CS182 January 15, 2021
Mehran Sahami

A Very Brief Introduction to Probability and Machine Learning with the Perceptron Algorithm

To aid our discussion of algorithmic decision-making, we present here a very brief introduction to probability theory, machine learning, and the Perceptron learning algorithm. Our goal in not to provide a thorough treatment of these topics (which are covered in several different course at Stanford, including CS109 and CS229). Rather we aim to provide a minimal background on these topics for those who may not already be familiar with them to make the technical material in the other readings and assignment in this unit more understandable. If you are already familiar with these topics, feel free to skim or skip parts of this document.

Probability

Informally speaking, a probability is simply the chance that some outcome will occur. For example, we can ask: what is the probability that a coin that is flipped comes up "heads"? If the coin is *fair* (i.e., both sides are equally likely to come up), then we would say that the probability of landing "heads" would be 50% or 0.5.

When working more formally with probabilities, we often define *variables* that take on numeric values, representing some outcome we may care about. For example, we could formalize the notion of a coin being flipped by defining a binary variable X which takes on the value 0 if the coin lands "tails" and 1 if the coin lands "heads". We could then write the probability that a coin lands "heads" as: $\Pr(X = 1)$.

If the coin was fair (as described previously), then we could conclude that $\Pr(X = 1) = 0.5$. For that same coin we would also know that $\Pr(X = 0) = 0.5$, since there are only two possible outcomes (either $X = 0$ or $X = 1$) and if we sum the probability of all possible outcomes of the coin flip, then the sum must be 1. This point is a property of probability generally—namely, if you sum the probability of all possible outcomes of a variable, then that sum must be 1 since one of the possible outcomes must occur.

We can also consider cases involving multiple variables when computing a probability. For example, say we have two coins (both fair), where the outcome of the coin #1 is represented by variable X and the outcome of the coin #2 is defined by variable Y. Now consider a process for flipping the two coins as follows:

- First, flip coin #1
- If $X = 1$ (i.e., coin #1 came up "heads"), then flip coin #2 to determine its outcome.
- However, if $X = 0$ (i.e., coin # comes up "tails"), then we intentionally set coin #2 "tails" up (i.e., we force $Y = 0$).

Given the process for flipping the two coins described above, we could ask a question like what is the probability that both coins come up "heads", or more formally, what is the value of:

$\Pr(X = 1, Y = 1)$

The notation "X = 1, Y = 1" means "X = 1 and Y = 1". This probability is 0.25, since there is a 50% chance of coin #1 coming up "heads" (i.e., X = 1) and then we would flip coin #2, which would have a 50% chance of coming up "heads". Multiplying the probability of those two cases yields: 0.5 * 0.5 = 0.25, which is the final result.

In this set-up, we could also ask: what is the probability that both coins come up tails, or more formally, what is the value of:

$\Pr(X = 0, Y = 0)$

Here the answer is 0.5. There is a 50% chance that coin #1 comes up "tails" (i.e., X = 0) and if it does then we are guaranteed (100% probability) that coin #2 will be tails as well (i.e., Y = 0). Again, multiplication (0.5 * 1.0) gives us the final result of 0.5.

To test your understanding, convince yourself that in this set-up $\Pr(X = 0, Y = 1) = 0$.

Conditional Probability

When computing probabilities, we often encounter situations where we would like to determine the probability of something happening given that something else has already occurred. Formally, this concept is known as *conditioning*, and gives rise to the notion of *conditional probability*. We write a conditional probability using a bar character '|' to separate the variables we are computing the probability of (on the left of the bar) and those whose value we already know (on the right of the bar). For example, we could write $\Pr(Y = 1 \mid X = 1)$, which would denote the probability that Y = 1 *given* that we know X = 1.

To illustrate this point more concretely, consider the set-up with two coins described above. Here, we could ask, what is the probability of that the outcome of coin #2 is "tails" (Y = 0) given that we know the outcome of coin #1 is "heads" (X = 1). Formally, we would write that as:

$\Pr(Y = 0 \mid X = 1)$.

The answer here is 0.5, since if we know that coin #1 came up "heads" (X = 1) then we simply flip coin #2 to determine its outcome and it has a 50% chance of landing "tails" (Y = 0). Note that this is a different value than if we were asked to compute $\Pr(Y = 0 \mid X = 0)$, or the probability that coin #2 would be "tails" (Y = 0) if we know that coin #1 landed "tails" (X = 0). Here $\Pr(Y = 0 \mid X = 0) = 1$, since in the case where we know coin #1 was "tails" (X = 0), then coin #2 would have to be "tails" (Y = 0) as well.

Note that conditioning on some variables does not always give us information about other variables. For example, say we had a set-up where we flipped coin #1 and then flipped coin #2 (regardless of the outcome of coin #1). Here, knowing the outcome of coin #1 (i.e., knowing that variable X had value *x*, where *x* is 0 or 1) gives us no information about the value of the variable Y (which we could denote with *y*). More formally, we would write:

$\Pr(Y = y \mid X = x) = P(Y = y)$

The statement above says that the probability that the variable Y takes on the value *y* is the same whether we know the value of the variable X or not. Intuitively, this means that the variable X gives us no information about the variable Y. When this property holds, we call variables X and Y ***independent***.

You will see the notion of independence appear in some formal definitions of fairness criteria we will examine in class. For example, the definition of *Classification Parity* is written as:

$$\Pr(d(X) = 1 \mid X_p) = \Pr(d(X) = 1),$$

where:

- X is a variable that represents a set (vector) of variables that are measured for an individual (such as the person's age, gender, race, prior criminal convictions, etc.),
- the function $d(X)$ represents some decision that is made about the individual represented by X (for example, $d(X) = 0$ may mean that individual X is released prior to trial and $d(X) = 1$ means the individual X is detained prior to trial),
- the variable X_p refers to the set of particular "protected" attributed of individual X (such as their race or gender).

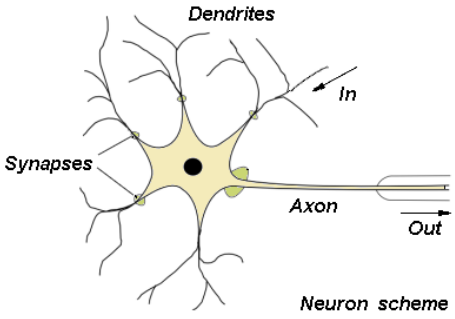
So the statement $\Pr(d(X) = 1 \mid X_p) = \Pr(d(X) = 1)$ is formally saying that the probability that a particular decision (determined by function *d*) is made about individual X is the same whether we know the protected attributes of individual X or not. In the case of criminal recidivism, it would mean that the probability individual X is released or detained prior to trial is the same whether we knew, say, their race or not. Equivalently, that would mean that the probability that individual X is released or detained prior to trial is *independent* of their race (or other protected attributes).

Machine Learning

Machine learning is a sub-discipline of Artificial Intelligence that addresses the problem of how computers can learn from data. Such learning takes many different forms (as mentioned previously, there are several classes at Stanford, such as CS229, devoted to the topic). Here we'll specifically deal with a particular method for learning, known as the Perceptron algorithm.

A Bit of History

In their pioneering work in the 1940's, Warren McCulloch and Walter Pitts posited that an "artificial" neuron could be created as a mathematical function. Drawing on work in neuroscience, McCulloch and Pitts wanted to simulate the function of a neuron (shown below).



Real neurons receive signals (called synapses) from other neurons. In a simplified view, when a neuron receives enough stimulation via synapses, it "fires" by producing an output from its axon.

The McCulloch-Pitts artificial neuron is this modeled by having a number of input variables (call them X_1 to X_n), which each have value 0 or 1. These variables represent whether there is a synapse present on each of the inputs to the neuron. The neuron then had a "threshold" value, and if the number of input variables with value 1 is greater than the threshold, then the neuron outputs a 1, otherwise it output a 0 (indicating if the neuron "fired" or not).

Given this description, it's straightforward to see that the McCulloch-Pitts artificial neuron is simply implementing a function from the inputs to the outputs. This model for an artificial neuron was further refined to be more expressive by allowing the inputs to have different weights. In the refined formulation, the artificial neuron still has *n* input variables (X_1 to X_n), which each have value 0 or 1. But, now, the neuron also stores a numeric valued (called a "weight") to correspond to each input. The weights are denoted by variables w_1 to w_n. In the weighted formulation, the artificial neuron outputs a 1 if the weighted sum of the inputs exceeds some threshold value. In other words, given the inputs X_1 to X_n, we compute the total weighted sum, S as:

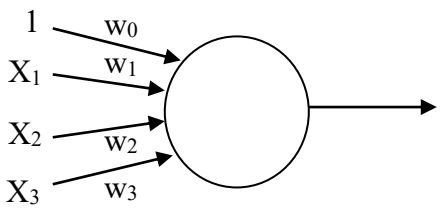
$$S = \sum_{i=1}^{n} X_i \cdot w_i$$

The artificial neuron outputs a 1 is S > T, where T is some chosen threshold value. Otherwise, the artificial neuron outputs a 0. Note that in some formulations (such as the one we'll be using in the assignment), the threshold T is simply fixed at 0. It is, however, easy to have a meaningful non-zero value for T. To make it easier to write the above rule (with non-zero values for T) mathematically, practitioners create a "dummy" input, X_0, which always has the value 1. They then create a corresponding weight, w_0, which is set to be -T. The weight w_0 is sometime called the *bias* of the neuron. (Note: the term "bias" in this context is in not the same as the notion of "biased" decisions when we talk about algorithmic decision-making). Then, we rewrite the sum above with the index variable *i* starting at 0:

$$S = \sum_{i=0}^{n} X_i \cdot w_i$$

And now the rule for the neuron firing (outputting a 1) is simply when S > 0, since the threshold has been incorporated into the weights of the neuron (as weight w_0).

Graphically, we can represent the neuron by showing the inputs as arcs (arrows) coming into a node from the left (where the weights associated with each input are shown on the respective arc), and the output is shown as an arc pointing to the right out of the node. Here is an example of a node with three inputs (note the "bias" weight w_0 represented by an extra arc with an associated input value of 1):



Note the resemblance of the picture above to that of a real neuron shown on the previous page.

The Perceptron: Learning with Artificial Neurons

In the 1950's Frank Rosenblatt built upon the McCulloch-Pitts artificial neuron by devising an algorithm that could automatically adjust the weights of the neuron to allow it to "learn" some function from inputs to output. This learning process works by presenting example input/output pairs from the function to be learned to the neuron, and then adjusting the weights in the cases where the neuron produces the wrong output. We now refer to this sort of learning as *supervised* learning, as there is a supervisor who is providing examples of what the correct answers should be.

For example, if we wanted to have an artificial neuron with three inputs model the logical function "X_1 and X_3" (i.e., only output 1 when the input variables X_1 and X_3 both had values 1), we could present examples of input/output pairs of this function to the neuron and have it adjust its weights accordingly. The input/output pairs we present could include examples such as:

X_1	X_2	X_3	Output
1	1	1	1
0	1	1	0
0	0	0	0

Note that the examples presented don't need to include every possible setting of the input variables. The set of available input/output examples is called the *training data*. Each row (example) in the data is called a "data instance" or just "instance" for short.

Perceptron Algorithm

The Perceptron algorithm starts by initially setting all the weights, w_0, ..., w_n to 0 (recall that w_0 is the *bias* weight and may not exist in some formulations of this algorithm). It then repeatedly cycles through any available training data, presenting them one at a time to the neuron for training. Each data instance represents a set of input variables (X_1, X_2, …, X_n) that are measurements for a single entity (e.g., attributes of a person, such as age, gender, etc.) and an output variable indicating the desired output (0 or 1 in this case) for that particular data instance.

Training proceeds as follows. First the input values from the chosen data instance are set as the inputs to the neuron. Then, the sum S is computed:

$$S = \sum_{i=0}^{n} X_i \cdot w_i$$

We then set the output of the neuron Q = 1 is S > 0, and set Q = 0 otherwise. If the "predicted" output Q matches the actual value of the output from the given training example, then we consider the weights of the neuron to be fine with respect to that example and don't make any changes. If, however, the value of Q does not match the output value given by the training example, then we note that an adjustment need to be made to the weights in the neuron as follows:

if Q = 1 then
    For all weights w_i (where i = 0 to n)
        Adjust value of w_i = old value of w_i – X_i
else
    For all weights w_i (where i = 0 to n)
        Adjust value of w_i = old value of w_i + X_i

Intuitively, we can think of the rule above as follows: If we predicted an output of 1 for the training example when the real output associated with it should be 0, then the weights in the neuron are *too high* as they allowed the weighted sum S to exceed 0. Thus, we should reduce those weights that contributed to the sum – which are precisely the weights associates with non-0 inputs. So, if we adjust all the weights w_i by subtracting X_i, we in fact only reduce the weights whose corresponding X_i value are non-0 (since subtracting a 0 would not affect the weight).

Similarly, if we predicted an output of 0 for the training example when the real output associated with it should be 1, then the weights in the neuron are *too low* as they did not allow the weighted sum S to exceed 0. Thus, we should increase those weights that contributed to the sum – again, these are the weights associates with non-0 inputs. So, if we adjust all the weights w_i by adding X_i, we increase the weights whose corresponding X_i value are non-0.

Note that we can simplify the weight update rule presented on the previous page with an equivalent mathematical form as:

Error = (actual value of output – Q)
For all weights w_i (where i = 0 to n)
    Adjust value of w_i = old value of w_i + (Error * X_i)

Here, Error = 0 if our prediction is correct, since Q will be the same as the actual value of the output. That case will result in no updates to the weights, since we'll just be adding 0's to every weight. If Error = -1, that means, Q = 1 and the actual value of the output was 0. That is equivalent to the Q = 1 case in our original definition of the update rule and will result in adjusting all weights w_i by subtracting X_i, since we multiple each X_i by -1 before adding it to the respective weight. Similarly, if Error = 1, that means, Q = 0 and the actual value of the output was 1. That is equivalent to the Q = 0 case in our original definition of the update rule and will results in adjusting all the weights w_i by adding X_i.

The set of weights in the neuron are often referred to as the ***model*** since the weights are all that is needed for defining the function that maps a set of input variables (X_1, X_2, …, X_n) to an output (0 or 1).

Updating the Weights Using Batch Training

Adjusting the weights in the model after processing each training data instance can potentially lead to an unstable result as the final model (set of weights) is heavily impacted by the order in which the training data is processed (i.e., the last few training data instances can cause weights to change in a way that would not lead to good results on data processed earlier). One way to mitigate this instability is just to record the change that we *should* have made to the weights after processing each training data instance (without actually updating the weights then). We call this change that we should have made to the weights the "difference" (which can be thought of as a discrete version of the derivative or gradient, for those who enjoy calculus). After running through all of the training data, we average all of the "differences" from each of the training instances together. This "average difference" is then used to actually change the weights in the model. Updating the weights using this "average difference" is known as *batch* updating, since updates to the weights are made as a result averaging all the differences from the whole "batch" of training data rather than one data instance at a time.

We generally make multiple passes through the training data, computing an average difference after each pass of the data and using that average difference to update the weights. Each such

pass through the data is referred to as an "epoch" of training. The number of epochs to use during training is a question that has many different answers (most of which are beyond our current scope). For simplicity, we just set the number of epochs to a reasonably large constant (e.g., 2,000) to make sure we have sufficiently many opportunities to make updates to the model weights.

Pocket Algorithm

A simple mechanism to make the Perceptron algorithm even more effective and stable is known as the Pocket algorithm, which was proposed by Stephen Gallant in 1990. The basic idea is that while training the Perceptron algorithm, on each epoch (pass through all the training data), the algorithm computes how many prediction errors were made on that pass through the data. If the number of errors made were lower than in any prior pass over the data, the current set of weights (model) is saved (i.e., the weights are "put in our pocket", which is where the name of the algorithm comes from). We continue to train the model as described in the previous section, but on any epoch of training where we discover a set of weights that performed better (lower number of prediction errors) than the previous model in our pocket, we put the newly discovered model (set of weights) with lower error in our pocket. After training is completed—we've run through the training data for a certain number of epochs—the algorithm returns the model (set of weights) in its pocket. That model represents the set of weights that resulted in the lowest error over all of the training data that was discovered during all training epochs.

Algorithm Pseudocode

Putting all of the concepts described above (Perceptron algorithm, batch updating, and the Pocket algorithm), on the next pages we provide the pseudocode for the full learning algorithm (one with Java-like syntax and the other with Python-like syntax, though both describe the same algorithm). Note that in the pseudocode shown below we do not use the "bias" term (so there is no weight w_0), although that could easily be added if desired. In the pseudocode, we index the array/list of weights and data instances starting with index 1 (to match the exposition above). In actual code, indexes in arrays/lists are likely to start at 0.

Pseudocode (Java-like syntax) for Batch Perceptron Pocket Algorithm

```
// Returns an array of the weights for a trained Perceptron model after training
// on the data passed in as a parameter

double[] BatchPerceptronPocket(data) {

  Initialize: w_i = 0 for all 1 ≤ i ≤ n

  maxCorrect = 0

  // "epochs" = number of passes over training data during learning
  for (j = 0; j < epochs; j++) {
     correct = 0                          // count correct predictions on each epoch

     Initialize: difference[i] = 0 for all 1 ≤ i ≤ n

     // Compute "average difference" to (eventually) update weights
     for each training instance (<x_1, x_2, …, x_n>, y) in data {

        sum = sum_{i=1}^{n} x_i · w_i

        if (sum > 0) {
           prediction = 1
        } else {
           prediction = 0
        }
                                          // Note: y is actual output for data instance
        error = y - prediction            // if (y != prediction), error is -1 or 1

        if (error == 0) {                 // if (error == 0), then prediction was correct
           correct++
        } else {                          // incorrect prediction, so update difference
           for (i = 1; i <= n; i++) {
              difference[i] += error * x_i     // Note: x_i is i-th input variable
           }
        }
     }

     // If made more correct predictions on this epoch, should store weights in "pocket"
     if (correct > maxCorrect) {
        pocket[i] = w_i for all 1 ≤ i ≤ n
        maxCorrect = correct
     }

     // Add average difference to the weights w_i for all 1 ≤ i ≤ n
     // Note: learningRate is just a constant value
     w_i += learningRate * (difference[i]/(number of training instances))
  }

  return (pocket)                         // weights in "pocket" represent final model
}
```

Pseudocode (Python-like syntax) for Batch Perceptron Pocket Algorithm

```
# Returns a list of the weights for a trained Perceptron model after training
# on the data passed in as a parameter

def batch_perceptron_pocket(data) {

  Initialize list: w_i = 0 for all 1 ≤ i ≤ n

  max_correct = 0

  # "epochs" = number of passes over training data during learning
  for j in range(epochs):
     correct = 0                          # count correct predictions on each epoch

     Initialize list: difference[i] = 0 for all 1 ≤ i ≤ n

     # Compute "average difference" to (eventually) update weights
     for each training instance (<x_1, x_2, …, x_n>, y) in data:

        sum = \sum_{i=1}^{n} x_i \cdot w_i

        if sum > 0:
           prediction = 1
        else:
           prediction = 0

                                          # Note: y is actual output for data instance
        error = y - prediction            # if (y != prediction), error is -1 or 1

        if error == 0:                    # if error == 0, then prediction was correct
           correct += 1
        else:                             # incorrect prediction, so update difference
           for i in range(1, n + 1):          # i counts from 1 to n
              difference[i] += error * x_i    # Note: x_i is i-th input variable
     # for loop over training instances ends here

     # If made more correct predictions on this epoch, should store weights in "pocket"
     if correct > max_correct:
        pocket[i] = w_i for all 1 ≤ i ≤ n
        max_correct = correct

     # Add average difference to the weights w_i for all 1 ≤ i ≤ n
     # Note: learningRate is just a constant value
     w_i += learningRate * (difference[i]/(number of training instances))

  # for loop over variable j ends here

  return pocket                           # weights in "pocket" represent final model
}
```