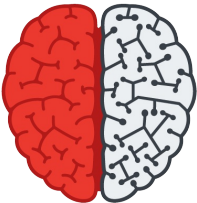


# Problem-Solving With Machine Learning

## What you'll do

- Define and reframe problems using machine learning concepts and terminology
- Identify the applicability, assumptions, and limitations of the k-Nearest Neighbors (k-NN) algorithm
- Simplify and make Python code efficient with matrix operations using NumPy, a library for the Python programming language
- Build a facial recognition system using the k-NN algorithm
- Implement loss functions to compute the accuracy of an algorithm



## Course Description

Data scientists employ machine learning techniques to solve complex problems and make decisions using computer programs that "learn" from past data. In this course, you will learn how to formalize real-world scenarios as machine learning problems and begin investigating how to implement, evaluate, and improve machine learning algorithms.

This course gives you firsthand experience using tools like Jupyter Notebook and the NumPy Python library to write code and implement machine learning algorithms. Ultimately, you will implement the k-Nearest Neighbors algorithm to build a facial recognition system that can identify individuals within a data set of facial images.

**System requirements:** This course contains a virtual programming environment that does not support the use of Safari, IE, Edge, tablets, or mobile devices. Please use Chrome or Firefox on a computer for this

course.

**Note:** *It is strongly recommended that **Linear Algebra: Low Dimension** and **Linear Algebra: High Dimension** be completed prior to beginning this course. This course assumes knowledge of linear algebra concepts including **dot products**, **norms**, and **matrix operations**. Please contact your facilitator if you have any questions about this material.*



**Kilian Weinberger**  
**Associate Professor**  
**Computing and Information Science, Cornell University**

**Kilian Weinberger** is an associate professor in the Department of Computer Science at Cornell University. He received his Ph.D. from the University of Pennsylvania in machine learning under the supervision of Lawrence Saul, and his undergraduate degree in mathematics and computer science from the University of Oxford.

During his career, Professor Weinberger has won several best paper awards at ICML (2004), CVPR (2004, 2017), AISTATS (2005), and KDD (2014, runner-up award). In 2011, he was awarded the Outstanding AAAI Senior Program Chair Award, and in 2012 he received an NSF CAREER award. He was elected co-program chair for ICML 2016 and for AAAI 2018. In 2016 he was the recipient of the Daniel M. Lazar '29 Excellence in Teaching Award.

Professor Weinberger's research focuses on machine learning and its applications. In particular, he focuses on learning under resource constraints, metric learning, machine-learned web-search ranking, computer vision, and deep learning. Before joining Cornell University, Professor Weinberger was an associate professor at Washington University in St. Louis and had previously worked as a research scientist at Yahoo! Research.

## Welcome Transcript

Machine learning is the science of how to make computers learn from experience. Recently, this has often also been referred to as data science. The way this works is that computers are programmed to discover patterns in data sets. The more data they see, the better they get at discovering these patterns. In this specific course, we will talk about supervised learning. Supervised learning is when you have a specific data set and you know exactly what you want to predict. So you want to teach the algorithm to predict a very specific thing. We will introduce you to the first supervised learning algorithm; it's called k-Nearest Neighbors. We will tell you what assumptions it makes and when it's the right algorithm to use. You will also learn how to implement it in Python. By the end of this class, you should be able to use this algorithm on your own data sets, and we will demonstrate this by letting you implement your own face recognition system.

# Table of Contents

---

## Module 1: Framing Problems for Supervised Machine Learning

- Module Introduction: Framing Problems for Supervised Machine Learning
- Watch: What Is Supervised Machine Learning?
- Read: The Machine Learning Setup
- Read: The Features of a Problem
- Read: The Labels of a Problem
- Ask the Expert: Laurens van der Maaten
- Solving Problems With Machine Learning
- Tool: Notations for Machine Learning
- Read: Formalizing the Machine Learning Setup
- Watch: How Do We Evaluate a Machine Learning Algorithm?
- Read: Selecting a Loss Function
- Activity: Which Loss Function Should You Use?
- Watch: Avoiding Memorization and Encouraging Generalization
- Read: Splitting Data Sets
- Split Your Data Set
- Watch: Identify Necessary Assumptions
- Part One — Frame a Machine Learning Problem
- Module Wrap-up: Framing Problems for Supervised Machine Learning

## Module 2: Classification With the k-Nearest Neighbors Algorithm

- Module Introduction: Classification With the k-Nearest Neighbors Algorithm
- Watch: The k-Nearest Neighbors Algorithm
- Read: k-Nearest Neighbors
- Tool: k-NN Cheat Sheet

- Watch: k-NN Hyperparameter
- Read: k-NN Hyperparameters
- Watch: Enhancing k-NN
- Read: Enhancing k-NN
- Activity: Find a Decision Boundary
- Watch: The Curse of Dimensionality
- Read: The Curse of Dimensionality
- Read: High-Dimensional Spaces
- Part Two — Application and Limitations of k-NN
- Module Wrap-up: Classification With the k-Nearest Neighbors Algorithm

## Module 3: NumPy and Jupyter Notebooks

- Module Introduction: NumPy and Jupyter Notebook
- Watch: Using the NumPy Library in Python
- Read: Why Vectorization Is Faster
- Tool: NumPy Cheat Sheet
- Code: Introduction to NumPy
- Code: Practice Matrix Multiplication
- Code: Additional NumPy Exercises
- Read: Using Jupyter Notebook
- Use a Jupyter Notebook
- Module Wrap-up: NumPy and Jupyter Notebook

## Module 4: Building a Facial Recognition System

- Module Introduction: Building a Facial Recognition System
- Activity: Compute Euclidean Distances in Matrix Form
- Code: Euclidean Distance Function Without Loops
- Ask the Expert: Laurens van der Maaten, Part Two
- Build a Facial Recognition System
- Module Wrap-up: Building a Facial Recognition System
- Read: Thank You and Farewell

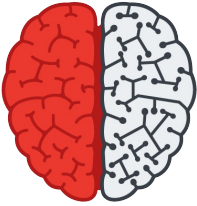
# Module 1: Framing Problems for Supervised Machine Learning

- Module Introduction: Framing Problems for Supervised Machine Learning
- Watch: What Is Supervised Machine Learning?
- Read: The Machine Learning Setup
- Read: The Features of a Problem
- Read: The Labels of a Problem
- Ask the Expert: Laurens van der Maaten
- Solving Problems With Machine Learning
- Tool: Notations for Machine Learning
- Read: Formalizing the Machine Learning Setup
- Watch: How Do We Evaluate a Machine Learning Algorithm?
- Read: Selecting a Loss Function
- Activity: Which Loss Function Should You Use?
- Watch: Avoiding Memorization and Encouraging Generalization
- Read: Splitting Data Sets
- Split Your Data Set
- Watch: Identify Necessary Assumptions
- Part One — Frame a Machine Learning Problem
- Module Wrap-up: Framing Problems for Supervised Machine Learning

---

[Back to Table of Contents](#)

# Module Introduction: Framing Problems for Supervised Machine Learning



Supervised machine learning is all about learning a function from existing data that can make predictions about new data. In this module, you will investigate the intuition behind how data scientists approach and frame machine learning problems. You'll then practice framing problems in terms of machine learning and review the basic mathematics behind the standard machine learning

setup.

Next, you'll examine how machine learning algorithms are evaluated using loss functions and determine the appropriate loss function for a specific set of data. Finally, you will practice decision making about splitting a data set in order to train and validate machine learning algorithms.

---

[Back to Table of Contents](#)

## Watch: What Is Supervised Machine Learning?

Supervised machine learning uses inputs, referred to as *features* ( $x$ ), and their corresponding outputs, referred to as *labels* ( $y$ ), to train a *function* ( $h$ ) that can predict the label of a previously unseen data point. This is useful in countless applications, from predicting the behavior of stock markets to identifying objects in an image. Professor Weinberger briefly provides an overview of machine learning then explores examples in more detail.

### Video Transcript

The general setup of supervised learning is that you would like to learn a function,  $h$ , from stuff that we do know to stuff that we don't know. The stuff that we do know is what we call features, and we denote it as  $x$ . And the stuff that we don't know is a label, and we denote that as  $y$ . So we learn a function  $h$  that goes from  $x$  to  $y$ .

Let's go through an example. Imagine we would like to predict whether a specific stock — for example, Coca-Cola stock — will go up tomorrow. So  $y$  is whether tomorrow the stock will go up or down. We don't know this, but what we do is we can collect some statistics, these features, that we think are relevant towards the Coca-Cola stock going up or down. For example, that could be whether the Pepsi stock went up today, or whether tomorrow is going to be hot weather in America; something like this. So we collect a bunch of statistics, and then what we do is we extract these statistics from past data. So, we knew last week, basically, on a certain day that had certain statistics, the next day the stock went up. And so then we can train our  $h$  to basically work really well in the past. So, in the past, from this feature vector  $x$ , it was able to predict the label  $y$  the next day. And once we are satisfied, we are convinced that this works well, then we can apply it to today's vector and predict tomorrow's stock.

There's many different examples; the stock market is just one of them. You could also have medical examples. Then, for example,  $y$  could be — you know, you're trying to predict if a patient is ill or not, and  $x$  could be symptoms, right, or patient statistics, like the age, the gender, the weight,



the blood pressure, etc., about the person. You could also have, you know, a face recognition system. Then  $y$  would be the identity of the person, and  $x$  could be the image of a person.

In general there's multiple categories for  $y$ , what  $y$  could be. And in this class, we will distinguish between three different types. The first one is a binary; that just means  $y$  is 1 or -1. So, for example, the stock market, it could go up or it could go down. The second one is multiclass; here  $y$  could be multiple categories. So, for example, given an image, and like to predict who is in that image. That's, for example, when you upload an image onto Facebook, Facebook recognizes the faces and then says which one of your 150 Facebook friends would it be. Here  $y$  is a number from 1 to 150. The last category for  $y$  is regression. Here, for example, given a house, I'm trying to predict the price of the house. Here I don't just have multiple categories; it could be any value between zero and maybe a million dollars. So if  $y$  is a real value, then we call it regression.

So we have binary classification, multiclass classification, and regression. Our feature vectors  $x$  could also take multiple forms. For example, they could be patient statistics, as you just saw. They could be words; if we, for example, have an email, and we would like to predict if an email is spam or not spam. Or it could be pixels if it's an image. Ultimately, one of the nice things about machine learning is that it's very universal. So in all of these settings, we can derive algorithms that actually work really, really well. And one job for you as a data scientist will be to identify which is the right algorithm for a specific setup.

---

[Back to Table of Contents](#)

## Read: The Machine Learning Setup

- Data scientists use supervised machine learning to create computer programs that learn from past data
- To learn from data, you must differentiate between what you know, the *features* ( $x$ ), and what you would like to infer, the *label* ( $y$ )

The purpose of machine learning is to make decisions from data. Following the approach of traditional computer science, one might be tempted to write a carefully designed program that follows some rules to make these decisions. Instead of writing a program in this traditional way, however, data scientists use supervised machine learning to create a computer program that is *learned* from past data.



To learn from data, you must differentiate between what you know, the **features** ( $x$ ), and what you would like to infer, the **label** ( $y$ ). For example, your features could describe a patient in a hospital (e.g., gender, age, body temperature, various symptoms) and the label could be if the patient is sick or healthy. You can use data from past medical records to learn a **function** ( $h$ ) **that is able to determine a future patient's diagnosis based on their symptoms.**

For an incoming patient, when you observe features ( $x$ ), **you can apply the function** ( $h$ ) **to predict whether this new patient is sick or healthy** ( $y$ ). **The initial stage where you use existing medical records to learn a function is called the *training stage*, and the latter where you apply the function to a new patient is called the *testing stage*.**

### The Hypothesis of a Problem

The function, often referred to as *hypothesis* and denoted as  $h$ , is the program that is learned from the data. You differentiate between the *hypothesis class*, which is the set of all possible functions that could be learned by the algorithm, and the *final hypothesis*, which is obtained after learning from our training data. Many possible hypothesis classes exist and they loosely correspond to different types of learning algorithms. All machine learning algorithms function differently and will have a variety of parameters for their hypothesis class. It is your job as a data scientist to identify which algorithm is most suitable for a given learning problem.

---

[Back to Table of Contents](#)

## Read: The Features of a Problem

- Features are the relevant characteristics or attributes that you believe may contribute to the outcome of an event
- The examples of feature vectors provided are bag-of-words features, pixel features, and heterogeneous features
- How a data instance is encoded into a vector and what data the vector contains will usually influence the outcome of the machine learning process

Features are the relevant characteristics or attributes that you believe may contribute to the outcome of an event. You use features to describe the data from which you are trying to make predictions. Throughout this course, you should assume that features are stored as a  $d$ -dimensional vector of feature values. How a data instance is encoded into a vector and what data the vector contains will usually influence the outcome of the machine learning process. Typically, here is where domain expertise can be helpful.

### Examples of Feature Vectors



**Heterogeneous features:** Patient data in medical applications typically contain many heterogeneous features. For example, these could include the patient's age in years, blood pressure, and height in centimeters. Here, each feature has its own unit and can vary widely in range and distribution. When working with heterogeneous data, a data scientist has to keep in mind that some feature values can be much larger than

others and must choose algorithms that can appropriately cope with such variety in scale. For instance, blood pressure and height are of different

scales; a blood pressure reading that is 10 units higher than the average and a height measurement of 10 units higher than the average have very different implications. If this difference in scale is not taken into account, it can easily happen that the algorithm ignores some important features, only because the differences are of very small magnitude.



**Bag-of-words features:** Text documents are often stored as bag-of-words features. This is a method to convert a text document with any number of words into a feature vector of fixed dimensionality. Before learning begins, one agrees on a finite set of possible words of interest, such as the  $d = 100,000$  most common words in the English language. The text document is then scanned for these words and represented as

a vector of word counts. In other words, the  $i^{th}$  dimension of the feature vector stores how many times the  $i^{th}$  word appears in this text document. Here, a text document is treated as a set of words whose order is disregarded for representational convenience. As most words in the English language are not present in any given document, the vector will consist mostly of zeros. A feature vector that contains far more zeros than non-zero entries is referred to as “sparse.”



**Pixel features:** Images are typically stored as pixels. These can be represented as a vector by simply “vectorizing” the image in one long chain of numbers. If an image has six megapixels, and each pixel has three numbers (one for red, green, blue) this yields an 18 million dimensional vector. Colors are typically stored by their saturation, ranging from zero to 255, so all feature values are non-negative and within

that range. More specialized algorithms (such as convolutional neural



networks) will preserve the 2D grid structure of an image and represent images as three saturation matrices, one for each color channel.

---

[Back to Table of Contents](#)

## Read: The Labels of a Problem

The label ( $y$ ) is what you want to predict for a given data instance. Labels can come in many different forms, but throughout this course we will only distinguish three different cases:

 A light blue circle containing a dark blue circle with a white "+1" and a red square with a white "-1".	<p><b>Binary</b></p> <p><b>There are only two possible label values.</b> For example, with spam email classification, an email is either spam or not spam. Spam could be mapped to "+1" and email not considered spam to "-1."</p>
 A light blue circle containing a dark blue circle, a red square, and a light blue triangle.	<p><b>Multiclass</b></p> <p><b>There are multiple distinct label values.</b> For example, in a facial recognition application, you would distinguish each individual as a separate class, such as:</p> <ul style="list-style-type: none"><li>• class1="Bill Gates"</li><li>• class2="Steve Jobs"</li><li>• class3="Linus Torvalds"</li></ul>
	<p><b>Regression</b></p> <p><b>There are infinite possible label values.</b> For example, if you want to predict for how</p>



much a particular house will sell, the label (the sale price) could be any non-negative value.

## Regression vs. Multiclass Classification

Some problems could be cast as either regression or multiclass; however, there is usually a most natural choice.

For example, assume you are trying to predict the height (label) of a person based on other data (features), such as their gender, weight, age, etc. If you cast this as a multiclass classification problem, you might define each class as a rounded incremental value of 1cm. If we try to classify someone who is actually 183cm tall and return a label of 182cm, we are just as wrong as if we had predicted 140cm. Most people would consider a predicted height of 182cm to be much closer to the truth (183cm) than 140cm is, but since each height is its own class and there is typically no assumption that one class is more similar to another in classification problems, we've simply failed to predict the height correctly. For this example, we should instead choose regression. In regression, we typically assume that, because the label can be any real value, you will almost never hit it exactly but hope to get very close.

An example problem where multiclass classification would be the best choice is facial recognition. Bill Gates, Steve Jobs, and Linus Torvalds would each be their own class. We see here that each class is separate and distinct; selecting the wrong class is simply an incorrect label. There is no notion of similarity between classes since you are either right or wrong. For image classification problems, multiclass classification is the appropriate choice.

---

[Back to Table of Contents](#)



## Ask the Expert: Laurens van der Maaten

---



**Laurens van der Maaten** is a research scientist at Facebook AI Research in New York, working on machine learning and computer vision. He previously worked as an assistant professor at Delft University of Technology and as a post-doctoral researcher at UC San Diego. Dr. van der Maaten received his Ph.D. from Tilburg University in 2009.

Dr. van der Maaten is interested in a variety of topics in machine learning and computer vision. Currently, he works on embedding models, large-scale weakly supervised learning, visual reasoning, and cost-sensitive learning.

---

### Question

How is machine learning helping us solve problems?

### Video Transcript

So I think what's really cool about machine learning right now is that a bunch of things have come together that have allowed us to solve some really important problems like image classification, like speech recognition, and machine translation. And basically the things that have come together are sort of a set of algorithms, but then also data, right? So the availability of sort of large data sets that we can use for training of these models for these kinds of problems, and computation. So what's been really important in recent years is the development of GPUs for the use of training machine learning models. And so I think that's what's really exciting; we've really sort of seen this confluence of these three different vectors that have allowed for breakthroughs in machine learning on a particular set of problems.

---

[Back to Table of Contents](#)

# Solving Problems With Machine Learning

## Discussion topic:

- How are you planning to apply machine learning to your current work?
- What are some questions you think could be answered or problems that could be solved by applying machine learning to data you already have?
- What future scenarios do you envision where machine learning could help you make a decision or predict an outcome?

## Instructions:

- You are required to participate meaningfully in all course discussions.
- The instructor will review the discussion and you will receive a "Complete" or an "Incomplete" grade based on the quality of your contributions.
- Limit your comments to 200 words.
- You are strongly encouraged to post a response to at least two of your peers' posts.

## To participate in this discussion:

Click **Reply** to post a comment or reply to another comment. Please consider that this is a professional forum; courtesy and professional language and tone are expected. Before posting, 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](#)

## Tool: Notations for Machine Learning

Use the [Notations for Machine Learning tool](#) as a reference for the math notations used in this and subsequent courses.

This tool provides a list of specific math notations that will be used throughout the Machine Learning courses. You'll find information on different variables and the notation styles used, which will help clarify math equations presented in this and subsequent courses.

---

[Back to Table of Contents](#)

## Read: Formalizing the Machine Learning Setup

To formalize the supervised machine learning setup, let's first look at how we express features and labels for our training data. Our training data comes in pairs of inputs and labels  $(\mathbf{x}, y)$ , where  $\mathbf{x} \in \mathcal{R}^d$  is the input instance and  $y \in \mathcal{C}$  is its label. The data is denoted as  $D = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\} \subseteq \mathcal{R}^d \times \mathcal{C}$  where:

$\mathbf{x}_i$  : input vector of the  $i^{\text{th}}$  sample (e.g., we refer to the  $a^{\text{th}}$  feature dimension of  $\mathbf{x}_i$  as  $[\mathbf{x}_i]_a$ )

$y_i$  : label of the  $i^{\text{th}}$  sample

$\mathcal{R}^d$  :  $d$  -dimensional feature space

$\mathcal{C}$  : label space

### Examples of label space:

<b>Binary classification</b>	$\mathcal{C} = \{0, 1\}$ or $\mathcal{C} = \{-1, +1\}$	Example: spam filtering. An email is either spam (+1) or not spam (-1).
<b>Multiclass classification</b>	$\mathcal{C} = \{0, 1, 2, \dots, K - 1\}$ or $\mathcal{C} = \{1, 2, \dots, K\}$ ( $K \geq 2$ )	Example: facial recognition. A person can be exactly one of $K$ identities (e.g., 1="Bill Gates", 2="Steve Jobs", etc.).
<b>Regression</b>	$\mathcal{C} = \mathbb{R}^+$ $\mathbb{R}^+$ : set of positive reals	Example: future sale price of a house. A house could sell for any

positive value.

Given a training data set  $D$  of label and feature pairs, the ultimate goal of supervised machine learning is to find a function  $h : \mathcal{R}^d \rightarrow \mathcal{C}$ , such that it can predict the label of data points that are *not* in the training set (we call this *testing* data):

$$h(\mathbf{x}_i) \approx y_i \text{ for all } (\mathbf{x}_i, y_i) \notin D$$

Since we do not have data outside of  $D$ , we learn  $h$  in order to reliably predict the label on the training data, which in turn will hopefully generalize to data outside the training set.

$$h(\mathbf{x}_i) \approx y_i \text{ for all } (\mathbf{x}_i, y_i) \in D$$

## Examples of feature vectors

We call  $\mathbf{x}_i$  a feature vector and  $d$  the dimensions of that feature vector that describe the  $i^{\text{th}}$  sample.

### Patient data in a hospital

The features describe the attributes of a patient and the label represents whether the patient is sick (+1) or healthy (-1). More specifically, the feature vector of the  $i^{\text{th}}$  patient is  $\mathbf{x}_i = ([\mathbf{x}_i]_1, [\mathbf{x}_i]_2, \dots, [\mathbf{x}_i]_d)$ . Here, the first dimension may represent the biological gender, encoded as 0 or 1. In our example below, Sarah Stevens is female, therefore  $[\mathbf{x}_i]_1 = 1$ . The second dimension we've chosen represents height in cm, so Sarah's second feature is 165. The third dimension,  $[\mathbf{x}_i]_3$ , stores the patient's age in years, which is 23 for Sarah. This continues for all of the features in our data set. In this case,  $d \leq 100$  and the feature vector is *dense* (i.e., the number of non-zero coordinates in  $\mathbf{x}_i$  is large relative to  $d$ ).

### Example:

Sarah Stevens, female, 165 cm, 23 years old, healthy:  $\mathbf{x}_1 = (1, 165, 23)$ ,  $y_1 = -1$

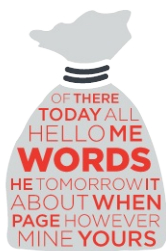
James Joyce, male, 167 cm, 65 years old, sick:  $\mathbf{x}_2 = (0, 167, 65)$ ,  $y_2 = +1$

Barry Michaels, male, 183 cm, 50 years old, healthy:  $\mathbf{x}_3 = (0, 183, 50)$ ,  $y_3 = -1$

Lisa Barrigan, female, 177 cm, 32 years old, sick:  $\mathbf{x}_4 = (1, 177, 32)$ ,  $y_4 = +1$

Carly Atworth, female, 142 cm, 18 years old, sick:  $\mathbf{x}_5 = (1, 142, 18)$ ,  $y_5 = +1$

Jasper Baker, male, 150 cm, 88 years old, healthy:  $\mathbf{x}_6 = (0, 150, 88)$ ,  $y_6 = -1$



### Text document in bag-of-words format

The feature vector represents a text document — an article, for example — and the label corresponds to a category (e.g., sports, politics, entertainment). The  $i^{\text{th}}$  document is represented as the vector

$\mathbf{x}_i = ([\mathbf{x}_i]_1, [\mathbf{x}_i]_2, \dots, [\mathbf{x}_i]_d)$ . Each dimension represents the

exact number of occurrences of one particular word (e.g.,  $[\mathbf{x}_i]_j$  is the number of occurrences of the  $j^{\text{th}}$  word in document  $i$ ). In this case,  $d$  could be between 100000 and 10M and the feature vector is sparse (i.e.,  $\mathbf{x}_i$  consists mostly of zeros).

### Example

Consider this corpus, a portion of Charles Dickens' "A Tale of Two Cities." For this example, let's imagine that each line is a distinct document:

It was the best of times,  
it was the worst of times,  
it was the age of wisdom,  
it was the age of foolishness,

The vocabulary is every unique word in the corpus:

it

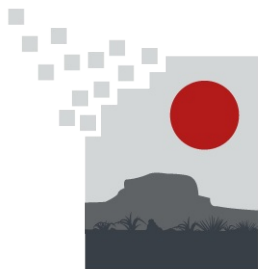
was  
 the  
 best  
 of  
 times  
 worst  
 age  
 wisdom  
 foolishness

The feature vectors for these four documents each contain ten dimensions, corresponding to the sequence of ten words in the vocabulary. If a word from the vocabulary is present in the document, we assign a "1" for the dimension that corresponds to that word in the vector. For example, the second document contains the word "worst," which is seventh in our vocabulary, therefore the feature vector that represents this document,  $\mathbf{x}_2$ , contains a "1" in the seventh dimension. There are two labels possible for each document, past tense (+1) and present tense (-1):

$$\begin{aligned} \mathbf{x}_1 &= (1, 1, 1, 1, 1, 1, 0, 0, 0, 0), y_1 = +1 \\ \mathbf{x}_2 &= (1, 1, 1, 0, 1, 1, 1, 0, 0, 0), y_2 = +1 \\ \mathbf{x}_3 &= (1, 1, 1, 0, 1, 0, 0, 1, 1, 0), y_3 = +1 \\ \mathbf{x}_4 &= (1, 1, 1, 0, 1, 0, 0, 1, 0, 1), y_4 = +1 \end{aligned}$$

You could also represent all four lines of the corpus as a single vector comprised of word counts for each instance of a word from the dictionary.

$$\mathbf{x} = (4, 4, 4, 1, 4, 2, 1, 2, 1, 1)$$



### Images in pixel space

The features correspond to images in raw pixel format and the label can correspond to an image category. For example, the image may be of a person and the label may be the person's name (e.g., "Bill Gates"). Here,



every input dimension corresponds to a color of a particular pixel:  
 $\mathbf{x}_i = ([\mathbf{x}_i]_1, [\mathbf{x}_i]_2, \dots, [\mathbf{x}_i]_{3k})$ , where  $[\mathbf{x}_i]_{3j-2}$ ,  $[\mathbf{x}_i]_{3j-1}$ , and  $[\mathbf{x}_i]_{3j}$  refer to the red, green, and blue values of the  $j$ th pixel in the image.

In this case,  $d$  could be between 100000 and 10M and the feature vector is dense. A 7MP camera results in  $7M \times 3 = 21M$  features.

## Example

Consider three images; one solid red, one solid green, and one solid blue. Each image is comprised of three pixels and is represented by nine features (the red, green, and blue values of each of the three pixels).

Each vector holds these values:

([pixel 1 red value], [pixel 1 green value], [pixel 1 blue value], [pixel 2 red value], [pixel 2 green value], [pixel 2 blue value], [pixel 3 red value], [pixel 3 green value], [pixel 3 blue value]).

There are three labels, red (+1), green (0), and blue (-1):

$$\mathbf{x}_1 = (1, 0, 0, 1, 0, 0, 1, 0, 0), y_1 = +1$$

$$\mathbf{x}_2 = (0, 1, 0, 0, 1, 0, 0, 1, 0), y_2 = 0$$

$$\mathbf{x}_3 = (0, 0, 1, 0, 0, 1, 0, 0, 1), y_3 = -1$$

---

[Back to Table of Contents](#)

# Watch: How Do We Evaluate a Machine Learning Algorithm?

Machine learning algorithms are only as good as the mistakes they make. When given a specific objective, your job as a data scientist is to minimize the loss function, which is the measure of how many mistakes your function made. In this video, Professor Weinberger introduces a few loss functions you'll use to improve your algorithms.

## Video Transcript

On the big picture, the way supervised learning works is that the computer learns from its own mistakes. So, what we do is we collect a training data set, and then the computer tweaks its function  $h$ , which makes predictions, such that it works as well as possible on this data set. Now, for a computer to do this, we have to be a little bit more specific, and that's where a loss function comes in.

A loss function quantifies what we mean by function  $h$  work as well as possible. Specifically, what a loss function does, it gives the computer an objective, a very clear objective. The computer will try to minimize the loss function, and the loss function measures how many mistakes that function  $h$  makes.

There's many different kind of loss functions; however, they all have two things in common. The first one is that lower loss is always better, and the second thing is that a loss of zero is perfect. And that's what we are aiming for. Let me introduce you two different examples. The first one is the zero-one loss. And the zero-one loss is commonly used for classification. It's very intuitive. Essentially what it does, you go over your data set, and you count how many errors do you make; how many times is your prediction not exactly the same as the label? So, for example, if you have an email spam classifier, you count how many times did you classify spam email as not-spam email, or not-spam email as spam. And it outputs the error rates, so 10% would mean 10% of my data points are misclassified.

Another example is the absolute loss. The absolute loss is useful for regression problems. So regression problems are where we try to predict a real valued output. So, for example, one task could be that given a house, we're trying to predict for how much money that house will be sold. So our vector  $x$  could be some specific attributes of that house, and  $y$  is the amount of money it was sold by. So the absolute loss would then take the predicted price for the house minus the actual price of the house and take the absolute value. So, for example, let's say a house was sold for \$120,000, and we predicted it would be sold for \$130,000. The difference is \$10,000, and that's how much loss we suffer for this particular house. The overall loss function just measures the average loss that we have per data point.

---

[Back to Table of Contents](#)

## Read: Selecting a Loss Function

- There are typically two steps in learning a hypothesis function: selecting the algorithm and finding the best function within the class of possible functions
- A loss function evaluates a hypothesis on our data and tells us how good or bad it is, helping us choose the best function
- Three examples of loss function are zero-one, squared, and absolute losses

There are typically two steps involved in learning a hypothesis function:

1. Selecting the appropriate algorithm for the problem
2. Finding the best function from all possible functions

### Select the Appropriate Algorithm

First, we select the type of machine learning algorithm that we think is appropriate for this particular learning problem. This defines the hypothesis class  $\mathcal{H}$  (i.e., the type of function we would like to learn).

### Find the Best Function

Second, we find the best function within the hypothesis class. This second step is the actual learning process and often — but not always — involves an optimization problem. Essentially, we try to find a function  $h \in \mathcal{H}$  within the hypothesis class that makes the fewest mistakes on our training data.

How can we find the best function? For this, we need some way to evaluate what it means for one function to be *better* than another. This is where the **loss function** comes in. A loss function evaluates a hypothesis on our training data and tells us how bad it is. The higher the loss, the worse it is — a loss of zero means it makes perfect predictions. It is common practice to divide the loss by the total number of training

samples,  $n$ , so that the output can be interpreted as the average loss per sample (and is independent of  $n$  ).

## Examples of Loss Functions

---

### Zero-One Loss

The simplest loss function is the zero-one loss. It literally counts how many mistakes a hypothesis function makes on a particular data set. For every single example that is predicted incorrectly, it suffers a loss of 1. The normalized zero-one loss returns the fraction of misclassified training samples, also referred to as the training error. The zero-one loss is often used to evaluate classifiers in multiclass/binary classification settings but rarely useful to guide optimization procedures because the function is non-differentiable and non-continuous.

Formally, the zero-one loss can be stated as:

$$\mathcal{L}_{0/1}(h) = \frac{1}{n} \sum_{i=1}^n \delta_{h(\mathbf{x}_i) \neq y_i}, \quad \text{where} \quad \delta_{h(\mathbf{x}_i) \neq y_i} = \begin{cases} 1, & \text{if } h(\mathbf{x}_i) \neq y_i \\ 0, & \text{otherwise.} \end{cases}$$

---

### Squared Loss

The squared loss function is typically used in regression settings. It iterates over all training samples and suffers the loss  $(h(\mathbf{x}_i) - y_i)^2$ . The squaring has two effects:

1. The loss suffered is always non-negative

2. The loss suffered grows quadratically with the absolute mispredicted amount

The latter property encourages no predictions to be really far off (or the penalty would be so large that a different hypothesis function is likely better suited). Squared loss can be problematic when your data has noisy labels, which could incur a large loss and distract the classifier.

Formally, the squared loss can be stated as:

$$\mathcal{L}_{\text{sq}}(h) = \frac{1}{n} \sum_{i=1}^n (h(\mathbf{x}_i) - y_i)^2.$$


---

## Absolute Loss

Similar to the squared loss, the absolute loss function is also typically used in regression settings. Because the suffered loss grows linearly with mispredictions, it is more suitable for noisy data (when some mispredictions are unavoidable and shouldn't dominate the loss).

Formally, the absolute loss can be stated as:

$$\mathcal{L}_{\text{abs}}(h) = \frac{1}{n} \sum_{i=1}^n |h(\mathbf{x}_i) - y_i|.$$


---

[Back to Table of Contents](#)

## Activity: Which Loss Function Should You Use?

Choosing the right loss function is a critical step in solving a problem with machine learning. Considerations include, first and foremost, determining whether the problem is a classification or regression problem, but also if the labels in the training data are accurate or if the data is noisy. In regression problems, it is additionally worth considering if it is better to make many tiny but few large mistakes (squared loss) or if it is OK to tolerate some larger mistakes yet drive most errors down to zero when possible (absolute loss).

Below you will find three scenarios. Determine whether zero-one, absolute, or squared loss is best suited to evaluate the performance of your function on the training data set.

[Click here to complete activity](#)

---

[Back to Table of Contents](#)

# Watch: Avoiding Memorization and Encouraging Generalization

In this video, Professor Weinberger explains how we split our data into training and test sets to ensure our algorithm can accurately predict labels for new data points. Since we should only use the test data set once, we can also use a validation data set to improve our algorithm before testing.

## Video Transcript

So in supervised learning, what we do is we collect the data set, and then we tweak an algorithm to make predictions on this data set, and we tweak it until it makes as few mistakes as possible. That's great, and we know how to do this now with the loss function; we minimize the loss function. However, ultimately, that particular data set that you're training on, we don't care about. All right? What we really want to do is create an algorithm that then does well on future data.

So, for example, let's say we make an email spam classification algorithm. We don't really care about the data that you're training it on; that's the data that's already in your email program labeled as spam and not-spam. What you really care about is tomorrow's email. So what we want to make sure is that our algorithm generalizes to new data points. And that gives rise to a problem, and that is that machine learning algorithms are very, very powerful, and what we are doing is we incentivize them to do as well as possible on one particular data set. So what they can do is they can just memorize that data set. Essentially, what they're learning is a database, but they just look up the samples and say, "Oh, for that sample, the label is the following." We want to avoid this from happening, and so there's a very simple trick to avoid. And that is we take the data set that we have and we split it into two parts: training and test. Usually, it's an 80-20 split. And the way we do this is we just take our algorithm and we only let it train on the 80% training data. and we never show the test data. And once we are convinced it's a good algorithm, then we evaluate it; not on the training data, but on the test data.



So essentially what we're doing is we are simulating the setup where we actually — actually sees — data has never seen before by simulating the deployment setting. There are, however, two caveats. The first one is when we take our data and we split it in train and test, we have to be careful that the test data is really the kind of data that you may see during deployment. So we can't just have all data of one kind going training, and all data of another kind during testing. And to make sure that doesn't happen, one way of doing this is we take the data and we just shuffle it uniformly at random, and then we just split it 80-20 into train and test. That's a good way of doing it, the shuffling.

However, there are some settings that it's not appropriate, and that is, for example, when we have a temporal component. So if your data was collected over time, and it changes over time. For example, actually, email spam. Email spam changes over time because spammers always kind of try new attacks. And so what you want to make sure is that you really simulate the case where you train on the past and predict the future. So in this case, the training data has to be past data; for example, data collected on Monday through Thursday. And then test data has to be future data; for example, Friday through Sunday. Another setting may be where we have medical data. Here, you want to make sure that you split on patients so that the training data consists of all measurements made by one set of patients. But then the test data is actually totally different patients because you want to make sure that your algorithm then works on new patients that come into the hospital.

There's a second caveat. And the second caveat is when we do the splitting to training and test, we are really only allowed to look at the test data once. What you're not allowed to do is look at the test data multiple times and tweak your algorithm to do well on that test data because now you're cheating in some sense. The problem is it will be very, very tempting to do so. And the reason is — let's say you've trained your algorithm on the training data set and you evaluate it on test. It could very well happen that your error is still too high.

Let's say you have a spam classifier; you evaluate it on test and it gets 5% error. But you really wanted 1% error. What do you do now? Well, you want to go back to the machine learning algorithm, tweak it a little, make some changes, train it again, and test again. All right? And you

may do this a hundred times. Well, after picking at the test data a hundred times, you've inadvertently changed your algorithm to do well on these particular test data points, and that's not allowed. Now, you may fool yourself into thinking that your algorithm does better than it actually does because you tweaked it to do well on these test data points. To avoid that, we do one more split. We essentially do the same thing again; we take our training data set and we split it into train and validation.

Now, we train our algorithm on train, and we evaluate it on validation data set. And we keep doing this, we keep tweaking our algorithm, until the validation error is low enough, so that you're satisfied. Then, you train your algorithm one more time on the union of train and validation, and you test it one time on test. And if you only look at the test data once, then it's actually an unbiased estimate, so it's a true estimate of the deployment error.

---

[Back to Table of Contents](#)

To avoid overfitting to the training set, you will usually split the data  $D$  into three mutually exclusive subsets:

$D_{TR}$  - training data (80%)

$D_{VA}$  - validation data (10%)

$D_{TE}$  - test data (10%)



A common choice is to split the data 80/10/10, respectively. You then choose a function based on the training data, improve it using the validation data, and evaluate it on the test data.

## How Do You Split a Data Set?

One has to be careful when splitting the training data into the three sets. All three sets should be drawn from the same distribution. A good rule of thumb is to make sure that the data sets simulate the real-life settings in which the algorithm will operate. The most common way of splitting the data is uniformly at random. However, if the data has a temporal component and changes over time, you should split the data by time to make sure that you never predict the past from the future and always the future from the past.

An example of such a data set with a temporal component is email spam. Spam emails change over time, as spammers adapt their emails to get past spam filters. What makes spam filtering so hard is that you don't know what clever tricks the spammers will deploy tomorrow. To address this problem, you could collect data for a few weeks to train your function. Then, collect data for a few days to use as your  $D_{VA}$  data set to validate your function. Finally, collect data for a few more days to generate your  $D_{TE}$  data set to test your function. This process is essentially simulating the real-world setting that a spam filter trained today should catch the spam emails of tomorrow.

## Why Is Validation Data Important?

A common scenario is that the first machine learning algorithm that is trained on  $D_{\text{TR}}$  does not perform well enough on  $D_{\text{TE}}$  and needs further refinement. For example, imagine you are building an email spam filter with the objective to catch at least 99% of all spam, with at most 1/1000 false positives. If the initial classifier is not accurate enough for your needs, you may have to use a different algorithm or change its hyperparameters. The problem is that you cannot tweak your algorithm to perform well on  $D_{\text{TE}}$ , otherwise you would be overfitting your function to this specific data set and not necessarily improving its accuracy on new data. The error obtained on  $D_{\text{TE}}$  is only an unbiased estimate of the true generalization error of the model if the model was trained independently of this test set. The moment you look at the test data once and make changes to the algorithm, it is no longer independent. This is where the validation data set comes in.

The validation data set is a proxy for the test set. In practice, you train your algorithm on the training set and evaluate it on the validation set. If your function is not satisfactorily accurate, you continue tweaking it until the validation error improves to an acceptable level. Then, complete a single and final evaluation of your function on the test set  $D_{\text{TE}}$  to find the unbiased estimate of the generalization error of your final model. Often, the final model is re-trained on the union of  $D_{\text{TR}}$  and  $D_{\text{VA}}$  so as to not waste the data points in the validation set.

## Formalized Process of Training and Evaluation

### Training the Function

Choose a function ( $h$ ) to minimize the training loss:

$$h^*(\cdot) = \operatorname{argmin}_{h \in \mathcal{H}} \epsilon_{\text{TR}}(h),$$

$$\epsilon_{\text{TR}}(h) = \frac{1}{|D_{\text{TR}}|} \sum_{(\mathbf{x}, y) \in D_{\text{TR}}} \ell(\mathbf{x}, y | h(\cdot)).$$

Here,  $\mathcal{H}$  is the hypothetical class (i.e., the set of all possible classifiers  $h(\cdot)$ ). In other words, you are trying to find a hypothesis  $h$  which would have resulted in the lowest possible value ( $\operatorname{argmin}$ ) of the loss function ( $\ell$ )

averaged over all the points in the training set which we will call  $\epsilon_{\text{TR}}$ .

## Evaluating the Function

Evaluate the function through the testing loss:

$$\epsilon_{\text{TE}}(h) = \frac{1}{|D_{\text{TE}}|} \sum_{(\mathbf{x}, y) \in D_{\text{TE}}} \ell(\mathbf{x}, y | h^*(\cdot)).$$

The function  $\epsilon_{\text{TE}}$  is the average of the loss function over all points in the test set.

## Generalization

The testing loss is an unbiased estimator (i.e., an approximation) of the **generalization loss** — the loss over unseen data. Minimizing the generalization loss is our true objective but cannot be done directly:

$$\epsilon(h) = \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{P}} [\ell(\mathbf{x}, y | h(\cdot))].$$

## Split Your Data Set

You will be given three machine learning application scenarios.

Determine how you would split the data effectively into training and test sets for the purpose of training and evaluating a function. **You must complete this quiz to unlock the final module. You may take the quiz as many times as you like to help you explore the concept of splitting data sets.**

---

[Back to Table of Contents](#)

## Watch: Identify Necessary Assumptions

Professor Weinberger introduces the "no free lunch" theorem and demonstrates the importance of regularity in data. Machine learning simply does not work if we view the world as arbitrary and unpredictable.

### Video Transcript

So let's play a little game. I'll give you some data points; here I have  $x$  and  $y$  coordinates. And what I want you to do is look at one particular location,  $x$ , and predict what the value of  $y$  should be. Look at it for a second, look at the other data points; what do you think  $y$  would be at this particular  $x$ ? Maybe 2.2, 2.3? I'm sorry; you're way off. It's actually 4.5. Why is it 4.5? Well, because I can choose it to be anything I want, and I choose just so you lose. Why am I doing this? I'm doing this because I want to show you that you cannot make predictions if the world is not very nicely behaved. If the world is arbitrary, there's no way that machine learning can work.

You have to assume somewhere that the world has some regularity. And in fact, every single machine learning algorithm that works does that, and we call this the "no free lunch" theorem. So your job as a data scientist is actually to know, for every single algorithm, what assumptions are made, and then to verify these assumptions hold on the data that you want to apply it to. And that's how you choose the right machine learning algorithm for the right problem. So in this particular data set, actually — you know, you probably made the implicit assumption that the data set is smooth and similar points in the  $x$ -axis have similar labels on the  $y$ -axis. And that's a reasonable assumption to make, and if you assume that, you can actually make predictions.

---

[Back to Table of Contents](#)

## Part One — Frame a Machine Learning Problem

For the first part of the course project, you will frame a problem that you are currently facing or would like to solve as a machine learning problem. You'll identify the features and labels, and determine the problem type and assumptions. *Completion of this project is a course requirement.*

### Instructions:

- Download the [course project document](#).
- Complete Part One, answering the questions provided within the table. Limit your answers to 100 words.
- Save your work as one of these file types: .doc, .docx, .txt, .pdf, .xls, .xlsx. No other file types will be accepted for submission.
- You will not submit your course project now; you will submit both Part One and Part Two at the end of the next module.
- Both written project parts will be evaluated and scored as a single project worth up to 20 points.

### 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: Framing Problems for Supervised Machine Learning

In this module, you explored the fundamental components of supervised machine learning and discussed problems that machine learning can help you solve. You chose loss functions appropriate for evaluating machine learning algorithms and determined the best options for splitting a data set based on the context. Finally, you identified a problem you'd like to solve and defined it through the lens of machine learning.

---

[Back to Table of Contents](#)



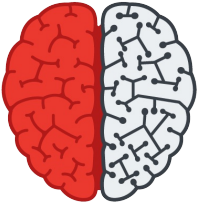
# Module 2: Classification With the k-Nearest Neighbors Algorithm

- [Module Introduction: Classification With the k-Nearest Neighbors Algorithm](#)
- [Watch: The k-Nearest Neighbors Algorithm](#)
- [Read: k-Nearest Neighbors](#)
- [Tool: k-NN Cheat Sheet](#)
- [Watch: k-NN Hyperparameter](#)
- [Read: k-NN Hyperparameters](#)
- [Watch: Enhancing k-NN](#)
- [Read: Enhancing k-NN](#)
- [Activity: Find a Decision Boundary](#)
- [Watch: The Curse of Dimensionality](#)
- [Read: The Curse of Dimensionality](#)
- [Read: High-Dimensional Spaces](#)
- [Part Two — Application and Limitations of k-NN](#)
- [Module Wrap-up: Classification With the k-Nearest Neighbors Algorithm](#)

---

[Back to Table of Contents](#)

## Module Introduction: Classification With the k-Nearest Neighbors Algorithm



In this module, you'll be introduced to your first machine learning algorithm, k-Nearest Neighbors. You will see how this algorithm is used to classify data points and explore its applications and limitations.

You will have a chance to investigate the curse of dimensionality, an aspect of your data that can make machine learning even more of a challenge. You'll discuss visualizing objects in high-dimensional space, and you'll take a quiz on how the curse of dimensionality impacts the viability of k-Nearest Neighbors.

---

[Back to Table of Contents](#)

## Watch: The k-Nearest Neighbors Algorithm

In this video, Professor Weinberger introduces the k-Nearest Neighbors algorithm. He explains the underlying concept behind how k-NN uses the known labels of  $n$   $d$ -dimensional training points that are assigned a class (+1 or -1) to infer the label of a test point.

### Video Transcript

So the k-Nearest Neighbor algorithm is one of the oldest and most well-known supervised learning algorithms. It was introduced by Cover and Hart in 1968. And it's quite, quite simple. So we take our training data; in this case we assume we have data just from two different classes, +1 and -1. Positive points are blue, the negative points are red. And we want to infer the label of a test point from these training points. So here's what we do. We just store the training data, we insert a test point, and now we want to infer the label of this green test point. The k-Nearest Neighbor algorithm makes an assumption, as all machine learning algorithms, and in this case the assumption is that similar points have similar labels.

So what we will try to do is find the  $k$  most similar data points and infer the label from them. So in this case, let's say  $k=3$ , then we find the three most similar data points. Amongst those, two are blue and one is red, so we have a majority vote. And you find out blue is more common, so we assign the blue label. And that's what we do. That's all there is to it. So you find the  $k$  most similar data points, you have a majority vote, and then assign that most common label to the test point.

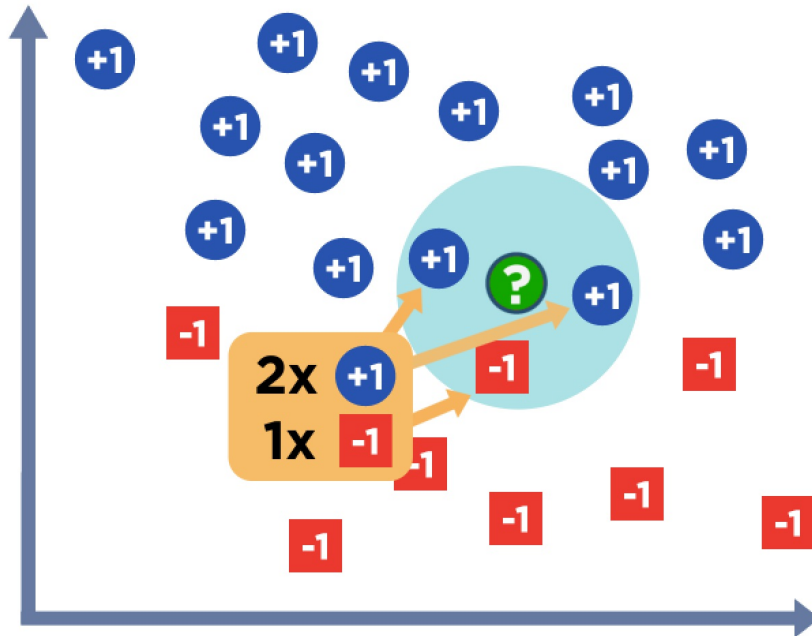
Despite that it's very, very simple and easy to implement, it's actually quite powerful, especially in multiclass setting, where you have many different classes. And because it's so easy to implement, it's often the first go-to algorithm that you build just to get an estimate of how well machine learning works on this particular data set. It works particularly well in low dimensions and if you have a metric that truthfully estimates similarity. That means the data points that are similar, according to our metric, are actually similar in some semantic way.

[Back to Table of Contents](#)

## Read: k-Nearest Neighbors

- Assumption: The k-Nearest Neighbors algorithm assumes similar inputs have similar outputs
- How one defines similarity depends on the application; one way to define similarity is to use the Euclidean distance between two points, where the shorter the distance, the more similar the points
- Classification rule: For a test point  $x$ , assign the most common label amongst its  $k$  most similar training points, also known as majority vote

The k-Nearest Neighbors algorithm is used to determine the label of a given data point based on the label of the other data points closest to it (i.e., its nearest neighbors). For example, in the figure below we have a training data set that contains data points that have been labeled using two different classes, where positive-labeled points are represented as blue circles, and negative-labeled points are represented as red squares. We want to use k-NN to infer the label of a test point using these training points as a reference.



Let's insert a green test data point and attempt to label it as positive or negative. The k-NN algorithm makes an assumption that similar data points have similar labels. So what we will try to do is find the k most similar data points and infer the label of our test data point from the existing data.

Let's use 3-NN and find the three most similar data points. In this case, we find two blue (+1) and one red (-1) data points. Now we have a majority vote, and since blue is more common, we assign the blue label to our test data point. That's all there is to it; you find the k most similar data points, have a majority vote, and then assign that most common label to the test point.

---

[Back to Table of Contents](#)

## Tool: k-NN Cheat Sheet

Use this [k-NN Cheat Sheet](#) when referring to algorithm parameters

This tool 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 k-NN.

---

[Back to Table of Contents](#)

## Watch: k-NN Hyperparameter

The k-NN algorithm has one parameter, "k," the size of the neighborhood. Professor Weinberger explains how changing this parameter affects the algorithm's classification accuracy and helps to manage noisy data.

### Video Transcript

The k-Nearest Neighbor classifier has exactly one parameter, and that is k: the size of the neighborhood that we pick to infer the label. Typical choices are odd numbers, but it can range from  $k=1$  up to  $k=15$  or something like this. So what happens when we change k? Here's a little illustration of a data set that has two different classes, like crosses and circles, and what we did here is we colored every single point in the space, either red or blue. Red means it will be classified as a cross. Blue means this particular point, if it was a test point, would be classified as a circle.

If you look at this data set, you actually observe something; that in the cluster of crosses, there's actually a little circle. And in the cluster of circles, there's a cross. This is probably a labeling mistake. This probably means that this data point has the wrong label. It should probably be a cross but is labeled as a circle. One thing you see is the k-Nearest Neighbor decision boundary, here as  $k=1$ , it's actually quite jiggly. So it picks up on any kind of little movement in the data set. And also, when you see those little islands — these kind of misclassified, mislabeled data points, it carves out, you know, little islands of a positive region in a negative sea, and vice-versa.

Just because around these mislabeled data points, like there's a small area where this data point is the closest, and test points would be classified as positive or negative, respectively. If this is not what you want — if you actually suspected your data set is noisy — then maybe the  $k=1$  is too sensitive to the specifics of your data set. If you increase k to 3, for example, what happens is now you're averaging local neighborhoods, and what happens for this one little mislabeled data point, it will always



get washed out and will always get outvoted by the other data points that are around it. And you can see that now these little islands disappear and the decision boundary becomes smoother. And this trend, you know, gets even stronger as you increase  $k$  further.

So, for example, if you said  $k=9$ , the decision boundary gets even smoother. So what  $k$  does,  $k$  is a measure of complexity. If  $k$  is low, we have a very complex decision boundary that picks up on any little intricate aspect of the data, but it's also very susceptible to noise because maybe these aspects are not real. If  $k$  becomes larger, the decision boundary becomes simpler, and you better protect against noise. However, you know, you may also lose some information. Another aspect of the  $k$ -Nearest Neighbor algorithm that you have to choose is the distance function.

So a natural choice is the Euclidean distance; that's the distance how we measure a 3D space distance between two objects. But there's many other distance functions. For example, there could be the Manhattan distance or the Mahalanobis distance, Minkowski distance, etc. I'm not going through it here in too much detail, but one thing to keep in mind is that if you apply the  $k$ -Nearest Neighbor algorithm, make sure that the distance metric that you're using to identify the  $k$ -Nearest Neighbor algorithm —  $k$ -Nearest Neighbors really measures some kind of semantic meaning of similarity; that data points that are similar according to their distance really are similar in the — you know, whatever you consider your data set to be.

One thing that's lately become very fashionable is to learn a distance metric. So, one thing that people, for example, do is they take deep convolutional neural networks to learn representations for images, and then in that learned representation they use the  $k$ -Nearest Neighbor algorithm. And this is, for example, how face classification algorithms work at Facebook, Google, and these kind of companies.

*Note: "Eucdist" refers to the use of Euclidean distance in the  $k$ -NN graph examples.*

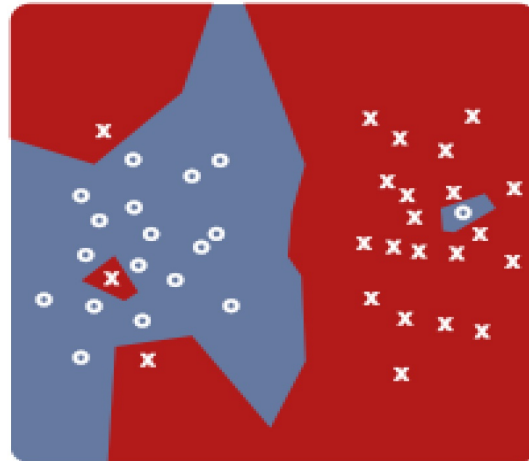
---

[Back to Table of Contents](#)

## Read: k-NN Hyperparameters

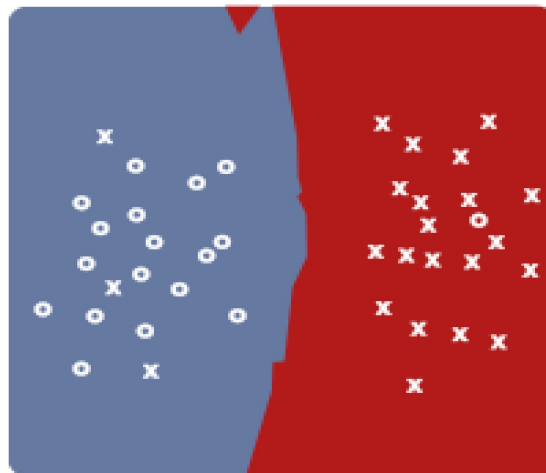
- The number of neighbors, or  $k$ , is the hyperparameter of k-NN that you can manipulate to improve the classification accuracy of the k-NN algorithm
- A relatively low  $k$  value will be more sensitive to variation in your data and may not be appropriate for noisy data
- A relatively high  $k$  value will have a simpler boundary and will handle noisy data well but might become too smooth for truly complex data sets

The k-NN classifier has exactly one parameter and that is  $k$ , the size of the neighborhood that we pick to infer the label. The best choice of  $k$  depends on the specific data set. We 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 we change  $k$ ?



euclidist  $k = 1$

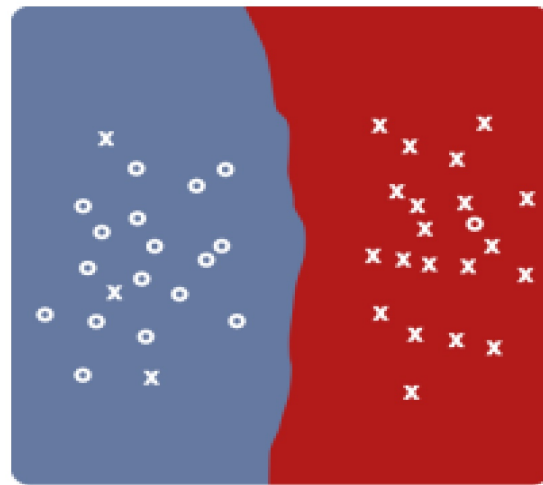
The k-NN decision boundary here for one nearest neighbor is actually quite rough and picks up on little movements in the data set. Also, you might see “islands” of misclassified, mislabeled data points, as k-NN carves out regions around these mislabeled data points. If you actually suspect your data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



euclidist  $k = 3$

If you increase  $k$  to 3, what happens is now you're averaging local neighborhoods and the little “islands” get washed out as they are outvoted by the other data points. The complexity of the boundary

decreases and becomes smoother with 3-NN.



euclidist  $k = 9$

The decision boundary becomes even 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. However, you may find 9-NN begins misidentifying significant groups of data points, which becomes more likely as you increase the number of neighbors used.

---

[Back to Table of Contents](#)

## Watch: Enhancing k-NN

There are a few things you can do to improve k-NN. In this video, you'll see how data augmentation and structures, as well as techniques for resolving ties, can make the k-NN algorithm even more accurate and efficient.

### Video Transcript

If you want to make the k-Nearest Neighbor algorithm truly competitive, there's a few more tweaks you can do to make it a lot more effective. The first one is to deal with ties. So if you have multiple classes, it could easily be that you actually get a tie within your neighborhood. So, for example, the following example, here you have a test point in the middle, and you have three nearest neighbors, but they all have a different label; a circle, a triangle, a square. So what should you do in this case? And a simple trick is to fall back on a lower k. So in this case,  $k=3$ . If  $k=3$  results in a tie, you just reduce k by two and you end up with  $k=1$ , and  $k=1$  never can result in a tie, so you are done. That's very effective; you can also use that in your project when you build your own k-Nearest Neighbor classifier.

Another trick that's very effective is called data augmentation. Data augmentation doesn't just work for k-Nearest Neighbors; it works for most machine learning algorithms, but it's particularly effective for k-Nearest Neighbors. And that's the following: Just take a data point, and — you take all your training data, and what you do is you apply some transformations that preserve the label. So imagine, for example, a data as an image; for example here, an image of a cat. One thing we can do is we can just take that image and translate it a little bit. Or you can flip it horizontally, or we can rotate it, or we can warp it a little bit or crop it differently. All of these transformations are cat-preserving, or technically label-preserving; that means the label of the image does not change with any of these transformations. And that's a very effective way to increase your training data set size.

The K-Nearest Neighbor algorithm gets better and better the more

training data you have, and so it's only natural to take the data that you have and kind of multiply it and make it denser. Finally, the last aspect of the k-Nearest Neighbor algorithm is addressed as a downside of it. It's easy to implement, it's naturally goes to multiclass settings — that's all great — but has one downside. And that is the bigger the data set gets, the slower it becomes. Because you have to compute the distance to every single training data point from your test point in order to find out if it is amongst the k-Nearest Neighbors. One way to speed that up is clever data structures; for example, k-d trees or ball-trees.

These data structures all effectively have the same kind of principle that underlies them, and that is the following: They kind of partition the space into boxes or balls, and then you take a test point and you compute distances to training points. And because training points are all kind of captured in boxes, what you do is before you compute the distance to a set of training points that are inside a box, you first compute the distance of the whole box. And if that whole box is already further away, then your current best or closest data point — then you know that no point inside can be amongst the k-Nearest Neighbors, so you just discard it. And you never actually have to compute these distances at all. So that can speed up your k-Nearest Neighbors algorithm tremendously, especially in low-dimensional data.

---

[Back to Table of Contents](#)

## Read: Enhancing k-NN

As Professor Weinberger previously described, the k-NN algorithm can be enhanced in a number of ways:

### Resolving Ties

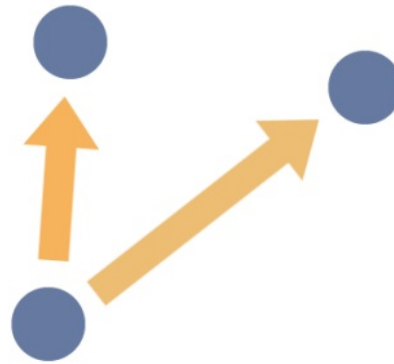
There can be cases without a unique most common label. For example, if  $k=4$ , you could retrieve two neighbors of one class and two of another. To avoid this, we typically pick odd values of  $k$ . However, even in those cases (e.g.,  $k=3$ ) your neighborhood vote may result in a tie. For example, 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 our 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 k-NN 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 k-NN 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 Augmentation

Data augmentation is a generic technique that can be used to improve any machine learning algorithm. For image data, there are many transformations that can be used to augment your data. For instance, you can flip, rotate, or translate an image, all while preserving the label of the image. You can safely add these transformed images into your training set without adding noisy labels to the data set. In return, you get a larger and more robust data set that will improve the generalization of our model.





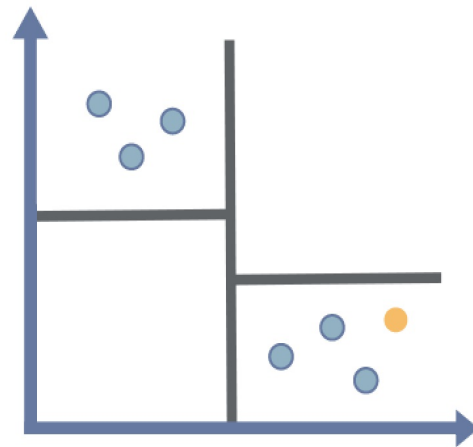
## Data Structure for Speedup

One downside of k-NN is that during test time, you have to compute distances from each test point to every training point. Testing becomes more intensive and slower as you increase the size of the training set. Formally, with  $d$ -dimensional vectors for each data point and  $n$  training points, calculating distances between a test point and each training point requires on the order of  $nd$  computations (naively calculating distance between any two points involves  $d$  subtractions,  $d$  exponentiations to power 2, then  $d - 1$  additions followed by a square root — adding these gives  $cd$  operations where  $c$  is a constant).

One way to speed up k-NN 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  $2^d$  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 the test point to all of the training points ( $nd$  computations) — even to the points in the

upper box that are clearly far — you simply compute the distance to the box containing those points in the upper box and ignore them if they are further away than the box is.

How is this faster than before? Finding the distance to a box involves finding distances to its edges. With points in  $d$ -dimensional space, you first compare the first coordinate of the test point to the split value on the first axis. We rule out any training points with a first coordinate distance from the test point's first coordinate that is greater than the distance of the first split value from the test point — those training points are clearly far away! In the next iteration, we check the next coordinate of the test point and split value of the corresponding axis to rule out any points that were not ignored in the previous iterations. Assuming each axis' split exactly rules out half of the remaining points, let's count the number of computations. Comparing on the first axis requires on the order of  $n$  computations. Comparing on the next axis now only requires  $n/2$  computations, since half of  $n$  training points were ruled out in the last iteration. The next axis requires  $n/4$  computations and



so on. Following the pattern, there are  $n + n/2 + n/4 \dots$  computations until we hit the required  $k$  nearest neighbors. Recall the sum of the infinite series  $1 + 1/2 + 1/4 + \dots = 2$ . Therefore, we perform less than  $2n$  computations. Since  $d$  is generally large, k-d trees do just order-of- $n$  computations compared to  $nd!$

There is a tradeoff, however. Naive nearest neighbors just involves storing the training points in memory, whereas building k-d trees also involves finding the axis split values. Hence, k-d trees are a little more computationally expensive during training but are cheaper during testing, whereas, naive nearest neighbors is cheap during training but expensive during testing. Depending on whether you have more training or testing data, you will have to decide which implementation of nearest neighbors to use.

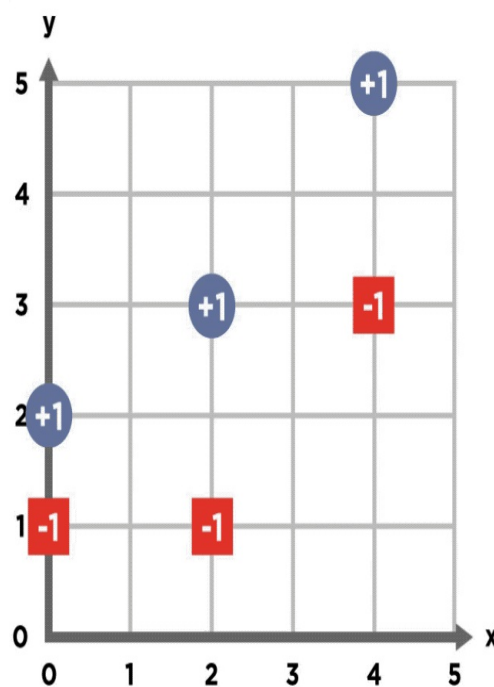
---

[Back to Table of Contents](#)

## Activity: Find a Decision Boundary

### Find a Decision Boundary

The k-NN 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, the decision boundary for a set of data might look like this:

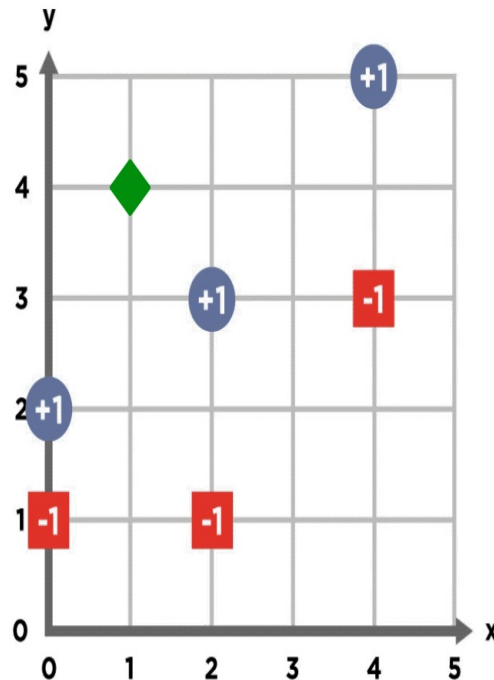


### Distance Metric

Note that our choice of distance metric also influences the determination of the decision boundary: If we choose L2 (Euclidean) distance as in the example above, the shortest distance between two points is along the line that connects them.

Alternatively, if we choose a different distance metric, such as L1 (taxicab) distance, we can only travel along vertical and horizontal axes, much like a taxicab driving through a city grid. For example, let's determine the L1 (taxicab) distance between the green diamond at (1,4)

and the positive labelled training point at  $(0,2)$  on the graph below. To get to the point  $(0,2)$  from  $(1,4)$ , we 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 by hand is to find two points of opposite class that you suspect are near the boundary (e.g., the +1 at  $(2,3)$  and the -1 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

Consider you have a data set and you want to draw a decision boundary. You have chosen to use 1-NN and will use the L2 distance metric. Suppose you have the following data points, strictly confined within a  $[0, 5] \times [0, 5]$  grid:

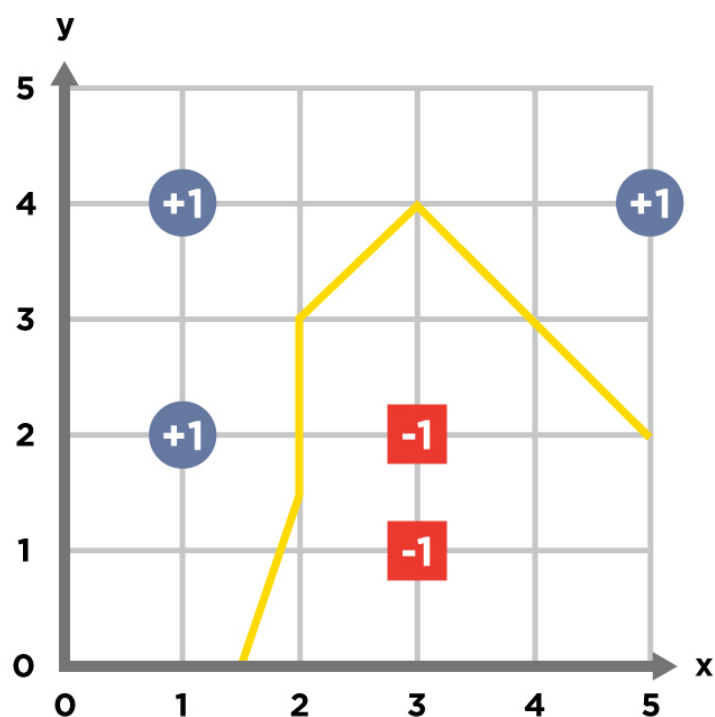
- Class +1:  $(1, 2)$ ,  $(1, 4)$ ,  $(5,4)$
- Class -1:  $(3, 1)$ ,  $(3, 2)$

---

Answer the following questions. Once you are ready, view below to reveal the answers.

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

Solution to question 1:



Solution to question 2: **the new point at (5,1) would be classified as "-1."**

---

[Back to Table of Contents](#)

## Watch: The Curse of Dimensionality

As we increase the number of dimensions for each data point in a data set, we find that individual data points become less and less similar to one another and the k-NN algorithm becomes less applicable. Professor Weinberger explains why this happens and when it can be avoided.

### Video Transcript

The k-Nearest Neighbor algorithm makes one crucial assumption, and that is that similar inputs have similar labels. It's interesting to think about when that assumption becomes less applicable, and that's exactly the case in high-dimensional spaces. What do we mean by high-dimensional spaces? Well, our data is represented as these vectors  $x$ , and  $x$  are basically these feature vectors; every single dimension is one attribute of a data point. The more such attributes you collect — so, for example, it could be your height or your weight, or something; the more we collect, right, the more high-dimensional that vector gets. And essentially — let's say we collect a hundred features; that means every single training point is just one point in a high-dimension — hundred-dimensional space.

Turns out in high-dimensional spaces there is something called the curse of dimensionality, and that means, loosely speaking, that no point is similar to any other point anymore. All the points are spread out. It becomes exceedingly unlikely that two points are very, very similar. So the assumption of the k-Nearest Neighbor algorithm, that similar points have similar labels, is useless, because nothing is similar to each other anymore. This is somewhat counterintuitive because we are so used to low-dimensional spaces, but let me walk you through a little example.

So imagine we have a training data set, patient data. So the first dimension is height, right, so we have the height of all our patients. Well, I'm 6 feet tall; there's many, many people who are 6 feet tall. So, many data points; many people are similar to me in that respect. So in one-dimensional space, no problem, but now we're adding more dimension. Second dimension that says age. Well, now, for a data point to be similar to me, you would have to match me in height, but also in age.



Now imagine we're also adding weight and blood pressure, and so on, and the entire medical history, so we add more and more different dimension. For someone to be similar to me, you would have to be similar in every single one of these dimensions. And once we get to 10,000 dimensions, for example, it would be exceedingly unlikely that someone has exactly my height, my age, and my medical history, and everything else. That basically means that the k-Nearest Neighbor algorithm becomes useless, all right, because there's no one like me anymore, so I can't make any predictions about someone like me.

So, essentially what that means is, when we have data, very high-dimensional feature vectors, high-dimensional data, the k-Nearest Neighbor algorithm stops working. That's a problem because most data is actually high-dimensional. So does the k-Nearest Neighbor algorithm not at all work in high dimensions? Well, it turns out, kind of. It doesn't work in high dimensions if the data is truly high-dimensional, but there is some rescue, and that is if the different dimensions are actually correlated with each other.

Let me give you an example. We just talked about blood pressure and weight, but actually those two are highly correlated; if you are overweight, you're also likely to have high blood pressure. So maybe these are not independent; maybe if you are similar in weight, you're also probably similar in blood pressure, right? So if that is the case, if dimensions are correlated, then you don't actually have a truly high-dimensional data set; you actually have lower-dimensional data that's just embedded in a high-dimensional. Then it's fine; the k-Nearest Neighbor algorithm is still fine because we still have similar data points.

Another example is images; let's say you have images of faces, and let's say I try to build a face recognition system. Given the face of a test person, I would like to find who is most similar in the training data set. Images of faces would be extremely high-dimensional, right? For example, a phone camera may have six megapixels; that's six million numbers, right, that actually describe a face. But imagine the following: Imagine someone steals your wallet and you go to the police and describe the face of the person to the police. How many questions would they ask you? They wouldn't ask you six million questions, right? They wouldn't ask you, "What's the color of the pixel in the top left corner?"

They would ask you something like, "What's the gender of the person? What's the age; middle age? Does the person have black hair?" etc. And with maybe 30, 50 questions, you could narrow down pretty well what the person looks like.

So essentially what's going on is, a few dimensions dominate the space, right? And if you are similar in those dimensions, you're probably similar. And the other dimensions are not very interesting; they're just small details about your skin, etc.; that's not very, very important. So, if your data is intrinsically low-dimensional, if features are correlated with each other or you basically have a few dimensions that dominate everything, then the k-Nearest Neighbor algorithm can still work even in high-dimensional data. It's your job as a data scientist to look at the data set and look and see if this is the case; if the data's actually too low-dimensional and if the k-Nearest Neighbor algorithm will still work. So keep that in mind; if the k-Nearest Neighbor algorithm does not work, it could be because your data set is just too high-dimensional, and then maybe you want to use a different algorithm.

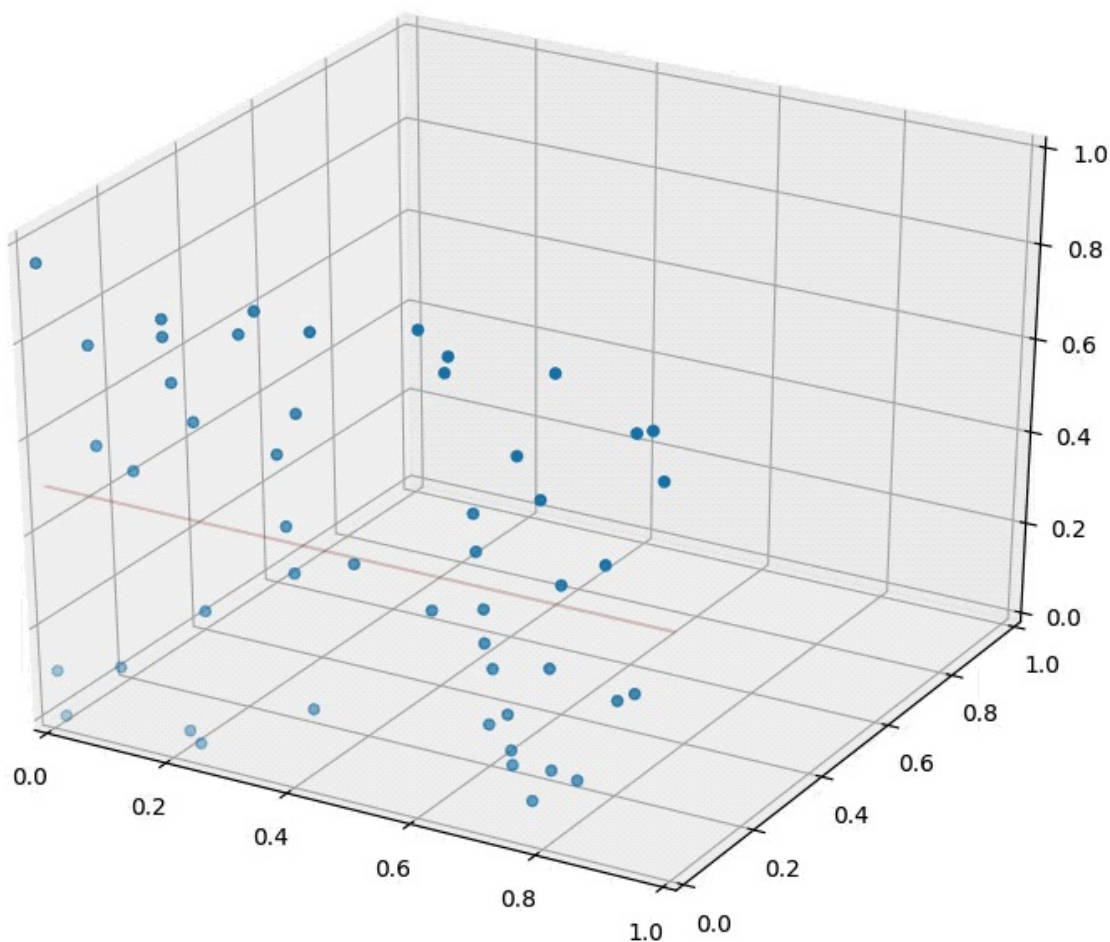
---

[Back to Table of Contents](#)

## Read: The Curse of Dimensionality

The curse of dimensionality is a particular challenge for our k-NN algorithm. As the number of dimensions increases — that is, as we include more and more features in our data set — all of our data points become more unique. Eventually, our data points are so dissimilar that the approach of finding close neighbors to predict the label of a test point is no longer feasible.

To better appreciate this behavior, let's visualize how the curse of dimensionality impacts our data set. The animation below shows what happens to data points with two dimensions after we add a third dimension.

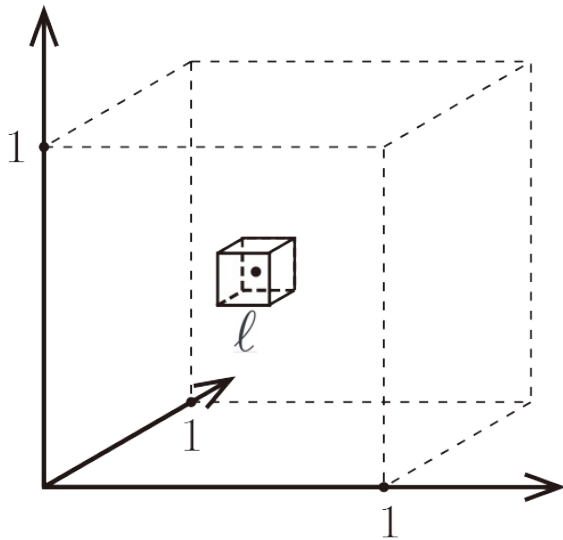


As the data points expand along the third dimension, their relative pairwise distances increase. However, the distance to the red plane (initially a line in 2D) stays constant. This is a typical behavior in high dimensions; randomly sampled points in high dimensions tend to be spread out from each other (with roughly equal distances) but tend to be comparable close to separating hyperplanes.

---

[Back to Table of Contents](#)

## Read: High-Dimensional Spaces



### The Curse of Dimensionality in Practice

Imagine  $X = [0, 1]^d$ , and all training data is sampled *uniformly* from  $X$ ,  
i.e.  $\forall i, x_i \in [0, 1]^d$

Let  $\ell$  be the edge length of the smallest hypercube that contains all  $k$ -NN  
of a test point. Because all training points  $n$  are spread uniformly  
throughout the cube,  $k$  neighboring points occupy roughly  $\frac{k}{n}$  of the total  
volume of the hypercube,  $\ell^d$ , which is equivalent to:

$$\ell \approx \left( \frac{k}{n} \right)^{1/d}$$

If  $n = 1000$  and  $k = 10$ , how big is  $\ell$  as we increase  $d$ ?

$d$	2	10	100	1000
-----	---	----	-----	------

$\ell$	0.1	0.63	0.955	0.9954
--------	-----	------	-------	--------

As the number of dimensions increases, the length of  $\ell$  quickly grows such that almost the entire space is needed to find the nearest neighbor. For example, when  $d = 100$ , the value of  $\ell = 0.955$ . In other words, the little cube is almost as big as the full hypercube (which has an edge length of 1). This expansion as we increase dimensionality means that our nearest neighbors are no longer near to our test point. Essentially, for each data point, all  $k$  nearest neighbors are far away, along the edges of the cube. Importantly, as the space along the edges is very thin, the  $k$  nearest neighbors are almost as far away as all other points, which are not nearest neighbors, and the k-NN algorithm is no longer effective.

---

[Back to Table of Contents](#)

## Part Two — Application and Limitations of k-NN

In this part of the course project, you will answer four questions about the application and limitations of k-NN. *Completion of this project is a course requirement.*

### Instructions:

- Download the [course project document](#), if you have not done so already.
- Complete Part Two, answering the questions provided within the table. Limit your answers to 100 words.
- Save your work as one of these file types: .doc, .docx, .txt, .pdf, .xls, .xlsx. No other file types will be accepted for submission.
- Now you will submit both parts of your completed course project for grading. Click the **Submit Assignment** button on this page to attach your completed course project document (Parts One and Two) and send it to your instructor 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: Classification With the k-Nearest Neighbors Algorithm

In this module, you learned about the k-NN algorithm and its underlying concepts. You had the opportunity to see firsthand how a decision boundary is formed and how the hyperparameter "k" impacts this boundary. Finally, you explored the curse of dimensionality and its impact on k-NN.

---

[Back to Table of Contents](#)



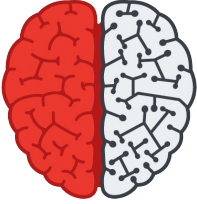
# Module 3: NumPy and Jupyter Notebooks

- [Module Introduction: NumPy and Jupyter Notebook](#)
- [Watch: Using the NumPy Library in Python](#)
- [Read: Why Vectorization Is Faster](#)
- [Tool: NumPy Cheat Sheet](#)
- [Code: Introduction to NumPy](#)
- [Code: Practice Matrix Multiplication](#)
- [Code: Additional NumPy Exercises](#)
- [Read: Using Jupyter Notebook](#)
- [Use a Jupyter Notebook](#)
- [Module Wrap-up: NumPy and Jupyter Notebook](#)

---

[Back to Table of Contents](#)

## Module Introduction: NumPy and Jupyter Notebook



Now that you have explored the concepts behind the k-NN algorithm, you are nearly ready to start coding your algorithm. In order to prepare you to implement k-NN, we must first introduce you to NumPy, a Python library for scientific computing. You will have the opportunity to practice using the key functions of NumPy that you will use in this and future courses to implement machine learning algorithms efficiently. You will also be introduced to Jupyter Notebook, a tool used by data scientists to create and share code, which we will use for exercises and projects later in the course.

---

[Back to Table of Contents](#)

## Watch: Using the NumPy Library in Python

In order to build an efficient facial recognition system, Python alone is not sufficient. In this video, Professor Weinberger demonstrates how loops in Python can slow down your code unnecessarily. As an alternative he shows you how to leverage NumPy functions to simplify and speed up your code.

### Video Transcript

The project in this class will be to build a face recognition system, and we want you to implement it in Python using NumPy. Now, you may be familiar with Python already, or maybe you're all in on NumPy, but in case you don't, it's a library to do linear algebra operations really, really fast, and it's really convenient.

Now, everything you implement in NumPy you could also implement in Python, but I want to convince you that NumPy is way more convenient and ridiculously much faster. Let me give you a little example. So for the k-Nearest Neighbor classifier, we have, let's say,  $n$  training points and we have  $m$  test points, and we need to compute the distance between any training point and any test point. In this little video, let's not compute distances; let's just compute inner products, which is very close to distance.

So, just for now, let's say we have  $n$  training points; we store them in a matrix  $X$ , where every single row is a training point. And we have  $m$  testing points, where every single row in this matrix  $Z$  is a test point. And we would like to compute a matrix  $G$  such that  $G[i,j]$  is the inner product between test point  $Z(i)$  and training point  $X(j)$ . So how do we do this? Let me show you a little example here in this Jupyter Notebook. So, for now we can just create random training and test points. So,  $X[m,i]$  training points,  $Z[m,i]$  test points, I now create 500 training points and 160 test points. So it's not a very large data set.

So here's a very simple, naive Python implementation. So here's what I would do; I'd say, "Well, in order to create this matrix  $G$ , I need to go over

all my training points and all my test points and compute the inner product between any two of them." So what I'm doing is  $i$  here iterates from one to  $n$  of my training points,  $j$  iterates over my testing points, and then I say  $G[i,j]$  — so that's the inner product between these two points — is now the sum of the cumulative product of every single dimension.

So what I'm doing,  $k$  now is a for-loop over all the dimensions, and it can be the product of every single dimension; that's the inner product, and I'm done. So it's three nested for-loops. I can take this code and execute it and run it on my training data set, and you can see it's running, running, running, running, running, and it takes about 4.2 seconds on this particular data set. So it's okay; it's not too bad. Let's, in comparison, do this with NumPy.

Now — well, one thing you need to figure out is that actually when you have these matrices, right,  $X$  and  $Z$ , we get a product — actually, we could just write this as a matrix operation;  $G$  equals  $X \cdot Z^T$ . So, you just paste this in; this is exactly — actually, NumPy is just one line of code. So  $G$  equals  $X$  times  $Z$  transpose. And then we return  $G$ . If you run that code instead — so the one thing you notice is this code is much, much, much shorter, and it describes exactly the linear algebra operations you need to do. It's just this matrix multiplication; you have no loops. The other thing is Python is actually really, really slow with loops. So before we had three nested loops; that's something Python is really bad at. So what we do instead is we run this — and NumPy, which is really, really fast and steady; we can now run it and see how long it takes us. So before it took us 4.2 seconds. Now if we run it, it takes about 1.4 milliseconds. If you run it one more time, it's less than 1 millisecond. So just to make sure to appreciate the difference in time here, the previous iteration was 4,000 times slower.

So imagine you write some code that takes one day to train; that's typical for a machine learning algorithm. If you have a large data set, right, and your company — you run something over this whole training data set, it can easily take you one day to train. If you take the fast implementation, it takes one day, if you take the slow implementation, it would take you over 11 years for the same code to finish. Right? So think about the difference between waiting one day or 11 years. That's the difference between implementing it in NumPy and implementing it in Python directly.

---

[Back to Table of Contents](#)

## Read: Why Vectorization Is Faster

**Vectorization** is the process of rewriting a loop so that instead of processing elements of an array one at a time, it processes several elements of the array simultaneously.

Under the hood, much of the NumPy library is written in C, which allows for more efficient lower-level operations than Python offers natively. In the case of vectorization, NumPy takes advantage of "single instruction, multiple data" operating, which is a type of parallel processing that calls for the processor to run the same operation simultaneously over multiple data points. Since such operations would have to be run sequentially if a Python for-loop was utilized, vectorization offers a significant performance improvement.

### Example of Vectorization

Say we wanted to square every element of the following list:

[2, 3, 4, 5]

Using a traditional loop, our code would process each element of the list one at a time, squaring each number to yield [4, 9, 16, 25].

Using vectorization, however, we send the array as a "batch" to lower-level code, which can perform the squaring operation simultaneously.

---

[Back to Table of Contents](#)

# Code: Introduction to NumPy

---

[Back to Table of Contents](#)

# Getting Started

*This guide leads you through a series of exercises that introduce you to NumPy and familiarize you with the virtual programming environment used in this course. There is a button in the bottom-right corner of this virtual programming interface that allows you to toggle fullscreen mode.*

---

NumPy is a Python package for scientific computing that is especially great for working with n-dimensional array objects, which you will use heavily in this and subsequent courses. You can find more information at <http://www.numpy.org/>.

For these exercises, you'll use the iPython interpreter. You'll also need to import NumPy to get started.

## Instructions

1. First, start the iPython interpreter by typing `ipython` at the command prompt (you can begin typing after `$`). The prompt for iPython will look a bit different, beginning with `In [ # ] : .` You can now run Python commands at this prompt.
2. Import NumPy with the following command:

```
import numpy as np.
```

*Note: Executing the `import` command will not return a response. Once you've completed this step, go to the next page of this guide.*



## Arrays in NumPy

NumPy's main object is an n-dimensional array, a table of the same data type. To create an array, you need to pass a list of objects into the function `np.array()`. Let's create a 2-D array, a matrix of size 2×3. Here's an example:

```
X = np.array([[1,2,3],[4,5,6]])
```

### Instructions

1. Create a multidimensional array `X` that has two rows and three columns using `np.array()`.
2. `print()` array `X` to confirm you've created a 2×3 array (matrix).

*Note: You can access the iPython help utility by typing `help()` at the command prompt.*

# Attributes of NumPy Arrays

Understanding how to look up attributes of an array can be very useful. There are three important attributes of a NumPy array that you may like to know:

- The shape of an array
- The dimension of an array
- The total number of elements of an array

## Instructions

1. Check the shape of array `X` using `X.shape`.
2. Check the dimension of array `X` using `X.ndim`.
3. Check the total number of elements of array `X` using `X.size`.

# Vectors in NumPy

You can use arrays to represent a vector, which is essentially a 1-D array. There are three ways to represent a vector using NumPy. You'll first start by using `np.array()` to create a 1-D array. Here's an example:

```
npvec = np.array([1,2,3])
```

This array, which we will refer to as a NumPy vector, has the shape of `(3,)`, meaning the array is indexed by a single index from 0 to 2.

You can then use `.reshape()` to transform a NumPy vector into a column or a row vector. For example, you could reshape our NumPy vector `npvec` into a column vector with the following line:

```
colvec = npvec.reshape((3,1))
```

## Instructions

1. Create a NumPy vector `v1` that contains `([3,4,5])` using the `np.array()` function.
2. `print()` the vector `v1` to confirm you've correctly created the vector.
3. `print()` the vector's shape `v1.shape`. You should find that its shape is `(3,)`
4. Using the function `.reshape()`, reshape the `v1` NumPy vector into a column vector `v2` by changing the shape to `(3,1)`
5. `print()` `v2` to confirm your data is stored in a column vector.
6. Reshape the `v1` NumPy vector into a row vector `v3` by changing the shape to `(1,3)`
7. `print()` `v3` to confirm your data is stored in a row vector.

# Flatten Vectors

The three representations of a vector (NumPy, column, and row) are often not compatible for most operations. Some operations might “work,” but will not return the expected output. To make this point, let’s try simple addition with a NumPy vector `v1` and a column vector `v2` in their current form.

## Instructions

- Add vectors `v1` and `v2` together and review the output.

The sum of your NumPy vector and column vector should look like this:

```
[[ 6  7  8]
 [ 7  8  9]
 [ 8  9 10]]
```

This is not the expected or desired output. In their current form, these vectors just can’t be added in the way we intend. Instead, you must first transform `v2` so that it is also in NumPy vector format like `v1`. For the purposes of our work, we will always prefer the NumPy vector format. You can transform any vector (matrix) into a NumPy vector with `.flatten()`.

## Instructions

- Create a new vector `v4` that is a flattened version of `v2` using `.flatten()`.
- Add vectors `v1` and `v4` together and review the output.

If you’ve correctly flattened `v2`, the sum of your NumPy vectors should look like this: `array([6, 8, 10])`. This is the desired output when summing vectors that both contain `[3, 4, 5]`

## Creating Common Matrices

There are also a few functions in NumPy for creating common matrices:

- `np.eye()` creates an identity matrix
- `np.zeros()` creates a matrix of all zeros
- `.arange()` creates a vector of equally spaced values

### Instructions

1. Create an identity matrix `np.eye(3)`.
2. Create an identity matrix `np.eye(5)`.
3. Create an identity matrix of zeros `np.zeros([2,3])`.
4. Create an identity matrix of zeros `np.zeros([3,6])`.
5. Create a vector of equally spaced values `np.arange(10)`.
6. Create a vector of equally spaced values `np.arange(1,10,2)`.
7. Create a vector of equally spaced values `np.arange(2,10,2)`.

*Note: `np.arange[start,end,step]`*

# Matrix Operations

Now you'll explore a few matrix functions that will be useful in future projects. Rather than writing functions in Python from scratch, there will often be preexisting NumPy functions that can save you time and make your code run faster.

## #### Multiplication

To perform matrix multiplication, use `np.dot()` or the `@` symbol. For example, you could multiply matrices `X` and `Y` by writing either `np.dot(X,Y)` or `X @ Y`.

### **Instructions**

1. To begin, create the following matrices:

```
X = np.array([[1,2], [3,4]])

Y = np.array([[5,6], [7,8]])

A = np.array([[1,2,3,4], [7,8,9,10]])

B = np.array([[4,5,6,7], [1,2,3,4]])
```

2. `print()` matrix `X` to see its content.
3. `print()` matrix `Y` to see its content.
4. Multiply `X` and `Y` using `np.dot()`
5. Multiply `X` and `Y` using `@`
6. Multiply `A` and `B` using `@`

You should have been able to successfully multiply `X` and `Y` but not `A` and `B`. This is because the number of columns in `A` does not match the number of rows in `B`. In order to successfully multiply these two matrices, we must first transpose one of them.

## #### Transpose

To transpose a matrix, use `.transpose()`. An alternative and more concise syntax is `.T`.

For example, you could transpose matrix `Y` by writing either `Y.transpose()` or `Y.T`.

### ***Instructions***

1. `print()` matrix B to see its content.
2. Transpose matrix B using `.transpose()`.
3. Transpose matrix B using `.T`.
4. Multiply A and the transpose of B.

# Self-Check #1

## Instructions

1. Use NumPy to calculate:

$$AB^T - Y$$

2. Check your results. The output should be:

```
array([[55, 24],  
       [185, 82]])
```



## Other Matrix Operations

Other useful functions you will use in your projects are `np.amax()`, `np.amin()`, `np.argmax()`, `np.argmin()`, and `np.sum(X)`. With all of these functions, you can specify the axis along which you want to perform the operation. If no axis is specified, the functions will perform the operation across all dimensions. Here are some examples of these functions in use:

Function	Example
Largest element in X	<code>np.amax(X)</code>
Largest elements along the first axis	<code>np.amax(X, axis = 0)</code>
Smallest element in X	<code>np.amin(X)</code>
Smallest elements along the second axis	<code>np.amin(X, axis = 1)</code>
Index of the smallest element in X	<code>np.argmin(X)</code>
Indices of the largest elements along the first axis	<code>np.argmax(X, axis = 0)</code>
Sum of all elements in X	<code>np.sum(X)</code>
Sum of elements along the first axis	<code>np.sum(X, axis = 0)</code>
Sort the matrix (in ascending order) along the last axis ( <code>axis=-1</code> )	<code>np.sort(D)</code>
Sort the matrix along the first axis	<code>np.sort(D, axis=0)</code>
Indices for sorting matrix (in ascending order) along the last axis ( <code>axis=-1</code> )	<code>np.argsort(D)</code>
Indices for sorting matrix along the first axis	<code>np.argsort(D, axis=0)</code>

### #### Instructions

1. `print()` matrix Y to see its content.
2. Find the largest element in Y.
3. Find the smallest elements along the first axis of Y.
4. Find the index of the smallest element in Y.
5. Find the indices of the largest elements along the second axis of Y.
6. Find the sum of elements in Y along the second axis.
7. Find the indices of the 2 largest elements along the second axis of Y using `np.argsort`. (Use negative slicing on the second axis after `argsort`-ing in ascending order by default).

## Find Multiple Smallest Values

You learned how to find the index for the smallest value for each row in a matrix. You might need the  $n$  lowest values in a row rather than simply the lowest. In this exercise, you will find the three smallest values for each row in a matrix. This is useful when you are implementing  $k$  nearest neighbors, when  $k \geq 1$ .

### Instructions:

1. Create the following matrix by typing:

```
D = np.array([[1, 4, 8, 5], [4, 8, 2, 9], [2, 0, 4, 5]])
```

2. Find the indexes for the three lowest values for each row. You should get a matrix in the form:

```
np.array([[0, 1, 3],  
[2, 0, 1],  
[1, 0, 2]])
```

### Implementation:

# Adding Columns and Rows to Matrices

The `np.vstack` and `np.hstack` functions can be used to add columns or rows to existing matrices. For example, let's look at matrices A and B:

```
[[ 1 2 3 4]
A: [ 7 8 9 10]]
```

```
[[4 5 6 7]
B: [1 2 3 4]]
```

Using the following function `np.vstack((A,B))`, vertical stacking concatenates vectors along the first axis, such as in this example output:

```
[[ 1 2 3 4]
 [ 7 8 9 10]
 [ 4 5 6 7]
 [ 1 2 3 4]]
```

Using the following function `np.hstack((A,B))`, horizontal stacking concatenates vectors along the second axis, such as in the example output:

```
[[ 1 2 3 4 4 5 6 7]
 [ 7 8 9 10 1 2 3 4]]
```

## Instructions

1. `print()` matrix X to see its content.
2. `print()` matrix Y to see its content.
3. Vertically stack X and Y.
4. Horizontally stack X and Y.

## Self-Check #2

### Instructions

1. Find the maximum elements in A along the first axis (axis = 0) and add it to the sum of elements in B along the first axis.
2. Check your results. Your output should be:

```
array([12, 15, 18, 21])
```

## Element-wise Matrix Operations

There are many operations that you might want to perform, such as taking the square root or exponent on each element of a NumPy array. Some examples of these functions are `np.exp()`, `np.sqrt()`, and `np.square()`. There are many NumPy functions that will perform element-wise operations on NumPy arrays. Refer to the official documentation for more information.

### Instructions

1. Create a new matrix `Z` using `np.arange()`.
2. `print()` matrix `Z` to see its content.
3. Find the exponential applied element-wise to `Z`.
4. Find the square root applied element-wise to `Z`.
5. Find the square applied element-wise to `Z`.

# Indexing and Slicing

NumPy arrays can be indexed and sliced, just like Python's list. Let's explore indexing and slicing on 1-D and multidimensional arrays.

## One-Dimensional Arrays

### *Instructions*

1. Create the following 1-D array:  
`x1 = np.arange(2,12,2)`
2. `print()` this new array.
3. Index a single element, such as `x1[3]`.
4. Slice a portion of the array, such as `x1[1:3]`.
5. Slice the last two numbers of the array using `x1[-2:]`.

## Multidimensional Arrays

### *Instructions*

1. Create the following 3-D array:  
`x2 = np.array([[1,2,3], [4,5,6], [7,8,9]])`.
2. `print()` this new array.
3. Find a full slice using `x2[:,:]`.
4. Index an element at the first row, third column using `x2[0,2]`.
5. Slice on both axes using `x2[1:, :2]`.
6. Index by row, selecting the first row using `x2[0]`.
7. Index by column, selecting the first column using `x2[:,0]`.
8. Iterate through each row in a matrix:  
`for row in x2: print(row)`.

## Self-Check #3

### Instructions

1. Return only the third and fourth column of matrix A
2. Check your results. Your output should be:

```
array([[3, 4],  
       [9, 10]])
```

Well done! You've completed this exercise. Move on to the next course page by clicking the **Next** button in the lower-right corner of the course page.

# Code: Practice Matrix Multiplication

---

[Back to Table of Contents](#)



# Import NumPy

In this section, we are going to practice applying what we've learned about NumPy to matrix manipulation.

## Instructions:

1. Start the iPython interpreter. Type `ipython` at the command prompt.
2. Import the NumPy library using:

```
import numpy as np
```

*Note: Executing the `import` command will not return a response. Once you've completed this step, go to the next page of this guide.*

# Matrix Multiplication

Let's try multiplying two matrices. We can create random matrices of any size using the `np.random.random()` function. We can then multiply those matrices using the `np.dot()` or `@` function.

Here's an example:

```
X = np.random.random((2,3))
```

```
Y = np.random.random((3,2))
```

```
np.dot(X,Y)
```

```
X @ Y
```

*Note: When multiplying two matrices, the number of columns in the first (3) must match the number of rows in the second (3).*

## Instructions:

1. Create two random 4x4 matrices using `np.random.random()`.
2. Compute and print the matrix product of the two matrices using `np.dot()`.
3. Compute and print the matrix product of the two matrices using `@`.

## Matrices of Different Dimensions

Now let's see what happens when we find the dot product of matrices with different shapes.

### Instructions:

1. Create a random 3x2 matrix and a random 2x4 matrix. Note: the `np.dot()` function adapts to matrices of different dimensions. [Review the documentation for more detail.](#)
2. Compute and print the product of the two matrices and assign it to a variable called `prod`.
3. Check that `prod` has the dimensions you expect using `prod.shape`.

# Multiplying Incompatible Matrices

As mentioned previously, multiplying matrices only works if the matrices shapes are compatible for such an operation (e.g. the number of columns in the first matrix matches the number of rows in the second).

## Instructions:

1. Create a random 5x3 matrix and a random 2x1 matrix.
2. Attempt to compute and print the product of the two, and observe the response.
3. You should see the following error:

```
ValueError: shapes (5,3) and (2,1) not aligned: 3 (dim 1) != 2 (dim 0)
```

## Multiplying by a Scalar

We can also multiply a matrix by a scalar.

### Instructions:

1. Create a random 9x4 matrix.
2. Compute and print the matrix scaled by a factor of 3.
3. Confirm that the result is what you expect by multiplying the first element of the first row by 3 and checking that it matches the first element of the first row of your output.

Note: the `np.dot()` function must be used when multiplying a matrix by a scalar.

# Multiplying by a Vector

Matrices can be multiplied by vectors (one-dimensional arrays).

## Instructions:

1. Create a random 5x6 matrix and a random 6x1 vector.
2. Compute and print the product of the two.

## Multiplying Row Vectors

The `np.dot()` function expects matrices to have compatible dimensions. Using the `np.outer()` function automatically transposes the first matrix before computing the product. Because the first matrix is transposed, the order of vectors matters. For example:

```
np.outer(x,y) != np.outer(y,x)
```

### Instructions:

1. Create two row vectors, one of length 4, one of length 6.
2. Compute and print the product of the two, first using `np.dot()`, then using `np.outer()`.

## Multiplying Column Vectors

Now let's use the `np.outer()` and `np.dot()` functions to compute the products of column vectors.

### Instructions:

1. Create two column vectors, one of length 3, one of length 8.
2. Compute and print the product of the two, first using `np.dot()`, then using `np.outer()`.

Well done! You've completed this exercise. Move on to the next course page by clicking the **Next** button in the lower-right corner of this course page.



# Code: Additional NumPy Exercises

---

[Back to Table of Contents](#)

## Compute a Mode Function

In this first exercise, you will write a *mode* function. The mode function takes as input a list and returns the most frequent element.

i.e. `mode([3,5,5,8,8,5]) = 5`

## Instructions

1. Launch the ipython shell.
2. `import numpy as np.`
3. Define a random list of integers using:  
`a = [np.random.randint(5) for x in range(10)].`
4. Implement the mode function and test it on list a, i.e. `mode(a)`.
5. Try this first yourself. Once you complete your function, review the alternative implementations provided below.

*Hint: Any list has a `count()` function. What does the following code return:*

`a.count(3)?`

## Alternative Implementations

## Mode Over Matrix Columns

Create a function that allows you to find the mode of columns.

# Instructions

1. Implement a `modeC` function that accepts a matrix as input and operates on each column.
2. Use the following matrix to test your function:  

```
M = np.int16(np.round(np.random.rand(10,8)*3.0))
```
3. First, try implementing it yourself. Once you complete your function, review the alternative implementations provided below.

### Alternative Implementations


## Add a Vector to a Matrix

Now you'll practice adding vectors to a matrix.

### Instructions

1. Use this matrix `M = np.random.rand(3,8);`.
2. Add vector `v = np.random.rand(3)` to each column of `M`.
3. Add vector `w = np.random.rand(8)` to each row of `M`.
4. First, try implementing it yourself. Once you complete your work, review the alternative implementations provided below.

### Alternative Implementations


## Set Negative Values to Zero

Now, replace negative vales in a matrix with zeros, which is useful when you need to compute square roots.

### Instructions

1. Set all negative values of matrix `Q = np.random.rand(5,5) - 0.5` to zero.
2. Compute the square root of the resulting matrix.
3. First, try implementing it yourself. Once you complete your code, review the alternative implementations provided below.

### Alternative Implementations


## Find Index of the Minimum Value for Each Row

Finding the index for the minimum value for each row.  
You will find the minimum value for each row in a matrix. This is useful  
when you are implementing nearest neighbors.

### Instructions

1. Create the following matrix by typing

```
D = np.array([[1, 7, 7, 8],  
              [4, 8, 2, 8],  
              [2, 0, 4, 5]])
```

2. Find the index for the minimum value for each row. You should get [0,  
2, 1]

### Alternative Implementations


## Repeat a vector to make a matrix

Repeat a vector to make a matrix

Here you will practice how a vector can be repeated along rows or columns to form a matrix. This will be helpful later in one of the projects.

### Instructions

1. Use the vector `v = np.random.rand(3)`.
2. Create a 3 by 8 matrix `M1`, where each column of the matrix is `v`. That is, all the entries in row `i` of `M1` are `v[i]`.
3. Next, create a 10 by 3 matrix `M2`, where each row of the matrix is `v`. That is, all the entries in column `j` of `M2` are `v[j]`.
4. First, try implementing it yourself. Once you complete your code, review the alternative implementations provided below.

### Alternative Implementations


## Tool: NumPy Cheat Sheet

Use this [NumPy Cheat Sheet](#) to reference NumPy functions

There are many useful functions to remember and utilize when working with NumPy. Here is a list of useful functions for your reference with a description of their uses and links to more information. If you would like to download or print this reference sheet, click the download link to the right.

<b>np.array()</b>	Usage: Create an array Reference: <a href="#">np.array()</a>
<b>np.reshape()</b>	Usage: Reshape the size of matrix X Reference: <a href="#">np.reshape()</a>
<b>np.flatten()</b>	Usage: Flatten a matrix to a NumPy vector Reference: <a href="#">np.flatten()</a>
<b>np.eye()</b>	Usage: Create an identity matrix Reference: <a href="#">np.eye()</a>
	Usage: Create a matrix of all zeros



<b>np.zeros()</b>	Reference: <a href="#">np.zeros()</a>
<b>np.arange()</b>	Usage: Create a vector of equally spaced values Reference: <a href="#">np.arange()</a>
<b>np.dot()</b>	Usage: Matrix multiplication. An alternative syntax is @ Reference: <a href="#">np.dot()</a>
<b>np.transpose()</b>	Usage: Transpose a matrix. An alternative syntax is .T Reference: <a href="#">np.transpose()</a>
<b>np.amax()</b>	Usage: Find the maximum value of an array Reference: <a href="#">np.amax()</a>
<b>np.amin()</b>	Usage: Find the minimum value of an array Reference: <a href="#">np.amin()</a>
<b>np.argmax()</b>	Usage: Find the indices of the maximum value of an array Reference: <a href="#">np.argmax()</a>

<b>np.argmin()</b>	Usage: Find the indices of the minimum value of an array Reference: <a href="#">np.argmin()</a>
<b>np.sum()</b>	Usage: Sum an array Reference: <a href="#">np.sum()</a>
<b>np.mean()</b>	Usage: Find the mean of an array Reference: <a href="#">np.mean()</a>
<b>np.vstack()</b>	Usage: Stacking arrays or vectors vertically Reference: <a href="#">np.vstack()</a>
<b>np.hstack()</b>	Usage: Stacking arrays or vectors horizontally Reference: <a href="#">np.hstack()</a>
<b>np.exp()</b>	Usage: Calculate the exponential of all elements of an array Reference: <a href="#">np.exp()</a>
<b>np.sqrt()</b>	Usage: Return the non-negative square root of each element of an array Reference: <a href="#">np.sqrt()</a>

**np.square()**

Usage: Return the square of each element of an array

Reference: [np.square\(\)](#)

---

[Back to Table of Contents](#)

# Read: Using Jupyter Notebook

## What Is Jupyter Notebook?

Jupyter Notebook is an open-source application that data scientists use to document, run, and share code. We'll use it in this and subsequent courses to implement and train our machine learning models. Future code exercises and projects in this and subsequent courses will be completed in the Jupyter Notebook environment, so it is important you understand the basics of how a notebook works.

## Using a Notebook

Once you've opened a notebook, you'll find a mix of text and code "cells." Some code cells can be modified and run as if you were working in a Python shell. You can use the controls in the menu to run code in the notebook or you can simply use the keyboard shortcut **shift + return** to run a code cell that is currently selected. Like in a Python shell, cells in a notebook must be executed sequentially.

## Feedback and Grading

Jupyter Notebooks in this and future courses may use an "autograder" script which contains test cases that will check your code and provide you with instant feedback. Exercise notebooks may have an autograder to give you feedback but the score will not be calculated in your final grade. Such exercises will be graded as Complete/Incomplete and will need to be completed in order for you to earn credit for the course. For project notebooks, an instructor will review your work in addition to the autograder. After reviewing your work, the instructor will determine your final grade and provide you with specific feedback when necessary.

---

[Back to Table of Contents](#)

# Use a Jupyter Notebook

---

[Back to Table of Contents](#)

## Module Wrap-up: NumPy and Jupyter Notebook

You are now ready to navigate Jupyter Notebook and use the NumPy package to begin implementing machine learning algorithms. The experience you've gained and the downloadable NumPy cheat sheet you now have should provide you with a solid foundation for coding up the k-NN algorithm to create a facial recognition system.

---

[Back to Table of Contents](#)

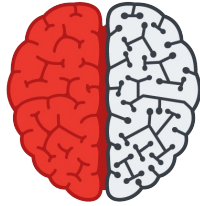
# Module 4: Building a Facial Recognition System

- [Module Introduction: Building a Facial Recognition System](#)
- [Activity: Compute Euclidean Distances in Matrix Form](#)
- [Code: Euclidean Distance Function Without Loops](#)
- [Ask the Expert: Laurens van der Maaten, Part Two](#)
- [Build a Facial Recognition System](#)
- [Module Wrap-up: Building a Facial Recognition System](#)
- [Read: Thank You and Farewell](#)

---

[Back to Table of Contents](#)

# Module Introduction: Building a Facial Recognition System



In this module, you will practice manipulating matrices using NumPy. You will then derive a Euclidean distance matrix and implement it in Python without loops. Finally, you will implement the k-NN algorithm in order to create a facial recognition system that will accurately label images of people in the provided data set.

---

[Back to Table of Contents](#)



## Activity: Compute Euclidean Distances in Matrix Form

### Compute Euclidean Distances in Matrix Form

Recall that the k-Nearest Neighbors algorithm is used to determine the label of a given data point based on the labels of the other data points closest to it. Thus, one must first determine pairwise distances between a testing data point and all training data points, which are represented as vectors. A commonly used distance metric for k-Nearest-Neighbors is the Euclidean distance (also called the L2 or squared distance). The following exercises demonstrate how you can compute Euclidean distances between data points.

In mathematics, typically a vector is represented as a  $d \times 1$  matrix (or column vector). However, in computer science it is often more efficient to store a vector as a row vector in a computer. For this reason, we are going to assume all vectors are now represented as  $1 \times d$  matrix (or row vector).

### Two points in $\mathbb{R}^d$ (two points with $d$ features)

The squared Euclidean distance between two points  $\mathbf{x}, \mathbf{z} \in \mathbb{R}^{1 \times d}$  can be expressed in these equivalent forms:

$$D^2(\mathbf{x}, \mathbf{z}) = \|\mathbf{x} - \mathbf{z}\|_2^2 = \sum_{i=1}^d ([\mathbf{x}]_i - [\mathbf{z}]_i)^2 = (\mathbf{x} - \mathbf{z})(\mathbf{x} - \mathbf{z})^\top = (\mathbf{x} - \mathbf{z}) \cdot (\mathbf{x} - \mathbf{z}) \in \mathbb{R}$$

To create a *vectorized* expression, we can distribute the terms using the matrix operations rules of linear algebra:

$$\begin{aligned} D^2(\mathbf{x}, \mathbf{z}) &= (\mathbf{x} - \mathbf{z})(\mathbf{x} - \mathbf{z})^\top = (\mathbf{x} - \mathbf{z})(\mathbf{x}^\top - \mathbf{z}^\top) \\ &= \mathbf{x}\mathbf{x}^\top - \mathbf{x}\mathbf{z}^\top - \mathbf{z}\mathbf{x}^\top + \mathbf{z}\mathbf{z}^\top = \mathbf{x}\mathbf{x}^\top - \mathbf{x}\mathbf{z}^\top - \mathbf{x}\mathbf{z}^\top + \mathbf{z}\mathbf{z}^\top \\ &= \mathbf{x}\mathbf{x}^\top + \mathbf{z}\mathbf{z}^\top - 2\mathbf{x}\mathbf{z}^\top \end{aligned}$$

Observe how the expression is similar to expanding  $(a - b)^2$  in 1 dimension:  $(a - b)^2 = a^2 + b^2 - 2ab$ . Euclidean distance with vectors in  $d$  dimensions is just a natural extension of Euclidean distance in 1 dimension.

## Two sets of points in $\mathbb{R}^d$

We will now use matrices to efficiently compute distances between two sets of points, say a training set of size  $n$  and a test set of size  $m$ .

Therefore, let us define our data matrices as  $\mathbf{X} \in \mathbb{R}^{n \times d}$ ,  $\mathbf{Z} \in \mathbb{R}^{m \times d}$ .

We can create a matrix  $D \in \mathbb{R}^{n \times m}$  of pairwise Euclidean distances where the  $ij^{th}$  entry is the distance between the  $i^{th}$  row vector  $\mathbf{x}_i$  in  $\mathbf{X}$  and  $j^{th}$  row vector  $\mathbf{z}_j$  in  $\mathbf{Z}$ . Note that we are using  $D$  to denote both the distance function and the resulting matrix, with its meaning being apparent from the context.

If we can calculate the matrix  $D$  (all pairwise distances between points in the training set and the test set) using some matrix operations, then we can leverage NumPy's optimized linear algebra functions to create an efficient program.

## Euclidean Distances using Matrices

Let us, for demonstration purposes, set  $n = 2$  and  $m = 3$ . Since the  $ij^{th}$  entry of the matrix  $D$  is exactly the distance  $D(\mathbf{x}_i, \mathbf{z}_j)$ , we can write the matrix  $D$  as:

$$\begin{aligned} D^2 &= \begin{bmatrix} D(\mathbf{x}_1, \mathbf{z}_1) & D(\mathbf{x}_1, \mathbf{z}_2) & D(\mathbf{x}_1, \mathbf{z}_3) \\ D(\mathbf{x}_2, \mathbf{z}_1) & D(\mathbf{x}_2, \mathbf{z}_2) & D(\mathbf{x}_2, \mathbf{z}_3) \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{x}_1\mathbf{x}_1^\top + \mathbf{z}_1\mathbf{z}_1^\top - 2\mathbf{x}_1\mathbf{z}_1^\top & \mathbf{x}_1\mathbf{x}_1^\top + \mathbf{z}_2\mathbf{z}_2^\top - 2\mathbf{x}_1\mathbf{z}_2^\top & \mathbf{x}_1\mathbf{x}_1^\top + \mathbf{z}_3\mathbf{z}_3^\top - 2\mathbf{x}_1\mathbf{z}_3^\top \\ \mathbf{x}_2\mathbf{x}_2^\top + \mathbf{z}_1\mathbf{z}_1^\top - 2\mathbf{x}_2\mathbf{z}_1^\top & \mathbf{x}_2\mathbf{x}_2^\top + \mathbf{z}_2\mathbf{z}_2^\top - 2\mathbf{x}_2\mathbf{z}_2^\top & \mathbf{x}_2\mathbf{x}_2^\top + \mathbf{z}_3\mathbf{z}_3^\top - 2\mathbf{x}_2\mathbf{z}_3^\top \end{bmatrix} \end{aligned}$$

Since multiplication distributes over addition in matrix algebra, the last expression can be written as:

$$D^2 = \underbrace{\begin{bmatrix} \mathbf{x}_1 \mathbf{x}_1^\top & \mathbf{x}_1 \mathbf{x}_2^\top & \mathbf{x}_1 \mathbf{x}_3^\top \\ \mathbf{x}_2 \mathbf{x}_1^\top & \mathbf{x}_2 \mathbf{x}_2^\top & \mathbf{x}_2 \mathbf{x}_3^\top \\ \mathbf{x}_3 \mathbf{x}_1^\top & \mathbf{x}_3 \mathbf{x}_2^\top & \mathbf{x}_3 \mathbf{x}_3^\top \end{bmatrix}}_S + \underbrace{\begin{bmatrix} \mathbf{z}_1 \mathbf{z}_1^\top & \mathbf{z}_1 \mathbf{z}_2^\top & \mathbf{z}_1 \mathbf{z}_3^\top \\ \mathbf{z}_2 \mathbf{z}_1^\top & \mathbf{z}_2 \mathbf{z}_2^\top & \mathbf{z}_2 \mathbf{z}_3^\top \\ \mathbf{z}_3 \mathbf{z}_1^\top & \mathbf{z}_3 \mathbf{z}_2^\top & \mathbf{z}_3 \mathbf{z}_3^\top \end{bmatrix}}_R - 2 \underbrace{\begin{bmatrix} \mathbf{x}_1 \mathbf{z}_1^\top & \mathbf{x}_1 \mathbf{z}_2^\top & \mathbf{x}_1 \mathbf{z}_3^\top \\ \mathbf{x}_2 \mathbf{z}_1^\top & \mathbf{x}_2 \mathbf{z}_2^\top & \mathbf{x}_2 \mathbf{z}_3^\top \\ \mathbf{x}_3 \mathbf{z}_1^\top & \mathbf{x}_3 \mathbf{z}_2^\top & \mathbf{x}_3 \mathbf{z}_3^\top \end{bmatrix}}_G$$

Therefore, we must compute the three matrices to compute  $D^2$  (and subsequently  $D$  by taking the square root of each element). Notice the horizontal and vertical repetition of terms in  $S$  and  $R$  respectively. This expression extends to any  $n$  and  $m$ . Let us recap these matrices:

- $D^2 \in \mathbb{R}^{n \times m}$  is the square Euclidean matrix, where  $D_{ij}^2 = (\mathbf{x}_i - \mathbf{z}_j)(\mathbf{x}_i - \mathbf{z}_j)^\top$ ; essentially, the entry at the  $i^{\text{th}}$  row and  $j^{\text{th}}$  column is the square of the Euclidean distance between  $\mathbf{x}_i$  and  $\mathbf{z}_j$ .
- $G \in \mathbb{R}^{n \times m}$  is the Gram matrix, where  $G_{ij} = \mathbf{x}_i \mathbf{z}_j^\top$
- $S \in \mathbb{R}^{n \times m}$ , where  $S_{ij} = \mathbf{x}_i \mathbf{x}_i^\top$  (note it is only a function of  $i$ ). Observe that  $S \neq \mathbf{X}\mathbf{X}^\top$ : the simplest reason is because  $S$  is of shape  $n \times m$  whereas  $\mathbf{X}\mathbf{X}^\top$  is of shape  $n \times n$ . More precisely,  $S_{ij} = \mathbf{x}_i \mathbf{x}_i^\top$  for all  $1 \leq i \leq n, 1 \leq j \leq m$  whereas  $[\mathbf{X}\mathbf{X}^\top]_{ij} = \mathbf{x}_i \mathbf{x}_j^\top$  for all  $1 \leq i \leq n, 1 \leq j \leq n$ .
- $R \in \mathbb{R}^{n \times m}$ , where  $R_{ij} = \mathbf{z}_j \mathbf{z}_j^\top$  (note it is only a function of  $j$ ). Again, observe that  $R \neq \mathbf{Z}\mathbf{Z}^\top$  for the same reason for why  $S \neq \mathbf{X}\mathbf{X}^\top$ .

Based on the four matrices define above, we can write  $D^2$

as a linear combination of  $G$ ,  $S$ , and  $R$ :

$$D^2 = S + R - 2G$$

**Now, let's work through a numerical example.**

**Compute  $D^2$  given two matrices:**

$$\mathbf{X} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad \mathbf{Z} = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

**When you think you know the answer, view below to see the solution.**

To begin, it may be helpful to first note  $D^2 \in \mathbb{R}^{n \times m}$  and given the provided matrices we can find that  $n = 2$  (number of columns or features),  $m = 3$  (number of rows or observations), and  $d = 2$  (the length of the row vectors). To start, we compute the  $G$  matrix:

$$G = \mathbf{XZ}^\top = \begin{bmatrix} 9 & 12 & 15 \\ 19 & 26 & 33 \end{bmatrix}$$

Then, we compute the two "norm" matrices  $S, R$ . Note that  $S$  and  $R$  are computed using the squared L2 norms of the vectors in  $\mathbf{X}$  and  $\mathbf{Z}$  respectively. We can first compute the squared norms of the individual vectors in  $\mathbf{X}, \mathbf{Z}$  and then construct the desired matrices  $S, R$ . The row vectors in  $\mathbf{X}$  have squared norms: 5, 25; the row vectors in  $\mathbf{Z}$  have squared norms: 17, 29, 45. We construct  $S, R$  by repeating the appropriate squared norms:

$$S = \begin{bmatrix} 5 & 5 & 5 \\ 25 & 25 & 25 \end{bmatrix} \quad R = \begin{bmatrix} 17 & 29 & 45 \\ 17 & 29 & 45 \end{bmatrix}$$

Using the formula, we know the squared Euclidean distances can be computed as:

$$D^2 = S + R - 2G = \begin{bmatrix} 4 & 10 & 20 \\ 4 & 2 & 4 \end{bmatrix}$$

You can verify that the distances are correct. Take for instance:

$$D_{11}^2 = D([1, 2], [1, 4]) = (1 - 1)^2 + (2 - 4)^2 = 0^2 + 2^2 = 4$$

or

$$D_{12}^2 = D([1, 2], [2, 5]) = (1 - 2)^2 + (2 - 5)^2 = 1^2 + 3^2 = 10$$

---

[Back to Table of Contents](#)

# Euclidean Distance Function Without Loops

---

[Back to Table of Contents](#)

## Ask the Expert: Laurens van der Maaten, Part Two



**Laurens van der Maaten** is a research scientist at Facebook AI Research in New York, working on machine learning and computer vision. He previously worked as an assistant professor at Delft University of Technology and as a post-doctoral researcher at UC San Diego. Dr. van der Maaten received his Ph.D. from Tilburg University in 2009.

Dr. van der Maaten is interested in a variety of topics in machine learning and computer vision. Currently, he works on embedding models, large-scale weakly supervised learning, visual reasoning, and cost-sensitive learning.

---

### Question 1

How does facial recognition work?

### Video Transcript

The basic idea of facial recognition is to try and extract some very small feature representation or template that describes properties of the face. So, for instance, a property that we know is very distinguishing between individuals is the ratio of the distance between the eyes and the length of the nose. If I would measure that for myself and I would measure it for you, it would be a very different value. And so what this template or what this feature representation for an individual face contains are features like that, are properties like that. Now, once you've extracted these features for a very large set of individuals who you would like to recognize, the problem becomes a nearest neighbor problem, right? You receive a new image of a face, you extract the same properties, and you ask the question, "What individual in my database has a face with most similar properties?" Which is essentially what a nearest neighbor model or nearest neighbor classifier is doing. So that's sort of how it works.

---

## Question 2

What is the purpose of facial recognition technology?

### Video Transcript

So there are a lot of applications of facial recognition. So, for instance, at Facebook, we use facial recognition to be able to organize your albums; like you can go to its app, see all photos of my friends, which is a useful thing to have. But we also use it, for instance, to make sure that people don't impersonate you on Facebook by making a fake profile that has your picture on it, because we can recognize that that picture is not this fake profile but it's you. We also use it to make Facebook more engaging for visually impaired users by being able to read out to visually impaired users what people are present in an image. And so, you know, inside a company like Facebook, there are a lot of nice applications of this technology, but you can also see facial recognition technology in other situations. So, for instance, in more and more countries, border control agencies are starting to roll out facial recognition as a way to make passport checks faster and more efficient. So there are a lot of applications of this technology in the real world.

---

## Question 3

Why is k-NN used for facial recognition?

### Video Transcript

Right; so in a facial recognition system, presuming you want to, let's say, recognize all the people who live in the United States, you would have 330 million individuals who you'd have to recognize, and you would only have a very small number of photos of each of these individuals. And so you're in a setting where you have a very large number of classes and you have a very small number of samples per class. And so this is very different from a lot of other machine learning problems where you will typically have a small number of classes and a large number of samples

per class. And so it's these properties that make an approach like nearest neighbor classification much more useful for facial recognition than for some other machine learning problems.

---

[Back to Table of Contents](#)



# Build a Facial Recognition System

---

[Back to Table of Contents](#)

## Module Wrap-up: Building a Facial Recognition System

You've now gained the necessary experience to start framing problems like a data scientist and the foundation to start implementing the code necessary to efficiently and accurately find answers. You also investigated your first machine learning algorithm, k-Nearest Neighbors, which is one of many important algorithms you as a data scientist will rely upon. Finally, you implemented k-NN to create a facial recognition system.

---

[Back to Table of Contents](#)

## Read: Thank You and Farewell



**Kilian Weinberger**  
Associate Professor  
Computing and Information Science  
Cornell University

Congratulations on completing "Problem-Solving with Machine Learning."

I hope that you now recognize the power and potential that machine learning offers. I hope the material covered here has met your expectations and prepared you to better meet the needs of your organization.

From all of us at Cornell University and eCornell, thank you for participating in this course.

Sincerely,

Kilian Weinberger

---

[Back to Table of Contents](#)

# Glossary

---

## Bayes optimal classifier

The optimal classifier if the data distribution  $P(Y|X)$  was known. The Bayes optimal classifier predicts the most likely label, given a feature vector  $x$ . In practice it cannot be used if  $P(Y|X)$  is not known; however, it can be useful as a lower bound on the error or if  $P(Y|X)$  is approximated.

## Binary classification

A type of classification that categorizes data instances into one of two groups. There are only two possible label values (e.g. +1, -1, or 0/1).

## Data augmentation

Data augmentation is a way to artificially increase the size of your training data by augmenting data instances through label-preserving heuristics. For example, natural images can often be flipped horizontally or rotated slightly without changing the class membership.

## Distance function

A distance function measures the dissimilarity between two input vectors according to some pre-specified metric. Examples are the Euclidean and Manhattan distance. A distance function is essential for the k-Nearest Neighbor classifier to retrieve the most similar training inputs for a given test point.

## Features

The relevant characteristics or attributes that we believe may be predictive of a data instance's class membership. For example, the features we might collect to identify fraudulent bank transactions might include dollar amount, type of transaction, country of origin, frequency, etc.

## High-dimensional data

Data is typically represented as vectors, where any single dimension contains a feature of a data point. The more attributes you collect, the more high-dimensional that vector gets. When we have data with very high-dimensional feature vectors — high-dimensional data — the k-Nearest Neighbor algorithm's performance may deteriorate due to the curse of dimensionality.



## Hypothesis (Function)

The function or hypothesis is commonly denoted as " $h$ " and represents the program that we learn from our training data. We apply this function to new data in order to make predictions during test time.

## k-Nearest Neighbors

A commonly used supervised learning algorithm that makes the assumption that similar points of data share similar labels. The algorithm predicts the label of a test point through a majority vote amongst its k-Nearest Neighbors within the training set.

## Labels

The label is what you want to infer about a data point. Your training data has labels so that you can train your function to predict the label of test points.

## Label preserving

A transformation is label preserving if it does not change the class membership of a given input.

## Loss function

A loss function gives the computer a clear objective, measuring how many mistakes the selected function makes. A lower loss is always better, and a loss of zero is perfect.

## Machine learning

The science of how to make computers learn from experience.

## Multiclass classification

A type of classification that sorts an element of data into more than two groups of labels (e.g., `class1="red"`, `class2="blue"`, `class3="yellow"`).

## NumPy

A Python library specialized for linear algebra operations. It is really fast and convenient.





## Regression

One of three categories for predicted labels,  $y$ , used when  $y$  is a real value; for example, the price of a house. (The other two categories are binary classification and multiclass classification.)

## Supervised learning

A type of machine learning in which a specific label  $y$  is predicted based on a specific data set of labeled features.

