Solvers and stuff

Jason G. Fleischer, Ph.D.
Asst. Teaching Professor
Department of Cognitive Science, UC San Diego

jfleischer@ucsd.edu

@jasongfleischer

https://jgfleischer.com

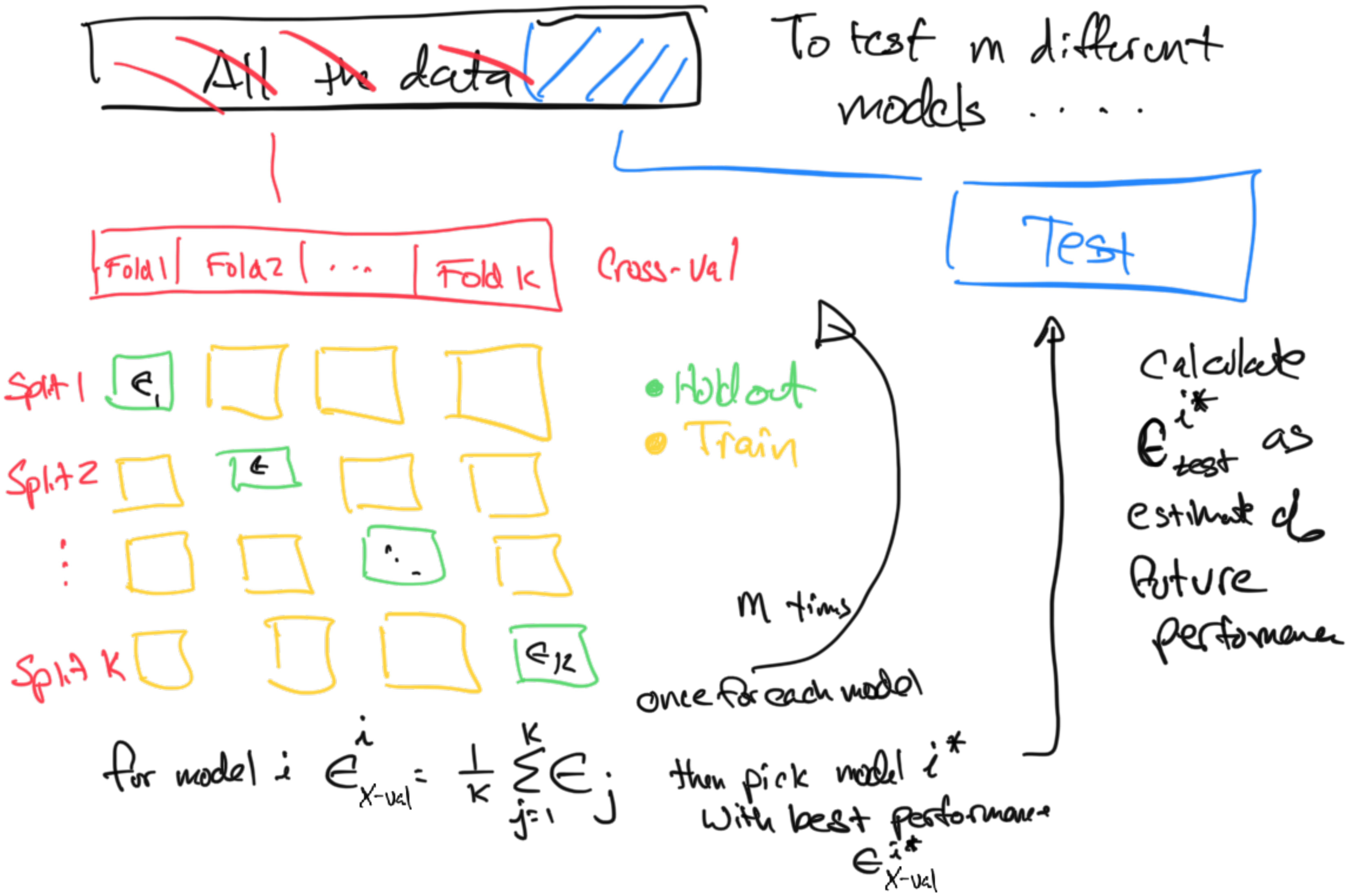
Slides in this presentation are from material kindly provided by Sebastian Rashka

METHOD 1 - with enough data to have a good test set

Let's say you had around 8k samples in a dataset

For each trial:

- training set ~ sample 5k (with or w/o) replacement from entire dataset
- Grid search of hyper parameters using k-fold cross validation on the training set
- Select best model from grid, train on entire training set
- Evaluate best model on the test set (everything not sampled for training)



METHOD 3 - AC or data efficient MC

<u>Nested Cross-validation</u>

For **algorithm comparison** if done the time/memory efficient way: store metrics on inner cv, pick best model, then store metrics of best model on outer cv for AC

…can be used for **model comparison** if done the inefficient way: metrics stored on inner and outer cv, calc best model post hoc on inner, use outer cv as test metric for best model

This for when you've got only ~2000 samples, which is barely enough to fit the data well let alone test

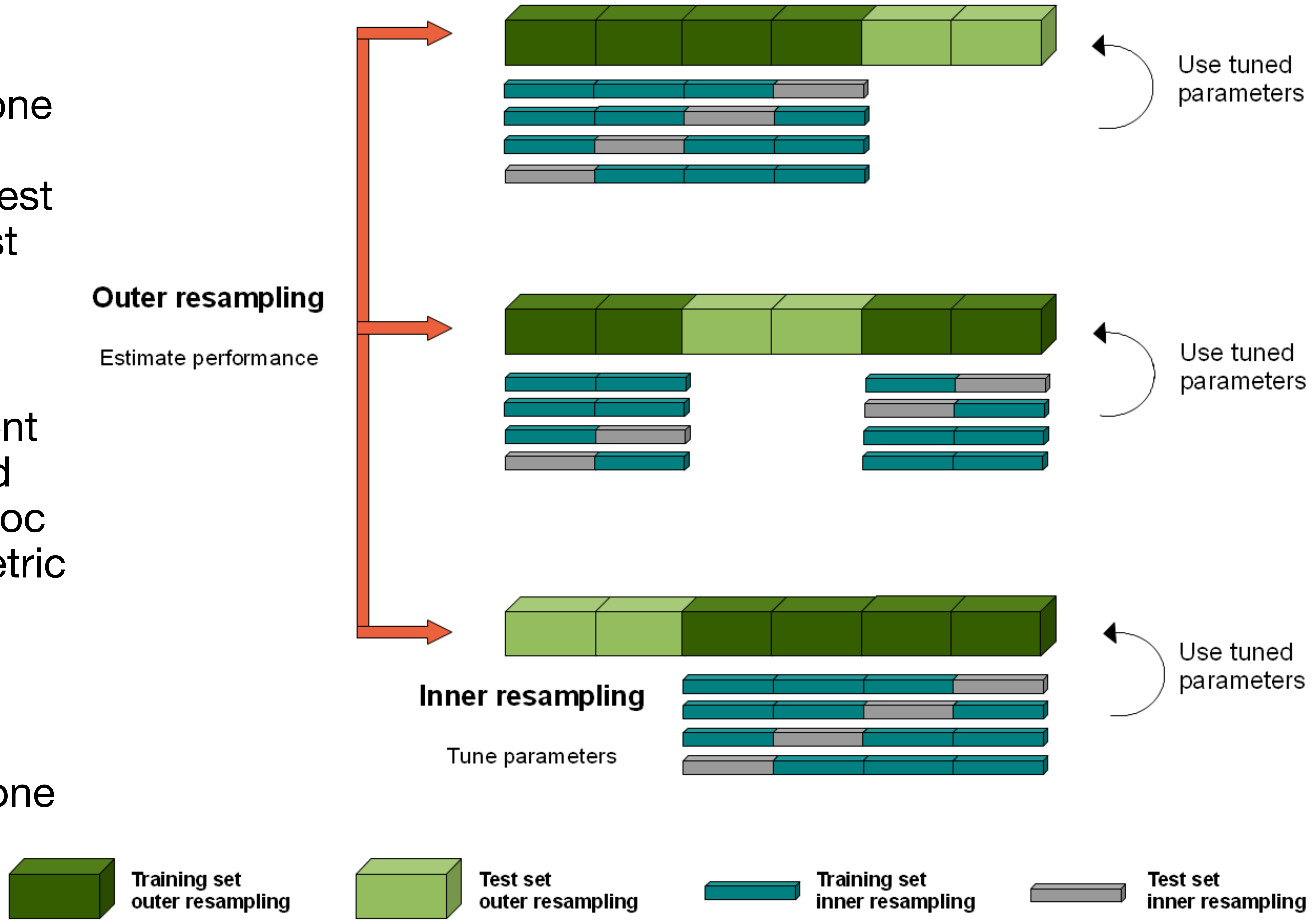


Figure from mlr R package docs

Method 3a - Nested CV

For doing algorithm selection

- Split off a test set for later
- [OPTIONAL] Outer loop… do this T times:
    - Do this M times, once for each algorithm
        - Use nested k-fold cross validation…
            - Inner CV estimates validation error for all the hyperparams tested for a given model
            - Outer CV estimates validation error for the best hyperparams from inner CV for a given algorithm.
- Pick the best algorithm based on its performance on the mean across [OPTIONAL trials and] the outer cross validation folds
- Train the chosen algorithm on entire nested CV dataseet. Use test set to estimate generalization performance
- Reasonably time/memory efficient as outer loop is only done once per algorithm and you don't have to store the inner loop results

Method 3b - Nested CV

For model selection on small datasets

- Do not split off a test set! This is why its data efficient.
- [OPTIONAL] Outer loop… do this T times:
  - Do this M times, once for each algorithm
    - Use nested k-fold cross validation…
      - Inner CV estimates validation error for all the hyperparams tested for a given model
      - Outer CV estimates test error for all the hyperparams tested for a given model
- Pick the best model based on its performance on the mean across [OPTIONAL trials and] folds of the inner cross validations and report its generalization performance on the mean across [OPTIONAL trials and] folds of the outer cross validation. This is post-hoc… you don't know which model is best until all the trials are done, so you have to calculate and store both inner and outer fold performance on every model.

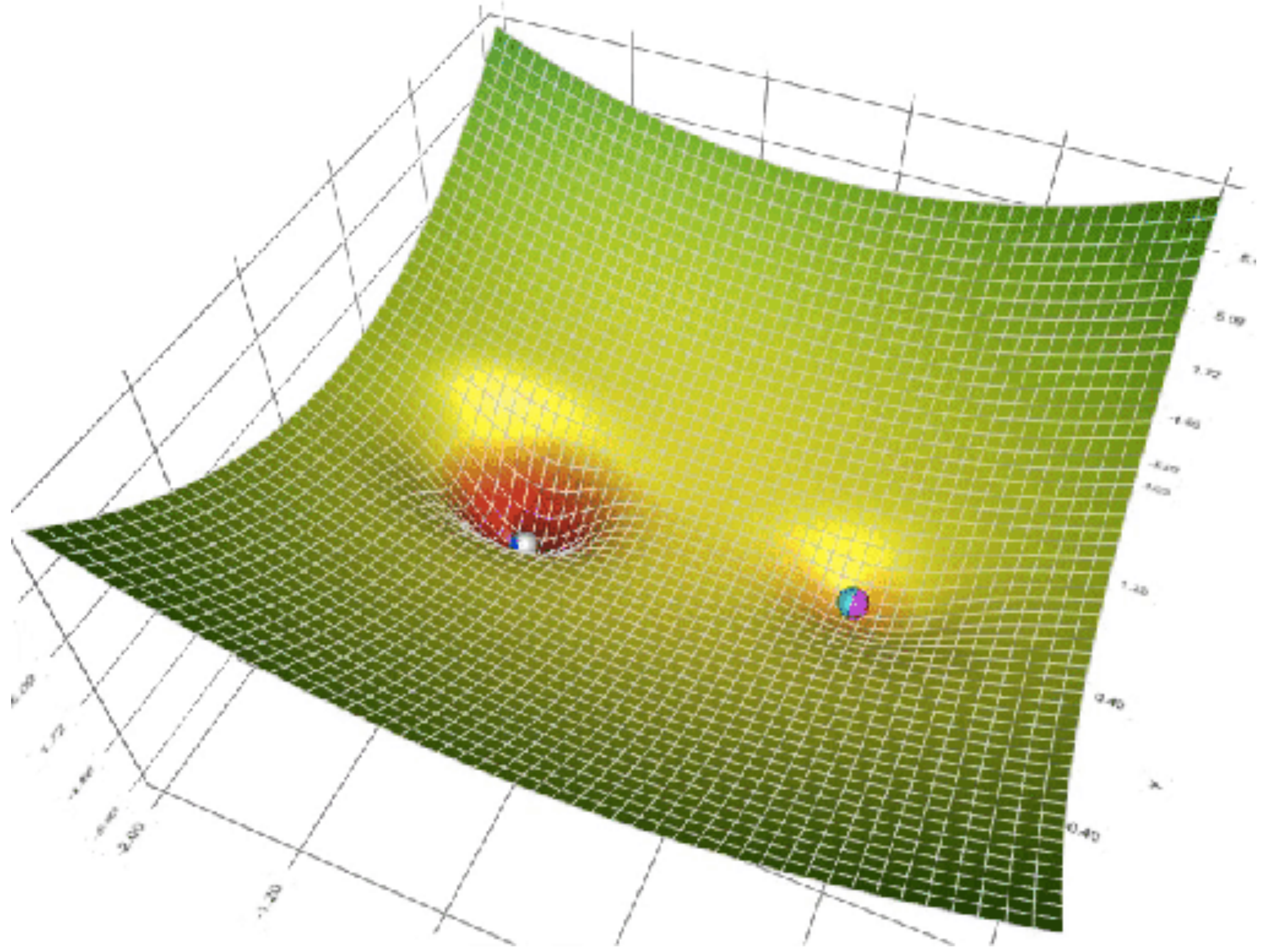
Gradient descent

$$\mathbf{w} = \mathbf{w} - \eta \frac{\partial \mathscr{L}(\mathbf{w})}{\partial \mathbf{w}}$$

$$= \mathbf{w} - \eta \nabla_{\mathbf{w}} \mathscr{L}(\mathbf{w})$$

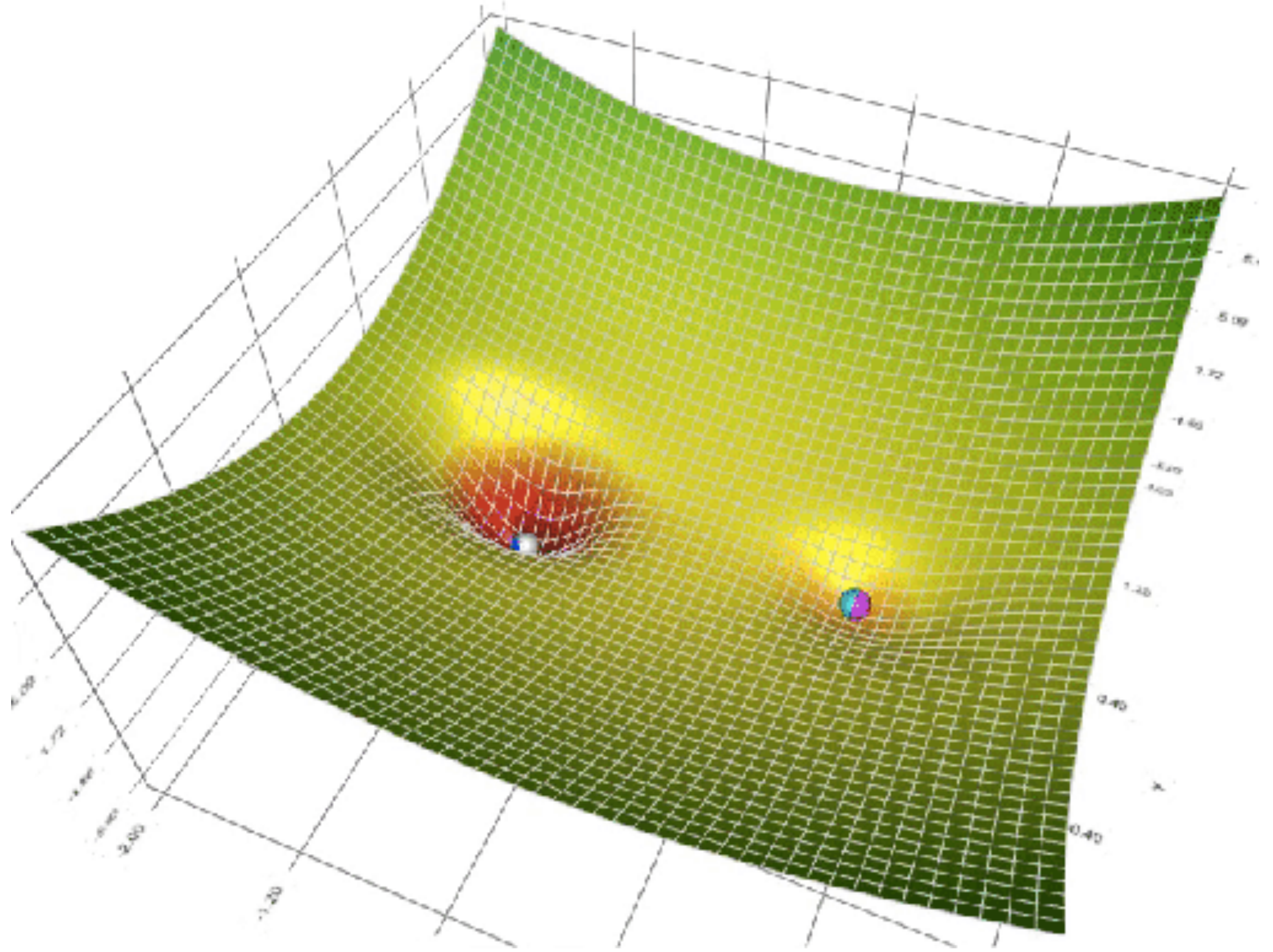
Why is the gradient term negative?

Batch vs stochastic vs minibatch



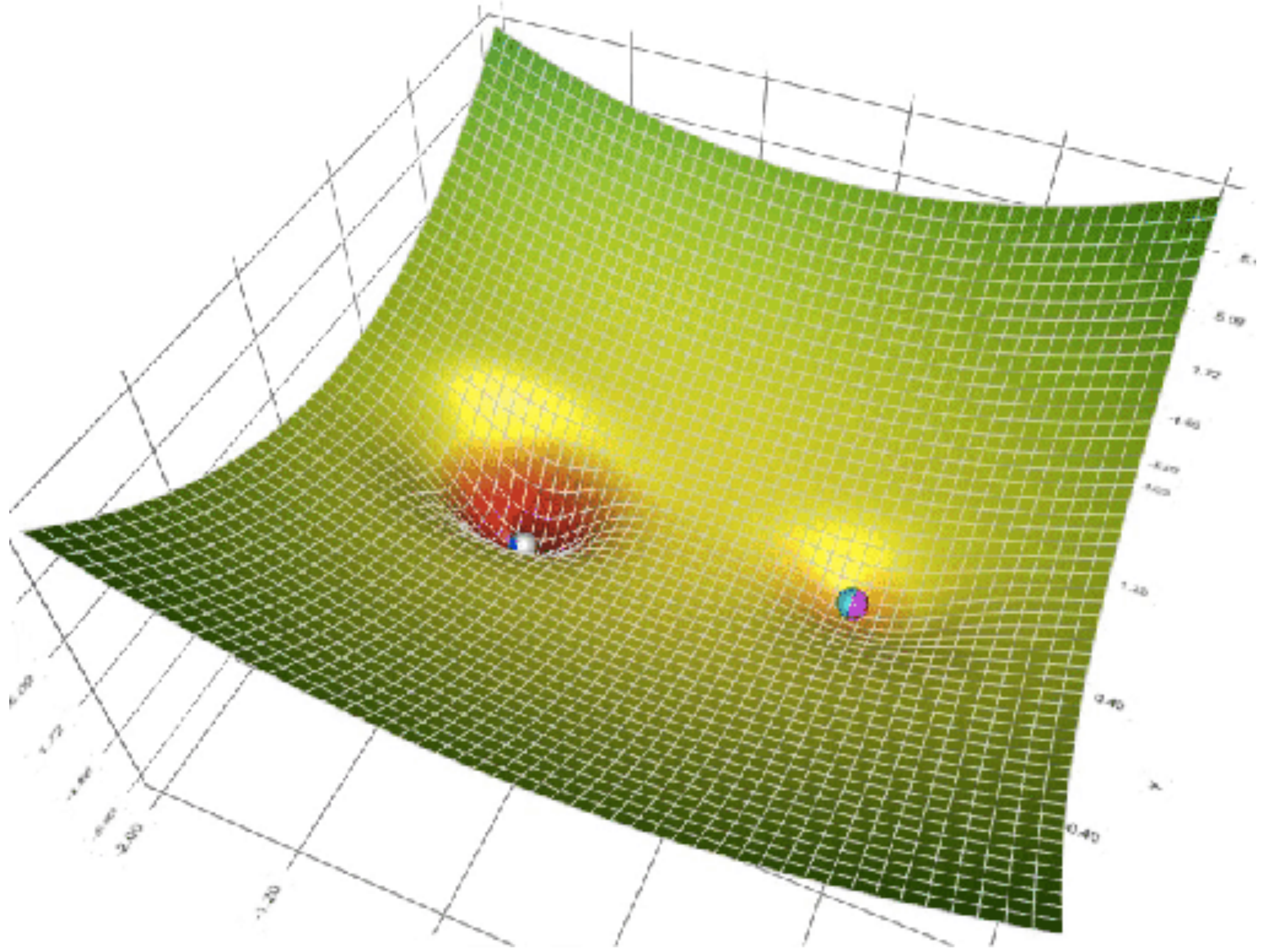
- Batch gradient descent
  - Takes all the training data, calculate gradient, take a step
  - Lots of memory required!
  - $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla_{\mathbf{w}} \mathscr{L}\left(\mathbf{w}; \mathbf{X}; \mathbf{y}\right)$

Batch vs stochastic vs minibatch



- Stochastic gradient descent
    - Pick a random data point in training, calculate gradient, take a step
    - Little memory required!
    - Not as accurate, Not repeatable
    - $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla_{\mathbf{w}} \mathscr{L}\left(\mathbf{w}; \mathbf{x}^{(i)}; y^{(i)}\right)$

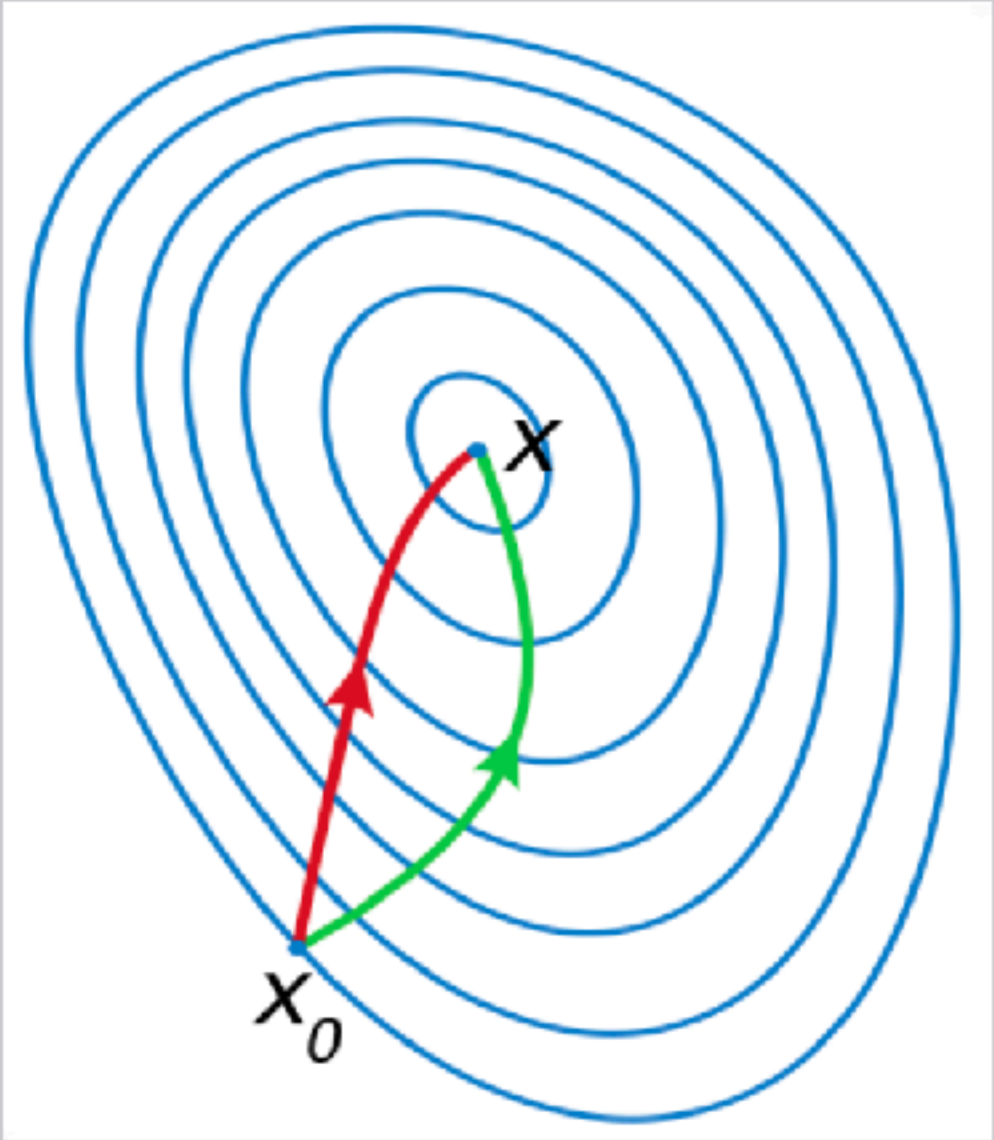
Batch vs stochastic vs minibatch



- Mini-batch gradient descent (ONLINE)
    - Takes a chunk of training data, calculate gradient, take a step
    - Goldilocks zone for memory, time, accuracy, repeatability
    - $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla_{\mathbf{w}} \mathscr{L}\left(\mathbf{w}; \mathbf{X}^{(i:i+n)}; \mathbf{y}^{(i:i+n)}\right)$

Higher order terms

Newton's method in optimization



A comparison of gradient descent (green) and Newton's method (red) for minimizing a function (with small step sizes). Newton's method uses curvature information (i.e. the second derivative) to take a more direct route.

In calculus, **Newton's method** is an iterative method for finding the roots of a differentiable function F, which are solutions to the equation $F(x) = 0$. As such, Newton's method can be applied to the derivative f' of a twice-differentiable function f to find the roots of the derivative (solutions to $f'(x) = 0$), also known as the critical points of f. These solutions may be minima, maxima, or saddle points; see section "Several variables" in Critical point (mathematics) and also section "Geometric interpretation" in this article. This is relevant in optimization, which aims to find (global) minima of the function f.

Newton's method [edit]

The central problem of optimization is minimization of functions. Let us first consider the case of univariate functions, i.e., functions of a single real variable. We will later consider the more general and more practically useful multivariate case.

Given a twice differentiable function $f : \mathbb{R} \to \mathbb{R}$, we seek to solve the optimization problem

$$\min_{x \in \mathbb{R}} f(x).$$

Newton's method attempts to solve this problem by constructing a sequence $\{x_k\}$ from an initial guess (starting point) $x_0 \in \mathbb{R}$ that converges towards a minimizer x_* of f by using a sequence of second-order Taylor approximations of f around the iterates. The second-order Taylor expansion of f around x_k is

$$f(x_k + t) \approx f(x_k) + f'(x_k)t + \frac{1}{2} f''(x_k)t^2.$$

The next iterate x_{k+1} is defined so as to minimize this quadratic approximation in t, and setting $x_{k+1} = x_k + t$. If the second derivative is positive, the quadratic approximation is a convex function of t, and its minimum can be found by setting the derivative to zero. Since

Quasi-Newton method

7 languages

Article Talk

Read Edit View history

From Wikipedia, the free encyclopedia

Quasi-Newton methods are methods used to either find zeroes or local maxima and minima of functions, as an alternative to Newton's method. They can be used if the Jacobian or Hessian is unavailable or is too expensive to compute at every iteration. The "full" Newton's method requires the Jacobian in order to search for zeros, or the Hessian for finding extrema.

As in Newton's method, one uses a second-order approximation to find the minimum of a function $f(x)$. The Taylor series of $f(x)$ around an iterate is

$$f(x_k + \Delta x) \approx f(x_k) + \nabla f(x_k)^{\mathrm{T}} \, \Delta x + \frac{1}{2} \Delta x^{\mathrm{T}} B \, \Delta x,$$

where (∇f) is the gradient, and B an approximation to the Hessian matrix.[4] The gradient of this approximation (with respect to Δx) is

$$\nabla f(x_k + \Delta x) \approx \nabla f(x_k) + B \, \Delta x,$$

and setting this gradient to zero (which is the goal of optimization) provides the Newton step:

$$\Delta x = -B^{-1} \nabla f(x_k).$$

The Hessian approximation B is chosen to satisfy

$$\nabla f(x_k + \Delta x) = \nabla f(x_k) + B \, \Delta x,$$

Quasi-Newton method

7 languages

Article Talk Read Edit View history

From Wikipedia, the free encyclopedia

Quasi-Newton methods are methods used to either find zeroes or local maxima and minima of functions, as an alternative to Newton's method. They can be used if the Jacobian or Hessian is unavailable or is too expensive to compute at every iteration. The "full" Newton's method requires the Jacobian in order to search for zeros, or the Hessian for finding extrema.

Limited-memory BFGS (**L-BFGS** or **LM-BFGS**) is an optimization algorithm in the family of quasi-Newton methods that approximates the Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS) using a limited amount of computer memory.[1] It is a popular algorithm for parameter estimation in machine learning.[2][3] The algorithm's target problem is to minimize $f(\mathbf{x})$ over unconstrained values of the real-vector $\mathbf{x}$ where f is a differentiable scalar function.

Like the original BFGS, L-BFGS uses an estimate of the inverse Hessian matrix to steer its search through variable space, but where BFGS stores a dense $n \times n$ approximation to the inverse Hessian (n being the number of variables in the problem), L-BFGS stores only a few vectors that represent the approximation implicitly. Due to its resulting linear memory requirement, the L-BFGS method is particularly well suited for optimization problems with many variables. Instead of the inverse Hessian $\mathbf{H}_k$, L-BFGS maintains a history of the past m updates of the position $\mathbf{x}$ and gradient $\nabla f(\mathbf{x})$, where generally the history size m can be small (often $m < 10$). These updates are used to implicitly do operations requiring the $\mathbf{H}_k$-vector product.

Momentum

See https://distill.pub/2017/momentum/

Adaptive step sizes

See https://www.ruder.io/optimizing-gradient-descent/

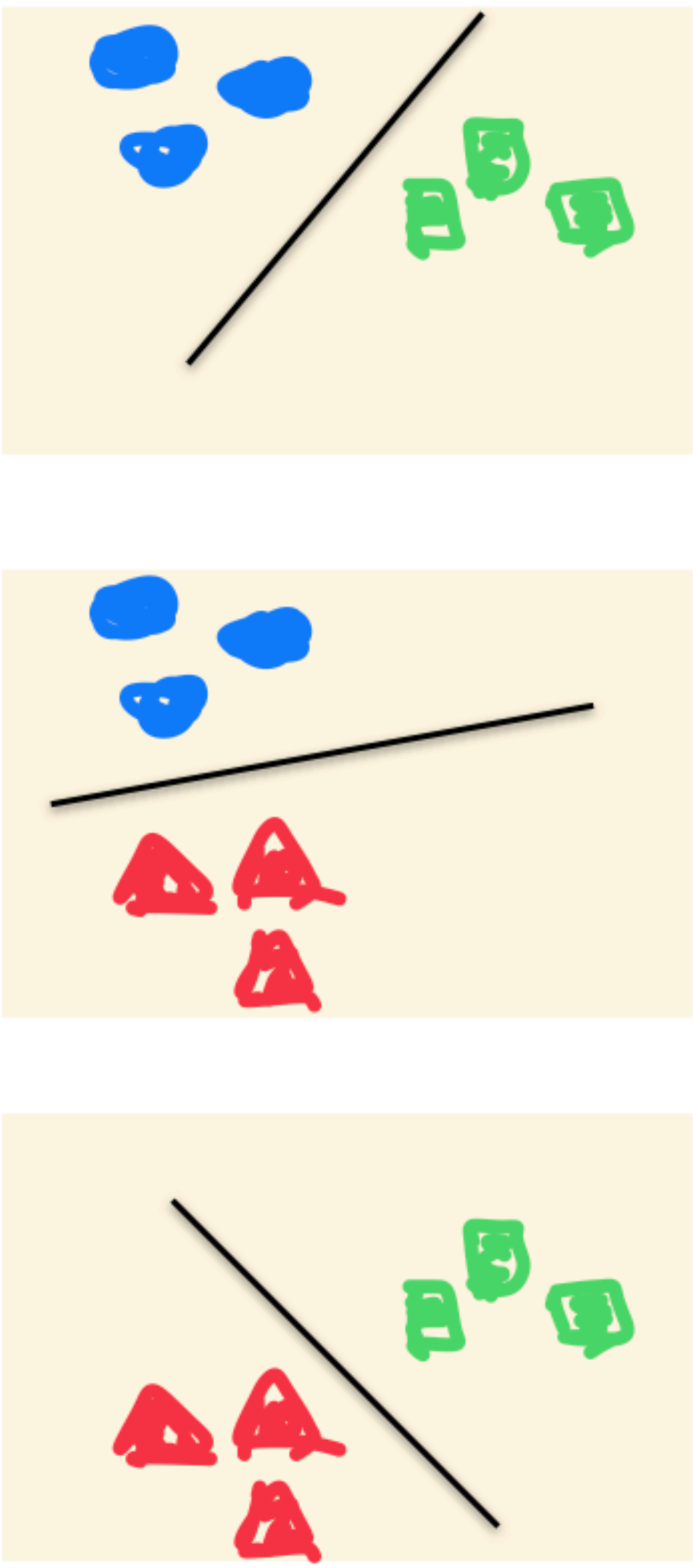
Extending binary classification to multi-class situations

$K=3$ classes

One v One uses class with highest confidence score

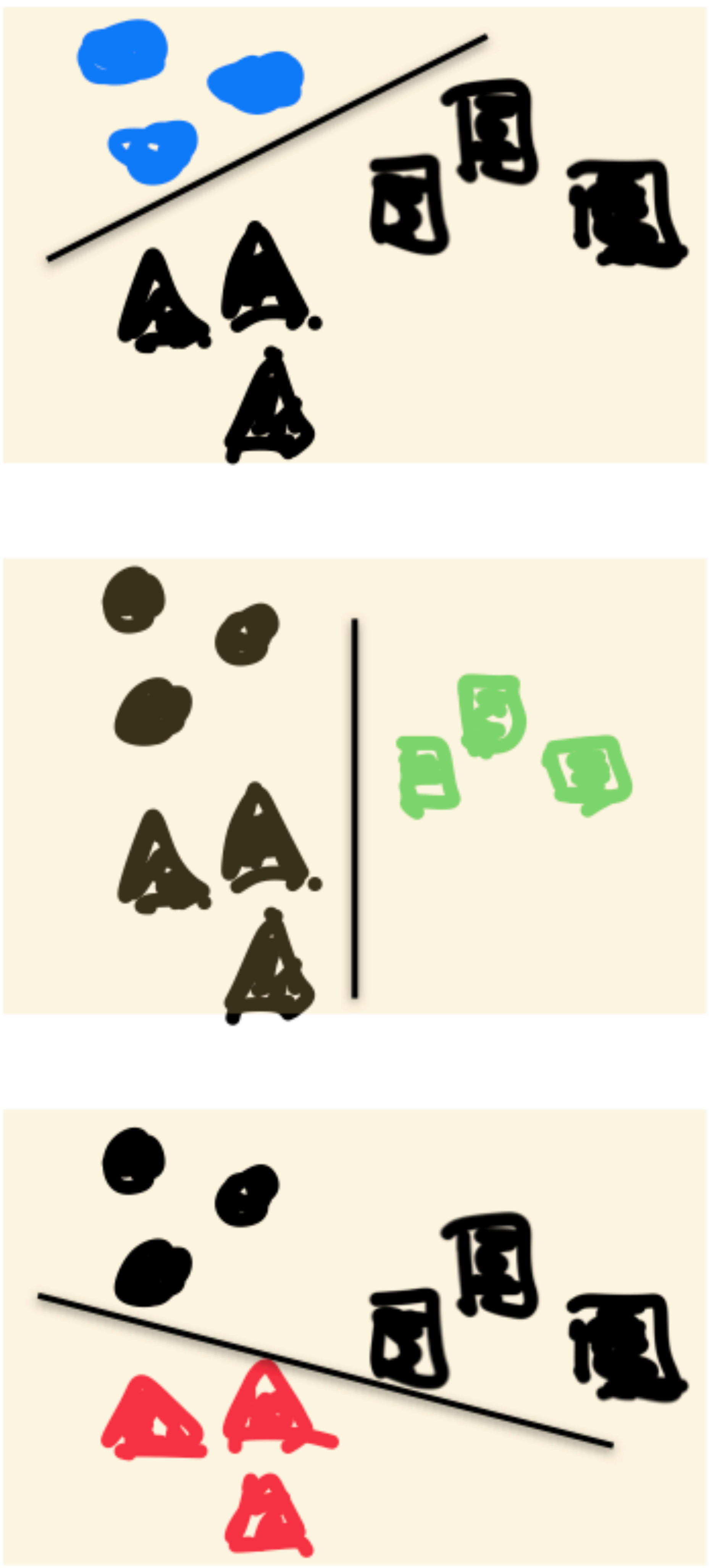
One v Rest uses plurality (mode) vote, tie breaker with confidence score

One v. One



$\frac{K(K-1)}{2}$ classifiers

One v. Rest



K classifiers

Multi-ball!

https://scikit-learn.org/stable/modules/multiclass.html