# DS3 Lecture: Classification and the Perceptron Algorithm

Jenn Wortman Vaughan

July 2, 2015

## 1 Binary Classification

Classification is the problem of predicting the category that an observation belongs to (called its *label*) based on *training data* that consists of a set of other observations and their labels. For example, we might have a set of emails labeled as "spam" or "not spam" that we'd like to use to come up with a way of predicting whether a new email we've never seen before is spam or not. Other examples of classification problems include handwriting recognition (where the observation is a handwritten character and the label is the real character it represents), face recognition (where the observation is an image of a face and the label is the name of the person pictured in the image), and medical diagnosis (where the observation is a patient's medical history and the label is whether or not they have a particular disease). Today we'll focus on the most simple case in which there are only two possible labels. This is referred to as *binary classification*.

Let's look at one of these examples in more detail. Suppose we would like to classify emails as spam or not spam. The first thing we need to do to turn this problem into something that can be solved algorithmically is to figure out how to represent our data (in this case, the email messages). This is typically done by representing each observation (that is, each individual email) as a vector of *features*. Designing features is a bit of an art and often requires domain-specific knowledge (for example, when building a classifier for medical diagnosis, it makes sense to consult a doctor who has intuition for which parts of a patient's history are likely to be important). For spam filtering, we might have one feature that is 1 if the phrase "Microsoft" appears in the email and 0 otherwise, another feature that is 1 if the sender is in our address book and 0 otherwise, and so on. Along with these feature vectors, we have the binary label for each email which tells us whether or not the email is spam.

The next thing we need to do is narrow our search to some reasonable set of prediction rules. For example, we might try to find a rule that predicts labels according to the value of a *disjunction* of some subset of the features (predict spam if and only if the sender is not known or "click here" appears in the message) or one that predicts labels according to a *threshold* function (predict spam if and only if "Jenn" in email + "Microsoft" in email + sender known $< 2$).

Finally, we need to design an algorithm to choose a prediction rule from the specified set based on the training data. Our hope is that the rule that we choose will generalize, that is, that it will perform well on data we haven't seen yet too. We can check this by examining the performance of the rule on data that we haven't seen before (our *test data*).

## 2 Linear Classifiers

Today we will discuss the Perceptron algorithm, which outputs a *linear classifier*. A linear classifier can be viewed as a threshold function with an assigned weight on every feature. Suppose we've decided to represent each observation as a vector $\mathbf{x}$ of $d$ features, which each component $x_i \in \{-1, 1\}$. A linear classifier is specified by a weight vector $\mathbf{w}$, also of length $d$. An observation $\mathbf{x}$ is classified as a positive

instance (label 1) if and only if

$$\sum_{i=1}^{d} w_i x_i \geq 0.$$

Intuitively, we can think about the sign of $w_i$ telling us whether the $i$th feature is more highly correlated with positive or negative observations, and $|w_i|$ telling us how important the $i$th feature is.

Let's think about a simple example. Suppose that $d = 2$, so we have only two features. For spam filtering, these may correspond to whether or not the sender is in the user's address book and whether or not the "click here" appears in the email. Suppose we have a linear classifier with $w_1 = 2$ and $w_2 = -1$. Our classifier then predicts positive if and only if

$$2x_1 - x_2 \geq 0.$$

The classifier is dividing the space of observations into those that will be labeled positive and those that will be labeled negative. What does this division look like? Well, let's think first about those $\mathbf{x}$ for which $2x_1 - x_2 = 0$. These $\mathbf{x}$ form a line. The linear classifier will classify all $\mathbf{x}$ on one side of this line as positive and all $\mathbf{x}$ on the other side as negative. This idea extends to higher dimensions $d$ too. In general, $\mathbf{w}$ will define a *hyperplane* that separates positive observations from negative observations.

Notice that no matter which $\mathbf{w}$ we've chosen, the point $\mathbf{x} = (0, 0)$ will always be labeled as positive. That is, the separating line always goes through the origin. To fix this, we can add an extra "dummy" feature $x_3$ to each observation and always set $x_3 = 1$. Now our classifier will predict positive if

$$w_1 x_1 + w_2 x_2 \geq -w_3$$

which allows us to "shift" the separator.

## 3 The Perceptron Algorithm

So, how do we figure out which weight vector $\mathbf{w}$ to use? There are many different algorithms for learning linear classifiers. The one that we'll focus on today is called the *Perceptron* algorithm. The most basic version of the Perceptron algorithm is shown below. Here we assume that each feature $x_i$ is either 1 or $-1$ and each label $y$ is also 1 or $-1$. We use $\hat{y}$ to denote the Perceptron's current guess for what the label $y$ should be.

PERCEPTRON ALGORITHM
Initialize $\mathbf{w} = \mathbf{0}$
For each training instance $(\mathbf{x}, y)$

- If $\mathbf{w} \cdot \mathbf{x} \geq 0$, $\hat{y} = +1$, else $\hat{y} = -1$
- If $y \neq \hat{y}$, update $\mathbf{w} \leftarrow \mathbf{w} + y\mathbf{x}$

Output $\mathbf{w}$

Let's get a little intuition for what's going on here. There are three cases to consider for each new instance $(\mathbf{x}, y)$:

- The predicted label $\hat{y}$ is equal to the real label $y$. In this case, the algorithm has already done the right thing and there is no need to update the weights.

- The predicted label $\hat{y}$ is negative but the true label is positive. This means that $\mathbf{w} \cdot \mathbf{x}$ was lower than it should have been. We would like to update $\mathbf{w}$ in such a way that this quantity increases so that we make the right prediction next time we see an observation like $\mathbf{x}$. We can verify that this is achieved since

$$(\mathbf{w} + y\mathbf{x}) \cdot \mathbf{x} = (\mathbf{w} + \mathbf{x}) \cdot \mathbf{x} = \sum_{i=1}^{d} (w_i + x_i)x_i = \sum_{i=1}^{d} w_i x_i + \sum_{i=1}^{d} (x_i)^2 \geq \sum_{i=1}^{d} w_i x_i = \mathbf{w} \cdot \mathbf{x}.$$

- The predicted label $\hat{y}$ is positive but the true label is negative. This means that $\mathbf{w} \cdot \mathbf{x}$ was higher than it should have been. We would like to update $\mathbf{w}$ in such a way that this quantity decreases so that we make the right prediction next time we see an observation like $\mathbf{x}$. Verify for yourself that this is achieved.

The algorithm is actually doing something a bit more sophisticated than just making sure the weights move in the right direction. It is closely related to gradient descent, though we won't get into the details.

In the version of the algorithm presented above, each training instance is examined only once. In some cases, better results can be obtained by looping through the training data multiple times. The Perceptron will continue to improve as it makes mistakes.

# 4 A Few Notes

In addition to being easy to implement, the Perceptron has some nice theoretical properties. For example, it is possible to prove that if the training data is "separable"—that is, if there exists some linear classifier that would perfectly classify the data—then the Perceptron will only make a bounded number of mistakes no matter how many times it loops through the training data. Since the weight vector is only updated when a mistake is made, this means the algorithm will converge. Unfortunately, when the data is not linearly separable, it will not converge.

The Perceptron algorithm can also be run "online." That is, instead of using a fixed set of training instances, we can use the Perceptron to label examples as they arrive and update the weights as mistakes are made. For example, we could use the Perceptron to label arriving email for a user, and update it when the user manually corrects a mistake.

# 5 Your Assignment

Your task is to implement the Perceptron algorithm in Python and run it on some data sets.

Everyone will start with the same data set in which the observations represent (very low resolution) images of handwritten digits, and the categories are either "three" or "five." These images and their labels are stored in two input files: `35_TrainingData.txt` and `35_TestData.txt`. The first contains 1400 labeled images for training your Perceptron, and the second contains 1400 labeled images for testing how it generalizes. Images are 8 by 8 pixels in size, and each pixel is either black or white. Input files are made of lines of the form

$$\text{label: } x_1\ x_2\ x_3\ x_4 \ldots x_{64}$$

where "label" is one of "three" or "five" and each $x_i$ is either 1 or -1. The vector $\mathbf{x}$ encodes an image when we line up the $x_i$ values in a grid:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ |
| $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ | $x_{16}$ |
| ... | | | | | | | |
| $x_{57}$ | $x_{58}$ | $x_{59}$ | $x_{60}$ | $x_{61}$ | $x_{62}$ | $x_{63}$ | $x_{64}$ |

and color in each cell of the grid either black or white corresponding to the feature value.

Before you work with this data, you will want to rewrite it into a more generic form, with the labels "three" and "five" replaced by 1 and $-1$. This will allow you to write the Perceptron algorithm in a way that is more easily reusable on other data sets with different sets of labels.

Your task can be broken down into the following steps:

1. Obtain the handwritten digit data from github and write a Python script to reformat it into a more generic input format. For example, your script may simply rename the labels. However, you are free to modify the data files in other ways if it will help you in Step 2.

2. Implement the Perceptron algorithm in Python. Your script should take as input a (preprocessed) training data file, a (preprocessed) test data file, and the number of times to loop through the training data. It should train a Perceptron on the training data, looping through the data the specified number of times. It should then output the training error (fraction of instances labeled incorrectly in the training data) and test error (fraction of instances labeled incorrectly in the test data) calculated using the final weight vector that is learned. Try running your algorithm with different values of the data looping parameter.

3. Augment your program so that it automatically adds a "dummy" feature with value 1 to each observation. How does this change the training error and test error?

4. Find another data set, break it into training data and test data, and run your Perceptron algorithm on it. You are free to use any data set you like. You are encouraged to find (or create!) your own, but if you need some inspiration, you're welcome to choose one of the data sets available at `http://archive.ics.uci.edu/ml/datasets.html`. If the data set you find has more than two natural categories (label values), just try to distinguish two of them. You will probably need a new preprocessing script.

5. (Optional) If you have extra time, examine how the training and test error change as a function of the number of observations in the training set. Limit your algorithm to using the first $n$ training instances for different values of $n$ and see what happens. Plot the training and test error as a function of $n$. (Training error should be calculated only on the $n$ instances that were used.)

6. (Optional) Try adding additional features to your data and see how the training error and test error change. For example, using the handwritten digit data, you could add one additional feature for each row of pixels that is 1 if at least half of the pixels are black and -1 otherwise. Make up additional features of your own and see what happens.