Exploring Approaches to Machine Learning

Module Overview

Hi and welcome to this course on designing a machine learning model. We'll start this course off by first exploring approaches to machine learning, going back to rule-based systems. In order to keep things interesting and work with something that you can identify with, we'll talk about data-driven decisions and actions using a case study of sentiment analysis. Our sentiment analysis case study will lead us to a discussion of rule-based approaches to learning. We'll understand the strengths and weaknesses of rule-based approaches and then move on to machine learning approaches. We'll see how the major strength of machine learning is in algorithms that can learn from your data so your algorithms can dynamically change their predictions when they are fed in new data or information. In this context we'll introduce traditional machine learning models; their featured selection is performed by experts, and we'll see how deep learning is a far more complex model, which can extract features from unstructured data without having any experts for feature selection. We'll round off this model with a discussion of the differences between traditional machine learning and deep learning, and we'll also discuss and understand a few traditional ML algorithms.

Prerequisites and Course Outline

Before we dive into the course content and the hands-on demos of this course, let's take a look at some of the pre-reqs that you need to have to make the most of your learning. This course assumes that you're comfortable programming in the Python language. Two of the modules of this course will cover demos written using Python 3 and Jupyter Notebooks. This course also assumes that you have some familiarity with machine learning models, you've built and trained ML models in the past and you have a good conceptual basis. Now this is not really a required prereq, but it would help to have some understanding of basic math; mean, standard deviation, and so on. If you feel that you lack some of the ML prereqs needed for this course, here are some other courses that you can watch before this one. Here are courses that will serve as introductory ML courses: Understanding Machine Learning with Python, How to Think About Machine Learning Algorithms, and Building Your First scikit-learn Solution. This course covers a lot of breadth and gives you a big-picture understanding of many concepts in ML. We'll start off by discussing different approaches to machine learning, starting off with rule-based systems. We'll then discuss how you would choose the right machine learning problem by focusing on different use cases and the kinds of models that you'll use. We'll then move on to choosing the right machine learning solution. We'll discuss the characteristics of the different ML models that you might choose to build. We'll then get hands-on and see how we can build simple ML solutions using Python. We'll move back to concepts and then speak about designing machine learning workflows. We'll discuss the significance of neural networks, ensemble learning, and other techniques as a part of this. And finally, we'll get hands-on once again and see how we can build ensemble learning and neural network solutions.

A Case Study: Sentiment Analysis

Before we explore the different approaches to machine learning, let's do a little case study of sentiment analysis. We'll talk about how the world is today, how we need to work with data, and where ML fits in. Here is a way to think about people's behavior on the internet over the last 30 years or so. For the first 10 years, the way they worked was surf and browse. Websites were static; if they were linked to one another, people would just click around and explore and see what's out there. Then from the year 2000 until about 2008, you can think of the behavior as search-find-obtain. This is when search engines like Google were really powerful. People knew what they were looking for, they would use a search engine, find what they need, and explore. From the year 2000 onwards until about now, we would say that the paradigm is share and discover. People automatically find things that they're interested in, will look for new things, and share it with others. Everyone in the world today is always online, and they have a network, not just one network, several different networks, friends, professional, family. They discover interesting details through their network; they have a number of online opinions, which they express on different pages of social media. These opinions which people share with the world are expressed on different channels online, and they take on a variety of different forms. They could be reviews, tweets and posts, messages sent to others, maybe swipes on a screen. All of these opinions are important; they contain information for businesses or for anyone else. If a business or an organization is interested in data today, here is where they need to look. A data analyst's job is to collect these opinions, extract information from them, and act on that information. How does this world of share and discover play out for a data analyst? In order to collect opinions, she might have to scrape/harvest comments, articles, and tweets. Once the information is available to her, she'll have to preprocess and extract information from this somehow. Now this extracting of information could be any kind of information. In our case study here, our data analyst is looking for product sentiment. She might perform sentiment analysis, and once she has this information, the company or the organization will act on this information. They might buy or sell stocks, target advertising spends a different way, make any number of decisions. The source of opinions today are varied and plenty, and depending on who you are and what you're looking for, you might look to a different source. Researchers may use public datasets, companies will have their own proprietary data, which they are constantly collecting. Scrapers will use media signals by scraping from social media. All of this is big, unstructured data. It's big because the quantity and number of sources are huge and diverse. It's unstructured because it could be text data, log messages, signals, which have no real structure. Once this data has been collected and preprocessed and cleaned, you can extract information from this data. With sentiment analysis, you might end up tagging all of the data items with value for sentiments; maybe this is a product, for a service. Once sentiment tags are available, you have categorical data about these opinions. The analysts will then analyze this categorical data in order to extract information. Sentiment data is categorical because you classify it as say, positive, negative, or a neutral, or you could have highly positive, highly negative, additional categories as well. There are really infinite kinds of analysis that you could perform on this data. Maybe you'd perform logistic regression to find the relationship between variables, or you can perform quadrant analysis where you find clusters of data with similar characteristics, and these clusters can then be used to target your ads or your spends. Once everything is categorized and analyzed, it's time for the organization to act on this data, make data-driven decisions. Maybe the company uses this information to trade on financial markets. Maybe budgets need to be reallocated and add spends need to be redirected based on sentiment. Opinions can also be used to tailor electoral strategy, and people have done so with a lot of success. You can even use opinions to decide product recall strategies. Here is a simple case study showing us how we could use opinions and online behavior to drive the actions of an organization or a company. Before we end, let's talk about how this sentiment analysis can be used in say, event-driven trading. If you're a company or a funder trading in the stock markets, you might choose to use sentiment analysis to determine your trading decisions. Now you're looking and waiting for some company earnings to come out, the forecast for these earnings are already available to you, and you are tracking whether this company exceeded forecast or missed its forecast, and that's what you're tracking along the rows of this 2x2 grid. You also have available to you, thanks to all of the data collection and analysis that you've done, the analyst sentiment before earnings are released. Now this analyst sentiment could have been positive or negative. These form the columns of our 2x2 grid. Now once the company releases its results, you'll have some information. Is it better or worse than analysts' expectations? And as a financial trader in this market, you'll be faced with a decision: do you buy or sell shares in this company? Here is a way where you could use the sentiment analysis information that you have available to determine how to act. If the analyst sentiment before earnings was very negative, but the company exceeded forecasts, that is their earnings were really good, you might choose to buy. The shares clearly beat expectations. Now at the other end, if the analyst sentiment before earnings release was very positive, but the company missed its forecast, you might choose to sell its shares. These two quadrants are fairly clear. Let's talk about when the analyst sentiment was positive and the company exceeded a forecast. That is, it did really well. Now this time, surprisingly enough, you might choose to sell, and I'll talk about why in just a bit. And finally, if analyst sentiment before earnings was negative and the company missed its forecast, you might choose to buy. Why would you do this? Well, the insight that you obtain when you work with sentiment data and actual earnings is buy the rumor, sell the news. Buy the rumor essentially says if market sentiment was negative, buy stock, even if the earnings are poor. Sell the news says if the market sentiment was positive, sell the stock even if earnings are great. This is of course, our case study; I wouldn't recommend that you take stock tips from me.

Sentiment Analysis as a Binary Classification Problem

Let's continue with our discussion of sentiment analysis, that is our case study, and talk about polarity detection. Now sentiment analysis systems can be built to assess many details of opinions or sentiment. It could choose to assess the polarity, is the sentiment positive or negative? You could build sentiment analysis systems to detect subjectivity that exists in the opinions. Is an article opinion or fact? Is it subjective or objective? You could also have sentiment analysis systems to explore the aspects of text. Do I just look at a part of a text, or do I look at the entire text? As you can see, opinions, whether expressed online or in the real world are very complex, they are very nuanced, and there are different ways to analyze these opinions, and analyzing these opinions can be a really hard problem, but sentiment analysis need not be hard if we set up the problem right, if we simplify the problem so we can solve it. Now it's been found that human brains are very, very efficient at making binary decisions. In fact, researchers have seen that even if there are three options available, humans tend to make bad decisions, but with just two options where they have to make binary decisions, they work well. Binary decisions can be of many types, hot or not, buy or sell, fight or flight, for or against, good or bad? Complex opinions can be simplified and made tractable if you model your sentiment analysis as a binary classification problem and you'll find that in the real world this is what we tend to do. The nuances of the individual opinions might be lost, but you'll get important information in the aggregate, and you'll get information that you can act on. So just go with positive or not positive. Once we've expressed sentiment in this way, things get more tractable and this problem can be solved. Binary classification is a well-studied, well-understood problem. In fact, binary classification problems have many applications, not just sentiment. Is a comment positive or negative? Well, that's sentiment analysis, but is it email spam or ham? That is a classic problem in binary classification. Are transactions fraudulent or legitimate? Another binary classification problem.

Rule Based vs. ML Based Analysis

Now that we know that our sentiment analysis is essentially just a binary classification problem, we can now talk about the kind of classifier that you might choose to build, whether it's a rule-based classifier or a machine learning-based classifier. If you zoom out a bit, what is this binary classifier what you want it to do? You want this classifier to take in a problem instance and give it a label. The binary classifier is just a function that does this. It looks at a problem instance and assigns it a label. Now you could assign a label based on rules and what you'd get is a rule-based classifier. The rules are drawn up by experts and these rules are used to assign a label to a problem instance. As a programmer, you can imagine these rules to be a bunch of if/else statements. If this is true, assign this label; if that is true, assign another label, and so on, or if your classifier is machine learning based or ML based, the label is assigned based on patterns displayed in aggregate data. The machine learning model has hopefully looked at lot of data. It understands patterns that exist and it applies these patterns to this problem instance to assign the label. We'll first talk about a rule-based classifier to assign a label to a problem instance. How did this rule-based classifier come to be? You have a corpus of data and you have expert analysts who would've studied this data. These are human experts who would have looked at this corpus and the pattern, and the information available in this corpus is their brain; this is what makes them experts. These are the experts who would then draw up the rules, which would be used to classify your problem instance. Whatever the rule-based classifier knows, and the way it approaches the problem has been drawn up and encoded by human experts. Another approach is to use a machine learning-based classifier. How does this work? It takes a problem instance and assigns it a label. This step remains the same; however, the machine learning classifier does not depend on human experts. We have a huge corpus of data and this is then fed into a classification algorithm and once this classification algorithm has identified the patterns in the data, you get an ML-based classifier. A machine learning-based classifier is essentially trained on the corpus of data itself. There are no human experts present. Now that we've understood this difference, let's compare and contrast machine learning-based classifiers and rule-based classifiers. The most significant difference and the reason why machine learning is so popular today is that ML-based classifiers are dynamic. The output or the prediction of the classifier will change based on changing patterns that it has encountered in the corpus of data. Rule-based classifiers are static. Rules are applied independent of the data being analyzed, even if the data changes, rules will not change without human intervention. With an ML-based classifier, there is really no expert intervention needed. There is far less need for expert skill, well, an expert has set up the machine learning-algorithm, and you could say that an expert needs to train the algorithm, but you don't need experts to formulate the rules. Whereas with the rule-based classifiers, experts are vitally important. These human experts have to constantly monitor the world and update the rules proactively if the classifier needs to stay relevant. With the ML-based classifier, in order to update the classifier to change its predictions based on new data, you simply update the corpus of data on which the classifier has been trained. With the rule-based classification, you'll need to update the rules. You'll need to re-code the model that you're using. Updating rule-based systems, as you can see, is much more heavyweight. The differences that I've highlighted here are the strengths of machine learning models and the weaknesses of rule-based models. Let's continue our discussion and we'll look at a few instances where rule-based classification may have an edge. You can't really build your ML-based classifier out of thin air; you need a large corpus. This has to be high quality; it has to be cleaned and preprocessed correctly to get a good ML model, whereas when you're working with rule-based systems, you don't really need a corpus. All of the patterns and information needed to formulate the rules is available with the experts. Now ML-based systems cannot operate on a single problem instance. Really, ML-based classifiers only work if you have a large corpus, which has already been tagged or labeled correctly. This is the training data that the ML-based classifier needs to understand patterns. You can't just feed it an isolated instance and look for a decision or a prediction. Whereas with rule-based classifiers, they can operate on the isolated problem instances. All of the magic is in the rules. Machine learning-based classifiers or really any machine-learning model has an explicit training step. This is the step where the huge corpus of data is fed in and the machine learning algorithm trains on that corpus to identify patterns. With the rule-based classification, there is no real training step. Once the rules are codified, that's it, you're ready to run. With the advent of ML, it does not mean that rule-based models are no longer needed. Here are some points to keep in mind when choosing rule-based analysis. The problem statement that you're working with is fairly simple. It's not too complex, you know the patterns. The rules that you need to formulate are straightforward and can be easily codified. Rule-based analysis also makes sense when the rules change infrequently. The rules are mostly static. Once you set them up, you don't need to worry about them too often. And finally, rule-based analysis is what you'll have to work with when you have just a few problem instances and they can be used to train your machine learning model. When would you choose to go with ML-based analysis? When the problem statement that you're working with is reasonably complex, you can't really formulate rules for these. There are patterns and other bits of information that you don't really know of. When it's hard to find patterns using visualizations and other exploratory tools, you might build an ML model. You'll also use ML when the decision variables are sensitive to data, and your decision variables need to change as new information is received. This essentially means that your decision parameters are highly dynamic. That's when you really need ML. You'll also used ML-based analysis if you have a large corpus available to train your ML models.

Traditional Machine Learning Systems

And this discussion of choosing between rule-based and machine learning-based systems leads us straight to a discussion of what exactly machine learning is, and we'll discuss this in the context of traditional machine learning algorithms. So what do machine learning algorithms really help us with? It allows us to work with a huge maze of data, which may not give you all of the insights that you're looking for using traditional methods of analysis. Machine learning can help you find patterns that exist in this data, and using these patterns, you'll be able to make intelligent decisions. A machine learning algorithm is an algorithm that is able to learn from data. This is a single sentence that completely defines what machine learning is. Machine learning algorithms have been drawn up by experts and in fact, there are many such algorithms available. All of these algorithms have been so structured that they're able to learn from the data that you feed it. Let's work with an example. Let's say you want to build an ML-based classifier to figure out whether whales are fish or mammals. Now we know that whales are members of the infraorder Cetacea, so they're probably mammals, but you also know that whales look like fish, swim like fish, and they move with fish; maybe they're fish. This is once again a binary classification problem. If you had a rule-based classifier, you would have human experts draw up the rules. The human experts would know exactly what rules to apply to make this classification. You'd simply say whale and this rule-based classifier would say, it's a mammal. Let's say you're working with an ML-based classifier. Things would change a little bit. This classifier would be trained on a huge corpus of data. The records in this corpus would be both fish, as well as mammals and their characteristics, and you'd feed into this ML-based classifier characteristics of the whale. If you were to feed in the information, the whale breeds like a mammal and gives birth like a mammal, the classifier would say the whale is a mammal, but let's say you give his ML-based classifier different information about the whale. You say that it moves like a fish and looks like a fish, the classifier is likely to say the whale is a fish. We had spoken earlier about how machine learning classifiers are trained. You have a huge corpus of data, which you feed into a classification algorithm. This algorithm trains or finds patterns in this data, and the final result of the training is an ML-based classifier, our machine learning model. What we've just described here is a traditional ML-based binary classifier. Why traditional? Well, keep this term in mind. We'll discuss what traditional means in this context in just a bit. Now the input that you feed into your traditional ML-based binary classifier is a feature vector. These are the characteristics of a whale that you're feeding in. The output prediction or classification that you get from this ML model is referred to as a label. Now it's possible that you had fed in a different set of characteristics about the whale. You told the classifier that the whale moves like a fish and looks like a fish. What you fed in here is a poor choice of features. The features that you've chosen to tell the classifier leads it to make the wrong prediction. The output of the classifier in this case might be an incorrectly predicted label. It thinks that the whale is a fish because of what you've told it about the whale. The input that you feed into your machine learning model, whether it's a classifier or some other kind of model are features. Features are attributes that the machine learning algorithm focuses on, during training and by making its prediction. Every data point is a list or a vector of such features, which is why these features are often referred to as feature vectors. The features tell your model what is important about the problem instance. Features are also referred to as X variables, and with this in mind, I'm ready to define what I meant by traditional. Traditional ML-based system rely on experts to decide what features to pay attention to. The features that you feed in, you are the one making this decision about what to feed into your model so that the output is correct. And this is an important attribute of traditional machine learning models. Machine learning models can also be representation ML-based systems. These figure out by themselves what features to pay attention to. You just feed in a bunch of features, these models figure out which features are important.

Representation Machine Learning Systems

In the last clip, we introduced and discussed traditional machine learning algorithms. In this clip, we'll discuss representation based systems. We'll understand what deep learning is all about. Deep learning is an example of representation machine learning-based systems. As we've discussed before, these systems figure out by themselves what features are important, what features to pay attention to. Here is how we had set up the traditional machine learning-based classifier model. We had a huge corpus of data. We fed this data in into a classification algorithm. This algorithm trained on this data and created an ML-based classifier. Let's dig a little deeper. While training your ML-based classifier, what features you'll use to train this classification algorithm were selected by experts. Once and done, your data processing cleaning and coding, it is you the expert who determines what features are important and what features you need to train this traditional ML-based system. Whereas if you're working with representation ML-based systems, there is a feature selection algorithm. This algorithm is a part of the ML system, and it's responsible for figuring out what features are important, and really you don't need to know the workings of this algorithm when you're working with representation-based systems. To intuitively understand what this means, let's talk about a traditional ML-based binary classifier. We fed in the features of a whale and the classifier trained on a huge corpus of data will hopefully tell us that it is a mammal. Representation-based systems may just take in a picture or a video of a whale and predict that it's a mammal. You've not indicated what features are important; you've not told your classifier what to look at in this picture or video. The classifier has figured out by itself. The design of the classifier in this representation ML-based system is such that it has a feature selection algorithm, which tells it what to look at, and deep learning systems are one type of representation systems. Deep learning systems are the most widely used representation systems today. They are very popular and the most widely used deep learning system is a neural network.

Deep Learning and Neural Networks

Now that we've understood the significant features of representation-based ML systems, we can move on to discussing deep learning and neural networks. These are complex and nuanced topics, and there are of course many ways to describe this. Here is one way to look at this that makes the most intuitive sense to me. Deep learning refers to those algorithms that learn what features matter in your data, and they're able to use those features to make important and insightful predictions. Deep learning is a vast field, and neural networks can be thought of as an implementation of deep learning. Neural networks are the most common class of deep learning algorithms. These are the most widely researched algorithms and these are the ones that are most used in practice, which is why deep learning and neural networks have almost become synonyms. The architecture of neural networks is such that they're made up of many layers, and they can be very, very deep, which is why they're considered to be deep learning, and what are neurons? Neurons are very simple building blocks that actually learn from your data. Neurons are just mathematical functions that have the ability to learn. Here is a big-picture overview of what a neural network looks like. This is an example of a deep learning-based binary classifier, and like I said, deep learning and neural networks are almost synonymous these days, so we'll talk specifically about neural networks. A deep learning binary classifier includes within it a feature selection algorithm, as well as the classification algorithm. There's actually no explicit separation between these two. It's just for your intuitive understanding. Observe that this neural network is divided into layers. Observe that the layers are arranged sequentially. This is one design of a neural network; there are several other designs, which we won't get into here. Each layer of the neural network is responsible for extracting different features from the data that you feed in. This data could be a corpus images, videos, or any other kind of data. Each of these layers are composed of active learning units; the neurons that actually learn from your data. The neurons in a neural network are arranged in layers and these layers can be thought of as an aggregation of those neurons, which extract similar features from the underlying data. For example, when you're working with images, the earlier layers in the neural network might extract more granular details from your image data. It might focus on pixels, lines, vertical and horizontal edges, corners, and so on. Higher layers might put all of this granular information together to identify object parts and other more abstract information. So what exactly is this neuron which is our active learning unit? A neuron is nothing but a mathematical function. In fact, it's a very simple mathematical function that operates on the inputs that it's fed in. A neuron will apply this mathematical function to the input X values and produce a result, which is the Y value of a neuron. A neuron in a neural network is said to be active when it's actively learning from the data. It's not dead of saturated. A neuron is said to be active when a change in inputs of the neuron triggers a corresponding change in the output. That is when a neuron is sensitive to its input. We saw how in a neural network neurons are laid out in layers and interconnected with neurons in other layers. The outputs of neurons feed into the neurons in the next layer. So there is an interconnection that exists, and each of these interconnections is associated with a weight usually represented by w. The value of the weights that are associated with these neuron interconnections are found during the training process of the neural network. If the output of the second neuron is very sensitive to the output of the first neuron, the connection between them gets stronger and the value of w increases. The more sensitive one neuron is to the output of another, the stronger the connection, the higher the weight, and this is the fundamental principle underlying any neural network. Cells that fire together, wire together. As more and more data, that is your training data, is fed through the neural network during the training process, the weights of these interconnections are constantly tweaked to improve the performance of your model, and all of these neurons arranged in layers and interconnected make a neural network, and these neural networks are amazingly powerful. They help find unknown patterns in massive datasets.

Traditional ML vs. Deep Learning

We are now finally ready to compare and contrast traditional machine learning versus deep learning algorithms. We've moved on from rule-based systems; both of these are machine learning systems, and both of these are dynamic in nature, and they learn from data, they change as the data changes. The most significant difference between these two is one that we've already discussed in some detail. When you work with traditional machine learning models, you don't need as many experts as with rule-based systems, but there is some need for expert skill to select features. With deep learning systems there is even less need for expert skill for feature selection, because the feature selection algorithm is part of the model. When you're training traditional machine learning models, you'll have experts select the right features to feed in to train your model. In the case of deep learning, you'll feed in all of the features that you have available, and there are algorithms which extract the right features, such as the layers of your neural network. Since there is some level of feature selection involved using experts, traditional ML typically works well with numeric data where the features are clearly defined. Deep learning models also work well with numeric data; they work well with all kinds of data. The algorithms that you'll use in traditional machine learning models are not really specialized for specific use cases. They're not specialized for image data, video, or text data. Whereas with deep learning models such as neural networks, you can customize them in such a way that they can deal with images, videos, and complex text, and in fact there are neural networks that are specifically customized to work on this specialized kind of data. Because feature selection is part of the neural network model, they work very well with unstructured data such as these. When you're working with traditional machine learning models, you'll see that there is an explicit algorithm, one that you can intuitively understand that learns from the data. With deep learning, there is no explicit algorithm. In fact, neural network models are often a black box. You'll find that many traditional machine learning algorithms make conceptual sense, as we'll see in just a bit. Model a tree structure on the data, find a hyperplane that classifies instances, fit a straight line. These are examples of traditional ML. With neural networks, it's hard to get a conceptual visual idea of what exactly is going on. You can design a highly custom neural network, you can design complicated interconnections; you'll train and see what works well. You have an intuitive understanding of the mathematics involved, but you don't really know what's happening under the hood. Traditional machine learning algorithms have a number of specified model parameters, and the training process explicitly fits these model parameter values. With deep learning, the training optimizes neural network weights and biases. The weights are the strengths of the interconnections between the neurons and different layers of the neural network.

Traditional ML Algorithms and Neural Network Design

We've spoken of the fact that traditional ML models are built using specific algorithms that have a conceptual basis, something that you can understand. These algorithms are not black boxes. For example, a linear regression that you can use for prediction is an example of a traditional algorithm that tries to fit a straight line on your data. Support vector machines is another popular traditional ML algorithm usually used for classification problems. SVMs try and find the best hyperplane that classifies your data into categories or classes. Decision trees are another example of a traditional ML algorithm, and these model a tree structure during the training process and the nodes in the tree are used to make decisions about the data. We'll look at each of these examples of traditional ML algorithms in some detail, not too much, but just to give you a big-picture overview. Linear regression tries to model cause/effect analysis where X is the cause or the explanatory variable and Y is the effect. The objective of the linear regression algorithm is to find the best fit line through all of your data points, and this line can then be used for prediction. Let's model this as a real-world problem. On the X axis you have the government bond yields, and on the Y axis you have oil prices. Oil prices are affected by government bond yields. You might fit a line through this data and then use this to make predictions. Say government bond yields have moved. Given a new value of X, we'll use this regression line to predict the corresponding value of Y; that is what will oil prices be at given that yield is 3%? Let's move on to understanding a different algorithm, support vector machines for classification. Let's say you have your data points set up in a two-dimensional plane as you see here on screen. You have this information about your data; the time when a particular review was posted, and the number of words in that review. You want to figure out whether a review is positive or negative. Support vector machines will try and draw a line through your data in order to classify your data points. It'll try and see whether such a line exists. Bi-dimensional data points can be represented using a plane and classified with a line. Support vector machines also work with multi-dimensional data. You can have data on multiple axes, N-dimensional data as it were. N-dimensional data can be represented using a hypercube and separated using a hyperplane. Support vector classifiers try to find that hyperplane that best separates points in a hypercube. We'll discuss one more example of an algorithm in traditional machine learning, and here we'll talk about decision trees for classification. Our example here is that you're building a classification model to determine whether a sports person is a jockey or a basketball player. Now we know some things intuitively about these sports people. We know that jockeys tend to be light to meet horse-carrying limits and basketball players tend to be tall, strong, and heavy. The height and the weight of a sports person is available to you as a part of your training data. The decision tree algorithm will try and fit this knowledge into rules and construct a tree in the process. Observe that every node in this tree is a decision node based on a particular feature, weight or height, and each of these features are associated with a threshold. The objective of the decision tree is to fit knowledge into rules. Each rule involves a threshold and these rules are used to make predictions. Deep learning models don't really have algorithms that are git to the data. They just have design and this design extracts features from the underlying data. Here are the most common kinds of deep learning neural networks that you'll encounter. You can have a dense, fully connected neural network where the neuron in a particular layer is connected to every neuron in the following layer and receives input from every neuron in the preceding layer. You can have a convolutional neural network which is a sparse neural network. Every neuron is not connected to every neuron in preceding and following layers. Convolutional neural networks specialize in working with images and they process images in two dimensions. Convolutional neural networks are designed to mimic the neurons in the eye and the visual cortex of the brain. Another popular neural network design is the recurrent neural network where the neurons hold additional state or memory. Recurrent neural networks work very well with time-series data and text sequences, and they're often used in natural language processing applications.

Module Summary

And on this note, we come to the very end of this module where we explore approaches to machine learning. We started this discussion off by talking about data-driven decisions and actions, and in fact, we studied this using a simple sentiment analysis case study. We saw a hypothetical example of how sentiment analysis can be used to drive decisions in event-driven trading. In this context, we discussed rule-based approaches to learning, its strengths and weaknesses. Rule-based approaches are static in nature. We moved on to learning dynamically from changing data using machine learning techniques. We saw that ML techniques work well when there are no formal rules that you can set up, and when your context or your data changes very quickly. We discussed how machine learning models are very sensitive to the features on which they are trained, and how representation based systems can extract features from even unstructured data. In this context, we discussed deep learning as an example of representation-based systems. We saw that the most popular class of deep learning algorithms were neural networks. We saw how neurons can learn from data. And finally, we rounded off this module by discussing the differences and similarities between traditional machine learning versus deep learning. In the next module we'll discuss approaches to choosing the right machine learning problem, and in that context we'll discuss unsupervised learning, supervised learning, and reinforcement learning.

Choosing the Right Machine Learning Problem

Module Overview

Hi, and welcome to this module on Choosing the Right Machine Learning Problem. You have your data and you have your use case. How do you use this information to figure out the right problem to solve? This is what we'll try and answer in this module. In this module we'll start off with a discussion of the canonical problems in machine learning. Machine learning techniques and algorithms have been developed to solve problems in four broad, commonly used categories. These are classification, regression, clustering, and dimensionality reduction. We'll discuss each of these machine learning problems in some detail, and we'll discuss and explore use cases. In addition to these more common problem categories, we'll also explore and discuss specialized problem categories such as reinforcement learning, association of rules detection, and recommendation systems. We'll categorize machine learning techniques that we've discussed into two broad categories; supervised learning that requires label data, and unsupervised learning that works with unlabeled data, but what if you have no data to work with at all and are in an unknown, uncertain environment? That's when you choose to reinforcement learning. We'll discuss what reinforcement learning is and contrast it with supervised and unsupervised learning techniques.

Choosing the Right Machine Learning Problem

You've collected a bunch of data and you know you want to use machine learning techniques to analyze this data. Let's see how you can choose the right machine learning problem for your use case. We'll start off our discussion with the broad problem categories. The first and the most intuitive of all of these problem categories is classification when you want to categorize or classify your data in some way. Spam or ham, good or bad, positive or negative? A second problem category is when you want your machine learning model to predict a continuous value; that is regression. What's the price of this house given its location and all of its features? What will be the price of this stock tomorrow given all of the past data? Your problem might be that you have a huge amount of data and you want to find patterns that exist within it, include things within your data that make sense. This requires clustering. And finally, you have data that is very huge, and every record has a huge number of features. You don't know which features are relevant or not; that's when you'll choose dimensionality reduction. Now that we've understood the broad problem categories, we can discuss each of these categories in detail and talk about use cases. When would you need to use classification? When you need to classify your input data into categories or classes. It turns out that predicting categories is a very common use case and these categories would be virtually anything. In terms of email, is this email spam or ham? Should I send it to the inbox or to the spam folder? As a financial trader, you're constantly monitoring stock information. Given the past performance of the market, the performance of the company, and the performance of this stock, should you buy, sell, or hold? You're working with image data and you want to do object recognition. Is this a cat, dog, or a mouse? That's an example of classification. You're working with text data and you want to analyze sentiment; positive, negative, or neutral; that's another example of classification. As you can see from all of these examples here, the output of a classification model is one category or class. Let's move on to our problem categories here and we'll now discuss regression. Your problem is one that needs a regression model when you want your model to predict continuous numeric values. You're a financial trader, and given current market sentiment, how a company has performed its last earnings, you need a model to predict the price of the stock tomorrow. The prediction is a continuous numeric value, you need a regression model. You're analyzing the performance of different cars available. Given the characteristics and attributes of a car, maybe its model and a bunch of other details, predict its mileage. Or you're performing analysis on housing prices in a particular state or a city. Given the location and the attributes of a particular house, how new it is, when it has been renovated, and so on, predict the price of this home. Observe the nature of the problem in each of the cases that we've discussed here. If we want to predict a continuous numeric value, we'll use regression. We'll now move on to the next broad problem category here, that is clustering, and discover patterns and groupings that exist in your data. You have an extremely large dataset, you have no idea what's inside it, you want to make some sense of it. That's when you'll tend to use clustering. Maybe you're performing document discovery. Find all documents related to homicide cases. You want to analyze only documents of this kind to see if you can find patterns. Social media ad targeting; find all users who are interested in sports so you can target sports-specific ads to them, maybe try and sell them a subscription to your sports-training channel. You have lots of data, each data point has many different attributes, but your data isn't set up in a way that you can use it to train a predictive model such as regression or classification models. Clustering allows you to self-discover patterns in such detail. We'll now move on to discussing dimensionality reduction as a problem category. This is what you'd use to perform feature detection on your data. Dimensionality reduction is also used as a preprocessing technique; it can also be used to find the latent factors that exist in your data such as find the latent drivers of stock price movements. You have 500 different variables; which of these are the most significant? Which should you pay more attention to? That's a problem of dimensionality reduction. Or feature detection. Now dimensionality reduction is used really often to preprocess your data to build more robust machine learning models, whether they're classification, regression, or some other kind of predictor model. Extracting latent features using dimensionality reduction greatly improves the performance of models during training and during prediction as well. You'll use dimensionality reduction to find latent factors when you have a lot of data to work with, and you don't have target Y labels or values in your data. I'm going to categorize these models into two broad categories that will lead us to our next discussion. Classification and regression models are examples of supervised learning. Clustering and dimensionality reduction techniques are examples of unsupervised learning. I won't go into the details of supervised and unsupervised learning just yet; that's what we'll cover in the next clip, but before we move on from here, let's talk about a few specialized problem categories for which you can use ML models. Recommendation systems, which you'll use to recommend products to users. Based on a user's past history and other users like him, do you think he'll like this product? That is recommendation. Association rules detection detect transactions that occur together. If a person has bought books, is he likely to purchase eyeglasses as well? And finally, another interesting problem category, reinforcement learning. Train an agent to navigate an uncertain environment. You have an environment you know nothing about, how do you get an agent to navigate it successfully? We'll discuss some of these techniques in more detail later on in this module, but before we end here, I'm going to cover a few broad solution categories. If you're working with image data and you want to use deep learning, convolutional neural networks will be your machine learning model of choice. If you're working with very complex text data, you might want to use recurrent neural networks that work well with sequences. If you're analyzing time series data, once again you'll choose recurrent neural networks. Anything to do with a sequence of data. We'll discuss the applications and users of neural networks in a little more detail in a later module.

Supervised and Unsupervised Learning

In the previous clip I introduced the terms supervised and unsupervised learning, but I didn't really talk about what these are. That's exactly what we'll discuss here in this clip. Specifically in the context of unsupervised learning, I'd like you to remember this famous saying: What lies behind us and what lies ahead of us are tiny matters compared to what lives within us, and this in essence captures the basic principle behind unsupervised learning. Here is a classification problem that we've discussed earlier. Whales, are they fish or mammals? They could be mammals because they are members of the infraorder Cetacea; they could be fish because they look like fish, swim like fish, and move with fish. You want to build and train a machine learning model that is a classification model to make this distinction. Classify whales as fish or mammals. What do you need to do? The first step to build your ML model is training. This is where you'll feed in a large corpus of data correctly classified. Your model, whether it's a traditional ML model or a neural network, will train on this data and once you have a fully trained model, you'll use it for prediction. You'll use it to classify new instances, which your model hasn't seen before. In order to get a machine learning model, the first step is training. You feed in a huge corpus data which has been correctly labeled, the model will give some kind of classification output. Initially it's quite likely that your model will get most of its predicted classifications wrong. This is then served as a feedback to train your model, and this feedback improves your model's parameters. Your training data has the right value for each record and it uses this correct value to correct and tweak your model's parameters to build a better model. you now have a classifier trained on a corpus of data, and it's able to make the right predictions, hopefully. Hopefully it's a good model. This whole process of training your model and using it for prediction is a key characteristic of supervised machine learning. The X variables are the features that you feed into your model to train your model. The Y values are what you want your model to predict, and your supervised machine learning techniques seek to learn the function f that links the features and the labels. The entire training process of the model is devoted to figuring out what exactly this f is that is the relationship between the input and the finding prediction. Now when your machine learning model is a linear regression model, this specifies upfront that the relationship between the input and the output is linear. It can be fit using this straight-line formula, but not all relationships can be so simply expressed. It's quite possible that the function f is something really complicated. Machine learning algorithms such as neural networks can learn or reverse engineer pretty much anything given the right training data. However complex f is, given the right data and the corresponding Y values or labels, your model can be trained to make good predictions. What we've discussed so far all apply to supervised learning. Unsupervised learning learns patterns in data without a labeled corpus. So you can train your model using the correct labels. There are no labels available at all. Machine learning algorithms can be divided into two broad categories. Supervised learning where labels associated with the training data is used to correct and tweak your algorithm. And on the other hand, we have unsupervised learning where your training data does not have any labels upfront. The model has to be set up just right in order to learn structure and patterns that exist in your data. Here is a quick summary of supervised learning. You have your input variable x and the output variable y, and your training data contains both of these values. Your algorithm is trained to learn the mapping function y is equal to f x. The entire process of training your model is to approximate this mapping function so that for any new value of x we can predict the corresponding y. This reverse engineering of this mapping function is made possible by using our training dataset to correct our mapping function approximation. Let's contrast this with unsupervised learning. The data that you have to work with only has x variables; it does not contain the corresponding y variables or labels, which means if you have to discover patterns in your data, you need to model the underlying structure to learn more about your data. Unsupervised learning algorithms self-discover patterns and structure that exists in the data. Unsupervised learning algorithms have to look within the data to find answers. Why look within? Well, in the real world there are several reasons, and many of these reasons are rather philosophical; to be emotionally self-sufficient, to learn what values matter to you, to identify others who share your values and those who don't, you eliminate unimportant features, things that don't matter, and in general, to train yourself to navigate the outside world. Philosophical or not, these same reasons apply to using unsupervised techniques in machine learning. Unsupervised techniques allow making unlabeled data self-sufficient. If you think about it, the vast majority of the data that exists in the world is unlabeled. You need a way to work with this data. You want to learn what values matter to you? Well, that corresponds to latent factor analysis, or find significant features in your data. You need to be able to identify others who share your values. Well, that is clustering, finding logical groupings in data. You also need to be able to identify those who do not share your values. Well, that is anomaly detection. You need to look within to eliminate what does not matter; that is quantization where the irrelevant levels are removed. You look within to train yourself to navigate the outside world. Well, unsupervised learning is pre-training for supervised learning problems such as classification or regression. Now that we've understood these machine learning techniques, let's apply them to use cases. To make unlabeled data self-sufficient, to identify photos of a specific individual. If you identify the same person across multiple photos, once you tag one photo, the remaining photographs will get tags as well. Latent factor analysis? Well, that's how you might find common drivers of 200 stocks. Clustering, find the relevant document in a corpus. Anomaly detection, flag fraudulent credit card transactions. An anomaly is a data point that is a deviant or different from the rest, a transaction that does not fit previous patterns. Quantization, compress true color, which is expressed using 24 bits to 8 bits. And as for unsupervised learning as pre-training for supervised learning, all of the machine learning use cases above can use this. For each machine learning technique here, there is a specific name; auto encoders, clustering, both of them put together. And if you remember our previous discussion in the previous clip, we'd spoken about clustering and dimensionality reduction as unsupervised learning techniques.

Reinforcement Learning

Now that we've understood supervised and unsupervised learning, let's move on to discussing reinforcement learning, which is a very different kind of machine learning problem. Here is a way to think about reinforcement learning. If you ask the wrong question, you will never get the right answer. At this point in time you clearly understand what supervised and unsupervised learning is all about. With supervised learning, once you have a trained model, given x, it predicts y. With unsupervised learning, once you've fed the data into your model, given x, it will simplify x. Observe that in both cases we have x. X here refers to your data, whether labeled or unlabeled. X is what you feed into the machine learning model. What if you have absolutely no idea what x is? What now? Can you use machine learning? Well, neither supervised nor unsupervised learning will work in an unknown environment where x is not available. Does this mean that you can't use machine learning in an unknown environment? That would be quite disappointing, but you don't need to be disappointed, because you have reinforcement learning. This is where you train decision makers to take actions to maximize rewards in an uncertain environment. Reinforcement learning is what you'll use when you want to get a handle on how to work or function in an uncertain, unknown environment. It talks about decision makers here. Decision makers are essentially software programs or agents. These are the agents that are responsible for navigating this unknown environment. Think about the robot that's learning to navigate an environment with obstacles. There are trees, there are boulders, the robot is the agent. The objective of this robot is to cross all of these obstacles and get to its destination so the robot needs to take actions so that its objective can be achieved. The actions are determined using an algorithm, and the algorithm that determines these actions is called the policy. The robot or the decision maker does not take actions at random. The actions are based on this policy, but which actions are right and which ones are wrong? The actions are so tailored so as to maximize rewards. Imagine that the robot is walking around and it manages to take five steps in a row. This action will be rewarded; the robot made progress, but it's possible that the robot walked around, that was the action, in a different direction where it promptly slammed into a tree. That is the punishment. This trains the robot to choose one direction over another. The robot will seek to maximize rewards. The robot will also seek to avoid punishments. Both rewards, as well as punishments are externally imposed by the environment in which the robot operates. The whole thing about reinforcement learning is that this environment is not known upfront. It's an unknown environment. The environment is complex so the reward and the punishment for the different actions is not usually known in advance, and this is the training process. You need to train the decision maker that is our software programmer agent to explore that uncertain environment, combining both caution, as well as courage. We want to ensure that our robot doesn't walk off a cliff at the same time the robot should make progress in the direction of the destination, and this in essence is what reinforcement learning is all about. I'll quickly summarize what we just discussed. Reinforcement learning uses an agent who is the decision maker in an unknown environment. This agent is responsible for observing the environment and then taking actions based on his or her observations. The agent's actions are tailored to maximize rewards and minimize punishment. The whole objective of this training is to maximize and get as many rewards as possible. The agent cannot take actions at random. There is an algorithm which determines what actions the action can take, and this algorithm is referred to as the policy. The policy of course is not known upfront; it's the training process that determines the policy. The policy is determined by exploring the environment. Reinforcement learning is an interesting and exciting machine learning technique, and it has several users. It's used in robotics. The self-navigating robot that I discussed as an example throughout, that's an example of reinforcement learning. Reinforcement learning is also used in text mining to generate summaries of a large corpus of text data. Reinforcement learning also finds uses in healthcare, for example to optimize medication dosing. As you can see, reinforcement learning is different from traditional supervised and unsupervised learning techniques. Let's compare and contrast the two. The objective of the reinforcement learning algorithm is to choose the best actions to maximize rewards. Supervised learning is used for prediction or classification, unsupervised learning for simplification or dimensionality reduction. Reinforcement learning is used when x variables are unknown, the environment is uncertain and not known upfront. Supervised and unsupervised learning works with a known environment, x is known. The training process in reinforcement learning involves an agent who's exploring the uncertain unknown environment to maximize rewards and minimize punishments. In the case of supervised or unsupervised learning, training involves finding patterns or relationships that exist in data. In certain unsupervised learning techniques training might be entirely absent. With the reinforcement learning, as the agent is navigating the uncertain environment, wrong actions taken by the agent get punished, right actions get rewarded. With supervised learning, the loss of the incorrect predictions are used to train the model. With reinforcement learning, the objective of the training process involves determining the best policy for actions. With supervised learning, training process involves fitting the best model. With unsupervised learning, there is really no training, and finally with reinforcement learning there is an explicit dependency of rewards on previous actions. Actions are linked together in this way. With supervised, as well as supervised learning, every data point is completely independent of other data points.

Recommendation Systems

In this clip we'll discuss another specialized machine learning problem, that of recommending products to users. If this is your problem statement, you'll build a recommendation system to solve this. Recommendation systems are often used in e-commerce sites, which is why I'm going to use that as my hypothetical example here. Any commerce site may have millions of users seeking to buy products, and it'll have many millions of products listed. You want to feed users and products into a recommendation engine, and you want to be able to match products with users. This is the entire objective of recommendation. With the right match, the users that are happy will buy more. There's a bunch of information about users and products that you might have such as what products has a user bought? How has he rated certain products? All of these bits of information serve as the input for your recommendation system. Now a recommendation system can be built using several different approaches. We'll discuss three specific approaches here. You can have content-based recommendations where you estimate the rating a user might give a product using just information about the user and information about a particular product, or you could build recommendation systems using collaborative filtering where you employ information about other users and other products in order to make your recommendations. You find other users who are like this particular user; you find other products that are like this particular product. The third alternative here would be to combine both approaches, content based, as well as collaborative filtering. This is the hybrid approach to recommendations. Let's start off by discussing content-based filtering where you estimate the rating using only information about a particular user and a particular product. You have individual users who are users of your e-commerce sites and they have viewed and bought several products. You use this information to give these users personalized recommendations, and with this, you get more information about users. A user might view a recommendation, might purchase a recommended product. All of this feeds back into this content-based filtering. With all of this information you've built up a user profile and a production description profile, and content-based filtering involves matching these two. Content-based filtering has two significant drawbacks. You require accurate rich product metadata, and in the real world this may not always be available. Content-based filtering is also hard to extend across different types of products. Let's move on to the next approach to building a recommendation system, collaborative filtering where you employ information about other users and other products. It's not just individual users and products that you're looking at here; you'll use the information that you have available about other users of this e-commerce site. If you analyze the data for all of the users across your site, you might find patterns. You might find other users who are similar to this user to whom you want to show a recommended product. For similar users, you'll have information about what they liked, what they viewed, and what they bought, and you'll use this information to recommend products. Collaborative filtering is based on the principle that you'll find similar users based on their characteristics. Users who have agreed in past will likely agree in the future, and these users will like similar kinds of items as they liked in the past. If this definition was way too verbose, just remember this. People who X also buy Y. Just figure out if an individual user is a person who buys X, then use that information to recommend Y.

Module Summary

And with this we come to the very end of this module on choosing the right machine learning problem. We started this module off with a discussion of canonical problems in machine learning, and in this context we discussed four problem categories: classification, regression, clustering, and dimensionality reduction. We saw that classification models can be used to predict classes or categories, regression models to predict continuous numeric values, clustering to find patterns in data, and dimensionality reduction to simplify your x data. In addition to these canonical ML problems, we discussed more specialized problem categories such as recommendation systems, association rule detection, and reinforcement learning. We compared and contrasted supervised and unsupervised learning techniques, and learned where unsupervised learning could be used. Both supervised, as well as unsupervised learning techniques require features or x data, but what if you're in an unknown uncertain environment? That's when you'll choose to use reinforcement learning. We discussed how reinforcement learning can be used to train an agent to navigate an uncertain environment, and we compared and contrasted reinforcement learning versus supervised and unsupervised learning techniques. In the next module we'll discuss how you can choose the right machine learning solution, and in that context we'll see how you can choose and evaluate regression, classification, clustering, and dimensionality reduction models.

Choosing the Right Machine Learning Solution

Module Overview

Hi, and welcome to this module on Choosing the Right Machine Learning Solution. In this module we'll focus our discussion on choosing and evaluating machine learning models for four important kinds of models: regression, classification, clustering, and dimensionality reduction. We'll start our discussion off by talking about regression models. We'll discuss the basic assumptions made by regression models. We'll see how we can choose the right regression algorithm, and we'll discuss techniques to evaluate regression models. We'll then move on to classification models and discuss the different types of classification tasks: binary classification, multi-class classification, multi-label classification, and multi-output classification. We'll discuss how we can choose the right classification model for our use case, and discuss accuracy, precision, and recall measures to evaluate classifiers. We'll then turn our attention to clustering models and discuss the objectives and outcomes of clustering. We'll also see how we can choose between different clustering techniques. And finally to round out our discussion on choosing the right machine learning solution, we'll talk about dimensionality reduction. We'll discuss the curse of dimensionality, how it affects building and training of our models, and we'll discuss techniques to reduce dimensionality. Before we get that, let's talk about the broad problem categories and the kind of model that you'll use for each of these categories. If you want to predict continuous values, you'll choose to build regression models. If you want your model to predict categorical values, that is classify your input instances, you'll build a classification model. If you want to find patterns within your data and you have no y values or labels in your data, you'll choose clustering or dimensionality reduction. And finally, if you want to simplify complex x data as a pre-processing technique before you feed into other machine learning models, you'll choose dimensionality reduction.

Regression Models

We'll start our discussion of the different machine learning solutions by talking about regression models. Regression models are what you'll build and train if you want to predict a continuous numeric value. Regression models model cause and effect. It's often used for cause-effect analysis. Cause here is the X variable, the explanatory variable or the independent variable represented on the X axis. The effect is the Y variable or the dependent variable represented on the Y axis. Linear regression involves looking at your data points and finding the best fit line that models your underlying data. Regression models of course go beyond linear regression. You have multiple regression and you have regression that finds very complex curves as well. I'm using the example of linear regression here as a stand-in for regression in general. The best fit line in linear regression is the one where the sum of the squares of the lengths of these dotted lines is minimum. Linear regression can be modeled as an optimization problem where your objective function is to minimize this mean square error, the sum of the squares of the lengths of these dotted lines. When you train your linear regression model on your training data, this is the best fit line that it tries to find. Finding this line is the objective of the regression problem. Now obviously there are some differences between the fitted values represented by this line and the actual Y values from your training data. When you perform regression analysis, the residuals of the regression are the difference between actual and fitted values of the dependent variable. In order to find the best fit line in linear regression, we need to make some assumptions about this regression error, that is the difference between the actual and the fitted values. Now there exists a fine distinction between errors and residuals, but for now we can just ignore it. And here are the assumptions that we make in a regression model. Ideally, the residuals should have 0 mean, common variance, be independent of each other, be independent of the x variable or the explanatory variable, and be normally distributed.

Choosing Regression Algorithms

Now regression models can be built using many different techniques. In fact, you can fit a formula and perform regression, that is the analytical technique, but in the world of machine learning techniques to perform regression, you have a huge choice of the regression algorithms that you can work with. How do you pick and choose the right one that fits your use case? What I'm about to discuss here using this 3x3 grid is one approach that you could follow to choose a regression algorithm. The first thing you'll need to take into account is the number of features that exist in your dataset. Are there many features, a moderate number of features, or just a few features? The second thing that we'll consider here is the size of the dataset as a whole. How many training samples do you have? Is it a small number of samples, a medium number of samples, or a large number of samples? If you have a very large dataset with very high dimensionality, that is many features and many training samples, a regression algorithm that works here will be the Stochastic Gradient Descent algorithm. The performance of the SGD algorithm is linear in the number of training examples. Stochastic Gradient Descent iteratively trains on your data by taking in one sample at a time during training and improves the models' parameters. If you're working with features of very high dimensionality, but your dataset is fairly small, you might choose to go with Least Angle Regression or LARS. Least Angle Regression is similar to forward stepwise regression. At every step, it finds that feature that is most correlated with the target value and includes that feature to be part of the regression analysis. Thus at every step it tries and improves the analysis by adding features. If you're working with a medium-size dataset and you have a moderate number of features in your regression analysis, the Lasso and ElasticNet regressors will work for you. The Lasso and ElasticNet regressors are regularized models that mitigate over-fitting on the training data. There is a regularization function that is a penalty, which is imposed on very complex models. Models that use too many regression coefficients. If you're working with a medium-sized dataset, but there are many features in each record, Ridge regression will work well for you. Ridge regression is also a regularized regression model. It imposes a penalty on your coefficients to force the algorithm to fit a simpler model. If you're working on a small dataset with just a moderate number of features per record, you might find that support vector regression using the Linear Kernel works well. This regression algorithm is based on support vector machines, which is generally used for classification models, but can also be used for regression. SVMs with the Linear Kernel fit a simpler model and generally works with linearly separable data. If your data seems to exhibit non-linear properties, that is it cannot be expressed as a sum of linear components, you might want to change the kernel. You won't use a linear kernel, you'll use a more complex kernel such as the RBF kernel. Support vector regression with the RBF kernel can work on small datasets with a few features where the data exhibits non-linearity. Non-linear kernels use a kernel trick to transform the data before performing regression. Using the kernel trick to transform your data is a common technique that is applied to simplify the data that you are working with. If you have a medium-sized dataset with just a few features, decision trees and ensemble techniques for regression work well. We'll talk more about ensemble solutions in a later model, but it's essentially using many models together to combine their predictions. And finally, if you have a large dataset with just a few features, you'll just go with Ordinary Least Squares regression. This is the simple regression technique that we discussed where we minimized the mean square error as our objective function. I've tried to give you a big-picture overview of choosing the right regression model to work with. If you're interested in learning more about how these different regression models work, and how you can implement them in a hands-on manner, there is a course on Pluralsight that you can watch, Building Regression Models with Scikit-Learn.

Evaluating Regression Models

You know that for your use case you need to build and train a regression model and you've also chosen your regression algorithm. Now that you have a fully trained model, how do you evaluate it? That's what we'll discuss here. The results of your regression analysis can be interpreted using many different techniques. You can use the Adjusted R Squared, Residuals, the F statistic, T-statistics, and R squared. We'll discuss what each of these measures mean and how they could help you evaluate your regression model, starting off with R-squared and Residuals. The measure that is the most widely used to evaluate your regression model is R-squared. R-squared is a measure of the overall quality of fit of your regression line. The higher the R-square, the better the model up to a point. If you find that your model has very high R-square on the training data, but low value of R-square on the test data, your model is likely overfitted on the training data. You can also use the residuals to interpret your regression results. Remember we had discussed basic regression assumptions earlier? You'd use these residuals to check whether your assumptions have been violated. If you find that your regression assumptions about residuals are violated, you may not be able to rely on your model. Let's discuss this R-squared measure in a little more detail. R-squared tries to capture how well the line or code that we'd fit using regression represents the underlying data. You can imagine that R-squared is a percentage measure of how much of the variants in the underlying data is captured by the line. Higher values of R-squared indicates that most of the variance is captured. Lower values of R-squared indicates that the line really doesn't represent the data well. An intuitive formula for the R-squared score of your regression model is explained variance upon total variance, the total variance that exists in the data and how much of that variance is explained by our regression model. As we've discussed, a higher value for R-squared indicates that a lot of the underlying variance is captured by the line or the code that we fit. We've got a better fit model. If you've used many features to train your regression model, you might find that rather than using R-squared, the Adjusted-R-squared is a better measure for evaluating your model. Intuitively, Adjusted-R-squared seeks to see if the additional features that you've added to your regression model are significant. It's possible for you to perform kitchen sink regression where you take a bunch of features and throw them all at your machine learning model, but you might have irrelevant features in there. Adjusted-R-squared will increase if irrelevant variables are deleted. Adjusted-R-squared is calculated from the original R-squared score, and you multiply it with a penalty for adding irrelevant variables. Thus, Adjusted-R-squared is a modified version of R-squared that has been adjusted for the number of predictors in the model. The Adjusted-R-squared increases only if every new term improves the model more than would be expected by chance, and finally, to evaluate your regression model, you can use statistical measures such as F-statistic and T-statistics. Both of these statistical measures are based on hypothesis testing. Conceptually here is what hypothesis testing means. You assume that two populations are the same; there is no difference between them, or you assume that a new factor is not significant. This is your null hypothesis. Your alternative hypothesis will claim that the two populations are different, the new factor is significant, and you try and perform hypothesis testing to prove or disprove the null hypothesis. That's all you need to know about hypothesis testing in order to understand regression t-statistics versus the F-statistic. Both of these are measures that evaluate how significant something is. There is one t-statistic calculated for each regression coefficient, and this will tell you whether individual coefficients are significant or not. There is one F-statistic for the regression model as a whole, and this tells you whether the regression model as a whole is significant or not. The null hypothesis for the t-statistic measure is that the corresponding coefficient value is 0. It doesn't really mean anything. The null hypothesis for the F-statistic, that is for the regression model as a whole basically states that all coefficient values of the regression are equal to 0. T-statistics are used to evaluate the utility or significance of that specific variable in a model, a specific feature. The F-statistic is what you'll use to evaluate the overall quality of the model, not individual coefficients. Now t-statistics are widely used. When you perform regression analysis, you'll often look at the t-statistics of individual coefficients to see if your individual features make sense. The F-statistic is relatively rarely used. You'll tend to use the R-squared or Adjusted-R-square instead.

Types of Classification

After having studied your data and your problem statement, you've figured out that a classification model is what you need to build and train. Before that you need to understand the type of classification that you need to perform. You've probably worked with binary classification. Output can be one of two categories, yes or no, male or female, true or false. When your classification model can predict more than two categories, that is you have multiple categories at the output, that is multi-class classification such as the classification of digit images. The image can be any digit from 0 through 9. Multi-label classification models predict more than one label at the output. So multiple categories can be assigned to the same record, true, as well as female; false, as well as female. So the output is a tuple of multiple binary variables, which may not be disjoint. And this brings us to the last type of classification task, the multi-output classification. This is a combination of multi-class and multi-label. The output here is in the form of a tuple. Multiple labels can be applied to the output, and each label can have more than two classes or categories. When you're working with machine learning algorithms, you'll find that many classification algorithms are inherently binary in nature such as logistic regression or support vector machines. Logistic regression fits an S curve on your data to calculate the probabilities of two classes; support vector machines draw a hyperplane through your data to divide them into two classes. These classifiers can be extended to perform multi-class classification as well. So inherently binary classifiers can be generalized for multi-class classification. There are other classification algorithms that are inherently multi-class such as the Naive Bayes model, which is based on Bayes' theorem of conditional probabilities. How would you extend binary classification algorithms to perform multi-class classification? Let's talk in terms of multi-class digit classification where you want to classify digits as 0, 1, 2, all the way up to 9. You can extend binary classification to multi-class using the one-versus-all technique. You'll then train 10 binary classifiers. The first one will check whether a digit is zero or not 0. A second classifier will check whether a digit is 1 or not 1, and a third one will check whether a digit is 2 or not 2, and so on. Once you've trained 10 classifiers, one corresponding to each class, the predicted label of your classifier will be the output of that detector or classifier which has the highest score. If five-versus-all has the highest score compared to other classifiers, five is the predicted label. Another way to extend binary classification to multi-class classification is to train one-versus-one classifiers. For multi-class digit classification this means you'll have to train 45 binary classifiers. How did I get to 45? Well, you'll need one classifier for each pair of digits. Think of these digits as playing tennis with one another. For each pair, you'll see which pair wins, 0 versus 1, 0 versus 2, 0 versus 3, and you'll have 1 versus 2, 1 versus 3, and so on. If you have N labels, you'll need N multiplied by N-1 divided by 2 detectors or classifiers. How did I get this? Well, this is combinations from back in high school. You don't have to worry about the actual math. So for N classes of digits, it's 10 multiplied by 9, divided by 2. That gives us 45 binary classifiers, and the predicted label is the output of that digit that wins the most duels or tennis matches.

Choosing Classification Algorithms

You've figured out that you want to perform classification, but how do you choose the right classification algorithm? That's what we'll discuss here. If your dataset is less than 100, 000 samples, there are different classification models that you can choose from amongst the traditional ML models. You may choose to perform classification using the linear support vector classifier. This of course works just with linearly separable data so you need to have data where you can fit a straight line or plane through the data in order to separate your classes or categories. If you feel that the linear support vector classifier is not giving you good results, and you're working with text data, you might choose to go with another algorithm. The Gaussian Naive Bayes algorithm, which works on the Bayes' theorem of conditional probabilities might work well here. If it's not text data that you're working with, and you've seen that the support vector classifier has not given you good results, you could use K-nearest neighbors to perform classification. The K-nearest neighbors algorithm groups your data based on what points are close together using a distance measure such as Euclidian distance. For a new data point, it'll see what other point this particular point is close to and categorize that new point accordingly. If you feel that the K-nearest neighbor algorithm does not work well, you might go for a support vector classifier with a more complex kernel such as the RBF kernel. As we discussed before, the RBF kernel uses the kernel trick allowing you to work with even non-separable data like the one you see here on screen. Often you might end up working with data that is not linearly separable. You can't really draw a line through all of the points and separate the two categories, the red points and the blue points. Such data is not linearly separable. When your data is in this format, you can make it easier to work with by performing a few smart transformations. You can transform this data using the kernel trick in such a way that the data is linearly separable. A smart transformation that might work in this example here is we have our original data. We simply calculate x squared. We square all of the original data points to get data, which is linearly separable. Thus using the kernel trick, which works with non-linear support vector machines, you can apply smart transformations to make your data easier to work with. If you feel that these classifications don't work well with your data, you can choose to go with an ensemble model. Ensemble classifiers train many individual predictors whose results are then combined to give the final prediction of the ensemble. If you're working on very large datasets, you might find that the traditional ML models don't really work well. You might choose to go with deep learning, or you could use the Stochastic Gradient classifier. Another option that you have here is to use kernel approximations on your data. This is similar to the kernel trick, but it's an actual preprocessing step, which applies kernel approximations to simplify your data and you can fit a simpler classification model. If you're interested in learning more about these classification techniques, how they work, and implementing them in a hands-on manner, Building Classification Models with Scikit-Learn is a course that you might find interesting.

Evaluating Classifiers

You've built and trained your classification model, how do you evaluate it, how do you know that it's a good one? Let's consider this binary classifier here. This is the all-is-well binary classifier. You feed in medical reports to this classification model, and it always classifies these medical reports as normal. It says there is no cancer present. A common measure to evaluate your classifier is the accuracy of the classifier. The accuracy of this model that we have here, especially when it's detecting very rare kinds of cancer might be very high. The accuracy of the model might be close to a hundred-percent, but is this a good model for detecting cancer? Let's see in just a bit. Now accuracy is not really a great metric to evaluate all kinds of classification models. Sometimes your dataset might be such that some labels are much more common or rarer than others. Such a dataset is set to be skewed, and when you train your classification model on such skewed data, using accuracy to evaluate the model is a poor metric. This means that you have to go for measures that go beyond accuracy, and we are going to set up something called a confusion matrix here to discuss other such measures. The confusion matrix is simply a grid representation of actual labels from the training data versus the predicted output of your classification model. Here we have actual labels along the rows and predicted labels along the columns. We pass in a bunch of problem instances to our machine learning model and we've categorized all of the outputs, actual labels versus predicted labels. Here are a few instances where our medical diagnostics model correctly predicted cancer. Here actual label is equal to predicted label. These are true positives or TP. At this grid here at the bottom left are instances where there was no cancer present actually, but our model still said cancer. These are false positives. Here at the bottom right are instances where our model's predictions were correct. These are true negatives. There was no cancer present and no cancer was detected. Actual label is equal to predicted label. And finally, the last grid here in this confusion matrix represents false negatives. Actual label is not equal to predicted label. Our model said no cancer, but cancer was actually present. All correct and wrong predictions from our model are succinctly summarized here in this confusion matrix, and we can use this confusion matrix to evaluate our classification model. The accuracy of our model is a measure of how many of our models' predictions were correct, where actual label is equal to predicted label. So you can say that the formula for accuracy looks like this. True positives plus true negative upon the total number of instances. In our example here, our model has an accuracy of 99.12 %. Now accuracy is not a good metric to evaluate whether this model performs well. If we want a medical diagnostics model that detects cancer well, this is not really a good model and you'll see why in just a bit. We'll move on to another metric to evaluate that classifier, and that is precision. Precision can be thought of as the accuracy of this model when our classifier flags cancer. Of the positive identifications of cancer made by this model, how many were actually correct? That's what precision tries to measure. Here is the formula for precision and by this measure, you can see that the precision of this model is just 66.67 %. One in three cases of cancer went undetected. Another measure that you can use to evaluate your classifier is to use recall. Recall is the accuracy of this model when cancer is actually present. Of all of the positive cancer cases in your data, how many was this model able to identify? Using the formula for recall, you can see that the recall of this particular model is about 71.42 %, two out of seven cancer cases were missed. Now that you understood the three main metrics used to evaluate classifiers, you need to choose the right one for your problem statement and your use case.

Clustering Models

We can now move on to discussing clustering algorithms. Clustering is an unsupervised learning technique that you can use to find patterns or logical groupings in your data, and clustering works with unlabeled data. It's an unsupervised learning algorithm. Let's say you've set up a very popular social media site and you want to figure out what kind of users you have on the site. You have a set of points, each representing a user. Now you may not have any tags or labels associated with these users, but you want to identify patterns, which is why you might choose clustering. Clustering techniques will allow you to group your users in such a way such that all users who belong to the same group are similar to one another, and users who belong to different groups are different from one another. How are these users similar to one another? How are they different? Well, that depends on the features that you'll use to feed into your clustering model. It's quite possible with a different set of features, you get a different clustering of your users. These are the implementation characteristics of your algorithm. The idea behind clustering is users in the same group should be similar and users in different groups should be different. As the CEO of this social media site, you might find it useful to figure out which users like the same kind of music, which users have gone to the same high school, which users have kids around the same age. All of these are potential clusters for your users. When you group your users in this manner, this gives you important information about your users, which is potentially monetizable. You might want to target a certain kind of advertisement to certain users in a cluster. Sport lovers see sports-related advertisements. Music lovers might see updates for the latest music concert. In order to group your users into clusters, you have to measure the distance between users. Users here are your data points and there are different distance measures that you can use based on your clustering algorithm. The distance between users in a cluster indicates how similar these users are. A measure of a good clustering algorithm is one which maximizes intra-cluster similarity. Users or data points in a cluster are as close to each other as possible. You have dense clusters. The distance between users in different clusters is a measure of how different the users are when they are separated by a cluster. The objective of clustering is to minimize intra-cluster similarity, and this is what clustering algorithms seek to achieve. Entities in the same group or cluster should be very similar, and entities which are placed in different groups are very different. Clustering techniques differ in how they are implemented and the distance measures that they use to evaluate the distance between data points, but their objective remains the same. There are a number of different clustering algorithms available and based on your dataset and the kind of problem that you're looking to solve, one might work better than the others. Here is a 3x3 grid. We are going to evaluate clustering algorithms based on the size of the data that we have to work with and the number of clusters that we want to find in our data. Number of clusters is the number of logical groupings. If you're working with a very large dataset and you want to find many groupings or many clusters within this data, Birch Agglomerative clustering works well. The Birch algorithm also detects and removes outliers in your data. Agglomerative clustering is a kind of hierarchical clustering where bottoms-up clusters are merged together to form larger clusters. If the size of the data that you're working with is fairly small and you want to find many clusters within this data, you might choose to go with mean-shift or Affinity Propagation clustering. Both of these techniques work well even when cluster sizes are uneven and you have a manifold shaped to your data. Your data is twisted and rolled in a higher dimension. Mean-shift clustering uses pair-wise distances between points to calculate clusters. Affinity propagation does not require you to specify the number of clusters upfront. It's different from other clustering techniques in this regard. If you have a fairly large dataset and you're looking for a moderate number of clusters within this data, K-means clustering and DBSCAN clustering techniques work well. K-means, in fact, is one of the most popular clustering algorithms out there, because of how scalable it is. K-means works with even cluster sizes and flat surfaces. DBSCAN works with uneven cluster sizes and manifold data; that is data which is non-linear in nature. If you have medium-sized data and just a few clusters that you want to find in this data, Spectral clustering is what you'll go with. Spectral clustering works in a fairly straightforward manner; it's simple to implement, it gives you intuitive results for data exploration. The cluster sizes that you get are fairly even and works with manifold data as well. If you're interested in clustering algorithms and learning more, the course that you ought to watch on Pluralsight is Building Clustering Models with Scikit-Learn.

The Curse of Dimensionality

In this clip we'll discuss another unsupervised learning technique, that is dimensionality reduction. Why do we need to reduce the dimensionality of our input data? Well, because of the curse of dimensionality. Let's see what I mean. We'll start our discussion with a very simple machine learning model. It takes in just one x variable. The weight of an individual, this is passed into a regression model that has been trained on a corpus of data and this regression model predicts the individual's height. Now this is too simple and maybe this didn't give you good results so you'll add more predictors, you'll add more x variables. Here is the same model with two x variables. The weight of an individual and the average height of the parents of the individual. So far, things seem fairly manageable. It's possible for you to get a good model with this data, but let's say you were to pass in a video clip to your classification model and use it for face classification. This is when you have dimensionality explosion. A video clip can be thought of as a sequence of images, and if you have a very high-resolution video, this can lead to many, many pixels that this classification model has to process to perform face classification. This can get out of control and you may not get a model that performs well and really are all of those pixels in the input image or video important? Maybe only a few are. What you now face when building and training your model is the curse of dimensionality. As the number of x variables or features that you need to train your model grows, several problems arise. As a practitioner of machine learning, you'll find that working with very high dimensionality data is a curse and there are several problems that you could encounter; problems in visualization, problems in the training process of your model, and problems when you use this model for prediction. Let's discuss each of these in a little more detail starting with problems that you might encounter when you visualize your data. Now before you dive right in to fitting a machine learning model on your data, it's important that you understand the kind of data that you're working with, and for this you need to perform Exploratory Data Analysis or EDA. This is an essential precursor to model building. Exploratory data analysis is useful for many different things. It can help you identify outliers that might exist in your data, detect anomalies, and can also help you choose the functional form of relationships. Based on your data, should you fit a linear model or should you go with a more complex polynomial model? Even if your data has many features, two-dimensional visualizations are powerful aids exploratory data analysis. Even three-dimensional data makes things hard to meaningful visualize. If you're working with data of very high dimensionality, such data is often imperfectly explored before fitting an ML model. So there is no guarantee that you've understood your data well. If you haven't really understood and explored your data, it's quite possible that you might inadvertently miss out on something important. Let's move on to discussing problems that you might encounter in training your ML models when you're working with very high dimensional datasets. The process of training your machine learning model, whether it's a traditional ML algorithm or a neural network, is the process of finding the best model parameters and complex models such as neural networks have thousands of parameter values. With complex data, it's likely that you need complex models to identify the relationships that exist, but if you have high-dimensional data where many of the parameters are useless, well, that does not give you a good model. With high-dimensional data, it's quite possible that many of the input features are useless or noise. If you train your ML model for too few epochs for too little time, this will lead to a bad model, which doesn't work well with your data. Higher dimensionality data requires more time for training. Now the number of parameters which you need to find, grows rapidly with dimensionality, and training your model becomes extremely time consuming. It becomes a very heavy-duty process, your model can't be trained quickly, and if you're working with cloud platforms to train your machine learning models, this can also get very, very expensive, because cloud platforms to pay for the time you use the resources. And finally, working with data of very high dimensionality leads to problems in prediction as well. What is prediction? Once you have a trained machine learning model, prediction involves finding training instances similar to the test instance. That is the problem instance that has just come in. When dimensionality increases, the search space for your model to find training instances similar to the test instance explodes. Also, when your data has a large number of x variables, many, many dimensions, there is a higher risk of over-fitting your model on the training data. An over-fitted model is one that does not work well in prediction. This is a model that has a low training error, but a very high test error. An over-fitted model does very well on the training data, but performs poorly in the real world when it works with test instances that it hasn't encountered before.

Dimensionality Reduction Techniques

So far, we've made the case for why dimensionality reduction of your input features is needed. In this clip, we'll talk about how this can be achieved. You can reduce the complexity of your data using two broad categories of techniques. Feature selection to select the best features to train your model, and dimensionality reduction. Let's talk about feature selection first. Here are examples of three techniques that you could choose to select significant features to train your model: variance thresholding, ANOVA analysis, mutual information regression. All of these techniques look through your x variables and features and find those features that are the most highly correlated in some way with the y variable or the dependent variable. You can reduce complexity in your data using dimensionality reduction, using three broad categories of techniques: projection, manifold learning, and auto-encoding. Feature selection involves choosing a subset of the x variables that you have. So you have an original set of variables, you'll choose just a few. You're not changing your x variables in any way. Whereas with dimensionality reduction techniques, you're actually transforming your original x variables onto new dimensions so they capture more information in the underlying data. Projection techniques are typically used with linear data, data that can be expressed in the form of linear combinations of components. Projection involves finding new better axes and then re-orienting your data along those axes. Projection techniques that you can use to reduce the dimensionality of linear data include principal components analysis, factor analysis, linear discriminate analysis, quadratic discriminate analysis, and so on. These work well with linear data. If you have non-linear data, you can use the kernel trick to transform your data to then use linear projection techniques. If you're working with non-linear data, data that cannot be expressed as linear combinations of components, you'll choose to reduce dimensionality using manifold learning. Manifold learning techniques unroll the data so that the twists and turns that exist in the data are smoothed out, making the data easy to work with. Manifold learning works best when the data lies around a rolled-up surface. Examples of data made up of rolled-up surfaces is a Swiss Roll or an S-curve. Examples of manifold learning techniques, which perform this unrolling and dimensionality reduction are multi-dimensional scaling, Isomap, locally linear embedding, kernel principal components analysis. And finally, if you're working with very high-dimensional data such as images or videos, you might choose auto-encoding to find latent features in your data. We'll build a neural network to simplify the data. Auto-encoders are built using neural networks.

Module Summary

And with this we come to the very end of this module on choosing the right machine learning solution. We started this module off by discussing broad problem categories and the kind of machine learning model that you'd choose to work with these kinds of problems. This module focused on helping you choose and evaluate the right machine learning model for regression, classification, clustering, and dimensionality reduction. We spoke about each kind of solution in a lot of detail. We discussed the assumptions of regression models, the different kinds of regression algorithms, and we discussed how you can evaluate your regression models using measures such as R-squared and Adjusted-R-squared. We then moved on to classification models. We discussed a number of different kinds of classifiers. We discussed different classification algorithms, and we saw how we could use accuracy, precision, and recall to evaluate classifiers. We then turned our attention to clustering as an unsupervised learning technique. We saw how the objective of clustering algorithms was to minimize intra-cluster similarity and maximize intra-cluster similarity. We discussed how high dimensionality input data is a curse in many ways when we are building and training ML models, and we discussed a variety of dimensionality reduction techniques. In the next module, we'll get hands-on and put the concepts that we've studied so far to practice. We'll see how we can build simple machine learning solutions.

Building Simple Machine Learning Solutions

Module Overview

Hi, and welcome to this module on Building Common Machine Learning Solutions. This module is a hands-on one, and we'll be using Python 3 to build and train our ML models. This module will apply in practice all of the concepts that we studied in earlier models. We'll start off by building and evaluating regression models analytically, as well as using machine learning techniques. We'll then move on to classification models. We'll build and evaluate classification models using accuracy, precision, and recall. From supervised learning techniques, we'll move on to implementing unsupervised learning with unlabeled data. We'll perform clustering using the k-means clustering algorithm. We'll try out a different number of clusters on our data and evaluate our model using the Silhouette score. From clustering we'll move on to dimensionality reduction on linear data. We'll see how we can perform Principal Component Analysis to find principal components that capture maximum variance in the underlying data. And finally, we'll round out this module by seeing how we can use manifold learning to perform dimensionality reduction on non-linear data.

Install and Set Up

Let's quickly explore the Python version that we're going to be working with and the environment which we're going to be using to write our demos. Here I am at the terminal window of my machine. The Python version that I'm working with is 3.7 .2 ; that's the latest version at the time of this recording. We'll write all of our code in Jupyter Notebooks and the Jupyter Notebook version that we're using here is 4.4 .0. If you don't have Jupyter installed, you can get it from the jupyter.org site. This is a browser-based shell where you can write interactive code in Python. We'll use the pip package manager to install any packages or models that we need, and the pip version that we're working with is 19.1 .1. I'll now invoke the Jupyter Notebook command to start running the Jupyter server with the Python 3 kernel. The server is available at this URL here. I'll copy this URL and paste it onto a new browser window, and this brings up Jupyter on my browser. Here is my current working directory where I'll put in my code notebooks. All of the datasets that you need for demos of this course I have within this dataset folder. Let's go back to the current working directory and click on the New drop-down here and bring up a Python 3 notebook. This Jupyter Notebook is currently untitled. I'm going to click on the name and rename it to be more meaningful. Because this is a regression problem that we're going to solve first, I'll call this regression. For all of the notebooks so that we more room to work with, I'm going to select the View option and toggle the toolbar, as well as the header. So for the remaining notebooks, I'll be doing exactly the same thing. Go ahead and make sure that you have the scikit-learn library available on your machine. We'll be using a lot of estimator objects and other models from this library.

Exploring the Regression Dataset

In this demo we'll work with the advertising dataset to perform regression using analytical, as well as machine learning techniques. In addition to scikit-learn, we'll work with other common data science libraries as well: pandas, NumPy, matplotlib. The scikit-learn version that we're working with is 0.21 .2. If you have a newer version and something doesn't work exactly like in the demo, you might want to check for version incompatibilities. For our regression model, we'll work with a simple and fairly interesting dataset. This is the advertising dataset available at this original source on Kaggle. The advertising dataset contains TV, radio, and newspaper spends for a particular company. This is a spend in thousands of dollars over a period of a month, and we have the corresponding sales of that company for each month as well, and we'll use the advertising spend to predict its sales. As you can see, it's a fairly simple and small dataset with about 200 records. If you want a quick statistical overview of what the data looks like, you can use the describe functional available in your Pandas DataFrame. This will give you information such as the mean, median, standard deviation, min, max, and percentile values for all of the numeric features in your data. Now we feel that there exists relationship between advertising and sales. I'll first explore the relationship between advertising spend on newspapers versus sales using a scatter plot, and this scatter plot seems to indicate there exists a linear relationship between spend on newspapers versus sales. Let's explore another relationship, advertising spend on the radio versus the sales of that company, and the scatter plot shows you that this linear relationship here is actually very marked. Increasing the advertising spend on radio seems to imply that sales numbers will increase correspondingly. Let's explore this same relationship, TV spends versus sales, and here the relationship is even more stark. Clearly, all three of our input features are good features to train a regression model. Let's see whether there exists any correlation between the variables in our data. The core function in a Pandas DataFrame will give you a correlation matrix. correlation values are between -1 and 1, and they are a measure of a relationship between two variables. A correlation of 1 indicates perfect positive correlation; every variable is perfectly positively correlated with itself. You can see that the advertising spends on the different kinds of media are also positively correlated with one another. Radio spends are correlated with newspaper spends, newspaper spends are slightly correlated with TV spends. Instead of working with raw numbers, if you want a visual representation of your correlation data, you can use a heatmap visualization, and here is what the heatmap looks like. The color of the cells indicate the correlation values.

Simple Regression Using Analytical and Machine Learning Techniques

We are now finally ready to perform a regression analysis using the linear regression estimator object available in scikit-learn. We'll use train\_test\_split to split our data into training data and the test subsets. We'll start off with simple regression using just one X feature, advertising spend on TV. We'll reshape the values in this column to be in the form of a 2D array where a second dimension has just one element. This is the format of the data expected by our linear regression estimator object. The Y value or the target values that we want to predict is the sales numbers for the company. We're using just one X feature to predict our target sales, and here is the shape of the arrays that we have set up. We'll use the train\_test\_split function available in scikit-learn to split our data into training data and the test subsets. Thirty-percent of the data will be test data that we'll use to evaluate the model. This is data that our model will not see during training. This is how we know that our model will be robust. We'll have the training data, which is 70% of our corpus. Those are the instances we'll use to train our regression model. We have 140 records to train our model, and we have the remaining 60 records to see if the model that we've built is a robust one. Before we use our linear regression estimator object, that is perform regression using machine learning, we'll perform regression analytically by fitting a formula, and this API is available in the StatsModels library. If you don't have StatsModels installed, you can get it with a simple pip install. We'll fit a regression formula with an intercept. Sm.add\_constant will add a constant column to our training data, which corresponds to the intersect. The OLS object in stats\_models performs Ordinary Least Squares regression, and it does so analytically by fitting a formula not using machine learning techniques. So instantiate this Ordinary Least Squares regression model, call fit on our training data, and print out a summary of the regression analysis that it has performed. Regression models are evaluated using the R-square measure. R-square gives you how much of the variance of the underlying data has been captured by a regression. Here the R-square is 0.555 and the Adjusted-R-square is 0.552. The Adjusted-R-square is a modified form of the R-square measure that has been adjusted for the number of predictors that you have in the model. The Adjusted-R-square increases only if a new predictor that you've added improves the model more than expected by chance. We are now ready to perform regression using machine learning. We'll instantiate the linear regression estimator object from scikit-learn, and we instantiate this estimator by specifying normalize=True as an input argument. Setting this to true will center our X data around 0 as a mean. So it'll subtract the mean from all of our X predictor values and divide by the L2 norm. Invoking the fit function on our estimator will start training our linear regression model. Once we have a fully trained model, this won't take more than a couple of seconds to run. Let's score this model to find the R-square score on the training data, and you can see that the R-square here is 0.55, exactly the same R-square that we got when we performed regression analytically. Now that we have a fully trained machine learning model, we can use it for prediction on the test data. Remember the test data is comprised of instances the model hasn't seen before. A robust model should have a good R-square score on the training data, as well as on the test data. Let's compute the R-square score on the test data, and you can see it's 0.72. The R-square on the test data is better than the R-square on the training data; this model is a robust one. Now scikit-learn does not have a built-in function to calculate the adjusted R-square score, but we can compute the score ourselves using the code that you see here on screen. Right now we have just a single predictor in our model, so Adjusted-R-square should be very close to R-square. Remember, this is adjusted for number of predictors in the model, and this score will increase only if the new predictor is actually meaningful. You don't really need to know the mathematical formula for Adjusted-R-square, you can simply accept it as is. In order to compute the Adjusted-R-square score, we pass in the R-square score of the test data and the test features and label, and the Adjusted-R-square for this model is 0.72, which is again, quite good. I'm now going to view how well the regression line fits on the underlying data so we'll plot the test data points and the predicted values from our model using matplotlib, and you can see here visually and using the R-square score that this model that we have built is a good one.

Multiple Regression Using Analytical and Machine Learning Techniques

So far what we performed was simple regression using just one predictor or X variable; let's perform multiple regressions. Our X variables include all of the columns, except sales; that is TV, radio, and newspaper spends, and our Y variable is the sales numbers. Once again, we'll split our X and Y values into training data and the test subsets. We have three features in our training data. That's what's different in this multiple regression. And the test data that we'll use to evaluate our model also contains the same three features. Let's fit an analytical regression model using the stats\_model API. We'll use a constant to our X training data so we perform a regression analysis with intercept. Once we've set up the object that performs Ordinary Least Squares regression, call fit on the model to fit on your data, and let's print out the summary of this model and see how it performs. With three features rather than just one, our R-square has shot up to 0.90, and our Adjusted-R-square is also high, indicating that each of the features of predictors that we added were meaningful. Let's perform regression using machine learning. We'll fit our linear regression estimator object on our training data exactly like we did before, and once we have a trained model, we'll print out the R-square score of this model on the training data. R-square is 90.4 %, the same value that we got when we performed regression using analytic techniques. How did this model do on data it hasn't seen before? Let's use this model for prediction on the test data and print out the test R-square score, and that's 0.87. The R-square score for the test data is very close to the R-square score for the training data. This is a good, robust model. Let's calculate the Adjusted-R-square score and that's 0.86. The new predictors that were added to this model, newspaper and radio spends, were both meaningful.

Exploring the Classification Dataset

In this demo we'll see how we can build a simple classification model using scikit-learn estimators. We'll write code for this model in a new notebook named classification. The dataset that we'll work with is the diabetes dataset that is available at this original URL here at Kaggle. This dataset contains help-related information about women who belong to a certain tribe and we'll use these characteristics to predict whether the woman has been diagnosed with diabetes or not. The test column here contains the target label we'll build our model to predict, whether there was a positive identification of diabetes or not. Let's take a look at the shape of this data frame. There are Azure Automation total of 392 records in this data frame. The describe function will give me a quick statistical overview of the numeric features that exist in my data. We have a number of different bits of information about the women here; how many times they were pregnant, their glucose levels, their insulin levels, and so on. You can see that the means and standard deviations of all of these features are very, very different. These numeric values are all at very different scales. We'll need to preprocess this data a little bit before we can fit our classification model. The first thing that we'll do is to label and code our target labels. We'll instantiate a label encoder and call fit\_transform on the test column. The test column contains the target labels in string format. We want to convert them to numeric identifiers so that we can use them in our model. Machine learning models can't work with strings directly; you have to encode strings in some numeric format before you feed them into ML models. The identifier 0 indicates that the diabetes test was negative, one indicates positive. The classes property on a label encoder will tell us what classes were converted to numeric integers. Negative corresponds to 0, positive corresponds to 1. This is an array which represents the order of the numeric identifiers. Let's see a scatter plot of glucose levels versus the diabetes test. Let's see if there is a relationship between the two. Now the test is a categorical variable, it can be positive or negative. There is a definite relationship here. People with diabetes seem to have higher glucose levels. Let's explore some of the relationships that exist between the X variables as well, age and insulin levels, and this scatter plot shows a clear linear relationship. There is a correlation between these two variables. Older individuals seem to have higher insulin levels. As we've discussed before, the correlation matrix is a quick way to get an idea of how the different variables in your dataset are correlated. You can see here that glucose and insulin levels are strongly correlated with a correlation value of 0.58, the diabetes pedigree function and age are just slightly correlated. Correlation value of 0.08. As before, let's view this correlation matrix information in the form of a heatmap. There is something interesting here. Observe that age and number of pregnancies are strongly correlated. This is something to be expected. If you're older, you're likely to have more pregnancies behind you.

Classification Using Logistic Regression

Now that we've understood what our data looks like, let's set up our training features and target labels. The features include all of the columns, except for the test column. That's what we'll try and predict. If you remember, the means and standard deviations of all of these numeric features were very different. We'll apply standardization to center all of these numeric features around the mean and express them in terms of standard deviations, or Z scores. Standardization scales all of the data to be centered around the mean, and it's a very common preprocessing technique used with numeric features. Fit\_transform of the standard\_scaler estimator object standardizes our data and returns a NumPy array. Here is the shape of the array. Let's convert this to a Pandas DataFrame so that it's easy to work with. We set up a new data frame of scaled features; the columns are the same as the original features data frame. I'll now use the describe function to explore this standardized dataset, and you can see here that all of the mean values across all numeric features are very close to 0, and standard deviations are close to 1. Standardization scales features to center around 0 and have unit variance. I'm going to create a single data frame with all of our standardized numeric features and the test column as well, and I'm going to write this data frame out to a CSV file. I'll call the CSV file PimaIndians\_processed.csv. Later on when we use this dataset for other demos, we can use the processed version of this file directly. Run an ls command under the dataset folder, and here is our processed.csv file available there for us to use. With that done, we are ready to set up the X features and target labels to fit our classification model. I'm going to use the train\_test\_split function to split our X and Y data into training sets and test subsets. Seventy-percent of our dataset will be used to train our model, that is 274 records, and the remaining 118 records we'll use to evaluate our model. The classification model that we'll fit on this data is the LogisticRegression classifier, which fits an S-curve on the underlying data and classifies categories based on the threshold. The l2 penalty is a regularization technique that we can apply on the model to prevent our model from over-fitting on the training data, and the solver equal to liblinear is simply the optimization algorithm that logistic regression will use under the hood. Classifier.fit will start the training process, and this will take just a few seconds on this small dataset. Let's now use this classifier for prediction. Here are the predicted values from the test data. Predictions are whether an individual has been diagnosed with diabetes or not. Before we use objective measures to evaluate this model's predictions, let's set up a simple data frame, which compares the values from our test dataset, that is actual labels versus predicted labels. This is just a data frame that I set up for quick comparison. You can see that our model got several predictions wrong on the test data, but there were other predictions that were correct as well. We can't really objectively evaluate our model just by looking at the data. So let's use a few classification metrics. We'll evaluate our model using accuracy, precision, and recall scores. The input to all of these coding functions are the actual values from your dataset and the predicted values from your model. Once the scores have been calculated, we'll print them out to screen and we'll see how good this model is. The accuracy of this model is roughly 72%. The accuracy tries to measure how many of our models' classifications were actually correct, whether they were positive identifications of diabetes or negative identifications. The precision score of this model is 62.5 %. Precision is a measure of how many of this model's positive classifications or identifications of diabetes were correct. Precision focuses only on those records where the model predicted diabetes, and the recall of this model is 48%. How many of the individuals with diabetes across our entire dataset were correctly classified by this model? This is what recall tries to measure.

Classification Using Decision Trees

Let's focus on classification for a little more time. The previous classifier model that we built was a logistic regression classifier. This time we'll build a decision tree classifier to train on the same data. Under the hood, the decision tree looks at your training data and tries to build a tree-like structure which it can use for predictions. We have specified that the max depth of our decision tree should be just 4. This is a design constraint or a hyperparameter that you can specify on your decision tree. Call fit on your training data to fit the decision tree model. As you can see here, decision trees have a number of different hyperparameters that you can tune. The discussion of each of these is beyond the scope of this particular course. Now we have a trained classification model, let's fit it on the test data and we'll see how this model compares with the previous one that we built using logistic regression. I'll set up the same data frame that I did before, which has the actual labels from the test data, along with the predicted labels from our model. It seems like our model did well, but we won't have an objective measure of how this model compares to the previous one until we specifically calculate its accuracy, precision, and recall scores. Let's print them out to screen and you can see that on all fronts, this model seems to have performed a little better than our logistic regression classifier. Accuracy is 73%, precision is 64%, but the biggest improvement is in the recall score, which is 53%. The previous model had a recall of just around 48%. You can view the actual labels versus predictor values from our model in the form of a confusion matrix using the pd.crosstab function. You can see that this confusion matrix is a 2x2 grid. This is a binary classification problem after all. Along the row you have predicted results from the model, and along the columns we have the actual results from the test data. I'll now calculate manually the accuracy, precision, and recall scores of this model and store it in variables. The variable TP holds the true positives. Predictions from our model were that the person has diabetes and the person actually did have diabetes. TN represents two negatives, the predictions from our model was the person does not have diabetes and in fact, she did not. Here are predictions our model got wrong. False positives, our model said a person had diabetes, but she actually did not, and we have false negatives. Our model was unable to detect diabetes when it was actually present. The accuracy score of our model is TP + TN upon the total number of predictions made by our model, and this verifies at 73%. The precision score is the number of positive identifications of diabetes made by our model upon the total number of positive identifications and that is 64%. And finally, we have the recall of the model, how many of the actual diabetics in our dataset were identified correctly, and that matches too at 53.6 %.

Clustering Using K-means

In this demo we'll build a machine learning model using unsupervised learning techniques. We'll build a clustering model using k-means clustering. We'll write code for this in a new notebook named Clustering. The dataset that we'll work with is the drivers dataset, and the original source of this dataset is here at this URL. This dataset contains information about car or truck drivers. These are the drivers used to make deliveries. Each driver has a unique id and has a distance score and a speeding score. How fast he drives and how far he drives. We'll try and find logical groupings that exist in this data using clustering to see which drivers are similar to other drivers. This dataset is a fairly large one. There are a total of 4000 records corresponding to 4000 different drivers. In order to ensure that our model doesn't inadvertently pick up patterns in the ordering of our data, I'm going to shuffle this dataset by invoking the sample function in Pandas. Frac is equal to 1 will ensure that we preserve all of the original records in this shuffling. we'll go ahead and drop the column for driver id from our dataset. It doesn't really hold any information for our clustering model so we are left with the distance feature and the speeding feature. Our data happens to be in two dimensions. This means that we can visualize this data using matplotlib. I'm going to plot a scatter plot of the different drivers and their distance and speeding features, and you can see at a glance that there are logical groupings present in this data. The dense clusters indicate that there are groups of drivers who are similar to one another and can be used for say the same tasks. We'll perform clustering using the k-means clustering algorithm, and this is available in the form of an estimator in scikit-learn. Instantiate the estimator object. K-means clustering requires that you specify the number of clusters that you want to find in your data as a hyperparameter or a design parameter. We're looking for four clusters in our data and we'll iterate for 1000 iterations. Once clustering is complete, the estimator will assign a cluster identifier to each of your data points, and that's available here using this labels property. You can see that the data points have been assigned clusters 1, 2, or 3. If you want to see the number of unique clusters that were identified, you can invoke the np.unique function on your labels and you can see that there are four clusters as we had specified. Let's create a list of tuples where we associate every data point from our original records with the corresponding cluster label. The zipped function in Python will allow us to create these tuples where we have the driver's data and the corresponding labels from our clustering algorithm. Let's take a look at same of this zipped list. You can see our original data in the form of a list and the corresponding cluster label, and this is actually useful when we want to visualize this data. If you want to know the coordinates of the cluster centroids as found by this k-means clustering model, you can access the cluster\_centers property. Four clusters correspond to four centroids. The centroids in your k-means clustering model need not be actual data points from your dataset. We are now ready to visualize the clusters that our k-means clustering model found. We'll display the data points in each of the clusters using a different color from this colors list. Once again, we'll represent our original drivers' data points in the form of a scatter plot, but this time each of the data points will be in a different color based on the cluster that it belongs to. In addition to the data points, I'll also plot the cluster centroids in the form of square markers. In addition, I will also annotate each of the cluster centers with its numeric identifiers. Let's take a look at the resulting visualization. You can see the four clusters that the k-means algorithm discovered in our data. Some of these are dense clusters, some of these are sparse clusters, and every cluster has its centroid more or less at the center of the cluster. The clustering algorithm seems to have done well, but how do you objectively evaluate it? Well, you can use the Silhouette score. The Silhouette score takes in the actual data points and the corresponding cluster labels, and then tries to calculate how similar an object is compared with other objects in its own cluster, that is a measure of cohesion, and how different that same object is from objects in other clusters. That is a measure of separation. Higher values of Silhouette scores are better. Let's take a look at the Silhouette score for this dataset when we have four clusters and the Silhouette score is 0.59, which is pretty decent. Let's try k-means clustering on the same dataset once again, but this time we'll change the number of clusters we want to find in our data. I'll set n\_clusters equal to 3. The k-means estimator will now look for three clusters in the underlying data and the unique labels are 0, 1, and 2. Let's set up the zipped list as before and associate every data point with its corresponding cluster so we have a list of tuples with this information, and let's access the centroids of the clusters that we found and placed them in the centroids variable. We can now visualize these clusters as we did before using matplotlib, and you can see in this visualization that we found three clusters in our dataset. All of the data points to the left of this graph, which were originally in two clusters, have been merged into a single cluster. But did our clustering improve with just three clusters instead of four? Well, the Silhouette score will give us this information and the Silhouette score here is 0.82, a great improvement over the previous one. This clustering is clearly better. Before we wind up this clustering demo, I'm curious; let's try clustering with n\_clusters equal to 2. We call fit on our training data, and if you look at the unique values of cluster labels, you'll see we have just two clusters identified by 0 and 1. As before, create a zipped list of the original data point and the cluster labels, we'll use the zipped list to visualize our clusters in matplotlib. Store the cluster centers in the centroids variable, and here are the two centroids for our two clusters. Let's visualize this in matplotlib. The different cluster data points will be in different clusters. Here are the two clusters that k-means clustering found in our dataset. What about the Silhouette score of this clustering with two clusters? And you can see that the Silhouette score is almost 85%. Two clusters is better than 3, which is better than 4 based on the Silhouette score.

Dimensionality Reduction Using Principal Component Analysis

In this demo we'll see how we can perform dimensionality reduction on linear data using Principal Components Analysis. Linear data, because this is data that can be expressed as a linear combination of basis vectors. Here we are on a new notebook and we'll work with the diabetes dataset. This is a dataset that we've already preprocessed and it's available for us to use. All of the numeric features have been standardized to have a center at 0 and unit variance. The features that we'll work with to perform dimensionality reduction are all of columns, except the last one. Here are the features. We'll perform dimensionality reduction on these features and then train a classification model. The dimensionality reduction technique that we'll use here is Principal Components Analysis. PCA will re-express the original features in our dataset in terms of its principal components. Each principal component is oriented in such a way that it captures the maximum variance in the underlying data. In order to perform dimensionality reduction using Principal Components Analysis, I'll use this helper function called apply\_pca, which takes as its input argument n, where n specifies the number of principal components that we want to use to train our model. Instantiate the PCA estimator object and specify n\_components = n that we passed in, and call fit\_transform on the features of the diabetes dataset. X\_new will be an array with n principal components where we specify n as an input argument. We'll return the pca estimator object, as well as x\_new in the form of a data frame. There are eight features in our original data. I'm going to apply Principal Components Analysis to find 8 principal components. The number of principal components is equal to the number of dimensions in your original data. I want all eight principal components. Once we have all eight principal components, we can select how many of them we want to use to train our model. The Explained Variance ratio property on the pca estimator object that was returned after fitting on our data will give us how much of the variance in the underlying data is explained by each of its principal components. You can see that for our dataset, the first four principal components capture almost 90% of the variance in the original data, and the remaining principal components account for about 10% of the variance. All of the principal components together should capture all of the variance in the original data so the sum of the explained variance ratio should be equal to 1. Let's visualize the cumulative sum of the explained\_variance\_ratio using matplotlib. Cumulative sum graphs are always upwards loading, but this gives you the important information that if you fit a model using just four or five principal components, that will capture almost 90% of the variance in your original dataset. You don't need to use all eight components or all eight original features. We are now ready to train our classification model using principal components. The Y labels are available in the test column. I'll perform dimensionality on the X features, and I'll use just four components. Four principal components to train my classification model. And here are the four principal components. The numbers won't really make much sense, because they are basically our original features projected onto principal components axes. I'll now use the train\_test function from scikit-learn to split these principal components into training data and the corresponding test data, and we'll fit a logistic regression model. Let's see how this model compares with the model that we fit on all of the original features. Once the model has been trained, let's use it for prediction on the test data. Once we have predicted values from our model, we can calculate the accuracy, precision, and recall scores for this model. Let's see the accuracy and it's 78%. Dimensionality reduction seems to have improved our model's performance. We used fewer features, but those features were expressed in terms of principal components capturing as much variance in the underlying data as possible. Let's take a look at the precision score next, and that has improved as well, 63%. And finally, here is the recall score of this classification model and that is at 60%.

Dimensionality Reduction Using Manifold Learning

In this demo we'll see how we can perform dimensionality reduction on non-linear data using manifold learning techniques. Non-linear data can be thought of as data that is twisted and turned and curved in higher dimensional space. It's data that can't be expressed in terms of linear combinations of basis vectors. We'll write our code on a brand-new notebook here and we'll work with non-linear data in three dimensions, which is why we need 3D plotting tools in matplotlib. Matplotlib has some helper utility functions, which allow us to generate artificial datasets of different types. Here we're using the make\_swiss\_roll function, which will generate non-linear data, which is shaped in the form of a Swiss roll. N\_samples = 2000 means we'll generate 2000 data points which make up the Swiss roll. Let's take a look at the shape of the data and the color variable. This dataset has three dimensions, which means we have three coordinates to express a data point. Observe that the color variable here is a one-D vector. This represents the univariate position of a particular sample according to the main dimension of points in the manifold. Points which are close to one another will have similar color. The Swiss roll is a manifold. That's because it's a simpler shape in two dimensions, it's essentially a carpet that has been folded and rolled up in three dimensions. And we'll see exactly this when we visualize these data points. Set up the X data in the form of a data frame, and let's plot it in 3D using matplotlib. This is a 3D scatter plot as you can see here. We'll also color the individual data points based on its value in the color variable, and here is what the Swiss roll data looks like. Observe that this is essentially a 2D carpet that has been folded up in three dimensions. Observe that the points that are close to one another have the same color. You can imagine that points which are close to one another are points which belong to the same category or class. In order to reduce the dimensionality of such non-linear data, we'll apply manifold learning techniques, and I'm going to set up a helper function for this called apply\_manifold\_learning. The input arguments are the data points in three dimensions and the method that we'll use for manifold learning. Method.fit\_transform will convert the original data points to two dimensions. We'll print out the shape of the new data, it should have reduced dimensionality, and we'll plot this reduced dimensionality data now expressed in two dimensions using a scatter plot. We'll use the original color variable to color these data points as well, and we'll return the manifold\_learning estimator object from this helper function. Now that we have this helper function set up, we can try out a number of different manifold learning techniques starting with MDS or multi-dimensional scaling. This technique reduces the dimensionality of the original data while trying to preserve the distances between the original points. We want the original data to be expressed using two dimensions, which is why n\_components is equal to 2. There are two flavors of this multi-dimensional scaling algorithm. What we're applying here is the non-metric version. The exact details of the differences between metric and non-metric is beyond the scope of this course. You can see that the reduced dimensionality data is expressed using two dimensions, and if you look at this visualization, you can see that non-metric mbs did not really do a good job. All of the points of the same color are scattered everywhere. Points that were originally close together in higher dimensions are now far apart. This is not really a good manifold learning technique for this data. Let's try metric multi-dimensional scaling and see if this technique performs better. If you look at the unrolled dataset, you can see that it's a lot better. The differences between the categories have been preserved in lower dimensions. Let's try another manifold learning technique to reduce dimensionality. This is locally linear embedding. Under the hood, this uses a nearest neighbor's algorithm. It measures how each instance relates to its closest neighbors and tries to find a lower dimensionality representation, which preserves these local relationships. We'll reduce our original 3D data to two dimensions, and we'll use 15 neighbors in the nearest neighbors algorithm. We'll first look at the standard flavor of the locally linear embedding algorithm, and when we look at the data in its reduced dimensionality form, you'll see that the carpet was unrolled pretty well. Points that were close together in higher dimensionality are close together in two dimensions as well, but this lower dimensionality representation is distorted and standard LLE tends to do that. Let's look at the LocallyLinearEmbedding algorithm, but use a different technique. This is the Hessian technique, which is more performant and gives better results. And here is the reduced dimensionality representation, our Swiss roll almost perfectly unrolled. The Hessian LLE method also eliminated the distortion that was present in standard LLE.

Module Summary

And with this demo, we come to the very end of this module on Building Common Machine Learning Solutions. We started this module off in a hands-on manner and ended it hands-on. We started with regression models. We saw how we could build and evaluate regression models analytically, as well as using machine learning techniques. We evaluated the model that we trained using R-square, as well as Adjusted-R-square. We then trained a classification model on the diabetes dataset and used it to predict whether an individual is likely to have diabetes or not. We evaluated our model using accuracy, precision, as well as recall scores. We then worked with the drivers dataset and performed clustering using k-means clustering. We tried a different number of clusters with this model and evaluated each model using the Silhouette score. We then saw how dimensionality reduction works as an unsupervised learning technique. We performed Principal Components Analysis to reduce the dimensionality of our input data. And finally, we implemented and visualized dimensionality reduction on non-linear data using manifold learning. In the next module, we'll discuss the various processes involved in a machine learning workflow. We'll also discuss the importance of ensemble techniques and discuss different kinds of neural networks.

Designing Machine Learning Workflows

Module Overview

Hi, and welcome to this module on Designing Machine Learning Workflows. Now we've discussed the different kinds of machine learning problems. You know how to choose the right machine learning solution for your use case. Now we'll discuss the machine learning workflow. At a high level, all of your machine learning models are trained using this exact same workflow. We'll examine the machine learning workflow end to end using a very interesting flow chart. We'll start off with raw data, preprocessing, choosing an algorithm, fitting a model, evaluating it, and how all of these processes have to come together in an iterative manner. We'll then talk about the different choices that you have to prototype and train your model. We'll talk about local, distributed, and cloud-based training and prediction using a case study of running PyTorch in the cloud. We'll then move on to discussing ensemble learning. We'll see what ensemble learning is all about, and how we can use this to build more robust models. And finally, we'll round off this module by discussing the most popular neural network designs, dense neural networks, convolutional neural networks for image data, and recurrent neural networks for sequential data.

The Machine Learning Workflow

You now have the tools that you need to compare and contrast different machine learning solutions and pick the right one for your use case. In this clip, we'll discuss the components of the machine learning workflow. This is the workflow that you'll use to build and train any machine learning model. Machine learning involves lots of intermediate steps and this is what a big picture of the workflow looks like. Now don't be intimidated when you look at this. We'll discuss each individual step and see where it fits in, in the context of the larger workflow. Machine learning starts with data collection. Now data is available in a wide variety of sources, and it may be present somewhere. It may be present in your data warehouse. You may have to scrape it up from the web. You first have to figure out what data you have to work with, and you'll have to figure out how to get this raw data to a place which you can access, and that takes us to the next step where you prepare your data. The first step in data preparation is to load and store the data where you can access it easily. You may have this data on the cloud somewhere. It may be on a distributed cluster of machines in your local data center. You'll need to bring all of this data together and preprocess your data so that it's in a form that is fit to feed into a machine learning model. Data cleaning and preprocessing is an important part of building robust models. You have to have the right kind of data to train your model. This step might involve detecting and removing outliers and anomalies that exist in your data. It might involve preprocessing your numeric features so that they are centered around 0. You might perform feature selection and dimensionality reduction. You might encode text data into numeric form. All of this makes up data preprocessing. Once you've worked with and processed your data, you might need to perform exploratory data analysis and choose the right algorithm for your use case. Your choice of algorithm might depend on the kind of model that you want to fit, whether it's a regression or classification model. How much data do you have to work with? How many features exist in each record? If it's complex, unstructured data, you might choose to go with neural network models. Once you've chosen your algorithm, you'll feed your data in to train and fit your model. Fitting a model is essentially just building your model once it has been trained on the data. Now of course with unsupervised learning, you may not have an explicit training process. You just feed your data in. Once you have a fully trained model that you can use for prediction, the workflow and process doesn't end there. You need to validate and evaluate your model to see if it's a good one. There are a variety of validation techniques that you can choose from. You can use cross-validation to find the best possible model where you break up your data into different subsets and train your model in different subsets of this data. You'll then need objective measures to evaluate your model. Accuracy, precision, or recall for classification models, R-square and Adjusted-R-square for regression models. With clustering models you might use the Silhouette score. There are other measures available as well. You'll score your model using these evaluative measures, and you'll see whether this model meets your expectations or not. Now if you're not satisfied with the model, you need to go back to the training process. You need to update your model. Maybe you'll change a hyperparameter, you'll change the design of your neural network. Maybe you'll try and get more data that you need to train your model. All of this part of model update. You'll then fit a new model and start this entire process once again. Get a fully trained model, validate that it's a robust one, score your model, check whether you're satisfied or not. This process continues until you're finally satisfied with your model, and once you're satisfied with this model, this is the model that you'll deploy to production and use for prediction. Now with machine learning, since it's constantly learning from data, the workflow doesn't end here. As new data for prediction comes in, you'll use this to update and improve your model parameters. There are of course nuances involved in every step, but this is the basic machine learning workflow.

Case Study: PyTorch on the Cloud

There are several techniques that you can use to prototype and train your machine learning models, and we'll study those techniques using a case study. PyTorch on the Cloud. What is PyTorch? It's one of the most popular deep learning frameworks out there. This is a deep learning framework that makes it very easy for you to prototype your neural network models and allows for fast, flexible experimentation. PyTorch is extremely popular because of how deeply it is tied to the Python programming language, the language of choice for data scientists. The approach in PyTorch is similar to NumPy and scikit-learn. You can create neural networks using Python and you can use existing Python libraries and debuggers. Before we go on to our PyTorch case study, let's talk about the training options that you have available for training any machine learning model. As you explore your data and seek to build an ML prototype, you'll likely start on your local machine and perform local training. This is the starter option which works perfectly well for prototyping, provided that you're not working with a very huge dataset and your model fits within the memory of your machine, but if you're training on the local machine, you cannot leverage GPUs or graphics processing units. You might be familiar with GPUs already. These are specialized chips with highly parallel architecture that makes them an order of magnitude faster than CPUs for training deep learning applications. If you want to move beyond prototyping, you'll find that your local machine will not really work well. You might run into out-of-memory issues. You just don't have the resources to handle data and training, and it might take hours and hours to train even simple models. Another option that you could choose to train your models is a local datacenter. The datacenter will likely have clusters of machines that you can use for distributed training. Machine learning frameworks such TensorFlow and PyTorch have built-in distributed training capabilities. You'll leverage these to run training on a cluster of machines. You'll run training for more epochs to build better models, but performance of your model will dramatically rise. When you're working on your local datacenter, you're actually working on real hardware, which means that scaling is a problem. If you need to scale your model to use additional machines, you might need to buy or purchase additional hardware. Also, the administrative overhead of managing the distributed hardware and distributed framework falls on you. Another alternative to perform distributed training for your models is to train on the cloud. On-cloud training is by far the most preferred option. Cloud training gives you pay-as-you-go functionality. You only pay for resources that you use. Cloud training is elastic and scalable. If you need more machines, you don't have to go out and buy them; you simply spin them up on the cloud. In fact, when you work on the popular cloud platforms such as AWS, Azure, and the GCP, your distributed training framework will take care of spinning additional instances for you during training. Cloud hardware also has built-in functionality for CUDA and GPU support. CUDA is simply the programming API that you use to work with GPUs. And finally, all cloud platforms have their own native support for distributed training using popular deep learning framework. This platform support makes distributed training very straightforward. Here are the choices that you have to make when building, training, and using your model for prediction. You have to figure out how you'll train your model and how you'll host it for prediction. Now training can happen locally; that's when you prototype your model or in a distributed manner. When you host your model for prediction, you can have it work in batch mode or online mode. Batch mode involves using your model for predictions on long-running jobs that work on huge amounts of data. Within online mode, this is a real-time prediction with your model. In the case study that we'll discuss next of running PyTorch in the cloud, we'll talk about distributed computing on the cloud. We'll specifically discuss how you can build and train PyTorch models on the cloud on the three most popular cloud platforms: AWS, Azure, and the GCP. You'll find that each of these cloud platforms offer similar solutions to prototype and train your models. Here are the broad machine learning solutions available for building and training PyTorch models on each of these cloud platforms. We'll now look at each of these solutions in turn, discuss what they are, and how they can be used. Notebooks are used to prototype your machine learning models. These are similar to the Jupyter Notebooks that you'll use locally, but they are cloud-hosted Python notebooks. Now these notebooks could be platform-agnostic, that is you're simply running Jupyter Notebook on the cloud, or they can be platform-specific. In the case of GCP, you have a platform-specific notebook called Datalab. On each of these platforms you can train your PyTorch models using a deep learning virtual machine. This is an incredibly powerful VM on the cloud embedded with GPU support. For each cloud platform, this deep learning VM is a cloud-specific virtual machine instance. It's EC2 on AWS, Google Compute Engine on the GCP. You can choose to enable these deep learning VMs with one or more GPUs and run distributed PyTorch training. If you don't want to work on a single virtual machine, you want to run distributed training on a cluster of machines, you'll use estimator objects on AWS, as well as Azure. At the time of this recording, GCP does not support a PyTorch estimator. GCP is optimized for distributed training for TensorFlow, the other popular open source deep learning framework. These estimators are a high-level, easy-to-use API. AWS has its own estimator, Azure has its own separate estimator. These are specific to the cloud platform. These help build, train, and deploy PyTorch models. They spin up machines on which you can run distributed training, the machines are automatically shut down when you're done using them. Estimators make it simple and intuitive to run distributed training on a cluster of machines.

Ensemble Learning

Some of the best neural networks built today for all kinds of use cases, regression, classification, you name it, are ensemble machine learning models. Ensemble machine learning models have also been the recipient of several prizes and competitions. They often come first. So what exactly is ensemble learning? What does is this magic? Now ensemble learning is not really a different algorithm per se. It's a machine learning technique in which several learners are combined to obtain a better performance than any of the learners individually. Ensemble learning models seek to channel the wisdom of the crowds. Several learners together will give us better predictions than one learner. One learner might be very limited in what it can do. An ensemble refers to a number of different learners, which are trained on the same dataset with a few nuances. We'll discuss those in just a bit, and together they give us a better performance. Let's say you have some training data and you want to build a classification model. Your training data is available, you want to build the best possible classifier. What you could do is use ensemble learning and train several different classification models, and each of these classification models should be as diverse and different as possible. You could train a logistic regression classifier, a Naive Bayes classifier, a random forest, maybe support vector machine classifier. Each of these will have their own biases, own strengths and weakness, but together they can give you better predictions than a single model. What we have here is an ensemble. Now there are important questions that need to be answered when you're using ensemble learning. The first question is, what kind of individual learners or predictors to use? The next important question is we have our training data, how should these individual learners be trained? Should we use all of the training data, some of it? Should we have some other constraint? Each of these individual learners will of course make their own predictions, whether for regression, classification, or any other kind of model. The third important question that you should answer when you're building an ensemble is how should the individual learners be combined? How will you combine their predictions? We'll discuss each of these questions in depth as we go through this module and through this course. Let's start with a quick overview. Let's discuss what kind of individual learners we should use. When you're building an ensemble learning model, the individual learners or predictors in your model can be absolutely of any type. There is no restriction on the algorithm or the individual models, but your ensemble as a whole will perform better when each learner is as different as possible from other learners. Now there a number of different machine learning algorithms to solve the problems, say classification. Each algorithm has its own strengths and weaknesses, and when you have different algorithms as individual learners, their strengths and weaknesses will offset each other and you'll get a better predictor overall. Now you'll find that for most ensemble techniques, even in our demos, decision trees are the model that we use the most often. When you build and ensemble model where the individual learners are decision trees, an ensemble of decision trees is what gives you a random forest. Why do you think it is that decision trees are the preferred individual learner in an ensemble? Why is that the random forest ensemble is so popular? That's because random forests make it very easy to build uncorrelated learners. When you use ensemble learning models, you're not working with just two or three predictors; you might have 100, 200, five-hundred predictors. If you want each individual predictor to be as different as possible in the ensemble, decision trees help make that easy, and this is why in the real world, ensemble learners are often random forests of decision trees. Now let's move on to answering the next question in our series, how should these individual learners be trained? There are a number of different choices available to you here. Now if your individual learners are very different, there are different algorithms, then each learner can be trained on the entire dataset because they learn different things from that dataset, but if you're building your ensemble with similar learners, let's say it's an ensemble of decision trees, that is a random forest, in order to have diverse predictors using the same algorithm, each model is often trained on a subset of the training data, and this subset is randomly selected. So each model looks at different random samples in the training data. In fact, you can take this further and have each individual model train on random sets of features. Thus each learner will learn different things from your data, and they will be as diverse as possible. We can now move on to answering the third question here, how should the predictions of individual learners be combined? Let's talk about this in terms of a classification model. You have a new problem instance, that is your prediction instance come in, and you have diverse predictors within your ensemble that make predictions on this problem instance. Now each of your diverse predictors might have a different classification. The classification of the models that make up your ensemble are the individual learners' predictions. And the final prediction of the ensemble has to combine these individual learners' predictions in some meaningful way, and there are a number of different techniques that you can use for this. If you're working with a classification model, you can use hard voting. The final prediction of the ensemble would be the majority vote of the individual learners in the ensemble. For a hypothetical instance, let's say the predictions from our individual learner, most of them were true. If this was a majority of the individual predictors, the ensemble would also predict true. An alternative to hard voting is soft voting. Soft voting can be used with classification, as well as with regression models. This is when the final prediction of the ensemble is a probability-weighted average of the individual learner's predictions, or if you want to get even fancier, you can build a model to combine predictions from your individual learners and this is called stacking. You'll train an additional model to combine predictions from the individual learners' outputs.

Averaging and Boosting, Voting and Stacking

Now that we've understood what ensemble learning is all about, let's discuss a few ensemble learning techniques. We discussed in detail in the last clip the important questions that you need to answer when you're setting up an ensemble learning model. One of those questions was, how should individual learners be trained? There are many answers to this question. We'll discuss two specific techniques, averaging and boosting. When you use averaging techniques to build an ensemble model, the individual learners or predictors in your model are trained in parallel, and the final prediction of the ensemble is the average scores of the individual predictors. You create diverse individual learners here by training your individual models on different samples of your training data. And the two most commonly used techniques to sample training data when you're working with averaging models is bagging and tasting. Another way to train ensemble models is to use boosting. Here the individual predictors in your ensemble are trained in sequence. They're actually built in sequence where each predictor learns from mistakes made by earlier predictors. This means that there exists a dependency between learners in a boosting model, the learners that are applied in a sequence depend on previous learners. There are two boosting techniques that are very popular. They work on the same principle, their implementation is different. Adaptive boosting or add a boost and gradient boosting. Let's now discuss two different ensemble techniques in the context of how should individual learners be combined. There are two ways you could approach this; voting and stacking. Voting can be either hard voting or soft voting. With hard voting, the majority vote of the individual predictors is the final prediction of the ensemble. When you use soft voting instead of hard voting, the weighted average of the individual predictors give you the final output of the ensemble. The individual learners in an ensemble can be combined in more interesting ways as well. You could use stacking or stacked generalization where you'd use another machine learning model on the individual predictions to get the final prediction of the ensemble. This additional ML model is often referred to as the blender or the meta learner.

Custom Neural Networks: Their Characteristics and Applications

In order to round out our discussion of different machine learning techniques, in this clip, we'll discuss the different kinds of neural network models and the use cases for those models. We've discussed earlier that neural networks are the most popular class of deep learning algorithms, and neural networks can be customized to be of many different types. What we'll discuss here are just three broad categories. The first of these is the dense, fully connected neural network where a neuron in one layer is connected to all neurons in the next layer and all neurons in the previous layer. Then we have convolution and neural networks, which are sparse neural networks. They're not made up of dense connections, and they work well with image data. And then you have recurrent neural networks when neurons have additional state or memory. Recurrent neural networks work very well with sequential of time-series data such as text. Let's discuss the characteristics of each of these neural network designs so that you understand when you would choose to use each. Dense neural networks work very well when your dataset contains mostly just numeric features. That is your data is not made up of text or image data. With dense neural networks, you can easily build traditional classification and regression models. Dense neural networks are often referred to as fade-forward neural networks where the layers of your neural network are arranged in a sequence and the input is fed forward through them. These neural networks are considered to be dense because all neurons in one layer are connected to all neurons in the previous, as well as the next layers. Now of course you can customize your dense neural network with different kinds of activations by using dropout to mitigate over-fitting and a bunch of other details. However, on the whole, dense neural networks are what I'd call general-purpose neural networks for traditional classification and regression models. Let's move on to discussing convolutional neural networks. These neural networks specialize in working with image data. Images as you know are two-dimensional; they have a height and a width, and convolutional neural networks are modeled to perceive image data in two dimensions. Convolutional neural networks have been designed to mimic the visual cortex of the brain. Individual neurons in our eye perceive only local regions of images and these are then combined together and aggregated by the visual cortex. Convolutional neural networks do something similar. Convolutional neural networks are sparse neural networks, which means that for a particular neuron in a layer, it's not connected to all neurons in the previous or the next layer. Convolutional neural networks are made up of two kinds of layers. Convolutional layers are used for feature detection. To detect objects and features in the input image, pooling layers are used for sub sampling of inputs. Pooling layers reduce the memory that you need to train your neural network model, which is pretty important. Pooling layers also add location in variance to the feature detection by convolutional layers. Let's move on to discussing the third popular neural network architecture, recurrent neural networks, which specializes in learning from sequential data such as text data or time-series data. Recurrent neural networks are widely used in natural language processing applications. The neurons are the active learning units in an RNN are different. They have additional memory or state that allows them to remember sequences. A recurrent neural network essentially comprises of a memory cell that is unrolled through time to learn from time-series or sequential data. Neural network layers represent instances in time.

Module Summary

And this brings us to the very end of this module on Designing Machine Learning Workflows. We started this module off by looking at a big-picture overview of the different processes involved in building and training a machine learning model. We discussed and understood every step of the machine learning workflow starting right from raw data, preprocessing, choosing an algorithm, fitting a model, validating it, evaluating it, and iterating. We then discussed the choices that you have to prototype and train your machine learning models. In this context we discussed local distributed cloud-based training and prediction. In this context, we worked with a case study running PyTorch on the cloud. We then turned our attention to ensemble learning techniques to build more robust models. We saw ensemble learning comprises of many individual learners whose predictions are then combined to give the final output of the ensemble, and finally we closed this module with a discussion of neural networks. There are many custom and popular neural network designs out there. We discussed the three most common models; dense neural networks, convolutional neural networks to work with image data, and recurrent neural networks to work with sequential text data. In the next module, we'll get hands-on once again. We'll see how we can build ensemble solutions, as well as neural network solutions.

Building Ensemble Solutions and Neural Network Solutions

Module Overview

Hi and welcome to this module on Building Ensemble Learning and Neural Network Solutions. We've understood the concepts in an earlier module; in this module we'll put them to practice once again. This module will be completely hands-on. We'll implement a number of different ensemble learning techniques using the scikit-learn library, starting off with a classification model that uses the voting classifier ensemble. We'll use this classifier to perform hard voting and soft voting. We'll then build ensemble models using averaging techniques. We'll use bagging or Bootstrap aggregation and pasting to train the individual learners of our model on different samples of the training data. We'll also see an example of a boosting technique. We'll use gradient boosting to build a regression model, and we'll see how the different learners in a sequence are fit on the residual errors of the previous learners. And finally, to cap it all, we'll build a regression model using the PyTorch deep learning framework.

Classification Using Hard Voting and Soft Voting

In this demo, we'll build a classification model using ensemble learning techniques, and the ensemble that we'll use is the voting classifier from scikit-learn. We'll use this voting classifier to perform hard voting where it computes the majority vote to make its final prediction, as well as soft voting where it averages the probability scores for the individual categories and then predicts that category with the highest probability score. We'll work with the dataset that we're familiar with. This is the diabetes dataset, which we've preprocessed and stored out to file. All of the numeric features in this data have been standardized. I'll now set up the features and target labels for our classification models. The X data frame contains all of the features, the Y data frame the test column that is our target labels. I'll use the train\_test\_split function to split the data into the training subset and the test subset. We'll use 80% of our data to train our model and 20% of the data to evaluate that classifier. The voting classifier ensemble is available as a built-in estimator object in scikit-learn. The individual learners, the classification models that we'll train on our training data is the logistic regression classifier, a support vector classifier, and a classifier that uses Gaussian Naive Bayes. In an ensemble it's important that the individual models or learners are as diverse as possible and that they're trained using different algorithms so that they learn different patterns from the underlying data. I'll now instantiate each of these classifier estimator objects. Here is the LogisticRegression classifier, here is the support vector classifier that uses a linear kernel, and here is the Gaussian Naive Bayes classifier that performs classification using Bayes' theorem of conditional probabilities. Once the individual learners have been created, we can set up the voting classifier estimator object. The input arguments to this voting classifier is the three learners whose individual predictions the voting classifier will aggregate, and you also need to specify the kind of voting that you want this classifier to perform. Hard voting basically means that the final prediction of this ensemble will simply be the majority vote of the predictions of the individual learners. If a majority of the individual learners say diabetes is present, the output of the ensemble will be it is present. With our estimator instantiated, let's fit on the training data to start training this ensemble learner. You can see the three individual learners that exist within this ensemble and their corresponding design parameters or hyperparameters. Once the ensemble has been trained, let's use it for prediction on the test data. The final prediction of the voting classifier will be the majority vote of the individual predictors, and let's calculate the accuracy score of this model on the test data, and you can see that it is 85% almost. If like me you're curious about the accuracies of the individual predictors in this model, let's run a little for loop to figure out what they are. So I'm going to iterate through each of the individual predictors, as well as the final voting classifier. I'm going to train them on our training data and call predict on the test data, and for each of these models I'll print out the accuracy score. You can see that for our data, the logistic regression classifier did really well, accuracy score of 86%. The other classifier models, including the final ensemble, were less accurate than the logistic regression classifier. The dataset that we're working with is a toy dataset so this is not really unusual. When you're working in the real world, you'll find that ensemble techniques often perform better than the individual predictors in the ensemble. You can use the voting classifier to perform soft voting as well. Here is where the probability scores, which is the output of the individual models is aggregated for each class or output category. By default, the support vector classifier does not output class probabilities. You need to instantiate this estimator object with probability equal to true so that this estimator outputs class probabilities that the voting classifier can use for soft voting. I'll now instantiate a voting classifier once again, this time to perform soft voting. Because soft voting involves averaging the probability scores for each output class or category, you can assign weights to individual estimators. The soft voting classifier will then take a weighted average of probability scores for each output category, and the final output will be that category with the highest weighted average. This time I'll directly set up the for loop where I'll perform predictions using the individual learner, as well as the final ensemble, the soft voting classifier. I'll train the individual learners, as well as the final ensemble on the training data and call predict on the test data and we'll print out the accuracy score for the individual learners, as well as the ensemble as a whole. And here are the accuracy score results with the soft voting classifier.

Exploring and Preprocessing the Regression Dataset

In this demo, we'll build an ensemble model to perform regression using both bagging, as well as pasting techniques. We'll write code for this model on a brand-new Jupyter Notebook called BaggingPasting. So that things are a little interesting, we'll perform regression analysis using a slightly different dataset. This is the medical insurance dataset, originally available at this source on Kaggle here. This is a dataset with a number of different individuals and certain characteristics of those individuals, and we'll use these characteristics to predict their insurance charges. We have age, gender, BMI, number of children, the region in which they live, and their insurance charges. If you look at the shape of this data frame, you can see that we have roughly 1350 records to work with. You're already familiar with using statistical measures and visualization to explore your data. I'm going to go straight to the correlation matrix. I want to see the correlations that exist between the different variables in my dataset. Here is a correlation matrix for all of the numeric features that exist in our dataset. The remaining features are all categorical in nature. If you want to visualize the correlation matrix, a heatmap is what you'll choose. Let's set up a heatmap for this insurance data, and here are two interesting observations. You can see that the age of an individual is positively correlated with the insurance charges he or she pays. It's a small positive correlation, but it exists. The other observation I found interesting in this dataset is that the age of an individual and number of children is not really correlated at all. The correlation value is very close to 0. There are certain columns in our dataset that are categorical or discrete values expressed in the form of strings. We need to label and code them so that they can be identified using numeric ids. The first column that I'll transform is the region column. I'll call fit\_transform on this label encoder so that we have numeric ids to represent the different regions. Region has been label encoded with numeric ids starting from 0. There are still two other columns, sex and smoker, which are categorical. The classes property in this label encoder that we use to encode the regions will tell us the regions that were encoded and the corresponding numeric ids. The northeast was encoded using 0, northwest with 1, and so on. The sex and smoker columns contain binary categorical values so I'm going to encode them using one-hot encoding. The pd.get\_dummies function will act on the columns specified and convert them to one-hot encoded form. So the sex column has been split into two; sex\_female, sex\_male, and a value of 1 in a column indicates whether an individual's gender is male or female. Now that we've preprocessed the insurance dataset, I'm going to write it out to a file called insurance\_processed.csv. I'm going to run an ls command in the dataset folder to make sure that this file has been saved now. We can use this preprocessed data directly by reading in this file if you need it later.

Regression Using Bagging and Pasting

I'm going to read our preprocessed file in into a data frame called insurance\_data. Here is the file that we just saved out, and you can see that it has all of the original 1338 records. Let's get set up to train a machine learning model, an ensemble model. This is a regression model to predict insurance charges, so the X variables will be all of the columns other than the charges column, and the Y target values is the data that's available in the charges column. Now that we have our X and Y variables, we can use train\_test\_split to split our data into the training subset and the test subset. Eighty-percent of our data, roughly 1000 records we'll use for training, and the remaining 20%, roughly 268 records, we'll use to evaluate our ensemble. We are now ready to build our ensemble model to perform regression. The individual predictors in our model will be decision trees, and for this we'll use the DecisionTreeRegressor object. Decision tree allows us to build diverse predictors when they are trained on different samples of data. We'll train our ensemble of decision trees using the bagging regressor, which will allow us to perform both bagging, as well as pasting. Instantiate the BaggingRegressor estimator object, you'll find that we use it just like we would any other scikit-learn estimator. You need to specify the kind of individual predictor you want to use within this ensemble. You want to use decision trees and we want to use 500 of them. In order to have our individual decision tree predictors, train on samples of the training data using bagging that is sampling fit replacement, we have to set Bootstrap to True. Each of our predictors will be trained on eighty-percent of the training data. So we'll sample 80% of the samples for each individual predictor we'll sample with replacement. This is bagging. The individual predictors in an averaging model can be trained in parallel. When you specify n\_jobs=-1, as many processes as are available on your CPU will be spun up to perform the training. When you specify oob\_score=True for your bagging model, this means that you want your ensemble model to be evaluated on out-of-bag instances so whatever instances were not used to train a particular predictor, those will be used to evaluate that predictor and an out-of-bag score will be returned. This out-of-bag evaluation is essentially on samples that an individual predictor hasn't encountered during training. Let's train our bagging regression model by calling fit on the training data, wait for the training to complete, and I'm first curious about this oob\_score\_ for this bagging regressor, and you can see that the oob\_score here is 0.83. This is the R-square score of the ensemble evaluated on out-of-bag instances, and this should actually be a very good predictor of the R-square score of this ensemble on test data, and let's see exactly that. Use this regression model to call predict on the test data, and let's calculate the R-square score on the test data. You can see that this regression model is a very good one. It has an R-square score on test of 0.86, which is very close to the 0.83 that we got using out-of-bag instances. This same bagging regressor ensemble estimator object can be used to train models using pasting. You only specify a different set of input arguments. We're using the DecisionTreeRegressor once again as an individual predictor, and we'll train 500 estimators. Observe that we have specified the Bootstrap argument to be equal to false. This basically means that you want to perform sampling without replacement, which is pasting. The individual predictors will be trained by sampling 90% of the data in the dataset. The sampling will be done without replacement, essentially pasting. Let's train our pasting ensemble by calling fit on the training data and let's take a look at the results. We'll use our pasting regressor to call predict on the test data and calculate the R-square score of this model on the test data and it is 83%. This is a good model as well. Typically models built using bagging tend to perform better than models built using pasting. Bagging techniques allow the ensemble to have a higher bias and lower variance.

Regression Using Gradient Boosting

In this demo we'll perform regression using gradient boosting. Gradient boosting involves applying your individual learners in a sequence where every learner learns from the mistakes of the previous learner. We'll build a regression model using the insurance dataset that we had preprocessed and saved out to file. Read in the data, this is the data that we'll use to predict the insurance charges for a particular individual. Let's set up two different data frames, one which holds the X variables or the features that we'll use to train our model, and another which holds the target values or the Y variables, which we'll use to train our model. We'll once again use the train\_test\_split function to split our data into the training subset and the test subset. Before we use an estimator object to perform gradient boosting directly, I'm going to train a number of individual decision tree regressors. I'm going to set up a number of these decision tree regressor models in sequence where each model will learn from the residual errors of the previous model. I'll instantiate the first model in our boosting sequence, DecisionTreeRegressor with a max depth of 3, and I'll call fit on the training data. Once this individual predictor has been trained, I'm going to use this regression model to predict on the training data. This will give us the predicted values from this model, and I'm going to subtract these predicted values from the actual values that I have in the dataset. Thus the variable y2 here will hold the residual errors of this first predictor. We'll now instantiate a second predictor in our boosting sequence and this decision tree regressor will train on the residual errors of the previous predictor. Observe that we use the same X variables or features to train this model, but we'll try and have this predictor predict the residual errors of the previous model. Once we've trained this second model in a sequence, we'll use this model for prediction and calculate the residual errors of this model as well that I store in the variable y3. I'm going to add one more decision tree regressor to our boosting sequence, max\_depth=3 once again, and this will train on the residual errors of the previous model in the sequence, that is y3. These three decision tree estimators applied in a sequence with every estimator training on the residual errors of the previous one gives us a boosting ensemble. We can calculate the final prediction of this ensemble on the test data by summing up the predictions from the individual trees. So I run a for loop through each of the individual trees in our gradient boosting sequence, and I'm going to invoke predict on that tree regressor on our test data. The sum of all of these predictions will be the final prediction of our ensemble, which I'll store in y\_pred. Let's calculate the R-square score of this gradient boosting ensemble on our test data, and you can see that it's 81.8 %. That's pretty good. Applying gradient boosting in the manner that we've just described where we fit individual predictors in a sequence is quite tedious, which is why there exists the GradientBoostingRegressor, which does this for us automatically. Instantiate the GradientBoostingRegressor and specify how many estimators you want to train in your ensemble. Here I specified number of estimators is equal to 3, and the max depth of each individual decision tree is also 3. This is the same gradient boosting model that we had applied individually just a few minutes ago. All you need to do is fit this model on the training data to train your gradient boosting regressor. Let's compare this regressor with our manually applied gradient boosting. Call predict on the test data and let's calculate the R-square score. It's exactly the same as what we got by manually applying predictors in a sequence. If you want to apply gradient boosting models, you can use this regressor estimator object directly instead of training and fitting individual models.

Regression Using Neural Networks

In this demo we'll see how we can build and train a regression model using neural networks, and we'll build our neural network using the PyTorch deep learning framework. PyTorch is one of the most popular deep learning frameworks out there today because of how easy and intuitive it is to use. Install the torch libraries onto your local machine with a simple pip install. Once you have all of the PyTorch libraries downloaded, you can simply start prototyping your model. Install the torchvision libraries as well. This will give you access to utilities that you might need to train your neural network. We'll continue to use helper libraries such as Pandas and matplotlib to work with data. Read in the advertising dataset, which we have seen before. This is the dataset that contains advertising spends on TV, radio, and the newspaper. We'll use that information to predict the sales revenue of an organization. It's a fairly small dataset with just 200 records, great for building a neural network prototype. Apparently it has been found that neural networks work well where all of your numeric data are expressed using very small values and all of the features have similar scales. That is your numeric features are roughly in the same range. From sklearn, I'm going to import the preprocessing module and use this to preprocess my data. Invoking the preprocessing.scale function will standardize all of the numeric features in this dataset. All features will be centered around 0. We'll subtract the mean and divide by the standard deviation. All features will have 0 mean and unit variance. So TV spends, radio spends, and newspaper spends have all been standardized and this is the data that we'll use to train our neural network. Set up the X and Y variables in exactly the same manner as we've seen so far. Split your data into training subsets and the test subset that you'll use to evaluate the model. We'll use 160 records for training and 40 records to test our regression model. We are now ready to set up our data so that it can be fed into a neural network. Import the torch module in PyTorch. Deep learning frameworks work with data stored in tensors. Tensors are essentially just matrix representations, multi-dimensional arrays like NumPy; however, tensors can be loaded onto a GPU device to run distributed processing. So you need to convert all of your input multi-dimensional arrays to the tensor format using the torch.tensor function. I've used torch.tensor to convert the training features, as well as the target values into the tensor format. For all practical purposes, when you are prototyping your models, you can work with tensors and treat them exactly like you would multi-dimensional NumPy arrays. You can see that tensors have a shape property, allowing you to view their dimensions, exactly like NumPy arrays do. Under the hood, tensors are incredibly powerful though, and they power the distributed computing abilities of deep learning frameworks. You're now ready to design and create the neural network that we'll use to perform regression on our advertising data. Here are some variables that I have initialized. Input and output here refers to the dimensionality of the input and output layers of our neural network. We have three features corresponding to every record in our model, which is why the dimensionality of the input is equal to 3. The output of our model is a single number, that is the sales prediction for the month. This is why the dimensionality of the output layer is equal to 1. We'll keep our neural network fairly simple with just one hidden layer, and this hidden layer will have 100 neurons. The loss function that we'll use to train our model parameters is the mean square error loss, and we can instantiate this loss function using the MSELoss object available in PyTorch. The learning rate that you use during training will determine how much your model will learn from your data for every epoch. An epoch is simply one pass that your model makes through your entire dataset. I've chosen a learning date of 0.0001. Let's set up a simple fully connected neural network. We'll use the torch.nn .Sequential model, which allows us to specify layers in a sequence. The sequential object is simply a container object for other neural network layers. The first layer is an input layer. Its input dimensionality is equal to inp, that is 3, and the output dimensionality of this layer is equal to 100. Remember, our hidden layer has 100 neurons. Our input layer will be followed by ReLU activation, and you can specify this using the built-in ReLU object available in PyTorch. ReLU activation is followed by another linear layer. Its import dimensionality is 100, output dimensionality is equal to 1. This is the layer that will give us our final output. During the training process, the model parameters of our neural network need to converge to their final values, and we'll use an optimizer for this. I've instantiated the Adam optimizer here, which is very popular and works very well in the real world. The Adam optimizer will update our model's parameters during training, and it'll use the learning rate that we have specified. I'm now going to train this neural network for 10, 000 epochs and I set up this training a simple for loop. This iterates through the entire dataset ten-thousand times. We'll then make a forward pass through this neural network to get the predictions from our current models' parameters. The current predictions we'll then compare against the actual values in our training data and calculate the loss. This loss will capture how far our model's predictions are from the actual values. Every thousand iterations will print out the current value of the loss to screen so that we can see that our model is converging. Once we've calculated the current loss of this model, we'll call loss.backward, which will calculate gradients, and these gradients will allow us to update our model's parameters to improve our model. The optimizer.step function in PyTorch will actually update the model's parameters, and this process that I just described will be repeated for every iteration of this for loop. We'll get the model's predictions made using its current parameters, calculate the loss. We'll make a backward pass through the model's layers in order to calculate gradients, which we'll then use to update the model's parameters. It's Shift+Enter to start the process of training your model, and after 10, 000 iterations, training will be complete. Let's see how our model performs on the test data. Pass in the x\_test\_tensor that we had set up earlier through the model, and we'll get the predicted values and store them in y\_pred\_tensor. If you want to convert this tensor representation to a NumPy format, you can simply call the NumPy function on a tensor. The predicted values from our model are now available in the form of a NumPy array. We can now see the scatter plot representation of actual values versus predicted values to see how close they are. If you look at the resulting visualization, the shape of the graph makes it very clear that the actual values are very close to the values predicted by our model, but before we complete this demo, let's evaluate our neural network regression model by calculating its R-square score on the test data. An R-square of 94% indicates that it's a good model.

Summary and Further Study

And with this demo we come to the very end of this module on building ensemble learning and neural network solutions. The first ensemble learning technique that we implemented was the voting classifier. We built a classification model using hard voting and soft voting. We then moved on to using averaging techniques to build regression models. In this context we studied bagging and pasting to sample random training data to build the individual learners. We saw that bagging involves sampling the training data with replacement. Pasting involves sampling the training data without replacement. We then built a regression model using boosting, specifically gradient boosting, and we saw how the individual learners will fit on the residual errors of previous learners. And at the very end, we saw how we could build and train a regression model using the PyTorch framework. And with this, we come to the very end of this course on designing a machine learning model. Now as a student of machine learning, if you're interested in studying further, here are some other courses on Pluralsight that you might find interesting. Foundations of PyTorch will get you started with building and training neural networks in PyTorch. Once you have your foundations covered, you can move on to building your first PyTorch solution. If it's not neural networks that you want to study, but you're interested in feature engineering, here are some courses that you might find interesting: Building Features from Numeric Data, Building Features from Text Data, and Building Features from Image Data. And with this, it's time for me to say goodbye. I hope you had fun watching this course. Thank you for listening!