Clarification of the pseudocode in the Pegasos paper

Richard Johansson

1 Introduction

In this document, we will clarify some of the details about the Pegasos algorithm (Shalev-Shwartz et al., 2011) for training a linear SVM. The pseudocode of the Pegasos algorithm itself is given in Algorithm 1. This corresponds to Figure 1 in the Pegasos paper, except that the notation has been changed to make things clearer.

Algorithm 1 The Pegasos algorithm.

Inputs: a list of example feature vectors $\boldsymbol{X}$
a list of outputs Y
regularization parameter λ
the number of steps T
$\boldsymbol{w} = (0, \ldots, 0)$
for t in $[1, \ldots, T]$
select randomly a training instance $\boldsymbol{x}_i$, with a corresponding output label y_i
$\eta = \frac{1}{\lambda \cdot t}$
score = $\boldsymbol{w} \cdot \boldsymbol{x}_i$
if $y \cdot \text{score} < 1$
$\boldsymbol{w} = (1 - \eta \cdot \lambda) \cdot \boldsymbol{w} + (\eta \cdot y_i) \cdot \boldsymbol{x}_i$
else
$\boldsymbol{w} = (1 - \eta \cdot \lambda) \cdot \boldsymbol{w}$
the end result is $\boldsymbol{w}$

Here are some practical hints and clarifications of the details:

- The optional projection step has been left out (the line in square brackets in the paper).
- Following scikit-learn conventions, we're using $\boldsymbol{X}$ and Y to denote the training examples and their corresponding outputs, instead of the S used in the paper.
- In the notation, we distinguish vectors ($\boldsymbol{w}$, $\boldsymbol{x}_i$) from numbers (for instance y_i, λ, η) by writing the vector names in a bold font.
- In the paper, several of the variables have an index t, for instance the weight vector $\boldsymbol{w}_t$. That is, there is a separate "version" of the weight vector for each step in the algorithm. This is just a conventional notation and doesn't matter in practice, so we've removed the index t from the variables in the pseudocode here.
- As usual, the outputs (in the list Y) are coded as +1 for positive examples and -1 for negative examples.
- Select a random training instance either by generating a random number for its position in the training set using `random.randint`, or by just picking directly from a list using `random.choice`).
- $\boldsymbol{x}_i$ is the feature vector at position i in $\boldsymbol{X}$, and y_i the output (+1 or -1) at position i in Y.
- The number η (Greek letter *eta*) is the step length in gradient descent.

- As we have discussed previously, the multiplication dot ($\cdot$) is used ambiguously. Be careful so that you understand the meaning of each of the dots in the pseudocode.
- The regularization parameter λ (Greek letter *lambda*) is not related to the keyword `lambda` in Python (which we use for anonymous functions). However, it's an unfortunate choice of parameter name, since the Python keyword `lambda` prevents us from using that word as a variable name.

2 Derivation of the algorithm in the pseudocode

In this section, we will see where Algorithm 1 comes from: how it is derived from the stochastic gradient descent applied to the SVM objective function. As we discussed in the lecture, and as stated in Equation 1 in the paper, in the SVM the weight vector $\boldsymbol{w}$ is defined as the vector that minimizes the following *objective function*:

$$f(\boldsymbol{w}, \boldsymbol{X}, Y) = \sum_i \text{Loss}(\boldsymbol{w}, \boldsymbol{x}_i, y_i) + \lambda \cdot \|\boldsymbol{w}\|^2$$

For the SVM, Loss is the *hinge loss function*:

$$\text{Loss}(\boldsymbol{w}, \boldsymbol{x}_i, y_i) = \max(0, 1 - y_i \cdot (\boldsymbol{w} \cdot \boldsymbol{x}_i))$$

The hinge loss function can be written more explicitly in this way:

$$\text{Loss}(\boldsymbol{w}, \boldsymbol{x}_i, y_i) = \begin{cases} 1 - y_i \cdot (\boldsymbol{w} \cdot \boldsymbol{x}_i) & \text{if } y_i \cdot (\boldsymbol{w} \cdot \boldsymbol{x}_i) < 1 \\ 0 & \text{otherwise} \end{cases}$$

What Pegasos does is to apply an optimization algorithm to find the $\boldsymbol{w}$ that minimizes the objective function f. As we saw in the lecture, *gradient descent* can be used to minimize a function. For efficiency reasons, we use a simplified version of this algorithm, *stochastic gradient descent* (SGD), where we consider just a single example at a time. The pseudocode of the general SGD is shown in Algorithm 2.

Algorithm 2 Stochastic gradient descent with a fixed number of steps.

Inputs: a list of example feature vectors $\boldsymbol{X}$
a list of corresponding outputs Y
the number of steps T
$\boldsymbol{w} = (0, \ldots, 0)$
for t in $[1, \ldots, T]$
  select randomly a training instance $\boldsymbol{x}_i$, with a corresponding output label y_i
  determine a step length η
  compute the gradient $\nabla(f)$ of the objective function $f(\boldsymbol{w}, \boldsymbol{x}_i, y_i)$
  $\boldsymbol{w} = \boldsymbol{w} - \eta \cdot \nabla(f)$
the end result is $\boldsymbol{w}$

As we saw in the lecture on optimization, gradient descent algorithm have some problems finding the minimum if the step length η is not set properly. To avoid this difficulty, Pegasos uses a variable step length:

$$\eta = \frac{1}{\lambda \cdot t}$$

Since we compute the step length by dividing by t, it will gradually become smaller and smaller. The purpose of this is to avoid the problems we saw in the lecture, where we "bounce around" as we get close to the optimum.

Then, how do we compute the gradient $\nabla(f)$ of the SVM objective function? Since we're considering just one single example, we compute the gradient with respect to just $\boldsymbol{x}_i$ and y_i. We won't go into the details about how to compute the gradient, but it can be shown that it is:

$$\nabla(f) = \lambda \cdot \boldsymbol{w} + \nabla(\text{Loss})$$

and the gradient of the hinge loss is

$$\nabla(\text{Loss}) = \begin{cases} -y_i \cdot \boldsymbol{x}_i & \text{if } y_i \cdot (\boldsymbol{w} \cdot \boldsymbol{x}_i) < 1 \\ (0, \ldots, 0) & \text{otherwise} \end{cases}$$

(To get an intuition of where this comes from: recall from the lecture that the gradient describes the *slope* of a function. And take a look at the plot of the hinge loss in the lecture: the function is falling at the left of the "hinge", and completely flat at the right.)

Strictly speaking, ∇ is not a gradient, but a *subgradient*: this is because of the "abrupt" shape of the hinge loss function. This doesn't have any practical consequences.

Now we have all the missing pieces to explain Algorithm 1. If we plug the gradient of the SVM objective function ($\nabla(f)$) into the update step ($\boldsymbol{w} = \boldsymbol{w} - \eta \cdot \nabla(f)$) in SGD, we get

$$\boldsymbol{w} = (1 - \eta \cdot \lambda) \cdot \boldsymbol{w} + \begin{cases} (\eta \cdot y_i) \cdot \boldsymbol{x}_i & \text{if } y_i \cdot (\boldsymbol{w} \cdot \boldsymbol{x}_i) < 1 \\ (0, \ldots, 0) & \text{otherwise} \end{cases}$$

3 Changing from SVM to logistic regression

While the SVM is based on the hinge loss, the logistic regression model instead uses a different loss function called the *log loss*:

$$\text{Loss}(\boldsymbol{w}, \boldsymbol{x}_i, y_i) = \log(1 + \exp(-y_i \cdot (\boldsymbol{w} \cdot \boldsymbol{x}_i)))$$

In the table on page 15 in the paper, we can see the gradient of the log loss. (This is actually a real gradient and not just a subgradient, since the log loss has a smooth shape unlike the hinge loss.)

$$\nabla(\text{Loss}) = -\frac{y_i}{1 + \exp(y_i \cdot (\boldsymbol{w} \cdot \boldsymbol{x}_i))} \cdot \boldsymbol{x}_i$$

When you use this in your code, please don't miss the following small details:

- The minus sign will be turned into a plus when you plug the gradient into the SGD algorithm, because it is canceled by the minus in the update step.
- When you code the gradient in Python, the function `math.exp` will crash because of an overflow error if its input is too large. We can avoid this problem by using `numpy.exp` instead. (It will give a warning you can ignore.) Alternatively, make sure you don't pass too large values to `math.exp`.
- We don't need the two separate cases that we had for the hinge loss.

4 Speeding up the vector scaling operation (optional)

If you changed your code to use sparse vectors instead of dense vectors, you probably saw a speed improvement if you're using the full feature set. However, there is still one part that can be made more efficient. At each step of the algorithm, we shrink the weight vector a bit.

$$\boldsymbol{w} = (1 - \eta \cdot \lambda) \cdot \boldsymbol{w}$$

If we are using many features and $\boldsymbol{w}$ is high-dimensional, this will be a bit slow because we need to go through all dimensions, and we need to do this for all the T steps! Section 2.4 in the paper describes (a bit tersely) a little trick that we can use to reduce the computation time.

The idea is that we define a scaling factor a that we use to aggregate all the scaling operations that we carry out: instead of rescaling the whole vector $\boldsymbol{w}$, we just change a. We initialize a to 1, and then we replace the vector scaling step above with the following:

$$a = (1 - \eta \cdot \lambda) \cdot a$$

We then need to change the other steps a bit, so that we take the scaling factor a into account. First, we change the dot product between the weight vector and the feature vector:

$$\text{score} = a \cdot (\boldsymbol{w} \cdot \boldsymbol{x}_i)$$

Then, we change the step where we add the feature vector to the weight vector, we need to "compensate" for the fact that we eventually will scale $\boldsymbol{w}$ by a:

$$\boldsymbol{w} += \frac{\eta \cdot y_i}{a} \cdot \boldsymbol{x}_i$$

Finally, when the algorithm is finishing, we carry out the scaling operation that we have postponed:

$$\boldsymbol{w} = a \cdot \boldsymbol{w}$$

References

Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, and Andrew Cotter. 2011. Pegasos: Primal estimated sub-gradient solver for SVM. *Mathematical Programming, Series B*, 127(1):3–30.