**breeze.optimize**

TODO: document breeze.optimize.minimize, recommend that instead.

Breeze's optimization package includes several convex optimization routines and a simple linear program solver. Convex optimization routines typically take a DiffFunction[T], which is aFunction1 extended to have a gradientAt method, which returns the gradient at a particular point. Most routines will require a breeze.linalg-enabled type: something like a Vector or aCounter.

Here's a simple DiffFunction: a parabola along each vector's coordinate.

scala> import breeze.optimize.\_

scala> val f = new DiffFunction[DenseVector[Double]] {

| def calculate(x: DenseVector[Double]) = {

| (norm((x - 3d) :^ 2d,1d),(x \* 2d) - 6d);

| }

| }

f: java.lang.Object with breeze.optimize.DiffFunction[breeze.linalg.DenseVector[Double]] = $anon$1@617746b2

Note that this function takes its minimum when all values are 3. (It's just a parabola along each coordinate.)

scala> f.valueAt(DenseVector(3,3,3))

Double = 0.0

scala> f.gradientAt(DenseVector(3,0,1))

breeze.linalg.DenseVector[Double] = DenseVector(0.0, -6.0, -4.0)

scala> f.calculate(DenseVector(0,0))

(Double, breeze.linalg.DenseVector[Double]) = (18.0,DenseVector(-6.0, -6.0))

You can also use approximate derivatives, if your function is easy enough to compute:

scala> def g(x: DenseVector[Double]) = (x - 3.0):^ 2.0 sum

scala> g(DenseVector(0.,0.,0.))

Double = 27.0

scala> val diffg = new ApproximateGradientFunction(g)

scala> diffg.gradientAt(DenseVector(3,0,1))

breeze.linalg.DenseVector[Double] = DenseVector(1.000000082740371E-5, -5.999990000127297, -3.999990000025377)

Ok, now let's optimize f. The easiest routine to use is just LBFGS, which is a quasi-Newton method that works well for most problems.

scala> val lbfgs = new LBFGS[DenseVector[Double]](maxIter=100, m=3) // m is the memory. anywhere between 3 and 7 is fine. The larger m, the more memory is needed.

scala> lbfgs.minimize(f,DenseVector(0,0,0))

scala> lbfgs.minimize(f,DenseVector(0,0,0))

breeze.linalg.DenseVector[Double] = DenseVector(2.9999999999999973, 2.9999999999999973, 2.9999999999999973)

scala> f(res6)

Double = 2.129924444096732E-29

That's pretty close to 0! You can also use a configurable optimizer, usingFirstOrderMinimizer.OptParams. It takes several parameters:

case class OptParams(batchSize:Int = 512,

regularization: Double = 1.0,

alpha: Double = 0.5,

maxIterations:Int = -1,

useL1: Boolean = false,

tolerance:Double = 1E-4,

useStochastic: Boolean= false) {

// ...

}

batchSize applies to BatchDiffFunctions, which support using small minibatches of a dataset.regularization integrates L2 or L1 (depending on useL1) regularization with constant lambda.alpha controls the initial stepsize for algorithms that need it. maxIterations is the maximum number of gradient steps to be taken (or -1 for until convergence). tolerance controls the sensitivity of the convergence check. Finally, useStochastic determines whether or not batch functions should be optimized using a stochastic gradient algorithm (using small batches), or using LBFGS (using the entire dataset).

OptParams can be controlled using breeze.config.Configuration, which we described earlier.

**breeze.optimize.linear**

We provide a DSL for solving linear programs, using Apache's Simplex Solver as the backend. This package isn't industrial strength yet by any means, but it's good for simple problems. The DSL is pretty simple:

val lp = new LinearProgram()

import lp.\_

val x0 = Real()

val x1 = Real()

val x2 = Real()

val lpp = ( (x0 + x1 \* 2 + x2 \* 3 )

subjectTo ( x0 \* -1 + x1 + x2 <= 20)

subjectTo ( x0 - x1 \* 3 + x2 <= 30)

subjectTo ( x0 <= 40 )

)

val result = maximize( lpp)

assert( (result.result - DenseVector(40.,17.5,42.5)).norm(2) < 1E-4)

We also have specialized routines for bipartite matching (KuhnMunkres and CompetitiveLinking) and flow problems.