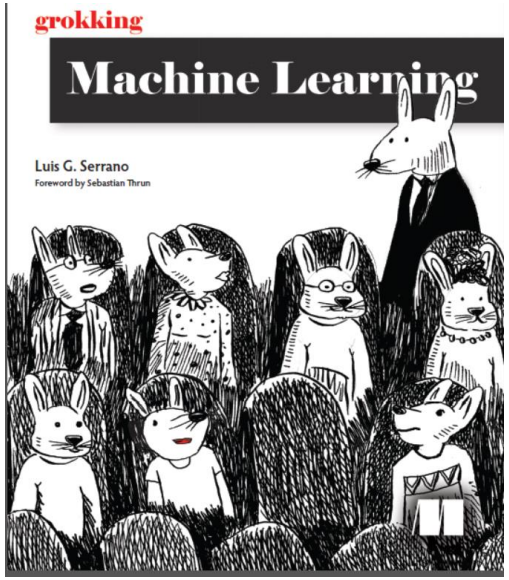


Machine Learning

| Perceptron

Prof. Dr. Mostafa Elhosseini

https://youtube.com/@ElhosseiniAcademy

Agenda

- Using lines to split our points

Classification

- Predict the labels of a dataset based on the features
- Predict a state or a category.
  - A recommendation model – watch a certain movie
  - An email model – (spam or ham)
  - A medical model – (sick or healthy)
  - An image-recognition model – (automobile , a bird, a cat, or a dog)
  - A voice recognition model – (particular command)

Perceptron

- A Perceptron is similar to a linear regression model, in that it uses a linear combination of the features to make a prediction
- The building block of neural networks
- The process of training a perceptron is similar to that of training a linear regression model

Sentiment analysis

- Predict the sentiment of a sentence.
- The model predicts whether the sentence is happy or sad.
- I feel wonderful! – is a happy sentence
- What an awful day! – is a sad sentence

Sentiment analysis applications

- When a company analyzes the conversations between customers and technical support, to evaluate the quality of the conversation
- Comments on social media or reviews related to its products
- When a social platform like Twitter analyzes the overall mood of a certain population after an event
- When an investor uses public sentiment toward a company to predict its stock price

How to Build Perceptron

- **Happy sentences** tend to contain **happy words**, such as wonderful, happy, or joy, whereas sad sentences tend to contain sad words, such as awful, sad, or despair

- A classifier can consist of a **“happiness” score** for every single word in the dictionary. Happy words can be given **positive scores**, and sad words can be given **negative scores**.

- Neutral words such as ‘’the’’ can be given a **score of zero**

How to Build Perceptron

- If the result is positive, then the classifier concludes that the sentence is happy.
- If the result is negative, then the classifier concludes that the sentence is sad.
- The goal now is to **find scores for all the words in the dictionary**. For this, we use machine learning.

Idea of Perceptron Classifier

- To train the model, we first need a **dataset** containing many sentences together with their **labels** (happy/sad).

- We start building our classifier by assigning **random scores** to all the **words**. Then we go over all the sentences in our dataset several times.

- For every sentence, we **slightly tweak the scores** so that the classifier **improves the prediction** for that sentence

Idea of Perceptron Classifier

- How do we tweak the scores?
  - We do it using a trick called the **perceptron trick**
  - Use an **error function**, then use **gradient descent** to minimize this function

Idea of Perceptron Classifier

- Wouldn't we **lose too much information** if we reduce a word to a simple score? The answer is **yes**
- We can still create a classifier that is **correct most of the time**
- The sentences, "I am not sad, I'm happy" and "I am not happy, I am sad" have the same words, yet completely different meanings
- A solution for this problem is to build a classifier that takes the order of the words into account
  - Hidden Markov models (HMM),
  - Recurrent neural networks (RNN), or
  - Long short-term memory networks (LSTM)

Example

- Aliens have two moods, happy and sad
- figure out if they are happy or sad based on what they say. In other words, we want to build a sentiment analysis classifier.
- Their language seems to only have two words: **aack** and **beep**

Example

- Alien 1: **Happy** Sentence: “Aack, aack, aack!”
- Alien 2: **Sad** Sentence: “Beep beep!”
- Alien 3: **Happy** Sentence: “Aack beep aack!”
- Alien 4: **Sad** Sentence: “Aack beep beep beep!”
- Aack beep aack aack ???

Example

- Aack beep aack aack ???
- We predict that this alien is happy because, even though we don't know the language, the word aack seems to appear more in happy sentences, whereas the word beep seems to appear more in sad sentences

Data

Aack aack aack!

Beep beep!

Aack beep aack!

Aack beep beep beep!

Prediction

Is this alien happy or sad?

Aack beep aack aack!

Example

- This observation gives rise to our **first sentiment analysis classifier**.
- This classifier makes a prediction in the following way: it **counts** the number of appearances of the **words aack and beep**.
- If the number of appearances of aack is larger than that of beep, then the classifier predicts that the sentence is happy. If it is smaller, then the classifier predicts that the sentence is sad.
- What happens when both words appear the same number of times? We have **no basis** to tell

Sentiment analysis classifier

Given a sentence, assign the following scores to the words:

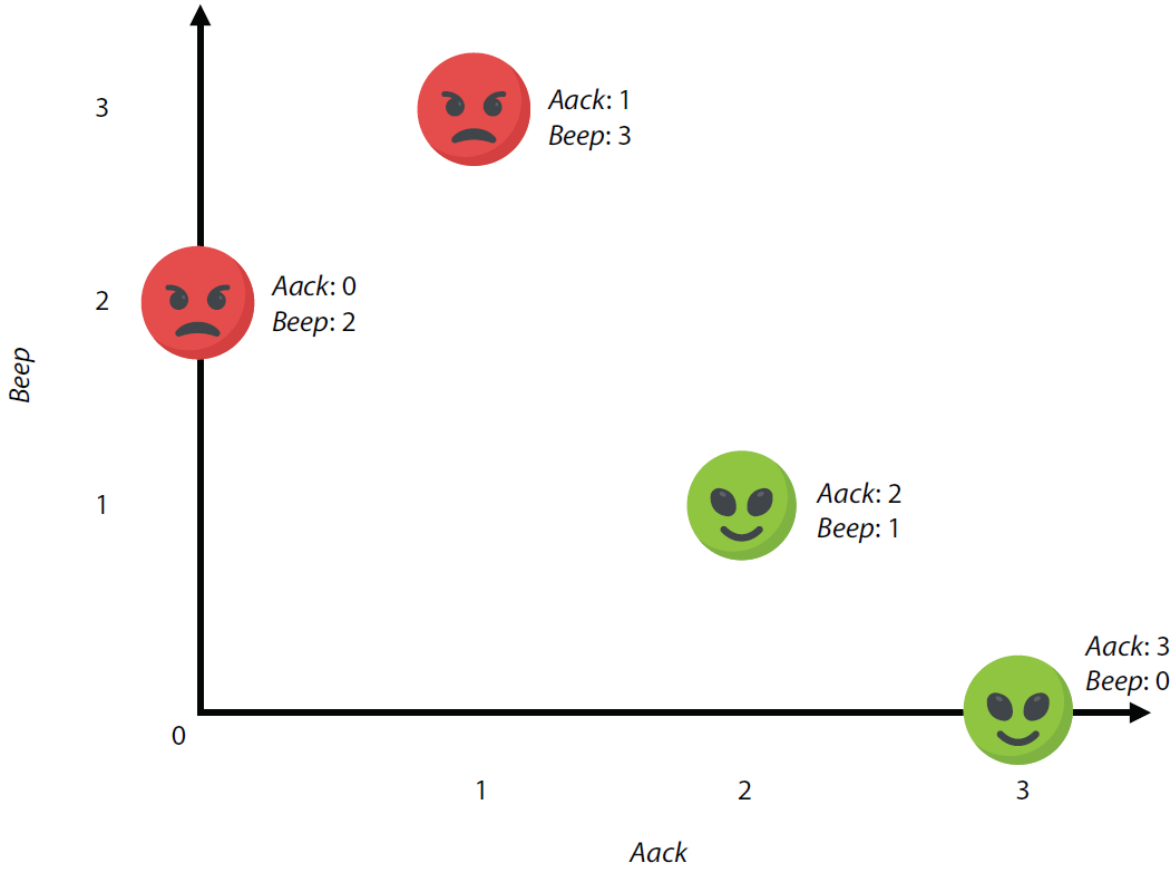
- Scores:
  - Aack: 1 point
  - Beep: –1 points
- Rule:

  Calculate the score of the sentence by adding the scores of all the words on it as follows:
  - If the score is positive or zero, predict that the sentence is happy.
  - If the score is negative, predict that the sentence is sad.

Sentiment analysis classifier

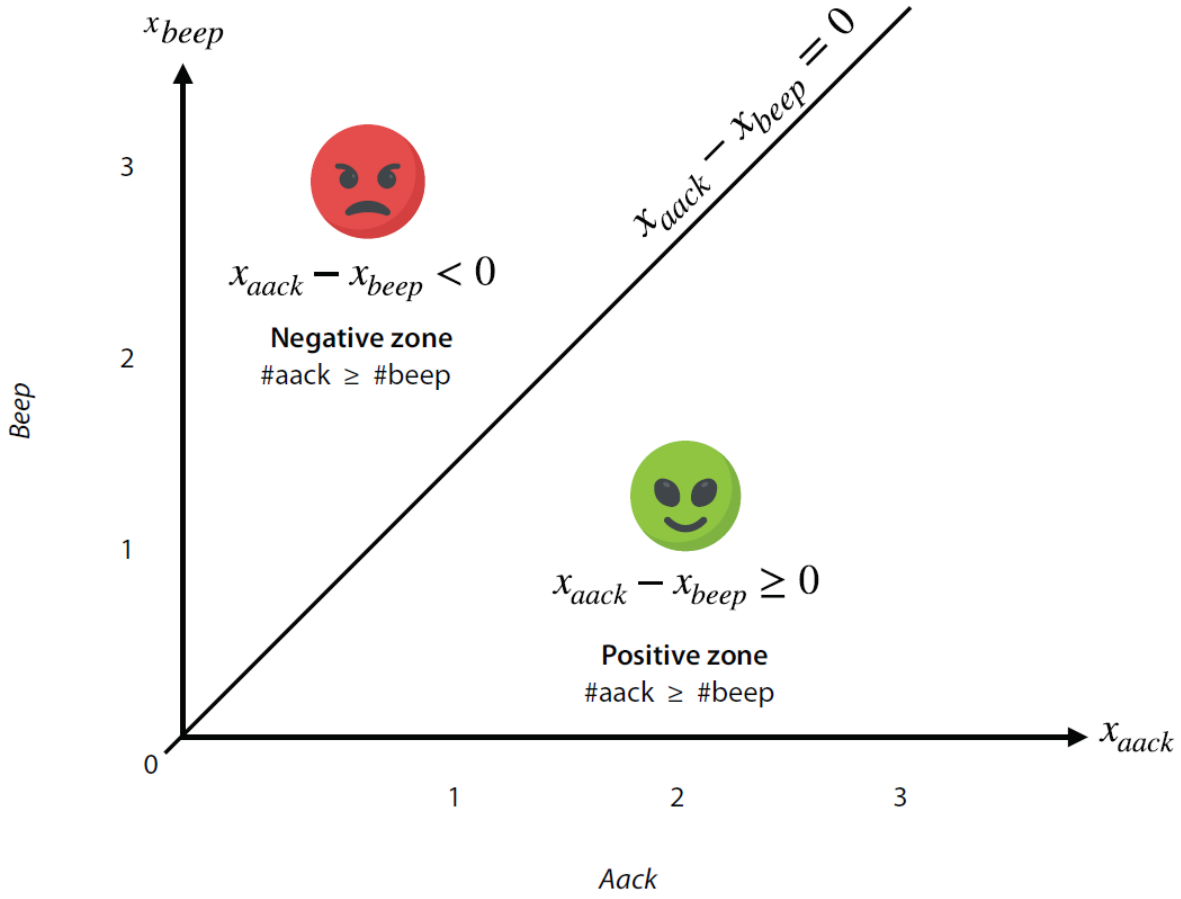
Sentence	*Aack*	*Beep*	Mood
Aack aack aack!	3	0	Happy
Beep beep!	0	2	Sad
Aack beep aack!	2	1	Happy
Aack beep beep beep!	1	3	Sad

Sentiment analysis classifier



- happy aliens are located on the bottom right, whereas the sad aliens are in the top left
- line formed by all the sentences with the same number of appearances of aack and beep divides these two regions
- #aack = #beep

Sentiment analysis classifier



- x_{aack} is the number of times the word aack
- appears, and x_{beep} is the number of times the word beep appears
- the equation of the classifier becomes $x_{aack} - x_{beep} = 0$
- **Positive zone**: The area on the plane for which $x_{aack} \geq xbeep$

Sentiment analysis classifier

- Our goal is to find the classifier that can put as many happy sentences as possible in the positive area and as many sad sentences as possible in the negative area

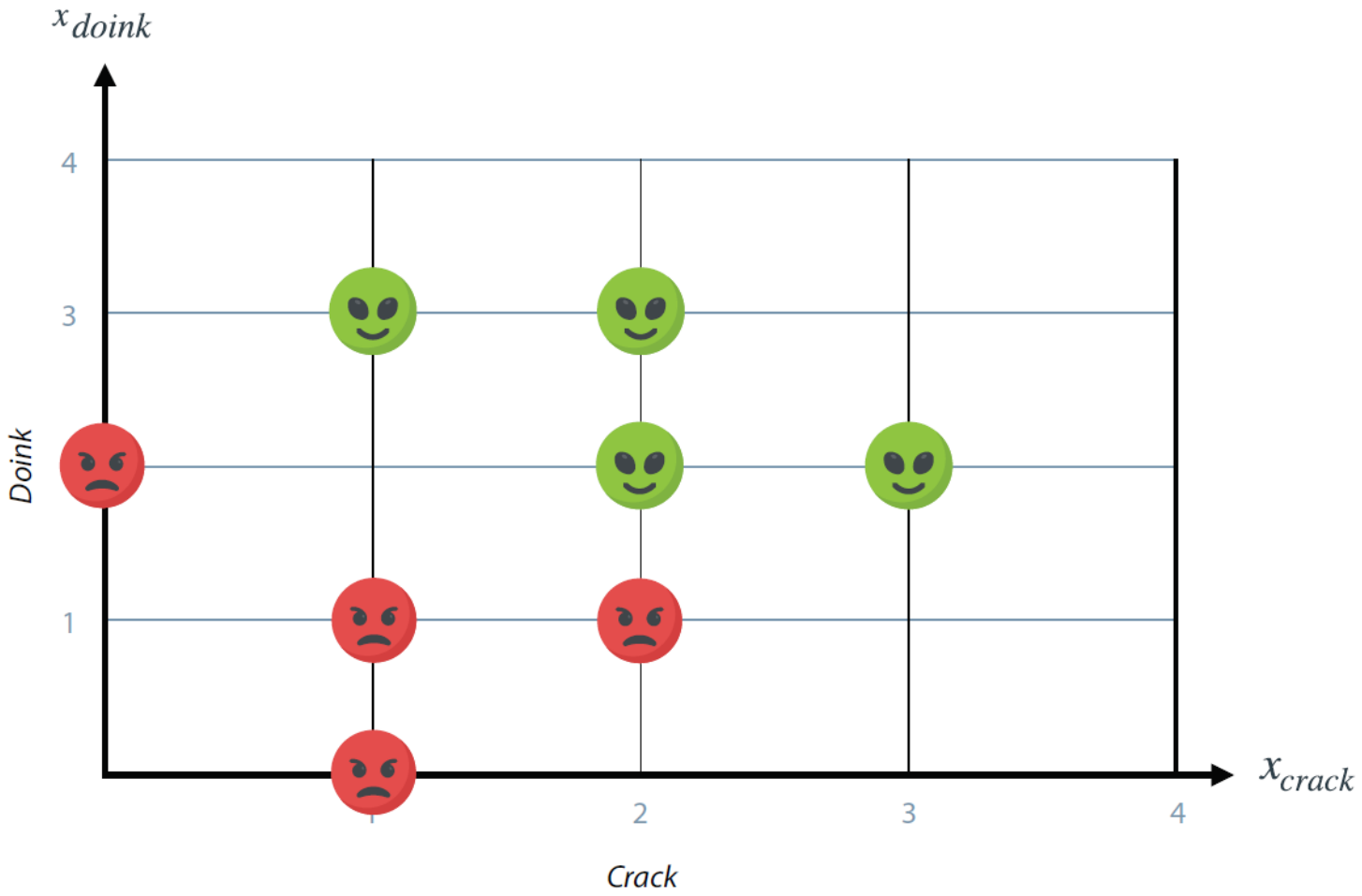
- For this small example, our classifier **achieves this job to perfection**. **This is not always the case**, but the perceptron algorithm will help us find a classifier that will perform this job really well.

Slightly more complex example

- In this section, we see a more complicated example, which introduces a new aspect of the perceptron: the **bias**
- The language in the new planet has two words: crack and doink

Sentence	*Crack*	*Doink*	Mood
Crack!	1	0	Sad
Doink doink!	0	2	Sad
Crack doink!	1	1	Sad
Crack doink crack!	2	1	Sad
Doink crack doink doink!	1	3	Happy
Crack doink doink crack!	2	2	Happy
Doink doink crack crack crack!	3	2	Happy
Crack doink doink crack doink!	2	3	Happy

Slightly more complex example



Slightly more complex example

- It classifies sentences with **three words or fewer as sad**, and the sentences with **four words or more as happy**
- Given a sentence, assign the following scores to the words:
- Scores:
  - Crack: one point
  - Doink: one point
- Rule:

  Calculate the score of the sentence by adding the scores of all the words on it.
  - If the score is four or more, predict that the sentence is happy.
  - If the score is three or less, predict that the sentence is sad.

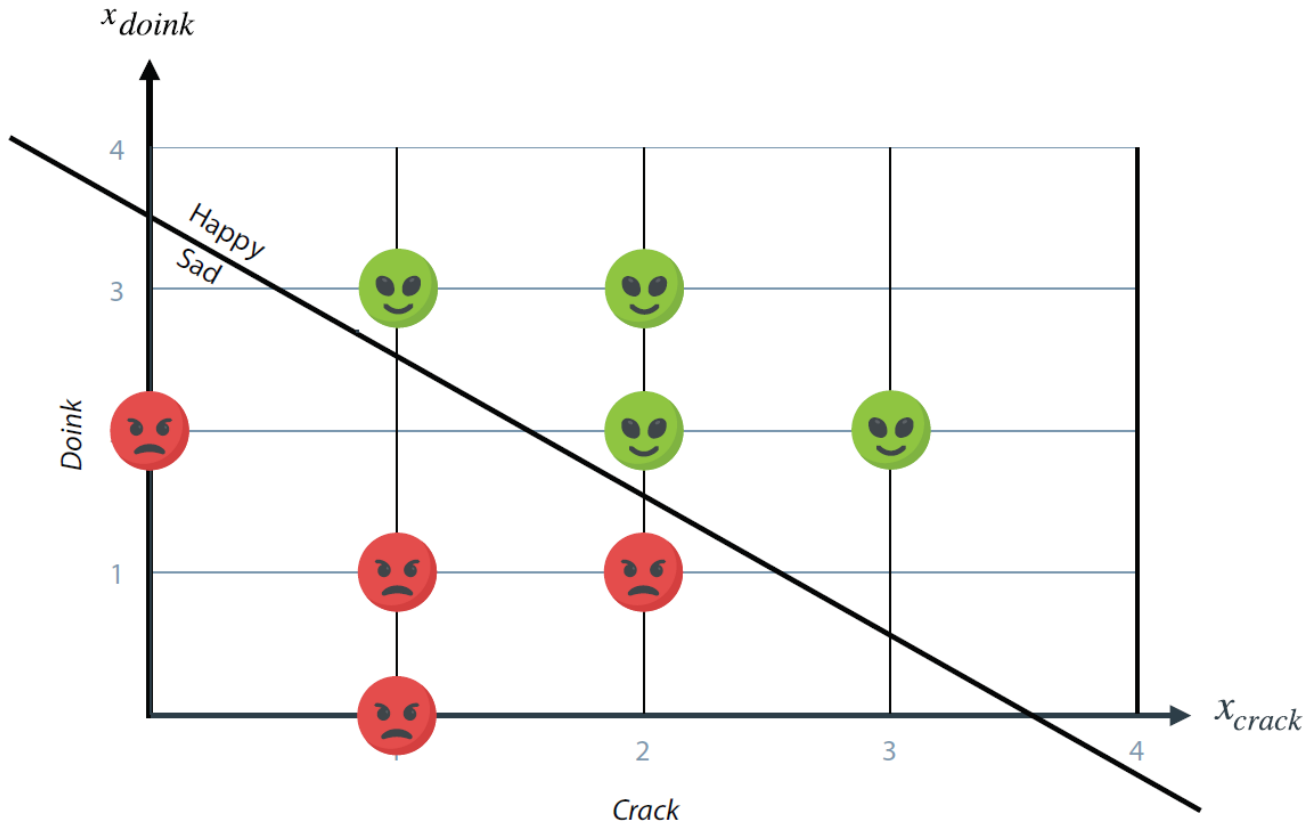
Slightly more complex example

- To make it simpler, let's slightly change the rule by using a **cutoff of 3.5**.
- Given a sentence, assign the following scores to the words:
- Scores:
  - Crack: one point
  - Doink: one point
- Rule:

  Calculate the score of the sentence by adding the scores of all the words on it.
  - If the score is 3.5 or more, predict that the sentence is happy.
  - If the score is less than 3.5, predict that the sentence is sad.

Slightly more complex example

- It seems that both words crack and doink are happy, because their scores are both positive
- Why, then, is the sentence “Crack doink” a sad sentence?
- The aliens that don’t speak much are sad, and those who speak a lot are happy



Bias

- Cutoff, or threshold
- Sentences with scores **higher than** or equal to the **threshold** are classified as **happy**, and sentences with scores lower than the threshold are classified as sad.
- However, thresholds are not common, and instead we use the notion of a bias
- The **bias** is the **negative of the threshold**, and we add it to the score

Bias

Given a sentence, assign the following weights and bias to the words:

- Weights:
  - Crack: one point
  - Doink: one point
- Bias: –3.5 points
- Rule:

  Calculate the score of the sentence by adding the weights of all the words on it and the bias.
  - If the score is greater than or equal to zero, predict that the sentence is happy.
  - If the score is less than zero, predict that the sentence is sad.

Bias

- The equation of the score of the classifier $\#crack + \#doink - 3.5 = 0$

- $\#crack + \#doink \geq 3.5$ Threshold
- $\#crack + \#doink - 3.5 \geq 0$ Bias

- $x_{crack} + x_{doink} - 3.5 = 0.$
- $x_{crack} + x_{doink} - 3.5 \geq 0$ Positive
- $x_{crack} + x_{doink} - 3.5 < 0$ Negative

Classifier quality

- Does our classifier need to be correct all the time? No
- The goal of the classifier is to classify the points as best as possible
- Impossible to perfectly split into two using a single line.
- The line in the picture does a good job, only classifying three points incorrectly.

General classifier equation

- let's call our words 1 and 2, and the variables keeping track of their appearances x_1 and x_2
- $x_1 - x_2 = 0$
- $x_1 + x_2 - 3.5 = 0$
- $ax_1 + bx_2 + c = 0$
  - a: Score of the word 1,
  - b: the score of the word 2, and
  - c: is the bias

General classifier equation

- $ax_1 + bx_2 + c = 0$
- Positive zone: $ax_1 + bx_2 + c \geq 0$
- Negative zone: $ax_1 + bx_2 + c < 0$

- if the word 1 has a score of 4, the word 2 has a score of –2.5, and the bias is 1.8, then the equation of this classifier is

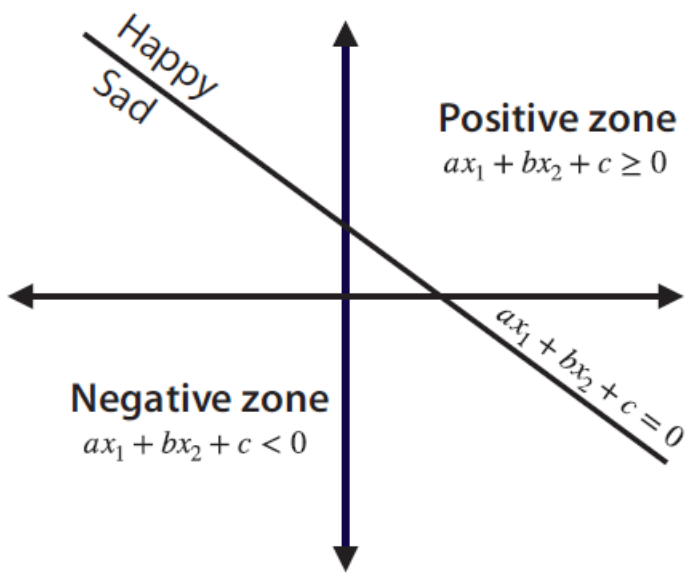
$$4x_1 - 2.5x_2 + 1.8 = 0$$

Equations of lines and zones in the plane

- Linear Regression: $y = mx + b$
- Perceptron Model, Logistic Regression: $ax_1 + bx_2 + c = 0$
- Why is this equation better for perceptron models?
  - $ax_1 + bx_2 + c = 0$: Defines a line and the two zones
  - $-ax_1 - bx_2 - c = 0$: The positive and negative regions flipped
  - We can draw **vertical** lines $1x_1 + 0x_2 + c = 0$, $x_1 = c$

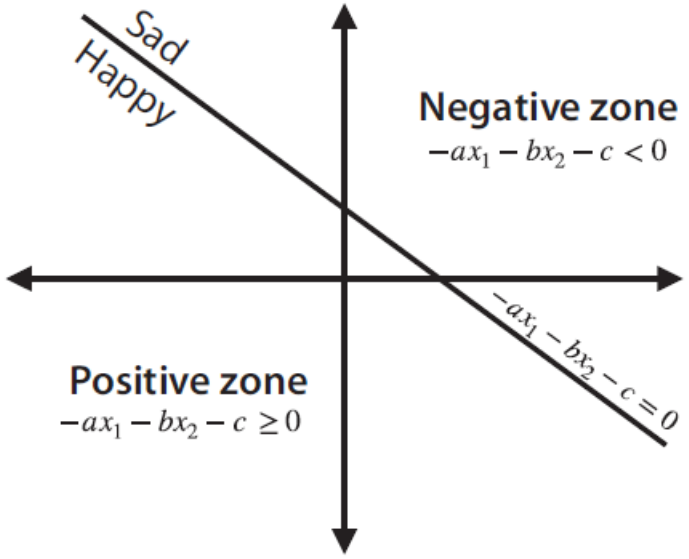
- For the line to be vertical, the x_1 coordinate must be constant, which implies that x_2 can take any value
- Vertical lines show up in classification models

Equations of lines and zones in the plane



Classifier with equation

$$ax_1 + bx_2 + c = 0$$



Classifier with equation

$$-ax_1 - bx_2 - c = 0$$

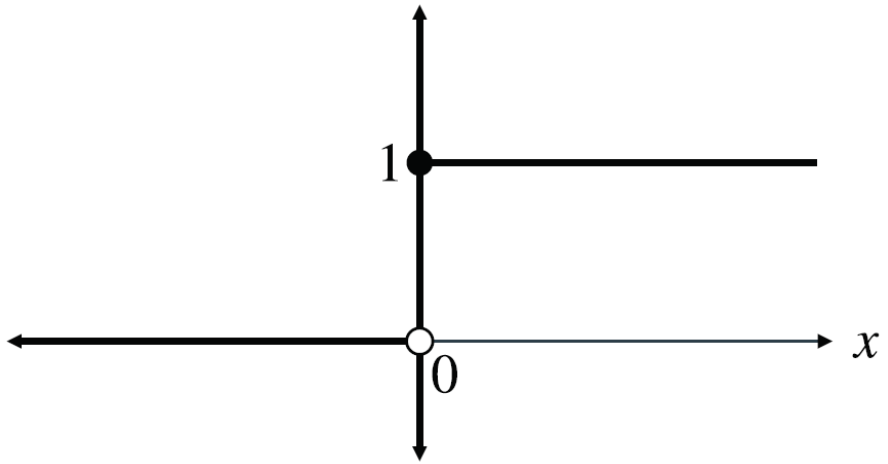
Step function

- The classifier predicts “happy” or “sad” based on the total score of the sentence; **if** this score is positive or zero, the classifier predicts “happy,” and **if** it is negative, the classifier predicts “sad.”

- We have a **more direct way** to turn the score into a prediction: using the **step function**

Step function

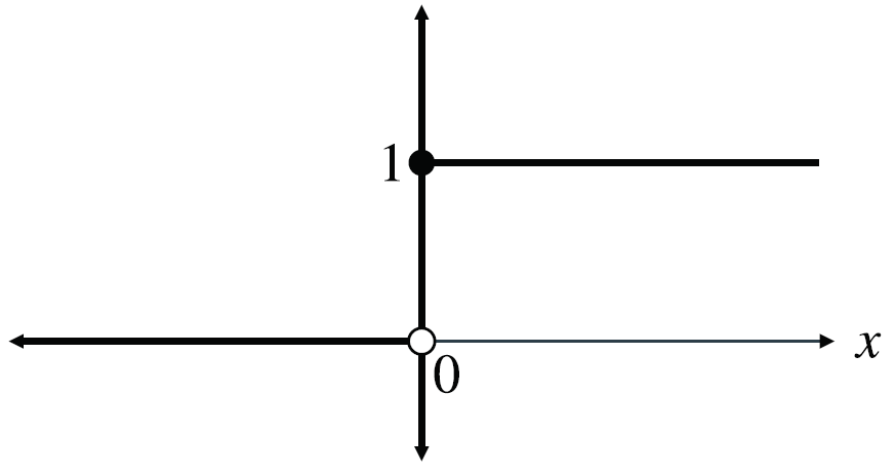
- The function that returns a 1 if the output is nonnegative and a 0 if the output is negative.
- $step(x) = 1 \; if \; x \geq 0$
- $step(x) = 0 \; if \; x < 0$



- $\hat{y} = step(ax_1 + bx_2 + c)$
- The step function is a specific case of an activation function
- Think of the activation function as a function we can use to turn the scores into a prediction.

Step function

- $\hat{y} = step(ax_1 + bx_2 + c)$
- The step function is a specific case of an activation function
- Think of the activation function as a function we can use to turn the scores into a prediction.
- The activation function is an important concept in machine learning, especially in deep learning



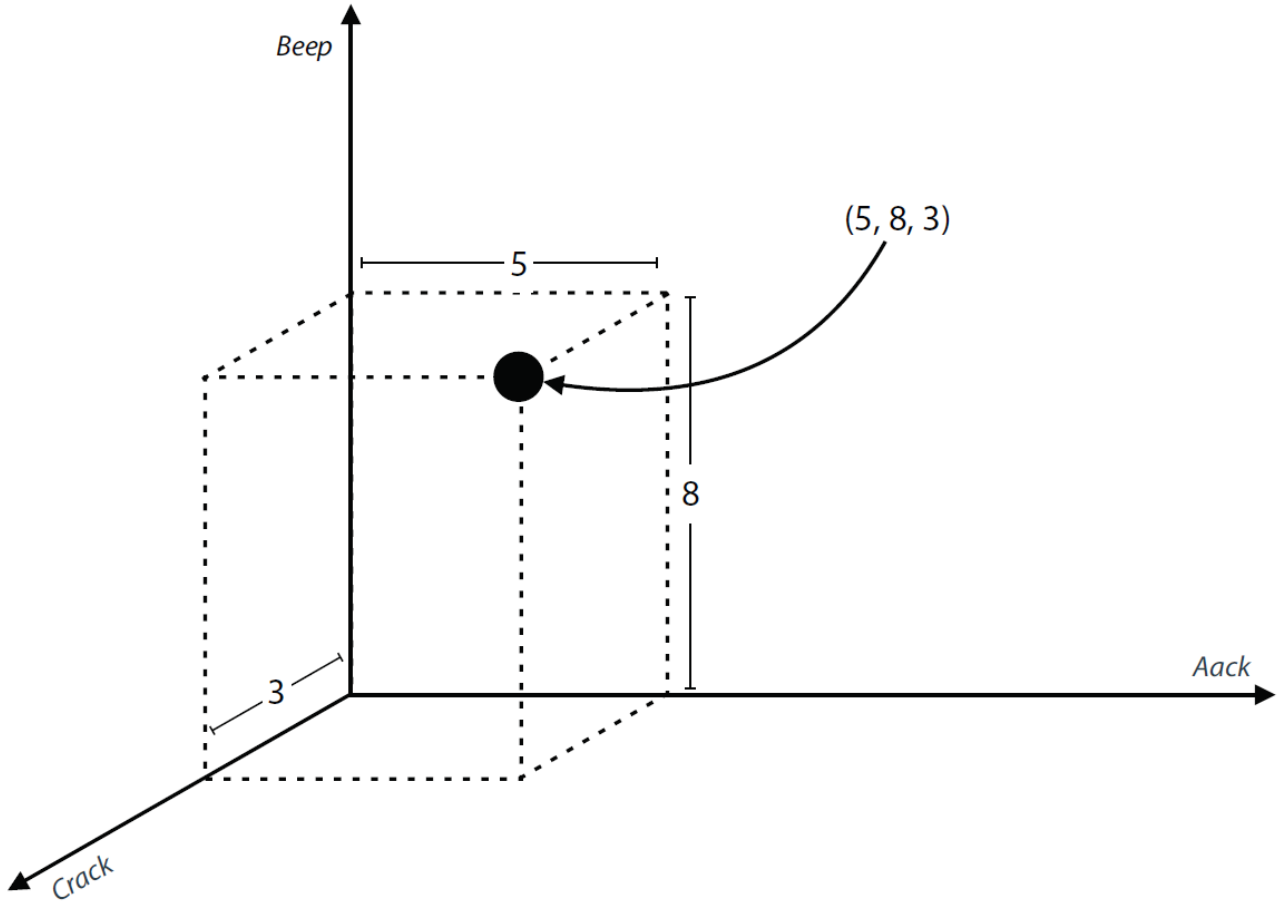
General definition of the perceptron classifier

- What happens if I have more than two words?
- if we had a language with three words, say, aack, beep, and crack
- $\hat{y} = step(ax_{aack} + bx_{beep} + bx_{crack} + d)$

- where a, b, and c are the weights of the words aack, beep, and crack, respectively, and d is the bias

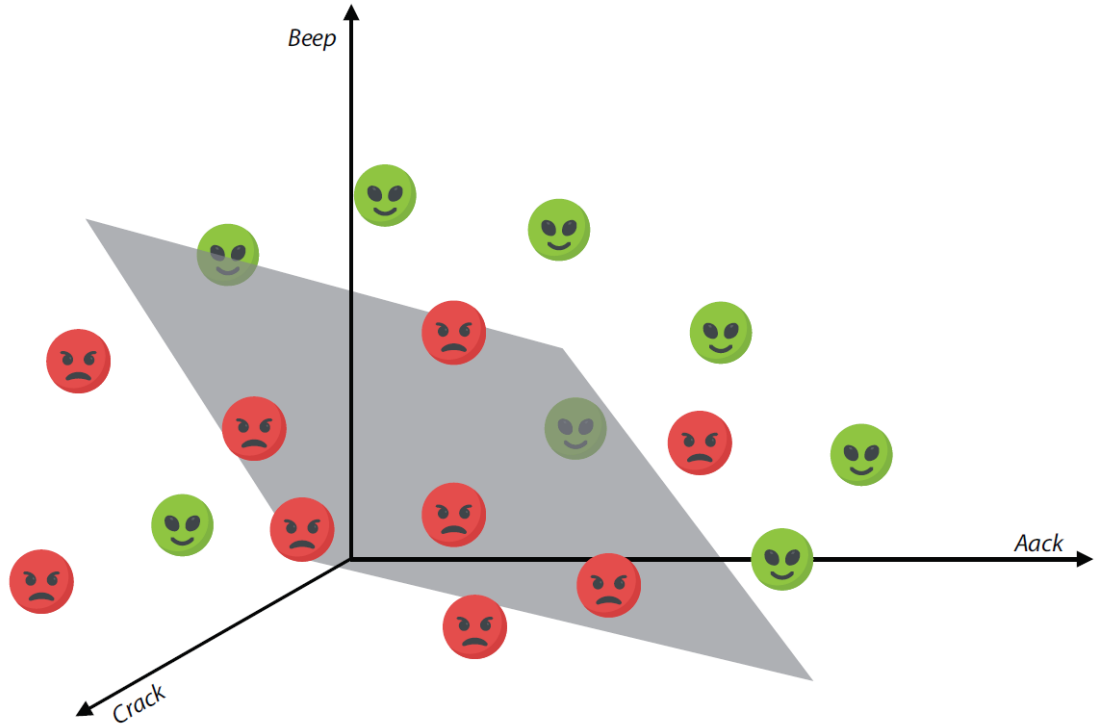
General definition of the perceptron classifier

- The sentence containing aack five times, beep eight times, and crack three times, corresponds to the point with coordinates (5, 8, 3).



General definition of the perceptron classifier

- The way to separate these points is using a plane $ax_{aack} + bx_{beep} + bx_{crack} + d$



General definition of the perceptron classifier

- Our language has n words, that we call $1, 2, \dots, n$. Our dataset consists of m sentences, which we call $x^{(1)}, x^{(2)}, \dots, x^{(m)}$
- Each sentence $x^{(i)}$ comes with a label y_i, which is 1 if the sentence is happy and 0 if it is sad
- Each sentence corresponds to a row in our dataset and can be seen as a vector, or an n-tuple of numbers $x^{(i)} = \left(x_1^{(i)}, x_2^{(i)}, \dots, x_n^{(i)}\right)$ where $x_j^{(i)}$ is the number of appearances of the word j in the $i-th$ sentence.

General definition of the perceptron classifier

- The perceptron classifier consists of n weights (scores), one for each of the n words in our language, and a bias.

- The weights are denoted w_i and the bias b. Thus, the prediction that the classifier makes for the sentence $x^{(i)}$ is

$$\hat{y}_i = step(w_1 x_1^{(i)} + w_2 x_2^{(i)} + \cdots + w_n x_n^{(i)} + b$$

General definition of the perceptron classifier

- The classifiers with two words can be represented geometrically as a line that cuts the plane into two regions, and the classifiers with three words can be represented as a plane that cuts the three-dimensional space into two regions.
- Classifiers with n words can also be represented geometrically. Unfortunately, we need n-dimensions to see them.
- Humans can see only three dimensions, so we may have to imagine an (n-1)-dimensional plane (called a hyperplane) cutting the n-dimensional space into two regions.
- However, the fact that we can't imagine them geometrically doesn't mean we can't have a good idea of how they work.

General definition of the perceptron classifier

- Imagine if our classifier is built on the English language. Every single word gets a weight assigned. That is equivalent to going through the dictionary and assigning a happiness score to each of the words
- **Weights (scores):**
  - A: 0.1 points
  - Aardvark: 0.2 points
  - Aargh: –4 points
  - …
  - Joy: 10 points
  - …
  - Suffering: –8.5 points
  - …
  - Zygote: 0.4 points
- **Bias:**
  - –2.3 points

General definition of the perceptron classifier

- To predict whether a sentence is happy or sad, we add the scores of all the words on it (with repetitions). If the result is higher than or equal to 2.3 (the negative of the bias), the sentence is predicted as happy; otherwise, it is predicted as sad.
- Furthermore, this notation works for any example, not only sentiment analysis. If we have a different problem with different data points, features, and labels, we can encode it using the same variables. For example, if we have a medical application where we are trying to predict whether a patient is sick or healthy using n weights and a bias, we can still call the labels y, the features x_i, the weights w_i, and the bias b.

Bias

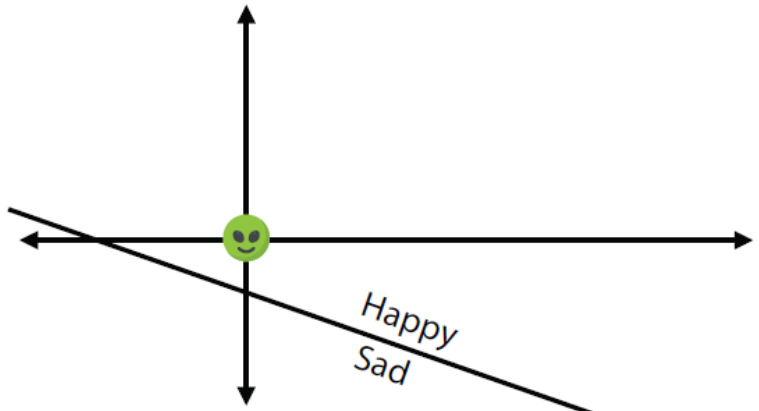
- Words with positive weights are happy, and words with negative words are sad
- In the perceptron model, the bias can be interpreted as the score of the empty sentence
- if an alien says absolutely nothing, is this alien happy or sad? If a sentence has no words, its score is precisely the bias
- Thus, if the bias is positive, the alien that says nothing is happy, and if the bias is negative, that same alien is sad.

Bias

- Geometrically, the difference between a positive and negative bias lies in the location of the origin (the point with coordinates (0,0)) with respect to the classifier.
- This is because the point with coordinates (0,0) corresponds to the sentence with no words.
- In classifiers with a positive bias, the origin lies in the positive zone, whereas in classifiers with a negative bias, the origin lies in the negative zone

Positive Bias

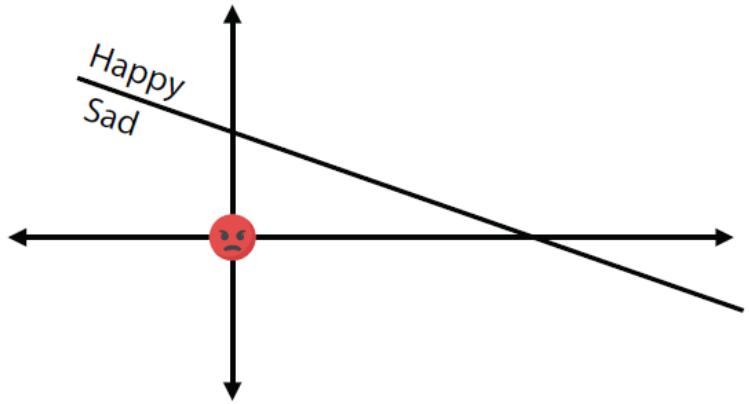
- A dataset of online reviews of a product
- Bad reviews tend to contain lots of words, because the customer is upset, and they describe their negative experience. However, many of the positive reviews are empty



Positive bias:
Quiet alien is happy

Negative Bias

- dataset of conversations with friends
- our friend says absolutely nothing, we imagine that they are mad at us or that they are very upset. Therefore, the empty sentence is classified as sad



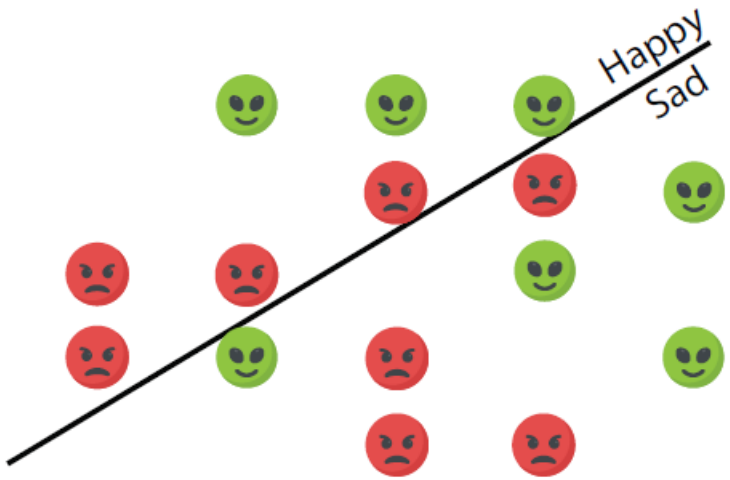
Negative bias:
Quiet alien is sad

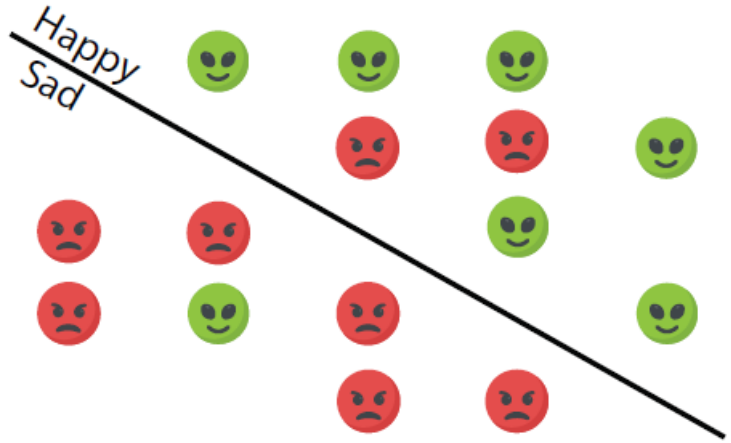
The error function

- How do we determine whether a classifier is good or bad?
- How do we find the perceptron classifier that best fits our data?
- How to evaluate the perceptron classifier
- Error function tell us whether a perceptron classifier fits our data well

The error function

- The one on the left is a bad classifier, and the one on the right is good

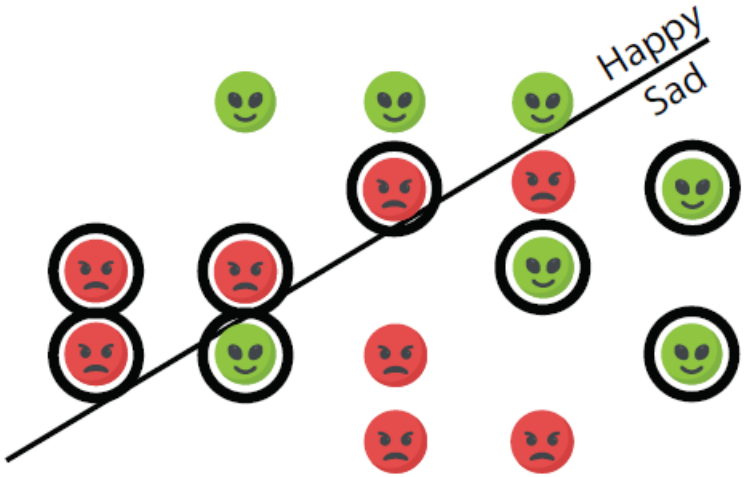




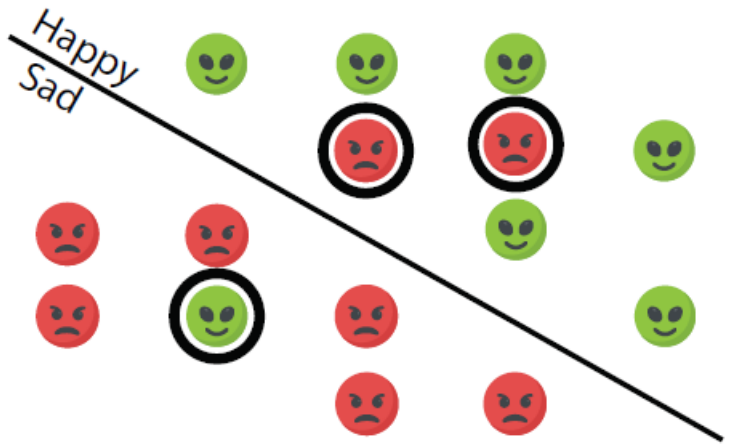
- Can we come up with a measure of how good they are?
- Can we assign a number to each one of them, in a way that the one on the left is assigned a high number and the one on the right a low number?

Error function 1: Number of errors

- Counting the number of mistakes, it makes
- Counting the number of points that it classifies incorrectly



Bad classifier
Error: 8



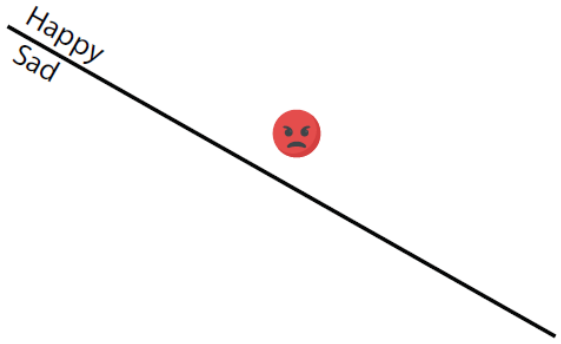
Good classifier
Error: 3

Error function 1: Number of errors

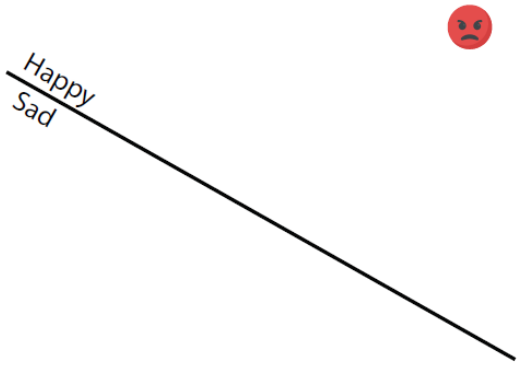
- This is a good error function, but it's not a great error function. Why?
- It tells us when there is an error, but it doesn't measure the gravity of the error
- if a sentence is **sad**, and the classifier gave it a **score of 1**, the classifier made a mistake. However, if **another classifier** gave it a score of **100**, this classifier made a much bigger mistake

Error function 1: Number of errors

- both classifiers misclassified a sad point by predicting that it is happy.
- However, the classifier on the left located the line close to the point, which means that the sad point is not too far from the sad zone. The classifier on the right, in contrast, has located the point very far from its sad zone.



Poorly classified



Very poorly classified

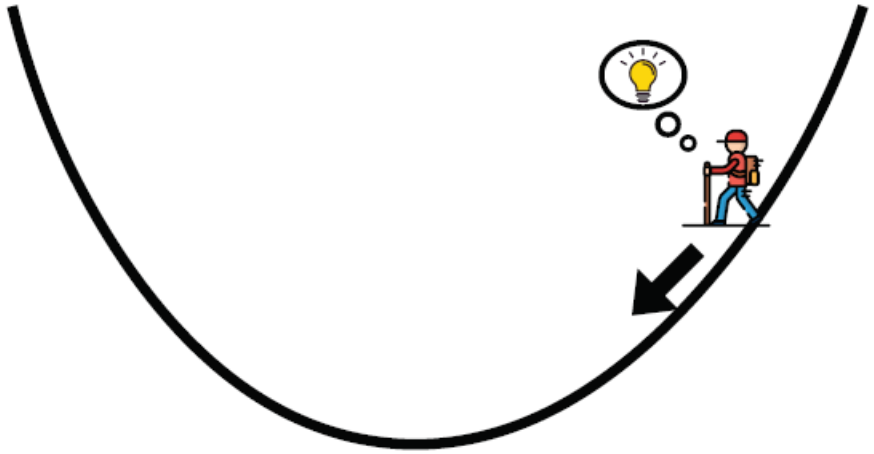
Error function 1: Number of errors

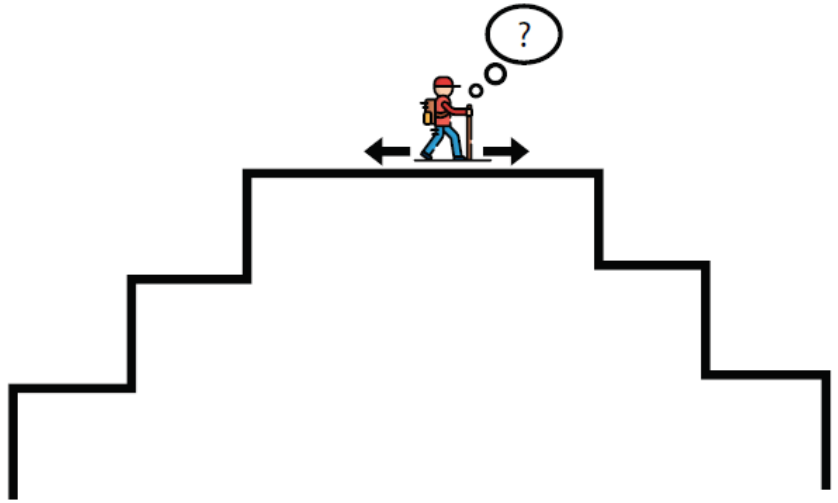
- Why do we care about measuring how bad an error is?
- The way to reduce an error is by decreasing it in small amounts, until we reach a point where the error is small

- If our error is calculated by counting the number of misclassified points, then this error will take only integer values. If we wiggle the line a small amount, the error **may not decrease at all**, and we **don't know** in which **direction** to move.

Error function 1: Number of errors

- The goal of gradient descent is to minimize a function by taking small steps in the direction in which the function decreases the most. If the function takes only integer values, this is equivalent to trying to descend from an Aztec staircase. When we are at a flat step, we don't know what step to take, because the function doesn't decrease in any direction



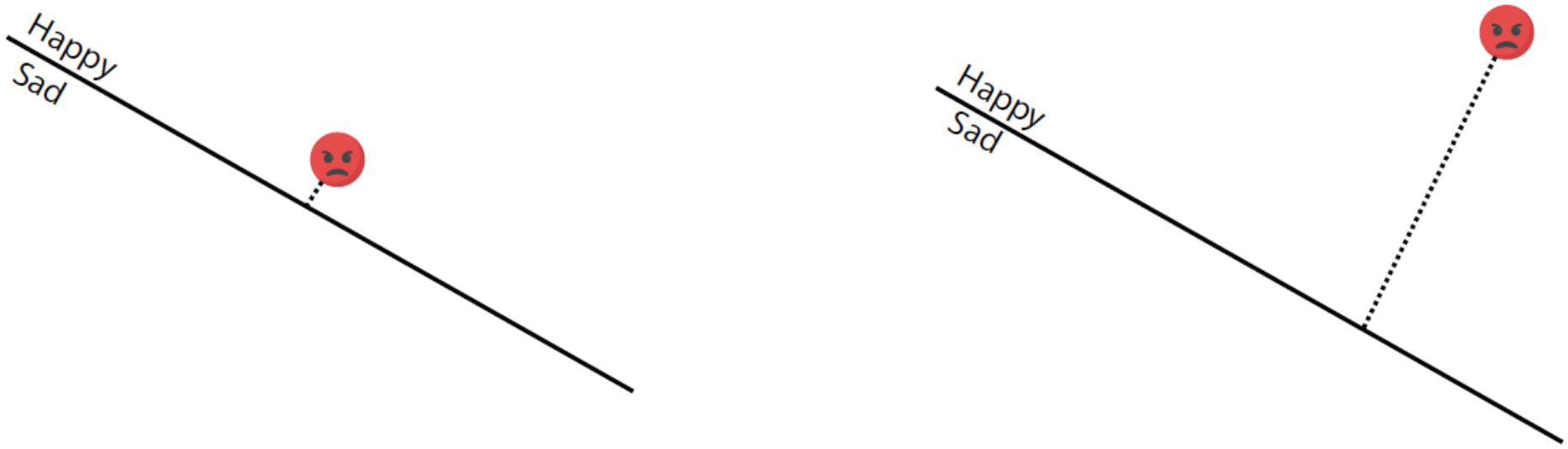


Error function 1: Number of errors

- We need a function that measures the magnitude of an error and that assigns a higher error to misclassified points that are far from the boundary than to those that are close to it.

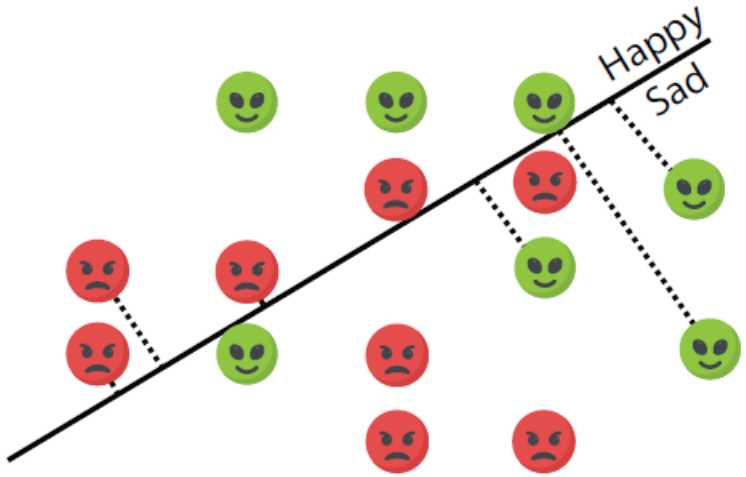
Error function 2: Distance

- Considering the perpendicular distance from the point to the line
- Points that are correctly classified produce an error of 0.
- Points that are misclassified produce an error equal to the distance from that point to the line.



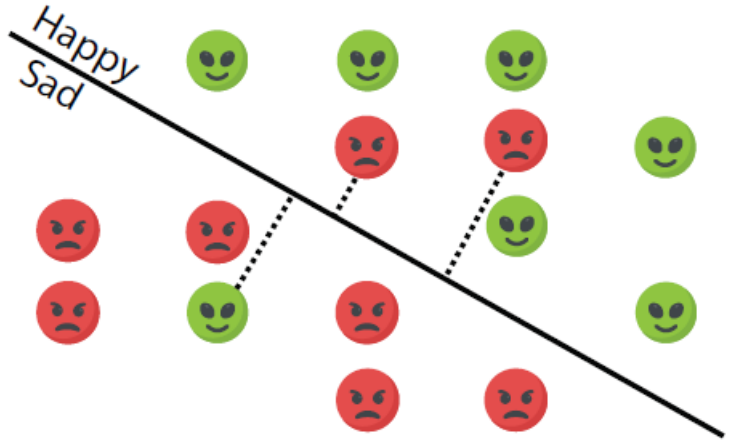
Error function 2: Distance

- Look only at the misclassified points and add the perpendicular distances from these points to the line



Bad classifier

Error: .....



Good classifier

Error: .....

Error function 2: Distance

- This is almost the error function we will use. Why don't we use this one?
- Because the distance from a point to a line is a **complicated formula**.
- The distance from a point (x_1, y_1) to a line given by the equation $ax + by + c = 0$ can be calculated using the following formula:

$$\frac{|ax_1 + by_1 + c|}{\sqrt{a^2 + b^2}}$$

- It contains a **square root**, because we calculate it using the Pythagorean theorem. Square roots have **complicated derivatives**, which adds unnecessary complexity the moment we apply the gradient descent algorithm

Error function 3: Score

let’s summarize the properties we want in an error function

- The error function of a correctly classified point is 0.

- The error function of an incorrectly classified point is a positive number.
  - For misclassified points close to the boundary, the error function is small.
  - For misclassified points far from the boundary, the error function is large.

- It is given by a simple formula.

Error function 3: Score

- The concept of **scoring a point** with respect to a **decision boundary** simplifies the calculation of the model's error function
- The points in the boundary have a score of 0.
- The points in the positive zone have positive scores.
- The points in the negative zone have negative scores.
- The points close to the boundary have scores of low magnitude (i.e., positive or negative scores of low absolute value).
- The points far from the boundary have scores of high magnitude (i.e., positive or negative scores of high absolute value).

Error function 3: Score

- Thus, We define the error as the absolute value of the score for misclassified points.

Perceptron error for a point (general)

- If the point is correctly classified, the error is 0.

- If the point is misclassified, the error is $|w_1x_1 + w_2x_2 + \cdots + w_nx_n + b|$

Mean perceptron error: Entire dataset

- Take the **average** of all the errors corresponding to all the points.

- We can also **take the sum** if we choose, although in this chapter we choose the **average** and call it the **mean perceptron error**.

Example

- Consider the dataset made of four sentences, two labeled happy and two labeled sad

Sentence	*Aack*	*Beep*	Label (mood)
Aack	1	0	Sad
Beep	0	1	Happy
Aack beep beep beep	1	3	Happy
Aack beep beep aack aack	3	2	Sad

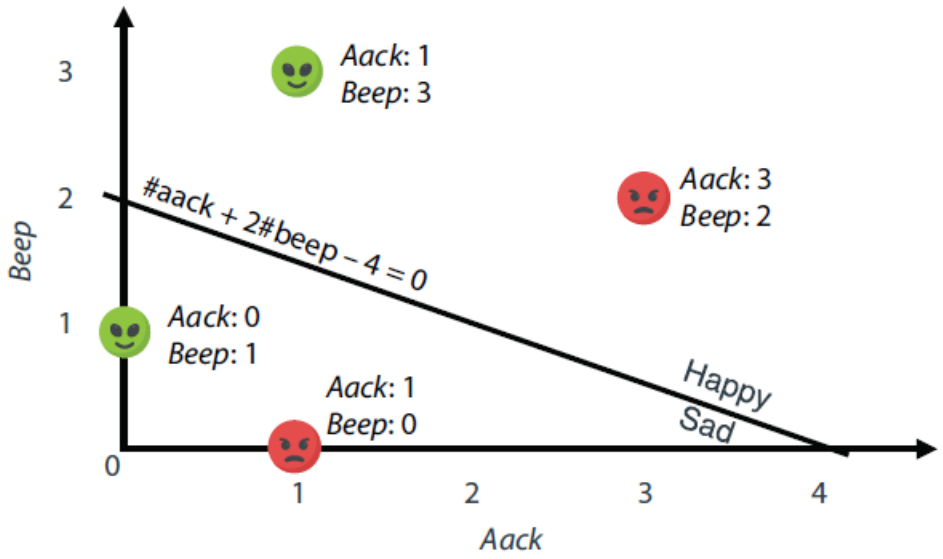
- We'll compare the following two classifiers on this dataset

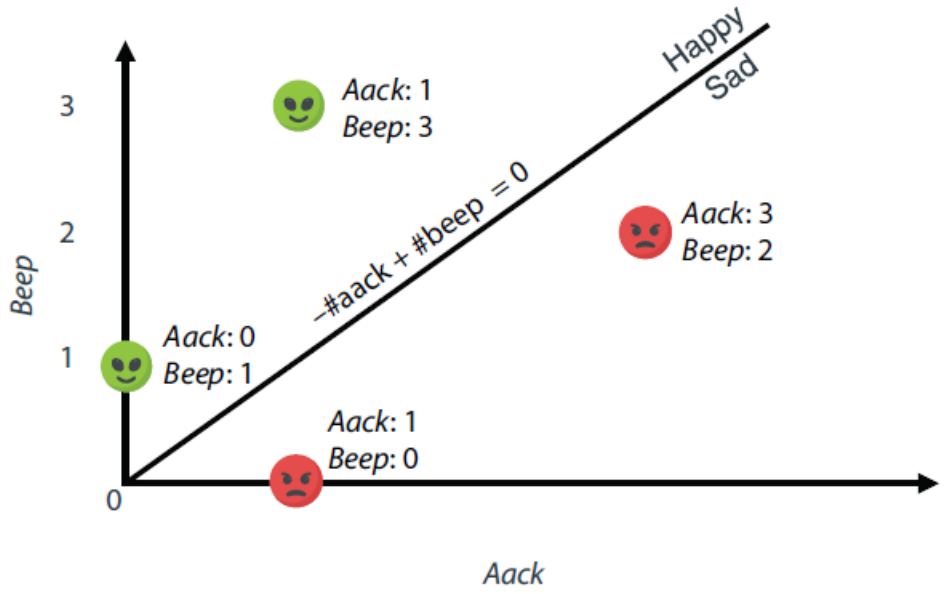
Example

Sentence	*Aack*	*Beep*	Label (mood)
Aack	1	0	Sad
Beep	0	1	Happy
Aack beep beep beep	1	3	Happy
Aack beep beep aack aack	3	2	Sad

- We'll compare the following two classifiers on this dataset
- $1x_{aack} + 2x_{beep} - 4$ $\quad \hat{y} = step(1x_{aack} + 2x_{beep} - 4)$
- $-1x_{aack} + 1x_{beep}$ $\quad \hat{y} = step(-1x_{aack} + 1x_{beep})$

Example


Aack: 1
Beep: 3
Aack: 3
Beep: 2
#aack + 2#beep – 4 = 0
Aack: 0
Beep: 1
Aack: 1
Beep: 0
Happy
Sad
Beep
Aack
0
1
2
3
4

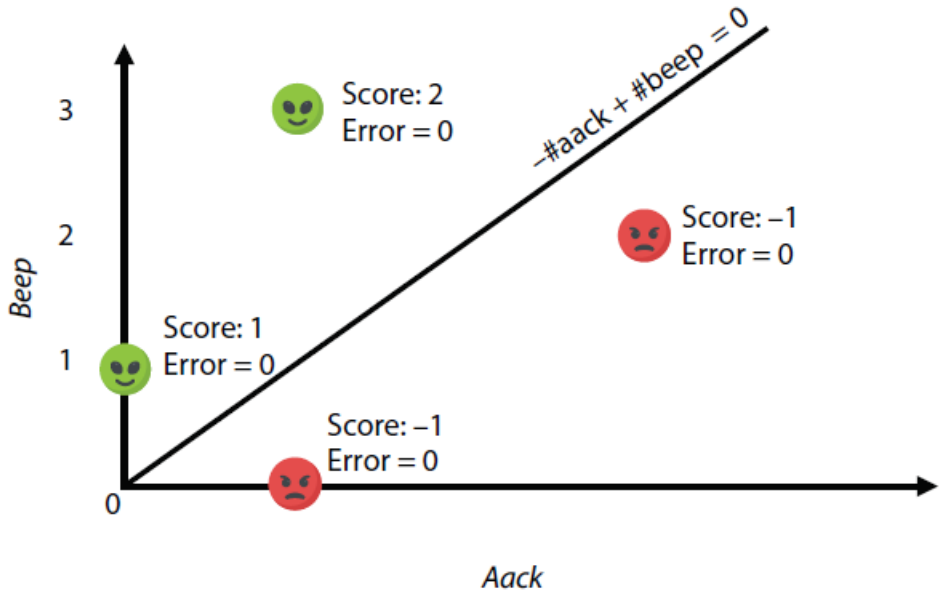

Aack: 1
Beep: 3
Happy
Sad
–#aack + #beep = 0
Aack: 3
Beep: 2
Aack: 0
Beep: 1
Aack: 1
Beep: 0
Beep
Aack
0
1
2
3

Example

Datapoint	Label	Classifier 1 score	Classifier 1 prediction	Classifier 1 error	Classifier 2 score	Classifier 2 prediction	Classifier 2 error
(1, 0)	Sad	-3	0	0	-1	0	0
(0, 1)	Happy	-2	0	2	1	1	0
(1, 3)	Happy	3	1	0	2	1	0
(3, 2)	Sad	3	1	3	-1	0	0
Mean Perceptron error				1.25			0

Example

Mean perceptron error = $\frac{1}{4}(0+3+0+2) = 1.25$



Mean perceptron error = $\frac{1}{4}(0+0+0+0) = 0$

The perceptron algorithm

- Start with a random perceptron classifier.
- Slightly improve the classifier. (Repeat many times.)
- Measure the perceptron error to decide when to stop running the loop.

The perceptron trick

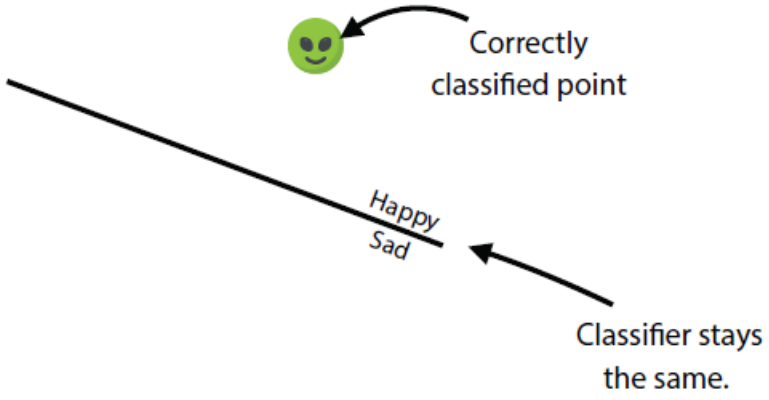
- The perceptron trick is a tiny step that helps us go from a perceptron classifier to a slightly better perceptron classifier

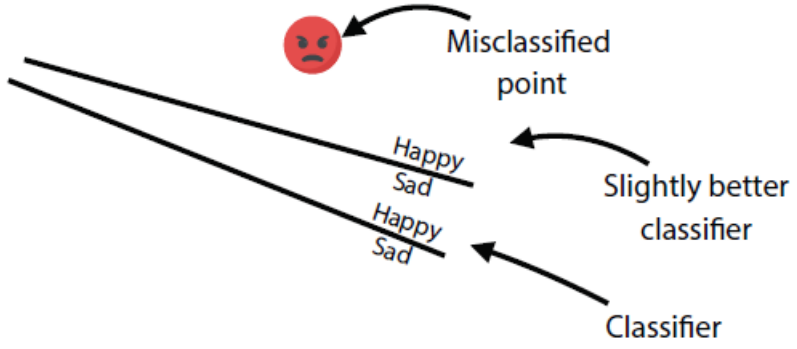
- Case 1: If the point is correctly classified, leave the line as it is.

- Case 2: If the point is **incorrectly classified**, **move** the line a **little closer** to the point.

The perceptron trick

- Moving the line closer to it may not put it on the right side, but at least it gets it closer to the line and, thus, closer to the correct side of the line.
- We repeat this process many times, so it is imaginable that one day we'll be able to move the line past the point, thus correctly classifying it





The perceptron trick

- Case 1: If the point is correctly classified, leave the classifier as it is.
- Case 2: If the point is **incorrectly classified**, that means it produces a **positive error**. Adjust the weights and the bias a small amount so that this error slightly decreases.
- **Decrease** the score is by decreasing all its parts, namely, the **weights** and the **bias**
- Decrease them by an amount equal to the **learning rate**

The perceptron trick

- we use a learning rate of $\eta = 0.01$
- $1x_{aack} + 2x_{beep} - 4$ $\quad\quad$ $\hat{y} = step(1x_{aack} + 2x_{beep} - 4)$

- Sentence 1: “Beep aack aack beep beep beep beep.”
- Label: **Sad** (0)

- Classifier 1 score: 8
- Classifier 1 prediction: 1 (**Happy**)

The perceptron trick

- we use a learning rate of $\eta = 0.01$
- $1x_{aack} + 2x_{beep} - 4$ $\hat{y} = step(1x_{aack} + 2x_{beep} - 4)$

- The sentence should have had a **negative score**, to be classified as sad. However, the classifier gave it a **score of 8**, which is **positive**. We **need to decrease** this score.
- One way to decrease it is to **subtract the learning rate** to the weight of aack to the weight of beep and to the bias, thus obtaining new weights
- Weight for Aack: 1−0.01=0.99
- Weight for Beep: 2-−0.01=1.99
- Bias: -4−0.01=−4.01

The perceptron trick

- we use a learning rate of $\eta = 0.01$
- $1x_{aack} + 2x_{beep} - 4$ $\hat{y} = step(1x_{aack} + 2x_{beep} - 4)$

- think about this: the word beep appeared many more times than the word aack. In some way, beep is more crucial to the score of the sentence than aack. We should probably decrease the weight of beep more than the score of aack

The perceptron trick

Inputs:

- A perceptron with weights a, b, and bias c
- A point with coordinates (x_1, x_2) and label y
- A small positive value η (the learning rate)

Output:

- A perceptron with new weights a', b', and bias c'

Procedure:

- The prediction the perceptron makes at the point is $\hat{y} = step(ax_1 + bx_2 + c)$.
- **Case 1:** If $\hat{y} = y$:
  - **Return** the original perceptron with weights a, b, and bias c.
- **Case 2:** If $\hat{y} = 1$ and $y = 0$:
  - **Return** the perceptron with the following weights and bias:
    - $a' = a - \eta x_1$
    - $b' = b - \eta x_2$
    - $c' = c - \eta x_1$.
- **Case 3:** If $\hat{y} = 0$ and $y = 1$:
  - **Return** the perceptron with the following weights and bias:
    - $a' = a + \eta x_1$
    - $b' = b - \eta x_2$
    - $c' = c + \eta x_1$.

The perceptron trick

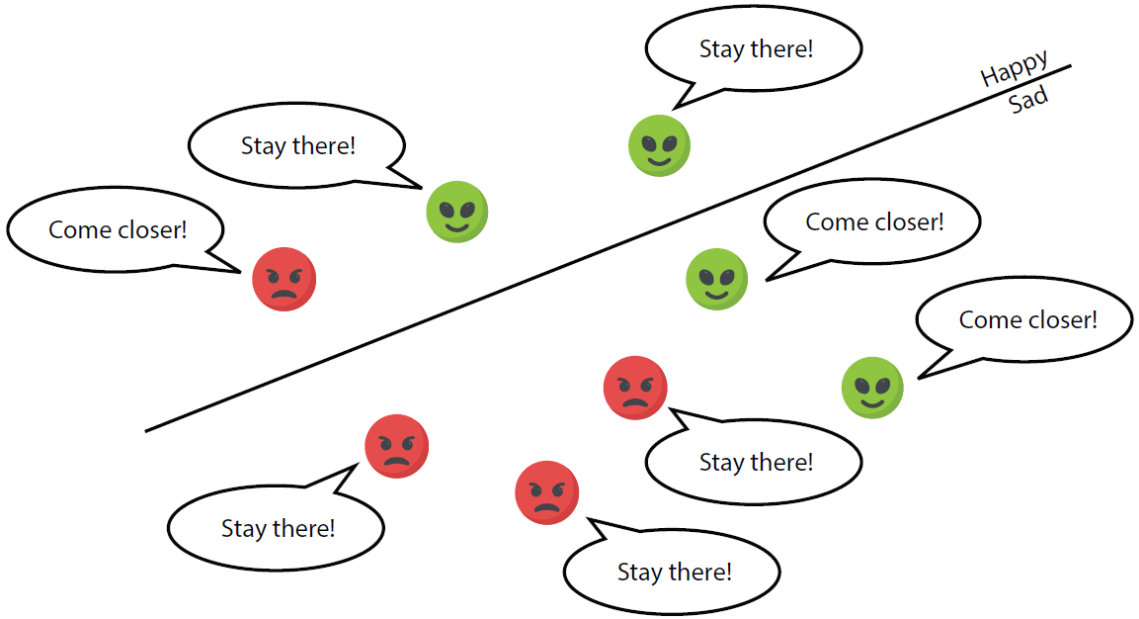
- Note that the quantity $y - \hat{y}$ is 0, –1, and +1 for the three cases in the perceptron trick. Thus, we can summarize it as follows:

Procedure:

- The prediction the perceptron makes at the point is $\hat{y} = step(ax_1 + bx_2 + c)$.
- **Return** the perceptron with the following weights and bias:
  - $a' = a + \eta(y - \hat{y})x_1$
  - $b' = b + \eta(y - \hat{y})x_2$
  - $c' = c + \eta(y - \hat{y})$

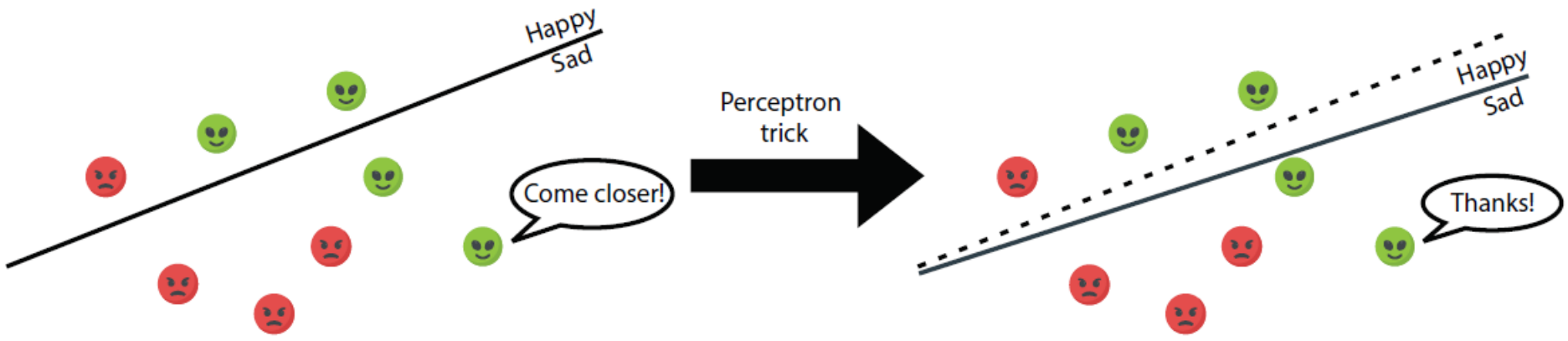
The perceptron Algorithm

- The first step of the algorithm is to draw a random line
- It doesn’t do a good job of splitting the happy and the sad sentences



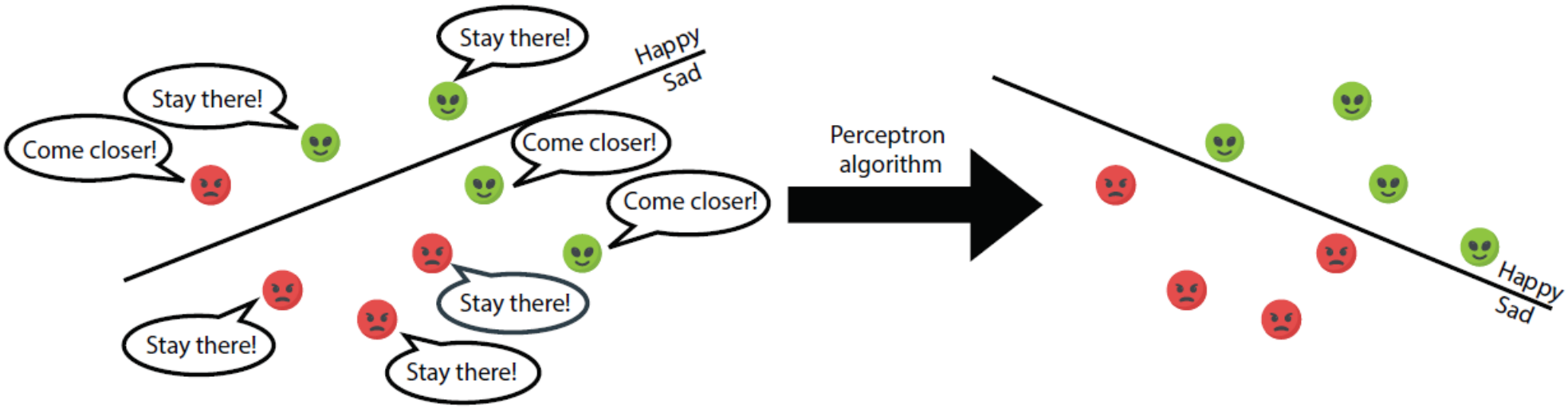
The perceptron Algorithm

- Picking one point at random
- If the point is correctly classified, the line is left alone.
- If it is misclassified, then the line gets moved slightly closer to the point



The perceptron Algorithm

- It is imaginable that if we were to repeat this process many times, eventually we will get to a good solution. This procedure doesn't always get us to the best solution. But in practice, this method often reaches to a good solution



The perceptron Algorithm

- The algorithm has two hyperparameters: the number of epochs, and the learning rate

- The number of times we run the algorithm is the number of epochs

The perceptron Algorithm

Pseudocode for the perceptron algorithm

- Inputs:
  - A dataset of points, labeled 1 and 0
  - A number of epochs, n
  - A learning rate η
- Output:
  - A perceptron classifier consisting of a set of weights and a bias that fits the dataset

The perceptron Algorithm

Pseudocode for the perceptron algorithm

Procedure:

- Start with random values for the weights and bias of the perceptron classifier.
- Repeat many times:
  - Pick a random data point.
  - Update the weights and the bias using the perceptron trick.
- **Return**: The perceptron classifier with the updated weights and bias.

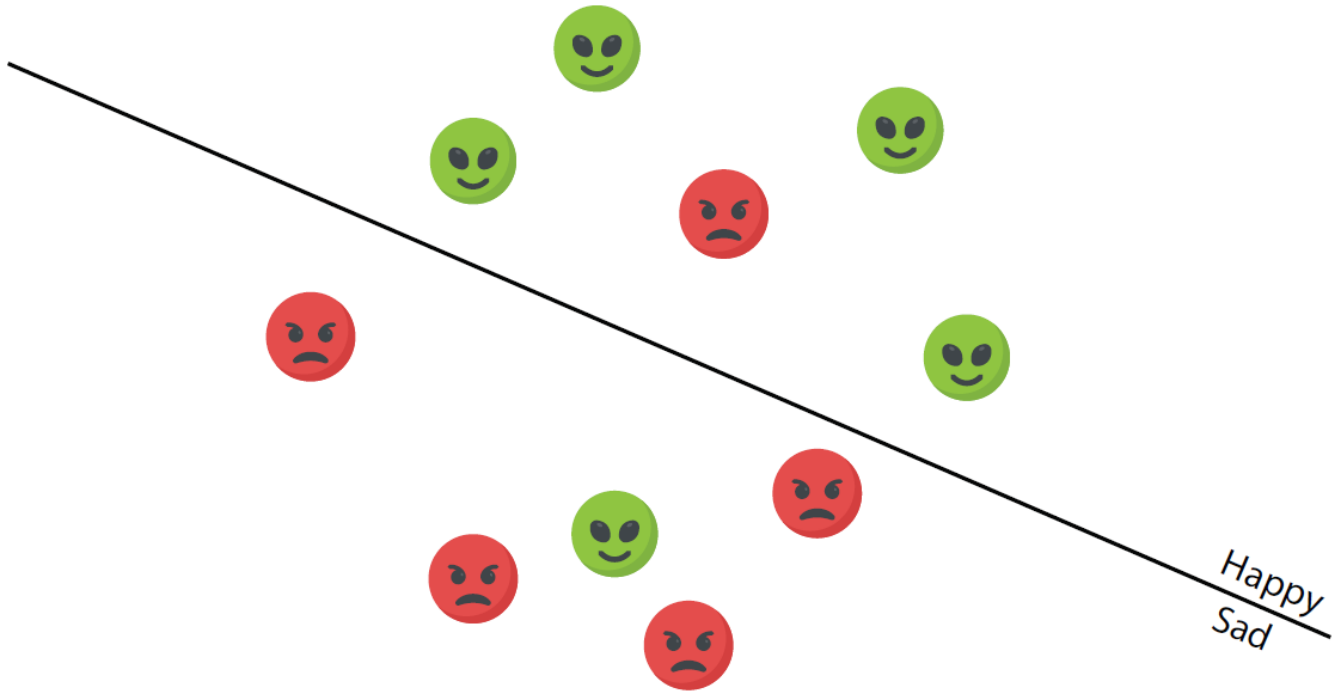
Stopping Criteria

How long should we run the loop? In other words, how many epochs should we use?

- Run the loop a fixed number of times, which could be based on our computing power, or the amount of time we have.
- Run the loop until the error is lower than a certain threshold we set beforehand.
- Run the loop until the error doesn't change significantly for a certain number of epochs.

Not Linearly separable

- It is impossible to find a line to separate the two classes in the dataset



Gradient Descent

- This process uses derivatives

Remind: (Perceptron Trick)

- Every time we "**move a small amount** in this direction," we are calculating in the background a **derivative** of the error function and using it to give us a direction in which to move our line

Gradient descent

- we want to draw a line that separates two sets of points in the best possible way.
- The perceptron algorithm is the process that starts with a **random line**, and it **slowly moves it step by step** to build a better separator.
- The analogy of descending from a mountain also works here.
- The only difference is that in this mountain, the **height** at each point is the **mean perceptron error**

Gradient Descent

- We have a metric called the **error function**, which tells us **how far a point from a classifier line**.
- Thus, if we could just **reduce mean perception error as much as possible**, we would **find the best separator** .

- This process, common in many areas in mathematics, is called **minimizing functions**, that is, finding the smallest possible value that a function can return.
- This is where **gradient descent** comes in: it is a **great way to minimize** a function.

Gradient Descent

- Gradient Descent doesn’t always find the exact minimum value of the function, but it may find something **very close** to it.
- In practice, gradient descent is **fast and effective** at finding points where the function is low

- Gradient descent is the equivalent of **descending from a mountain**.

Gradient Descent

- Gradient descent is the equivalent of **descending from a mountain**.
- We wish to **descend**, but it is **very foggy**, and we can see only about one meter away. What do we do? A good method is to look around ourselves and figure out **in what direction** we can take **one single step**, in a way that we **descend the most**.

- We could reach the bottom, or we could also reach a valley and then we have nowhere else to move

Gradient Descent

Gradient descent works as follows:

- Start somewhere on the mountain.
  - Find the best direction to take one small step.
  - Take this small step.
- Repeat steps 2 and 3 many times.

Stochastic and Batch Gradient Descent

- Repeatedly taking one point at a time and adjusting the perceptron (line) to be a better fit for that point. This is called an **epoch**.

- The better approach is to take a **batch of points at a time** and adjust the perceptron to be a better fit for those points in one step.

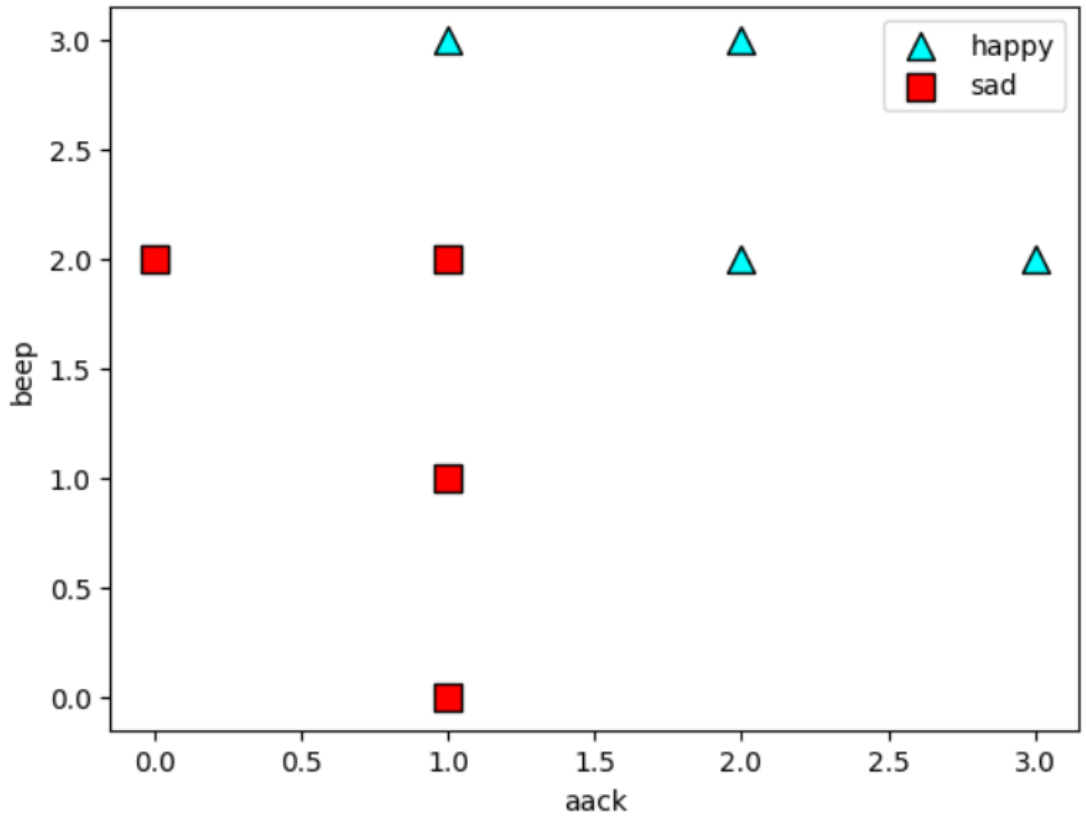
- The extreme case is to **take all the points** in the set at a time and adjust the perceptron to fit all of them better in one step

Stochastic and Batch Gradient Descent

- Batch Gradient Descent
- Mini-batch learning: Dividing data into many minibatches.
- Stochastic gradient descent: When we use one point at a time

Python Session

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



Aack	*Beep*	Happy/Sad
1	0	0
0	2	0
1	1	0
1	2	0
1	3	1
2	2	1
2	3	1
3	2	1

Python Session

```
def score(weights, bias, features):
    return features.dot(weights) + bias
```

```
def step(x):
    if x >= 0:
        return 1
    else:
        return 0
```

```
def prediction(weights, bias, features):
    return step(score(weights, bias, features))
```

```
def error(weights, bias, features, label):
    pred = prediction(weights, bias, features)
    if pred == label:
        return 0
    else:
        return np.abs(score(weights, bias, features))
```

```
def mean_perceptron_error(weights, bias, features, labels):
    total_error = 0
    for i in range(len(features)):
        total_error += error(weights, bias, features[i], labels[i])
    return total_error/len(features)
```

Python Session

```
#weights = [1,1]
#bias = -3.5
weights = [1,2]
bias = -4
for i in range(len(features)):
    print(prediction(weights, bias, features[i]), error(weights, bias, features[i], labels[i]))
```

```
0 0
1 0
0 0
1 1
1 0
1 0
1 0
1 0
```

Python Session

```
# First perceptron trick
def perceptron_trick(weights, bias, features, label, learning_rate = 0.01):
    pred = prediction(weights, bias, features)
    if pred == label:
        return weights, bias
    else:
        if label==1 and pred==0:
            for i in range(len(weights)):
                weights[i] += features[i]*learning_rate
            bias += learning_rate
        elif label==0 and pred==1:
            for i in range(len(weights)):
                weights[i] -= features[i]*learning_rate
            bias -= learning_rate
    return weights, bias
```

Python Session

```
# Shorter version of the perceptron trick
def perceptron_trick(weights, bias, features, label, learning_rate = 0.01):
    pred = prediction(weights, bias, features)
    for i in range(len(weights)):
        weights[i] += (label-pred)*features[i]*learning_rate
        bias += (label-pred)*learning_rate
    return weights, bias
```

Python Session

```
perceptron_trick(weights, bias, features[6], 0)
```

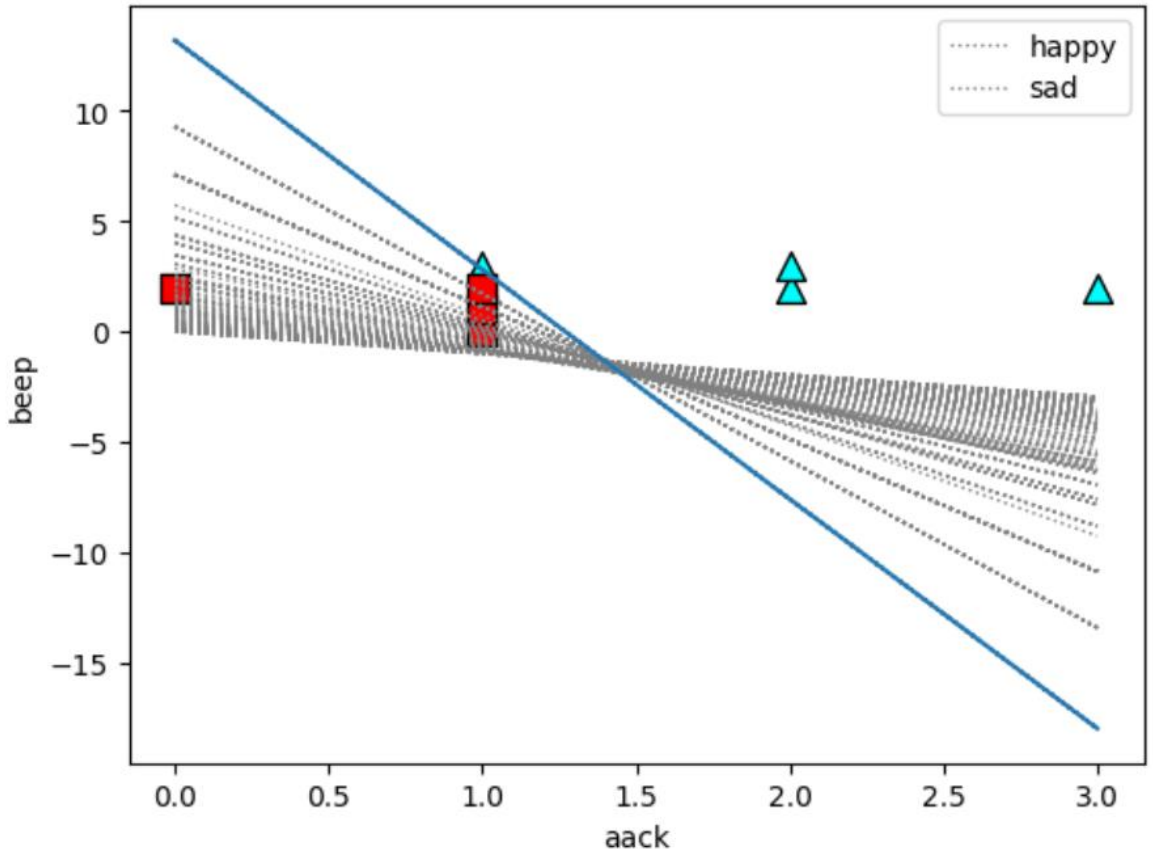
```
([0.98, 1.97], -4.01)
```

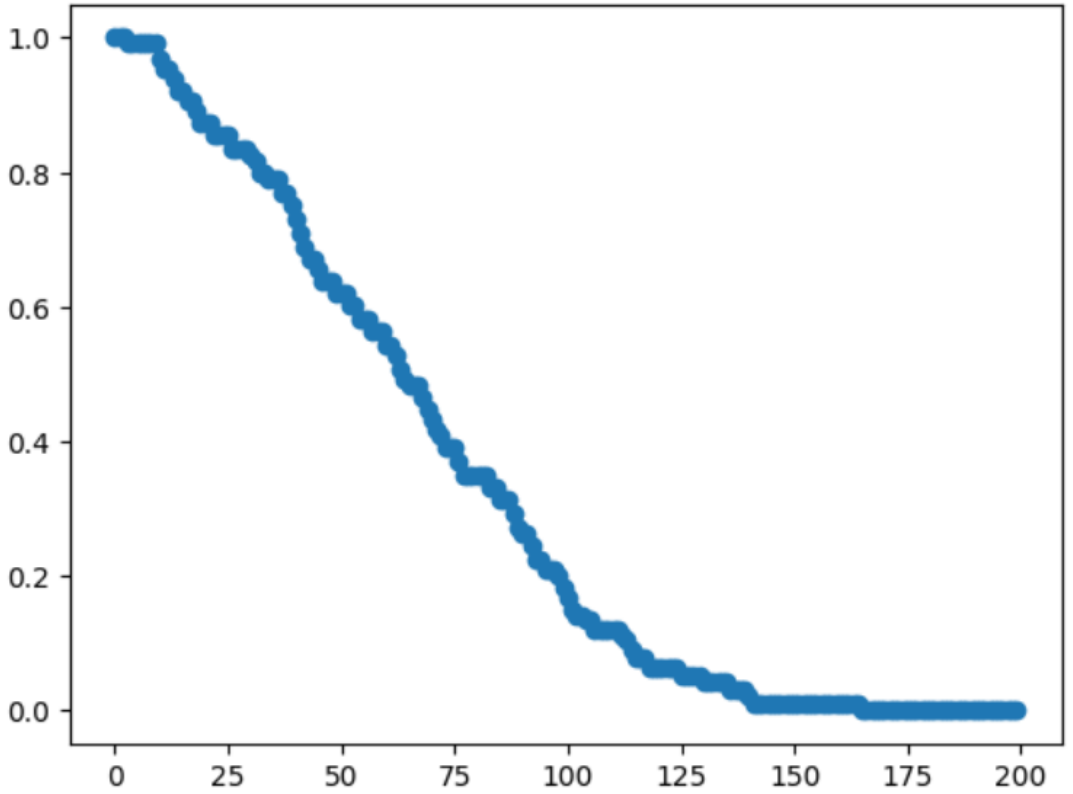
Python Session

```
random.seed(0)
def perceptron_algorithm(features, labels, learning_rate = 0.01, epochs = 200):
    weights = [1.0 for i in range(len(features[0]))]
    bias = 0.0
    errors = []
    for epoch in range(epochs):
        # Coment the following line to draw only the final classifier
        utils.draw_line(weights[0], weights[1], bias, color='grey', linewidth=1.0, linestyle='dotted')
        error = mean_perceptron_error(weights, bias, features, labels)
        errors.append(error)
        i = random.randint(0, len(features)-1)
        weights, bias = perceptron_trick(weights, bias, features[i], labels[i])
    utils.draw_line(weights[0], weights[1], bias)
    utils.plot_points(features, labels)
    plt.show()
    plt.scatter(range(epochs), errors)
    return weights, bias
```

```
perceptron_algorithm(features, labels)
```

Python Session






```
([0.5199999999999996, 0.049999999999999364], -0.6600000000000004)
```

Applications of the perceptron algorithm

- The perceptron algorithm has many applications in real life.
- Virtually every time we need to answer a question with yes or no.

  - Spam email filters
  - Recommendation Systems
  - Health care
  - Computer vision

Spam email filters

- Identify and block unwanted or unsolicited emails, commonly known as spam.
- Spam emails often include advertisements, phishing attempts, malware, or other potentially harmful content.

- Spam filters help manage and reduce the number of these emails reaching users' inboxes, thereby improving the email experience and protecting against security threats.

Spam email filters

- Identify and block unwanted or unsolicited emails, commonly known as spam.
- Spam emails often include advertisements, phishing attempts, malware, or other potentially harmful content.

- Spam filters help manage and reduce the number of these emails reaching users' inboxes, thereby improving the email experience and protecting against security threats.

Spam email filters – How it works

Spam filters typically use a combination of techniques to detect spam, including:

- **Keyword Filtering**: Analyzing the content of emails for specific words or phrases commonly associated with spam, such as "free," "win," "urgent," etc.
- **Blacklists**: Maintaining lists of known spammer IP addresses or domains. Emails from these sources are automatically flagged or blocked.
- **Heuristic Analysis**: Using rules based on common characteristics of spam emails, such as the presence of many links, suspicious attachments, or unusual sender names.

Spam email filters – How it works

Spam filters typically use a combination of techniques to detect spam, including:

- Bayesian Filtering: A statistical method that uses probabilities based on the frequency of certain words or features in known spam and legitimate emails. Over time, this filter learns to identify spam more accurately.
- Machine Learning: Advanced filters use machine learning algorithms to analyze large datasets of emails and identify patterns indicative of spam. These systems can continuously learn and improve based on user feedback and new data.

Spam email filters – How it works

Spam filters typically use a combination of techniques to detect spam, including:

- Header Analysis: Examining the metadata in email headers, such as the sender's IP address, routing information, and inconsistencies that might indicate spoofing.
- Content Analysis: Scanning the email body for scripts, executable files, or other potentially harmful content that might be embedded in attachments or links.

Spam email filters – Benefits

- Protection from Malicious Content: Helps prevent phishing attacks, malware, and other security threats.
- Improved Productivity: Reduces the time users spend sorting through unwanted emails.
- Better Email Management: Keeps the inbox clean and organized, ensuring important emails are not lost among spam.

Spam email filters – Description of Features

- Spam Keywords: Number of keywords in the email that are commonly associated with spam.
- Number of Links: Total number of hyperlinks present in the email.
- Size of Attachment (KB): The size of any attachments included in the email, measured in kilobytes.
- Number of Recipients: Number of recipients the email is addressed to. Multiple recipients can sometimes indicate spam.

Spam email filters – Description of Features

- Suspicious Domain: Binary value where 1 indicates the sender's domain is known for spam activity, and 0 indicates it is not.
- Known Sender: Binary value where 1 means the sender is in the recipient's contact list, and 0 means they are not.
- Spam Label: The label indicating whether the email is classified as spam (1) or not (0).

Spam email filters – Description of Features

Email ID	Spam Keywords	Number of Links	Size of Attachment (KB)	Number of Recipients	Suspicious Domain	Known Sender	Spam Label
1	5	10	500	1	1	0	1
2	0	1	0	1	0	1	0
3	8	15	1200	3	1	0	1
4	2	2	50	2	0	1	0
5	6	5	800	1	1	0	1
6	1	0	0	1	0	1	0
7	4	3	300	1	0	1	0
8	7	7	450	1	1	0	1
9	0	1	0	1	0	1	0
10	5	12	600	1	1	0	1

Spam email filters – Description of Features

- $\hat{y} = step(w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6)$

- x_i: Feature
- w_i: Weight
- w_0: Bias
- x_1: Spam Keywords
- x_2 : Number of Links
- x_3 : Size of Attachment
- x_4 : Number of Recipients
- x_5 : Suspicious Domain
- x_6 : Known Sender

Spam email filters – Description of Features

- $\hat{y} = step(-1 + 2x_1 + 1.5x_2 + 0.002x_3 + 1.2x_4 + 2.5x_5 + -3x_6)$

- x_1 : Spam Keywords – A higher weight for this feature (2.0) seems reasonable, as the presence of certain keywords is often a strong indicator of spam.
- x_2: Number of Links – A weight of 1.5 suggests a moderate correlation between the number of links and the likelihood of an email being spam, which is plausible.
- x_3:Size of Attachment – The weight of 0.002 indicates a very small contribution to the decision. This makes sense as the size of an attachment alone may not strongly indicate spam.

Spam email filters – Description of Features

- $\hat{y} = step(-1 + 2x_1 + 1.5x_2 + 0.002x_3 + 1.2x_4 + 2.5x_5 + -3x_6)$

- x_4: Number of Recipients – A weight of 1.2 suggests that emails sent to multiple recipients may be more likely to be spam, which is a reasonable assumption.
- x_5: Suspicious Domain – A weight of 2.5 indicates a strong association with spam, which is realistic given that many spam emails come from questionable domains.

Spam email filters – Description of Features

- $\hat{y} = step(-1 + 2x_1 + 1.5x_2 + 0.002x_3 + 1.2x_4 + 2.5x_5 + -3x_6)$

- x_6: Known Sender – A negative weight of -3.0 indicates that known senders significantly decrease the likelihood of an email being classified as spam. This is reasonable because emails from known contacts are typically not spam.
- Bias Term: The bias term (-1) helps shift the decision boundary. Its reasonableness depends on the overall distribution of spam and non-spam emails. A negative bias suggests that the model is slightly biased towards predicting non-spam unless there's strong evidence to classify an email as spam.

Spam email filters – Description of Features

- Currently, the perceptron algorithm (and its more advanced counterparts, logistic regression and neural networks) and other classification models are used as a part of spam classification pipelines by most of the biggest email providers, with great results.

- We can also categorize emails using classification algorithms like the perceptron algorithm.

Spam email filters – Description of Features

- Classifying email into personal, subscriptions, and promotions is the exact same problem.

- Even coming up with potential responses to an email is also a classification problem, except now the labels that we use are responses to emails.

Recommendation Systems

- In many recommendation systems, recommending a video, movie, song, or product to a user boils down to a yes or no answer

  - Will the user click on the video/movie we're recommending?
  - Will the user watch the entire video/movie we're recommending?
  - Will the user listen to the song we're recommending?
  - Will the user buy the product we're recommending?

Recommendation Systems

- The features can be anything, from demographic (age, gender, location of the user), to behavioral (what videos did the user watch, what songs did they hear, what products did they buy?).
- You can imagine that the user vector would be a long one! For this, large computing power and very clever implementations of the algorithms are needed.
- Companies such as Netflix, YouTube, and Amazon, among many others, use the perceptron algorithm or similar, more advanced classification models in their recommendation systems.

Description of Features

Predict whether a user will click on a recommended video or movie

- Age: The age of the user (continuous numerical feature).
- Gender: Coded as 0 for male and 1 for female (binary feature).
- Location: Coded as integers representing different regions (1, 2, 3, etc.).
- Video Genre: Coded as integers representing different genres (1 for Action, 2 for Comedy, etc.).

Description of Features

- Previous Watch Time (hrs): Total hours the user has spent watching videos (continuous numerical feature).
- Number of Previous Clicks: The number of times the user has clicked on recommended videos previously (integer).
- Purchase History: Binary indicator of whether the user has made purchases on the platform (0 for no, 1 for yes).
- Click Label: The target variable indicating whether the user clicked on the recommended video (1) or not (0).

Dataset

User ID	x_1 (Age)	x_2 (Gender)	x_3 (Location)	x_4 (Video Genre)	x_5 (Previous Watch Time (hrs))	x_6 (Number of Previous Clicks)	x_7 (Purchase History)	Click Label
1	25	1	1	3	15	5	0	1
2	34	0	2	2	10	2	1	0
3	19	1	3	1	20	10	0	1
4	45	0	1	4	5	0	1	0
5	28	1	2	5	30	7	1	1
6	32	0	3	2	8	3	0	0
7	22	1	1	3	18	8	0	1
8	27	0	2	1	12	6	1	1
9	40	1	3	4	7	1	0	0
10	23	1	1	5	25	9	1	1

Recommendation Systems

- $\hat{y} = step(w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6 + w_7x_7)$

- x_i: Feature
- w_i: Weight
- w_0: Bias
- x_1: Age
- x_2 : Gender
- x_3 : Location
- x_4 : Genre
- x_5 : Previous Watch Time (hrs)
- x_6 : Number of Previous Clicks
- x_7: Purchase History

Correlation

- Age ($w_1 = 0.05$): A positive but small weight suggests a slight tendency for older users to click on recommended videos more frequently. This could reflect a higher engagement level or a preference for certain content types associated with age.

- Video Genre ($w_4 = 0.3$): A significant positive weight indicates that certain video genres are more likely to lead to clicks. This is a key feature as users often have specific genre preferences that influence their engagement.

Correlation

- Previous Watch Time ($w_5 = 0.04$): The positive weight here, albeit small, suggests that users who have spent more time watching videos are slightly more likely to click on recommendations. This could indicate higher overall engagement with the platform.

- Number of Previous Clicks ($w_6 = 0.5$): A relatively large positive weight implies that users who have clicked on recommendations before are more likely to do so again. This feature is a strong indicator of a user's engagement with recommended content.

Correlation

- Purchase History ($w_7 = 0.7$): This is one of the strongest indicators, with a significant positive weight. Users who have made purchases may be more engaged or invested in the platform, increasing their likelihood of clicking on recommended videos.

- Bias Term ($w_0 = -1.0$): The negative bias term suggests a general tendency towards predicting no click unless sufficient positive evidence (from the features) shifts the decision towards predicting a click. This may reflect a default assumption that clicks are not as frequent as non-clicks, potentially due to click rates generally being lower than non-click rates.

Correlation

- Gender ($w_2 = 0.2$): The moderate positive weight suggests some influence of gender on click behavior, but it is not as strong as other features. This could imply that while gender may play a role, it is not a predominant factor.
- Location ($w_3 = 0.1$): The small positive weight indicates a slight correlation between the user's location and their likelihood of clicking. This could reflect regional preferences or differences in content availability and interests, but the effect is relatively minor.

Health care

- Does the patient suffer from a particular illness?
- Will a certain treatment work for a patient?
- The features for these models will normally be the symptoms the patient is suffering and their medical history.

Health care

- For these types of algorithms, one needs very high levels of performance.

- Recommending the wrong treatment for a patient is much more serious than recommending a video that a user won’t watch

Computer vision

- Classification algorithms such as the perceptron algorithm are widely used in computer vision, more specifically, in image recognition.

- Imagine that we have a picture, and we want to teach the computer to tell whether the picture contains a dog. This is a classification model in which the features are the pixels of the image.

Computer vision

- The perceptron algorithm has decent performance in curated image datasets such as MNIST, which is a dataset of handwritten digits. However, for more complicated images, it doesn’t do very well.

- For these, one uses models that consist of a combination of many perceptrons. These models are aptly called multilayer perceptrons, or neural networks