CS 337: Artificial Intelligence

Amit Rajaraman

Last updated September 4, 2021

Contents

0 Notation **2**

1 Introduction **2**
1.1 What and Why? . . . 2
1.2 Linear Regression . . . 2
1.2.1 The method of least squares . . . 3
1.3 Probabilistic Interpretations . . . 4
1.3.1 Maximum likelihood estimation . . . 4
1.3.2 Bayesian estimation . . . 4
1.3.3 Relating probabilistic interpretations and regression . . . 5
1.4 Regularisation . . . 5
1.4.1 Regularised regression . . . 5
1.4.2 Understanding regularisation . . . 6

2 Classification **7**
2.1 Perceptron Classification . . . 7
2.1.1 The basic idea . . . 7
2.1.2 Convergence . . . 8
2.1.3 Different loss functions . . . 8

§0. Notation

Given a vector $w \in \mathbb{R}^n$, w_i denotes the ith component of w.
Given a vector w, $\|w\|_0$ is equal to the number of non-zero components $|\{i : w_i \neq 0\}|$ of w.

§1. Introduction

1.1. What and Why?

Machine learning involves learning from past experiences to perform a job better.
Broadly, the goal of the field is to make a computer emulate human pattern recognition, which is second nature to us. What do each of these words mean in the context of computer science? "Learning" and "better" usually depend heavily on the context surrounding our goals, and past experiences usually refers to a data set of some form. Broadly, there are two types of learning: *supervised* and *unsupervised*.

In the former, we have access to some **training data**, usually consisting of a set of pairs of input and output. The supervised learning algorithm then analyzes this data to produce an inferred function, which can then be used for determining what unknown inputs (not in the training data) must map to. This requires the algorithm to go from a smaller set of training data to a much broader set of inputs in a way that "makes sense". For example, given the heights and weights of a thousand people, we might be asked to determine the weight of a person of a person of some new height.
In unsupervised learning on the other hand, we do not have access to any prior information, so we must classify them in some sensible manner. For example, given a set of fruits, we may wish to classify them into various groups. If we divide on the basis of colour (alone), we might get two groups – apples and cherries in one and oranges and peaches in the other. If we divide on the basis of both colour and size, we might further divide the first of these groups into two. The computer is forced to create some compact internal representation of the features of these fruits, and from this we might even be able to generate *new* fruits.

How does all this work? It might be simpler to grasp if we dive headlong into an example.
Suppose we are given a data set $\mathcal{D} = \{\langle x_1, y_1\rangle, \ldots, \langle x_k, y_k\rangle\}$. We wish to determine a function f^* such that $f^*(x)$ is the best "predictor" of y with respect to $\mathcal{D}$.
To quantify how good a prediction is, we introduce an *error function* $\varepsilon(f, \mathcal{D})$ that reflects the discrepancy of the function with respect to $\mathcal{D}$. We also assume that our f^* is taken from some base class of functions, say $\mathcal{F}$. We then set

$$f^* = \arg\min_{f \in \mathcal{F}} \varepsilon(f, \mathcal{D}).$$

Typically, we present our function in terms of some "basis functions" $(\phi_i)_{i=1}^n$ each from the set of inputs to $\mathbb{R}$. More compactly, we have a single function ϕ from the set of inputs to $\mathbb{R}^n$ that shows all the relevant things we can glean about any input. Suppose our set of outputs is in $\mathbb{R}^k$. We then learn a function $f : \mathbb{R}^n \to \mathbb{R}^k$ and given a new input x', we predict the corresponding output as $y' = f(\phi(x'))$.
Due to this, it is often helpful to think of the input as an element of $\mathbb{R}^n$, where n is the number of basis functions.

Remark. It is important to note that even though our original input x may be in $\mathbb{R}$, we could have more than just 1 basis function. For instance, we could choose the basis functions as $\{(x \mapsto x^i) : 1 \leq i \leq 100\}$. This is especially important in the coming topic of linear regression, where linearity in the basis functions should not be mixed up with linearity in the original input.

1.2. Linear Regression

We first discuss perhaps the simplest example of machine learning. In this, the class $\mathcal{F}$ of functions we consider is merely the class of all linear functions over the basis functions. That is,

$$\mathcal{F} = \{(x \mapsto w^\top \phi(x)) : w \in \mathbb{R}^n\}.$$

Observe that if one of our basis functions is 1 (or some constant), this is equivalent to the set of all *affine* functions (linear functions plus a constant).

1.2.1. The method of least squares

Suppose our data set is $\mathcal{D} = \{\langle x_i, y_i\rangle : 1 \le i \le m\}$, where each y_i is in $\mathbb{R}$. In the method of least squares, our choice of error function is given by

$$\varepsilon(f, \mathcal{D}) = \sum_{i=1}^{m} (f(x_i) - y_i)^2.$$

This is an extremely common error function for various reasons – it is convex, non-negative, and well-behaved.
In the case of linear regression, where any function f is uniquely determined by a $w \in \mathbb{R}^n$, we can express the error in terms of this vector as

$$\varepsilon(w) = \sum_{i=1}^{m} (w^\top \phi(x_i) - y_i)^2.$$

Now, set

$$\Phi = \begin{pmatrix} \phi_1(x_1) & \cdots & \phi_n(x_1) \\ \vdots & \ddots & \vdots \\ \phi_1(x_m) & \cdots & \phi_n(x_m) \end{pmatrix} \text{ and } y = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}.$$

So,

$$\varepsilon(w) = \|\Phi w - y\|_2^2.$$

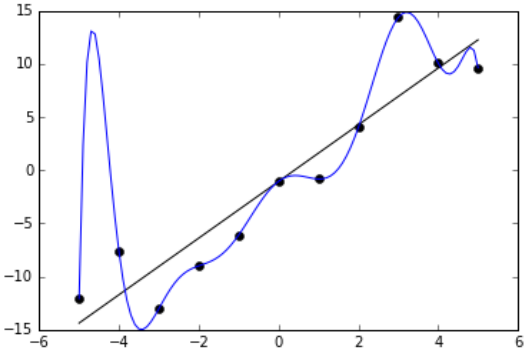
The minimum value of the above is just the distance from y to $\mathcal{C}(\Phi)$, the column space of Φ! Let $\hat{w} = \arg\min_w \varepsilon(w)$.
In particular, if $y \in \mathcal{C}(\Phi)$, it is possible to get the cost to 0.
The least square solution is then the distance from y to the projection $\hat{y}$ of y on $\mathcal{C}(\Phi)$. How do we find $\hat{y}$ and $\hat{w}$?
The line joining y and $\hat{y}$ is orthogonal to $\mathcal{C}(\Phi)$. As a result, $(\hat{y} - y)^\top \Phi = 0$. Therefore,

$$\begin{aligned} \hat{y}^\top \Phi &= y^\top \Phi \\ (\Phi \hat{w})^\top \Phi &= y^\top \Phi \\ \hat{w} &= (\Phi^\top \Phi)^{-1} \Phi^\top y. \end{aligned}$$

This is well-defined only if $(\Phi^\top \Phi)$ is invertible (Φ has full column rank).

Remark (Overfitting). One might think that more basis functions means a better approximation. Indeed, this would mean that we have more "freedom", which should allow us to get a better function. However, this is not the case. While adding more basis functions allows us to emulate the *training* data better, we might mimic it too closely and lose sight of the overall behaviour, which results in bad performance when it comes to the general *testing* data.
This is very clearly seen in the following example where the output is slightly noisy linear data, and we have taken two cases: one wherein the basis function are just $\phi(x) = (1, x)$ (fitting a degree 1 polynomial), and the second where $\phi(x) = (1, x, \ldots, x^{10})$ (fitting a degree 10 polynomial).

FIGURE 1 – Overfitted data



1.3. Probabilistic Interpretations

Let us jump away from regression for a moment and look at what happens if instead of trying to learn a function with inputs and outputs, we are instead trying to determine the parameters of some random variable given some datapoints drawn from it.
To do this, suppose that any observation z is drawn from the random variable Z with probability density/mass function g_θ (depending on the parameter θ, which we are trying to learn). For example, if Z is a gaussian with mean μ and variance σ^2, we might have $\theta = (\mu, \sigma^2)$ and

$$g_\theta(z) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right).$$

1.3.1. Maximum likelihood estimation

Suppose we are given a dataset $\mathcal{D} = \{z_1, \ldots, z_m\}$.
Also suppose that we have discrete outputs for the moment (g_w is a probability *mass* function). Then, define the **likelihood** function

$$\mathcal{L}(\mathcal{D} \mid \theta) = \Pr[\mathcal{D} \mid \theta].$$

Making the mild assumption that the various datapoints in $\mathcal{D}$ are drawn iid, this can be rewritten as

$$\mathcal{L}(\mathcal{D} \mid \theta) = \prod_{i=1}^{m} g_\theta(z_i).$$

Even in the case where we have a continuous output (so a probability density function) instead, this expression remains the same.
As might be expected, in *maximum likelihood estimation* (MLE), we choose that w which maximizes the likelihood. Often, we work with the *log*-likelihood ℓ as well, which is just defined by $\ell(\mathcal{D} \mid w) = \log \mathcal{L}(\mathcal{D} \mid w)$.

1.3.2. Bayesian estimation

In all that we have done so far, we have arrived at a single estimate for w or θ. However, this may not necessarily be the best option – might it not be better to arrive at a probability distribution for w which represents how likely various values are?
Bayesian estimation is one simple way of doing this.

We make the assumption initially that θ is drawn from some **prior** probability distribution p. Given the dataset $\mathcal{D} = \{z_1, \ldots, z_m\}$ (and the function g_θ from earlier), we try to improve on this prior to get to a **posterior** probability distribution that better reflects how θ behaves, now that we have knowledge about the dataset. To do this, we use Bayes' law:

$$\begin{aligned}\Pr[\theta \mid \mathcal{D}] &= \frac{\Pr[\mathcal{D} \mid \theta]\Pr[\theta]}{\Pr[\mathcal{D}]} \\ &= \frac{\Pr[\mathcal{D} \mid \theta]p(\theta)}{\int \Pr[\mathcal{D} \mid \alpha]p(\alpha)\,\mathrm{d}\alpha}.\end{aligned} \tag{1.1}$$

Note the contrast between this and MLE; in the the former we use $\Pr[\theta \mid \mathcal{D}]$, whereas in the latter we use $\Pr[\mathcal{D} \mid \theta]$.
We can then use the posterior to return some estimate of θ.
In *maximum a posteriori estimation* (MAP estimation), we return the mode of the posterior density.
In *Bayesian estimation*, we return the mean of the posterior density.
In the pure Bayesian estimate, we do not return any point estimate for θ at all, and we instead use Bayes' rule to calculate the output given an input as

$$\Pr[x \mid \mathcal{D}] = \int p(x \mid \theta)p(\theta \mid \mathcal{D})\,\mathrm{d}\theta$$

Now, the integral in the denominator of (1.1) can be quite daunting. To deal with this, we introduce the idea of a **conjugate prior**. Such a prior is of the form that the base distribution of both the prior and the posterior are the same. This then enables us to skip the calculation of the denominator, since it merely leads to a constant to normalize the distribution (its integral must be 1), and we know this constant from the structure of the distribution anyway. This can be better understood by an example. Suppose that the datapoints are drawn from a Bernoulli distribution with parameter θ. We then have

$$\Pr[\mathcal{D} \mid \theta] = \theta^r(1-\theta)^{m-r},$$

where r is the number of 1s in $\mathcal{D}$. If we choose our prior to be a Beta distribution with parameters α and β, then the numerator of (1.1) is

$$\theta^r(1-\theta)^{m-r} \cdot \theta^{\alpha-1}(1-\theta)^{\beta-1}\frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}.$$

This is proportional to $\theta^{\alpha+r-1}(1-\theta)^{m-r+\beta-1}$. Since the denominator only gives a constant multiplicative factor, the posterior must just be $\mathrm{Beta}(\alpha+r, \beta+m-r)$.

Some more examples are that the Dirichlet distribution is the conjugate prior to the multivariate Bernoulli and the Gaussian is the conjugate prior to the Gaussian.
Bayesian estimation takes care of overfitting

1.3.3. Relating probabilistic interpretations and regression

Let us jump back to regression. While we typically assume that our input/output pairs are initially taken from some basic (fixed) function f, this is typically not the case. Indeed, we usually have some amount of error when taking the observations itself. This error is commonly taken to be a Gaussian with 0 mean and some (small) variance σ^2. The output as a whole is then just a random variable, with the parameters involving our base parameter θ *and* the input.

For example, let our training data set be $\mathcal{D} = \{\langle x_1, y_1\rangle, \ldots, \langle x_m, y_m\rangle\}$ where the x_i are in $\mathbb{R}^n$ and the y_i in $\mathbb{R}$. Let us perform normal linear regression, but also add a Gaussian error of known variance σ^2. That is, if the input is x, we predict the distribution of the corresponding output y to be $\mathcal{N}(w^\top x, \sigma^2)$.
Supposing we use MLE, we wish to determine

$$\hat{w} = \arg\max_w \frac{1}{(\sqrt{2\pi}\sigma)^m}\exp\left(-\frac{1}{2\sigma^2}\sum_{i=1}^{m}(y_i - w^\top x_i)^2\right).$$

It is not too difficult to see that this is in fact the same solution as that given by the discussion in the method of least squares.

1.4. Regularisation

As mentioned in a remark towards the end of Section 1.2.1, overfitting tends to make the error over the test data too large, due to large fluctuations in the model we learn (to emulate the training data better). However, this also leads to large coefficients of the learnt vector, and thus large norm. Regularisation attempts to fix this, by further adding in the constraint that $\|w\|_2$ is not too large.

1.4.1. Regularised regression

This leads to the *penalized regularised least squares regression problem*, where the cost function is replaced with

$$\|\Phi w - y\|_2^2 + \lambda\Omega(w).$$

We are said to be performing

- *ridge regression* if $\Omega(w) = \|w\|_2^2$,

- *lasso*[1] *regression* if $\Omega(w) = \|w\|_1$, and
- *support-based penalty regression* if $\Omega(w) = \|w\|_0$.

The closed-form solutions to the first two can be computed by setting the gradient to 0. In particular, the closed-form solution for ridge regression is

$$\hat{w} = (\Phi^\top \Phi + \lambda I)^{-1} \Phi^\top y.$$

Also, this turns out to be the same as performing Bayes/MAP estimation with a multivariate Gaussian prior of mean 0 and variance $(1/\lambda)I$.

Alternatively, we have the *constrained regularised least squares regression problem*, where the cost function is the normal $\|\Phi w - y\|_2^2$, but we have the added constraint that $\Omega(w) \leq \theta$ for some fixed θ.
We again have ridge, lasso, or support-based penalty regression depending on Ω.
It in fact turns out that the above two constraints are equivalent – for any penalized formulation with a λ, there is a corresponding constrained formulation problem with some θ (that depends on both λ and the data), and vice-versa.

Why is lasso regression interesting? Observe that the level curves of $\|w\|_1$ (namely cross-polytopes) have many corners on the axes, and as a result, it is likely that many components of w are close to 0 (as opposed to ridge regression where the level curves are rotationally invariant). That is, the learnt vector is likely very sparse, which is desirable.
The closed form solution is the MAP estimate of linear regression subject to the prior being the Laplace distribution with parameters 0 and θ. Note that the Laplace distribution is extremely similar to the Gaussian, except that we use the L^1 norm in the exponential instead of the L^2. The intuition that lasso regression leads to many parameters being close to 0 is strengthened by the sparsity of the Laplace distribution (the density at 0 is high).
However, there is no known closed form solution for the solution to lasso regression. Instead, it is solved iteratively with normal gradient descent. We have

$$\begin{aligned}\hat{w} &= \arg\min_w \|\Phi w - y\|_2^2 + \lambda \|w\|_1 \\ &= \arg\min_w E_{\text{LS}}(w) + \lambda \|w\|_1 .\end{aligned}$$

At the kth step, set

$$w_{\text{LS}}^{k+1} = w_{\text{Lasso}}^k - \eta \nabla E_{\text{LS}}(w_{\text{Lasso}}^k)$$

$$(w_{\text{Lasso}}^{k+1})_i = \begin{cases} (w_{\text{LS}}^{k+1})_i - \lambda\eta, & (w_{\text{LS}}^{k+1})_i > \lambda\eta, \\ (w_{\text{LS}}^{k+1})_i + \lambda\eta, & (w_{\text{LS}}^{k+1})_i < -\lambda\eta, \\ 0, & \text{otherwise.} \end{cases}$$

We then estimate $\hat{w}$ as w_{Lasso}^r for a suitably large r.

1.4.2. Understanding regularisation

Ridge regression might seem slightly silly from the perspective of just training error. Indeed, the the (minimized) error for the ridge cost function is always greater than the (minimized) error for normal least squares.

Suppose we are performing regression with a true function g and let f be our predicted/fitted function. We sample a bunch of points from g to get a dataset, with possibly some noise that causes small perturbations.
There are three primary sources of test error:

- *Bias*: This occurs due to the difference between the nature of the true function and fitted function. It is just equal to $\mathbf{E}[f(x)] - g(x)$.
- *Variance*: This occurs due to the choice of our dataset. It is equal to $\mathbf{Var}(f(\hat{x}))$.

[1] this is an abbreviation for *l*east *a*bsolute *s*hrinkage and *s*election *o*perator

- *Noise*: This occurs due to the implicit noise that is present when we take measurements from the true function. For example, there may be an additive Gaussian noise with mean 0 and variance σ^2. The noise is then just equal to the variance σ^2 of this component.

Now, suppose the noise is some random variable ε with mean 0 and variance σ^2. Then, given an input x, we observe the output y as being drawn from $g(x) + \varepsilon$.
Suppose we are given a dataset $\mathcal{D} = \{\langle x_i, y_i\rangle : 1 \leq i \leq m\}$, and using it we predict the function f.
The question we would like to answer is: what is the expected test error $\mathbf{E}[(f(\hat{x}) - (g(\hat{x}) + \varepsilon))^2]$ for a new input $\hat{x}$?
The expectation is taken over both the sampled dataset and the distribution of ε.
Let $\hat{y} = g(\hat{x}) + \varepsilon$. We then have

$$\begin{aligned}
\mathbf{E}[(f(\hat{x}) - \hat{y})^2] &= \mathbf{E}[f(\hat{x})^2] + \mathbf{E}[\hat{y}^2] - 2\mathbf{E}[f(\hat{x})]\mathbf{E}[\hat{y}] && \text{(assuming the dataset and } \varepsilon \text{ are independent)} \\
&= \mathbf{E}[f(\hat{x})]^2 + \mathbf{Var}(f(\hat{x})) + \mathbf{E}[\hat{y}^2] - 2\mathbf{E}[\hat{f}(x)]g(\hat{x}) \\
&= \mathbf{E}[f(\hat{x})]^2 + \mathbf{Var}(f(\hat{x})) + \mathbf{E}[g(\hat{x})^2 + \varepsilon^2] - 2\mathbf{E}[\hat{f}(x)]g(\hat{x}) \\
&= \mathbf{E}[f(\hat{x})]^2 + g(\hat{x})^2 - 2\mathbf{E}[\hat{f}(x)]g(\hat{x}) + \mathbf{Var}(f(\hat{x})) + \sigma^2 \\
&= \underbrace{(\mathbf{E}[f(\hat{x})] - g(\hat{x}))^2}_{\text{Bias}^2} + \underbrace{\mathbf{Var}(f(\hat{x}))}_{\text{Variance}} + \underbrace{\sigma^2}_{\text{Noise}}.
\end{aligned}$$

Given the bias-variance tradeoff, how do we choose the best predictor? How do we set the model's parameters?
In **bootstrap sampling**, we repeatedly sample observations from a dataset with replacement. Given dataset $\mathcal{D}_b$, we repeatedly choose a V_b, train on $\mathcal{D}_b \setminus V_b$, then test on V_b.
So, the training set is that we use to actually train the algorithm. V_b is the *validation* or *development* set, and is used for model selection and tuning hyperparameters. The test/evaluation set is that we use for testing finally.

§2. Classification

Suppose we wish to intelligently classify emails into two categories depending on whether they are spam or not. How do we learn a method to classify an email as spam?
It is not similar to regression where the function maps to $\mathbb{R}$, but we instead map to a discrete set $\{c_1, c_2, \ldots, c_k\}$.
The simple hack of mapping c_i to i, say, and then performing usual regression. However, the ordering of the c_i influences it significantly then, which is very undesirable, since the ordering doesn't mean anything.
A better idea would be to take c_i as

$$y_i = e_i \in \mathbb{R}^k,$$

to encode the idea that the various components are 'independent'.
We present these ideas over the next few sections.

2.1. Perceptron Classification

2.1.1. The basic idea

Rather than mapping directly to $\mathcal{C} = \{c_1, \ldots, c_k\}$, we could map to some intermediate space which in turn maps to $\mathcal{C}$ (or a sequence of spaces and functions, the last of which is $\mathcal{C}$).

First, consider the binary classification problem, where $\mathcal{C} = \{-1, 1\}$. Let $\mathcal{D} = \{\langle x_i, y_i\rangle : 1 \leq i \leq m\}$, where each $y_i \in \{-1, 1\}$. The ordering does not matter then (both orderings are equivalent), so we can learn a linear classifier.
Suppose $f(x) = w^\top \phi(x)$.
Given a new input x, we predict the new output as 1 if $f(x) \geq 0$ and -1 if $f(x) < 0$. So, we want

$$y \cdot w^\top \phi(x) \geq 0.$$

We learn w as follows. Starting with $w^{(0)} = 0$, we iterate over the dataset. If a point is correctly classified, we leave $w^{(r+1)} = w^{(r)}$. Otherwise, we marginally correct the weights by performing

$$w^{(r+1)} = w^{(r)} + \eta y_{r+1} \phi(x_{r+1}),$$

for some (typically small) learning rate η.
Observe that the quantity $w^\top \phi(x)$ is the distance of x_r from the hyperplane $\{x : w^\top \phi(x) = 0\}$.
So,

$$y_{r+1}(w^{(r+1)})^\top \phi(x_r) = y_{r+1}(w^{(r)})^\top \phi(x_{r+1}) + \eta \left\|\phi(x_{r+1})\right\|^2.$$

This makes the quantity $y_{r+1} w^\top \phi(x_r)$ less negative. We repeat the above process until $y_i w^\top \phi(x_i) \geq 0$ for all i (or until we terminate).
This **perceptron** algorithm does not attempt to find the best separating hyperplane, it merely tries to find any separating hyperplane – it need not provide any 'breathing room' for the inputs. This is addressed by support vector machines, which we shall read later.
One issue with perceptron classification is that it does not acknowledge how much the estimate differs from the expected ± 1 – we only care that $yw^\top \phi(x) \geq 0$, not how positive it is.

2.1.2. Convergence

Now, suppose there exists some u such that for all i, $y_i u^\top \phi(x_i) \geq 0$. It turns out that the perceptron update rule will then converge in a finite number of iterations. That is, perceptrons can model linearly separable functions.

Proof. Since we can normalize u suppose that $\|u\| = 1$. We can also normalize the dataset itself, so assume $\left\|\phi(x_i)\right\| \leq 1$ for all i.
Define the *margin* $\gamma = \min_i |u^\top x_i|$.
Further make the assumption that $\eta = 1$ (the general case follows similarly.)
We claim that in at most $t = 1/\gamma^2$ updates, we will have $y_i {w^{(t)}}^\top x_i > 0$ for all i.
If we update $w^{(k)}$ to $w^{(k+1)} = w^{(k)} + y\phi(x)$ on reading the input $\langle x, y \rangle$,

$${w^{(k+1)}}^\top u = (w^{(k)} + y\phi(x))^\top u \geq {w^{(k)}}^\top u + \gamma.$$

and

$$\left\|w^{(k+1)}\right\|^2 = \left\|w^{(k)} + y\phi(x)\right\|^2 < \left\|w^{(k)}\right\|^2 + 1.$$

As a result, after k updates,

$${w^{(k)}}^\top u \geq k\gamma \text{ and } \left\|w^{(k)}\right\|^2 \leq k$$

so

$$\sqrt{k} \geq \underbrace{\left\|w^{(k)}\right\| \geq u^\top w^{(k)}}_{\text{Using Cauchy-Schwarz}} \geq k\gamma.$$

This gives $k \leq 1/\gamma^2$, completing the proof. ∎

2.1.3. Different loss functions

Consider the loss function

$$L_P(w^\top(\phi(x)), y) = \max\{0, -yw^\top \phi(x)\}.$$

We can then apply the *stochastic gradient descent* algorithm. Let $f_w(x) = w^\top \phi(x)$ and choose some example $\langle x_t, y_t \rangle$. We then compute the gradient of the loss of $f_w(x_t)$ with respect to w, then update the weight vector using it as

$$w \leftarrow w - \nabla_w L(f_w(x_t), y).$$

Perceptron update is then just stochastic gradient descent on this "hinge loss".
Similarly, we could use different loss functions.

Now, what w should we use at the time of testing? Using the final w is not always a good idea due to overfitting and the stochastic nature of the algorithm.
So, we usually use an intermediate w. The *voted perceptron* algorithm counts votes for weight vectors tallying how many examples they have correctly classified. The *averaged perceptron* algorithm uses the averaged weight vector (not necessarily the average of all weight vectors, but a suitable subset).