

Machine Learning Foundations

Unit 3

Table of Contents

- Unit 3 Overview
- Tool: Unit 3 Glossary

Module Introduction: Introduction to Model Training

- Watch: Training With the Goal of Generalization
- Read: Overfitting and Underfitting
- Read: Training and Test Data Sets
- Read: Hyperparameters
- Watch: The Core Sklearn API
- Module Wrap-up: Introduction to Model Training

Module Introduction: Implement K-Nearest Neighbors

- Watch: Introduction to KNN
 - Read: An Overview of KNN
 - Tool: KNN Cheat Sheet
 - Code: KNN Demo
 - Watch: Distance Functions
 - Read: Common Distance Functions
 - Assignment: Computing the Euclidean Distance
 - Activity: Find a Decision Boundary
 - Watch: KNN Optimization
 - Read: The Curse of Dimensionality
 - Read: Enhancing KNN
 - Assignment: Optimizing KNN
 - Code: One-Hot Encoding of Features
-

- **Module Wrap-up: Implement K-Nearest Neighbors**

Module Introduction: Implement Decision Trees

- Watch: Information Theory Overview
- Read: Information Theory Formalized
- Quiz: Check Your Knowledge: Entropy Scenarios
- Watch: Building a Decision Tree
- Watch: Making Predictions Using the Decision Tree
- Tool: Decision Trees Cheat Sheet
- Quiz: Check Your Knowledge: Decision Trees Part 1
- Quiz: Check Your Knowledge: Decision Trees Part 2
- Watch: Optimizing a Tree
- Read: Model Complexity Revisited
- Assignment: Optimizing Decision Trees
- Assignment: Unit 3 Assignment: Building a DT After Feature Transformations
- Assignment: Unit 3 Assignment: Written Submission
- Module Wrap-up: Implement Decision Trees
- Assignment: Lab 3 Assignment

Unit 3 Overview

Video Transcript

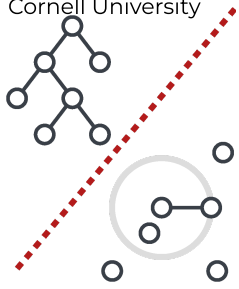
We are now ready to dive into the core of machine learning, which is the process of creating predictive models from data. So far, we've discussed supervised learning, which is where we built a model using labeled data. In this section, we will cover three core methods used in supervised learning. These methods are k-nearest neighbors, decision trees, and logistic regression. Our goals will be to understand the core mechanics of these methods, but also gain practical knowledge on how to build and optimize them. Each of these three algorithms has a unique way in which they are trained. I'll show you how to set this up and train them using scikit-learn. In addition to the mechanics of these core algorithms, we'll discuss a few other fundamental concepts that apply to supervised learning. One such concept is loss functions, which are functions that quantify how well a model is performing or fitting the data. Additionally, we'll cover model complexity and how it leads to overfitting. Overfitting is a model failure mode that causes model generalization performance to be poor. We will learn how to control overfitting by adjusting model hyperparameters, where hyperparameters are algorithm specific inputs that control how the model is built. Each of our core algorithms has different hyperparameters and choosing them carefully is a key to getting good generalization performance.

What you'll do:

- Define the core foundational elements of model training and evaluation
- Develop intuition for different classes of algorithms
- Analyze the mechanics of two popular supervised learning algorithms: decision trees and k-nearest neighbors
- Develop intuition on trade-offs between different algorithmic choices

Unit Description

Over the next few weeks, you will be introduced to the next steps in the machine learning life cycle: model training and evaluation. You will focus on the model training



and evaluation process for supervised learning models, and will explore a few supervised learning algorithms that are commonly used.

Recall that supervised learning attempts to discover the relationship between features and some associated labels for the purpose of prediction. This week Mr. D'Alessandro introduces the model training

for two popular supervised learning algorithms: k-nearest neighbors (KNN) and decision trees (DT). Although both models can be used for regression and classification problems, the focus of this week will mostly be on their applicability to classification problems. You will practice creating your own machine learning models using a popular Python package for machine learning called scikit-learn.

Note: Throughout this program, you may often see classification models referred to as "classifiers." Recall that in classification, the labels are discrete or categorical values. You may often see "labels" referred to as "classes" or "class labels."

[Back to Table of Contents](#)

Tool: Unit 3 Glossary

Most of the new terms you will learn about throughout the unit are defined and described in context, but there are a few vital terms that are likely new to you but undefined in the course videos. While you won't need to know these terms until later in the unit, it can be helpful to glance at them briefly now. Click the link to the right to view and download the new terms for this unit.

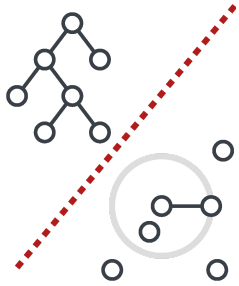


Download the Tool

Download and save this **Unit 3 Glossary Tool** to use as a reference as you work through the unit and encounter unfamiliar terms.

[Back to Table of Contents](#)

Module Introduction: **Introduction to Model Training**



In this module, Mr. D'Alessandro will introduce the training and evaluation process for supervised learning models. Recall that the goal is to train a model that can generalize well to new data. You will familiarize yourself with new terms, key concepts, and the methods used in the training process with the goal of generalization in mind.

You will also be introduced to the scikit-learn package, a key package for doing machine learning in Python. This package supports techniques for supervised learning and will make programming your model much easier.

[Back to Table of Contents](#)

Watch: Training With the Goal of Generalization

The goal of the training process is to create an accurate model that generalizes well to new data. There are a few things to consider in order to reach that goal.

First, you want to minimize loss, or the measure of how many mistakes your model made. Loss is a way of measuring the difference between a training example's label and the label that the model predicts. The goal is to produce a model in which there are little to no prediction errors, and therefore zero to no loss. This measure is often referred to as training loss or training error.

Second, you want to avoid one common problem that may occur when training your model with the goal of having zero to no loss. While a model may successfully have a low training loss, it may adapt so well to training data that it cannot generalize to new data. This is called overfitting. The goal is to create the best model possible without overfitting.

In this video, you will be introduced to these core concepts. And please note that logistic regression will be further discussed next week.

Video Transcript

We are now ready to start formally learning the algorithms behind supervised learning. There are several algorithms we'll learn. Each varies in how they learn a prediction model, but they all have the same goal in the end, which is to be able to generalize to new data. In this course, we won't cover all the possible algorithms to learn. I chose a core set with two goals in mind. The first goal is to learn the fundamental conceptual building blocks that will ultimately help you better understand any learning algorithm. These building blocks are represented by the three algorithms and the less complex group here, which are k-nearest neighbors, decision trees, and logistic regression. The second goal is to pick up practical experience on the algorithms that are most generally used in practice. These are represented in the more complex group here, which are random forest, gradient boosting, and neural networks. Before we discuss any one algorithm, I want to introduce a few concepts that are important for all of them. I have mentioned several times that the goal of machine learning is generalization, which is the model's ability to adapt to new previously unseen data. For supervised learning, I can make this more explicit. Instead of saying adapt to new data, I can say, we want to be able to

accurately predict the label in previously unseen data. This idea of generalization can be expressed mathematically. The tool we use to do this is called the loss function. A loss function is a specific mathematical expression that measures how good your predictions are. The most common loss function is the idea of error, i.e. on average, how often is your prediction exactly equal to the label? We will use loss functions in the model building process, which is called training, as well as in evaluation. There are several loss functions to choose from, and the one to use will generally be driven by the problem. Our general goal with supervise learning is to optimize the loss function. Depending on the specific loss function, this can be a minimization or maximization problem, but there is a very important distinction to make. We don't want to fully optimize the loss function on the data we have or the data that we've observed. Instead, we want to optimize the loss on data that could come from the same data generating process that created the data we have. This is called the expected loss. This is a bit abstract, so, let's illustrate this a bit. The plot on the left represents a theoretical data distribution. This is the data that could be generated from a data generating process. A data generating process is any process that is creating the data points you see. On the right, we have our observed data sample. These data points come from the same distribution as the left, but we should assume it is finite. When we build a model, we try to optimize the loss on the training sample. There is a possibility that we optimize it so well, though, that when we apply the model on the theoretical data and actually doesn't perform well. The idea of applying the model on the theoretical data might seem like just a thought exercise but it isn't. When we implement the model and apply our new data points, we can think of those data points as being part of the left-hand distribution here as opposed to the right. Thus, the real goal is to optimize the loss on new data. Here is another illustration of this concept. In this case, we have a scatterplot with a classification label as represented by the color of the points, the line representing our supervised learning model, and the goal has been to optimally separate the two labels. If this data represents our training data, then the green line makes perfect predictions. At face value, this might seem great, but when new data comes in, we might observe that this green line fit too much of the nuances or the noise of the specific training dataset. The black line is a lot smoother and we might observe in practice that it is more accurate on the new data. The green line represents a concept we call overfitting, and I'll define it again. When we overfit, our model adapts to the training data so well that the model does not generalize well. Pretty much everything we've learn from this point on has the core goal of creating the best

model possible without overfitting the data. This is essentially how we will approach the goal of generalization.

Note: At 1:53, Mr. D'Alessandro says "Depending on the loss function this can be a minimization or maximization problem." The goal of supervised learning is to minimize a loss function rather than maximizing it.

[Back to Table of Contents](#)

Read: Overfitting and Underfitting

The goal of supervised learning is to build a model from known labeled examples to predict outcomes, or labels, from unlabeled data. As you have seen, generalization is a machine learning model's ability to accurately predict new, unseen data. A good machine learning model generalizes well to new data.

There are two failures that can occur when training your model: your model can "overfit" the training data, or "underfit" the training data. Overfitting occurs when the model is too complex, and underfitting occurs when the model is too simple.

What is overfitting?

Just because a model performs well and has low training loss (training error) on training data does not mean it will also perform well with new, unseen data. If the model learns the idiosyncrasies that are particular to **only** the training data set, it will fail to generalize to new data. This is known as overfitting.

The model has learned relationships among features and labels that are too specific to the training data only, and therefore cannot be used to accurately infer anything about unseen data. This happens when a model is too **complex**, which allows it to fit extremely well against all detailed information that is thrown at it. While this may sound like a good thing, it is not. Although training loss may be low, the model becomes so fixated on the training data that it may not be able to make accurate predictions on unseen data, as the model makes decisions based on patterns which exist only in the training data and not in the broader data distribution.

Consider this example. You are training your model to recognize trees. You would start off by giving it a bunch of images of various trees during training. In this case, you

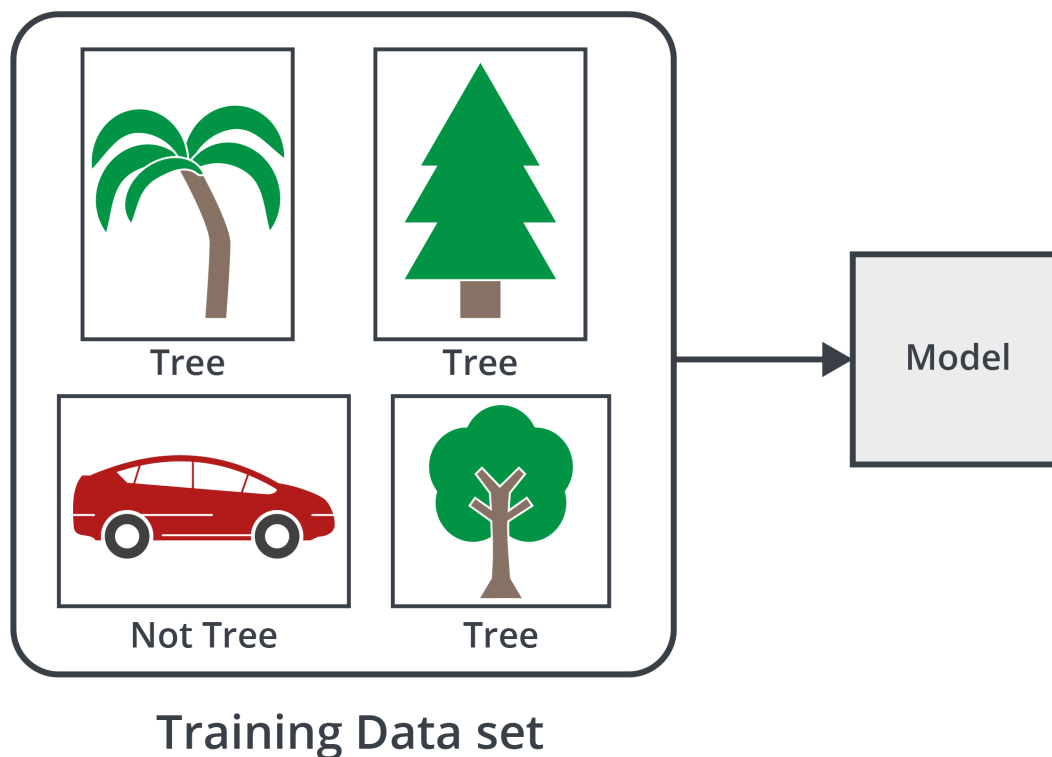
☆ Key Points

Having a low training loss does not always mean that your model will generalize well to new data.

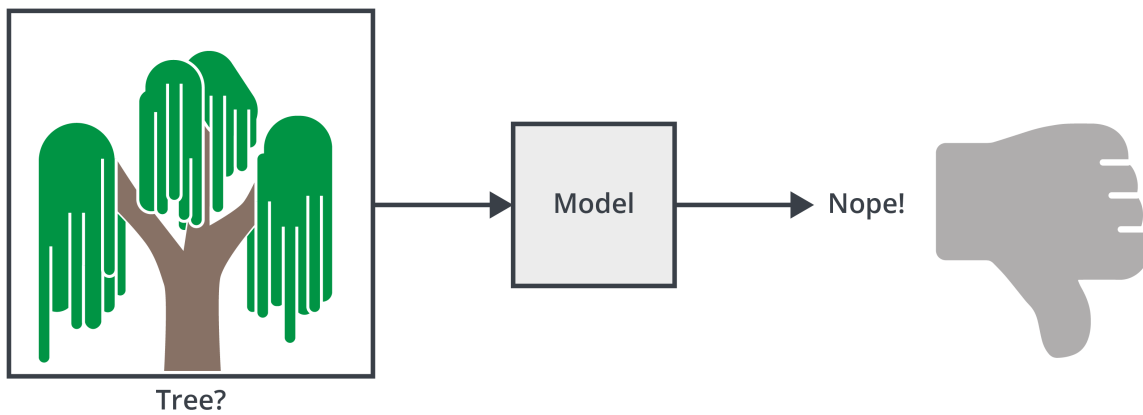
Overfitting occurs when a model is too complex; an overfit model has low training error but poor generalization.

Underfitting occurs when the model is too simple; an underfit model has high training error and poor generalization.

were able to obtain various images of palm, oak, and spruce. Your goal is to have your model recognize green leaves and a brown trunk, so that next time a different kind of tree with these characteristics shows up, your model will be able to confidently predict it as a tree.



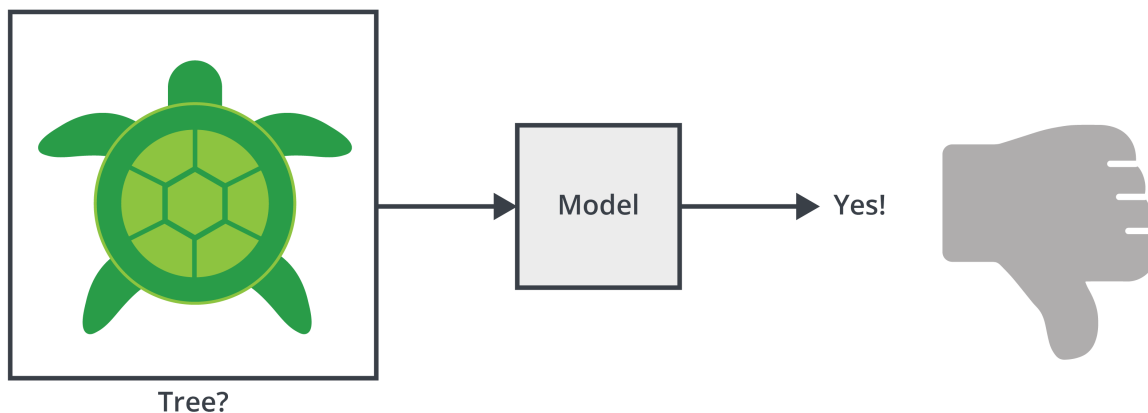
If your model is overly complex, it will start to learn characteristics such as leaf shape, trunk shape, and the height-to-width ratio of these trees from the training data set. As such, your tree recognizer, in this case, becomes very good at recognizing just palm, spruce, and oak. If the model sees a willow tree, it will predict it as a non-tree even though it also has green leaves and a brown trunk.



What is underfitting?

While an **overfit** model performs well on training data, but generalizes poorly to new data, an **underfit** model never performs well on training data from the beginning. An underfit model has high training error and therefore can't generalize well to new data either. In the case of an underfit model, it is too **simple** and has not captured relevant details and relationships among features and labels necessary to make proper predictions. Therefore, it will be unable to make predictions on new data and will perform poorly.

In the tree recognizer example, an underfit model may have only learned to recognize a tree as anything that is green.



[Back to Table of Contents](#)

Read: Training and Test Data Sets

How to avoid overfitting?

You have seen that due to overfitting, a low training error does not necessarily mean that your model will perform well on new, unlabeled data. How can you properly evaluate your model's performance and prevent overfitting? To evaluate your model's ability to generalize well to new data, you have to test your model on data that your model has not seen before. However, how can you test your model on new data if you only have the data set your model was trained on? A common technique in machine learning is to split your original data set into two partitions: training data and test data.

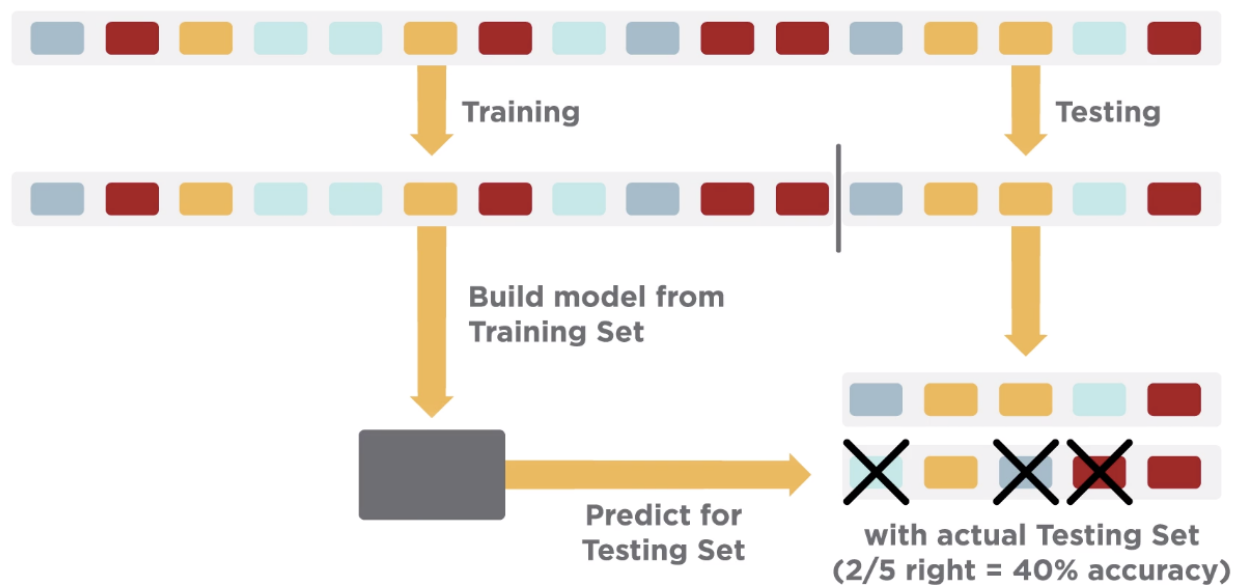
- **Training set** — a partition to train a model
- **Test set** — a partition to test the model

You will train your model on the training data set and evaluate the model on your test data set. The beauty of this technique is that since you will be making your test set from your initial labeled data set, you will have both the features and the labels. This means you will have all of the predicted values in advance. You will test your model on the features, and compare your model's predictions with the labels. For example, in a classification problem, you will be able to see what fraction of the email examples were accurately classified as "spam" emails. In a regression problem, you will be able to see the difference between the predicted and actual values of housing prices.

☆ Key Points

Splitting your data set into training, validation, and test sets allows you to properly evaluate your model's performance and prevent overfitting.

Split Full Data Set Into a Training Set and a Test Set



There are pitfalls in using the test set over and over. After evaluating your model's performance, you may find that you must improve your model. However, you cannot tweak your model, then train and test again using the same test data set. Continuing this process will inadvertently change the model to do well on the test data set, and lead you to overfit your model on the test set. Testing more than once on a test set may incorrectly lead you to believe that your model will perform well on unseen data. Since you should only use the test data set **once**, you can also use a validation data set to improve your model before testing.

Therefore, your data set should be initially split into three partitions:

- **Training set** — a partition to train a model
- **Validation set** — a partition to validate the model's performance
- **Test set** — a partition to test the model

Split Full Data Set Into a Training Set, a Validation Set, and a Test Set



You train your model on the training set, evaluate the model on the validation data set, adjust the model accordingly, and continue this process as many times as needed until you improve your model and the validation error (validation loss) improves to an acceptable level. Once you feel that you are ready to test your model to obtain an accurate assessment of the model's performance, you can then complete an evaluation of your model on the test set. This will allow for an unbiased evaluation of the model's performance.

Note: Training and validation sets are often referred to as just a training set.

Evaluation techniques

There are different ways to partition your data set to make train, validation, and test sets. There are also different ways to analyze your model to determine loss and accuracy during both the training and validation phase. These techniques can vary per model. Some of these techniques are:

- loss functions
- evaluation metrics

You will cover these techniques in a later unit. In this unit you will begin by using just two sets: a training data set and a test data set. You will train the model on the training set, and evaluate the model on the test set using a very simple evaluation metric built into scikit-learn for classification models: accuracy. While the measure of loss and accuracy are not the same, the accuracy metric you will use is essentially equivalent to the zero-one loss function used to evaluate binary and multi-class classification models. Both count the number of predictions that are correct and incorrect.

[Back to Table of Contents](#)

Read: Hyperparameters

One way to avoid the problems of overfitting and underfitting is to incorporate a validation phase after model training. In the validation phase, you can evaluate your model's performance, make necessary changes, and then retrain the model. One key mechanism of a supervised learning model that can be adjusted during the validation phase is a model's hyperparameters.

Hyperparameters are the knobs that programmers tweak in machine learning algorithms. The best way to think about hyperparameters is to view them as the settings of an algorithm that can be adjusted to optimize performance, just as you might turn the knobs of a radio to get a clearer signal. You choose hyperparameter values prior to commencing training. In order to improve a machine learning model's performance after training, a model's hyperparameters may be adjusted during the validation phase. The model will subsequently go through the training process again with these adjustments. You will read in further detail about what hyperparameters are and why they are so important.

What is a hyperparameter in a machine learning model?

A model hyperparameter is a configuration that is external to the model itself. Hyperparameters declare the mechanics of the model, such as its complexity, and determine how the model is trained, such as how fast it learns.

Hyperparameters are often:

- Used to adapt a model to a particular setting

☆ Key Points

Hyperparameters are the settings of an algorithm that can be adjusted to optimize performance; it's the same idea as turning the knobs of a radio to get a clearer signal.

To ensure your model is not susceptible to overfitting or underfitting, you want to tune the model's hyperparameters to some optimal values.

A machine learning engineer's job is to experiment with various hyperparameters values and evaluate the models to ensure that the models developed are not overfitted or underfitted.

- Specified by the practitioner
- Set using heuristics
- Tuned for a given predictive modeling problem

Hyperparameter optimization

When creating a machine learning model, you'll be presented with design choices as to how to define your model architecture, including a model's hyperparameters. Often, you won't immediately know what the optimal model hyperparameters should be for a given model. You cannot know the best value of a model hyperparameter for a given problem and cannot estimate these values from data. To find the optimal hyperparameters for your model, you may use common approaches, copy values used on other models, or search for the best value by trial and error. You can explore a range of possibilities. When a machine learning algorithm is tuned for a specific problem, essentially you are tuning the hyperparameters of the model to discover the model that results in the most accurate predictions. This is known as hyperparameter optimization.

Examples of model hyperparameters

Below are some hyperparameters you will encounter throughout the course:

- **Size of neighborhood** in KNN
- **Depth of tree** in decision trees
- **Learning rate**, or step size, in gradient descent

[Back to Table of Contents](#)

Watch: The Core Sklearn API

One of the reasons Python is the number one language for machine learning is the wealth of available open-source packages out there. One of the packages commonly used in the machine learning community is scikit-learn (sklearn for short). Not only does sklearn contain a comprehensive collection of classes and methods for performing common machine learning tasks, but it is also well-documented. In this video, Mr. D'Alessandro will go over the core API in sklearn, including model instantiation, model fitting/training, and model prediction and evaluation.

Video Transcript

I think it is fair to say that great software is one of the reasons machine learning has become so popular. The discipline has been around for several decades in academic circles. The emergence of the tech industry has also brought the data and use cases that were perfect for machine learning. In the early 2000s, it still took deep and specialized knowledge to be a machine learning engineer. This is because the software we have today did not exist. If you wanted to build a model, you often had to build the model training code from scratch, and this significantly slowed down our limited execution progress. There are now many great software options available today, but arguably the most influential has been scikit-learn and Python. In this video will introduce the core APIs that make this software so powerful and popular. Scikit-learn has a very wide range of algorithmic options covering regression, classification, and unsupervised learning. It also provides rich libraries for data preparation, model selection, and evaluation. One of the most powerful design features of the package is the consistency and ease of use of actually training a model. Independent of the algorithm, scikit-learn has reduced modeling to three core steps. These are model specification where you instantiate the model class and tell scikit-learn how you want to configure the specific algorithm, then is model fitting, which is another way to say training the model. At that point, you have your model. When you want to use the model for evaluation or prediction purposes, the third step kicks in. Here are the same steps represented as actual code. There are four lines here because you have to import the specific algorithm first, of course. In the first line I am creating a model object from the model class. Across the different algorithms, this is where the code will be different because each of the core algorithms will have its own class as well as different input parameters. Next, we call the fit method. We pass in the data which

consists of the features represented by X here and the label represented by Y. DF here is just a variable that holds a data frame. This is where scikit-learn runs the actual optimization routines. This method doesn't return anything, but under the hood, the model object is now filled with data that represents the actual model. At this stage, the training is complete. When we want to use the model to either evaluate or make actual predictions, we call the predict method. We'll cover this more later, but this particular method returns the predicted class. If you want to predict a probability score, for instance, you'll want to use a different one. When we build models, we commonly want to try out different algorithms. Scikit-learn makes this particularly easy with its common API. In this code example, we can call the training and a loop. First, we import the different algorithms. Notice that each one comes from a separate submodule of scikit-learn. I then created a dictionary here where each element is a different algorithm. After that, I call the fit method within the loop. The common API gives us the ability to automate the process of testing and choosing different algorithms. This is a process called model selection that we'll discuss in more detail later. Despite some common APIs, scikit-learn still has a lot to remember. I personally don't remember it all after many years of using it. Even if I know the name of the class like I need, I still might forget the exact names of the input parameters, or what the defaults are. The scikit-learn documentation is quite robust and accessible though. A common sequence in your workflow will be to Google scikit-learn plus the name of the action you want to do. The documentation usually comes up first, and I think scikit-learn has some of the best documentation of any Python package out there. As you get more advanced, the source code is linked to the documentation itself. So if you want to really understand how an algorithm works, there's no better thing to do than study the actual source code.

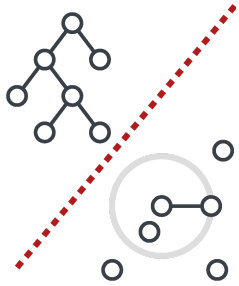
[Back to Table of Contents](#)

Module Wrap-up: Introduction to Model Training

In this module, Mr. D'Alessandro explained the basics for training and properly evaluating a supervised learning model. You explored how to create a model with low training loss and ways to overcome the perils of overfitting your model to training data. Mr. D'Alessandro also introduced a popular Python package known as scikit-learn, a package that is popular in machine learning. You will use this package to implement and evaluate future supervised learning models.

[Back to Table of Contents](#)

Module Introduction: Implement K-Nearest Neighbors



The first supervised learning algorithm Mr. D'Alessandro introduces is the k-nearest neighbors (KNN) algorithm. KNN is a simple algorithm in that it can make predictions about an example based on the labels of other examples "near" it.

KNN can be used in regression or classification problems. For regression problems, it can make predictions for a *continuous* label Y . In this case, the predicted value is the average of the Y labels for the k-nearest neighbors. For classification problems, KNN can be used to classify a *categorical* label Y . In this case, we look at the distinct Y categories for the k-nearest neighbors, and the category that occurs most often is our predicted label (class).

In this module, Mr. D'Alessandro will focus on how k-nearest neighbors is used in classification. You will explore the k-nearest neighbors classifier and see how it is used to classify examples and explore its applications and limitations.

[Back to Table of Contents](#)

Watch: Introduction to KNN

In this video, Mr. D'Alessandro introduces you to your first machine learning algorithm known as k-nearest neighbors (KNN). KNN is straightforward to implement and is therefore often used to introduce the key components of a machine learning algorithm. Watch as Mr. D'Alessandro explains how this model works and what it means for KNN to be an instance-based learning algorithm.

Video Transcript

Here, we'll cover the learning algorithm called k-nearest neighbors or KNN for short. KNN is unique amongst the algorithms you'll commonly use. It's an example of what we call instance-based learning. Nearly all other algorithms can be represented by a parameterized data object. These objects typically take the form of equations or logical tree-based structures. KNN, on the other hand, is represented by the training data. As a matter of fact, one doesn't technically train a KNN model. Predictions are made at prediction time and involve just a query on the original training examples. To illustrate how KNN works, let's start with a common picture. Here we have a scatter of two features called X_1 and X_2 . Each point represents a different example with those two features. Each example has a class label which we represent with the color. You can see here that most of the green points tend to cluster to the upper right, but we also see some overlap between the two classes. Recall that all of our learning algorithms work by attempting to find a reasonable separating line between the two colors. With KNN, we start with the point we are predicting. In this case, it's the large red dot. When we look for the K closest points using some distance formula that we choose, this is where the k-nearest neighbor name comes from. Once we have found the K closest points a separating boundary is essentially a circle around our point, and the radius would be the distance to the farthest point in that set of k-neighbors. Once we have our neighborhood and a set of points within it, we make a prediction by looking at the class labels of the neighbors. Just as it is illustrated in this chart, to make a prediction for a single example, we can ignore all points outside of the circle. Our predictions can be the most common class if this is a classification problem; we can also take the averages of the label. This means we can use the exact same algorithm for regression problems, or classification problems when we're trying to predict a class label probability instead of a class label alone. This process will be repeated for every new example we are predicting. Each

example will get its own custom neighborhood and the prediction will be based only on the points in that specific neighborhood. In this example, we can see how label values tend to be concentrated in different regions of the feature space. Pretty much all classification algorithms operate under this basic assumption. The more that labels cluster, the better performing our algorithms likely will be. KNN is the most explicit algorithm capturing a fundamental concept in supervised learning. This concept is that examples that are near each other in the feature space are likely to have the same label.

[Back to Table of Contents](#)

Read: An Overview of KNN

Most supervised learning models are represented by some underlying data structure derived from training examples, such as the tree structure that is produced by the decision tree supervised learning algorithm. This is known as model-based learning. Another type of supervised learning is known as instance-based learning. Instance-based learning models simply store training examples in memory and utilize those examples on-demand to make predictions for a new, previously unseen example. K-nearest neighbors, KNN for short, is one such algorithm that belongs in instance-based learning. One way to think of it is that there really is no such thing as training in instance-based learning. All that you are doing is making the data available at prediction time, so you can consult the data on the fly as you make predictions.

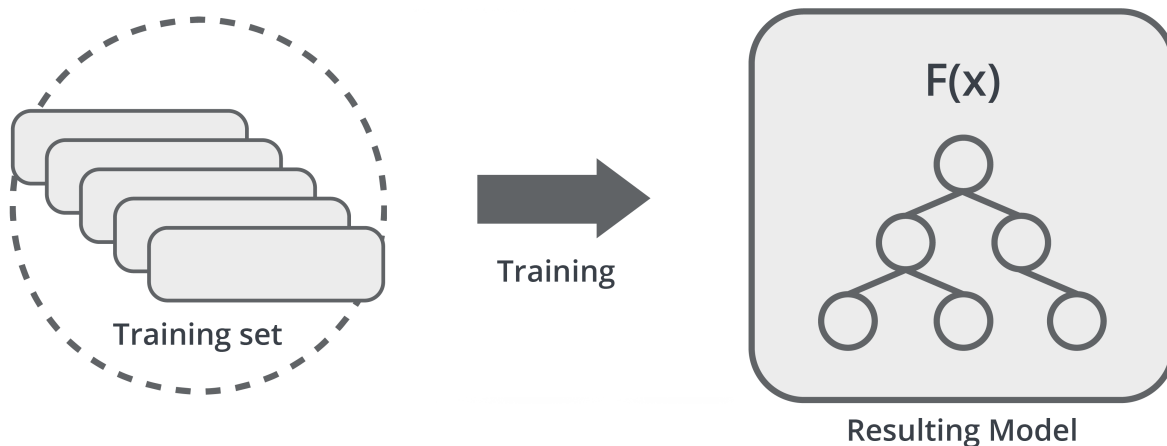
☆ Key Points

KNN is an instance-based type of supervised learning model that stores training examples in memory and uses them to predict the label of new examples.

Distance functions are used to measure similarity (nearness) between two points.

K is a hyperparameter chosen by the machine learning engineer; it can be determined based on domain-knowledge, heuristics, or experimentation.

Model-based Learning



Instance-based Learning

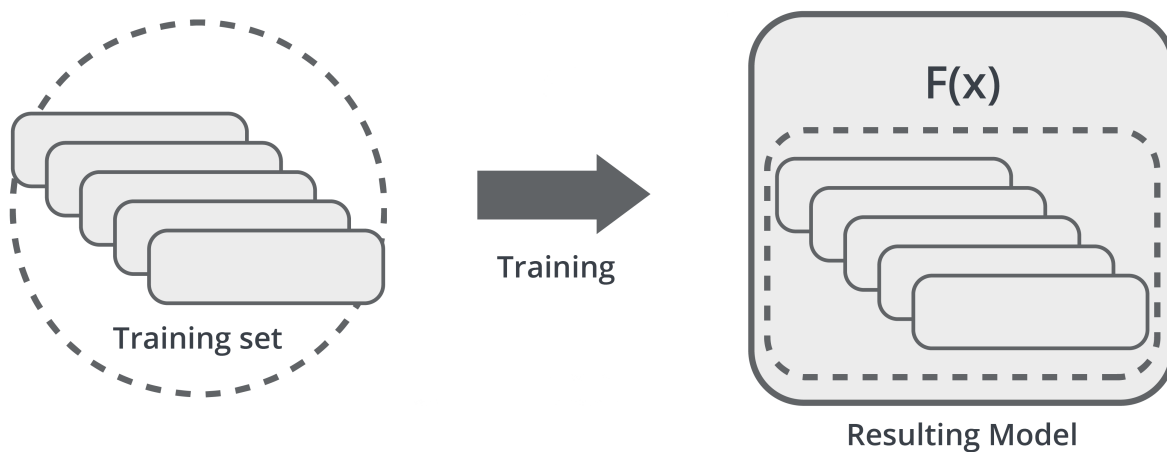


Figure 1. Model-based learning vs. Instance-based learning

Take a look at an example of KNN in action. In Figure 2 below, blue and red markers represent training examples containing two features — x_1 and x_2 . When a new, unlabeled example comes in, as indicated by the green marker, how should we classify it? We first choose a size for K (K =number of nearest neighbors), in this case three. Then, we find the K closest examples to our green marker and look for the most common class label in this group. Since there are two blue markers and one red marker, we determine that our green example should be classified as blue, given that it is the most common marker in this group.

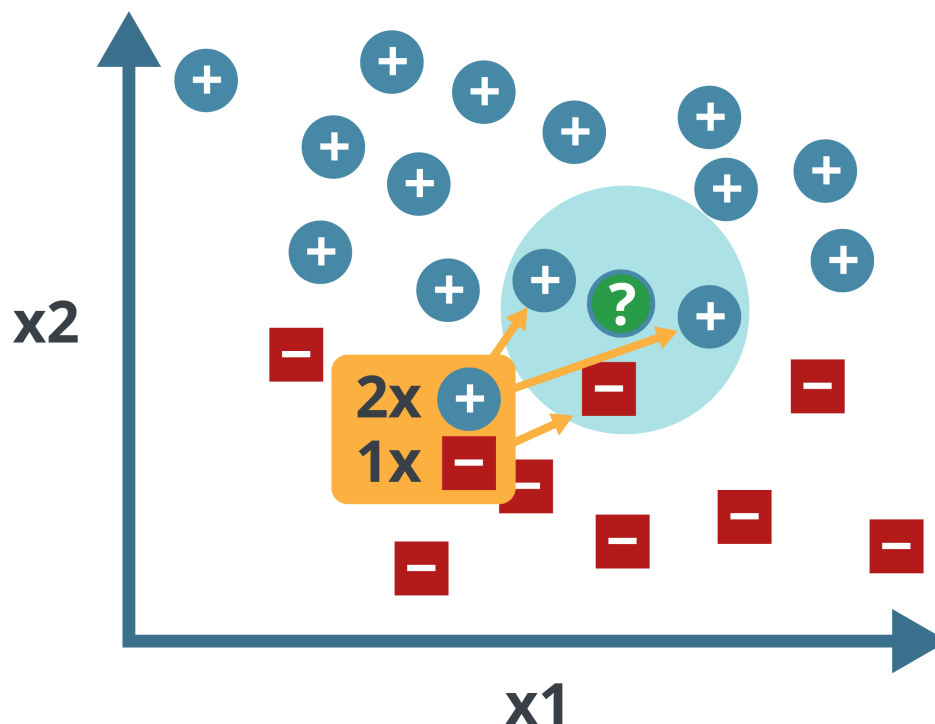


Figure 2. KNN example

In KNN, the K is also known as a **hyperparameter**, and represents the number of neighboring examples that will be used to make a prediction. This number is usually determined based on domain knowledge, heuristics, or experimentation, though experimentation is recommended. Too low of a K or too high of a K can both lead to poor performance. You will get into more details on hyperparameter tuning in the later lessons.

Another concept is the notion of closeness or similarity. In our example above, we use Euclidean distance as our similarity measure. However, there are other methods for measuring similarities such as Manhattan distance and Mahalanobis distance. These similarity measures are known as **distance functions**, which you will go over in greater detail in the subsequent sections.

To summarize, KNN is an instance-based learning that stores training examples in its memory. When a new, previously unseen example comes in, KNN first looks for the K most similar examples based on some distance function and makes the prediction based on the labels of those examples.

[Back to Table of Contents](#)

Tool: KNN Cheat Sheet

The tool linked to this page provides an overview of the k-nearest neighbors algorithm for your reference. You'll find information about the applicability, underlying mathematical principles, assumptions, and other details of KNN.



Download the Tool

Use this **KNN Cheat Sheet** when referring to algorithm parameters.

[Back to Table of Contents](#)

Code: KNN Demo

In this activity, you will see how to implement a KNN model using scikit-learn. You will practice how to create training and test data sets from a raw data set, use the training data to train a KNN model, and use the test data set to evaluate your model's performance. You will work with a popular data set, the **Iris Data Set**, which is used to classify the species of an iris flower.

Note: This week you will use just two sets: a training data set and a test data set. You will train the model on the training set, and evaluate the model on the test set. You will not use a validation data set.

[This activity will not be graded.](#)

This content cannot be rendered properly in the course transcript. Please log into the course to view it.

[Back to Table of Contents](#)

Watch: Distance Functions

The core of KNN lies in the concept of distance function. Distance function allows KNN to evaluate the similarity between two different data points (examples). In this video, Mr. D'Alessandro goes over how distance functions work in detail and its role in KNN. He further explains how to compute common distance with a demonstration of Euclidean distance.

Video Transcript

K-nearest neighbors is probably the simplest supervised learning algorithm from a mathematical perspective: The most important mathematical elements are what we use to evaluate the idea of nearness. We do this using a special type of function called a distance function. Distance functions are typically defined between two points — let's start there. The concept is easiest to illustrate in two dimensions, but everything we discuss here applies to any number of dimensions. When I use the word dimension here, I simply am referring to the number of features in your data. The simplest way to think of distance is the length of a straight line connecting two points. The most commonly used distance function is called Euclidean distance. The distance function represents the concept I just mentioned, which was the straightest distance between two points. If you look at the formula, it should be actually pretty familiar. This is none other than the Pythagorean theorem. This formula is easily generalized to multiple dimensions. Again, there are a few other distance functions that are relevant for k-nearest neighbors, but Euclidean distance is the most common starting point. Once you learn the different options and circumstances in which you may use each one, you can specify the distance functioned when building the KNN model in scikit-learn. As shown here in the scikit-learn documentation, the parameter for distance function in the code is called metric. The choice of distance metric can be another type of hyperparameter you can test to ultimately find your best model.

[Back to Table of Contents](#)

Read: Common Distance Functions

There are many common distance functions that are used to compute the similarity of two data points (examples). The optimal distance function for a problem can be selected based on domain-knowledge, heuristics, and experimentation. You will begin with three distance functions commonly used in KNN: Euclidean, Mahalanobis, and Manhattan distance.

Euclidean distance

Most people are familiar with Euclidean distance, as it is what is used to calculate the hypotenuse of a triangle. The equation below is the generalized form of Euclidean distance function for a n -dimensional feature space where x_i^a is the i -th feature of a given example A and x_i^b is the i -th feature of another example B that you are comparing against.

$$d(A, B) = \sqrt{\sum_{i=1}^n (x_i^b - x_i^a)^2}$$

☆ Key Points

Euclidean distance is the most straightforward distance metric, but it does not account for the correlation and scale among features.

Mahalanobis distance can be thought of as an improved version of Euclidean distance that accounts for both correlation and scale among features.

Manhattan distance is another common distance metrics that is calculated by taking the sum of the absolute difference between data points.

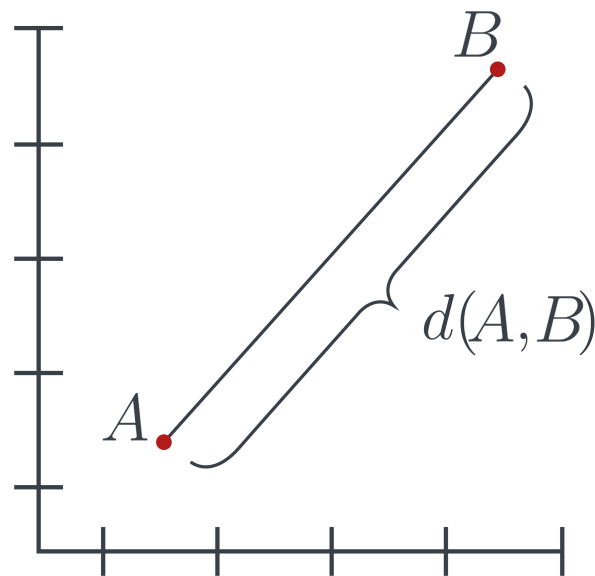


Figure 1. Euclidean Distance

Euclidean distance is straightforward to work with and familiar to most people. However, it does not account for covariance and scale among features. If you want to account for these among features, you will need to use something more elaborate such as Mahalanobis distance. If you need a refresher on what covariance is, please refer back to the "Correlation, Covariance, and Mutual Information" page in Module 2.3.

Mahalanobis distance

Mahalanobis can be thought of as an improved version of Euclidean distance in order to account for correlation among features and scale. Here S is the covariance matrix. If S is the identity matrix, meaning that there is no correlation among any of the features, then Mahalanobis distance simply reduces to Euclidean distance.

$$d(A, B) = \sqrt{(\vec{x}^b - \vec{x}^a)^T S^{-1} (\vec{x}^b - \vec{x}^a)}$$

In the example below, even though the Euclidean distance between red/blue and between green/blue are the same, green would be considered a lot closer to blue in Mahalanobis distance, since it lies along the first principal component of the dataset. The first principal component is basically the direction of the highest variance. Intuitively speaking, it is the direction where data is most stretched out. Principal component analysis is a useful technique under feature engineering, but it is outside

the scope of this course. You are encouraged to read up on it when it is convenient for you.

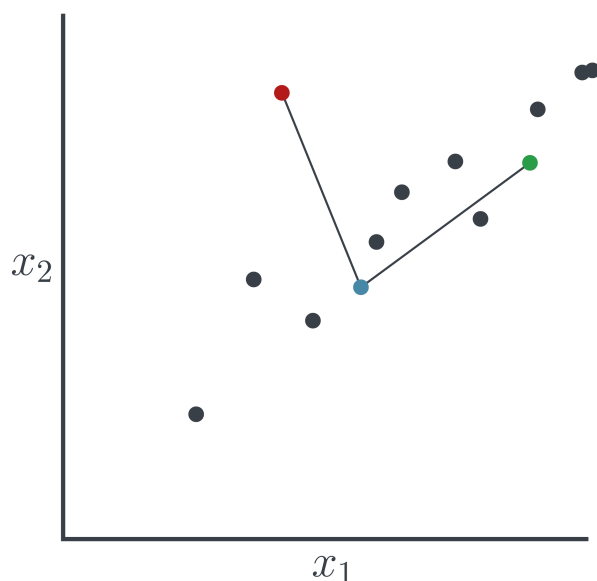


Figure 2. Mahalanobis Distance

Manhattan distance

Manhattan distance takes the sum of the absolute difference between each feature. It is also known as the taxicab metrics due to its likeness to the distance traveled in the city with grid layout.

$$d(A, B) = \sum_{i=1}^n |x_i^b - x_i^a|$$

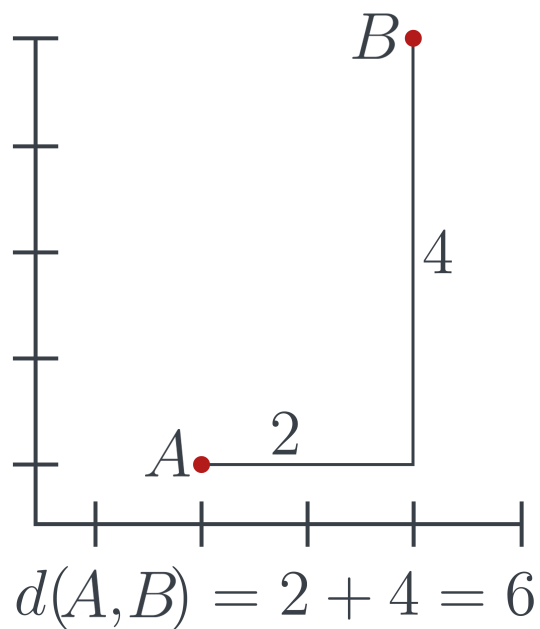


Figure 3. Manhattan Distance

Summary

The type of distance to use for a given problem can be determined based on domain-knowledge, heuristics, or experimentation. Oftentimes, it is worthwhile to experiment with various distance functions to find the one that yields the best performance.

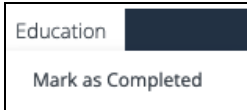
[Back to Table of Contents](#)

Assignment: Computing the Euclidean Distance

In this exercise, you will see how the Euclidean distance is used to find the k-nearest neighbors of an unlabeled example. You will write a function to find the Euclidean distance. In this assignment and for the remaining activities and assignments in this unit, you will work with a new data set called **Cell2Cell**, which is used to predict whether a customer will remain with its current telecom service.

This exercise will be graded.

When you finish your work:

1. Save your notebook by selecting the "Save and Checkpoint" entry from the "File" menu at the top of the notebook. If you do not save your notebook, some of your work may be lost.
2. Submit your work by clicking **Education** → **Mark as Completed** in the upper left of the Activity window.A screenshot of a web interface showing a dropdown menu. The menu is open, displaying two options: 'Education' (which is the parent menu) and 'Mark as Completed' (which is the selected option).
3. After submission, the Jupyter Notebook will always remain accessible in the first tabbed window of the exercise. To reattempt the work, you will first need to click **Education** → **Mark as Uncompleted**, then proceed to make edits to the notebook. Once you are ready to resubmit, follow steps one and two.

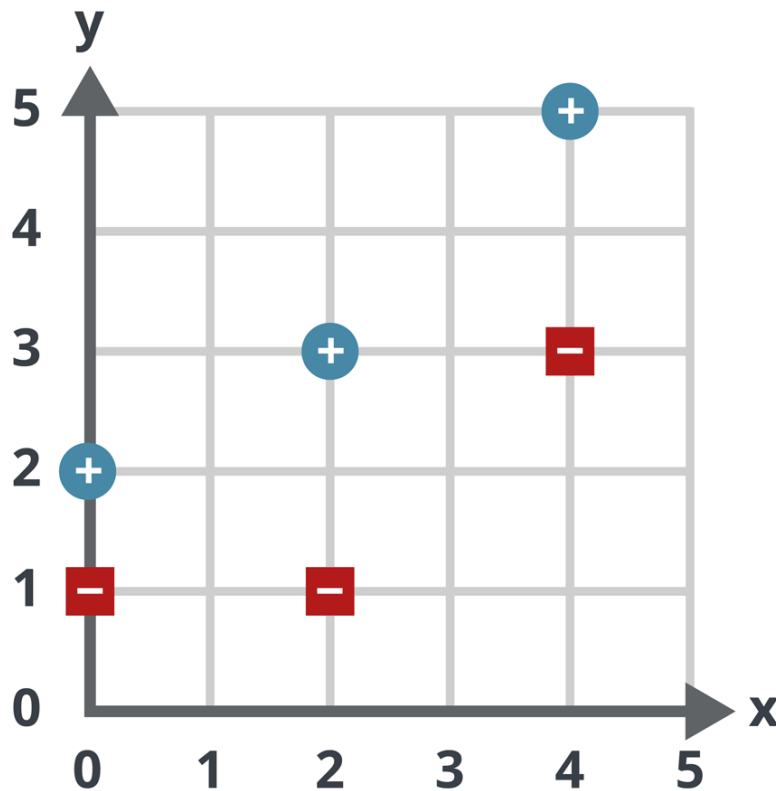
This exercise will be auto-graded.

This content cannot be rendered properly in the course transcript. Please log into the course to view it.

[Back to Table of Contents](#)

Activity: Find a Decision Boundary

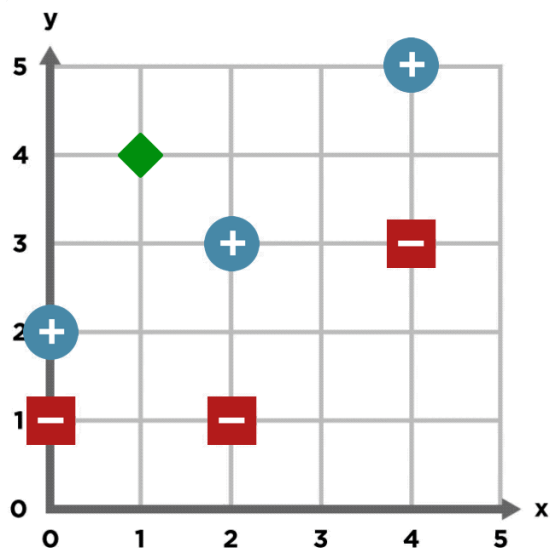
The KNN decision boundary is influenced by the number of nearest neighbors and the position of points from each class. In the case of binary classification using 1-NN (1-nearest neighbor), the decision boundary for a set of data might look like this:



Distance metric

Note that your choice of distance metric also influences the determination of the decision boundary: If you choose Euclidean distance as in the example above, the shortest distance between two points is along the line that connects them.

Alternatively, if you choose a different distance metric, such as taxicab (Manhattan) distance, you could only travel along vertical and horizontal axes, much like a taxicab driving through a city grid. For example, you need to determine the taxicab distance between the green diamond at (1,4) and the positive labeled data point at (0,2) on the graph below. To get to the point (0,2) from (1,4), you have to take 2 steps down and one step left. Thus, the taxicab distance between these points is 3.



Determining the boundary by hand

One trick to finding the boundary for 1-NN (1-nearest neighbor) by hand is to find two points of opposite class that you suspect are near the boundary (e.g., the + at (2,3) and the - at (2,1)). Their midpoint (2,2) is likely to be on the boundary. You can continue to look for points whose two closest neighbors are of opposite classes; their midpoint will also be on the boundary.

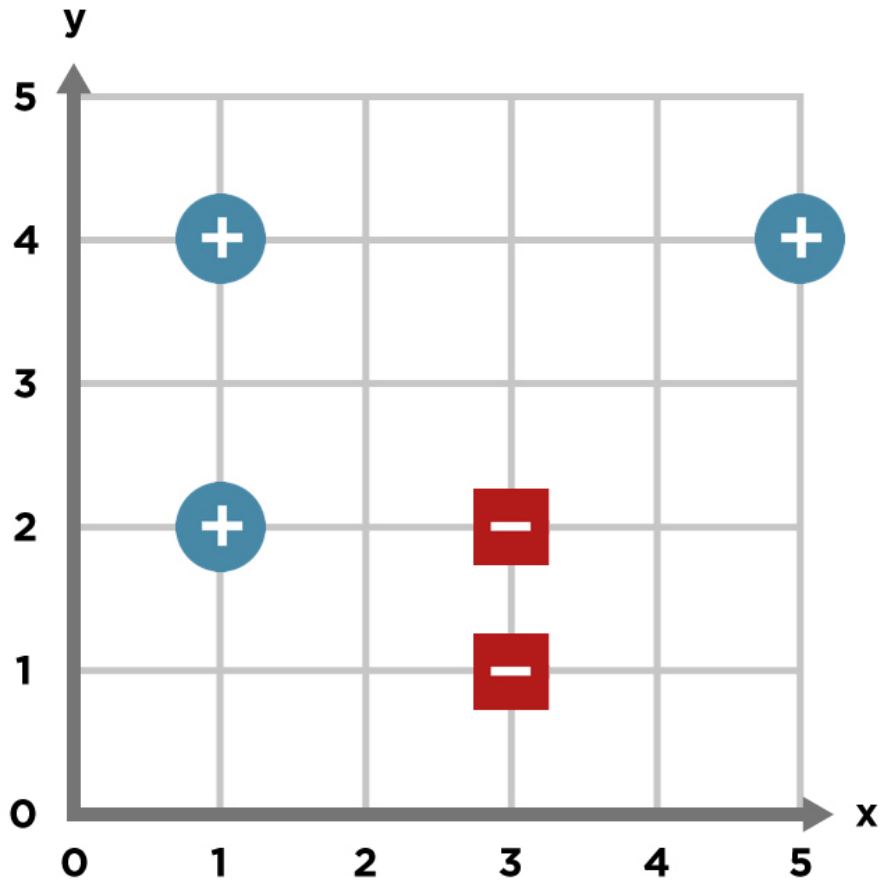
Activity instructions

Suppose you have a data set and you want to draw a decision boundary. You have chosen to use 1-NN (1-nearest neighbor) and you will use the Euclidean distance metric. You have the following data points, strictly confined within a $[0, 5] \times [0, 5]$ grid:

- Class +: (1, 2), (1, 4), (5,4)
- Class -: (3, 1), (3, 2)

Answer the following prompts. When you are ready, click the **Show Solution** button below to reveal the answers.

1. Find the decision boundary for the data displayed in the graph below (using 1-nearest neighbor). You may find it helpful to plot these points out and draw the boundary by hand.
2. How would a new test data point (5,1) be classified, given your decision boundary?



Hide SolutionShow Solution
[Back to Table of Contents](#)

Watch: KNN Optimization

In most cases, machine learning algorithms need to be optimized based on the data you are working with. KNN is no exception. In this video, Mr. D'Alessandro will introduce the idea of hyperparameters. Hyperparameters are tunable parameters that are used to improve model performance. Mr. D'Alessandro also explains the idea of model complexity and how it relates to a model's ability to generalize, as well as the functionality of normalization and the effect it has on KNN's performance. You will discover what it means for a model to overfit and underfit in the context of model complexity.

Video Transcript

Every supervised learning algorithm has input parameters that control what we call the complexity of the model. These input parameters defined how flexible the model is in terms of its ability to fit small variations in the data. We'll generally refer to these input parameters as model hyperparameters, and the process of finding the best ones is called hyperparameter optimization. A model's complexity is its ability to adapt to small variations in the data. We run a hyperparameter optimization to choose a complexity that enables us to generalize the best. This idea of complexity is an incredibly important concept, and we'll discuss every algorithm through this lens. The main complexity parameter for k-nearest neighbors is usually just called K or the number of neighbors to be used for prediction. It is advisable to always spend some time to test different options for K and to find a value that is best for your data. There are other design parameters that are important to consider as well. We have choices for distance function, and even within a given function, we have the option to weigh each instance using some function of the distance. Another important design choice is normalization, which is a process to scale each feature so that they have similar magnitudes. We'll discuss why this is so important. As with all hyperparameters that control a model's complexity, we need to empirically test a range of options to see which performs the best. We'll cover exactly how to do this in a different lecture. This picture illustrates three different levels of K. On the left, we have a high value of K. Given a fixed set of data points, increasing K means using points that are farther away. We can see in the left-hand plot that the circle is so wide that we're mixing in too much of the different classes. At the most extreme case, the circle covers all points and we're just averaging over the whole data set. Using far away or unrepresentative

points leads to what we call underfitting. Underfitting is when the model doesn't learn real nuances of the data, and we will also learn later that underfitting is associated with a type of error we call model estimation bias. On the right-hand side, we have a small value of K . This solves the problem we introduced with high K . We can see that within any small circle, the points are more likely to have similar labels, but at the same time, the predictions depend heavily on only a few points. When the data is noisy, which in this case translates to the different colored points being in the same circle, there was a high, random chance that the very closest neighbors to a point are there from the opposite class. This heightened sensitivity to just a few nearby points leads to what we call overfitting, and again, we'll show later that overfitting is associated with a type of model error called estimation variance. The best option is likely to be somewhere in the middle, a smaller circle means the nearby points are likely to be representative of the example being predicted, but we want it to be large enough to not be too sensitive to just a few points. This idea of finding the middle ground will be common in a lot of algorithms we use. Next, let's discuss the idea of normalization and why it's important. Normalization would be a preprocessing step, so you wouldn't specify this when calling the scikit-learn KNN class. Normalization generally improves performance with this algorithm, so I highly recommend doing it. Let's use a simple Euclidean distance example to illustrate the point. In this case, we have age and income as features, and we're computing the distance between two points. Imagine the average age is 40 and the average income is 50,000. The two red arrows start from the same point and jump the same relative distance. The size of the arrows are misleading because each of these features are on different scales. But, in this case, moving 50 percent along the x-axis results in a distance value of 20, but on the y-axis, it is 25,000. So without normalizing, similar relative movements on features that have a larger scale will dominate the distance function. Let's also see this another way. Going from age 40 to 60 is pretty significant, but raising one's annual income by \$20 likely isn't, at least for prediction purposes. With Euclidean distance, these changes are considered equivalent, so the bottom line is that features with higher scale will receive a higher implicit weight in KNN. The distance functions will be more sensitive to these features, sometimes to the point where other features don't even matter. This is generally undesirable, so we normalize to equalize each features contribution to the model. This process of normalization maintains the predictive properties of an individual feature, but changes its distribution. Again, we do this to ensure each feature has a similar scale of distribution, which results in them having equal importance in the distance function. Here is an illustration on how

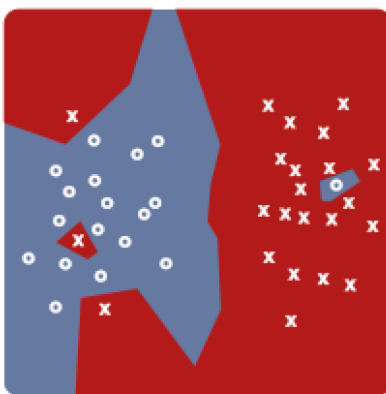
normalization affects the shape of the distribution. We can see that the new distribution is shifted, so its mean is zero, and it also has a lower variance, which is depicted by the width of the bell curves. When we normalize, we perform this shift and rescaling on each features so that the distributions look more like the orange plot here. After normalization, each feature would have a distribution similar to the orange one here, no matter its starting point. This process doesn't hurt the predictive performance because the normalized values still have the same correlation with the label.

[Back to Table of Contents](#)

Techniques for Improving KNN

How can you improve the performance of a KNN model? To optimize KNN, you can experiment with different values of its hyperparameter K (the size of the neighborhood) to find the best model complexity and avoid underfitting or overfitting. You can also optimize KNN by choosing the appropriate distance function. You can even perform pre-processing steps to normalize each feature's contribution to the model, thus avoiding the pitfalls of a distance function weighing some features more heavily than others, resulting in certain features being "ignored." Below, you will see how choosing different values for hyperparameter K affects performance, and some normalization techniques.

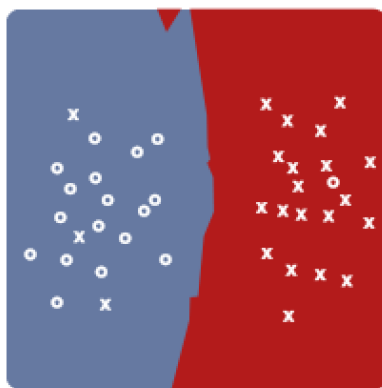
As you will recall, a model's complexity defines how flexible a model is in terms of its ability to fit small variations in data. Hyperparameters are the configurable aspects of your model that can be adjusted to control model complexity and improve a model's performance. The KNN model has exactly one hyperparameter and that is k , the size of the neighborhood that you pick to infer the label. The best choice of k depends on the specific data set. You typically choose odd numbers to avoid ties, especially with binary classification problems, though you can set k equal to any number. So what happens when you change k ?



euclidist $k = 1$

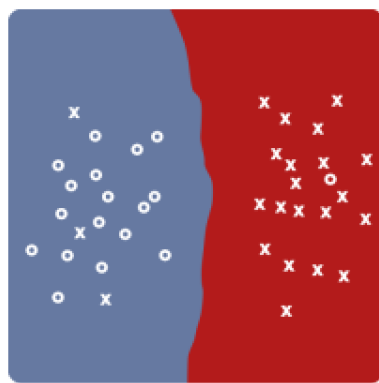
The KNN decision boundary here for 1-nearest neighbor is actually quite rough and picks up on little movements in the data set. Also, you might see “islands” of misclassified and mislabeled examples, as KNN carves out regions around these mislabeled examples. If you suspect your data set is noisy, meaning some points are

misclassified in your data, 1-NN might be too sensitive to accurately generalize to future data.



euclidist $k = 3$

If you increase k to 3, you're now averaging local neighborhoods and the little “islands” get washed out as they are outvoted by the other examples. The complexity of the boundary decreases and becomes smoother with 3-NN.



euclidist $k = 9$

The decision boundary becomes smoother and less susceptible to noise as you increase k further. In this example, the decision boundary of 9-NN is well defined and even smoother than 3-NN. You may, however, find 9-NN begins misidentifying significant groups of data points, which becomes more likely as you increase the number of neighbors used.

Some machine learning models are sensitive to the range of feature values: KNN is one such algorithm. For example, say you want to predict whether a loan will default based on a person's age and income level. A typical range for age will be between 18 and 100, while income level can be upward of thousands to hundreds of thousands. In

this case, income level will have a much higher influence in our distance calculation than age does. In Figure 1, you would typically expect red and green dots to be considered closer to each other than blue and green. However, based on un-normalized values, our distance function determines that blue and green are actually closer than red and green.

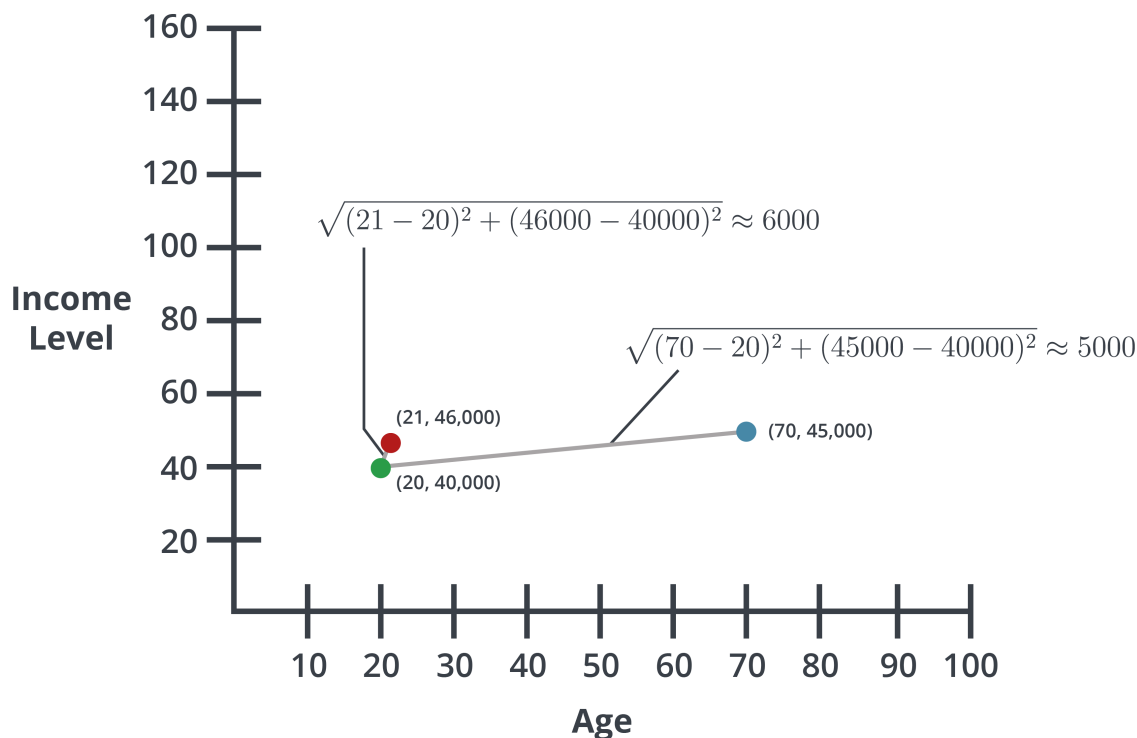


Figure 1. Loans with two features, age, and income level.

To combat the behavior illustrated above, introduce the technique of normalization. Normalization essentially transforms the values of features into the same scale. This allows distance function to give equal weight to each feature. The two most common normalization techniques are standardization and min-max normalization.

(Note: some literature uses normalization interchangeably with min-max normalization).

Standardization

In standardization, you transform the values within a feature to have a mean of 0 and a standard deviation of one. You do this by subtracting the mean for each feature and then dividing this by the feature's standard deviation. Another name for

standardization is standard scaler. Standard scaler typically works well with features that follow normal distribution and are less sensitive to outliers.

$$x_{i,normalized} = \frac{x_i - \mu}{\sigma}$$

where μ is the mean and σ is the standard deviation

Min-max normalization

In min-max normalization, you transform the values within a feature to be between certain min/max range. Here, subtract the min value of the feature from the example and then divide this by the range of the feature. Another name for min-max normalization is min-max scaler. Min-max normalization typically works well with features that don't follow normal distribution and are more sensitive to outliers.

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

Summary

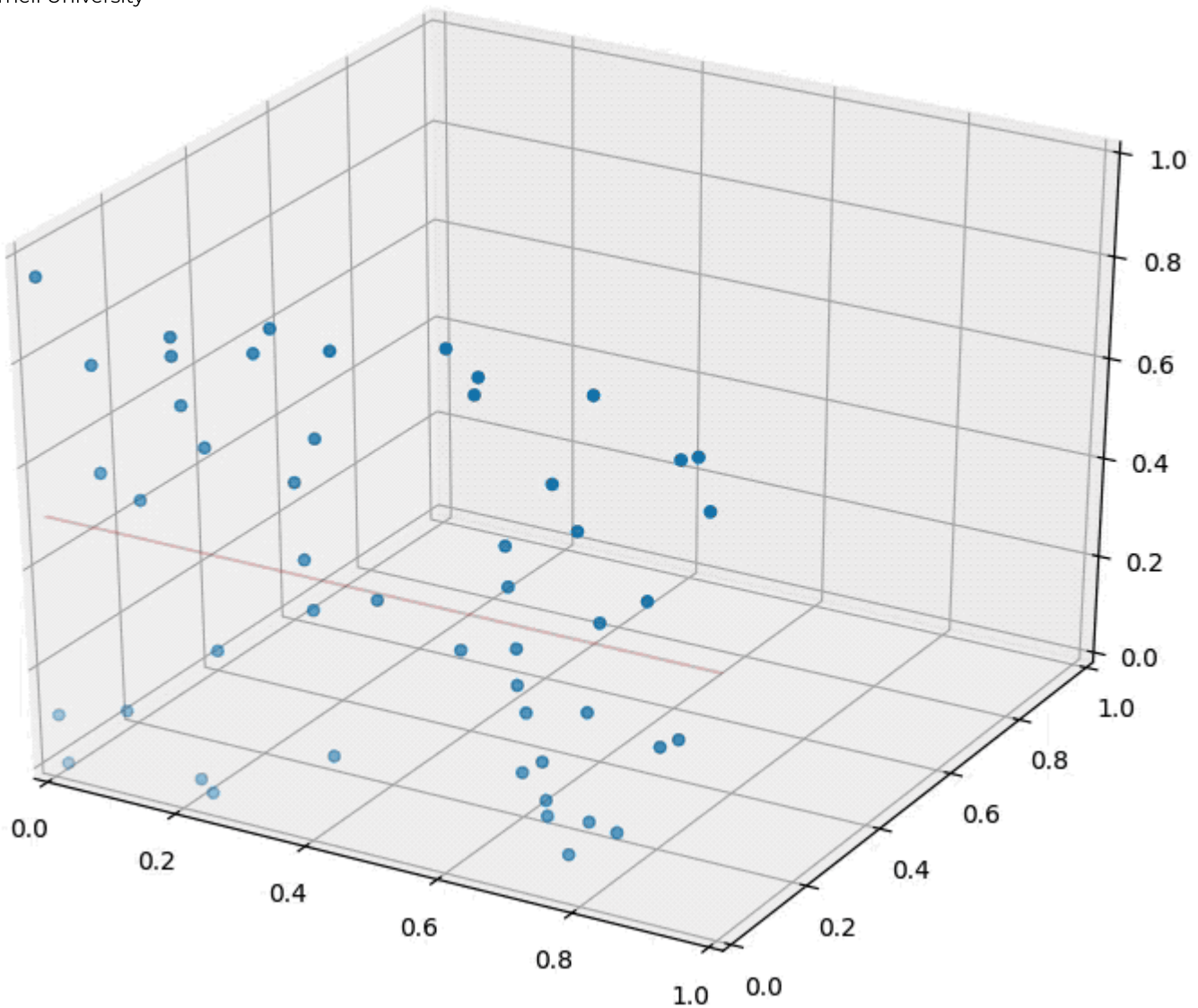
Many machine learning algorithms perform better when features are normalized. The two most common normalization methods are standardization and min-max normalization. While it is possible to implement these normalization methods from scratch, sci-kit learn comes prepackaged with StandardScaler class and MinMaxScaler class for such purpose.

[Back to Table of Contents](#)

Read: The Curse of Dimensionality

High dimensional data is data that contains a large number of features. Note that when referring to the number of dimensions, you are referring to the number of features. High-dimensional data is a particular challenge for the KNN model. As the number of dimensions increases — that is, as you include more and more features in your data set — all of your data points (or examples) become more unique and less similar to one another. Eventually, your data points are so dissimilar that the approach of finding close neighbors to predict the label of a test data point is no longer feasible. This is known as the curse of dimensionality.

To better appreciate this behavior, visualize how the high dimensionality impacts your data set. The animation below shows what happens to examples with two dimensions after adding a third dimension. This data set is randomly sampled, but a similar effect can be observed on real-world, correlated data.



In this example, let the true classification boundary be the red line (plane). Points above it are positive, and points below it are negative. Knowing this boundary, you know that the only real feature that matters is the vertical axis.

If you developed a KNN model for the dataset with two dimensions, you could run it on a test data point near the boundary, but still above the line. This test data point may have several close neighbors, all positive, and several nearby-but-less-close neighbors that are negative. The algorithm could predict that this data point is positive without much difficulty.

Once adding the third dimension, however, suddenly all its neighbors are farther apart. Say you are classifying a data point just above the center of the cube. Its closest neighbors may change; suddenly, a negative data point that used to be farther away, but happened to be relatively close to your test data point along the new axis, might be the closest data point.

In general, when increasing dimensions, data points become farther away from each other, and the distances between data points all become larger and less distinguishable. As the data points expand along the third dimension, their relative pairwise distances increase. However, the distance to the red plane (initially a line in 2D) stays constant. This is a typical behavior in high dimensions; randomly sampled data points in high dimensions tend to be spread out from each other (with roughly equal distances) but tend to be comparably close to separating hyperplanes.

[Back to Table of Contents](#)

Read: Enhancing KNN

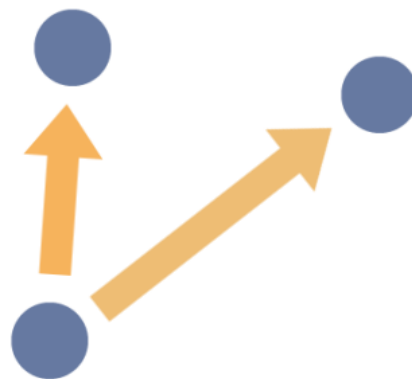
The KNN model can be enhanced in a number of ways.

There can be cases without a unique most common label. For example, if $k=4$, you could retrieve two neighbors of one class and two of another. To avoid this, you typically pick odd values of k . Even in those cases (e.g., $k=3$), however, your neighborhood vote may result in a tie. For example, the image to the right shows our test point (a red star) and three neighbors, all from different classes. A common approach to resolve ties is to fall back onto the majority label within the $k-2$ closest neighbors. In the scenario on the right, since 3-NN would result in a tie, you can fall back to 1-NN and get a definitive label.



Choosing distance function

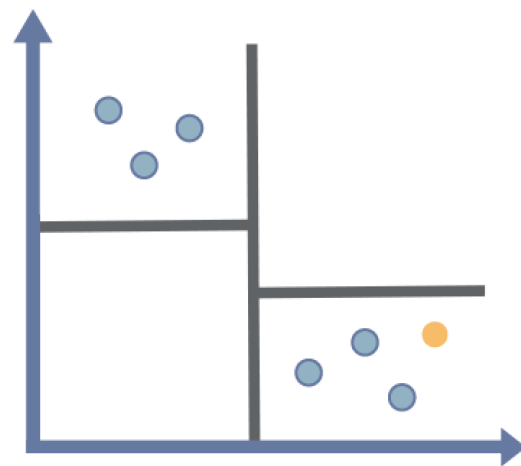
The distance function is a critical component of KNN and has a major impact on the "neighborhoods" derived from your data; how you determine the distance between two points is critical to the accuracy and performance of your classifier. Using the Euclidean distance (also known as L2 distance) with KNN is common but may be suboptimal in some settings where features follow particular structures (e.g., are normalized). Depending on your data set, there may be more suitable distance



metrics, such as the L1 (taxicab) or Minkowski distance.

Data structure for speedup

One downside of KNN is that during test time, you have to compute distances from each test point to every training point. This process becomes more intensive and slower as you increase the size of the data set. One way to speed up KNN is to use data structures such as k-d trees or ball trees. In this example, you see a k-d tree structure that splits the data space into boxes. If you are trying to find the nearest neighbors for the yellow test point in the lower box, rather than computing the distance from our test point to all of the points in the upper box, you simply compute the distance to the box containing those points. You can very quickly determine that the upper box is further away than points closer to our test point, thereby ruling out any points contained in that upper box with a single calculation. Note that Scikit-learn uses this technique when fitting a KNN model.



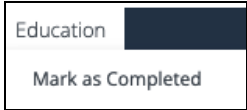
[Back to Table of Contents](#)

Assignment: Optimizing KNN

In this exercise you will implement a KNN classification model using scikit-learn. You will train different models using different values for hyperparameter K and compare the accuracy of each model.

This exercise will be graded.

When you finish your work:

1. Save your notebook by selecting the "Save and Checkpoint" entry from the "File" menu at the top of the notebook. If you do not save your notebook, some of your work may be lost.
2. Submit your work by clicking **Education** → **Mark as Completed** in the upper left of the Activity window.
3. After submission, the Jupyter Notebook will always remain accessible in the first tabbed window of the exercise. To reattempt the work, you will first need to click **Education** → **Mark as Uncompleted**, then proceed to make edits to the notebook. Once you are ready to resubmit, follow steps one and two.

This exercise will be auto-graded.

This content cannot be rendered properly in the course transcript. Please log into the course to view it.

[Back to Table of Contents](#)

Code: One-Hot Encoding of Features

Recall that to implement a k-nearest neighbors model, you must use features that allow you to compute the distance; this means using features that are not of string-value. In the previous exercise, you removed these values from the data set before implementing the KNN model, but there is another way to handle these features with string values. The one-hot encoding feature transformation technique can turn string values (i.e., categorical data) into numerical ones that can be used by the KNN model. In this activity, you will practice using one-hot encoding to transform feature values from strings into numbers.

This activity will not be graded.

This content cannot be rendered properly in the course transcript. Please log into the course to view it.

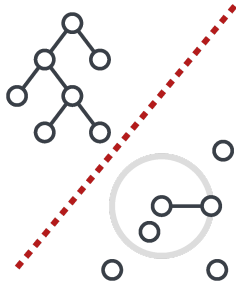
[Back to Table of Contents](#)

Module Wrap-up: Implement K-Nearest Neighbors

In this module, you explored the KNN supervised learning model and its underlying concepts. Mr. D'Alessandro explained the different distance functions that can be used with the KNN algorithm and you learned how to optimize a KNN algorithm for performance. You saw an example of KNN in practice using the Iris data set and you implemented your own KNN algorithm using the cell2cell data set.

[Back to Table of Contents](#)

Module Introduction: **Implement Decision Trees**



In this module, Mr. D'Alessandro introduces another common supervised learning algorithm: decision trees. As is the case with KNN, decision trees can be used for both regression and classification problems. You will focus on how to apply it to classification problems. In addition, you will have the opportunity to investigate how to partition the data into a tree that minimizes training error, and explore techniques to avoid overfitting and underfitting to the training data while maximizing your model's accuracy.

[Back to Table of Contents](#)

Watch: Information Theory Overview

In this video, Mr. D'Alessandro introduces another very popular and versatile machine learning algorithm: decision trees. In order to understand decision trees, you will first need to explore the basics of Information Theory. Mr. D'Alessandro will introduce the idea of entropy and information, both of which are instrumental to how decision trees are built.

Video Transcript

Decision trees are one of the most useful algorithms in the machine learning toolkit. They have a lot of high performing properties on their own, but they're also a fundamental building block for some of the most widely used algorithms in practice. This idea of building blocks will be important, particularly in the study of decision trees. If full tree itself is built using a recursive algorithm. Once we understand the base step, we can essentially understand the entire process. At the root of decision, trees are some powerful formulas that come from a field of study called information theory. These two formulas are called entropy and information gain. These formulas are useful on their own, but they are instrumental in how decision trees are built. We'll start with entropy because this is the more fundamental of the two concepts. Here is a conceptual description along with the actual formula. First, let's start with a discrete random variable. This is the same as a categorical variable. Here are two examples of a variable with four values in different distributions. The bars just show the relative distribution across the four values. One way to think about entropy is it's a measure of dispersion or uncertainty of a discrete random variable. In the formula, each discrete value of the variable would have an associated probability or P , which is represented by the bar height. The total entropy of the variable is the sum of each P times the logarithm of P . The top chart is actually the highest entropy you can measure. This is a uniform distribution over the four values. We say this has the highest uncertainty because predicting where a particular example falls is as good as guessing at random. The bottom is different. We see more mass towards the middle two values. If you were to draw a random example and had to bet its value, you'd be better off betting one of the two middle values. Here, we quantify the entropy of each example and using the entropy formula. For each distribution, focus on the labeled probabilities. Our entropy formula again is a sum of these probabilities multiplied by their own logarithms. The negative sign means this will be a positive number and a higher number means more

entropy. The upper chart is uniform or equal probability for each value. When this occurs again, we have maximum entropy. The other extreme is when all examples have the same value. This would have an entropy of zero. We often don't use entropy on its own. Instead, we use it to compute a quantity called information gain. In words, information gain is a way to measure how much average entropy changes after we segment our data. I'll explain the motivation behind the segmentation later. For now, I want to emphasize the actual formula. We will use H as our function for entropy. The first term represents the entropy on a variable Y on a sample of data. I changed the formula to use Y instead of X here because I want to start thinking of computing entropy and information gain on a label that might be used for supervised learning. In the right-hand side of the equation, the C stands for child and represents a separate segment of the data. I use this notation of parent and child because we are building up the foundations for decision tree, and I want to use the notation common with tree based structures. Let's next assume we split the data into two or more partitions, again, with each partition being called a child of the original data. The second term is essentially a weighted average of the entropy of each partition, with the weights proportional to the amount of examples in each partition. Now, let's illustrate this graphically, this time using scatter plots that are related to binary classification. The left-hand side represents the full data: We call this the parent. The title shows the proportion of points that belong to the positive class, represented by green points, as well as the computed entropy. Now let's say we split the data into two parts to produce the plots on the right. These are what we would call the children of the parent. Notice in total, we have the same exact points, but they're organized in a specific way. To compute the information gain, we compute the entropy of each child and take a weighted average, where the weight is the proportion of points in the child. The difference between the parent entropy and this weighted average is the information gain. The most important concept is not the math here, though. Notice how in each of the children, we have a higher concentration of a specific class label. Ideally, we want the average value of Y in a child partition to be as close to zero or one as possible. When this happens, we say that the node has more purity. From a guessing perspective, imagine you had to place a bet on the value of Y for a randomly chosen point. Absent any information, your accuracy would be as good as the average value of Y on all the data. But, imagine someone gave you a piece of information. This information is the segment you sample the point from amongst the child partitions. With that information, your guessing accuracy would dramatically increase. We call

this increase in certainty the information gain, and this exact step is the base building block of decision trees.

[Back to Table of Contents](#)

Read: Information Theory Formalized

Information theory is widely utilized in the implementation of decision tree-based classifiers. The core concepts of information theory revolve around the idea of entropy and information gain; both of which relate to the idea of uncertainty. You'll explore these concepts in greater detail and how they contribute to the construction of a decision tree.

Entropy

Entropy is essentially the measure of uncertainty in a random variable.

When the value of a random

variable is predictable, the entropy

is low. When the value is unpredictable, entropy is high. For a random variable with a binary outcome, the range of entropy is between 0 and 1. As an example, a fair coin toss has the highest possible entropy of 1 since it is not possible to predict the outcome better than randomly guessing. Conversely, a coin rigged to always show heads has an entropy of 0 since you know it will always toss heads. Mathematically, this can be explained by the following equation, where each y_i is a type of outcome (heads or tails).

$$H(y) = - \sum_{i=1}^n P(y_i) \log_2 P(y_i)$$

For example, if you have a coin that has a 30% chance of flipping heads and 70% chance of flipping tails, then its entropy is calculated as follows:

$$H(y) = - (.30 * \log(.30) + .70 * \log(.70)) = .88$$

For a fair coin:

$$H(y) = - (.50 * \log(.50) + .50 * \log(.50)) = 1$$

☆ Key Points

Entropy measures the uncertainty of a random variable, with the highest uncertainty being 1 and the lowest uncertainty being 0 when the random variable is binary.

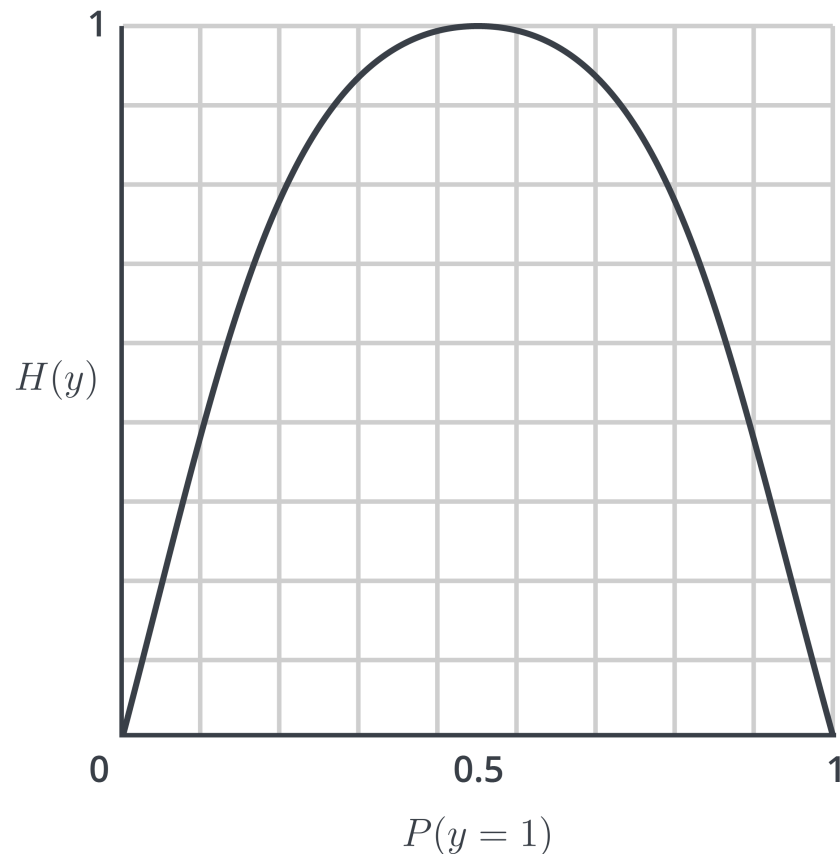
Information gain measures how much entropy has reduced once a certain condition has occurred.

Information gain in the context of a decision tree measures how much entropy has been reduced when a data set is split by some feature value.

For a coin that always flips heads:

$$H(y) = -(1 * \log(1)) = 0$$

If you plot this out for various probabilities with a binary outcome, you get the following plot.



Decision Tree Intuition

So how does this all relate to decision tree and machine learning? Imagine you have a training dataset with binary labels as shown in Figure 2, where there are the same number of red dots as there are blue dots. Without considering the positioning (feature values) of the dots, all the information we have to go by is that half of them is red and half of them is blue. When a new, unlabeled point is introduced, our best chance of predicting its color correctly can be no better than 50%. The intuition behind training a decision tree is to have an algorithm that automatically looks at all

the points and subsequently grouping these points into regions where the dots are primarily red and primarily blue - hence reducing the overall entropy of the system.

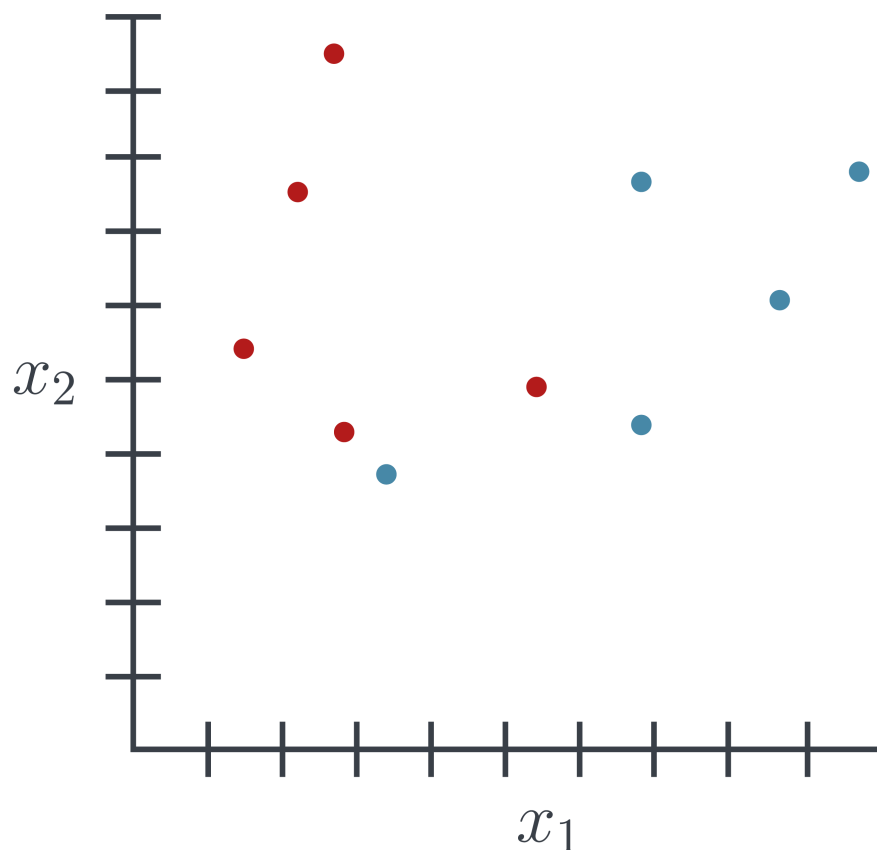


Figure 2. A training set with 5 red labels and 5 blue labels

To separate points into regions, what a decision tree will do is to "draw splits" based on a selected feature value. In Figure 3 below, we explore two split options. The first option is to split by $x_2 = 7$, creating two regions of high entropies, whereby each region still has the same number of blue and red dots. The second option is to split by $x_1 = 6$, which is a much better option as it creates two regions - a left hand side where most points are red and a right hand side where all points are blue. Evidently, option 2 is better as it reduces entropy by a large amount. The amount of entropy reduced is called information gain, and it is what a decision tree use to determine the best feature, and value, to split on.

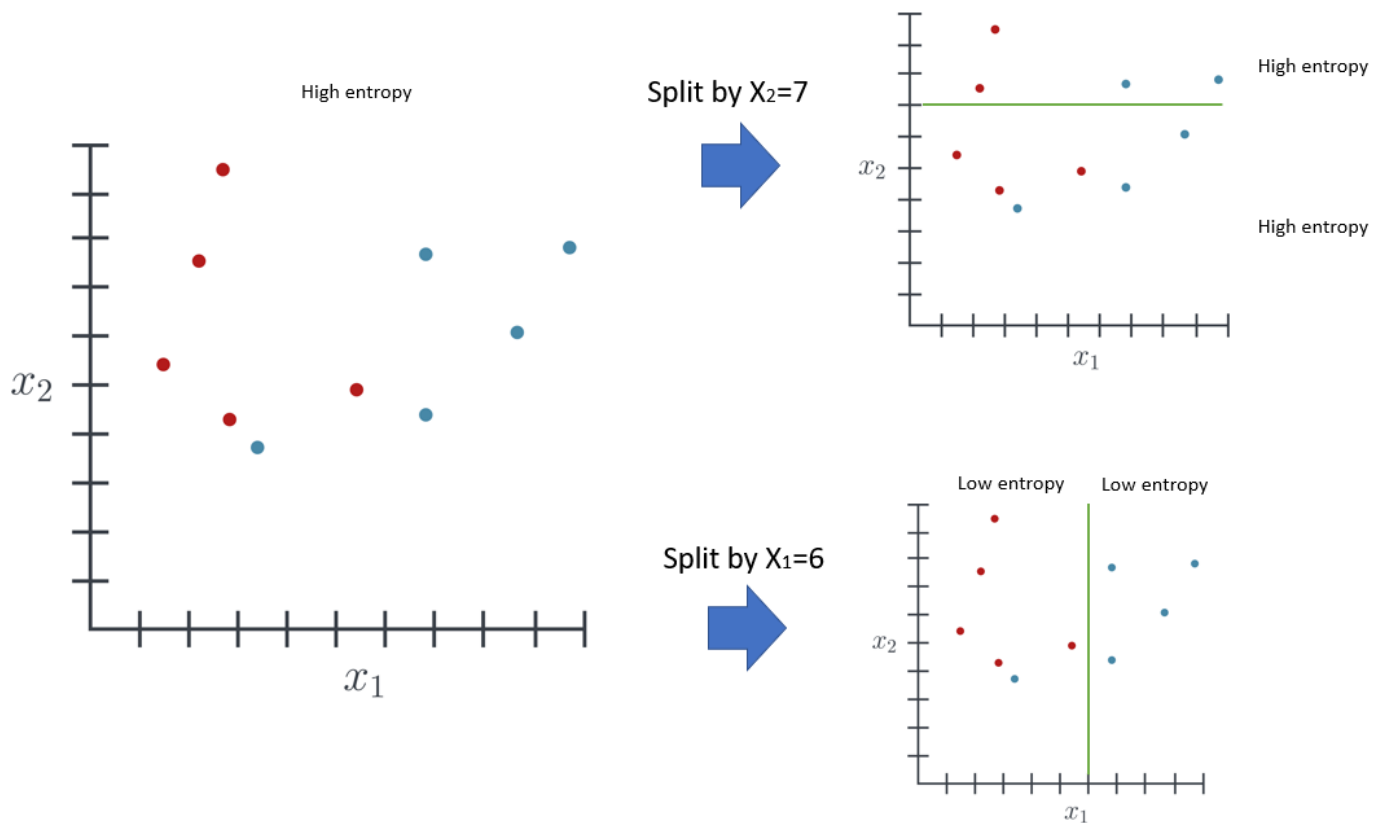


Figure 3. Two example splits by a decision tree.

Information gain

Information gain tells us how much reduction in entropy has taken place once some condition has been met. The condition in the context of a decision tree is a split. The formula for information gain is as follows.

$$IG = H(y|parent) - \sum_{c \in \mathcal{C}} p(y|c)H(y|c)$$

- $H(y|parent)$ is the entropy prior to a split.
- $H(y|c)$ is the entropy of the data points within a child region created by the split.
- $p(y|c)$ is the number of data points in a child region divided by the number of total data point prior to a split.

The specific information gain formula for binary tree is as follows. This is particularly useful as most decision trees are constructed as binary trees. The L and R here denotes the left and right node of a binary tree, which represents the two child regions created by a split.

$$IG = H(y|parent) - p(y|c_L)H(y|c_L) - p(y|c_R)H(y|c_R)$$

- $H(y|c_L)$ and $H(y|c_R)$ are the entropy of each child region - L and R.
- $p(y|c_L)$ and $p(y|c_R)$ are the number of data point in each child region divided the number of data points prior to the split.

Let's walk through an example. In Figure 4 below, we are making a split at $x_1=1$ to create two child regions - c_L and c_R

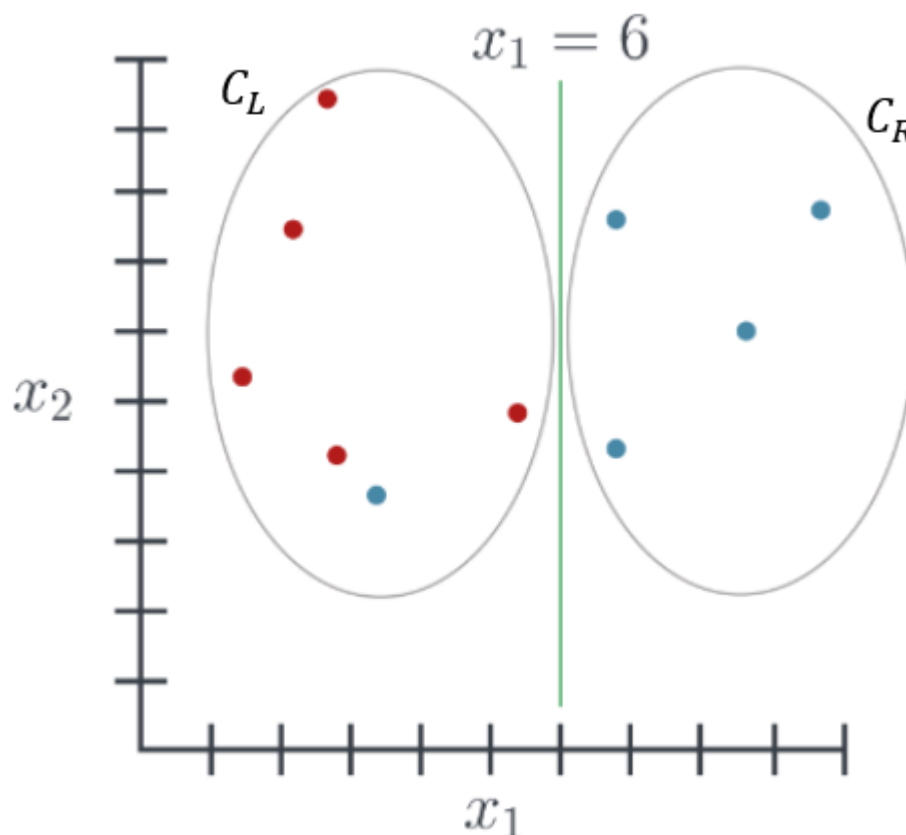


Figure 4. Split by $x_1 = 6$ to create two child regions.

$$H(y|parent) = -\left(\frac{5}{10}\log_2\left(\frac{5}{10}\right) + \frac{5}{10}\log_2\left(\frac{5}{10}\right)\right) = 1$$

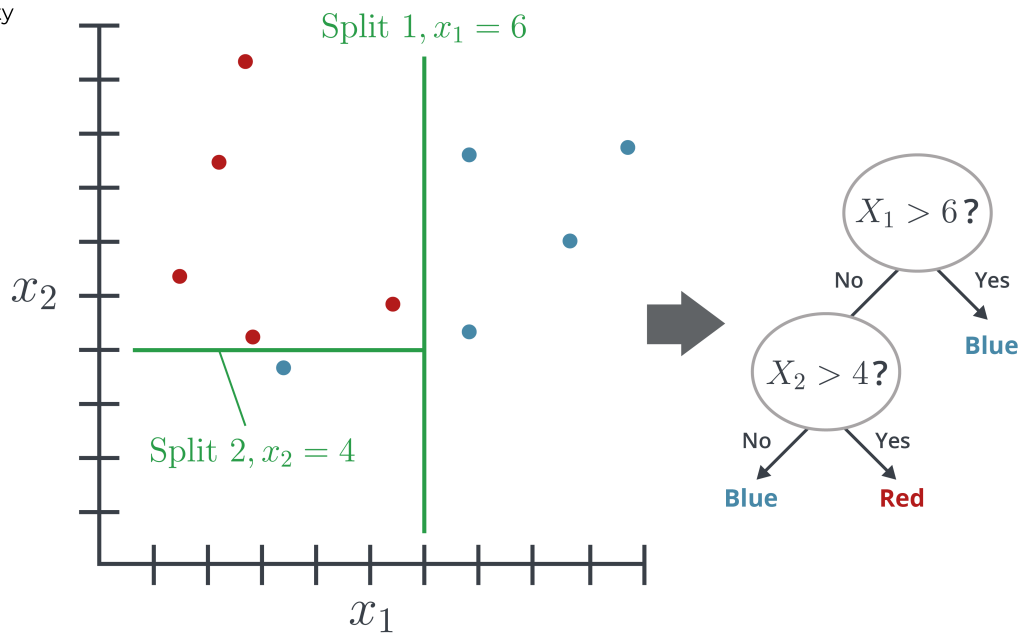
$$p(y|c_L)H(y|c_L) = \left(\frac{6}{10}\right)\left(-\left(\frac{1}{6}\log_2\left(\frac{1}{6}\right) + \frac{5}{6}\log_2\left(\frac{5}{6}\right)\right)\right) = .39$$

$$P(y|c_R)H(y|c_R) = \left(\frac{4}{10}\right)\left(-\left(\frac{4}{4}\log_2\left(\frac{4}{4}\right)\right)\right) = 0$$

$$IG = 1 - .39 - 0 = .61$$

Based on the split above, we arrive at a information gain of .61, which is a fairly high number, considering the highest possible information gain 1.0. Now, when a new, unlabeled data comes in, you simply need to look at its \mathbf{x}_1 value. If its \mathbf{x}_1 is larger than 6, then you would classify it as blue since all the points on the right group is blue. If its \mathbf{x}_1 is smaller than 6, we would classify it as red, since most of the points in the left group are red.

In practice, there can be more than thousands of examples in our training data, and consequently many splits will need to be drawn. For your very simple data set, you would only need two splits to completely separate the examples. When a new, unlabeled example comes in, you will just need to traverse down the tree to determine what the color of the label should be.



In summary, entropy and information gain allows you to find the best splits in your features in order to construct a decision tree. There are other split criteria, such as Gini impurity and Chi-square, although information gain is the most commonly used.

[Back to Table of Contents](#)

Quiz: Check Your Knowledge: Entropy Scenarios

In this quiz, you will calculate the entropy to determine how to build the most effective decision tree, given a specific scenario and data set.

Scenario Setup

Imagine you are building a decision tree to predict whether a personal loan given to a person would result in a **payoff** (i.e., the person pays off the loan) or **default** (the person fails to pay back the loan).

- Your entire data set consists of 30 examples:
 - 16 belong to the "**default**" class
 - 14 belong to the "**payoff**" class
- The examples contain two features, "Balance" and "Residence".
 - "Balance" refers to the amount of money the person has in their savings and checking accounts at the time of the loan, which can take on two values: "< \$50K" or "≥ \$50K".
 - "Residence" refers to whether or not the person owns their home or rents and can take on two values: "OWN" or "RENT".

The entropy H over a leaf containing a set S of examples is $H(S) = -\sum_{k=1}^c p_k \log(p_k)$.

The entropy H over an intermediate or a root node of a set S of examples with two branches S_L and S_R is $H(S) = \frac{|S_L|}{|S|} H(S_L) + \frac{|S_R|}{|S|} H(S_R)$.

If we don't divide the set of 30 examples and treat it as a leaf, the entropy will be:

$$H(\text{undivided}) = -\frac{16}{30} \log_2\left(\frac{16}{30}\right) - \frac{14}{30} \log_2\left(\frac{14}{30}\right) \approx 0.99.$$

To complete the quiz:

Your task is to determine which feature — "Balance" or "Residence" — provides the lowest entropy split. There are two calculation questions to answer.

You may take this quiz as many times as you like.

This content cannot be rendered properly in the course transcript. Please log into the course to view it.

Watch: Building a Decision Tree

Now that you have a better understanding of the basics of Information Theory, you are prepared to explore building a decision tree. In this video, Mr. D'Alessandro will show you the inner working of a decision tree, and how you can construct such a tree by splitting data that maximizes information gain.

Video Transcript

Now we're going to cover the fundamental steps of a decision tree. This involves taking a base segment and finding a split of the data that maximizes our information gained. We'll start with data representing a binary classification problem with two features. Our ultimate goal is to split this data into as many partitions as we think is necessary to get the best cumulative information gain. Let's start with the basic structure of a partition. This is deliberately simple. At each step, we only want to split the data based on a single feature. This process looks like the diagram here. The code shows a simple Boolean operation. If the example feature is greater than some value K , then we split to the right node, otherwise, we split to the left node. To build a good predictive model, we want to find both features and values of K that lead to the best possible increase in information gain. The base step in the decision tree algorithm is to determine which feature to split on and what is the best split value K . The process used to answer this question is illustrated in the picture. It is somewhat of a brute force approach. We loop through each feature and a range of split values for each of the features. For each feature, we find the split value that maximizes the information gained for that specific feature. This is represented by the red lines. Then, we look at the feature that produces the maximum information gain across each of its splits. In this case, it is the left-hand side or the feature marked X_1 . Again, this algorithm loops over both features and possible split values to find the combination that yields the highest information gain. At this stage, our full sample of data has been split into a left and a right partition. The next step is to loop through each of these partitions and apply the same exact algorithm to each partition. This would leave us with a set of partitions that look like this. In our first split here, we split using the feature X_1 . Now, within each of the new partitions, we split on the feature X_2 , but notice how each partition splits at a different value of the feature. That's because each of these later partition steps are completely independent of each other. Our decision tree building algorithm will continue running this splitting process until some termination criteria is

met. Those specific criteria are actually the input parameters we'll give to scikit-learn when we initialize the class. The process we illustrated so far can be represented as a tree. This is, of course, where we get the phrase "decision tree." This tree represents the conditions we used to split and each node and some key stats in each node. In this example, the F and T represent whether the Boolean condition was met. Remember, these are Boolean statements on the features. In this case, I'm using X_1 and X_2 as generic feature names. P of Y represents the average value of the label in each node. When it comes time to make a prediction, we follow a simple process. Starting from the top, we evaluate the new example against the splitting logic. We proceed down the tree until we reach a terminal node. The prediction here is just the distribution or average of the label at this terminal node. You can see here that without a model, our best guess about the expected value of the label would be 20 percent — which is the average of the label — but, with our well learned tree guided by increasing information gain, we can guess that the expected value is 95 percent. Assuming this tree generalizes well, the accuracy of this prediction would be much higher thanks to the process.

[**Back to Table of Contents**](#)

Watch: Making Predictions Using the Decision Tree

The purpose of a supervised learning model is to make predictions. In this video, Mr. D'Alessandro will demonstrate how to make prediction on an unlabeled example by traversing down a decision tree. You will see how prediction is performed in both regression and classification problems. You will also touch on various hyperparameters for optimizing decision trees, such as minimum samples per leaf and maximum tree depth.

Video Transcript

Now, I want to share some practical elements regarding decision trees. We'll cover the components of a decision tree and provide an example implementation in scikit-learn. Here is a visualization of a simple decision tree. One nice property of decision trees is their inherent interpretability. The tree is defined by a set of logical conditions, which means we can trace the exact set of steps that leads to a specific prediction. However, I should share that in most realistic cases, it will be challenging to actually display the entire working tree. Most trees are produced with dozens of features or more and can easily end up with thousands of terminal nodes, so fitting them into one display will be mostly impractical. Let's dig into the components of the tree now. We can start with what we call the nodes. A decision tree is a type of graph structure, which is why the individual elements are referred to as nodes. This type of directed graph is directed in the arrows indicate the direction of some logical flow. Any starting node is called a parent node, and the endpoint of an arrow is called a child node. Once we get to the end of the tree, the terminal nodes are often referred to as the leaves of the tree. These leaves contain the information we use for predictions. Next, we'll introduce a few key components to find within the tree. The depth of the tree tells us how many splits there are down a single path. When making a prediction on a given point, this is the number of logical comparisons you would have to make. Because each split results in two child nodes, the number of nodes in the tree grows exponentially with the depth. Additionally, each node contains data, and there are a few aspects of that data that are important. The first is the node sample size. This is simply the examples that made it to that node in the training process. Then there is the label distribution. Decision trees can be built for both binary and multiclass classification, as well as regression. The main difference between classification and regression in the algorithm is the function used to define the splits. The distribution of the labels in the leaf nodes is what

determines a prediction. For classification, we can take the most common class label as the prediction, or if applicable, we can take the average value to get a probability. For regression, we would just take the average label within the leaf node. Decision trees follow the standard scikit-learn API. Here we instantiate the model. In this case, I specified values for the key input parameters. The max depth tells us how large we can grow the tree. The other two relate to the number of elements in the nodes. The split parameter says we won't split a node unless we have that many examples in the splitting node. The min sample leaf says we won't split unless each of the child node gets at least that number of examples. These are important elements for optimizing the tree's performance, and we'll cover that in a subsequent lecture. Finally, the last two lines are for predictions. We have two choices. We can either predict the class directly, or we can predict that the probability of being in each class. You can see that there are different methods for each. The choice of whether to predict a class or a probability will be problem dependent.

[Back to Table of Contents](#)

Tool: Decision Trees Cheat Sheet

The tool linked to this page provides an overview of classification and regression trees for your reference. You'll find information about the applicability, underlying mathematical principles, assumptions, and other details of this algorithm.

 [Download the Tool](#)

Use this [Decision Trees Cheat Sheet](#) as a quick way to review the details of how the algorithm works.

[Back to Table of Contents](#)

Quiz: Check Your Knowledge: Decision Trees Part 1

To further explore how decisions trees work, you will complete two exercises that require you to review and make decisions based on some data. The example data consists of "good" and "bad" individuals and some basic characteristics that describe each individual. Using the data, you will answer a series of questions about decision trees.

Take a look at the table below, which contains training data consisting of individuals (inputs), characteristics of those individuals (features), and whether that individual is "good" or "bad" (labels). Answer the questions that follow based on the data.

Training Data

	Mask	Cape	Tie	Ears	Smokes	Height	Class
Batman	Y	Y	N	Y	N	180	Good
Robin	Y	Y	N	N	N	176	Good
Alfred	N	N	Y	N	N	185	Good
Penguin	N	N	Y	N	Y	140	Bad
Catwoman	Y	N	N	Y	N	170	Bad
Joker	N	N	N	N	N	179	Bad

You may take this quiz as many times as you like.

This content cannot be rendered properly in the course transcript. Please log into the course to view it.

[Back to Table of Contents](#)

Quiz: Check Your Knowledge: Decision Trees Part 2

To further explore how decisions trees work, you will complete two exercises that require you to review and make decisions based on some data. The example data consists of "good" and "bad" individuals and some basic characteristics that describe each individual. Using the data, you will answer a series of questions about decision trees.

Take a look at the table below, which contains a few test examples consisting of individuals (inputs) and characteristics of those individuals (features) but without labels. Based on the training data, there is a simple decision tree below that accurately classifies the training data. Use the data and tree to answer the following questions.

Testing Data

	Mask	Cape	Tie	Ears	Smokes	Height	Class
Batgirl	Y	Y	N	Y	N	165	?
Riddler	Y	N	N	N	N	182	?
Fred	N	N	Y	Y	Y	181	?

You may take this quiz as many times as you like.

This content cannot be rendered properly in the course transcript. Please log into the course to view it.

[Back to Table of Contents](#)

Watch: Optimizing a Tree

Just as K is the hyperparameter to optimize KNN, the hyperparameters to optimize a decision tree are minimum samples per leaf, maximum tree depth, and minimum samples split. In this video, Mr. D'Alessandro goes over in detail how each of these hyperparameters is used to control the model complexity of a decision tree.

Finding the ideal hyperparameters helps control the model complexity, which in turn reduces model estimation errors due to bias and variance. Bias error arises when a model is too simple, and hence it is not capable of "bending" itself to fit against training data — a condition known as underfitting. On the other hand, when a model is too complex, it has all the abilities to bend at-will, hence able to fit against the training data too closely — a condition known as overfitting. This will result in a high variance error. Mr. D'Alessandro discusses the bias-variance trade-off and how the ideal trade-off between bias and variance lies in finding the right hyperparameters, leading to a model with balanced complexity.

Video Transcript

Now we'll discuss how to tune a decision tree to get optimal performance. Remember, every supervised learning algorithm has hyperparameters that determine its complexity. With a decision tree, the complexity is related to the size of the tree. Each leaf of the tree is a separate and distinct partition of the feature space. The more leaves or partitions we use, the more the tree is able to learn the nuances of the data. For us, tuning the size of the tree leads to the best generalization. A bigger tree results in more complexity; with more complexity we have a higher likelihood to overfit, and the model will have what we call model estimation variance. With less complexity, we have a higher likelihood to underfit, and the model will have what we call model estimation bias. High variance in a model means it is sensitive to slight variations of the data. For instance, if you were to randomly split your data into two training sets and train a model on each data set, a model algorithm that has high estimation variance will produce very different models between the two training sets. Although each random subset comes from the same distribution, the process of sampling will cause slight variations in those two distributions. This is even worse when you have small sample sizes. On the other hand, a model with higher bias tends to not fit the data well. An example is with curve fitting where you're trying to fit a parabola with a straight line. The right complexity is always a balance between these two extremes of

high and low complexity. The main thing to remember and consider when tuning your tree is how to get more or less complexity with each of the main hyperparameters. In this case, we'll cover: "max_depth," where higher values lead to larger trees and higher complexity; "min_sample_split," where higher values lead to smaller trees and lower complexity; and "min_samples_leaf," where higher values lead to smaller trees in lower complexity. Max_depth, again, controls how many splits we can make, and min_sample_split controls the minimum number of samples and a parent node required to split it. Then, min_samples_leaf dictates how many samples can be in the leaf nodes. Given a large tree will usually be difficult to actually visualize the complexity of the tree, or basically visualize the entire tree, since each leaf of a tree represents a specific partition of the data, we can instead visualize the tree by showing how it partitions the feature space. The graph here shows three different output scenarios for trees trained on the same data. The left-most chart shows the original data for your reference. Each subsequent chart shows what we call the decision surface. The colors of these plots represent how new examples will be predicted. This is binary classification, so we're predicting two classes, represented by the two colors. Essentially, when a new point comes in, we can see its corresponding color and the decision surface, and use that to assign it a label. This visualization is aimed at showing the typical complexity trade off. At a high max_depth, we can see a lot of very small partitions where neighboring partitions predict different classes. Meanwhile, on the left with a very small tree, there isn't much nuance. As a general rule, when looking at decision surfaces like this, we want to see curved, but smooth lines, without having many alternating colors in a small region. When we spend time optimizing our models, we'll empirically test different values of the input hyperparameters and try to test both low and high complex models to find the right fit. Now, I want to make one last point about decision trees that is separate from hyperparameter optimization. This relates to optimizing the performance of the tree in general, but can also be leveraged for learning general insights from the data. The decision tree algorithm in scikit-learn automatically computes an attribute that they call feature importance. The technical definition of feature importance, here, is the cumulative information gain that feature contributes in the learning process. Here's an example of feature importance on some real data. The data and learning problem doesn't matter so much, so I won't get into that, but the plot here shows what the above array stores. By looking at this data, we can quickly determine which features are contributing the most to our predictions. The plot is sorted, and we can see that features on the right-hand side have little to no predictive value. We can use information like this to better

explain the process we're trying to model, as well as to drop features we think aren't adding any value.

[Back to Table of Contents](#)

Read: Model Complexity Revisited

To revisit, there are the two equally problematic cases which can arise when training a model: underfitting and overfitting. If the model isn't able to distinguish important aspects of the training data, it will fail to classify new data accurately. On the other hand, if the model learns the idiosyncrasies that are particular to only the training dataset, it will fail to generalize to new data.

Underfitting: The model is too simple. The model has not captured the relevant relationships and details among features and labels that are necessary to make proper predictions. In this case, both the training error and the test error will be high and the model will not be able to make predictions about new, unseen data.

Overfitting: The model is too complex. The model has learned relationships and details among features and labels that are too specific to the training data only. The model therefore cannot be used to accurately infer anything about new, unseen data. Although training error may be low, test error will be high, as the model can only make decisions based on patterns which exist only in the training set and not in unseen data.

Bias-variance trade-off

Finding the ideal hyperparameters helps control the model complexity, which in turn reduces model estimation errors due to bias and variance.

- **Bias:** Model bias expresses the error that the model makes (how different is the prediction from the training data). Bias error arises when a model is too simple and is underfitting.

An overfit model is too complex; it has captured the idiosyncrasies of the training data and will not generalize to unseen data.

An underfit model is too simple; it has not captured the predictive nuances present in the data and will perform poorly on new data.

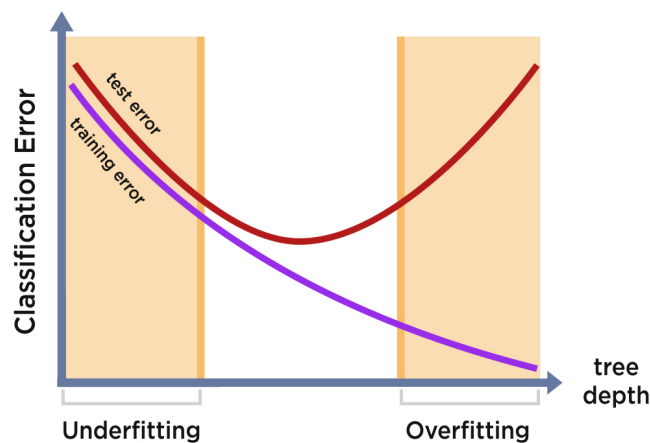
Finding the ideal hyperparameters helps control the model complexity, which, in turn, reduces model estimation errors due to bias and variance.

- **Variance:** Model variance expresses how consistent the predictions of a model are on different data. High variance is a sign that the model is too complex and is overfitting to the particular data set it is trained on.
- The ideal trade-off between bias and variance lies in finding the right hyperparameters, leading to a model with balanced complexity.

Hyperparameter tuning to manage complexity

To ensure your model is not susceptible to overfitting or underfitting, you want to tune the model's hyperparameters to some optimal values. As a refresher, hyperparameters are tune-able properties to control the behavior of a model. Most models have some kind of hyperparameter to control a model's complexity. For a decision tree, it can be the maximum allowable depth of the tree. Note in the graph below how changing the tree depth of a decision tree can affect underfitting and overfitting.

A machine learning engineer's job is to experiment with various hyperparameters values to ensure that the models developed are not overfitted or underfitted.



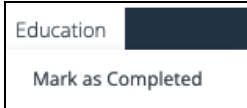
[Back to Table of Contents](#)

Assignment: Optimizing Decision Trees

In this exercise you will implement a decision tree classification model using scikit-learn. You will train different models using different values for hyperparameter max depth and compare the accuracy of each model.

This exercise will be graded.

When you finish your work:

1. Save your notebook by selecting the "Save and Checkpoint" entry from the "File" menu at the top of the notebook. If you do not save your notebook, some of your work may be lost.
2. Submit your work by clicking **Education** → **Mark as Completed** in the upper left of the Activity window.A screenshot of a web interface showing a dropdown menu. The menu is open, displaying the text 'Education' at the top and 'Mark as Completed' as one of the options below it.
3. After submission, the Jupyter Notebook will always remain accessible in the first tabbed window of the exercise. To reattempt the work, you will first need to click **Education** → **Mark as Uncompleted**, then proceed to make edits to the notebook. Once you are ready to resubmit, follow steps one and two.

This exercise will be auto-graded.

This content cannot be rendered properly in the course transcript. Please log into the course to view it.

[Back to Table of Contents](#)

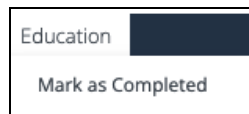
Assignment: Unit 3 Assignment: Building a DT After Feature Transformations

In this assignment you will put into practice everything that you have learned this unit to implement a decision tree classifier using scikit-learn. You will prepare your data by one-hot encoding some features and creating training and test data sets. You will implement and train a few different decision tree classifiers using different hyperparameter values, and you will plot the resulting accuracy scores. You can accomplish this assignment by using all of the techniques you practiced and implemented in the different exercises and activities this unit.

This assignment will be graded.

When you finish your work:

1. Save your notebook by selecting the "Save and Checkpoint" entry from the "File" menu at the top of the notebook. If you do not save your notebook, some of your work may be lost.
2. Submit your work by clicking **Education** → **Mark as Completed** in the upper left



3. of the Activity window.
3. After submission, the Jupyter Notebook will always remain accessible in the first tabbed window of the exercise. To reattempt the work, you will first need to click **Education** → **Mark as Uncompleted**, then proceed to make edits to the notebook. Once you are ready to resubmit, follow steps one and two.

This assignment will be graded by your facilitator.

This content cannot be rendered properly in the course transcript. Please log into the course to view it.

[Back to Table of Contents](#)

Assignment: Unit 3 Assignment: Written Submission

In this part of the assignment, you will answer six questions about building and evaluating your models using algorithms such as decision trees and k-nearest neighbors.

The questions will prepare you for future interviews as they relate to concepts discussed throughout the unit. You've practiced these concepts in the coding activities, exercises and coding portion of the assignment.

Completion of this assignment is a course requirement.

Instructions:

1. Download the [Unit 3 Assignment document](#).
2. Answer the questions.
3. Save your work as one of these file types: .doc or .docx. No other file types will be accepted for submission.
4. Submit your completed Unit 3 Assignment document for review and credit.
5. Click the **Start Assignment** button on this page, attach your completed Unit 3 Assignment document, then click **Submit Assignment** to send it to your facilitator for evaluation and credit.

Before you begin:

Please review [eCornell's policy regarding plagiarism](#) (the presentation of someone else's work as your own without source credit).

[Back to Table of Contents](#)

Module Wrap-up: Implement Decision Trees

Now that you've completed this module, you can see how building a decision tree by repeatedly dividing data into partitions allows you to predict labels in larger data sets more efficiently. You continued to explore hyperparameter tuning and talked about the different hyperparameters for a decision trees. Remember that regulating the complexity of a model by using the correct hyperparameters avoids both overfitting and underfitting, improving the likelihood that your model performs well on new data.

[Back to Table of Contents](#)

Lab 3 Overview

In this lab, you will practice optimizing a decision tree. You will be working in a Jupyter Notebook.

This 3-hour lab session will include:

- **5 minutes** - Icebreaker
- **30 minutes** - Concept Overview + Q&A
- **30 minutes** - Breakout Groups (Big-Picture Questions)
- **15 minutes** - Sharing of Big-Picture Group Responses
- **15 minutes** - Break
- **85 minutes** - Breakout Groups (Lab Assignment)
- **5 minutes** - Survey

By the end of Lab 3, you will:

- Convert categorical features to one-hot encoded values.
- Train decision tree classifiers with various hyperparameter values.
- Train KNN classifiers with various hyperparameter values.
- Visualize the models' accuracies.

[Back to Table of Contents](#)

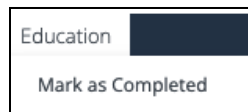
Assignment: Lab 3 Assignment

In this lab, you will continue working with the Airbnb NYC "listings" data set.

This assignment will be graded.

When you finish your work:

1. Save your notebook by selecting the "Save and Checkpoint" entry from the "File" menu at the top of the notebook. If you do not save your notebook, some of your work may be lost.
2. Submit your work by clicking **Education** → **Mark as Completed** in the upper left of the Activity window.



3. After submission, the Jupyter Notebook will always remain accessible in the first tabbed window of the exercise. To reattempt the work, you will first need to click **Education** → **Mark as Uncompleted**, then proceed to make edits to the notebook. Once you are ready to resubmit, follow steps one and two.

[This lab assignment will be graded by your facilitator.](#)

This content cannot be rendered properly in the course transcript. Please log into the course to view it.