Clarification of the pseudocode in the Pegasos paper

Richard Johansson

1 Introduction

In this assignment, we will implement code to train linear SVC and logistic regression classifiers. Recall that these two classifiers both are based on finding the "best" weight vector $\boldsymbol{w}$, as defined by minimizing the following *objective function*:

$$f(\boldsymbol{w}) = \frac{1}{N}\sum_i \text{Loss}(\boldsymbol{w}, \boldsymbol{x}_i, y_i) + \frac{\lambda}{2} \cdot \|\boldsymbol{w}\|^2$$

The difference between SVC and LR is that in the SVC, Loss is the *hinge loss*, and in LR it is the *log loss*.

To find the best $\boldsymbol{w}$, we will use an algorithm called *Pegasos* (Shalev-Shwartz et al., 2011). The Pegasos algorithm is actually just the *Stochastic Gradient Descent* algorithm applied to the SVC and LR objective functions, plus a cleverly selected approach to gradually decreasing the step length (which we will see later). In this document, we will clarify some of the details about the Pegasos algorithm.

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 be more similar to the notation we have used in the course.

Algorithm 1 The Pegasos algorithm.

Inputs: a list of example feature vectors $\boldsymbol{X}$
a list of outputs Y
regularization parameter λ
$\boldsymbol{w} = (0, \ldots, 0)$
repeat
  select a training pair (input $\boldsymbol{x}$, output y)
  $t = t + 1$
  $\eta = \frac{1}{\lambda \cdot t}$
  score = $\boldsymbol{w} \cdot \boldsymbol{x}$
  if $y \cdot$score < 1
    $\boldsymbol{w} = (1 - \eta \cdot \lambda) \cdot \boldsymbol{w} + (\eta \cdot y) \cdot \boldsymbol{x}$
  else
    $\boldsymbol{w} = (1 - \eta \cdot \lambda) \cdot \boldsymbol{w}$

Here are some practical hints about your coding, and some comments about the notation:

- You will need to decide how you translate the lines **repeat** and **select a training pair** into practical code. You may decide to go through the training set a fixed number of times as in the perceptron code, or (as in the paper) pick a fixed number of randomly selected training pairs.
- As usual, the outputs (in the list Y) are coded as +1 for positive examples and -1 for negative examples.
- In the notation, we distinguish vectors ($\boldsymbol{w}$, $\boldsymbol{x}$) from numbers (for instance y, λ, η) by writing the vector names in a bold font.

- If you decide to pick instances randomly, you can 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`).

- The number η (Greek letter *eta*) is the step length or *learning rate* in gradient descent.

- As we have discussed previously in the course, 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.

Here are some differences in notation between our pseudocode and Fig. 1 in the Pegasos paper.

- In the Pegasos paper, the dot product is written $\langle w, x \rangle$ instead of $\boldsymbol{w} \cdot \boldsymbol{x}$

- 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.

- In the pseudocode in the paper, there is an optional step (the line in square brackets in the paper). This is not necessary and has been left out here.

- In the paper, S means the training set: we instead use $\boldsymbol{X}$ and Y, following scikit-learn. $|S|$ means the size of the training set.

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 SVC objective function. As we discussed in the lecture, and as stated in Equation 1 in the paper, in the SVC the weight vector $\boldsymbol{w}$ is defined as the vector that minimizes the following *objective function*:[1]

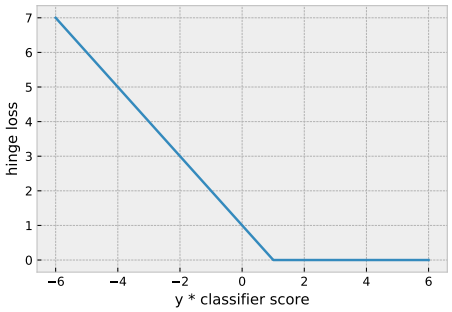
$$f(\boldsymbol{w}, \boldsymbol{X}, Y) = \frac{1}{N} \sum_i \text{Loss}(\boldsymbol{w}, \boldsymbol{x}_i, y_i) + \frac{\lambda}{2} \cdot \|\boldsymbol{w}\|^2$$

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

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

The hinge loss represents the "price" we pay for each training instance: the loss is nonzero if the instance is too close to the decision boundary, or on the wrong side of it. Here is a plot of the hinge loss as a function of $y \cdot$ the classifier score.

[1] Some formulations of the SVC objective use a parameter C instead of λ: where C is multiplied by the loss, not the regularizer. This is how the SVC works in scikit-learn, for instance.



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

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

2.1 Stochastic gradient descent

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, *stochastic gradient descent* can be used to minimize a function. The pseudocode of the general SGD is shown in Algorithm 2.

Algorithm 2 Stochastic gradient descent.

Inputs: a list of example feature vectors $\boldsymbol{X}$
a list of corresponding outputs Y
initialize the weight vector $\boldsymbol{w}$
repeat
  select a training pair $\boldsymbol{x}, y$
  determine a step length η
  compute the gradient $\nabla(f)$ of the objective function $f(\boldsymbol{w},\boldsymbol{x},y)$
  update the weight vector: $\boldsymbol{w} = \boldsymbol{w} - \eta \cdot \nabla(f)$

2.2 Setting the step length

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

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

Here, t is a "counter" that increases by one for each training instance we process. 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.

2.3 Gradient of the SVC objective

Then, how do we compute the gradient $\nabla(f)$ of the SVC objective function? Since we're considering just one single example, we compute the gradient with respect to just $\boldsymbol{x}$ and y. 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

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

(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 above: the function is falling at the left of the "hinge", and completely flat at the right.)[2]

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

if $y \cdot (\boldsymbol{w} \cdot \boldsymbol{x}) < 1$
  $\boldsymbol{w} = (1 - \eta \cdot \lambda) \cdot \boldsymbol{w} + \eta \cdot y \cdot \boldsymbol{x}$
else
  $\boldsymbol{w} = (1 - \eta \cdot \lambda) \cdot \boldsymbol{w}$

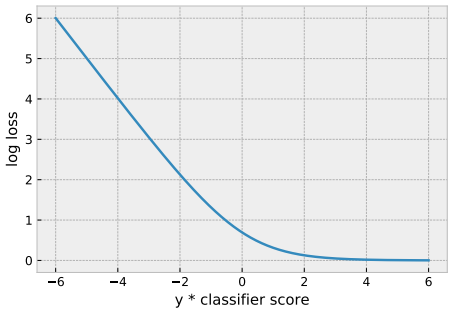
Self-check: Make sure that you understand why we arrive at Algorithm 1 if we combine Algorithm 2 with the variable step length formula and the update formula.

3 Changing from SVC to logistic regression

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

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

Here is a plot of the log loss as a function of $y \cdot$ the classifier score.



In the table on page 15 in the Pegasos paper, we can see the gradient of the log loss:

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

When you replace the hinge loss with the log loss, 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.
- We don't need the two separate cases that we had for the hinge loss.
- Don't forget that the weight vector still needs to be rescaled in each step.

[2] 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 in our case.

- The logarithm is `np.log` and the exponential is `np.exp`. Don't use Python's built-in function `math.exp` in this case, because it will crash with an overflow error. If you use `np.exp`, you'll just get a warning message that you can ignore. Alternatively, make sure you don't pass too large values to the exponential function.

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 carry out a vector rescaling operation.

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

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 every training instance! 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.