**Course Overview: Building Your First Scikit-learn Solution**

Hi, my name is Janani Ravi, and welcome to this course on Building Your First scikit-learn Solution. A little about myself, I have a master's degree in electrical engineering from Stanford and have worked at companies such as Microsoft, Google, and Flipkart. At Google, I was one of the first engineers working on real time collaborative editing in Google docs and I hold four patents for its underlying technologies. I currently work on my own startup, Loonycorn, a studio for high quality video content. In this course, you will gain the ability to identify the situations where scikit-learn is exactly the tool you are looking for, and also those situations where you need something else. First, you will learn how scikit-learn's niche is traditional machine learning, as opposed to deep learning or building neural networks. Next, you will discover how seamlessly scikit-learn integrates with core Python libraries. You will then understand the typical set of steps needed to work with models in scikit-learn. Finally, you will round out your knowledge by building your first scikit-learn regression and classification models. When you're finished with this course, you will have the skills and knowledge to identify precisely the situations when scikit-learn ought to be your tool of choice, and also how best to leverage the formidable capabilities of scikit-learn.

Exploring scikit-learn for Machine Learning

Module Overview

Hi, and welcome to this course on Building Your First scikit-learn Solution. Now this course assumes that you have no idea of what machine learning is, so we start off with a high level overview of how algorithms can be learned from data. We'll then talk about scikit-learn for data and machine learning modeling. We'll see the features and libraries that scikit-learn has to offer, which make it so popular with data scientists. Scikit-learn is an integral part of the Python data science stack, and it has a close relationship with NumPy, SciPy, Pandas, Matplotlib, and other Python libraries. In this module, we'll also understand supervised and unsupervised learning techniques in machine learning. We'll see how supervised learning requires labeled data, whereas unsupervised learning learns from patterns within the data. We also talk about scikit-learn to build traditional machine learning models, as opposed to representation-based ML models, which are built using frameworks such as TensorFlow, Keras, and other deep learning frameworks.

Prerequisites and Course Outline

Before we dive into the contents of the course, let's take a look at some of the prereqs that you need to have to make the most of your learning. You need to know basic Python programming so that you can build and train your ML models at the very end of this course. And really, that's the only requirement. This is intended to be your very first machine learning course, you need not have worked on ML before, there is no prior ML knowledge required. In this course, we'll start off with a high level overview of what exactly machine learning is all about and how algorithms can be set up to learn from data. We'll then introduce the scikit-learn library, explore what it has to offer, and talk about why it's so popular. We'll also understand the different categories of ML models, traditional and representation-based, supervised and unsupervised learning. We'll then move on to talking about the basic machine learning workflow and how the scikit-learn library fits into this workflow. And finally, we'll build and train two simple machine learning models with scikit-learn, a regression model and a classification model.

Introducing Machine Learning

With all the hype around machine learning nowadays, machine learning seems like a technology that is extremely complex, hard to understand and implement. While there is indeed some truth to the first point, you'll find that the remaining two points are not true at all, and we'll see that in this course. A machine learning model is built using an algorithm, just like other algorithms in computer science. However, it's an algorithm that is able to learn from data. The logic that you code into this algorithm is able to parse an input data set and find a structure which you can then use for predictions. A machine learning algorithm should have the ability to work with a huge maze of data. The data sets that we use to feed into such algorithms are getting larger by the day. It's not thousands, not hundred thousands, but millions and hundreds of millions of records. The machine learning algorithm is structured in such a manner that it's able to find patterns in the data that you feed in, and after those patterns have been found, the algorithm is able to make intelligent decisions. For example, you might apply a machine learning algorithm to parse all of the email received on a particular server. So the input maze of data is emails on a server and you want your algorithm to figure out whether the emails are spam or ham. Once the algorithm has figured this out, it'll make intelligent decisions about whether the email belongs in the trash or in your inbox. Another example might be your machine learning algorithm works with many, many images represented in the form of pixels. Your algorithm is then able to identify edges, colors, and shapes in these images and put them all together to figure out that this image is that of a little girl or a bird or a cat. Machine learning is a field that's constantly evolving, and you'll find that there are new types of machine learning algorithms being developed almost every day. For example, you might have heard of reinforcement learning or generative adversarial networks. But before we get to those, for a student of machine learning it's important to understand the four most common categories of machine learning problems. A type of problem that is simple and intuitive to understand is classification. Is an email spam or ham? Is this review positive or negative? Is the image of a cat, dog or a bird? Classification problems try to determine the class or category of input. Another common type of machine learning problem is regression. Regression is what you use to predict a continuous value. What is the price of an automobile given its make, model, engine size, and top speed? Regression models are built to predict a continuous value, such as price, not a discrete category. Another common use of machine learning is for clustering problems. This is where you have a large corpus of data and you want of find some logical groupings in the underlying data. You have a large number of newspaper articles, which of these articles are about sports? Which are about entertainment? Which are current affairs? These are clusters in your data. And finally, machine learning algorithms are commonly used for dimensionality reduction. When you see data in the real world, every entity has a huge number of attributes or features, as they are called. Not all of these attributes are interesting or even useful. Dimensionality reduction techniques try to find latent factors or significant features in this data so that every entity can be represented with lower dimensionality. Of all of the inputs that a stock market receives, what are the most significant factors that drive stock prices? That's an example of dimensionality reduction.

Learning from Data: Training and Prediction

Let's use an example of a model built for classification to understand what exactly machine learning is and how your algorithm can learn from data. Let's say you were asked to build a machine learning model to solve this problem, whales, are they fish or mammals? As an intelligent human, you would reason this out and say, well whales are members of the infraorder Cetacea, which means they could be mammals. On the other hand, you would also say, whales look like fish, they swim like fish, and they live with fish in the sea. They could be fish as well. What you're essentially doing is, as a human expert, you're applying some rules to the input and figuring out what the right output should be. In this case, you'll see an instance of a whale, you'll apply a few rules and make the decision that the whale is a mammal. You have built in knowledge of what rules to apply. You know what features of a whale are significant, what features you need to pay attention to, and then you can make the classification that the whale is a mammal. If you were just to code up these rules in an algorithm, which in its simplest form would be just a bunch of if statements, that would be a rule-based classifier. But if anything changes in the underlying data, you'll have to re-code your algorithm to account for the changes. There's absolutely no way for your code to keep up with the changes in the data. Instead of having human experts formulate rules, let's say you had a whole corpus of training data, instances of many, many kinds of animals, fish, and so on, which you've used to train your machine learning-based classifier. Your machine learning algorithm, based on this corpus of data available to it, has figured out the significant features of mammals and fish. To your machine learning model, you wouldn't simply pass in the term whale, you would pass in important or significant features of a whale. The fact that it breeds like a mammal and gives birth like a mammal. And your ML-based classifier will be able to correctly classify the whale as a mammal. Here, of course, I'm assuming that this is a good machine learning model. Your machine learning-based classifier is not just a series of static rules that it applies to the input, instead, you've used data to train model parameters and these trainable model parameters have picked up significant features from the data in order to make their prediction. Model parameters refer to variables within your machine learning algorithm that can be trained using data. Now let's talk about the two phases for building your ML-based classifier, or any machine learning model. When you first choose a machine learning algorithm, the algorithm has no idea how to predict things correctly. Your classifier algorithm has to be first trained by feeding in a large corpus of data where all of the data has the right classifications. So if it's review data that you want to classify into positive or negative reviews, you'll feed in a bunch of reviews, correctly marked as positive or negative. This is the training phase of your ML model and your learnable model parameters will be trained in this phase. Once your machine learning algorithm has been trained using data, what you get is a model, and this is the model that you can now use for prediction. The term machine learning model applies to machine learning algorithms when used loosely, but once an algorithm has been trained, what you get is a model and you can use this model to classify new instances that the model hasn't seen before. This is the prediction phase. Let's now get a high level understanding of how this training of your machine learning model works. You have your ML-based classifier algorithm and you feed it a corpus of data, and with whatever parameters it has at this point in time, it'll make some kind of prediction. It'll be a classification. Now this prediction, of course, initially will not be correct, and this incorrect prediction will be a feedback into your model in the form of some kind of loss function. The algorithm will look at the correct output, will see its current prediction and see, well, this is not really correct, this is lossy, and it'll use that information to improve the model parameters to make a better prediction. Now this description of how a model is trained is, of course, a 10, 000 foot view. Each algorithm performs the step a little differently. But the essential idea and objective of training remain the same. So you have a corpus of data for your specific use case, images, reviews, no matter what, and you feed this corpus into some kind of classification algorithm. That is your machine learning algorithm. And once this classification algorithm has been trained, you get your ML-based classifier model. Now that we've understood how models are trained, we can go back to this visualization that we saw earlier. We have a fully trained model, which we now use for prediction. The input that you feed into your model for prediction, or even for training, are referred to as feature vectors. They are the features or attributes of your training and prediction instances, they're also often referred to as x variables. The output of your model, that is the model's prediction, is often referred to as a label. The term label is typically used when you have a classifier model. Other terms that can be used for the output of your model are y values, or predicted values. Now let's say you have a fully trained machine learning classifier and you feed an input for a whale that looks like this. You tell the model that the whale moves like a fish and looks like a fish. Well, based on the information available, your model is very likely to classify the whale as a fish, which is clearly wrong. So this is kind of significant for traditional machine learning models. If you pass in as input a wrong choice of features, your output from the machine learning model will be incorrect, the predicted label will not be equal to the actual label.

Traditional and Representation ML Models

Now that we've got a big picture understanding of how machine learning algorithms work, let's talk about traditional and representation-based machine learning. Here's what we know so far about machine learning using a classifier as a canonical example. We feed the corpus of data to a classification algorithm and the output is an ML-based classification model. The classification algorithm here could be any machine learning algorithm that learns from data. There are a variety of classifiers, decision trees, support vector machines, Naive Bayes classifiers, all of these learn from data in different ways. There are a wide variety of algorithms to choose from and each kind of algorithm has its own strength and weaknesses. The choice of algorithm is determined by experts. Now this algorithm has to be trained using the training data that you specify, that is your problem use case. The features fed into the algorithm are also determined by experts. What we just discussed is true of traditional machine learning models. These can be regression models, such as Linear regression, Lasso, Ridge, Support Vector Regression, that is SVR. Classification models, such as Naive Bayes, Support Vector Machines, Decision trees, and Random forests are dimensionality reduction techniques for manifold learning that is curved data in higher dimensional space or factor analysis, extracting significant factors from the underlying data. Traditional machine learning models can also be used for clustering where you find logical groupings in your data. Examples are K-means clustering, DBSCAN clustering, Spectral clustering, and so on. As a student of machine learning, you first start understanding and working with traditional ML models. These models have a fundamental algorithmic structure to solve problems, and different models have different algorithmic structures. An example of an algorithmic structure would be a model could fit a curve to solve a problem, draw a line. The algorithm is then fed data, which trains the algorithms parameters, and the algorithms parameters are what we refer to as model parameters. Model parameters are the variables within a model that learn from data. So here are some examples of algorithmic structures that can be used to learn from data. You might have a traditional ML model that builds a tree-like structure and uses this tree to classify instances. This is a decision tree. Or you might have an algorithm that tries to fit a line or a curve on the underlying data and uses this line or curve to make predictions. These are examples of regression models. Or the machine learning algorithm might be structured in such a way that it applies probabilities on input data and gets output in the form of probabilities as well. This is a simplified explanation of how Naive Bayes models work. All of the characteristics that we've discussed so far in this clip talk about traditional ML-based systems. Traditional ML systems rely in experts like you to decide what features they need to pay attention to and how those features should be structured. As your input data gets larger and you want to build models for more complex use cases, you might turn to representation-based machine learning models. These machine learning models are used to solve the same kind of problems, classification, regression, clustering, and dimensionality reduction. The difference here is the fact that they don't need experts to tell them what features to pay attention to. They're capable of learning significant features from the underlying data. And an example of representation-based ML models are deep learning models, such as neural networks. In a nutshell, representation ML-based systems figure out by themselves what features to pay attention to and how to pay attention to those features. Deep learning models are the most common examples of representation machine learning-based systems. Deep learning basically refers to those algorithms that learn what features matter and the most common class of deep learning algorithms are neural networks. Neural networks are made up of active learning units called neurons, and these neurons are very simple building blocks, essentially just mathematical functions that actually learn from the data that you feed it. Neural networks are machine learning models which are structured in the form of layers, so you feed in a corpus of training data to active learning units, which are arranged in layers, where every layer receives the output of the previous layer and passes its output on to the next layer. Every layer in a neural network is made up of the active learning unit, that is neurons. And all of these layers are actually neurons interconnected with one another, and these interconnections can get quite complex. At a very, very high level, the way neural networks work is that every layer tries to extract a little bit of information from the underlying data and the information extracted by each layer is different. If you're feeding in images to a neural network, you might have one layer extracting granular information from pixels, one layer might look at edges, another layer corners, a third layer might put together object parts. With this understanding of traditional and representation-based ML models, let's quickly compare and contrast the two. In traditional ML models, features used in models are explicitly chosen by domain experts; whereas in deep learning, features are implicitly chosen by the model itself. With traditional ML models, you feed in structured data, such as numbers or probabilities. In the case of deep learning ML models, you can pass in unstructured data as well, images, movies. Similar problems can be solved using both these types of models, all of the common machine learning problem types. The scikit-learn library that we'll work with primarily supports traditional ML models, and you'll find that there are a wide range of problem specific solution techniques available. Whereas with deep learning models, neural networks are by far the most common solution technique. Each solution technique in traditional ML has its own characteristic approach. Build a tree, fit a line, fit a curve. All solution techniques in deep learning models rely on neurons and the interconnection between neurons. But those can get quite complex. As a user of a traditional machine learning model, it's possible for you to have a high-level conceptual understanding of what the model does and you have more insight into the mechanics and internals of these models. Whereas with neural networks, they typically tend to be black-box models that are hard to question and reverse-engineer. If you want to work with traditional machine learning models, the most popular library out there is scikit-learn. For deep learning ML models, you have some choice, TensorFlow, Keras, PyTorch, are all deep learning frameworks.

The Niche of scikit-learn in ML

With this big picture understanding of machine learning algorithms in mind, we are now ready to talk about scikit-learn. Scikit-learn is a Python library that is very easy to use, comprehensive, and efficient. If you want to build and train traditional machine learning models, scikit-learn is the library of choice for data scientists. Scikit-learn has been around for awhile. Scikit-learn was originally developed as a Google summer of code project back in 2007, and it currently has 30+ active contributors. It's sponsored by a number of different companies and organizations, INRIA, Google, Tinyclues, and the Python software foundation. As a student of machine learning, this is the first library that you will typically tend to use to build your ML models because it is easy to use, it's comprehensive, and it's efficient. And scikit-learn for all its ease of use is not a toy in my library, it's actually used in the real world for production models. Let's talk about each of these attractive features of scikit-learn in turns, talking with how easy it is to use. Scikit-learn offers a higher level API called the estimator API for a consistent interface to all your machine learning models. No matter what model you're building, you can use the estimator API. No matter what algorithm you choose, you work with the estimator object in the same way. You first create and instantiate a model object, you'll then invoke a method to fit this model to your training data, and once you have a fully trained model, you will use this estimator object to predict for new data. Individual estimator objects can then be combined and chained together for complex operations. Estimator APIs exist, not just for machine learning models, but also for the preprocessing of your data. We'll now move on to talking about how comprehensive the scikit-learn library is. You'll find that when you work with scikit-learn, all common families of models are supported. There are tree-based models, probability-based models, support vector machines-based models. There are a range of models for regression, classification, dimensionality reduction, and clustering. In addition to machine learning algorithms that you can use to build models, scikit-learn also has a huge library of data preprocessing, cleaning, feature selection, and extraction functions. All of these are techniques that you'll use to wrangle data before you feed it into train a model. And finally, once you've trained a model, there are libraries available in scikit-learn for model validation and evaluation. These are the libraries that you'll use to see how good your model really is. When we explore the scikit-learn documentation and webpage, you'll see that all categories of models are supported. In addition, you'll see that feature extraction and selection can be performed using statistical techniques, as well as dimensionality reduction techniques. All of these techniques will help you get rid of noisy features so that your model can be more robust. In addition, scikit-learn offers a wide variety of preprocessing techniques, such as standardization and normalization for numeric data. Machine learning models don't really work with any kind of data and these techniques help get the right kind of data to feed your model. In addition, scikit-learn has a number of data generation libraries for you to generate artificial data sets that you can work with. You can generate complex nonlinear data in the form of Swiss rolls or S-curves. You also have a number of real world data sets available within scikit-learn. Once you've built your machine learning model, there are techniques that you use to evaluate whether your model is robust. Cross-validation is one such technique, and scikit-learn has libraries to perform cross-validation on your models. When you work with machine learning models, you'll find that there are parameters which are part of the design or construction of your model. These are referred to as hyperparameters. How do you know your model is designed right? Well, you need to perform hyperparameter tuning on your data. And there are libraries available for this as well. Another attractive feature of scikit-learn is that it's actually a very efficient library and very performant. And why is that? Because the implementation of individual machine learning models are highly optimized. Engineers have really worked hard to get the best out of these models. Scikit-learn is an integral part of the Python data science stack. It's built on top of the SciPy library, which is why it has the scikit prefix. And you'll find that it's very easy to work with scikit-learn and all other Python data science libraries. Here are some of the most popular libraries in the Python data science stack, and you'll find that scikit-learn works with all of these. We have NumPy, which is an n-dimensional array package. We have SciPy, which is a fundamental library for scientific computing. If you're visualizing your data, Matplotlib is what you'll tend to use for both 2D and 3D plotting. Sympy is a symbolic mathematics library. And finally, we have Pandas, which allow you to represent data in a tabular format and analyze data. All of these Python libraries interoperate with scikit-learn and scikit-learn is built using NumPy, SciPy, Matplotlib. These three are the foundational libraries of scikit-learn.

Exploring scikit-learn Libraries

Let's get a little hands on. We'll explore the scikit-learn webpage and explore the documentation packages and libraries associated with scikit-learn. When you're working scikit, this is a page that you'll visit often. So it's best to be familiar with it. The scikit-learn page is available at scikit- learn.org. You can see here that scikit-learn offers you simple and efficient tools for data mining and analysis. You can see that it's open source, commercially useable under the BSD license. You'll often use this top tab bar here to navigate the scikit-learn webpage. Home is the current page. You can view installation details, documentation, and examples with scikit-learn here. You can see from this home page here that scikit-learn offers you a wide variety of ML models to solve classification and regression problems. Classification and regression are examples of supervised machine learning techniques, which use BI labels to train your ML models. We'll talk a little more about supervised and unsupervised techniques in the next clip. Scikit-learn also offers you many machine learning algorithms for clustering and dimensionality reduction. Both of these are categories of models, which are part of unsupervised learning. Unsupervised learning techniques don't use label training data, instead try to find patterns in the data itself. If you want to know how to get the scikit-learn package installed onto your local machine, click on the Installation tab to the top of your page. This will take you to a page that has detailed steps for getting scikit-learn onto any operating system, Windows, Mac, and Linux. If you want to view the docs for a particular machine learning algorithm or a preprocessing library, the Documentation tab is what you'll use. There are tutorials, quick starts, user guides, APIs, everything available here. If you're looking for specific scikit-learn examples, use the Examples tab. Look for a specific use case from the left navigation pane, click on that, and walk through the examples there. Let's head back to the Home tab here and from the main webpage, let's explore some of the machine learning models that you can build using scikit-learn. Let's start with Classification. Click through and let's take a look at what scikit-learn has to offer. A number of different types of models can be used to solve classification problems. For example, generalized linear models, or you can have linear or quadratic discriminant analysis, kernel ridge regression, support vector machines. All of these might just be jargon or terms that you don't know at this point in time. However, they are important machine learning techniques and all available in the scikit-learn library. The purpose of this exploration is to show you that all of this is available and if you're using scikit-learn, you can go hunting in the right places. You can scroll all the way down here and see the wide variety of models available. Let's head back though, and let's click through and view the regression models that can be built using scikit-learn. You'll find that this takes you to the same page that we visited earlier for classification models. This is because the same techniques can be used for regression, as well as classification in many cases. As I've mentioned before, both regression and classification are examples of supervised learning techniques. We'll talk about that in more detail in the next clip. Let's go back and let's click through to the clustering models that are supported by scikit-learn. This takes you to a very nice page, which explains the different kinds of clustering and you can use this left nav pane here to navigate to a particular model, understand what the model does, and how you can apply it. If you scroll down on this page, you'll find this nice little table here, which shows you which clustering model is the right one to choose based on your circumstances or your use cases, and also the strengths and weaknesses of each kind of model. Let's go back and continue our exploration here. I'm going to now click through to the dimensionality reduction techniques supported by scikit-learn. As you dive further into machine learning techniques, you'll find that often you have a lot of data and a lot of the data might be noisy or not needed, which is why dimensionality reduction is so important. Training your model with data which has many attributes or dimensions can also lead to a poor model. And this is why scikit-learn supports all of these dimensionality reduction techniques to improve the performance of your model. At this point, you know the problem that you want to solve and you know the data that you want to feed in to train your model. But is the model that you've chosen the right one? And has the model been designed correctly? Well, that is why scikit-learn offers a whole host of techniques for model selection where you can compare, validate, and choose the right kind of model parameters. Cross-validation techniques are what you'll use to measure the performance of the model that you've build. Hyperparameter tuning allows you to design your model well. So you have this very large data set, which you're confident can train a good model, but the data set is not set up right. This is where you need preprocessing techniques so that your data is set up the right way to train a robust model and scikit-learn has a large number of preprocessing libraries available for you to use. Before we move on from our exploration of the scikit-learn page, there is one last important link that we should visit. Click on Documentation and from the dropdown, click on API. This is what allows you to view the entire API interface offered by scikit-learn. When you're working hands on with the scikit-learn library, this is a link that you'll visit often. Here are the base class APIs for scikit-learn estimators. You can scroll down and view other APIs as well. Here are the models that you can build for clustering using scikit-learn. If you scroll down further, you can see that scikit-learn has a number of built in data sets available in the datasets namespace. I'm just sampling some of the API links here. If you scroll down further, here are the ensemble models that you can build with the scikit-learn library. Ensemble models in machine learning are ML algorithms that use more than one ML technique under the hood. Scikit-learn is primarily used to build and train traditional machine learning models. But if you scroll down at the very bottom, you'll see limited support for neural networks as well. Well, there is a lot more to explore on this page, and I'll leave that to you to do as self study.

Supervised and Unsupervised Learning

In an earlier clip we had referenced supervised and unsupervised ML techniques, and in this clip we'll understand what exactly they are. Now these are two types of ML algorithms. The big picture understanding of machine learning that we've spoken of so far basically talks about supervised learning. Here you have training data with labels associated, and these labels are used to correct the algorithm. Whereas with unsupervised learning, you have no correctly labeled training data. The model itself has to be set up right so that it can learn structure in the data. As a student of machine learning, it's common to start with supervised learning techniques first, which is what we did. We talked about an ML-based classifier to classify whales as fish or mammals. We picked the machine learning algorithm for our classifier and the algorithm learned from data in the training phase. This is the phase where we feed in a large corpus of data, classified correctly, and our algorithm uses this data to train the model parameters. Once we have a fully trained model, we could then use it for prediction, where the classifier could be used to classify instances it hasn't encountered before. This entire idea of training your model parameters with a corpus of data, with a prediction from your model, was fed back in the form of a cost function or loss function to improve your model parameters, is a highlight of supervised learning. The input that you feed into your machine learning algorithm during training or prediction, are referred to as features or x variables. X variables are the attributes of your entity that the machine learning algorithm focuses on. They are called features, and every data point is a list or vector of such features. So every record that you feed in for training or prediction is a vector of features. And this is why the input ML algorithm is referred to as a feature vector. If you're using your ML model to predict the price of a car, all of the attributes of a car make up a feature vector. The term features, feature vector, or x variables can be used interchangeably, so it's important to remember this. Supervised learning techniques also have y variables or labels. These are the attributes that the machine learning algorithm tries to predict after it has been trained. The price of a house or car, whether a review was good or bad, all of these are labels. Now there are different types of labels. You can have categorical labels, which have discrete values. These are typically the output of classification models. Or you can have continuous numeric variables, for example, the price of a car or a house. This is typically the output of regression models. These labels or predictions are also called y variables. Let's think about supervised learning from a slightly different approach. What supervised machine learning techniques try to do is to seek to learn the function f that links your input data, x, with the output predicted value, y. So the whole idea of training your model is to help your model figure out what exactly this f is. Now linear regression is often thought of as the simplest possible machine learning model. Let's think of what exactly linear regression is all about in the very simplest case. Here you have some data in two dimensions. X on the x axis form the features. Let's assume you have just one feature. And y on the y axis is your output prediction. All of the data that you see on screen here have x and y values that you use to train your model. Once your model has been trained, you basically want to see, given a particular x, what is the predicted value of y. The linear regression model helps you do this by trying to find the best fit line on your data using the training process. What we are looking for from our linear model is to try to find that line that best represents the underlying data. Once we've found this best fit line, we can use this line to predict values of y, given an x. If you get back to our definition of supervised learning where we are trying to figure out the function f that connects the input with the output, linear regression specifies up front that this function f is linear. And this makes the process of figuring out this function, f, a little simpler. We know that a linear function is of the form Wx + b. We only have to figure out the values for W and b for our training data. But really, linear regression is the simplest possible example. There are machine learning algorithms, such as neural networks, that can learn or reverse engineer pretty much any function linking the input with the output. Provided, of course, that you have the right kind of training data. Classification and regression, both of which we've discussed, are the most common techniques in supervised learning. With this big picture understanding of supervised learning, let's move on and talk about unsupervised learning where the model has to be set up right to learn from data. The most important detail that you need to keep in mind about unsupervised machine learning techniques is the fact that unsupervised learning does not have y variables or a labeled corpus, so you don't have correctly classified instances to train your classifier, or the right predicted values to train your regression model. With unsupervised learning techniques, you only have the input x variables to work with. You have no output predicted labels or y values. So your model doesn't really have the information it needs to correct itself or tweak the model parameters to become better. So what the model does in unsupervised learning is to understand or find patterns in the structure of the data itself. The model has the data alone to work with and unsupervised learning algorithms self discover the patterns and structure in the data that you feed it. As you might imagine, classification and regression can't really be performed by unsupervised techniques alone, but often unsupervised learning techniques preprocess data that you use for classification and regression. Common techniques in unsupervised learning and those which are supported by the scikit-learn library, are clustering. This is what you'd use to identify patterns or logical groupings in data. A very popular clustering algorithm is K-means clustering. Unsupervised learning techniques are also used for dimensionality and reduction of your data. Dimensionality reduction is what you would use to identify significant factors within your data. Principle components analysis is a very common dimensionality reduction technique, and it's often abbreviated as PCA.

Installing scikit-learn Libraries

In this demo, we'll use the pip package manager in Python in order to install the scikit-learn libraries and its dependencies on your local machine. Here I am at the terminal window of my machine. My current working directory for this course is called BuildingFirstScikitLearnSolution. All of the code that I'm going to write in this course is going to be within this top level directory. For this course, I assume that you're familiar with Python and you already have Python installed on your local machine. We'll be writing Python code using Jupyter Notebooks. The current version of the scikit-learn libraries is version 0.20 .3. This is at the time this course was recorded, and this version of scikit-learn needs Python version greater than or equal to 2.7 or greater than and equal to 3.4. For future releases of scikit-learn, specifically version 0.21 onwards, only Python versions greater than or equal to 3.5 will be supported. So make sure that you have the right version of Python in order to be able to run scikit-learn. The version of Python that I'll be working with throughout this course is Python 3.7 .1. We'll write our scikit-learn code using Jupyter Notebooks, which is a browser-based interactive shell for executing Python code. If you haven't worked with Jupyter Notebooks before, I highly recommend it. Go ahead to jupyter.org and install it. Installation is very straightforward. Jupyter Notebooks allow you to quickly execute code and see results right there within your browser. The Jupyter Notebook version that we'll use for all of our demos is version 4.4 .0. Pip is the standard Python installer used to install libraries and other dependencies. Pip stands for Python installs packages, or pip installs packages. The current version of pip that I have installed on my local machine is pip 18.1. You can see that my current Python and pip installation uses the Anaconda distribution, which is a free and open source distribution of the Python in our programming languages for scientific computing. Once you have pip and Python installed, installing the scikit-learn libraries is very straightforward. Simply call pip install -U scikit-learn from your terminal window. This will install scikit-learn and all of the other libraries that it depends on. You can see from the messages here on my screen that I already have scikit-learn installed. I also have NumPy and SciPy installed. These are the libraries that scikit-learn depends on. If you wanted to ensure that you have the latest version of NumPy, NumPy is a package that you'll use for array computation and matrix operations, you can simply call pip install -U numpy, and get the latest version on your local machine. And you can run the same pip install -U command for SciPy, if you wanted to install SciPy separately. Remember that when you run a pip install for scikit-learn, it will pull in all of the dependent packages that it needs. You don't need to run these commands separately. Let's go ahead and bring up Jupyter Notebook in order to start coding in Python. Run Jupyter Notebook on your terminal window and this will bring up a URL that you can paste into your browser to run your notebook. However, in most cases, Jupyter should automatically open up a browser window for you, pointed at this URL. And here is Jupyter Notebook up and running on my machine. I'm ready to get started exploring scikit-learn.

Summary

And with this, we come to the very end of this introductory module where we spoke about machine learning algorithms that can learn from data and the scikit-learn library to build traditional ML models. We saw that the scikit-learn library is extremely popular with data scientists because of how easy and intuitive it is to work with. It provides a comprehensive library of ML models for classification, regression, clustering, and dimensionality reduction. In addition, it also provides library functions to evaluate your trained model and for preprocessing the data that you feed into your models. Once we had a basic understanding of machine learning, we studied the different categories of machine learning techniques, specifically supervised and unsupervised learning. We also compared and contrasted traditional machine learning models with representation based ML models. We saw that scikit-learn is typically used to work with traditional ML models, whereas frameworks, such as TensorFlow, Keras, and PyTorch, are used to build deep learning models, such as neural networks. In the next module, we'll get a big picture understanding of the typical workflow to build and train your ML model, and we'll see where exactly the scikit-learn library fits in.

Understanding the Machine Learning Workflow with scikit-learn

Module Overview

Hi, and welcome to this module on Understanding the Material Learning Workflow with scikit-learn. Now we'll first get introduced to the basic steps involved in building and training a machine learning model. We'll then understand where the scikit-learn library fits into this typical machine learning workflow. You'll find that the scikit-learn library offers objects and functions to build every step in an ML workflow. We'll then talk about the high level estimator API in scikit-learn that you can use to build and train your models. We'll also discuss the design principles underlying this estimator API and talk about how individual estimators can be used as building blocks to build scikit-learn pipelines. You'll see that scikit-learn, in addition to offering a comprehensive suite of machine learning algorithms, also offers a whole host of \_\_\_\_\_ functions and objects for model evaluation and transformation. And finally, we'll get a little more hands on in this module where we'll use scikit-learn to load in data, clean and transform our data, and use Matplotlib to visualize our data.

The Machine Learning Workflow

In this clip, we'll talk about the steps involved in building and training a machine learning model. This is the machine learning workflow. And this flow chart here should give you a big picture understanding of the machine learning workflow. Now this looks really complicated, but you'll find that every step is intuitive and follows through from the previous step. And we'll study this diagram in detail, one component, or one step, at a time. Before you get to any of the individual steps here, you need to have an understanding of the problem that you're trying to solve. Are you trying to classify email as spam or ham? Are you trying to identify objects within images? Are you trying to predict stock prices? Once you have that understanding, you'll need raw data that is relevant to your particular use case. You need to figure out what data you have to work with. What format is this data in? Do you need to scrape the web or do you have this data available in a database somewhere? For a stock price prediction model, you might want to use stock prices from the last 10 years. Once you've identified the raw data, the next step is to prepare your data for machine learning. You need to load the data so you're ML models and preprocessing pipelines can access the data. If you just want to prototype your model, this could just be on your local machine in the form of CSV files. But for large scale distributed training, you might want to upload your data to the cloud somewhere. Once this data is available to your program, the next step is to get this data in a format that can be used by our model. You need to perform a bunch of data preprocessing to clean your data and get it in the right format. Machine learning models can only work with numeric data, which means that strings or categories in your data have to be converted to the numeric format. That is part of data preprocessing. Once your data is all ready, as a human expert, you need to choose the algorithm that you want to fit on your underlying data. As we've spoken earlier, in traditional ML models, different algorithms use different techniques to learn from data. Your algorithm could involve decision trees, support vector machines, all of these are common ML models. Once you've made your choice of algorithm, you will then use this ML algorithm to learn from your data. This is what it means to fit a model on your data or to train your model on your data. At the end of the training phase, what you'll have is a fully trained machine learning model that is ready for prediction. But in the real world, it's not that easy. How do you know that the model that you have trained performs well? This is where validation comes in, and there are a variety of different validation techniques that you can choose to evaluate your model. Validation techniques will help you score your model to see whether it's predictions are good or meet a certain bar. Validating your model often involves examining the fit of your model on instances it hasn't encountered before on real world data. You will then see whether you're satisfied with the result of your model. If your model performs well, you're all set. But if it does not, as it often happens in the real world, you might want to update your model. Updating your model is rather a broad statement. This could be anything. You might choose a different algorithm, which works better on your data. You might include more data in your training phase. You might run training for a longer period of time. Once you've updated your model, you might repeat this entire sequence of steps once again. You have an updated model, you choose a validation method, examine the fit of your model, see whether you're satisfied or not. Until, finally, the answer is yes, I'm satisfied. You're happy with this model, it does well with predictions on your data. You'll finally deploy this model to production. There you'll feed in new data for prediction in the real world. And it's also possible for your model to learn from this new data as well. And this completes our machine learning workflow step by step to get a production ready model. And you'll see that it's possible to use the scikit-learn library for every step in the machine learning workflow. Preprocessing and cleaning your raw data, all the way through to building your model, validating it, and using it for prediction.

Using scikit-learn in the Machine Learning Workflow

Scikit-learn has APIs that you can use to work with every step of the machine learning process, and all of these APIs are built using this higher level estimator object. Almost all of the library classes in scikit-learn use this estimator API, which means you have a consistent interface that you can work with. The estimator API is not available just for your machine learning algorithms, it's also available for preprocessing techniques, model selection, and evaluation techniques as well. Whatever machine learning model you choose to use, the basic steps involved will be the same. You create or instantiate a model object, which you then fit to the training data by calling the same Fit method or Fit and Transform method. Once you have a fully trained model, you can use that model for predictions on new data. And the same estimator API becomes part of pipelines that you can construct for very complex operations. The estimator API in scikit-learn has been designed with a lot of care and consideration, keeping in mind a few high level design principles. The first and most important of these principles is consistency. All objects share the same interface, and they are drawn from a limited set of methods, and the documentation you will find is fairly consistent. All estimator objects allow you to inspect model parameters in a very straightforward manner. All specify parameter values are exposed as public properties or attributes. Scikit-learn APIs follow the standard Python format and have a limited object hierarchy. Only machine learning algorithms are represented by Python classes. Data sets are represented in standard formats, such as NumPy arrays or Pandas data frames. Often the implementation of more complex machine learning models use simpler machine learning models as building blocks, and scikit-learn follows this wherever possible. ML models are built using composition, so that machine learning tasks are expressed as a sequence of more fundamental algorithms. And finally, you'll find that the creators have made a real effort to provide sensible defaults for input parameters. Wherever models require user specified parameters, the library defaults make sense for a vast majority of use cases. Let's go back to the machine learning workflow that we saw earlier and see what the scikit-learn has to offer for every step in the process, starting off with the raw data. You'll find that scikit-learn interoperates seamlessly with Pandas and NumPy. So if you load in your raw data in the form of NumPy arrays or data frames, you can directly use that data with scikit-learn. The sklearn.preprocessing namespace offers a host of functions and libraries to perform preprocessing on your numeric, as well as categorical data. You can apply techniques, such as standardization, normalization, and scaling techniques, to bring your numeric features into the right form. There are also preprocessing techniques that you can use to convert your discrete categories, or strings, to numeric form. Often ML algorithms may not work well with missing values or outlier data, and there are a number of different preprocessing techniques that you can use to deal with missing values, outliers, and noisy data. Choosing an algorithm, well, scikit-learn has a comprehensive suite of algorithms. And we got a little taste of this when we explored scikit-learn's webpage. There are a host of algorithms available for regression, classification, clustering, and dimensionality reduction. All of these ML algorithms expose a consistent interface, allowing you to train your model in a very straightforward manner to find the best model parameters. Fitting a model on our underlying data, simply involves invoking the Fit function or the Fit Transform function on your estimator API. Once you have a fully trained model, the next step is know whether this model is a good one or not. And for this, scikit-learn makes available to you a comprehensive suite of cross validation tools. The simple estimator API is available for several different cross validation techniques, K fold, group K fold, and a variety of others. You'll find that different kinds of models have different metrics that you'll use to evaluate those models, and all of these metrics are available in the form of simple functions in scikit-learn. Regression models might use the R squared metric. Classification models might use accuracy, position, and recall. All of these are easy to use scikit-learn library functions. Scikit-learn also allows you to build pipelines using estimator objects as building blocks. This will allow you to train and evaluate your models until you are finally satisfied with the model that you have. Once you have a fitted model, that can be used in production, and you can have your machine learning model make predictions for you by invoking the predict method on the estimator. The estimator object takes as its input new prediction instances and outputs a predicted value from your model. Besides estimator objects for preprocessing, model validation, and the actual ML algorithms, scikit-learn also offers a higher level pipeline estimator object. A pipeline is composed of other estimators that are building blocks of the pipeline and it sequentially applies several transforms on your input data. The scikit-learn pipeline can include all of the steps involved in the machine learning workflow and the pipeline can be evaluated and tuned as a whole. Working with scikit-learn pipelines is no different than working with an individual estimator. Pipeline objects are estimators as well, only that they apply transforms sequentially on the input data. What a pipeline returns is a fitted estimator, and the final step in the pipeline only implements fit, it does not transform the data. When you first start working with scikit-learn, you may not use pipelines at all. But as you progress further and build more complex models, you'll find pipelines extremely useful and efficient. Pipelines cache intermediate results from your training runs, and it's possible for you to tune the pipeline as a whole instead of just tuning individual estimators. Pipelines can also include cross-validation techniques, and you have the ability to switch in and switch out individual steps in the pipeline. Thus scikit-learn pipelines give you very fine grain control over the machine learning workflow.

Choosing the Right Estimator: Classification

When you dive into scikit-learn, you'll be confronted with a whole host of machine learning techniques and estimator objects. But which is the right one to choose for your use case? That's what we'll see in this clip. The first thing that you need to be clear about is the kind of problem that you want to solve, and this is extremely important. Focus first on defining the right problem, and then you can focus on choosing the right estimator to solve this problem. It's important that you get the higher level objective right first, before you move on to implementation details with scikit-learn. The scikit-learn documentation offers this very nifty flow chart, which you can use as a quick cheat sheet or reference to figure what the right estimator is for your use case. There are a lot of steps here, and in fact, this is not the entire flow chart, there is more on other pages, but every step involves asking yourself a question about your problem and your data and then following a part down this flow chart. The very first question you need to ask yourself is, how much data do you have to work with? Do you have at least 50 samples of training data, which are correctly labeled or classified? Well, if your answer to this question is no I don't have 50 samples, then that's a problem. I don't think machine learning can really help you, neither can the scikit-learn library. You need more data. If you have more than 50 records of training data, you're in luck, let's proceed further and ask more questions. What are you trying to achieve? What is the problem that you're trying to solve? Are you trying to predict a category? Are you trying to determine whether reviews are positive or negative? And email is spam or ham? If the answer to this question is yes, that is a classification problem, which means you need labeled data. If the training data that you have to work with is correctly classified, has the right labels associated with each record, well you're in luck. This is a classification problem, and scikit-learn has several different machine learning algorithms that work for classification. Now this brings us to another question about how much training data you have. If your data size is medium, that is you have less than 100, 000 samples, then an estimator that you might want to use is the linear support vector classifier. This is a classifier model that belongs to the family of support vector machines, a particular kind of ML algorithm. Let's say that the model that you built using the linear SVC is not really working well. If the data that you're working with is text data in the form of reviews or newspaper articles, you might want to go with the NaïveBayes classifier. If it's not text data and the linear SVC hasn't worked well for you, another estimator object that you might try is the KNeighbors classifier. If KNeighbors also doesn't give you a good model, you might go with other classification techniques, such as general purpose support vector classifiers or ensemble classifiers. Ensemble methods use a variety of different machine learning techniques under the hood to give you your final prediction. On the other hand, if you have a very large dataset, greater than 100, 000 records, a classifier that you might choose is the stochastic gradient descent, or the SGD classifier. If this does not work well for your use case, another technique available in the scikit-learn library is kernel approximation. Kernel techniques involve applying a mathematical function to your data to transform your raw data before feeding it into a model.

Choosing the Right Estimator: Clustering

Here we are back to the beginning of this flow chart where we are trying to figure out the right estimator for our use case. Let's go back to this question here about what it is that we want our model to do. We want our model to predict a category or a label. But what if you don't really have correctly labeled training data? What will you do then? Well, this is where you would apply clustering techniques in order to find patterns in your data, and scikit-learn has a whole host of estimator objects that allow you to perform clustering. The first question you need to ask yourself here is, do you know the number of categories into which your data will be divided? For example, if you are trying to identify hand written digits from 0 through 9, you know that you have 10 categories. If you're trying to identify images of a cat, dog, and a bird, you know that you have three categories. If the number of categories in your data is known up front and you have less than 10, 000 samples, an algorithm that you might choose to use to perform clustering is the K-means clustering algorithm. If you find that K-means does not work, there are other clustering algorithms available as well in scikit-learn, such as spectral clustering and Gaussian mixture models. But if your dataset is medium to large sized data with more than 10, 000 samples, you might want to go with MiniBatch K-Means. This algorithm is essentially the same as K-means, but it works on batches of data, which means it can scale to very large datasets. If your data is a little more unknown, in that you don't know the number of categories in your data and you find that you have less than 10, 000 records or samples, clustering techniques that you might choose to use are MeanShift clustering and VBGMM, which stands for variational inference for Gaussian mixture model. These may all just be terms for you at this point in time, but it's important to get a big picture understanding of what all scikit-learn has to offer. If the number of categories in your data is not known up front and you have a very large dataset, greater than 10, 000 records, you're out of luck as far as the scikit-learn library is concerned.

Choosing the Right Estimator: Regression and Dimensionality Reduction

Here we are back at the beginning of this flow chart, and we have to figure out what is the type of problem that we are trying to solve? Let's say we are not really predicting a category, instead we want to predict a quantity or a numeric value. If this part is clear, then you know that this is a classic situation where you would choose to use a regression model, and scikit-learn has a host of regression models for you. Let's assume here that the dataset that you have to train your machine learning regression model is fewer than 100, 000 samples. Every entity or record in your dataset has attributes or features. How many of those features really matter? If you find that only a few features matter, that is the data that you're working with is of a fairly low dimensionality, you might choose to go with the Lasso or the ElasticNet regression models. But if your data is of a very high dimensionality, every record has many features that are significant and important, other regression techniques might make sense. You might choose to go with RidgeRegression or SVR, support vector regression, with the linear kernel. If this results in a poor model, you will want other techniques. You might choose to do support vector regression using a different kernel, such as the RPF kernel, or you might choose an ensemble regressor. If your dataset is very large, greater than 100, 000 samples, the estimator object that works well is the SGD, or the stochastic gradient descent regressor. That completes our explanation of regression models. Let's go back to the beginning of this flow chart. Let's say you're not really predicting a quantity and you're just looking and exploring your data. You want to figure out the significant features or latent factors in your data. Well, that's what dimensionality reduction is all about, and this is also supported by the scikit-learn library. Scikit-learn has an estimator API for one of the most popular dimensionality reduction techniques, principle components analysis, or PCA. Don't worry if PCA does not work on your data, there are other techniques that you can choose from in scikit-learn. If your dataset contains less than 10, 000 records, dimensionality reduction techniques that might work well are Isomap and Spectral Embedding. And if those do not work, you can always go to the LLE, or locally linear embedding. Or if you have more than 10, 000 samples in your dataset kernel approximation techniques are what you would choose. As we discussed before, kernel functions apply a mathematical function to transform your data before applying other algorithms. Let's go back to the beginning of our flow chart once again. You're not predicting a category, you're not predicting a quantity, and you're not just looking at your data. But instead, you want to predict the structure of your data. Well, tough luck. Scikit-learn doesn't really have any libraries for that. Well, that was a long and complicated flow chart, which introduced to you the names of a number of different machine learning algorithms. Keep this big picture in mind as you dive into studying machine learning.

Exploring Built-in Datasets in scikit-learn

In this demo, we'll explore the built-in datasets available in scikit-learn for a student of machine learning. Here we are on our browser with Jupyter Notebooks up and running. Click on the New dropdown to the top right of your screen in order to open up a new Python notebook. We'll have this notebook running the Python 3 kernel, and we'll write all of our development code in Python 3. This brings up a new browser window and you can see that this notebook is currently untitled. If you select the title, you'll be able to change and rename your notebook as you please. I'm going to call this notebook ExploringDatasets, because that's exactly what we are going to do. As a student of machine learning, you'll be working with different kinds of algorithms and models. Now all of these models need training data on which to train, and there are a number of datasets built into scikit-learn, which you can use out of the box. These datasets have been made freely available by the original authors and are great resources for any machine learning student. Let's first set up the import statement for the Python modules that we'll use. I'm going to import sklearn, I'm going to import the numpy library and alias it as np, and the scipy library and alias it as sp. And this is for those who haven't used Jupyter Notebooks before. You'll type code into a cell and in order to execute that bit of code, you'll simply press down the Shift and Enter keys. Shift+Enter to run code. I'll quickly print out the versions of these packages that I have installed. If you have later versions, you might find that things have changed a little bit. I'm currently using sklearn version 0.20 .2. The numpy version used across all of the demos in this course is version 1.16 .1. And the scipy version that scikit-learn will use is version 1.2 .1. When you're building machine learning models, you'll need to wrangle and clean your data. This is best done using Pandas. Pandas represents your data in a tabular format and is very easy and intuitive to work with. I'm working with Pandas version 0.23 .4. If you don't have Pandas already installed on your machine, you can get it with a simple pip install. All of the datasets available in scikit-learn for a machine learning student to use is present in the sklearn datasets namespace. Go ahead and import the load\_breast\_cancer function and this will allow us to load the breast\_cancer\_dataset. Loading this dataset into your program is simply a matter of invoking the load\_breast\_cancer function, and you have this dataset all loaded and ready to go in the form of numpy arrays. Now of course you're curious about what this dataset is all about. You can print out the description attribute of this dataset in order to get more details. The breast\_cancer\_dataset is a very common one typically used for classification problems. It's originally from Wisconsin and there are a total of 569 records in this dataset. This dataset contains a number of numeric attributes of tumors found in patients and then a class label which tells you whether this patient was actually detected with breast cancer or not. This dataset is typically used for binary classification. Was the tumor malignant or benign? You can see that the descriptions give you summary statistics for all of the numeric attributes within this data. If you're going to site or use this dataset within a research paper, here is the original source information. The structure of the dataset that we just loaded is in the form of a dictionary of numpy arrays. Let's take a look at the keys of this dictionary. You can see that this dataset contains the data. These are the attributes or the features that we'll use to predict breast cancer, and a number of other bits of useful information. We've already explored the description of this dataset, let's explore some of these other dictionary attributes. Let's take a look at the feature names and you can see, these are the numeric features present in our dataset. For this particular dataset, all of these features are numeric and they relate to attributes of the tumor found. The actual numeric data is in the form of a numpy array and is present in the data field. If you take a look at the shape of the data, you can see that we have a total of 569 records and every record has 30 attributes. The breast\_cancer\_dataset.target\_names member variable will give you the target that you'll use to predict with this dataset. You'll predict whether the tumor was malignant or benign. The shape of the target will show you that we have a total of 569 labels corresponding to each of our records. When you use scikit-learn's built-in datasets, they are available in the form of NumPy arrays, however, it's much more intuitive to work with datasets in Pandas, and it's very straightforward to convert these NumPy arrays to Pandas data frames. As you might already know, converting NumPy arrays to data frames is straightforward. Simply instantiate a data frame with the data in our dataset and set is column names to be equal to the feature names. We set up the features for our ML model. We can then instantiate a separate data frame with the target data. And the column names, you can specify based on your use case. If you want a single data frame with the features, as well as the target labels, you can simply concatenate these two data frames along the rows. The data frame.head function will allow you to view the first five records in this data frame, and you can see that we have the features, as well as the target, all neatly concatenated and ready for you to use. The df.shape variable will give us the rules and columns for this data frame. There are 569 rules corresponding to our data and 31 columns, 30 feature columns and 1 target column.

Exploring the Boston Newsgroups and Digits Datasets

Now that we've understood how to work with the datasets available in scikit-learn, let's quickly explore some of the other datasets available for different kinds of ML problems. The Boston housing prices dataset is a standard dataset used for regression. Load this dataset as a dictionary of NumPy arrays by calling the load\_boston function. Just like with the breast\_cancer dataset, you can explore the keys of this dataset. You have the data, the target, the feature names, the description, the file name, and so on. Whenever you're working with a new dataset, take a look at the description so that you can orient yourself with what attributes this dataset contains and what you'll predict with this dataset. This house crisis dataset is from Boston. It contains 506 records, 13 attributes, and you have 1 target that you'll predict, the median value of an owner-occupied house. You can see that the features are interesting bits of details, such as the age of the house, the per capita crime for that neighborhood, the pupil-teacher ratio for that town, and so on. If you want to work with text data for topic classification problems, the 20newsgroups dataset is available for you. You can either fetch the training subset or the test subset for this dataset. They are separate sets for this newsgroups data, subset is equal to train will give us the training data. Once again, this dataset is in the form of a dictionary of NumPy arrays. The description, of course, is where all of the interesting details are. You can see that this dataset contains 18, 000 newsgroups posts on 20 topics. And the standard use case for this particular dataset is topic classification. We'll take a look at all of the posts and try to figure out which topic the particular post belongs to. If you take a sample of the data, you'll find that all of it is text data in the form of newsgroups posts. The topics into which these posts can be categorized are available in the target\_names, and you can see that there are 20 categories here, sports, baseball, for sale, autos, politics, and religion are some examples here. These are the actual topics, the numeric labels representing these topics are available in the target array. You can see that these are numbers from 0 through 19. Let's take a look at one last dataset here before we move on to building our first scikit-learn solution. This is an images dataset, available using the load\_digits function. These are images of handwritten digits from 0 through 9. Each image is an 8 x 8 image, 8 pixels by 8 pixels. Let's take a look at the keys of this dataset, the keys are familiar to us. You can print out the description in order to understand this dataset further. There are a fleet 5600 images in this dataset. The number of attributes is equal to 64, 8 pixels by 8 pixels equals to 64 attributes. Let's use the Matplotlib visualization library in order to display one of these images so that we can see what they look like. Call plt.imshow on the images variable, and we access the image at index 1. And here is what the 8 x 8 handwritten digits image looks like. If you're familiar with the MNIST dataset, you can think of this images dataset as a simplified version of MNIST.

California Housing Dataset: Exploring Numeric and Categorical Features

In order to build our first scikit-learn solution, we'll get a little more adventurous and choose a dataset available on kaggle.com. Under our current working directory, I've created a folder called datasets, where I'm going to store this dataset and its present in a CSV file called house.csv. This is the California housing dataset, housing prices, but in California rather than Boston, and it's available here on kaggle.com. I'll be writing code in this new notebook here called ExploringTheCaliforniaHousingDataset. We'll use visualization tools and Pandas in order to explore this data. Go ahead and import the packages that we need, I import pandas, the pyplot library from Matplotlib, and the seaborn visualization library. Seaborn is a higher level library built on top of Matplotlib and is very useful for statistical visualizations. If you don't have seaborn installed on your machine, you can get it with a simple pip install seaborn. In order to work with the housing data, I'm going to read in the csv file using Pandas. Pd.read\_csv is the function that I use, and I'm going to access the housing.csv file under my datasets folder. Once this data is loaded in the housing\_data data frame, let's invoke the head function in order to take a look at this dataset. You can see that this csv file includes headers, which have been set up as our data frame column names, and it contains a bunch of different information about houses in California. You can see that some of this information is location-based information. We have the latitude and longitude of the houses and how close it is to the ocean. There are also details about the home itself, the housing median age, total rooms, and total bedrooms. Now if these numbers look a little large, don't get worried. This is because these are not details for an individual house, rather these are details that correspond to a number of houses that are situated in a particular neighborhood or block in California. That's why you can see that the age is the median age of houses in that neighborhood. Total rooms refers to the total number of rooms across all of these houses, as does total bedrooms. We also have a little bit of information about people living in those houses. We have the population for that neighborhood, the number of households, and the median income. If you take a look at the numbers for median income, you'll find that it seems a little low, that's because it's expressed in terms of ten thousands of dollars. So 8.3 is $83, 000. The column that we can predict using the features that we just saw is the median\_house\_value. This is the target for prediction that we'll use in regression and we'll also use this in classification. You'll see how in a little bit. These are the median housing prices in dollars. The head function in Pandas allows you to view the first few records of data. If you want to view a random sampling of records, the sample function on a data frame is a better choice. Here I'm going to sample five rows from my dataset. And the interesting thing here in the resulting sample is you see the different values that are possible for ocean proximity. You can see that certain neighborhoods are inland, some are close to a bay, and some are less than 1 hour from the ocean. This is categorical data comprising of categories or discrete values into which our homes are grouped. The shape variable gives us the number of rows and columns for our Pandas data frame. You can see that there are about 20, 000 records in this dataset, and we have 10 columns, 1 of those is the target or the column that we want to predict. We have nine columns that are features. When you're working with real world data, it's quite possible that there are certain fields that have missing values, values that are not available. That means when you're working with your ML model, you'll have to do a little bit of data cleaning before you can use this dataset in your algorithm. Pandas data frames offer this very useful function called dropna, which will drop all records in your dataset that have missing fields. I'm going to call to dropna, and you can now see that my housing\_data has just 20, 433 records. Roughly 200 records which had some field values missing have been dropped. The describe function on your Pandas data frame will give you basic statistical information about all columns with numeric values. Here in this result, you can see that other than the ocean proximity column, which is categorical nature, all of our other columns are numeric, and we have mean standard deviation and percentile information described here. We are going to be using these 20, 433 records in order to train and test our ML model. You can see that the average age of a house in a neighborhood is roughly 28 years, the minimum age is 1 and the maximum age is 52. You can also see that the average value of a house in a neighborhood is roughly $206, 000, the minimum median house value is $14, 999, and the maximum is $500, 001. This $500, 001 seems a little strange. We'll get to that in a bit as we explore the data further. This gives us a bit of insight into the numeric values in our dataset. Let's understand the one categorical column, that is the ocean\_proximity column. The .unique function will give you the unique value set are present in that column. So here are the ocean\_proximity categories for the different neighborhoods in California. We can be near the bay, less than 1 hour from the ocean, inland, near the ocean, or be on an island. As you might imagine, housing prices are quite likely to vary based on this data.

California Housing Dataset: Exploring Relationships in Data

Exploring the dataset is a key part of the process of building your machine learning model. At this point, we have done a basic exploration of each of the attributes in our dataset, numeric, as well as categorical. Now it's time for some visualizations to understand relationships. Here I'm using the Matplotlib visualization library to plot a scatter plot of the median house value on the Y axis and the total number of rooms on the X axis. This will allow me to visualize the relationship between total rooms and the median price of a house in that neighborhood. And here is what the resulting scatter plot looks like. There is some kind of shape here, you can see that it's kind of tilted upwards indicating as the number of rooms increase, it's possible that house prices go up. Visualizations offer us a first level of understanding of the underlying data. There is another thing we can observe here, notice this little cluster at the very top. At around exactly $500, 000, there is a group of points indicating that there is some kind of gap here on this dataset at $500, 000. We'll keep this in mind as we continue exploring this dataset. I'm going to plot another scatter plot here. Scatter plots, as you might imagine, are great for visualizing pairwise relationships. In this scatter plot, I want to visualize how the median age of a house affects the median house value. And here is what this scatter plot looks like. And it's really hard to discern any kind of pattern here. It's possible that the older the houses are, the cheaper they are, but maybe older houses have better locations. So it's really hard to say. There is another clustering of points here at around the age 52. Once again, 52 seems like some kind of gap on the age of a house. We'll now visualize another pairwise relationship that might be useful, the relationship between house prices and the median income of people in that neighborhood. I have multiplied the median\_income column by 10, 000 so that its value is represented in terms of actual dollars. And this scatter plot has a clear upward flow, showing us a clean positive relationship between median income and house prices. Once again, this cluster of points shows you that house prices seem to be capping out at around $500, 000. Plotting scatter plots for pairwise relationships allow you to view the impact of certain features on our predicted value, in this case housing price. But it's also possible that your features themselves have some kind of interplay between themselves, some kind of correlation. In order to view the correlation in your features, you can simply call the corr function on your data frame. The result of the corr function is the corr relation between every pair of columns in your dataset. Correlation and statistics is a measure that indicates the extent to which two or more variables fluctuate together. If two variables move in the same direction together, they are said to be positively correlated. If they move in different directions, they are said to be negatively correlated. Observe that the correlation of every feature with itself, longitude with longitude, housing median age with median age, median house value with itself, is equal to 1. They are perfectly positively correlated. Correlation values typically range between -1 and 1, where 1 denotes perfect positive correlation and -1 denotes perfect negative correlation. A value of 0.68 here in the cell shows you that median income is positive correlated with the median house value. This is something we knew about when we viewed the scatter plot between median income and median house value. You can also see that total\_rooms is slightly positively correlated with the median\_house\_value. This kind of leads us to believe that median income is a better predicted of house prices, as compared with total rooms. Rather than viewing raw correlation numbers, correlations are best visualized using something called a heat map, and you can call the heat map function, available in the seaborn library, in order to view the correlations present in our dataset. The annotation equal to true parameter that we pass in will print out correlation values on our heat map. And here is what a heat map looks like. Observe that the colors indicate how positively or negatively correlated two features are. Lighter colors indicate high positive correlation. A correlation of 1 is a bright cream in color. Darker colors indicate negative correlation, a correlation of -1 will be almost black in color. You can see that the feature columns of our dataset form the rows and columns of this heat map. And a correlation of 0.69 in this cell here indicates that median income is positively correlated with the median house value.

Summary

And this demo brings us to the end of this module. In this module, we were introduced to the basic steps involved in a machine learning workflow and we saw how the scikit-learn library offers objects and functions to implement every step in the ML workflow. The scikit-learn library offers the high level estimator API for data preprocessing, cleaning outlier detection, training and building your model, validating your model, and finally using your model for prediction. Most of the ML algorithms, preprocessing steps, and other techniques in scikit-learn are implemented using the high level estimator API. The estimator API has a very simple and intuitive set of interface methods that you can work with. We studied how this estimator API has been explicitly designed, keeping in mind principles of consistency, inspection, a limited object hierarchy, composition, and having sensible defaults. We also discussed how individual estimators can be sequentially brought together into pipelines, which is also an estimator. We also discussed model evaluation and transformation as a part of the machine learning workflow and discussed scikit-learn support for this. And finally, we moved on to hands on demos where we loaded and explored the built-in datasets available in scikit-learn. We also loaded in an external dataset, cleaned up the data, transformed it, and visualized it using Matplotlib. In the next module, we'll work with the scikit-learn library to build and train a few simple machine learning models. One for regression and one for classification.

Building a Simple Machine Learning Model with scikit-learn

Module Overview

Hi, and welcome to this module on Building a Simple Machine Learning Model with scikit-learn. Here in this module, we'll work with some classic problems in machine learning, regression and classification. Regression models are what we choose to use when we want to predict continuous data. Stock price on a particular day, the price of a car, how long you'd have to wait for your Uber driver. Classification models are what you'll build when you want to predict categorical data, where your output is in the form of distinct classes or categories. Is the review good or bad? Is this is an image of a cat or a rabbit? We'll focus first on getting a high level understanding of linear regression and logistic regression for classification. We'll then see how we can build and train these simple models using the scikit-learn library. We'll work with some real world data as well.

Understanding Linear Regression

Before we go on to build our linear regression model and use it for prediction, let's understand what exactly regression is all about. As we've discussed before, regression problems are used to predict a numeric value, such as the price of a stock, the price of a house, the estimated wait time for your taxi. All of these are problems that can be solved using regression. Regression models can be built when you know that you have some value X, which form the features of your data, which causes Y. X here is referred to as the cause, that is an independent variable, Y here is the effect, that is a dependent variable. Y depends on X. X here refers to a single variable or to a feature vector, which contains many different features or variables. X is called the explanatory variable. So when you're going into your training data to train your regression model, you know that there exists some relationship between X and Y and it is this relationship that you're trying to find using machine learning. Regression, as we've discussed before, is an example of a supervised learning technique. In order to allow us to visualize linear regression, we'll keep things simple and imagine that data isn't just two dimensions. We have a single X variable and a single Y output for each X variable. Regression models can get arbitrarily complex, and of all these models, linear regression is the simplest and linear regression involves finding the best fit line on your data so that you can use this line for prediction. Linear regression assumes that a straight line is enough to model or represent your data. The term linear comes from the fact that our machine learning model tries to fit a line on our data and not some kind of curve. If you think back, you might remember some of your high school mathematics. A line can be represented using the equation, y = A1 + B1X, or y is equal to MX + C. This is the equation for line 1 that you see here on screen. Now let's think of another line that we can draw, this is line 2, which has the equation A2 + B2X. If you compare these two lines, which do you think is a better fit on the underlying data? Well, because our visualization is simple, it's very easy for you to say that line 1 fits better. But mathematically, how do you arrive at this? What you really want to do is to see how far away the data points are from the line that you fit. You'll drop vertical lines from each point to the lines 1 and 2. So here are vertical lines to line 1 and here are vertical lines dropped to line 2, from every data point. Once you drop these lines, it's pretty clear that the better fit model has smaller vertical lines. And this intuition is something that can be captured using a mathematical formula. The best fit line is the one where the sum of the squares of the lengths of these dotted lines is minimum. And the term used to refer to this mathematical computation is the least square error. So when you're training your machine learning model, you want to minimize the least square error of your linear model. You want to find that straight line that fits on your data, which has the minimum value for least square error. Let's parse what this mathematical computation is all about. So we have the lengths of these dotted lines that we'll drop from the original points to the fitted line. We then find the sum of the squares of the lengths of these lines. The squares are so that we don't have any negative values. And finally, we find the best fit line where these errors are minimized. The distance of your original points from the fitted line can be thought of as errors in your model. And when you use the scikit-learn library to fit a linear regression model on your data, this is what it does behind the scenes. The scikit-learn estimator object will move the fitted line to find the best regression line by minimizing the least square error.

Data Preparation for Machine Learning

In this demo, we'll use scikit-learn estimators to build a simple machine learning model to perform linear regression. We'll train our model on the California housing dataset that we just explored and use it to predict the prices of houses. We'll build a solution in this Jupyter Notebook, LinearRegressionForPricePrediction. Let's import a few of the Python libraries that we'll use up front, Pandas and the PyPlot module from Matplotlib. Once again, we'll use pd.read\_csv to read in the CSV file that contains our dataset. It's present in datasets/ housing.csv. Here is a simple of housing records from our dataset. We've already explored this in a lot of detail here, so there's nothing new that we need to get into at this point in time. I'm going to call the dropna function on my data frame in order to clean up all records with missing values. This will drop all records with missing fields. So we are now left with about 20, 433 records to play with. When we have explored our dataset using visualizations, we had seen a number of points clustered around the housing price $500, 000. That seemed to be some kind of upper cap here. We'd also noticed that the highest value associated with a particular house was $500, 001. I'm going to use this Pandas command to see how many records in our dataset have a median house value of $500, 001. And with this count function, you can see that there are 958 records. So nearly 1000 records out of 20, 000 are at this upper cap of house price. Now something like this can really skew the training of our machine learning model, so it's best to drop these records from our dataset. If you're working with skewed datasets where you have lots of data clustered around a single value or in a single class, there are, of course, techniques that you can employ in the real world to work with such data. For our first scikit-learn solution, we want to simply ignore these records. We'll first calculate the index positions of all of the records with median\_house\_value equal to 500, 001, and we'll go ahead and drop these records from our dataset. Assign the resulting data frame, after these records have been dropped, to the house\_data variable. You can see that housing\_data now contains 19475 records. These are the records that we'll use to build and train our machine learning model. You can call the head function in order to sample the first few records of this data frame. Now all of the features in our dataset have numeric values, except for one column, that is the ocean\_proximity. This column contains categorical or discrete values, not numbers. As you'll probably aware, machine learning models can only work with numeric data. If you have categories or discrete values in your dataset that are represented using string values, you'll need to convert these categorical values to numeric form. There are, of course, several ways in which this can be done, but the most common way is to use one-hot encoding. We'll now use one-hot encoding to convert the ocean\_proximity column to numeric form. This is easiest done using the pd.get\_dummies function, pd.get\_dummies will convert the column ocean\_proximity to one-hot form and remove the original column from your dataset. If you take a look at the shape of your data frame after you've one-hot encoded your ocean\_proximity column, you can see that our data frame now has 14 columns, rather than the original 10. Let's view a sample of our dataset now and take a look at the one-hot encoded column values. Here are all of the numeric values, which we haven't changed. And if you scroll to the right here, you'll see the values for ocean\_proximity in one-hot encoded form. Observe that we now have a column for each discrete value or category in our original ocean\_proximity column. There's a column for the neighborhood being less than 1 hour from the ocean, inland, island, near bay, and near ocean. If the original location of the neighborhood was inland, one-hot encoding places a value of 1 in that particular column, all other columns have 0s. Having converted all of our features into the right form to feed into a machine learning model, we are now finally ready to set up the features and targets for our model. The X values are features or data that we'll use to train our machine learning model and the Y values are our predictions. X values are all of our input columns, except the median\_house\_value. Go ahead and create a new data frame with our X values by dropping the median\_house\_value from our original data frame. The Y values, or the target of our ML model, are what we want to predict using our ML algorithm. These are the actual Y values that we'll use in the training phase of our machine learning model. Access the median\_house\_value column from our housing\_data data frame. X.columns will give you the names of the features that we use to train our model. Notice all of the features are present except median\_house\_value and median\_house\_value is the target that we'll use to train our model. That's what we want to predict with our algorithm. You can see in our features that we have five columns for ocean\_proximity, one for each category or discrete value. This is a categorical variable represented with one-hot encoding. When you're using data to train your ML model, the common practice is to split your dataset into a test set and a training set. The training set is what you'll use to train your model parameters, that's what you'll use to build your model. The test data is a little bit of your original dataset that you'll hold out, that you won't let your model see during the training phase. This test data is what you'll use to measure your model's performance once it has been trained. You'll use this test data to measure how well your trained model performs on instances it has never seen before. The test train split function in the scikit-learn library is an extremely useful function that allows you to quickly split up your dataset into a training set and a test set. You'll invoke the train\_test\_split function with your training features, as well as the target values, both X and Y, and you'll specify as a fraction, how much of the original dataset you want to hold out in the form of test data. Typically when you're building your ML models, you'll do an 80/20 split. Eighty percent of the original data you'll use to train your model and 20% to test your model. X\_train will contain the features of your training data and y\_train will contain the corresponding targets. X\_test will have all of the records for your test data and y\_test will contain the actual targets for your test data. In addition to splitting your data into a test set and a training set, train\_test\_split will also shuffle the records in your dataset. Shuffling your dataset is an important step because your machine learning model should not pick up inadvertent patterns in your data. Let's take a look at the shape of our X data, in our training and test sets, you can see that we have about 15, 580 records to train our model and the rest, 3895 records, will be used to tests our model. Our model during the training phase will not see any of the rules in our test data. Let's take a look at the shape of the Y values or the targets of the training and test data as well. We are now ready to use our first scikit-learn estimator for linear regression to train a model and use it to make predictions.

Training and Prediction Using Linear Regression

Estimator objects in scikit-learn give you a high level API that you can use to build and train your machine learning models. You can form linear regression in scikit-learn using the LinearRegression estimator object. Import the LinearRegression estimator for use in your Python notebook from the sklearn.linear\_model namespace. An important reason that scikit-learn is so popular with machine learning enthusiasts is because of how easy it is to build and train models using estimators. In order to perform linear regression, you'll simply instantiate a LinearRegression object. Here is our LinearRegression estimator instantiated using the normalize is equal to true parameter. Normalization is a process that scales all numeric features in our dataset so that their values are between 0 and 1. This is an often an important preprocessing step in machine learning because having all features in the same scale of 0 to 1 can vastly improve the performance of your model. You can see how easy it is to normalize your numeric features when you use the LinearRegression estimator object. Simply specify normalize is equal to true. You will have to perform an additional preprocessing step. The fit function on the estimator fits your linear model on the training data, that is, this is what you use to train your linear model using our features and target values. The fit function takes in two input parameters, your X features and your Y values or target. When you execute this bit of code, your model will be instantiated and will start training. Because our dataset is fairly small, you'll find that our model trains quickly in scikit-learn. Let's take a look at our training score. Once you have a fully trained model, the score function allows you to evaluate this model on different kinds of data. We've passed in the training data here, which means we'll evaluate how this model performed on the training dataset. And here is the training score in terms of R-square for our linear regression model. Now R-square, as we've already studied, is a measure of how well our linear model captures the underlying variable in our data. And because we've calculated this R-square on our training data, it captures the underlying variation in the training data. Because this is a linear regression model, the training score is the R-squared. For other models, the training score will be different, it'll be different for a classifier, for instance. A training score of 61% indicates that our linear model captured 60% of the variation in our training data. That isn't fantastic, but it's not too bad. I'm going to assign to a variable named predictors, all of the features that we use to train our model, so you can see that predictors contains a list of columns which are our features. As you know, the formula to represent a linear model is of the form Y is equal to WX + C, where X represents our features. The linear model would have calculated a coefficient for each input feature in your dataset, and those are the coefficients we are going to view here. Here we are going to instantiate a Panda series, giving us the names of each of our features and the corresponding coefficients. And here are the coefficients for the linear model that we just trained. For each of our input features, there is a corresponding coefficient or \_\_\_\_\_. You can think of these coefficients as a measure of how much that particular feature contributed towards the prediction. We have our coefficients in a sorted order, which is why negative values are first. Negative values intuitively indicate that these are the features that tend to lower the price of a house. For example, if the neighborhood was at an inland location, not close to the ocean, or if the population of that neighborhood was high. Here are features that positive coefficients, indicating that they tended to increase the median price of a house. You can see that the feature with the highest coefficient is ocean\_proximity\_ISLAND, indicating the most expensive houses in our dataset were located on islands. You can see that the higher the median income of the population in that neighborhood, housing prices tended to go up, and near the bay or less than 1 hour from the ocean also had a positive effect on prices. Now that we have a fully trained linear regression model, let's use it to predict something. Let's call linear\_model.predict on our test data. When we make predictions, we only pass in the X data or the features, and the resulting predictions are stored in y\_pred. So how did our model perform? Let's create a little data frame that compares our predicated values with actual values that we have in our dataset. The columns of this data frame are predicted and actual, and we'll run the head function to sample the first 10 records. And you can see that our predicted values are pretty close. The actual price of a house here was around $204, 000. Our model predicted $198, 000. That's pretty good. The actual price of a house here was around $150, 000, our model said $141, 000. Once again, not bad. Eyeballing the data is good when you want to double check your predictions. However, the best way to evaluate a model is to calculate an R-squared score on your test data. The sklearn.metrics namespace is very useful and has a number of different scoring algorithms for different kinds of models. We are working with linear regression, import the r2\_score function, and call r2\_score on your test data. Pass in the predicted values from your model and the actual values from their dataset, and you can see that the R-square score on your test data is almost as good as it was on training instances. So your model performs as well on instances it hasn't seen before as on the training data. You can also use a scatter plot to visualize actual price values against predicted values, and see how close they are clustered together. The R-square score, of course, is an objective measure of your model, but visualizations give you a big picture feel for how your model works. Our test dataset contains nearly 4000 records. I'm going to sample 100 of these predictions versus actual values from the original data frame. I use the .sample function in Pandas for this and specify that I want 100 samples. I also reset the index on my Pandas data frame so that the index positions start with 0. I'll invoke the head function on this sample to take a look at some of the records. Here are predicted values from our model and actual values from our dataset. We already know from the R-square score that our model is decent, if not fantastic. Let's go ahead and plot actual versus predicted values using a line chart. I call the plot function on my PyPlot library and I plot the predicted values for the label predicted and actual values for the label actual. And here is another cool way for you to visualize actual versus predicted values. Predicted values are in blue, actual values are in orange, as you can see from the legend on the top right. And thus, we've successfully built our first scikit-learn solution using a linear regression estimator and used this estimator to predict housing prices.

Understanding Logistic Regression

Now that we've understood linear regression, we'll now build a classification model using scikit-learn. And we'll build a classification model using logistic regression. Now the first time I heard the term logistic regression, I was a little confused. Remember that logistic regression is actually for classification, not regression. We'll first get a high level understanding of how logistic regression works before we build and train a model in scikit-learn. Now there are two approaches you could follow to meet a particular deadline. One way is you start 5 minutes before a deadline and work really, really crazily. Well, good luck with that technique. Another extreme could be that you start working towards the deadline 1 year before. Well, this isn't really ideal as well, this might be overkill. Neither approach, if you go to see, is optimal. Let's see what's wrong with each of these approaches. Let's say you start a year in advance, the probability of you meeting the deadline is 100%, but the chances of you getting other important work done may be close to 0. Starting a year in advance means that all your attention is focused on this single deadline and nothing else. But what if you start 5 minutes before the deadline? The probability of meeting the deadline is clearly 0%, but the probability that you got other important work done is 100%. Neither of these approaches is really very good. What we are looking for is the Goldilocks solution. At one end, we can work very fast, we start very late and hope for the best. Other extreme, we work extremely hard. We start very early, well in advance of the deadline, and do little else. The Goldilocks solution is where you work smart. We start as late as possible to be sure that we make the deadline. So where does logistic regression fit in? Wait, we're getting there. This is all the lead up. Let's say we were to plot this in the form of a graph. On the X axis, we have the time to deadline, and on the Y axis, we have the probability of meeting that deadline. If you were to start work 5 minutes before the deadline, your point would be here at the bottom. Five minutes left of the deadline, 0% probability of hitting the deadline, or close to 0. If you're really hard, you would be at the other extreme here in this graph. You start 1 year before the deadline, you have 100% probability of hitting the deadline. We now have the two extreme solutions, work fast and work hard, plotted on this graph here. What we are really looking for is the middle ground where we work smart. When do we start so that we have a 95% probability of hitting the deadline. For different values of time to deadline, when you plot these probabilities, you'll find that they often resemble a nice S curve. This means you fit an S curve on this data, and once you have this S curve representation, you can use it for prediction. When do I need to start work so that I have a 95% probability of hitting my deadline? If you use this S curve, you might get a sensible answer. You need to start work on the project 11 days before the deadline. It's important that you understand the conceptual idea behind this S curve, even though we might gloss over the mathematical details, because this is the intuition that lies behind logistic regression. The whole purpose of logical regression is to help you find how the probabilities of a particular outcome are changed by the actions that go into that outcome. So let's make a more general version of the S curve that we were working with. We have the time to deadline on the X axis, and we have a probability on the Y axis. This entire region here at the very left of this S curve are when you've started too late. You start too late and you'll definitely miss your deadline. All of the region here at the very right of your S curve are when you've started work on your project too early. You start too early, you'll definitely make it, but you don't have much chance of getting other stuff done. Working with smart is really this region here at the center where the S curve slopes up. This dotted line at the center can be thought of as a threshold, below the threshold you have less than 50% chance of meeting your deadline, and above the threshold you have greater than 50% chance of meeting your deadline. And this threshold is something that you can tweak typically with ML models. The logistic regression machine learning model fits an S curve on your data, just like a linear model fit a straight line. Y is hit or miss, 0 or 1. X is the start time before deadline, if you consider our example. And p(y) is what we plot on the Y axis, the probability that we'll meet our deadline or y = 1. The output of classification models is a probability and you choose that outcome which has the higher probability. If the probability that you'll meet your deadline is p(y), the probability that you'll miss your deadline is 1 minus p(y). For those who are curious about the mathematics involved, logistic regression uses this formula that you see on the left to estimate the probabilities of outcomes. This formula is that of an S curve and logistic regression involves finding the best fit S curve on your data. Here you can see A and B are the variables. A is the intercept and B is the regression coefficient. And the constant e here is Euler's number, something that you might remember from back in high school. Well, I barely remember, but that's okay. You don't need deep mathematics to build ML models, unless you're getting into research. You just need a good high level understanding. Let's go back to the classification example that we worked with throughout this course. Whales, are they fish or mammals? So we feed in a corpus of data to train our logistic regression model, and this model will try to fit an S curve on our data using this formula. Remember that the output of a classification model is always a probability score. This is the probability score for a certain outcome, say the whale is a fish, and we classify this output based on a threshold. For our example here, let's assume that the probability of Y that we get at the output is the probability that the whale is a fish. Logistic regression also uses a threshold to calculate outcomes. For simplicity sake, we can assume that this threshold is say 50%. If the output probability from our machine learning classifier basically says that the probability that the whale is a fish is less than Pthreshold, then we classify whales as mammals. On the other hand, if the output probability from our machine learning model says that the probability of whales being fish is greater than Pthreshold, then we classify the whale as a fish.

Training and Prediction Using a Logistic Regression Classifier

In order to cement our understanding of working with scikit-learn to build machine learning models, let's now perform logistic regression to classify our house prices as above the median or below the median. We'll write our code in this new Jupyter notebook called LogisticRegressionForPriceClassification. We'll be using pandas to import the Pandas library and the PyPlot module from Matplotlib. Read in our housing.csv file using pd.read\_csv. Here is a sample of five records from our dataset. This is a dataset that we are now intimately familiar with. Let's clean up and prepare our data as we did before. Drop all records which have missing fields. We now have about 20, 000 records to work with. We've seen from the earlier demo that we have a number of houses whose houses are capped at 500, 001. These might distort our ML model. I'm going to go ahead and drop all of these records from our training and test data. Once we've dropped around 1000 records, we now have about 19, 475 records. These will be part of our training and test data. All of our other features are numeric, except for ocean proximity, which contains categorical values. Let's one-hot encode these values by calling pd.get\_dummies. You should have realized by now that our data preparation and cleaning steps tend to be the same, whether we are building a model for regression or classification. After having one-hot encoded, our ocean\_proximity categories, we now have a housing\_data data frame with 14 columns. Let's sample this and take a look at how our one-hot encoded values look. If you scroll over to the right in the display of this data frame, you can see the one-hot encoded features. When a particular neighborhood is at an ocean proximity of less than 1 hour from the ocean, there is a 1 in that particular column. All other columns are 0s. If a neighborhood is inland, there is a 1 in the column corresponding to ocean\_proximity\_INLAND, all other columns are 0s. This is how one-hot encoding works to encode your categorical data into numeric form. For every neighborhood, we have a median housing value, which means this is typically a dataset that's used for regression. We can use this same dataset for classification by calculating the median of housing prices and then trying to predict whether a particular neighborhood will have a house above the median value or below the median value. This is how we convert our simple regression problem to a classification problem. Call the median function on the median\_house\_value column in order to calculate the median and store it in a variable. The median value of houses in our dataset is roughly 173, 000. We'll now add a new column in our dataset called above\_median. Above\_median is a column with Boolean values. It'll contain the value true if the value of a particular house is above the median, otherwise it'll be false. So subtract the median from every house value and check whether it's always 0. If yes, above\_median will be true, otherwise it will be false. Let's sample the resulting data frame and take a look at what this above\_median column looks like. You can see that it contains true and false values. The record is true if the value of that particular house is above the median value of $178, 000 that we had calculated. Let's now set up the features and Y values or labels of our classification problem. The X values are all of the features in our data frame except the median\_house\_value and above\_median. We create a new data frame, X, by dropping these columns from our original data frame. And the Y values are true or false. We extract this from the above\_median column in our data frame. As you can see, this is a binary classification problem. Our output will be one of two labels, true or false. True if the house price prediction is above the median, false otherwise. Just to remind ourselves, here are the features that we'll use to train our logistic regression classifier. These are all the numeric and one-hot encoded features from our dataset. In exactly the same way as we did in our previous demo, we want to call the train\_test\_split function in order to split our dataset into training data that we'll use to train our model and test data that we use to evaluate our model. Once again, we'll use 80% of the data for training and 20% as a hold out to test our model. If you take a look at the shape of x\_train and x\_test, you'll see that we have 15, 000 records, roughly, for training, and 4000 records for testing. And here are the corresponding shapes of y\_train and y\_test. With our data all set up, we are now ready to perform logistic regression to classify our data. This is binary classification. If you're using scikit-learn, things are easy. Simply import the LogisticRegression estimator object from sklearn.linear\_model. Building and training a model simply involves instantiating this LogisticRegression estimator object and calling fit on our training data, along with the corresponding y labels. The fit function on our estimator object kick starts the training process. There is one interesting thing here, solver is equal to liblinear. Solver specifies to scikit-learn the algorithm that it should use under the hood to solve this LogisticRegression optimization problem. The liblinear solver is a good choice when your dataset is fairly small. And the classification that you want to perform is binary classification, when your output is one of two categories, true or false, 0 or 1. That's all we need to know about the liblinear solver at this point in time. Since both of this is true, our dataset is fairly small with just 20, 000 or so records, and we have a binary classification problem. This is the solver we use. And just like with our linear model, once our model has been trained, we can evaluate it on the training data by calling model.score. We've passed in the x values and the y labels that we use for training, and this will give us a score for our model, which is 0.82. When you're building a model for classification, the default way to score your model is the accuracy of your predictions. How many of the model predictions were correct on the training data? That's what this number gives us. Eighty-two percent of the predictions were correct. There are, of course, other scoring techniques that you can use to measure and evaluate your classifier. We don't need to worry about them now, they're out of scope for this particular discussion. We'll stick with the simplest of all measures, the accuracy of our classifier. How many of our model predictions were correct? Our model got about 82% right on the training data, but how does it perform on test data? Call logistic\_model.predict, pass in x\_test, which contains your test dataset, and save the results in y\_pred. Before we take a look at how our predictions were, let's create a new data frame with the predicted versus actual values. The values or binary classification output by our model are present in the predicted column. The actual values from our test data are present in the actual column. And here is what a sample of the resulting data frame looks like. You can see that the predictions are pretty accurate, except for this one last record at the bottom where our model predicted that the house value was not above the median, but it actually was. Sampling the actual and predicted values is fine, but a way to get the accuracy of your model on test data is to import the accuracy score function from sklearn.metrics. Pass in the labels from your model predictions, as well as the actual labels, and you'll get an accuracy score, and you can see that it's pretty high for this particular logistic regression model, 81.2 %. Eighty-one point two percent of our model predictions were correct.

Summary and Further Study

And with this demo, we come to the very end of this module and to the very end of this course as well. In this module, we discussed and implemented classic problems in machine learning. We performed regression and classification. We used regression to predict continuous data. Regression is typically used when the output is as continuous value. And after that, we built and trained a classification model for predicting categorical data. We first focused on getting a high level understanding of linear regression, which fits a line through our data, and logistic regression, which fits an S curve on our data. And once we've understood these basic concepts, we moved on and implemented a simple linear and logistic regression model using the scikit-learn estimator API. And with this, we come to the end of this course on building our first scikit-learn solution. If you're interested in scikit-learn, there are other interesting courses on Pluralsight that you can watch. Building Clustering Models with scikit-learn will focus on clustering. Employing Ensemble Methods with scikit-learn will talk about ensemble learning techniques, which bring together multiple machine learning models to make a prediction. And finally, Building Neural Networks with scikit-learn will talk about scikit-learn support for neural networks.