Classification

Piyush Patil

March 10, 2017

Classification is problem in a sub-domain of machine learning called *function learning*, in which one is given a set of data points (x_i, y_i) and tries to build a mapping from the x_i's to the y_i's, so as to accurately predict the y of a completely new, unseen x. The process of building the mapping is called *training*.

Conventionally, x_i and y_i are vectors in $\mathbb{R}^n$, so when approaching real-world problems it's necessary to *featurize* data; that is, to extract a set of numerical descriptors, called *features*, about a data point that are together sufficient to determine the data point. In practice, it's difficult to find a pragmatic featurization that fully determines the data, so the choice of features is very important.

For example, a common featurization of image data is a row-by-row vector of 2-D pixel data, and a common featurization of text data is the bag of words model, in which words are mapped to the frequency of their occurrence in the whole text. When trying to featurize data involving human subjects, for example in predicting if a subject has cancer, it would be extremely difficult to fully featurize a person; rather, choosing the right features, such as age, number of cigarettes smoked per day, number of family members with lung cancer, etc. can yield a very good approximation.

Classification is a special case of function learning where y_i only takes on values in a finite set; regression, another special case of function learning, is when y_i is real-valued.

1 Linear Classifiers

Consider binary classification, in which $y_i \in \{-1, 1\}$ by convention. Each x_i is a point in n-dimensional space, and belongs to one of two classes; the simplest way to classify new data points is to search for a linear separator - a hyperplane (a generalization of a separating line in the special case $n = 2$ in the plane) for which every x_i on one side belongs to one class and every x_i on the other side belongs to the other.

The data points shown above are *linearly separable* since a line can be drawn to partition the data points, but in practice, this is very rare. Let's now formally define linear separators.

(Definition) Linear separator: A function $f : \mathbb{R}^n \to \mathbb{R}$ characterized by parameters $w \in \mathbb{R}^n$, $\beta \in \mathbb{R}$ and given by

$$f(x) = w^\intercal x + \beta = \sum_{i=1}^{n} w_i x_i + \beta$$

where we classify data points as $\hat{y} = \text{sgn}\,(f(x)) \in \{-1, 1\}$. Accordingly,

(Definition) Linearly separable: *A data set $\{(x_i, y_i), 1 \leq i \leq n\}$ is **linearly separable** if there exists a linear separator f such that*

$$\text{sgn}\,(f(x_i)) = y_i, \forall i \in \{1, \cdots, n\}$$

The separating hyperplane associated with a linear separator is known as the *decision boundary* and is given by the set of points x s.t. $f(x) = 0$. We want the decision boundary to be as far from all data points as possible (without compromising accuracy), since the closer it is to a given data point, the more uncertain that point's classification is. This leads us to the idea of the *margin* of a linear separator, which is related to the distance from the decision boundary to a data point.

(Definition) Margin: *The **margin** τ of a data point x with respect to a linear classifier f is the distance from x to the decision boundary of f, and is given by*

$$\tau = -\frac{f(x)}{|w|}$$

Motivation: We can compute the distance from a data point x to the decision boundary of a linear separator $f(x) = w \cdot x + \beta$ as follows: the shortest distance is perpendicular to the decision boundary and through x. The decision boundary is a

hyperplane, and by definition of a plane, w is orthogonal to the plane; thus, to compute the distance from x to the decision boundary, we first construct a vector by starting at x and reaching to the decision boundary:

$$x + \tau \frac{w}{|w|}$$

where τ is the distance. To determine τ notice that the above vector should lie on the decision boundary, which means

$$f(x + \tau \frac{w}{|w|}) = w \cdot \left(x + \tau \frac{w}{|w|} \right) + \beta = 0$$

Solving for τ, we find the distance

$$\tau = -\frac{w \cdot x + \beta}{|w|} = -\frac{f(x)}{|w|}$$

To make the math work out easier, we often include the shifting term β as the last component of the weight vector w, and append a one to the end of any input vector x. This allows us to use the more compact form $f(x) = w^\intercal x$. Usually we want to find the hyperplane that both fits the data and maximizes margin; the hyperplane that maximizes its minimum distance to any data point. To fit the data, we want the linear classifier to satisfy the inequalities

$$f(x_i) \geq 1 \text{ if } y_i = 1,$$
$$f(x_i) \leq -1 \text{ if } y_i = -1$$

Combining the inequalities, we want our classifier to satisfy $y_i f(x_i) \geq 1$. Recall that the distance from a data point to the decision boundary was the value of the classifier at the data point divided by the magnitude of the weights used. As just discussed, the value of a classifier that fits the data is given by $y_i f(x_i)$, so the problem of maximizing the minimum margin becomes

$$\max_w \left(\min_i \frac{y_i f(x_i)}{|w|} \right) \text{ subject to } y_i f(x_i) \geq 1$$

which, due to the fact that under the constraint the minimum value is 1, reduces to

$$\max_w \left(\frac{1}{|w|} \right) = \min_{w \in \mathbb{R}^{d+1}} (|w|^2), \text{ subject to } y_i w^\intercal x_i \geq 1$$

We square the magnitude above so the minimized function is convex. Thus, we have now reduced the problem of linearly classifying data to a simple optimization problem.

2 Soft-Margin Classifiers

Notice that the above optimization problem assumes that the data is linearly separable, and moreover is very sensitive to outliers. In practice, this assumption is rarely true, and we may want to ignore outliers to some degree, allowing our model to mis-classify very rare, badly behaved points. To incorporate this idea, we introduce slack variables to the optimization problem. Essentially, we'll allow for a non-zero error rate, by relaxing the constraint $y_i f(x_i) \geq 1$, but penalize mis-classified points proportional to how poorly they are mis-classified, which we quantify as the distance of the point from the separating hyperplane.

To formalize this idea, we allow the i^{th} data point some "slack" ϵ_i, which can be thought of as the maximum margin of error we're willing to accept for that data point, and relax the classification constraint to $y_i f(x_i) \geq 1 - \epsilon_i$. Of course, we still want to minimize the amount of slack we "use", since ideally we'd allocate as small a margin of error to each data point as possible, so as to find the best classifier. Thus, although we relax constraints by adding slack, we penalize the magnitude of the slack used in classifying any given data point. Our new soft-margin optimization becomes

$$\min_{w \in \mathbb{R}^{d+1}, \epsilon_i \in \mathbb{R}} \left(|w|^2 + C \sum_{i=1}^{n} \epsilon_i \right), \text{ subject to } y_i w^\intercal x_i \geq 1 - \epsilon_i$$

for some scaling factor C. This is equivalent to

$$\min_{w \in \mathbb{R}^{d+1}} \left(|w|^2 + C \sum_{i=1}^{n} \max(0, 1 - y_i w^\intercal x_i) \right) \text{ subject to } y_i w^\intercal x_i \geq 1 - \epsilon_i$$

The larger we make C, the more harshly we penalize mis-classification; hard-margin classifiers can be seen as solving the above optimization problem with $C = \infty$, and run the risk of overfitting. Conversely, the lower C is, the larger the margin of error is for mis-classifying data points, and thus the classifier is less sensitive to outliers but runs the risk of underfitting.

3 Perceptrons

A *perceptron* is another kind of linear classifier. While maximum-margin linear classifiers try to find an optimal separating hyperplane by reducing the problem to an optimization problem, perceptrons start off by viewing the hyperplane as completely determined by its parameters, and try to converge onto the optimal parameters, data point by data point. Every time the perceptron mis-classifies a labeled data point, it updates the parameters of the hyperplane to move a bit closer to the missed point, and every time the classification is correct, it moves in the other direction. As opposed to our previous discussion on linear classifiers, which compose a class of models known as *support vector machines*, or SVMs, perceptrons learn "online", rather than all at once with a closed form solution. From a high level, the algorithm is:

procedure PERCEPTRON(w, β)
  Pick x_i
  if $y_i \cdot (w \cdot x_i + \beta) < 0$ **then**
    $w \leftarrow w + y_i x_i$
    $\beta \leftarrow \beta + y_i$
  end if
end procedure

The intuition behind the algorithm is that if the perceptron incorrectly classified the data point, to shift the weight vector (which is orthogonal to the separating hyperplane and thus determines its orientation) slightly towards the data point, so as to correctly classify it in the future; the y_i factor ensures that the weight vector shift is in the right direction. Similarly, the bias update simply moves the hyperplane towards the point. The following theorem formalizes this intuition.

(Theorem) Convergence of perceptron algorithm: *Given linearly separable data the perceptron algorithm will terminate with a perfect linear classifier $f(x) = w \cdot x + \beta$, independent of the order of iteration through the dataset.*
Moreover, the number of updates is bounded above by

$$\frac{R^2}{\gamma^2} \text{ where } R = \max_i |x_i|, \gamma = \text{minimum margin}$$

Proof: Since the data is linearly separable, there exists some optimal weight vector w^* which produces a separating hyperplane, so that

$$|w^*| = 1 \text{ and } y_i \cdot (w^* \cdot x_i + \beta) \geq \gamma$$

We will prove that the perceptron algorithm converges to w^* within $\frac{R^2}{\gamma^2}$ iterations by defining w^k to be the weight vector in iteration k and showing that k is bounded by the above bound.
We use proof by induction on k to show that

$$k^2\gamma^2 \leq |w^{k+1}|^2 \leq kR^2$$

The base case $k = 0$ is trivially satisfied if we suppose that the perceptron algorithm initializes weights to 0, so that $w^1 = \mathbf{0}$. Suppose, as the inductive hypothesis, that

$$w^k \cdot w^* \geq (k-1)\gamma \text{ and}$$
$$|w^k|^2 \leq (k-1)R^2$$

Notice that both of the above are satisfied in the base case. If the k^{th} error is made on data point t, then we can use the first inductive hypothesis to derive a lower bound on $|w^k|$:

$$\begin{aligned} w^{k+1} \cdot w^* &= (w^k + y_t \mathbf{x_t}) \cdot w^*, \text{ by the perceptron update rule} \\ &= w^k \cdot w^* + y_t \mathbf{x_t} \cdot w^* \\ &\geq w^k \cdot w^* + \gamma, \text{ since } \gamma \text{ is the minimum margin} \\ &= (k-1)\gamma + \gamma, \text{ by the first inductive hypothesis} \\ &= k\gamma \end{aligned}$$

Therefore,

$$|w^{k+1}||w^*| \geq w^{k+1} \cdot w^*, \text{ and } |w^*| = 1 \rightarrow |w^{k+1}| \geq k\gamma \rightarrow |w^{k+1}|^2 \geq k^2\gamma^2$$

Using the second inductive hypothesis and again assuming that the k^{th} error is made on data point t, we can derive an upper bound on $|w^k|$:

$$\begin{aligned} |w^{k+1}|^2 &= |w^k + y_t \mathbf{x_t}|^2 \\ &= |w^k|^2 + 2y_t \mathbf{x_t} \cdot w^k + y_t^2 |\mathbf{x_t}|^2 \\ &\leq |w^k| + R^2, \text{ since } y_t^2 |\mathbf{x_t}|^2 = |\mathbf{x_t}|^2 \leq \max_i |x_i|^2 = R^2 \\ &= (k-1)R^2 + R^2 \\ &= kR^2 \end{aligned}$$

It follows that

$$k^2\gamma^2 \leq |w^{k+1}|^2 \leq kR^2 \rightarrow k \leq \frac{R^2}{\gamma^2}$$

which bounds the number of iterations k of the perceptron algorithm. $\square$

4 Unconstrained Optimization and Gradient Descent

Because we deal with loss functions quite a bit in machine learning, in this section we study how to efficiently minimize loss functions, with the assumption that we don't have any specific constraints to worry about (in practice, this isn't too strong an assumption since we can often incorporate our constraints directly into the function to be minimized). More generally, for $f : \mathbb{R}^n \rightarrow \mathbb{R}$ we wish to compute

$$\min_{w \in \mathbb{R}^n} f(w)$$

where we define a **minimum** $w^* \in \mathbb{R}^n$ by $\forall w \in \mathbb{R}^n : f(w^*) \leq f(w)$. We further define the notion of a **local minimum** by a $w^* \in \mathbb{R}^n$ where $\exists R \in \mathbb{R}$ s.t. $\forall w \in \mathbb{R}^n$ with $||w - w^*|| \leq R$, we have $f(w^*) \leq f(w)$. Note that we are interested in finding global minima, not local minima.

If f is differentiable, we can find the minimum by simply solving $\nabla f = 0$. In practice, we usually can't explicitly compute ∇f, since we don't have a complete description of the function. However, if we can measure the gradient at certain points, then we can iteratively converge to a local minimum. This is based on the fact that at a given point the gradient points in the direction of greatest increase, so if we keep moving away from the gradient, we'll eventually descend to a minimum. More formally,

procedure GradientDescent($f : \mathbb{R}^n \rightarrow \mathbb{R}$)
  Choose $w_0 \in \mathbb{R}^n$
  for $k \in \{1, \cdots, N\}$ **do**
    $w_{k+1} \leftarrow w_k + \alpha_k \nabla f(w_k)$
  end for
  return w_N
end procedure

where N is the number of iterations to run for and $\alpha_k \in [0, 1]$ is the learning rate, or step size, which controls how quickly we'll converge to the minimum; a larger step size will converge faster, but is also more likely to overshoot the minimum and be more inaccurate. Often the update loop is run simply until convergence rather than for a preset number of trials, where convergence is defined as $w_k = w_{k+1}$.

Often we choose our cost function to be **convex**, as convex functions are guaranteed to have only a single global minima, and gradient descent is guaranteed to converge to a global minimum. Intuitively, a convex function is one where the entire function lies beneath the line segment through any two points on the function. Formally,

(Definition) Convex function: *A function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is **convex** if for any $w_1, w_2 \in \mathbb{R}^n$, we have*

$$\forall t \in [0, 1] : f\left((1-t)w_1 + tw_2\right) \leq (1-t)f(w_1) + tf(w_2)$$

Because convex functions have the property that their gradient vanishes only at the global minimum, which is guaranteed to exist, finding a convex loss function is very useful in machine learning. Lastly, let's prove the above fact that convex functions have one global minimum.

Proof: Let $f : \mathbb{R}^n \rightarrow \mathbb{R}$ be a convex function, and let x^* be a local minimum, so that $\exists R > 0$ s.t. $\forall x \in \mathbb{R}^n : ||x - x^*|| \leq R \rightarrow f(x^*) < f(x)$. We prove that x^* is a global minimum. Assume for the sake of contradiction that $\exists x_0$ s.t. $f(x_0) < f(x^*)$. Then

$$f((1-t)x^* + tx_0) \leq (1-t)f(x^*) + tf(x_0) < (1-t)f(x^*) + tf(x^*) = f(x^*)$$

Notice that $\lim_{t \rightarrow 0}(((1-t)x^* + tx_0) - x^*) = 0$, so for sufficiently small t we have

$$||((1-t)x^* + tx_0) - x^*|| < R \rightarrow f(x^*) < f((1-t)x^* + tx_0)$$

since x^* is a local minimum. We've now shown that $f(x^*) < f((1-t)x^* + tx_0) < f(x^*)$, a contradiction. Thus, x_0 can't exist. $\square$

5 Stochastic Gradient Descent

We've now discussed the gradient descent algorithm for minimizing functions, which is used in machine learning in the context of minimizing loss functions. Loss functions are simply computable functions which quantify the error in our model when used as a predictive model for a given dataset. That is, given a dataset $\{(x_i, y_i), x_i \in \mathbb{R}^n, y_i \in \mathbb{R}\}$ of inputs and expected outputs, a loss function $L(w, (x_i, y_i))$ maps the error in a model with parameters w when predicting on the input x_i as

compared to the expected output y_i.
Typically we try to minimize the average loss over the entire dataset (though, of course, the minimum is unaffected by positive constant factors and so the $\frac{1}{n}$ factor does nothing):

$$\min_w \frac{1}{n}\sum_{i=1}^{n}(L(w,(x_i,y_i)) + R(w))$$

where R is some regularization term. In the context of support vector machines and linear classifiers with hyperplanes, which were covered in the first lecture, we were trying to solve the special case

$$\min_w \frac{C}{n}\sum_{i=1}^{n}(1 - y_i w^\intercal x_i) + ||w||_2^2$$

Regularization: As a sidenote, we briefly review regularization parameters. A regularization term is a function R of our model's parameters which tries to quantify in some sense how "complex" the model is. By imposing the additional constraint of minimizing R in addition to the average loss over the dataset, we can better ensure that the model we converge to performs well not only on the dataset, but also generalizes to unseen data. Accordingly, regularization is a very common technique to prevent overfitting, the phenomenon in which a model has, during training, essentially "memorized" the training dataset through the use of an extremely complex, highly specific model that achieves very high accuracy on the training data but which doesn't generalize to unseen data at all. Overfitting is common because in practice a given training dataset will inevitably have some amount of random noise, and so trying to fit the training data as closely as possible doesn't account for this noise and thus will not generalize well. The underlying assumption of regularization terms is that a good model that will both achieve high accuracy on the training data and generalize well to unseen data is one which is as simple as possible, accounting for random noise. Philosophically, this assumption is based on Occam's razor, the idea that simpler models with equal predictive or explanatory power as more complex models should be preferred.

Letting $f(w) := \frac{1}{n}\sum_{i=1}^{n} L(w,(x_i,y_i))$, be the average loss, let's apply gradient descent to compute the minimum over w. Differentiation is a linear operator, so note that $\nabla f(w) = \frac{1}{n}\sum_{i=1}^{n}\nabla L(w,(x_i,y_i))$. We'd need to compute this for gradient descent, which would take $O(n)$ time. It turns out that if we introduce some randomness into our algorithm and accept a slightly higher error rate, we can approximate gradient descent by computing the gradient in $O(1)$ time. We want to find some function $g(w)$ such that, in expectation, we attain the gradient of f, and is fast to compute (as opposed to ∇f). Of course, by introducing randomness this will always be merely an approximation. That is, we want to converge to a function g for which

$$\mathbb{E}(g(w)) = \nabla f(w)$$

We can then use g in our gradient descent algorithm. Here's one very natural and intuitive way of computing one such g, which is constant time:

$$g(w) = \nabla L(w,(x_i,y_i)), \text{ where } i \text{ is sampled uniformly from } \{1,\cdots,n\}$$

So we set $g(w)$ to be the gradient of L only at the point (x_i, y_i). It follows easily that $\mathbb{E}(g(w)) = \nabla f$, since

$$\mathbb{E}(g(w)) = \sum_{i=1}^{n}\Pr(i \text{ was sampled})\nabla L(w,(x_i,y_i)) = \frac{1}{n}\sum_{i=1}^{n}\nabla L(w,(x_i,y_i)) = \nabla\left(\frac{1}{n}\sum_{i=1}^{n}L(w,(x_i,y_i))\right) = \nabla f$$

If we repeat this many times in our gradient descent algorithm, on any given iteration there will be some random error in our approximation of ∇f with g, but the error will average out and decrease over time. Thus, if we used gradient descent to minimize f, we would have to first expend an $O(n)$ operation to compute ∇f followed by the weight updates. With the above method, called **stochastic gradient descent** to emphasize the use of randomness, we can approximate ∇f and update weights both at once in a single iteration, and converge to the correct ∇f as the number of iterations increases. This leads us to the following algorithm.

procedure SGD(L)
  $w_0 \leftarrow 0$
  for $k \in \{1,\cdots,N\}$ **do**
    Uniformly sample i from $\{1,\cdots,n\}$
    $w_{k+1} \leftarrow w_k - \alpha_k \nabla L(w,(x_i,y_i))$
  end for
end procedure

As with gradient descent, the above loop is often run simply until convergence, when $w_k = w_{k+1}$. This has significant speed advantages over gradient descent; to get gradient descent to converge, it may require multiple runs of gradient descent, each time having to compute ∇f, but with stochastic gradient descent, we can perform constant time updates every iteration

until converged, effectively performing as much work as a single pass of gradient descent.
We've shown that after passing through the data, we'll have a very reasonable approximation of ∇f. However, this doesn't explain why we iterate through the data by randomly sampling. It turns out that introducing randomness in the iteration process is an important way to ameliorate error due to biases in the data, which will help speed up convergence. If there are any pre-existing biases in the data, then because as we iterate through the data we maintain a partially converged model, partway through the data we may have encountered the biased portion of data and thus have a partial model that's very biased and will have a harder time converging in the future. Iterating randomly solves this problem.
Finally, we discuss some minor optimizations to the original stochastic gradient descent algorithm.

Learning rate decay: To prevent oscillating around the minimum, it can be beneficial to decrease the learning rate over time. If timed correctly, the learning rate can start to decrease right as the algorithm begins to converge, so the weights can "home in" on the minimum without overshooting it and oscillating.
One possible strategy is to use α for n steps, $\frac{\alpha}{2}$ for $2n$ steps, $\frac{\alpha}{4}$ for $4n$ steps, etc.

Momentum: In the same vein, if we simply go towards the direction of steepest descent every time, we run the risk of highly variable oscillation as well as, for non-convex functions, being stuck at local minima. To guard against this, we can borrow the idea of inertia - objects in motion tend to stay in motion - if we think of the weight vector as a particle in space moving towards a minimum, its descent to the minimum will involve not just first order changes (i.e. linear changes in the position), but also second-order changes, i.e. acceleration. When applied to gradient descent, the result is that weight vectors move towards the minimum with a rate of convergence that tends to increase the more and more the vector descends, and decreases sharply with changes in direction. This can help avoid being stuck in local minima because if the vector descends towards a minimum, it may accumulate momentum that might be sufficient to pull it through and over the minimum, allowing to potentially reach a point where another, deeper minimum can be "spotted".
To incorporate momentum, the following weight update equation is used:

$$w_{k+1} \leftarrow w_k - \alpha_k \nabla L(w_k, (x_i, y_i)) + \beta_k \cdot (w_k - w_{k-1})$$

so the amount the weight vector updates by, in the direction of steepest descent, accounts for the magnitude of the previous weight update, allowing "momentum" to accumulate slowly in the rate of convergence. $beta_k$ is simply a scaling factor for how much to factor the momentum in during the weight update.