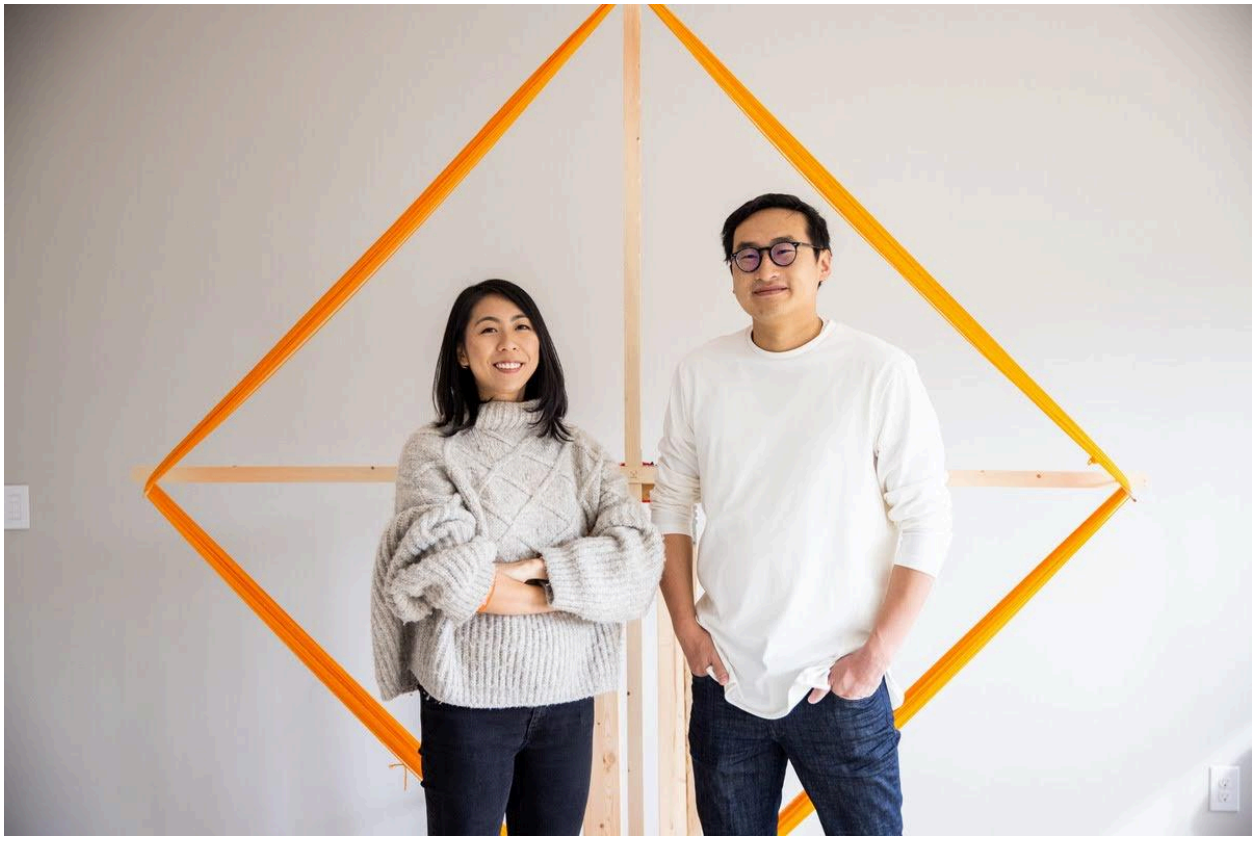
Tab 1

Case study: The Woobles: The power of recommendation systems to drive sales

Previously, you learned that data professionals use machine learning to discover patterns in data and make informed predictions. This case study describes how The Woobles, an online retailer based in Cary, North Carolina, uses machine learning and website analytics to grow their business and enhance customer experience. You'll learn how machine learning generates personalized product recommendations to save time and increase sales. You also learn how chatbots enabled by machine learning can improve customer service.

Company background



Founders, Justine Tiu and Adrian Zhang, established The Woobles as an online retailer of crochet amigurumi kits. Amigurumi is the art of crocheting (or knitting) miniature figures. The word is a mashup of two Japanese words, “ami,” which means crochet, and “nuigurumi,” which means stuffed doll. The penguin, shown below, is one of its most popular kits.

When Justine began teaching herself how to make amigurumi animals, she encountered many roadblocks to learning - from having to piece together videos, blogs, and incomplete instructions. Only after many mistakes and multiple iterations did a cute penguin emerge out of her efforts! Justine thought there must be a better way, and that inspired the creation of The Woobles. Each kit offered in the online store contains everything anyone—from a beginner to an advanced creator—needs to be successful with crochet amigurumi. Resonating with the current trend of people desiring to upskill, The Woobles store on the Shopify e-commerce platform helps people prove to themselves (in a fun way) that they're capable of learning something new.

The challenge

E-commerce stores display product recommendations as links to products a customer might also be interested in purchasing. Product recommendations on The Woobles' website help drive product sales. The biggest challenge is making sure that the right products are recommended to the right customers at the right time, all in hopes of increasing sales.

The Woobles gets some help from Shopify's machine learning. Businesses that implement product recommendations with machine learning don't have to worry about the details to create or personalize them for each customer; they can focus their time on other important matters. As Adrian described it, "Working with machine learning is a black box. You plug things in and hope something good happens. It's helpful to know what machine learning is looking at, but as an entrepreneur, I wouldn't want to know all the ins and outs."

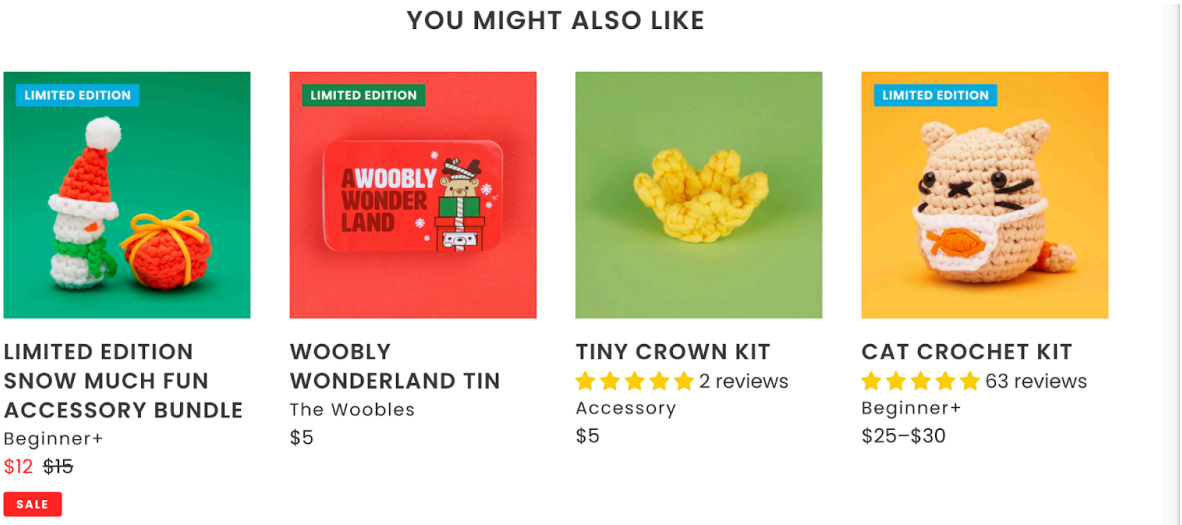
As entrepreneurs and business owners, Adrian and Justine do not have the time or expertise to develop their own recommendation systems. To them, these systems seem like a black box, but to the data scientists who create them, they're quite understandable!

The approach

The Woobles relies on a combination of machine learning and analytics to ensure that product recommendations displayed to customers lead to the greatest increase in sales. It uses an automated and manual approach to managing product recommendations on its website. Website metrics from Google Analytics and analytics data from Triple Whale inform both machine-based and human decisions about which recommendations to display.

Automated recommendations on product pages

The Woobles displays product recommendations at the bottom of each product page to encourage customers to continue browsing and shopping.



Because The Woobles is hosted on the Shopify e-commerce platform, Shopify's machine learning algorithm automatically generates these recommendations. The algorithm relies on purchase history data to recommend products that have historically been purchased together. It also relies on product descriptions in the store to recommend products with similar descriptions.

These recommendation algorithms are complex and closely guarded proprietary secrets, as they are very valuable in today's e-commerce landscape. They can make use of elements from both content-based and collaborative filtering. They also can make use of some natural language processing (NLP) techniques, as is the case here.

In this case, the algorithm might use content-based filtering to compare all of the items that The Woobles sells and assign each to a place in a featurespace. For instance, beginner-level kits might be closer to each other in this featurespace than expert-level kits.

At the same time, the model can use collaborative filtering in conjunction with average item scores on the website. For example, if a person shows interest in product A, and the highest similarity score from a content-based filtering approach returns product B with a 3.9-star rating, and product C is in second place by similarity score but it has a 4.9-star rating, it may combine this information to offer product C to the customer.

Finally, the algorithm uses NLP techniques to help determine product similarities. To do this, it might use a “bag of words” technique. This is a method in which, for each product, the description is “tokenized,” or separated into its individual words, which are then used as features. The words are then counted. So, if the words “needle,” “pattern,” and “expert” appear five, three, and two times, respectively, these counts are captured. This is repeated for all of the products, and the resulting word counts are put into what’s known as a “document-term matrix”—essentially a table with words as columns and products as rows. Then the algorithm calculates the distance between each product vector. There are many different ways of doing this, and they can be very complex. Ultimately, the products with the least distance between them or the greatest similarity are then recommended.

Key benefits of Shopify’s product recommendations

- Shopify generates the links to the recommended products and the process is automated using Shopify’s machine learning algorithm.
- Shopify uses customer transaction data to ensure its product recommendations algorithm has up-to-date information when selecting products to recommend.
- The cost to increase sales via auto-generated product recommendations is minimal.

The results

Automated product recommendations enhance the customer shopping experience by helping them find relevant products, while maximizing cart order values. There are many ways that online retailers can try to increase sales, but implementing product recommendations is a cost-effective option that has been proven to pay off for The Woobles. In particular, product recommendations that are auto-generated by Shopify's machine learning algorithm provide a personalized shopping experience to each customer. Customers save time by having product links presented to them, and this personalization is achieved without any manual work or additional time commitments by The Woobles staff. In fact, The Woobles estimates Shopify's product recommendations on product pages saves five hours per month in manual labor since the process is automated through machine learning.

The future

Product recommendation isn't the only way for a small business to leverage machine learning. As their business has grown, so too has the number of questions related to product information, shipping, billing, and other routine aspects of commerce. A lot of time is spent responding to these inquiries. To help them, The Woobles would like to implement another solution enabled by machine learning: chatbots. Chatbots are software programs designed to simulate conversation with human users, especially over the Internet. The Woobles envisions that chatbots could help answer questions potential customers have after they watch any of the crochet tutorials offered on the website. Chatbots can assist with certain aspects of customer acquisition, such as customer education.

Like recommendation systems, chatbots are highly complex, multilayer algorithms that make use of multiple techniques. For example, a chatbot might break down sentences into bags of words or "n-grams" (combinations comprised of *n* number of words), then use unsupervised learning to reduce the dimensionality of the resulting vector matrix and cluster it, and use supervised learning to predict the words most likely to follow a

given prompt. The result, ideally, is an experience that is indistinguishable from chatting with a human being.

Key takeaways

Below is a summary of the main insights from this case study.

- Recommendation engines can use combinations of content-based and collaborative filtering. Products can be compared based on intrinsic similarities *and* offered based on collaborative satisfaction ratings. NLP can be used to help determine content-based similarity metrics.
- Businesses that implement product recommendations with machine learning save time. By using an ML-driven solution to product recommendations, The Woobles don't have to regularly spend time manually updating recommendations as new data and products come in.
- Analysis of current sales data and customer trends is critical in selecting the products featured in product recommendations. An automated approach to managing online product recommendations is an effective component of The Woobles' strategy to increase sales and maximize cart order values.
- Chatbots enabled by machine learning could also be implemented to assist with other time-consuming tasks, such as the customer education aspects of customer acquisition.

Tab 2

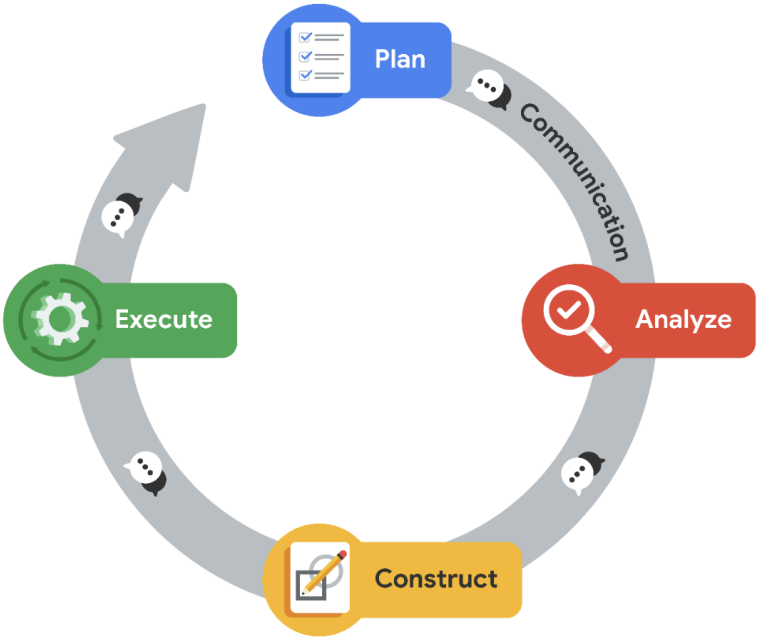
More about planning a machine learning project

The PACE workflow is something that can be used to keep the most experienced data professionals on track in their projects. In this reading, you will learn more about the Plan stage of PACE and the things that must be considered and determined to ensure a smooth and successful model development process.

The Plan Stage

The PACE workflow is something that you can use to keep you on track, no matter the project you're working on. Each step is important to get to your final product. However, like many things, the most important part is setting up the foundations of your project - The Plan Stage.

The Plan Stage is the part of the process where you first start thinking about what the problem actually is, and what needs to be done to find a solution. You start to consider what tools you have available to you, and how you'll need to manipulate the dataset. Sometimes this can be as straightforward as needing to create some visualizations for the data. Or, it can get as complex as needing to make a predictive model using the dataset.



The plan that you create during this stage will be carried through the whole process, so it is important to really make sure you've considered all the aspects and constraints of the project. However, that isn't to say that the plan you create must stay unchanging, you can absolutely reassess as you progress. It is there to get you started heading in the right direction.

What should your Plan Include?

This section of the course focuses on machine learning algorithms, so we will use those types of projects as the example here. However, you need to think about whether you need a model in the first place! Many analytical tasks do not require the creation of a model, and you could spend time creating something that is not necessary to what you're trying to achieve.

Knowing What You Need For a Problem

The first thing to do when forming your plan is to consider the end goal. What exactly are you trying to model, and what types of results from the model are needed? Something that can be determined immediately is what type of machine learning model

you'll need. The two types that you've seen so far are Supervised and Unsupervised models.

Supervised models are used to make predictions about unseen events. These types of models use labeled data, and the model will use these labels and the predictor variables present to learn from the dataset. And when given new data points, they're able to make a prediction of the label. So, for example, if you're tasked with predicting rainfall amounts, you already know that you will need a supervised learning model.

Unsupervised models, on the other hand, don't really make predictions. They are used to discover the natural structure of the data, finding relationships within unlabeled data. So, for example, if you're tasked with discovering relationships between customer habits and segment users, you know you'll need an unsupervised model.

Now, let's go back to the rainfall example. Just from that problem statement alone, we know we need a supervised learning model. However, not all supervised learning models are the same. The two main types of supervised learning are Regression and Classification. There are different types of regression models that you have practiced, with different models able to perform regression or classification tasks.

Linear regression models are used when the result must be a continuous variable. As you have learned, continuous variables are numerical values that can have an unlimited number of values between the highest and lowest points of measurement. So if you need rainfall amounts in inches or centimeters, you know a linear regression model is needed.

However, what if we don't need exact rainfall amount predictions, but just whether or not it will rain that day? This is where a classification model, such as a logistic regression model, would be more appropriate. Classification models will deliver results as a categorical variable, where there is a finite set of values that the variable can be. In this example, the model would only ever predict two results: Will Rain or Won't Rain.

Figuring out the tools you need

After you've determined what type of model you're going to need, you must consider what you have at your disposal to complete the project. Most importantly, you need to figure out if you have the data you'll need to build the model.

If your dataset only has one or two predictor variables, it probably will not produce a model that will be useful. Or, if it has very few data points, the model's performance will similarly suffer. On the other hand, your dataset might be large and unwieldy, meaning that you'll either need to clean it up or cut it down to get it into a format that you can use to train the model. Having these issues means that you'll have to put in a little extra work to get it to usable form, or look elsewhere for data that will be helpful to create the model.

Key Takeaways

- The PACE workflow for machine learning is very useful for planning out and solving data driven problems.
- The Plan stage of PACE is one of the most important, setting you up for success throughout the rest of the process
- In the Plan stage, you first consider the problem at hand and what will be needed to solve it
- You also verify that you have the tools and resources you need to solve the problem
- The Plan is not set in stone. It just serves as a foundational starting point for the rest of the project

Tab 3

Explore feature engineering

In this reading, you will learn more about what happens in the Analyze stage of PACE—namely, feature engineering. The meaning of the term “feature engineering” can vary broadly, but in this course it includes feature selection, feature transformation, and feature extraction. You will come to understand more about the considerations and process of adjusting your predictor variables to improve model performance.

Feature Engineering

When building machine learning models, your model is only ever going to be as good as your data. Sometimes, the data you have will not be predictive of your target variable. For example, it’s unlikely that you can build a good model that predicts rainfall if you train it on historical stock market data. In this case, it might seem obvious, but when you’re building a model, you’ll often have features that plausibly could be predictive of your target, but in fact are not. Other times, your model’s features might contain a predictive signal for your model, but this signal can be strengthened if you manipulate the feature in a way that makes it more detectable by the model.

Feature engineering is the process of using practical, statistical, and data science knowledge to select, transform, or extract characteristics, properties, and attributes from raw data. In this reading, you will learn more about these processes, when and why to use them, and what good feature engineering can do for your model.

Feature Selection

Feature selection is the process of picking variables from a dataset that will be used as predictor variables for your model. With very large datasets, there are dozens if not hundreds of features for each observation in the data. Using all of the features in a dataset often doesn’t give any performance boost. In fact, it may actually hurt

performance by adding complexity and noise to the model. Therefore, choosing the features to use for the model is an important part of the model development process.

Generally, there are three types of features:

1. Predictive: Features that by themselves contain information useful to predict the target
2. Interactive: Features that are not useful by themselves to predict the target variable, but become predictive in conjunction with other features
3. Irrelevant: Features that don't contain any useful information to predict the target

You want predictive features, but a predictive feature can also be a redundant feature. Redundant features are highly correlated with other features and therefore do not provide the model with any new information—for example, the steps you took in a day, may be highly correlated with the calories you burned. The goal of feature selection is to find the predictive and interactive features and exclude redundant and irrelevant features.

The feature selection process typically occurs at multiple stages of the PACE workflow. The first place it occurs is during the Plan phase. Once you have defined your problem and decided on a target variable to predict, you need to find features. Keep in mind that datasets are not always prepackaged in nice little tables ready to model. Data professionals can spend days, weeks, or even months acquiring and assembling features from many different sources.

Feature selection can happen once more during the Analyze phase. Once you do an exploratory data analysis, it might become clear that some of the features you included

are not suitable for modeling. This could be for a number of reasons. For example, you might find that a feature has too many missing or clearly erroneous values, or perhaps it's highly correlated with another feature and must be dropped so as not to violate the assumptions of your model. It's also common that the feature is some kind of metadata, such as an ID number with no inherent meaning. Whatever the case may be, you might want to drop these features.

During the Construct phase, when you are building models, the process of improving your model might include more feature selection. At this point, the objective usually is to find the smallest set of predictive features that still results in good overall model performance. In fact, data professionals will often base final model selection not solely on score, but also on model simplicity and explainability. A model with an R2 of 0.92 and 10 features might get selected over a model with an R2 of 0.94 and 60 features. Models with fewer features are simpler, and simpler models are generally more stable and easier to understand.

When data professionals perform feature selection during the Construct phase, they typically use statistical methodologies to determine which features to keep and which to drop. It could be as simple as ranking the model's feature importances and keeping only the top *a*% of them. Another way of doing it is to keep the top features that account for ≥ b% of the model's predictive signal. There are many different ways of performing feature selection, but they all seek to keep the predictive features and exclude the non-predictive features.

Feature Transformation

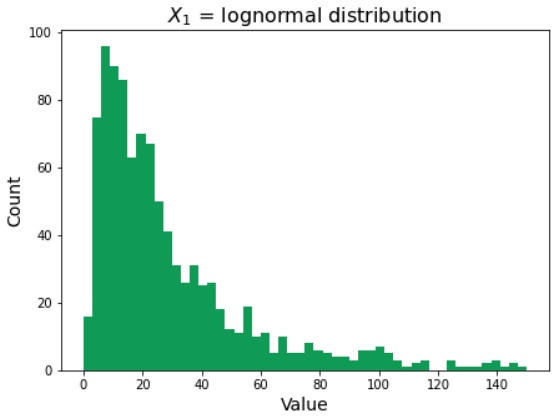
Feature transformation is a process where you take features that already exist in the dataset, and alter them so that they're better suited to be used for training the model. Data professionals usually perform feature transformation during the Construct phase,

after they've analyzed the data and made decisions about how to transform it based on what they've learned.

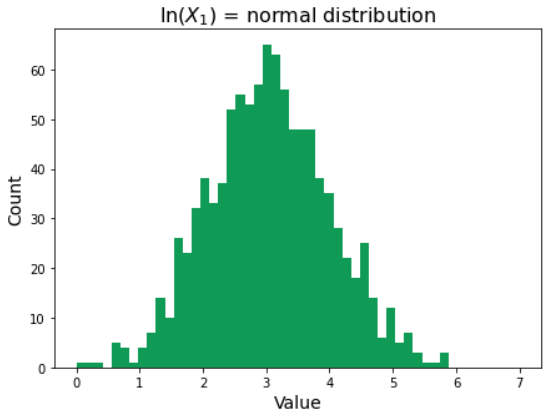
Log normalization

There are various types of transformations that might be required for any given model. For example, some models do not handle continuous variables with skewed distributions very well. As a solution, you can take the log of a skewed feature, reducing the skew and making the data better for modeling. This is known as log normalization.

For instance, suppose you had a feature *X*1 whose histogram demonstrated the following distribution:



This is known as a log-normal distribution. A log-normal distribution is a continuous distribution whose logarithm is normally distributed. In this case, the distribution skews right, but if you transform the feature by taking its natural log, it normalizes the distribution:



Normalizing a feature's distribution is often better for training a model, and you can later verify whether or not taking the log has helped by analyzing the model's performance.

Scaling

Another kind of feature transformation is scaling. Scaling is when you adjust the range of a feature's values by applying a normalization function to them. Scaling helps prevent features with very large values from having undue influence over a model compared to features with smaller values, but which may be equally important as predictors.

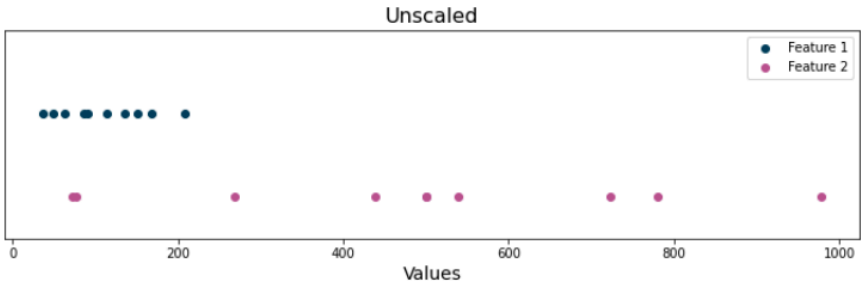
There are many scaling methodologies available. Some of the most common include:

Normalization

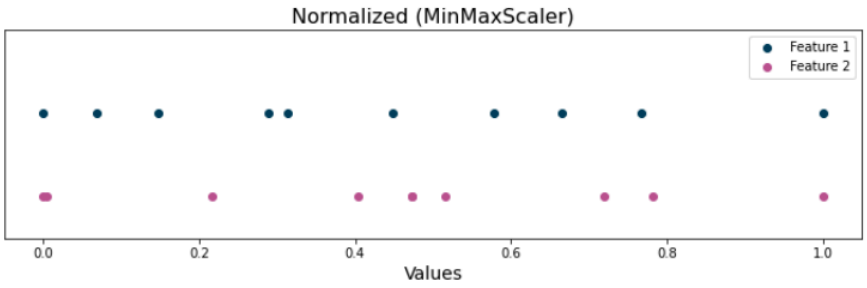
Normalization (e.g., **MinMaxScaler** in scikit-learn) transforms data to reassign each value to fall within the range [0, 1]. When applied to a feature, the feature's minimum value becomes zero and its maximum value becomes one. All other values scale to somewhere between them. The formula for this transformation is:

$$x_{i,normalized} = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

For example, suppose you have feature 1, whose values range from 36 to 209; and feature 2, whose values range from 72 to 978:



It is apparent that these features are on different scales from one another. Features with higher magnitudes of scale will be more influential in some machine learning algorithms, like K-means, where Euclidean distances between data points are calculated with the absolute value of the features (so large feature values have major effects, compared to small feature values). By min-max scaling (normalizing) each feature, they are both reduced to the same range:

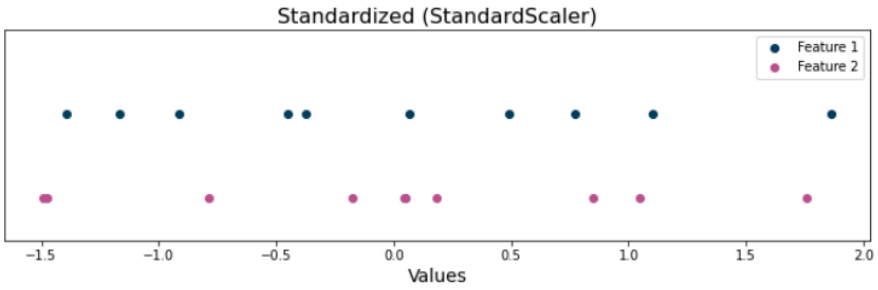


Standardization

Another type of scaling is called **standardization** (e.g., **StandardScaler** in scikit-learn). Standardization transforms each value within a feature so they collectively have a mean of zero and a standard deviation of one. To do this, for each value, subtract the mean of the feature and divide by the feature's standard deviation:

$$x_{i,standardized} = \frac{x_i - x_{mean}}{x_{stand.dev.}}$$

This method is useful because it centers the feature's values on zero, which is useful for some machine learning algorithms. It also preserves outliers, since it does not place a hard cap on the range of possible values. Here is the same data from above after applying standardization:



Notice that the points are spatially distributed in a way that is very similar to the result of normalizing, but the values and scales are different. In this case, the values now range from -1.49 to 1.76.

Encoding

Another form of feature transformation is known as encoding. Variable encoding is the process of converting categorical data to numerical data. Consider the bank churn dataset. The original data has a feature called "Geography", whose values represent each customer's country of residence—France, Germany, or Spain. Most machine learning methodologies cannot extract meaning from strings. Encoding transforms the strings to numbers that can be interpreted mathematically.

The "Geography" column contains nominal values, or values that don't have an inherent order or ranking. As such, the feature would typically be encoded into binary. This process requires that a column be added to represent each possible class contained within the feature.

Geography	Is France	Is Germany	Is Spain
France	1	0	0
Germany	0	1	0
Spain	0	0	1
France	1	0	0

Tools commonly used to do this include `pandas.get_dummies()` and `OneHotEncoder()`. Often methods drop one of the columns to avoid having redundant information in the dataset . Note that information isn't lost by doing this. If you have this…

Customer	Is France	Is Germany
Antonio García	0	0

… then you know this customer is from Spain!

Keep in mind that some features may be inferred to be numerical by Python or other frameworks but still represent a category. For example, suppose you had a dataset with people assigned to different arbitrary groups: 1, 2, and 3:

Name	Group

Rachel Stein	2
Ahmed Abadi	2
Sid Avery	3
Ha-rin Choi	1

The “Group” column might be encoded as type `int`, but the number is really only representative of a category. Group 3 isn’t two units “greater than” group 1. The groups could just as easily be labeled with colors. In this case, you could first convert the column to a string, and then encode the strings as binary columns. This is a problem that can be solved upstream at the stage of data generation: categorical features (like a group) should not be recorded using a number.

A different kind of encoding can be used for features that contain discrete or ordinal values. This is called ordinal encoding. It is used when the values *do* contain inherent order or ranking. For instance, consider a “Temperature” column that has values of cold, warm, and hot. In this case, ordinal encoding could reassign these classes to 0, 1, and 2.

Temperature	**Temperature (Ordinal encoding)**
cold	0
warm	1

hot	2

This method retains the order or ranking of the classes relative to one another.

Feature extraction

Feature extraction involves producing new features from existing ones, with the goal of having features that deliver more predictive power to your model. While there is some overlap between extraction and transformation colloquially, the main difference is that a new feature is created from one or more other features rather than simply changing one that already exists.

Consider a feature called “Date of Last Purchase,” which contains information about when a customer last purchased something from the company. Instead of giving the model raw dates, a new feature can be extracted called “Days Since Last Purchase.” This could tell the model how long it has been since a customer has bought something from the company, giving insight into the likelihood that they’ll buy something again in the future. Suppose that today’s date is May 30th, extracting a new feature could look something like this:

Date of Last Purchase	**Days Since Last Purchase**
May 17th	13
May 29th	1
May 10th	20

May 21st	9

Features can also be extracted from multiple variables. For example, consider modeling if a customer will return to buy something else. In the data, there are two variables: “Days Since Last Purchase” and “Price of Last Purchase.” A new variable could be created from these by dividing the price by the number of days since the last purchase, creating a new variable altogether.

Days Since Last Purchase	**Price of Last Purchase**	**Dollars Per Day Since Last Purchase**
13	$85	$6.54
1	$15	$15.00
20	$8	$0.40
9	$43	$4.78

Sometimes, the features that you are able to generate through extraction can offer the greatest performance boosts to your model. It can be a trial and error process, but finding good features from the raw data is what will make a model stand out in industry.

Key takeaways

- Analyzing the features in a dataset is essential to creating a model that will produce valuable results

- Feature Selection is the process of dropping any and all unnecessary or unwanted features from the dataset
- Feature Transformation is the process of editing features into a form where they’re better for training the model
- Feature Extraction is the process of creating brand new features from other features that already exist in the dataset

Tab 4

More about imbalanced datasets

Imbalanced Datasets

As you may have noticed as you've gone through this program, many raw datasets that you will encounter require varying levels of work to get them in a place where they are ready for modeling. There might be missing values, or the variables might not be in the exact format that you need. You perform an exploratory data analysis to get a good understanding of the data. When working with classification models, however, there is something else to consider before moving forward: class balance.

For categorical variables, the different possible values that each can take are known as classes. This is true for both predictor variables and target variables. If you were trying to classify the weather on a given day as rainy or sunny, these would be considered two classes. The number of classes is equal to the number of unique values in the variable.

The number of occurrences of each class in the target variable is known as the class distribution. When predicting a categorical target, problems can arise when the class distribution is highly imbalanced. If there are not enough instances of certain outcomes, the resulting model might not be very good at predicting that class.

This is where the process of class balancing comes in, a process that allows you to manipulate the dataset, or the model fitting process, in a way that the class imbalance that exists doesn't affect the performance of the resulting model. In this reading, you will explore more about imbalanced datasets, the problems that can occur when working with them, and some methods of adjusting the class distribution to minimize the imbalance.

Situations of Imbalance

Classification is a very broad field, with many applications across different industries. Some business needs, however, require a model to be good at classifying things that occur relatively rarely in the data. These types of problems have several names that are used commonly, including rare event prediction, extreme event prediction, and severe class imbalance. However, they all refer to the same thing: at least one of the classes in the target variable occurs much less frequently than another.

Consider this example. You are tasked with creating a model that will classify emails that are sent to the company either as “spam” or “not spam.” The company receives thousands and thousands of emails daily, not to mention all the emails they’ve received in the past. However, the number of examples of spam is very small. For the sake of this example, say that 10 emails per day are identified as spam manually.

All the emails are collected into a dataset to train the model, with each example labeled as spam or not spam. The problem is that the dataset contains many, many more examples of not spam than spam. In this case, spam is known as the minority class and not spam is known as the majority class.

This doesn't have the makings of a very good model. With so few examples of spam relative to examples of not spam, the model can have difficulty detecting the minority class, resulting in its being biased toward the majority class or possibly never predicting the minority class at all.

Balancing a Dataset

Class balancing refers to the process of changing the data by altering the number of samples in order to make the ratios of classes in the target variable less asymmetrical. It is a large field of study on its own, and there are several methods that allow you to balance the classes while maintaining the integrity of the data. Here, you’ll learn about some of the most common methods that can be used to create a better model.

There are two general strategies to balance a dataset, and the method that is better to use generally is decided by how much data you have in the first place.

Downsampling

Downsampling is the process of making the minority class represent a larger share of the whole dataset simply by removing observations from the majority class. It is mostly used with datasets that are large. But how large is large enough to consider downsampling? Tens of thousands is a good rule of thumb, but ultimately this needs to be validated by checking that model performance doesn't deteriorate as you train with less data.

One way to downsample data is by selecting some observations randomly from the majority class and removing them from the dataset. There are some more technical, mathematically based methods, but random removal works very well in most cases.

Upsampling

Upsampling is basically the opposite of downsampling, and is done when the dataset doesn't have a very large number of observations in the first place. Instead of removing observations from the majority class, you increase the number of observations in the minority class.

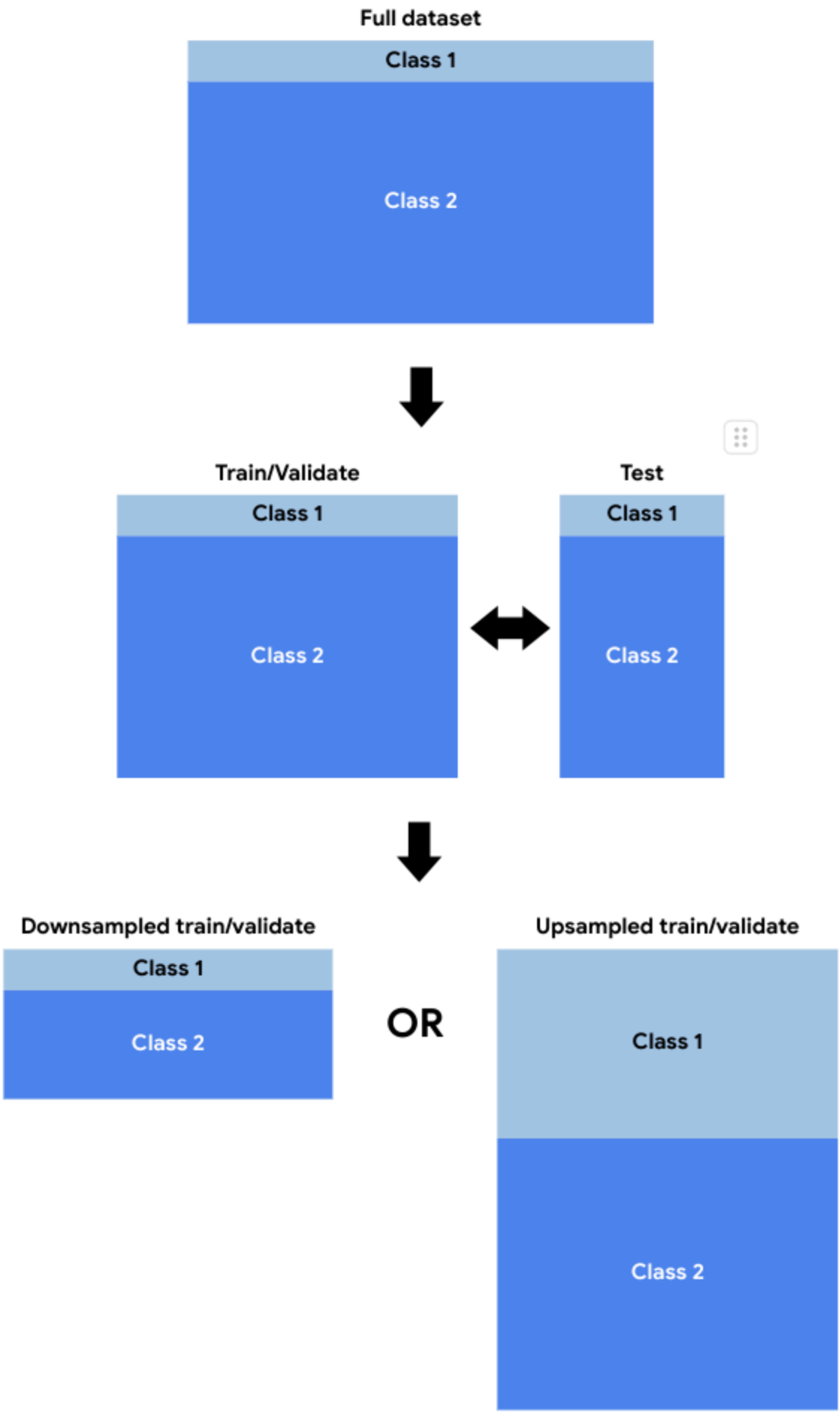
There are a couple of ways to go about this. The first and easiest method is to duplicate samples of the minority class. Depending on how many such observations you have compared to the majority class, you might have to duplicate each sample several times over.

Another way is to create synthetic, unique observations of the minority class. On the surface, there seems to be something wrong about editing the dataset like this, but if the goal is simply to train a better-performing model, it can be a valid and useful technique. You can generate these synthetic observations from the observations that currently exist. For example, you can average two points of the minority class and add the result

to the dataset as a sample of the minority class. This can even be done algorithmically using publicly available Python packages.

How to do it

In both cases, upsampling and downsampling, it is important to leave a partition of test data that is unaltered by the sampling adjustment. You do this because you need to understand how well your model predicts on the actual class distribution observed in the world that your data represents. In the case of the spam detector example, it’s great if your model can score well on resampled data that is 80% not spam and 20% spam, but you need to know how it will work when deployed in the real world, where spam emails are much less frequent. This is why the test holdout data is not rebalanced.


Full dataset
Class 1
Class 2
Train/Validate
Class 1
Class 2
Test
Class 1
Class 2
Downsampled train/validate
Class 1
Class 2
OR
Upsampled train/validate
Class 1
Class 2

Consequences

Manipulating the class distribution of your data doesn't come without consequences. The first consequence is the risk of your model predicting the minority class more than it should. By class rebalancing to get your model to recognize the minority class, you might build a model that over-recognizes that class. That happens because, in training, it learned a data distribution that is not what it will be in the real world.

Changing the class distribution affects the underlying class probabilities learned by the model. Consider, for example, how the Naive Bayes algorithm works. To calculate the probability of a class, given the features, it uses the background probability of a class in the data.

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

In the numerator, *P*(*A*) (the probability of class *A*) depends on class probabilities encountered in the data. If the data has been enriched with a particular class, then this probability will not be reflective of meaningful real-life patterns, because it's based on altered data.

When to do it

Class rebalancing should be reserved for situations where other alternatives have been exhausted and you still are not achieving satisfactory model results. Some guiding questions include:

- How severe is the imbalance? A moderate (< 20%) imbalance may not require any rebalancing. An extreme imbalance (< 1%) would be a more likely candidate.
- Have you already tried training a model using the true distribution? If the model doesn't fit well due to very few samples in the minority class, then it could be worth rebalancing, but you won't know unless you first try without rebalancing.

- Do you need to use the model's predicted class probabilities in a downstream process? If all you need is a class assignment, class rebalancing can be a very useful tool, but if you need to use your model's output class probabilities in another downstream model or decision, then rebalancing can be a problem because it changes the underlying probabilities in the source data.

Key Takeaways

- Imbalanced datasets can be a problem when working on classification problems
- Class imbalance isn't always a problem. The likelihood and severity of it negatively affecting model performance generally increase as the degree of the imbalance increases.
- Downsampling involves removing some observations from the majority class, making it so they make up a smaller percent of the dataset than before.
- Upsampling involves taking observations from the minority class and either adding copies of those observations to the dataset or generating new observations to add to the dataset.

Tab 5

Naive Bayes classifiers

Sometimes, the simplest solutions are the most powerful ones. When it comes to supervised machine learning techniques, Naive Bayes is a perfect example of that. The theoretical foundations of the model date back nearly 300 years, and though the field of data science and machine learning has grown immensely in recent years, Naive Bayes models remain relevant because they are simple, fast, and good predictors. In certain situations, Naive Bayes is also known to outperform much more advanced classification methods. Even if a more advanced model is required, producing a Naive Bayes model can also be a great starting point. Therefore, the Naive Bayes classifier is something that every data professional needs in their machine learning skill set.

How do Naive Bayes models work?

A Naive Bayes model is a supervised learning technique used for classification problems. As with all supervised learning techniques, to create a Naive Bayes model you must have a response variable and a set of predictor variables to train the model.

The Naive Bayes algorithm is based on Bayes' Theorem, an equation that can be used to calculate the probability of an outcome or class, given the values of predictor variables. This value is known as the posterior probability.

That probability is calculated using three values:

- The probability of the outcome overall P(A)
- The probability of the value of the predictor variable P(B)
- The conditional probability P(B|A) (Note: P(B|A) is interpreted as *the probability of B, given A.*)

The probability of the outcome overall, P(A), is multiplied by the conditional probability, P(B|A). This result is then divided by the probability of the predictor variable, P(B), to obtain the posterior probability.

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

The goal of Bayes' Theorem is to find the probability of an event, A, given that another event B is true. In the context of a predictive model, the class label would be A and the predictor variable would be B. P(A) is considered the prior probability of event A before any evidence (feature) is seen. Then, P(A|B) is the posterior probability, or the probability of the class label after the evidence (feature) has been seen.

In a predictive model, this calculation is carried out for each feature, for each class. Then the probabilities are multiplied together. The class with the highest resulting product is the model's final prediction for that sample.

These models make a number of assumptions about the data to work properly. One of the most important is the assumption that each predictor variable (different Bs in the formula) is independent from the others, conditional on the class. This is called conditional independence. Variables B and C are independent of one another *on the condition that* a third variable, A, exists such that:

$$P(B|C, A) = P(B|A)$$

This equation can be interpreted as "the probability of B, given C and A, is equal to the probability of B*,* given A." In other words, given A, introducing C does not change the probability of B. Note that two features can only be considered conditionally independent of each other when considered in relation to a third variable. Furthermore, it's possible for variables B and C to be conditionally independent of one another with respect to A*,* but not with respect to another variable, say, Z.

In Naive Bayes, the predictor variables (B and C in the equation above) are assumed to be conditionally independent of each other, given the target variable (A). This is an assumption that very often is not actually true. However, Naive Bayes models still often perform well in spite of the data violating the assumption. The assumption is made to simplify the model. Otherwise, long probabilistic chains would have to be calculated to determine the probability of a feature's value with respect to the values of every other variable.

Another assumption of the data is that no predictor variable has any more predictive power than any other predictor. In other words, the individual predictor variables are assumed to contribute equally to the model's prediction. Like the assumption of class-conditional independence between the features, this assumption is also often violated by real-world data, but Naive Bayes still often proves a good model in spite of this.

Positives and negatives

Of all the classification algorithms that are still used today, Naive Bayes is one of the simplest. However, it is still able to produce valuable results. Its simplicity comes as an asset because it is one of the most straightforward algorithms to implement. In spite of their assumptions, Naive Bayes classifiers work quite well in many industry problems, most famously for document analysis/classification and spam filtering.

Additionally, the training time for a Naive Bayes model can sometimes be drastically lower than for other models because the calculations that are needed to make it work are relatively cheap in terms of computer resource consumption. This also means it is highly scalable and able to work with large increases in the amount of data it must handle.

One of the biggest problems with Naive Bayes is the data assumptions that were mentioned earlier. Few datasets have truly conditionally independent features—it is

something that is very rare in the world today. However, Naive Bayes models can still perform well even if the assumption of conditional independence is violated.

Another issue that could arise is what is known as the “zero frequency” problem. This occurs when the dataset you’re using has no occurrences of a class label and some value of a predictor variable together. This would mean that there is a probability of zero. Since the final posterior probability is found by multiplying all of the individual probabilities together, the probability of zero would automatically make the result zero. Library implementations of the algorithms account for this by adding a negligible value to each variable count (usually 1) to ensure a non-zero probability.

Implementations in scikit-learn

There are several implementations of Naive Bayes in scikit-learn, all of which are found in the `sklearn.naive_bayes` module. Each is optimized for different conditions of the predictor variables. This reading will not delve into the mechanics of each variation. It is intended as a basic guide to using these models. Feel free to explore them on your own!

`BernoulliNB`: Used for binary/Boolean features

`CategoricalNB`: Used for categorical features

`ComplementNB`: Used for imbalanced datasets, often for text classification tasks

`GaussianNB`: Used for continuous features, normally distributed features

`MultinomialNB`: Used for multinomial (discrete) features

Of course, datasets aren’t always limited to features of just a single type. In these cases, it’s often best to try the one that makes the most sense given your data. Often it’s useful to try several. It might also be the case that none of them work very well. Remember that modeling can be a messy process. Things will break. Assumptions will

be violated. Nothing will be perfect, so don't let perfect be the enemy of good. Careful planning, sound decision-making, and a lot of perseverance can go a long way.

Key Takeaways

- Naive Bayes is a classification technique that is based on Bayes' Theorem.
- The model will calculate the posterior probability of an event, given the values of the predictor variables.
- This model assumes class-conditional independence of the predictor variables, which can sometimes lead to poor performance in conditions when this assumption is violated.
- Naive Bayes models can be good to use because they are relatively simple to create and are highly scalable depending on the business need.
- scikit-learn has different implementations of the algorithm, each optimized for different conditions of the data.

Tab 6

More about evaluation metrics for classification models

You have learned much about classification models, from logistic regression to Naive Bayes, and you will encounter yet others elsewhere in this course. Classification tasks are among the most common applications of machine learning. Knowing these techniques will empower you to take on data challenges from fraud detection to predicting stock market events and World Cup winners.

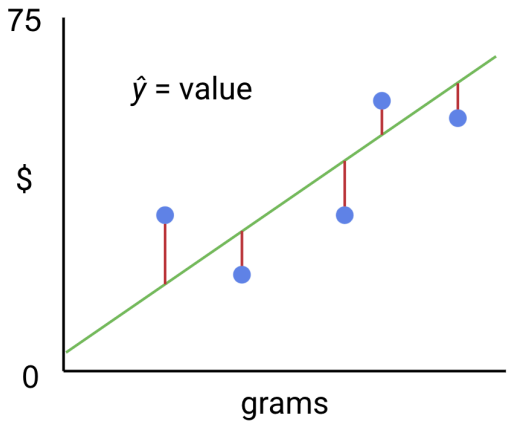
In this reading, you will reexamine some different ways to evaluate the performance of classification models, and also learn some new ones. You will review the confusion matrix and how it relates to accuracy, precision, and recall. (To revisit this information, please review Key metrics to assess logistic regression results and Common logistic regression metrics in Python.) You'll also learn about model evaluation using receiver operating characteristic (ROC) curves and area under the ROC curve (AUC), as well as F_1 score and F_β score.

Evaluation metrics for classification models

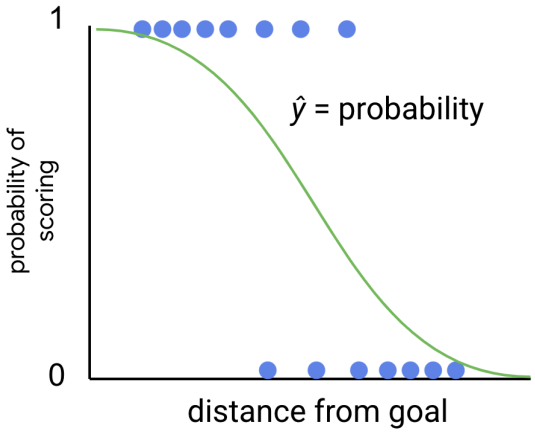
Previously you learned about linear regression models, their assumptions, theory, and use cases. You also learned how to evaluate these models using metrics like R^2, mean squared error (MSE), root mean squared error (RMSE), and mean absolute error (MAE). These metrics are all useful when evaluating the error of a prediction on a continuous variable.

You also learned about logistic regression, which does not predict on a continuous variable, but rather on a binary variable. It predicts a class. As such, it is a type of classification model.

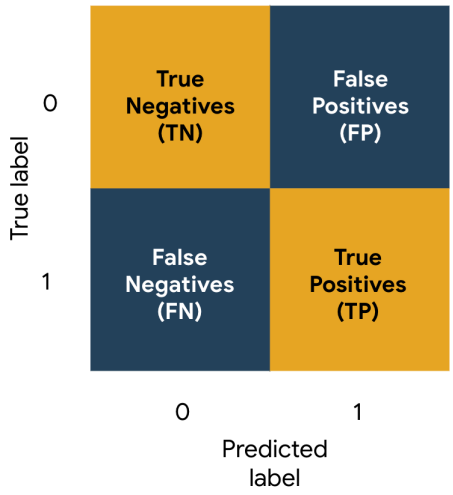
Classification models cannot be evaluated using the same metrics as linear regression models. Consider why. With a linear regression model, your model predicts a continuous value that has a unit label (e.g., dollars, kilograms, minutes, etc.) and your evaluation pertains to the residuals of the model's predictions—the difference between predicted and actual values:



But with a binary logistic regression, for example, your observations are represented by two classes; they are either one value or another. The model predicts a probability, and assigns observations to a class based on that probability.



Your evaluation must therefore pertain to the class assignment of your model. There are a number of such evaluation metrics that can be used with classification models. Some of these were introduced previously: accuracy, precision, and recall. These can all be derived from a confusion matrix, which is a graphical representation of your model.



Accuracy

Accuracy is the proportion of data points that are correctly classified. It is an overall representation of model performance, represented as:

$$\text{Accuracy} = \frac{\text{true positives + true negatives}}{\text{total predictions}}$$

Accuracy=

$$\frac{\text{true positives + true negatives}}{\text{total predictions}}$$

A, c, c, u, r, a, c, y, equals, start fraction, start text, t, r, u, e, space, p, o, s, i, t, i, v, e, s, space, plus, space, t, r, u, e, space, n, e, g, a, t, i, v, e, s, end text, divided by, start text, t, o, t, a, l, space, p, r, e, d, i, c, t, i, o, n, s, end text, end fraction

Accuracy is often unsuitable to use when there is a class imbalance in the data, because it's possible for a model to have high accuracy by predicting the majority class every time. In such a case, the model would score well, but it may not be a useful model.

Precision

Precision measures the proportion of positive predictions that are true positives. It is represented as:

$$\text{Precision} = \frac{\text{true positives}}{\text{true positives + false positives}}$$

$$\textit{Precision} = \frac{\text{true positives}}{\text{true positives + false positives}}$$

P, r, e, c, i, s, i, o, n, equals, start fraction, start text, t, r, u, e, space, p, o, s, i, t, i, v, e, s, end text, divided by, start text, t, r, u, e, space, p, o, s, i, t, i, v, e, s, space, plus, space, f, a, l, s, e, space, p, o, s, i, t, i, v, e, s, end text, end fraction

Precision is a good metric to use when it's important to avoid false positives. For example, if your model is designed to initially screen out ineligible loan applicants before a human review, then it's best to err on the side of caution and not automatically disqualify people before a person can review the case more carefully.

Recall

Recall measures the proportion of actual positives that are correctly classified. It is represented as:

$$\text{Recall} = \frac{\text{true positives}}{\text{true positives + false negatives}}$$

$$\textit{Recall} = \frac{\text{true positives}}{\text{true positives + false negatives}}$$

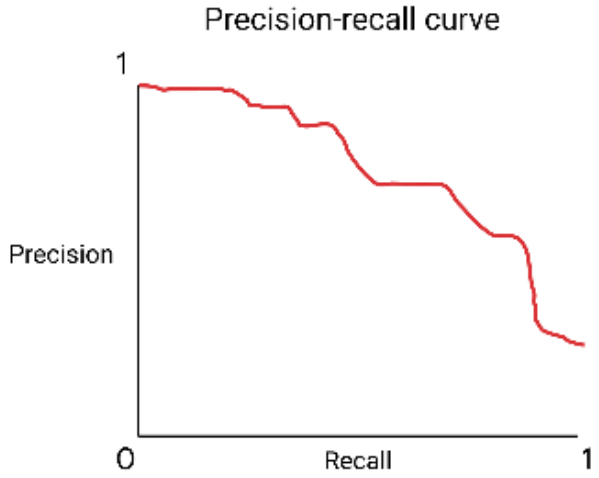
R, e, c, a, l, l, equals, start fraction, start text, t, r, u, e, space, p, o, s, i, t, i, v, e, s, end text, divided by, start text, t, r, u, e, space, p, o, s, i, t, i, v, e, s, space, plus, space, f, a, l, s, e, space, n, e, g, a, t, i, v, e, s, end text, end fraction

Recall is a good metric to use when it's important that you identify as many true responders as possible. For example, if your model is identifying poisonous mushrooms, it's better to identify all of the true occurrences of poisonous mushrooms, even if that means making a few more false positive predictions.

Precision-recall curves

A precision-recall curve is a way to visualize the performance of a classifier at different decision thresholds (a.k.a. classification thresholds). In the context of binary classification, a decision threshold is a probability cutoff for differentiating the positive class from the negative class. In most modeling libraries—including scikit-learn—the default decision threshold is 0.5 (i.e., if a sample's predicted probability of response is ≥ 0.5, then it's labeled "positive").

In a precision-recall curve, necessarily, as one metric increases, the other generally decreases, and vice versa. For example, suppose your model's decision threshold is very high: 0.98. Your model will not predict a positive class unless it's at least 98% sure. Your precision will therefore be very high—your model's predicted positives will likely all be true positives. But the model will probably catch very few actual positives because it's too strict, so the recall will be very low.



The same is true for the inverse. If your model casts a very wide net and predicts many positives, it will likely capture many true positives, but it will also mistakenly label many negative samples as positive. Thus, the model might have a perfect recall but a very poor precision. The art and the science is to find a model that maximizes both metrics and best suits your use case.

ROC curves

Receiver operating characteristic (ROC) curves are similar to precision-recall curves in that they visualize the performance of a classifier at different decision thresholds. However, this curve is a plot of the true positive rate against the false positive rate.

1. True Positive Rate: Equivalent/synonymous to recall

$$\text{True positive rate} = \frac{\text{true positives}}{\text{true positives + false negatives}}$$

start text, T, r, u, e, space, p, o, s, i, t, i, v, e, space, r, a, t, e, end text, equals, start fraction, start text, t, r, u, e, space, p, o, s, i, t, i, v, e, s, end text, divided by, start text, t, r, u, e, space, p, o, s, i, t, i, v, e, s, space, plus, space, f, a, l, s, e, space, n, e, g, a, t, i, v, e, s, end text, end fraction

2. False Positive Rate: The ratio between the false positives and the total count of observations that should be predicted as False

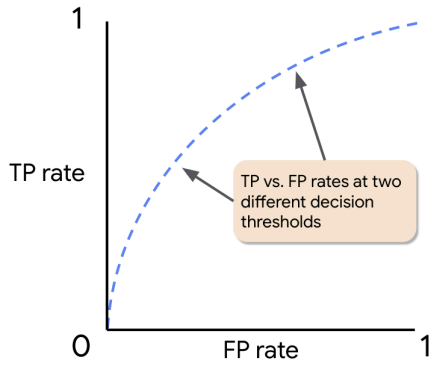
False positive rate=false positivesfalse positives + true negatives

False positive rate=

$$\frac{\text{false positives}}{\text{false positives + true negatives}}$$

start text, F, a, l, s, e, space, p, o, s, i, t, i, v, e, space, r, a, t, e, end text, equals, start fraction, start text, f, a, l, s, e, space, p, o, s, i, t, i, v, e, s, end text, divided by, start text, f, a, l, s, e, space, p, o, s, i, t, i, v, e, s, space, plus, space, t, r, u, e, space, n, e, g, a, t, i, v, e, s, end text, end fraction

For each point on an ROC curve, the horizontal and vertical coordinates represent the false positive rate and the true positive rate at the corresponding threshold.



The false positive rate and true positive rate change together over the different thresholds.

How does the curve appear for an ideal model? An ideal model perfectly separates all negatives from all positives, and gives all real positive cases a very high probability and all real negative cases a very low probability. In other words, on the graph above, it

would be in the upper-left-most corner. It would have a perfect TP rate (1) and a perfect FP rate (0).

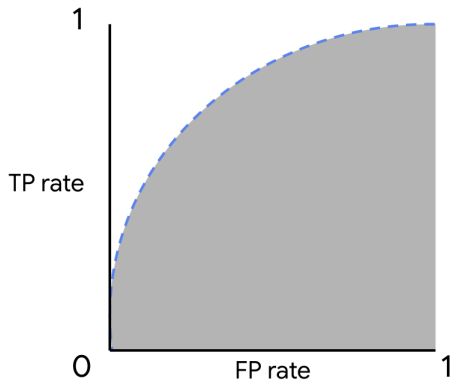
To contrast, now imagine starting from a decision threshold just above zero: 0.001. In this case, it's likely that all real positives would be captured and there would be very few—if any—false negatives, because for a model to label a sample "negative," its predicted probability must be < 0.001. The true positive rate would be ≈ 1, which is excellent, but its false positive rate would also be ≈ 1, which is very bad (refer to the formulas above). This decision threshold would result in TP/FP rates that would be at the upper-right-most area of the graph above.

To reiterate: graphically, the closer your model's TP/FP rate is to the top-left corner of the plot, the better the model is at classifying the data.

Remember, by default most modeling libraries use a decision threshold of 0.5 to separate positive predictions and negative predictions. However, there are some cases where 0.5 might not be the optimal decision threshold to use. For a demonstration of this topic applied to an actual model, refer to the Bonus section at the end of the Exemplar: Course 6 Waze project notebook.

AUC

AUC is a measure of the two-dimensional area underneath an ROC curve. AUC provides an aggregate measure of performance across all possible classification thresholds. One way to interpret AUC is to consider it as the probability that the model ranks a random positive sample more highly than a random negative sample. AUC ranges in value from 0.0 to 1.0. In the following example, the AUC is the shaded region below the dotted curve.



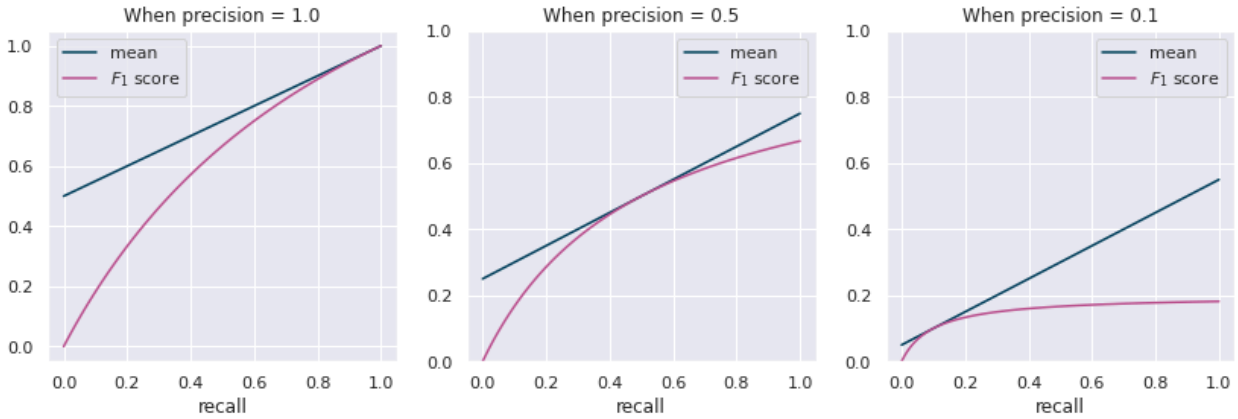
F_1 score

F_1 score is a measurement that combines both precision and recall into a single expression, giving each equal importance. It is calculated as:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

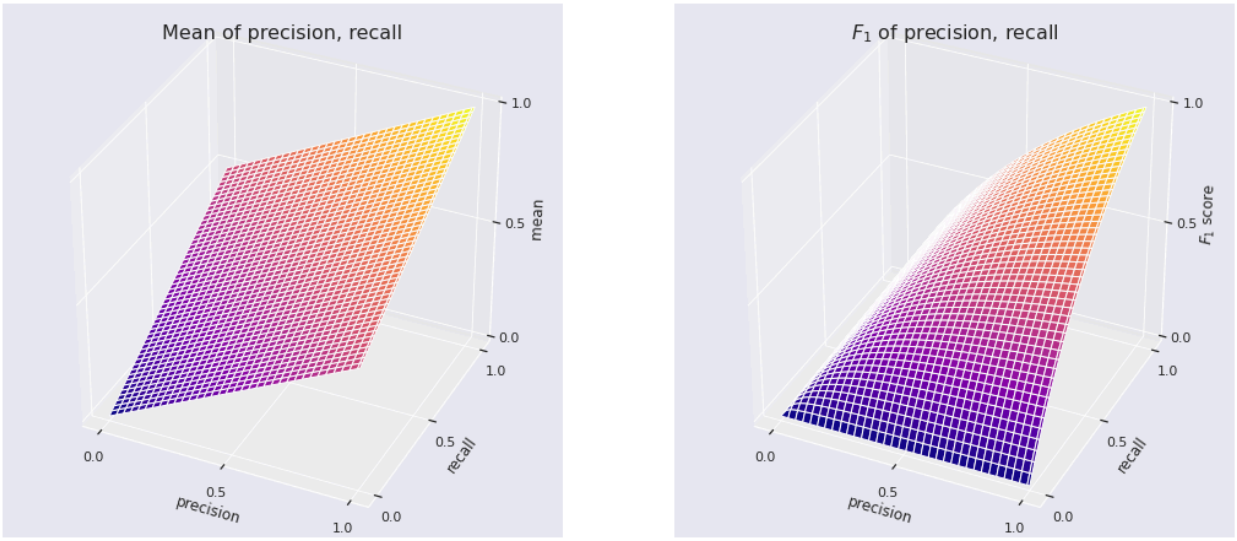
This combination is known as the harmonic mean. F_1 score can range [0, 1], with zero being the worst and one being the best. The idea behind this metric is that it penalizes low values of either metric, which prevents one very strong factor—precision or recall—from "carrying" the other, when it is weaker.

The following figure illustrates a comparison of the F_1 score to the mean of precision and recall. In each case, precision is held constant while recall ranges from 0 to 1.



The F_1 score never exceeds the mean. In fact, it is only equal to the mean in a single case: when precision equals recall. The more one score diverges from the other, the more F_1 score penalizes. (Note that you could swap precision and recall values in this experiment and the scores would be the same.)

Plotting the means and F_1 scores for all values of precision against all values of recall results in two planes.



While the coordinate plane of the mean is flat, the plane of the F_1 score is pulled further downward the more one score diverges from the other. This penalizing effect makes F_1 score a useful measurement of model performance.

F_β score

What if you still want to capture both precision and recall in a single metric, but you consider one more important than the other? There's a metric for that! It's called F_β score (pronounced F-beta). In an F_β score, β is a factor that represents how many times more important recall is compared to precision. In the case of F_1 score, $\beta = 1$, and recall is therefore 1x as important as precision (i.e., they are equally important). However, an F_2 score has $\beta = 2$, which means recall is twice as important as precision; and if precision is twice as important as recall, then $\beta = 0.5$. The formula for F_β score is:

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}}$$

You can assign whatever value you want to β. However, in scikit-learn, while most modules have built-in F_1 and F_β scorers, some modules may only have an F_1 scorer and require you to define your own scoring function if you want to set a custom β value.

Key takeaways

- There are many metrics used to evaluate binary classification models, including accuracy, precision, recall, ROC curve, AUC, and F score.
- Accuracy captures overall model performance, while precision measures true positives and recall measures false negatives.
- F_β score combines precision and recall into a single metric.
- You can set β to any value you wish. β is a factor that determines how many times more important recall is than precision in the score.
- F_1 score is simply an F_β score where $\beta = 1$. For F_1 score, precision is equally important to recall.

Tab 7

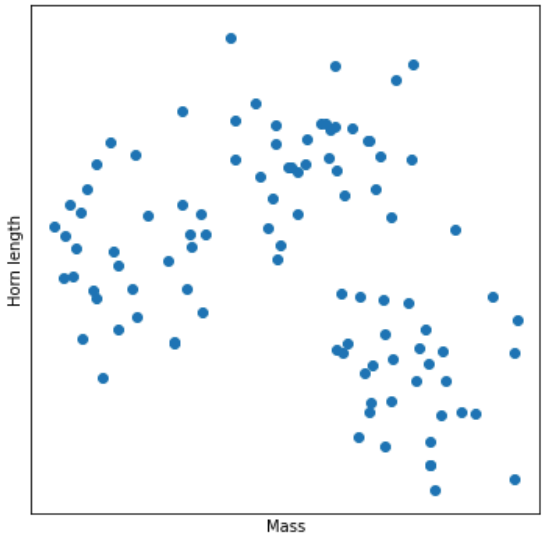
More about K-means

Previously, you learned about the K-means algorithm, an unsupervised learning technique used to cluster unlabeled data. You have a foundational understanding of its underpinning theory. In this reading, you will examine this methodology in greater depth by studying how the algorithm behaves when different *k* values are used to cluster two-dimensional data. You will gain a deeper understanding of K-means by comparing cases where it works well, and those where it works poorly. Being aware of the strengths and limitations of this methodology will enable you to use it appropriately and effectively as a data professional.

Consider the data

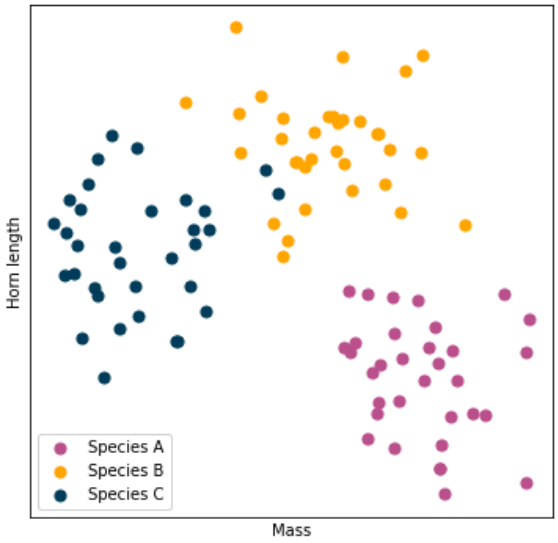
Suppose you're presented with data for 100 samples selected from three species of horned beetles.

The data compares total body mass to horn length. Because it's two-dimensional, you can plot it and examine the results to identify patterns or trends.



What do you notice about the data in the scatter plot above? Does it fall into groups? Can you tell that three species are represented in this data?

What if we color the same data by species?



Hopefully you were able to discern similar clusters on your own. Notice that the points in this scatter plot are loosely grouped into three clusters. In this case, each color represents a different species of beetle. You can use K-means to cluster this data.

Cluster with K-means

Before continuing, it's important to note that in this illustrative case:

1. You know there are three species of beetles.
2. By using a scatter plot, you can verify that they are clustered into three groups.

Remember, you will not always know how many clusters you *should* have, and you probably won't be able to visualize your data so easily because it will likely have more than three features (i.e., dimensions). We use this example because it is effective for teaching concepts to learners.

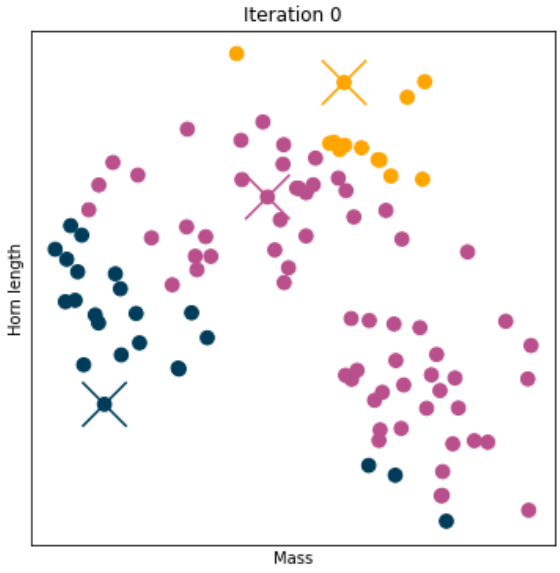
Modeling: *k* = 3

Since you know there are three clusters, you know to set *k* equal to three when you build your model. Recall the steps of the K-means algorithm:

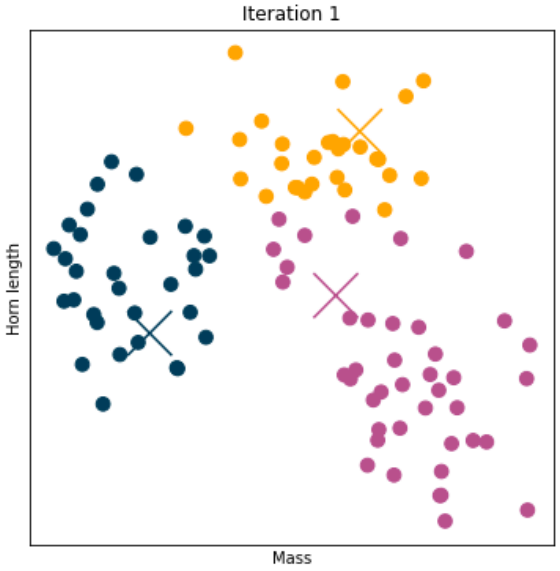
1. Randomly place centroids in the data space.
2. Assign each point to its nearest centroid.
3. Update the location of each centroid to the mean position of all the points assigned to it.
4. Repeat steps 2 and 3 until the model converges (i.e., all centroid locations remain unchanged with successive iterations).

Now, consider these steps as the K-means algorithm iterates over the data.

Iteration 0: The centroids are placed at random, and the points are assigned to their nearest centroid. (Note: Centroids are indicated by "X".)

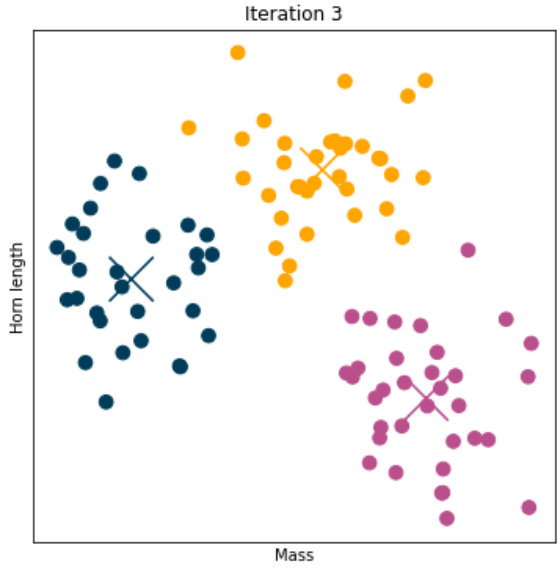


Iteration 1: Update centroid locations, reassign points to their nearest centroid.



Iteration 2: Repeat.

Iteration 3: Repeat.



Iteration 4: Repeat.

Stop.

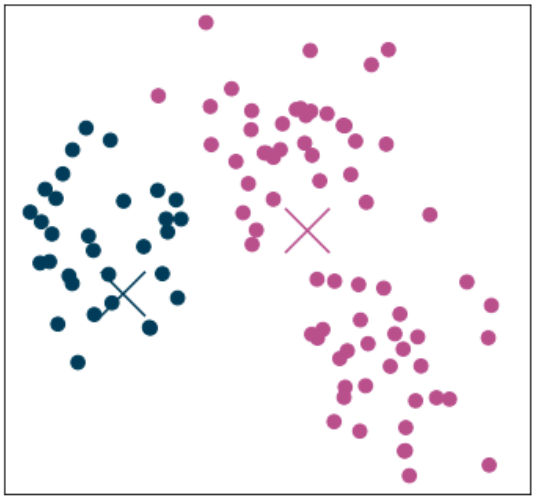
Why stop here? Well, notice that there is no meaningful difference between the results of iteration 3 and iteration 4. The algorithm has converged (the centroids have stopped moving and cluster assignment has stabilized). The three points marked with squares were assigned to the wrong clusters by the model. Except for these points, the cluster assignments of the data were correct after just two iterations, and the centroid locations converged after three. K-means is both effective and efficient in clustering this data.

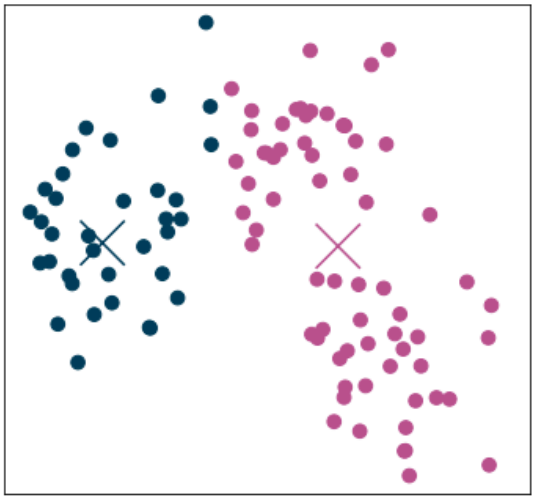
Note that in the case above, the centroids may still be moving after iteration 4, but if so, all movement is below a convergence tolerance that can be set by the modeler. The default in scikit-learn is 0.0001. If all centroid locations move less than this distance, the model is declared converged.

Modeling: *k* = 2

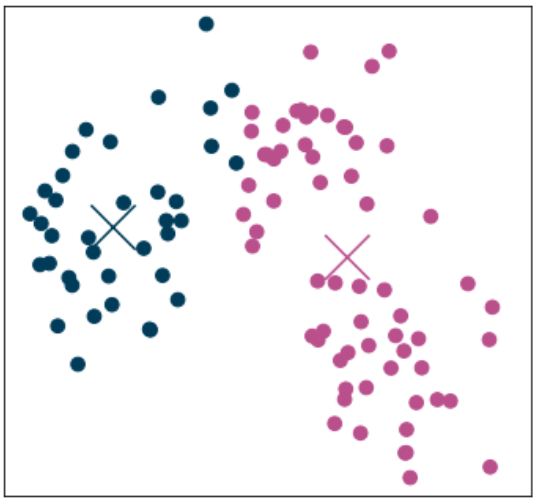
Suppose you didn't know how best to cluster your data, and you decided to set the value of *k* to two. Note how the clustering converges in this case:


Iteration 0

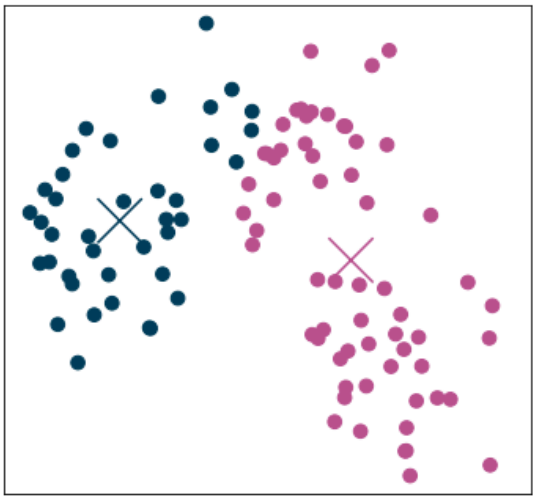

Iteration 1


Iteration 2

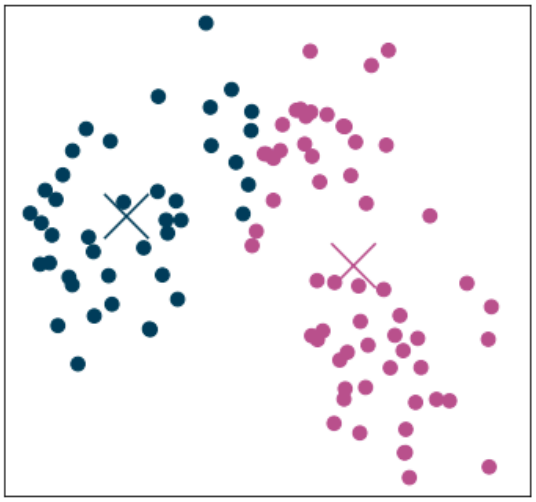
Iteration 3

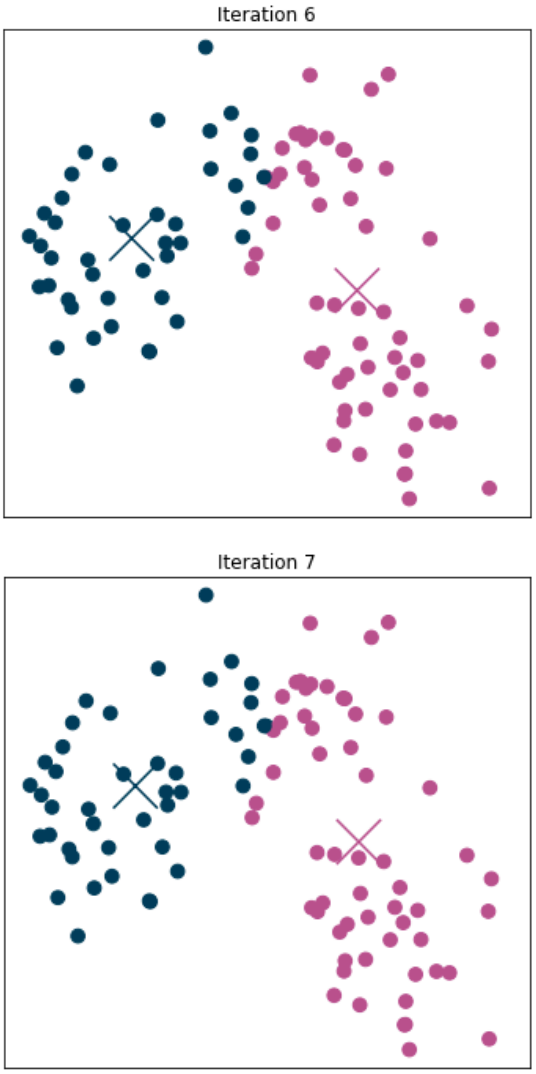


Iteration 4



Iteration 5





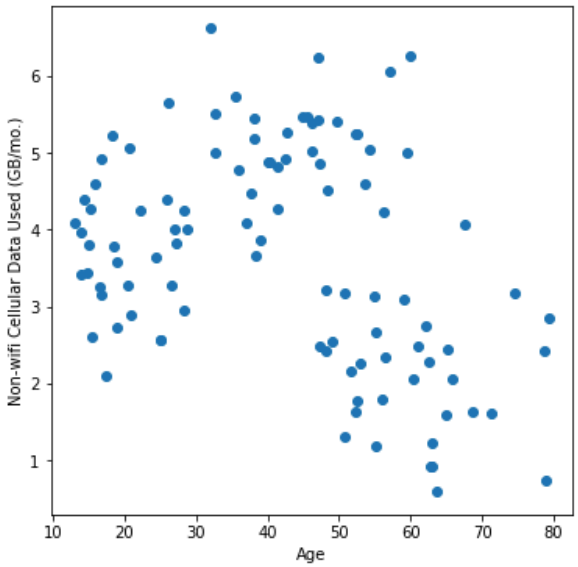
Not only did it take six iterations to converge, but the data in the top cluster was split, resulting in class assignment that, in this case, you know to be incorrect (because there are three species of beetle). This is an example of what can happen when you don't use the best value for *k.* If you were to set *k* to a value greater than three, the algorithm could similarly split the data into unintuitive clusters.

When there are no correct answers

In the previous examples, you knew that there were three different kinds of beetles, and therefore to set *k* to three. The purpose of using this labeled data for an unsupervised learning demonstration was to illustrate that K-means can effectively group data.

However, more often, you won't have any class labels to determine how "good" your model is.

Suppose you have the same data points, but they represent something else—in this new case, mobile phone owners plotted by age and cellular (non-wifi) data usage.



You may want to perform a market segmentation analysis on this data. In this scenario, there is no "correct" answer. You cannot verify your model's clustering against a baseline truth. It's possible that you could make a case to use two, three, four, or more clusters. You might not have any idea how many to choose. There are tools to help you make this decision when you cannot visualize your data or don't otherwise know how many clusters you should have, which you'll learn later in this lesson.

A note on K-means++

Earlier in this course, you learned the importance of running K-means multiple times with different initial positions of the centroids to help avoid using a model that gets stuck in local minima. Fortunately, most machine learning packages have improved implementations of K-means that make it easier for you by removing this requirement.

In scikit-learn, this implementation is called K-means++. K-means++ still randomly initializes centroids in the data, but it does so based on a probability calibration. Basically, it randomly chooses one point within the data to be the first centroid, then it uses other data points as centroids, selecting them pseudo-randomly. The probability that a point will be selected as a centroid increases the farther it is from other centroids. This helps to ensure that centroids aren't initially placed very close together, which is when convergence in local minima is most likely to occur.

K-means++ is the default implementation when you instantiate K-means in scikit-learn. If you don't want to use this implementation and would prefer to start with truly random centroids, you can change this by setting the "init" parameter to "random", but rarely would you want to do this.

Key takeaways

When using K-means to model your data, it's important to understand how the methodology works to cluster data. If you know the most common ways by which K-means can arrive at both good and bad clustering results, it will help you understand how to choose the best value for *k* and ultimately to make better decisions when building models as a data professional.

Tab 8

Clustering beyond K-means

As you now know, K-means is a powerful and straightforward way to group data based on its proximity to other data. However, as with all models, K-means has its limitations. This reading will review the strengths of K-means and also demonstrate some of its weaknesses. Additionally, you'll learn about two additional clustering methodologies to explore as alternatives:

- DBSCAN
- Agglomerative clustering

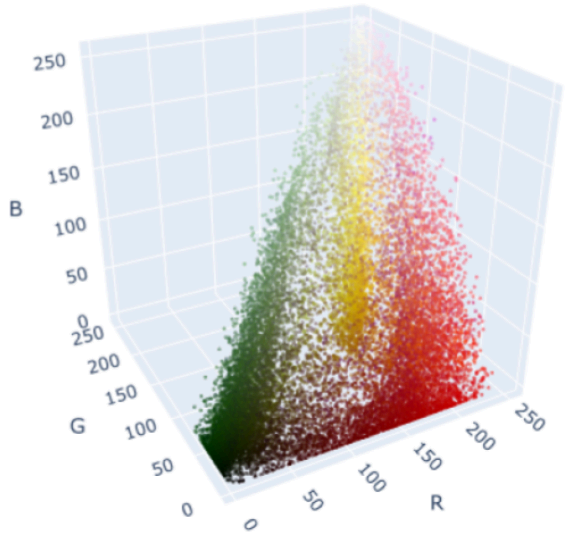
The purpose is not to make you an expert in clustering, but rather to give you a map of the terrain that you can use to guide you if you'd like to learn more about this branch of unsupervised learning.

Why use other algorithms?

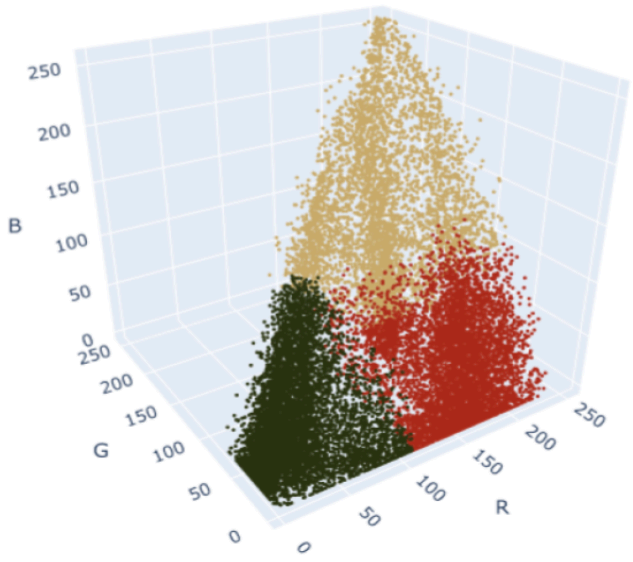
In the K-means demonstration video, you saw how the K-means algorithm clustered the pixels of a photograph of some tulips. Recall that the color of every pixel is represented by a vector with three values, each with a range of [0, 255]. These three values correspond to the amounts of red, green, and blue (RGB) that are combined to make the color of each pixel. The RGB values can be plotted in a three-dimensional space in order to visualize how the colors in the photograph are related to each other.

In this way, you could go from this:

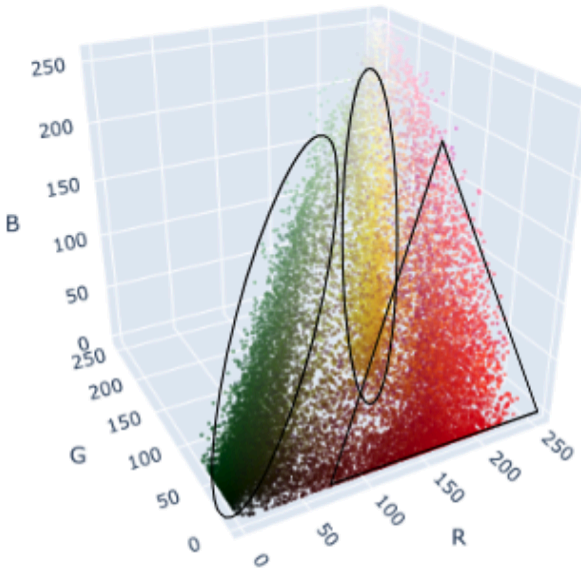
To this:



From this graph, it is clear that all of the pixels fall within a space that resembles a lopsided pyramid. In this same demonstration, the pixels were grouped into three clusters using K-means. When the RGB values for each pixel were replaced with those of its nearest centroid, the results were this:



Perhaps you were thinking that you would have clustered this data differently. After all, there do appear to be denser areas of points along the vertices of the pyramid:



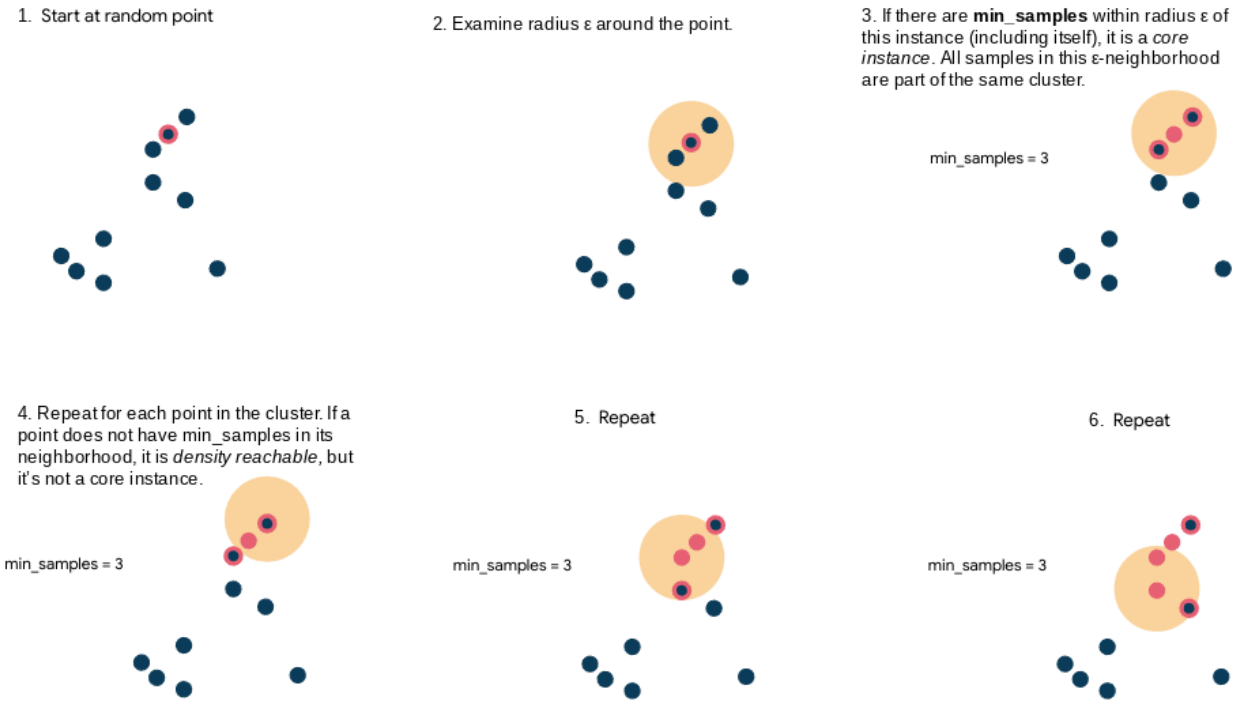
So why were the K-means clusters so boxy? Well, this is because K-means works by minimizing intercluster variance. In other words, it aims to minimize the distance between points and their centroids. This means that K-means works best when the

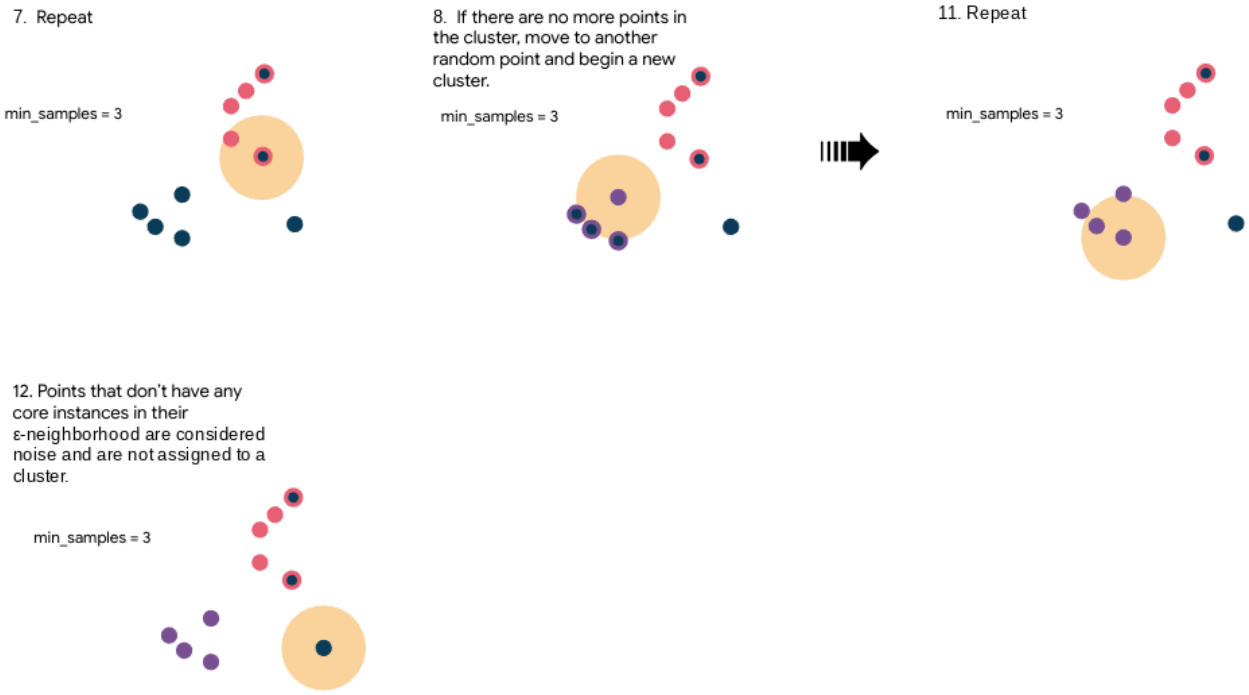
clusters are round. If you aren't satisfied with the way K-means is clustering your data, don't worry, there are many other clustering methods available to choose from.

DBSCAN

DBSCAN stands for density-based spatial clustering of applications with noise. Instead of trying to minimize variance between points in each cluster, DBSCAN searches your data space for continuous regions of high density. Here's how it works.

Note: You can also find text alternative versions of these conversations in the DBSCAN addendum transcript.





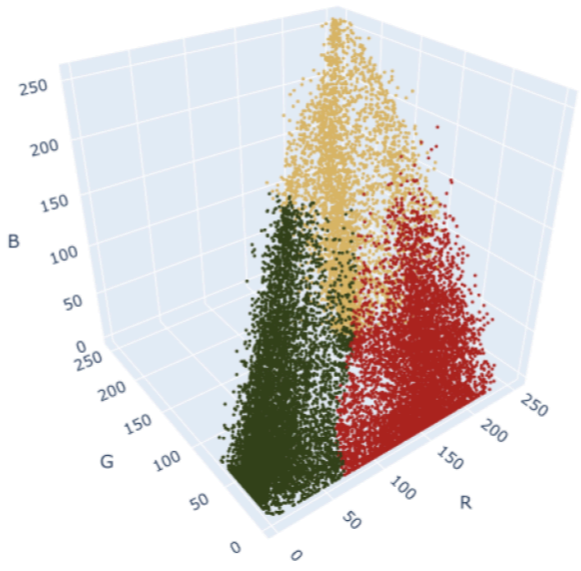
Hyperparameters

The most important hyperparameters for DBSCAN in scikit-learn are:

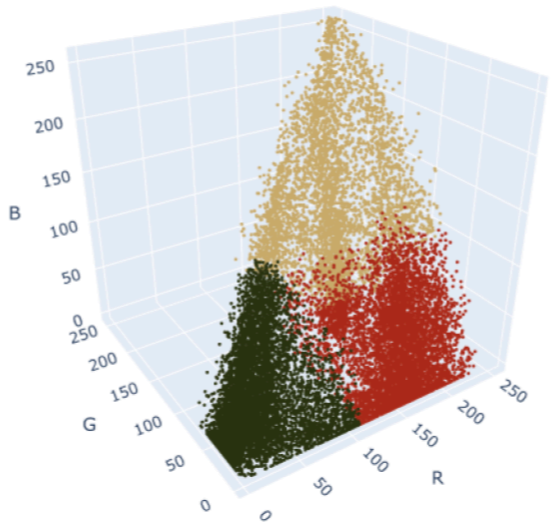
- eps: Epsilon (ε) - The radius of your search area from any given point
- min_samples: The number of samples in an ε-neighborhood for a point to be considered a core point (including itself)

Since DBSCAN is designed to find clusters based on density, the shape of the cluster isn't as important as it is for K-means. Here's what the tulips data looks like when clustered using DBSCAN compared to K-means:

DBSCAN



K-means

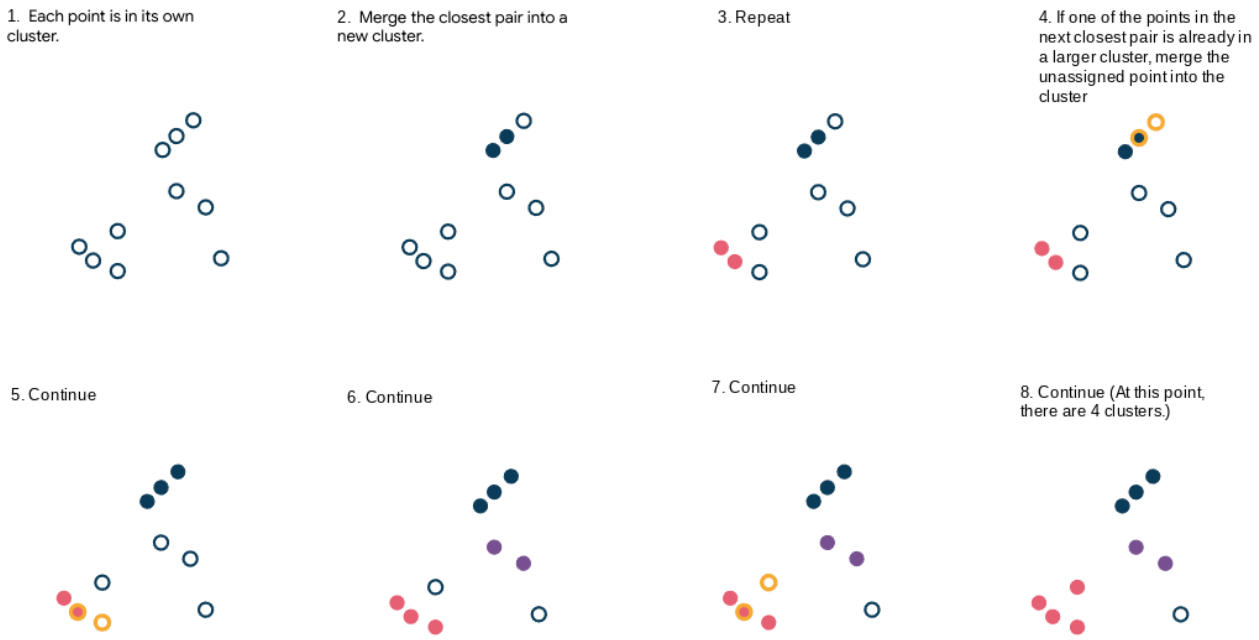


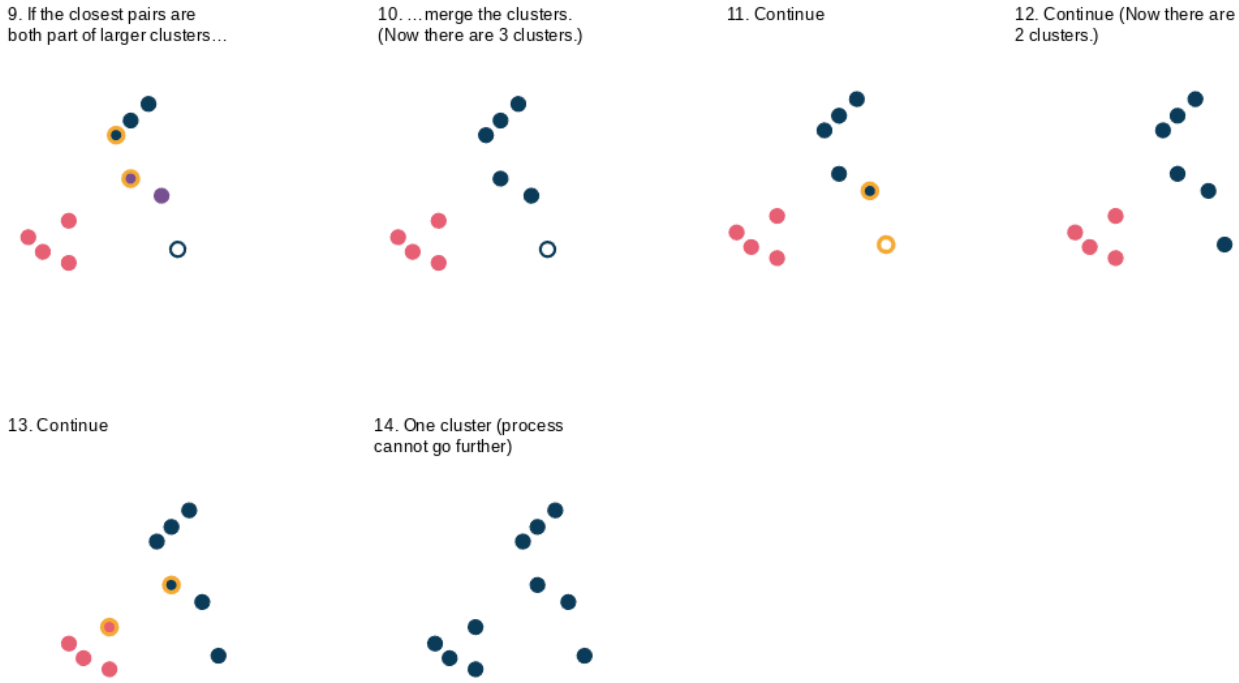
Notice that the clusters aren't as block-like as they were for K-means, and they correspond with the vertices of the pyramid a lot more closely than K-means. Also, of course, different clustering arrangements result in different colors for each of the three centroids.

Agglomerative clustering

Another way to cluster data is by using a technique called agglomerative clustering. Agglomerative clustering works by first assigning every point to its own cluster, then progressively combining clusters based on intercluster distance. Here's how it works.

Note: You can also find text alternative versions of these conversations in the [Agglomerative clustering transcript].





Agglomerative clustering requires that you specify a desired number of clusters or a distance threshold, which is the linkage distance (explained further in the next section) above which clusters will not be merged. If you do not specify a desired number of clusters, then the distance threshold is an important parameter, because without it the model would converge into a single cluster every time.

Linkage

There are different ways to measure the distances that determine whether or not to merge the clusters. This is known as the linkage. Here are some of the most common.

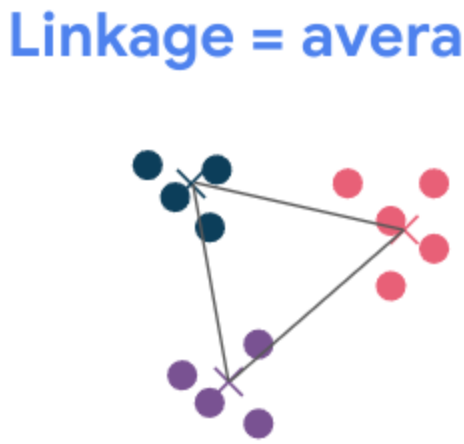
- Single: The minimum pairwise distance between clusters

Linkage = single

- Complete: The maximum pairwise distance between clusters

Linkage = complete

- Average: The distance between each cluster’s centroid and other clusters’ centroids.



- Ward: This is not a distance measurement. Instead, it merges the two clusters whose merging will result in the lowest inertia.

Note that these linkage strategies aren't specific just to agglomerative clustering. You'll encounter them in many other clustering methodologies if you choose to continue learning about them, and even beyond clustering in other areas of data science.

When does it stop?

The agglomerative clustering algorithm will stop when one of the following conditions is met:

1. You reach a specified number of clusters.
2. You reach an intercluster distance threshold (clusters that are separated by more than this distance are too far from each other and will not be merged).

Hyperparameters

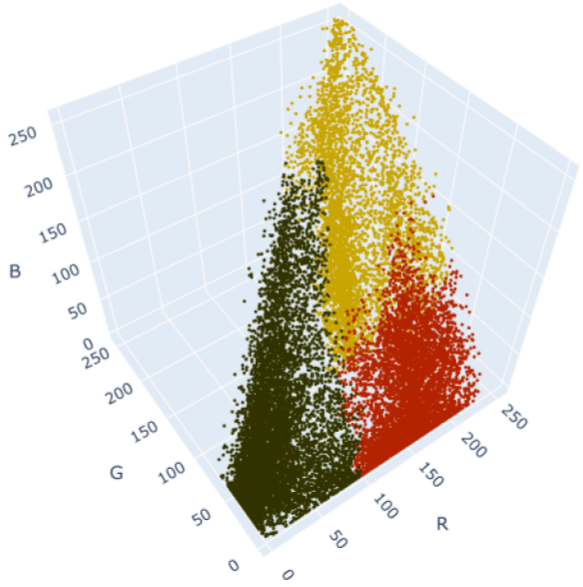
There are numerous hyperparameters available for agglomerative clustering in scikit-learn. These are some of the most important ones:

- n_clusters: The number of clusters you want in your final model

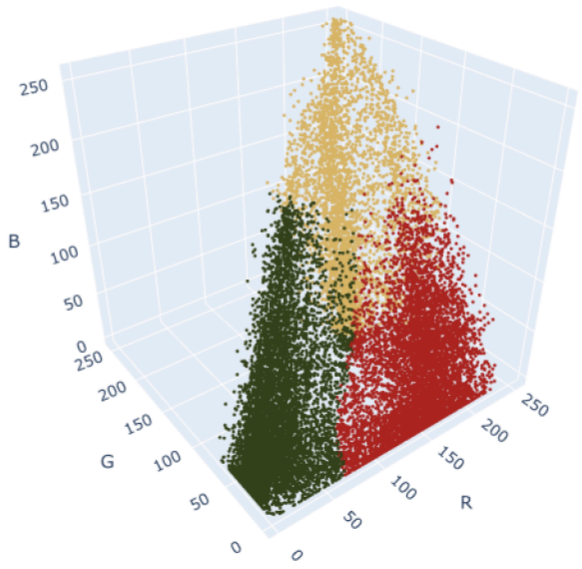
- linkage: The linkage method to use to determine which clusters to merge (as described above)
- affinity: The metric used to calculate the distance between clusters. Default = euclidean distance.
- distance_threshold: The distance above which clusters will not be merged (as described above)

Agglomerative clustering can be very effective. It scales reasonably well, and it can detect clusters of various shapes. Compare how agglomerative clustering performs on the tulip data compared to DBSCAN and K-means:

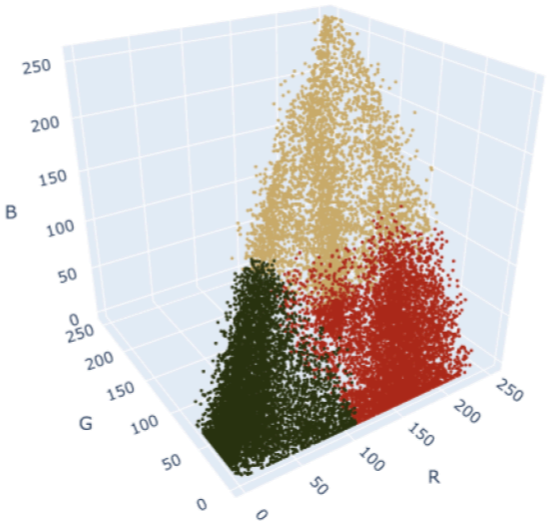
Agglomerative clustering:



DBSCAN



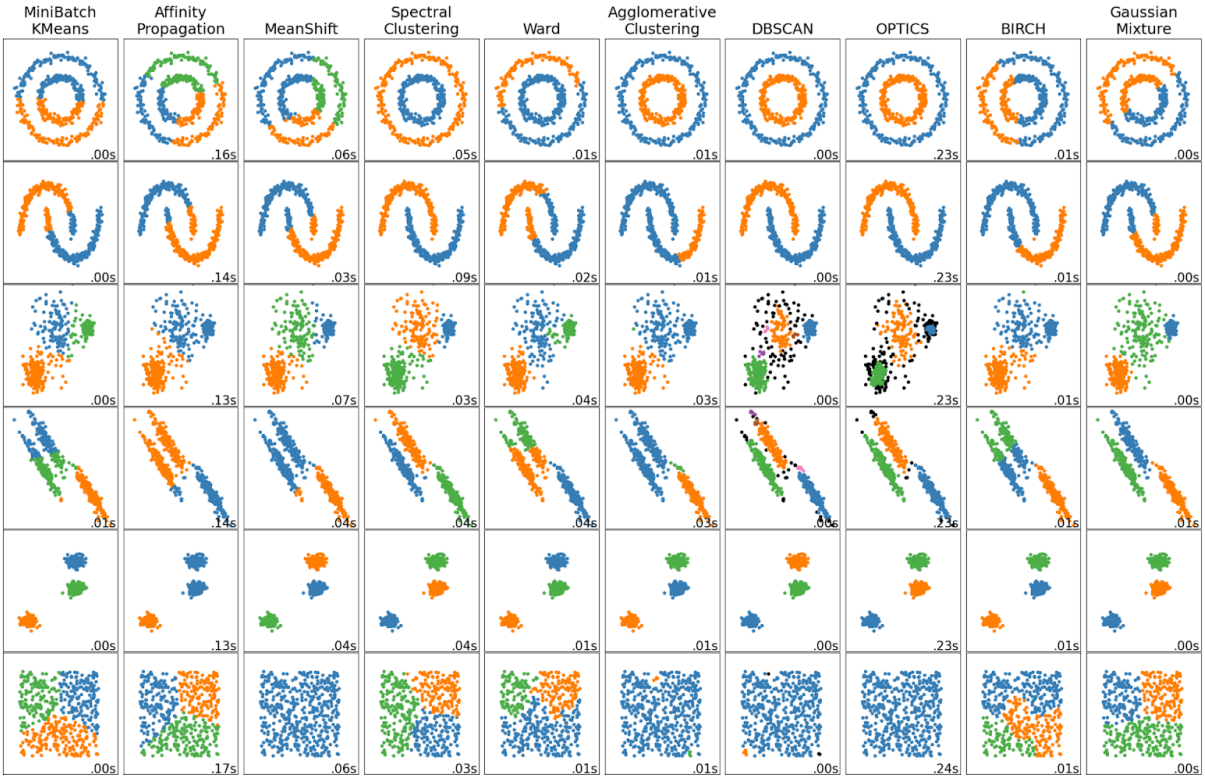
K-means



Agglomerative clustering gives even more definition to the data clustered along the vertices of the pyramid, which most closely represents the clusters that appear to the eye.

Other clustering algorithms

There are many other ways to cluster data than what is covered here. Scikit-learn's documentation provides a helpful reference that illustrates some of the strengths and weaknesses of each methodology by running them on a series of toy datasets.



As always, feel free to explore some of these algorithms on your own!

Key takeaways

K-means is a simple yet powerful clustering algorithm, but it's not the only one. Depending on the nature of the problem you're trying to solve, other algorithms might outperform it. DBSCAN and agglomerative clustering are two methodologies that are accessible to novices and effective, and they can cluster your data even when the clusters themselves have unusual shapes.

Tab 9

More about inertia and silhouette coefficient metrics

You know that the evaluation metrics you used for supervised learning models don't apply to unsupervised learning models. This is because unsupervised learning model results cannot be categorized as "correct" or "incorrect." While supervised learning models use predictor variables to predict a defined target variable, unsupervised learning methods have metrics that seek an underlying structure within the data.

Clustering models are a type of unsupervised learning that do this by grouping observations together. Data professionals often use inertia and silhouette scores to evaluate their clustering models and help them determine which groupings make sense. This reading reviews these concepts and examines them in greater detail.

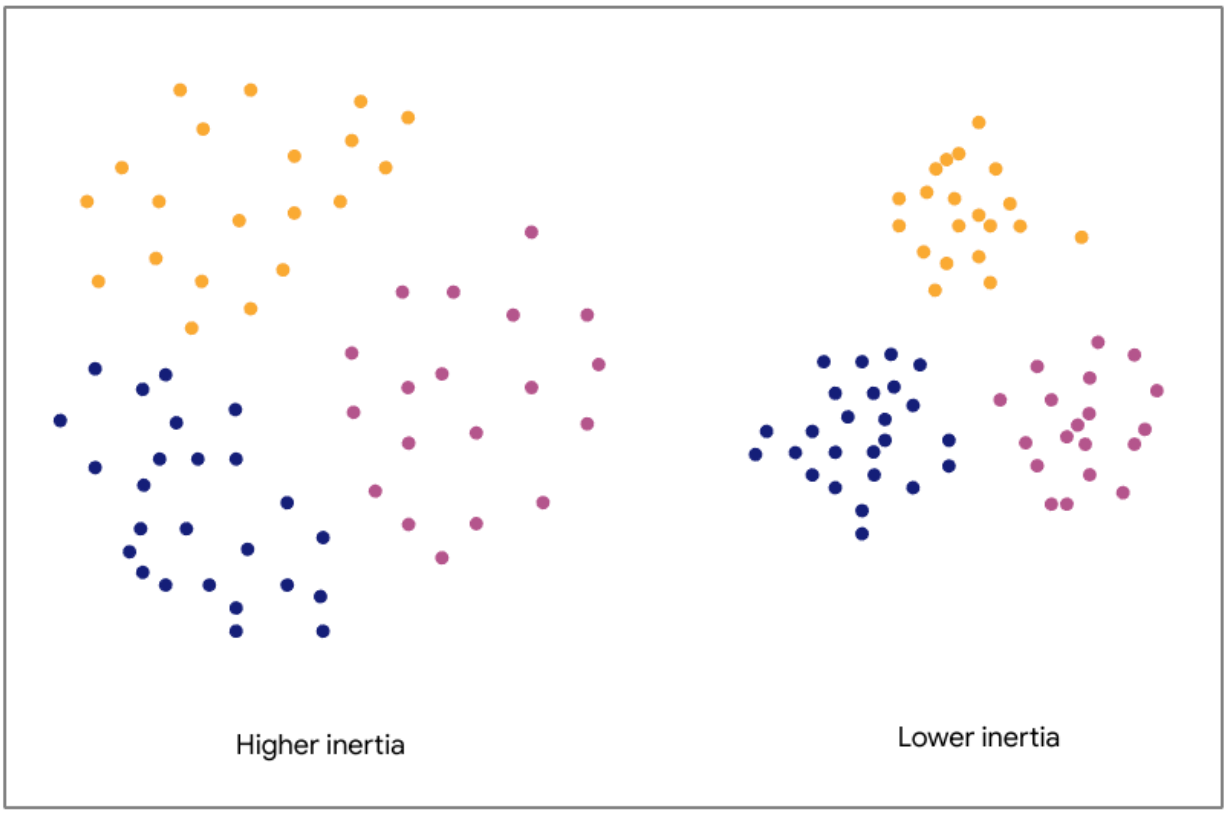
Inertia

Inertia is a measurement of intracluster distance. It indicates how compact the clusters are in a model. Specifically, inertia is the sum of the squared distance between each point and the centroid of the cluster that it's assigned to. It can be represented by this formula, where:

- n = the number of observations in the data,
- x_i = the location of a particular observation,
- C_k = the location of the centroid of cluster *k*, which is the cluster to which point x_i is assigned.

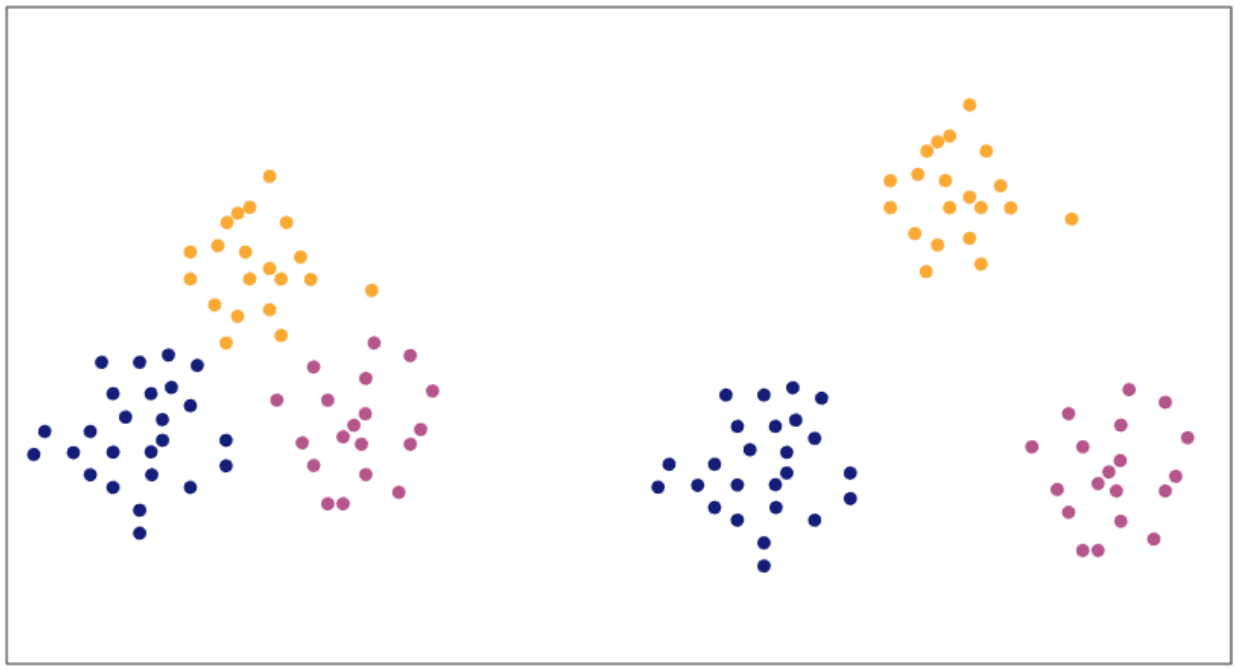
$$\text{Inertia} = \sum_{i=1}^{n} (x_i - C_k)^2$$

The greater the inertia, the greater the distances between points and their centroids, which means the points within each cluster are farther apart from each other. In the following figure, the three clusters on the left have higher inertia than the three clusters on the right, because they are less compactly positioned around their respective centroids.



The three clusters on the left have higher inertia than the three on the right.

Note, however, that inertia only measures intracluster distance. Therefore, both of the clusterings in the figure below have the same inertia.



The three clusters on the left have the same inertia as the three on the right.

For the same dataset and the same number of clusters, lower inertia values are typically better than higher values, because low values indicate that points are closer together within their clusters.

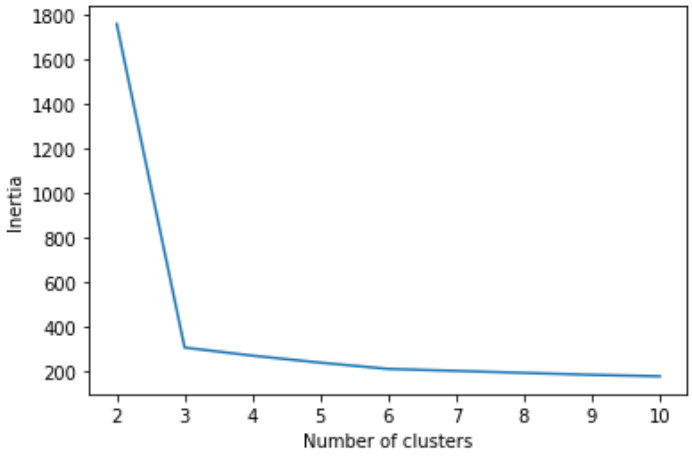
Why is it more meaningful for points in a dataset to be close together within their clusters? Well, notice that you're trying to make sense of data by identifying observations that are related to each other somehow. Clustering models do this by grouping points together based on their similarity. For some techniques, "similarity" is about the distance between points themselves (and so points that are close to each other belong to the same cluster). With others, it's about the distance between points and a cluster center (such that points that are close to the same center belong to the same cluster)—like K-means. In both cases, observations that are closer together are assumed to be more similar to each other. In other words, a tighter cluster could be indicative of greater similarity between the real-world observations that are represented by those data points.

Evaluating inertia

Inertia is a useful metric to determine how well your clustering model identifies meaningful patterns in the data. But it's generally not very useful *by itself.* If your model has an inertia of 53.25, is that good? It depends. The measurement becomes meaningful when it's compared to the inertia values and *k* values of other models on the same data. As you increase the number of clusters *(k)*, the inertia value will drop, but there comes a point where adding more clusters will have only small changes in inertia. And it's this transition that we need to detect.

The elbow method

The elbow method is a great way to find this point of transition. It's a way to help decide which clustering gives the most meaningful model of your data. It uses a line plot to visually compare the inertias of different models. With K-means models, this is done as a comparison between different values of *k.* Here's an example:



This plot compares the inertias of nine different K-means models—one for each value of *k* from two through 10. It's clear that inertia begins very high when the data is grouped into two clusters. The three-cluster model, however, has much lower inertia, creating a steep negative slope between two and three clusters. After that, the rate of inertial decline slows down dramatically, as indicated by the much flatter line in the plot.

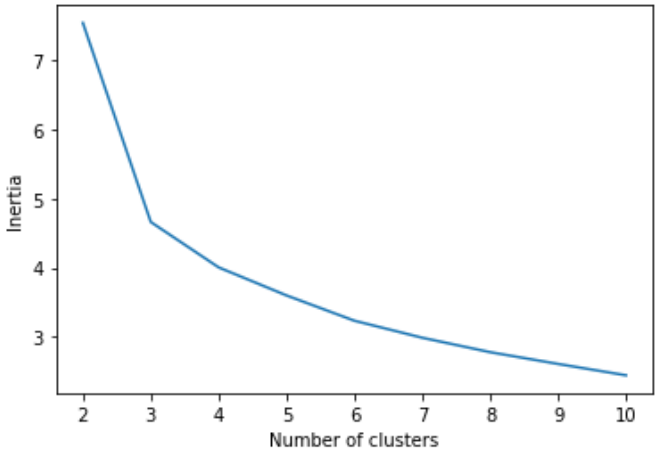
To use the elbow method to evaluate a model, you want to find the part of the curve that looks like an elbow—the sharpest bend in the curve. That's usually the model that will

give you the most meaningful clustering of your data, because if inertia is dropping significantly with added clusters, it means distance between points and their centroids is shortening significantly. This implies denser clusters and thus more similarity between points in a given cluster. But when the drop in inertia is very minor, the distance between points and their centroids isn't changing much, while your model is becoming more complex due to the additional cluster. Adding clusters is not capturing real structure in the data. A model with k = 1,000 will have low inertia, but is it any good? What meaning can you take away from 1,000 clusters?

Remember, you want inertia to be low, but if you add more and more clusters with only minimal improvement to inertia, you're only adding complexity without capturing real structure in the data.

There's not always an obvious elbow.

In the last example, the elbow was very clear. Often it won't be so obvious. Consider this, for example:



In this case, it seems that the elbow occurs at the three-cluster model, but there's still a considerable decline in inertia from three to four clusters. There's not necessarily a "correct" answer. It might be worth doing some analysis on the cluster assignments of both models to check for yourself which is more meaningful. There are also other tools at your disposal to help you decide.

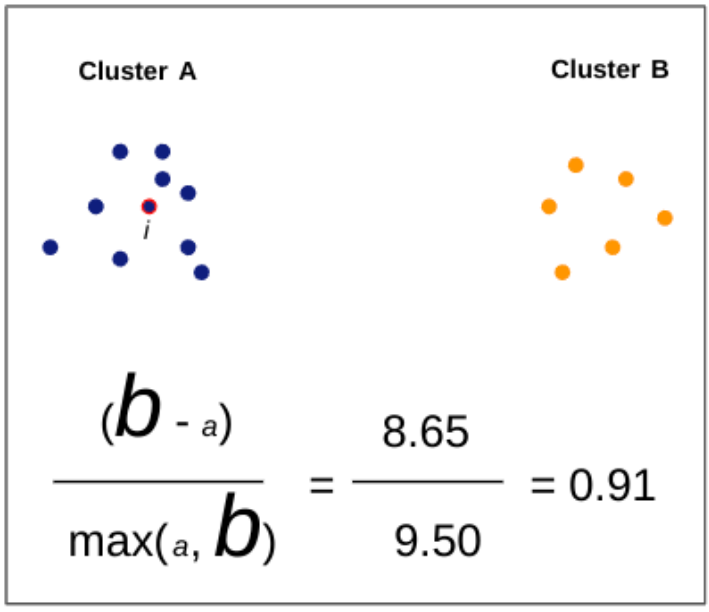
Silhouette analysis

One of these tools is a silhouette analysis. A silhouette analysis is the comparison of different models' silhouette scores. To calculate a model's silhouette score, first, a silhouette coefficient is calculated for each instance in the data. An instance's silhouette coefficient is defined by the following formula, where:

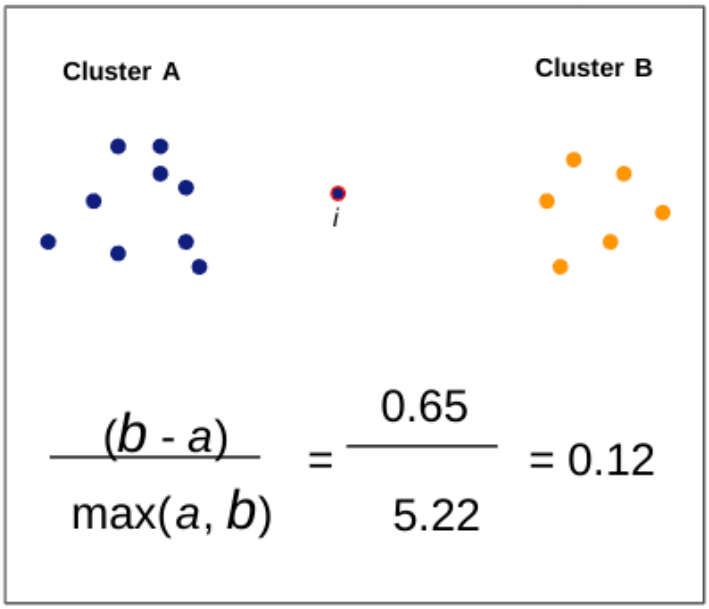
- a = the mean distance between the instance and each other instance in the same cluster
- b = the mean distance from the instance to each instance in the nearest other cluster (i.e., excluding the cluster that the instance is assigned to)
- $max(a, b)$ = whichever value is greater, a or b

$$\text{Silhouette coefficient} = \frac{(b-a)}{max(a,b)}$$

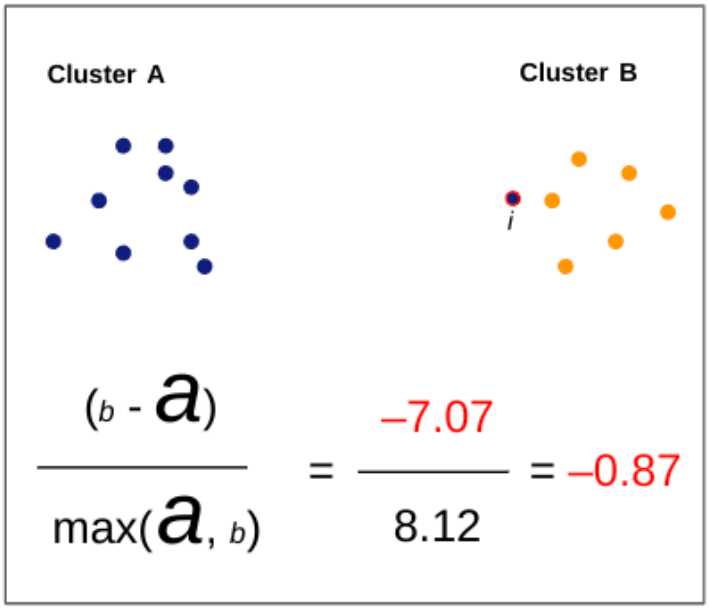
A silhouette coefficient can range between -1 and +1. A value closer to +1 means that a point is close to other points in its own cluster and well separated from points in other clusters.



A value closer to zero means that a point is between clusters.

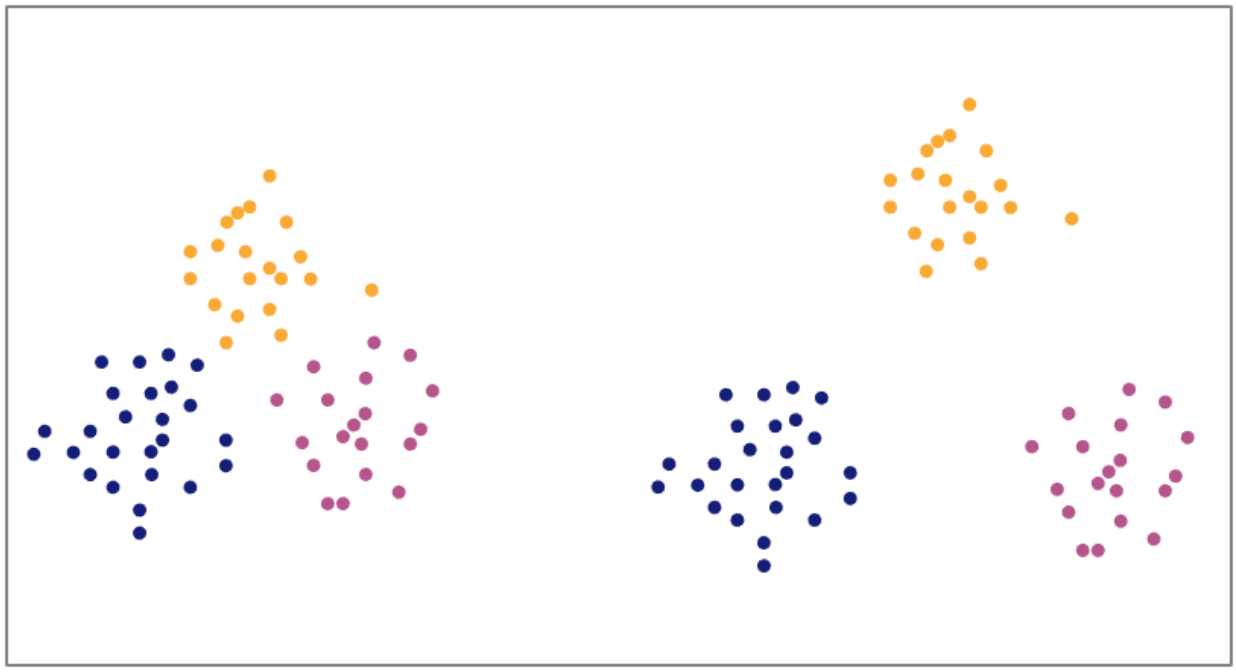


And a value closer to -1 means that a point is probably assigned to the wrong cluster, because it is closer to the points of another cluster than to the points in its own.



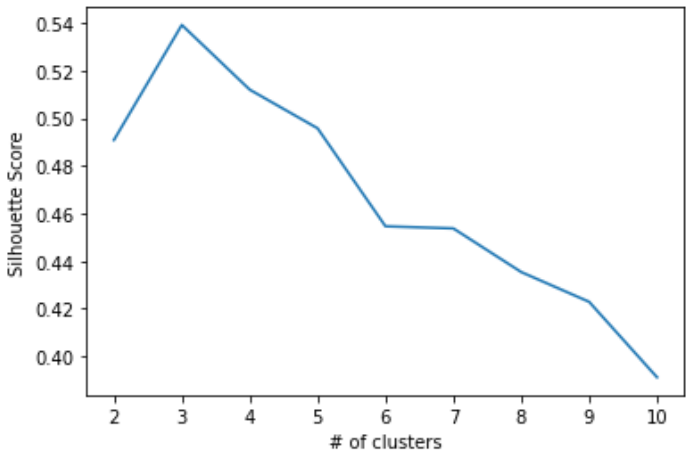
The silhouette score is the mean silhouette coefficient over all the observations in a model. The greater the silhouette score, the better defined the model clusters, because the points in a given cluster are closer to each other, and the clusters themselves are more separated from each other.

Note that, unlike inertia, silhouette coefficients contain information about both intracluster distance (captured by the variable *a*) and intercluster distance (captured by the variable *b*).



The points in the three clusters on the left have lower silhouette coefficients than the points in the three on the right.

As with inertia values, you can plot silhouette scores for different models to compare them against each other.



In this example, it's evident that a three-cluster model has a higher silhouette score than any other model. This indicates that this model results in individual clusters that are tighter and more separated from other clusters, more so than any other model tried. Based on this diagram, the data is probably best grouped into three clusters.

Key takeaways

Inertia and silhouette score are useful metrics to help determine how meaningful your model's cluster assignments are. Both are especially helpful when your data has too many dimensions (features) to visualize in 2-D or 3-D space. Use these metrics together to help inform your decision on which model to select.

Inertia:

- Measures intracluster distance
- Equal to the sum of the squared distance between each point and the centroid of the cluster that it's assigned to
- Used in elbow plots
- All else equal, lower values are generally better

Silhouette score:

- Measures both intercluster distance and intracluster distance
- Equal to the average of all points' silhouette coefficients
- Can be between -1 and +1 (greater values are better)

Tab 10

Explore decision trees

Explore decision trees

As you know, tree-based learning is one of the most effective machine learning techniques that are currently used in industry today. Many different algorithms use a tree-based architecture to make their predictions. The decision tree is the basic building block of these algorithms, as well as a powerful predictive algorithm in itself. Data professionals rely on decision trees as powerful tools that enhance modeling decisions. In this reading, you will take a deep dive into decision trees, how they are structured, how they work, and how they are built.

What is a decision tree?

Decision trees are a flowchart-like structure that uses branching paths to predict the outcomes of events, the probability of certain outcomes, or to reach a decision. They can be used for classification problems, where a specific class or outcome is predicted—like whether or not a sports team will win a game. They can also be used for regression problems, where a continuous variable is predicted—like the price of a car. This reading focuses on classification trees, but in both cases, the models depend on the same underlying decision process.

Decision trees have been used for decades for analyzing problems. However, technological advances have made it so decision trees can be created by computers, offering much deeper and more accurate analysis than humans alone could ever achieve.

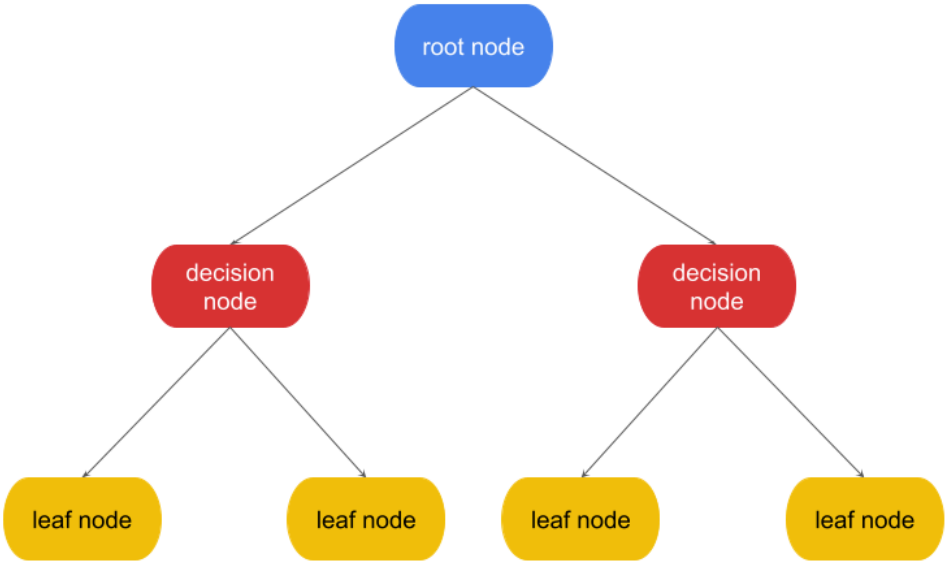
Note: All examples in this reading depict nodes that split into just two child nodes, because this is how they are implemented in modeling libraries, including scikit-learn. While it's theoretically possible to split each node into three, four, or even more new groups (as depicted previously in the "Shall I play soccer?" example), this is an

impractical approach because of its computational complexity. A two-level binary split is functionally equivalent to a single-level three-way split, but much simpler in terms of computational demand.

The structure of a classification tree

Decision trees only resemble actual trees if you flip them upside down, because they start with the root at the top and grow downward so the “leaves” are at the bottom. Decision trees are made of nodes. Nodes are groups of samples. There are different types of nodes, depending on how they function in the tree. The first node in a decision tree is called the root node. The first split always comes off of the root node, which divides the samples into two new nodes based on the values they contain for a particular feature.

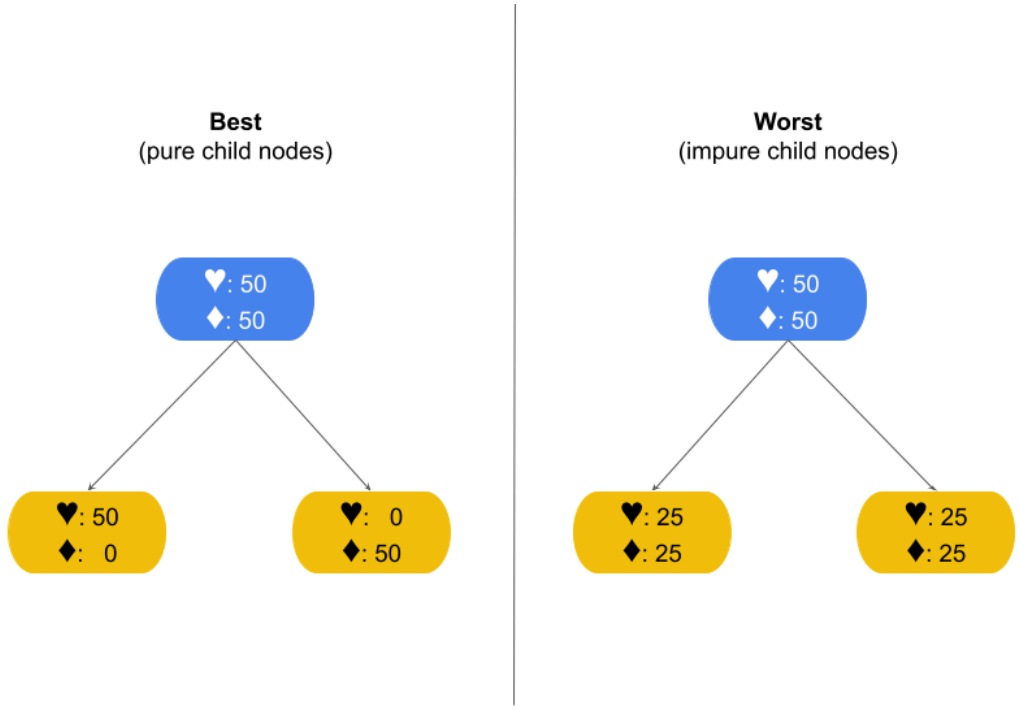
These two new nodes are referred to as child nodes of the root. A child node is any node that results from a split. The node that the child splits from is known as the parent node. Each of these two new child nodes in turn splits the data again, based on a new criterion. This process continues until the nodes stop splitting. The bottom-level nodes that do not split are called leaf nodes. All the nodes above the leaf nodes are called decision nodes, because they all make a decision that sorts the data either to the left or to the right.



Decision tree diagram. Root node at top. It splits into 2 decision nodes in the middle & those each split into 2 leaf nodes at bottom.

Decisions and splits

In a decision tree, the data is split and passed down through decision nodes until reaching a leaf node. A decision node is split on the criterion that minimizes the impurity of the classes in their resulting children. Impurity refers to the degree of mixture with respect to class. Nodes with low impurity have many more of one class than any other. A perfect split would have no impurity in the resulting child nodes; it would partition the data with each child containing only a single class. The worst possible split would have high impurity in the resulting child nodes; both of the child nodes would have equal numbers of each class.



2 trees. “Best” splits into child nodes of pure classes. “Worst” splits to equal numbers of each class in both nodes.

When building a tree and growing a new node, a set of potential split points is generated for every predictor variable in the dataset. An algorithm is used to calculate the “purity” of the child nodes that would result from each split point of each variable. The feature and split point that generate the purest child nodes are selected to partition the data.

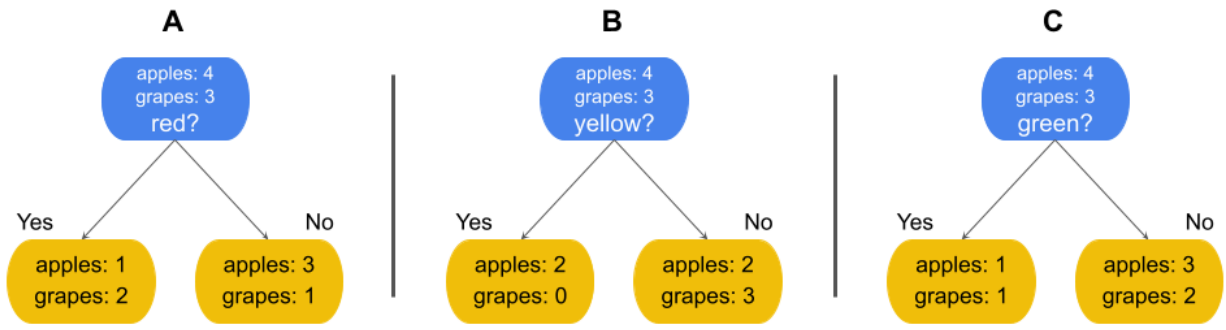
To determine the set of potential split points that will be considered for a variable, the algorithm first identifies what type of variable it is—such as categorical or continuous—and the range of values that exist for that variable.

Categorical variables

If the predictor variable is categorical, the decision tree algorithm will consider splitting based on category. Here’s a small dataset of fruits, to illustrate. The data contains samples of fruit that are either apples or grapes. It also has the fruits’ color (yellow, red, or green) and diameter in centimeters. Each of these variables affects the decision tree algorithm.

Color	Diameter (cm)	Fruit(target)
Yellow	3.5	Apple
Yellow	7	Apple
Red	2	Grape
Red	2.5	Grape
Green	4	Grape
Green	3	Apple
Red	6	Apple

First, the algorithm will consider splitting based on the categorical variable, color. Since there are three categories, three options are considered. Note that “yes” always goes to the left and “no” to the right.



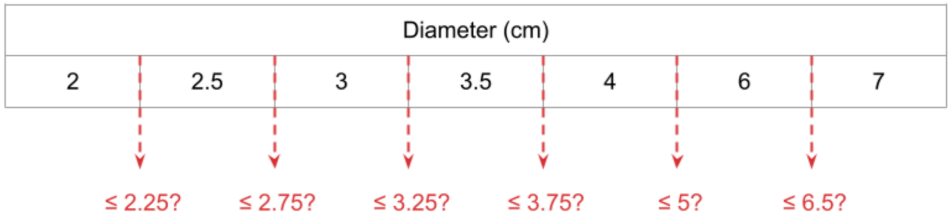
Three trees, labeled A through C, each sorting the fruit based on its color: red, yellow, or green.

Continuous variables

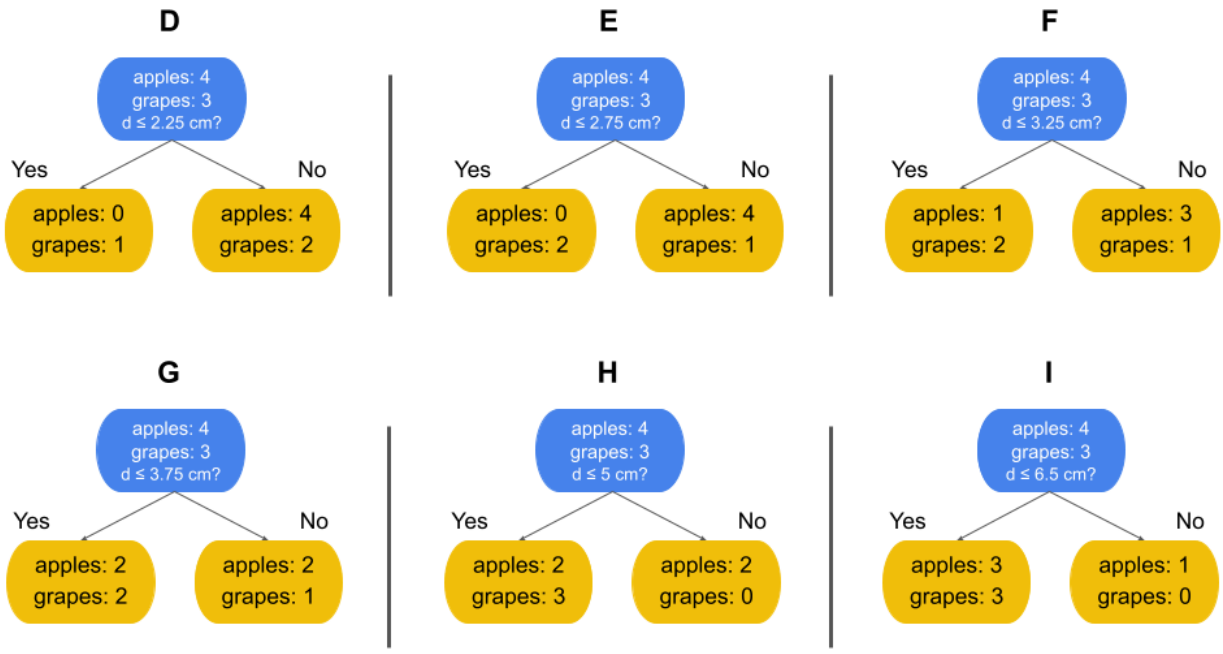
If the predictor variable is continuous, splits can be made anywhere along the range of numbers that exist in the data. Often the potential split points are determined by sorting the values for the feature and taking the mean of each consecutive pair of values. However, there can be any number of split points, and fewer split points can be considered to save computational resources and time. It is very common, especially when dealing with very large ranges of numbers, to consider split points along percentiles of the distribution.

In the case of the fruit example, “Diameter” is a continuous variable. One way a decision tree could handle this is to:

1. Sort the values, identify average of consecutive values:



1. Examine splitting based on these identified means:



Six trees, labeled D through I, each sorting the fruit based on diameter: 2.25, 2.75, 3.25, 3.75, 5, and 6.5 centimeters.

These are the six potential split points for the Diameter feature that were identified by the algorithm. Each option includes the children of that split, but note that at this point none of them has been evaluated yet. That's the next step.

Choosing splits: Gini impurity

Now you know how to determine the *potential* split points. In the fruit example, there are nine options to choose from: A–I. But how do you decide which split to use? This is where the "purity" of the child nodes becomes relevant. Generally, splits are better when

each resulting child node contains many more samples of one class than any other, like in example E above, because this means the split is effectively separating the classes—the primary job of the decision tree! In such cases, the child nodes are said to have low impurity (or high purity). The decision tree algorithm determines the split that will result in the lowest impurity among the child nodes by performing a calculation.

There are several possible metrics to use to determine the purity of a node and to decide how to split, including Gini impurity, entropy, information gain, and log loss. The most straightforward is Gini impurity, and it's also the default for the decision tree classifier in scikit-learn, so this reading will focus on that method. The Gini impurity of a node is defined as:

$$Gini\ impurity = 1 - \sum_{i=1}^{N} P(i)^2$$

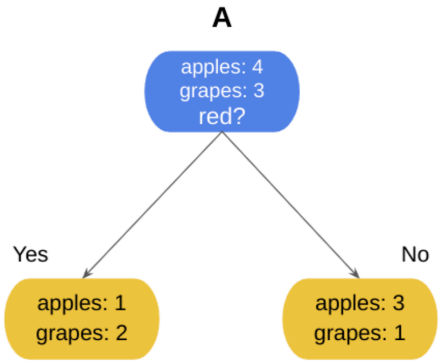
where i= class,

$P(i)$ = the probability of samples belonging to class *i* in a given node.

In the case of the fruits example, this becomes:

$$Gini\ impurity = 1 - P(apple)^2 - P(grape)^2$$

The Gini impurity is calculated for each child node of each potential split point. For example, there are nine split point options in the fruit example (A–I). The first potential split point is Color=red:



Calculate Gini impurity of each child node:

$$Gini\ impurity = 1 - P(apple)^2 - P(grape)^2$$

$$= 1 - (\frac{number\ of\ apples\ in\ node}{total\ number\ of\ samples\ in\ node})^2 - (\frac{number\ of\ grapes\ in\ node}{total\ number\ of\ samples\ in\ node})^2$$

For the “red=yes” child node:

Gini impurity = 1 - $(1/3)^2$ - $(2/3)^2$

= 1 - 0.111 - 0.444

= 0.445

And for the “red=no” child node:

Gini impurity = 1 - $(3/4)^2$ - $(1/4)^2$

= 1 - 0.5625 - 0.0625

= 0.375

Now there are two Gini impurity scores for split option A (whether or not the fruit is red)—one for each child node. The final step is to combine these scores by taking their weighted average.

Calculate weighted average of Gini impurities

The weighted average accounts for the different number of samples represented in each Gini impurity score. You cannot simply add them together and divide by two, because the first child node contained three samples and the second child node contained four. The weighted average of the Gini impurities (*Gi)* is calculated as:

Total Gi:

$$= (\frac{number\ of\ samples\ in\ LEFT\ child}{number\ of\ samples\ in\ BOTH\ child\ nodes}) * Gi_{left\ child} + (\frac{number\ of\ samples\ in\ RIGHT\ child}{number\ of\ samples\ in\ BOTH\ child\ nodes}) * Gi_{right\ child}$$

= (3/7 * 0.445) + (4/7 * 0.375)

Gi_{total} = 0.405

Repeat this process for every split option

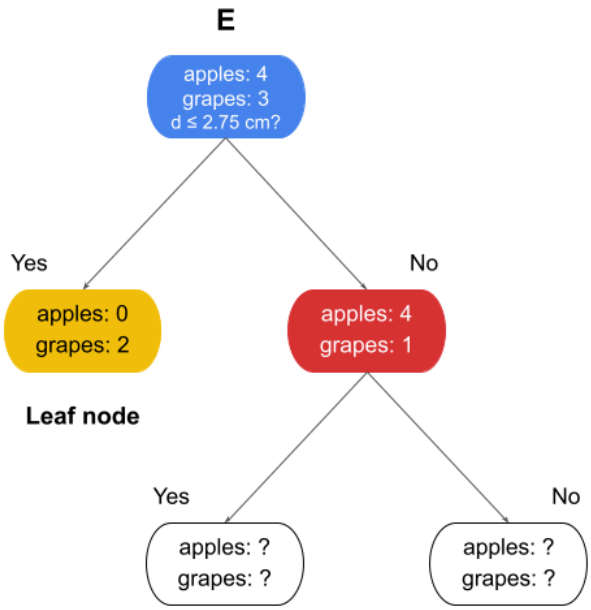
This same process is repeated for every split option. The fruit example has nine options (A–I):

Now there are nine Gini impurity scores ranging from 0.229 to 0.486. Since it's a measure of impurity, the best scores are those closest to zero. In this case, that's option E. The worst of the nine options is option C, because it doesn't separate classes well. The worst possible Gini impurity score is 0.5, which would occur when each child node contains an equal number of each class.

Now that the algorithm has identified the potential split points and calculated the Gini impurity of the child nodes that result from them, it will grow the tree by selecting the split point with the lowest Gini impurity.

Grow the tree

In the case of the given example, the root node would use split option E to split the data. The left child becomes a leaf node, because it contains just one class. However, the right child still does not have class purity, so it becomes a new decision node (in the absence of some imposed stopping condition). The steps outlined above will repeat on the samples in this node to identify the feature and split value that would yield the best result.



Splitting would continue until all the leaves are pure or some imposed condition stops the splitting.

You may have noticed that this process involves *a lot* of computation—and this was only for a dataset with two features and seven observations. As with many machine learning algorithms, the theory and methodology behind decision trees are fairly straightforward and have been around for many years, but it wasn't until the advent of powerful computing capabilities that these solutions were able to be put into practice.

Advantages and disadvantages of classification trees

Advantages:

- Require relatively few pre-processing steps
- Can work easily with all types of variables (continuous, categorical, discrete)
- Do not require normalization or scaling
- Decisions are transparent
- Not affected by extreme univariate values

Disadvantages:

- Can be computationally expensive relative to other algorithms
- Small changes in data can result in significant changes in predictions

Key takeaways

Decision trees are powerful predictive tools that can identify patterns in data that other algorithms might not be able to. They’re user-friendly because they require relatively few preprocessing steps compared to other models. They consist of a series of decision nodes, beginning at a root node and ending at leaf nodes. They operate by splitting data at particular feature values that are identified as thresholds. These split thresholds are determined by calculating the impurity of the child nodes that result from them, and selecting the split that yields the least impurity.

Tab 11

Hyperparameter tuning

In this reading, you will learn about hyperparameters, and how tuning them can affect model performance. You will develop a deeper understanding of some of the most commonly tuned hyperparameters for decision trees. Additionally, you will learn one process that is used to find optimal sets of hyperparameters.

Hyperparameter tuning

Throughout this program, you've learned about modeling techniques to make predictions or better understand data. Sometimes, the models perform well right out of the box, using the basic or default application of their underlying theory. Most often, however, this is not the case. Each dataset presents its own set of characteristics for which the data professional must tailor the model.

Depending on the characteristics of both the data and the algorithm used to model it, a model might overfit or underfit the data. Remember, the aim of a predictive model is to identify underlying, intrinsic patterns and characteristics in data that are representative of *all* such distributions, and use these characteristics to make predictions on new data.

Overfitting is when the model learns the training data so closely that it captures more than the intrinsic patterns of *all* such data distributions, and ends up learning noise or idiosyncrasies particular to *just* the training data. This results in a model that scores very well on the training data but considerably worse on unseen data because it cannot generalize well.

On the other hand, underfitting is when the model does not learn the patterns and characteristics of the training data well, and consequently fails to make accurate predictions on new data. It's typically easier to identify underfitting, because the model performs poorly on both training and test data. The best models neither underfit nor

overfit the data. They identify intrinsic patterns within it, but do not capture randomness or noise.

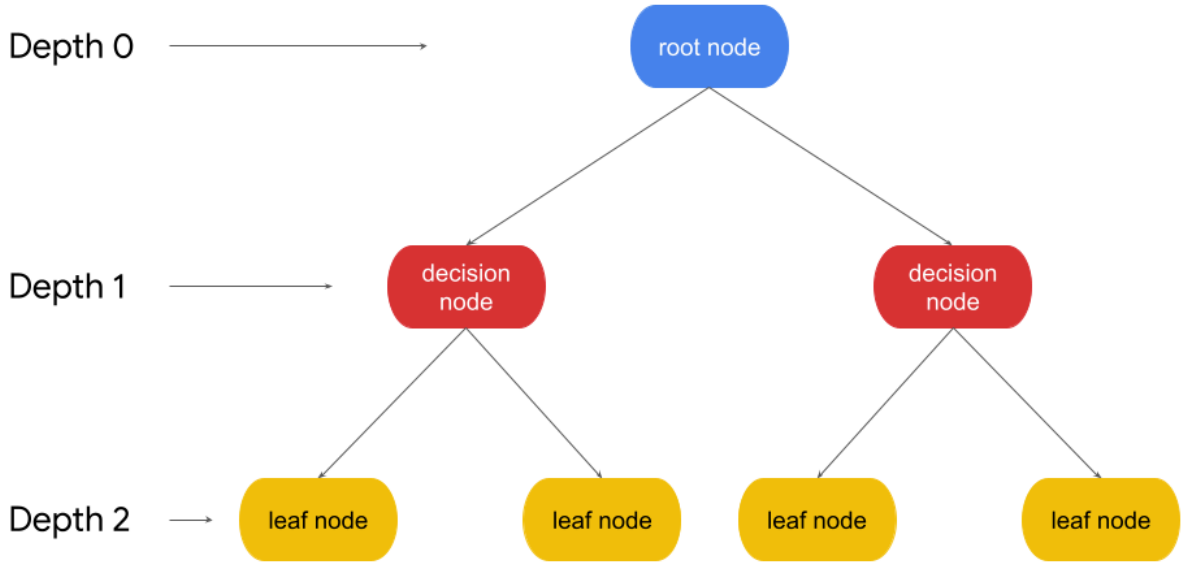
One way of helping to achieve this balance is through the use of hyperparameters. Hyperparameters are aspects of a model *that you set* before the model is trained, and that affect how the model fits the data. They are not derived from the data itself. Hyperparameter tuning is the process of adjusting the hyperparameters to build a model that best fits the data. (Note that you'll often hear them referred to as "parameters," much like you might hear the word "theory" used to mean "idea," when it actually has a very specific scientific meaning. It's okay, as long as you understand the difference.)

Hyperparameters for decision trees

There are many different hyperparameters available to control how a decision tree grows. Each hyperparameter affects something very specific related to the growth conditions. One might affect what causes a node to split, while another might limit how deep the tree is allowed to grow, and yet another might change the way node purity is calculated. This reading will introduce you to three hyperparameters: max_depth, min_samples_split, and min_samples_leaf.

max_depth

max_depth defines how deep the tree is allowed to grow. The depth of the tree is the distance, measured in number of levels, from the root node to the furthest leaf node. The root node would have a depth of zero, the child of the root node would have a depth of one, and so on.

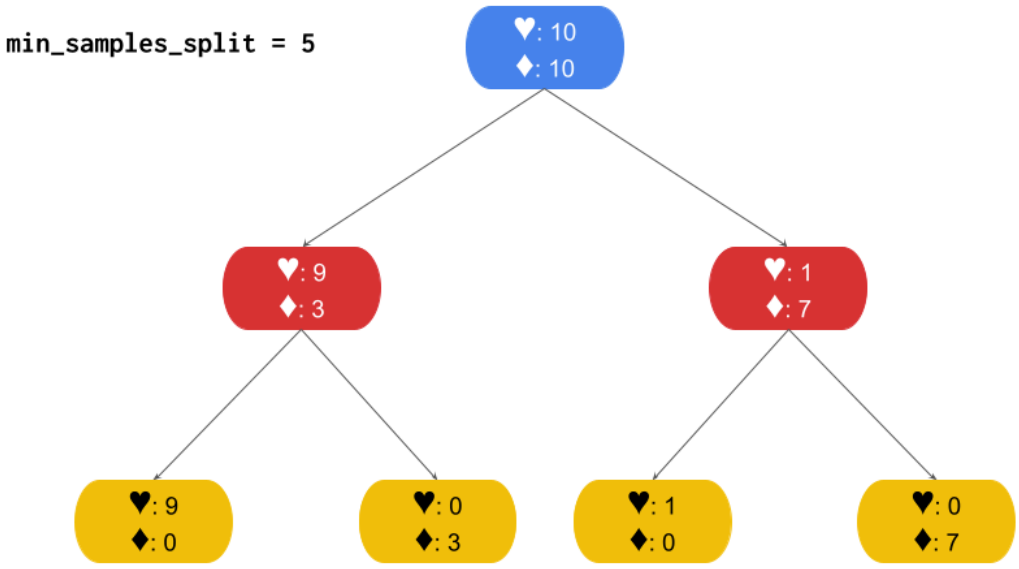


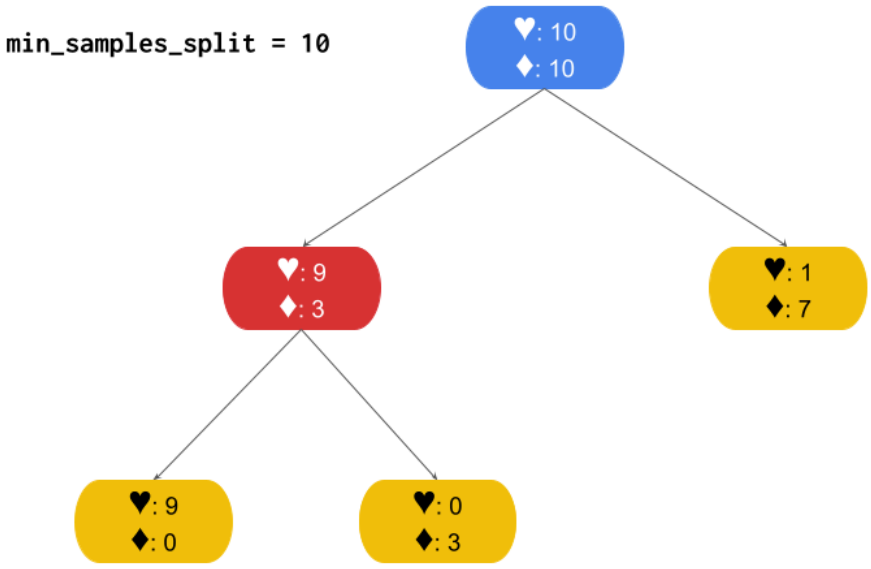
An unrestricted decision tree will continue splitting until every leaf node contains only a single class. As you increase the max depth parameter, the performance of the model on the training set will continue to increase. It's possible for a tree to grow so deep that leaves contain just a single sample. However, this overfits the model to the training data, and the performance on the testing data would probably be much worse. A quick intuition of why this happens: if you let a tree have so many nodes representing so many specific decision rules that perfectly align with the details of the training data, how likely do you think it is for those exact decision nodes to match new data in the real world? On the other hand, a tree that is not allowed to grow deeply enough will have high bias and fail to make accurate predictions. The best decision tree models are neither too shallow nor too deep, but just right.

min_samples_split

min_samples_split is the minimum number of samples that a node must have for it to split into more nodes. For example, if you set this to 10, then any node that contains nine or fewer samples will automatically become a leaf node. It will not continue splitting. However, if the node contains 10+ samples, it may continue to split into child

nodes. The greater the value you use for min_samples_split, the sooner the tree will stop growing. The minimum possible value is two, because two is the smallest number that can be divided into two separate child nodes.

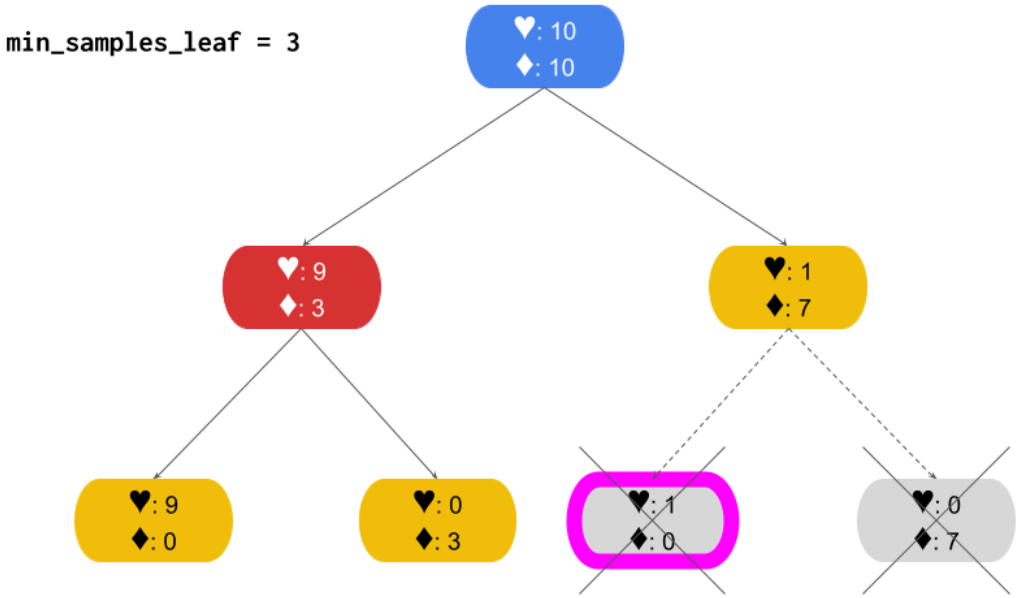




Notice how the tree in the example with min_samples_split=5 continues splitting until all leaves are pure, while the tree with min_samples_split=10 stops splitting on one side even though this node still contains a mix of classes. It stops because it contains only eight samples—below the threshold of 10 indicated by min_samples_split.

min_samples_leaf

min_samples_leaf is similar to min_samples_split, but with an important difference. Instead of defining how many samples the *parent* node must have *before* splitting, min_samples_leaf defines the minimum number of samples that must be in each *child* node *after* the parent splits. Consider the following example, where min_samples_leaf is set to three.



Notice that the right branch of the tree becomes a leaf at depth=1, while the left branch continues splitting until reaching class purity in the leaf nodes at depth=2. The right branch stops because splitting it would result in one of the child nodes containing just a single sample, which is below the threshold of three indicated by min_samples_leaf.

Finding the optimal set of hyperparameters

The values that these hyperparameters can take are limited only by the number of samples in your dataset. That leaves open the possibility for millions of combinations! How do you know what the values should be? The answer is to train a lot of different models to find out. There are a number of ways to do this. Performing a grid search is one of the more popular methods.

Grid search

A grid search is a technique that will train a model for every combination of preset ranges of hyperparameter values. The aim is to find the combination of values that results in a model that both fits the training data well and generalizes well enough to predict accurately on unseen data. After all these models have been trained, you then compare them to find this ideal model—if it exists.

Here is a very basic example of how a grid search might be used to find the best combination of two hyperparameters: max_depth and min_samples_leaf. You can define the set of max_depth values as [6, 8, None], and the set of min_samples_leaf values as [1, 3, 5].

		max_depth		
		6	**8**	**None**
min_samples_leaf	**1**	(1, 6)	(1, 8)	(1, None)
	3	(3, 6)	(3, 8)	(3, None)
	5	(5, 6)	(5, 8)	(5, None)

A tree is grown for each combination of values for each hyperparameter. Note that this grid search results in nine models—the product of the number of values being tried for each hyperparameter (3 x 3). If you wanted to include min_samples_split values of [2, 4, 6, 8] in your search as well, then you'd be growing 3 x 3 x 4 trees, or 36 different

models. Note also that, of these combinations of hyperparameters, the (5, 6) combination restricts growth regularizes the most, while the (1, None) combination allows the model to grow unrestricted.

Note: For decision trees (and all tree-based models), restricting growth is a form of regularization. Recall from the regression course that regularization refers to the process of reducing model complexity to prevent overfitting. The greater the complexity of a model, the more susceptible it is to overfit the training data. Regularization helps to make the model more generalizable to new data.

With more hyperparameters and a more expansive array of values to search over, grid searches can quickly become computationally expensive. One helpful search strategy is to try a wider array of values for each hyperparameter—say, [3, 6, 9], instead of [3, 4, 5]. If the best model has 6 as the value for this hyperparameter, perhaps try another grid search using [5, 6, 7] as potential values. This technique uses multiple search iterations to progressively home in on an optimal set of values.

Another technique is to define a more comprehensive set of search values from the beginning—say, [3, 4, 5, 6, 7, 8, 9]—and let the model train for what could be a very long time. Which approach you take will depend on your computing environment, your computational resources, and how much time you have.

Key takeaways

- Hyperparameters are aspects of a model that are set before the model trains, affecting the training itself.
- Different model types have different sets of hyperparameters that can be changed
- For decision trees, a few but some of the most important are:
    - max_depth: The maximum depth the tree will construct to before stopping
    - min_samples_split: The minimum number of samples that a node must have to split into more nodes.

    - min_samples_leaf: The minimum number of samples that must be in each child node for the split to complete.
- You can find the optimal set of hyperparameters using different types of algorithms. GridSearch is one of the more popular techniques, and involves specifying in advance all the values you want to try for each hyperparameter, and then training the model for every combination of those values.

Tab 12

More about validation and cross-validation

Previously, you learned how to split a dataset into training and testing data. Then, you fit a model to the training data and evaluated its performance on the test data. This basic process of building models with training data and evaluating them with held-out data is a fundamental part of machine learning and something that you should become very familiar with as a data professional. In this reading, you'll learn about validation and cross-validation, which are more rigorous ways of training and selecting a model.

Model validation

Fitting a model to training data and evaluating it on test data might be an adequate way of evaluating how well a single model generalizes to new data, but it's not a recommended way to compare multiple models to determine which one is best. That's because, by selecting the model that performs best on the test data, you never get a truly objective measure of future performance. The measure would be optimistic.

This may seem difficult to grasp or counterintuitive. You may be asking yourself: How is it not objective? After all, I'm not using the test data to tune my models. Well, if you're comparing how all of the models score on this data and then selecting the model with the best score as champion, in a way, you're "tuning" another hyperparameter—the model itself! The selection of the final model would itself behave as a tuning process, because you'd be using the test data to go back and make an upstream decision about your model. Put simply, if you want to use the test data to get a true measure of future performance, then you must never use it to make a modeling choice. Only use the test data to evaluate your final model. As a data professional, you will likely encounter scenarios where the test data is used to select a final model. It's not best practice, but

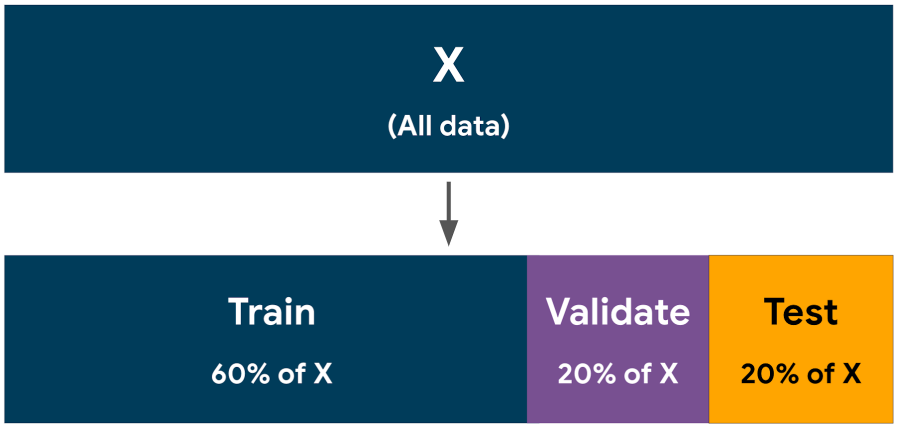
it's unlikely that it will break your model. However, there are better, more rigorous ways of evaluating models and selecting a champion.

One such way is through a process called validation. Model validation is the whole process of evaluating different models, selecting one, and then continuing to analyze the performance of the selected model to better understand its strengths and limitations. This reading will focus on evaluating different models and selecting a champion. Post-model-selection validation and analysis is a discipline unto itself and beyond the scope of this certification.

Validation sets

The simplest way to maintain the objectivity of the test data is to create another partition in the data—a validation set—and save the test data for after you select the final model. The validation set is then used, instead of the test set, to compare different models.

Here is one common way of splitting data, but note that these proportions are not required. You can split to whichever ratios make the most sense for your use case.



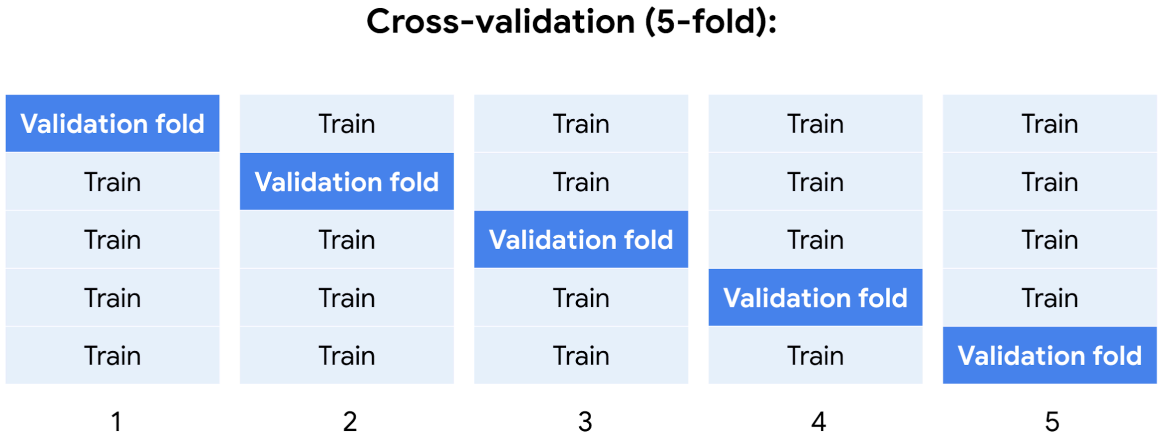
This method—using a separate validation set to compare models—is most commonly used when you have a very large dataset. The reason for this is that the more data you use for validation, the less you have for training and testing. However, if you don't have enough validation data, then your models' scores cannot be expected to give a reliable

measure that you can use to select a model, because there's a greater chance that the distributions in the validation data are not representative of those in the entire dataset.

When building a model using a separate validation set, once the final model is selected, best practice is to go back and fit the selected model to all the non-test data (i.e., the training data + validation data) before scoring this final model on the test data.

Cross validation

There is another approach to model validation that avoids having to split the data into three partitions (train / validate / test) in advance. Cross-validation makes more efficient use of the training data by splitting the training data into k number of "folds" (partitions), training a model on k – 1 folds, and using the fold that was held out to get a validation score. The training process occurs k times, each time using a different fold as the validation set. At the end, the final validation score is the average of all k scores. This process is also commonly referred to as k-fold cross validation.



After a model is selected using cross-validation, that selected model is then refit to the entire training set (i.e., it's retrained on all *k* folds combined).

The cross-validation process maximizes the usefulness of your data with the goal of getting a more accurate measure of model performance. It does so by averaging out the randomness imparted when splitting into training and validation folds. In other words, any time a dataset is split, the specific samples that go into each partition are usually

random, which makes it possible for the distributions in each partition to diverge from those found in the full dataset.

Cross-validation reduces the likelihood of significant divergence of the distributions in the validation data from those in the full dataset. For this reason, it's often the preferred technique when working with smaller datasets, which are more susceptible to randomness. The more folds you use, the more thorough the validation. However, adding folds increases the time needed to train, and may not be useful beyond a certain point.

Model selection

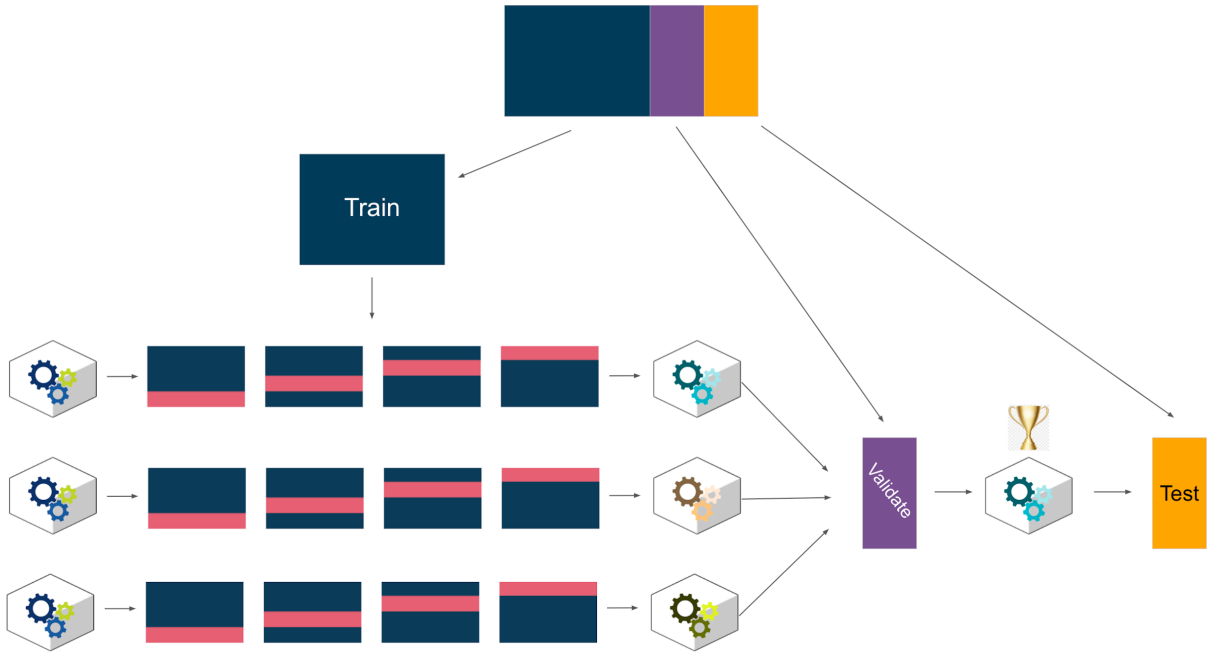
Once you've trained and validated your candidate models, it's time to select a champion. Of course, your models' validation scores factor heavily into this decision, but score is seldom the only criterion. Often you'll need to consider other factors too. How explainable is your model? How complex is it? How resilient is it against fluctuations in input values? How well does it perform on data not found in the training data? How much computational cost does it have to make predictions? Does it add much latency to any production system? It's not uncommon for a model with a slightly lower validation score to be selected over the highest-scoring model due to it being simpler, less computationally expensive, or more stable.

Once you have selected a champion model, it's time to evaluate it using the test data. The test data is used only for this final model. Your model's score on this data is how you can expect the model to perform on completely new data. Any changes you make to the model at this point that are based on its performance on the test data contaminate the objectivity of the score. Note that this does not mean that you can't make changes to the model. For instance, you might want to retrain the champion model on the entire dataset (train + validate + test) so it makes use of all available data prior to deployment. This is acceptable, but understand that at this point you have no way of meaningfully evaluating the model unless you acquire new data that the model hasn't encountered.

A review of the model development process

There is no single way to develop a model. Project-specific conditions will dictate the best approach. Over the course of your development as a data science professional, you'll likely encounter different variations of the train-validate-test process, some of which are more rigorous than others.

A rigorous approach to model development might use both cross-validation *and* validation. The cross-validation can be used to tune hyperparameters, while the separate validation set lets you compare the scores of different algorithms (e.g., logistic regression vs. Naive Bayes vs. decision tree) to select a champion model. Finally, the test set gives you a benchmark score for performance on new data. This process is illustrated in the diagram below.



For variations of this process, refer to the appendix.

Key takeaways

- Model validation is the whole process of evaluating different models, selecting one, and then continuing to analyze the performance of the selected model to

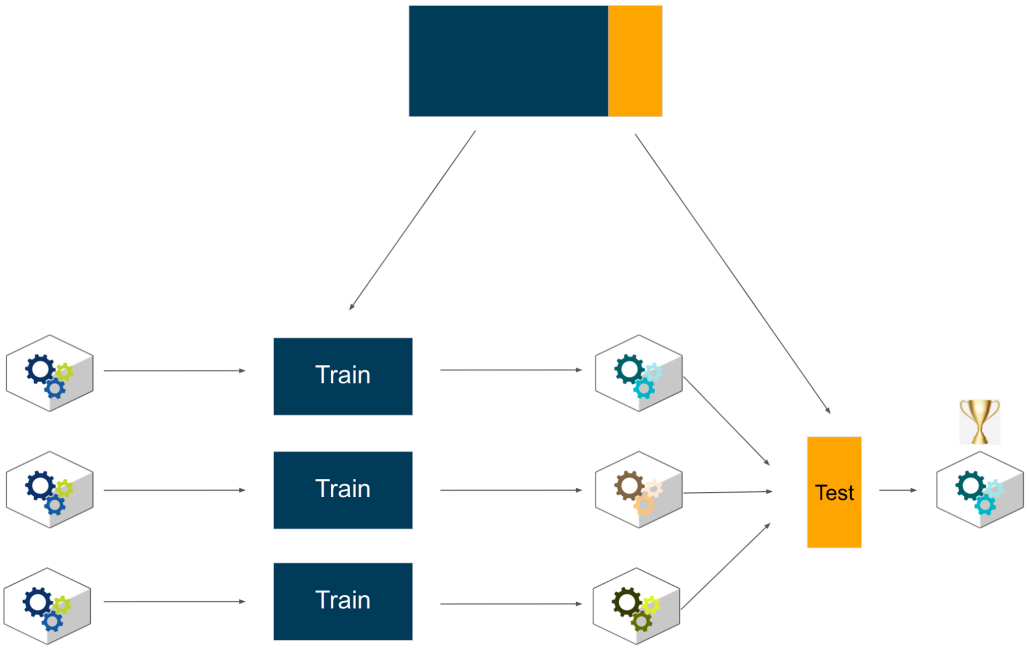
better understand its strengths and limitations. (Note that each unique combination of hyperparameters is a different model).

- Validation can be performed using a separate partition of the data, or it can be accomplished with cross-validation of the training data, or both.
- Cross-validation splits the training data into *k* number of folds, trains a model on *k* – 1 folds, and uses the fold that was held out to get a validation score. This process repeats *k* times, each time using a different fold as the validation set.
- Cross-validation is more rigorous, and makes more efficient use of the data. It's particularly useful for smaller datasets.
- Validation with a separate dataset is less computationally expensive, and works best with very large datasets.
- For a truly objective assessment of model performance on future data, the test data should not be used to select a final model.

Appendix

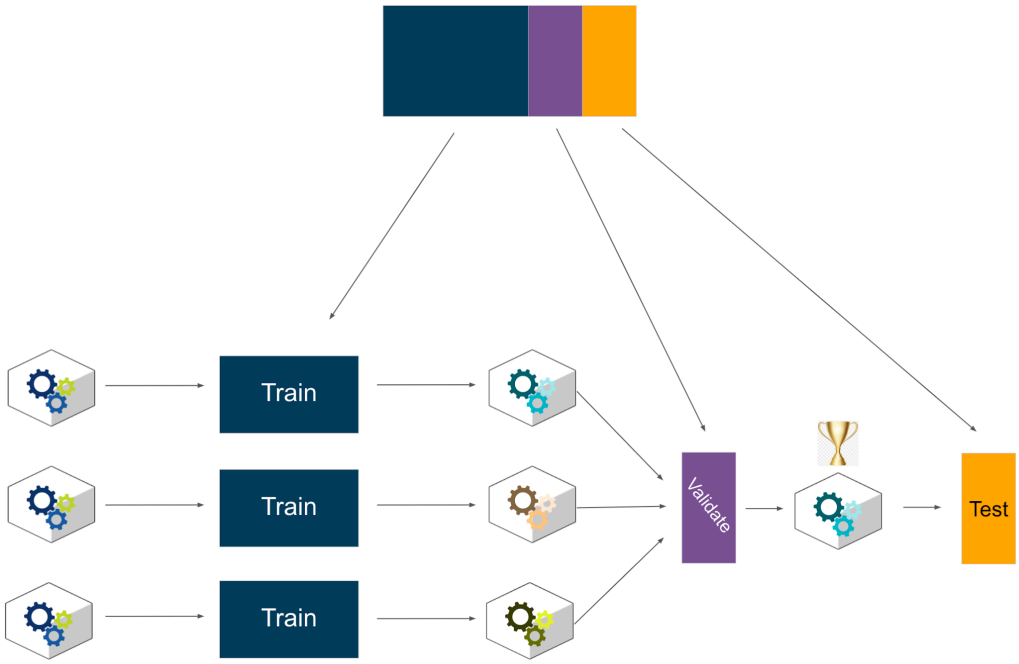
The following diagrams depict some variations of the model development process. Consider the implications of each choice.

A



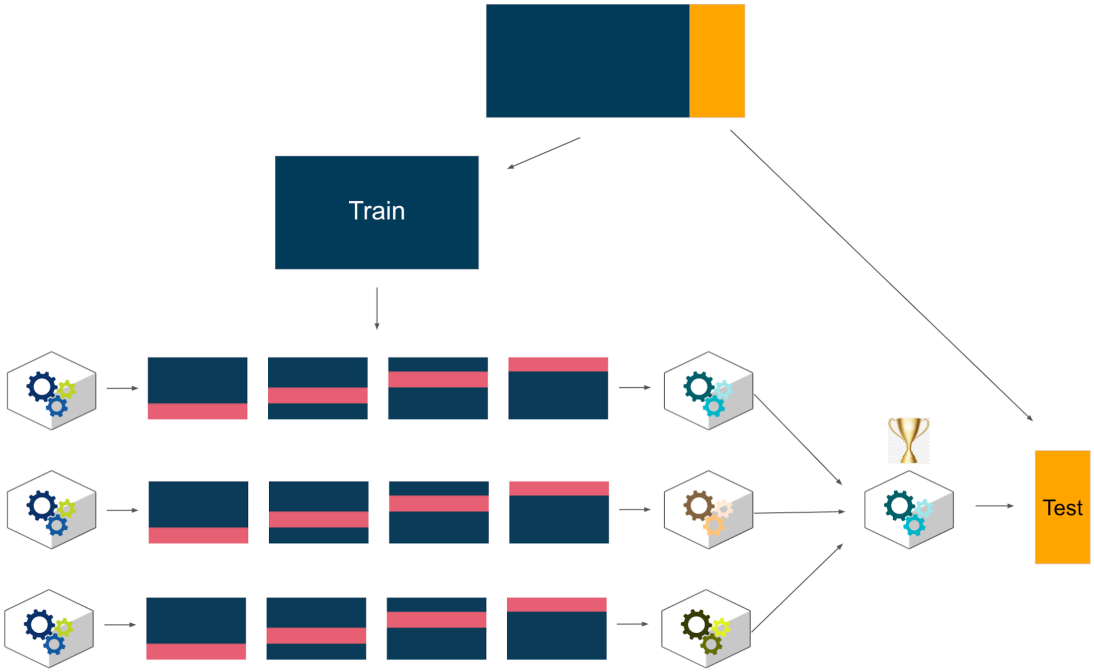
In this simple development scheme, data is split into train and test sets. Models are trained on the train data and all tested on the test data. The model with the best score on the test data is selected as champion. This approach does not iteratively tune hyperparameters or test the champion model on new data. The champion model's score on the test data would be an optimistic indicator of future performance.

B



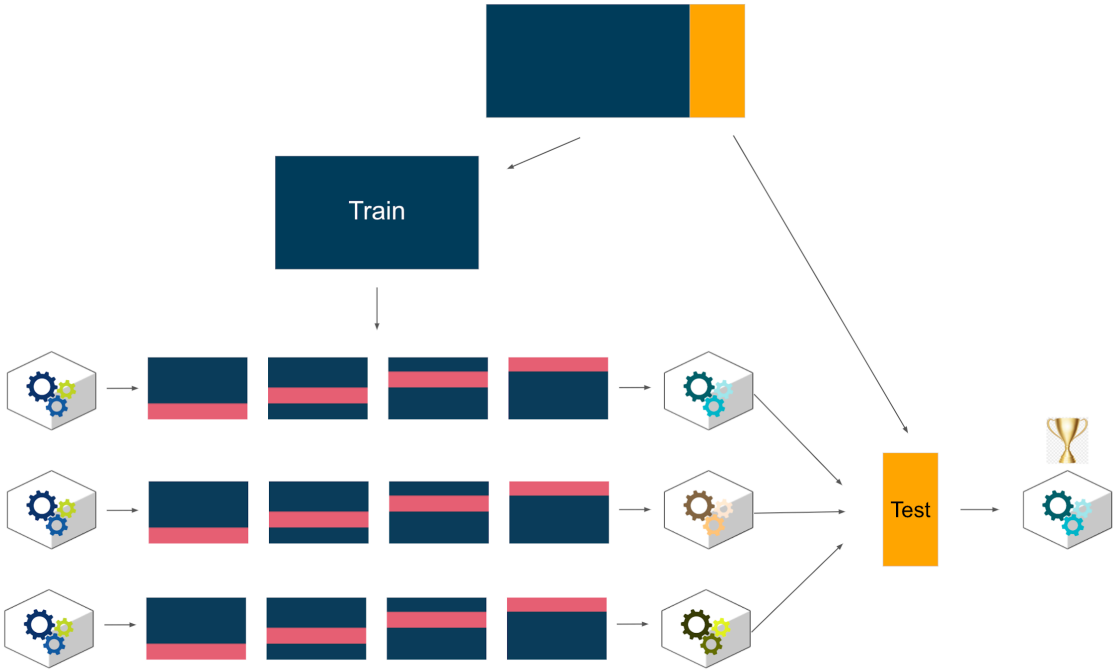
In this approach, the data is split into training, validation, and testing sets. Models are fit to the training data, and the champion model is the one that performs best on the validation set. This model alone is then scored on the test data. This is the same approach as example A, but with the added step of testing the champion model by itself to get a more reliable indicator of future performance.

C



This method splits the data into train and test sets. Models are trained and cross-validated using the training data. The model with the best cross-validation score is selected as the champion model, and this model alone is scored on the test data. The cross-validation makes the model more robust, and using the test data to evaluate only the champion model allows for a good understanding of future performance. However, selection of the champion model based on the cross-validation results alone increases the risk of overfitting the model to the training data.

D



In this variant, the data is split into training and testing sets. Models are trained and cross-validated using the training data, then all are scored on the test data. The model with the best performance on the test data is the champion. This is a very common approach, but note that it does not score the champion model on completely new data, so expected future performance may be optimistic. However, compared to approach C above, this method mitigates the risk of overfitting the model to the training data.

Tab 13

Reference guide: Random forest tuning

Reference guide: Random forest tuning

Previously, you learned about random forest models and studied how to build and tune them. This reading is a quick-reference guide to help you when you're building models of your own. It includes:

- Import statements
- Hyperparameters

Save this course item

You may want to save a copy of this guide for future reference. You can use it as a resource for additional practice or in your future professional projects. To access a downloadable version of this course item, click the link below and select "Use Template."

Reference guide: Random forest tuning

OR

If you don't have a Google account, you can download the item directly from the attachment below.

Import statements

Models

For classification tasks:

from `sklearn.ensemble` import `RandomForestClassifier`

For regression tasks:

from `sklearn.ensemble` import `RandomForestRegressor`

Evaluation metrics

For classification tasks:

from `sklearn.metrics` import

accuracy_score(y_true, y_pred, *[, ...])	Accuracy classification score.
average_precision_score(y_true, ...)	Compute average precision (AP) from prediction scores.
confusion_matrix(y_true, y_pred, *)	Compute confusion matrix to evaluate the performance of the training of a model .
f1_score(y_true, y_pred, *[, ...])	Compute the F1 score, also known as balanced F-score or F-measure.

fbeta_score(y_true, y_pred, *, beta)	Compute the F-beta score.
metrics.log_loss(y_true, y_pred, *[, eps, ...])	Log loss, aka logistic loss or cross-entropy loss.
multilabel_confusion_matrix(y_true, ...)	Compute a confusion matrix for each class or sample.
precision_recall_curve(y_true, ...)	Compute precision-recall pairs for different probability thresholds.
precision_score(y_true, y_pred, *[, ...])	Compute the precision.
recall_score(y_true, y_pred, *[, ...])	Compute the recall.
roc_auc_score(y_true, y_score, *[, ...])	Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.

For regression tasks:

from `sklearn.metrics` import

mean_absolute_error(y_true, y_pred, *)	Mean absolute error regression loss.
mean_squared_error(y_true, y_pred, *)	Mean squared error regression loss.
mean_squared_log_error(y_true, y_pred, *)	Mean squared logarithmic error regression loss.
median_absolute_error(y_true, y_pred, *)	Median absolute error regression loss.
mean_absolute_percentage_error(...)	Mean absolute percentage error (MAPE) regression loss.
r2_score(y_true, y_pred, *[, ...])	R^2 (coefficient of determination) regression score function.

Hyperparameters

The following are some of the most important hyperparameters for random forest classification models in scikit-learn.

Hyperparameter	What it does	Input type	Default Value	Considerations
`n_estimators`	Specifies the number of trees your model will build in its ensemble	int	100	A typical range is 50–500. Consider how much data you have, how deep the trees are allowed to grow and how many samples are bootstrapped to grow each tree (you generally need more trees if they're shallow, and more trees if your

				bootstrap sample size is smaller). Also consider if your use case has latency requirements.
`max_depth`	Specifies how many levels your tree can have. If None, trees grow until leaves are pure or until all leaves contain less than `min_samples_split` samples.	int	None	Random forest models often use base learners that are fully grown, but restricting tree depth can reduce train/latency times and prevent overfitting. If not None,

				consider values 3–8.
`min_samples_split`	Controls threshold below which nodes become leaves If float, then it represents a percentage (0–1] of `max_samples`.	int or float	2	Consider (a) how many samples are in your dataset, and (b) how much of that data you're allowing each base learner to use (i.e., the value of the max_samples hyperparameter). The fewer samples available, the lesser the number of samples

				may need to be allowed in each leaf node (otherwise the tree would be very shallow).

`min_samples_leaf`	A split can only occur if it guarantees a minimum of this number of observations in each resulting node. If float, then it represents a percentage (0–1] of `max_samples`.	int or float	1	Consider (a) how many samples are in your dataset, and (b) how much of that data you're allowing each base learner to use (i.e., the value of the max_samples hyperparameter). The fewer samples available, the lesser the number of samples may need to be allowed in each leaf

				node (otherwise the tree would be very shallow).
`max_features`	Specifies the number of features that each tree randomly selects during training If int, then consider `max_features` features at each split. If float, then `max_features` is a fraction and `round(max_features * n_features)` features are considered at each split. If “sqrt”, then `max_features=sqrt(n_features)`.	{“sqrt”, “log2”, None}, int or float,	“sqrt”	Consider how many features the dataset has and how many trees will be grown. Fewer features sampled during each bootstrap means more base learners would be needed. Small max_features values on datasets

	If “log2”, then `max_features=log2(n_features)`. If None, then `max_features=n_features`.			with many features mean more unpredictive trees in the ensemble.

`max_samples*`	Specifies the number of samples bootstrapped from the dataset to train each base model If float, then it represents a percentage (0–1] of the dataset. If None, then draw `X.shape[0]` samples.	int or float	None	Consider the size of your dataset. When working with large datasets, it can be beneficial to limit the number of samples in each tree, because doing so can greatly reduce training time and yet still result in a robust model. For example, 20% of 1 billion may be enough to capture patterns in the data,

				but if you only have 1,000 samples in your dataset, you'll probably need to use them all.

* Note that `max_samples` was not used in the "Build and validate/cross-validate a random forest model" videos and notebook, but is included here so you can use this hyperparameter in your own work. Remember that using fractions of the data to train each base learner can possibly improve model predictions and certainly speed up execution times.

Key takeaways

When building machine learning models, it's essential to have the right tools and to understand how to use them. While there are numerous other hyperparameters to explore, the ones in this reference guide are among the most important. Be inquisitive and try different approaches. Discovering ways to improve your model is a lot of fun!

Tab 14

Case Study: Machine learning model unearths resourcing insights for Booz Allen Hamilton

Booz Allen Hamilton, a global consulting firm, employs experts in analytics, digital solutions, engineering, and cyber solutions who support a wide variety of clients. One of the ways they've used machine learning to help their business is by constructing a model to better understand the likelihood of winning contracts. Insights derived from the model help Booz Allen Hamilton anticipate its demand signal and subsequently understand and plan for future resourcing and staffing needs.

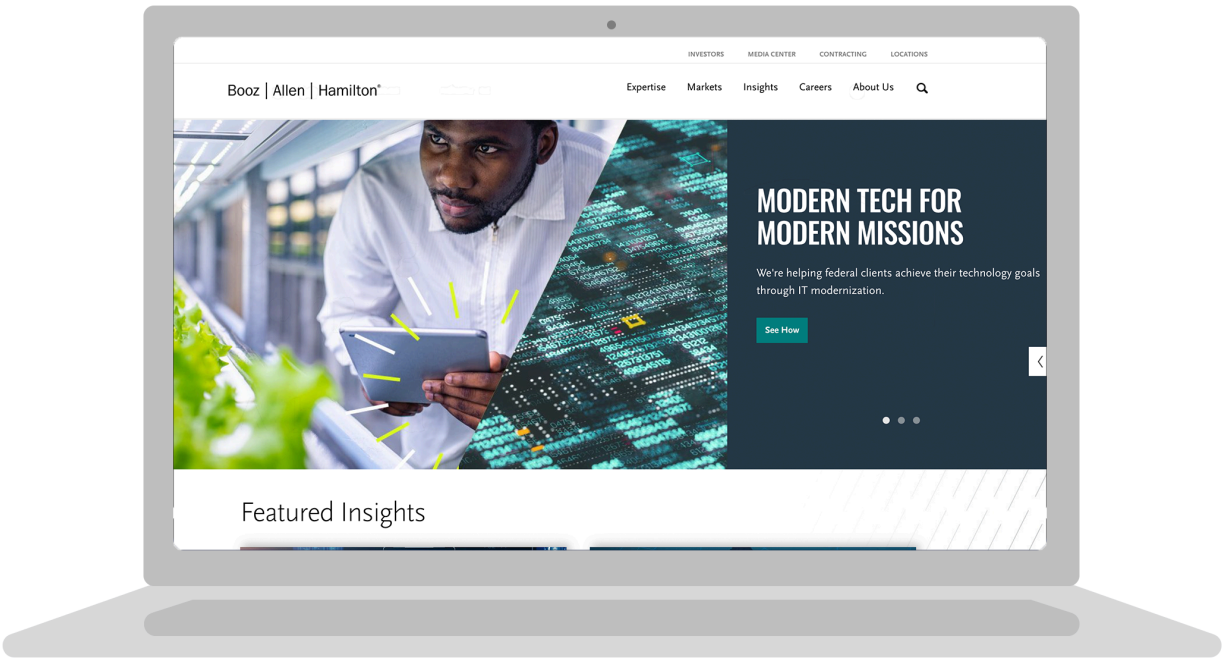
Booz | Allen | Hamilton

Refer to this case study to learn more about how a machine learning model is developed and deployed at an enterprise level.

Company background

For more than 100 years, government, military, and business leaders have turned to Booz Allen Hamilton to solve their most complex problems. Their experts in analytics, digital solutions, engineering, and cyber work together to find solutions that help organizations transform. They are a key partner on some of the most innovative programs for governments worldwide and are trusted by these governments' most sensitive agencies. They work in partnership with clients, using a mission-first approach to choose the right strategy and technology to help them realize their vision. With global

headquarters in McLean, Virginia, Booz Allen Hamilton employs nearly 29,300 people globally as of June 30, 2022, and had revenue of $8.4 billion for the 12 months ending March 31, 2022. To learn more, visit www.boozallen.com.



The challenge

Booz Allen Hamilton's work comes from contracts awarded by federal government and commercial clients. To get this work, the company bids on contracts. The bidding process involves reviewing the job requirements and submitting a proposal for a solution and a cost to implement it. The organization that solicited the bids will then review these proposals and award a contract to the company whose bid best suits their needs and budget. Upon award, Booz Allen must meet those requirements and deliver the best possible solution to their client. For a company of its size, the uncertainty that comes with bidding and winning work can make staffing and resource planning difficult. With this machine learning model they are able to better understand which contracts they'd be more likely to win and therefore understand the workforce and skills needed to meet its future demand.

Step 1: The stakeholders pave the way

Even though the model would only be used internally at Booz Allen Hamilton, it still involved numerous stakeholders, all of whom needed to work together to develop and deploy the model. At large organizations, this can take days, weeks, or even months. For this model, the stakeholders include:

- Corporate Finance team: This is the team that will use the model. They have domain expertise of the model's applications and use cases. They dictate what the model should do, and they know what data is relevant and available to build it.
- Enterprise Data Science team: This is the team responsible for training, validating, and deploying the model. Throughout model development, they work closely with the model end-users (the Corporate Finance team) to collect feedback, explore use cases, and validate results.
- Enterprise Platforms and Engineering team: This is the team responsible for storing and maintaining the data in an enterprise data lake (a large repository of structured and unstructured data). Large organizations typically have so much data that there is a team tasked with managing all of it. The Platforms and Engineering team also is responsible for setting up the necessary computing platform and infrastructure where the model is developed and deployed.

Step 2: Develop the model

To provide more insight to leadership and decision-makers, the Data Science team created a tree-based classification model that predicts the likelihood of winning a contract.

Target variable and model output: The target variable was binary: whether or not a submitted bid won Booz Allen Hamilton the contract. The model's final output is the probability of winning a given contract.

Algorithms considered: The Data Science team explored a number of different algorithms for the model, including logistic regression, support vector machines, decision tree, and random forest. A random forest model was ultimately selected as the champion solution.

Class balance: There was a minor imbalance between the classes of the target variable. Both upsampling and downsampling were tried, but ultimately neither provided any substantial lift over the baseline.

Evaluation metrics: The model was evaluated based on its performance with respect to four different metrics:

1. Area under the ROC curve (ROC/AUC)
2. F1 score
3. Accuracy
4. Log-loss

Splitting the data: The data comprises several full years of historical data as well as resolved bids from the current fiscal year. The data was split into training and test sets.

- Training data: Several years of historical data and 50% of resolved bids from the current fiscal year
- Test data: The other 50% of resolved bids from the current fiscal year

Model training & tuning: The Data Science team tuned four main hyperparameters using 5-fold cross-validation:

- Max samples: The number of observations sampled with replacement
- Max features: The number of features to consider when looking for the best split
- Number of trees: The number of base learners grown in the ensemble
- Tree depth: The level to which each tree is allowed to grow

Feature selection: Of the 250+ initial features, approximately 40 features of varying types (numerical, categorical, and Boolean) were selected for use in the final model based on their relative importance.

Model selection: The final model was a random forest model that was selected based on its performance on the test data as indicated by the four metrics listed above.

Step 3: Deploy the model

Once the Data Science team finished building the model, they drafted a report to present to the Corporate Finance team. This report contains details of the model development and validation process, including model architecture, performance results, important features, and relevant visualizations that support their conclusions. One of the most important criteria for model approval and deployment is that the model provides an improvement over existing methods of estimation. In this case, the final model proved to be 12% more accurate than existing methods.

After receiving buy-in from the Corporate Finance team, the Data Science team, with support from the Platforms and Engineering team, worked to deploy the model so end users can begin using its predictions. Enterprise-level data science is not performed on a personal laptop. The volume of data, computing requirements, and risk are all too great for this. The model must be developed and deployed on a platform that is powerful, reliable, and secure enough to support it. The Data Science team developed and deployed this model leveraging the enterprise data lake infrastructure, which includes a notebook-based model development environment as well as a platform to streamline machine learning development, management, and deployment.

Key takeaways

Below is a summary of the main insights from this case study.

- Machine learning solutions are transforming the way businesses operate, and these solutions are not all customer-facing; they're also used to aid in internal processes and decision-making.
- Development of machine learning solutions at large businesses involves many stakeholders with diverse responsibilities and fields of expertise.
- Many of the algorithms taught in this certification—including logistic regression, decision trees, and random forests—are used by the biggest companies in the world, and their modeling process closely resembles that which is presented in this course's notebooks.
- Even when a model performs well, its use must still be carefully explained and justified before deployment. It is not enough to simply build a model that scores well.

Tab 15

More about gradient boosting

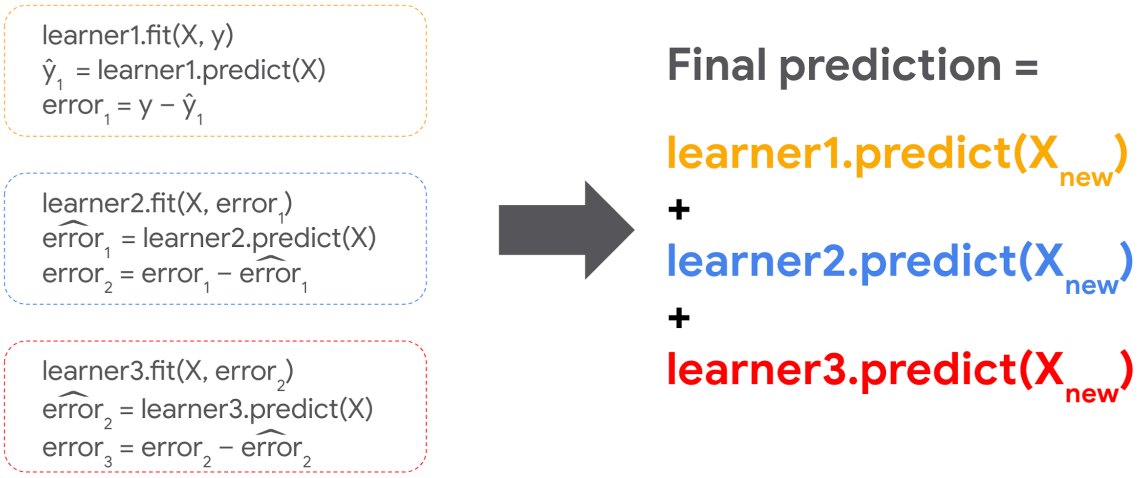
Previously, you learned about gradient-boosting machines (GBMs). Gradient boosting is one of the most powerful supervised learning techniques. It's important to understand how gradient boosting works because, as a data professional, you'll likely encounter this type of model frequently. In this reading, you'll review how gradient boosting works and then explore it in greater depth through a worked example.

Review

Recall that gradient boosting is a supervised learning technique that uses model ensembling to predict on a target variable. Although GBMs do not have to be tree-based, tree ensembles are the most common implementation of this technique. There are two key features of tree-based gradient boosting that set it apart from other modeling techniques:

1. It works by building an ensemble of decision tree base learners wherein each base learner is trained successively, attempts to predict the error—also known as "residual"—of the previous tree, and therefore compensate for it.
2. Its base learner trees are known as "weak learners" or "decision stumps." They are generally very shallow.

Here is a review of the pseudo-code outline of gradient boosting for an ensemble of just three trees:



With commented explanations:

learner1.fit(X, y) # Fit the data

$\hat{y}_1$ = learner1.predict(X) # Predict on X

$error_1 = y - \hat{y}_1$ # Calculate the error → (actual – predicted)

learner2.fit(X, $error_1$) # Fit tree 2, but target = error from tree 1

$\hat{error}_1$ = learner2.predict(X) # Predict on X

$error_2 = error_1 - \hat{error}_1$ # Calculate the new error

learner3.fit(X, $error_2$) # Fit tree 3, but target = error from tree 2

$\hat{error}_2$ = learner3.predict(X) # Predict on X

$error_3 = error_2 - \hat{error}_2$ # Calculate the new error

For these observations and any new samples being predicted by this model:

Final prediction = learner1.predict(X) + learner2.predict(X) + learner3.predict(X)

A worked example

Here is a demonstration of this process in action. For this scenario, consider a vendor selling bottles of water along a bike path. Below is a table of how many bottles of water he sold on different days and the noontime temperature of each day. The model will try to predict how many water bottles the man sold based on the day's temperature in degrees Celsius.

Temperature (℃)	Sales (bottles)

14	72
17	80
20	98
23	137
26	192
29	225
32	290
35	201
38	95
41	81

Step one is to fit a model to the data. This example uses temperature in Celsius as a single X feature and sales as the target variable. The model is a regular decision tree that is only allowed to grow to a maximum depth of one (i.e., it only makes one split), to replicate the weak learners used by GBMs.

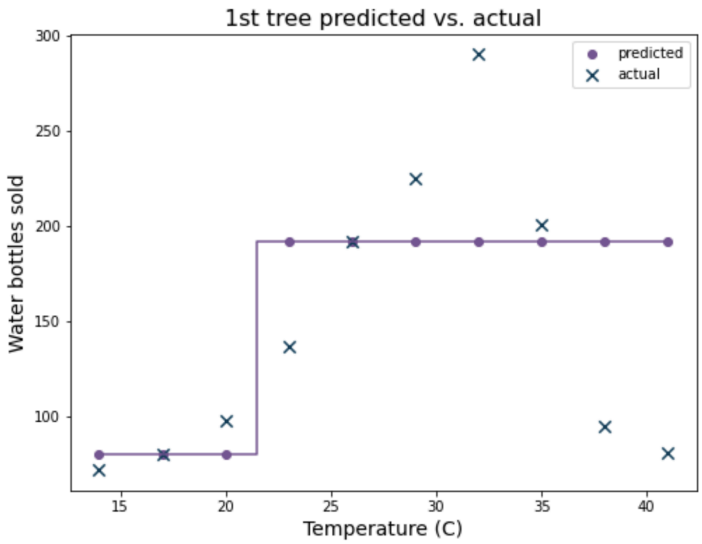
index	temp (℃)	sales	tree 1 predictions	tree 1 error	tree 2 predictions	tree 2 error	tree 3 predictions	tree 3 error	final prediction
0	14	72	80	-8	4.5	-12.5	-4.5	-8	80
1	17	80	80	0	4.5	-4.5	-4.5	0	80
2	20	98	80	18	4.5	13.5	-4.5	18	80
3	23	137	192	-55	4.5	-59.5	-4.5	-55	192

4	26	192	192	0	4.5	-4.5	-4.5	0	192
5	29	225	192	33	4.5	28.5	7	21.5	203.5
6	32	290	192	98	4.5	93.5	7	86.5	203.5
7	35	201	192	9	4.5	4.5	7	-2.5	203.5
8	38	95	192	-97	-104	7	7	0	95
9	41	81	192	-111	-104	-7	7	-14	95

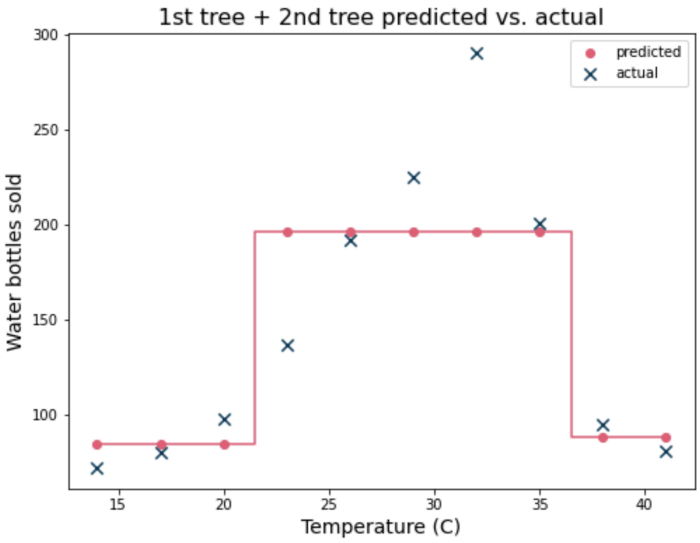
The table above contains information including the temperature, number of water bottles sold, predictions of three base learner trees, their error, and their final prediction for each day. The first tree tries to predict the number of sales. Each subsequent tree tries to predict the error of the tree that came before it.

Take the day at index 0 for example. The number of sales for that day was 72. Tree 1 predicted 80. To calculate the residual error, simply subtract actual minus predicted. In this case: 72 – 80 = -8. Notice that -8 is the value in the “tree 1 error” column. The next tree (tree 2) tries to predict tree 1’s error, and so on. The final prediction represents the sum of the predictions of all three trees. In the case of the day at index 0, this is: 80 + 4.5 – 4.5 = 80.

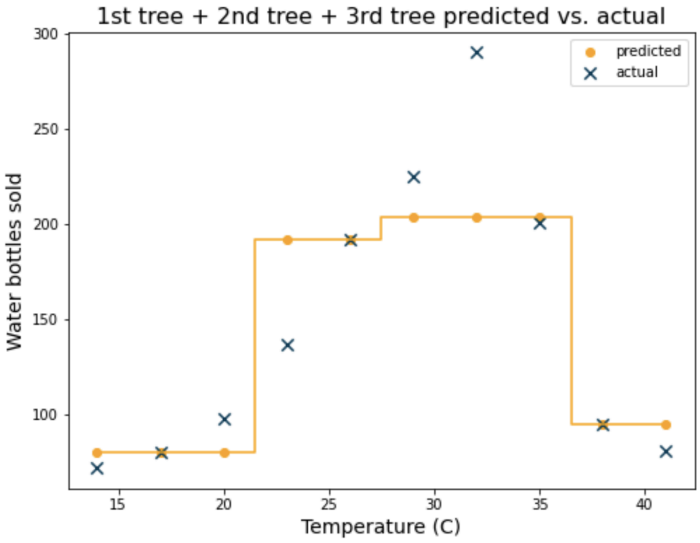
Here are some figures that depict the predicted vs. actual number of bottles sold for each step of the process.



In the first graph, the Xs indicate the actual number of water bottles sold and the purple dots indicate the predicted number. Each vertical part of the line that connects the dots represents a split, or decision boundary, of the model. Notice that for a single tree there’s only one vertical line, because it only makes one split.

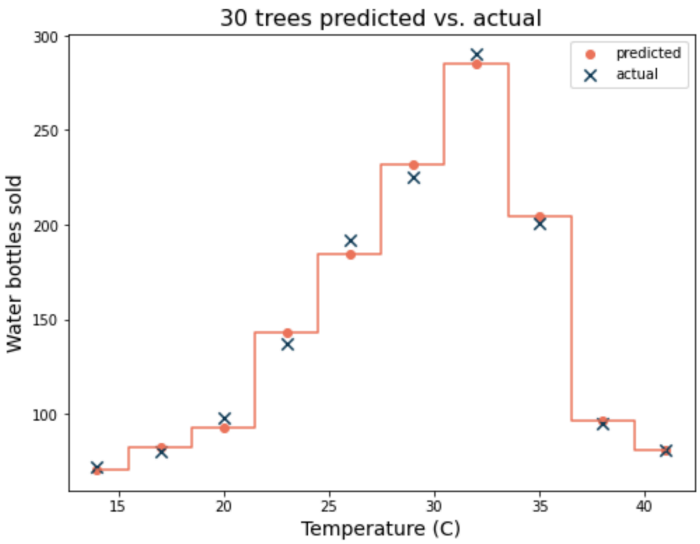


The next graph depicts the predictions of the first two trees vs. the actual values. Each predicted point represents the sum of the predictions of the first and second trees. The predictions match the actual values more closely, but the model still underfits the data.



The graph above depicts the next iteration of the model. Now there are three trees whose predictions are summed. Each additional base learner adds more nuance to the final prediction; it adds a decision boundary (represented here by the vertical segments of the yellow line), which allows the predictions to become more accurate. In this case, every sample is assigned to one of four different values, whereas in the previous example every sample was assigned to one of three different values.

Here is the prediction curve for 30 trees:



The predicted values of this model ensemble very closely match the actual values of the samples. Note, however, that they are not perfect predictions. Indeed, a single decision tree could fit this data perfectly if it were allowed to grow to a depth of five. However, a benefit of ensemble methods like gradient boosting is that they are less likely to overfit the data than a single decision tree.

Key takeaways

Gradient boosting is a powerful and straightforward technique that uses an ensemble of weak learners to make a final prediction. GBM models are more resilient to high variance that results from overfitting the data due to being comprised of high-bias, low-variance weak learners. The bias of each weak learner in the final model is mitigated by the ensemble.

Tab 16

Reference guide: XGBoost tuning

Previously, you learned about gradient boosting machine models and studied how to build and tune them with XGBoost's scikit-learn API. This reading is a quick-reference guide to help you when you're building XGBoost models of your own. It includes information on the following components:

- Import statements
- Hyperparameters

Save this course item

You may want to save a copy of this guide for future reference. You can use it as a resource for additional practice or in your future professional projects. To access a downloadable version of this course item, click the following link and select "Use Template."

Reference guide: XGBoost tuning

OR

If you don't have a Google account, you can download the item directly from the following attachment.

Import statements

The following are some of the most commonly used import statements for gradient boosting models using the XGBoost library together with scikit-learn.

Models

For classification tasks:

```
from xgboost import XGBClassifier
```

For regression tasks:

```
from xgboost import XGBRegressor
```

Evaluation metrics

For classification tasks:

```
from sklearn.metrics import
```

accuracy_score (y_true, y_pred, *[, ...])	Accuracy classification score
average_precision_score(y_true, ...)	Compute average precision (AP) from prediction scores
confusion_matrix(y_true, y_pred, *)	Compute confusion matrix to evaluate the performance of the training of a model
f1_score(y_true, y_pred, *[, ...])	Compute the F1 score, also known as balanced F-score or F-measure

fbeta_score(y_true, y_pred, *, beta)	Compute the F-beta score
metrics.log_loss(y_true, y_pred, *[, eps, ...])	Log loss, aka logistic loss or cross-entropy loss
multilabel_confusion_matrix(y_true, ...)	Compute a confusion matrix for each class or sample
precision_recall_curve(y_true, ...)	Compute precision-recall pairs for different probability thresholds
precision_score(y_true, y_pred, *[, ...])	Compute the precision
recall_score(y_true, y_pred, *[, ...])	Compute the recall
roc_auc_score(y_true, y_score, *[, ...])	Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores

For regression tasks:

```
from sklearn.metrics import
```

mean_absolute_error(y_true, y_pred, *)	Mean absolute error regression loss
mean_squared_error(y_true, y_pred, *)	Mean squared error regression loss
mean_squared_log_error(y_true, y_pred, *)	Mean squared logarithmic error regression loss
median_absolute_error(y_true, y_pred, *)	Median absolute error regression loss
mean_absolute_percentage_error(...)	Mean absolute percentage error (MAPE) regression loss
r2_score(y_true, y_pred, *[, ...])	R^2 (coefficient of determination) regression score function

Hyperparameters

The following are some of the most important hyperparameters for gradient boosting machine classification models built with the XGBoost library. These are the hyperparameters that data professionals typically reach for first, because they are

among the most intuitive and they control the model at different levels using a diverse variety of mechanisms.

n_estimators

Hyperparameter	What it does	Input type	Default Value
n_estimators	Specifies the number of boosting rounds (i.e., the number of trees your model will build in its ensemble)	int	100

Considerations:

A typical range is 50–500. Consider how much data you have, how deep the trees are allowed to grow, and how many samples are bootstrapped from the overall data to grow each tree (you generally need more trees if they're shallow, and more trees if your bootstrap sample size represents just a small fraction of your overall data). For an extreme but illustrative example, if you have a dataset of 10,000, and each tree only bootstraps 20 samples, you'll need more trees than if you gave each tree 5,000 samples. Also keep in mind that, unlike random forest, which can grow base learners in parallel, gradient boosting grows base learners successively, so training can take longer for more trees.

max_depth

Hyperparameter	What it does	Input type	Default Value
max_depth	Specifies how many levels your base learner trees can have. If None, trees grow until leaves are pure or until all leaves have less than min_child_weight.	int	6

Considerations: Controls complexity of the model. Gradient boosting typically uses weak learners, or “decision stumps” (i.e., shallow trees). Restricting tree depth can reduce training times and serving latency as well as prevent overfitting. Consider values 2–6.

min_child_weight

Hyperparameter	What it does	Input type	Default Value

min_child_weight	Controls threshold below which a node becomes a leaf, based on the combined weight of the samples it contains. For regression models, this value is functionally equivalent to a number of samples. For the binary classification objective, the weight of a sample in a node is dependent on its probability of response as calculated by that tree. The weight of the sample decreases the more certain the model is (i.e., the closer the probability of response is to 0 or 1).	int or float	1

Considerations: Higher values will stop trees splitting further, and lower values will allow trees to continue to split further. If your model is underfitting, then you may want to lower it to allow for more complexity. Conversely, increase this value to stop your trees from getting too finely divided.

learning_rate

Hyperparameter	What it does	Input type	Default Value
learning_rate	Controls how much importance is given to each consecutive base learner in the ensemble's final prediction. Also known as *eta* or *shrinkage*.	float	0.3

Considerations: Values can range from (0–1]. Typical values range from 0.01 to 0.3. Lower values mean less weight is given to each consecutive base learner. Consider how many trees are in your ensemble. Lower values typically benefit from more trees.

colsample_bytree*

Hyperparameter	What it does	Input type	Default Value
colsample_bytree*	Specifies the percentage (0–1.0] of features that each tree randomly selects during training	float	1.0

Considerations: Adds randomness to the model to make it robust to noise. Consider how many features the dataset has and how many trees will be grown. Fewer features sampled means more base learners might be needed. Small colsample_bytree values on datasets with many features mean more unpredictive trees in the ensemble.

subsample*

Hyperparameter	What it does	Input type	Default Value
subsample*	Specifies the percentage (0–1.0] of observations sampled from the dataset to train each base model.	float	1.0

Considerations: Adds randomness to the model to make it robust to noise. Consider the size of your dataset. When working with large datasets, it can be beneficial to limit the number of samples in each tree, because doing so can greatly reduce training time and yet still result in a robust model. For example, 20% of 1 billion might be enough to capture patterns in the data, but if you only have 1,000 samples in your dataset then you'll probably need to use them all.

*Note that `colsample_bytree` and `subsample` were not used in the Tune a GBM model video and its accompanying notebook; they are included here so you can use these hyperparameters in your own work. Remember that using fractions of the data to train each base learner can possibly improve model predictions and certainly speed up training times.

Key takeaways

When building machine learning models, it's essential to have the right tools and understand how to use them. Although there are numerous other hyperparameters to explore, the ones in this reference guide are among the most important. Be inquisitive and try different approaches. Discovering ways to improve your model is a lot of fun!

Tab 17