Building Machine Learning Models in Python with scikit-learn

by Janani Ravi

Course Overview

Course Overview

Hi, my name is Janani Ravi, and welcome to this course on building machine learning models in Python with scikit-learn. A little about myself, I have a master's degree in electrical engineering from Stanford and have worked with companies such as Microsoft, Google, and Flipkart. At Google, I was one of the first engineers working on realtime collaborative editing in Google Docs, and I hold four patents for it's underlying technologies. I currently work on my own startup, Loonycorn, a studio for high-quality video content. This course is a beginner's course for engineers and data scientists who want to understand and learn how to build machine learning models using scikit-learn, one of the most popular ML libraries in Python. This course covers scikit-learn support for data processing and feature extraction. You'll learn how to use libraries for working with continuous, categorical, ex as well as image data. This course goes beyond ordinary regression models. You'll understand and learn to implement specialized regression models such as lasso and ridge regression. Classification algorithms such as support vector machines and ensemble learning techniques such as gradient boosting and scikit-learn are also covered. In addition to supervised learning techniques, you'll also understand and implement unsupervised models such as clustering using the mean shift algorithm and dimensionality reduction using

principle component analysis. At the end of this course, you will have a good understanding of the pros and cons of the various regression, classification, and unsupervised learning models covered, and you'll be extremely comfortable using the Python scikit-learn library to build and preen your models.

Processing Data with scikit-learn

Module Overview

Hi, and welcome to this course on building machine learning models in Python using scikit-learn. Scikit-learn is an extremely popular open source Python library with implementations for a wide range of machine learning problems, such as classification, regression, clustering, dimensionality reduction, and so on. This is typically the first library that a student encounters when she starts her study of machine learning, which is why we'll start off this course by understanding the different types of ML algorithms and where they might be used. We'll then see how we can work with numerical as well as categorical data. In the world of machine learning there are well understood ways in which we deal with data in a continuous range, and data which can only take on discreet values. We'll see how we can standardize numerical data when we want to feed it as an input to an ML model. Standardization of numerical data is an important prereq for many machine learning algorithms to get a stable and robust solution. ML algorithms only recognize numeric input, but we want our machine learning models to work with text data as well, such as for sentiment analysis. This is an important tree processing step here. We need to be able to represent our text in numerical form. We also want our machine learning algorithms to be able to work with images, which means we ought to know how we can represent pixel intensities and extract other features from images. This module will cover that as well.

Prerequisites and Course Overview

Before we dive into the actual contents of this class, let's see what the prereqs are so that you can make the most of what is covered. We start off by saying that this is an introductory course to scikit-learn. If you've never used scikit-learn before, or you have only a limited idea of what it has to offer, then this course is for you. This course does assume though that you have a basic

understanding of Python programming and are very comfortable with Python constructs. There are many good courses available on Pluralsight that will allow you to get the Python experience that you need for this course. Python: Getting Started, Python Fundamentals are all examples. If you are looking for more, the Advanced Python course is for you. If you are looking for other introductory courses in machine learning, How to Think About Machine Learning Algorithms is a good place to start. This introduces machine learning and covers the different techniques and approaches you might follow. Understanding Machine Learning with Python is also a good course, as is Understanding the Foundations of TensorFlow, which focus on the TensorFlow deep learning framework. Some of the prerequisites are that you be very comfortable programming in Python. We'll be using Python 3 for all the demos in this course. You also need to be comfortable working with Jupyter Notebooks. We won't be diving into the details of how they work. It would be useful for you to have some basis understanding of machine learning. However, this is absolutely not required. This course assumes no prior knowledge. This course starts at the very beginning. The first step for any machine learning workflow is data preparation. This is where you explore your data and convert it to a form that your ML model understands. We'll see how we can represent text as numbers, how we can represent images as matrices and so on. In the following module we'll talk about regression. We'll talk about the simple linear regression using the alternate release squares method, and compare and contrast that with specialized regression models such as Lasso and Ridge regression, and Support Vector regression. We'll have hands on demos for all of these. A widely used machine learning model for classification is support vector machines. Support vector machines are the basis for support vector regression that we say in the previous module. We'll see how SVMs work for text and image classification, and we'll then move on to gradient posting. Gradient posting is an example of ensemble learning, which takes takes many weak machine learning modules and brings their output together to get a more robust implementation. And finally we'll cover clustering and dimensionality reduction in scikit-learn. These are unsupervised learning techniques which allow you to find interesting patterns in your underlying data. We specifically cover mean shift clustering and principal components analysis.

Machine Learning Use Cases and scikit-learn

A one sentence explanation of what exactly machine learning is is that it's an algorithm that is able to learn from data. You might have a huge maze of data, data that is connected from the various clicks on your product website, data that is stored in from business reports and so on. In order to discover anything interesting at all within this data you need techniques such as machine learning. You'll find patterns within this data that will allow you to make intelligent decisions. A

classic machine learning use case that you might be familiar with is emails on a server. We all receive emails of all kinds. Some of them are important, some of them are junk. Machine learning algorithms have proven to be extremely useful at identifying these emails as spam or ham, and based on this classification moving them to trash or to your inbox. Machine learning is also widely used to identify objects within images. Images are represented by pixels, and neural networks can identify the different parts of these images, corners, edges, shapes and so on, and bring them together to identify say the photo of a little girl. A large number of the most commonly used machine learning algorithms can be broadly divided into these four categories. The first of these is classification. Is the email spam or ham? Was the credit card transaction fraudulent or legit? Is this a photo of a dog or a cat? Machine learning is also used for regression where you predict a value from a continuous range of numbers. What is the price of an automobile given that you know it's make, model, the fittings that it has, the engine horsepower and so on? How much would a 2, 000 square foot house 30 miles from the city center cost? If you have a large input data set, you can find patterns within the data itself. That's where clustering algorithms come in. Let's say you have a wide variety of emails, and you want to fit these emails into individual topics. Emails which are about technology, emails which are about personal stuff and so on. That is an example of clustering. If you've ever worked on an ecommerce product, you know that a very great driver of traffic is the recommendations that the site shows to its users. Recommendation systems are examples of rule-extraction algorithms. If you shop for mobile phones, are you more likely to buy tablets? Let's understand some basic concepts of machine learning by using an example of classification. We want to figure out whether whales are fish or mammals. Now this particular classification can be pretty confusing because whales are members of the infraorder Cetacea. They also give birth to young and feed them. But whales also look like fish, swim like fish and live in the sea with fish. Now in order for your machine learning algorithm to understand and make the decision as to whether whales are fish or mammals, it needs to lean on a huge corpus of data where it can understand the various characteristics of a variety of different living things, and whether they are fish or mammals. This is our training purpose, and is fed as an input to our machine learning classification algorithm. Once we have the training data, we need to make a choice of our classification algorithm. Which model do we want to use? Do we want to use support vector machines, naive-based decision trees, neural networks and so on. Once this corpus of data is fed to our algorithm, the algorithm has been trained and it gives us an ML-based model. This is our machine learning classifier. So here we have it, an ML-based classifier trained on this corpus of data, through which we feed in our input information, information about the whale. Let's say we give it information that the whale breathes like a mammal and gives birth like a mammal. It's likely to classify the whale as a mammal. This data that we fed in as an input to our

machine learning model is called the feature vector. It's also called an S instance or a prediction instance because it is one record for which we want a prediction result. The output of the machine learning model, which is our classification output, is called the label or the predicted label. Now let's say we have the same machine learning classifier, and we feed it some different information about the whale. We tell it that the whale moves like a fish, looks like a fish and lives in the sea with other fish. Then it's likely that this ML-based classifier will think that the whale is a fish. The prediction of our model has changed completely based on what data we actually fed in as its input, so the prediction of our model is sensitive to the input feature vector. We focus the attention of our machine learning model on features that were misleading. The input feature vector spoke about how the whale was like a fish, which resulted in an incorrect prediction. It predicted the whale to be a fish. Here the predicted label is not equal to the actual label. As we all know, the whale is actually a mammal. And this is how traditional machine learning models behave. Traditional ML-based systems rely on experts, that is you, to decide what features they need to pay attention to. If you direct their attention to the wrong features, you'll get incorrect predictions. Some examples of traditional machine learning models are the linear regression, Lasso and Ridge regression, support vector regression, all of which we'll cover in this course. Examples of traditional classification models are Naive Bayes, SVM, decision trees. Machine learning research is constantly making advances, and we now have representation ML-based systems which figure out by themselves what features are important in the input data, what features the model needs to pay attention to to get the right result. A common example of representation ML-based systems are deep learning models such as neural networks. Scikit-learn, which is a very popular open source machine learning library built in Python, has some support for neural networks. However, it is primarily used for working with traditional ML-based models. You can learn more about scikit-learn if you visit its website at scikit-learn. org. You can see that it's built using other Python libraries that data scientists find very useful, such as NumPy, SciPy and matplotlib. And here are all the categories of algorithms that scikit-learn supports. Classification, regression, clustering, dimensionality reduction. It also offers tools which allow you to find the best possible model for your data, and a number of tools and utilities for data processing and preparation.

Supervised and Unsupervised Learning Techniques

If you have been studying and following machine learning, you might have heard the terms supervised and unsupervised learning. These are two broad categories that all machine learning algorithms are divided into. Supervised learning algorithms have labels associated with the

training data that we feed into the model, and these labels are used to correct the algorithm and tweak them until we get a good model. Data fed to unsupervised learning models do not have any associated labels which can be used to correct the algorithm. Instead, the model has to be set up exactly right to learn patterns or structures within the data itself. Classification models, which we spoke about briefly in the last video clip, are examples of supervised learning models. If you want to classify whales as fish or mammals using an ML-based classifier, you need to set up the classifier to work in two different phases. The first is the training phase where we feed in a large corpus of data, all of which have been classified correctly. This is the data that our model uses to learn from. Once the model has been trained, we'll run it in its prediction phase. We'll use the model to classify new instances which it hasn't seen before. During the training phase we'll feed in a large corpus of correctly labeled instances. For each of these corpus instances our ML-based classifier will give an output classification. We'll compare out model prediction with the actual prediction or or the actual labels that we have associated with the training data, and we use this as feedback to tweak our model parameters. This is the loss function or the cost function, and it's primary purpose is to improve our model parameters and build a stronger model. Once training is complete our model is ready for prediction. This is where we feed in our input data, such as characteristics of a whale, which will allow our model to identify it as a mammal. A typical supervised ML algorithm has X variables, and these are the attributes that our algorithm focuses on. These are called features, and every data point is a list or vector of these features, which is why they are called feature vectors. So if you heard the terms X variables, feature vectors, features, these are all the input to our supervised learning model. On the other hand, supervised learning models also have Y variables. These are the attributes that our algorithm tries to predict. Y variables are also called labels. Depending on your use case and the kind of ML algorithm that you choose, you can have different kinds of labels. Your labels can be categorical. That is typically the output of a classification model. Categorical labels take on discrete values, days of the week, months of the year, spam or ham, fraudulent or not. Or the label output of your machine learning model can be continuous, such as the output of regression models. Continuous value labels are typically numbers within a certain range. Y variables labels or predicted values are the output of our machine learning algorithm. The entire objective of supervised learning techniques is to learn the function F that links the features represented by X with the labels represented by Y. The most common ML technique is linear regression where you have a bunch of points represented in space, and you want to find the best fit line that passes close to all of these points. In linear regression the function F that links our features and labels is a linear function represented by Y is equal to Wx plus B. If you remember you high school mathematics, this is the equation of a straight line where B is the intercept, and W is the slope. Linear regression is the simplest possible

ML example, though. You have a really complicated machine learning model such as neural networks, which can learn or reverse engineer pretty much anything. This of course presupposes that we are feeding our model the right training data. Machine learning techniques can also be unsupervised, which means we do not have labels associated with the training data. We only have the data itself and unsupervised techniques find patterns or structure within this data. Unsupervised learning techniques do not have Y variables. They only have the input features that go into our model. They do not work off of a labeled corpus. This is no explicit training phase. Let's quickly contrast this with supervised learning where we have the input variable X and the output variable Y, and the job of the model is to learn the mapping function F, which makes X correspond to Y. The mapping function that the model discovers is an approximation, and what we are trying to do in our training process is to improve this approximation. This mapping function exists so that when we feed in new values of X, we are able to predict Y, and this is where the labeled corpus comes in. We use the existing data set to correct and tweak our mapping function to improve our prediction. Contrast this with unsupervised learning techniques where the only input that you'll feed into your machine learning model is the input X data, the features themselves. There are no corresponding labels. The objective of the model is to find patterns within the data itself. It will model the underlying structure to learn more about the data. The algorithms are forced to self discover patterns inside logical groupings and so on. A very popular unsupervised learning technique, which applies to many use cases, is clustering where we use algorithms such as K-means, mean shift, hierarchically classing etc., to find logical grouping within data. Unsupervised learning techniques are also used for dimensionality reduction of your input data set. We want to identify the significant factors that drive data. In fact, dimensionality reduction algorithms such as principal components analysis often serves as a pretraining step for other supervised learning techniques such as classification or regression.

Demo: Useful Python Packages

Here are some things you need to know before you get started with demos in this course. I am using the Anaconda three distribution from jupyter. org. I'll be writing all of my code in Python 3. We'll be using Jupyter Notebooks as our Python shell. Jupyter Notebooks are great for demos because they allow you to show comments, code and the results of running that code in one interactive shell. Jupyter runs on both Windows and Mac machines. You can go ahead and pick what environment you want if you don't want to use Jupyter. Here are some common Python libraries that I'll be using. The first is NumPy, which gives me a powerful and dimensional array object, and this array is perfectly compatible with (mumbling). We'll also use SciPy for certain

statistical calculations. We won't explicitly be using this package in this course, but it's a useful one to have around. We'll use pandas to work with tabular representations of data. And finally, matplotlib to plot (mumbling) graphs in our code. You can get all of the packages installed onto your local machine using pip3, pip if you're using Python 2. 7.

Mean and Variance

There are two broad types of data that machine learning models have to work with, continuous data and categorical data. Continuous data can take on an infinite set of values within a certain range, such as the height or weight of an individual, the income and so on. Categorical data, on the other hand, can only take on a finite set of values. True or false, male or female, day of the week, month of the year and so on. The output of many common machine learning models are binary, true or false, male or female, left the company, stayed in the company, fraud or not. Categorical variables that take on just two values are called binary variables. Working well with continuous variables requires some basic understanding of statistics. Nothing beyond high school, don't worry about that. Certain machine learning models work well when the input data into these models is standardized. For this we need to understand mean as well as variance. If you take a look at all the points that you can see on screen, you can see that they belong to a single number line. The mean or average of these points is a single number which best represents all of these data points. The mean is calculated by summing up the values of all of these points and dividing by the total number of these points. The second most important piece of information about these data points is the variation. These answer the question, do the numbers jump around? If yes, how much do they jump around, or are they confined to a particular range? The range is one way to express variation in data. Here we subtract the value of the lowest point from the highest. The range is not the perfect way to express variation in data. It completely ignores the mean, for one, and it's heavily swayed by outliers in data, the highest and the lowest values, which is why we need another statistic. That is variance. If the mean of the data points is the headline, then the variance is the second most important number to summarize any set of data points. Calculating the variance requires a series of steps. The first of these is to calculate the mean deviation of the data points. Mean deviation refers to how far away every data point is from the average of those data points. Squared mean deviation simply squares this result. This is the square of the distance of every data point from the mean, and now we can finally calculate the variance of these data points. Variance is the sum of the squares of the distances of every individual data point from the mean divided by the total number of data points. It has been mathematically proven, and we don't need to go into that in this course. The estimate of the

variance can be improved by tweaking the denominator of this function. This tweak is called the Bessel's Correction. So instead of using N for all of the data points, we'll simply use N minus 1 as the denominator. Understanding the mean and variance of data is typically one of the first things that you study in statistics. That's because mean and variance succinctly summarize a set of numbers. Another common term that you might have come across is standard deviation, which once again measures how much the numbers jump around. Standard deviation is nothing but the square root of variance. Now let's assume that you have a number of different collections of data points, that every collection represents something different. One collection could represent the height of individuals, another the weight of individuals and so on. Here every collection is represented as one column of data. If you want to be able to compare and contrast these different collections of data points in some manner, you need to standardize them. Standardizing typically involves finding the average of these individual columns, and then finding the standard deviations of these columns. Remember these are the two values which succinctly summarize our data points. We standardize every column of data points by subtracting the mean from every individual point and dividing by the standard deviation. By applying this to every collection of data points, we'll essentially have every column of the standardized data with mean zero and a variance of one. A standardization of data allows you to bring different collections of data points into the same form, making it easier to understand the data and perform comparisons. Data standardization is a very common technique you'll see when you're feeding input data to machine learning models because many techniques work best on standardized data. If you perform standardization of your input X features, it will prevent some high variance data series from dominating the output of your machine learning model. The resulting ML model will be more robust. Examples of techniques which benefit from standardization are principal components analysis and Lasso and Ridge Regression. We'll cover all of these techniques in this course.

Demo: Scaling Numeric Data

We've done a lot of theory so far. Let's start working with numerical and categorical data in scikit-learn. If you're using the Anaconda three installation like I am, scikit-learn is a part of Ananconda three, and there is no additional install that you need to do to get this (mumbling). If not, you'll need to perform a pip install. Pip3 install sklearn will scikit-learn library. All the code I'll write today will be under this demo directory. A simple ls will show you that there are two directories under here, one for code and one for the data sets that we'll use in our demos. Here are all the data files for the different data sets. We'll be using all of these, and we'll be referencing them from this data directory. All our code is present in the code directory. This is the directory where I'll start the

Jupyter Notebook server. Remember to install Anaconda 3 from jupyter. org if you don't have it already. Here is how we can set up a new notebook. Click on New at the top right corner, and choose the Python 3 kernel. This will create a new Jupyter notebook, which you can then rename by clicking underneath. This is called mi-demo1-CategoricalAndNumbericData. We'll first import a few Python libraries that are useful, such as pandas. You can click on the Run button on the title, or use Shift+Enter in order to execute code in a single cell of your Jupyter Notebook. In order to demonstrate how we can work with numerical and categorical data, we'll use a very simple exams data set. We'll read in this data set from our local machine by using the pandas read_csv function. This data set contains students course for math, reading and writing, along with other details about the students. All of these course seem to be out of a hundred, but it's totally possible that different exams have different scoring patterns. Math may be out of a hundred, reading may be out of 50, writing may be out of 20 and so on. Standardization is an important technique that we can use to compare these scores. Let's start off by finding the mean of each of these scores. In pandas you read in the entire in tabular form. You can simply access an individual column and call the mean function on it. The average of our math, reading and writing scores are as you see on screen. Now scikit-learn offers a very useful toolkit called preprocessing, which you can use to standardize your data set. All you need to do is call the scale function within the preprocessing library, and pass in the column that you want standardized. The scale function will divide all the data points, for example the math scores, by the average and subtract the standard deviation. After standardization if you now examine your data, you'll find that all of the scores will be within the same range. You can now find the average of these standardized scores. The average should be equal to or very close to zero because it subtracted the mean and divided by the standard deviation, and you will find that this is indeed the case. The mean for each of these sets of scores is zero or almost zero, and the standard deviation will be one. There are other classes and library functions that you can use within scikit-learn to scale your data, but preprocessing that scale is the most common.

Categorical Data and One-hot Encoding

Categorical variables which take on discrete values may need special treatment and preprocessing before you can feed them into a machine learning module. This is because machine learning modules can only accept numeric data. Let's understand a few characteristics of categorical variables first. Continuous data can typically be ordered. Categorical data, maybe not. In some cases if you want to classify the output of your module as low, medium, high, there might be some inherent ordering, but that's not always the case. Because ML algorithms are

constructed to only accept numeric values, categorical data will require special preprocessing to be encoded as numbers, and there are a variety of different ways in which you can do this. The numerical encodings of categorical data should never be ordered, though you might have numerical representations which imply some order. For example, a zero for low, one for medium and two for high. Such ordered numeric representations of categorical values are typically for our understanding. Our ML model won't use them in any way. There are many ways in which categorical data can be encoded as numbers. Simply mapping categories to numbers is one of them, but another common way is one-hot encoding. Consider the days of the week, Sunday through Saturday. One-hot encoding will assume that each of these days represent one position in an array, with Sunday at position zero and Saturday at position six. If you want to have a numeric vector which represents Monday, it will have the number one at the position Monday. The numeric vector for Monday will have the number one at index position one. The numeric vector for Thursday will have the number one at index position four. All the other numbers will be zero. Similarly, Saturday will be represented by the number one at index position six. All the other numbers in that vector will be zero. One-hot encoding identifies discrete values by the location of the number one in the feature vector.

Demo: Representing Categorical Data in Numeric Form

We will continue with the same demo that was saw earlier. We will see how we can encode categorical variables as number is scikit-learn. An extremely useful library function that scikit-learn offers is the label encoder. This is typically when there are meaningful comparisons between the categorical variables, the low, medium, high example that I've spoken of earlier. These comparisons for categorical variables are meaningful only to you as a developer, and not to the machine learning model. The label encoder offers a fit transform method which will go through all the categorical values in the gender column and transform them to numbers. The numerical values for gender will be assigned to the gender column in our pandas data frame. Examine the data and you can see that the gender column will now be made up of numbers. The number one represents a male, and the number zero represents a female. The classes underscore variable in the label encoder will give you the unique values of gender that that column holds. The pandas library offers a very easy way to convert your data to one-hot representation, which is why we'll use that instead of scikit-learn here. The get_dummies method allows you to represent any categorical value in its one-hot form. In our data set every student has an associated race, which can be anything from group A to group E. The race of the student identified by row number zero is group E. The student at row one belongs to group C. You can assign the one-hot

representation of race to be part of our tabular data in the exam_data data frame. If you now exam the data frame, you'll see that race ethnicity information is now represented in numeric form using one-hot notation. The get_dummies function can work on multiple columns at the same time. Here we want the parental level of education, lunch and test preparation course all to be represented in one-hot notation, and we can do that with a single command. In the resulting data frame here is the one-hot representation of the parental level of education, whether they have an Associate's degree, Bachelor's degree, high school diploma and so on. Similarly the categorical values of the lunch column, whether the students qualify for reduced lunch or standard lunch, is also converted to one-hot form, and test preparation course, whether they've completed it or not, is also in one-hot form.

Representing Text in Numeric Form

Machine learning problems are very commonly used with X input, especially in the case of problems which involve sentiment analysis. This is the worst restaurant in the metropolis by a long way. We want to figure out the sentiment of this statement. The input here is in the form of a sentence. Now machine learning models, all of them including neural networks, only process numeric input. They don't work with plain text, which means we have to figure out a way to represent this sentence in numeric form. This sentence, which we can think of as a single document, can be modeled as an ordered sequence of words. The first step to representing this document in numeric form is tokenizing the document into individual words. Every individual word in this sentence can be modeled as a number. So let's say this is word W0. It will have some numeric encoding, which is X0. Likewise, every word in this document will have its own corresponding numeric encoding. Every sentence can then be represented as a vector of numbers where every number corresponds to an individual word. So the entire document can be represented as a tensor. A tensor is basically just a matrix with N dimensions, where N can be anything. Now the big question is how can we best represent words as numeric data? Do we just randomly assign numbers to individual words? Or is there a science to it? The numbers used to represent individual words are called word embeddings, and there are three ways in which this can be done. The first is one-hot representation. We are already familiar with that. The second is frequency-based embeddings, and the third, prediction-based embeddings. Deep learning framewords such as neural networks use prediction-based embeddings where the numeric representations of X capture the meanings as well as the semantic relationships between words. However, this is not going to be covered in this course. It's an advanced topic. Before we move on to frequency-based embeddings, let's quickly look at how we can represent words using the one-

hot format. Consider all of these reviews with the number of different words. Here uppercase D is used to represent the entire corpus of reviews, and B of I is one review in the corpus. The first step is to find all words that exist within our corpus of documents. This is a set of all words. The words will not be repeated. Once we have the master vocabulary of all the words in our corpus, every review can be represented as a couple of 1-0 elements. The review with the single word "amazing" will have one corresponding to that word from our vocabulary. The "worst movie ever" will have three one's corresponding to those words, and "two thumbs up" will have three ones corresponding to those words as well. The one-hot notation has several drawbacks, which makes it not very suitable for X representation. If you have a very large vocabulary, your feature vectors will be enormous. If you have 10, 000 words in your vocabulary, every feature vecture will be 10, 000 elements long. You've also completely lost information on the order in which those words occur. You've lost the context in which a particular word was stated. One-hot notation also causes the loss of frequency information. You don't know whether the word amazing was said once, twice or thrice in a particular review. One-hot encoding is a pretty dumb representation of words within a sentence. It does not capture an semantic information or relationship between the individual words.

Frequency Based Encoding: Count Vectors

In order to preserve frequency information of individual words in text, you can use frequency-based embedding, and that's what we study next. There are several different ways in which we can generate frequency-based representations of words in a document. We'll look at two of the most common ones. The first is count vectors, which tracks how many times a particular word occurred in one document. The second is the TF-IDF algorithm. This stands for term frequency-inverse document frequency, and it's a coding algorithm which tracks how important a particular word is to the document and to the corpus as a whole. Count-based embeddings capture how often a particular word occurs within a document. These are the word counts or the frequency of the words. So let's say we have two reviews of movies. The movie was bad, the actors were bad, the sets were bad. The first step is to tokenize each document into its individual words. Create a vocabulary of all the words which occur in our document corpus, and then we express each review, that is each document, as a frequency of the words which appear in that review. The movie was bad, here every word occurs exactly once. The second review which says, "The actors were bad, "sets were bad", the word bad appears twice, which is why its frequency count is set to two. Now an important drawback of frequency-based encoding is the fact that if you have a very large vocabulary, you'll have enormous feature vectors, which are sparse because not all words

will be present in each review. An alternative that you can choose to have an upper bound on the size of your feature vectors is by saying you'll only consider the top N words based on frequency. So the top N most commonly used words are the only ones you'll represent in your vocabulary. You can mark the remaining words as unknown. The drawbacks of frequency-based encoding are similar to one-hot encoding, the large vocabulary, the fact that it is unordered. You don't know in what order the words occurred in the original text. We've lost the semantic and word relationships that might have existed. Semantic and word relationships are captured only by advanced predication-based techniques, which we won't cover here. Let's focus on the large vocabulary. There is another way to mitigate this, and that is by hashing individual words to buckets. The number of buckets that you choose will be the size of your vocabulary. Let's say that the original number of words that you had in your vocabulary was 10, 000, and you choose to use 8, 000 buckets. This means you have reduced the feature vector representation for every document. There is a trade off here. Make sure that you choose enough buckets so that collisions are rare. It's rare that two words map to the same bucket. Let's take a look at this example here. Let's say that the word actors and the word sets hash to the same bucket, and this bucket was represented by an integer. The word actors as well as the word sets both hash to bucket four. Our feature vector will have just one entry for this bucket, so we won't have two instances of four. We'll just have one, and we'll sum up the frequencies. So actor and sets both hash to the same bucket, and the bucket has a frequency of two in our second review. We've lost some information. We don't know that these were two different words originally, but we've reduced the size of our feature vector.

Frequency Based Encoding: TF/IDF

Before we move on to an actual demo, let's first understand how the TF-IDF algorithm works in frequency-based encoding. This captures two pieces of information, how often a particular words occurs within a document, as well as how often it occurs across the entire corpus. We represent a document in numeric form as a feature vector by tokenizing the document into words and representing every word as an integer. This tokenized document is a tensor. A tensor is nothing but a matrix. In the TD-IDF representation every word is encoded using a score, and this score is a multiplication of two terms, the term frequency and the inverse document frequency of that word. The TD-IDF algorithm is based on two principles. If it sees that a word occurs more frequently in a single document, it assumes that word is important, and it upweighs the score of that word. On the other hand, if it notices that the word occurs very frequently across the document corpus, it assumes that it is probably a common word such as "a", "an", "this", "that"

and so on. It down weighs the importance of such words. Thus the encoding of a word I in a particular document J depends on the word, that document and also on the entire corpus. Every word in every document is assigned two scores which are then multiplied together. The first of these is the term frequency, which is a measure of how frequently word I occurs in document G. The more often a word occurs, the higher its score. The second score is the inverse document frequency, which is a measure of how infrequently a word I occurs across the entire corpus B. The more common a particular word is across the document corpus, the lower its IDF score. Thus the TF-IDF score for a particular word will be high for word I in document J if the word occurs frequently in that document, but rarely in other documents in the corpus. Using TF-IDF as our word encoding algorithm has several important advantages. The first of these is that the feature vector used to represent a document is much more tractable in size because it does not depend on the size of the vocabulary. In addition, every score captures the frequency as well as the relevance of that word. An important drawback that still remains is that the context in which a particular word was mentioned is not captured.

Demo: CountVectorizers, TfidfVectorizer, HashingVectorizer

In this demo we'll study some scikit-learn libraries which allow us to represent X data in numerical form. This includes the CountVectorizer, the TfidfVectorizer, and the HashingVectorizer. Every vectorizer implements a fit method to which we pass in a corpus of documents. This method generates unique IDs for all words in the corpus. This is where the vectorizer learns the vocabulary of our input data set. A vectorizer also implements the transform method to which you pass in some input data. This is used to assign the generated unique IDs to words in the corpus that we just passed in as an input argument. Using the fit method and the transform method separately makes sense if you want to generate unique IDS using one corpus, and assign those IDs to another corpus. If you're working on just one data set, you'll typically use fit and transform to generate unique IDs and assign it to words in that corpus. This is the second demo in this module. You'll find the code for this in M1 Demo two. Let's import some libraries from scikit-learn that we'll use. The first is the CountVectorizer. This is a basic frequency-based representation of words in documents. We'll work with a very simple corpus of data so that we can see exactly what's going on. This corpus has four documents, which we've specified in the form of an array. Initialize the CountVectorizer class and generate a bag of words in numeric form. Simply set up the CountVectorizer, and then call fit_transform on our corpus of data. The bag of words is a sparse 4 by 12 matrix, four documents and a total vocabulary of 12 words. Print out the contents of this bag of words. Notice that every word is identified by the document in which that

word occurred. There are four documents in our input corpus, which is why every document is given a unique ID from zero all the way to three. Every word in our document corpus also has a unique individual ID. Here are the five words in document one, the five words in document two. If you look closely at document one and two, you'll see that the only word that is different is the word "first" and "second". Those are represented by different integers. The other words in these two documents are the same. Their integer representations are also the same. Notice that word frequencies are captured in this bag of words representation. The word "four" appears twice in the last document. It's frequency is two. You can access the ID that corresponds to a particular word by calling vectorizer. vocabulary. get on that word. The word document corresponds to ID zero. Typically the most frequently used words are assigned lower IDs. The word document appears four times across our corpus. It's the most frequently used word. Vectorizer. vocabulary will give you access to all words in our vocabulary. Here are the 12 different words. Let's view this bag of words in a tabular format in order to understand how exactly it is set up. We'll use the pandas dataframe for this. This dataframe will be in the form of a 2D array where the rows are the individual documents and the columns are the words in our vocabulary. The numbers which are present in the cells of this tabular format represent the frequency of individual words in each document. Let's now look at the TF-IDF vectorizer. This associates scores with every word in our document corpus. Here is a bag of words representation which is the output of the TF-IDF vectorizer. Every word in every document is associated with a score. Every document has a unique ID. Every word has a unique ID as well, and a document ID word ID combination is associated with a score. You can access the IDs assigned to individual words just like you did before. You can also represent this in a dataframe format. The cells of this dataframe contain TF-IDF scores and not word frequencies. And finally here is our complete vocabulary, all of 12 words. If you have a very large vocabulary of words, we can choose to use the HashingVectorizer rather than the CountVectorizer. The use of hashing buckets to represent words allows us to scale large data sets when we use the HashingVectorizer. The input argument to this vectorizer is the number of hash buckets, which in our case is set to eight. The result you see on screen is the numeric representation of all the words in our four documents. Notice that word IDs are from zero to seven because we have a total of eight buckets. Because the size of our vocabulary is larger than the number of buckets, which is how it should be, multiple words can hash to the same bucket. One disadvantage of the HashingVectorizer is that there is no way to get back to the original word from its hash bucket value. The frequencies of each token is not represented in raw number form. This is some kind of normalized form.

Representing Images in Numeric Form

A very common input to machine learning models are images, and images are typically represented as a matrix. Every cell in this matrix represents a particular pixel in the image, and each pixel holds some value based on the kind of image this is. For a color image, every pixel is represented using three separate values. These are RGB values where each value is a number between zero and 255. For example, the color red will be represented by 255, 0, 0, the color green by 0, 255, 0, and the color blue by 0, 0, 255. Color images are called three channel images because each pixel requires three values to represent its information. Grayscale image, on the other hand, require just one value to represent the information in one pixel. This value represents the intensity of that pixel, and is typically a number between zero and one. It can be a number between 0 and 255 as well. We typically divide it by 255 to get a number between zero and one. A pixel could have an intensity value of 0. 5, as an example. One value to represent intensity, a single channel image. Single channel grayscale images and multi-channel color images can both be represented by a three-dimensional matrix. The first two dimensions represent the height and width of an individual image, and the third dimension represents the number of channels. The dimensions of the matrix representing the grayscale image on the left may be six cross six cross one. The dimensions of the matrix representing the color image on the right will be six cross six cross three. Typically in machine learning all the images that you deal with are of the same size, in which case you can use a single four-dimensional matrix to represent a list of images. Let's consider a matrix with four dimensions, 10, six, six and three. The last dimension is the number of channels in an image. A three-channel image is a colored image. The second and third dimensions are the height and width of every image in the list, and the first dimension represents the number of images. Image processing can get very specialized indeed, and there are multiple algorithms that deal exclusively with images, which is why scikit-learn has a special library called scikit-image, which is a collection of algorithms for image processing. That's not covered here in this course. That requires a course of its own. We'll use open CB for image processing in our demo that's coming up.

Demo: Extracting Features from Images

In this demo we'll see how we can extract features from images. We'll focus on color as well as grayscale images. We'll perform all the image processing using the OpenCV library. The first step is to install the OpenCV library. You can use the pip install command for this. OpenCV-Python is what we need. Make sure you specify bank in front of the pip install if you're doing it from within a Jupyter Notebook. OpenCV is a powerful and easy to use image processing library. You can

learn more about it at the URL below, if you are interested. Go ahead and import the CV2 module, and read in a jpeg image of a dog that's present in our data directory. Use the library function cs2. imread and specify the image path. We'll use the plot library from matplotlib in order to display this image to screen. Matplotlib inline will allow us to display this image inline within our Jupyter Notebook. Here is our very cute dog image in color. The image variable where this is image is stored is simply a matrix. Let's print out the shape of this matrix, and you can see that this is a color image, 130 by 173 with three channels. You can print out this 3D image matrix and see its individual pixel values. Notice that every pixel has an RGB value, and here is the RGB value for the very first pixel. OpenCV allows you to resize images. Let's resize this to a 32 by 32 image, and display it using matplotlib. You'll see that this is still a color image, but its height and width have been changed. You can view the three-dimensional matrix. We've seen that before. In some cases you might find that when you feed an image into your model, you might want to represent it as a 1D array. This can be done using the flatten operation on your image matrix. The flatten function flattens all of the dimension of the array. There were three dimensions in the original array. The final array has just a single dimension. It is a vector, and the length of that vector is 3072, which is 32 multiplied by 32 multiplied by three. You can also read in an image as a single channel grayscale image. Specify the imread_grayscale option. Go ahead and view this image using matplotlib, and you can see it's the same dog, but this time in grayscale. The shape of the image will be different, though. You will see that it is a 130 by 173 image, but the last dimension is one because this is a single channel image. The last dimension isn't explicitly specified here because it's represented as a scaler. If you display the actual image matrix, you will see that there is a single intensity value for each pixel, and this intensity number is between zero and 255. You'll work with matrices a lot when you're working on machine learning, which is why the NumPy library is so important. It makes working with arrays very, very simple. Here we use the expand_dims function to basically expand the third axis, the axis at index two for our image. This will add an additional dimension to represent the pixel intensity. So now the shape of our image will be 130 by 173 by one. You can now confirm how the matrix looks by printing it to screen. Notice that the final scaler value is within an additional dimension now. And here we come to the very end of this module. In this module we started off by understanding the different types of machine learning algorithms and the use cases where they make sense. We also saw how we could prepare input data to feed into our machine learning models. We worked with numeric as well as categorical data. We used the mean and variance of numeric data to standardize information. Categorical data can be represented as numbers by simply assigning integers to specific values or using one-hot representation. Machine learning models only work with numeric data, which means we have to convert text to numbers. This can be done using word encodings.

We studied specific scikit-learn libraries, the CountVectorizer, the TF-IDF Vectorizer and the HashingVectorizer. We also saw how we could work with images, which can be inputs to our machine learning models. We saw how we could extract features from color as well as grayscale images. This was all about data preparation. In the next module we'll actually build machine learning models. We'll focus on building specialized regression models such as Lasso and Ridge and support vector regression.

Building Specialized Regression Models in scikit-learn

Module Overview

Hi, and welcome to this module on building specialized regression models using scikit-learn. Regression is a very common machine learning technique which is used to predict an output which is a continuous variable. The price of a house given where it's located, the length of a sports person's career given his health and fitness, these are all examples of regression. We'll talk about regression models and how we measure the fit of a model to its underlying data. When you're building your machine learning model it's possible that your model does very well during training but performs poorly on the test data. Such models are called overfitted models. We'll speak about those and the bias variance trade off that is inherent when you build any machine learning model. Lasso and Ridge regression are alternative methods that you can use. Beyond ordinary least squares regression these mitigate the problem of overfitting. We'll also study another form of regression called support vector regression. These are built using the same principles as support vector machines for classification but use a different objective function.

Ordinary Least Square Regression

Regression is one of the most common use cases for machine learning. Regression is where we try to predict an output value from a continuous range based on input features. Regression is the machine learning algorithm that you'll use when your problem statement is of the form X causes Y. Here X is the cause or the independent variable and Y is the effect or the dependent variable.

The cause is also sometimes referred to as the explanatory variable. For simplicity's sake let's assume X and Y are both numbers and they can be plotted on a two dimensional plane with X on the X axis and Y on the Y axis. Linear regression involves finding the best fit line that passes through all of these data points. You can think of this line as modeling the X and Y data such that we can figure out the Y value given an X value using the linear equation Y is equal to A one plus B one X. So how do we find this best fit regression line? That is our machine learning problem. Let's consider two separate lines, line one and line two. Line two has the linear equation Y is equal to A two plus B two X. Now intuitively you might say that line one is the best fit regression line. Let's formalize this mathematically. The best fit line can be found by minimizing something called the least square error. This least square error is the objective function of any linear regression. Let's understand this concept of the least square error by dropping vertical lines from every point in our data to the lines one as well as two. The best fit line is one where the sum of the squares of the length of these dotted lines is minimum. Let's parse the statement. Let's consider the lengths of the dotted lines. These lines are longer when they're dropped or projected onto line two. We find the lengths of each of these vertical lines and find the sum of their squares. We do this for all possible regression lines that we can draw on this plane and find that line for which the errors are at a minimum. That is our best fit regression line. This is what it means when we say we want to minimize the least square error. This technique will give us our best fit regression line. That is line one. Another common term that you might have heard when talking about regression is the residual of a regression. Residual are the difference between the actual value of a data point and the fitted value of the dependent variable on the line. On screen you can see there is a difference between Y actual and Y fitted on the regression line. That is a residual. Solving the linear regression machine learning problem requires us to minimize the sum of the squares of the residuals. A regression model which uses this as its objective function is called an ordinary least squares regression.

Measuring Fit Using R-squared

How do we know whether we have good regression model? Our model is said to be good if it fits our data well. We need a measure for that. Let's understand regression as an optimization problem first before we start measuring fit. A linear regression line is represented by the equation Y is equal to A plus B X where A is the intercept and B is the slope of the line. We find the best fit regression line for our underlying data by trying to minimize the mean square error. For any given input X the predicted value Y for that input is called the fitted value of a dependent variable. The actual Y value for the point X might be different. The fitted line given by Y is equal to A plus B X

gives us a different set of values called fitted values. If you look at this image on screen here you'll see that the point in green which is on the regression line is the fitted value for Y I. All of the X values of our data points will have corresponding Y values on the regression line. All of these are their fitted values. Residuals can now be expressed as the differences or the errors between the actual and fitted values of the dependent variable Y. We've spoken about how the regression line tries to minimize the mean square error. The mean square error is nothing but the variance of the residuals. Thanks to the vast array library functions that we have for calculating all of this you don't need to understand the exact mathematics behind this. It's important to understand the intuition though. The regression that we've spoken of so far is simple regression where we have a single cause and a single effect. The MSE minimization technique extends to multiple regression as well where data is present in more than two dimensions. You can have multiple causes for a single effect. Simple regression has one independent variable which is the explanatory variable. Multiple regression has multiple independent variables. The input feature set has more than one feature. The simple regression line has an equation of the form Y is equal to A plus B X. The multiple regression plane has an equation of the form Y is equal to A plus B one X one B two X two plus B three X three and so on. When we solve the linear regression problem using the ordinary least squares method we get the best linear unbiased estimator, or BLUE estimator. It's called the best because the coefficients of the X variables have minimum variance and it's called unbiased because the residuals have zero mean and are uncorrelated with each other. So, how do we measure how good our regression is? This is done using a metric called R squared and R squared is calculated by dividing the ESS with the TSS. ESS stands for explained sum of squares and TSS stands for total sum of squares. ESS is the variance of the fitted values that are present on the regression line, how much do those values jump around. TSS is the variance of the actual values to which we have fitted our regression model. By expressing our square as a ratio we are trying to capture how well our regression model represents the underlying data. Higher the R squared, better the quality of the regression. The upper bound of R squared is 100%. Let's understand this visually. The original data points have some variance. Remember, variance captures how the points jump around. This variance is called the TSS or the total sum of squares. When you're representing this data as a linear regression model each of these points will have some projection or some fitted value on the linear regression line. The fitted data points have their own variance, that is ESS. R squared is a ratio of the ESS, the explained sum of squares, to the total sum of squares. It tries to measure how much of the variance in the original set of data points is captured in the fitted values. The ordinary least squares method that we've spoken of so far which minimizes the variance of the residuals is the line which has the best possible R squared. There are other types of regression models though, which make sense in different use cases.

These regression models alter the objective function of the optimization. The objective function is no longer the mean square error. Examples of such a regression are the Ridge regression, the Lasso regression, Elastic Net, and the SVR.

Demo: Data Preparation

Before we move on to the more interesting Lasso and Ridge regressions let's first see how to implement linear regression is scikit-learn. We'll write all our code in M two demo one. Import the pandas library which we'll use to read in and explore our data. For our regression problems we'll use the automobile price prediction data. This data is available as a part of UCI's vast collection of machine learning data sets. UCI is the University of California at Irvine. We've downloaded this file and it's currently present as a CSV file in our data directory. The data is separated by commas and we're using the Python engine within pandas to read in this file. This data contains 20 plus features for over 200 vehicles, features such as the make of the car, the fuel type, the aspiration, the number of doors, body style, and so on. Let's explore this data. Notice that missing values in this data are represented by the question mark. Numeric packages typically have special functions that allow you to deal with not a number or NAN values. So it's useful to replace our question marks with NANs which we'll do using the NumPy library. If you now examine the data you'll find that some columns have NAN values, not a number. The describe function on a pandas data frame is very useful indeed to give you a quick overview of how your data looks. This will have information like the total count of rows, the mean value, the standard deviation, min and max values, and the values at various quartiles. If you look at the data carefully you'll find that the price field which was present in the original data set is missing here. Price is our label or our target value for the training data. There's a reason for this. We'll see what it is in just a bit. You can simply say describe and include is equal to all to get information about all the data and you'll get additional information as well. If you scroll over to the very end you'll see that the price field is now included. But it has many NAN values. If you examine the price column alone you'll find that this is of type object. An object cannot be summarized with numerical statistics. We need to convert this column to be of type float. Pandas offers a very easy way to do this using the pd. to numeric library function. This converts the price column to numeric values. We coerce on errors. That means we want to ignore errors and force the conversion. If you now examine the price column you'll see that it is of type float 64. As we examine the input feature set we'll find that there are certain columns whose values make no sense in determining the price of a particular vehicle. One such column is the normalized losses. This has to do with insurance payment, the loss on insurance payment for a particular kind of vehicle. That's not a predictor of automobile

price. So we go ahead and drop this column. Once again go ahead and call describe on the data frame to see what additional processing we need to do to prepare our data for linear regression. You can see that the price column is now numeric but there are other columns that might need to be processed. The horsepower column is in non numeric form as well. It is of type object. Go ahead and use pd. to numeric to convert it to float values. With that complete, let's turn our attention to the number of cylinders. You'll find that this is of type object as well and these are categorical values when they could be numeric and have some meaningful value. A more powerful and more expensive car is likely to have more cylinders. We'll perform a mapping using the Python dictionary from categorical values to numeric values which make sense. Please note that this step isn't encoding the categorical value. Instead we're assigning numerical values which make more sense and the number of cylinders that our car has will drive the price of our automobile. Let's examine the data once again. We'll turn our attention now to the make of the car. This is clearly a categorical variable with a finite set of values. Another example is the number of doors, another categorical variable. A third example is the body style of the car. All of these will affect the automobile price but they have specific discrete values. Let's use the pd. get dummies method to convert all of these categorical values into one hot representation. We'll do this for the columns that we just examined and other categorical columns as well as you can see on screen. Now if you examine the data all columns are numeric. Notice the one hot representation of the engine type. We've performed a bunch of data transformations to feed into our machine learning model but we haven't cleaned up our data yet. Pandas offers a very helpful function drop any which allows us to drop all rules which have not a number values. We'll now check to see whether our data set contains any null values. There are zero rows, so we are good there. Our data is cleaned up and in the right form. We're now ready to feed in this data to our machine learning model. Scikit-learn offers a very handy function for exactly this purpose. The train test split offers a very easy way to split our data set into training instances and test instances. While we're building machine learning models we want to make sure that our model works well on new instances it has never seen before. It has seen all of the training data before. That's what we used to tune our model parameters. The test data is what we use to check whether our model performs well. The features which form the input to our model includes all of the columns in our data frame except the price column. The price column is the label. Go ahead and drop the price column. It's not part of our X variables. The price column is our Y variable. These are the actual labels that we'll use to train our machine learning model. Use the train test split function from scikit-learn to split our input data set into training and test data. Test size is equal to 0. 2 gives us the proportion of test data. 20% of the input data is reserved for testing.

Demo: Linear Regression Using Estimators

Something you'll notice over and over again when you're working with scikit-learn is that the actual concepts of the machine learning algorithm are hard to understand and it takes a while for you to wrap your head around them. But when you go to implement those algorithms using the scikit-learn library it's very simple indeed. For example, linear regression requires that you instantiate a linear regression object called an estimator. An estimator is a high level API that scikit-learn offers which implements the fit method. Any object in scikit-learn which learns from data which can be trained is an estimator. Here linear regression is our estimator object and we call the fit method passing in the X data as well as the Y variables to start the training process for this estimator. Output you'll see printed out the estimator object with its default values and any values that you passed in. Copy underscore X is set to true, this means that the X variables that we've passed in will be copied over, will not be overwritten. Remember that linear regression involves fitting a line using a slope as well as an intercept. Fit intercept by default is set to true, indicating that intercepts ought to be used. The data is not centered around zero. Normalize is set to false indicating that we do not want to normalize our data. Normalization is a process similar to standardization. It involves subtracting the mean and dividing by the L two norm, not the standard deviation. We'll discuss the L one and L two norm in a little more detail later on in this module. We can get the R square of this regression model which measures how good our regression line fit is by calling the score method on the linear model. Our R square is 96. 7 which is a pretty high value. This is the R square of the training data. Note that the regression model captures 96% of the variance in our training data set. You can view the coefficients on the regression by calling linear model. coef underscore. This will show you the weights for all the features in our data set. Seeing the weights alone gives us no information. What is really useful, to see the weights associated with a particular feature. We first get the feature names from the columns of our training data set and then set up a panda series that will associate the coefficients with the feature names and sort them by the coefficients. This association will allow us to see how much weight that is given to a particular feature in the final regression model. We can see what factors really drive the price of automobiles. First off you can immediately see that some features are associated with negative coefficients which means these features are negatively correlated to the price. Cars with these features are cheaper. Peugot, Plymouth, Isuzu, and Mitsubishi cars are cheaper than others. The coefficient of certain features are very small, very close to zero, which means these features have very little effect on the vehicle price. The highest coefficients are associated with those features which drive up the price of a vehicle. The fuel system IDI, fuel type diesel, cars of type BMW, Saab, or Porsche have higher prices. So far we've only used the training

data set to run training on our machine learning model. Let's now use this model for prediction. Estimators have the predict function which you can call in order to predict with an input data set. This is our X underscore test data set. The predicted prices of the automobiles will be stored in Y underscore predict. To see how closely our predicted labels follow the actual labels let's plot it out in a graph using math plot lib. We'll have two lines on the graph, one for predicted values and one for actual values, and we'll see how close they are. And here is our result. The predicted values are depicted by the blue lines and the actual values by the orange line. You can see at the various points of prediction that these are reasonably close. How well our regression model works on the test data set is given by the R square of the test data set. This is 63%. This is not as good as what we got with the training data but that was to be expected. The mean square error is the objective function of the ordinary least squares method of regression. Let's calculate the mean square error with this very handy function from the scikit-learn library. The mean square error is calculated on the predicted labels and the actual labels and the mean square error for our model is of the order of 26 million. Another interesting metric for any regression model is the root mean square error or the RMSE which is the square root of the mean square error that we just calculated. The root mean square error describes on average how much our predicted label will differ from the actual label. The RMSE is around 5, 100 for our model which means on average the price predicted by our model is around $5, 100 away from the actual price. This can be in the positive or negative direction.

L1 and L2 Norm

Before we move on to studying Lasso and Ridge regression there are some concepts that we need to understand. The first of these concepts is the distance measure. There are different ways in which we can measure the distance between two points. One of the ways is called the L one distance, also known as snake distance, city block distance, Manhattan distance. This involves traversing from point A to point B along city blocks. From the image here it's pretty clear why it's called snake, city block, and Manhattan distance. Let's focus on why it's called the L one distance. Let's assume that the coordinates of these two points are one comma zero and five comma four. L one refers to how the distance between the two points is calculated. Let's consider the X coordinates, five and one. We first subtract one from five and get four and then look at the Y values. We subtract zero from four and get four. The sum of these values gives us the distance between these two points. The absolute difference between the coordinates of the two points gives us the L one distance. To calculate the L one distance we first find the absolute value of the difference between the coordinates and find the sum of all of these absolute values. A discussion

of the L one distance was important because it will help us understand the L one norm. The L one norm for any set of numbers is the absolute value of those numbers. The term one in L one comes from the fact that all of these numbers are raised to the power of one. The L two norm for the same set of numbers are the absolute values of all of these numbers raised to their squares, raised to two. As you know from basic mathematics the squares of numbers are always positive. So you can get rid of the absolute value of the modular sign and simply raise the number to the power of two and sum up all the numbers. The L two norm can be better understood if you understand the L two distance between two points. The distance between two points can also be calculated as the crow flies. This is called Euclidean distance. For the same two points that we saw earlier the L two distance is five minus one squared plus four minus zero squared, 16 plus 16 equal to 32. The one difference between the L two norm and the L two distance is the fact that distance is the square root of this number. Here is the complete formula for the L two distance. We first find the difference between the points and square this number. We then find the sum of squares and find the square root. The most important thing that you need to take away from this clip is the understanding of L one and L two norm. The discussion of distances was simply to facilitate that understanding. To summarize, the L one norm is the absolute value of the numbers raised to the power of one and summed up and the L two norm is the sum of the squares of those numbers.

Overfitting and The Bias-variance Trade-off

When you are working with machine learning models an important concept to understand is the bias variance trade off because it is this trade off that determines how you will set up your model. Let's say you were given a number of points on two dimensional plane and your task was to fit the best curve through these points. Now, this definition of best is kind of iffy here. Let's say you had a curve which went like this. Would you call this a good fit? Yes, this curve has a good fit if the distances of points from the curve are small. There is a problem with this definition though because you could technically draw a pretty complex curve to fit these points. In fact, you could have the curve pass through every single data point that you have. In such a situation you could say your curve is a very good fit for the data points that you see on screen. But if you use the model represented by this very complex curve on a new set of points this model might perform quite poorly. Our machine learning model which is represented by this complex curve is so tuned to the training data that the test data represented in blue would perform very poorly with that model. This happens more often than you would think with machine learning. This is called overfitting, when your model has great performance during training and very poor performance

in real usage. Overfitting is when your model instead of extracting insights from your training data has simply memorized that data set. Instead of this very complex curve where the distance from every data point is minimized you could have a simple straight line, which is what linear regression does. This might perform worse during training, but it'll work better with test data. By simplifying your model and allowing it to perform worse during the training phase you're building a better model for prediction. The characteristics of an overfitted model is that it has low training error. The model performs extremely well on the training data set. However, it has a high test error. When you actually use the model for prediction it'll perform poorly with real data. Overfitting is very common in machine learning. In fact certain models are more prone to overfitting than others. Overfitting is the result of a suboptimal choice that you as a developer made in the bias variance trade off. An overfitted model is one which has a high variance error and a low bias error. Let's study what these terms mean. A model is said to have a low bias error when we make few assumptions about the underlying data before we fit that model. Fewer assumptions result in fewer constraints on the model parameters as a whole. On the other hand a high bias indicates that we made more assumptions about the underlying data. For example, in linear regression our assumption is that a straight line can model our data. That is a classic example of high bias. A model which has low bias tends to be overly complex. The training data becomes extremely important. The actual model parameters or constraints count for little. The model tries to follow the training data as closely as possible. On the other hand when models have a high bias the model can end up being too simple or simplistic. The importance of the model parameters far outweigh the importance of the training data. So much for bias. Let's now understand variance. A high variance model is one which changes significantly when the training data changes. You can think of a high variance model as highly sensitive to the training data. On the other hand, the model is said to have a low variance when it doesn't change much when the training data changes. It's not very sensitive to the data points in training. A high variance model can end up being far too complex because it very closely tries to mimic the training data. Even a slight change in the training data causes a significant change in the model. A low variance model on the other hand can end up being too simple. It's not very sensitive to the training data. It may not represent it very well. A model that's far too complex is said to have a high variance error. A model that is so simple so as to be simplistic is said to have a high bias error. So when you're choosing your machine learning model depending on your use case and the training data that you have you need to have the right bias variance trade off. You should understand certain characteristics of the different machine learning algorithms. High bias algorithms tend to have simple parameters and a regression is an example of a high bias algorithm. High variance

algorithms tend to have much more complex parameters. They might overfit the data. Decision trees and dense neural networks are examples of high variance algorithms.

Multicollinearity in Regression

Overfitting tends to be a common problem with machine learning algorithms which is why there have been a bunch of ways developed to mitigate this. Regularization, cross validation, ensemble learning of which dropout is a part, are all ways to mitigate overfitting. Regularization is a technique where we penalize complex models. We add an additional parameter where if the model coefficients get too complex we add a penalty to the objective function. This is the technique that we use in regression. This is what Lasso and Ridge regression do. Cross validation is a technique where we split the training data into training and validation. The actual training phase is run on the training data. How good the model is is evaluated using the validation data. And finally, dropout is a kind of ensemble learning. We'll study more about ensemble learning later on in this course. Dropout is a technique to mitigate overfitting in neural networks. During the training phase certain neurons in the network are intentionally turned off and do not participate in training. It's a different neuron for every epoch of training. This forces the active neurons to do more work to identify significant patterns in the data. Of all of these techniques for preventing overfitting let's study regularization in a little more detail because Lasso and Ridge regression use this method. Regularization can mitigate overfitting a model to training data by penalizing extremely complex models. This is done by adding a penalty to the objective function. The objective function is the function that the model tries to minimize. The most common objective function for linear regression is the mean square error. In a regression model the penalty is a function of the regression coefficients forcing the optimizer to keep the coefficients simple. The more complex the coefficients the higher the penalty. Multiple regression specifically is prone to overfitting. Imagine a multiple regression model set up like you see on screen. Multiple regressions have many X variables or X features. When you perform a linear regression with multiple variables it involves finding K plus one coefficients. K for the explanatory X variables and one for the intercept. Even though regression models tend to be high bias and low variance a big risk with multiple regression is multicollinearity, the X variables which contain the same information. If two X variables are highly correlated one is not required. If our feature data is highly correlated that can cause overfitting. Let's say we wanted to use multiple regression to determine what makes a good salesperson. We think the causes are the number of cold calls, years of experience in sales jobs, and so on and the effect is the bonus the person will receive as a part of the sales team. Let's say you identified 10 different features that could be the reason for

the salesperson's success. You could have a kitchen sink regression which uses all of these input features. However if you analyze your input data carefully you might find that many of the explanatory variables are highly correlated with each other. You might find that all of these explanatory variables are caused by an underlying personality trait and that personality trait is extroversion. Once you've determined that extroverted individuals are probably better at sales you simply measure extroversion and use it instead of all those other features. So rather than using a kitchen sink regression which focuses on N or 15 different causes for a successful salesperson you might want to perform some kind of factor analysis which reduces the many observed causes to a few underlying causes and use those underlying causes to figure out how successful an individual is in his or her sales role. And it is this multicollinearity in the underlying features that are fed into the regression which might lead to overfitting which is why we need other regression models which mitigate the overfitting problem and lead to a better regression model. Alternative regression techniques are Ridge and Lasso regression amongst others.

Lasso and Ridge Regression

Preventing overfitting in regression requires the use of regularized regression models. Lasso regression penalizes large regression coefficients. Ridge regression also works the same way with a slightly different objective function. Elastic net regression simply combines Lasso and Ridge. Remember that the bias variance trade off pulls in opposite directions. Regularization reduces the variance error but increases the bias error. We have to find the right balance in our trade off. Here's the objective function of our ordinary least square regression. This is the mean square error. This is what we set out to minimize. By minimizing the mean square error we find the values of A and B which give the best fit line. In Lasso regression it's not the mean square error that is minimized. An additional term is added to the objective function which we seek to minimize. This additional term is multiplied by alpha which is a hyperparameter than we specify. If you observe this additional term you'll see that it is the L one norm of the coefficients A and B. We want to find the A and B that still define the best fit line subject to this new objective function which includes the L one norm of the coefficients. Ridge regression is also a regularized regression model. It works exactly like the Lasso regression except that the penalty function is the L two norm of the coefficients. Both Ridge and Lasso regression penalize complex models by adding this additional term to the objective function. Let's quickly see the characteristics of Lasso regression. We add a penalty for large coefficients and this penalty is the L one norm of the coefficients. The penalty is weighted by the hyperparameter alpha. The value of alpha is something that you can tweak to find the best possible model for your data. The value of alpha

determines how large of a penalty you want to have for complex coefficients. If you perform a regular ordinary least squares regression alpha is equal to zero. The additional penalty term doesn't exist at all. If you have a very large alpha where alpha tends to infinity this forces small coefficients to be zero. Insignificant features are not considered in your regression. Alpha is a hyperparameter. It's a tuning parameter that you have for your machine learning model in order to find the best possible model for your data set. Alpha helps you eliminate unimportant features. Lasso stands for least absolute shrinkage and selection operator. The actual math for this regression model actually turns out to be quite complex. In fact, it does not have a closed form solution meaning it does not have a solution that can be expressed in terms of a formula. The Lasso regression formula requires a numeric solution where you perform many iterations, build many models 'til you get to the best possible model. The math is complex for Lasso regression because it uses absolute values in its objective function. Ridge regression differs from the Lasso regression in exactly one significant detail. It uses the L two norm of the coefficients in its penalty function. The characteristics of Ridge regression are very similar to that of Lasso regression. It adds a penalty for large coefficients. The penalty term is the L two norm of the coefficients. That's what is added to the objective function and the penalty is once again weighted by the hyperparameter alpha. The use of the L two norm which uses the squares of the coefficients rather than the L one norm means that the Ridge regression has a closed form solution. Ridge regression can be expressed in terms of a formula. Unlike the Lasso regression if you tune the alpha parameter to be very high it'll not force coefficients to zero because Ridge regression has a closed form solution. It does not perform model selection. It doesn't build many different models and choose the best one. The best possible model can simply be calculated using a formula.

Demo: Lasso Regression

In this demo we'll implement Lasso as well as Ridge regression in scikit-learn using estimators. Once again you'll find that the concepts are harder to wrap your head around. The actual implementation is very simple. We'll continue working on the same data set for automobile price prediction and use the same file where we had set up our linear model. Just import the Lasso class which is the estimator for Lasso regression from the scikit-learn library and pass in the alpha parameter. Alpha if you remember is the regularization parameter to prevent overfitting of our model. Here we choose to normalize our data. We'll center the data by subtracting the mean and dividing by the L two norm. Call the fit function on the training data and the Y variables to train our Lasso regression model. Get the R square to see how well our Lasso regression model fits our training data. You can see that the R square value here is a little lower than the R square that we

got when we used simple MSE regression. Let's print out the coefficients or the weights of the various features in our input data set. Large negative values for coefficients show that these features are inversely correlated with the price of the automobile. Lasso regression is a regularized regression model which means it penalizes complex coefficients. You can see a large number of features from the middle of the set have been eliminated completely by reducing their coefficients to zero. If you look at the features at the bottom which have high weights you can see that the coefficients themselves are smaller but the features with the high weights remain the same between the Lasso regression model and the linear regression model. We'll go ahead and predict the automobile prices for our test data set and then plot the predicted values against the actual values using math plot lib and really it's possible to see from the simple plot how close the actual values are to the predicted values. The fit is in fact much better and this we can confirm by calculating the R square on the test data. The R square on the test data for our Lasso regression model is 88%. It's much better than the linear model which was around 63%. You can tell that the Lasso regression model is better for prediction by calculating the root mean square error as well. Compared with 5108 for the linear model this is around 2800. Let's change the value of the alpha hyperparameter and see how our model changes. From 0. 5 we'll change it to five. Go ahead and hit shift enter and execute all cells which come after that. You can see that the R square on our training data fell a little bit. It's now 94. 9. If you observe your feature coefficients carefully you'll find that many more features have had their coefficients go to zero which means these features no longer play a part in our model. If you plot the predicted values of Y and the actual values of Y the automobile prices you'll find that the prediction still seems pretty good. In fact, the predictions on the test data actually seem to be better, closer to the actual values. Visual inspection isn't enough though. If you plot the R square on the test data you can find that it has increased. Increasing the value of alpha has forced more significant to drive our prediction. Our model has improved. It's not overfitted. The root mean square error has also fallen. Getting the right value of alpha to build the best possible model for your data set requires a little tuning and tweaking. Scikit-learn offers some tools to make this easier. We'll study those in later on in this course.

Demo: Ridge Regression

Let's continue working on the same automobile price prediction data set. This time we'll set up the Ridge regression estimator. Use the Ridge class. Pass in a value of alpha and set normalize to true. Call fit to train our estimator with these parameter values and call score to see how well this model performs on our training data set. The R square on training data is pretty good. It's around

95. 38%. Examine the coefficient values that have been assigned to the various features in our input data set. You'll find something interesting. Notice that the high coefficient values are much lower than the high coefficient values in Lasso as well as simple linear regression. The penalty function has really constrained the complexity of our coefficients. As before, we'll run predictions on our test data and plot the predicted Y values against actual Y values using math plot lib. And from this graph it seems that the model performs pretty well, at least from a visual inspection. The real test of whether this is a better model is our R square value on the test data and it comes out to be around 88. 75. This is a high value of R square and this value is very similar to what we got from the Lasso regression model with alpha set to 0. 5. The root mean square error here is around 2800. Again, similar to our first Lasso model. Let's increase the penalty, the alpha value for our Ridge regression model from 0. 05 to 0. 5. Go ahead and hit shift enter to execute the other cells and notice that our R square on the training data has fallen. It's now 92%. Examine the coefficients for the various input features. You'll see that they seem to be a little lower after increasing alpha. The actual features which drive our automobile price though remain the same. Predict the Y values on test data and plot the predicted versus the actual values. The graph makes it seem like our prediction was quite good. A true measure of whether this model is actually better than our previous Ridge model is the R square. The R square on test data has increased which means this is a better model. And as we would expect the root mean square error has fallen. What if we go back and increase our alpha hyperparameter value further, from 0. 5 to 1. 0? Will this give us a better model? Let's check. Execute the code in the various cells by hitting shift enter. Notice that the R square value on the training data has fallen further. The coefficients for the various features in our training data seem to be around the same. There's not much to see there. Predict values on the test data. The actual graph doesn't seem to be very different. Let's calculate R square and here you can see that the R square value has actually fallen a little. It was 92% previously. It's around 90. 7 now, which means blindly increasing our alpha parameter can hurt the accuracy of our model. There is an entire process which involves tuning your hyperparameters to find the best model for you. This involves training multiple models and finding the one that works best on the test data. This process is called hyperparameter tuning.

Support Vector Regression

Now that we've understood Lasso and Ridge regression let's move onto a different regression model, the support vector regression or the SVM regression problem. Support vector machines are typically used for classification problems, spam or ham, fraudulent or not. Support vector regression uses the same underlying principles as support vector machines for classification

except that the objective function for the model is different and that makes all the difference in the world. Let's quickly understand how SVM classification works. We'll study it in more detail in the next module. Unidimensional data points can be represented using a line such a number line and unidimensional data points can be separated or classified using a single point. Let's assume that we're dealing text data such as reviews and this number line represents the number of words in a review. You've determined that shorter reviews tend to be negative and longer reviews tend to be positive. With this information you could choose one of the points on the number line. Let's say the point is where number of words in the review is equal to 10 and use this to classify your reviews. You could say all reviews with number of words less than 10 are negative reviews and all reviews with number of words greater than 10 are positive. Let's say you're still working with movie reviews but now you have two pieces of information about every review. You have the time when the review was posted and the number of words in a review. The red reviews are the negative reviews and the blue reviews are the positive reviews. So you have a line that can be drawn separating your positive from the negative reviews. Two dimensional data can be separated using a line. Now let's say you have one more piece of information about the reviews. You know the number of words in a review, the time when the review was posted, and whether the review was posted by a B2B or a B2C customer. Thus as you add dimensions to your data you need to add additional axes to represent those dimensions. N dimensional data can be represented using an N dimensional hypercube and N dimensional data can be classified using a hyper plane. Support vector machine classifiers tries to find that hyper plane that best separates points in a hypercube. This is far easier to explain and visualize using two dimensions, so let's go back to our two D data. Ideally your data is linearly separable, that is, you can draw a line and separate all points so that they belong to one side or to the other. Also hopefully your reviews are also cleanly separated with all the negative reviews on one side of the line and positive reviews on the other side of the line. This is a hard decision boundary. On either side of the lin you will have points that are the closest to your decision boundary. The nearest instances on either side are called the support vectors. For a very clean classification of your data points these support vectors should be as far apart as possible. You want to fit the widest possible margin. Having a wide margin means that your classification was clean and this is what support vector machines try to do. SVM tries to find the widest street between the nearest points on either side. This is a big picture understanding of how support vector machines work for classifying data. Now let's understand support vector machines for regression by seeing what the differences between the two are. The objective of SVM classification is very simple. We want to find the widest margin which has the most distance between the support vectors. As in the case of all regression problems we want to find that line that best fits the points. By finding the classification boundary

in support vector machines we want to ensure that there are no points within the margin. We want to find as clean a classification boundary as possible. In SVM regression on the other hand we want to maximize the number of points inside the margin. We want to fit a line within a certain margin such that as many points as possible are within that margin. In the case of SVM classification the more points we have further away from the boundary the better our classification is. That improves our objective function value. For support vector regression though the points far from the margin are bad. We want all points to be within the margin. Points further away from the margin make our objective function worse. In our classification problem it's always possible to have outliers. Outliers on the wrong side of the line are penalized. We don't want any data point misclassified. For support vector regression the penalty is a little different. Points which are far from the margin are penalized. It does not matter on what side of the margin they lie. Notice that we have a penalty factor C in both cases. You multiply C by the magnitude of the margin violation. C is a hyperparameter that you can tweak in both SVM classification as well as regression. When we use support vector machines for classification the width of the margin is determined by the optimization function when we try to fit our model to the training data. The optimization function tries to make this margin as wide as possible. In the support vector regression problem the width of this margin is another hyperparameter called epsilon. You need to specify the width of the margin when you're performing the regression. It's not determined by the optimizer.

Demo: Support Vector Regression

Let's build another specialized regression model, support vector regression using scikit-learn. We start off with a new Jupyter I Python notebook M two demo two dash SVR. We go ahead and import the pandas library. Pandas is extremely useful. It's always good to have it around. For this regression we'll once again work with automobile data but with a different data set. This data set is also available at the UCI website. We'll use this data set to try and predict the mileage for a particular automobile given a bunch of input features. This data is white space delimited and it does not have a header and here are the names of the various columns. These form the features of our data set. The miles per gallon is our target value. Go ahead and explore this data set. One of the columns that should jump out at you is the name of the car. If you think about it the name of the car has nothing to do with the miles it offers per gallon. The name of the car is a feature which has 305 unique values in a data set which is just 398 long. It's pretty obvious that the name of the car is not really going to give us information about the miles per gallon. So we can go ahead and drop this particular column. Here is our current data. Notice the last column, the origin.

The origin determines where the car is from, whether it's from America, Europe, or Asia. It makes sense to encode this column using one hot encoding. We'll first replace one, two, and three with more meaningful categorical values, America, Europe, and Asia. Once the column values are meaningful we can now use the pd. get dummies function in order to convert these to one hot encoding. This data set also contains question marks for missing values. We can go ahead and replace those with NANs and then use the drop NA function on a data frame to drop all rows which have NANs. Use the train test split to split up your data into training data and test data. Make sure you drop the MPG column from your X variables. That is the label or the Y variable that we have to predict. 20% of our data will be test data. The rest will be for training. Performing the actual regression in scikit-learn is very straightforward as you have seen before. Instantiate the SVR estimator and pass in a linear kernel. We want to perform linear regression. The penalty factor for points which lie outside of our support vector margin is one. The default value for the epsilon hyperparameter which determines how wide our support vector margin is 0. 1. Let's see the coefficients of this regression model. Notice that the coefficients are specified as a list of lists. Use the score method to see how well our support vector regression model fits the training data. The R square here is about 61. 9%, so not that great. This time we'll do something a little different. Let's plot our model coefficients using math plot lib. This will allow us to see the magnitude of our model coefficients. For each one of our features we have an associated coefficient and the magnitude of the coefficients are seen on this bar graph. You can see from these coefficients that European cars where the origin is Europe have a positive effect on the miles driven per gallon. American cars tend to have a negative effect. The coefficient is negative. We'll now go ahead and predict the Y values, the miles per gallon, for our test data and plot the actual values against the predicted values. Notice that our R square on the test data is much higher than the R square on the training data. It's 71%. With this calculating the mean square error and the root mean square error of our model is very straightforward. We follow the same steps as we did earlier. Remember that the RMSE indicates that on average the predicted miles per gallon by our model is about four miles off either in the positive or negative direction.

Demo: SVR Reduced Penalty

Let's change the value of C, the penalty hyperparameter, and see how that affects our model. From 1. 0 we'll reduce it to 0. 5. Intuitively reducing the penalty parameter increases our emphasis on finding the best fit model for our underlying data and reduces our emphasis on penalizing points which are far away from the margin. Hit shift enter to execute the code in the various cells. Notice the R square of this new regression model on our training data. It's almost 81%. It's a much

better R square. Here is a plot of the new model coefficients and our graph showing the predicted values versus the actual values. The R square of the new model on the test data is almost 83%. You can see that the overall fit of our model is much better. Now that we've reduced the penalty factor the mean square error and thus the root mean square error have also fallen. This is overall a much better model. You can try this for yourself but increasing the penalty factor C to say two will make the model worse. The R square will be worse, the RMSE will be bigger. And on this note, we come to the very end of this module on building specialized regression models. We studied the Lasso, Ridge, and the support vector regression models and understood how we can use R square for measuring the model fit. We studied the concept of overfitting your ML model to the training data and we saw that this was a common occurrence. We saw that preventing overfitting requires a good understanding of the bias variance trade off. We saw how multiple regression can be prone to overfitting if there are multiple features that are correlated. Lasso and Ridge regression are regularized regression models which allow us to mitigate overfitting. In addition, we also studied the support vector regression model which is built on the same principles as support vector machine classification but uses a different objective function. In the next module, we'll study SVMs for classification in much more detail. We'll also study how we can use gradient posting for regression.

Building SVM and Gradient Boosting Models in scikit-learn

Module Overview

Hi and welcome to this module on building support vector machines and gradient boosting models using scikit-learn. We studied support vector machines briefly in a previous module when we spoke of support vector regression. Support vector machines are actually a very popular ML technique for classification. In this module, we will see how we can use SVMs to work on text as well as images. We'll use SVMs to classify documents by topic and to find the digit represented by an image. We spoke briefly about ensemble learning earlier which allows us to mitigate the over fitting problem in machine learning models. Often you can have many ML models work

together as an ensemble to build a stronger model. We will see how this can be done in gradient boosting regression, which uses several weak decision trees to build a stronger regression model.

Support Vector Machines for Classification

Let's briefly review the SVM classification model that we studied in an earlier module. If you have data in one dimension, it can be represented using a number line and it can be classified using a single point. For example, if you find that shorter reviews are negative and longer reviews are positive, you could have your classification point be number of words in a review equal to 10. Points on one side are negative, points on the other are positive. If your data has two features that needs two dimensions to be represented. Number of words in a review and time when the review was posted. In order to classify these points, you need to draw a line through this plane. Typically data that you would use as an input to a machine learning model can comprise of many dimensions. Here is data in three dimensions in order to classify this data you need a plane that passes through this cube. Extending this to N dimensional data. N dimensional data represented using a hypercube requires a hyper plane for classification. And that's the objective of a support vector machine. Support vector machine classifiers find that hyper plane that best separates points in a hypercube. Depending on the data set it's possible that you might find a hard decision boundary that clearly separates our data. In such situations, the data points, which are the closest to the decision boundary are called the support vectors. For a good clean classification, we want the support vectors to be as far away as possible. We want to fit the widest possible margin and that's what support vector machines do. Now you could have a classifier that is a hard margin classifier. These are classifiers which are very sensitive to outliers. In fact hard margin classifiers do not allow any outliers on the wrong side of the boundary. This is actually a very hard constraint and maybe impossible to meet in real data. Hard margin classifiers require perfect linear separability in data. Soft margin classifiers on the other hand are a little more forgiving. They allow some violations of the decision boundary. Here in this example we have exactly one data point that has violated the classification boundary. SVMs are very powerful indeed and can be used to classify data sets which might seem non-separable upfront. SVMs internally use something called the kernel trick, which are smart transformations that can help resolve many such hard used cases. This kernel trick allows SVM classification to be extended to almost any data set. We won't go into the details of how this works. The mathematics is beyond this course. However, let's consider a simple example where the original data is not linearly separable. Let's apply a simple transformation where we convert our x values to x square and plot our data. A plot of the squares of the original data might look like this, this data set is now linearly separable.

Setting up the SVM Classification Problem

In this clip, let's try and understand the intuition behind how the SVM classification problem is set up. We've already seen data in two dimensions, where one dimension is the number of words in the review. And the second dimension is the time when the review was posted. We want to draw a line through these data points and classify these reviews as negative or positive. The time when the review was posted is one x variable. We can call it x1 and the number of words in the review is another x variable. We can call it x2. The support vector machine draws a line such that the points on one side of the line are negative. Their y values are negative. And the points on the other side of the line are positive. This decision boundary is a line that can be expressed as a function of x1 and x2. W1 of x1 plus w2 of x2 plus b. Here we've used colors to represent positive and negative reviews. Actually positive or negative will be a score that will be represented in a third dimension, which is why we need three dimensions to visualize this decision boundary correctly. Here are the three dimensions in our data, x1 and x2 are features. Y is the label. Here is our decision boundary which is a hyperplane expressed as a function of x1 and x2. And this decision boundary is a plane that separates the various data points based on the value of this function whether this function is equal to less than or greater than zero. For example your classification can be such that for all data points where the value of this function is greater than zero, they are positive reviews. They lie kind of on the right side of this decision plane for data points, where the value of this function is less than zero, they lie on the left side of this decision plane. And for points which lie exactly on this decision plane, this function is equal to zero. So whenever a new data point comes in, we can apply this function to that data point. If the result is greater than zero, y predicted will be positive, if the function is less than zero, y predicted will be negative. And for points on the decision boundary we can classify those as either negative or positive. Here we've chosen to classify them as negative. The objective of the SVM classifier is to find the best values of the weights w1 and w2 and the coefficient b. The values of w1, w2 and b should be such that we can use them to classify our data points. If a formula using these coefficients gives a value less than or equal to zero, our prediction will be negative and if the value is greater than zero, our prediction will be positive. Finding the best possible value for w1 w2 and b is the optimization problem that our support vector machine tries to solve. And this optimization problem finds the widest street that can classify our data points. The definition of best here is those values of w1, w2 and b where the widest margin exists between the support vectors but there is another constraint that we haven't considered so far. Our best decision boundary must also avoid or minimize outliers. We need to penalize outliers in some way during the optimization process to find the decision boundary if a point happens to be on the wrong side of a boundary we need to first figure out the magnitude

of the violation, the distance between that point and our margin. Calculate this magnitude for every point that is in violation. The magnitude of the violation is then multiplied by a penalty factor which is a hyper parameter for the SVM model. The penalty can be expressed as C multiplied by the magnitude of the margin violation. We can use this penalty factor C, to define hard margin and soft margin classification that we studied earlier. If C is very high the penalty for outliers is very high, that is hard margin classification. If the value of the C hyperpparameter is very small the penalty for outliers is small that is soft margin classification. Going back to our optimization problem, we want to find the widest margin between support vectors but penalize each margin violation by a factor C. The last couple of minutes might have seemed a little mathematical for you but the best part of using SVM library is that you do not need to know the precise math. You only need to know the intuition. The best decision boundary used to classify our data points is one which seeks to maximize the width of the street. The width of the support vectors. It also seeks to minimize the margin violations. The two objectives of support vector machine classification are in conflict with each other. This is where the tuning that you perform comes in the penalty C is a hyper parameter that you can tweak.

Demo: SVM Text Classification

Let's perform some document classification with SVMs and scikit-learn, because they're dealing with text. We will get to use some of our text extraction features that we studied in an earlier module. The data set that we'll use for this problem is available with the scikit-learn library. Scikit-learn contains a number of data sets that can be used to train and validate models. We'll use the fetch 20 newsgroups module to retrieve the data set. This contains roughly 20, 000 newsgroup documents which are split across 20 newsgroups. If you want to learn more about this data set, you can go to the URL that you see on screen. Each one of these 20 newsgroups corresponds to a particular topic. The return value from this function is a dictionary and these are the keys within the dictionary. The data key is what contains our training data. These are our newsgroup documents. Target names are the newsgroups to which these documents belong. These are the labels or the y values associated with each document. Here is a document sample from the training data set. This is an email related to cars. Here are the 20 newsgroups to which these documents belong. They range from sports to computers to politics. Our categorical variables need to be expressed in numeric form and that's what the target key holds. Every document is x data and and we will represent it in numeric form using the count vectorizer. We'll call the count vectorizers fit transform method on our training data. The output of the count vectorizer is a sparse matrix. Every word is identified uniquely using its document ID and its unique word ID and

the frequency of the word in that document is specified. Here is the shape of our sparse matrix. You can explore the output of the count vectorizer on the very first document. Here you can see the document ID, the word ID and the associated frequency. I'm going to pause this output of the count vectorizer through a TfidfTransformer. The TfidfTransformer is different from the Tfidf vectorizer in one significant way. The Tfidf vectorizer worked directly on documents and produced a bag of words with corresponding Tfidf course. The TfidfTransformer on the other hand requires a bag of words as its input. That's why the output of the count vectorizer be passed into the TfidfTransformer. The count vectorizer plus the TfidfTransformer is equal to the tfidfvectorizer. Let's print this course for one document. Here you can see a mapping of document ID word ID and the corresponding tfidf score. Once we've set up our data correctly, using a support vector machine estimator is very easy. We instantiate the linear SVC and pass in a number of input arguments. The penalty function that we are going to use is the L2 norm penalty. Another criteria that we can specify is the tolerance for stopping training on our model. If the losses that we calculate on the objective function go below this tolerance value, we'll assume that our model is good enough and stop training. Notice that in order to prepare the data set, we performed a series of transformations on it. We first used the count vectorizer and then the TfidfTransformer and then pass that output to our linear SVC estimator. Scikit-learn provides a very handy tool called the pipeline which allows a linear sequence of data transformations to be changed. Notice that we instantiate a pipeline and within the pipeline we specified the series of transformations that we want performed. We can now simply execute this pipeline by calling the fit method and our training data will be passed through each of these steps. We don't need to call these steps individually. Let's go ahead and get the test data and we can pass this test data through the same pipeline. The last step of the pipeline we'll use our linear support vector classifier for prediction. This is the same classifier that we just trained. One way to measure how well our classifier performs is to calculate the accuracy of our predictions. Let's see how many of our predicted labels are equal to the actual labels. Our support vector machine classifier performs pretty well. It has an accuracy of 85%. Let's tweak our model a little bit. Instead of using the L2 norm as our penalty function, we will use the L1 norm. Go ahead and hit shift + enter to execute all the cells. You'll find that the accuracy of this model is around 81. 5%. It has fallen a bit. Let's make a little change to our pipeline. Instead of using the output of the TfidfTransformer, we'll use the output of the count vectorizer directly to feed into our support vector classifier. Train the pipeline and perform predictions on the test data. You can see that the accuracy is around 79. 8%. Compare this with the accuracy of 85% when we use the exact same model but the output of the Tfidfvectorizer. This gives us an idea of the impact of the Tfidf scores in our classification model.

Demo: SVM Image Classification with Grid Search

SVMs can be used for image classification as well. In this demo, we'll see how. In this demo, we'll work on images from the MNIST data set. The MNIST data set consists of images of handwritten digits where every digit is in grayscale. This is a very popular data set, created by Yan Lakon and it is often used by beginners when they're first getting into machine learning. Every image in this data set is standardized to be of size 28 x 28 containing a total of 784 pixels. The images are all in grayscale. These are single channel images. Every pixel holds a single value for intensity. Here is how the digit four would be represented in the MNIST data set. Notice the intensity values corresponding to the strokes of four are numbers between zero and one. All other pixel intensity values are zero. Every image has an associated label which tells us what digit corresponds to that image. We'll use the support vector machine classifier to classify these MNIST images based on what digit they represent from zero through nine. We will use the PANDAS library as usual and read in the CSV file which contains the training data for MNIST. This file was originally from the Kaggle website. Notice the columns and the data. The first column is the label and the remaining columns are the pixel intensity values. Let's set up the features and labels for our machine learning models. The x variables are all the pixel values and the y variable is the label. Use 10% of the data set for test, the rest for training. The pixel intensity values are expressed as integers between zero and 255, divide by 255 to get intensity values between zero and one. Instantiate the liner support vector classifier and call the fit method on it. In order to run training on the MNIST data set. We'll then call predict on the test data and measure the accuracy of our predictions. And the accuracy of this SVM model is 91%, that means 91% of our test instances were classified correctly. When your model has a number of hyper parameters, we've spoken earlier of the need to tune them to find the best possible model on your data set. Scikit-learn offers some specialized tools to perform exactly this tuning. It will help you choose the best possible model by using a few different values of the hyperparameters that you specify. This is done using the GridSearchCV. Let's assume that we want to tweak two different parameters. We want the penalties of the SVM model to be either the L1 norm or the L2 norm. We don't know which one might be better and we want to try out three different tolerances for our model. Remember that this tolerance is our stopping criteria for our training. This is called grid search because we set up a grid or a matrix specifying the various parameter values that we want to use. Instantiate a GridSearchCV estimator and pass in a LinearSVC estimator within it. When we instantiate our LinearSVC estimator which is going to be trained with various combinations of the parameters that we've specified in the grid, we can also pass in other arguments, which will remain constant during training. The second argument to our GridSearchCV is the grid which contains our

hyperparameter values. GridSearchCV will now run training on our data with every possible model parameter combination. The CV parameter specifies that we want this model to be cross validated to mitigate over fitting. CV is equal to three means that the input data set will be divided into three different parts. This is threefold cross validation. The training data will be two out of three parts and the validation data will be the third part. One thing you ought to be aware of though is that GridSearchCV can take a very long time because it has to train many different models to find the best possible one, it can take a while. This particular grid search took about 35 minutes on my machine. Grid Search will spit out the best possible model parameters. The penalty should be the L1 norm and the tolerance should be 0. 001. We will now use the parameters found by grid search to instantiate our linear SVC estimator. Run training on the model and then measure the accuracy of our predictions on test data. When you get the accuracy number you will see that it is around 91. 19%. Marginally better than the 91. 02% accuracy that we got earlier.

Decision Trees

We will study gradient boosting models for regression next but before we get into gradient boosting it's important to have a good understanding of how decision trees work because decision tree models are what make up gradient boosting in scikit-learn. Let's say you had the height and weight of a number of sports persons and you want to be able to classify them as jockeys or basketball players. Now you know that jockeys tend to be light to meet horse carrying limits and basketball players are just the opposite. They tend to be tall, strong, and heavy. These categorizations that we've made based on height and weight are very intuitive. We know certain characteristics about jockeys and basketball players. And we've used these characteristics as rules on the underlying data. Knowledge of these rules is what helps us classify the sports persons. What we've done almost unconsciously has built up a tree model based on the rules that we know. The decision factors or the nodes in this tree model are the weight and the heights of the players and based on which category they fall to, we classify them as jockeys or basketball players. What we've done here is fit the specialized knowledge that we have about jockeys and basketball players in the form of rules and applied the rules to the underlying data. The rules are not standalone. Every rule is associated with a threshold which determines which way the classification goes. The high threshold here is a 150 pounds. Weight greater than 150 pounds more likely to be a basketball player less than or equal to 150 pounds more likely to be a jockey. We have a corresponding height threshold at six feet. The whole idea behind decision trees is that these rules are not specialized rules that you are aware of. Instead they are determined by the

machine learning model. The rules themselves are important and the order in which these decision variables are applied are also important. Our machine learning model needs to be able to pick the right decisions to make, ask the right questions, and also apply these decisions in the right order. This decision tree model that you see depicted on the right is a CART tree, where cart stands for classification and regression tree. This model can be used for both classification problems as well as regression problems. Once you've completely trained your decision tree classifier, you will feed in the height and weight of a particular sports person at the input of this ML based classifier and based on the internal decision tree model, you'll receive a classification at the output, whether this player is a jockey or a basketball player. Assume that your training data has constructed a decision tree like the one you see on the left of your screen. In order to solve the classification problem once you've given the height and weight of a sports person, you need to traverse the tree to find the right node, where the sports person belongs. Here the weight of the individual that we passed in is less than a 150 pounds, but he is taller than six feet. From your training data you know that at that particular node, the most frequent label that is present is that of a basketball player. At that node, we have more basketball players than jockeys. That is our predicted label. An example of a regression problem using decision trees would be something like this. We feed in the weight and the height of a particular sports person and we predict the number of years that he might spend in his career. Once again our training data has constructed exactly the same decision tree that we've seen earlier in order to solve this regression problem. We will traverse the tree to find the right leaf node where our particular player belongs. Our player in this example is identified as a jockey because its weight is less than equal to 150 pounds. And its height is less than six feet. Based on the training data that we used to build this decision tree model that particular node has been labeled as a jockey which means at that node, we have more jockeys than basketball players. We can now calculate an average of the number of years that these sports persons spend in their careers and this average can be a regression prediction. That's great. We now have our decision tree model. However, what if we fed in the input data of Muggsy Bogues. Muggsy Bogues is the shortest player ever to play in the NBA at five feet three inches and a 135 pounds. Our tree would've classified him as a jockey. The machine learning model that we built up using our training data would be clearly wrong in this case. That's because no threshold is ever perfect. Any ML model is likely to get outliers wrong. Decision tree models for either classification or regression optimize tree construction. Their job is to get the best possible decision tree to represent the data. And their objective is to minimize the impurity of each node. At this point you know that every leaf node is associated with a label. A node is considered to be impure if it has misclassified data points. In our decision tree model the bottom left leaf node has been classified as a basketball node. But within that node you might find eight basketball players

and two jockeys from actual data. That node is considered impure. In our training data, we have two jockeys whose weight is above 150 pounds they are the impure elements in this node. The bottom right node on our decision tree model has been classified as a jockey node. But within that node we might find from our training data that we have seven jockeys and three basketball players. This is once again an impure node. The basketball players are the impure elements at this node. In a decision tree model, there are two ways by which you can measure impurity. One is the Gini impurity and the other is the Entropy. We'll look at Gini impurity in a little more detail in this course. We won't be looking at Entropy. But the important thing for you to remember here is that both of these yield very similar trees. Let's understand Gini impurity in a little more detail. We will consider a part of our decision tree that you see on the left. When we build classification and regression trees, our machine learning model seeks to minimize the Gini impurity at each node. And this Gini impurity is found from rule violations in our training data. Let's consider the specific example of the left leaf node that has been classified as basketball. Our training data yields 100 samples at this node. All players with height greater than six feet. Of these players in our training data 95 are basketball players, but there are five jockeys as well. So the Gini impurity is calculated as one minus 95% square minus 5% square, which gives us a value of 0. 095. Let's consider another node. The node on the bottom right, which we've classified as a jockey node. There are 100 samples or 100 training data points at this node as well, of which there are zero basketball players and a 100 jockeys. This node is an example of a completely pure node and its Gini impurity will be equal to zero. Gini impurities can be calculated at intermediate nodes as well, not just leaf nodes. Let's consider the height node here. Within the height node there are 200 samples of data from our previous examples. If you go back a little in this video clip, you will know that 95 of these samples are basketball players and a 105 of these samples are jockeys. This gives this node a Gini impurity of 0. 49875. Decision trees are very good machine learning models to use in certain used cases. One advantage of decision trees is that it is White Box ML. You don't need to leave it completely to the machine learning model. You can leverage the opinion of experts to build up your decision variables. Decision trees are also non-parametric machine learning models. There isn't a lot of hyper parameter tuning involved. The machine learning model that you get is typically the best possible model. An additional advantage is that it requires little data preparation. You can usually feed in your training data as is. There are always trade-offs for any ML model. Decision trees come with their own drawbacks. Decision trees are highly prone to over fitting. This is a common risk when you have non-parametric ML models. As we've studied before, overfitted models tend to be high variance models which means small changes in data causes big changes in the model. This is what makes decision trees slightly unstable.

Random Forests

The decision tree over fitting problem can be mitigated by using random forest. Random forests work on the principle if everyone in the room is thinking the same thing, then somebody isn't thinking. Random forests are basically a collection of decision tree models. Every decision tree within a random forest has been trained on a random sample of data. Different data sets for each tree. The output of each of these individual decision trees are then combined together in some intelligent way in order to get the final predicted label. If you're using a random forest algorithm for regression, then you might simply average the output of all your individual decision trees or if you're using a random forest algorithm for classification, you might use the mode. This will be most frequently occurring label output for each of your decision tree models within the random forest. A short discussion of random forests is useful because this is a first introduction to an ensemble learning technique. Random forests are an extremely powerful technique. They build on the output of many individual ubiq models combine the outputs together in an intelligent way to produce a stronger more robust ML model. Ensemble learning works best when the individual models that make up the ensemble are as different from each other as possible.

Gradient Boosting Regression

Armed with an understanding of decision trees we are now ready to move on to starting gradient boosting for regression. Gradient boosting algorithms work on this principle. Build up your weaknesses until they become your strong points and that's exactly what gradient boosting does. Gradient boosting uses many weak decision tree models in order to build a strong one. So you have the first decision tree, which tries to learn from data, whatever that tree could not learn is passed on to the next decision tree, which tries to learn from the previous tree's mistakes. And this process continues. Every tree tries to correct and work on the mistakes made by the previous trees. Thus our gradient boosting algorithm tries to boost the performance of its machine learning model by having many ML models come together to work on the same training data. All of these individual ML models are weak learners, but many weak learners make a strong robust model. That is the gradient boosting model. Let's use a very tiny bit of mathematics in order to understand this a little better. Let's assume that each of our individual learners are linear learners. They use linear regression. A1 and B1 are the coefficients of model one. E1 is the residual. The portion that remained unlearned by this model. The second of our weak learners will focus on these unlearned bits. Model two will try to capture E1 using the coefficients E2 and B2. Being a weak learner it may not capture this perfectly. It will have a residual E2 as well. Model three will now try to learn from the mistakes of model two. All three of these models together will give you

a combined model, which looks like what you see on screen. The residual of the combined model is E3. That is the residual of model three. The beauty here is that each of these individual models that we are using within gradient boosting are weak learners. The models themselves can be fairly simple. They in fact should not be very complex. All of these weak learners put together and combined in an intelligent way gives us a strong learner. A good machine learning model. It's useful to look carefully at what each model tries to learn. The first model learns from the training data and produces residual E1 that portion of the information from the training data that model one fail to learn, we will try to learn in model two. The main job of model two is not focus on the entire data set. Focus only on the mistakes of the previous model. Model two being a weak learner by itself might once again fail to capture some information in the underlying data. The residuals from model two will be learnt by model three. It's the job of model three to focus on what the previous model, that's model two failed to learn. Till finally the only residual that remained unlearned by our combination of models is E3. These individual weak learners are our ensemble of learners and this ensemble can comprise of a 100 200 even 500 models. The strong learner is our gradient boosting model and the combined model is the sum of the outputs of weak learners. So what machine learning algorithm do these weak learners implement? It has been found in practice that when these weak learners implement MSE regression or ordinary least squares regression, gradient boosting does not work. Gradient boosting regression works beautifully with decision trees, which is why our gradient boosting algorithm in scikit-learn uses decision trees as its weak learner.

Gradient Boosting Regression and Shrinkage Factor

We don't need to cover the exact mathematical details but it is important for you to know that decision trees work very well as weak learners in a gradient boosting model, while regression models issues ordinary least squares do not. Which is why we use decision trees in scikit-learn. Decision trees are a non parametric model. The only hyper parameter that you can tune is the depth of the tree. The weak learner decision trees in our gradient boosting algorithm will typically be very shallow trees. Shallow trees might be weak learners but they're also less prone to over fitting the training data. In all our examples showing weak learner so far we've shown you three maybe four weak learners but actually gradient boosting works best when there are a large number of weak learners of the order of several hundred. The output of every learner is fed on to the next learner. The early learners, the ones which come at the beginning of this chain learn the most. Later learners learn from the mistakes of the early learners. Just like the random forest machine learning model that we spoke of in an earlier clip, gradient boosting is a form of

ensemble learning, where the word ensemble is together in french. Many weak learners come together to produce a stronger model. We first mentioned ensemble learning in this course as a standard technique to mitigate over fitting on the training data. Ensemble learning can be thought of as yet another regularization technique, just like lasso and ridge regression. The implementation details of how gradient boosting regularizes models is different. It doesn't add a penalty to the objective function. Instead it reduces over fitting and the variance error by using many weak learners together. Gradient boosting also uses something called the shrinkage factor. This is a multiplicative term, which is applied to each one of our weak learners which make up our ensemble. The shrinkage factor scales the output of individual weak learners by a constant factor. This mitigates over fitting and makes the model learn more slowly. You will see that until passing the shrinkage factor as an input argument for gradient boosting model. This scales the output of each model by a constant factor. Higher the value specified for the shrinkage factor less the importance of an individual learner in our model. The objective of using such a shrinkage factor is to slow down the learning process. Slower the learning, less overfitted our model. The shrinkage factor is a way of forcing subsequent learners to focus on the important patterns within the data. When you are setting up your gradient boosting model, you should adjust your shrinkage factor based on the number of learners that you plan to use. If you plan to use a large number of learners, set up your shrinkage factor such that you shrink a lot. You will choose a shrinkage factor less than one. If you're using just a few learners you will shrink just a little bit. Your shrinkage factor should be close to one in that case. Typical values for the shrinkage factor is 0. 1 or 1.

Demo: Gradient Boosting Regression with Grid Search

It's finally time for a demo once again. We will see how we can perform gradient boosting regression in scikit-learn. Even though gradient boosting is an example of ensemble learning and has many weak learners, using the gradient boosting algorithm is exactly the same as using any other estimator in scikit-learn. Input arguments to the gradient boosting regressor include the number of estimators. These are the number of weak learners that our algorithm will use. You can also constrain the individual weak learners by specifying the maximum depth of each decision tree. Remember these decision trees should be shallow, so that they don't overfit on the training data and are weak learners. In this demo, we will use the same automobile price prediction data set, that we set up in the lasso and ridge regression example. So we can simply open up that iPython notebook and make a copy. Once the copy has been created, we can go ahead and rename this iPython notebook to reflect that it will now have the gradient boosting algorithm. This is the same automobile price prediction data. We will read in the data in exactly the same

way and the rest of the steps for data preparation also remain exactly the same as before. The thick red line that you see here on screen marks the end of the data preparation phase. This is where we've accessed the training and test data set. We can go ahead and delete all the cells which follow this red line. Those relate to the linear regression lasso and ridge regression model. Go ahead and execute all the cells from the beginning up to the current cell. That completes our data preparation. At this point we have the training and test data and can go ahead with building the gradient boosting model. The scikit-learn library contains an estimator for the gradient boosting regressor. Here are all the parameters that you can pass into the model. The number of estimators is equal to 500. These are the number of boosting stages to use in our model. We want to constrain the depth of each decision tree which is our weak learner. We will constrain it to six. We can also specify a condition for splitting a node. A node should have at least two samples before we can split it. The loss calculation for our gradient boosting regressor uses the least squares regression loss. Go ahead and run training on the model on the automobile price prediction data. And let's see what the R square of the trained model is on the training data set. It's at 99%, which is a really high value for the r square. Let's perform some predictions using this model. We will perform predictions on the test data and as usual let's plot a matplotlib graph of the predicted y values against the actual y values and they seem to be rather close. Our model seems to have performed reasonably well on the test data and we can confirm this by calculating the R square on the test data and this is really high as well at 93. 2%. This is clearly a good machine learning model. Let's calculate the root mean square error. We get the mean square error first and then calculate the square root and this is at 2. 93, which is pretty decent given all our previous examples. The gradient boosting regressor has a number of hyper parameters, which means it's a great candidate for us to use grid search to find the best possible model. Here are the different parameters that we want to pass into grid search. We want to try models for all of these combinations. Num estimators, learn rates and max depths of individual decision trees. The learn rates represent the shrinkage factor that we apply to each weak learner. Set up the parameter grid. This is the dictionary and then instantiate the GridSearchCV with the gradient boosting regressor. We will pass in the parameter grid and specify cross validation. We will also pass in a different argument, written train score is equal to true. For every model that grid search trains, we want a score indicate how well that model performed. Wait for a little bit so that grid search can complete its training. And here's the best possible model. The learning rate should be 0. 1. The max depth of a decision tree should be four and the number of estimators should be 200. You can access the results of every individual model that was set up using Grid Search by using the CV_results number variable. This grid search has actually set up 36 machine learning models. There are 36 parameter combinations. Three estimators, four learn rates and three max

depth parameters. We will use this for loop to retrieve and find the rank of each machine learning model. Here you can see the results. The worst model with rank 36 was the one with learning rate 0. 01 max depth four and just 100 estimators. You can scroll through and see the ranks of the various parameter combinations. Our best model is here with rank one. Now that we know the hyper parameters that we have to use for the best possible model, let's instantiate a gradient boosting regressor with these parameters. And then run training and prediction using this regressor. When we plot the predicted values against the actual values on the test data, we will see that they are very close but calculating the R square will give us how good our model really is. The R square on test data is 97%, which is really high. And the root mean square error, you will find is correspondingly very low. Just around 1386. And on this note we come to the very end of this module. This module we covered support vector machines which are a very popular technique for classification algorithms. SVMs work on both text as well as image data. We saw demos of both. In this module, we also saw a real live example of ensemble learning. gradient boosting regressors which use decision trees as their weak learners. In the next module, we will focus on implementing unsupervised learning techniques in scikit-learn. Specifically the mean shift clustering algorithm and principal components analysis.

Implementing Clustering and Dimensionality Reduction in scikit-learn

Module Overview

Hi, and welcome to this module on unsupervised learning techniques. We'll see how we can implement clustering and dimensionality reduction in scikit-learn. Clustering is a popular and elegant unsupervised learning technique which helps find patterns in the underlying data. Clustering does not use any Y variables or labels on the data. It looks at the data structure itself. Common clustering algorithms are K-means algorithm, hierarchically clustering, and mean shift clustering. Today the problem is no longer of scarcity of data. We have a lot of data and a lot of that data might be meaningless. Dimensionality reduction represents the input data in terms of their most significant features, and tend to improve the performance of machine learning models. One of the most widely use techniques for dimensionality reduction is principal components

analysis. We studied early on in this course that machine learning models can be divided into two broad categories. Supervised learning techniques require labeled training data. Unsupervised learning techniques do not need label instances. Instead, they try to find patterns within the data itself.

Clustering

Let's first understand how clustering works and how we can use it with any kind of data. The important principle behind clustering is that anything can be represented by a set of numbers. Whether it's an object, a person, a document, or a webpage, all of these can be represented in some numeric form. Let's consider a person. A person is of a certain age that can be represented on a number line. A person may be of a certain height. All you need to do then is to represent this information in two dimensions. The person is a point on this plane. Let's say you were to add a third dimension. A person has a certain weight. Now this individual is represented using three distinct pieces of data. Now, assuming that you have a whole bunch of other information about this person, you could then use an N dimensional Hypercube to represent the set of N numbers. The basic principle is that all the information about a particular person can be represented in numeric form. Now let's take the example of Facebook users. Facebook users have certain characteristics. Different users have different characteristics. Hypothetically, you could have a set of Facebook users where each user is a point in an N dimensional Hypercube. Clustering involves finding groups of people within this data who have the same characteristics. What those characteristics are can differ. It could be that they like the same music, they went to the same high school, anything. Clustering results in the formation of groups within the data where people within the same group are similar. People who are in different groups are different. Let's say you were to change the features on the basis of which you performed clustering. You could end up with a completely different set of groups. One of these groups could be parents with children under five. Another group could be parents of teenagers. If you think about the Facebook example, clustering of users is important because then you can target specific ads to specific groups. So, how well did your algorithm cluster the underlying data? This can be measured by considering the distance between individual points in a cluster. Smaller this distance, better the clustering. The distance between users in a cluster is a measure of how similar the users are and the goal of clustering is to maximize intra-cluster similarity. In addition, we also want our clustering algorithm to ensure that the distance between users who are in different clusters is as large as possible. We want to minimize inter-cluster similarity. A good clustering algorithm will try

and achieve both of these objectives to the best of its ability, maximize intra-cluster similarity, and minimize inter-cluster similarity.

K-means Clustering

One of the most popular machine learning algorithms to perform clustering which allows us to maximize inter-cluster similarity and minimize intra-cluster similarity is the K-means clustering algorithm. In this module, we'll primarily study the mean shift clustering algorithm, however, it's important to understand how K-means clustering works. The basic principles are very similar to mean shift clustering. Let's say we have a number of points in two dimensional space. This can be extended to N dimensional space. We'll work with two dimensions because that's simpler to visualize. We start off by initializing K centroids or the K-means of the clusters. In K-means clustering you have to specify this value of K up front, how many clusters you want your data to be divided into. Once you have K cluster centers assign each point to a particular cluster. In order to do this, we calculate the distance between every point and every cluster center. A point is assigned to that cluster whose cluster center it is the closest to. Once you've assigned all the points, you'll see cluster set up like this. At this point in time, use the existing points in each cluster to recalculate the mean for each cluster. Once the cluster centers have been recalculated you'll find that certain points will move to another cluster. We recalculate the distance from all cluster centers and reassign the points. This process of recalculating the means of each cluster and then reassigning the points once the new means have been calculated continues 'til the points reach their final position. When the cluster centers and the corresponding points don't move anymore, that's when the algorithm has converged. After convergence, you can think of every cluster being represented by a single point and this point is the reference vector. This reference vector is the center of the cluster and because it is calculated as an average of all points that belong to a cluster it's called the centroid of the cluster.

Mean Shift Clustering

Now that we've understood how K-means clustering works let's look at mean shift clustering. We start off with the same points in space. We'll assume two dimensional data because that's easy to visualize. But this can be extended to N dimensions as well. The final neighborhood for every point, a good way to imagine this neighborhood is to imagine a circle around every point. All points within the circle are neighbors of that point. Once every point has a neighborhood defined the next step is to calculate a function based on all points that are present within a certain

neighborhood. This function is called the kernel. This kernel function is applied to all the points in the neighborhood of a particular point. By changing the size of the neighborhood you can have this kernel function applied to more points or fewer points. The simplest of all kernels to envision is the flat kernel, which is a simple sum of all points which are present in the neighborhood of a certain point. Each point gets the same weight. No point is more important than the other. However, in mean shift clustering it's more common to use a Gaussian kernel or an RBF kernel. RBF stands for radial basis function. Rather than a simple sum of neighborhood points this kernel would apply a probability weighted sum of points. The probability distribution used is the RBF or the Gaussian probability distribution. This is a probability distribution function that you're hopefully familiar with from high school statistics. You don't really need to know much about it, except that it has this characteristic bell shaped curve and a Gaussian probability distribution is defined by its mean represented by mew and the standard deviation represented by sigma. The mean of this Gaussian kernel is the center point, the point mew around which we've defined this neighborhood. The standard deviation of this Gaussian kernel is proportional to something called the bandwidth. The bandwidth is a hyperparameter that is used in the mean shift clustering algorithm. Though it's not exactly the same, you can think of the bandwidth as the radius of the neighborhood around a certain point. In the mean shift clustering algorithm this RBF or Gaussian kernel is applied to every point in our data set. Every point has its own neighborhood and a kernel function applied. Based on the number of points in each neighborhood and the way the points are distributed the magnitude of this RBF kernel is different for each point. Now let's color code all the points in our data set based on the magnitude of the RBFs. The points that are darker are those that correspond to a high RBF value. These are the peaks of the probability distribution. Lightly colored points correspond to low RBF values. These are the troughs of our distribution. Mean shift clustering is an iterative process wherein all points start shifting towards the nearest peak. Imagine that these points are climbing the hills of the probability distribution functions. This process where the points climb the hills is called mean shift and this continues 'til the algorithm converges. The algorithm converges when the points stop moving. They've found their closest peaks. Let's talk about the bandwidth hyperparameter that we mentioned earlier. The standard deviation of the Gaussian kernel that is applied to every point is proportional to the bandwidth hyperparameter. This is the only hyperparameter for the mean shift clustering algorithm. Tweaking this parameter is similar to adjusting the radius of the neighborhood around each point. Bandwidth is not exactly the radius though, so make sure you don't confuse the two. If you have a small bandwidth you end up with a tall, skinny kernel. If you have a large bandwidth your kernel will be flat. Here is an example of a tall, skinny kernel. A kernel with this probability distribution will tend to ignore those points that are far away from the mean. It'll give a higher weight to

points that are closer to the mean. For a larger bandwidth the kernel will tend to be flatter. This will consider or take into account points which are even far away from the mean. For a completely flat kernel all points in the neighborhood have the same weight.

K-means vs. Mean Shift Clustering

In order to understand mean shift clustering better it's helpful to compare and contrast it with K-means clustering. In K-means clustering you need to specify the number of clusters you want to divide your data into up front. In the case of mean shift clustering however you do not need to specify how many clusters you'll have at the end. The algorithm will decide for itself. One disadvantage that K-means clustering tends to have, that it cannot handle some complex non linear data. Mean shift clustering on the other hand because it has a density function associated with every point can handle even non linear data such as pixels. When you perform K-means clustering there's almost no hyperparameter tuning required. The only hyperparameter is key the number of clusters. In order to get a good clustering model using mean shift clustering you require hyperparameter tuning of the bandwidth parameter. In terms of how computationally heavy the algorithms are mean shift clustering is far more computationally intensive as compared with K-means clustering. This is because of the difference in computational complexity for the individual algorithms. K-means clustering is order of N, where N is the number of data points. For mean shift clustering it's order of N squared. However, if you have data points that are outliers, not really very close to some clusters, K-means clustering struggles with that data. Mean shift clustering copes better with outliers.

Demo: Mean Shift Clustering

Time once again for a demo. Let's see how we can implement mean shift clustering in scikit-learn. We'll use the I Python notebook with the prefix M four demo one for this code. Set up the import statement for the pandas library as usual. For this particular demo, we'll use the famous Titanic data set. You can download the original data from kaggle. com. This data set contains information about several passengers who were on the Titanic. There's a bunch of information that we have on each of the passengers, the class the passenger was traveling in, the name of the passenger, sex, age, whether they were traveling with siblings, and so on, and we can use all of these passengers to predict whether or not they survived the sinking of the Titanic. In our clustering algorithm though we'll use this data set a little differently. We'll try to find groupings of the passengers with similar characteristics. When you examine the input data you'll find that all of the

features are not useful. There are some features that are too specific to an individual passenger, such as passenger ID, name, the ticket, the cabin, and so on. Let's drop these features, and focus our attention on the remaining data. Here's the data set now with far fewer columns. The survived column tells us whether the passenger survived the sinking of the Titanic or not. A value of zero indicates that the passenger did not survive. One indicates that the passenger survived. These columns are pretty self explanatory. The class in which the passenger was traveling, first, second, or third, whether the passenger was male or female, how old the passenger was, and the fare column is how much their ticket cost. The middle two columns might require some additional explanation. This is the number of siblings or parents who were also aboard the Titanic for that passenger. These passengers were traveling with family. The embarked column represents where the passenger boarded the ship. The port of embarkation could be one of the following, C stands for Cherbourg, Q for Queenstown, and S for Southampton. Let's start pre processing this data to feed into our machine learning model. The first step is to convert the gender column which contains categorical variables into numeric form. Here we have the gender converted to numbers using the label encoder. One is male and zero is female. We'll convert the data in the embark column to one hard representation, C, S, and Q. Let's check to see whether there are any invalid values in our data set. We'll have to clean up those rows. Yes indeed, there are 177 rows with data missing. Let's go ahead and drop those rows from our data. With our data set up let's instantiate the estimator for the mean shift clustering algorithm. The only hyperparameter that we pass in is the bandwidth. Remember that the standard deviation is proportional to the bandwidth. Smaller values of bandwidth will result in tall, skinny kernels. Larger values of bandwidth will result in short and fat kernels. Go ahead and call the fit method to start training on the data. The bandwidth parameter that we pass in when we instantiated our mean shift estimator was largely a guess, however, you'll not need to guess at the bandwidth parameter. There is an estimate bandwidth function that scikit-learn provides which gives you a good value of bandwidth for your data. If you do not specify a value for the bandwidth this is the function that our estimator will call under the hood. Now, this function takes a long time to run. It runs in quadratic time where N is the number of samples. N is also a parameter that you can specify. For our Titanic data set this function estimates that 30 is a good value for bandwidth to get proper logical groupings of our data. Let's continue with our estimator which we set up using a bandwidth of 50 and call training on our Titanic data. Let's see how many clusters our data was distributed into. A bandwidth of 50 produces three clusters. Each of these three groups will contain passengers with similar characteristics. Let's add a new column called cluster group to our data frame. This column will contain information as to what cluster a particular passenger belongs to. Include the clusters generated by the mean shift algorithm in the same data frame where the data resides. Here you

can see that the new column has been successfully added to the very end of our data frame. Before we study the characteristics of passengers who belong to the same cluster let's take a look at the entire data set. We have information on 714 passengers who were on the Titanic. Of these, 40% survived, the rest did not. The average class across all of these passengers is 2. 23 which indicates that there more people in lower classes such as second and third class as compared with higher classes such as first class. In this data set of 714 passengers there are more males than females. A value of one represents a male. A value of zero represents a female. The average fare paid across all passengers in this data set is around $35.

Demo: Examine Mean Shift Clusters

Now that we've seen the characteristics of the entire data set let's group this data by cluster and see how similar the passengers in each cluster are. Perform a group by using the cluster group column that we just added and find the mean for each cluster. This will be the average of all characteristics within each group. We have the averages for each of the characteristics but this information would be far more useful if you could also see the number of samples that exist in each cluster. Add a counts column to our data frame and set its value to the size of each cluster. This should immediately show you that we have two clusters where there are sufficient data points for it to actually be significant. The third cluster has just three people within it. There's not really that much we can learn from that cluster. The first cluster with 680 people were mostly individuals who did not survive. The survival rate was around 38%. They belonged to the lower classes, second and third classes mostly. The 0. 65 in the sex column shows that there are more males than females in this cluster. Remember one represents a male and zero a female. The average fare paid by the passengers in this cluster is around 25. The second cluster comprises mostly of people who survived the Titanic disaster. You can see that the survival rate is 74%. Notice that they're mostly from first class and they're mostly female. A 0. 25 in the sex column shows us that. Also the average fare paid by these passengers is 192, far higher than the previous cluster. The last cluster has just three data points. That's not really significant. We can't really read too much into this. Let's study one of these clusters, the second cluster with 31 passengers in it, in a little more detail. Take a look at this data. These are mostly survivors and see whether anything interesting pops out for you. You can filter the contents of the data frame to see all the passengers that belong to this cluster. The one thing that struck me here was how high their fares were. These are clearly first class passengers. Of the list that you can see on screen there are only four passengers who did not survive the sinking of the Titanic. Three out of these four passengers are young males. The fourth passenger who did not survive was a two year old girl. Let's try mean

shift clustering once again, but this time we'll change the bandwidth parameter to be 30. Remember this is the value that we got from our estimate bandwidth function. This is the before that our mean shift estimator might have used had we not explicitly specified the bandwidth. Go ahead and execute all the cells by hitting shift enter. Notice that we have five groupings now in the underlying data. Execute the rest of the cells. Include the cluster group and the number of people in each cluster as a part of your data frame. Notice there are five clusters. Only the last cluster is insignificant with just three passengers. The first cluster mostly includes people who did not survive the disaster. They usually occupied cabins in the second or third class, were mostly male, and had paid a low fare for their ticket. The second cluster includes passengers that were more likely to survive. The survival rate was 61%. These were mostly first and second class passengers with an even split of male and female. They paid an average fare of 65 for their ticket. As we move to other clusters the passengers are more likely to have survived. These two clusters comprise entirely of first class passengers and are mostly female. They've also paid a very high average fare. The last cluster has just three individuals, so it's hard to read much into this cluster, but it comprises mostly of males and they've paid a very high fare and they've survived.

Principal Components Analysis: Intuition

Principal components analysis is a very popular dimensionality reduction technique which is often used with large data sets with many features. The idea behind PCA is to express complex data in terms of a few well chosen vectors. These vectors are called principal components and these principal components more efficiently capture the variation in that data. If that was a mouthful and you didn't really follow what was going on, don't worry. Let's try and understand PCA intuitively. Let's say that you had a bunch of data points scattered in this two dimensional plane. Our objective is to find the best directions in which to represent this data. In order to represent data we need dimensions. We can start off by drawing a line in some direction on this plane and then projecting the data onto that line. How do you know whether the direction that you've chosen for your line is a good one to represent this data? The greater the distances between the projections of your data the better the direction that you've chosen. A projection is said to be a bad one when information is lost. Here, all of the individual points project onto a single point on the X axis. This is a bad projection. A projection where information is preserved and the distances between the projected points are maximized is a good projection. Let's go back to our data in two dimensions. The first direction which you'll choose to project these points is the one where variance is maximized, and this is the first principal component of your data. We'll then seek to find the next best direction in which to project the points. The next best direction is at right

angles to the first principal component and is called the second principal component. We don't need to understand the mathematics behind why principal components should be at right angles to each other, however, intuitively it makes sense that directions at right angles help express the most variation with the smallest number of directions. The first principal component will capture the most variation in the input data. The second principal component will capture a little less variance, and so on. If your original data set is expressed in two dimensions the number of principal components for this data will also be equal to two. In general, there are as many principal components as there are number of dimensions in the original data. Principal components analysis involves expressing any data set in terms of its principal components. So we are reorienting the data along the new axis. These axes are the principal components. If the number of principal components required to express our data is the same as the number of dimensions in our data what was the point of this reorientation? Once data is expressed in terms of principal components if the variance along the second principal component is small enough we can simply ignore it and use just one dimension to represent the data. This is dimensionality reduction. For the data that you see on screen variation is present along two dimensions which means we require two principal components to represent this data. However, if the data was set up in a slightly different way like what you see on screen now the variation is mostly present along one dimension. You can just have one principal component to represent this data. Let's go back to our definition of principal components analysis and see what it holds. We want to express complex data. This is data which of a high dimensionality. Data which has many features, and we want to express this data using only its most significant components. These are a smaller number of mew dimensions. Every feature vector with N features X one through to X N will be re-expressed with fewer features. Let's say F one and F two. The trick is that F one and F two are so chosen that it captures the maximum possible variation in the data. Very little information is lost.

Demo: Principal Components Analysis

We've come to the final demo of this module and of this course. Here we'll implement principal components analysis in scikit-learn. We'll write code in the I Python notebook with the prefix M four demo three. Go ahead and import pandas. We always use pandas to read in our data set. For this example we are going to choose the wine quality data set. This data set is available for download at the University of California Irvine's website. Here are the characteristics of the wine which form our feature set. When we read in the data we want to skip the header row and the separator used within the data set is the semi colon. This data set is great for students of machine learning. It's a very clean data set. There are no missing values and no values that need to be

cleaned up. All values are numeric. There are no categorical variables. In this demo we'll perform principal components analysis and use the reduced dimensionality data set for prediction. We want to predict the quality of the wine. Here wine quality is a categorical variable expressed in numeric form already. There are seven possible quality scores. Random guesses about the quality of the wine will give us an accuracy of just about 14%. Set up our X and Y variables. In principal components analysis we'll only use the X variables. It's unsupervised learning technique. We'll standardize our X data by subtracting the mean and diving by the standard deviation. We'll do this using the pre processing library. We'll go ahead and split our data into training and test data in the usual way using train test split and then let's use a support vector machine classifier to perform classification using all the features of the input data. We haven't performed PCA yet. Perform predictions on the test data and let's calculate the accuracy of our test predictions. Here the accuracy is around 49%. This is a baseline score using our model. This accuracy isn't really that bad. Remember random guesses would give us about 14% accuracy which means our model does much better. However, what we're most interested in in this demo is how reducing the number of features that we use to train the model affects the accuracy. Now, dimensionality reduction only helps if features are correlated with each other. Here I'm going to do something interesting. We'll use the seaborn Python visualization library in order to see which of the input features are strongly correlated with other input features. The seaborn library can be accessed using a simple pip install. We'll use seaborn to set up a heat map. A heat map is a visualization tool which will allow us to see which features are strongly correlated with one another. And here is our heat map. Cells which are in light green indicate strong correlations. Notice that the entire diagonal is in light green. Every feature is strongly correlated with itself. There are some other strong correlations that you can see here. The total sulfur dioxide content of a wine is strongly correlated with the free sulfur dioxide content. Another strong correlation is the density of the wine with the residual sugar in the wine. Performing principal components analysis for dimensionality reduction should definitely help with this data set. Let's go ahead and import the PCA library and instantiate our estimator. To start off we won't really reduce the number of input dimensions. There are 11 features in our data set. We'll use 11 principal components. Call fit transform to run PCA on the data set. The explained variance member variable gives us the magnitude of variation captured by each of the principal components. Absolute numbers here don't make much sense. So we'll use the explained variance ratio. This gives us the same information but in percentage terms. You can see that some dimensions play a large role in defining the data, others do not. One of the choices that you need to make when you're using principal components to represent your data is how many dimensions do you need to capture most of the information in your data. A great way to identify this visually is by plotting the

dimensions on the X axis and the explained variance ratio on the Y axis. This graph is called a scree plot. Scree plots typically have an elbow. Notice that most of the variation in our input data is captured using just two dimensions. Let's now use the principal components of our training data in order to perform classification using support vector machines. We train the model and then run prediction on the test data and then we see what the accuracy of our predictions are. The accuracy is exactly the same with the transform dimensions. Remember, we haven't reduced the dimensions yet. Let's actually reduce the number of dimensions that we use to represent our data. We change this to nine. The two dimensions which are the least relevant, the last two principal components, will be dropped. Go ahead and execute all the cells. Run training, run prediction on the test data and let's calculate the accuracy of our test. The accuracy is 49. 38%, which is marginally higher in spite of having to remove two dimensions. Let's reduce the number of dimensions in our training data further. From nine we'll go down to six. The last three dimensions will be dropped. Once again, run training, calculate predictions, and let's see the accuracy of our predictions. It's 45%. This is pretty good considering we are using less and less of our training data. This time we'll get serious about dimensionality reduction. Let's see how well our predictions perform if we use just one dimension, the most important one. Run training, then predict, and you'll see that the accuracy on the test data hasn't fallen much at all. It is 44%. This is very close to the 49% accuracy that we got when we used all the dimensions. If you have a large data set with many features you'll find that many of those features are unwanted. You'll be able to make your training faster if you use a dimensionality reduction technique such as the PCA.

Summary and Further Study

And this brings us to the end of this module, and to the end of this course on scikit-learn. In this module, we covered unsupervised learning techniques. We saw that clustering is an important algorithm which helps find patterns in the underlying data. The K-means clustering algorithm is one of the most popular to find logical groupings of data. In this particular module we studied mean shift clustering. We studied how this was implemented in scikit-learn and how it compares with K-means clustering. When you're working with large data sets that have many features dimensionality reduction is an important technique that allows you to improve your training process. Instead of using many features that might be correlated you'll only use those features that are the most significant which represent the underlying data best. We saw how we could use principal components analysis to extract latent factors in our data. If you're a student of machine learning and you wish to take your studies further here is a great book that you can use, Hands-On Machine Learning with Scikit-Learn and TensorFlow by Aurelien Geron. I highly recommend

this book. It's easy to read. It's fun and very interesting. Other courses on Pluralsight that will teach you other machine learning techniques and libraries are How to Think About Machine Learning Algorithms, Understanding Machine Learning with Python, and Understanding the Foundations of TensorFlow. I hope you enjoyed this course. Thank you for listening.

Course author

Janani Ravi

Janani has a Masters degree from Stanford and worked for 7+ years at Google. She was one of the original engineers on Google Docs and holds 4 patents for its real-time collaborative editing...

Course info

Level	Beginner
Rating	★★★★½ (94)
My rating	★★★★★
Duration	3h 13m
Released	30 Apr 2018

Share course