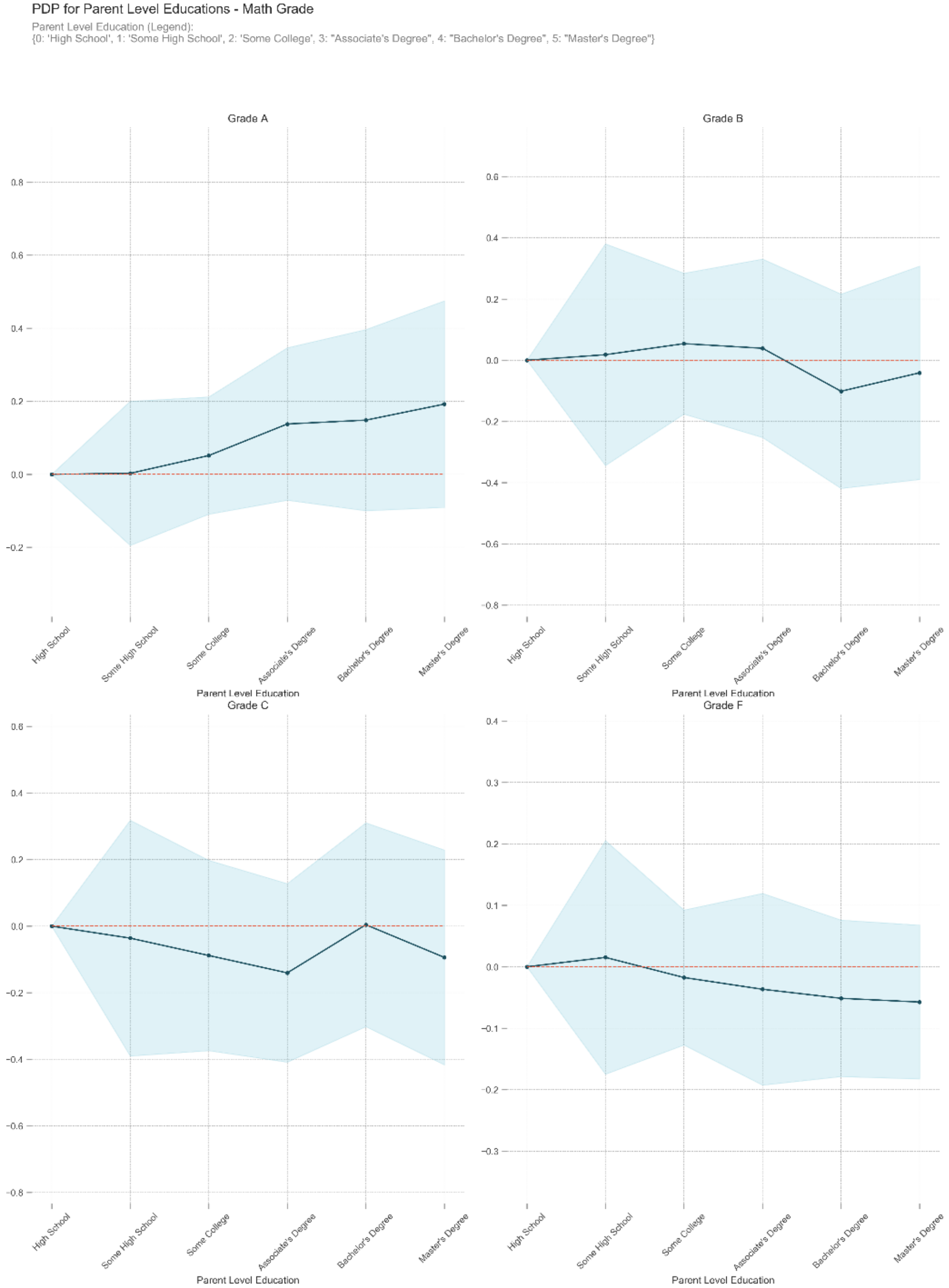

PDP for Parent Level Educations - Math Grade
Parent Level Education (Legend):
{0: 'High School', 1: 'Some High School', 2: 'Some College', 3: "Associate's Degree", 4: "Bachelor's Degree", 5: "Master's Degree"}
Grade A
0.8
0.6
0.4
0.2
0.0
−0.2
High School
Some High School
Some College
Associate's Degree
Bachelor's Degree
Master's Degree
Parent Level Education
Grade B
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
High School
Some High School
Some College
Associate's Degree
Bachelor's Degree
Master's Degree
Parent Level Education
Grade C
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
High School
Some High School
Some College
Associate's Degree
Bachelor's Degree
Master's Degree
Parent Level Education
Grade F
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
High School
Some High School
Some College
Associate's Degree
Bachelor's Degree
Master's Degree
Parent Level Education

Figure 3.10: PDP of various Parent Levels of Education for Math Grades A, B, C and F

In Figure 3.11, we have zoomed into the PDP for math grade A. We saw in Section 3.1.1 that the proportion of students getting grade A in math is higher when the parent has a master's degree than when the parent has a high school degree, see Figure 3.4. Has the random forest model learned this pattern? We can see from Figure 3.11 that the impact on getting grade A increases as the parent level of education increases. For a parent with high school education, the effect on predicting grade A in math is negligible – close to 0. This means that having high school education does not change anything for the model and other features besides the parent level of education come into play when predicting grade A. We can however see a high positive impact of roughly +0.2 when the parent has a master's degree. This means that on average, a master's degree pushes the probability of a student getting grade A by roughly 0.2.

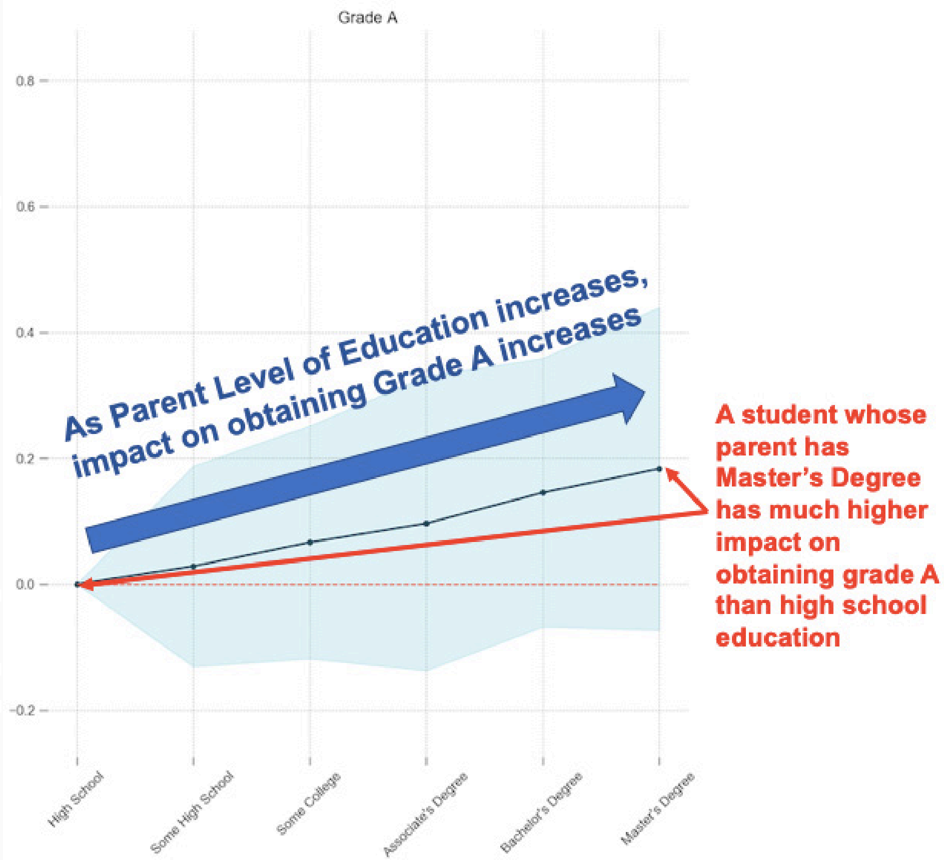


Figure 3.11: Interpreting the Parent Level of Education PDP for Math Grade A

In Figure 3.12, we have zoomed into the PDP of math grade F. We can notice a downward trend for grade F, i.e. the more educated the parent is, it has a more negative impact on predicting grade F. We can see that a student whose parent has a master's degree has a negative impact of roughly -0.05 on average in predicting grade A. This means that having a master's degree decreases the likelihood of getting grade F thereby increasing the likelihood of getting grade A. This insight is great and would not have been possible by just looking at the feature importance. The end user of this system (i.e. the superintendent) will therefore have more trust in the model that she is using.

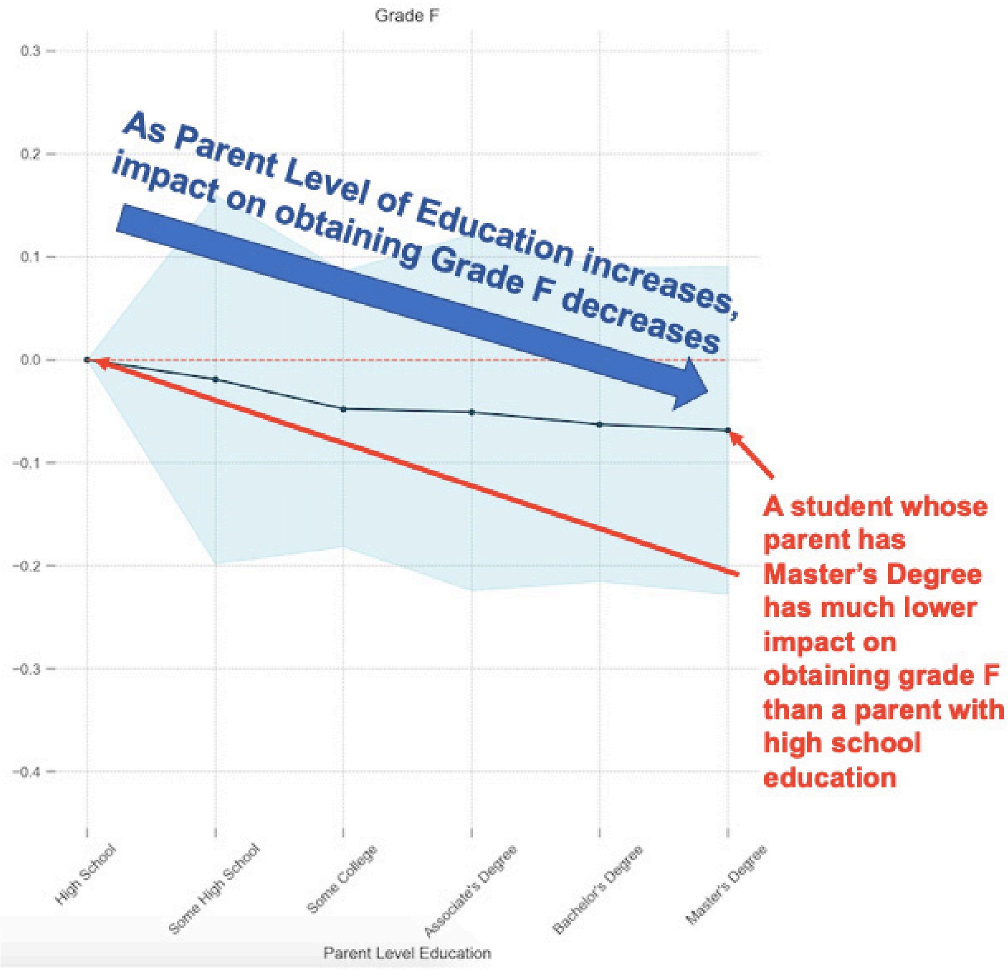


Figure 3.12: Interpreting the Parent Level of Education PDP for Math Grade F

As an exercise, I encourage you to extend the PDP code for math grade and parent level of education to the other subject areas – reading and writing. You can check if the patterns observed in Section 3.1.1 (see Figures 3.4) are learned by the random forest models. You can also extend the code to other features. As an exercise, you can pick the second most important feature which is the race or ethnicity of the student and generate the PDP for that feature.

3.4.2 Feature Interactions

PDPs can be extended to understand feature interactions. In the equation seen in Section 3.4.1, we will now look at two features in set S and marginalize on the rest. Let’s now look at the interactions between the two most important features – parent level of education and student ethnicity – in predicting grades A, B, C and F in math. Using PDPBox, we can easily visualize pairwise feature interactions as shown in the code snippet below.

```
pdp_race_parent = pdp.pdp_interact(model=math_model, #A
                                   dataset=df, #B
                                   model_features=feature_cols, #C
                                   features=['race_le', 'parent_le']) #D

# Parameters for the Feature Interaction plot
plot_params = {
    # plot title and subtitle
    'title': 'PDP Interaction - Math Grade',
    'subtitle': 'Race/Ethnicity (Legend): \n%s\nParent Level of Education (Legend): \n%s' %
       (race_title, parent_title),
    'title_fontsize': 15,
    'subtitle_fontsize': 12,
    # color for contour line
    'contour_color':  'white',
    'font_family': 'Arial',
    # matplotlib color map for interact plot
    'cmap': 'viridis',
    # fill alpha for interact plot
    'inter_fill_alpha': 0.8,
    # fontsize for interact plot text
    'inter_fontsize': 9,
}

# Plot feature interaction in matplotlib
fig, axes = pdp.pdp_interact_plot(pdp_race_parent, ['Race/Ethnicity', 'Parent Level of
       Education'],
                                  plot_type='grid', plot_pdp=True, plot_params=plot_params)
axes['pdp_inter_ax'][0]['_pdp_x_ax'].set_xlabel('Race/Ethnicity (Grade A)')
axes['pdp_inter_ax'][1]['_pdp_x_ax'].set_xlabel('Race/Ethnicity (Grade B)')
axes['pdp_inter_ax'][2]['_pdp_x_ax'].set_xlabel('Race/Ethnicity (Grade C)')
axes['pdp_inter_ax'][3]['_pdp_x_ax'].set_xlabel('Race/Ethnicity (Grade F)')
axes['pdp_inter_ax'][0]['_pdp_x_ax'].grid(False)
```

The plot generated by the code above is shown in Figure 3.13. There are four plots generated one for each grade. Feature interactions are visualized in a 2D grid where the 6 parent levels of education are on the y-axis and the 5 ethnicities are on the x-axis. I’ll zoom into grade A to decompose and explain this plot further.

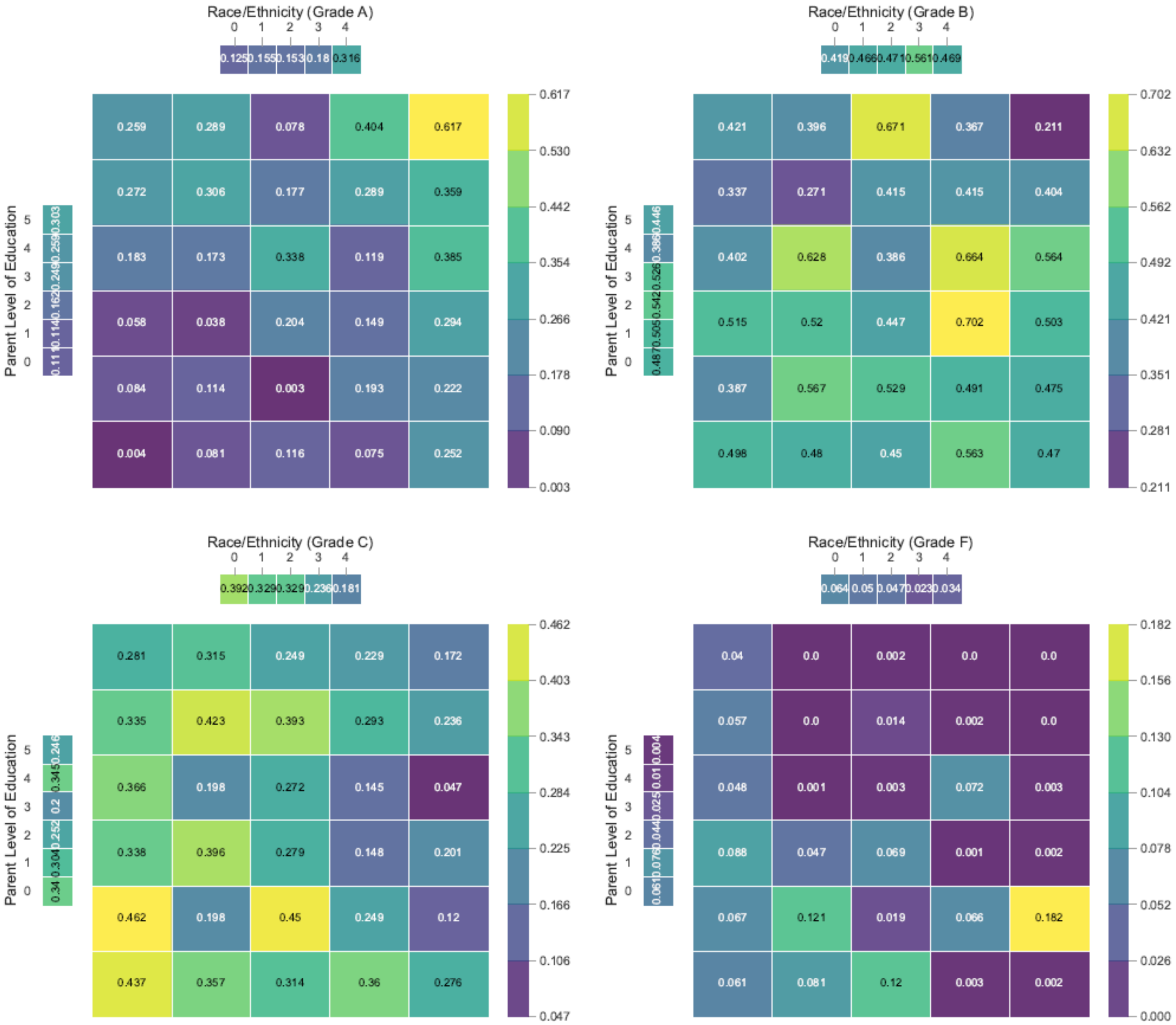
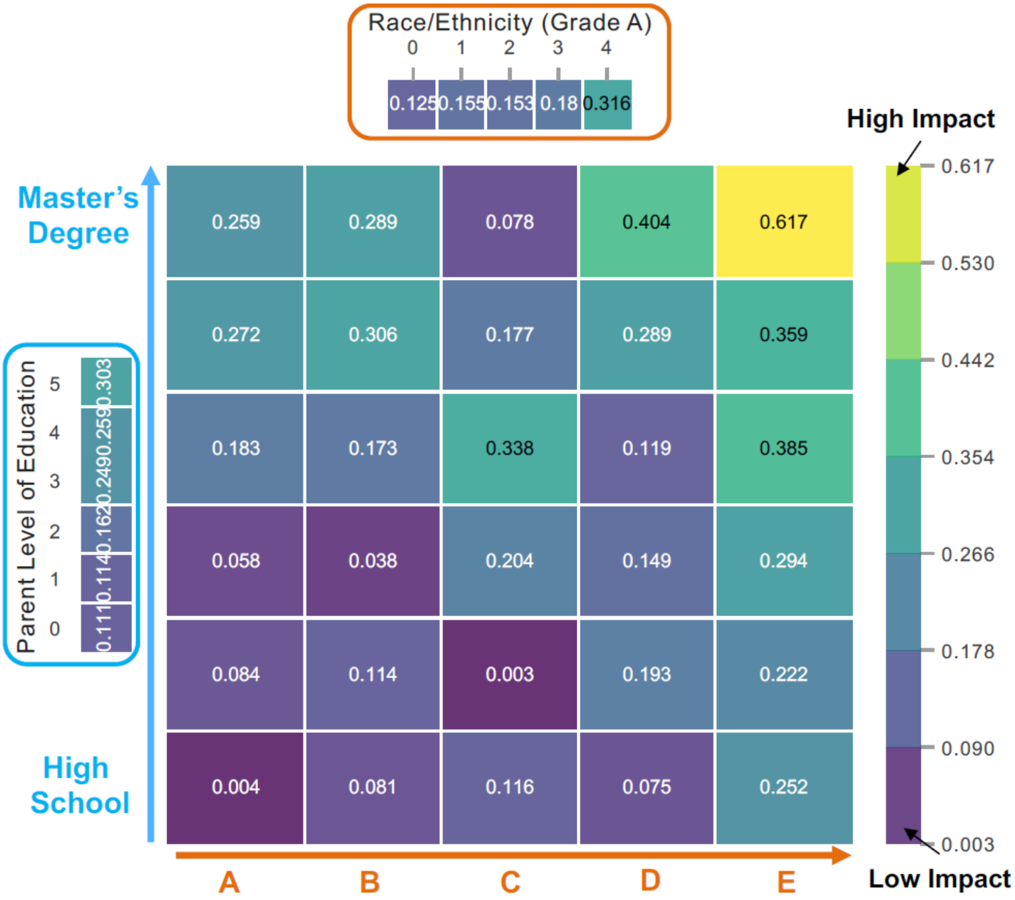


Figure 3.13: Interaction between Parent Level of Education and Ethnicity for all Math Grades A, B, C and F

Figure 3.14 shows the feature interaction plot for math grade A. The parent level of education is on the y-axis and the anonymized ethnicity of the student is on the x-axis. As you go from bottom to the top on the y-axis, the parent level of education increases from high school all the way to a master's degree. High school education is represented by a value of 0 and a master's degree is represented by a value of 5. The x-axis shows the five distinct ethnicity groups – A, B, C, D and E. Ethnicity group A is represented by a value of 0, group B is represented by a value of 1, group C by a value of 2 and so on. The number in each cell represents the impact of a given parent level of education and student ethnicity on getting grade A. For instance, the cell in the bottom-most row and the left-most column represents the average impact of a student belonging to ethnicity group A and whose parent has high school education in getting grade A. Please also note the numerical values and colors in each cell of the grid– lower number represents lower impact and higher number represents a higher impact in predicting grade A. The colors in the heatmap can also be used to study the level of impact – violet/blue represents low impact in predicting grade A and green/yellow represents high impact.



	A	B	C	D	E
Master's Degree	0.259	0.289	0.078	0.404	0.617
	0.272	0.306	0.177	0.289	0.359
	0.183	0.173	0.338	0.119	0.385
	0.058	0.038	0.204	0.149	0.294
	0.084	0.114	0.003	0.193	0.222
High School	0.004	0.081	0.116	0.075	0.252

Figure 3.14: Zooming into math grade A and decomposing the feature interaction plot

Now let’s focus on ethnicity group A, which is the left-most column in the grid. This is highlighted in Figure 3.15. You can see that as the parent level of education increases, the impact on predicting grade A also increases. This makes sense as it shows that the level of education has more influence on the grade than the ethnicity. This is validated by the feature importance plot shown in Figure 3.9 as well. The model has therefore learned this pattern well.

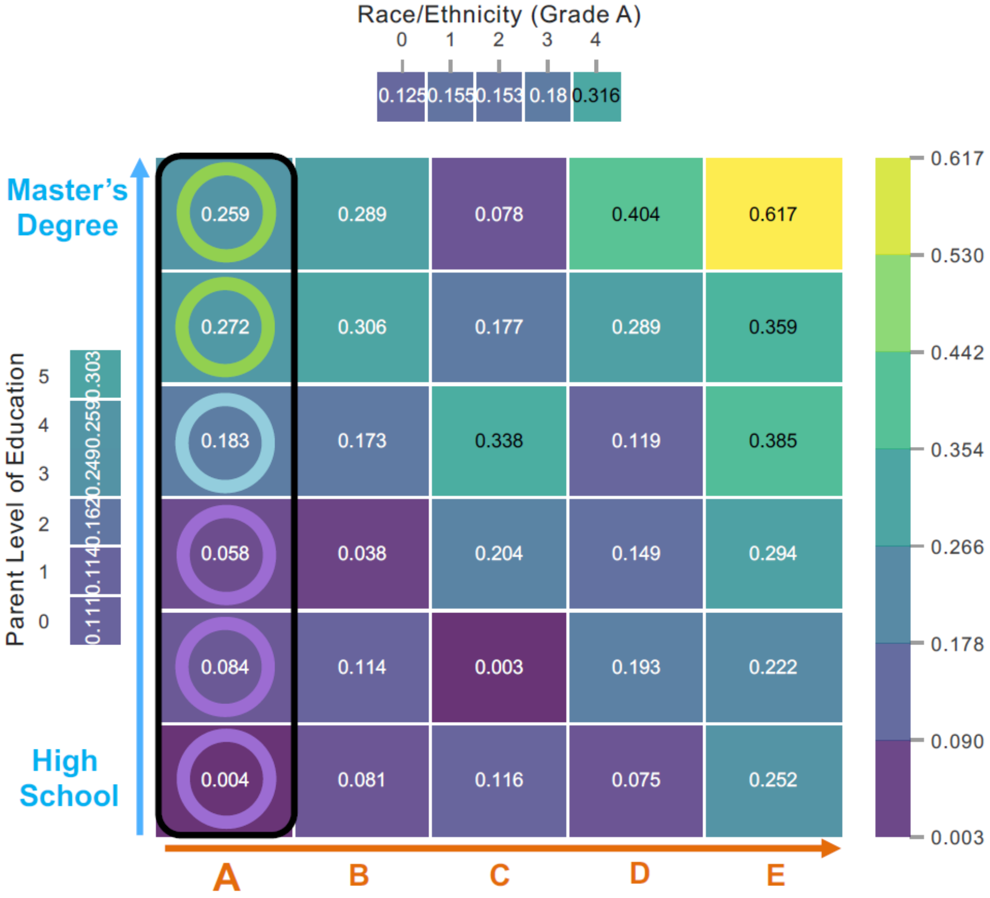


Figure 3.15: Impact of predicting grade A by conditioning on ethnicity group A

But what is going on with ethnicity group C, the third column? This is highlighted in Figure 3.16. It looks like a student whose parent has a high school degree has a higher positive impact in predicting grade A than a student whose parent has a master’s degree (compare the bottom-most cell with the top-most cell for the highlighted column). It also looks like a student whose parent has an associate degree has the highest positive impact in predicting grade A than any other level of education (see the third cell from the top in the highlighted column).

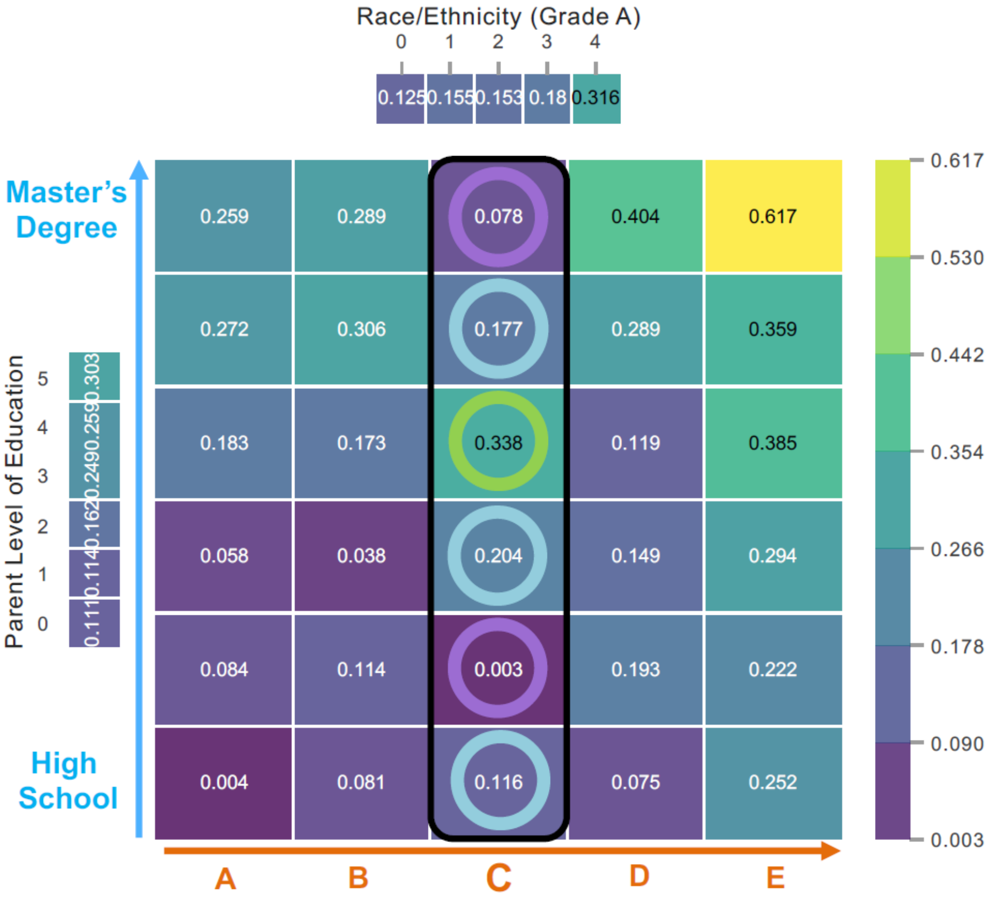


Figure 3.16: Impact of predicting grade A by conditioning on ethnicity group C

This is a bit concerning as it exposes any of the following problems:

- The parent level of education might be correlated with the ethnicity feature thereby resulting in untrustworthy feature interaction plots.
- The dataset does not properly represent the population, especially ethnicity group C. This is called sampling bias.
- The model is biased and has not learned the interaction between parent level of education and ethnicity well.

- The dataset exposes a bias that is systemic in society.

The first problem exposes the limitation of PDPs and we will discuss this limitation in the following paragraph. The second problem can be solved by collecting more data that is representative of the population. We will learn about other forms of bias and how to mitigate them later in Chapter 8. The third problem can be solved by adding or engineering more features or by training a better, more complex model. The last problem is much harder to solve, requiring better policies and laws, and this is beyond the scope of this book.

To check if the first problem exists, let's look at the correlation between the parent level of education and ethnicity. We have seen in Chapter 2 how to compute and visualize the correlation matrix. We used the Pearson correlation coefficient to quantify the correlation between the features for that problem. This coefficient can only be used for numerical features and not for categorical features. Since we are dealing with categorical features in this example, we have to use a different metric. The **Cramer's V statistic** can be used here as it measures the association between two categorical variables. This statistic can be between 0 and 1 where 0 signifies no correlation/association and 1 signifies maximum correlation/association. The following helper function can be used to compute this statistic.

```
import scipy.stats as ss

def cramers_corrected_stat(confusion_matrix):
    """ Calculate Cramers V statistic for categorial-categorial association.
        uses correction from Bergsma and Wicher,
        Journal of the Korean Statistical Society 42 (2013): 323-328
    """
    chi2 = ss.chi2_contingency(confusion_matrix)[0]
    n = confusion_matrix.sum().sum()
    phi2 = chi2/n
    r,k = confusion_matrix.shape
    phi2corr = max(0, phi2 - ((k-1)*(r-1))/(n-1))
    rcorr = r - ((r-1)**2)/(n-1)
    kcorr = k - ((k-1)**2)/(n-1)
    return np.sqrt(phi2corr / min( (kcorr-1), (rcorr-1)))
```

The correlation between parent level of education and ethnicity can be computed as follows.

```
confusion_matrix = pd.crosstab(df['parental level of education'],
                               df['race/ethnicity'])
print(cramers_corrected_stat(confusion_matrix))
```

By executing the above lines of code, we can see that the correlation or association between parent level of education and ethnicity is **0.0486**. This is quite low, and we can therefore rule out the issue of the feature interaction plot or PDP being untrustworthy.

We have seen in Figure 3.5 that students belonging to group C perform better in general than students belonging to group A. It could be the case that the model has learned this pattern. We can validate it by looking at the top-most legend in Figure 3.14, 3.15 and 3.16. We can see that if the student belongs to group C, it has a positive impact of +0.153 on predicting grade A which is greater than the impact that student has when belonging to group

A, which is +0.125. Let us now look at the difference in the distributions of the parent level of education between ethnicity groups A and C. This is shown in Figure 3.17.

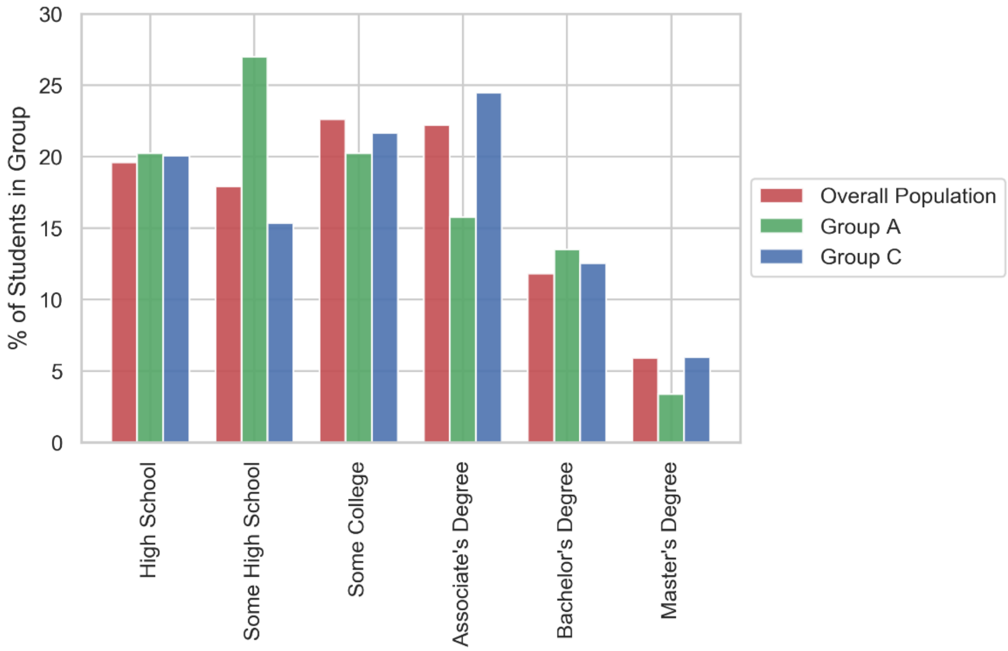


Figure 3.17: Comparison of the Parent Level of Education Distributions of the Overall Population with Ethnicity Groups A and C

From Figure 3.17, we can see that parents of students belonging to ethnicity group A are much more likely to have high school or some high school education than the overall population and students belonging to group C. It also looks like group C has a higher proportion of students whose parents have an associate degree than the overall population and group A. The differences in distributions are quite striking. We are not sure if the dataset represents the overall population and each ethnicity group accurately. As a data scientist, it is important to highlight this problem to the stakeholder (superintendent in this example) and ensure that the dataset is legitimate and that there is no sampling bias.

The important point to take away from this section is that interpretability techniques especially PDPs and feature interactions are great tools to expose potential problems with the model or the data before the model is deployed into production. None of the insights in this section would have been possible by just looking at the feature importance. As an exercise, I encourage you to use the PDPBox package on other black-box models such as gradient boosting trees.

Accumulated Local Effects (ALE)

We have seen in this chapter that PDPs and feature interaction plots based on them are not trustworthy if the features are correlated with each other. An interpretability technique that is unbiased and overcomes the limitation of PDPs is Accumulated Local Effects (ALE). This technique was proposed in 2016 by Daniel W. Apley and Jingyu Zhu. At the time of writing this book, ALE is only implemented in the R programming language. There is a Python implementation that is still a work in progress and does not support categorical features yet. Since the implementation of ALE is not mature yet, we will cover this technique in greater depth in a later release of this book.

3.5 Summary

- Model-agnostic interpretability techniques are not dependent on the specific type of model being used. It can be applied to any model as it is independent of the internal structure of the model.
- Interpretability techniques that are global in scope will help us understand the entire model as a whole.
- To overcome the problem of overfitting, there are two broad ways of combining or ensembling decision trees - bagging and boosting.
- Using the bagging technique, multiple decision trees are trained in parallel on separate random subsets of the training data. These individual decision trees are used to make predictions and they are combined by taking an average to come up with the final prediction. Random Forest is a tree ensemble using the bagging technique.
- Like in bagging, the boosting technique also trains multiple decision trees but in sequence. The first decision tree is typically a shallow tree and is trained on the training set. The objective of the second decision tree is to learn from the errors made by the first tree and to further improve the performance. Using this technique, multiple decision trees are strung together, and they iteratively try to optimize and reduce the errors made by the previous one. Adaptive boosting and gradient boosting are two common boosting algorithms.
- A random forest model for classification tasks can be trained in Python using the `RandomForestClassifier` class provided by the scikit-learn package. This implementation will also help you easily compute the global relative importance of features.
- The adaptive boosting and gradient boosting classifiers can be trained by using the `AdaBoostClassifier` and `GradientBoostingClassifier` classes respectively, provided by scikit-learn. There are variants of gradient boosting trees that are faster and scalable such as `CatBoost` and `XGBoost`.
- For tree ensembles, we can compute the global relative importance of features, but this computation cannot be extended to other black-box models like neural networks.
- Partial Dependence Plot (PDP) is a global, model-agnostic interpretability technique that can be used to understand the marginal or average effects of different feature values on the model prediction. PDPs cannot be trusted if features are correlated with each

other. PDPs can be implemented using the `PDPBox` Python package.

- PDPs can be extended to understand feature interactions as well. PDPs and feature interaction plots can be used to expose possible issues such as sampling bias and model bias.

In the next chapter, we will focus on neural networks and how to interpret them using model agnostic techniques that are local in scope. We will specifically cover the techniques LIME, SHAP and Anchors.

4

Model Agnostic Methods – Local Interpretability

This chapter covers:

- **Characteristics of deep neural networks**
- **How to implement deep neural networks that are inherently black box**
- **Perturbation-based model agnostics methods that are local in scope such as LIME, SHAP and Anchors**
- **How to interpret deep neural networks using LIME, SHAP and Anchors**
- **Strengths and weaknesses of LIME, SHAP and Anchors**

In the previous chapter, we looked at tree ensembles especially Random Forest models and learned how to interpret them using model agnostic methods that are global in scope such as Partial Dependence Plots (PDPs) and feature interaction plots. We saw that PDPs are a great way of understanding how individual feature values impact the final model prediction at a global scale. We were also able to see how features interact with each other using the feature interaction plots and also how they can be used to expose potential issues such as bias. They are easy and intuitive to understand but a major drawback of PDPs is that it assumes features are independent of each other. Higher order feature interactions can also not be visualized using feature interaction plots.

In this chapter, we will look at more advanced model agnostic techniques that overcome these difficulties. We will specifically focus on techniques such as Local Interpretable Model-agnostic Explanations (LIME), Shapley Additive exPlanations (SHAP) and Anchors. Unlike PDPs and feature interaction plots, these techniques are local in scope. This means that it can be used to interpret only a single instance or prediction. We will also now switch to interpreting neural networks that are inherently black box, specifically focusing on Deep Neural Networks (DNNs).

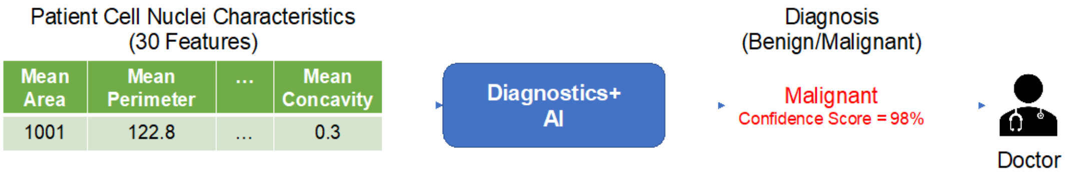
We will follow a similar structure as the previous chapters. We will start off with a concrete example where the objective is to build a model for breast cancer diagnosis. We will explore this new dataset and learn how to train and evaluate DNNs in Pytorch. We will then learn how to interpret them. It is worth reiterating that although the main focus of this chapter is on interpreting DNNs, we will also be covering basic concepts of DNNs and how to train and test them. Since the learning, testing and understanding stages are quite iterative, it is important to cover all three together. There are also some key insights and concepts that will be covered in the earlier sections that will be useful during model interpretation. Readers who are already familiar with DNNs and how to train and test them are free to skip the earlier sections and jump straight to Section 4.4 which covers model interpretability.

4.1 Diagnostics+ AI – Breast Cancer Diagnosis

Let’s look at a concrete example. We’ll go back to the Diagnostics+ center introduced in Chapters 1 and 2. The center would like to extend its AI capabilities to now diagnose breast cancer. The center has digitized the images of a fine needle aspirate of breast masses from around 570 patients. Features were computed from these digitized images that described the characteristics of cell nuclei present in the images. For each cell nucleus, 10 features are used to describe its characteristics. They are:

- Radius
- Texture
- Perimeter
- Area
- Smoothness
- Compactness
- Concavity
- Concave points
- Symmetry, and
- Fractal dimension

For all the nuclei present in an image for a patient, the mean, standard error and the largest or worst values are computed for each of the 10 features above. For each patient, therefore, there are 30 features in total. Given these input features, the goal of the AI system is to predict if the cancerous cell is benign or malignant and to provide a confidence score for the doctor to help with his diagnosis. This is summarized in Figure 4.1.



Mean Area	Mean Perimeter	…	Mean Concavity
1001	122.8	…	0.3

Figure 4.1: Diagnostics+ AI for Breast Cancer Diagnosis

Given this information, how would you formulate this as a machine learning problem? Since the target of the model is to predict if a given breast mass is benign or malignant, we can formulate this problem as a **binary classification** problem.

4.2 Exploratory Data Analysis

Let’s now try to understand this dataset a bit better. Exploratory data analysis is an important step in the process of model development. We will specifically be looking at the volume of the data, the target class distribution, and whether features like the cell area, radius and perimeter can be used to differentiate between benign and malignant cases. A lot of the insights gleaned in this section will be used to determine what features should be used for model training, what metrics should be used for model evaluation and how to validate the model interpretations obtained using the techniques that will be covered later in this chapter.

There are 569 patient cases in this dataset and 30 features in total. The features are all continuous. Figure 4.2 shows the proportion of cases that are benign and malignant. Out of the 569 cases, 357 of them (roughly 62.7%) are benign and 212 (roughly 37.3%) are malignant. This shows that the dataset is skewed or imbalanced. As we’ve seen in Chapter 3, we say that the data is imbalanced when there is a disproportionate number of examples or data points for a given class. Most machine learning algorithms work best when the proportion of samples for each class is roughly the same. This is because most algorithms are designed to minimize error or maximize accuracy and these algorithms tend to naturally bias towards the majority class. To recapitulate, there are two things to note when dealing with imbalanced datasets:

- Use the right performance metrics (like precision, recall and F1) when testing and evaluating the models.
- Resample the training data such that the majority class is either under-sampled or the minority class is over-sampled.

We will discuss this further in Section 4.3.2.

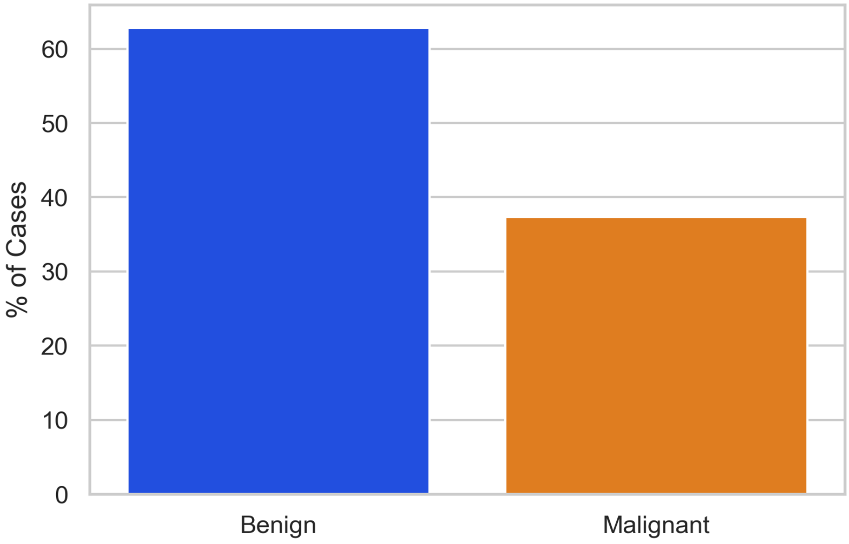


Figure 4.2: Distribution of Benign and Malignant Cases

Let’s now look at the distributions of the cell area, radius and perimeter and see if there are any major differences between the benign and malignant cases. Figure 4.3 shows the distributions of the mean cell area and worst or largest cell area, comparing the benign cases in blue and the malignant cases in orange.

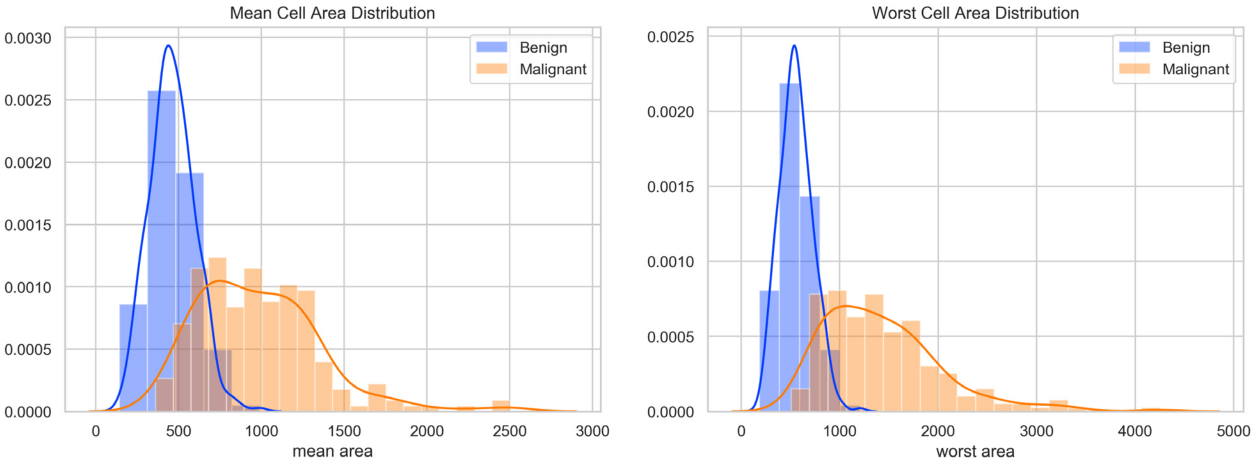


Figure 4.3: Cell Area Distribution Comparison of Benign and Malignant Cases

From Figure 4.3, we can see that if the mean cell area is greater than 750, then the case is much more likely to be malignant than benign. Also, if the worst or largest cell area is greater than 1000, then the case is much more likely to be malignant. There seems to be a good but weak separation between the malignant and benign cases by looking at just two features related to the cell area.

How about the cell radius and perimeter? Figures 4.4 and 4.5 show the distributions of the radius and perimeter respectively separating the benign cases in blue from the malignant cases in orange. We see a similar separation between the benign and malignant cases. For instance, a case with mean radius that is greater than 15 is much more likely to be malignant than benign. Also, a case with worst or largest cell perimeter of 100 is much more likely to be malignant.

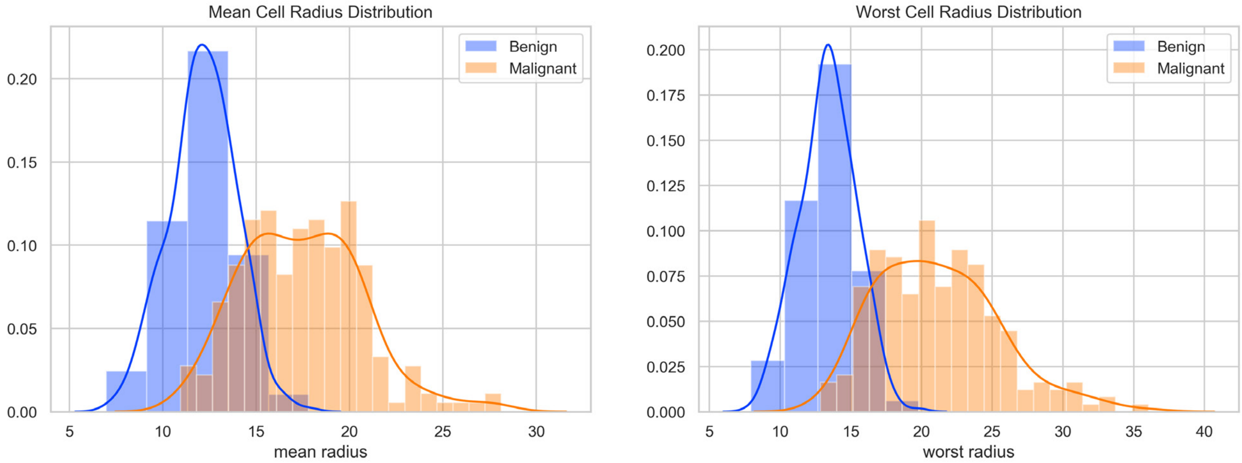


Figure 4.4: Cell Radius Distribution Comparison of Benign and Malignant Cases

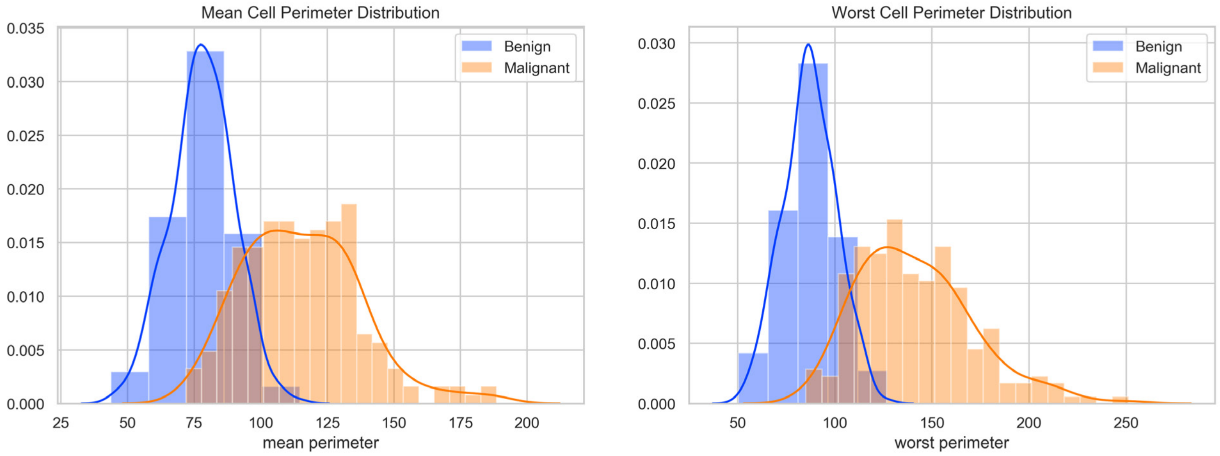


Figure 4.5: Cell Perimeter Distribution Comparison of Benign and Malignant Cases

The purpose of this analysis is to get a sense of how good the features are in being able to predict the target variable, i.e. whether a given case is benign or malignant. By looking at the distributions in Figures 4.3, 4.4 and 4.5, we can see pretty good signal in the six features that we've considered where there's good separation between the benign and malignant cases. We will also use these insights to validate the interpretations obtained through the LIME, SHAP and Anchors later in this chapter.

Let's finally look at how correlated each of the input features are with each other and the target variable. We know that the input features are continuous, but the target variable is discrete and binary. In the dataset, a malignant case is encoded as 0 and a benign case is encoded as 1. Since the input features and the target are all numerical values, we can use the Pearson or standard correlation coefficient to measure correlation. As we've seen in Chapter 2, the Pearson correlation coefficient measures the linear correlation between two variables and has a value between +1 and -1. If the magnitude of the coefficient is above 0.7, then that means really high correlation. If the magnitude of the coefficient is between 0.5 and 0.7, then that means moderately high correlation. If the magnitude of the coefficient is between 0.3 and 0.5, then that means low correlation and a magnitude is less than 0.3, then that means little to no correlation. You can easily compute the pairwise correlations using the `corr()` function provided by `pandas`. As an exercise, please reuse the code learned in Chapter 2 (Section 2.2) to compute and plot the correlation matrix. The code to load the dataset can be found in Section 4.3.1. The resulting plot for the breast cancer dataset is shown in Figure 4.6.

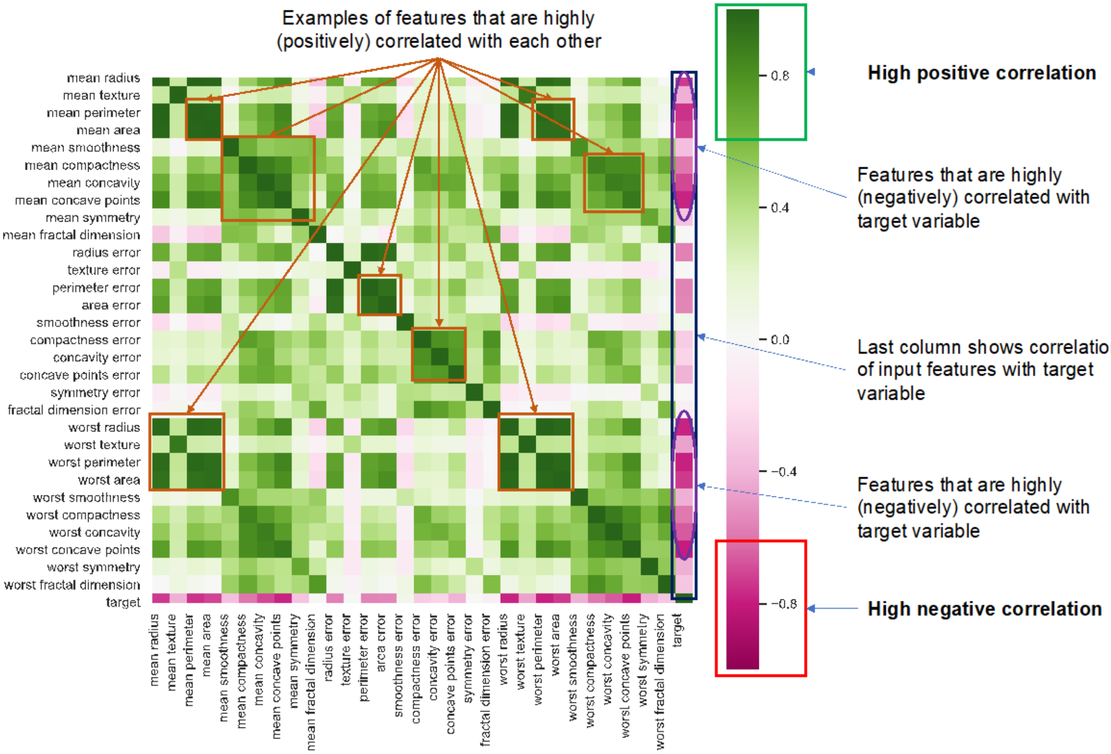


Figure 4.6: Correlation Plot of Input Features and the Target Variable

In Figure 4.6, let's first focus on the last column which shows the correlation of all the input features with the target variable. We can see that features like mean cell area, radius and perimeter are highly correlated with the target class. The correlation coefficient is however negative, which means that the larger the value for the features, the smaller the value for the target variable. This makes sense since the target class has a smaller value (i.e. 0) for the malignant class and a higher value (i.e. 1) for the benign class. As we've seen in Figures 4.3, 4.4 and 4.5, the larger the value for these features, the more likely that the case is malignant. We can also see that there are quite a few features that are highly correlated with each other. For instance, features like mean cell radius, area and perimeter are highly correlated with worst cell radius, area and perimeter. As we've seen in Chapter 2, features that are correlated with each other are said to be multicollinear or redundant. One way of dealing with multicollinearity is to remove redundant features for the model. We will discuss this further in the following section.

4.3 Deep Neural Networks

An Artificial Neural Network (ANN) is a system that is designed to loosely model a biological brain. It belongs to a broad class of machine learning methods called deep learning. The central idea of deep learning based on ANNs is to build complex concepts or representations from simpler concepts or features. An ANN learns a complex function mapping the input to the output composing of many simpler functions. In this chapter we will focus on ANNs consisting of multiple layers of units or neurons that are fully interconnected with each other. These are also called **Deep Neural Networks (DNNs)**, Fully Connected Neural Networks (FCNNs) or Multi-Layer Perceptrons (MLPs). In subsequent chapters, we will cover Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) that are more advanced structures of neural networks used for complex computer vision and language understanding tasks.

Figure 4.7 illustrates a simple ANN consisting of three types of layers – the input layer, the hidden layer and the output layer. The input layer acts as the input for your data. The number of units in the input layer is equal to the number of features in your dataset. In Figure 4.7, we are considering only two features from the breast cancer dataset namely, mean cell radius and mean cell area. This is why there are two units in the input layer.

The input layer is then connected to all the units in the first hidden layer. The hidden layer transforms the inputs based on the activation function used for its units. In Figure 4.7, the function `f` is used to represent the activation function for all the units in the hidden layer. The units in one layer are connected with units in another layer using edges. Each edge is associated with a weight which defines the strength of the connection between the units that it connects. Note that there is also a bias term connected to each of the units in the hidden layer and an edge weight of 1 is used for the bias term. A weighted sum of the inputs and the bias term is taken before it gets transformed by the activation function. If there is more than one hidden layer, then the ANN is said to be "deep". Hence, an ANN with two or more hidden layers is called a **DNN**.

The units in the final hidden layer are then connected to units in the output layer. In Figure 4.7, there is one unit in the output layer since for the breast cancer detection task, we have binary output where the given cell is either malignant or benign. The unit in the output layer also has an activation function `g` which transforms the inputs to that unit to an output prediction. One of the challenges in creating neural networks is to determine the structure of the neural network, i.e. how deep (number of hidden layers) and how wide (number of units in each layer) the network should be. We will briefly talk about how to determine and interpret the structure of the neural network in Section 4.4 and cover it in more detail in subsequent chapters when we look at CNNs and RNNs.

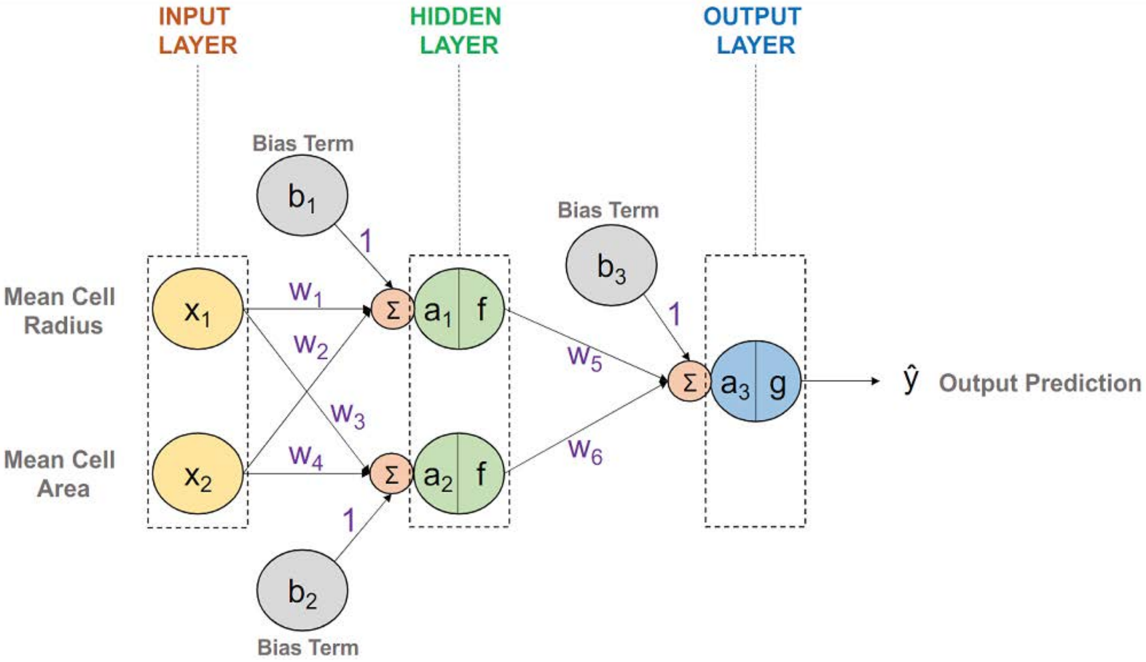


Figure 4.7: Illustration of an ANN

Let's now see how the input data is transformed into the output as it passes through the ANN. This is called forward propagation and is illustrated in Figure 4.8. The input data is fed through the units in the input layer. The values of the input units for the two features are represented as x_1 and x_2. These values are then propagated through the network in the forward direction through the hidden layers. At each unit in the hidden layer, a weighted sum of the inputs is first taken and then passed through an activation function. In Figure 4.8, the first unit in the hidden layer computes the weighted sum of the inputs x_1 and x_2 and the bias term b_1 to obtain the pre-activation value a_1. This is then passed through the activation function f to obtain $f(a_1)$. A similar set of operations happen at the second unit in the hidden layer. Note that the same activation function is used for both units in the hidden layer. We will discuss activation functions in more depth later in this section.

Once we have computed the outputs for the units at the hidden layer, these outputs are then fed as inputs to the units in the subsequent layer. In the illustration below, the outputs from the two units at the hidden layer are fed as inputs to the one unit in the output layer. Just as before, first a weighted sum of the inputs together with the bias term is taken to obtain the pre-activation value a_3. This is then passed through the activation function g to obtain the output of the unit as $g(a_3)$. The output of this final unit is meant to be an estimate of the target variable y, represented as $\hat{y}$. The weights for all the edges in the network will be randomly initialized at the start.

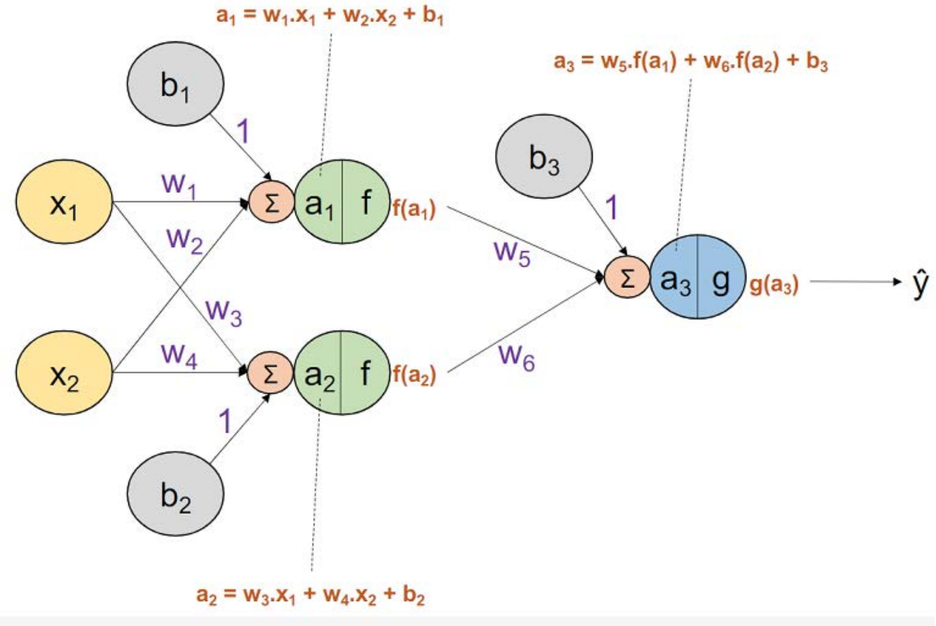


Figure 4.8: Illustration of forward propagation in an ANN

Now, the objective of the learning algorithm is to determine the weights of the edges or the strength of the connections between the units such that the output prediction is as close to the actual value for the target variable. How do you learn these weights? We will apply the same technique that we learned in Chapter 2 to determine the weights for a linear regression model, i.e. gradient descent. An optimum set of weights are those that minimize a cost or loss function. For regression problems, a common cost function is the squared error or squared difference between the predicted output and the actual output. For binary classification problems, a common cost function is the log loss or the Binary Cross Entropy (BCE) loss function.

The squared error cost function and its corresponding derivative with respect to the predicted output are shown in the following equations.

$$J(\hat{y}|y) = \frac{(\hat{y} - y)^2}{2}$$

$$J'(\hat{y}|y) = \hat{y} - y$$

The log loss or BCE loss function and its corresponding derivative with respect to the predicted output are shown below.

$$J(\hat{y}|y) = \begin{cases} -\log(\hat{y}), \text{ if } y = 1 \\ -\log(1-\hat{y}), \text{ if } y = 0 \end{cases}$$

$$J'(\hat{y}|y) = \begin{cases} -\frac{1}{\hat{y}}, \text{ if } y = 1 \\ \frac{1}{1-\hat{y}}, \text{ if } y = 0 \end{cases}$$

The cost function is said to be at a minimum (global or local) when the gradient of the cost function is 0 or close to 0. We can easily determine the weights for a linear regression or logistic regression type of problem as the number of weights is equal to the number of input features (plus an additional bias term). For a DNN, on the other hand, the number of weights is dependent on the structure of the network. The number of weights can easily explode as we add more units and layers in the network. Applying the gradient descent algorithm directly will not be computationally feasible. An efficient algorithm to determine these weights in a DNN is backpropagation.

The backpropagation algorithm for the simple ANN structure seen earlier is illustrated in Figure 4.9. Once you have evaluated the output of the network after forward propagation, the next step is to compute the cost or loss function and the gradient of the cost function with respect to the predicted output. Then visit nodes in reverse order and propagate an error signal that can be used to compute the gradient with respect to the weights for all the edges in the network. Let's go through it step by step by parsing Figure 4.9 from right to left.

We first compute the gradient of the cost function with respect to the predicted output variable. This is represented as `J'` in Figure 4.9. This gradient is then passed in the reverse direction through the unit in the output layer. Within the output layer, the local gradient of the activation function `g` is stored. This is represented as `g'`. The pre-activation value a_3 evaluated during forward propagation is also stored. These values are used to compute the output error signal of the unit, represented as e_1. The computed value is shown in Figure 4.9. It is essentially the gradient of the loss function weighted by the local gradient of the activation function. This error signal e_1 is then propagated to the two units in the hidden layer. The process is then repeated to compute the output error signals of the hidden units. Once the error signals have been propagated through the network and we have reached the input layer, then the gradient with respect to each edge weight can be computed by multiplying the error signal flowing through it during backward propagation by the value that flowed through it during forward propagation. There are multiple online resources and books that explain backpropagation and the mathematical concepts in great depth. We will therefore not be covering these concepts in more depth in this chapter.

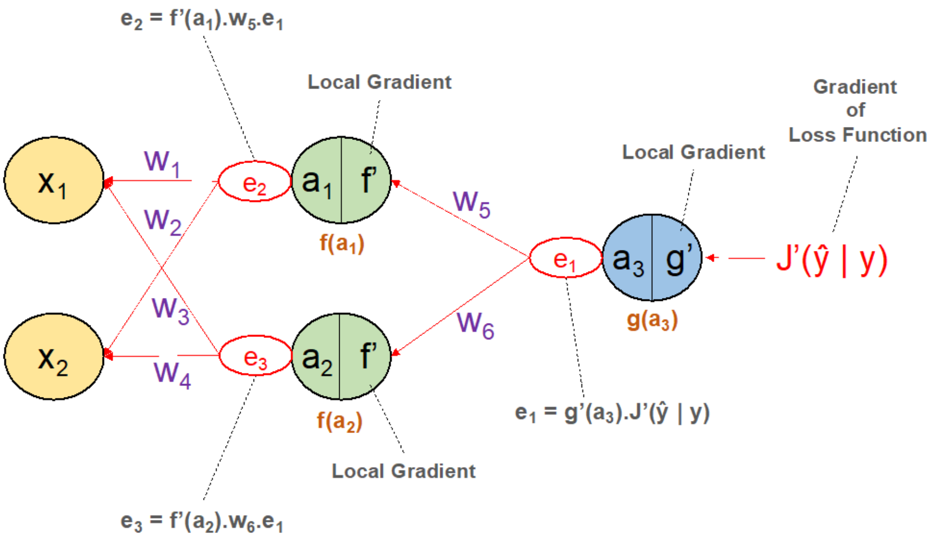
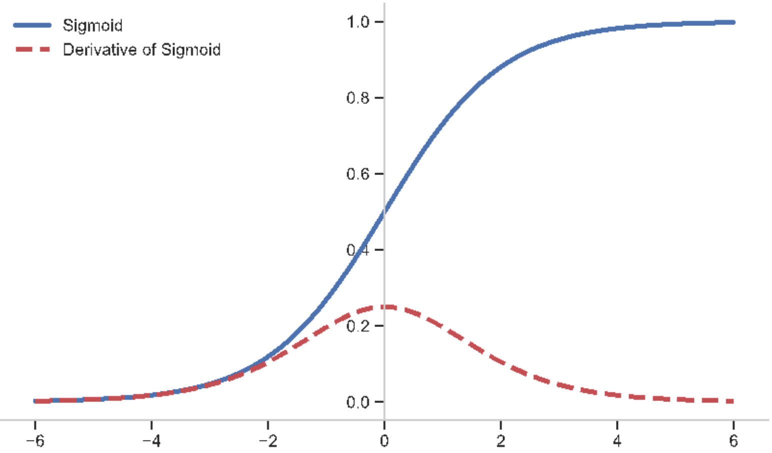
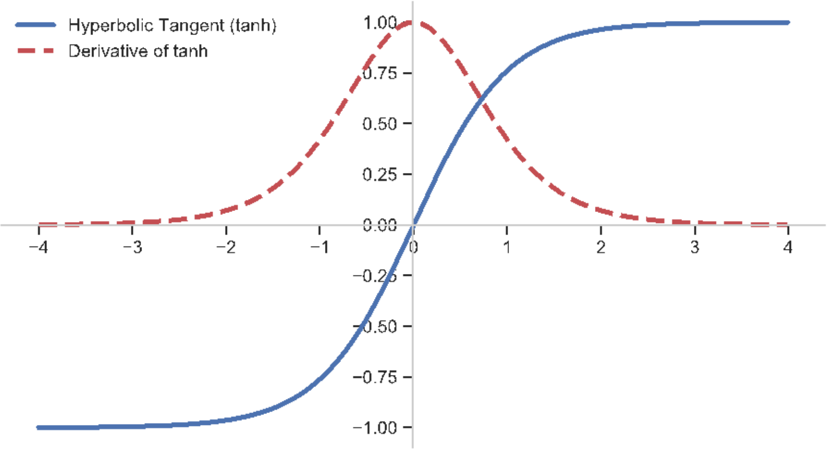


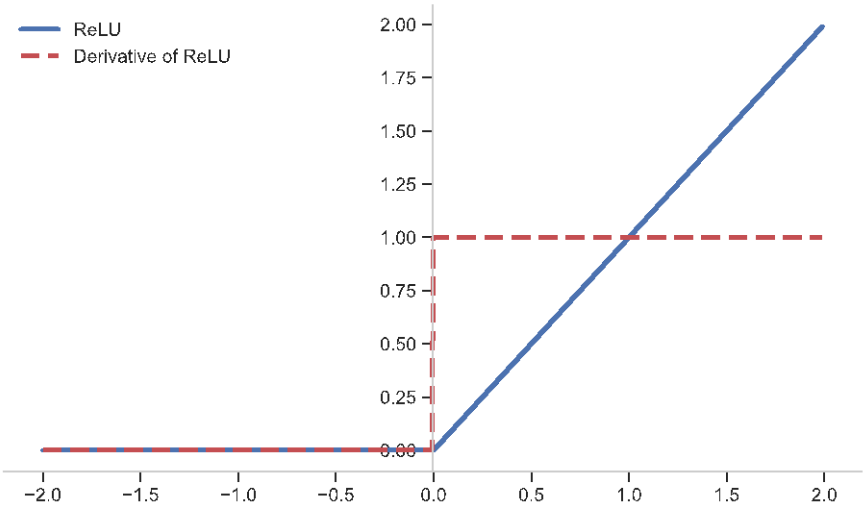
Figure 4.9: Illustration of backward propagation of error signals in an ANN

The activation function is a very important feature within a neural network. It decides whether a neuron should be activated and by how much. The properties of an activation function are that it is differentiable (i.e. the first derivative exists) and monotonic (i.e. it is either entirely non-decreasing or non-increasing). Common activation functions used in neural networks are the sigmoid function, hyperbolic tangent (tanh) and the Rectified Linear Unit (ReLU). These functions are defined in Table 4.1.

Table 4.1: Common activation functions used in neural networks

Activation Function	**Description**
Sigmoid	The sigmoid function is defined as follows: $sigmoid(x) = 1 \,/\, (1 + \exp(-x))$

	 The output of the function ranges from 0 to 1. It is differentiable and is monotonic as shown in the figure above.
Hyperbolic Tangent (tanh)	The hyperbolic tangent function is defined as follows: $\tanh(x) = 2 * sigmoid(2x) - 1$  The output of the function ranges from -1 to 1. It is also differentiable and monotonic as shown in the figure above.
Rectified Linear Unit (ReLU)	The ReLU function is defined as follows: $ReLU(x) = \max(0, x)$ The output of the function ranges from 0 to the infinity (depending on the value

	of the input x). It is differentiable and is monotonic as shown in the following figure. 

The sigmoid activation function is typically used for classifiers as the output of the function ranges from 0 to 1. For the breast cancer detection problem in this chapter, we will be using the sigmoid function as the activation function g in the output layer. The hyperbolic tangent function has similar properties as the sigmoid but the output ranges from -1 to 1. Both the sigmoid and hyperbolic tangent activation functions suffer from the problem of vanishing gradients. This is because the gradients for both of these functions are 0 (also said to be saturated) for very large or small values of the input, as seen in Table 4.1.

ReLUs are the most widely used activation functions in neural networks as it handles the vanishing gradient problem well. We can see that the value of the ReLU is 0 if the input is negative. Due to this property, not all neurons are activated at the same time. It is therefore much more computationally efficient. In practice, for simplicity, the same activation function is used for all the units in the hidden layers of the neural network.

4.3.1 Data Preparation

Let’s now train a DNN for the breast cancer detection problem. We will be using Pytorch to build and train the network. PyTorch is a library that facilitates building neural networks in Python. PyTorch is gaining popularity among researchers and machine learning practitioners in the industry due to its ease of use. There are other libraries such as TensorFlow and Keras that can be used to build neural networks as well, but we will be focusing on PyTorch in this book. Since the library is pythonic, it will be easier for data scientists and engineers who are already familiar with Python to use it. To learn more about PyTorch, please see Appendix A.

The first step before training the DNN is to prepare the data. The code below shows how to load the data, split it into training, validation and test sets, and then transform them into inputs for the PyTorch implementation of the network.

```
import numpy as np #A

from sklearn.datasets import load_breast_cancer #B
from sklearn.model_selection import train_test_split #C

import torch #D
from torch.autograd import Variable #D

data = load_breast_cancer() #E
X = data['data'] #E
y = data['target'] #E

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.3, random_state=24) #F
X_val, X_test, y_val, y_test = train_test_split(X_val, y_val, test_size=0.5, random_state=24)
       #G

X_train = Variable(torch.from_numpy(X_train)) #H
X_val = Variable(torch.from_numpy(X_val)) #H
y_train = Variable(torch.from_numpy(y_train)) #H
y_val = Variable(torch.from_numpy(y_val)) #H
X_test = Variable(torch.from_numpy(X_test)) #H
y_test = Variable(torch.from_numpy(y_test)) #H
```

#A: Import numpy used for loading the dataset as vectors and matrices
#B: Import the breast cancer dataset available in scikit-learn
#C: Import the train_test_split function available in scikit-learn
#D: Import Pytorch and the Variable data structure to store the input dataset as tensors
#E: Load the breast cancer dataset and extract the features and target
#F: Split into train and validation/test sets
#G: Split the validation/test set into two equal sets - val and test
#H: Initialize the train, val, test sets into Pytorch tensors

Note that 70% of the data is used for training, 15% for validation and the remaining 15% as the hold-out test set. Let’s now check to see if the distribution of the target variable is similar across the three sets. This is shown in Figure 4.10. We can see that roughly 60-62% of the cases are benign (where target variable = 1) and 38-40% of the cases are malignant (where target variable = 0) in all three sets.

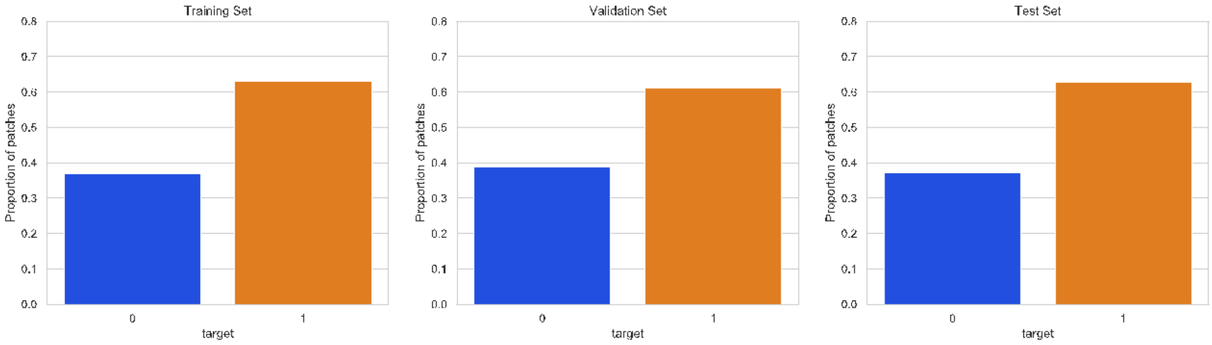


Figure 4.10: Target variable distribution across the training, validation and test sets

4.3.2 Training and Evaluating DNNs

Now that we've prepared the data, the next step is to define the DNN. We will be creating a class where the number of layers and units can be passed in as attributes.

```
class Model(torch.nn.Sequential): #A
    def __init__(self, layer_dims): #B
        super(Model, self).__init__() #C
        for idx, dim in enumerate(layer_dims): #D
            if (idx < len(layer_dims) - 1): #E
                module = torch.nn.Linear(dim, layer_dims[idx + 1]) #E
                self.add_module("linear" + str(idx), module) #E
            else: #F
                self.add_module("sig" + str(idx), torch.nn.Sigmoid()) #F
            if (idx < len(layer_dims) - 2): #G
                self.add_module("relu" + str(idx), torch.nn.ReLU()) #G
```

#A: Create a Model class that inherits from the Pytorch Sequential class
#B: Pass the number of layers and units for each layer as an array to the constructor
#C: Initial the Pytorch Sequential super class
#D: For each element in the array, extract the index and the number of units for that layer
#E: Create a layer module containing all Linear units until the final output layer
#F: Use the sigmoid activation function for the unit in the output layer
#G: Use the ReLU activation function for all the units in the hidden layers

Note that the DNN `Model` class inherits from the Pytorch `Sequential` class, which layer modules in the sequential order that they are initialized. For the input layer and hidden layers, `Linear` units are used to compute a weighted sum of all the inputs to that unit. For the hidden layers, the ReLU activation function is used. The final output layer consists of a single unit where the sigmoid activation function is used. The output of a sigmoid activation function is a score that is between 0 and 1. This acts as a proxy for a probability measure for the positive class in the classification task. In this case, the positive class is benign. Now that we have the `Model` class, let's initialize it as follows.

```
dim_in = X_train.shape[1] #A
dim_out = 1 #B
layer_dims = [dim_in, 20, 10, 5, dim_out] #C
model = Model(layer_dims) #D
```

#A: Number of units for the input layer is equal to the number of features in the training set
#B: Number of units in the output layer is 1 since we are dealing with a binary classification problem
#C: Initialize the layer dimensions array to define the structure for the DNN
#D: Initialize the DNN model with the predefined structure

If you print the model, using the command `print(model)`, you will get the following output that summarizes the structure of the DNN.

```
Model(
  (linear0): Linear(in_features=30, out_features=20, bias=True)
  (relu0): ReLU()
  (linear1): Linear(in_features=20, out_features=10, bias=True)
  (relu1): ReLU()
  (linear2): Linear(in_features=10, out_features=5, bias=True)
  (relu2): ReLU()
  (linear3): Linear(in_features=5, out_features=1, bias=True)
  (sig4): Sigmoid()
)
```

From the output above, you can see that the DNN consists of 1 input layer, 3 hidden layers and 1 output layer. The input layer consists of 30 units since there are 30 input features in the dataset. The first hidden layer consists of 20 units, the second hidden layer consists of 10 units and the third hidden layers consist of 5 units. The ReLU activation function is used for all the units in the hidden layers. Finally, the output layer consists of a single unit with a sigmoid activation function. The number of units in the input and output layers must be 30 and 1 respectively for this dataset because the number of features is 30 and only a single output is required for the binary classification task. You are however free to tune the number of hidden layers and the number of units in each hidden layer depending on which structure gives the best performance. You can use the validation set to determine these hyperparameters.

With the model in place, let's now define the loss function and the optimizer that will be used to determine the weights during backpropagation.

```
criterion = torch.nn.BCELoss(reduction='sum') #A
optimizer = torch.optim.Adam(model.parameters(), lr=0.001) #B
```

#A: Initialize the Binary Cross Entropy (BCE) loss as the criterion for optimization
#B: Use the Adam optimizer with learning rate of 0.001 to determine the weights during backprop

As mentioned in the previous section, the BCE loss is used as the criterion of optimization for binary classification problems. We are also using the Adam optimizer here with a predefined initial learning rate to determine the edge weights during backpropagation. The Adam optimizer is a technique that adaptively determines the learning rate for the gradient descent algorithm. You can find more details on the Adam optimization technique in this blog post. Finally, train the model as follows.

```
num_epochs = 300 #A
for epoch in range(num_epochs):
    y_pred = model(X_train.float()) #B
    loss = criterion(y_pred, y_train.view(-1, 1).float()) #C

    optimizer.zero_grad() #D
    loss.backward() #E
    optimizer.step() #F
```

#A: Initialize the number of epochs to 300
#B: In each epoch obtain the output of the DNN for the training set
#C: Compute the BCE loss for the training set
#D: Zero out the gradients before backpropagating
#E: Compute the gradient with respect to every parameter/edge weight
#F: Update the weights based on the current gradients

Note that we are training the model over 300 epochs. An epoch is a hyperparameter that defines the number of times we propagate the entire training set in the forward and backward directions through the neural network. During each epoch, we first obtain the output of the DNN by propagating the training set through the network in the forward direction. We then compute the gradient with respect to every parameter or edge weight and update the weights during backpropagation. Note that the gradients are set to 0 in each epoch before starting backpropagation because PyTorch accumulates gradients during backward passes by default. If we do not set the gradients to 0, the weights will not be updated correctly.

The next step is to evaluate the model performance using the test set. Since this is a classification problem, we will be using the same metrics that we used in Chapter 3 for the student grade prediction problem. The metrics we will use are precision, recall and the F1 score. We will compare the performance of the trained DNN model with a reasonable baseline model. As seen in Section 4.2, majority of the cases in the dataset are benign. We will therefore consider a baseline model that always predicts benign. This is not ideal as we will get all the malignant cases wrong. In a real-life situation, the baseline model will typically be predictions made by a human or expert or an existing model that the business is using. For this example, unfortunately, we do not have access to that information and so will compare the model with a baseline that always predicts benign.

Table 4.2 shows the 3 key performance metrics used to benchmark the models, i.e. precision, recall and F1. If we look at the recall metric, the baseline model does better than the DNN. This is expected since the baseline model is predicting the positive class all the time and will therefore get all the positive cases right. The recall with respect to the negative class will however be 0 for the baseline model. Overall, though, the DNN model does much better than the baseline achieving a precision of 98.1% (+35.4% better than the baseline) and an F1 score of 96.2% (+19.1% better than the baseline).

As an exercise, I highly encourage you to tune the hyperparameters of the model and see if you can improve the performance of this model. You can tune the structure of the network by changing the number of hidden layers and units in each layer, and also the number of epochs used for training. In Section 4.2 (Figure 4.6), we also saw that some of the input features are highly correlated with each other. The performance of the model could be further improved by removing some of the redundant features. As another exercise, perform feature selection and determine the best subset of the features that maximizes the performance of the model.

Table 4.2: Performance Comparison of Baseline Model with the DNN Model

	Precision (%)	Recall (%)	F1 Score (%)
Baseline Model 1	62.7	100	77.1
DNN Model	98.1 (+35.4)	94.4 (-5.6)	96.2 (+19.1)

With the DNN model performing better than the baseline, let’s now interpret it and understand how the black-box model arrived at the final prediction.

4.4 Interpreting DNNs

As we’ve seen in the previous section, to make a prediction with a DNN, data is passed through multiple layers, each layer consisting of multiple units. The inputs to each layer go through a non-linear transformation based on the weights and the activation function used for the units. A single prediction can involve a lot of mathematical operations depending on the structure of the neural network. For the relatively simple architecture used in the previous section for breast cancer detection, a single prediction involved roughly 890 mathematical operations based on the number of training parameters or weights (as shown below).

```
+----------------+------------+
|    Modules     | Parameters |
+----------------+------------+
| linear0.weight |    600     |
|  linear0.bias  |     20     |
| linear1.weight |    200     |
|  linear1.bias  |     10     |
| linear2.weight |     50     |
|  linear2.bias  |     5      |
| linear3.weight |     5      |
|  linear3.bias  |     1      |
+----------------+------------+
Total Trainable Parameters: 891
```

This can very easily explode into millions of operations as we add more hidden layers and units per hidden layer. This is what makes DNNs black boxes as it becomes really difficult to understand what transformations each layer is doing and how the model arrives at the final prediction. We will see in later chapters that it becomes even more difficult with more complex structures like CNNs and RNNs.

One way of interpreting DNNs is by looking at the weights or the strengths of the edges connected to the units in the input layer. This could as a proxy to determine the overall influence of the input features on the output prediction. It will however not give us an accurate measure of the feature importance as we saw for white-box models and tree ensembles in the previous chapters. The main reason is because the neural network learns a representation of the input at the hidden layers. The initial input features are transformed into intermediate features and concepts. Therefore, the importance of those input features is not just dictated by the edges connected to the units in the input layer. So how do we interpret DNNs?

There are multiple ways of interpreting DNNs. We can use model agnostic methods learned in the previous chapter that are global in scope. In the previous chapter, we learned about PDPs and feature interaction plots. These were model agnostic techniques, i.e. they were interpretability techniques that could work with any machine learning model. They were also global in scope, i.e. they were techniques that looked at the overall influence of the model on the final prediction. PDPs and feature interaction plots are easy and intuitive, and they are great tools to shed light into how specific feature values influence the model output. We also learned how they could be used to uncover potential issues like data and model bias. We could very easily apply these techniques to the DNN model trained for breast cancer detection. For PDPs and feature interaction plots to work however, the input features for the model have to be independent and we have seen in Section 4.2 that they are not.

In the subsequent sections, we will learn about more advanced model agnostic techniques – specifically focusing on LIME, SHAP and Anchors. These interpretability techniques are local in scope, i.e. they focus on a specific instance or example to interpret. In later chapters, we will learn about feature attribution methods that aim to quantify the contribution of each input feature on the final prediction and also learn how to dissect the neural network and visualize the features learned by the intermediate hidden layers and units.

4.5 LIME

LIME is an acronym that stands for Local Interpretable Model-agnostic Explanations and it was proposed in 2016 by Marco Tulio Ribeiro and team. Let's break this technique down. In the previous section, we trained a DNN that learned how to separate the benign cases from the malignant cases using 30 features. Let's simplify this by collapsing the feature space into 2D space as shown in Figure 4.11. The figure illustrates the complex decision function learned by the DNN where the model separates the benign cases in blue from the malignant cases in red.

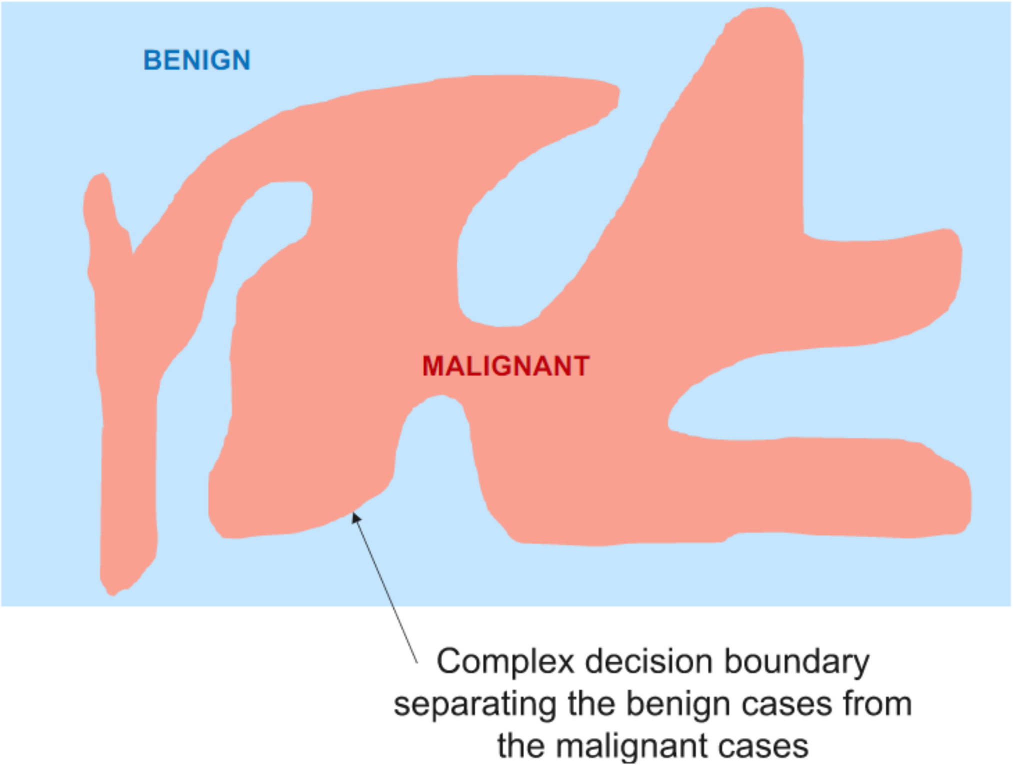

BENIGN
MALIGNANT
Complex decision boundary separating the benign cases from the malignant cases

Figure 4.11: 2D illustration of a complex decision boundary learned by the DNN (or any black-box model) to separate the benign cases from the malignant cases

The way LIME works is to first pick an example to interpret. This is shown in Figure 4.12 where we have picked one malignant case to interpret. The aim is to probe the model as often as needed to interpret how the model comes up with the prediction for that picked example. You can probe the model by **perturbing** the dataset to get the model predictions for that new dataset.

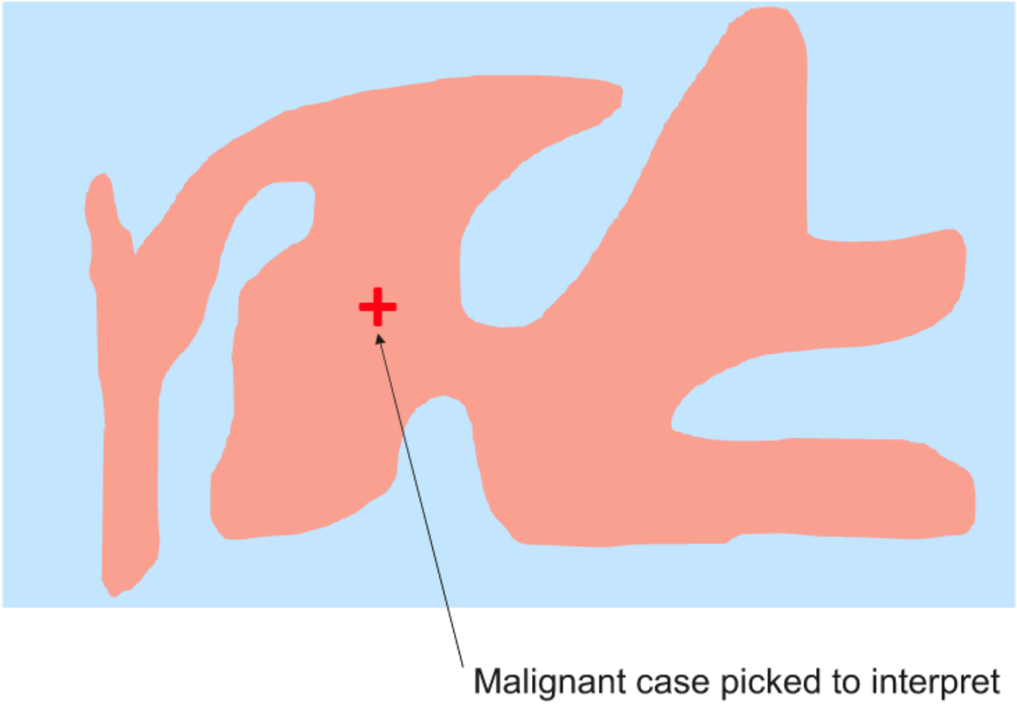


Figure 4.12: Illustration of an instance picked to interpret using LIME

How do you create this new **perturbed dataset**? Given the training data, calculate the key summary statistics for each feature. For numerical or continuous features, calculate the mean and standard deviation. For categorical features, compute the frequency of each value. Then create a new dataset by sampling based on these summary statistics. For numerical features, data is sampled from a Gaussian distribution given the mean and standard deviation for that feature. For categorical features, sample based on the frequency distribution or probability mass function. Once you've created this dataset, probe the model by getting predictions for them. This is shown in Figure 4.13. The picked instance is shown as the big red plus. The malignant and benign predictions on the perturbed dataset are shown as small red pluses and blue circles respectively.

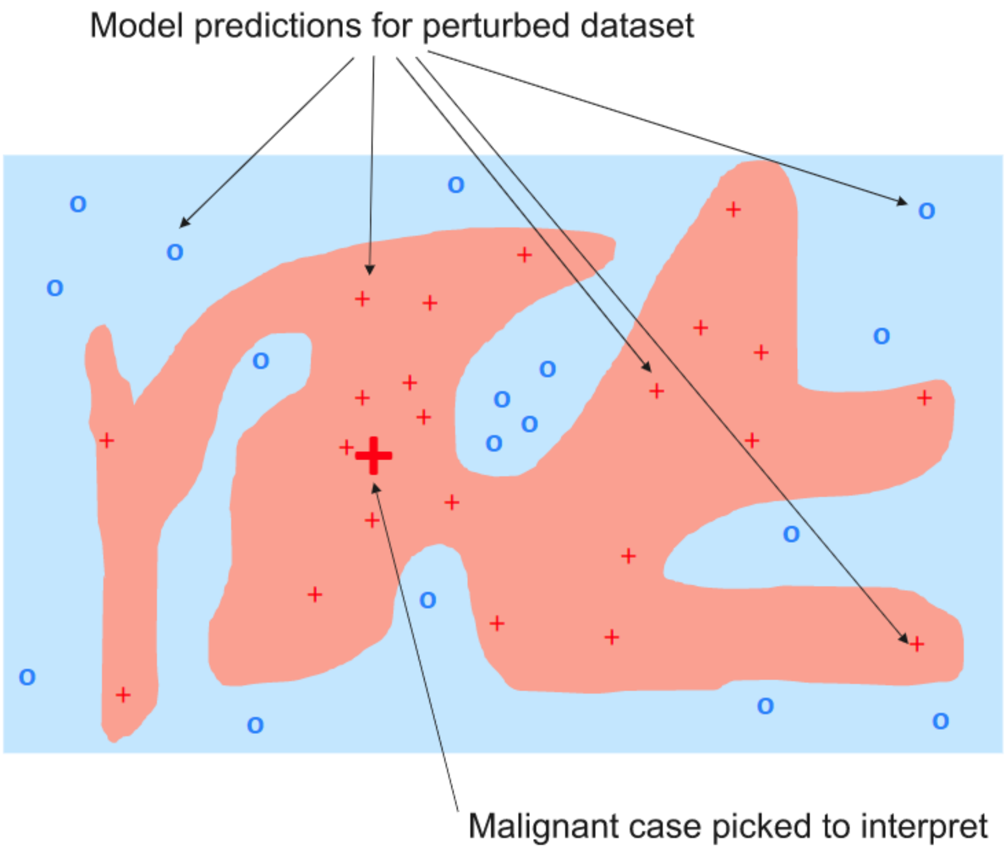

Model predictions for perturbed dataset
Malignant case picked to interpret

Figure 4.13: Illustration of generated or perturbed dataset and the corresponding model predictions

Once you have created the perturbed dataset and obtained the model predictions for them, weight these new samples by their proximity to the picked instance. This is done to interpret the picked instance by looking at cases similar to it in terms of features. The locality of the interpretation is captured by this weighting. Hence, the “local” in the acronym LIME. Figure 4.14 shows the perturbed samples that are close to the picked instance given a higher weight.

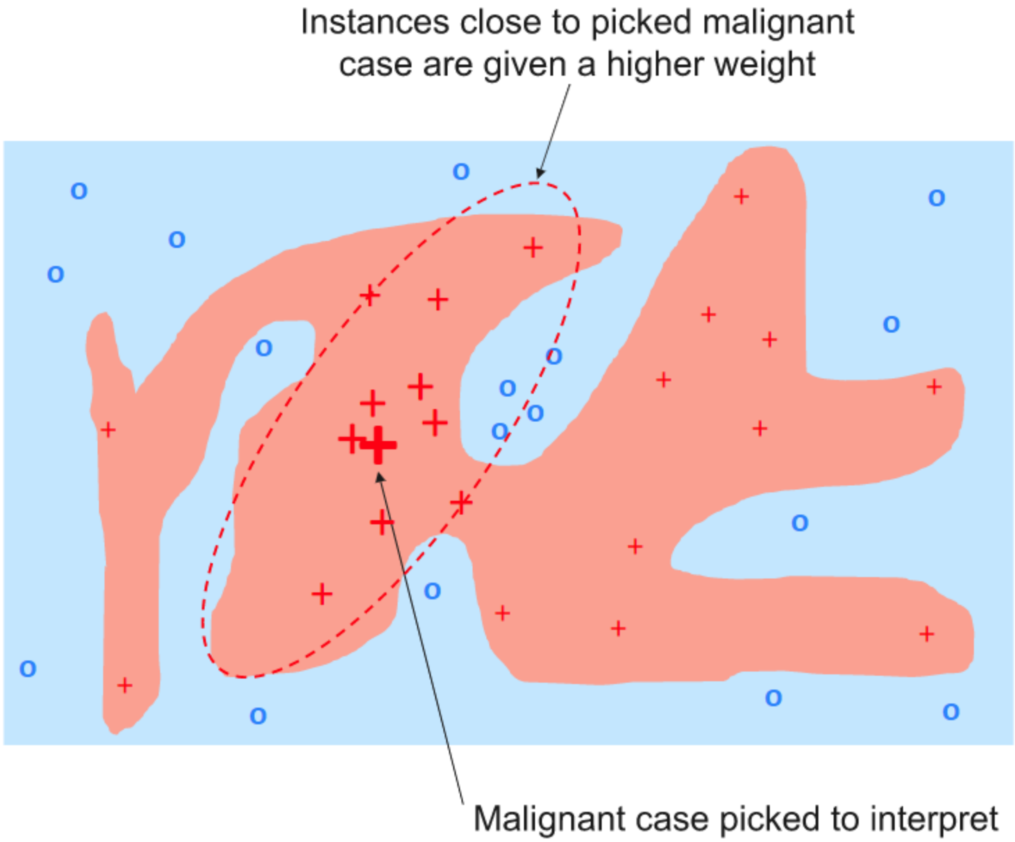

Instances close to picked malignant case are given a higher weight
Malignant case picked to interpret

Figure 4.14: Illustration of weighted instances in close proximity to the picked instance to interpret

Now, how do you weight the samples based on its proximity to the picked instance? In the original paper, the authors use the exponential kernel function. The **exponential kernel function** takes two parameters as inputs:

1. **Distance of perturbed sample from picked instance**: For the breast cancer dataset (or tabular data in general), Euclidean distance is used to measure the distance of the perturbed sample from the picked instance in the feature space. Euclidean distance is also used for images. For text, the cosine distance measure is used.
2. **Kernel width**: This is a hyperparameter that can be tuned. If the width is small, only samples that are close to the picked instance will influence the interpretation. If the width is large, however, samples that are further away can influence the interpretation. This is an important hyperparameter and its impact on the interpretation will be studied in greater depth later. By default, the kernel width is set to $0.75 * \sqrt{Number\ of\ features}$.

So, for the model with 30 input features, the default kernel width is 4.1. The value of the kernel width can range from 0 to infinity.

The property of the exponential kernel function is that samples closer to the picked instance in terms of distance will have a larger weight than samples further away.

The final step is to fit a white-box model that is easily interpretable on the weighted samples. In LIME, linear regression is used and as we've seen in Chapter 2, the weights of the linear regression model can be used to interpret the importance of features for that picked instance. Hence the "interpretable" in the acronym LIME. We therefore get an interpretation that is locally faithful and since we're fitting a linear surrogate model, LIME is totally agnostic of the DNN or black-box model. Hence the "model-agnostic" in the acronym LIME. Figure 4.15 illustrates the linear surrogate model (shown by the dashed grey line) that is faithful to the region near and around the instance picked to interpret.

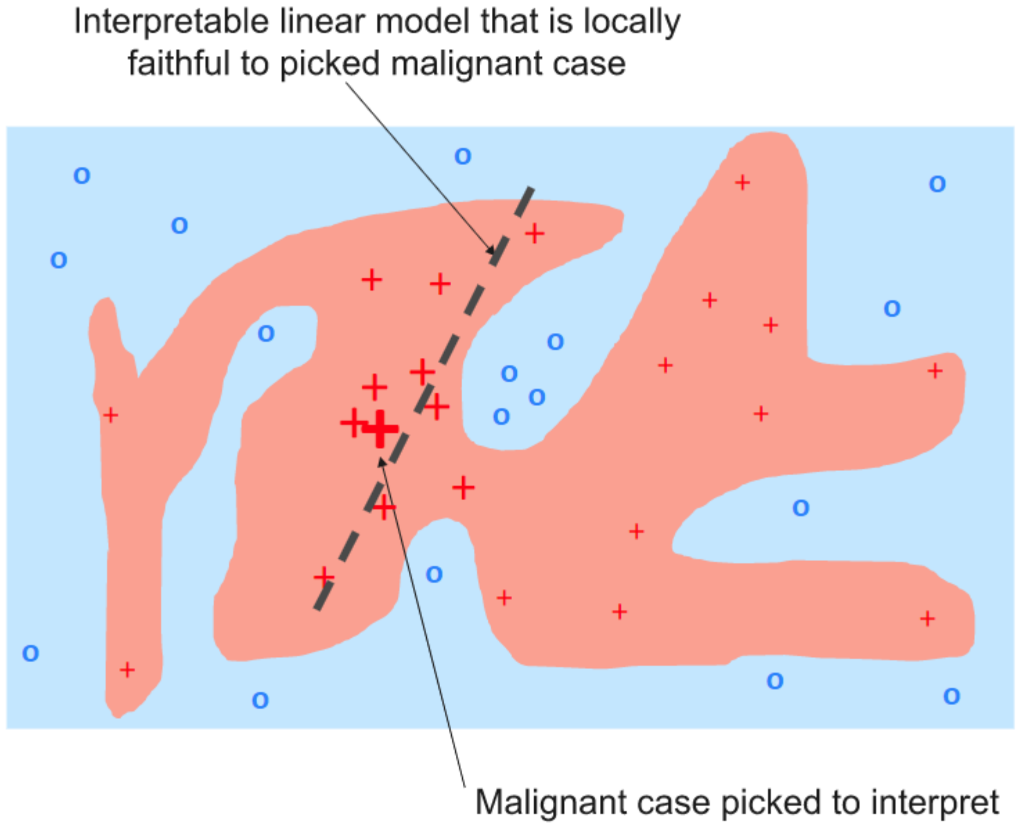

Interpretable linear model that is locally faithful to picked malignant case
Malignant case picked to interpret

Figure 4.15: Illustration of linear model that is used to interpret the picked instance using the weighted samples around it

Let’s now get our hands dirty and see LIME in action for the breast cancer diagnostics DNN model that we trained earlier. First, install the LIME library using pip as follows.

```
pip install lime
```

After installing, the first step is to initialize a LIME explainer object. Since the dataset is tabular, we will be using the `LimeTabularExplainer` class. Other explainer classes are `LimeImageExplainer` to explain models that use images as inputs and `LimeTextExplainer` for text. We will make use of the `LimeImageExplainer` class in the next chapter when we deal with images.

```
import lime #A
import lime.lime_tabular #A

explainer = lime.lime_tabular.LimeTabularExplainer(X_train.numpy(), #B
                          feature_names=data.feature_names, #C
                          class_names=data.target_names, #D
                          discretize_continuous=True) #E
```

#A: Import the library and the relevant modules
#B: Initialize the explainer using the training dataset
#C: Provide the feature names
#D: Provide the target class names (benign/malignant)
#E: Discretize the continuous variables to reduce computational complexity

Let’s now pick two cases to interpret – one benign and one malignant. We will use the test set here where we pick the first benign and malignant cases as shown in the following code.

```
benign_idx = np.where(y_test.numpy() == 1)[0][0]
malignant_idx = np.where(y_test.numpy() == 0)[0][0]
```

We will need to create a helper function to provide the predictions of the DNN model for the perturbed dataset. This is shown below.

```
def prob(data):
    return model.forward(Variable(torch.from_numpy(data)).float()).\
     detach().\
     numpy().\
     reshape(-1, 1)
```

We will also need to create another function to plot the LIME interpretation in `matplotlib`. You can create this plot using the library, but it doesn’t allow customizations. This is why we’ve created this helper function so that you add titles, labels, change the colors and even create your own plots using the LIME interpretation.

```
def lime_exp_as_pyplot(exp, label=0, figsize=(8,5)):
    exp_list = exp.as_list(label=label)
    fig, ax = plt.subplots(figsize=figsize)
    vals = [x[1] for x in exp_list]
    names = [x[0] for x in exp_list]
    vals.reverse()
    names.reverse()
    colors = ['green' if x > 0 else 'red' for x in vals]
    pos = np.arange(len(exp_list)) + .5
    ax.barh(pos, vals, align='center', color=colors)
    plt.yticks(pos, names)
    return fig, ax
```

Let’s now interpret the first benign case. This is shown below where we pass the picked benign case to the LIME explainer.

```
bc1_lime = explainer.explain_instance(X_test.numpy()[benign_idx], #A
                                      prob, #B
                                      num_features=5, #C
                                      top_labels=1) #D
f, ax = lime_exp_as_pyplot(bc1_lime) #E
```

#A: Pass the features of the picked benign case to the function
#B: Pass the helper function that provides the predictions for the perturbed dataset
#C: Limit the number of features for the linear surrogate model to 5
#D: The top label or positive class is 1
#E: Use the helper function to plot the LIME interpretation

Note that we are limiting the number of features for the linear surrogate model to 5. LIME uses a ridge regression model as the surrogate model by default. Ridge regression is a variant of the linear regression model that allows for variable selection or parameter elimination through regularization. By using a high regularization parameter, we can create sparse models that pick only a few top features for prediction. We can use a low regularization parameter for less sparsity. The resulting LIME interpretation for the benign case is shown in Figure 4.16.

For the benign case used to interpret using LIME, the DNN model predicted that it was benign with probability 0.99 or confidence 99%. In order to understand how it arrived at that prediction, the figure below shows the top five most important features for the linear surrogate model and their corresponding weights or importance. It looks like the most important feature was the worst area with a large positive weight. According to LIME, the reason why the model predicted benign was because the worst area value was between 511 and 683.95. How did LIME get these range of values? It is based on the standard deviation of the weighted perturbed dataset used by the linear surrogate model. Now, does this interpretation make sense? In order to validate this, we must go back to the exploratory data analysis that we did in Section 4.2. We saw in Figure 4.3 that when the worst or largest cell area is less than 700, then there are a lot more cases that are benign than malignant.

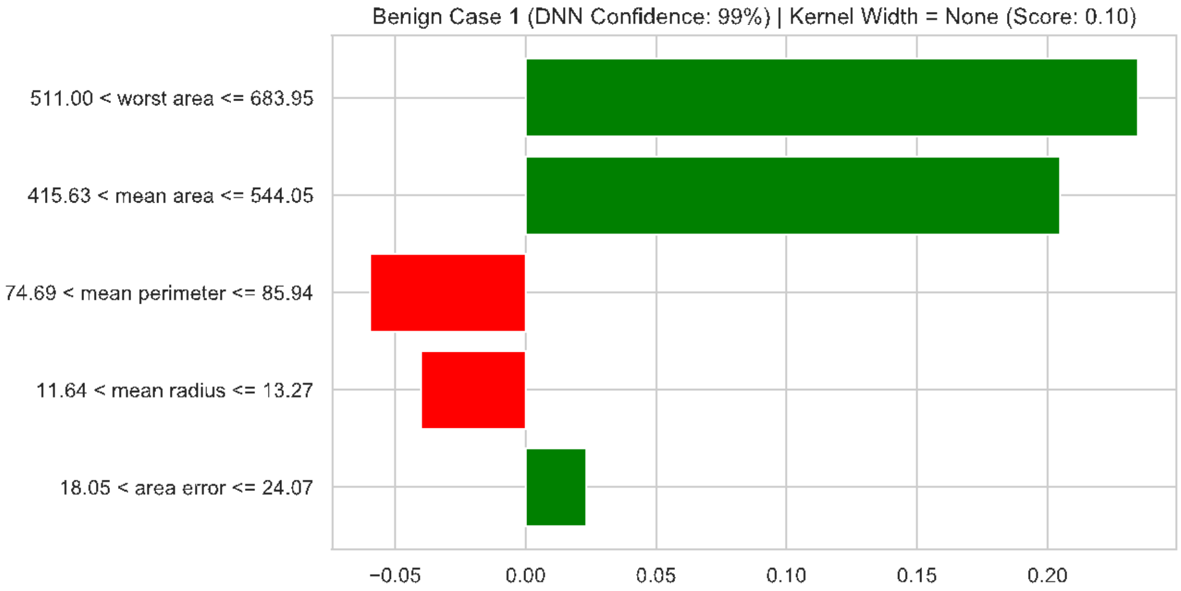


Figure 4.16: LIME Interpretation of Benign Case 1 where DNN Model predicted Benign with Confidence 99%

If we now look at the second most important feature identified by LIME, we can see that if the mean area is between 415.63 and 544.05, then it is much more likely for the case to be benign. This is further validated by our observation made in Figure 4.3 earlier. We can also make a similar observation for the third most important feature, i.e. mean perimeter. You might have observed the kernel width and a score in the title in Figure 4.16. We will come to this in a bit.

Let’s now look at the first malignant case in the test set to interpret using LIME. We can use the same code as before but just remember to pick the right feature values from the test set using `malignant_idx`. As an exercise, I encourage you to do that yourself. The resulting LIME interpretation is shown in Figure 4.17. The two most important features are the same as the benign case, but the range of values are different. Moreover, the weight for the most important features (worst cell area) is also negative. This makes sense because we expect the feature to have a negative effect on the model’s output. The DNN is trained to predict the probability of the positive class, which is in this case, benign. Therefore, if the case is malignant, we expect the output of the model to be as low as possible, i.e. the probability that the case is benign must be as low as possible.

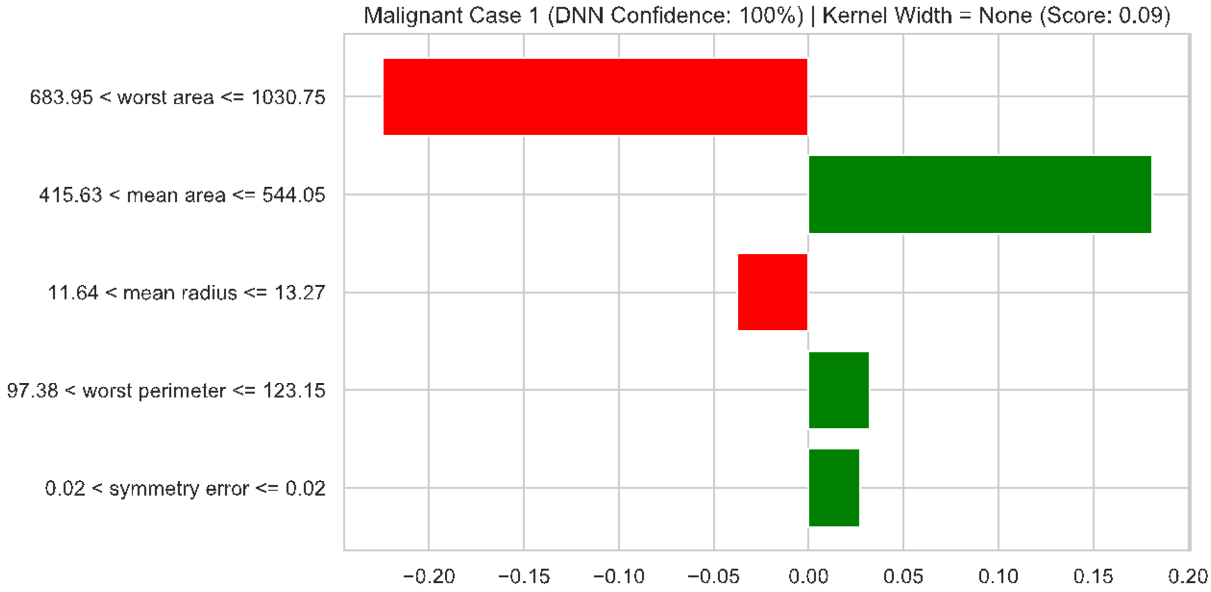


Figure 4.17: LIME Interpretation of Malignant Case 1 where DNN Model predicted Malignant with Confidence 100%

For this malignant case, the DNN predicts that the case is benign with probability 0. This means that the model is 100% confident that the case is malignant. Now let's inspect the feature value ranges. We can see that the model predicted malignant because the worst or largest cell area is greater than 683.95 but less than 1030.75. This makes sense since from the exploratory analysis, we observed more malignant cases than benign cases in that range (see Figure 4.3). We can make similar observations for the other features.

IMPACT OF THE KERNEL WIDTH

The kernel width is an important hyperparameter for LIME and it is important to call this out. Picking the right kernel width is important and it has an impact on the quality of the interpretation. We can't pick the same kernel width for all instances that we wish to interpret. The choice of width has an impact on the weighted perturbed samples that LIME considers for the linear surrogate model. If we choose a large kernel width, samples further away from the picked instance will have an influence on the linear surrogate model. This may not be desirable as we want the surrogate model to have as locally faithful to the original black-box model. By default, the LIME library uses a kernel width that is the square root of the number of features multiplied by a factor of 0.75. So, if `kernel_width = None`**, then the default value is used. It may be the case that the same kernel width may not be applicable for all instances that need to be interpreted using LIME. In order to evaluate the quality of the interpretation, LIME provides an explanation or fidelity score. The parameter is called** `score` **for the resulting LIME explanation. A higher score means that the linear model used by LIME is a**

good approximation of the black-box model. The kernel width and the LIME fidelity score are shown in the title for Figures 4.16 and 4.17.

Let's now look at the impact of the kernel width by looking at another benign case. We have picked the second case here from the test set as shown below.

```
benign_idx2 = np.where(y_test.numpy() == 1)[0][1]
```

The LIME explainer created earlier was using the default value which is `0.75 * sqrt(number of features)`. This evaluates to a kernel width of 4 since the number of features in the dataset is 30. We will also create another LIME explainer that is initialized with a smaller kernel width of 1 to see the impact on the interpretation. The code below shows how to create a LIME explainer with kernel width = 1.

```
explainer_kw1 = lime.lime_tabular.LimeTabularExplainer(X_train.numpy(),
feature_names=data.feature_names,
                    class_names=data.target_names,
                  kernel_width=1, #A
                    discretize_continuous=True)
```

#A: kernel_width parameter set to 1

The resulting LIME interpretations using the default kernel width and kernel width of 1 for the second benign case are shown in Figure 4.18 (a) and Figure 4.18 (b) respectively.

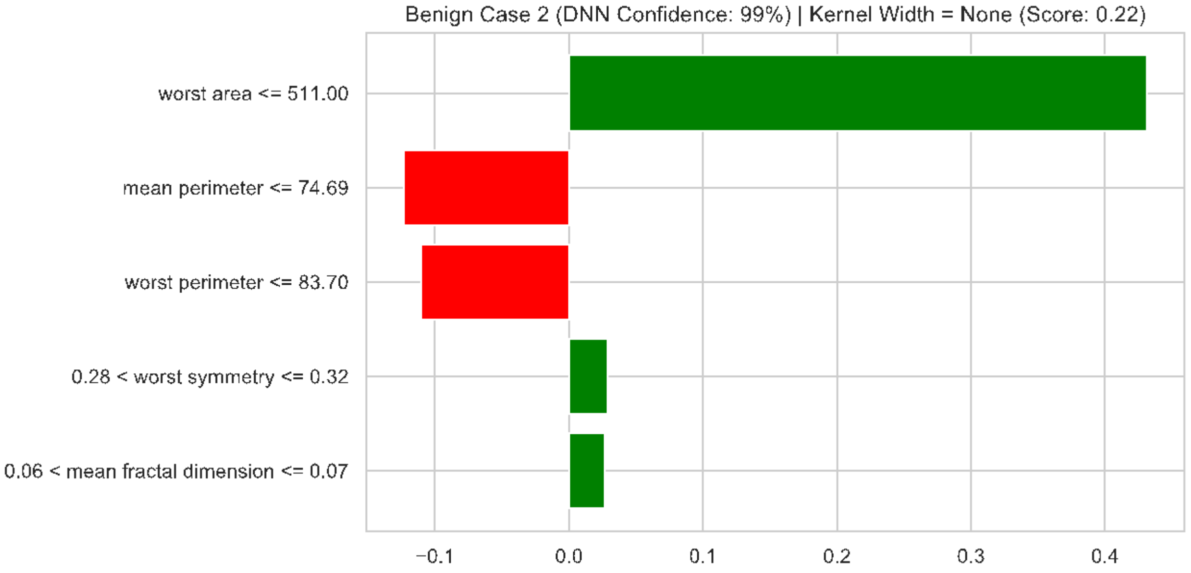


Figure 4.18 (a): LIME Interpretation of Benign Case 2 with Default Kernel Width

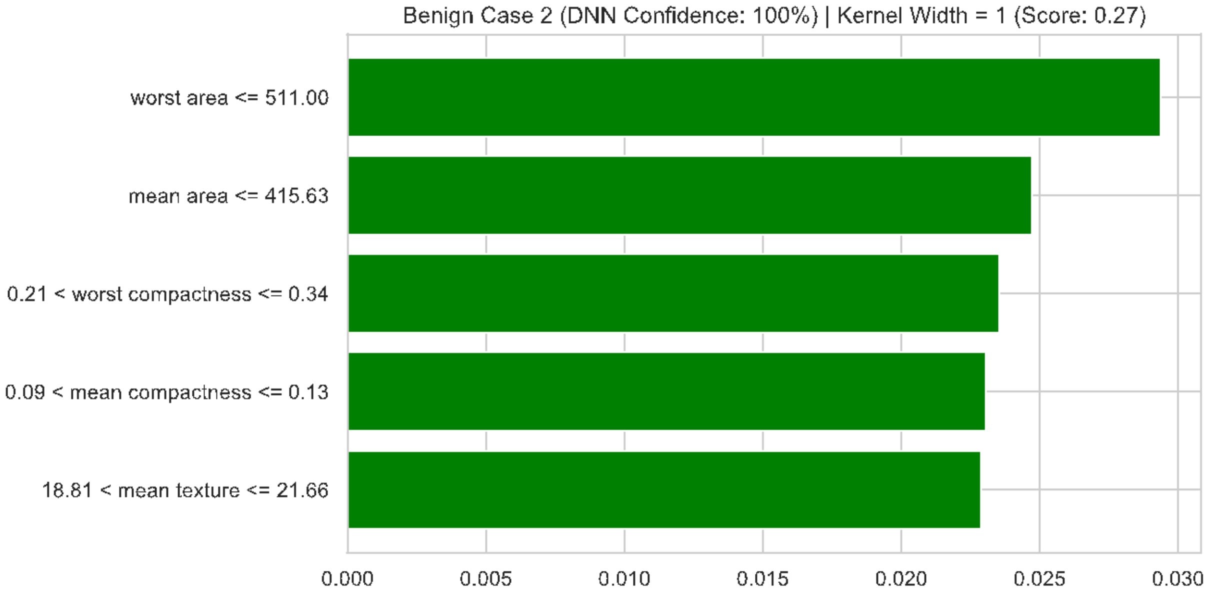


Figure 4.18 (b): LIME Interpretation of Benign Case 2 with Kernel Width=1

Let’s first compare the default LIME interpretation of the second benign case with the first case done earlier. The first most importance feature is the same. We can however see that the range of values for the features is different. For the second benign case, we see that the model predicted benign because the worst cell area was less than 511, as opposed to being in between 511 and 683.95 as seen for the first case. This is still a valid prediction as there are a lot more cases that are benign when the worst area is less than 511. The fidelity score is also higher for default LIME interpretation of the second benign case. This means that the linear model in LIME reflects the DNN model a lot more closely for this case than the first one.

If we now switch to Figure 4.18 (b), we can see how different the interpretation is if we use a smaller kernel width. The top-most feature is still the same, but we see different features and a much smaller range of values for them. This is because a small kernel width focuses the linear surrogate model on perturbed cases that are very close to the picked instance. Which kernel width is better for the second benign case? We can see that a kernel width of 1 achieves a fidelity score of only 0.27 as opposed to 0.22 for the default. Therefore, kernel width of 1 is better in this case. As an exercise, I highly encourage you to increase the kernel width for the second case to see if you can achieve a higher fidelity score and to analyze the resulting LIME plot. I would also highly encourage you to tune the kernel width hyperparameter for the first case to see if you can get a better interpretation that is much more faithful to the DNN.

Figure 4.19 (a) and Figure 4.19 (b) show the LIME interpretations for the second malignant case for two kernel widths – one default and the other with width 1. As an exercise, I encourage you to compare these interpretations with the first malignant case and see which kernel width gives you a higher quality interpretation.

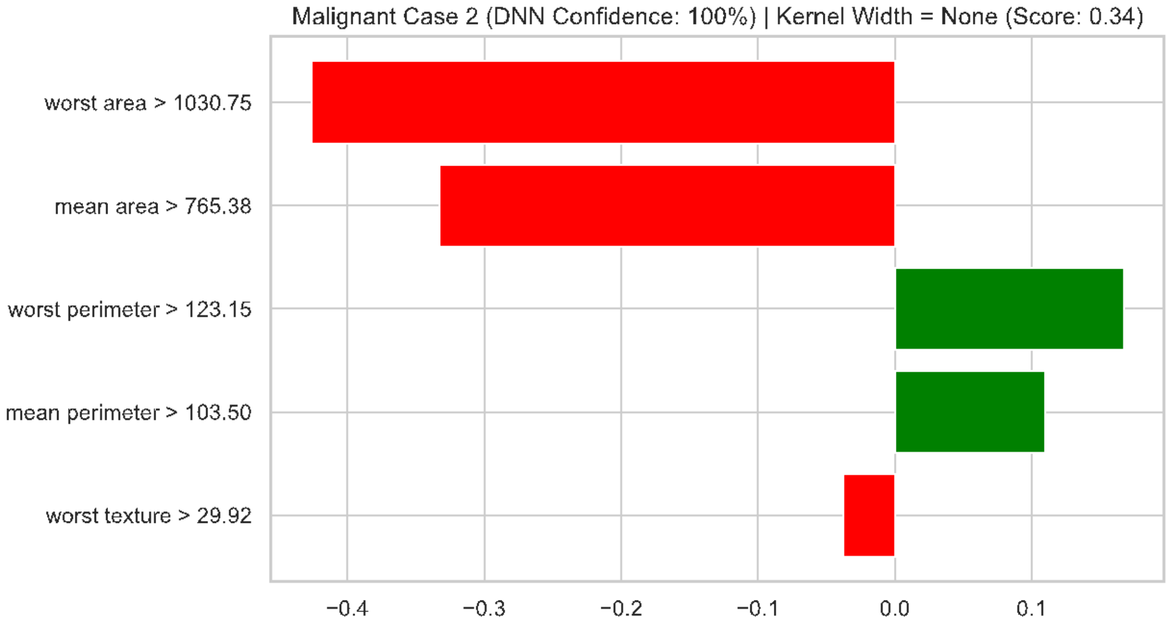


Figure 4.19 (a): LIME Interpretation of Malignant Case 2 with Default Kernel Width

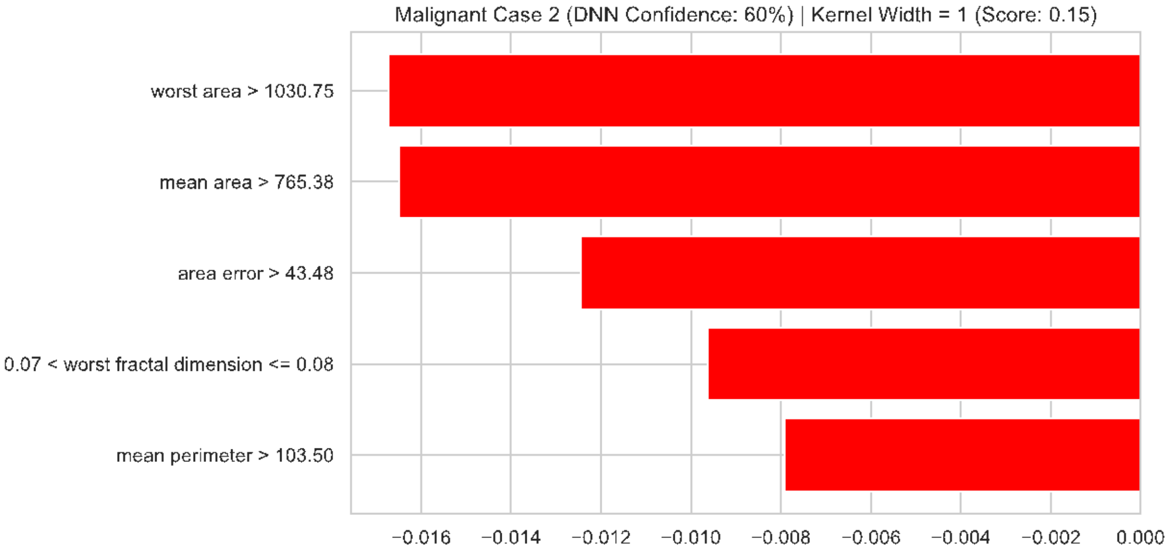


Figure 4.19 (b): LIME Interpretation of Malignant Case 2 with Kernel Width=1

LIME is a great tool to interpret black-box models. It is model agnostic and can work with different types of models. LIME can also work with different types of data – tabular data, images and text. We have seen it in action using tabular data in this section. We will explore images and text data in later chapters, and you can find examples in the library documentation here. It is a widely used library with lots of active contributors.

The quality of the LIME interpretation, however, depends greatly on the choice of the kernel width which is an input to the kernel function that is used to weight the perturbed samples. It is a very important hyperparameter and we have seen that the width could be different for different examples that we pick to interpret. We can use the fidelity score provided by the library to determine the right width, but the selection of the right kernel width is still ambiguous. Another limitation of LIME is that the perturbed dataset is created by sampling from a Gaussian distribution and it ignores correlations between features. The perturbed dataset may therefore not have the same characteristics as the original training data.

4.6 SHAP

SHAP is an acronym that stands for Shapley Additive exPlanations and it was proposed in 2017 by Scott M. Lundberg and Su-In Lee. It unifies the idea behind LIME (and linear surrogate models) and game theory and provides more mathematical guarantees on the accuracy of the explanations than LIME. A **Shapley value** is a **game-theoretic** concept that quantifies the impact of a **coalition** of players in a **cooperative game**. Without going too deep into game theory, we can easily draw parallels with model interpretability. Shapley values could be used to quantify the impact of features (à la players) and their interactions (à la player coalitions) on a model prediction (à la cooperative game). Let’s breakdown the SHAP interpretability technique by looking at Figure 4.20.

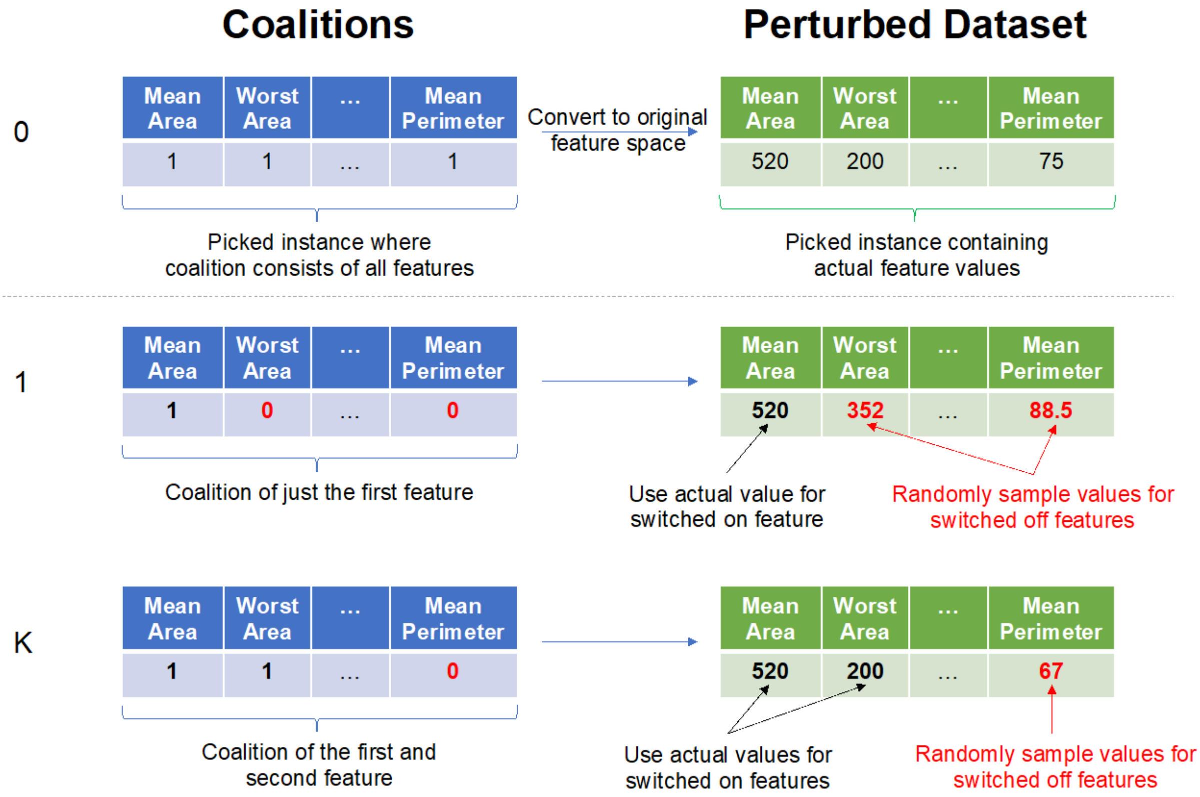

Coalitions
Perturbed Dataset
0
Mean Area
Worst Area
…
Mean Perimeter
1
1
…
1
Convert to original feature space
Mean Area
Worst Area
…
Mean Perimeter
520
200
…
75
Picked instance where coalition consists of all features
Picked instance containing actual feature values
1
Mean Area
Worst Area
…
Mean Perimeter
1
0
…
0
Mean Area
Worst Area
…
Mean Perimeter
520
352
…
88.5
Coalition of just the first feature
Use actual value for switched on feature
Randomly sample values for switched off features
K
Mean Area
Worst Area
…
Mean Perimeter
1
1
…
0
Mean Area
Worst Area
…
Mean Perimeter
520
200
…
67
Coalition of the first and second feature
Use actual values for switched on features
Randomly sample values for switched off features

Figure 4.20: Illustration of creating the perturbed dataset for SHAP

The idea behind SHAP is quite similar to LIME. The first step is to pick an instance to explain. In Figure 4.20, the picked instance is shown as the first row in index 0. Since SHAP uses game-theoretic concepts, the picked instance consists of a coalition of all the features. When all the features are selected or "switched on", it is represented by a vector containing all 1s for all the features in the dataset. The first column in Figure 4.20 shows the **coalition vector** as a table. For the picked instance, since the coalition vector consists of all 1s, we pick all the actual feature values for that instance when we convert that vector into the feature space. This **feature vector** is shown as a table in the second column in Figure 4.20.

Once you've picked the instance to interpret, the next step is to create the perturbed dataset. This is the same as LIME but unlike LIME, the idea in SHAP is to generate a bunch of coalition vectors where features are randomly "switched on" or "switched off". If a feature is switched on, then its value in the coalition vector is 1. If the feature is switched off, then its value in the coalition vector is 0. We know how to represent the feature in the feature space when it is switched on – we just have to pick the actual value from the instance that we've picked to interpret. If, however the feature is switched off, then we pick a value randomly from the training set for that feature. These are shown in red in both the coalition vector and the feature vector in Figure 4.20.

After creating the perturbed dataset, the next step is to the weight the dataset based on its proximity to the picked instance. This is again similar to LIME but unlike LIME, SHAP uses the **SHAP kernel** to determine the weights for the samples in the perturbed dataset as opposed to the exponential kernel function. The SHAP kernel function gives higher weight to coalitions that consist of very low or very high number of features. The next steps are then the same as LIME, which is to fit a linear model on the weighted dataset and then return the coefficients or weights of the linear model as the interpretation for that picked instance. These coefficients or weights are called Shapley values.

Let's now see SHAP in action on the breast cancer diagnostics model trained earlier. The authors of SHAP have created a Python library in Github. We can install this library using pip as follows.

```
pip install shap
```

We will be using the same helper function called `prob` (introduced in the previous section on LIME) to provide the DNN model predictions for the perturbed dataset. You can now create the perturbed dataset and initialize the SHAP explainer as follows.

```
import shap
shap.initjs() #A

shap_explainer = shap.KernelExplainer(prob, #B
                                      X_train.numpy(),
                                      link="logit") #C
```

#A: Initialize javascript for interactive visualizations
#B: Use the prob helper function to obtain the DNN predictions
#C: Use the logit link function since the DNN is a classifier

Note that the logit link function is used for the linear surrogate model since we are dealing with a binary classifier that outputs a probability estimate for the positive class. For regression problems, you can switch the `link` parameter to `identity`. Next, obtain the SHAP values for all the data in the test set as follows.

```
shap_values = shap_explainer.shap_values(X_test.numpy())
```

You can now obtain the SHAP interpretation for the first benign case as a matplotlib plot as follows.

```
plot = shap.force_plot(shap_explainer.expected_value[0],
                    shap_values[0][benign_idx,:],
                    X_test.numpy()[benign_idx,:],
                    feature_names=data['feature_names'],
                    link="logit")
```

The resulting plot is shown in Figure 4.21. Recall that for the first benign case, the DNN model predicted it was benign with a probability of 0.99 or confidence of 99%.

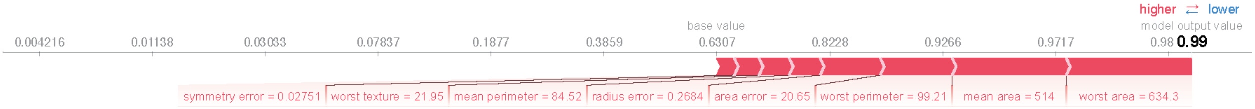


Figure 4.21: SHAP Interpretation of Benign Case 1 where DNN Model predicts Benign with Probability 0.99 (or Confidence of 99%)

The SHAP library provides much nicer visualizations where you can see how each feature value pushes the base prediction up or down. In Figure 4.21, you can the base value at around 0.63. This is the positive class rate representing the proportion of benign cases. When we explored the data in Section 4.2, we observed that in the dataset roughly 63% of the cases were benign. The idea behind the SHAP visualization is to see how the feature value pushes the baseline prediction probability from 0.63 up to 0.99. The positive Shapley values are shown as red and negative Shapley values are shown as blue. The impact of the feature is shown by the length of the bar. We can see from the figure that the worst cell area and mean cell area features have the largest Shapley values and it pushes the base prediction the most. The next most important feature is worst cell perimeter.

Figure 4.22 shows the SHAP interpretation for the second benign case where the DNN model predicted benign with probability 0.99.

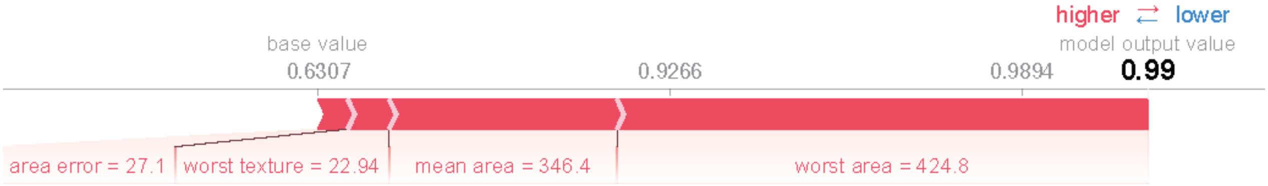


Figure 4.22: SHAP Interpretation of Benign Case 2 where DNN Model predicts Benign with Probability 0.99 (or Confidence of 99%)

We can see that the two most important features here are the worst cell area and the mean cell area. Since the worst area and mean area are quite low, with values of 424.8 and 346.4 respectively, it was enough to push the baseline prediction all the way to 0.99. As an exercise, please modify the code shown earlier to interpret the two malignant cases. The resulting plots are shown in Figures 4.23 and 4.24.

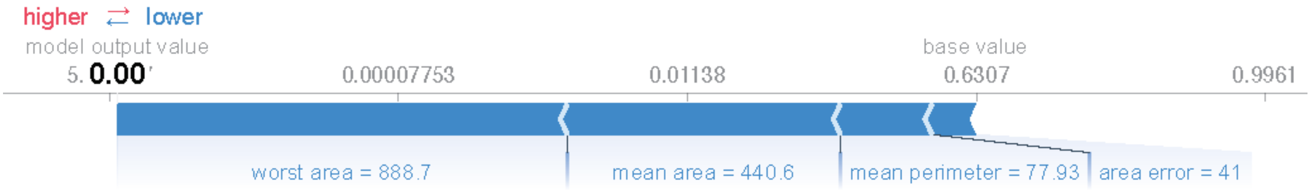


Figure 4.23: SHAP Interpretation of Malignant Case 1 where DNN Model predicts Benign with Probability 0 (or Malignant with Confidence of 100%)

For the first malignant case, the model predicted that it was benign with a probability of 0. In Figure 4.23, we can see how the feature values push the baseline prediction probability down to 0. It looks like the features that have the most influence on the final prediction are the worst cell area, mean cell area and perimeter.

For the second malignant case, the model also predicted that it was benign with a probability of 0. We can see that the most influential feature is again the worst cell area. Since the value was quite large, greater than 1417, it was enough to push the baseline prediction probability down to 0.

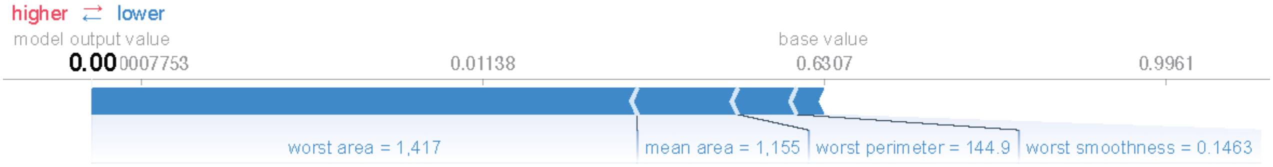


Figure 4.24: SHAP Interpretation of Malignant Case 2 where DNN Model predicts Benign with Probability 0 (or Malignant with Confidence of 100%)

SHAP is another great tool to interpret black-box models. Like LIME, it is model agnostic, and it uses concepts from game theory to quantify the impact of features on the model prediction of a single instance. It provides more mathematical guarantees on the accuracy of the explanations than LIME. The library also provides great visualizations of the impact of features showing how the feature values push the baseline prediction up or down to the final prediction. Computing the Shapley values based on the SHAP kernel is however computationally intensive. The computational complexity increases exponentially with the number of input features.

4.7 Anchors

Anchors is another model agnostic interpretability technique that is local in scope. It was proposed in 2018 by the same creators of LIME. It improves on LIME by providing high precision rules or predicates for how the model arrived at the prediction and also by quantifying the coverage of these rules in terms of global scope. Let's break this down.

In this technique, model interpretations are generated in the form of anchors. An **anchor** is essentially a set of **if-conditions** or **predicates** that contains the picked instance that we would like to interpret. This is shown by the solid violet box in Figure 4.25. The anchor illustrated in the figure can be interpreted as two if-conditions where the two features in the 2D feature space is bounded by a lower bound and upper bound, thereby forming a bounding box around the picked instance. The first objective of the algorithm is to form high precision anchors that contain the picked instance in terms of the target prediction. The **precision** is a measure of the quality of the anchor and is defined as the ratio of the number of perturbed samples with the same target prediction as that of the picked instance to the total number of samples within the anchor. An important hyperparameter for the algorithm is the **precision threshold**.

Once the algorithm has come up with a set of high precision anchors, the next step is to quantify the scope of each anchor. The scope of an anchor is quantified by a metric called **coverage**. The coverage metric measures the probability that the anchor (or the set of predicates) will be present in other samples or other parts of the feature space. With this metric, we will be able to tell how applicable the anchors interpretation is at a global scale. The objective of the algorithm is to pick the anchor with the highest coverage.

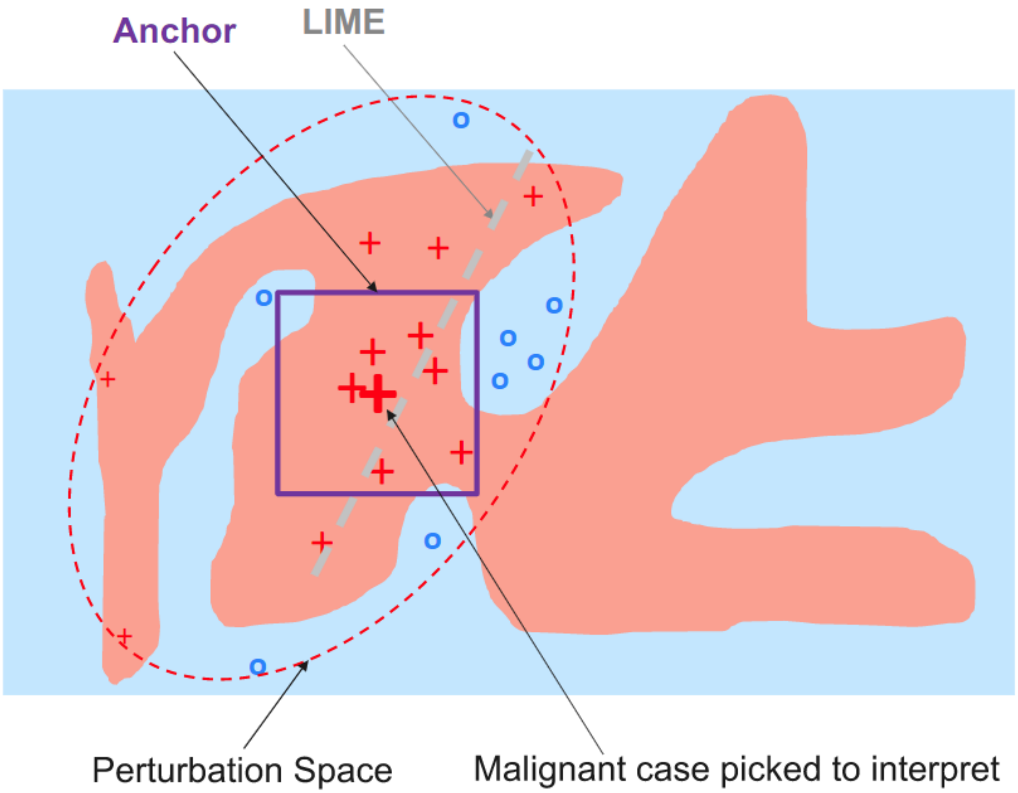


Figure 4.25: Illustration of an anchor

Determining all possible predicates that meet the precision threshold and the coverage requirement is a computationally intensive task. The authors of the algorithm have used a bottom up approach in constructing the predicates or rules. The algorithm starts off with an empty set of rules and in each iteration, the algorithm incrementally constructs an anchor that meets the precision threshold and the coverage requirement and adds it to the set. To estimate the precision of a anchor, the authors have formulated this problem as a multi-armed bandit problem and specifically used the KL-LUCB algorithm to identify the rules with the highest precision.

Let's now interpret the breast cancer DNN model using anchors. The authors of the paper have created a library in Python that can be found in Github. You can install the library using pip as follows.

```
pip install anchors_exp
```

As we had done with LIME and SHAP, let’s now create the anchors tabular explainer for the breast cancer dataset.

```
from anchor import anchor_tabular #A

anchor_explainer = anchor_tabular.AnchorTabularExplainer(
    data.target_names, #B
    data.feature_names, #C
    X_train.numpy(),
    categorical_names={}) #D
anchor_explainer.fit(X_train.numpy(), #E
                     y_train.numpy(), #E
                     X_val.numpy(), #E
                     y_val.numpy()) #E
```

#A: Import the anchor_tabular module from the library
#B: Set the target label names
#C: Set the feature names for the dataset
#D: Provide categorical feature names if any
#E: Fit the anchors explainer on the train and val sets

We will need to create a different helper function for anchors that provides the DNN predictions as discrete labels rather than probabilities. This helper function is shown below.

```
def pred(data):
    pred = model.forward( #A
        Variable(torch.from_numpy(data)).float()).\ #A
    detach().numpy().reshape(-1) > 0.5 #A
    return np.array([1 if p == True else 0 for p in pred]) #A
```

#A: Predict 1 if output probability is greater than 0.5, else 0

Let’s now interpret the first benign case using Anchors. The code below shows how to interpret the instance, extract the predicates or rules and also obtain the precision and coverage of the interpretation.

```
exp = anchor_explainer.explain_instance(X_test.numpy()[benign_idx], #A
                                        pred, #B
                                        threshold=0.95) #C
print('Prediction: ', anchor_explainer.class_names[pred(X_test.numpy()[benign_idx])][0]) #D
print('Anchor: %s' % (' AND '.join(exp.names()))) #E
print('Precision: %.3f' % exp.precision()) #F
print('Coverage: %.3f' % exp.coverage()) #G
```

#A: Pass the picked instance as the first parameter
#B: Provide the helper function that provides the model label predictions
#C: Set the precision threshold

#D: Print the label prediction made by the model
#E: Print the rules or predicates
#F: Print the precision of the anchor
#G: Print the coverage of the anchor

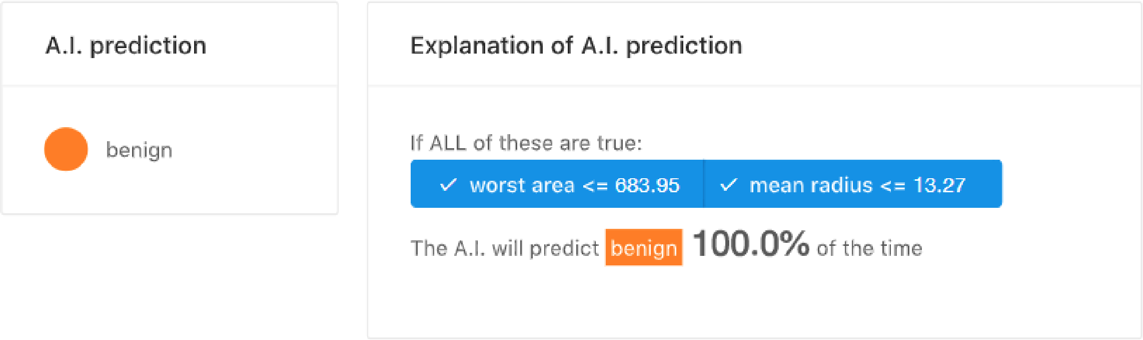
Note that the precision threshold was set to 0.95. The rules or predicates are obtained as a list of strings and they are strung together using the AND clause. The resulting output from the code is shown below.

```
Prediction:  benign
Anchor: worst area <= 683.95 AND mean radius <= 13.27
Precision: 1.000
Coverage: 0.443
```

You can see that the model predicted benign correctly and the interpretation or anchor with the highest precision consists of two rules or predicates. If the worst area is less than or equal to 683.95 and the mean radius is less than or equal to 13.27, then the model predicts benign 100% of the time in the region around the picked instance. In terms of coverage, this anchor does pretty well with a coverage of 44.3%. This means that the rule is applicable to quite a lot of benign cases globally. You can also obtain an HTML visualization of this interpretation using the following line of code.

```
exp.save_to_file('anchors_benign_case1_interpretation.html')
```

The resulting visualization is shown in Figure 4.26. The Anchors library as it stands now does not provide matplotlib visualizations.



A.I. prediction

● benign

Explanation of A.I. prediction

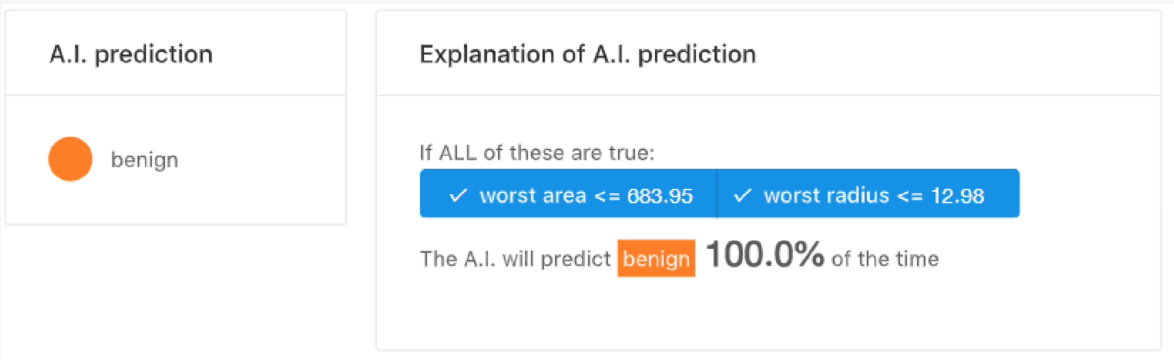
If ALL of these are true:

✓ worst area <= 683.95 ✓ mean radius <= 13.27

The A.I. will predict benign 100.0% of the time

Figure 4.26: Anchor Interpretation of Benign Case 1 where Precision is 100% and Coverage is 44.3%

As an exercise, please extend the code above to the other benign and malignant cases. The resulting visualization for the second benign case is shown in Figure 4.27. You can see that the model predicted benign correctly and the anchors algorithm came up with two rules with precision 1 which is – if the worst cell area is less than or equal to 683.95 and the worst cell radius is less than or equal to 12.98, then the model predicts benign 100% of the time. The coverage of this anchor is however 20.9% lower than the first benign case. This means that the interpretation for the second benign case is a lot more local than the first one.



A.I. prediction

● benign

Explanation of A.I. prediction

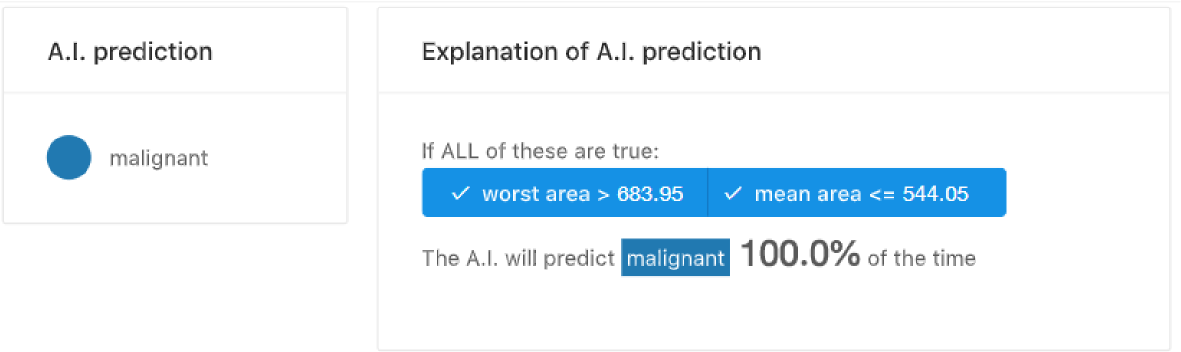
If ALL of these are true:

✓ worst area <= 683.95 ✓ worst radius <= 12.98

The A.I. will predict benign 100.0% of the time

Figure 4.27: Anchor Interpretation of Benign Case 2 where Precision is 100% and Coverage is 20.9%

The anchors interpretation for the first malignant case is shown in Figure 4.28. The model correctly predicted it was malignant and the interpretation consists of two rules or predicates with a precision of 1. The rules are – if the worst cell area is greater than 683.95 and the mean cell radius is less than or equal to 544.05, then the model predicts malignant 100% of the time. The anchor however has a very low coverage of just 1.2%. The interpretation is therefore extremely local and is not really applicable to a lot of the other malignant cases.



A.I. prediction

● malignant

Explanation of A.I. prediction

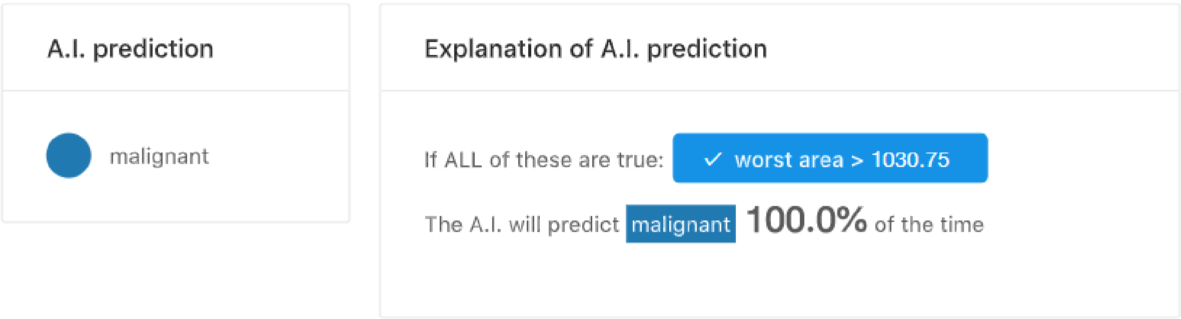
If ALL of these are true:

✓ worst area > 683.95 ✓ mean area <= 544.05

The A.I. will predict malignant 100.0% of the time

Figure 4.28: Anchor Interpretation of Malignant Case 1 where Precision is 100% and Coverage is 1.1%

Finally, the anchors interpretation of the second malignant case is shown in Figure 4.29. The model again predicted malignant correctly and the interpretation consists of one rule with a precision of 1. The rule is – if the worst cell area is greater than 1030.75, then the model predicts malignant 100% of the time. The coverage of this anchor is a lot better than the first case at 27.1%. This makes sense as if we go back to the exploratory analysis we did in Section 4.2 and look closely at Figure 4.3, we see a lot more malignant cases with worst cell area greater than 1030.



A.I. prediction

● malignant

Explanation of A.I. prediction

If ALL of these are true: ✓ worst area > 1030.75

The A.I. will predict malignant 100.0% of the time

Figure 4.29: Anchor Interpretation of Malignant Case 2 where Precision is 100% and Coverage is 27.1%

Anchors is a powerful model agnostic interpretability technique as it provides interpretations as a set of high precision rules or predicates or human readable if-conditions. The technique also gives us a sense of the coverage or scope of the rules, i.e. how applicable the rules are at a global scale. The Python library however is still a work in progress and is not as actively developed as LIME or SHAP.

4.8 Summary

- An Artificial Neural Network (ANN) is a system that is designed to loosely model a biological brain. It belongs to a broad class of machine learning methods called deep learning. The central idea of deep learning based on ANNs is to build complex concepts or representations from simpler concepts or features.
- An ANN with two or more hidden layers is called a Deep Neural Network (DNN).
- An efficient algorithm to determine the weights in a DNN is backpropagation.
- The activation function is a very important feature within a neural network. It decides whether a neuron should be activated and by how much. The properties of an activation function are that it is differentiable and monotonic.
- ReLUs are the most widely used activation functions in neural networks as it handles the vanishing gradient problem well. It is also more computationally efficient.
- There are multiple ways of interpreting neural networks. We can use model agnostic methods that are global in scope such as PDPs. In this chapter, we learned about more

advanced perturbation-based model agnostic techniques such as LIME, SHAP and Anchors. These interpretability techniques are local in scope, i.e. they focus on a specific instance or example to interpret.

- LIME stands for Local Interpretable Model-agnostic Explanations. It is based on picking an example, randomly perturbing it, weighting the perturbed samples based on its proximity to the picked instance and lastly fitting a simpler white-box model on the weighted samples.
- The quality of the LIME interpretation depends greatly on the choice of the kernel width which is an input to the kernel function that is used to weight the perturbed samples. It is a very important hyperparameter and we have seen that the width could be different for different examples that we pick to interpret. We can use the fidelity score provided by the library to determine the right width, but the selection of the right kernel width is still ambiguous.
- Another drawback of LIME is that the perturbed dataset is created by sampling from a Gaussian distribution and it ignores correlations between features. The perturbed dataset may therefore not have the same characteristics as the original training data.
- SHAP stands for Shapley Additive exPlanations. Like LIME, it is model agnostic, and it uses concepts from game theory to quantify the impact of features on the model prediction of a single instance. In theory, SHAP provides more mathematical guarantees on the accuracy of the explanations than LIME.
- The SHAP library provides great visualizations of the impact of features showing how the feature values push the baseline prediction up or down to the final prediction.
- Computing the Shapley values based on the SHAP kernel is however computationally intensive. The computational complexity increases exponentially with the number of input features.
- Anchors is another technique that improves on LIME by providing interpretations as a set of high precision rules or predicates or human readable if-conditions. The technique also gives us a sense of the coverage or scope of the rules, i.e. how applicable the rules are at a global scale. The Python library however is still a work in progress and is not as actively developed as LIME or SHAP.

In the next and subsequent chapters, we will go deeper into the world of neural networks and learn about more complex structures like CNNs and RNNs. We will also learn how to perform feature attributions on neural networks and also how to dissect them to get a much better understanding of what the network has learned.