Lecture 5: More on logistic regression. Second-order methods. Kernels

- Logistic regression
- Regularization
- Kernelizing linear methods

Recall: Logistic regression

- Hypothesis is a logistic function of a linear combination of inputs:

$$h(\mathbf{x}) = \frac{1}{1 + \exp(\mathbf{w}^T \mathbf{x})}$$

- We interpret $h(\mathbf{x})$ as $P(y = 1|\mathbf{x})$
- Optimizes the *cross-entropy error function* :

$$J_D(\mathbf{w}) = -\left(\sum_{i=1}^{m} y_i \log h(\mathbf{x}_i) + (1 - y_i) \log(1 - h(\mathbf{x}_i)) \right)$$

using gradient descent (or ascent on the log likelihood of the data)

Maximization procedure: Gradient ascent

- First we compute the gradient of $\log L(\mathbf{w})$ wrt $\mathbf{w}$
- The update rule is:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \nabla \log L(\mathbf{w}) = \mathbf{w} + \alpha \sum_{i=1}^{m} (y_i - h_{\mathbf{w}}(\mathbf{x_i}))\mathbf{x_i} = \mathbf{w} + \alpha \mathbf{X}^T(\mathbf{y} - \hat{\mathbf{y}})$$

  where $\alpha \in (0, 1)$ is a step-size or learning rate parameter
- If one uses features of the input, we have:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \mathbf{\Phi}^T(\mathbf{y} - \hat{\mathbf{y}})$$

- The step size α is a parameter for which we have to choose a "good" value

Another algorithm for optimization

- Recall Newton's method for finding the zero of a function $g : \mathbb{R} \to \mathbb{R}$
- At point w^i, approximate the function by a straight line (its tangent)
- Solve the linear equation for where the tangent equals 0, and move the parameter to this point:

$$w^{i+1} = w^i - \frac{g(w^i)}{g'(w^i)}$$

Application to machine learning

- Suppose for simplicity that the error function J has only one parameter
- We want to optimize J, so we can apply Newton's method to find the zeros of $J' = \frac{d}{dw}J$
- We obtain the iteration:

$$w^{i+1} = w^i - \frac{J'(w^i)}{J''(w^i)}$$

- Note that there is *no step size parameter*!
- This is a *second-order method*, because it requires computing the second derivative
- But, if our error function is quadratic, this will find the global optimum in one step!

Second-order methods: Multivariate setting

- If we have an error function J that depends on many variables, we can compute the *Hessian matrix*, which contains the second-order derivatives of J:

$$H_{ij} = \frac{\partial^2 J}{\partial w_i \partial w_j}$$

- The inverse of the Hessian gives the "optimal" learning rates
- The weights are updated as:

$$\mathbf{w} \leftarrow \mathbf{w} - H^{-1} \nabla_{\mathbf{w}} J$$

- This is also called Newton-Raphson method for logistic regression, or Fisher scoring

Which method is better?

- Newton's method usually requires significantly fewer iterations than gradient descent
- Computing the Hessian requires a batch of data, so there is no natural on-line algorithm
- Inverting the Hessian explicitly is expensive, but almost never necessary
- Computing the product of a Hessian with a vector can be done in linear time (Pearlmutter, 1993), which helps also to compute the product of the inverse Hessian with a vector without explicitly computing $\mathbf{H}$

Newton-Raphson for logistic regression

- Leads to a nice algorithm called *iterative recursive least squares*
- The Hessian has the form:

$$\mathbf{H} = \mathbf{\Phi}^T \mathbf{R} \mathbf{\Phi}$$

  where $\mathbf{R}$ is the diagonal matrix of $h(\mathbf{x}_i)(1 - h(\mathbf{x}_i))$ (you can check that this is the form of the second derivative.

- The weight update becomes:

$$\mathbf{w} \leftarrow (\mathbf{\Phi}^T \mathbf{R} \mathbf{\Phi})^{-1} \mathbf{\Phi}^T \mathbf{R} (\mathbf{\Phi} \mathbf{w} - \mathbf{R}^{-1} (\mathbf{\Phi} \mathbf{w} - \mathbf{y}))$$

Regularization for logistic regression

- One can do regularization for logistic regression just like in the case of linear regression
- Recall regularization makes a statement about the weights, so does not affect the error function
- Eg: L_2 regularization will have the optimization criterions:

$$J(\mathbf{w} = J_D(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

Probabilistic view of logistic regression

- Consider the additive noise model we discussed before:

$$y_i = h_{\mathbf{w}}(\mathbf{x}_i) + \epsilon$$

  where ϵ are drawn iid from some distribution
- At first glance, log reg does not fit very well
- We will instead think of a latent variable $\hat{y}_i$ such that:

$$\hat{y}_i = h_{\mathbf{w}}(\mathbf{x}_i) + \epsilon$$

- Then the output is generated as:

$$y_i = 1 \text{ iff } \hat{y}_i > 0$$

Graphical model for logistic regression

Other versions of logistic regression

Linear regression with feature vectors revisited

- Find the weight vector $\mathbf{w}$ which minimizes the (regularized) error function:

$$J(\mathbf{w}) = \frac{1}{2}(\mathbf{\Phi w} - \mathbf{y})^T(\mathbf{\Phi w} - \mathbf{y}) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

- Suppose instead of the closed-form solution, we just take the gradient and rearrange the terms
- The solution takes the form:

$$\mathbf{w} = -\frac{1}{\lambda}\sum_{i=1}^{m}(\mathbf{w}^T\phi(\mathbf{x}_i) - y_i)\phi(\mathbf{x}_i) = \sum_{i=1}^{m} a_i\phi(\mathbf{x}_i) = \mathbf{\Phi}^T\mathbf{a}$$

  where $\mathbf{a}$ is a vector of size m (number of instances) with $a_i = -\frac{1}{\lambda}(\mathbf{w}^T\phi(\mathbf{x}_i) - y_i)$
- Main idea: *use $\mathbf{a}$ instead of $\mathbf{w}$ as parameter vector*

Re-writing the error function

- Instead of $J(\mathbf{w})$ we have $J(\mathbf{a})$:

$$J(\mathbf{a}) = \frac{1}{2}\mathbf{a}^T\mathbf{\Phi}\mathbf{\Phi}^T\mathbf{\Phi}\mathbf{\Phi}^T\mathbf{a} - \mathbf{a}^T\mathbf{\Phi}\mathbf{\Phi}^T\mathbf{y} + \frac{1}{2}\mathbf{y}^T\mathbf{y} + \frac{\lambda}{2}\mathbf{a}^T\mathbf{\Phi}\mathbf{\Phi}^T\mathbf{a}$$

- Denote $\mathbf{\Phi}\mathbf{\Phi}^T = \mathbf{K}$
- Hence, we can re-write this as:

$$J(\mathbf{a}) = \frac{1}{2}\mathbf{a}^T\mathbf{K}\mathbf{K}\mathbf{a} - \mathbf{a}^T\mathbf{K}\mathbf{y} + \frac{1}{2}\mathbf{y}^T\mathbf{y} + \frac{\lambda}{2}\mathbf{a}^T\mathbf{K}\mathbf{a}$$

- This is quadratic in $\mathbf{a}$, and we can set the gradient to 0 and solve.

Dual-view regression

- By setting the gradient to 0 we get:

$$\mathbf{a} = (\mathbf{K} + \lambda \mathbf{I}_m)^{-1}\mathbf{y}$$

- Note that this is similar to re-formulating a weight vector in terms of a linear combination of instances
- Again, the *feature mapping is not needed* either to learn or to make predictions!
- This approach is useful if the feature space is very large

Kernel functions

- Whenever a learning algorithm can be written in terms of dot-products, it can be generalized to kernels.
- A *kernel* is any function $K : \mathbb{R}^n \times \mathbb{R}^n \mapsto \mathbb{R}$ which corresponds to a dot product for some feature mapping ϕ:

$$K(\mathbf{x}_1, \mathbf{x}_2) = \phi(\mathbf{x}_1) \cdot \phi(\mathbf{x}_2) \text{ for some } \phi.$$

- Conversely, by choosing feature mapping ϕ, we implicitly choose a kernel function
- Recall that $\phi(\mathbf{x}_1) \cdot \phi(\mathbf{x}_2) = \cos \angle(\mathbf{x}_1, \mathbf{x}_2)$ where $\angle$ denotes the angle between the vectors, so a kernel function can be thought of as a notion of *similarity*.

Example: Quadratic kernel

- Let $K(\mathbf{x},\mathbf{z}) = (\mathbf{x}\cdot\mathbf{z})^2$.
- Is this a kernel?

$$\begin{aligned} K(\mathbf{x},\mathbf{z}) &= \left(\sum_{i=1}^{n} x_i z_i\right)\left(\sum_{j=1}^{n} x_j z_j\right) = \sum_{i,j\in\{1\ldots n\}} x_i z_i x_j z_j \\ &= \sum_{i,j\in\{1\ldots n\}} (x_i x_j)(z_i z_j) \end{aligned}$$

- Hence, it is a kernel, with feature mapping:

$$\phi(\mathbf{x}) = \langle x_1^2,\ x_1x_2,\ \ldots,\ x_1x_n,\ x_2x_1,\ x_2^2,\ \ldots,\ x_n^2\rangle$$

  Feature vector includes all squares of elements and all cross terms.
- Note that computing ϕ takes $O(n^2)$ but *computing K takes only $O(n)$*!

Polynomial kernels

- More generally, $K(\mathbf{x}, \mathbf{z}) = (\mathbf{x} \cdot \mathbf{z})^d$ is a kernel, for any positive integer d:

$$K(\mathbf{x}, \mathbf{z}) = \left(\sum_{i=1}^{n} x_i z_i \right)^d$$

- If we expanded the sum above in the obvious way, we get n^d terms (i.e. feature expansion)
- Terms are monomials (products of x_i) with total power equal to d.
- If we use the primal form of the SVM, each of these will have a weight associated with it!
- *Curse of dimensionality:* it is very expensive both to optimize and to predict with an SVM in primal form
- However, *evaluating the dot-product of any two feature vectors can be done using K in $O(n)$*!

The “kernel trick”

- If we work with the dual, we do not actually have to ever compute the feature mapping ϕ. We just have to compute the similarity K.
- In our case, we kernelized linear regression, as we do not need to look at features to compute the parameter vector, but only at dot-products of features.

Some other (fairly generic) kernel functions

- $K(\mathbf{x}, \mathbf{z}) = (1 + \mathbf{x} \cdot \mathbf{z})^d$ – feature expansion has all monomial terms of $\leq d$ total power.
- Radial basis/Gaussian kernel:

$$K(\mathbf{x}, \mathbf{z}) = \exp(-\|\mathbf{x} - \mathbf{z}\|^2 / 2\sigma^2)$$

  The kernel has an infinite-dimensional feature expansion, but dot-products can still be computed in $O(n)$!
- Sigmoidal kernel:

$$K(\mathbf{x}, \mathbf{z}) = \tanh(c_1 \mathbf{x} \cdot \mathbf{z} + c_2)$$

Making predictions in the dual view

- For a new input $\mathbf{x}$, the prediction is:

$$h(\mathbf{x}) = \mathbf{w}^T\phi(\mathbf{x}) = \mathbf{a}^T\mathbf{\Phi}\phi(\mathbf{x}) = \mathbf{k}(\mathbf{x})^T(\mathbf{K} + \lambda\mathbf{I}_m)^{-1}\mathbf{y}$$

  where $\mathbf{k}(\mathbf{x})$ is an m-dimensional vector, with the ith element equal to $K(\mathbf{x}, \mathbf{x}_i)$
- That is, the ith element has the similarity of the input to the ith instance
- The features are not needed for this step either!

Logistic regression

- The output of a logistic regression predictor is:

$$h_{\mathbf{w}}(\mathbf{x}) = \frac{1}{1 + e^{\mathbf{w}^T\phi(\mathbf{x}) + w_0}}$$

- Again, we can define the weights in terms of support vectors: $\mathbf{w} = \sum_{i=1}^{m} \alpha_i \phi(\mathbf{x}_i)$
- The prediction can now be computed as:

$$h(\mathbf{x}) = \frac{1}{1 + e^{\sum_{1=1}^{m} \alpha_i K(\mathbf{x}_i, \mathbf{x}) + w_0}}$$

- α_i are the new parameters (one per instance) and can be derived using gradient descent

Kernels

- A lot of current research has to do with defining new kernels functions, suitable to particular tasks / kinds of input objects
- Many kernels are available:
  - Information diffusion kernels (Lafferty and Lebanon, 2002)
  - Diffusion kernels on graphs (Kondor and Jebara 2003)
  - String kernels for text classification (Lodhi et al, 2002)
  - String kernels for protein classification (e.g., Leslie et al, 2002)

  ... and others!

Example: String kernels

- Very important for DNA matching, text classification, ...
- Example: in DNA matching, we use a sliding window of length k over the two strings that we want to compare
- The window is of a given size, and inside we can do various things:
  - Count exact matches
  - Weigh mismatches based on how bad they are
  - Count certain markers, e.g. AGT
- The kernel is the sum of these similarities over the two sequences
- How do we prove this is a kernel?

Establishing "kernelhood"

- Suppose someone hands you a function K. How do you know that it is a kernel?
- More precisely, given a function $K : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, under what conditions can $K(\mathbf{x}, \mathbf{z})$ be written as a dot product $\phi(\mathbf{x}) \cdot \phi(\mathbf{z})$ for some feature mapping ϕ?
- We want a general recipe, which does not require explicitly defining ϕ every time

Kernel matrix

- Suppose we have an arbitrary set of input vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_m$
- The *kernel matrix (or Gram matrix)* K corresponding to kernel function K is an $m \times m$ matrix such that $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$ (notation is overloaded on purpose).
- What properties does the kernel matrix K have?
- Claims:
  1. K is symmetric
  2. K is positive semidefinite
- Note that these claims are consistent with the intuition that K is a "similarity" measure (and will be true regardless of the data)

Proving the first claim

If K is a valid kernel, then the kernel matrix is symmetric

$$K_{ij} = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j) = \phi(\mathbf{x}_j) \cdot \phi(\mathbf{x}_i) = K_{ji}$$

Proving the second claim

If K is a valid kernel, then the kernel matrix is positive semidefinite

Proof: Consider an arbitrary vector $\mathbf{z}$

$$\begin{aligned}
\mathbf{z}^T K \mathbf{z} &= \sum_i \sum_j z_i K_{ij} z_j = \sum_i \sum_j z_i \left(\phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j)\right) z_j \\
&= \sum_i \sum_j z_i \left(\sum_k \phi_k(\mathbf{x}_i) \phi_k(\mathbf{x}_j) \right) z_j \\
&= \sum_k \sum_i \sum_j z_i \phi_k(\mathbf{x}_i) \phi_k(\mathbf{x}_j) z_j \\
&= \sum_k \left(\sum_i z_i \phi_k(\mathbf{x}_i) \right)^2 \geq 0
\end{aligned}$$

Mercer's theorem

- We have shown that if K is a kernel function, then for any data set, the corresponding kernel matrix K defined such that $K_{ij} = K(\mathbf{x_i}, \mathbf{x_j})$ is symmetric and positive semidefinite
- Mercer's theorem states that the reverse is also true:

  Given a function $K : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, *$K$ is a kernel if and only if, for any data set, the corresponding kernel matrix is symmetric and positive semidefinite*
- The reverse direction of the proof is much harder (see e.g. Vapnik's book for details)
- This result gives us a way to check if a given function is a kernel, by checking these two properties of its kernel matrix.
- Kernels can also be obtained by combining other kernels (see next homework), or by learning from data
- Kernel learning may suffer from overfitting (kernel matrix close to diagonal)

Support Vector Regression

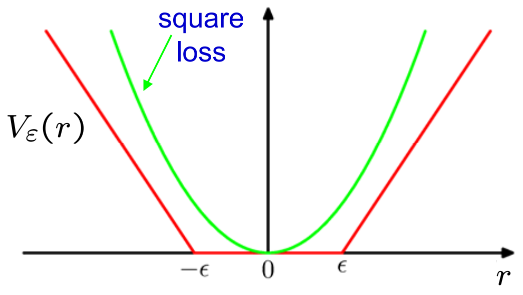
- In regression problems, so far we have been trying to minimize mean-squared error:
$$\sum_i (y_i - (\mathbf{w} \cdot \mathbf{x_i} + w_0))^2$$
- In SVM regression, we will be interested instead in minimizing absolute error:
$$\sum_i |y_i - (\mathbf{w} \cdot \mathbf{x_i} + w_0)|$$
- This is more *robust to outliers* than the squared loss
- But we cannot require that all points be approximated correctly (overfitting!)

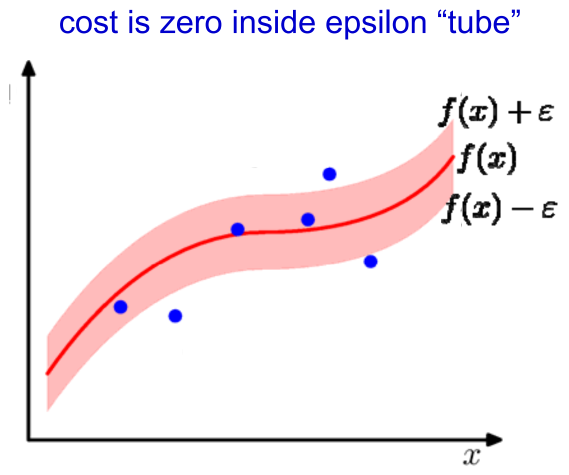
Loss function for support vector regression

In order to allow for misclassifications in SVM regression (and have robustness to noise), we use the *ϵ-insensitive loss*:

$$J_\epsilon = \sum_{i=1}^{m} J_\epsilon(\mathbf{x_i}), \text{ where}$$

$$J_\epsilon(\mathbf{x_i}) = \begin{cases} 0 & \text{if } |y_i - (\mathbf{w} \cdot \mathbf{x_i} + w_0)| \leq \epsilon \\ |y_i - (\mathbf{w} \cdot \mathbf{x_i} + w_0)| - \epsilon & \text{otherwise} \end{cases}$$





1

Solving SVM regression

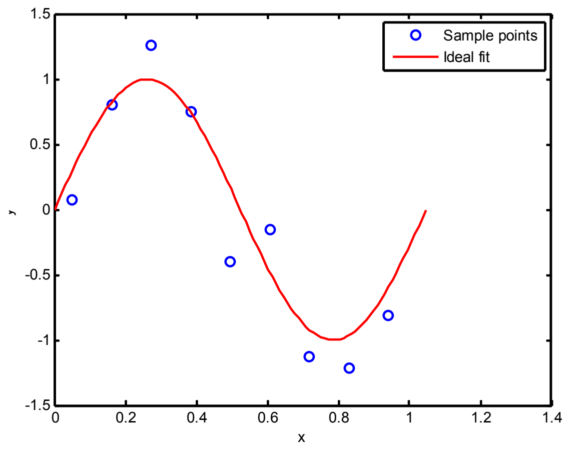
- We use ideas similar to the soft margin classifiers
- We introduce slack variables, ξ_i^+, ξ_i^- to account for errors outside the tolerance area
- We need two kinds of variables to account for both positive and negative errors

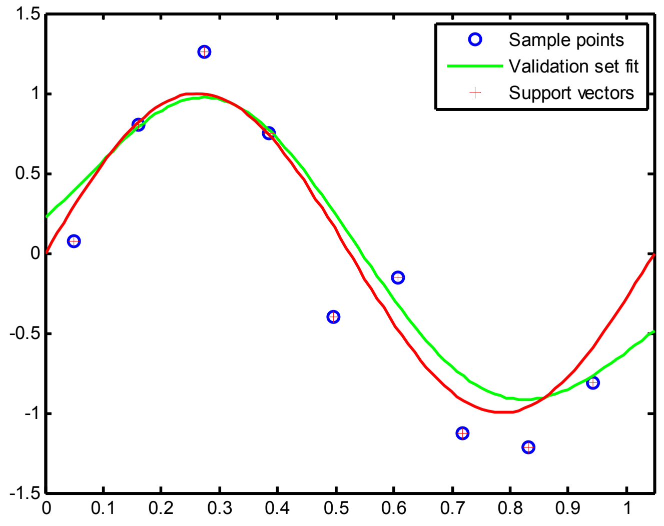
The optimization problem

$$\begin{array}{rl}
\text{min} & \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_i(\xi_i^+ + \xi_i^-) \\
\text{w.r.t.} & \mathbf{w}, w_0, \xi_i^+, \xi_i^- \\
\text{s.t.} & y_i - (\mathbf{w}\cdot\mathbf{x}_i + w_0) \leq \epsilon + \xi_i^+ \\
 & y_i - (\mathbf{w}\cdot\mathbf{x}_i + w_0) \geq -\epsilon - \xi_i^- \\
 & \xi_i^+, \xi_i^- \geq 0
\end{array}$$

- Like before, we can write the Lagrangian and solve the dual form of the problem
- Kernels can be used as before to get non-linear functions

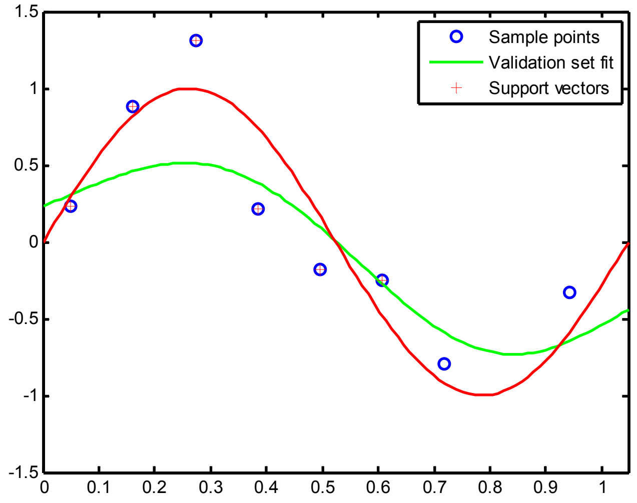
Effect of ϵ





epsilon = 0.01

epsilon = 0.8



[2]

- As ϵ increases, the function is allowed to move away from the data points, the number of support vectors decreases and the fit gets worse

[2] Zisserman course notes