# ***Goals***

* To learn about object-oriented programming concepts.
* To learn about machine learning.
* To implement the *perceptron algorithm*; the perceptron algorithm is a simple and beautiful example of machine learning (ML) in action.

# ***Before you begin***

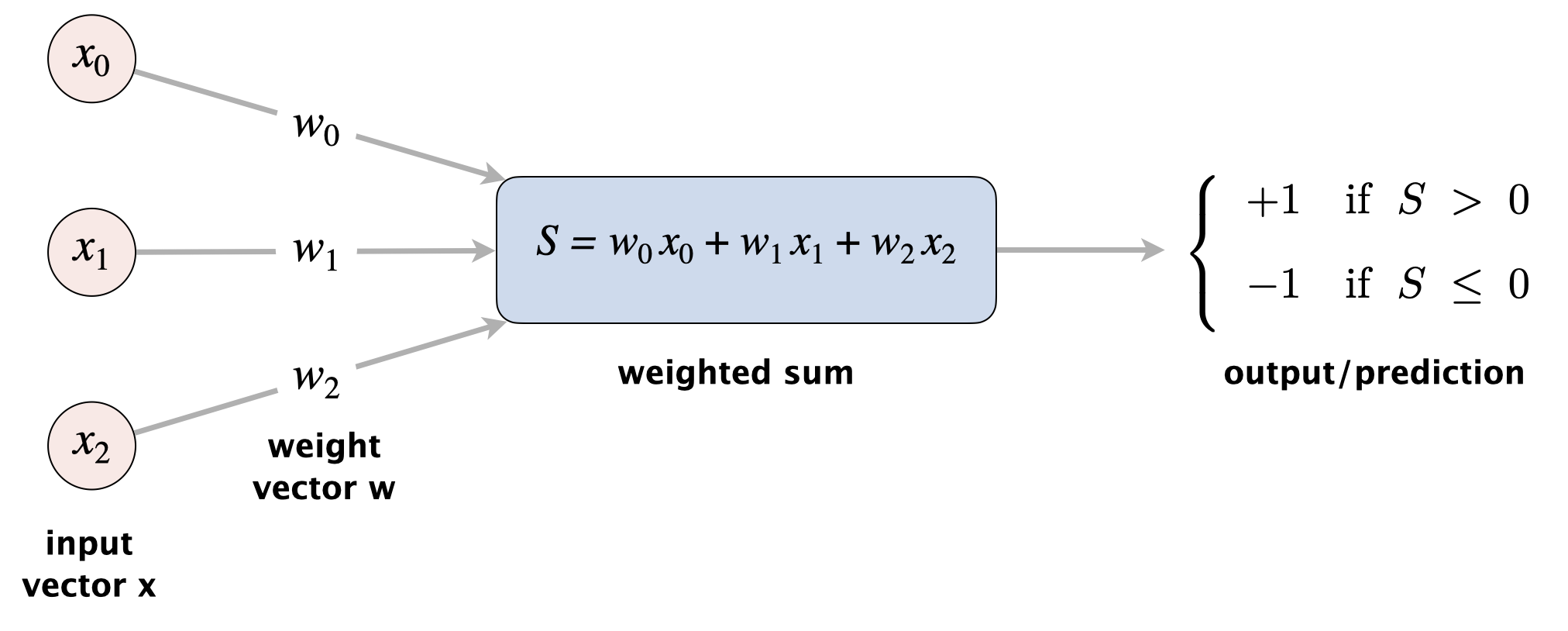
**Getting started.** Before you begin coding, do the following:

* Review Sections 3.1 and 3.2 of the textbook to learn the basics of object-oriented programming.

# ***Background***

**Perceptrons.** A *perceptron* is a simplified model of a biological neuron. It is a function that takes a vector of real numbers as input and outputs (or predicts) either +1 or −1. A perceptron is characterized by a vector of real numbers known as the *weight vector*. The perceptron computes the weighted sum

and outputs the *sign* of that number.



**Context.** A perceptron addresses the binary classification problem, in which we seek to classify input vectors into one of two classes. By convention, we use the labels +1 (positive) and −1 (negative) to denote the two classes. For the handwritten digit application, a perceptron will be trained to predict +1 for images that correspond to a target digit; and −1 otherwise. The input vector holds the grayscale values of each pixel in an image; the weight vector is pre-computed by a process described in the next paragraph.

# ***Approach***

**Perceptron algorithm.** How do we determine the values of the weight vector so that the perceptron makes accurate predictions? The core idea is to use the training data of known input–output pairs to incrementally refine the weights. Specifically, we initialize the weights to 0 and then process the labeled inputs one at a time. When we process a labeled input, there are three possibilities:

1. *Correct prediction*: the perceptron predicts the correct label (+1 or -1) for the given input vector . In this case, we leave unchanged.
2. *False positive:* the given input vector is labeled -1 but the perceptron predicts +1. In this case, we adjust as follows:

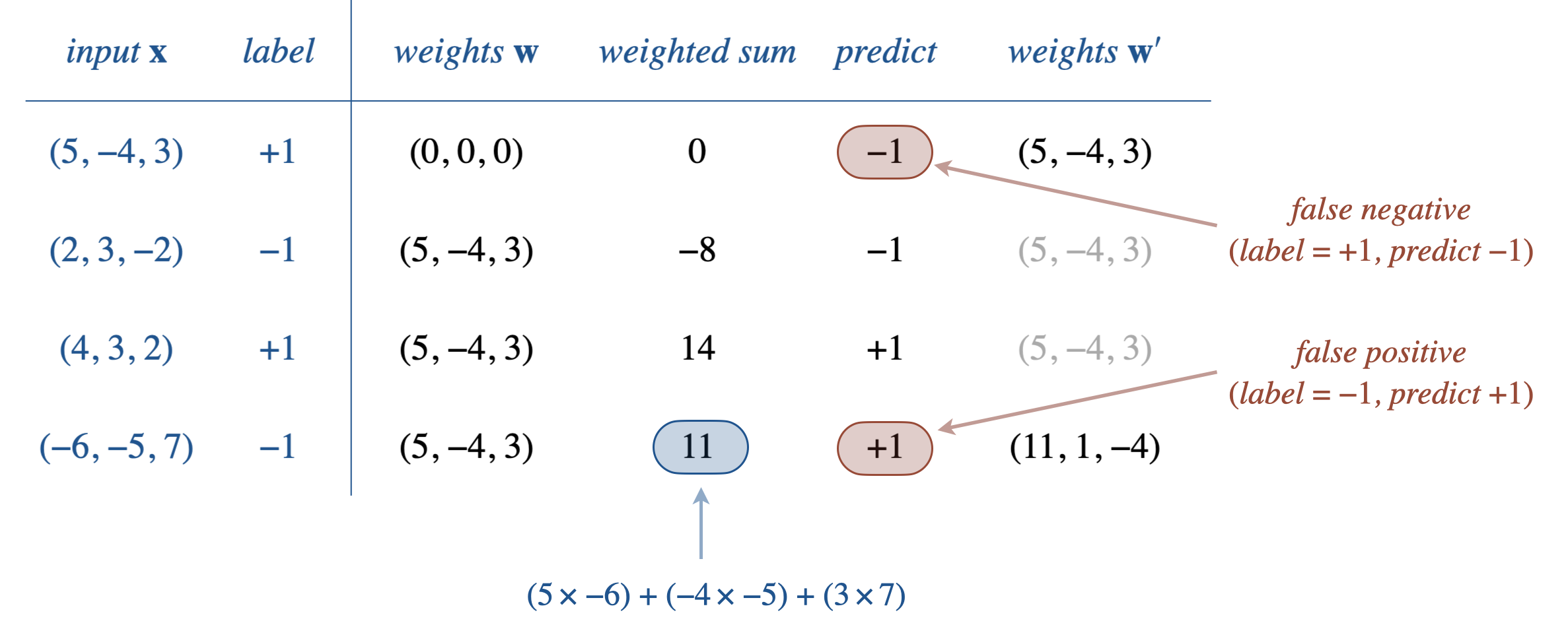
for each :

1. *False negative:* the given input vector is labeled +1 but the perceptron predicts -1. In this case, we adjust as follows:

for each :

The perceptron algorithm is one of the earliest and simplest examples of a *supervised learning* algorithm.

**Example.** Here is an example trace of the perceptron algorithm using 4 labeled inputs, each of length :



# 

# ***Implementation Tasks***

## **Overall Requirements**

* You must provide a comment describing the purpose of every method, using the names of the argument variables in your description.
* You should comment all instance variables.
* As usual, your code must conform to the specified APIs. If a method has a different signature or does not behave as specified, you will lose a substantial number of points. Producing undocumented side effects (such as printing to standard output in any of the constructor or instance methods) is also an API violation. You may not add public methods to the API; however, you may add *private* methods (which are accessible only in the class in which they are declared).

### **Perceptron data type.**

Create a data type that represents a perceptron by implementing the following API:

| **public class Perceptron {**  // Creates a perceptron with n inputs.  **public Perceptron(int n)**  // Returns the number of inputs n.  **public int numberOfInputs()**  // Returns the weighted sum of the weight vector and x.  **public double weightedSum(double[] x)**  // Predict the label (+1 or -1) of input x.  **public int predict(double[] x)**  // Trains this perceptron on the labeled (+1 or -1) input x.  **public void train(double[] x, int label)**  // Returns a string representation of this perceptron.  **public String toString()**  // Tests this class by directly calling all instance methods.  **public static void main(String[] args)**  **}** |
| --- |
|  |

We recommend representing a Perceptron using **two** instance variables: an integer n and an array of floating-point numbers weights[].

Here is some more information about the required behavior:

* **Perceptron(int n)***.* The constructor creates an array of n weights and initializes them to 0. Assume n >= 1.
* **double weightedSum(double[] x)***.* The weighted sum of the input vector **x** is

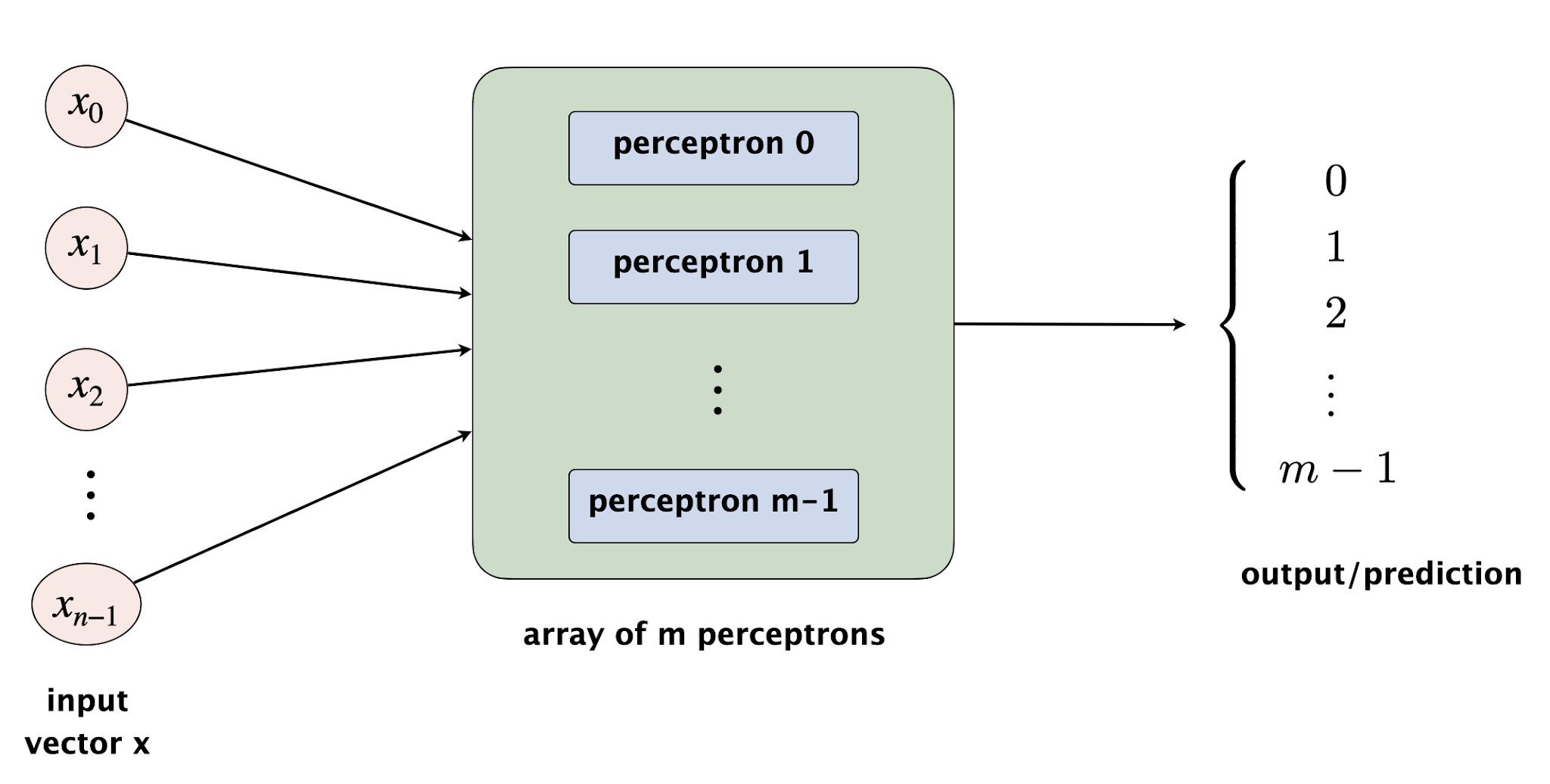
* **int predict(double[] x)***.* Returns +1 if the weighted sum is positive and −1 if it is negative (or zero).
* **void train(double[] x, int label)***.*  Checks whether the perceptron correctly classifies the labeled input **x** and updates the weight vector accordingly. Assume that the argument label is either +1 or −1
* **String toString()**. Returns a string representation of the weight vector, with the individual weights separated by commas and enclosed in parentheses.

| **(2.0, 1.0, -1.0, 5.0, 3.0)** |
| --- |

* *Corner cases.*  You may assume that the arguments to the constructor and instance methods are valid. For example, you may assume that the label is either +1 or −1 , the input vector x is of length n, and n >= 1.

### **Multiclass classification.**

In the previous section, we used a single perceptron for the binary classification problem. For a multiclass classification problem with classes, we create an array of perceptrons, each solving its own binary classification problem. For our handwritten digit application, each perceptron solves a binary classification problem of the form *does the image correspond to the digit 6*?



We train each perceptron independently and make predictions by distilling the results from the perceptrons.

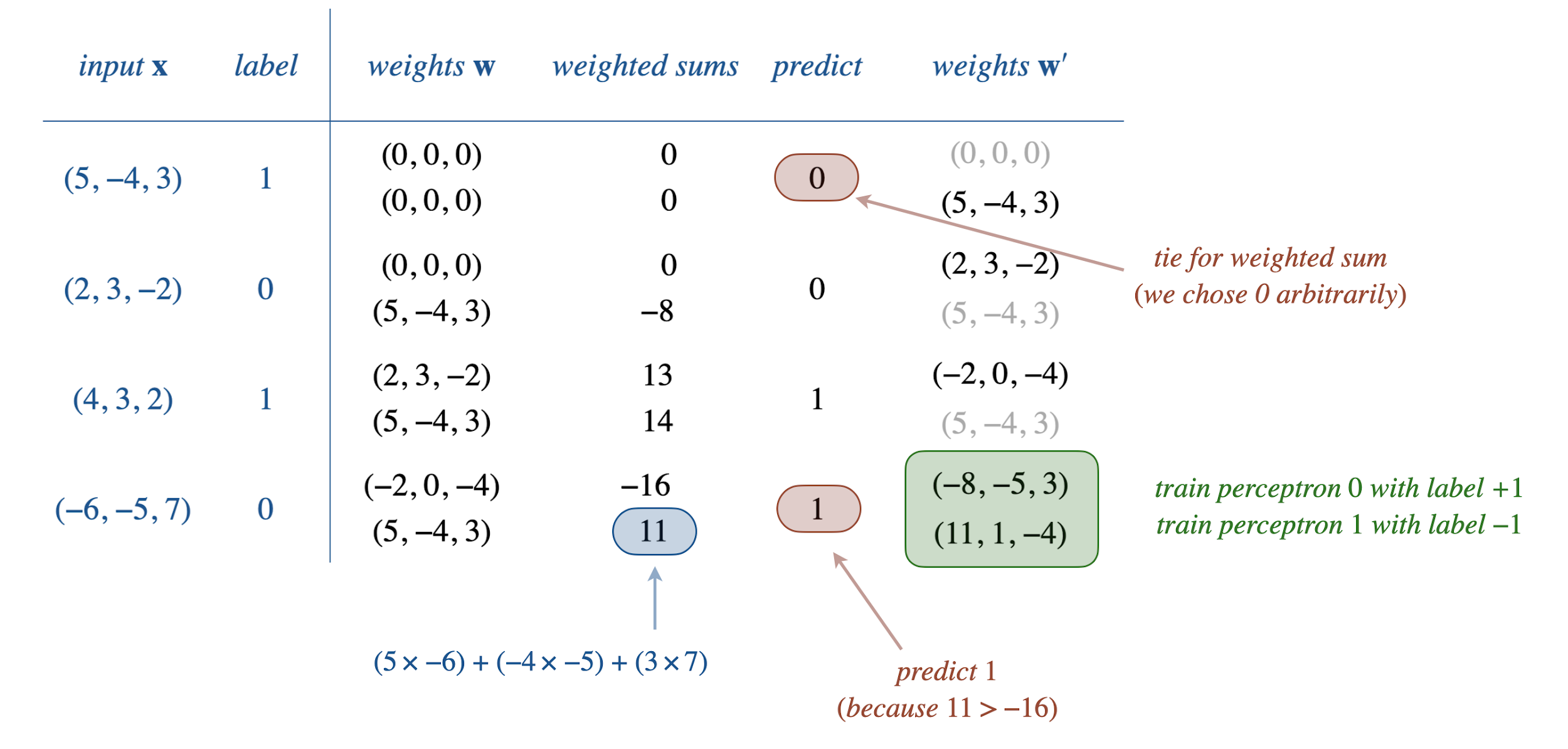
* *Multiclass training*. Initialize the weight vector of each of the perceptrons to be zero and process the labeled training inputs one at a time. To train the perceptrons on an input vector with multi-class label (0 to m - 1):
  + Train perceptron on input vector with the label = +1
  + Train the other perceptrons on input vector with the label = -1

That is, when training perceptron , we treat an input vector labeled as a positive example and an input vector with any other label as a negative example.

* *Multiclass prediction*. To make a prediction for an input vector , compute the weighted sum for each of the perceptrons on that input. The multiclass prediction is the index of the perceptron with the largest weighted sum. Intuitively, each perceptron with a positive weighted sum predicts that is a positive example for its class; the perceptron with the largest weighted sum makes that prediction with the most intensity, so we use that perceptron for the prediction.

This *one-vs-all* strategy decomposes a multiclass classification task into *m* binary classification tasks. In computer science, this decomposition is known as a *reduction*; this particular kind of reduction is used all over machine learning.

Example. Here is an example of a multi-perceptron with m = 2 classes and n = 3.



### **MultiPerceptron data type.**

Create a data type that represents a perceptron by implementing the following API:

| **public class MultiPerceptron {**  **Int m,**  **Int n**  **Perceptron[] perceptrons;**  // Creates a multi-perceptron object with m classes and n inputs.  **public MultiPerceptron(int m, int n)**  // Returns the number of classes m.  **public int numberOfClasses()**  // Returns the number of inputs n (length of the feature vector).  **public int numberOfInputs()**  // Returns the predicted label (between 0 and m-1) for the given input.  **public int predictMulti(double[] x)**  // Trains this multi-perceptron on the labeled (between 0 and m-1) input.  **public void trainMulti(double[] x, int label)**  **perceptrons[label].train(x, +1);**  **perceptrons[AllOtherClasses].train(x, -1);**  // Returns a string representation of this MultiPerceptron.  **public String toString()**  // Tests this class by directly calling all instance methods.  **public static void main(String[] args)**  **}** |
| --- |
|  |

We recommend representing a MultiPerceptron using **three** instance variables: an integer m (for the number of classes), an integer n (for the number of inputs), and an array of Perceptron objects perceptrons[].

Here is some more information about the required behavior:

* **MultiPerceptron(int m, int n)***.* The constructor creates an array of m perceptrons, each with n inputs and all weights initialized to 0. Assume m >= 1 and n >= 1.
* **String toString()***.*  Returns a string representation of the perceptrons, with the string representations of the perceptrons separated by commas and enclosed in parentheses. Here is an example with m = 4 and n = 3:

| **((-10.0, -8.0, 5.0), (3.0, 4.0, 5.0), (6.0, 4.0, -2.0), (1.0, 9.0, -10.0))** |
| --- |

* **int predictMulti(double[] x)***.*  If two (or more) perceptrons tie for the largest weighted sum in predictMulti(), return the index of *any* such perceptron (between 0 and m − 1).
* *Corner cases*. You may assume that the arguments to the constructor and instance methods are valid. For example, you may assume that the argument label is an integer between 0 and m − 1, x[] is an array of length n, and m >= 1 and n >= 1.

# ***Possible Progress Steps***

These are purely suggestions for how you might make progress. You do not have to follow these steps. If you get stumped or frustrated on some portion of the assignment, you should not hesitate to consult a preceptor.

**Perceptron.java:**

1. Implement the Perceptron constructor, Perceptron(int n), and the method, int numberOfInputs().
   1. Test: In the main() method, instantiate a few Perceptron objects and print the number of inputs for each object.
2. Implement the String toString() method.
   1. Test: In the main() method, print the various Perceptron objects. What should the output be for a newly instantiated Perceptron object?
3. Implement the double weightedSum(double[] x) method.
   1. Test: In the main() method, print the result of invoking the weightedSum method on the various Perceptron objects (using, of course, appropriately sized arrays.
4. Implement the int predict(double[] x) method.
5. Implement the void train(double[] x, int label) method. Note - train should call predict.
6. You can test by using the code in the ***Testing*** section (below) and submit to CodePost. Do not move onto MultiPerceptron until Perceptron is working properly.

**MultiPerceptron.java:**

1. Implement the MultiPerceptron constructor, MultiPerceptron(int m,int n), and the methods, int numberOfClasses() and int numberOfInputs().
   1. Test: In the main() method, instantiate a few MultiPerceptron objects and print the number of classes and inputs for each object.
2. Implement the String toString() method.
   1. Test: In the main() method, print the various MultiPerceptron objects. What should the output be for a newly instantiated MultiPerceptron object?
3. Implement the int predictMulti(double[] x) method.
4. Implement the void trainMulti(double[] x, int label) method.
5. You can test by using the code in the ***Testing*** section (below) and submit to CodePost.

# ***Testing***

**Perceptron.java:** Here is Java code that trains a **Perceptron** four input vectors (of length 3) from the assignment specification:

| double[] training1 = { 5.0, -4.0, 3.0 }; // yes  double[] training2 = { 2.0, 3.0, -2.0 }; // no  double[] training3 = { 4.0, 3.0, 2.0 }; // yes  double[] training4 = { -6.0, -5.0, 7.0 }; // no  int n = 3;  Perceptron perceptron = new Perceptron(n);  StdOut.println(perceptron);  perceptron.train(training1, +1);  StdOut.println(perceptron);  perceptron.train(training2, -1);  StdOut.println(perceptron);  perceptron.train(training3, +1);  StdOut.println(perceptron);  perceptron.train(training4, -1);  StdOut.println(perceptron); |
| --- |

And the desired output:

| (0.0, 0.0, 0.0)  (5.0, -4.0, 3.0)  (5.0, -4.0, 3.0)  (5.0, -4.0, 3.0)  (11.0, 1.0, -4.0) |
| --- |

**MultiPerceptron.java:** Here is Java code that trains a **MultiPerceptron** on four input vectors (of length 3) from the assignment specification:

| int m = 2;  int n = 3;  double[] training1 = { 5.0, -4.0, 3.0 };  double[] training2 = { 2.0, 3.0, -2.0 };  double[] training3 = { 4.0, 3.0, 2.0 };  double[] training4 = { -6.0, -5.0, 7.0 };  MultiPerceptron perceptron = new MultiPerceptron(m, n);  StdOut.println(perceptron);  perceptron.trainMulti(training1, 1);  StdOut.println(perceptron);  perceptron.trainMulti(training2, 0);  StdOut.println(perceptron);  perceptron.trainMulti(training3, 1);  StdOut.println(perceptron);  perceptron.trainMulti(training4, 0);  StdOut.println(perceptron); |
| --- |

And the desired output:

| ((0.0, 0.0, 0.0), (0.0, 0.0, 0.0))  ((0.0, 0.0, 0.0), (5.0, -4.0, 3.0))  ((2.0, 3.0, -2.0), (5.0, -4.0, 3.0))  ((-2.0, 0.0, -4.0), (5.0, -4.0, 3.0))  ((-8.0, -5.0, 3.0), (11.0, 1.0, -4.0)) |
| --- |

**Integration Testing:** Recall from Part 1 of this assignment that you implemented a client, ImageClassifier, of *our* implementation of MultiPerceptron. Now that you have implemented *your* version of MultiPerceptron and (Perceptron), you can test your client with your MultiPerceptron implementation.

Copy your ImageClassifier.java file from Part 1 (last week) to this assignment’s project folder. Compile and run ImageClassifier.java, which now uses your implementation of MultiPerceptron. Do you get the same error rates as reported in Part 1?

# ***Analysis***

Using your ImageClassifier (from Part 1), in addition to printing the misclassified image filename, also print the label associated with the filename and the incorrectly predicted digit, for example:

| jar:file:digits.jar!/testing/9/4560.png label: 9 predicted: 4 |
| --- |

Run the following experiment (you may wish to redirect the standard output to a file):

| % java-introcs ImageClassifier digits-training60K.txt digits-testing1K.txt |
| --- |

1. What digit is misclassified the most frequently?
2. For this digit, what are the top two digits that your MultiPerceptron incorrectly predicts?
3. Examine some of these misclassified images. Provide an explanation of what might have caused these misclassifications.

Provide your answers in your readme.txt file.

# ***Submission***

Submit Perceptron.java, MultiPerceptron.java, and a completed readme.txt file.

# ***Enrichment***

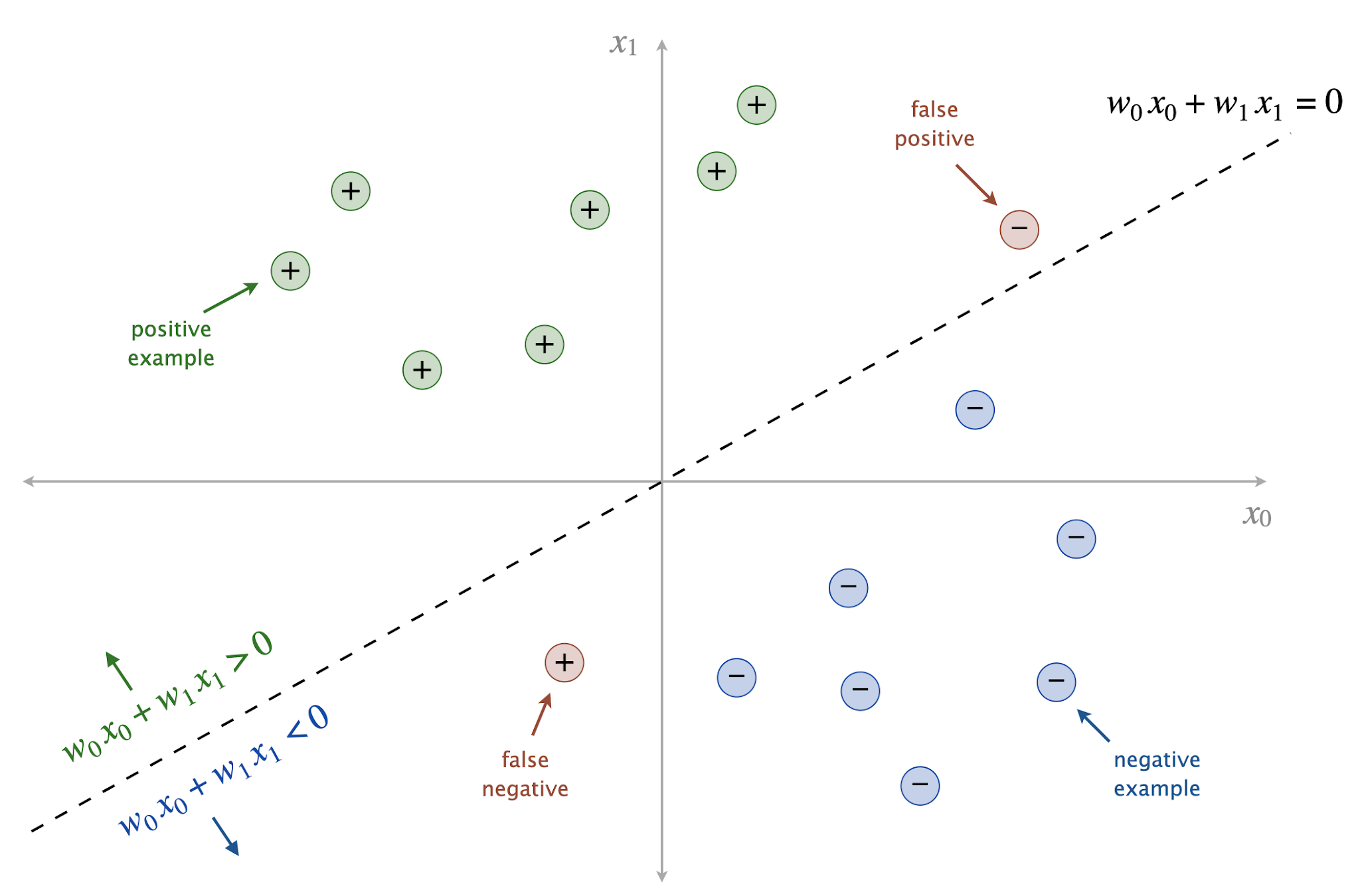
*When was the perceptron algorithm first discovered?* It was proposed in 1958 by Frank Rosenblatt, an American psychologist. It is one of the earliest machine-learning algorithms. At the time, researchers were overly optimistic about its short-term potential and grossly underestimated the formidable challenge of building intelligent machines.

*What is the significance of the perceptron algorithm?* It is a really simple, beautiful algorithm that, nevertheless, can do something interesting.

* The perceptron algorithm is one of the most fundamental algorithms in an area of ML called online learning (learning from samples one at a time).
* The perceptron algorithm is closely related to the support-vector machines algorithm, another fundamental ML algorithm.
* The perceptron algorithm has some beautiful theoretical properties. For example, if the training data set is linearly separable (i.e., there exists some weight vector that correctly classifies all of the training examples), and if we cycle through the training data repeatedly, the perceptron algorithm will eventually (and provably) find such a weight vector.
* Perceptrons are a building block in neural networks. In neural networks, many perceptrons (or other artificial neurons) are connected in a network architecture, in which the outputs of some neurons are used as the inputs to other neurons. Training multi-layer neural networks requires a more sophisticated algorithm to adjust the weights, such as gradient descent.

*Does the perceptron algorithm produce a different weight vector depending on the order in which it processes the training examples?* Yes. We've randomized the training data. It would be a bad idea, for example, to process all of the handwritten 0s before all of the handwritten 6s.

*Any intuition for why the perceptron algorithm works?* Geometrically, you can view each input vector as a *point* in and the weight vector as a *hyperplane* through the origin. The goal of the perceptron algorithm is to find a hyperplane that separates the positive examples from the negative examples. Using vector terminology, the weighted sum is known as the *dot product*; its sign determines on which side of the hyperplane the point lies.



During training, when we encounter a point that is on the wrong side of the hyperplane (i.e., a false positive or negative), we update the weight vector, thereby rotating the hyperplane slightly. After the rotation,is either on the correct side of the hyperplane or, if not, at least a bit closer to the hyperplane (in terms of Euclidean distance).

*How can I improve accuracy of the perceptron algorithm?* Here are three simple ideas:

* *Multiclass perceptron.* Instead of training all m perceptrons on each input vector, when there is a prediction error (multiclass perceptron predicts but correct label is , train only two perceptrons: train perceptron (with label −1) and perceptron (with label +1).
* Adjust the weights with a fraction of +1 or -1 for correct or incorrect predictions (this helps with a smoother convergence) and iterate over the training step multiple times, each time training the perceptron with the same set of training data (randomized in order).
* Normalize the Features array data to have values between 0 and 1 (divide the values with 255) and initialize the perceptron weights to random values (with uniform random or gaussian random to be less than 1 and on average 0).
* *Averaged perceptron.* Instead of using the last weight vector, take the average of the weight vectors that are computed along the way.
* *Incorporate more features.* Instead of using the feature vector create additional features. In particular, for each pair of features and , create a new feature . You could also keep going, adding not just pairs of features, but also triples, etc. This can significantly improve accuracy, but it becomes prohibitively expensive in terms of computation.

See this [paper](http://rob.schapire.net/papers/FreundSc98.pdf) for additional ideas, including the kernel trick and the voted-perceptron algorithm.

*I noticed that the weights are always integral throughout the perceptron algorithm. Why is this?* Adding or subtracting an integer to an integer yields an integer. The weights are always adjusted by either adding or subtracting the input vector; for the image classification problems we consider in this assignment, the input vector elements are integers between 0 and 255 (grayscale values).

*As a perceptron is trained with more and more data, the weights increase in absolute value. Is this to be expected?* Yes. This implies that the algorithm makes smaller and smaller relative changes to the weights as it learns from more and more input–output pairs.

*Why not use the testing data for training?* We seek a model that can make good predictions for unseen input–output pairs. If we use the testing data when training, the algorithm could “memorize” the input–output pairs. While this might achieve higher accuracy on those inputs, the results may not generalize to unseen inputs.

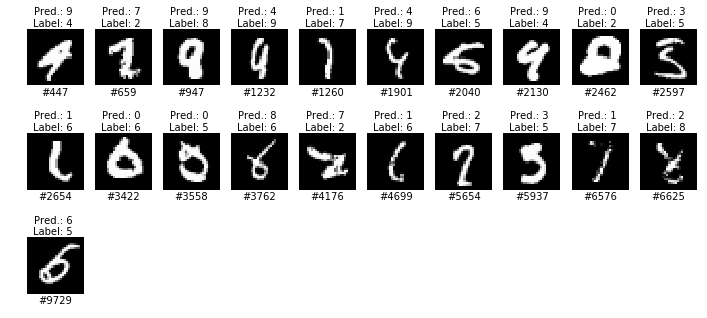
*What is the difference between supervised and unsupervised learning?* In supervised learning, the algorithm has access to a training data for which we know the correct labels. These labels act as a teacher supervising the learning process. In unsupervised learning, the training data is unlabeled, so there are no correct answers; instead, the goal may be to cluster similar inputs together.

*How to recognize my own handwritten digits?* Use the same technique but must be careful to size-normalize and center the images, as is done in the [MNIST training data](https://en.wikipedia.org/wiki/MNIST_database).

*Can ML algorithms be trained to classify alphabetic and mathematical symbols?* Yes. Here are web apps that let you draw and find the most similar Unicode characters or LaTeX symbols.

*Any games based on classifying hand-drawn images?* Yes. Try [Quick, Draw](https://quickdraw.withgoogle.com/), which uses neural networks and the world’s largest doodling data set.

*What is the best performing algorithm for classifying handwritten digits using the MNIST dataset?* The current champion uses convolution neural networks and achieves a 99.79% accuracy rate on the MNIST testing database consisting of 10,000 images. Here are the 21 incorrect predictions:



There is not much room for improvement; indeed, some of the errors appear to be due to incorrectly labeled (or ambiguous) inputs.

This assignment was developed by Robert DeLuca, Maia Ginsburg, Alan Kaplan, Kevin Jeon, and Kevin Wayne

Copyright © 2020, [Kevin Wayne](http://www.cs.princeton.edu/~wayne).