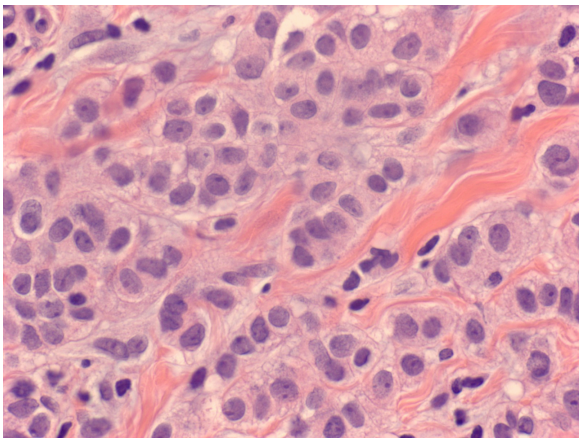
machine learning lab

Coding the Matrix, 2015

`For auto-graded problems, edit the file machine_learning_lab.py to include your solution.`

In this lab you will use a rudimentary machine-learning algorithm to learn to diagnose breast cancer from features.



The core idea is the use of *gradient descent*, an iterative method to find a "best" hypothesis. Gradient descent is useful in finding a point that nearly minimizes a nonlinear function. In each iteration, it approximates the function by a linear function.

Disclaimer: For this particular function, there is a much faster and more direct way of finding the best point. We will learn it in Orthogonalization. However, gradient descent is useful more generally and is well worth knowing.

1 *The data*

You are given part of the Wisconsin Diagnostic Breast Cancer (WDBC) dataset. For each patient, you are given a vector $\boldsymbol{a}$ giving features computed from digitized images of a fine needle aspirate of a breast mass for that patient. The features describe characteristics of the cell nuclei present in the image. The goal is to decide whether the cells are malignant or benign.

Here is a brief description of the way the features were computed. Ten real-valued quantities are computed for each cell nucleus:

- radius (mean of distances from center to points on the perimeter)
- texture (standard deviation of gray-scale values)
- perimeter
- area
- smoothness (local variation in radius lengths)
- compactness (perimeter^2/ area)

- concavity (severity of concave portions of the contour)
- concave points (number of concave portions of the contour)
- symmetry
- fractal dimension ("coastline approximation")

The mean, standard error, and a measure of the largest (mean of the three largest values) of these features were computed for each image. Thus each specimen is represented by a vector $\boldsymbol{a}$ with thirty entries. The domain D consists of thirty strings identifying these features, e.g. `"radius (mean)"`, `"radius (stderr)"`, `"radius (worst)"`, `"area (mean)"`, and so on.

We provide two files containing data, `train.data` and `validate.data`.

The procedure `read_training_data` in the `cancer_data` module takes a single argument, a string giving the pathname of a file. It reads the data in the specified file and returns a pair $(A, \boldsymbol{b})$ where:

- A is a Mat whose row labels are patient identification numbers and whose column-label set is D
- $\boldsymbol{b}$ is a vector whose domain is the set of patient identification numbers, and $\boldsymbol{b}[r]$ is 1 if the specimen of patient r is malignant and is -1 if the specimen is benign.

Ungraded Task: Use `read_training_data` to read the data in the file `train.data` into the variables $A, \boldsymbol{b}$.

2 *Supervised learning*

Your goal is to write a program to select a *classifier*, a function $C(\boldsymbol{y})$ that, given a feature vector $\boldsymbol{a}$, predicts whether the tissue is malignant or benign. To enable the program to select a classifier that is likely to be accurate, the program is provided with *training data* consisting of *labeled examples* $(\boldsymbol{a}_1, b_1), \ldots, (\boldsymbol{a}_m, b_m)$. Each labeled example consists of a feature vector $\boldsymbol{a}_i$ and the corresponding label b_i, which is +1 or -1 (+1 for *malignant*, -1 for *benign*). Once the program has selected a classifier, the classifier is tested for its accuracy on unlabeled feature vectors $\boldsymbol{a}$ for which the correct answers are known.

3 *Hypothesis class*

A classifier is selected from a set of possible classifiers (the *hypothesis class*). In this case (as is often the case in machine learning), the hypothesis class consists of linear functions $h(\cdot)$ from the space $\mathbb{R}^D$ of feature vectors to $\mathbb{R}$. The classifier is defined in terms of such a function as follows:

$$C(\boldsymbol{y}) = \begin{cases} +1 & \text{if } h(\boldsymbol{y}) \geq 0 \\ -1 & \text{if } h(\boldsymbol{y}) < 0 \end{cases}$$

For each linear function $h : \mathbb{R}^D \longrightarrow \mathbb{R}$, there is a D-vector $\boldsymbol{w}$ such that

$$h(\boldsymbol{y}) = \boldsymbol{w} \cdot \boldsymbol{y}$$

Thus selecting such a linear function amounts to selecting a D-vector $\boldsymbol{w}$. We refer to $\boldsymbol{w}$ as a *hypothesis vector* since choosing $\boldsymbol{w}$ is equivalent to choosing the hypothesis h.

You will write a procedure that calculates, for a given hypothesis vector $\boldsymbol{w}$, the number of labeled examples incorrectly predicted by the classifier that uses function $h(\boldsymbol{y}) = \boldsymbol{w} \cdot \boldsymbol{y}$. To make this easier, you will first write a simple utility procedure.

Task 1: Write the procedure `signum(u)` with the following spec:

- *input:* a Vec $\boldsymbol{u}$
- *output:* the Vec $\boldsymbol{v}$ with the same domain as $\boldsymbol{u}$ such that

$$\boldsymbol{v}[d] = \begin{cases} +1 & \text{if } \boldsymbol{u}[d] \geq 0 \\ -1 & \text{if } \boldsymbol{u}[d] < 0 \end{cases}$$

For example, `signum(Vec({'A','B'}, {'A':3, 'B':-2}))` is
`Vec({'A', 'B'},{'A': 1, 'B': -1})`

Task 2: Write the procedure `fraction_wrong(A, b, w)` with the following spec:

- *input:* An $R \times C$ matrix A whose rows are feature vectors, an R-vector $\boldsymbol{b}$ whose entries are $+1$ and -1, and a C-vector $\boldsymbol{w}$
- *output:* The fraction of of row labels r of A such that the sign of (row r of A) $\cdot\, \boldsymbol{w}$ differs from that of $\boldsymbol{b}[r]$.

(Hint: There is a clever way to write this without any explicit loops using matrix-vector multiplication and dot-product and the `signum` procedure you wrote.)

Pick a simple hypothesis vector such as $[1, 1, 1, ..., 1]$ or a random vector of $+1$'s and -1's, and see how well it classifies the data.

4 *Selecting the classifier that minimizes the error on the training data*

How should the function h be selected? We will define a way of measuring the error of a particular choice of h with respect to the training data, and the program will select the function with the minimum error among all classifiers in the hypothesis class.

The obvious way of measuring the error of a hypothesis is by using the fraction of labeled examples the hypothesis gets wrong, but it is too hard to find the solution that is best with respect to this criterion, so other ways of measuring the error are used. In this lab, we use a very rudimentary measure of error. For each labeled example $(\boldsymbol{a}_i, b_i)$, the error of h on that example is $(h(\boldsymbol{a}_i) - b_i)^2$. If $h(\boldsymbol{a}_i)$ is close to b_i then this error is small. The overall error on the training data is the sum of the errors on each of the labeled examples:

$$(h(\boldsymbol{a}_1) - b_1)^2 + (h(\boldsymbol{a}_2) - b_2)^2 + \cdots + (h(\boldsymbol{a}_m) - b_m)^2$$

Recall that choosing a function $h(\cdot)$ is equivalent to choosing a D-vector $\boldsymbol{w}$ and defining $h(\boldsymbol{y}) = \boldsymbol{y} \cdot \boldsymbol{w}$. The corresponding error is

$$(\boldsymbol{a}_1 \cdot \boldsymbol{w} - b_1)^2 + (\boldsymbol{a}_2 \cdot \boldsymbol{w} - b_2)^2 + \cdots + (\boldsymbol{a}_m \cdot \boldsymbol{w} - b_m)^2$$

Now we can state our goal for the learning algorithm. We define a function $L : \mathbb{R}^D \longrightarrow \mathbb{R}$ by the rule

$$L(\boldsymbol{x}) = (\boldsymbol{a}_1 \cdot \boldsymbol{x} - b_1)^2 + (\boldsymbol{a}_2 \cdot \boldsymbol{x} - b_2)^2 + \cdots + (\boldsymbol{a}_m \cdot \boldsymbol{x} - b_m)^2$$

This function is the *loss* function on the training data. It is used to measure the error of a particular choice of the hypothesis vector $\boldsymbol{w}$. The goal of the learning algorithm is to select the hypothesis vector $\boldsymbol{w}$ that makes $L(\boldsymbol{w})$ as small as possible (in other words, the *minimizer* of the function L).

One reason we chose this particular loss function is that it can be related to the linear algebra we are studying. Let A be the matrix whose rows are the training examples $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_m$. Let $\boldsymbol{b}$ be the m-vector whose i^{th} entry is b_i. Let $\boldsymbol{w}$ be a D-vector. By the dot-product definition of matrix-vector multiplication, entry i of the vector $A\boldsymbol{w} - \boldsymbol{b}$ is $\boldsymbol{a}_i \cdot \boldsymbol{w} - b_i$. The squared norm of this vector is therefore $(\boldsymbol{a}_1 \cdot \boldsymbol{w} - b_1)^2 + \cdots + (\boldsymbol{a}_m \cdot \boldsymbol{w} - b_m)^2$. It follows that our goal is to select the vector $\boldsymbol{w}$ minimizing $\|A\boldsymbol{w} - \boldsymbol{b}\|^2$.

In Orthogonalization, we learn that this computational problem can be solved by an algorithm that uses orthogonality and projection.

~~Task 3:~~ Write a procedure `loss(A, b, w)` that takes as input the training data $A, \boldsymbol{b}$ and a hypothesis vector $\boldsymbol{w}$, and returns the value $L(\boldsymbol{w})$ of the loss function for input $\boldsymbol{w}$. (Hint: You should be able to write this without any loops, using matrix multiplication and dot-product.)

Find the value of the loss function at a simple hypothesis vector such as the all-ones vector or a random vector of +1's and -1's.

5 *Nonlinear optimization by hill-climbing*

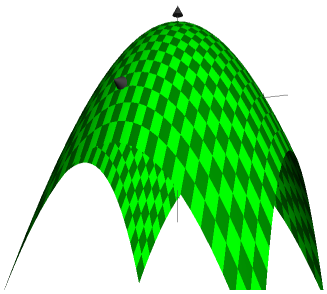
In this lab, however, we use a generic and commonly used heuristic for finding the minimizer of a function, *hill-climbing*. I call it *generic* because it can be used for a very broad class of functions; however, I refer to it as a *heuristic* because in general it is not guaranteed to find the true minimum (and often fails to do so). Generality of applicability comes at a price.

Hill-climbing maintains a solution $\boldsymbol{w}$ and iteratively makes small changes to it, in our case using vector addition. Thus it has the general form

initialize $\boldsymbol{w}$ to something
repeat as many times as you have patience for:
  $\boldsymbol{w} := \boldsymbol{w} + change$
return $\boldsymbol{w}$

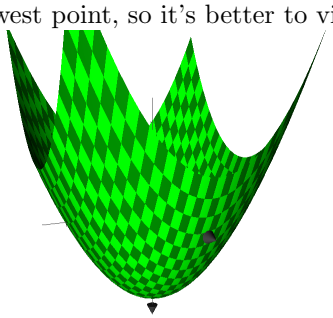
where *change* is a small vector that depends on the current value of $\boldsymbol{w}$. The goal is that each iteration improves the value of the function being optimized.

Imagine that the space of solutions forms a plane. Each possible solution $\boldsymbol{w}$ is assigned a value by the function being optimized. Interpret the value of each solution as the altitude. One can visualize the space as a three dimensional terrain.



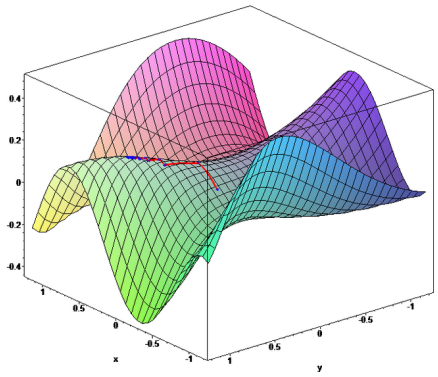
If we were trying to find a *maximizer* of the function, the algorithm gradually move the solution $\boldsymbol{w}$ towards the top of the terrain, thus the name *hill-climbing*.

In our case, the goal is to find the lowest point, so it's better to visualize the situation thus:



In this case, the algorithm tries to climb *down* the hill.

The strategy of hill-climbing works okay when the terrain is simple, but it is often applied to much more complicated terrains, e.g.



In such cases, hill-climbing usually terminates with a solution that is not truly a minimum for that function. Intuitively, the algorithm descends a hill until gets to the lowest point in a valley. It has not reached the point of smallest elevation—that point is somewhere far from the valley—but the algorithm cannot proceed because it is only allowed to descend, and there is nowhere nearby that has lower elevation. Such a point is called a *local* minimum (as opposed to a *global* minimum). This is an annoying aspect of hill-climbing but it is inevitable since hill-climbing can be applied to functions for which finding the global minimum is a computationally intractable problem.

6 *Gradient*

How should the *change* vector be selected in each iteration?

Example 0.2: Suppose the function to be minimized were a linear function, say $f(\boldsymbol{w}) = \boldsymbol{c} \cdot \boldsymbol{w}$. Suppose we change $\boldsymbol{w}$ by adding some vector $\boldsymbol{u}$ for which $\boldsymbol{c} \cdot \boldsymbol{u} < 0$. It follows that $f(\boldsymbol{w} + \boldsymbol{u}) < f(\boldsymbol{w})$ so we will make progress by assigning $\boldsymbol{w} + \boldsymbol{u}$ to $\boldsymbol{w}$. Moving in the direction $\boldsymbol{u}$ decreases the function's value.

In this lab, however, the function to be minimized is not a linear function. As a consequence, the right direction depends on where you are. for each particular point $\boldsymbol{w}$, there is in fact a direction of steepest descent *from that point*. We should move in that direction! Of course, once we've moved a little bit, the direction of steepest descent will have changed, so we recompute it and move a little more. We can move only a little bit in each iteration before we have to recompute the direction to move.

For a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$, the *gradient* of f, written ∇f, is a function from $\mathbb{R}^n$ to $\mathbb{R}^n$. Note that it outputs a vector, not a single number. For any particular input vector $\boldsymbol{w}$, the direction of steepest ascent of $f(\boldsymbol{x})$ for inputs near $\boldsymbol{w}$ is $\nabla f(\boldsymbol{w})$, the value of the function ∇f applied to $\boldsymbol{w}$. The direction of steepest *de*scent is the negative of $\nabla f(\boldsymbol{w})$.

The definition of the gradient is the one place in this course where we use calculus. If you don't know calculus, the derivation won't make sense but you can still do the lab.

Definition 0.3: The gradient of $f([x_1, \ldots, x_n])$ is defined to be

$$\left[\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right]$$

Example 0.4: Let's return once more to the simple case where f is a linear function: $f(\boldsymbol{x}) = \boldsymbol{c} \cdot \boldsymbol{x}$. That means, of course, $f([x_1, \ldots, x_n]) = c_1 x_1 + \cdots + c_n x_n$. The partial derivative of f with respect to x_i is just c_i. Therefore $\nabla f([x_1, \ldots, x_n]) = [c_1, \ldots, c_n]$. This function disregards its argument; the gradient is the same everywhere.

Example 0.5: Let's take a function that is not linear. For a vector $\boldsymbol{a}$ and a scalar b, define $f(\boldsymbol{x}) = (\boldsymbol{a} \cdot \boldsymbol{x} - b)^2$. Write $\boldsymbol{x} = [x_1, \ldots, x_n]$. Then, for $j = 1, \ldots, n$,

$$\begin{aligned} \frac{\partial f}{\partial x_j} &= 2(\boldsymbol{a} \cdot \boldsymbol{x} - b) \frac{\partial}{\partial x_j} (\boldsymbol{a} \cdot \boldsymbol{x} - b) \\ &= 2(\boldsymbol{a} \cdot \boldsymbol{x} - b) a_j \end{aligned}$$

One reason for our choice of loss function

$$L(\boldsymbol{x}) = \sum_{i=1}^{m} (\boldsymbol{a}_i \cdot \boldsymbol{x} - b_i)^2$$

is that partial derivatives of this function exist and are easy to compute (if you remember a bit of calculus). The partial derivative of $L(\boldsymbol{x})$ with respect to x_j is

$$\begin{aligned} \frac{\partial L}{\partial x_j} &= \sum_{i=1}^{m} \frac{\partial}{\partial x_j} (\boldsymbol{a}_i \cdot \boldsymbol{x} - b_i)^2 \\ &= \sum_{i=1}^{m} 2(\boldsymbol{a}_i \cdot \boldsymbol{x} - b_i) a_{ij} \end{aligned}$$

where a_{ij} is entry j of $\boldsymbol{a}_i$.

Thus the value of the gradient function for a vector $\boldsymbol{w}$ is a vector whose entry j is

$$\sum_{i=1}^{m} 2(\boldsymbol{a}_i \cdot \boldsymbol{w} - b_i) a_{ij}$$

That is, the vector is

$$\nabla L(\boldsymbol{w}) = \left[\sum_{i=1}^{m} 2(\boldsymbol{a}_i \cdot \boldsymbol{w} - b_i) a_{i1}, \ldots, \sum_{i=1}^{m} 2(\boldsymbol{a}_i \cdot \boldsymbol{w} - b_i) a_{in} \right]$$

which can be rewritten using vector addition as

$$\sum_{i=1}^{m} 2(\boldsymbol{a}_i \cdot \boldsymbol{w} - b_i) \boldsymbol{a}_i \qquad (1)$$

$2 \cdot [A \cdot \bar{w} - b]^T A$

Task 4: Write a procedure `find_grad(A, b, w)` that takes as input the training data $A, \boldsymbol{b}$ and a hypothesis vector $\boldsymbol{w}$ and returns the value of the gradient of L at the point $\boldsymbol{w}$, using Equation 1. (Hint: You can write this without any loops, by using matrix multiplication and transpose and vector addition/subtraction.)

7 *Gradient descent*

The idea of gradient descent is to update the vector $\boldsymbol{w}$ iteratively; in each iteration, the algorithm adds to $\boldsymbol{w}$ a small scalar multiple of the negative of the value of the gradient at $\boldsymbol{w}$. The scalar is called the *step size*, and we denote it by σ.

Why should the step size be a small number? You might think that a big step allows the algorithm to make lots of progress in each iteration, but, since the gradient changes every time the hypothesis vector changes, it is safer to use a small number to as not to overshoot. (A more sophisticated method might adapt the step size as the computation proceeds.)

The basic algorithm for gradient descent is then

> Set σ to be a small number
> Initialize $\boldsymbol{w}$ to be some D-vector
> repeat some number of times:
> $\boldsymbol{w} := \boldsymbol{w} + \sigma(\nabla L(\boldsymbol{w}))$
> return $\boldsymbol{w}$

Task 5: Write a procedure `gradient_descent_step(A, b, w, sigma)` that, given the training data $A, \boldsymbol{b}$ and the current hypothesis vector $\boldsymbol{w}$, returns the next hypothesis vector.

The next hypothesis vector is obtained by computing the gradient, multiplying the gradient by the step size, and subtracting the result from the current hypothesis vector. (Why subtraction? Remember, the gradient is the direction of steepest ascent, the direction in which the function increases.)

Ungraded Task: Write a procedure `gradient_descent(A, b, w, sigma, T)` that takes as input the training data A, $\boldsymbol{b}$, an initial value $\boldsymbol{w}$ for the hypothesis vector, a step size σ, and a number T of iterations. The procedure should implement gradient descent as described above for T iterations, and return the final value of $\boldsymbol{w}$. It should use `gradient_descent_step` as a subroutine.

Every thirty iterations or so, the procedure should print out the value of the loss function and the fraction wrong for the current hypothesis vector.

Ungraded Task: Try out your gradient descent code on the training data! Notice that the fraction wrong might go up even while the value of the loss function goes down. Eventually, as the value of the loss function continues to decrease, the fraction wrong should also decrease (up to a point).

The algorithm is sensitive to the step size. While in principle the value of `loss` should go down in each iteration, that might not happen if the step size is too big. On the other hand, if the step size is too small, the number of iterations could be large. Try a step size of $\sigma = 2 \cdot 10^{-9}$, then try a step size of $\sigma = 10^{-9}$.

The algorithm is also sensitive to the initial value of $\boldsymbol{w}$. Try starting with the all-ones vector. Then try starting with the zero vector.

Ungraded Task: After you have used your gradient descent code to find a hypothesis vector $\boldsymbol{w}$, see how well this hypothesis works for the data in the file `validate.data`. What is the percentage of samples that are incorrectly classified? Is it greater or smaller than the success rate on the training data? Can you explain the difference in performance?