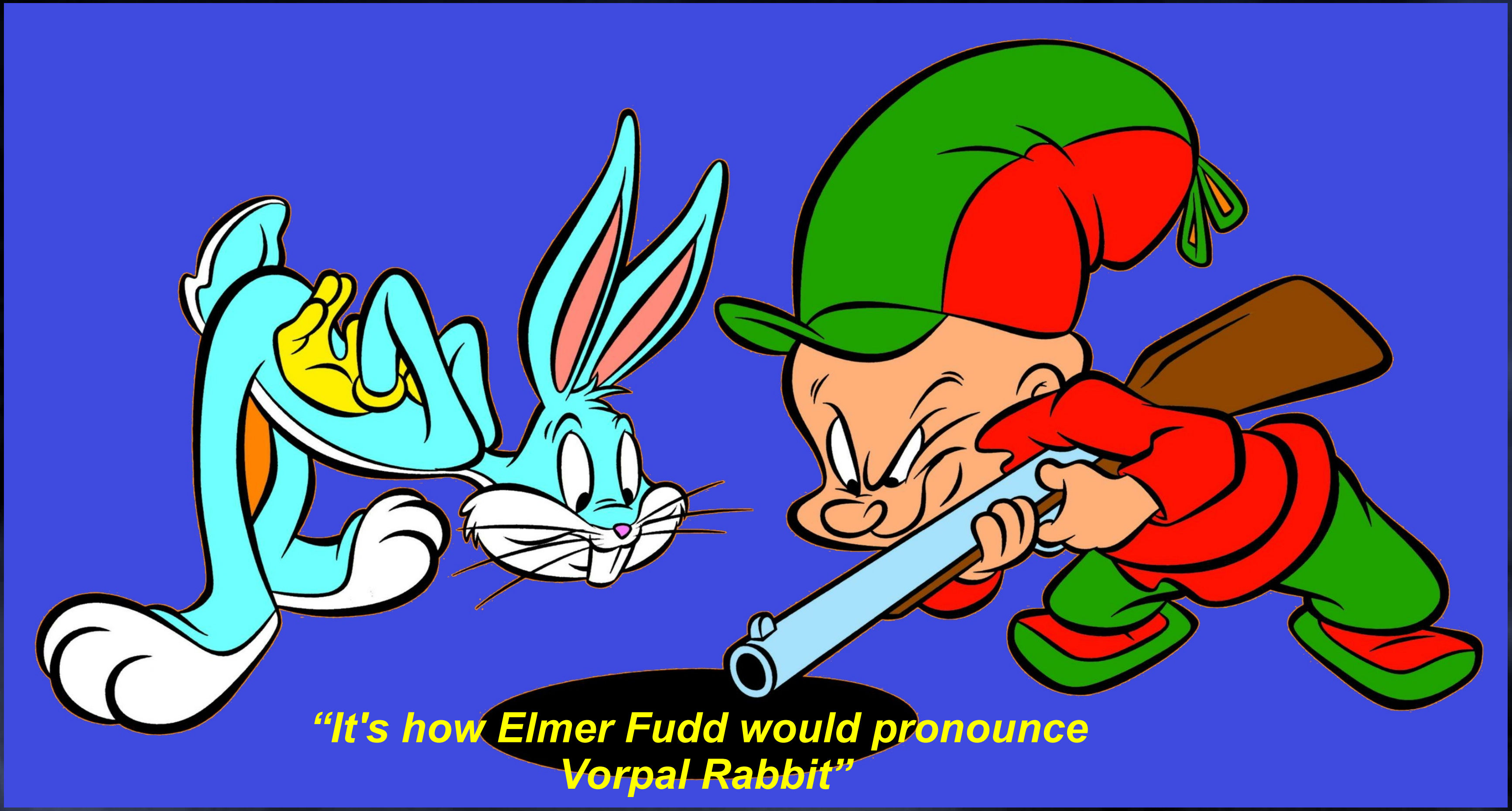
Vowpal Wabbit

(fast & scalable machine-learning)

ariel faigon



What is Machine Learning?

In a nutshell:

- The process of a computer (self) learning from data

Two types of learning:

- ***Supervised:*** *learning from labeled (answered) examples*
- ***Unsupervised:*** *no labels, e.g. clustering, segmentation*

Supervised Machine Learning

$$y = f(x1, x2, \ldots, xN)$$

y : *output/result we're interested in*

X1, … , xN : *inputs we know/have*

Supervised Machine Learning

$$y = f(x1, x2, \ldots, xN)$$

Classic/traditional computer science:

- *We have:* *x1, … , xN* *(the input)*
- *We want:* *y* *(the output)*

We spend a lot of time and effort thinking and coding ***f***

We call ***f*** *"the algorithm"*

Supervised Machine Learning

$$y = f(x1, x2, \ldots, xN)$$

In more modern / AI-ish computer science:

- *We have:* $x1, \ldots, xN$
- *We have:* y

We have a lot of past data, i.e. many instances (examples) of the relation $y = f(x1, \ldots, xN)$ *between input and output*

Supervised Machine Learning

$$y = f(x1, x2, \ldots, xN)$$

*We have a **lot** of past data, i.e. many instances (examples) of the relation $y = ?(x1, \ldots, xN)$ between input and output*

So why not let the computer find f for us ?

When to use supervised ML?

$$y = f(x1, x2, \ldots, xN)$$

3 necessary and sufficient conditions:

1) We have a goal/target, or question y which we want to predict or estimate

2) We have lots of data including y's and related X_i's:
   i.e: tons of past examples $y = f(x1, \ldots, xN)$

3) We have no obvious algorithm f linking y to $(x1, \ldots, xN)$

Enter the vowpal wabbit

- Fast, highly scalable, flexible, online learner
- Open source and Free (BSD License)
- Originally by John Langford
- Yahoo! & Microsoft research

Vorpal (adj): deadly
(Invented by Lewis Carroll to describe a sword)

Rabbit (noun): mammal associated with speed

vowpal wabbit

- Written in C/C++
- Linux, Mac OS-X, Windows
- Both a library & command-line utility
- Source & documentation on github + wiki
- Growing community of developers & users

What can vw do?

Solve several problem types (many via reductions):

- Linear regression
- Classification (+ multi-class) [using multiple reductions/strategies]
- Matrix factorization (SVD like)
- LDA (Latent Dirichlet Allocation)
- More ...

vowpal wabbit

Supported optimization strategies

(method used to find the gradient/direction towards the optimum/minimum error):

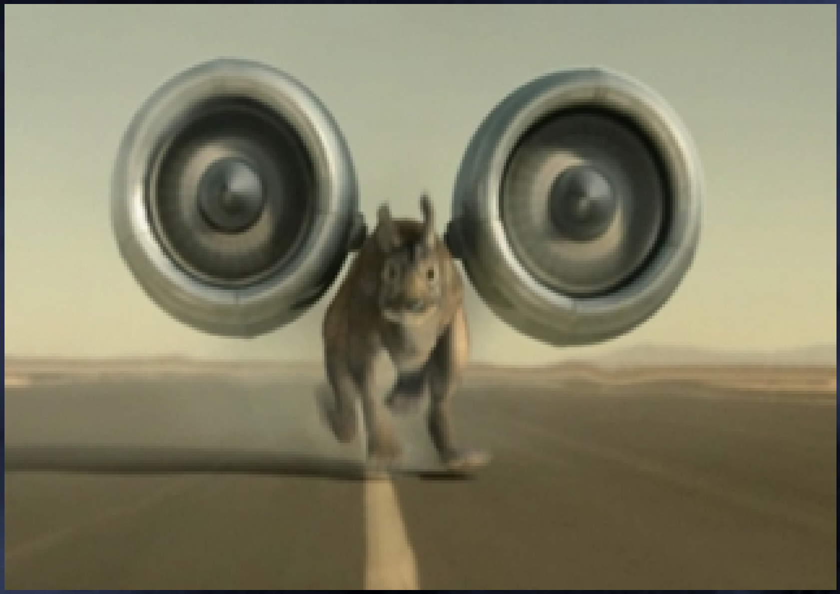
- ***Stochastic Gradient Descent (SGD)***
- ***BFGS***
- ***Conjugate Gradient***

vowpal wabbit

During learning,
which error are we trying to optimize-for (minimize)?

VW supports multiple loss (error) functions:

- squared
- quantile
- logistic
- hinge

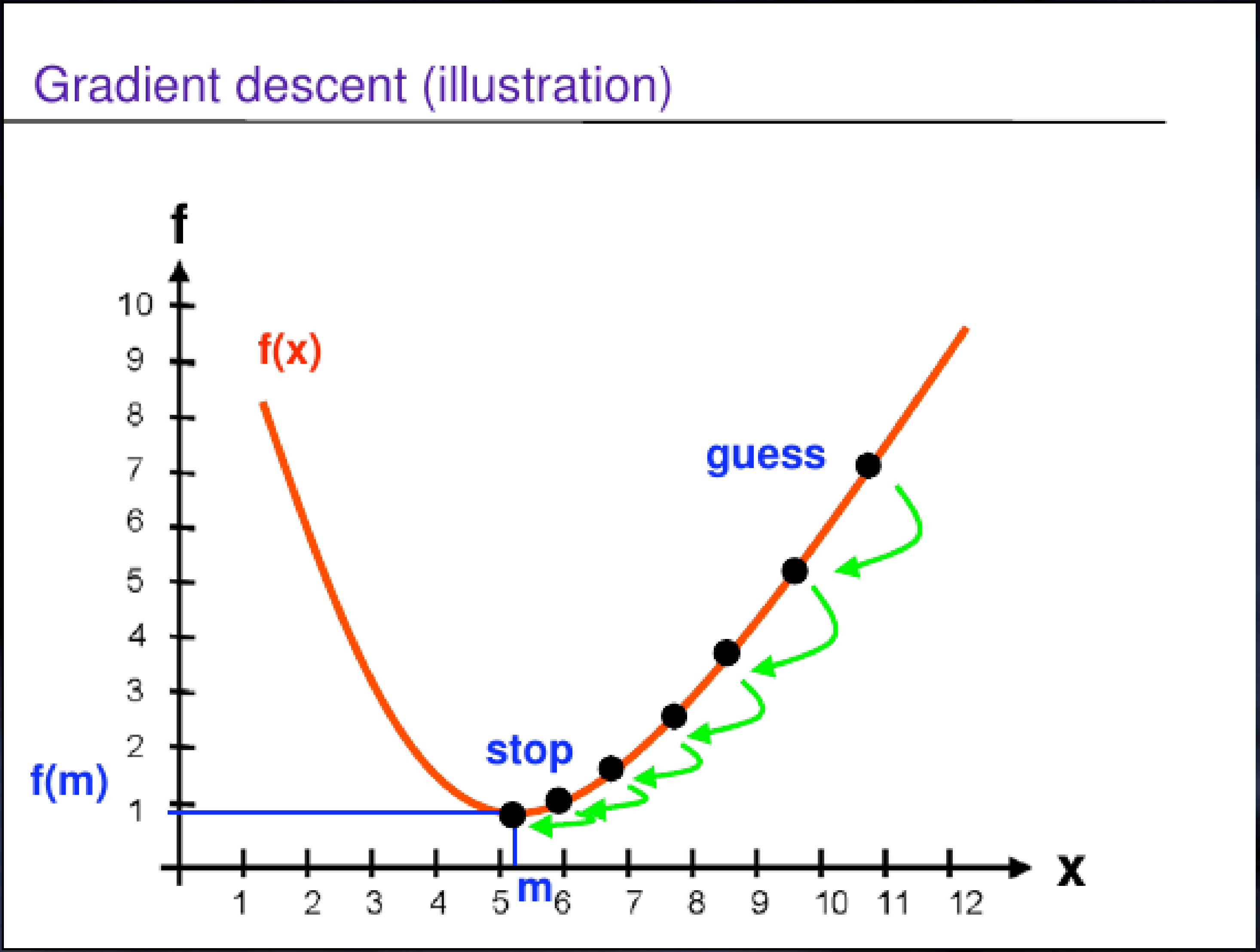


vowpal wabbit

Core algorithm (in inner loop):

- Supervised machine learning
- Online stochastic gradient descent
- With a 3-way iterative update:
  - --adaptive
  - --invariant
  - --normalized

Gradient Descent in a nutshell

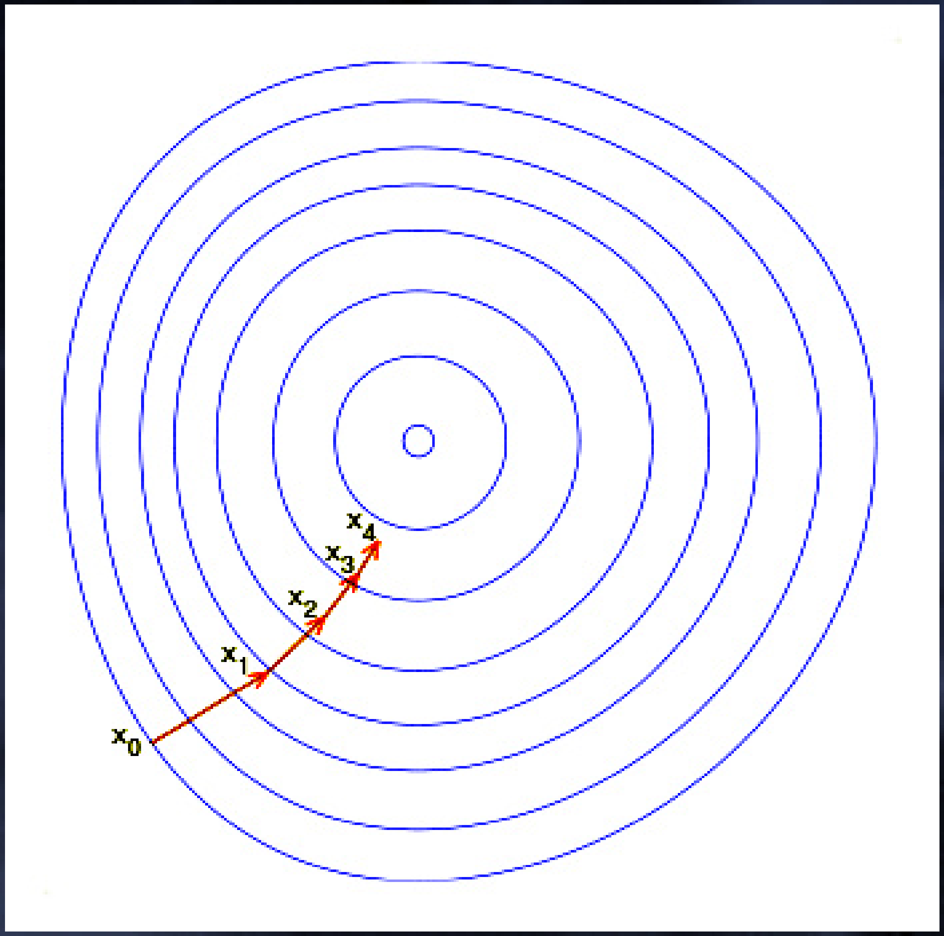


Gradient Descent in a nutshell

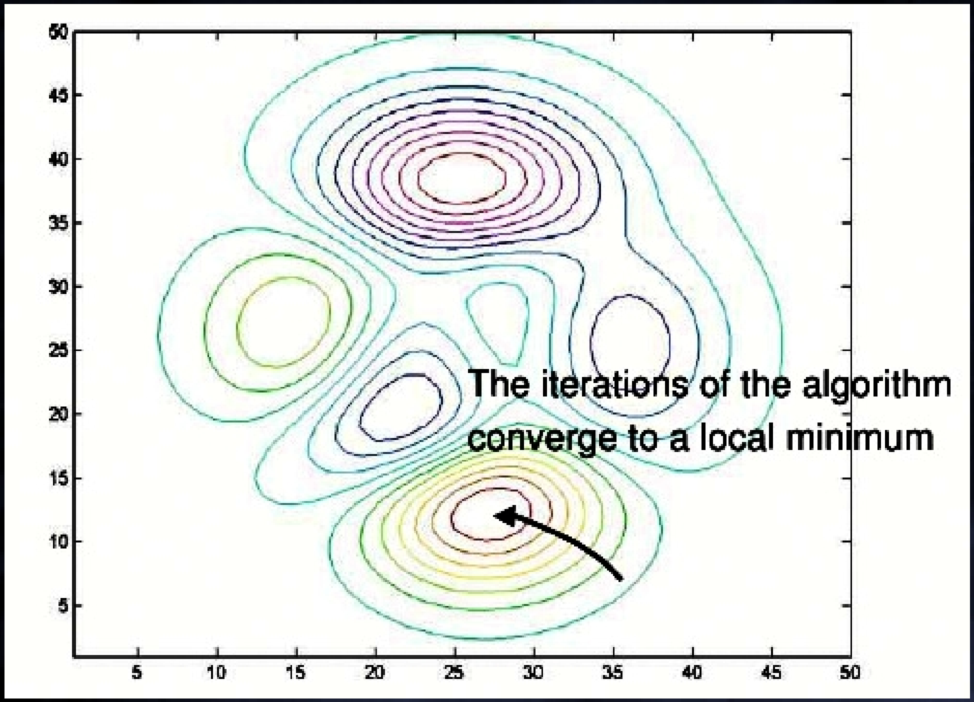
from 1D (line) to 2D (plane)
find bottom (minimum) of valley:

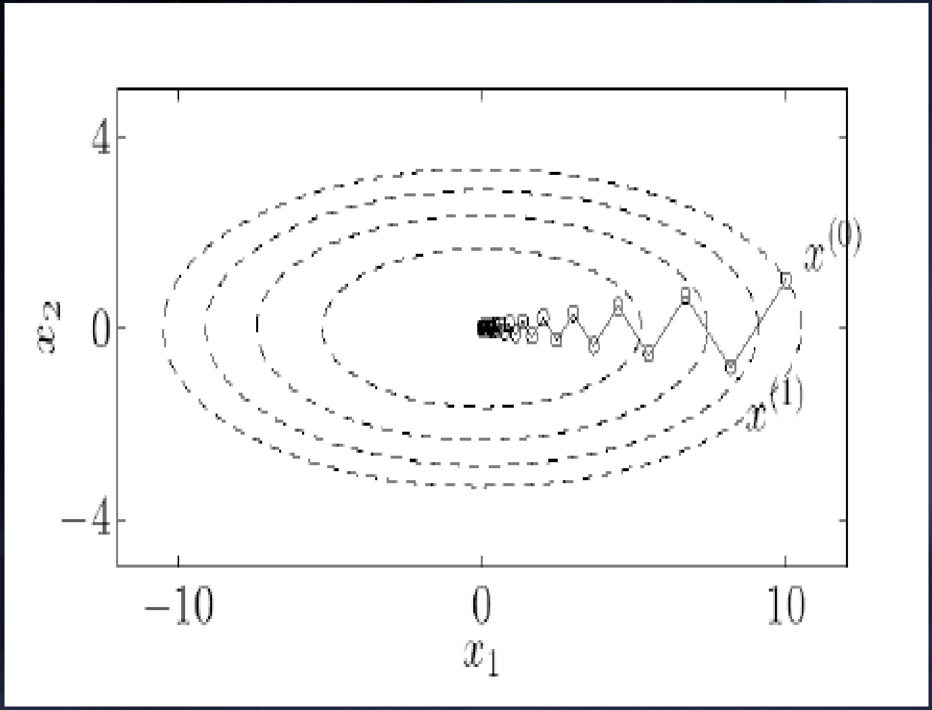
We don't see
the whole picture,
only a local one.

Sensible direction
is along
steepest gradient



Gradient Descent: challenges & issues





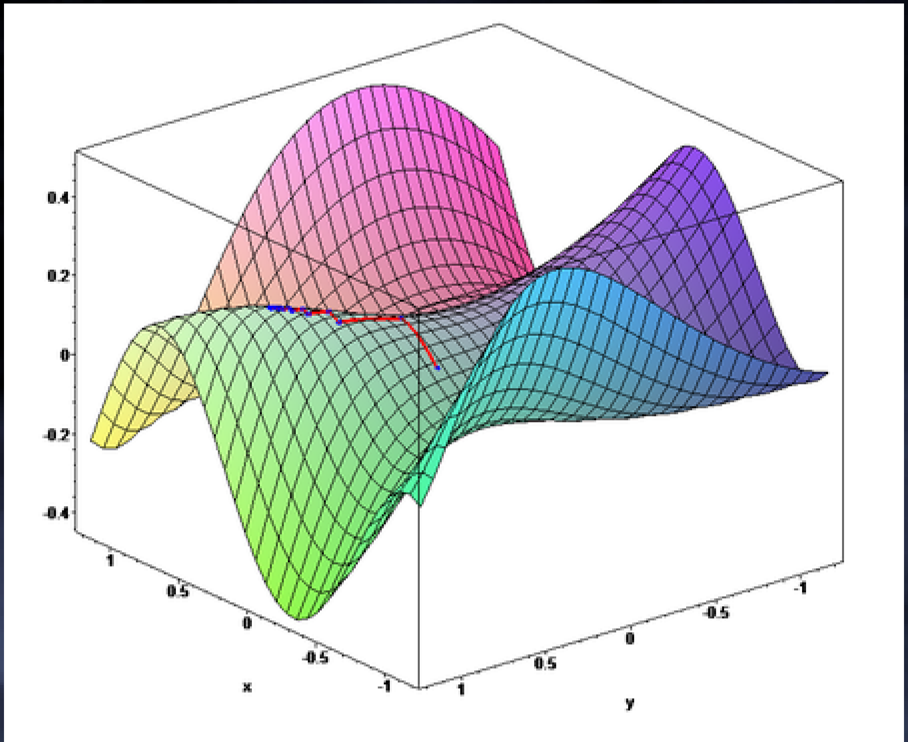
Local vs global optimum

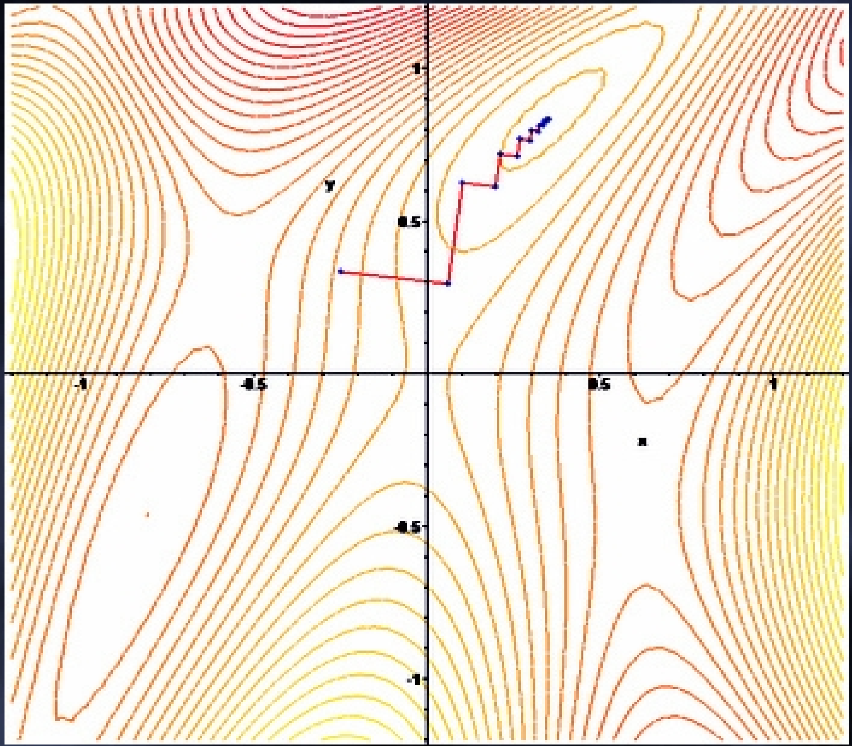
Non normalized steps

Step too big / overshoot

Gradient Descent: challenges & issues

- Saddles
- Unscaled & non-continuous dimensions
- Much higher dimensions than 2D

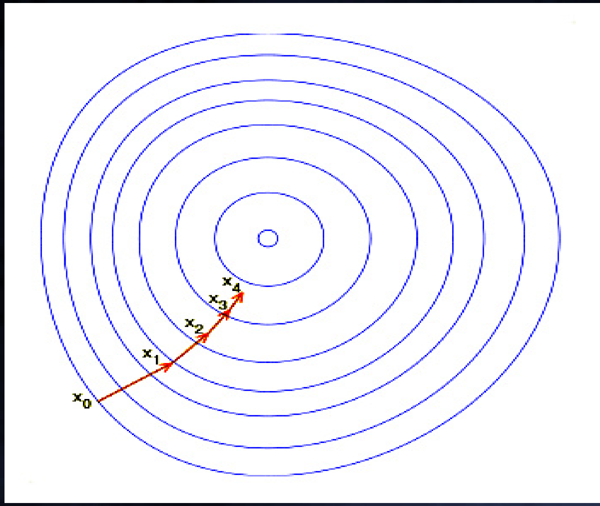


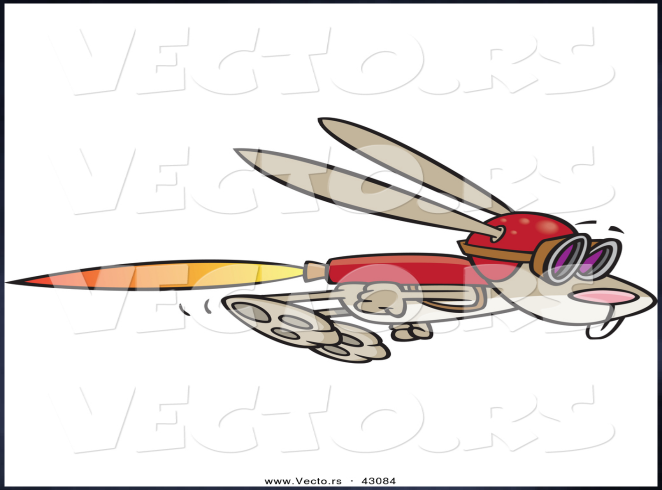


What sets vw apart?

SGD on steroids:

- ***invariant***
- ***adaptive***
- ***normalized***





What sets vw apart?

SGD on steroids

Auto-adaptive to feature scale, importance & rarity:

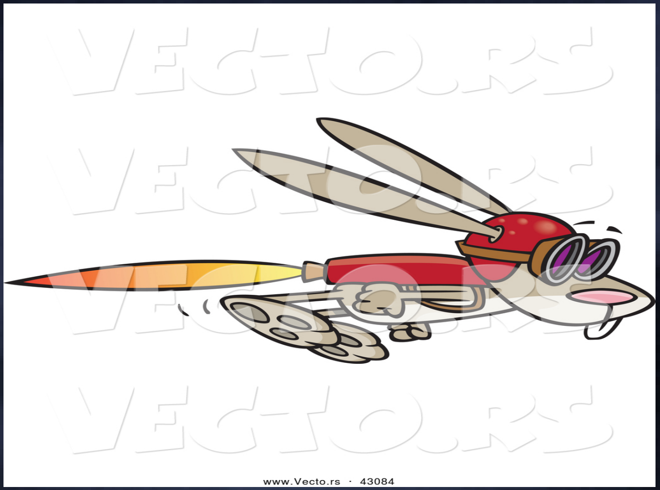
- *No need to pre-normalize feature value ranges*
- *Takes care of unimportant vs important features*
- *Adaptive & separate per feature learning rates*

feature = one dimension of input

What sets vw apart?

Speed and scalability:

- Unlimited data-size (online learning)
- ~5M features/second on my desktop
- Oct 2011 learning speed record:
  10^{12} (tera) features in 1h on 1k node cluster



What sets vw apart?

The "hash trick": `num:6.3  color=red  age<7y`

- *Feature names are hashed fast (murmur hash 32)*
- *Hash result is index into weight-vector*
- *No hash-map table is maintained internally*
- *No attempt to deal with hash-collisions*

What sets vw apart?

Very flexible input format:

- *Accepts sparse data-sets, missing data*
- *Can mix numeric, categorical/boolean features in natural-language like manner (via the hash trick):*

```
size:6.3  color=turquoise  age<7y  is_cool
```

What sets vw apart?

Name spaces in data-sets:

- *Designed to allow feature-crossing*
- *Useful in recommender systems*
- *e.g. used in matrix factorization*
- *Self documenting:*

```
1 |user age:14  state=CA … |item books       price:12.5 …
0 |user age:37  state=OR … |item electronics price:59 …
```

Crossing users with items:

```
$ vw -q ui did_user_buy_item.train
```

What sets vw apart?

Over-fit resistant:

- On-line learning: learns as it goes

  - Compute **y** from x_i ... based on current weigths

  - Compare with actual (example) **y**

  - Compute error

  - Update model (per feature weights)

  Advance to next example & repeat...

- New data is always “out of sample” (exception: multiple passes)

What sets vw apart?

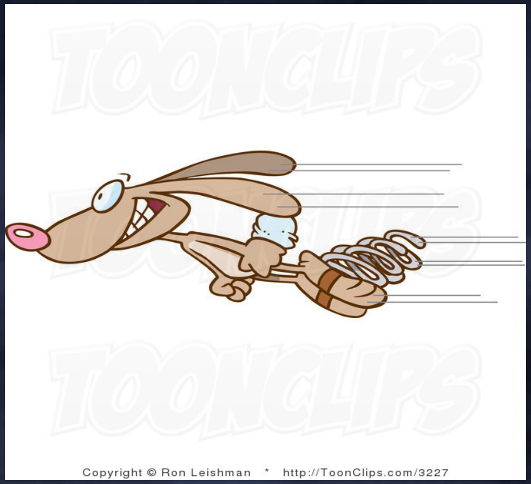
Over-fit resistant (cont.):

- *Data is always "out of sample" …*
- *So model error estimate is realistic (test like)*
- *Model is linear (simple) – hard to overfit*
- *No need to train vs test or K-fold cross-validate*

Biggest weakness

Learns simple models

- Can be partially mitigated by:
  - Quadratic / cubic (-q / --cubic options) to automatically cross features on-the-fly
  - Single hidden layer neural-net –nn <N>
  - Early feature transform (ala GAM)



Demo

(How to separate a signal from surrounding noise)

Demo

Step 1:

Generate a random train-set: Y = a + 2b - 5c + 7

```
$ random-poly -n 50000   a + 2b - 5c + 7  >  r.train
```

Demo

Random train-set: $Y = a + 2b - 5c + 7$

```
$ random-poly  -n 50000   a + 2b - 5c + 7  >  r.train
```

Quiz:

Assume random values for (a, b, c) are in the range [0 , 1)

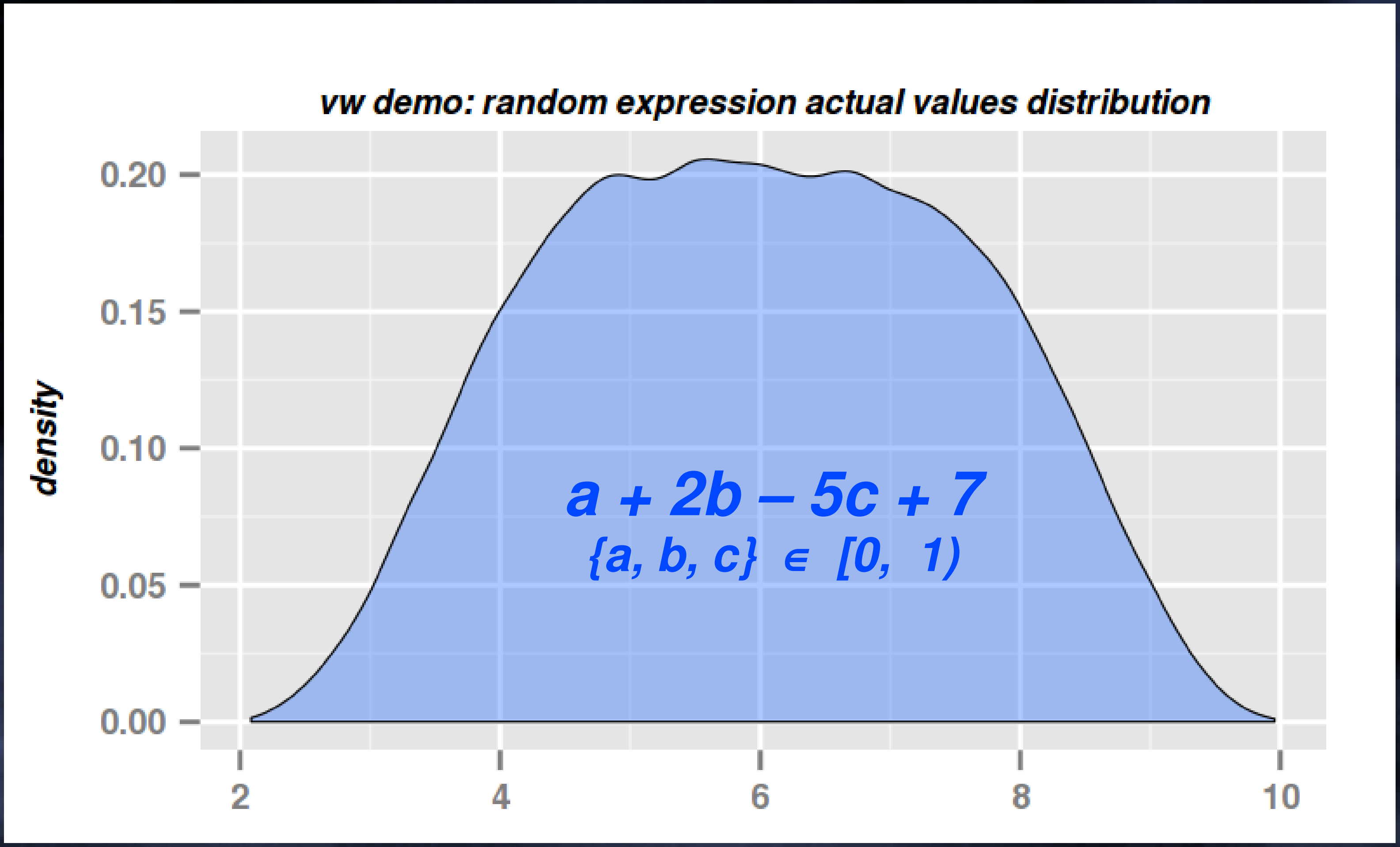
What's the min and max of the expression?

What's the distribution of the expression?

getting familiar with our data-set

Random train-set: $Y = a + 2b - 5c + 7$

Min and max of Y: (2, 10)
Density distribution of Y (related to, but not Irwin-Hall):



Demo

Step 2:

Learn from the data & build a model:

$ vw -l 5 r.train -f r.model

Quiz: how long should it take to learn from
(50,000 x 4) (examples x features)?

Demo

Step 2:

$ vw -l 5 r.train -f r.model

Q: how long should it take to learn from (50,000 x 4) (examples x features)?

A: about 1 /10th (0.1) of a second on my little low-end notebook

Demo

Step 2 (training-output / convergence)

$ vw -l 5 r.train -f r.model

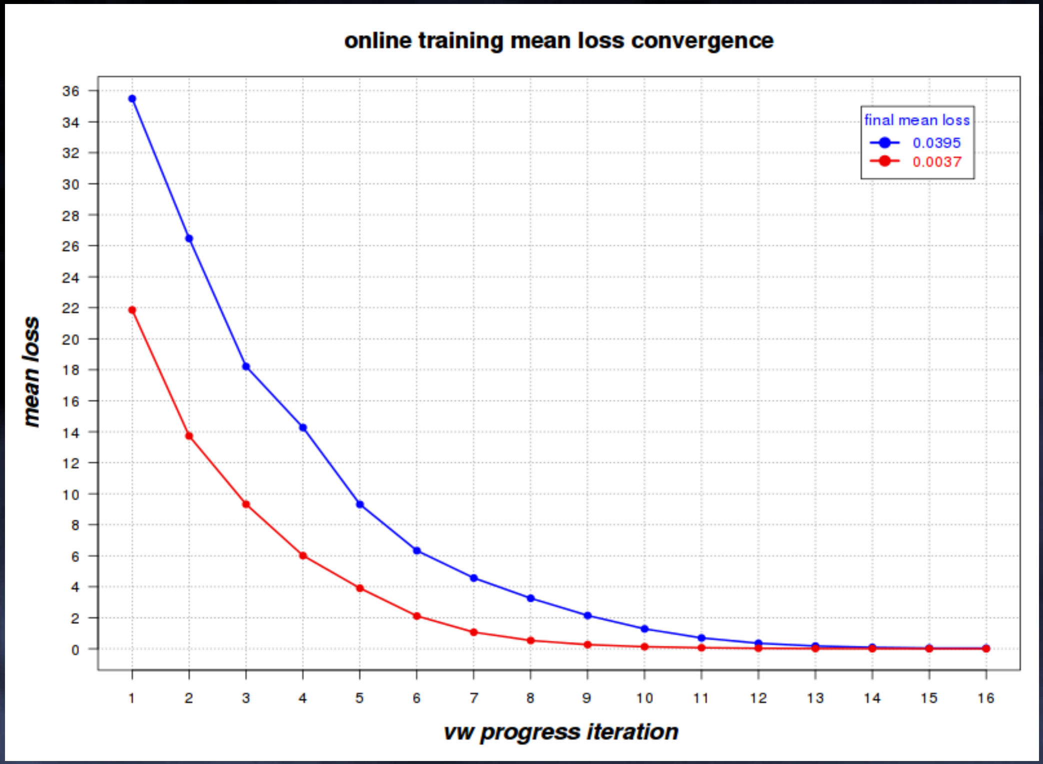
```
average    since        example     example  current  current  current
loss       last         counter      weight    label  predict features
22.438119  22.438119          3         3.0   4.0602   4.4325        4
13.288925  4.139732           6         6.0   6.5879   7.9563        4
10.334829  6.789914          11        11.0   4.8486   8.1888        4
6.939150   3.543470          22        22.0   6.0161   6.6145        4
4.358768   1.778385          44        44.0   8.4484   6.7984        4
2.777721   1.159907          87        87.0   6.5252   5.1801        4
1.561782   0.345843         174       174.0   6.4677   6.0781        4
0.797207   0.032632         348       348.0   6.6580   6.5860        4
0.398842   0.000476         696       696.0   5.0679   5.0723        4
0.199421   0.000000        1392      1392.0   8.8758   8.8758        4
0.099711   0.000000        2784      2784.0   7.4089   7.4089        4
0.049855   0.000000        5568      5568.0   8.7209   8.7209        4
0.024930   0.000000       11135     11135.0   9.5274   9.5274        4
0.012465   0.000000       22269     22269.0   7.9403   7.9403        4
0.006233   0.000000       44537     44537.0   3.4829   3.4829        4

finished run
number of examples = 50000
weighted example sum = 50000
weighted label sum = 299829
average loss = 0.00555188
best constant = 5.99657
total feature number = 200000
```

Demo

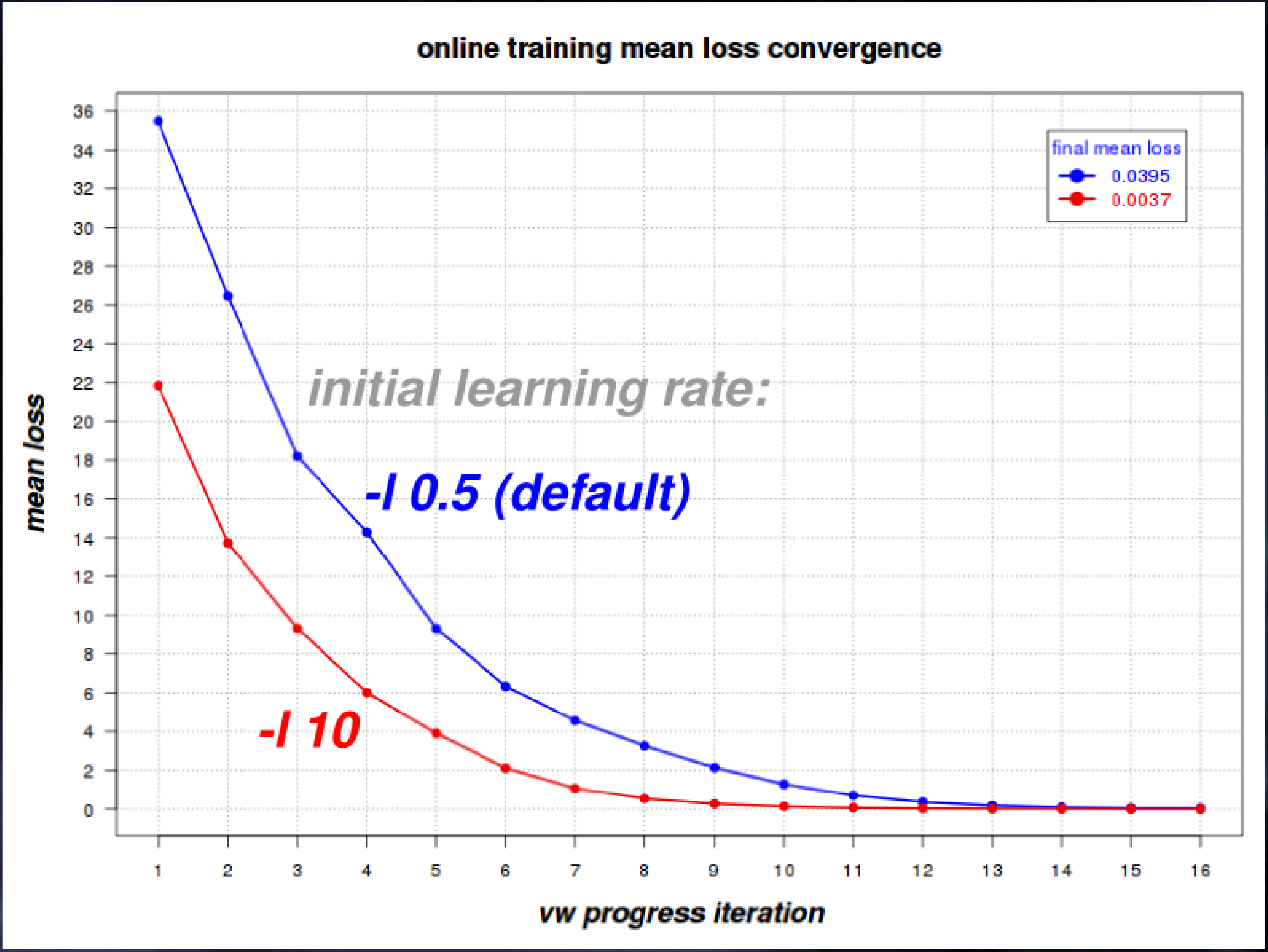
error convergence towards zero w/ 2 learning rates:

$ vw r.train

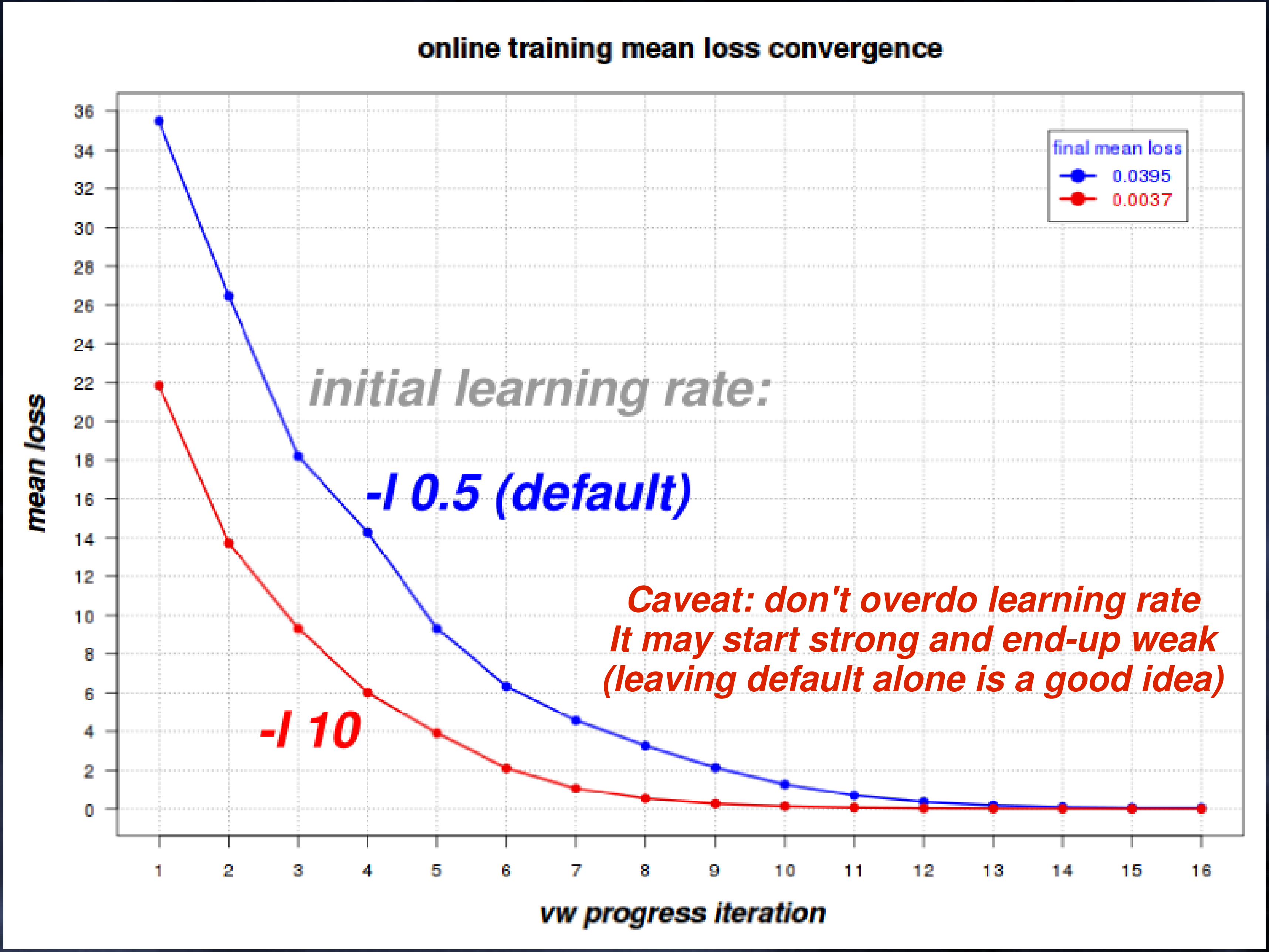
$ vw r.train -l 10



vw error convergence w/ 2 learning rates



vw error convergence w/ 2 learning rates



Demo

(separate a signal from surrounding noise)

Step 2 (looking at the trained model weights):

$ vw-varinfo -l 5 -f r.model   r.train

```
=== 3: Train: look at the model weights
$ vw-varinfo -l 5 r.train
FeatureName        HashVal   MinVal   MaxVal    Weight   RelScore
f^b                 146788     0.00     1.00   +2.0000     40.00%
f^a                  14355     0.00     1.00   +1.0000     20.00%
Constant            116060     0.00     0.00   +7.0000      0.00%
f^c                 253856     0.00     1.00   -5.0000   -100.00%
```

Demo

(separate a signal from surrounding noise)

Step 2 (looking at the trained model weights):

$ vw-varinfo -l 5 -f r.model r.train

```
=== 3: Train: look at the model weights
$ vw-varinfo -l 5 r.train
FeatureName         HashVal   MinVal   MaxVal    Weight   RelScore
f^b                  146788     0.00     1.00   +2.0000     40.00%
f^a                   14355     0.00     1.00   +1.0000     20.00%
Constant             116060     0.00     0.00   +7.0000      0.00%
f^c                  253856     0.00     1.00   -5.0000   -100.00%
```

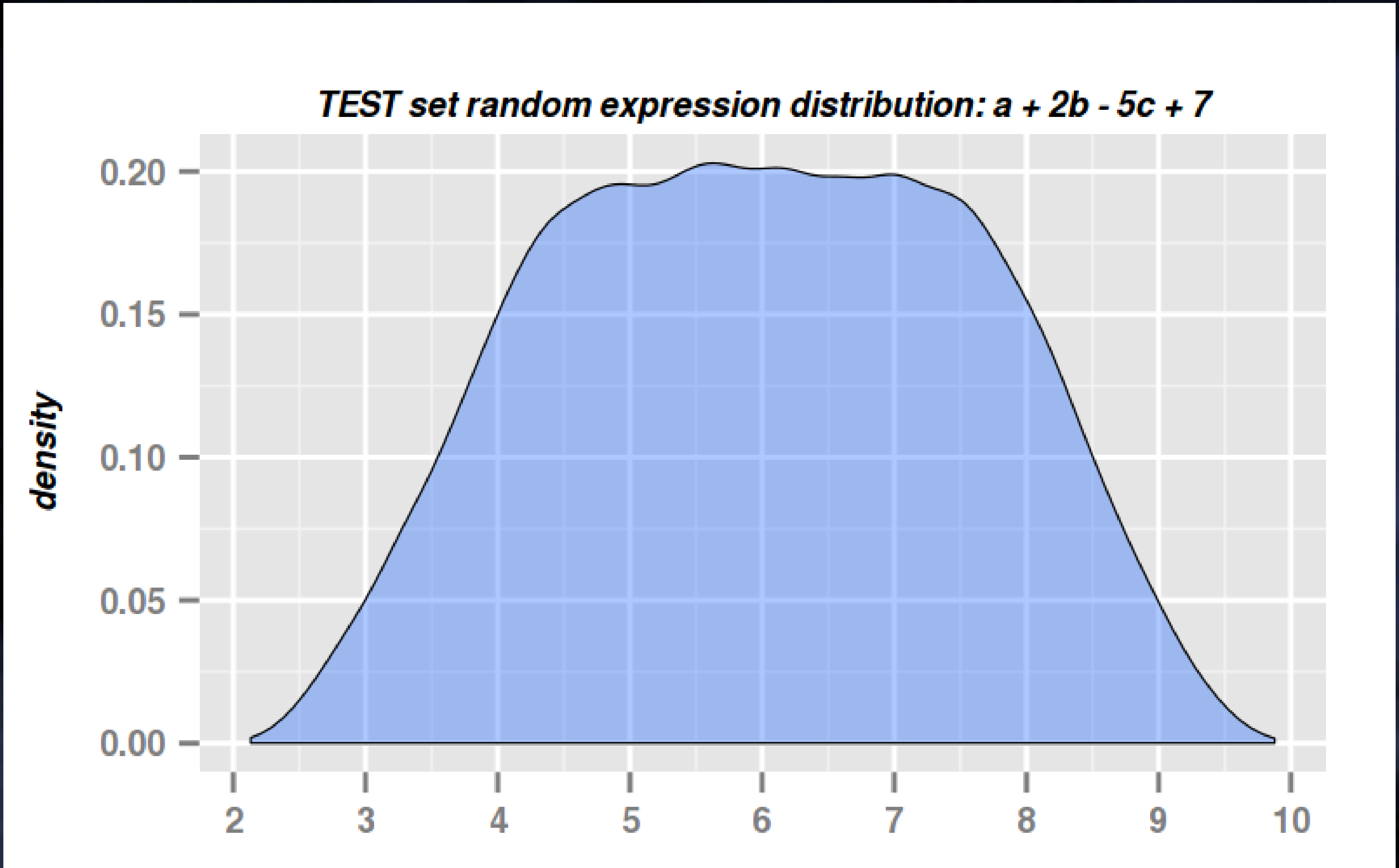
Perfect weights for {a, b, c} & the hidden constant

Q: how good is our model?

Steps 3, 4, 5, 6:

- Create independent random data-set for same expression: $Y = a + 2b - 5c + 7$
- Drop the Y output column (labels) Leave only input columns (a, b, c)
- Run vw: load the model + predict
- Compare Y predictions to Y actual values

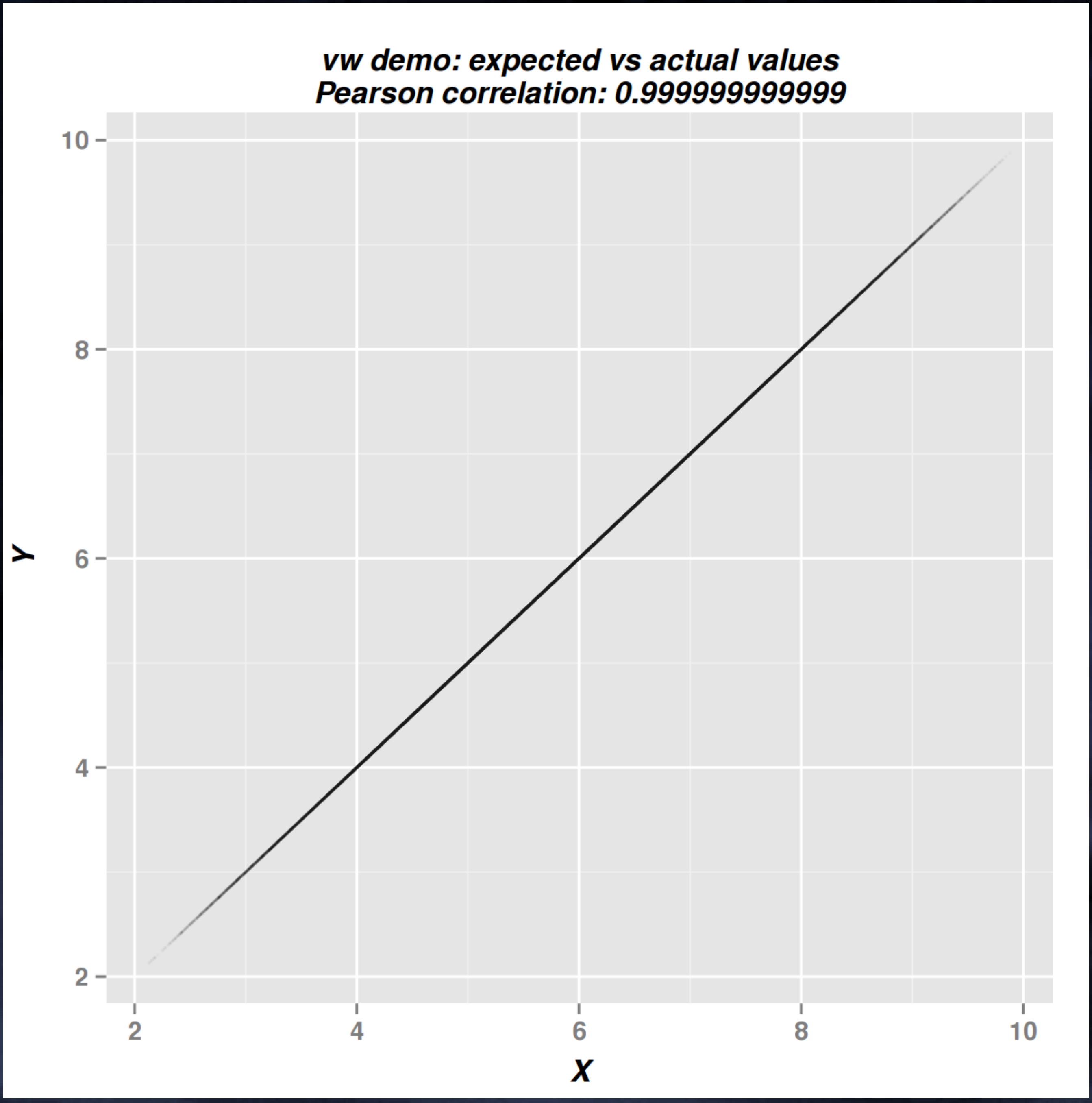
test-set Ys (labels) density



predicted vs. actual (top few)

predicted		actual
8.455564	|	8.455560
7.594127	|	7.594125
5.321825	|	5.321826
6.509799	|	6.509795
7.354873		7.354873
4.561502	|	4.561500
8.095616	|	8.095618
6.707353		6.707353
4.268953	|	4.268952
6.679539	|	6.679541
4.642760	|	4.642761
5.364318	|	5.364319
7.818297		7.818297
6.362800	|	6.362796
4.910164		4.910164

Q: how good is our model?



Q.E.D

Demo – part 2: adding noise

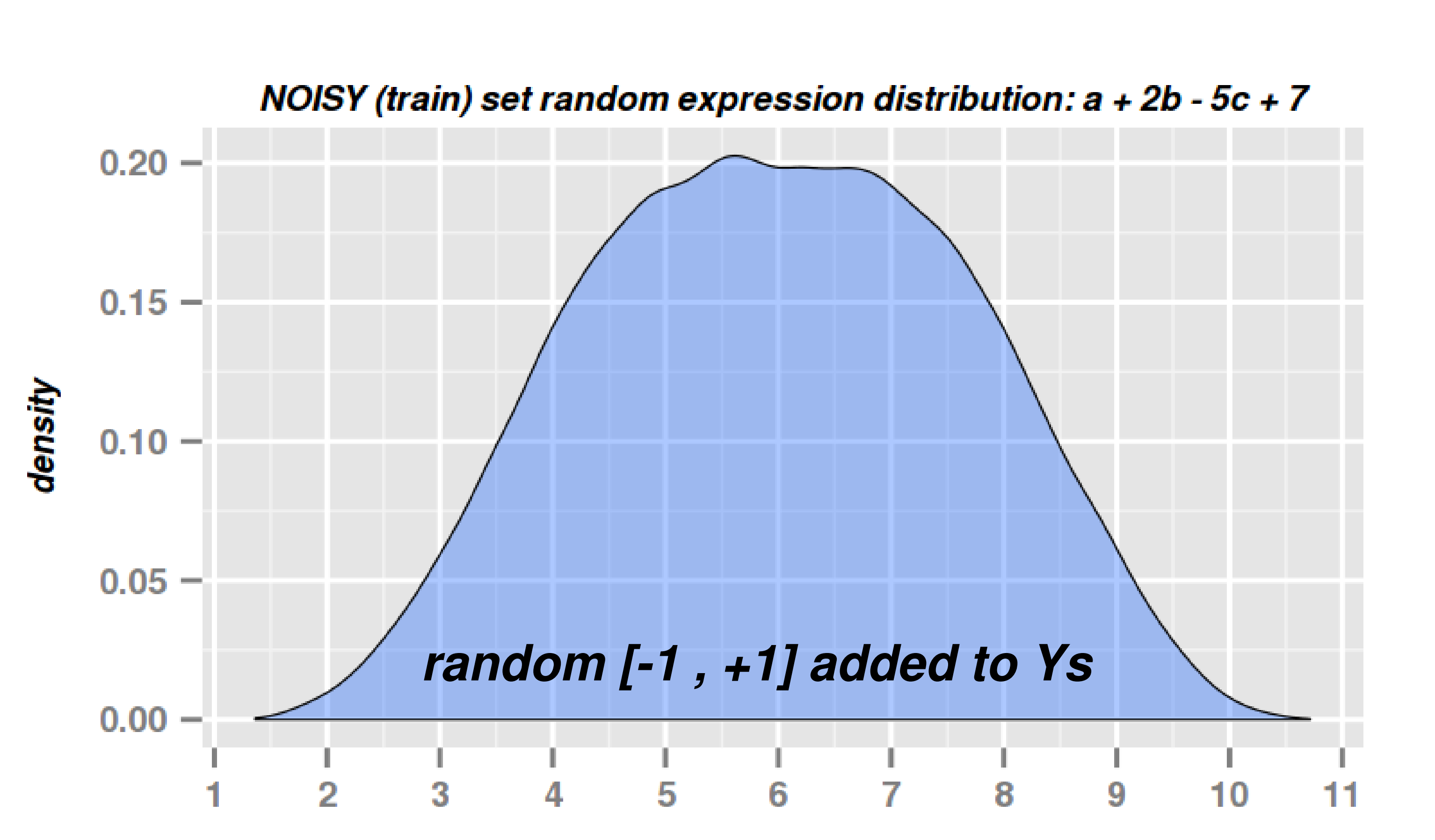
Unfortunately, real life is never so perfect

so let's repeat the whole exercise
with a distortion:

Add "global" noise to each train-set result (Y)
& make it "wrong" by up to [-1 , +1]

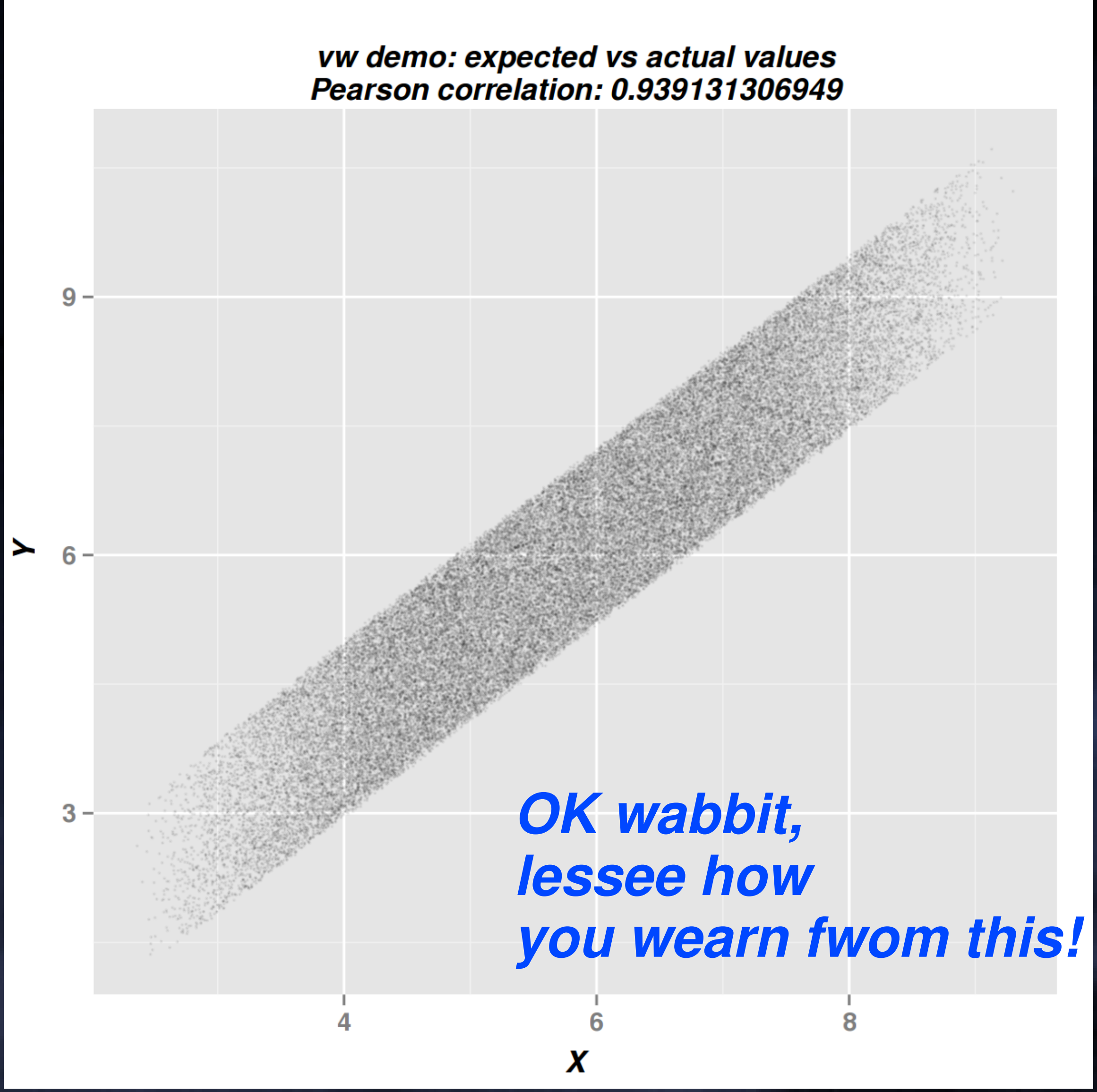
$ random-poly -n 50000 -p6 -r -1,1 a + 2b - 5c + 7 > r.train

NOISY train-set Ys (labels) density



range falls outside [2 ,10]
due to randomly added [-1 , +1]

Original Ys vs NOISY train-set Ys (labels)



train-set Ys range falls outside [2 ,10]
due to randomly added [-1 ,1]

NOISY train-set – model weights

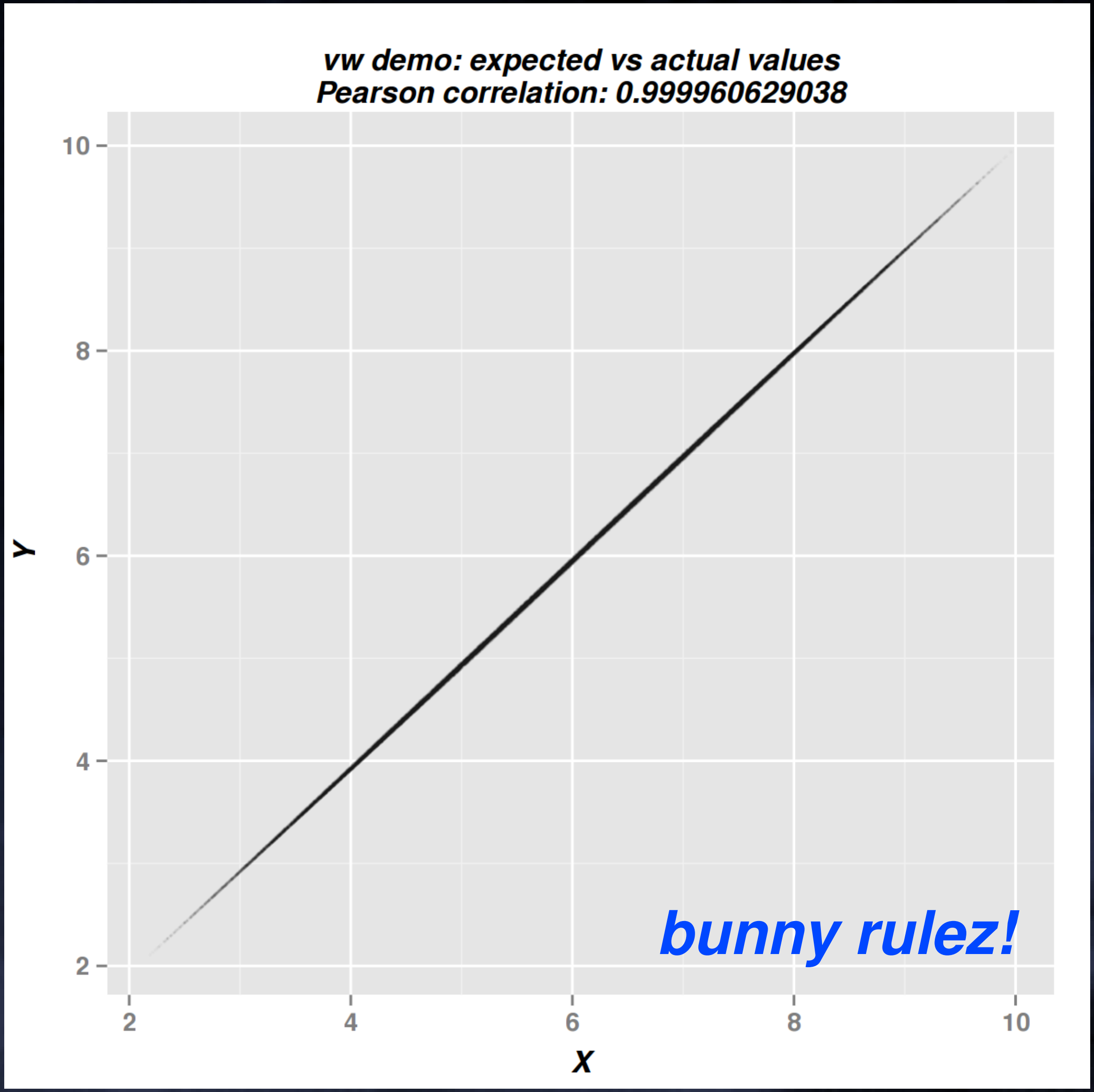
```
=== 3: Train: look at the model weights
$ vw-varinfo -k  r.train
FeatureName          HashVal   MinVal   MaxVal    Weight   RelScore
f^b                   146788     0.00     1.00   +2.0121     40.92%
f^a                    14355     0.00     1.00   +1.0105     20.55%
Constant              116060     0.00     0.00   +6.9974      0.00%
f^c                   253856     0.00     1.00   -4.9168   -100.00%
```

no fooling bunny
model built from global noisy data
has still near perfect weights {a, 2b, -5c, 7}

global-noise predicted vs. actual (top few)

predicted	actual
7.667181	7.646101
6.384394	6.351331
7.156300	7.131273
7.381573	7.364534
4.589863	4.490818
4.224433	4.140517
3.965925	3.896666
3.264382	3.179833
5.869455	5.822634
6.504361	6.466419
5.710673	5.632021
8.812504	8.782471
6.927242	6.892944

predicted vs test-set actual w/ NOISY train-set



surprisingly good
because noise is unbiased/symmetric

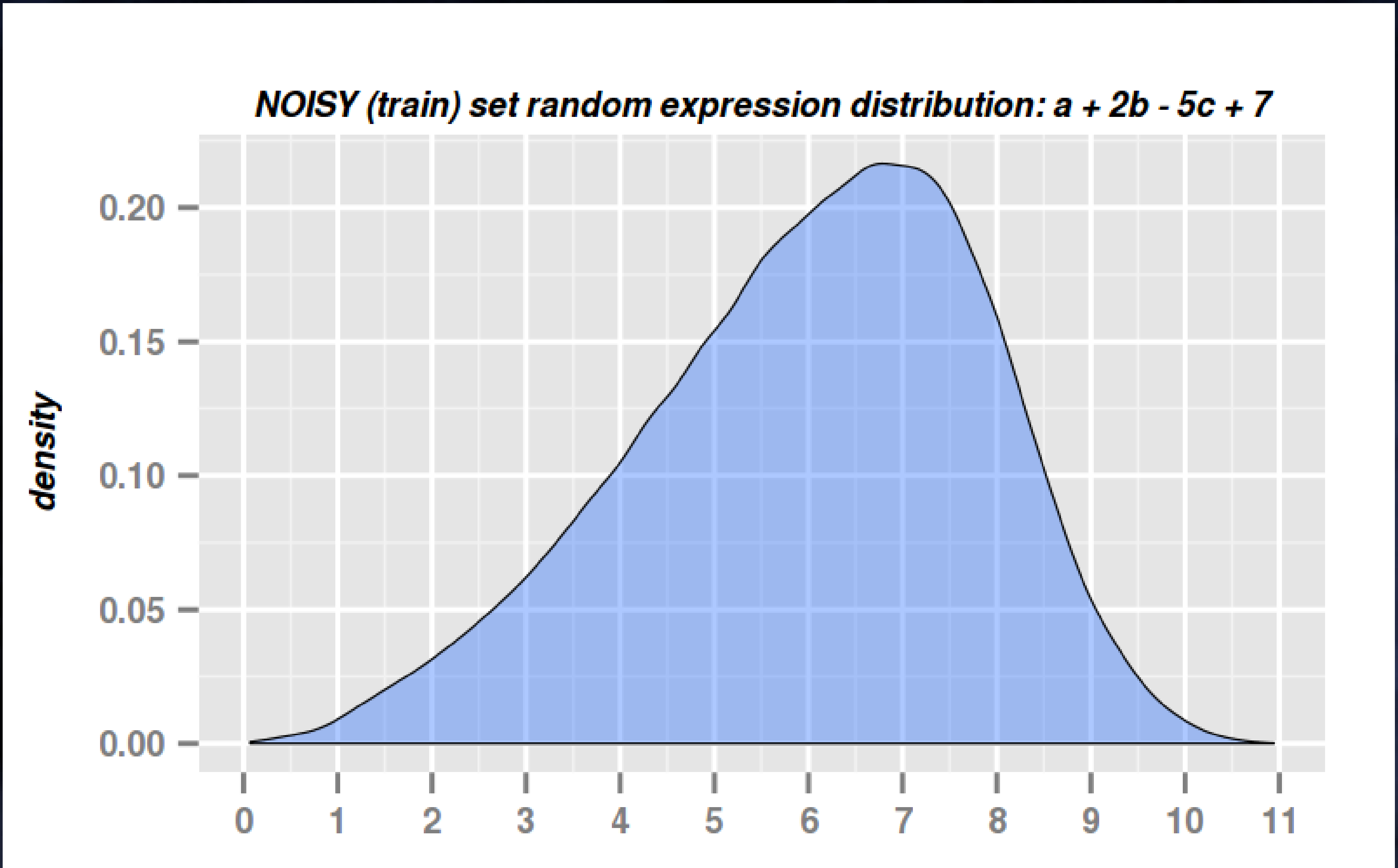
Demo – part 3: more noise

Let's repeat the whole exercise with a more realistic (real-life) distortion:

Add noise to each train-set variable separately & make it "wrong" by up to +/- 50% of its magnitude:

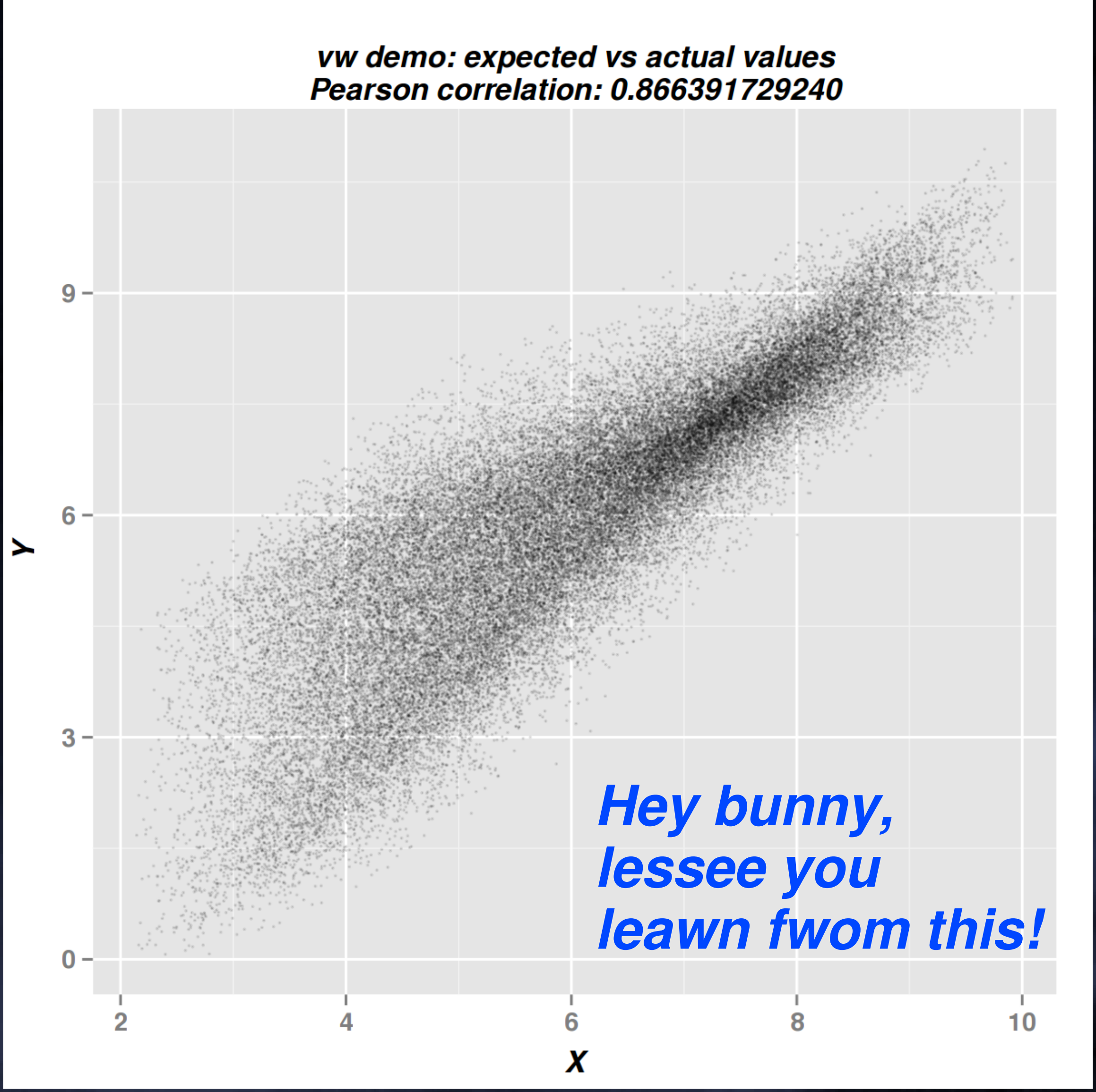
$ random-poly -n 50000 -p6 -R -0.5,0.5 a + 2b - 5c + 7 > r.train

all-var NOISY train-set Ys (labels) density



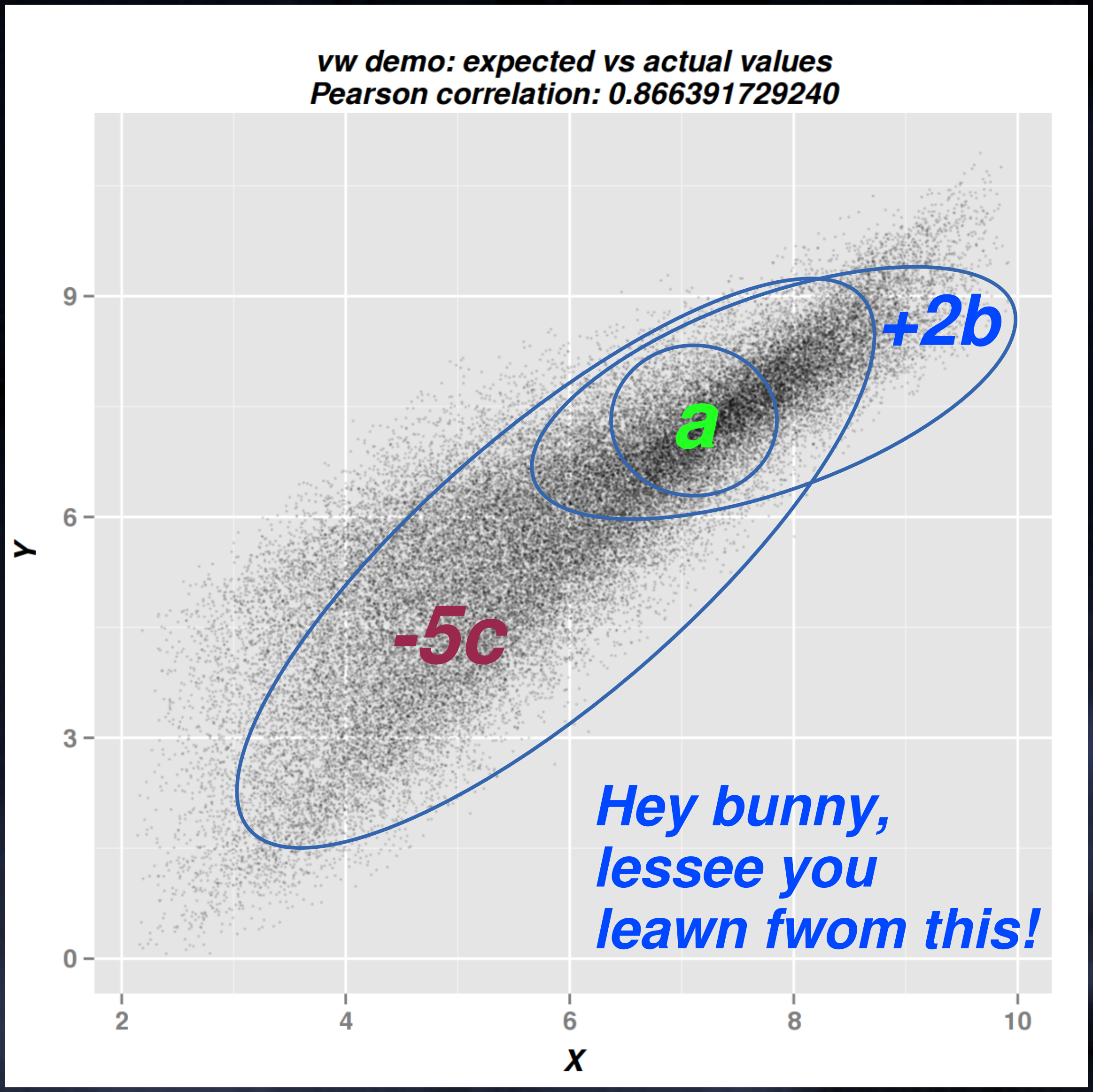
range falls outside [2 ,10] + skewed density due to randomly added [+/- 50% per variable]

expected vs per-var NOISY train-set Ys (labels)



Nice mess: skewed, tri-modal, X shaped due to randomly added +/- 50% per var

expected vs per-var NOISY train-set Ys (labels)



Nice mess: skewed, tri-modal, X shaped due to randomly added +/- 50% per var

per-var NOISY train-set – model weights

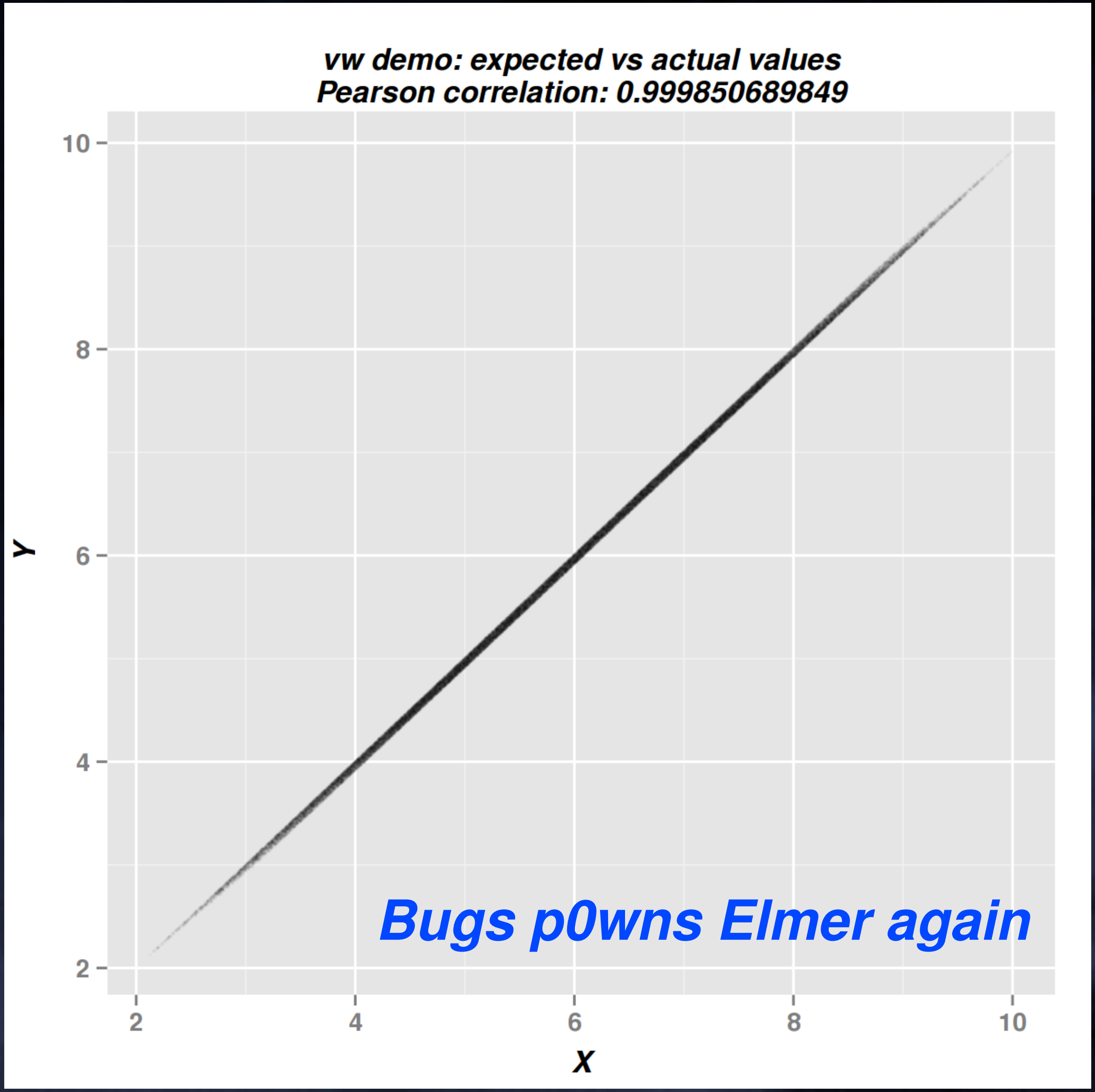
```
=== 3: Train: look at the model weights (w/ per var noise)
$ vw-varinfo  r.train
FeatureName          HashVal   MinVal   MaxVal    Weight   RelScore
f^b                   146788     0.00     1.00   +2.0074     40.28%
f^a                    14355     0.00     1.00   +1.0929     21.93%
Constant              116060     0.00     0.00   +6.9782      0.00%
f^c                   253856     0.00     1.00   -4.9842   -100.00%
```

model built from this noisy data is still remarkably close to the perfect {a, 2b, -5c, 7} weights

per-var noise predicted vs. actual (top few)

predicted	actual
4.512985	4.433761
8.281034	8.269412
5.897462	5.891922
5.736421	5.672354
3.581511	3.529558
8.136392	8.104806
8.123308	8.100060
6.782677	6.778263
5.992764	5.953797
8.590860	8.566768
8.738648	8.690760
5.022583	4.954888
6.615203	6.614312

predicted vs test-set actual w/ per-var NOISY train-set



remarkably good
because even per-var noise is unbiased/symmetric

there's so much more in vowpal wabbit

Classification
Reductions
Regularization
Many more run time options
Cluster mode / all-reduce…

The wiki on github is a great start

“Ve idach zil gmor” (Hillel the Elder)

“As for the west - go leawn” (Elmer's translation)

Questions?