Understanding Machine Learning with Python

by Jerry Kurata

Start Course

Bookmark Add to Channel Download Course

Table of contents Description **Transcript** Exercise files Discussion Learnin

Course Overview

Course Overview

Hi, my name is Jerry Kurata, and welcome to my course, Understanding Machine Learning with Python. These days, machine learning is all around us, from helping doctors diagnose patients to assisting us in driving our cars. As we go about our day, we may be utilizing machine learning applications and not even realize it. It silently scans our email inboxes for spam emails and ensures that stores are stocked with the goods we want to buy when we need them. This course will introduce you to machine learning and the technology behind it. You will see why companies are in such a rush to use machine learning to grow their business and increase profits. You will learn how developers and data scientists use machine learning to predict events based on data, specifically you will learn how to format a problem to be solvable, where to get the data, and how to combine that data with algorithms to create models that can predict the future. Throughout this course, we'll utilize Python and a number of its libraries to make creating machine learning solutions easy. However, you do not need prior experience with Python. In this course, we learn by doing, and the code we will use will be explained as we create our solution. By the end of this course, you will know the how, when, and why of building a machine learning solution with Python. You will have the skills you need to transform a one-line problem statement into a tested prediction model that solves the problem. I look forward to you joining me on this journey of understanding machine learning with Python from Pluralsight.

Getting Started in Machine Learning

Introduction

Hi I'm Jerry Kurata. Welcome to the Pluralsight course on Understanding Machine Learning with Python. In this course you'll learn how to apply machine learning to solve problems that are difficult, and some might say impossible to solve with standard coding techniques. This first module will provide some basic information about machine learning. This includes examples of machine learning, a definition of machine learning, and importantly, how machine learning differs from traditional programming. We will go over the two basic types of machine learning, supervised and unsupervised. We will see each of these types in action, which will clarify how they differ and when each type of machine learning should be used. After that we will review the content of this course, and the skills you need and the skills you do not need for this course. We will then have a brief discussion of how machine learning fits into the larger subject of data science. There's also a section on programming in Python and using the Jupyter Notebook environment we will use in the demos.

What Is Machine Learning?

Machine learning is one of those technologies that is all around us and we may not even realize it. For example, machine learning is used to solve problems like determining if an email is spam, how cars can drive themselves, and what product a person is likely to buy. Every day you see these types of machine learning solutions in action. When you open your email and your messages are automatically scanned, classified as spam, and moved to your spam folder. For the past few years, Google, Tesla and others have been building self-drive systems that will soon augment or replace the human driver. And data giants like Google and Amazon have been able to use your search history to predict which items you're looking to buy, and ensure you see adds for those items on each and every webpage you visit. All of this useful, and sometimes annoying and creepy behavior, is the result of machine learning. But before we proceed, what do you think of when you hear the term machine learning? Perhaps something like this? While the thinking robots we see in movies likely use machine learning, defining the complex logic that lets the robots think is unfortunately beyond the scope of this course. So let's define machine learning in general, and the specific type of machine learning this course focuses on. For this course we'll define machine learning to be building a model from example inputs to make data-driven predictions versus strictly following static program instructions. This definition points out the key feature of machine learning, namely that the system learns how to solve the problem from example data, rather than you writing specific logic. This is significant departure from how most programming is done. In more traditional programming we carefully analyze the problem and write code. This code reads in data and uses its control logic to determine the correct parts to execute, which then produces the correct result. With traditional programming we use constructs such as if statements, case statements and control loops implemented with while and until statements. Each of these statements has tests that have to be defined. And the changing data typical of machine learning problems can make defining these tests very difficult. In contrast with machine learning, we do not write this logic that produces the results. Instead, we gather the data we need and modify its format into a form which machine learning can use. We then pass this data to an algorithm. The algorithm analyzes the data and creates a model which implements the solution to solve the problem based on the data.

Types of Machine Learning

Machine learning algorithms learn from data by utilizing one of two techniques. Supervised or unsupervised machine learning. While this course is focused on supervised machine learning, it is important to understand the difference between the two techniques and when you would use one versus the other. In supervised machine learning, the subject of this course, each row of data has fields containing feature values, and the value we want the algorithm to predict. For example, let's assume we want an algorithm to predict house prices. To accomplish this, we would pass a set of training data with each row of data containing features of the house such as its size, number of bedrooms, and the year the house was built, and the value we want to predict, namely the price. We pass many rows of this training data to the algorithm. The algorithm analyzes the features, and the resultant price. It determines the relationship between the features and the price and creates a model that is trained to predict the price of the house based on the features. Then when the trained model is presented with data for a new house, it executes its logic and accurately predicts the price of the new house. Let's contrast this to unsupervised machine learning. In unsupervised learning, we're looking for clusters of like data. The algorithm analyses input data and identifies groups of data that share the same traits. For example, let's start with a recording of a room full of people talking. We can convert this recording into data containing values of vocal attributes such as pitch, intonation and inflection. We can then pass this mixed voice data to an unsupervised learning algorithm. The algorithm can analyze the voices and create a model that classifies clusters of data, words in this case, that have certain speech patterns of pitch, intonation and other vocal features. This results in the ability to isolate an individual voice from the mixtures of voices. Since it is important to understand the correct technique to use to solve a problem. Let's recap the difference between supervised and unsupervised machine learning. A primary difference is what type of problem you're trying to solve. If you're trying to predict a value, like the price of the house, then you're likely looking at a supervised machine learning problem. If you're going into a set of data and trying to find groups or clusters of like data, then this likely an unsupervised machine learning problem. The data we have also matters. Supervised machine learning requires that we have some training data that has the value we are trying to predict. With unsupervised machine learning we don't have the values, we are trying to figure out the values. As we saw in the examples, in supervised machine learning, we use the data with the value to train the model so it can predict values on new models, like a new house. In contrast, with unsupervised machine learning, we get clusters of like data from the model. And importantly, in this course, we'll cover supervised machine learning and not unsupervised machine learning. This choice was made because many of the machine problems you're likely to run into are predictions and are thus solved with supervised machine learning.

Course Overview

Now let's take a few moments and go over the content of the course. We will start with an overview of the machine learning workflow. The workflow will provide the framework we'll use to approach our problem and ensure we do not forget any critical steps in developing a solution to the problem. Over the years, this framework has been followed to solve countless machine learning problems such as fraud detection, identifying spam email and weather prediction. Once we understand the structure of the workflow, we will spend the majority of the course diving deep into each of the steps of the workflow. We will do this by applying the workflow steps to the sample problem of predicting if a person will develop diabetes. Diabetes is one of the leading causes of death in the world today. Up to 8. 5% of the world's adults have diabetes. Prediction methods like the one we are building can help identify people at risk from this deadly disease. At the end of the course we'll have a short review of what you have learned and where to go from here on your journey to learn more about machine learning. But before we go further, let's review the skills and experience you should have to get the most from this course. Let's start by discussing the skills I do not expect you to have. First, as the with Python in the title implies, we're going to be doing our programming in Python. However, you do not need to have any prior experience with Python. Throughout this course, we will learn by doing. That is we will introduce the parts of Python as we use them. And in this course, we will use version 3. 5 of Python. We will use Jupyter Notebook as our development environment. And as with Python, you don't need to have prior experience with Jupyter Notebook. There's a quick getting started guide at the end of this introduction for those people who want to get a jump start on using Python within Jupyter Notebook. Finally, I do not expect you to have advanced math or statistics knowledge. While the algorithms are based on advanced math and statistics, we will be focused on using existing algorithms rather than creating new algorithms. Now let's take a look at the skills and experience you do need to have. First you need to have some sort of programming experience. That can be in C, C#, VBJava, or almost any programming language. The language specifics don't matter, but you need to be able to understand basic programming constructs such as assignment, conditional statements, loops and passing parameters to functions. Second, you need to understand and have some experience working with data in table and lists. That is, you need to understand that in a table, a given column contains data of the same type, and that rows contain one or more columns. And that in lists, all of the data is the same type. Third, you need to have some basic math and statistics knowledge. The statistics need not be much more than summation, means, median, maximum and minimum. But I am assuming at least this level of knowledge. Likewise, nothing more than basic algebra math skills are assumed. And finally, and most importantly, you need to have a curious mind and be enthusiastic. This is a course about seeking understanding through data. You will learn with a little manipulation, you can create models that can predict the future, but getting there will take a little bit of effort.

Why This Course?

I'm going to assume that if you've made it this far it's because you have an interest in machine learning. But just in case you have some doubts, let's talk about why you want to watch the rest of the course. One obvious reason for taking this course would be to add a skill set, namely machine learning. As developers we always want to constantly add skills and learn how to do things in new and interesting ways. And just like when you learned a new programming language, you needed a set of resources like this course to get you started. This course is about machine learning, and machine learning is one of the key technologies used in the broader field of data science. And as we see, data science itself is a blend of math and statistics, software development and expertise in the problem subject area. Machine learning is one of the most important parts of data science and is at the intersection of software development, and math and statistics. I know you have the software development expertise, so in this course we'll blend that with using math and statistical algorithms and models. And if you add some subject matter expertise, you can become a tech unicorn. One of the things that has made data science so exciting now is the recognition of what machine learning can do for companies. In particular, the ability to predict outcomes can directly impact a company's bottom line by defining new business opportunities and bringing in new customers. And with this new awareness has been a substantial increase in the salaries of data science professionals. Especially for tech unicorn. In fact, Forbes recently had this quote. But beyond the financial gains, I hope this course can satisfy your curiosity about machine learning and how it is applied. Once you understand the basics of machine learning, who knows where your skill will take you. Perhaps this will be your next project. If so, I for one welcome our new robot overlords.

Installing Python and Jupyter Notebook

You now have a high level understanding of machine learning and how it uses data instead of explicit program logic to determine how to solve a prediction problem. Next we have a quick introduction to Python and Jupyter Notebook. If you are familiar with Python and Jupyter Notebook, feel free to skip this information and head over to the next module on the machine learning workflow. If you've been around programming for any length of time, you've no doubt heard about Python. Python is an easy to learn language, sometimes used as the first language to teach programming. It has clear and powerful object oriented features. It uses a structure based on indentation that makes programs easy to read. And most importantly for us, comes with standard libraries that support many common programming tasks, including machine learning. Right now there are two versions of Python, 2. 7 and 3. x commonly in use. Python 3 came out in 2010 and is the future of the language going forward. However there are some incompatibilities with Python 2. 7. Python 2. 7 is the last version of the Python 2 family, and has been largely static since 2012. Many people still use 2. 7 because their code has syntax or library dependencies on Python 2. At this time, most of the major libraries support Python 3 and Python 3 is the recommended version of the language. In this course, we'll use Python 3. More specifically, we'll use Python 3. 5, and the appropriate Python 3 version of the libraries. Speaking of libraries, for machine learning, the libraries we are most interested in are numpy, which supports scientific computing through n dimension arrays, has linear algebra support, and many other numeric capabilities. Pandas, which provides data frames that make it easy to access and analyze the data.

We will use these data frames to hold and manipulate our data. Matplotlib, which is a 2D plotting library which produces publication quality graphics on our data. During the course, we will use several plots to help us understand the data. And finally, scikit-learn, which provides simple and effective tools for performing many of our machine learning operations. This includes implementing the algorithms we train into predictive models, methods that make it easy to pre-process data, methods that assist in the evaluation and comparison of model performance, and support for many more machine learning tasks. We will use all the libraries and a few more, but as you might expect, scikit-learn is the library we will use the most. Of course as developers, we know that the language is just part of the story. The development environment and associated tools are just as important. For these we use Jupyter Notebook, which was previously called IPython Notebook. The Jupyter Notebook environment creates a single, interactive document containing text, graphics and code. This is a great environment for the iterative trial and error nature in which machine learning solutions are built. As we will see, the result documents are presentation quality and these documents can be shared and worked upon by multiple users. The notebook now supports over 40 languages via kernels, which are language plugins. Languages supported include Python 2. 7 and 3. x, Ruby, R, Haskell, C#, PHP, Scala and many others. Now let's get everything installed. There are numerous ways to get Python and Jupyter Notebook set up on your system. But the way I suggest is to use the Anaconda Distribution. This distribution of Python will install Python, Jupyter Notebook and all the applicable libraries all at once. To get the installer go to the listed URL, and select the installation of Python for your system. I strongly recommend you select the installer for the latest 3. x version of Python, unless you need an older version of Python for some reason. And if you do need to support multiple versions, check out creating Python version specific environments with conda package and environment manager. These environments can make supporting Python 2. 7 and 3, side by side, pretty painless. Next, we'll see Jupyter Notebook and Python in action.

Python and Jupyter Notebook Demo

Now let's see a quick demo of Jupyter Notebook and Python 3. 5. To launch Jupyter Notebook you need to start a new shell window. Do not close this window until you have completed your editing session, since the Jupyter web server will be running in this shell. Once the window comes up, we launch Jupyter Notebook by going to the command line and entering Jupyter Notebook. As you see, this launches a web server that will start in the current folder. You should also see a new browser window with the Jupyter file browser appearing. It will start from the folder that was current when you launched the notebook. Navigate to wherever you want to create your notebook. If you need to create a folder, select the New drop down, and select Folder. The folder will be created with the name Untitled Folder. You can select this folder and then Rename it. Once you get to the folder where you want to create your notebook, go to the New drop down and select Python 3 to create a new notebook associated with the Python 3 kernel. Congratulations, you just created a new notebook. Notice that the title of the notebook is Untitled. Let's change the title by clicking on it, and entering a new title in the dialog. Now let's review the interface which may look a little strange at first. We see a single cell displayed into which we can type code. What you type in depends on the cell type. By default, new cells are for entering code. Let's change this cell type to markdown, so that we can type in a title and description of the notebook. We can change the cell type to markdown by using the drop down box. Or by pressing Escape + M. In a similar manner, you can change the cell type to code by using the drop down or by pressing Escape + Y. As we see, markdown denotes a header by a line that starts with one of more # followed by a space. The level of the header is determined by the number of #s. A level one header is the most major, and is denoted by one #. A level two has two #s, et cetera. The levels are similar to the H1, H2, H3 you have seen in HTML. The second line is plain text except for Jupyter and Python 3 which have *s denoting emphasis. Like headers, if you put one *, you get level one of emphasis. Two *s causes level two emphasis, et cetera. But right now, the cell contains raw markdown text and it looks pretty ugly. That is because we have not process the markdown yet. We process it by pressing Shift + Enter anywhere in the box. And look at the difference. The markdown has been processed to produce nicely formatted text. There are a lot of markdown commands you can use. And if you do a web search, you'll find a number of great markdown tutorials. Now, let's try some code. Of course, our first code example has to be some variant of Hello World. Let's create a cell with the header for this Hello World example. Again, we need to ensure it is a markdown cell, and since it is a header for this example, we'll enter a level two header. What should I type? Correct. I should enter two #s and a space for a level two header. Now we enter some code below this title. Let's do a classic Hello World with a little twist. Here we will see string concatenation in action. The editor has completion, so if I type in my and hit Tab it looks up the variables and automatically completes the variable as my name. We are now ready to print our string, but I know the syntax for the print statement changed between versions two and three. And I'm not quite sure which syntax I should use. No problem. I type print and press Shift + Tab. It shows the documentation for the print statement. I see that print is now a function, so that I need to put my text and variables in between parentheses. But what is that end argument about? I press Shift + Tab twice and see the end argument lets us define a character to append, like new line, to the end of the line. Notice that there are two types of arguments used in Python. Value is a position based argument that is the first argument to the function. There are also keyword arguments such as end, for which the caller must specify the keyword followed by an =. Also notice that the keyword end has a default value of \n which is a new line character. If we wanted a blank line after our print statement, what character string would we pass? That's right. We would pass \n \n to get two new line characters after the text. One to end the current line, and one to create a blank line. Now that we know the parameters, let's complete the print statement. I type in H-e-l and hit TAB. We now see there are two items starting with h-e-l, the command help and the variable hello statement. I select hello statement and press Return. Then enter the parameter End and press Tab. To be correct syntax, an = is needed, so it's added automatically. Then we move to the next cell, and nothing happens. What is going on? Oh, silly me, I forgot to press Shift + Return to tell Jupyter Notebook to execute the contents of the cell. Once I do this, the contents of the cell are executed, and we see Hello Jerry, followed by a blank line. That's great. Let's try another example. Let's try a loop. First we add a description. Oops, all that red means I started typing text into a code cell. No problem, we can just change the cell type to markdown. And then press Shift + Enter to process a markdown. Much better. And then let's enter in a For statement. And below that we enter in the operations we want to perform within the loop. Notice that the operations in the loop are indented further than the for statement. This indentation is how Python knows that the code belongs in the loop. There are no curly braces or other characters that define the enclosing structure of code in Python. Indentation defines parentage and makes the code very easy to read. Also notice that by default, the print statement supports format strings where all occurrences of that curly brace zero, and curly brace one get replaced by the first and second arguments in the format function. Now let's run the loop by pressing shift return. Uh-oh, syntax error. Looks like I forgot to define x. I'll define it in the first code cell to make the second code cell look neater. Then we rerun the loop by entering Shift + Enter. What, still broken. What's going on? This shows one of the basic features of the Jupyter Notebook. Namely, only the code in the cell that is current is executed when you press Shift + Enter in that cell. When I entered x 10, I forgot to do this, so x is undefined. If I go back to the first cell, and press Shift + Enter it reruns the contents of that cell. Then I can go to the loop cell and everything works as expected. The need to press Shift + Enter in both cells is both good and bad. On the one hand it means that you can make a change to a cell, and only evaluate the cells where you explicitly enter Shift + Enter. On the other hand, it means that if you change something the cells below will not see the change until you rerun them. As you can imagine, this can be a frustrating bug to find, especially when you've made a number of change in different cells. To get around this, you can just rerun the entire notebook. Go to the Menu item Kernel, Restart & Run All option. This will rerun all the cells in your notebook from top to bottom. And now our loop works. Okay, now that the loop is running, let's take a look at the output. As you see, the loop now prints out the expected numbers, that is, those are the expected values if you're already a Python programmer. Developers coming from other languages would likely have expected the range to return one, two, three, four and five. Not just one, two, three, four. This is a feature of Python looping. In Python you specify the starting point and the value not to exceed, not the ending value. Well, surprisingly, in these few examples, we've gone over enough to get us started. So we can press the save button to save our work and everything is written to a file with the notebook name plus and extension of ipynb. So in our case, the file name is pythonfirststeps. ipynb. The ipynb extension comes from the old name for Jupyter Notebook, IPython Notebook. Now that you know now to use the notebook, we can head off to the next module, where we'll go over the machine learning workflow. See you there.

Understanding the Machine Learning Workflow

Machine Learning Workflow Overview

Building a machine learning solution is a complex set of tasks. Over the years, a machine learning workflow has been developed to organize these tasks in an efficient manner and ensure that no critical steps are missed. I am Jerry Kurata and welcome back to the Pluralsight course on Understanding Machine Learning with Python. In this module, you'll get an overview of various tasks that comprise the machine learning workflow and tips on how to best use the workflow. But first, let's define the machine learning workflow. While there are numerous definitions to the machine learning workflow, I like this definition. As you can see, the machine learning workflow is orchestrated and repeatable. This means work has been purposely organized into defined tasks, and the same workflow can be used to develop solutions to different problems. Transforms and processes information. That is, before we can use data in creating solutions, it must be transformed into a format we can use for processing. And, finally, the product of the workflow is a prediction solution. Implicit in this is the prediction solution must meet the performance criteria we specify. With this definition in mind, let's take a look at the steps in the machine learning workflow. Our first step is to define the question we want the solution to answer. We need to define this question in a way that guides the remaining steps in the correct direction. This requires thinking carefully about the goals we want to achieve, the data we need, and the processing we can perform. Once the question is defined, we can gather the data we need to answer the question. This can be tricky, because data are often incomplete, inaccurate, and conflicting. When we have the question defined, and the data we need, we can consider which algorithm to use. This is not an easy task, as there are many algorithms available. But if we have properly defined the question, the question will help us select the proper algorithm. Once we have the algorithm selected, we need to use a subset of the data we have to train the algorithm. This training process will result in creating a trained model that predicts results on similar data. Once we have a trained model, we need to test its accuracy on new data that was not used to train the model. This testing will generate statistics with which we can determine if the model will meet our requirements or needs to be refined. If refinement is needed, it may be necessary to go back into the workflow and alter or get more data, change algorithms, adjust training parameters, and sometimes some combination of all three. When using the machine learning workflow, there are some important things to keep in mind. First, there is an inherent hierarchy of these steps, with the earliest steps being the most important, since the latter steps are dependent upon them. That is, you need to correctly define the question for which you are creating a solution. Then you have to get correct data, which will allow you to train your algorithm to come up with a prediction. Only when you have the model trained can you evaluate its accuracy. When moving through the workflow, it's not unusual to have to return to a previous step. For example, as you work with data, it may be apparent that you are asking the question incorrectly. And regarding data, data that you find will almost never be in the format that you need. Expect to spend a considerable amount of time locating and transforming the data into the structure that you can use. Also, within reason, more data is usually better. Remember the mathematical equations needed to model your data may be complex with strange quirks. The more corner cases you can cover with the data, the better the model will be trained, and the more accurate your resulting predictions will be. Finally, try not to push a bad solution. It's easy to fall into the trap of thinking, with just a few a more tweaks the stars will align and your model will start performing correctly. If you find yourself in this situation, it's better to take a step back and ask yourself, "Do I have the right data? "Do I need to pre-process data? "Or do I even have enough information to continue? " Now that we have seen the structure of the machine learning workflow, let's see how we can apply it and use the workflow to build a solution that will predict whether a person will develop diabetes.

Asking the Right Question

From Question to Solution Statement

[Autogenerated] as with any software project, having a clear definition of what we want the machine learning solution to achieve is critical. I am Jared Kurata and welcome back to the portal site course and understanding machine. Learning with Python in this module will go through the first step in the machine learning workflow. Asking the right question will convert our general goal of predicting of her person will develop diabetes into a solution statement that could be used to guide our work. At first glance, it might seem that asking the right question is easy. After all, don't we already know that we want to predict if a person will develop diabetes? But this is a case of appearances being deceiving? We need a question that will direct invalidate our work. That is the question which drive the direction of the data we gather, how we mold the data, how we interpret the solution and the criteria we used to validate solution. What we need is more than a simple, one lying question. Rather, we need a solution statement to finding the in goal starting point and how we achieve the end goal. Let's state the goals for our solution statement. The final solution statement should define the scope of the solution, including data sources. Defined darker performance measurements for the solution. Determine the context for using the solution and find how the solution will be created. Let's start adding these requirements and see how they impact the solution statement. We will use our original goal of predicting if a person will develop diabetes as the starting point based on the needs of the solution. We can see if there are some assumptions that can help with the definition of the scope of the solution. A review of the data on the American Diabetes website shows the diabetes has several factors that could help guide us thes our age. Older people are more likely to get diabetes and race certain _______ groups, including African Americans, American Indians, nation Americans are more likely to have diabetes. Interestingly, gender does not seem to be a factor of these factors. Race stands out of something we may be able to use as a selector. Specifically, we can see if we can select data from one or more of the higher diabetes risk _______ groups. With the race feature, we can do a preliminary evaluation of data sources. There are several sources of data on diabetes, but one stands out. Is the best data source for our purposes. That is the Pieman Indian Diabetes Study available at the UC I machine Learning Repositories. This is a study of diabetes and women in the Pieman Indian population. Located in the Phoenix, Arizona area. The study was conducted in the nineties in as a classic data set. When we incorporate the use of the Pema Indian data set. That statement changes to using Pema Indian diabetes data predict which people will develop diabetes. One important aspect in any prediction. Algorithms how accurate it is. So for our prediction, what prediction Accuracy should we reasonably expect? We're defining the evaluation in a binary fashion, diabetes or not diabetes, and we want to be a Zachary. It as is reasonable. Historically, disease predictions are notoriously bad. This is due to many factors, but the biggest is that we're own genetically different, even people that are genetically identical at birth. Identical twins experience different factors during their lives. Such a CE lifestyle, environmental exposure and sickness that lead to different health results. Against this backdrop, 70% accuracy is a reasonable goal. So let's use that as the bottom of the acceptable range. This changes the statement to using Pema Indian data predict, with 70% of greater accuracy which people will develop diabetes. The context in which the prediction will be used needs to be considered. Since this contrived decisions on data to capture and mold, for example, we're predicting disease. What does it mean to predict disease doesn't mean we're absolutely sure about the prediction. What are the common practices in the field of medical research In medical research? We see that there are many variables in each individual and many of these air unknown with respect to how they affect the probability of developing the disease as we discussed earlier, even identical twins with the exact same DNA show variance. So in medical research, what is predict is the likelihood of developing and disease. Let's incorporate this likelihood estimate into the statement statement is now using Pema. Indian data predict, with 70% of greater accuracy, which people are likely to develop diabetes. It is always useful to put down on paper how you're going to approach the problem with machine learning. I like to put down the general steps for our solution. These are used the workflow process. The Pima Indian diabetes data transformed data as required. This changes our solution statement quite a bit. The statement is now used the machine learning workflow to process and transform Pema Indian data to create a prediction model. This model must predict which people are likely to develop diabetes with 70% or greater accuracy. We now have the final solution statement. The statement defines the data We will use the performance expected and even how we will go about creating the solution joined me in the next module where we will see how we can use the machine learning workflow to implement the solution we have to find.

Preparing Your Data

Introduction to Data Preparation

I am Jerry Kurata, and welcome back to the Pluralsight course on Understanding Machine Learning with Python. In the previous module, we worked through the first step of the Machine Learning Workflow, asking the right question. In that module, we developed a solution statement for our prediction. In this module, we will use that solution statement to guide us in getting and preparing data. As you know, machine learning is based on data, so it should come as no surprise that getting and preparing data is a major step in the machine learning workflow. And just like preparing a surface before painting, good data preparation can make the task of creating a well-trained model much easier. Likewise, poor preparation can make the task painful and full of corrective and redo work. With this in mind, let's take a look at what we will do in this module to get and prepare our data. We will start by discussing the data we need. Once we have the data, we need to load it into Python and inspect and clean the data. We also explore the data to understand its structure and make any necessary alterations. Finally, we mold the data into Tidy data, which works best with machine learning algorithms. We will do all this work in Python, running inside Jupyter Notebook. But right now, you may be asking yourself, "What is this Tidy data that we are trying to produce? " Here is the definition of Tidy Data. If you have any SQL experience, as you read through this definition, it may strike you as very familiar. In fact, Tidy Data has attributes of what we try to achieve in database normalization. We want each variable to be a column, each row a unique observation, and we store them in a table which is in code's third normal format. But Tidy Data is a bit more pragmatic than normal forms, and focuses on presenting data in a clean, readable format, versus underlying structures like keys. I like to think the data is Tidy when it looks like it was produced by a well-designed view, and getting our data to a Tidy form can take a little time. In the literature, you'll find estimates of 50 to 80 percent of the time in a machine learning project is spent getting, cleaning, and organizing data. At first, I didn't believe this, but experience has taught me that these numbers are accurate. Why is this?

Getting Data

Today, we are swimming in seas of data, but even with all that data, the data we need may be difficult to find, so where can we look? Google, obviously, but be careful. Not surprisingly, quality varies wildly. Be extra careful when looking for data on controversial topics like global warming, vaccine side effects, etc. Fake, invalidated data abounds online. Government sources are often good. There's usually lots of data, and it's peer reviewed. And the data comes with good documentation on the meaning of the data, that is metadata. Professional groups and companies can be a good source of data, also. Professional societies provide data sets. Twitter provides access to tweets, and lots of analysis has been done on tweets to determine the tenor of them. Financial data can be retrieved from free APIs from companies like Yahoo. If you work in a company, your own IT department could be a good source of company specific data. Similarly, your department likely maintains department specific data. And sometimes, it's just all of the above. A lot of the time, you need data from many sources, and you have to weave them together into Tidy data. With these sources in mind, let's see where we can get the data we need to solve our problem of predicting diabetes. When it comes to getting data, we're lucky. The large UCI Machine Learning Repository has a version of the data we need in its Pima Indian Diabetes Data set. However, the Pima Indian Diabetes Data from the repository is too perfect. It does not have many of the issues you will find with the data you encounter in real world situations. Because of this, we will use a slightly modified version of the data from the repository that is provided in the demo file pima-data. csv. The data in the csv file is from female patients at least 21 years old. There are 768 patient observation rows and each row contains ten columns. Nine of these columns are feature columns, such as number of pregnancies, glucose concentration, and insulin levels. One column contains the class, diabetes, which has the value of either true or false. Using this data, we will be able to predict how likely it is that a person will develop diabetes. The data in the csv is not perfect, and as such, we will illustrate Data Rule #1: The Closer the data is to what you are predicting, the better. This may sound obvious, but it's true. Having a single field saying this patient has diabetes, or not, is great. In a lot of cases, you don't have what you are trying to predict so clearly defined in the source data. Rather, you have to infer it by combining columns from multiple sources. And each combination brings up a potential area of misinterpretation or otherwise, coming to the wrong conclusion. But no matter how good the data appears at first glance, it will almost never be quite what we need, or better stated, Data Rule #2: Data will never be in the format you need. If you have done any data processing, you know how true this is. You will always have to do something to the data to reformat it. In Python, we will utilize Pandas DataFrames to make this reformatting easier. Pandas DataFrames appear as a table with a column for each feature in the class, and a row containing the observation of these features in the class for each patient.

The GitHub Repository

As previously mentioned, the demos in this course utilize a modified version of the Pima Indian Data. To get you started, I have created a GitHub repository with this modified version of the Pima Indian csv data file, and the notebooks used in this course. You can find the repository at this URL. When you get to the GitHub repository, you should see a screen similar to this one. If you're familiar with GitHub, you can clone the repository using this URL. If you're not familiar with GitHub, you can get the repository by pressing this button to download the ZIP version of the repository. All of the notebooks are in the notebooks folder. Also in the notebooks folder is a sub folder named Data that contains the file pima-data. csv, which is the modified version of the Pima Indian Diabetes Data.

Loading, Cleaning, and Inspecting Data

In this demo, we will go over loading our version of the Pima Indian Diabetes Data, exploring that data to see what sort of information it provides, and cleaning the data of extraneous fields that might cause us difficulties in utilizing the data. Prior to beginning this demo, I placed the csv file containing our version of the Pima Indian Data into a folder named Data. Let's go into Jupyter Notebook and start looking at our version of the Pima Indian Diabetes Data. At the command prompt, we enter the command Jupyter Notebook to launch the notebook server. We create a new notebook by selecting the Python3 kernel, and we change the name of the notebook to something useful; Pima Prediction. We change the first cell to a mark down cell, and enter the top level header for the notebook, and then we add a header for the import statements. We need to import some basic libraries. Pandas provides dataframes, Matplotlib provides plotting support, and NumPy provides scientific computing with N-dimensional object support. Finally, we add this Jupyter Notebook magic function that causes plots to be displayed inline instead of in a separate window. We are now ready to load our Pima Data into the Pandas DataFrame. We use the Pandas reads csv function. By default, this function assumes that the first row of the csv file contains the column headers. Let's check the structure of the data by using the shape function. The shape function displays data in the format number of rows, comma, number of columns. So, we have 768 rows with ten columns. Let's inspect the newly loaded data. We use the head method to see the beginning rows of the dataframe. Passing five is the argument tells the method how many lines to return. Likewise, we use the tail function to see the end of the dataframe. Notice that each row has a zero based index. Also notice that the Pandas DataFrame has column names. We use these names to access specific columns. We can also use these column names with a row indice to access the value in a specific column in a specific row. These columns represent logical features in the results in the data. So often, you hear the term feature and column used interchangeably, but they are different. A feature is something that's used to determine a result. A column is a physical structure that stores the value of a feature or a result. We will need to refer to the definition of the columns from time to time. To ensure the definitions are available, let's add a feature definition table to our notebook. In general, we want to eliminate any column that we do not need. In particular, we want to eliminate columns that don't have any values, or are full of empty values, but we also want to eliminate columns that are duplicates, or provide the same information in a different format. We can visually inspect columns to see if they really are the same, but visual inspection is error prone, and does not deal with the critical issue or correlated columns. Correlated columns state the same information in a different way, such as an ID in the text value for the ID. These correlated columns do not add any information about how the data causes changes in the result. And worse, they can amplify a bias because some algorithms naively treat every column as being independent and just as important. For example, imagine what would happen if we were predicting house prices, and had the size of the house in both square feet and square meters. Size could result in being twice as important as it should. Pandas makes it easy to see if there are any null values in the dataframe. The isnull method will check each value in the dataframe for null values, and then the any method will return true if any nulls were found. Great, looks like we don't have any nulls. But how about those correlated values? To make it easier to visualize correlations, let's use the matplot library and create a function that cross-plots features so that we can see when they are correlated. The colors in this cross-plot run from blue, meaning not very well correlated, to dark red, meaning very positively correlated. Now, let's invoke the correlation cross-plot. Notice the column names running across the horizontal and vertical axes. This is a matrix showing which columns have data values that are correlated with values in other columns. Dark red means the columns are highly positively correlated. Since we have the same columns along the horizontal and vertical axes, we expect red squares on a diagonal from the upper left to the lower right, since this is comparing a feature with itself. But what about the red squares for skin and thickness? This means a strong correlation between these two columns. Let's check the correlation numbers to verify the result shown on the graph, and see just how correlated the skin and thickness columns are. Notice the 1. 00 running down the diagonal of the table. Again, this is because we are seeing the correlation of each column with itself, which is, of course, a perfect correlation. But also notice the 1. 0 correlation between the thickness column and skin row, and the skin column and the thickness row. This means that the columns are changing in value in lockstep, in an exact proportions to one another. Having two features that move together adds no additional information. We can either make note of this and make sure we try to avoid using both columns at the same time, or we can drop one of them. I am going to drop the skin column. By the way, the skin column was created by converting the thickness, which was in millimeters, to inches. Let's look at the head of the dataframe again. Look at this data, and tell me if you can see the one to one correlation between thickness and skin. Maybe, but it's pretty easy to miss something like this, and one reason we use features like Pandas core function to make the relationship obvious. And plotting the correlation in color makes the relationship hard to ignore. Now that we found the extraneous correlated columns, let's use the del command to delete the skin column. We can verify that the column was dropped by checking the head again. The skin column is now gone. Let's take another look at the plot to verify that there are no other correlated columns. The correlations look good. There appear to be no other correlated columns. Now we can move on to molding the data.

Molding Data

We have done some good clean up by removing some columns we did not need. Now, we need to mold the data into data that we can use for training. This involved ensuring that the data types of the columns are what we need, and if required, creating new columns based on existing data. Let's now mold the data. Let's start by inspecting the data to see if there are any issues. Algorithms are largely mathematical models. As such, they work best with numeric quantities, and in fact, some algorithms require that all the features and classification results be numerical. We still have categorical results, Boolean true and false, for the diabetes feature. We need to change these Booleans to integers to ensure they'll work with all algorithms. Specifically, we need to change true to one, and false to zero. We do this with the Pandas DataFrame map method. We first need to define a mapping dictionary that defines true as one and false as zero. Then, we use the map method to change the values from true and false, to one and zero. Note that the Pandas DataFrame did all the work of iterating through the data. We didn't have to write any sort of loop iteration code. Pandas makes dealing with huge tables of data very simple and fast. Now, let's check the dataframe again with the head function. And we see now that the diabetes class is all one and zero. Reviewing the data, it looks like we have it in a form that we can use. We are just about ready to use this data to train the algorithm. However, before we use this data for training or prediction, we need to ensure the distribution of the data will allow us to train or prediction. This is driven by Data Rule #3: Accurately predicting rare events is difficult. The basic issue with a rare event is it's rare. So, the probability of you having them in your data is low, and the probability of accurately training the algorithm is also low. We only need a few percentage of the event to be able to train, but more is better. To ensure that we have a reasonable chance of correctly predicting if a person is likely to develop diabetes, let's check our newly cleaned data. Let's check the number of true and false diabetes cases. Pandas will let us easily search the columns by name, and see how many times diabetes is true, one, and how many times it is false, zero. We can then calculate the percentage of the time the diabetes class is true, and the percentage of the time the diabetes class is false. We have about 35% diabetes, which means we can reasonably train our prediction model using the data we have. If the vast majority of the data were either true or false, the standard learning techniques we will apply might not work very well. In cases where we were trying to predict something that did not happen very often, we need to use special techniques, which are beyond the scope of this course. But with 35% true, and 65%

false, standard prediction techniques can be used. So, congratulations, we have our training data. But does anybody remember what we did to get here? This brings up Data Rule #4: Track how you manipulate data. A lot of manipulation of data that you do in Machine Learning is trial and error. When you manipulate the data, it's very easy to change the meaning of the data. This can be done intentionally, but it can also be done unintentionally. By keeping track of the manipulations of the data, you can always reproduce your data any step, and see where you went wrong. Since we are using Jupyter Notebook, keeping track of our changes is done for us automatically. We have the best of both worlds. We have the interactivity of the Python interpreter, with tracking via code cells, plus markup cells where we can enter detailed code descriptions and links to reference sites. Additionally, like all coding, we need to ensure that we use some sort of source control to track changes and allow us to restore things that we might accidentally delete or alter. Let's summarize how we prepared our data. First, we loaded the diabetes data from our csv file into a Pandas DataFrame. We used a function to visually identify correlated features. From there, we cleaned the data by removing the extraneous column of the correlated features. Then, we molded the data by changing the data types of columns into what we needed. We also verified that we had enough true and false values to use the data for prediction. And along the way, we discussed data rules that guided our work. Join me in the next module, where we will select the algorithm that we will train with our data.

Selecting Your Algorithm

The Role of the Algorithm

I'm Jerry Kurata, and welcome back to the Pluralsight course in Understanding Machine Learning with Python. In the previous modules, we went through the steps of asking the right question and preparing our data. In this module, we'll go through the process of selecting algorithms. We will use our problem knowledge to help us decide the algorithm to use. More specifically, in this module we will discuss the role of the algorithm in the machine learning process, select our initial algorithm by utilizing the requirements identified in the solution statement as a guide, and discuss at a high level the characteristics of some specific algorithms. Finally, we will select one algorithm to be our initial algorithm. Notice that I said "initial" algorithm. As mentioned previously, in machine learning we often cycle through the workflow. In our search to find the best solution, it is likely we will need to train and evaluate multiple algorithms. It is important that we understand the role of the algorithm in the machine learning process. So let's review how the algorithm is involved in the process. One could say that the algorithm is the engine that drives the entire process. And we will see this illustrated as we walk through the process of using the algorithm to create and use a trained model. For our prediction problem, we will use training data containing examples of the results we are trying to predict and the feature values we are using to predict that result. When the training function, often named "fit" in scikit-kearn, is called, the algorithm executes its logic and processes the training data. Using the algorithm's logic, the data is analyzed. This analysis evaluates the data with respect to a mathematical model and logic associated with the algorithm. The algorithm uses the results of this analysis to adjust internal parameters to produce a model that has been trained to best fit the features in the training data and produce the associated class result. This best fit is defined by evaluating a function specific to the particular algorithm. The fit parameters are stored, and the model is now said to be trained. Later, the trained model is called the prediction function, often named "predict" in scikit-learn. When this prediction function is called, real data is passed to the trained model. Using only the features in the data, the trained model uses its code and parameter values set during training to evaluate the data's features and predict the class result, diabetes or not diabetes, for this new data. Now that we understand the role of the algorithm, let's select our initial algorithm.

Narrowing the Selection

Since machine learning can be applied to a lot of problems, there are a lot of machine learning algorithms available. A quick check of the scikit-learn library shows that there are over 50, and more being created. So how do we decide which algorithm to use? To decide which algorithms would be suitable, we can compare the algorithms on a number of factors. Not surprisingly, data scientists have different opinions on what factors are important in selecting algorithms. The decision factors presented here represent one possible set of factors. As you gain experience, you will likely develop your own set of factors and decision criteria. For our example, we are going to use these decision factors to select our initial algorithm: what learning type they support, the result type the algorithm predicts, the complexity of the algorithm, and whether the algorithm is basic or enhanced. We will use our solution statement and knowledge of the workflow to help guide us in the evaluation of these factors. Each algorithm has a set of problems with which it works best. One way to divide them is to look at the type of learning they support. With that in mind, let's go back to the solution statement and see what guidance it offers. Reading the statement, we see that our solution is about prediction. Prediction means supervised machine learning, so we can eliminate all algorithms that do not support supervised machine learning. That reduces the number of choices, but we still have a lot. Let's see what else we can do. Let's see how the result type can help. Prediction results can be divided into two categories, regression and classification. Regression means a continuous set of values. Back in our example of determining the price of the house, we used the features of the house, size, number of bedrooms, et cetera. We put these features into an equation and produced a price. Any change in any feature resulted in a direct change in the price. In contrast, classification problems have a discrete set of values, such as: small, medium, or large; one to 100, 101 to 200, 201 to 300; or true and false. Changes in the feature values may or may not change the classification. So what type of problem do we have? Again, the solution statement comes to our aid. From the solution statement, we see that the algorithm must predict which people are likely to develop diabetes. Remember, diabetes is true or false, and therefore diabetes is a binary outcome. Since we are predicting a binary outcome, diabetes or not, we can eliminate any algorithms that do not support classification in general and binary classification in particular. That disqualified some more algorithms, but perhaps not as many as you might have thought. This is because many algorithms support both regression and classification. We are down to 20 algorithms, but that is still too many. Since this is our initial algorithm, let's keep it simple. Let's also eliminate algorithms that are ensemble algorithms. These are special algorithms that combine multiple algorithms under one interface. These algorithms are more often used when we need to tune a model to increase performance. And we are still trying to do our initial training. Also, ensemble algorithms tend to be complex and difficult to diagnose and troubleshoot when the results start going awry. We still have a field of algorithms left, and we can divide these into two groups, basic and enhanced. Enhanced algorithms are a variation on basic algorithms. They have been enhanced to perform better and/or add additional functionality. As a result, they are more complex to understand and use properly. Since this is our initial algorithm, let's stick to the basic algorithms. These have the advantage of being less complex, and therefore easier to understand.

Selecting Our Initial Algorithm

For our problem, let's look at three basic algorithms as possibilities for use in our initial training and evaluation. We will look at Naive Bayes, Logistics Regression, and Decision Tree. Each of these algorithms are classic machine learning algorithms, and understanding a little bit about each one can greatly help in understanding more complex algorithms that use these algorithms as building blocks. The Naive Bayes algorithm is based on Bayes' theorem. This theorem calculates a probability of a diabetes by looking at the likelihood of diabetes based on previous data combined with the probability of diabetes on nearby feature values. In other words, so how often does the person having high blood pressure correlate to diabetes? It makes the naive assumption that all of the features we pass in our independent of each other and equally impact the result. For our data, this means that blood pressure is as important as BMI, which is also as important as the number of pregnancies. This assumption that every feature is independent to the others allows for fast conversions and therefore requires a small amount of data to train. The Logistics Regression algorithm has a somewhat confusing name. In statistics, regression often implies continuous values. But Logistics Regression returns a binary result. The algorithm measures the relationship of each feature and weights them based on their impact on the result. The resultant value is mapped against a curve with two values, one and zero, which is equivalent to diabetes or no diabetes in our case. The Decision Tree algorithm can be nicely visualized. The algorithm uses a binary tree structure with each node making a decision based upon the values of the feature. At each node, the feature value causes us to go down one path or another. A lot of data may be required to find the value which defines taking one path or another. As we see, decision trees have the advantage of having tools available to produce a picture of the tree. This makes it easy to follow along and visualize how the trained model works. From these three, let's select Naive Bayes as the initial algorithm for training and evaluation. Naive Bayes has a number of characteristics that make it useful as an initial algorithm. First, it is simple to understand. It uses how often a feature is associated with a result to determine if the value is diabetes or not diabetes. Second, the algorithm is fast. Training time of some of the more complex algorithms can run 100 times or more longer. In a machine learning, you often have repeated cycles of running the algorithm, evaluating the result, and tweaking the data. Having a fast algorithm makes these cycles much quicker. Finally, the algorithm is stable. With some algorithms, as you work with the data, small changes in training parameters cause the results to vary widely. Or worse, the algorithm fails and returns nonsensical results. Both of these can send you on a long and frustrating debugging cycle. Naive Bayes does not exhibit these behaviors. So let's summarize our algorithm selection process. First, we need to acknowledge that there are a lot of algorithms available and more being created every day. With all these algorithms available, we need to select one we will initially train with our data into a model. To select this algorithm, we can use the features of our problem that we captured when we defined our solution statement. From the solution statement, we know we have a supervised machine learning problem, which removes all of the unsupervised machine learning algorithms. We wanted binary results, not continuous values, which further reduces the number of viable algorithms. Since this is initial algorithm training and evaluation, we also eliminated all ensemble algorithms that combine multiple algorithms. We also decided that for initial training, we only wanted basic versions of the algorithms. After evaluating the basic algorithms, we selected Naive Bayes as our initial algorithm, because it is fast, simple, and stable. In the next sections, we will train this algorithm and see how well it can predict if a person is likely to develop diabetes.

Training the Model

Introduction to Training

Hi, I'm Jerry Kurata. Welcome back to the Pluralsight course, Understanding Machine Learning with Python. In the previous modules, we covered the workflow steps of asking the right question, where we defined our solution statement, preparing data in which we obtained raw data and transformed it into the data we will use for training, and selecting the algorithm, where we selected the initial algorithm we will train and evaluate. In this module, we'll put the pieces together and train the algorithm we selected with the data we prepared. When we are done with this training process, we will have a model that can predict if a person is likely to develop diabetes. In this module, we will get a detailed understanding of the training process, introduce the Scikit-learn library, which can make the training and evaluation process much easier. Then, we will go back to Python and train our algorithm with our diabetes data and produce a trained model. A good definition of Machine Learning training is letting specific data teach a Machine Learning algorithm to create a specific prediction model. Notice the use of the term specific. Data drives the training, and if the data changes over time or new data is used, in man case, we need to retrain. And we want to retrain if the data changes. Retraining will ensure that our model can take advantage of the new data to make better predictions. And also verify the algorithm can still create a high-performance model with the new data. Now let's dive deeper into Machine Learning training process.

The Training Process

Let's review the supervised Machine Learning training tasks we need to perform. First, we split the prepared diabetes data into two data sets. One for training the model, and one for testing the trained model. About 70% of the data goes into the training set, and about 30% goes into the testing set. Then, we train the algorithm with the training data and hold the test data aside for evaluation. This training process produces a train model based on the logic in the algorithm and the values of the features in the training data. But what does this training process look like? To make this more concrete, let's take a look at a hypothetical training session of some hypothetical disease data. Note, this is just an example. I am not using our diabetes data in these plots. We are trying to understand if a person is likely to develop a disease based on a set of health test results. In Machine Learning, these test results are the features. Let's plot two of these features, Feature X and Feature Y, and the disease results from our hypothetical test data. We show not disease as blue circles and disease as red X's. When we train, we expose code in the algorithm to data with which it interacts. Let's create some model whose parameters are set by the feature values to define the boundary between the disease and not disease results. We call this the decision boundary. The key point is the structure of this boundary is defined by the combination of the algorithm and the training data which adjusts the model's parameters. As you can imagine, this relationship can get quite complex. Also notice that the training boundary is not perfect. That is, we see some blue circles in the red x side and vice versa. We will talk about this later, but for now, know that perfect accuracy against training data is not the goal. The goal is accuracy against real world data. So, now we understand what we use our training data for. But I bet you're thinking, what about the test data? Why are we only using the training data to train? What are we going to do with the test data? You are correct to question this. Any thoughts on why we split the data and then only used only one of the sets of data in the training? Well, if we used all of the data to train, the produced model would be trained with both training and test data sets. Since data drives training of the model, this additional data would impact the workings of the model. Also, when we evaluated the model with this test data, it would likely falsely perform well since the model has seen the data before and knows about the test data's biases. But the model could perform poorly when we use real world data that has a different set of biases. To prevent us from being lulled into thinking the model performs better than it really does, we need some real world data to test against. We use the previously unseen test data as the stand in for real world data. And since our data comes from the real world data, using the test data, we are testing against a set of real world data. We are just about ready to train our algorithm and create the trained model. But before we get started training, we need to select the features we will train with. Since data drives how the model is trained, we want to ensure that we only train with the minimum number of features. This makes the training go faster and often more accurately. We can look at the data and reason about the context of the problem to make our selection. When I reviewed the features, it looks like now that we have tidied up the data, we can use all of the fields we have remaining.

Python Training Tools

Now that we understand in detail how to do the training, we're just about ready to code. But before we start coding, let me ask you a question. Do you usually write all your code from scratch? I bet not. I bet most of the time, you're like me and will try to find a library that can do some of the repetitive tasks for you. In Python, there's such a library for handling Machine Learning training and evaluation tasks. The library is called Scikit-learn. Scikit-learn provides a set of simple and efficient tools that can handle many of the tasks we routinely perform in Machine Learning. Let's take a look at the Scikit-learn library and see how it can lessen our workload. As the name Scikit-learn implies, Scikit-learn's purpose is to support Machine Learning. Scikit-learn is built on oft-used Python libraries such as, NumPy, SciPy and Matplotlib, and supports these and Pandas' dataframes. The Scikit-learn library is a tool set that makes it easier to perform training and evaluation tasks, such as splitting the data into training and test sets, pre-processing data before we train with it, selecting which features of the data are most important, creating our trained models, tuning the model for better performance, and another of other common tasks. In addition to these features, the library provides a common interface for accessing the algorithms. And now that we know about Scikit-learn, let's see it in action as we split our data and train our model.

Splitting Data and Training the Algorithm

[Autogenerated] Let's go into Jupiter Notebook where we will split our data into training and test data sets, perform any required post split data preparation and train our initial algorithm with our training data. Now that we have our data in a form we can use, we can split the data into training and test sets. We will train our algorithm with the training set, and we will use the test set to verify the quality of the training we will never use. The test set for training to do so would train the model to the features in the test set and invalidate any testing. Perform with this test data Psych it learn contains the train test split method, which makes it easy to split the data. We use this method to split our data, with 70% being put in the training set and 30% into the test set. We start by importing the train test split method next week to find lists of the future columns in the predicted column. Diabetes. We split our prepared data into two data frames, one containing the future columns and the other with the predicted diabetes result for its observation row these data frames in the test size 0.30 for 30% are passed to the train test. Split function trained tests split splits the original data frames in returns. Four. Numb pyre. Rays of data. The rays contained the values of the test and training feature columns and the test and training Diabetes result. A minor but important feature is that we set random state to 42. That sets the seed for the random number generator used as part of the splitting process, setting the seed to a constant insurers that if we run the function again, the split will be identical. Any number can be used for random State. Once the function is run, we have our data split into training and test sets. By the way, we pre fixed variable names with X and Y since in algebra it is common practice to say why equals F of X, which means that the result is a function of the features we need to ensure that we have the desired 70 30% split. These numbers are very close to the 70 30% split of training to test data we desired, however well, the number of rose and the training and test data sets look correct. This is insufficient to ensure that we have the data split correctly. We also need to ensure that the ratio of true and false cases of diabetes in the test in training data sets is the same. We can verify the percentage of true versus false on the training and test data sets match those in the original data. We do this by checking the percentage of rose with diabetes equal zero versus the percentage of rose with diabetes equal one. As we see, the percentage arose with diabetes equals zero and the percentage of rows of diabetes equals one was preserved when we split the data. Therefore, we can move forward with the test in training data sets. Once we have the data split into training and test sets, we may need to do some additional data preparation, which includes some data transformations. However, unlike our early data preparation steps, we will separately apply the transformations to the training and test data sets. This will ensure that the separation between training and test data is maintained. Previously, we checked the data for all values by using the is no function and did not find any and all values. But is this really true? Sometimes no values can be hiding in plain sight. Let's take another look at the head of our data frame. Notice. The thickness is zero in the third row. Having a skin thickness of zero is not physically possible. This is a hidden missing value, likely somewhere along the way zero was entered for the missing values. Do we have other hidden missing values? Let's check the other columns that might have zero values to indicate missing data. While we have a lot of calls with zero values, which Rose are really an issue in which could validly have zero. This is where little domain expertise can come in handy. Unfortunately, I'm not a doctor or medical researcher, so we will have to make do with the next best thing. The Internet. Doing a few Google searches reveals that of these fields with zero on Lee won, insulin might have a valid zero value, and even in this cereal is likely not correct. So now the question is, what do we do about this missing data? Missing data is a common problem, and we have a few options and how to deal with them. We could ignore the missing values that is. Do nothing. Delete all the rows with missing values or replace the missing values with another value. Let's look at the numbers for some guidance. We have 768 rows of these 374 missing insulin readings. Can we ignore the fact that all the state is missing values? Maybe. But although zero values will cause a bias, how about the leading? The rose was zero values that might be okay for just a few rows, but we probably don't want to delete half our data. Therefore, we need to look at replacing those values with something else. This is called Imputing. Computing is a common practice for handling missing data. With imputing, we have a couple of options. We could replace the cereal fields with their mean median or other statistic for the column. Another, perhaps better option is to replace the value with the new value derived from the remaining features in the row, as determined by an expert medical researcher again. Since I'm not a doctor or an expert medical researcher, I don't know how to derive reasonable numbers for each of the missing features based on the values of the other features. So we'll go with another method. Using the mean of the column. Using the mean of the column is a responsible tact, since it reinforces what data we already have. Psychic learn contains an impudent class that makes it very eased, impute data and the in pewter class already sports mean based imputation. Let's import Thean pewter class. Here we create a new in pewter object and specify that it should replace the missing values. Zero with the mean for all values on the Axis zero, which is column. We used the imputed fit transform function to create a new numb pyre. A with any future value of zero replaced by the mean for the column. We do this for both the training and test feature value. Note that we must compute this mean separately for training and test data, since the values of for the features and therefore the means for each data set are different now that we have dealt with zeroes in the data, we're ready to train our initial algorithm naive bays before we train with the naive bays algorithm. We need to import it in the case that now you've based their multiple implementation algorithms. We're using the galaxy, an algorithm that assumes that the feature date is distributed, a galaxy in which looks like the classic bell curve, with most of the data near the mean. Then we call the fit method too great a model trained with the training data, and that's it. The Naive Bays model has been created and trained. Let's briefly summarize what we did in this module. We started with an overview of the training process. This provided us with an understanding of the task we needed to do to split the date into training and test data sets and create a trained model. We then went in to buy Thon and split our data and trained our model to help us with some tests. We use the psych it and learn library. This library comes with functions that make it easy to split the data into training and test data sets and then train the model. We also discovered that we had a lot of missing data. We discussed this and decided to use mean imputation to replace the missing values. Once we dealt with missing data we created in trained the Naive Bays model with our training data. Join me in the next module where we will evaluate the performance of this newly trained model.

Testing Your Model's Accuracy

Introduction to Evaluating the Model

Hello, I'm Jerry Kurata. Welcome back to the Pluralsight course in Understanding Machine Learning with Python. In the previous modules, we went through the workflow steps of defining our solution statement, getting the data, and selecting an initial algorithm. In the last module, we trained our initial algorithm with our training data and produced a trained Naive Bayes model. In this module, we will evaluate this trained model and see how well it can predict the likelihood of a person developing diabetes. We will evaluate our trained model by using a set of test data. Remember this data was not used to train the model, so it should give us an accurate evaluation of the real-world performance of our model. This evaluation will provide us with a series of results that we can use to decide if the performance of the model is acceptable. The results will also give us some ideas on how we might revise the workflow steps to improve the performance. Throughout our evaluation process, we need to keep in mind that statistics only provide us data. We are the ones that interpret this data and decide if it is good or bad, and we need to define good or bad in the context of how we will use our model. But enough theory for now. Let's go back to Python and evaluate the model.

Evaluating the Naive Bayes Model

Now that we have a training model, let's see how well it can predict values. We do this by passing feature data to the model's predict function. The predict function will return an array of 1 and 0s representing true and false. Let's first predict against the training data. X_train is the training feature data we use to train the model. To see the accuracy, we load the scikit-learn metrics library. Metrics has methods that let us get the statistics on the model's predictive performance. We are getting over 75% accuracy against the training data. Now let's see how our Naive Bayes model performs on the testing data. The accuracy looks good. We have greater than 70% accuracy. Wasn't that our goal? Let's take a look at the numbers in more detail before we declare victory. We can get more details and exactly how this accuracy number was derived if we look at the interestingly named confusion matrix. We'll also take a look at the classification report, which calculates some other performance statistics. The confusion matrix provides a matrix that compares the predicted natural result for diabetes. The columns are the predicted values. The left column is predicted false. The right column is predicted true. The rows are the actual values. The top row is actual false. The bottom row is actual true. Let's label these values with letters so we can use them in equations. Going clockwise from the upper left, TN is true negative, actual not diabetes and predicted to be not diabetes; FP is false positive, actual not diabetes, but predicted to be diabetes; TP is true positive, actual diabetes and predicted to be diabetes; FN is false negative, actual diabetes, but predicted to be not diabetes. Note that these are the default position for scikit-learn's confusion matrix function. These positions can be altered by using the function's labels parameter. Also, the value's position may not be the same in a different language or library. Make sure you review the appropriate documentation for the details. A perfect classifier that classifies everything correctly would return true positive equals 80, true negative equals 151, false negative equals 0, and false positive equals 0. As you can see, our classifier is far from perfect. But is it good enough to meet our problem statement's goal of predicting with 70% or greater accuracy which people are likely to develop diabetes? To see this, we need to calculate some statistics or rather have scikit-learn calculate some statistics for us. That is why we added the classification report. The classification report generates statistics based on the value shown in the confusion matrix. Based on our problem statement, we want to focus on the probability of a true result, meaning the patient has the disease. This is the recall for class 1, or diabetes. Recall is also known as the true positive rate and sensitivity. It is how well the model is predicting diabetes when the result is actually diabetes. Mathematically, recall equals the true positive divided by the sum of the true positive, plus the false negative. We are currently at 0.65, or 65%. We need to get this number to be greater than or equal to 70%. Precision is also known as the positive predictor value. This is how often the patient actually had diabetes when the model said they would. Mathematically, precision equals true positive divided by true positive plus false positive. We would also like to increase this number since that would mean fewer false positives.

Performance Improvement, Take 1

We have several options for improving performance. We can try adjusting the current algorithm. Often algorithms have special parameters called hyperparameters that let us tune the model's performance. Unfortunately, the Naive Bayes model has no hyperparameters. Or we could go back to preparing the data and get additional data or try improving the data we have. But we are using all the data in the Pima dataset and have already processed quite a bit. Another option would be to improve how we train the data. We'll revisit this option later. But for now, let's not do this. The final option is to select a new algorithm that works better with our data. But which algorithm? In general, simple is better, but powerful algorithms can make doing a lot of work simple, so let's try Random Forest. Random Forest is an ensemble algorithm. It's based on decision trees. It creates multiple trees, hence the forest part of the name, with random subsets of the training data. The results of these trees are average. This usually results in improved performance and can reduce the tree algorithm's tendencies to overfit. We'll talk more about overfitting soon. So let's go back to our notebook and train and evaluate the Random Forest algorithm with our data. Now let's train the Random Forest model. We import the RandomForestClassifier and train the model. As before, let's check the accuracy on the training data. Wow, nearly perfect accuracy with the training data. Let's see how it does on the test data. Seventy-one percent. That is a big drop in accuracy from the training data. Also, the recall is low at 54%. Something is happening. Notice the accuracy with training data is near perfect, 0.987, 99%, and with the test date is considerably worse, 0.71, 71%. Whenever you see differences this big, you are seeing the classic signs of an algorithm that is overfitting the training data. That is the model has learned the training data too well.

Why Overfitting Is Bad

The algorithms we use are powerful and can sometimes be too powerful and too accurate. They learn the training data too well. Suppose we have this data distribution. The red X represents positive values and the blue circle negative values. We train our algorithm with this data. The algorithm analyzes the data and trains itself to create a high mathematical order model based on the data. The exact values of this equation do not matter. But notice the X2 cubed and the X3 to the 8th feature values. These high-order terms let this equation define a precise decision boundary between the positive and negative values. But as a result, the training process has created a model that works very well on training data, but poorly when asked to predict values based on data it is not trained on. This is the classic overfit problem and is an issue that must be handled to create machine learning models that work well not only on the training data, but also on real-world data. There are two primary techniques used to handle overfitting. Algorithms often have special tuning parameters called hyperparameters. Using these hyperparameters, we can define how the algorithm learns and operates. The hyperparameter that impacts overfitting goes by different names, but the general term regularization is common. Setting the value of the regularization hyperparameter allows the developer to control how much the algorithm focuses on precisely fitting every corner case of the training data. In this equation, the Lambda term is an example of a regularization hyperparameter. Different settings for Lambda define how much value is produced in the trained model are dampened via the regularization term. This dampening decreases the accuracy of the model on trained data, but potentially increases the accuracy of the trained model on new data. Many algorithms have regularization parameters, and each algorithm reacts differently to the data and to the values of its hyperparameters, like those controlling regularization. Some experimentation is required to get the best values for each hyperparameter with a specific set of data. You may have also noticed that the effect of regularization in this particular example equation is larger when the value of the regularization is smaller. This inverse relationship is common, but by no means universal. Be sure to read the documentation of your algorithm carefully to ensure you understand how the regularization hyperparameter works in your selected algorithm. Another solution to overfitting is through a process called cross-validation. This is where we use multiple subsets of the training data during the training process. We will discuss cross-validation in detail later. Also, the use of regularization and cross-validation are not mutually exclusive. Both can be used at the same time. With these options, we are trying to come up with a compromise between accuracy with training data and accuracy with testing in real-world data. This is often called the bias-variance tradeoff and as something we must consider in almost all supervised machine learning algorithms. The end result is that we need to sacrifice some perfection in training for better overall performance with tests and real-world data.

Performance Improvement, Take 2

Let's review and evaluate our options for further performance improvement. We could try adjusting the current algorithm, but the Random Forest hyperparameter that is likely to impact overfitting is oob_score=True, and we do not have enough data to use that. Getting more data is one of the best ways to help Random Forest not overfit, but we are already using all the data in the Pima study. We can try improving the training by using cross-validation, but that does not always have as much of an impact with Random Forests since the algorithm is essentially doing cross-validation by building multiple trees and comparing them. Finally, we could switch algorithms, but to what algorithm? We went from the simple Naive Bayes to a more powerful ensemble algorithm, Random Forest. However, we also discussed going with a simpler algorithm. Let's do that now. Let's use one of the basic algorithms, logistic regression. Logistic regression is simple in form, but turns out to work quite well in many classification scenarios. Let's try logistic regression, and if we detect overfitting, see if we can adjust its hyperparameters to compensate. As usual, we start by importing LogisticRegression. In the constructor, we set C, the regularization hyperparameter, to 0.7 as a starting guess. Then we train our algorithm, and then we evaluate the algorithm against the test data. Finally, we print the results. The scores look promising, but we are still not achieving the recall score we want. Is there some way to automate selecting the value of the regularization parameter? How about a loop which selects the regularization parameter that returns the highest recall? This while loop will try C values from 0 to 4.9 in increments of 0.1. For each C value, its logistic regression object is created and trained with the training data and then used to predict the test results. Each test recall score is computed, and the highest recall score is recorded. That score is used to get the C value that produced the highest recall score. We also plot the recall scores versus regularization value so that we can get an idea of how recall changes with different regularization values. We are still not getting a recall over 70%. What else can we try? Remember, our data had more non-diabetes results than diabetes results. Perhaps this imbalance is causing an issue.

Understanding and Fixing Unbalanced Classes

Unbalanced classes are very common in datasets. They occur when we have more than one class result, diabetes versus not diabetes in our data, than the other. If you remember from our earlier work, 65% of our results are not diabetes and 35% are diabetes. This imbalance in the class can decrease the performance of an algorithm. Luckily, some algorithm implementations, including logistic regression in scikit-learn, include a hyperparameter that instructs the algorithm to compensate for the class imbalance. This is equivalent to balancing a scale, and the result is a shift in the predicted class boundary. Let's enable this hyperparameter in our model and see if it fixes the class imbalance and, more importantly, improves our model's predictive capabilities. Let's repeat the steps we did before, but this time with balanced classes. We start by executing the loop to find the best value of C, the regularization hyperparameter, but this time we also pass the class_weight="balanced" hyperparameter. Now we do logistics regression just as before with two changes. First, we set the C value to the best score from the loop above, and second, we include the class_weight="balanced" hyperparameter. Let's see how these changes impact the recall score. Wow, we are successful. We achieved our goal of 70% or greater accuracy in predicting diabetes. I guess we are all done. But we have only tested this on one set of test data, and by looping through the values, we sort of cheated by setting the regularization value based on the test data. Is there some way we could test on more data and ensure that we have the algorithm tuned to perform well on a wide set of data? Of course there is, but we may trade off some performance on the test set for better performance on a wider set of data. We previously mentioned cross-validation in reference to handling overfitting, but cross-validation can do so much more.

What Is Cross Validation?

By now, you are familiar with the training/test split we have used to divide our data into training and test datasets. We always hold the testing dataset in reserve to test the model's predictive capabilities. As we try to improve performance of a model, we make adjustments to the model and retest the model's performance with the test data. However, there is a danger in this approach. We are starting to let the checking of accuracy with the test data affect our result. Thus, we are suddenly tuning the model to the test data. However, we need some sort of data to help us tune hyperparameters. It would be great to be able to hold the test data aside and only use it for final testing and have some other data for tuning. Where can we get this other data we can use for tuning? One approach is to go back and re-split our data into three data sets, training, validation, and testing. We use the validation set to tune and validate our model and only use the testing data for a final test. This is a common technique, but has a couple of issues. First, how do we choose the validation set? We want the best validation data. We could just re-split the set to say 50% training, 25% validation, and 25% testing. But sometimes, like in our data, we don't have a lot of extra data. Also, does this address the issue of overtraining? Maybe, depending on the data, but there is a better alternative. Let's talk about cross-validation, and not just any Cross-validation, k-fold cross-validation. With k-fold cross-validation, we do not use the testing data for evaluation. We split our training data into nfolds, each of the same size. We then select one fold to be the validation set. We make the other folds the training set. We train the algorithm on the training set and evaluate it with a fold that is the validation set. We repeat the training and evaluation with the second fold as the validation set, and then again and again until each of the folds has been used as the validation data. The result of each loop through a fold is a set of values for each parameter and associated scores for the accuracy and other performance metrics. This k-fold structure provides capability to generate multiple values with which to tune the hyperparameters. A routine can be written to determine the best value of a hyperparameter, such as regularization for each fold. When all of the folds have been processed, the hyperparameter for the model can be set to the average of the fold's best hyperparameter values. This provides a model with a hyperparameter that may not perform best on a specific fold or other subset of data, but in general performs well on all data. We can use the scikit-learn cross-validation library to access cross-validation methods that make it easy to perform k-fold cross-validation. Then you would need to add code to handle determining the best type of parameter value for each fold and then train the model with the average of the best hyperparameter values. But wouldn't it be great to use a model that did all this work for us? We can. Scikit-learn provide special ensemble versions of the algorithms that contain the code to determine the optimal hyperparameter value and set the model to that value. Scikit-learn has a series of classes that implement variants of the algorithms with the use of cross-validation to determine hyperparameters built in. The naming convention for them is the base class name plus capital CV. Just like other predictor classes, they implement fit, predictor, and other methods that you have seen before. You can use the parameters in the constructor to specify things such as the number of folds, and then when you run fit, k-fold validation is run with a specified number of folds on the training data. Other parameters in the constructor let you define how the optimal value for hyperparameters, such as regularization, is determined. How cool is that? The bottom line is that you can use these CV variants just like their non-CV variants; however, they take a little longer to run since they are doing certain operations multiple times.

Implementing and Evaluating Cross Validation

Since we previously used logistic regression, let's add cross-validation to it. Scikit-learn has an ensemble algorithm that combines logistic regression with k-fold cross-validation in one-easy-to-use interface. This is the LogisticRegressionCV class. Let's use this now and see how easy it is to incorporate cross-validation with logistics regression. As expected, we first have to import the LogisticRegressionCV library, and then we invoke the constructor to create our object. All of the splitting of data and training and evaluation cycles performed in cross-validation can be compute intensive. We need to take advantage of all the resources our system can provide. We do this by setting the parameter n_jobs to -1 to tell the application to use all the cores on our system. We also set the number of folds to 10. We then set the CS param 3. The CS param's documentation say that setting it to an integer value defines the number of values it will try trying to find the best value for the regularization parameter for each fold. Wow! Notice all the parameters available. Ensemble algorithms like LogisticRegressionCV have a lot of parameters. That's a lot of settings one could play with. Now let's predict on test data. Our recall score is not quite what we achieved by tuning the score against the test data, but by using cross-validation, we will likely score better on real-world data. Also remember that with these in ensemble algorithms there are a lot of parameters that we can adjust, and some are quite powerful. With all these parameters, it is likely that some combination of settings will push the scores even higher, but I will leave playing with these parameters as an exercise for you. Just know that with performance improvement, sometimes the hardest thing is to know when to stop, or as Edison said, and as we have seen, Edison was so right.

Summarizing the Evaluation

In this module, we evaluated the performance of our initial Naive Bayes trained model against the data. We did this by using the predict function to generate a prediction of how likely a person was to develop diabetes. We used the confusion matrix and other evaluation functions to review the predictive performance of the model and inform us of areas which needed improvement. We chose to improve the performance by using a different algorithm, Random Forest, but we ran into issues with overfitting. To handle overfitting, we switched to the logistic regression algorithm which supports regularization. We found that with this algorithm we could meet the performance goals of predicting with 70% or greater accuracy if a person was likely to develop diabetes. But to do this, we had to utilize the test data to tune our regularization hyperparameter. A better solution is to use cross-validation to tune the hyperparameter against multiple subsets of training data. This performed a few percent worse on our specific test data, but with real-world data should perform better than a model tuned on a specific set of test data. In the next module, we will summarize the course and go over some additional information that will be helpful in your machine learning journey. See you there.

Summary

The Journey so Far

As we complete this course, let's spend a few minutes re-capping our machine learning journey so far, and going over some suggestions for future learning. We started this course describing some of the ways machine learning is working in our lives. We saw how machine learning affects our lives in many subtle, and not so subtle ways. The key point brought out early on is that machine learning is data driven with data defining the logic used to solve the problem. We discussed that the machine learning workflow provides a defined path for creating machine learning solutions. Throughout the remainder of the course this workflow guided our work. Let's review how we use the workflow. We started by defining the question we are trying to answer. We used requirements and knowledge to enhance and transform our simple question of whether a person was likely to develop diabetes. The end result was a solution statement defining where we would get our data, how we would use that data to create a solution, and the performance we expected from our solution. Following the workflow, we decided to use data from a version of the Pima Indian database using Python and Panda as we cleaned the data, we removed columns that contain correlated values that might cause the training to be misled. We then molded the data by transforming the data types of columns into numeric values that work best for our machine learning. Once we had our data, we selected an algorithm to use. From the large selection of available algorithms, we made our initial algorithm selection by matching the learning type they supported to our problem, ensuring the result I produced was what we needed, reviewing the complexity of their implementation, and choosing basic over enhanced versions of the algorithm.

After we selected our initial algorithm, we used it with our data to create a trained model. We did all this in Python utilizing the scikit-learn library. This involves splitting the data into training and testing data sets. We decided to use 70% of the data for training, and 30% was reserved for testing. We passed this training data to the algorithm to create our trained model. We evaluated this train model with scikit-learn features. We used the predict method to create a prediction with the test data. We used a confusion matrix to evaluate the prediction. We tried several algorithms, and a combat overfitting used the logistics regression algorithm and set its regularization hyperparameter. In the end, this resulted in achieving the result of predicting with 70% or greater accuracy whether a person was likely to develop diabetes. However, we did this by using the test data to tune the regularization hyperparameter to the specifics of the test data set. So we used logistics regression cross-validation to create a model that would perform better on general data sets. This model was within 1% of achieving the target performance. Tuning to increase the performance was left as an additional exercise.

Guides for Your Journey

There are a number of resources available to you to help you continue on your machine learning journey. These Pluralsight courses can be very helpful. We used Python and learned some of its features and functions. Python Fundamentals can help you take the next step and dig into Python to get the most from the language. There are two major languages used in machine learning, Python and R. Understanding machine learning with R is a sister class to this course, and covers similar material using the R language. Another source that can be useful is the UCI Machine Learning Repository. This site contains data sets that can be readily used to learn about machine learning. Many of the data sets are classics in machine learning and are referenced in many articles you will find online. And of course the Jupyter site has more information about using Jupyter Notebook.

The Journey from Here

So where do you go from here? You now know some of the basics of machine learning and quite a bit about supervised machine learning. But as the data science diagram shows, there's a lot more to know. You started this journey with good development skills. Throughout this course, we focused on adding machine learning skills. We have done this through both code and understanding a little bit about math and statistics. And beyond machine learning, we have also tried throughout to add subject matter expertise in the area of disease prediction. So hopefully you moved a little closer to becoming a tech unicorn. And who knows where this new knowledge will take you. Maybe here, but no matter where it takes you, I hope you enjoy the journey. Just remember the words of Mahatma Gandhi, and go forth and have fun. Build something cool and amazing. Happy programming!

Course author

Jerry Kurata

Jerry has Bachelor of Science degrees in Geology and Physics. His plans to work in the oil exploration industry were sidetracked when he discovered he preferred to work with computers on simulation...

Course info

Level	Beginner
Rating	★★★★☆ (568)
My rating	★★★★★
Duration	1h 53m
Released	17 May 2016

Share course