Numerical Optimization

Signal Data Science

Finding maxima and minima

Gradient descent

Stochastic gradient descent

Newton's method

The expectation–maximization algorithm

The expectation–maximization (EM) algorithm is a standard technique for iteratively finding maximum likelihood estimates of parameters in statistical models where the parameters are themselves dependent on latent variables.

For intuition, consider a classic example of the EM algorithm in use, k-means clustering: we posit that there exist k clusters with means located at certain locations (parameters of the model) and that each data point belongs to the closest of the k clusters (the cluster assignment is a hidden variable). We clearly see that the model parameters and hidden variables are circularly dependent—with an estimate of cluster assignment, we can calculate an estimate of the cluster means (by taking the mean location of each point assigned to a particular cluster), and similarly, with an estimate of cluster means, we can calculate an estimate of the cluster assignments (by checking for each point what the closest cluster mean is). This process is repeated until convergence.

More generally, we have a dataset $\mathbf{X}$, hidden ("latent") variables $\mathbf{Z}$, and model parameters $\boldsymbol{\theta}$. Typically, one latent variable is associated with each data point, and the latent variables can take on one of a fixed number of values (*i.e.*, they are discrete). The model parameters are usually of two classes—first, those associated with the entire dataset, and second, those associated only with the data points whose corresponding latent variable takes on some particular value.

If we knew the true values of the latent variables $\mathbf{Z}$, we would be able to obtain the best estimates of the parameters $\boldsymbol{\theta}$ by averaging some function of the data points $\mathbf{X}$ (either over the entire dataset or over only the data points whose

corresponding latent variables take on some particular value). Similarly, if we knew the true values of the model parameters $\boldsymbol{\theta}$, finding the values of $\mathbf{Z}$ which maximize the likelihood associated with the entire model is a straightforward matter of iterating over possible values of the latent variable for each data point. This suggests an iterative method in the case where both $\mathbf{Z}$ and $\boldsymbol{\theta}$ are unknown, *i.e.*, the EM algorithm.

k-means clustering

Gaussian mixture models

Markov chain Monte Carlo

Iterative principal component extraction

The technique of principal component analysis (PCA) can be conceptualized in the following manner: First, we find the direction along which the data varies the most (analogous to the semi-major axis of an ellipse). That is the first principal component. We then 'subtract off' that dimension of variation from the data and then find the direction of greatest variation in the *reduced* data. This is the second principal component. Repeating this process, we end up with p principal components for a dataset of dimension p (containing p variables).

With a matrix of data $\mathbf{X}$, PCA is ordinarily calculated through the computation of the singular value decomposition of $\mathbf{X}$ or an eigendecomposition of the covariance matrix of $\mathbf{X}$. However, the procedure outlined above illustrates more of the *intuition* behind PCA. The first principal component (PC1) is the dimension which accounts for as much variation as possible, PC2 accounts for as much variation as possible after PC1 is removed, PC3 accounts for as much variation as possible after PC1 and PC2 are removed, and so on and so forth.

In the following, we will implement our own version of PCA using the method of iterative principal component extraction described above and compare our answer to the PCA method implemented in R's `prcomp()` (which uses a SVD-based method).

- Initialize a random matrix with `set.seed(1); X = matrix(sample(0:1, 100, replace=TRUE), nrow=10); X = scale(X)`.

Extracting the first principal component

Given $\mathbf{X}$, the vector representing the direction of greatest variation is given by the vector $\mathbf{w}$ which *maximizes* the expression $(\mathbf{w}^\intercal \mathbf{X}^\intercal \mathbf{X} \mathbf{w}) \,/\, (\mathbf{w}^\intercal \mathbf{w})$, where

$\mathbf{X}^\intercal$ represents the transpose operation (calculated in R with `t()`). We call the expression which we wish to maximize our objective function.

- Write a function `objective(w, X)` which calculates the value of the objective function for a given matrix `X` and vector `w`. Verify that `objective(1:10, X)` returns 6.89004.

Instead of implementing our own numerical optimization algorithm, we will use R's `optim()`, which implements a variety of different optimization techniques.

- Write a function `vnorm(v)` which calculates the magnitude (*i.e.*, the L^2 norm) of `v`. Verify that `vnorm(1:10)` returns 19.621.
- Write a function `extract_pc(X, method)` which uses `optim()` to maximize the objective function and calculate the first principal component of `X` using the optimization method specified by `method`. Keep the following in mind, referring to the documentation as necessary:
  - By default, `optim()` tries to *minimize* the objective function. Modify the parameters as necessary for *maximization*.
  - Change the default behavior of `optim()` so that the maximum number of iterations is 10,000.
  - For an initial starting point (the `par` parameter of `optim()`), use a vector of all 1s with length equal to the dimensionality of `X`.
  - Passing in the matrix `X` to `objective()` can be accomplished by passing in `X=X` to `optim()` (accomplished via the `...` parameter).
  - Return the calculated vector `w` which maximizes `objective(w, X)` *normalized* by dividing by its

Moreover, we can use `prcomp()` to verify that `extract_pc()` works as desired.

- Write a function `prcomp_pc(X)` which runs `prcomp()` on `X` and returns the loadings of the first principal component.

In order to compare the outputs of `extract_pc()` and `prcomp_pc()`, we need to check if their output vectors point in the same direction. To determine if two vectors point in the same direction, we can't just compare the elements of the vectors; for example, $(1, 1)$ and $(10, 10)$ point in the same direction but are clearly quite different on an element-by-element basis.

The key is to look at the dot product of the two vectors, given by $\mathbf{v} \cdot \mathbf{w} = \sum_i v_i w_i$. We can use the identity $\mathbf{v} \cdot \mathbf{w} = \|\mathbf{v}\| \, \|\mathbf{w}\| \cos\theta$, where $\|\mathbf{v}\|$ is the L^2 norm of $\mathbf{v}$ and θ is the *angle* between the two vectors $\mathbf{v}$ and $\mathbf{w}$. If the angle between two vectors is close to either 0 or 180°, then they point along the same direction (an angle of 180° indicates that one vector points in the *exact opposite* direction of the other, like $(1, 1)$ and $(-1, -1)$).

- Write a function `dot_prod(v, w)` which calculates the dot product of `v` and `w`.

- Write a function `angle(v, w)` which calculates the angle θ between `v` and `w` in degrees rather than radians and returns the smaller of θ and $180-\theta$. You may find the inverse cosine function `acos()` useful. Verify that `angle(1:5, 6:10)` returns 15.214 and that `angle(1:5, -5:-1)` returns 50.479.

Finally, we're ready to compare the output of our home-grown `extract_pc()` with the R-based `prcomp_pc()`. We will use the Nelder–Mead method (also called the downhill simplex method or the amoeba method), a classic technique of numerical optimization developed in 1965, to perform our optimization.

- Use `extract_pc()` with `method="Nelder-Mead"` to estimate the first principal component of `X`. Calculate the angle between the output of `extract_pc()` and the output of `prcomp_pc()`. Your answer should be 6.065 degrees.

Unfortunately, the Nelder–Mead method does not perform well on more difficult optimization problems; indeed, it can often converge to suboptimal points. However, `optim()` also implements the Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS), which is an approximation of Newton's method that does not need to directly calculate the second derivatives of the objective function. (Such methods are known as Quasi-Newton methods; BFGS is one of the most popular.)

- How well does `extract_pc()` perform if you use BFGS for the optimization instead of Nelder–Mead?

Extracting every principal component

After having calculated the *first* principal component of $\mathbf{X}$, we need to remove all the information in $\mathbf{X}$ which is captured by the direction of PC1 before we can extract the *second* principal component. In general, given an "intermediate" matrix of data $\mathbf{X}^\star$ (which may be equal to $\mathbf{X}$ but may also already have had multiple principal components "removed"), we can remove the information captured by the direction of the vector $\mathbf{w}$ by computing $\mathbf{X}^\star - \mathbf{X}\mathbf{W}$, where $\mathbf{W} = \mathbf{w}\mathbf{w}^\intercal$ is the outer product of $\mathbf{w}$ with itself. ($\mathbf{W}$ is a *matrix*, in contrast with $\mathbf{w}^\intercal\mathbf{w}$, which evaluates to a single number.)

- Write a function `remove_pc(Xorig, Xinit, pc)` which takes the *original* data matrix $\mathbf{X}$ as `Xorig`, the *intermediate* data matrix $\mathbf{X}^\star$ as `Xinit`, and a vector of weights `pc` corresponding to the principal component to be removed from $\mathbf{X}^\star$. It should return the result of removing the information along the direction of `pc` from `Xinit` as described by the above explanation. Verify that `norm(remove_pc(X, X, 1:10))` returns 1475.395.
- Write a function `extract_all(X, method)` which combines `extract_pc()` and `remove_pc()` to calculate every principal component of `X` with the optimization method specified by `method`, returning a matrix where the ith

column corresponds to the weights of the ith principal component. Write a function `prcomp_all(X)` analogously.

We're ready to see how well our method performs for extraction of *every* principal component.

- Write a function `angles_all(X, method)` which runs both `extract_all()` and `prcromp_all()` on `X` and returns a vector where the ith entry corresponds to the angle between the ith principal component as computed by our home-grown optimization method and the ith principal component as computed by R's `prcomp()`.
- Use `angles_all()` to compare the performance of Nelder–Mead and BFGS for principal component extraction.
- Why is the last value of the output of `angles_all()` so much higher than the others? Provide an intuitive explanation for this phenomenon.