CSC2515 Fall 2007
Introduction to Machine Learning

Lecture 6: Optimization

Regression/Classification & Probabilities

- The "standard" setup
- Assume data are iid from unknown joint distribution $p(y, \mathbf{x}|\mathbf{w})$ or an unknown conditional $p(y|\mathbf{x}, \mathbf{w})$
- We see some examples $(y_1, \mathbf{x}_1)(y_2, \mathbf{x}_2)...(y_n, \mathbf{x}_n)$ and we want to infer something about the parameters (weights) of our model
- The most basic thing is to optimize the parameters using *maximum likelihood* or *maximum conditional likelihood*
- A better thing to do is maximum penalized (conditional) likelihood, which includes regularization terms such as factorization, shrinkage, input selection, or smoothing
- An even better thing to do is to go Bayesian, but this is often too computationally demanding

Maximum Likelihood

- Basic ML question: For which setting of the parameters is the data we saw the most likely?
- Assumes training data are iid, computes the log likelihood, forms a function $\ell(\mathbf{w})$ which depends on the fixed training set we saw and on the argument w:

$$\begin{aligned}\ell(\mathbf{w}) &= \log p(y_1, \mathbf{x}_1, y_2, \mathbf{x}_2, ..., y_n, \mathbf{x}_n|\mathbf{w}) \\ &= \log \prod_n p(y_n, \mathbf{x}_n|\mathbf{w}) \quad \text{since iid} \\ &= \sum_n \log p(y_n, \mathbf{x}_n|\mathbf{w}) \quad \text{since } \log \prod = \sum \log\end{aligned}$$

- Maximizing likelihood is equivalent to minimizing sum squared error, if the noise model is Gaussian and datapoints are iid:

$$\ell(\mathbf{w}) = -\tfrac{1}{2\sigma^2} \textstyle\sum_n (y_n - f(\mathbf{x}_n; \mathbf{w}))^2 + \text{const}$$

Maximum A Posteriori (MAP)

- MAP asks question: Which setting of the parameters is most likely to be drawn from the prior $p(\mathbf{w})$ and then to generate the data from $p(X, Y|\mathbf{w})$?
- It assumes training data are iid, computes the log posterior, and maximizes a function $\ell(\mathbf{w})$ which depends on the fixed training set we saw and on the argument w:

$$\begin{aligned}\ell(\mathbf{w}) &= \log p(\mathbf{w}) + \log p(y_1, \mathbf{x}_1, y_2, \mathbf{x}_2, ..., y_n, \mathbf{x}_n|\mathbf{w}) + \text{const} \\ &= \log p(\mathbf{w}) + \log \prod_n p(y_n, \mathbf{x}_n|\mathbf{w}) \\ &= \log p(\mathbf{w}) + \sum_n \log p(y_n, \mathbf{x}_n|\mathbf{w})\end{aligned}$$

- E.g., MAP is equivalent to ridge regression, if the noise model is Gaussian, the weight prior is Gaussian, and the datapoints are iid:

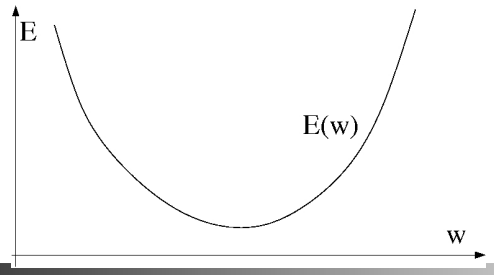
$$\ell(\mathbf{w}) = -\frac{1}{2\sigma^2}\sum_n (y_n - \mathbf{w}^T\mathbf{x}_n)^2 - \lambda \sum_i w_i^2 + \text{const}$$

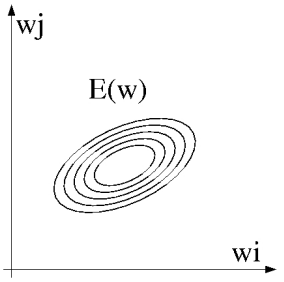
Going Bayesian

- Ideally we would be Bayesian, applying Bayes rule to compute
$$p(\mathbf{w}|y_1, \mathbf{x}_1, y_2, \mathbf{x}_2, ..., y_n, \mathbf{x}_n)$$
- This is the *posterior distribution* of the parameters given the data. A true Bayesian would integrate over the posterior to make predictions:
$$p(y^{new}|x^{new}, Y, X) = \int p(y^{new}|x^{new}, \mathbf{w})p(\mathbf{w}|Y, X)d\mathbf{w}$$
but often this is analytically intractable and/or computationally difficult
- We can settle for maximizing and using the argmax $\mathbf{w}^*$ to make future predictions: this is the *maximum a posterior* (MAP) approach
- Many of the penalized maximum likelihood techniques we used for regularization are equivalent to MAP with certain parameter priors:
  - Quadratic weight decay (shrinkage) $\Leftrightarrow$ Gaussian prior
  - Absolute weight decay (lasso) $\Leftrightarrow$ Laplace prior
  - Smoothing on multinomial parameters $\Leftrightarrow$ Dirichlet prior
  - Smoothing on covariance matrices $\Leftrightarrow$ Wishart prior

Error Surfaces and Weight Space

- End result: an "error function" $E(\mathbf{w})$ which we want to minimize
- $E(\mathbf{w})$ can be the negative of the log likelihood or log posterior
- Consider a fixed training set; think in weight (not input) space. At each setting of the weights there is some error (given the fixed training set): this defines an error surface in weight space.
- Learning == descending the error surface
- Notice: if the data are iid, the error function E is a sum of error functions E_n, one per data point





Quadratic Error Surfaces and IID Data

- A very common form for the cost (error) function is the quadratic:

$$E(\mathbf{w}) = \mathbf{w}^T \mathbf{A} \mathbf{w} + 2\mathbf{w}^T \mathbf{b} + c$$

- This comes up as the log probability when using Gaussians, since if the noise model is Gaussian, each of the E_n is an upside-down parabola (a "quadratic bowl" in higher dimensions).
- Fact: sum of parabolas (quadratics) is another parabola (quadratic)
- So the overall error surface is just a quadratic bowl
- It is easy to find the minimum of a quadratic bowl:

$$E(w) = a + bw + cw^2 \quad \Rightarrow \quad w^* = -b/2c$$

$$E(\mathbf{w}) = a + \mathbf{b}^T \mathbf{C} \mathbf{w} \quad \Rightarrow \quad \mathbf{w}^* = -\tfrac{1}{2}\mathbf{C}^{-1}\mathbf{b}$$

- For linear regression with Gaussian noise:

$$\mathbf{C} = \mathbf{X}\mathbf{X}^T \quad \text{and} \quad \mathbf{b} = -2\mathbf{X}\mathbf{y}^T$$

Partial Derivatives of Error

- Question: if we wiggle w_k and keep everything else the same, does the error get better or worse?
- Calculus provides the answer: $\frac{\partial E}{\partial w_k}$
- Plan: use a differentiable cost function E and compute *partial derivatives* of each parameter with respect to this error:

$$\nabla E(\mathbf{w}) = (\frac{\partial E}{\partial w_1}, ..., \frac{\partial E}{\partial w_M})$$

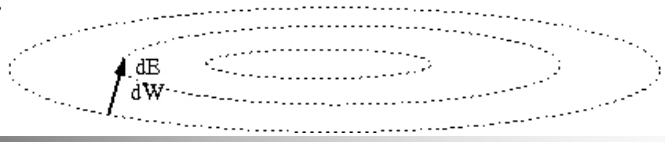
- Use the *chain rule* to compute the derivatives
- The vector of partial derivatives is called the gradient of the error. The negative gradient points in the direction of steepest error descent in weight space.
- Three fundamental questions:
  1. How do we compute the gradient efficiently?
  2. Once we have the gradient, how do we minimize the error?
  3. Where will we end up in weight space?

Steepest Descent

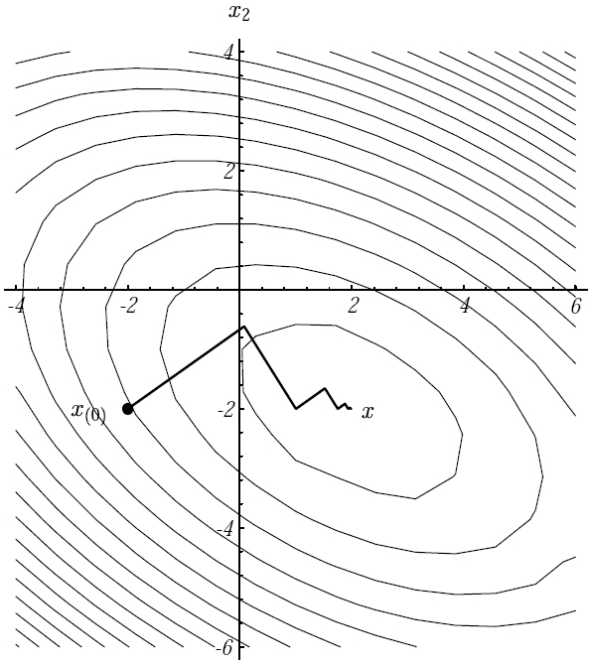
- Once we have the gradient of the error function, how do we minimize the weights?
- Steepest descent:

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \epsilon \nabla E(\mathbf{w})$$

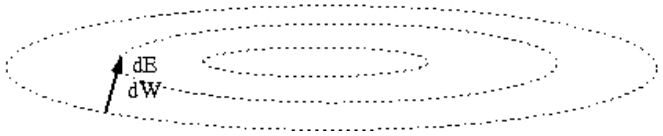
- If the steps are small enough, then this is guaranteed to converge to at least a local minimum
- But if we are interested in the rate of convergence, this may not be the best approach
  - Stepsize is a free parameter that has to be chosen carefully for each problem
  - The error surface may be curved differently in different directions. This means that the gradient does not necessarily point directly to the nearest local minimum.



Steepest Descent: Far from Ideal



Error Surface: Curvature



- The local geometry of curvature is measured by the Hessian: the matrix of second-order partial derivatives: $H_{ij} = \partial^2 E / \partial w_i w_j$
- Eigenvectors/eigenvalues of the Hessian describe the directions of principal curvature and the amount of curvature in each direction.
- Maximum sensible stepsize is $2/\lambda_{max}$
- Rate of convergence depends on $(1 - 2\frac{\lambda_{min}}{\lambda_{max}})$

Adaptive Stepsize

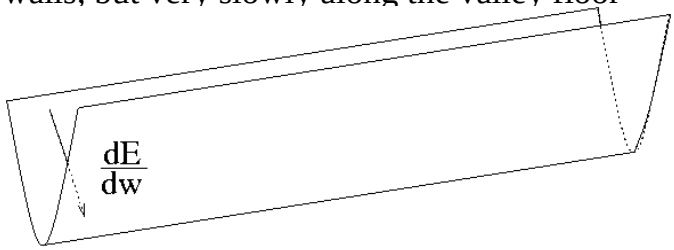
- No general prescriptions for selecting appropriate learning rate; typically no fixed learning rate appropriate for entire learning period
- "Bold driver" heuristic: monitor error after each *epoch* (sweep through entire training set)
  1. If error decreases, increase learning rate: $\epsilon = \epsilon * \rho$
  2. If error increases, decrease rate, reset parameters:

$$\epsilon = \epsilon * \sigma; \quad \mathbf{w}^t = \mathbf{w}^{t-1}$$

- Sensible choices for parameters: $\rho = 1.1, \quad \sigma = 0.5$
- This is *batch* gradient descent

Momentum

- If the error surface is a long and narrow valley, gradient descent goes quickly down the valley walls, but very slowly along the valley floor



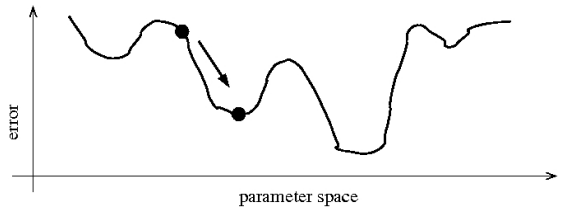
- We can alleviate this problem by updating parameters using a combination of the previous update and the gradient update:

$$\Delta w_j^t = \beta \Delta w^{t-1} + (1-\beta)\left(-\epsilon\, \partial E / \partial w_j(\mathbf{w}^t)\right)$$

- Usually β is set quite high, about 0.95.
- This is like giving momentum to the weights

Convexity, Local Optima

- Unfortunately, many error functions while differentiable are not unimodal.
- When using gradient descent we can get stuck in local minima; where we end up depends on where we start.



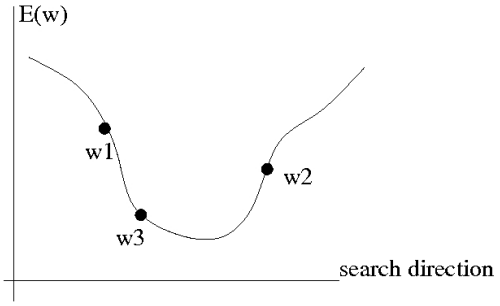
- Some very nice error functions (e.g. linear least squares, logistic regression, lasso) are *convex*, and thus have a unique (global) minimum.
- Convexity means that the second derivative is always positive. No linear combination of weights can have greater error than thelinear combination of the original errors.
- But most settings do not lead to convex optimization problems.

Mini-Batch and Online Optimization

- When the dataset is large, computing the exact gradient is expensive
- This seems wasteful since the only thing we use the gradient for is to compute a small change in the weights, then throw this out and recompute the gradient all over again
- An approximate gradient is useful as long as it points in roughly the same direction as the true gradient
- One easy way to do this is to divide the dataset into small batches of examples, compute the gradient using a single batch, make an update, then move to the next batch of examples: mini-batch optimization
- In the limit, if each batch contains just one example, then this is the ‘online’ learning, or stochastic gradient descent mentioned in Lecture 2.
- These methods are much faster than exact gradient descent, and are very effective when combined with momentum, but care must be taken to ensure convergence

Line Search

- Rather than take a fixed step in the direction of the negative gradient or the momentum-smoothed negative gradient, it is possible to do a search along that direction to find the minimum of the function

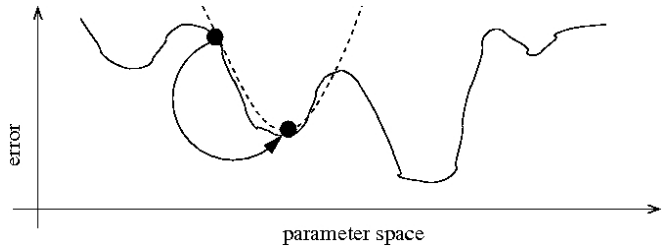


- Usually the search is a bisection, which bounds the nearest local minimum along the line between any two points such that there is a third point $\mathbf{w}_3$ with $E(\mathbf{w}_3) < E(\mathbf{w}_1)$ and $E(\mathbf{w}_3) < E(\mathbf{w}_2)$

Local Quadratic Approximation

- By taking a Taylor series of the error function around any point in weight space, we can make a *local quadratic approximation* based on the value, slope, and curvature:

$$E(\mathbf{w} - \mathbf{w}_0) \approx E(\mathbf{w}_0) + (\mathbf{w} - \mathbf{w}_0)^T \frac{\partial E}{\partial \mathbf{w}} + (\mathbf{w} - \mathbf{w}_0)^T \frac{\mathbf{H}(\mathbf{w}_0)}{2} (\mathbf{w} - \mathbf{w}_0)$$



- Newton's method: jump to the minimum of this quadratic, repeat:

$$\mathbf{w}^* = \mathbf{w} - \mathbf{H}^{-1}(\mathbf{w}) \frac{\partial E}{\partial \mathbf{w}}$$

Second Order Methods

- Newton's method is an example of a *second order optimization* method because it makes use of the curvature or Hessian matrix
- Second order methods often converge much more quickly, but it can be very expensive to calculate and store the Hessian matrix.
- In general, most people prefer clever first order methods which need only the value of the error function and its gradient with respect to the parameters. Often the sequence of gradients (first order derivatives) can be used to *approximate* the second order curvature. This can even be better than the true Hessian, because we can constrain the approximation to always be positive definite.

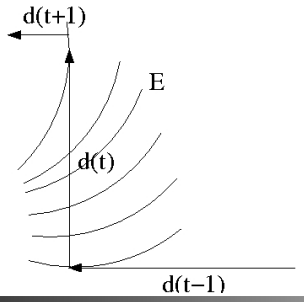
Newton and Quasi-Newton Methods

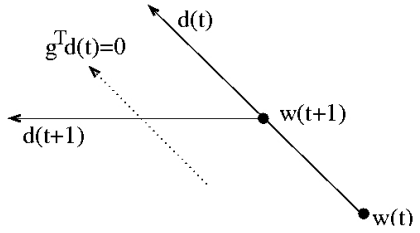
- Broyden-Fletcher-Goldfarb-Shanno (BFGS); Conjugate-Gradients (CG); Davidon-Fletcher-Powell (DVP); Levenberg-Marquardt (LM)
- All approximate the Hessian using recent function and gradient evaluations (e.g., by averaging outer products of gradient vectors, but tracking the ``twist'' in the gradient; by projecting out previous gradient directions...).
- Then they use this approximate gradient to come up with a new search direction in which they do a combination of fixed-step, analytic-step and line-search minimizations.
- Very complex area -- we will go through in detail only the CG method, and a bit of the limited-memory BFGS,

Conjugate Gradients

- Observation: at the end of a line search, the new gradient is (almost) orthogonal to the direction we just searched in.
- So if we choose the next search direction to be the new gradient, we will always be searching successively orthogonal directions and things will be very slow.
- Instead, select a new direction so that, *to first order*, as we move in the new direction the gradient parallel to the old direction stays zero. This involves blending the current negative gradient with the previous search direction:

$$\mathbf{d}(t+1) = -\mathbf{g}(t+1) + \beta(t)\mathbf{d}(t)$$





More Conjugate Gradients

- To first order, all three expressions below satisfy our constraint that along the new search direction $\mathbf{g}^T\mathbf{d}(t) = 0$

$$\mathbf{d}(t+1) = -\mathbf{g}(t+1) + \beta(t)\mathbf{d}(t)$$

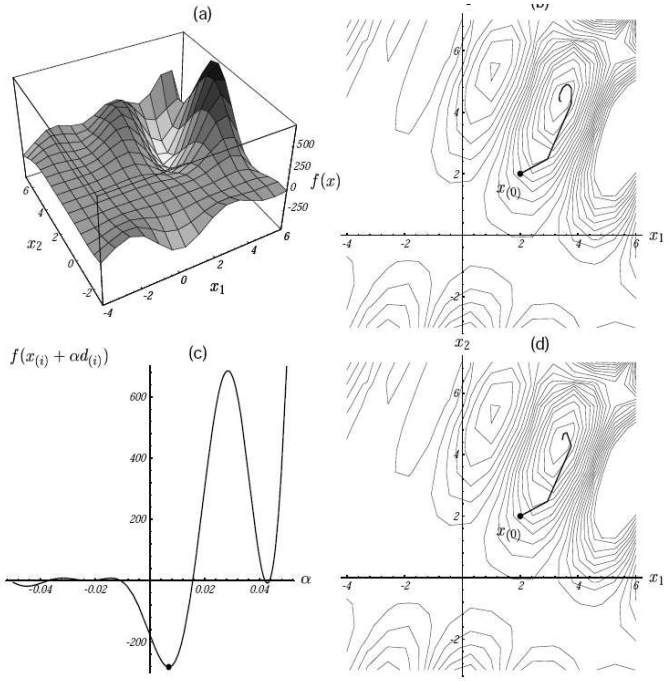
$$\beta(t) = \frac{\mathbf{g}^T(t+1)\boldsymbol{\Delta}\mathbf{g}(t+1)}{\mathbf{d}^T(t)\boldsymbol{\Delta}\mathbf{g}(t+1)} \qquad \text{Hestenes} - \text{Stiefel}$$

$$\beta(t) = \frac{\mathbf{g}^T(t+1)\boldsymbol{\Delta}\mathbf{g}(t+1)}{\mathbf{g}^T(t)\mathbf{g}(t)} \qquad \text{Polak} - \text{Ribiere}$$

$$\beta(t) = \frac{\mathbf{g}^T(t+1)\mathbf{g}(t+1)}{\mathbf{g}^T(t)\mathbf{g}(t)} \qquad \text{Fletcher} - \text{Reeves}$$

where $\boldsymbol{\Delta}\mathbf{g}(t+1) = \mathbf{g}(t+1) - \mathbf{g}(t)$

Conjugate Gradients: Example



BFGS

- One-step memoryless quasi-Newton method
- A *secant* method – iteratively constructing approximation to Hessian matrix

$$\mathbf{d}(t) = -\mathbf{g}(t) + \mathbf{A}(t)\boldsymbol{\Delta}\mathbf{w}(t) + \mathbf{B}(t)\boldsymbol{\Delta}\mathbf{g}(t)$$

$$\text{where } \boldsymbol{\Delta}\mathbf{w}(t) = \mathbf{w}(t) - \mathbf{w}(t-1)$$

$$\mathbf{A}(t) = -\left(1 + \frac{\boldsymbol{\Delta}\mathbf{g}(t)^T\boldsymbol{\Delta}\mathbf{g}(t)}{\boldsymbol{\Delta}\mathbf{w}(t)^T\boldsymbol{\Delta}\mathbf{g}(t)}\right)\frac{\boldsymbol{\Delta}\mathbf{w}(t)^T\mathbf{g}(t)}{\boldsymbol{\Delta}\mathbf{w}(t)^T\boldsymbol{\Delta}\mathbf{g}(t)} + \frac{\boldsymbol{\Delta}\mathbf{g}(t)^T\mathbf{g}(t)}{\boldsymbol{\Delta}\mathbf{w}(t)^T\boldsymbol{\Delta}\mathbf{g}(t)}$$

$$\mathbf{B}(t) = \frac{\boldsymbol{\Delta}\mathbf{w}(t)^T\mathbf{g}(t)}{\boldsymbol{\Delta}\mathbf{w}(t)^T\boldsymbol{\Delta}\mathbf{g}(t)}$$

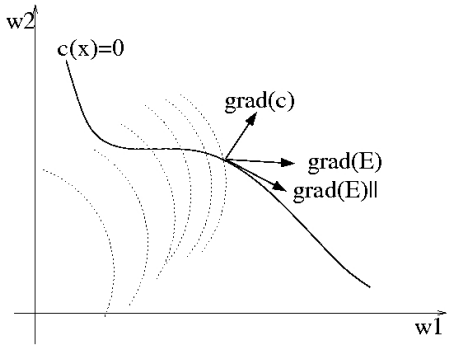
- Every N steps, where N is number of parameters, search is restarted in the direction of the negative gradient

Constrained Optimization

- Sometimes we want to optimize with some constraints on the parameters, e.g.,:
  - Variances are always positive
  - Priors are non-negative and sum to unity (live on the simplex)
- There are two ways to get around this:
  1. Re-parametrize so that the new parameters are unconstrained. For example, we can use log(variances) or use softmax(inputs) for priors.
  2. Explicitly incorporate the constraints into our cost function

Lagrange Multipliers

- Imagine that our parameters have to live inside the constraint surface $c(\mathbf{w}) = 0$
- To optimize our function $E(\mathbf{w})$ we want to look at the component of the gradient that lies *within* the surface, i.e., with zero dot product to the normal of the constraint.
- At the constrained optimum, this gradient component is zero. In other words the gradient of the function is parallel to the gradient of the constraint surface.



More Lagrange Multipliers

- At the constrained optimum, the gradient of the function is parallel to the gradient of the constraint surface: $\frac{\partial E}{\partial \mathbf{w}} = \lambda \frac{\partial c}{\partial \mathbf{w}}$
- The constant of proportionality is called the *Lagrange multiplier*. Its value can be found by forcing $c(\mathbf{w}) = 0$
- In general, the Lagrangian function has the property that when its gradient is zero, the constraints are satisfied and there is no gradient within the constraint surface

$$\begin{aligned} L(\mathbf{w}, \lambda) &= E(\mathbf{w}) + \lambda^T \mathbf{c}(\mathbf{w}) \\ \partial L / \partial \mathbf{w} &= \partial E / \partial \mathbf{w} + \lambda^T \partial \mathbf{c} / \partial \mathbf{w} \\ \partial L / \partial \lambda &= \mathbf{c}(\mathbf{w}) \end{aligned}$$

Lagrange Multipliers: Example

- Find the maximum over x of the quadratic form

$$E(\mathbf{x}) = \mathbf{b}^T\mathbf{x} - \tfrac{1}{2}\mathbf{x}^T\mathbf{A}^{-1}\mathbf{x}$$

subject to the K conditions $c_k(\mathbf{x}) = 0$

- Answer: use Lagrange multipliers:

$$L(\mathbf{x}, \lambda) = E(\mathbf{x}) + \lambda^T\mathbf{c}(\mathbf{x})$$

Now set $\frac{\partial L}{\partial x} = 0$ and $\frac{\partial L}{\partial \lambda} = 0$:

$$\begin{aligned}\mathbf{x}^* &= \mathbf{A}\mathbf{b} + \mathbf{A}\mathbf{C}\lambda \\ \lambda &= -4(\mathbf{C}^T\mathbf{A}\mathbf{C})\mathbf{C}^T\mathbf{A}\mathbf{b}\end{aligned}$$

where the kth column of C is $\partial c_k(\mathbf{x})/\partial\mathbf{x}$