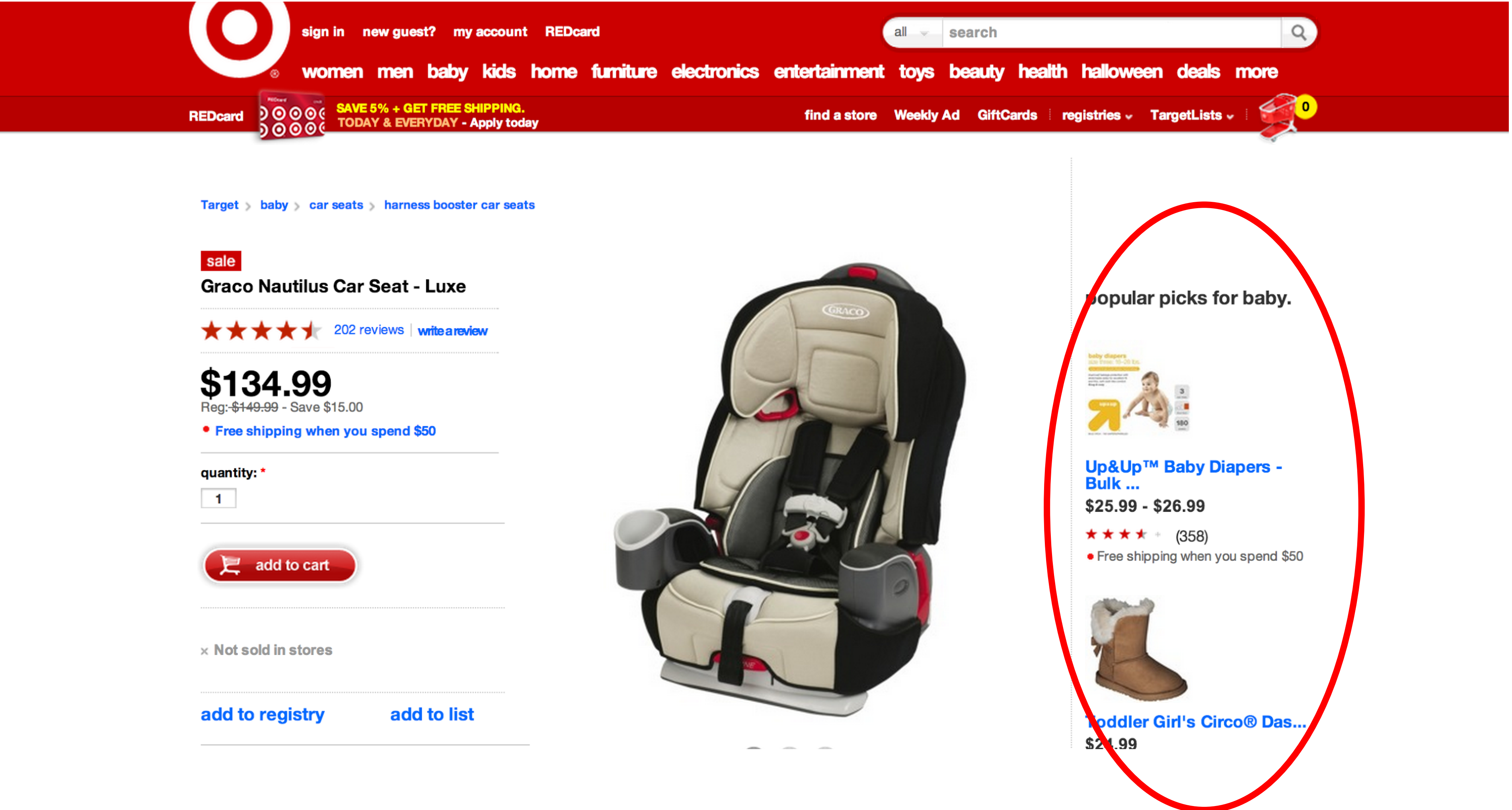
Spire

by example

Tom Switzer
@tixxit

Code: http://github.com/non/spire
Learn: http://typelevel.org

{rr}



sign in
new guest?
my account
REDcard
all
search
women
men
baby
kids
home
furniture
electronics
entertainment
toys
beauty
health
halloween
deals
more
REDcard
SAVE 5% + GET FREE SHIPPING.
TODAY & EVERYDAY - Apply today
find a store
Weekly Ad
GiftCards
registries
TargetLists
0
Target
baby
car seats
harness booster car seats
sale
Graco Nautilus Car Seat - Luxe
202 reviews
write a review
$134.99
Reg: $149.99 - Save $15.00
Free shipping when you spend $50
quantity: *
1
add to cart
Not sold in stores
add to registry
add to list
GRACO
popular picks for baby.
Up&Up™ Baby Diapers - Bulk ...
$25.99 - $26.99
(358)
Free shipping when you spend $50
Toddler Girl's Circo® Das...
$24.99

The Problem

Some things depend on a set of **weights**.

Weights were chosen "**heuristically**."

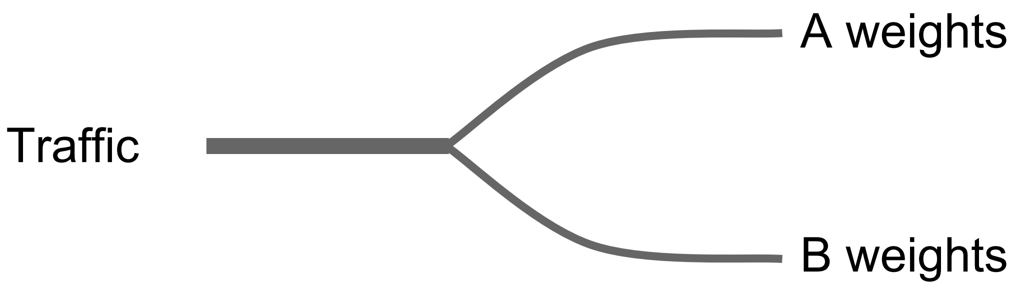
Can we optimize the weights?

Tools

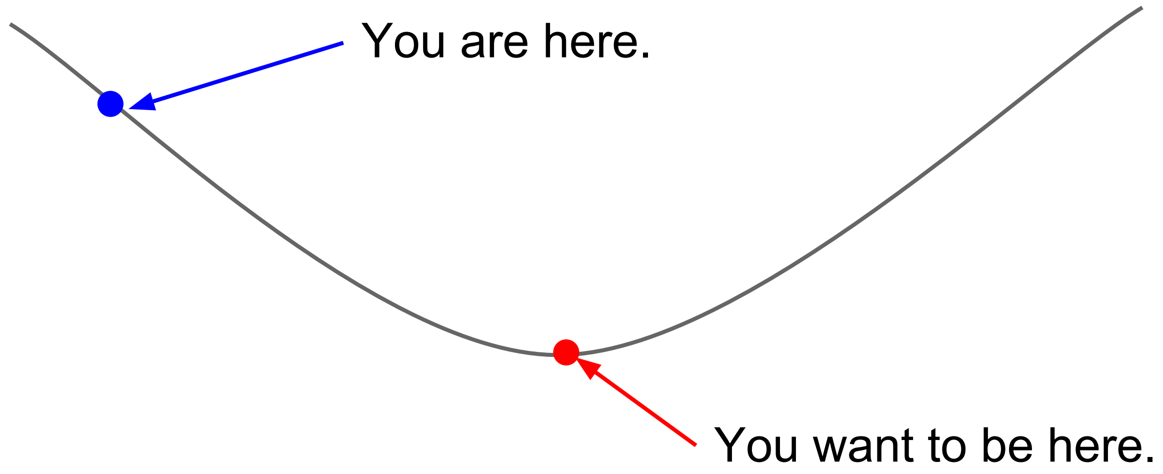
Split traffic to run A/B tests on weights.

Calculate metrics like RPS.

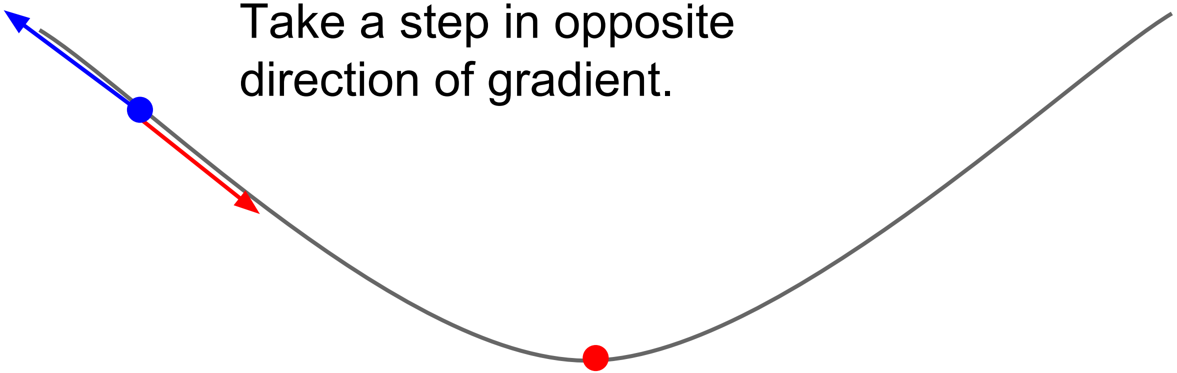
Is this enough?



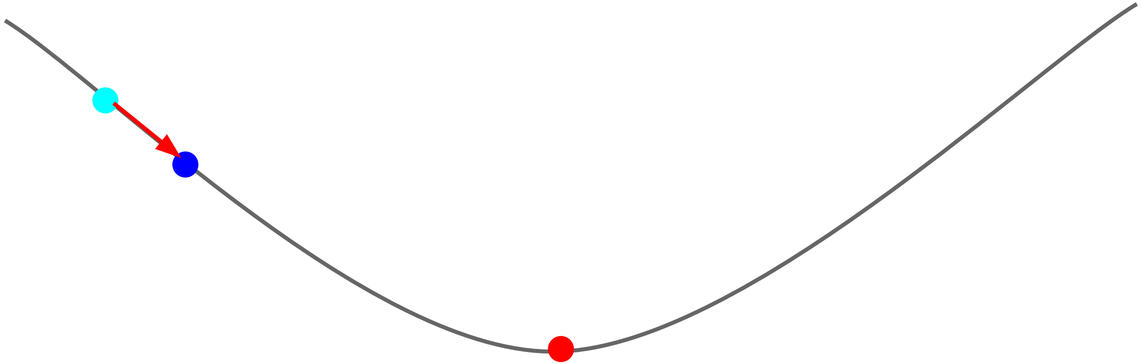
Gradient Descent



Gradient Descent

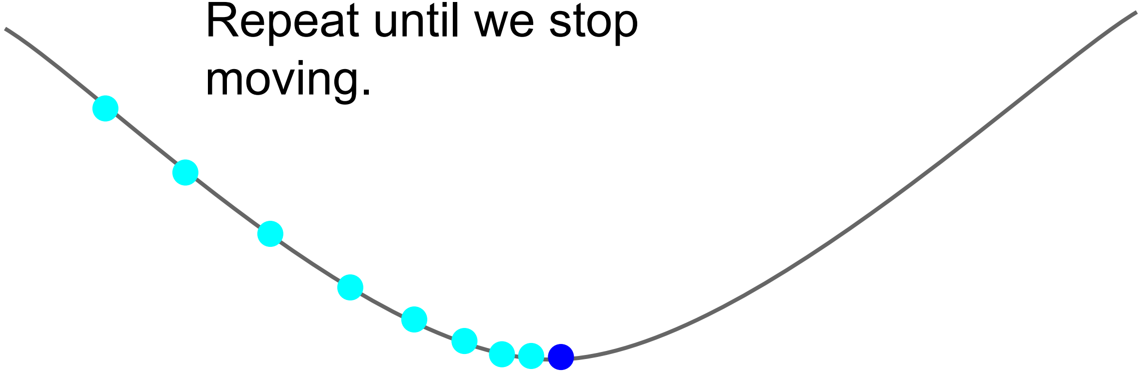


Gradient Descent



Gradient Descent

Repeat until we stop moving.



Requirements for Optimizing F

Need a starting position: w_0

Need a "step" size (learning rate): α

Need to calculate the gradient: ∇F

$$w_{n+1} = w_n - \alpha \nabla F(w_n)$$

In a Language We Understand

```
trait GradientDescent {
  type V = Array[Double]
  type A = Double

  def grad(i: Int, w: V): V

  def nextWeight(i: Int, w: V, a: A): V =
    w - (a *: grad(i, w))
}
```

In a Language We Understand

```
trait GradientDescent {
  type V = Array[Double]
  type A = Double

  def grad(i: Int, w: V): V

  def nextWeight(i: Int, w: V, a: A): V =
    w - (a *: grad(i, w))
}
```

the vector type

the scalar type

$$w_{n+1} = w_n - \alpha \nabla F(w_n)$$

What Spire Gave Us

```
trait GradientDescent {
  type V = Array[Double]
  type A = Double

  def grad(i: Int, w: V): V

  def nextWeight(i: Int, w: V, a: A): V =
    w - (a *: grad(i, w))
}
```

Multiply vector by a scalar

What Spire Gave Us

```
trait GradientDescent {
  type V = Array[Double]
  type A = Double

  def grad(i: Int, w: V): V

  def nextWeight(i: Int, w: V, a: A): V =
    w - (a *: grad(i, w))
}
```

Subtract 2 vectors

Why Does It Work?

`Double` is a “field”

`Array[Double]` is a “vector space”

We can add/subtract vectors.

And multiply them by scalars.


ROAD CLOSED
3 MILES AHEAD
LOCAL TRAFFIC ONLY
DETOUR
22

Algebras Used In This Talk

```
Field[A]
VectorSpace[V,A]
InnerProductSpace[V,A]
Order[A]
```

Field[A]

Has +, -, *, and /.

Instances for Float, Double, BigDecimal, Rational, Complex, etc.

Example

```
import spire.algebra.Field
import spire.implicits._

def mean[A: Field](xs: A*) =
  xs.foldLeft(Field[A].zero)(_ + _) / xs.size

mean(1F, 2F, 3F)
mean(1D, 2D, 3D)
mean(r"1/2", r"2/3", r"4/5")
```

Example

```
import spire.algebra.Field
import spire.implicits._

def mean[A: Field](xs: A*) =
  xs.foldLeft(Field[A].zero)(_ + _) / xs.size

mean(1F, 2F, 3F)
mean(1D, 2D, 3D)
mean(r"1/2", r"2/3", r"4/5")
```

Provides instances and syntax

Example

```
import spire.algebra.Field
import spire.implicits._

def mean[A: Field](xs: A*) =
  xs.foldLeft(Field[A].zero)(_ + _) / xs.size

mean(1F, 2F, 3F)
mean(1D, 2D, 3D)
mean(r"1/2", r"2/3", r"4/5")
```

Short-hand for:

```
implicitly[Field[A]].
zero
```

Example

```scala
import spire.algebra.Field
import spire.implicits._

def mean[A: Field](xs: A*) =
  xs.foldLeft(Field[A].zero)(_ + _) / xs.size

mean(1F, 2F, 3F)
mean(1D, 2D, 3D)
mean(r"1/2", r"2/3", r"4/5")
```

Division is exact

Example

```scala
import spire.algebra.Field
import spire.implicits._


def mean[A: Field](xs: A*) =
  xs.foldLeft(Field[A].zero)(_ + _) / xs.size
```

```scala
mean(1F, 2F, 3F)
mean(1D, 2D, 3D)
mean(r"1/2", r"2/3", r"4/5")
```

Works on Float, Double, Rational, etc.

VectorSpace[V,A]

Operations on vectors V with scalar A.

The scalar A must be a Field.

Instances for Array, Vector, List, etc.

InnerProductSpace[V,A]

Like VectorSpace, but adds an inner product.

```
Vector(1D, 2D) dot Vector(3D, 4D)
```

Order[A]

Provides a total ordering on A.

```
def sort[A: Order](xxs: List[A]) = xxs match {
  case x :: xs =>
    val (ls, rs) = xs.partition(_ <= x)
    sort(ls) ++ (c :: rs)
  case Nil => Nil
}
```

Also has >, ===, cmp, min, etc.


END
DETOUR

Back to the Example

```
trait GradientDescent {
  type V = Array[Double]
  type A = Double

  def grad(i: Int, w: V): V

  def nextWeight(i: Int, w: V, a: A): V =
    w - (a *: grad(i, w))
}
```

Do we really care what V and A are?

Making it Generic

```
trait GradientDescent[V, A] {
  implicit def V: VectorSpace[V, A]
  implicit def A: Field[A]

  def grad(i: Int, w: V): V

  def nextWeight(i: Int, w: V, a: A): V =
    w - (a *: grad(i, w))
}
```

The Optimization Loop

```
trait GradientDescent[V, A] {
  …
  def optimize(a: A): V = …
  …
}
```

The Optimization Loop

```
trait GradientDescent[V, A] {
  implicit def V: InnerProductSpace[V, A]
  implicit def order: Order[A]
  …
  def optimize(a: A): V = …
  …
}
```

We'll need the dot product.

The Optimization Loop

```
def optimize(a: A): (Int, V) = {
  val eps = 0.0000001 * a
  def loop(w0: V, i: Int): (Int, V) = {
    val w1 = nextWeight(i, w0, a)
    val dw = (w0 - w1)
    if ((dw dot dw) > eps) loop(w1, i + 1) else (i, w1)
  }
  loop(V.zero, 0)
}
```

Step/learning rate

The Optimization Loop

```
def optimize(a: A): (Int, V) = {
  val eps = 0.0000001 * a
  def loop(w0: V, i: Int): (Int, V) = {
    val w1 = nextWeight(i, w0, a)
    val dw = (w0 - w1)
    if ((dw dot dw) > eps) loop(w1, i + 1) else (i, w1)
  }
  loop(V.zero, 0)
}
```

Initial weight

The Optimization Loop

```
def optimize(a: A): (Int, V) = {
  val eps = 0.0000001 * a
  def loop(w0: V, i: Int): (Int, V) = {
    val w1 = nextWeight(i, w0, a)
    val dw = (w0 - w1)
    if ((dw dot dw) > eps) loop(w1, i + 1) else (i, w1)
  }
  loop(V.zero, 0)
}
```

Current iteration

The Optimization Loop

Perform 1 step

```
def optimize(a: A): (Int, V) = {
  val eps = 0.0000001 * a
  def loop(w0: V, i: Int): (Int, V) = {
    val w1 = nextWeight(i, w0, a)
    val dw = (w0 - w1)
    if ((dw dot dw) > eps) loop(w1, i + 1) else (i, w1)
  }
  loop(V.zero, 0)
}
```

The Optimization Loop

```
def optimize(a: A): (Int, V) = {
  val eps = 0.0000001 * a
  def loop(w0: V, i: Int): (Int, V) = {
    val w1 = nextWeight(i, w0, a)
    val dw = (w0 - w1)
    if ((dw dot dw) > eps) loop(w1, i + 1) else (i, w1)
  }
  loop(V.zero, 0)
}
```

Check if we've "converged"

Testing it Out: Linear Least Squares

Problem: Fit a line to a set of points.

- Line: y = *m*x + *b*
- Points: [(x_0,x_1), (x_0,x_1), … , (x_n,x_n)]

Minimize: $\sum(mx_i + b - y_i)^2$

Gradient Descent Requirements

✓ Need a starting position.

✓ Need a “step” size (learning rate).

✓ Need to calculate the **gradient**.

The Derivative: 2 Dimensions

$F(w) = \sum(mx_i + b - y_i)^2$

$f_i(m, b) = (mx_i + b - y_i)^2$

$f'_i(m, b) = [2x_i(mx_i + b - y_i), 2(mx_i + b - y_i)]$

$\nabla F(w) = \sum[2x_i(mx_i + b - y_i), 2(mx_i + b - y_i)]$

The Derivative: N Dimensions

$p = [x_0, x_1, \ldots, x_n, y]$

$\beta = [x_0, x_1, \ldots, x_n, 1]$

$f_p(w) = (\beta \cdot w - y)^2$

$f'_p(w) = [2x_0(\beta \cdot w - y), 2x_1(\beta \cdot w - y), \ldots, 2(\beta \cdot w - y)]$

$\quad = \mathbf{2(\beta \cdot w - y)\beta}$

$\nabla F(w) = \sum 2(\beta_p \cdot w - y_p)\beta_p$

In Spire

```scala
class LeastSquares[V, A](betas: Iterable[V], ys: Iterable[A])
    extends GradientDescent[V, A] {

  val grads: Iterable[V => V] =
    (betas zip ys) map {  case (b, y) =>
      { (w: V) => (2 * ((b dot w) - y)) *: b }
    }

  def grad(i: Int, v: V): V =
    grads.foldLeft(V.zero)(_ + _(v)) :/ grads.size
}
```

In Spire

```
class LeastSquares[V, A](betas: Iterable[V], ys: Iterable[A])
    extends GradientDescent[V, A] {

  val grads: Iterable[V => V] =
    (betas zip ys) map {  case (b, y) =>
      { (w: V) => (2 * ((b dot w) - y)) *: b }
    }

  def grad(i: Int, v: V): V =
    grads.foldLeft(V.zero)(_ + _(v)) :/ grads.size
}
```

$2(\beta \cdot w - y)\beta$

In Spire

```
class LeastSquares[V, A](betas: Iterable[V], ys: Iterable[A])
    extends GradientDescent[V, A] {

  val grads: Iterable[V => V] =
    (betas zip ys) map { case (b, y) =>
      { (w: V) => (2 * ((b dot w) - y)) *: b }
    }

  def grad(i: Int, v: V): V =
    grads.foldLeft(V.zero)(_ + _(v)) :/ grads.size
}
```

Scalar division

Testing it Out

```
import spire.random.Dist

def linear[A: ...](coeffs: Array[A]): Dist[Array[A]] = …

val points = linear(Array(0.7, 2.3)).sample(40)
val coeffs = LeastSquares(points).optimize(0.01)
```

Testing it Out

Random distributions

```
import spire.random.Dist

def linear[A: ...](coeffs: Array[A]): Dist[Array[A]] = …

val points = linear(Array(0.7, 2.3)).sample(40)
val coeffs = LeastSquares(points).optimize(0.01)
```

Create a sample with 40 points

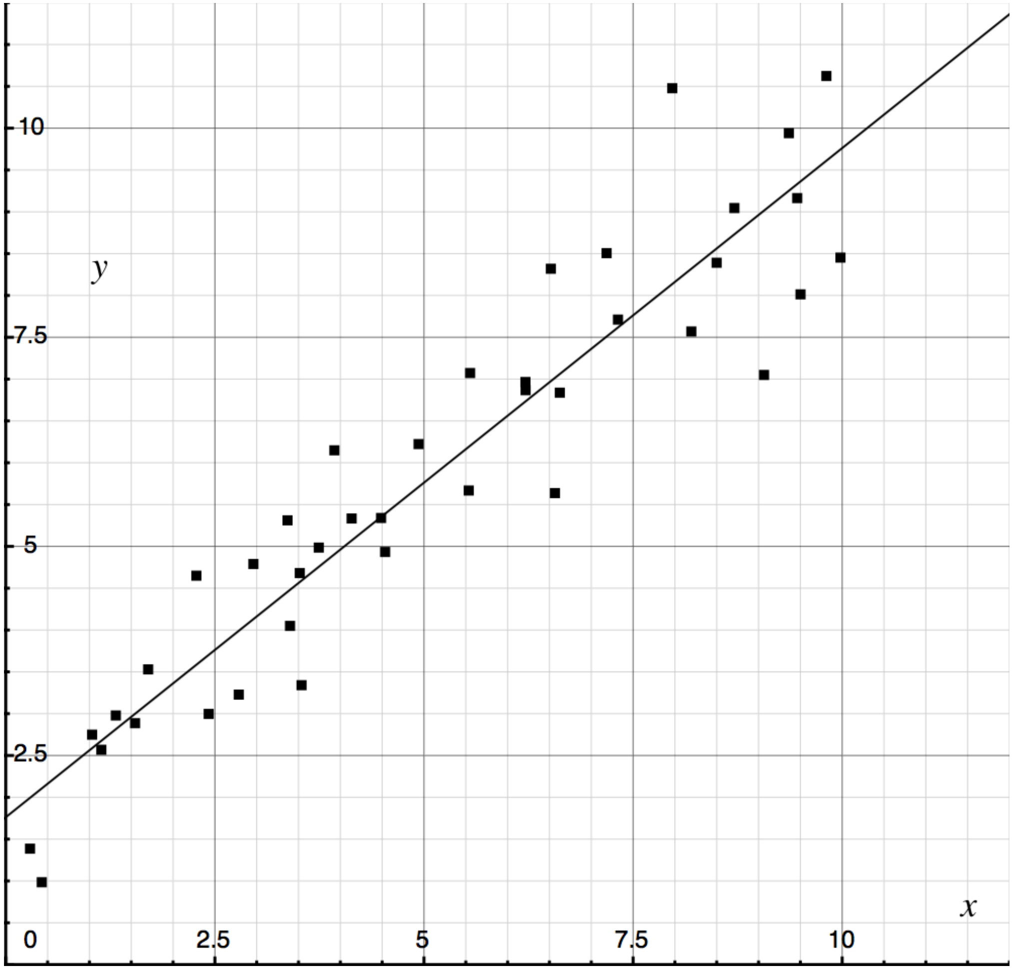
Testing it Out

```
import spire.random.Dist


def linear[A: ...](coeffs: Array[A]): Dist[Array[A]] = …


val points = linear(Array(0.7, 2.3)).sample(40)
val coeffs = LeastSquares(points).optimize(0.01)
```

Optimize using gradient descent!



$$f(x) = 0.8x + 1.76$$

Back to Our Actual Problem

Optimize “weights” in personalization product.

Can we use gradient descent?

Requirements

✓ Need a starting position.

✓ Need a “step” size (learning rate).

✗ Need to calculate the **gradient**.

Requirements

✓ Need a starting position.
✓ Need a “step” size (learning rate).
✗ Need to calculate the **gradient**.

Unknown F, unknown ∇F

Approximate the Gradient

Sample around w.

Fit hyperplane (line) to samples.

The coefficients approximate the gradient.

Approximate the Gradient

Sample around w.

Fit hyperplane (line) to samples.

The coefficients approximate the gradient.

Approximate the Gradient

```
abstract class SampledGradientDescent[ A: ClassTag: Uniform]
    (f: (Int, Array[A]) => A, llsLearningRate: A)
    extends GradientDescent[Array[A], A] {
  def timespan(i: Int): Range
  def sample(v: Array[A], d: A): Array[Array[A]] = …
  def grad(i: Int, v: Array[A]): Array[A] = …
}
```

Approximate the Gradient

```
…
def sample(v: Array[A], d: A): Array[Array[A]] = {
  import Dist._
  val noise = array(v.length)(uniform[A](-d, d))
  (constant(v) + noise)
      .sample[Vector](v.length + 1)
      .toArray
}
…
```

Approximate the Gradient

```
…
def sample(v: Array[A], d: A): Array[Array[A]] = {
  import Dist._
  val noise = array(v.length)(uniform[A](-d, d))
  (constant(v) + noise)
      .sample[Vector](v.length + 1)
      .toArray
}
…
```

Methods from Dist._

Approximate the Gradient

…

```
def sample(v: Array[A], d: A): Array[Array[A]] = {
  import Dist._
  val noise = array(v.length)(uniform[A](-d, d))
  (constant(v) + noise)
      .sample[Vector](v.length + 1)
      .toArray

}
```

Adding “distributions” together

Approximate the Gradient

…

```
def grad(i: Int, v: Array[A]): Array[A] = {
  val vs = sample(v, llsLearningRate * 4)
  val g = LeastSquares(vs, vs map { v =>
    timespan(i).map(f(_, v)).qmean
  }).optimize(llsLearningRate)._2.init
}
```

…

Approximate the Gradient

…

```
def grad(i: Int, v: Array[A]): Array[A] = {
  val vs = sample(v, llsLearningRate * 4)
  val g = LeastSquares(vs, vs map { v =>
    timespan(i).map(f(_, v)).qmean
  }).optimize(llsLearningRate)._2.init
}
```

…

Use our linear least squares
implementation to fit hyperplane!

Approximate the Gradient

…

```
def grad(i: Int, v: Array[A]): Array[A] = {
  val vs = sample(v, llsLearningRate * 4)
  val g = LeastSquares(vs, vs map { v =>
    timespan(i).map(f(_, v)).qmean
  }).optimize(llsLearningRate)._2.init
}
```

…

Average over a span of time

Approximate the Gradient

…

```
def grad(i: Int, v: Array[A]): Array[A] = {
  val vs = sample(v, llsLearningRate * 4)
  val g = LeastSquares(vs, vs map { v =>
    timespan(i).map(f(_, v)).qmean
  }).optimize(llsLearningRate)._2.init
}
```

…

Drop the value axis intercept

So… What’s the “timespan” Thing?

People shop differently on different days.

Time is an implicit parameter.

For testing, make it explicit.

A New Challenger Arrives

Evaluating F at w:

- Split traffic: some see weights w
- Wait X hours/days/weeks
- Calculate our metric

A New Challenger Arrives

Evaluating F at w:

- Split traffic: some see weights w
- Wait X hours/days/weeks
- Calculate our metric

Convergence may take a while!

A New Challenger Arrives

Evaluating F at w:

- Split traffic: some see weights w
- Wait X hours/days/weeks
- Calculate our metric

How does X affect the ***time*** to convergence?

Test It Out!

Our Assumption:

- F varies periodically over a week

Test It Out!

Our Assumption:

- F varies periodically over a week

Strategies:

- Fix X to 1 or 7
- Linear ramp-up from 1-7
- Run it at 1 at first, then jump to 7

Strategy 1: Fix X

```
def period: Int


def timespan(i: Int): Range =
  (i * period) until ((i + 1) * period)
```

Strategy 2: Ramp-up X

```
def iters: Int
def max: Int
def timespan(i: Int): Range = {
  val p = spire.math.min(max, (i / iters) + 1)
  val j = i - (p - 1) * iters
  val start = p * j + iters * (p * (p - 1)) / 2
  val end = start + p
  start until end
}
```

Strategy 2: Ramp-up X

Note: not scala.math!

```
def iters: Int
def max: Int
def timespan(i: Int): Range = {
  val p = spire.math.min(max, (i / iters) + 1)
  val j = i - (p - 1) * iters
  val start = p * j + iters * (p * (p - 1)) / 2
  val end = start + p
  start until end
}
```

Strategy 3: Jump Up

```
def fst: Int
def snd: Int
def jump: Int
def timespan(i: Int): Range = {
  if (i < jump) {
    (i * fst) until ((i + 1) * fst)
  } else {
    val start = jump * fst + (i - jump) * snd
    start until (start + snd)
  }
}
```

The Data: Creating a Cyclic F

```
def cyclic[A: Field: Order](p: Int,
    ps: (Int, Array[A] => A)*): (Int, Array[A]) => A = {
  val (indices, fs) = ps.toArray.qsortedBy(_._1).unzip

  def eval(t: Int, v: Array[A]): A = {
    val i = indices.qsearch(t)  // Note, along w/ qselect and qsort.
    …
  }

  { (t, v) => eval(t % period, v) }
}
```

The Data: Creating a Cyclic F

```
def cyclic[A: Field: Order](p: Int,
    ps: (Int, Array[A] => A)*): (Int, Array[A]) => A = {
  val (indices, fs) = ps.toArray.qsortedBy(_._1).unzip

  def eval(t: Int, v: Array[A]): A = {
    val i = indices.qsearch(t)  // Note, along w/ qselect and qsort.
    …
  }

  { (t, v) => eval(t % period, v) }
}
```

Sorting!

The Data: Creating a Cyclic F

```
def cyclic[A: Field: Order](p: Int,
    ps: (Int, Array[A] => A)*): (Int, Array[A]) => A = {
  val (indices, fs) = ps.toArray.qsortedBy(_._1).unzip

  def eval(t: Int, v: Array[A]): A = {
    val i = indices.qsearch(t) // Note, along w/ qselect and qsort.
    …
  }

  { (t, v) => eval(t % period, v) }
}
```

Binary search!

Other Methods on Seq and Array

Sorting (`xs.qsort`)

Searching (`xs.qsearch(x)`)

- Never implement binary search again

Selection (`xs.qselect(k)`)

- Linear time selection

A Week in Code

```
def quad[A: Field](a: Double, h: Double, k: Double): Array[A] => A =
  { v => a * ((v(0) - h) ** 2) + k }

def weekly[A: Field: Order]: (Int, Array[A]) => A =
  cyclic(7, 0 -> quad(2  , 2  , 5),
            1 -> quad(1.5, 2.5, 3),
            3 -> quad(1  , 3  , 1),
            5 -> quad(1.5, 2.5, 3),
            6 -> quad(2  , 2  , 5))
```

A Week in Code

```
def quad[A: Field](a: Double, h: Double, k: Double): Array[A] => A =
  { v => a * ((v(0) - h) ** 2) + k }

def weekly[A: Field: Order]: (Int, Array[A]) => A =
  cyclic(7, 0 -> quad(2  , 2  , 5),
            1 -> quad(1.5, 2.5, 3),
            3 -> quad(1  , 3  , 1),
            5 -> quad(1.5, 2.5, 3),
            6 -> quad(2  , 2  , 5))
```

The weekend

A Week in Code

```
def quad[A: Field](a: Double, h: Double, k: Double): Array[A] => A =
  { v => a * ((v(0) - h) ** 2) + k }


def weekly[A: Field: Order]: (Int, Array[A]) => A =
  cyclic(7, 0 -> quad(2  , 2  , 5),
            1 -> quad(1.5, 2.5, 3),
            3 -> quad(1  , 3  , 1),
            5 -> quad(1.5, 2.5, 3),
            6 -> quad(2  , 2  , 5))
```

The weekend

Minimum over the week is ~2.3

What We Want

Minimize # of days to convergence.

Minimize error of result.

What We Want

Minimize # of days to convergence.
Minimize error of result.

These goals may compete with each other!

Results: No Free Lunch

Strategy	# of iterations	# of days	Converged To	RMSE
Fixed: 7 days	475	3325	2.245	0.73
Fixed: 1 day	1189	1189	2.21	**1.23**
Ramp-up: 1-7	727	2659	2.23	0.99
Jump: 200	569	2789	2.27	0.71
Jump: 300	559	2119	2.26	0.88
Jump: 400	727	2659	2.23	0.99

Results: No Free Lunch

Strategy	# of iterations	# of days	Converged To	RMSE
Fixed: 7 days	475	3325	2.245	0.73
Fixed: 1 day	1189	1189	2.21	1.23
Ramp-up: 1-7	727	2659	2.23	0.99
Jump: 200	569	2789	2.27	0.71
Jump: 300	559	2119	2.26	0.88
Jump: 400	727	2659	2.23	0.99

Questions?