Lecture 2: Linear models

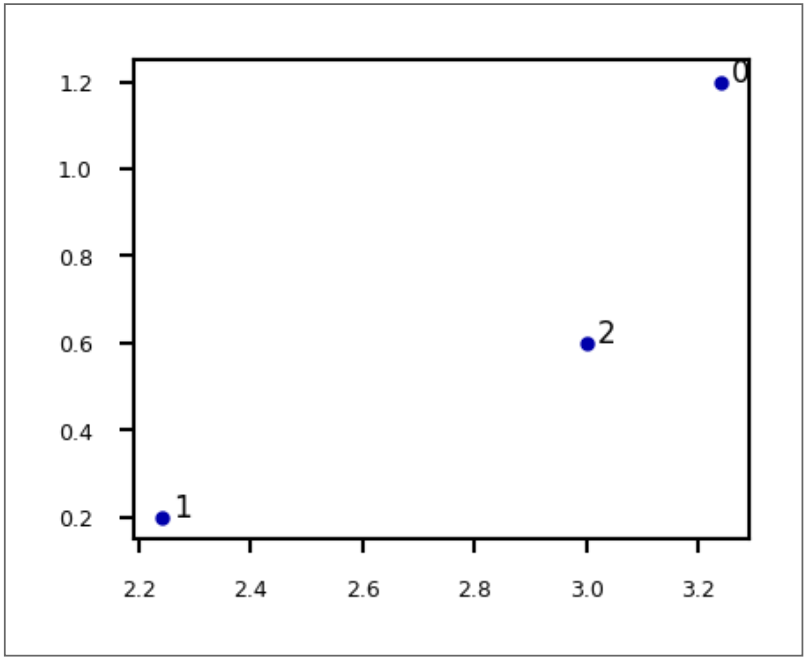
Basics of modeling, optimization, and regularization

Joaquin Vanschoren

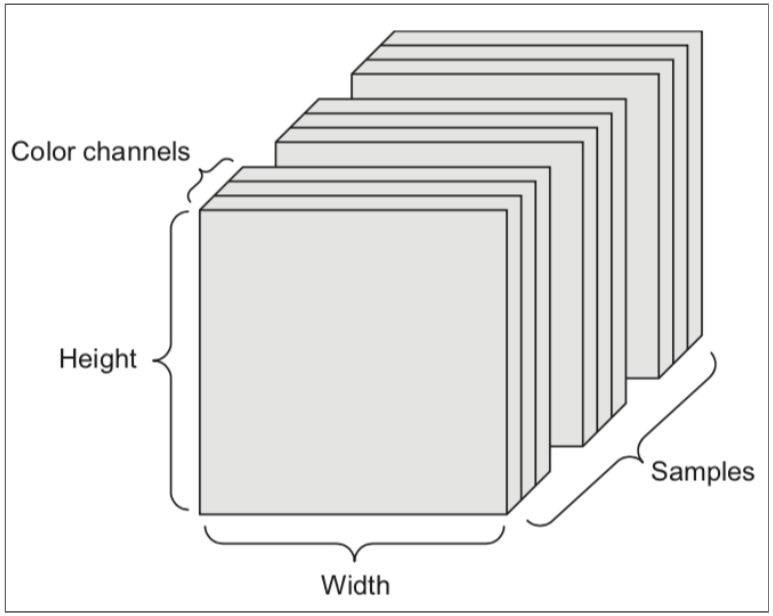
Notation and Definitions

- A *scalar* is a simple numeric value, denoted by an italic letter: $x = 3.24$
- A *vector* is a 1D ordered array of *n* scalars, denoted by a bold letter: $\mathbf{x} = [3.24, 1.2]$
    - x_i denotes the ith element of a vector, thus $x_0 = 3.24$.
        - Note: some other courses use $x^{(i)}$ notation
- A *set* is an *unordered* collection of unique elements, denote by caligraphic capital: $\mathcal{S} = \{3.24, 1.2\}$
- A *matrix* is a 2D array of scalars, denoted by bold capital: $\mathbf{X} = \begin{bmatrix} 3.24 & 1.2 \\ 2.24 & 0.2 \end{bmatrix}$
    - $\mathbf{X}_i$ denotes the ith *row* of the matrix
    - $\mathbf{X}_{:,j}$ denotes the jth *column*
    - $\mathbf{X}_{i,j}$ denotes the *element* in the ith row, jth column, thus $\mathbf{X}_{1,0} = 2.24$

- $\mathbf{X}^{n \times p}$, an $n \times p$ matrix, can represent n data points in a p-dimensional space
  - Every row is a vector that can represent a *point* in an n-dimensional space, given a *basis*.
  - The *standard basis* for a Euclidean space is the set of unit vectors
- E.g. if $\mathbf{X} = \begin{bmatrix} 3.24 & 1.2 \\ 2.24 & 0.2 \\ 3.0 & 0.6 \end{bmatrix}$



- A *tensor* is an *k*-dimensional array of data, denoted by an italic capital: T
    - *k* is also called the order, degree, or rank
    - $T_{i,j,k,...}$ denotes the element or sub-tensor in the corresponding position
    - A set of color images can be represented by:
        - a 4D tensor (sample x height x width x color channel)
        - a 2D tensor (sample x flattened vector of pixel values)



Basic operations

- Sums and products are denoted by capital Sigma and capital Pi:

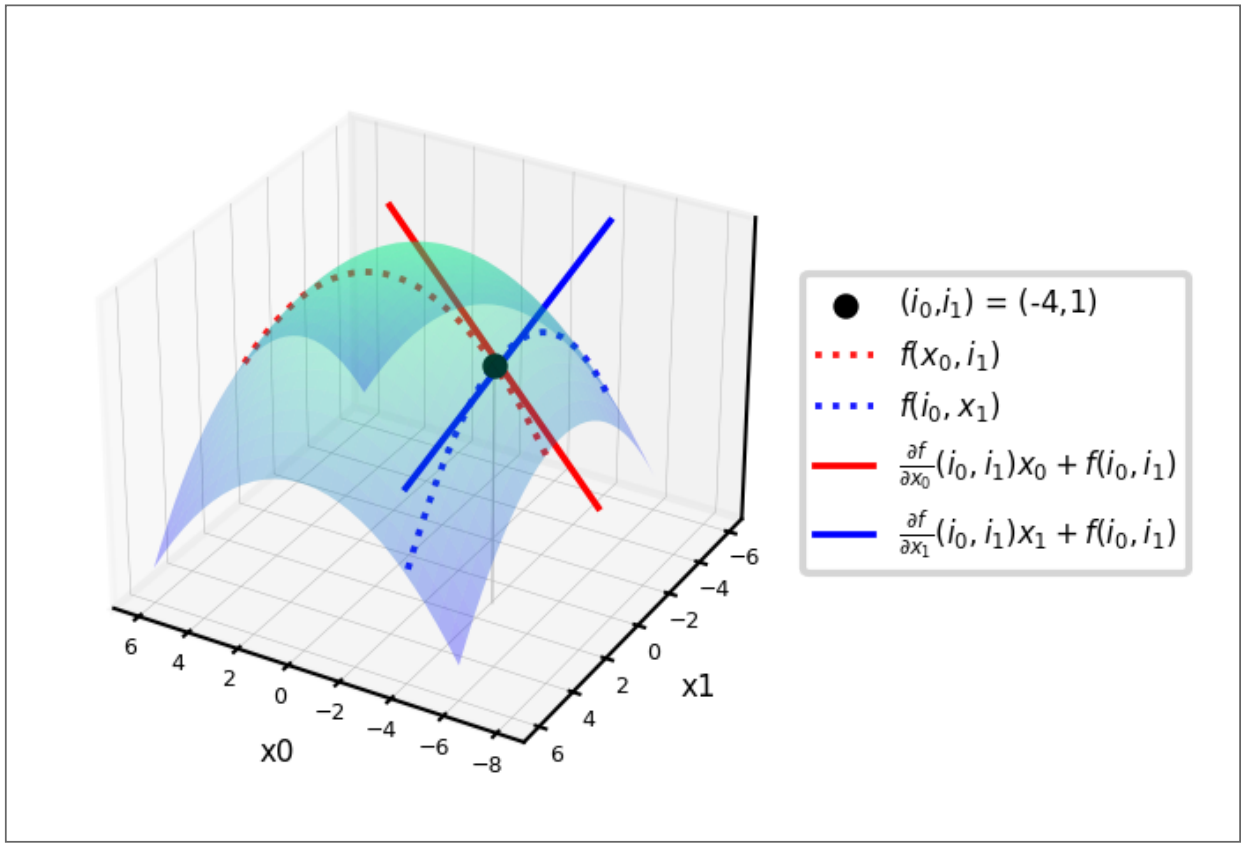
$$\sum_{i=0}^{p} = x_0 + x_1 + \ldots + x_p \quad \prod_{i=0}^{p} = x_0 \cdot x_1 \cdot \ldots \cdot x_p$$

- Operations on vectors are element-wise: e.g. $\mathbf{x} + \mathbf{z} = [x_0 + z_0, x_1 + z_1, \ldots, x_p + z_p]$
- Dot product $\mathbf{wx} = \mathbf{w} \cdot \mathbf{x} = \mathbf{w}^T\mathbf{x} = \sum_{i=0}^{p} w_i \cdot x_i = w_0 \cdot x_0 + w_1 \cdot x_1 + \ldots + w_p \cdot x_p$
- Matrix product $\mathbf{Wx} = \begin{bmatrix} \mathbf{w_0} \cdot \mathbf{x} \\ \ldots \\ \mathbf{w_p} \cdot \mathbf{x} \end{bmatrix}$
- A function $f(x) = y$ relates an input element x to an output y
  - It has a *local minimum* at $x = c$ if $f(x) \geq f(c)$ in interval $(c - \epsilon, c + \epsilon)$
  - It has a *global minimum* at $x = c$ if $f(x) \geq f(c)$ for any value for x
- A vector function consumes an input and produces a vector: $\mathbf{f}(\mathbf{x}) = \mathbf{y}$
- $\max_{x \in X} f(x)$ returns the highest value f(x) for any x
- $\operatorname{argmax}_{x \in X} f(x)$ returns the element x that maximizes f(x)

Gradients

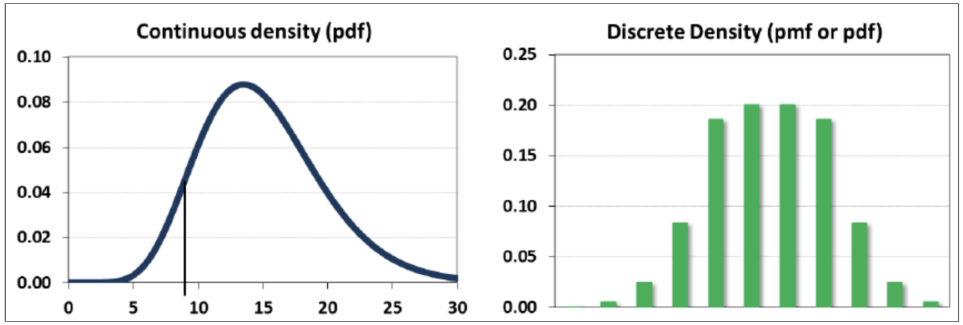
- A *derivative* f' of a function f describes how fast f grows or decreases
- The process of finding a derivative is called differentiation
    - Derivatives for basic functions are known
    - For non-basic functions we use the chain rule: $F(x) = f(g(x)) \rightarrow F'(x) = f'(g(x))g'(x)$
- A function is *differentiable* if it has a derivate in any point of it's domain
    - It's *continuously differentiable* if f' is itself a function
    - It's *smooth* if $f', f'', f''', \ldots$ all exist
- A *gradient* ∇f is the derivate of a function in multiple dimensions
    - It is a vector of partial derivatives: $\nabla f = \left[\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, \ldots\right]$
    - E.g. $f = 2x_0 + 3x_1^2 - \sin(x_2) \rightarrow \nabla f = [2, 6x_1, -cos(x_2)]$

- Example: $f = -(x_0^2 + x_1^2)$
  - $\nabla f = \left[\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}\right] = [-2x_0, -2x_1]$
  - Evaluated at point (-4,1): $\nabla f(-4, 1) = [8, -2]$
    - These are the slopes at point (-4,1) in the direction of x_0 and x_1 respectively



Distributions and Probabilities

- The normal (Gaussian) distribution with mean μ and standard deviation σ is noted as $N(\mu, \sigma)$
- A random variable X can be continuous or discrete
- A probability distribution f_X of a continuous variable X: *probability density function* (pdf)
    - The *expectation* is given by $\mathbb{E}[X] = \int x f_X(x) dx$
- A probability distribution of a discrete variable: *probability mass function* (pmf)
    - The *expectation* (or mean) $\mu_X = \mathbb{E}[X] = \sum_{i=1}^{k} [x_i \cdot Pr(X = x_i)]$



Linear models

Linear models make a prediction using a linear function of the input features X

$$f_{\mathbf{w}}(\mathbf{x}) = \sum_{i=1}^{p} w_i \cdot x_i + w_0$$

Learn w from X, given a loss function $\mathcal{L}$:

$$\underset{\mathbf{w}}{\operatorname{argmin}} \mathcal{L}(f_{\mathbf{w}}(X))$$

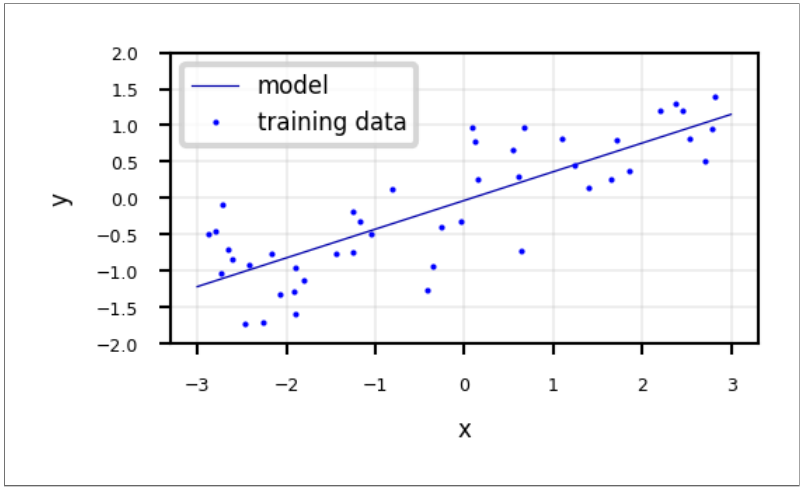
- Many algorithms with different $\mathcal{L}$: Least squares, Ridge, Lasso, Logistic Regression, Linear SVMs,...
- Can be very powerful (and fast), especially for large datasets with many features.
- Can be generalized to learn non-linear patterns: *Generalized Linear Models*
    - Features can be augmentented with polynomials of the original features
    - Features can be transformed according to a distribution (Poisson, Tweedie, Gamma,...)
    - Some linear models (e.g. SVMs) can be *kernelized* to learn non-linear functions

Linear models for regression

- Prediction formula for input features x:
  - $w_1 \ldots w_p$ usually called *weights* or *coefficients* , w_0 the *bias* or *intercept*
  - Assumes that errors are $N(0, \sigma)$

$$\hat{y} = \mathbf{w}\mathbf{x} + w_0 = \sum_{i=1}^{p} w_i \cdot x_i + w_0 = w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_p \cdot x_p + w_0$$

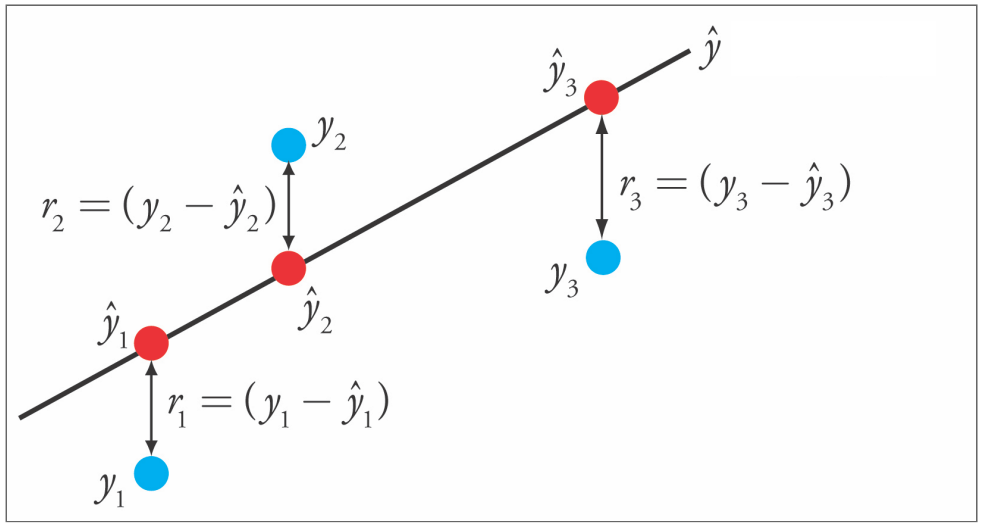
```
w_1: 0.393906  w_0: -0.031804
```



Linear Regression (aka Ordinary Least Squares)

- Loss function is the *sum of squared errors* (SSE) (or residuals) between predictions $\hat{y}_i$ (red) and the true regression targets y_i (blue) on the training set.

$$\mathcal{L}_{SSE} = \sum_{n=1}^{N} (y_n - \hat{y}_n)^2 = \sum_{n=1}^{N} (y_n - (\mathbf{w}\mathbf{x_n} + w_0))^2$$



Solving ordinary least squares

- Convex optimization problem with unique closed-form solution:

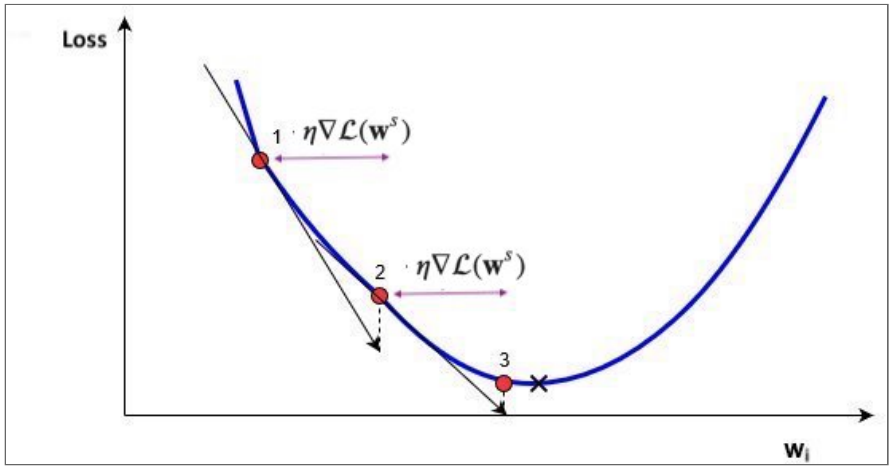
$$w^* = (X^T X)^{-1} X^T Y$$

  - Add a column of 1's to the front of X to get w_0
  - Slow. Time complexity is quadratic in number of features: $\mathcal{O}(p^2 n)$
    - X has n rows, p features, hence $X^T X$ has dimensionality $p \cdot p$
  - Only works if $n > p$
- *Gradient Descent*
  - Faster for large and/or high-dimensional datasets
  - When $X^T X$ cannot be computed or takes too long (p or n is too large)
- **Very easily overfits**.
  - coefficients w become very large (steep incline/decline)
  - small change in the input *x* results in a very different output *y*
  - No hyperparameters that control model complexity

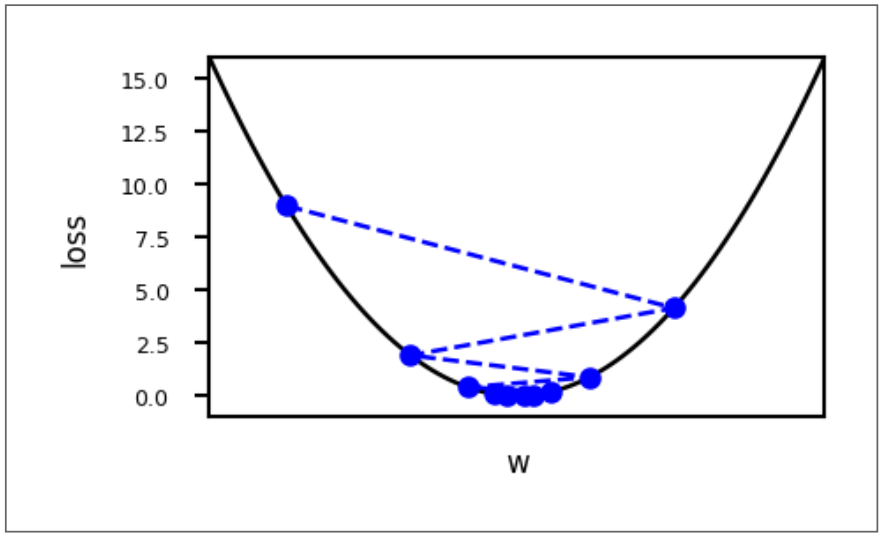
Gradient Descent

- Start with an initial, random set of weights: $\mathbf{w}^0$
- Given a differentiable loss function $\mathcal{L}$ (e.g. $\mathcal{L}_{SSE}$), compute $\nabla\mathcal{L}$
- For least squares: $\frac{\partial \mathcal{L}_{SSE}}{\partial w_i}(\mathbf{w}) = -2\sum_{n=1}^{N}(y_n - \hat{y}_n)x_{n,i}$
    - If feature $X_{:,i}$ is associated with big errors, the gradient wrt w_i will be large
- Update *all* weights slightly (by *step size* or *learning rate* η) in 'downhill' direction.
- Basic *update rule* (step s):

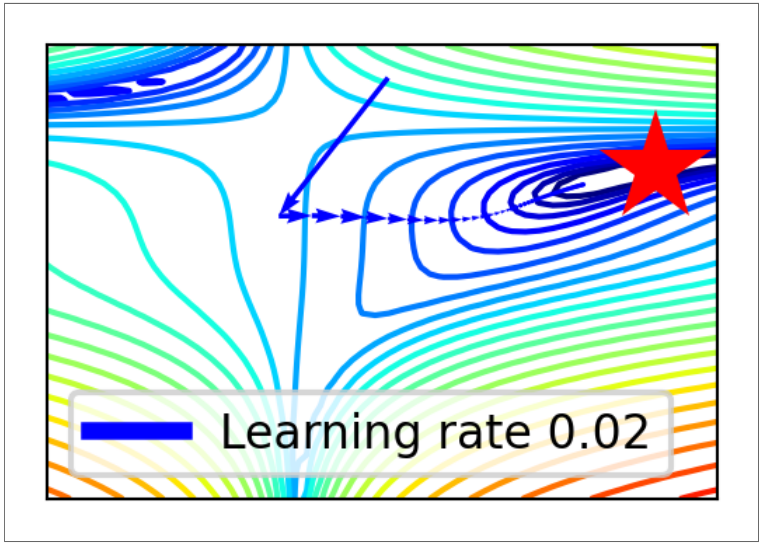
$$\mathbf{w}^{s+1} = \mathbf{w}^s - \eta\nabla\mathcal{L}(\mathbf{w}^s)$$



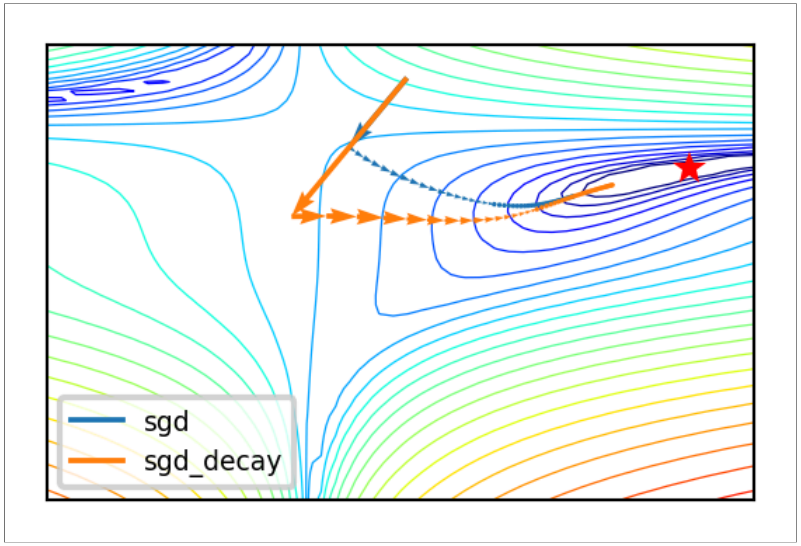
- Important hyperparameters
    - Learning rate
        - Too small: slow convergence. Too large: possible divergence
    - Maximum number of iterations
        - Too small: no convergence. Too large: wastes resources
    - Learning rate decay with decay rate k
        - E.g. exponential ($\eta^{s+1} = \eta^0 e^{-ks}$), inverse-time ($\eta^{s+1} = \frac{\eta^s}{1+ks}$),...
    - Many more advanced ways to control learning rate (see later)
        - Adaptive techniques: depend on how much loss improved in previous step



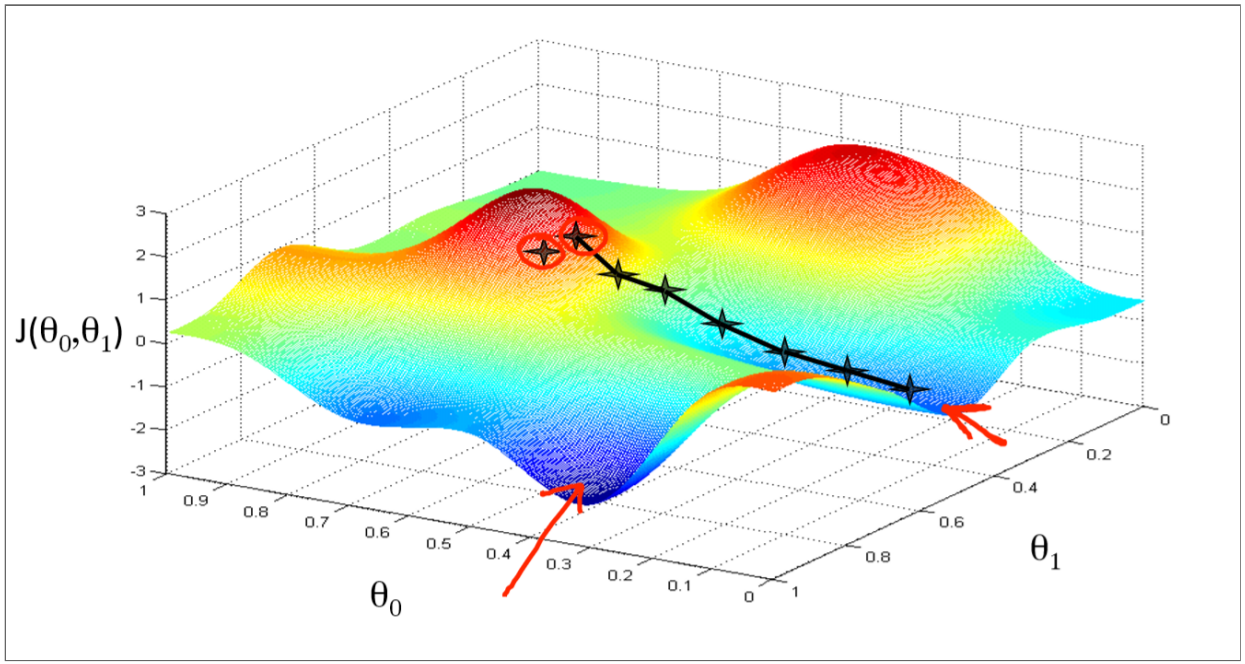
Effect of learning rate



Effect of learning rate decay

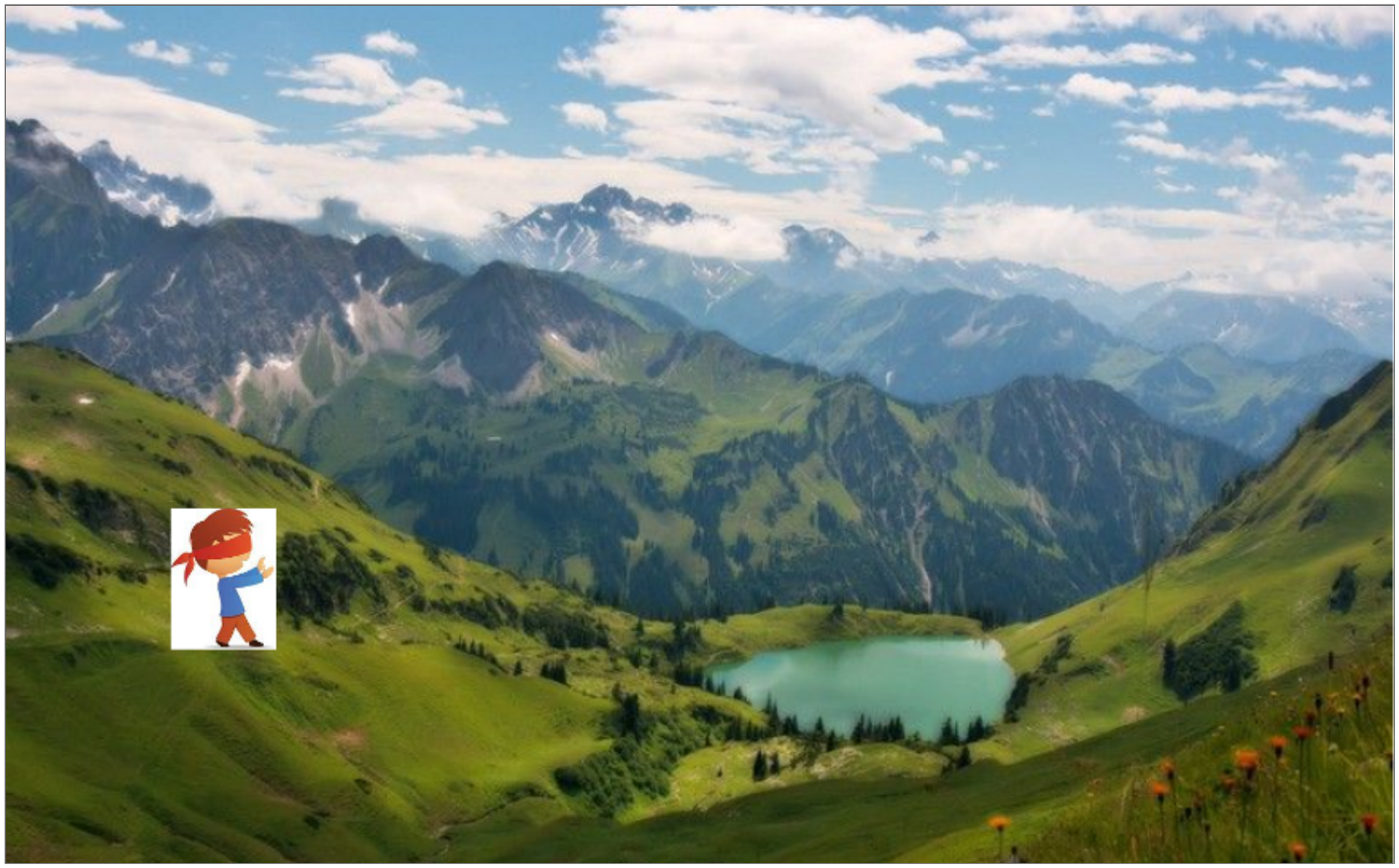


In two dimensions:



- You can get stuck in local minima (if the loss is not fully convex)
    - If you have many model parameters, this is less likely
    - You always find a way down in some direction
    - Models with many parameters typically find good local minima

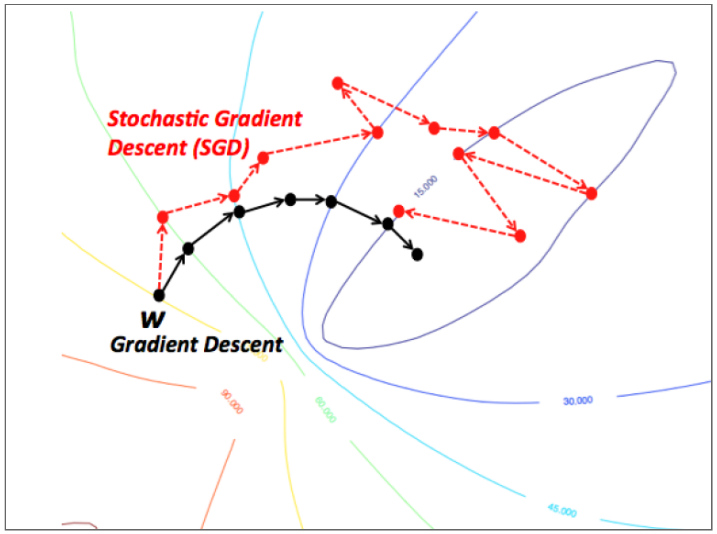
- Intuition: walking downhill using only the slope you "feel" nearby



(Image by A. Karpathy)

Stochastic Gradient Descent (SGD)

- Compute gradients not on the entire dataset, but on a single data point i at a time
  - Gradient descent: $\mathbf{w}^{s+1} = \mathbf{w}^s - \eta \nabla \mathcal{L}(\mathbf{w}^s) = \mathbf{w}^s - \frac{\eta}{n} \sum_{i=1}^{n} \nabla \mathcal{L}_i(\mathbf{w}^s)$
  - Stochastic Gradient Descent: $\mathbf{w}^{s+1} = \mathbf{w}^s - \eta \nabla \mathcal{L}_i(\mathbf{w}^s)$
- Many smoother variants, e.g.
  - Minibatch SGD: compute gradient on batches of data: $\mathbf{w}^{s+1} = \mathbf{w}^s - \frac{\eta}{B} \sum_{i=1}^{B} \nabla \mathcal{L}_i(\mathbf{w}^s)$
  - Stochastic Average Gradient Descent (**SAG**, **SAGA**). With $i_s \in [1, n]$ randomly chosen per iteration:
    - Incremental gradient: $\mathbf{w}^{s+1} = \mathbf{w}^s - \frac{\eta}{n} \sum_{i=1}^{n} v_i^s$ with $v_i^s = \begin{cases} \nabla \mathcal{L}_i(\mathbf{w}^s) & i = i_s \\ v_i^{s-1} & \text{otherwise} \end{cases}$



In practice

- Linear regression can be found in `sklearn.linear_model`. We'll evaluate it on the Boston Housing dataset.
    - `LinearRegression` uses closed form solution, `SGDRegressor` with `loss='squared_loss'` uses Stochastic Gradient Descent
    - Large coefficients signal overfitting
    - Test score is much lower than training score

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression().fit(X_train, y_train)
```

```
Weights (coefficients): [ -412.711   -52.243  -131.899   -12.004   -15.511    28.716    54.704
   -49.535    26.582    37.062   -11.828   -18.058   -19.525    12.203
  2980.781  1500.843   114.187   -16.97     40.961   -24.264    57.616
  1278.121 -2239.869   222.825    -2.182    42.996   -13.398   -19.389
    -2.575   -81.013     9.66      4.914    -0.812    -7.647    33.784
   -11.446    68.508   -17.375    42.813     1.14 ]
Bias (intercept): 30.93456367364179


Training set score (R^2): 0.95
Test set score (R^2): 0.61
```

Ridge regression

- Adds a penalty term to the least squares loss function:

$$\mathcal{L}_{Ridge} = \sum_{n=1}^{N}(y_n - (\mathbf{w}\mathbf{x_n} + w_0))^2 + \alpha \sum_{i=1}^{p} w_i^2$$

- Model is penalized if it uses large coefficients (w)
    - Each feature should have as little effect on the outcome as possible
    - We don't want to penalize w_0, so we leave it out
- Regularization: explicitly restrict a model to avoid overfitting.
    - Called L2 regularization because it uses the L2 norm: $\sum w_i^2$
- The strength of the regularization can be controlled with the α hyperparameter.
    - Increasing α causes more regularization (or shrinkage). Default is 1.0.
- Still convex. Can be optimized in different ways:
    - Closed form solution (a.k.a. Cholesky): $w^* = (X^T X + \alpha I)^{-1} X^T Y$
    - Gradient descent and variants, e.g. Stochastic Average Gradient (SAG,SAGA)
        - Conjugate gradient (CG): each new gradient is influenced by previous ones
    - Use Cholesky for smaller datasets, Gradient descent for larger ones

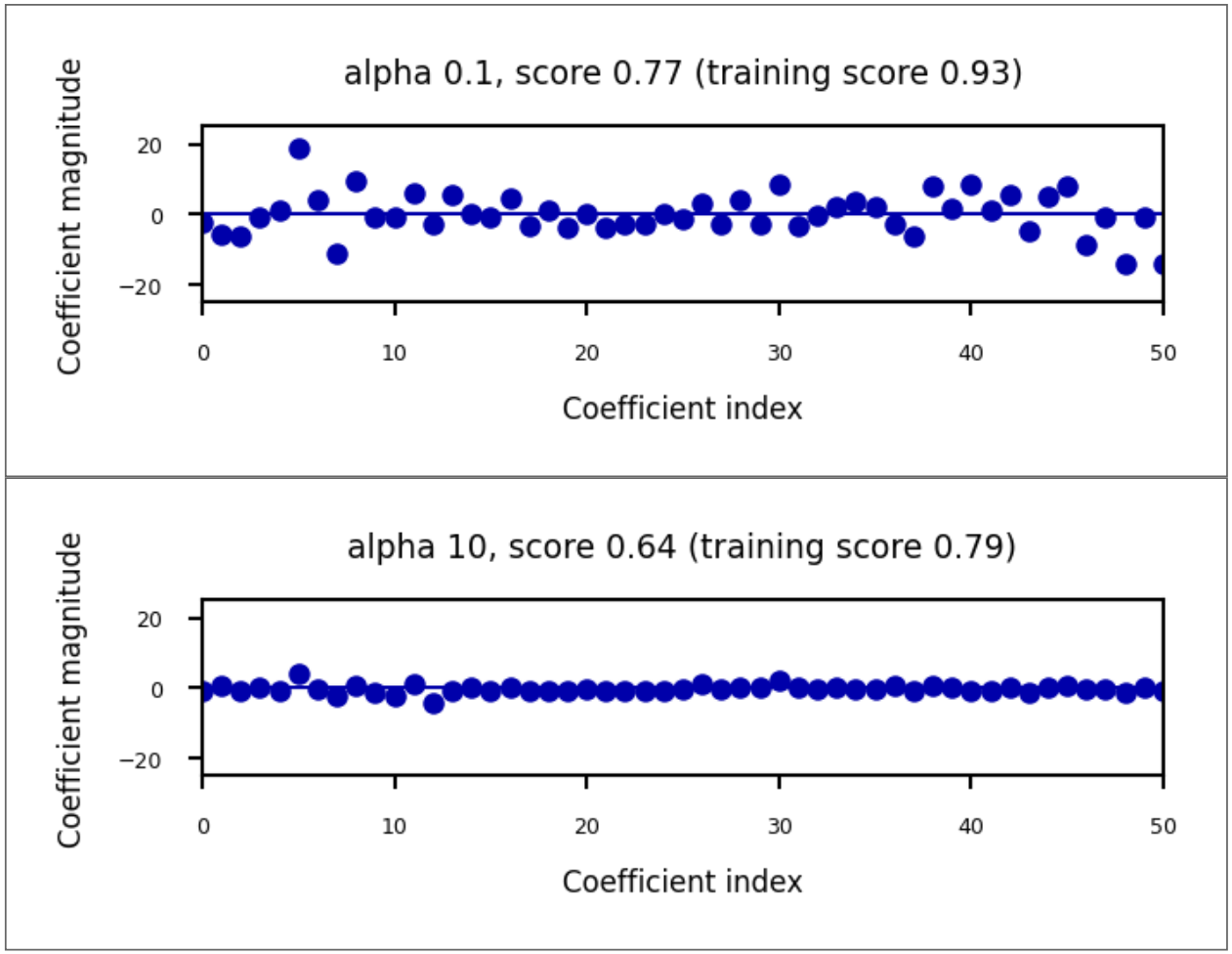
In practice

```
from sklearn.linear_model import Ridge
lr = Ridge().fit(X_train, y_train)
```

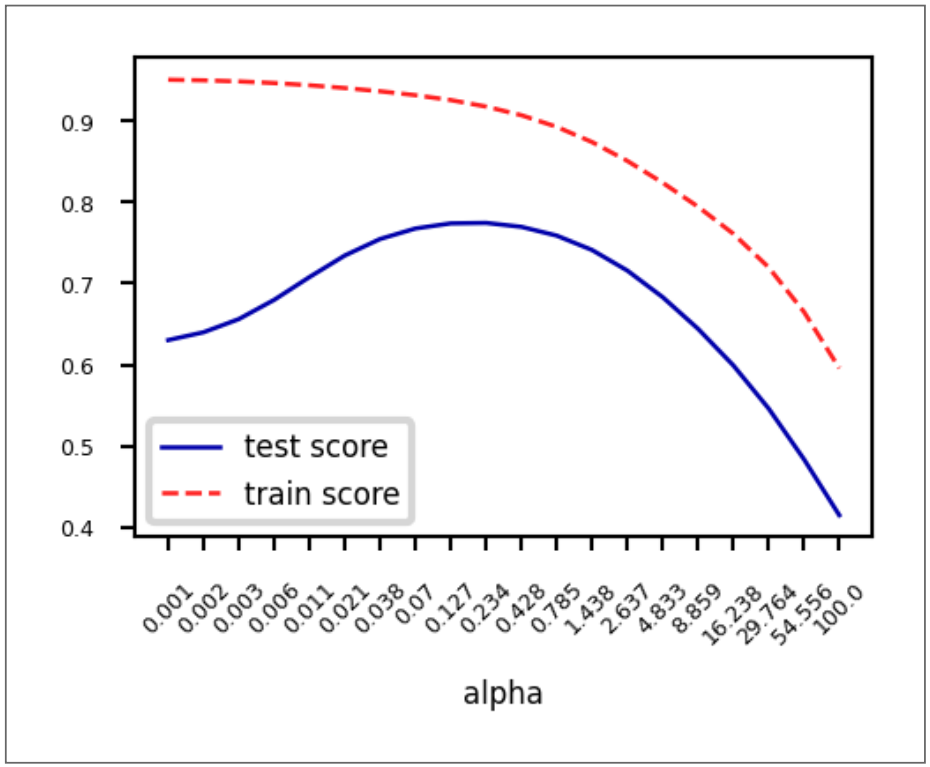
```
Weights (coefficients): [-1.414 -1.557 -1.465 -0.127 -0.079  8.332  0.255 -4.941  3.899 -1.059
 -1.584  1.051 -4.012  0.334  0.004 -0.849  0.745 -1.431 -1.63  -1.405
 -0.045 -1.746 -1.467 -1.332 -1.692 -0.506  2.622 -2.092  0.195 -0.275
  5.113 -1.671 -0.098  0.634 -0.61   0.04  -1.277 -2.913  3.395  0.792]
Bias (intercept): 21.39052595861006
Training set score: 0.89
Test set score: 0.75
```

Test set score is higher and training set score lower: less overfitting!

- We can plot the weight values for differents levels of regularization to explore the effect of α.
- Increasing regularization decreases the values of the coefficients, but never to 0.

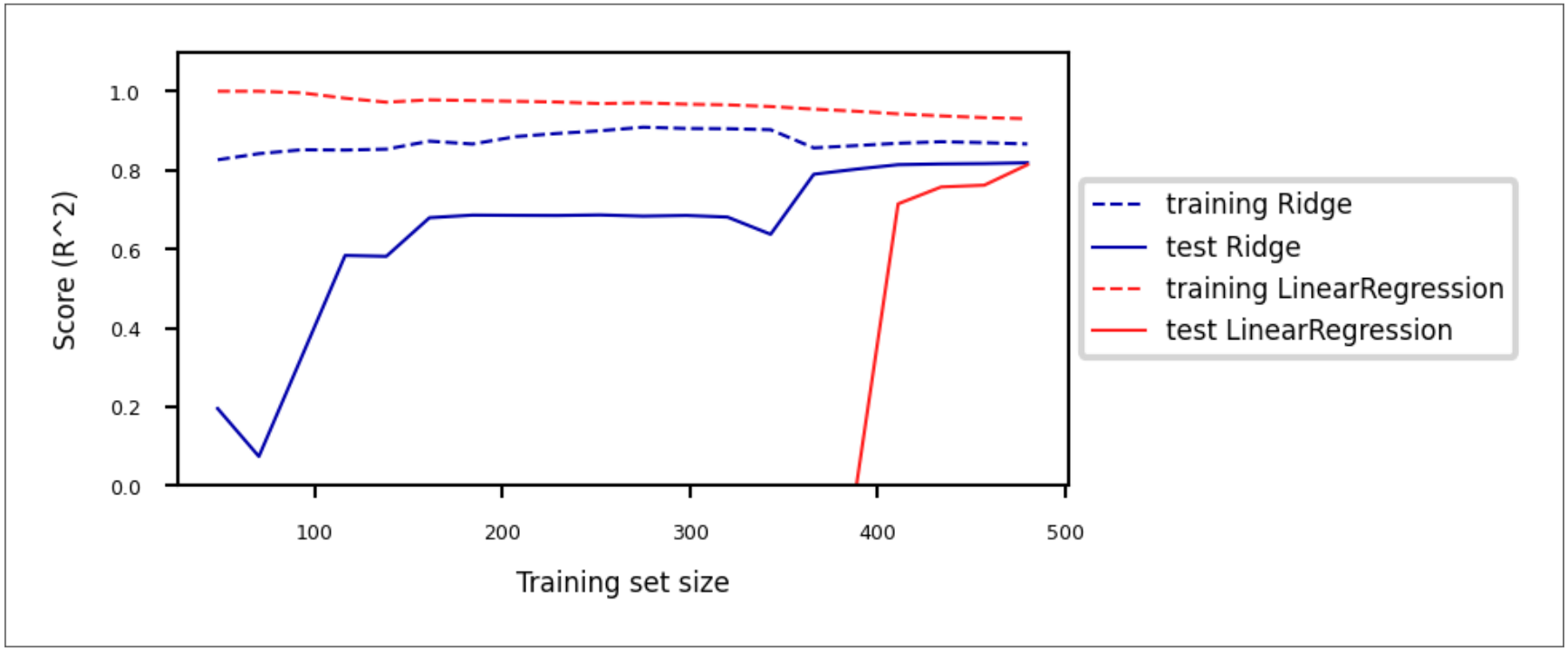


- When we plot the train and test scores for every α value, we see a sweet spot around $\alpha = 0.2$
    - Models with smaller α are overfitting
    - Models with larger α are underfitting



Other ways to reduce overfitting

- Add more training data: with enough training data, regularization becomes less important
    - Ridge and ordinary least squares will have the same performance
- Use fewer features: remove unimportant ones or find a low-dimensional embedding (e.g. PCA)
    - Fewer coefficients to learn, reduces the flexibility of the model
- Scaling the data typically helps (and changes the optimal α value)



Lasso (Least Absolute Shrinkage and Selection Operator)

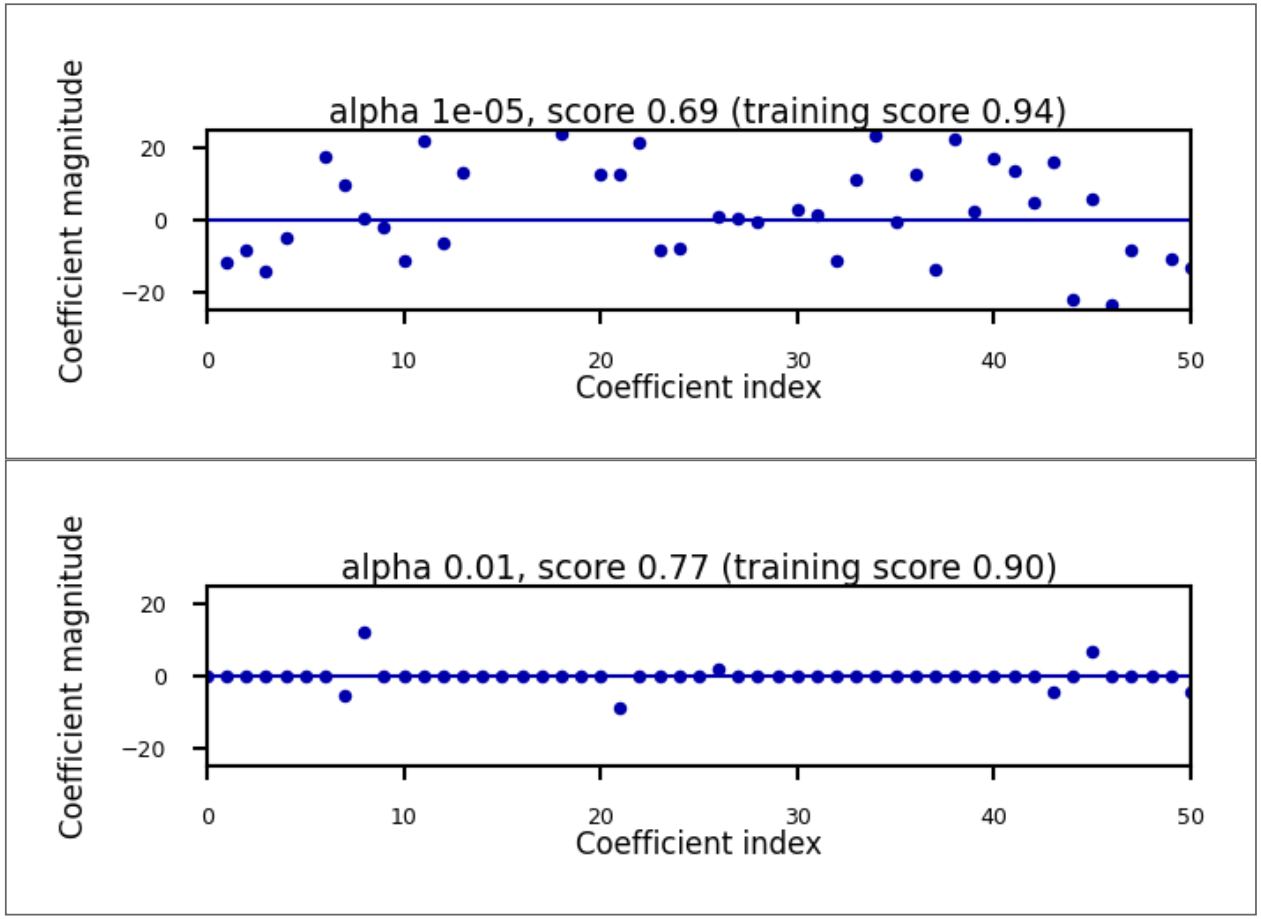
- Adds a different penalty term to the least squares sum:

$$\mathcal{L}_{Lasso} = \sum_{n=1}^{N}(y_n - (\mathbf{w}\mathbf{x_n} + w_0))^2 + \alpha \sum_{i=1}^{p} |w_i|$$

- Called L1 regularization because it uses the L1 norm
    - Will cause many weights to be exactly 0
- Same parameter α to control the strength of regularization.
    - Will again have a 'sweet spot' depending on the data
- No closed-form solution
- Convex, but no longer strictly convex, and not differentiable
    - Weights can be optimized using *coordinate descent*

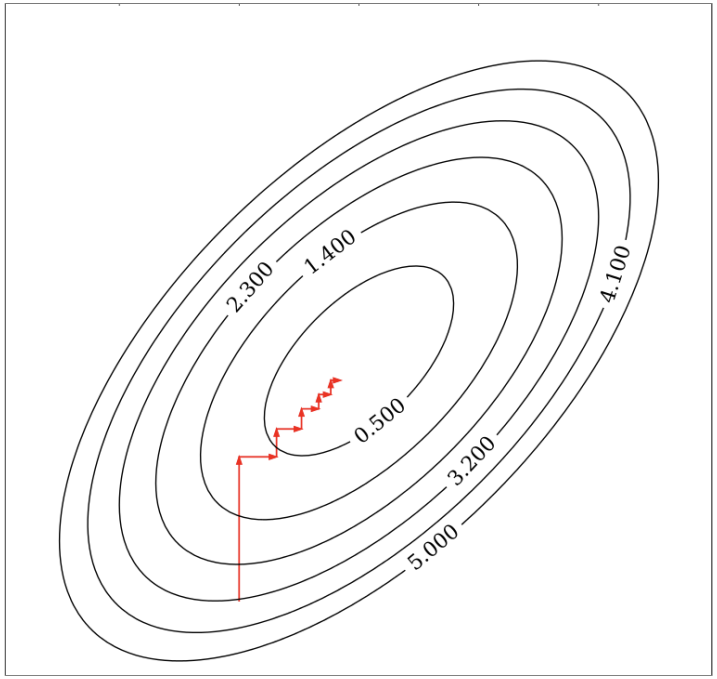
Analyze what happens to the weights:

- L1 prefers coefficients to be exactly zero (sparse models)
- Some features are ignored entirely: automatic feature selection
- How can we explain this?



Coordinate descent

- Alternative for gradient descent, supports non-differentiable convex loss functions (e.g. $\mathcal{L}_{Lasso}$)
- In every iteration, optimize a single coordinate w_i (find minimum in direction of x_i)
    - Continue with another coordinate, using a selection rule (e.g. round robin)
- Faster iterations. No need to choose a step size (learning rate).
- May converge more slowly. Can't be parallellized.



Coordinate descent with Lasso

- Remember that $\mathcal{L}_{Lasso} = \mathcal{L}_{SSE} + \alpha \sum_{i=1}^{p} |w_i|$
- For one w_i: $\mathcal{L}_{Lasso}(w_i) = \mathcal{L}_{SSE}(w_i) + \alpha |w_i|$
- The L1 term is not differentiable but convex: we can compute the ***subgradient***
  - Unique at points where $\mathcal{L}$ is differentiable, a range of all possible slopes [a,b] where it is not
  - For $|w_i|$, the subgradient $\partial_{w_i} |w_i|$ = $\begin{cases} -1 & w_i < 0 \\ [-1, 1] & w_i = 0 \\ 1 & w_i > 0 \end{cases}$
  - Subdifferential $\partial(f + g) = \partial f + \partial g$ if f and g are both convex
- To find the optimum for Lasso w_i^*, solve

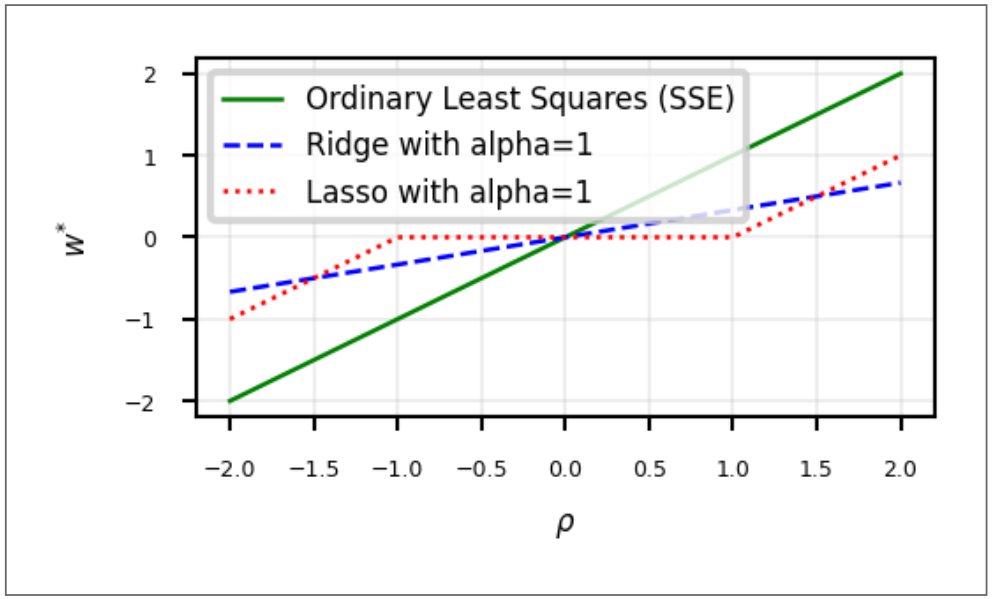
$$\begin{aligned} \partial_{w_i} \mathcal{L}_{Lasso}(w_i) &= \partial_{w_i} \mathcal{L}_{SSE}(w_i) + \partial_{w_i} \alpha |w_i| \\ 0 &= (w_i - \rho_i) + \alpha \cdot \partial_{w_i} |w_i| \\ w_i &= \rho_i - \alpha \cdot \partial_{w_i} |w_i| \end{aligned}$$

  - In which ρ_i is the solution for $\mathcal{L}_{SSE}(w_i)$

- We found: $w_i = \rho_i - \alpha \cdot \partial_{w_i} |w_i|$
- Lasso solution has the form of a *soft thresholding function* S

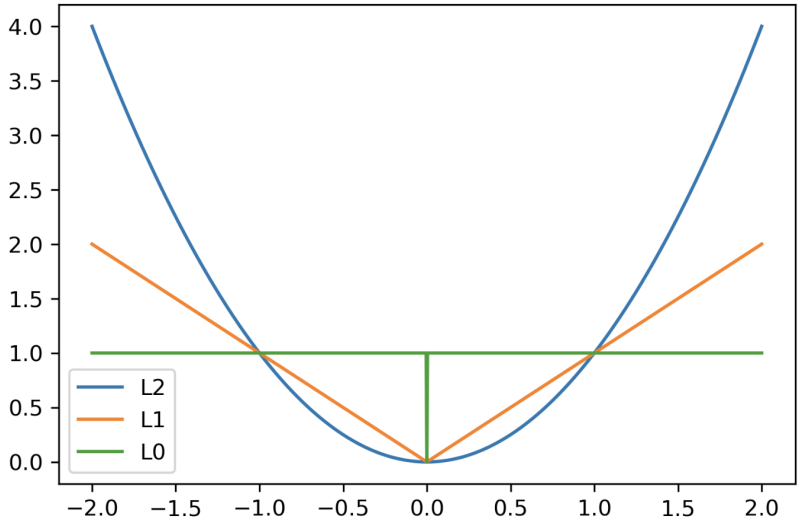
$$w_i^* = S(\rho_i, \alpha) = \begin{cases} \rho_i + \alpha, & \rho_i < -\alpha \\ 0, & -\alpha < \rho_i < \alpha \\ \rho_i - \alpha, & \rho_i > \alpha \end{cases}$$

  - Small weights become 0: sparseness!
  - If the data is not normalized, $w_i^* = \frac{1}{z_i} S(\rho_i, \alpha)$ with z_i a normalizing constant
- Ridge solution: $w_i = \rho_i - \alpha \cdot \partial_{w_i} w_i^2 = \rho_i - 2\alpha \cdot w_i$, thus $w_i^* = \frac{\rho_i}{1+2\alpha}$



Interpreting L1 and L2 loss

- L1 and L2 in function of the weights



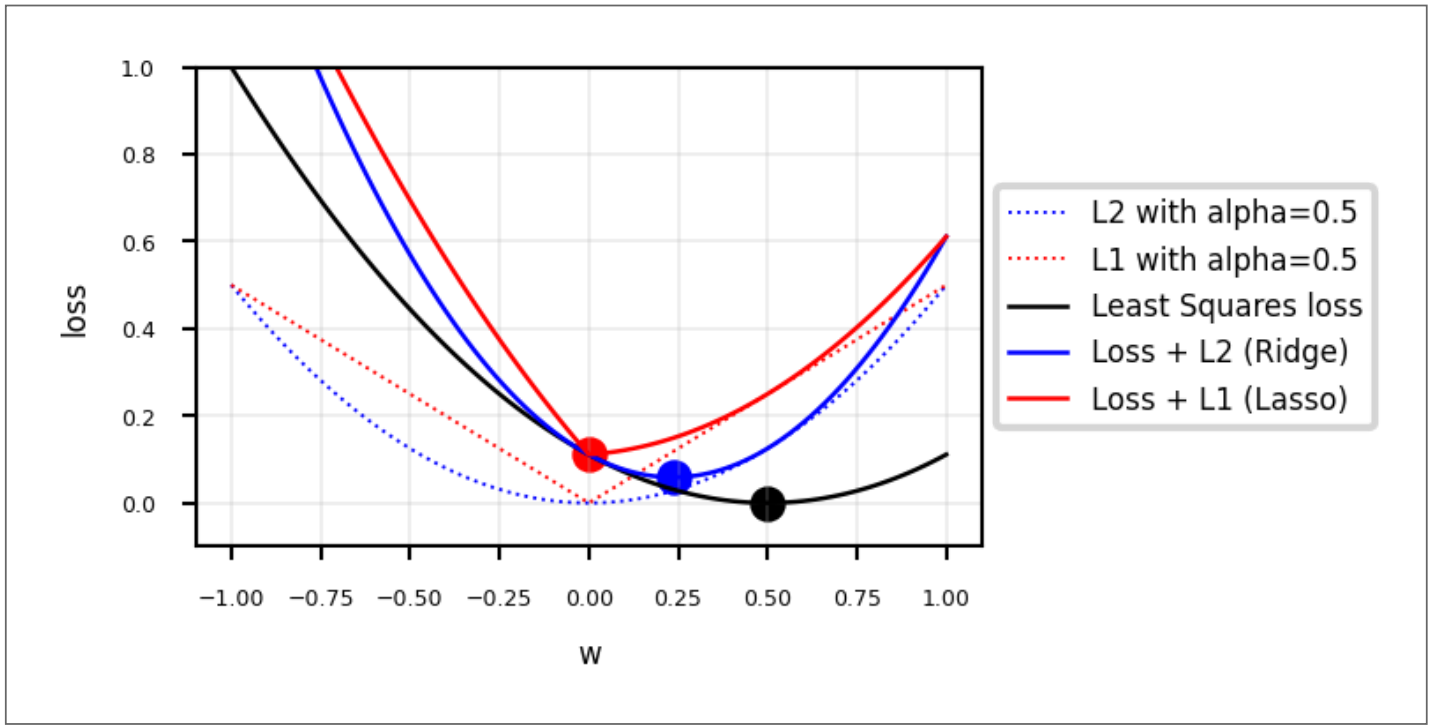
$$\ell_2(w) = \sqrt{\sum_i w_i^2}$$

$$\ell_1(w) = \sum_i |w_i|$$

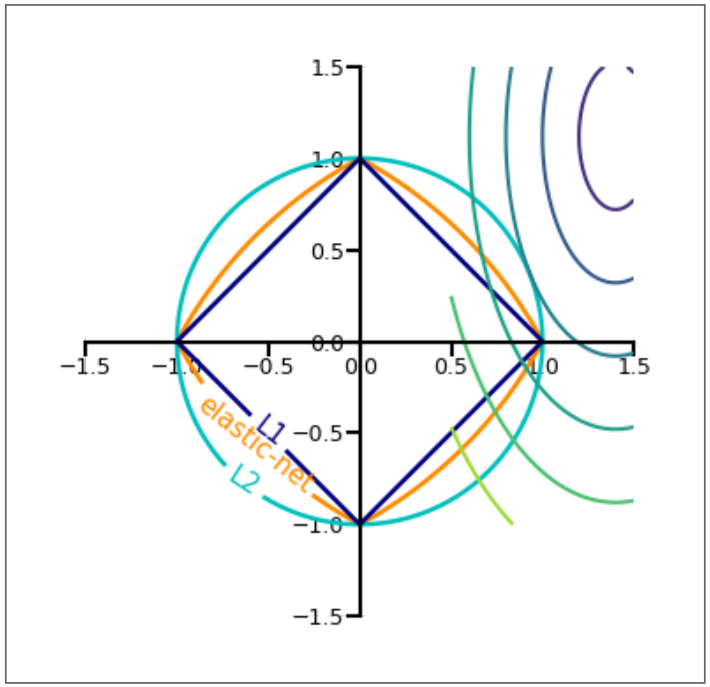
$$\ell_0(w) = \sum_i 1_{w_i != 0}$$

Least Squares Loss + L1 or L2

- Lasso is not differentiable at point 0
- For any minimum of least squares, L2 will be smaller, and L1 is more likely be 0



- In 2D (for 2 model weights w_1 and w_2)
    - The least squared loss is a 2D convex function in this space
    - For illustration, assume that L1 loss = L2 loss = 1
        - L1 loss ($\Sigma|w_i|$): every $\{w_1, w_2\}$ falls on the diamond
        - L2 loss (Σw_i^2): every $\{w_1, w_2\}$ falls on the circle
    - For L1, the loss is minimized if w_1 or w_2 is 0 (rarely so for L2)



Elastic-Net

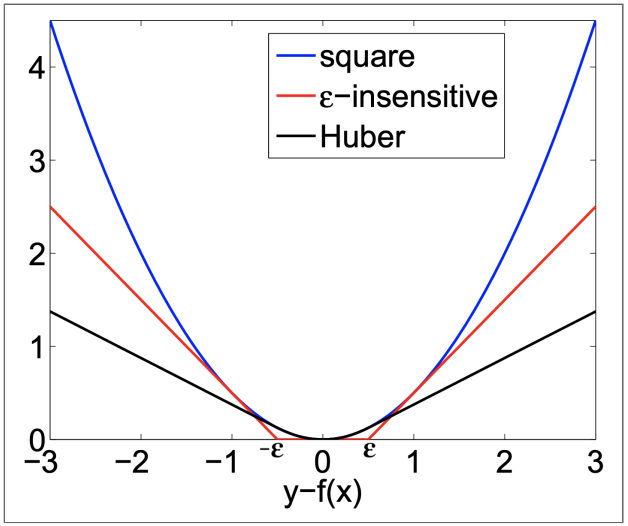
- Adds both L1 and L2 regularization:

$$\mathcal{L}_{Elastic} = \sum_{n=1}^{N}(y_n - (\mathbf{wx_n} + w_0))^2 + \alpha\rho\sum_{i=0}^{p}|w_i| + \alpha(1-\rho)\sum_{i=0}^{p}w_i^2$$

- ρ is the L1 ratio
    - With $\rho = 1$, $\mathcal{L}_{Elastic} = \mathcal{L}_{Lasso}$
    - With $\rho = 0$, $\mathcal{L}_{Elastic} = \mathcal{L}_{Ridge}$
    - $0 < \rho < 1$ sets a trade-off between L1 and L2.
- Allows learning sparse models (like Lasso) while maintaining L2 regularization benefits
    - E.g. if 2 features are correlated, Lasso likely picks one randomly, Elastic-Net keeps both
- Weights can be optimized using coordinate descent (similar to Lasso)

Other loss functions for regression

- Huber loss: switches from squared loss to linear loss past a value ϵ
    - More robust against outliers
- Epsilon insensitive: ignores errors smaller than ϵ, and linear past that
    - Aims to fit function so that residuals are at most ϵ
    - Also known as Support Vector Regression (`SVR` in sklearn)
- Squared Epsilon insensitive: ignores errors smaller than ϵ, and squared past that
- These can all be solved with stochastic gradient descent
    - `SGDRegressor` in sklearn

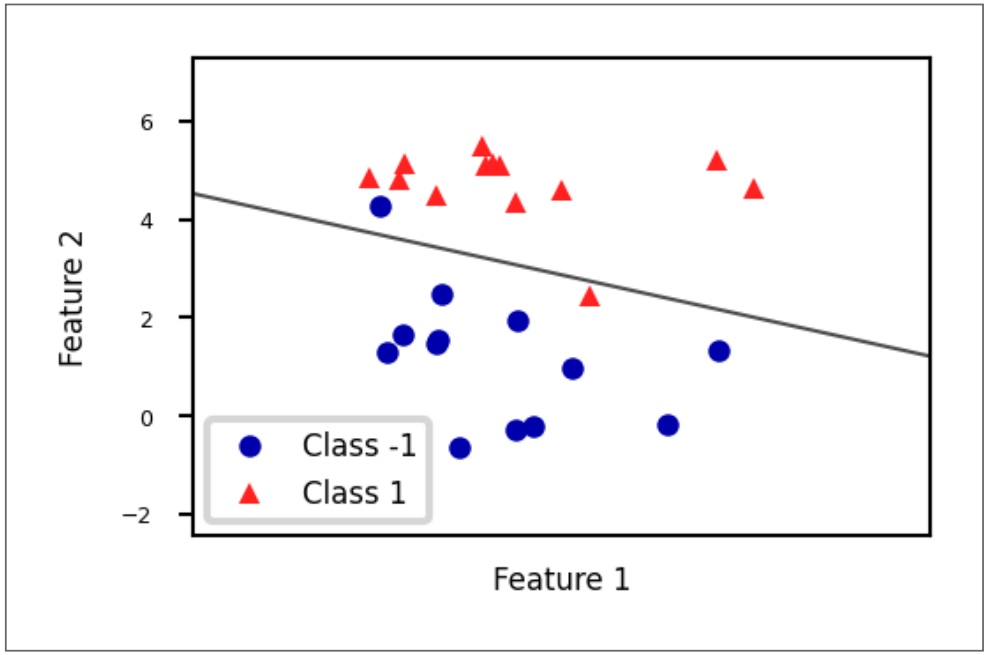


Linear models for Classification

Aims to find a hyperplane that separates the examples of each class.
For binary classification (2 classes), we aim to fit the following function:

$\hat{y} = w_1 * x_1 + w_2 * x_2 + \ldots + w_p * x_p + w_0 > 0$

When $\hat{y} < 0$, predict class -1, otherwise predict class +1



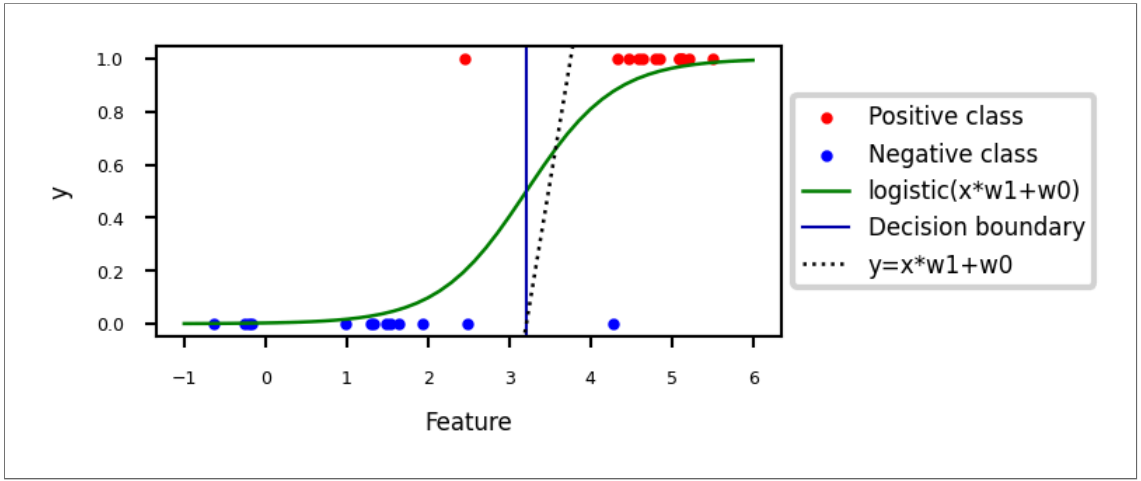
- There are many algorithms for linear classification, differing in loss function, regularization techniques, and optimization method
- Most common techniques:
    - Convert target classes {neg,pos} to {0,1} and treat as a regression task
        - Logistic regression (Log loss)
        - Ridge Classification (Least Squares + L2 loss)
    - Find hyperplane that maximizes the margin between classes
        - Linear Support Vector Machines (Hinge loss)
    - Neural networks without activation functions
        - Perceptron (Perceptron loss)
    - SGDClassifier: can act like any of these by choosing loss function
        - Hinge, Log, Modified_huber, Squared_hinge, Perceptron

Logistic regression

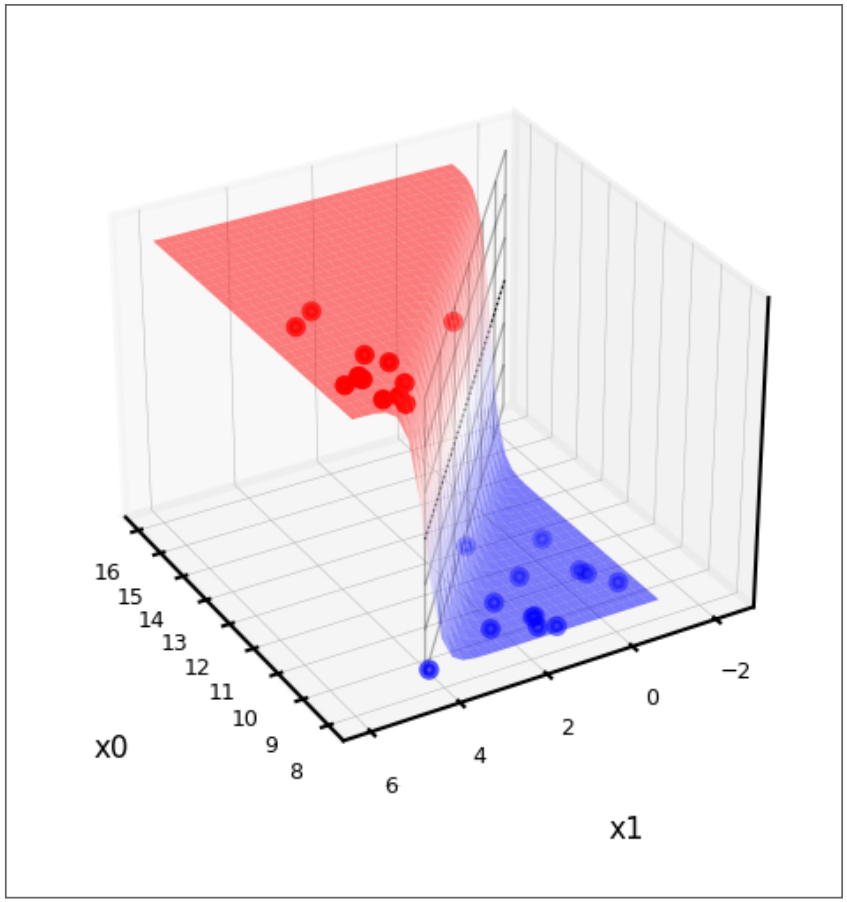
- Aims to predict the *probability* that a point belongs to the positive class
- Converts target values {negative (blue), positive (red)} to {0,1}
- Fits a *logistic* (or *sigmoid* or *S* curve) function through these points
    - Maps (-Inf,Inf) to a probability [0,1]

$$\hat{y} = \text{logistic}(f_\theta(\mathbf{x})) = \frac{1}{1 + e^{-f_\theta(\mathbf{x})}}$$

- E.g. in 1D: $\text{logistic}(x_1 w_1 + w_0) = \frac{1}{1+e^{-x_1 w_1 - w_0}}$



- Fitted solution to our 2D example:
    - To get a binary prediction, choose a probability threshold (e.g. 0.5)

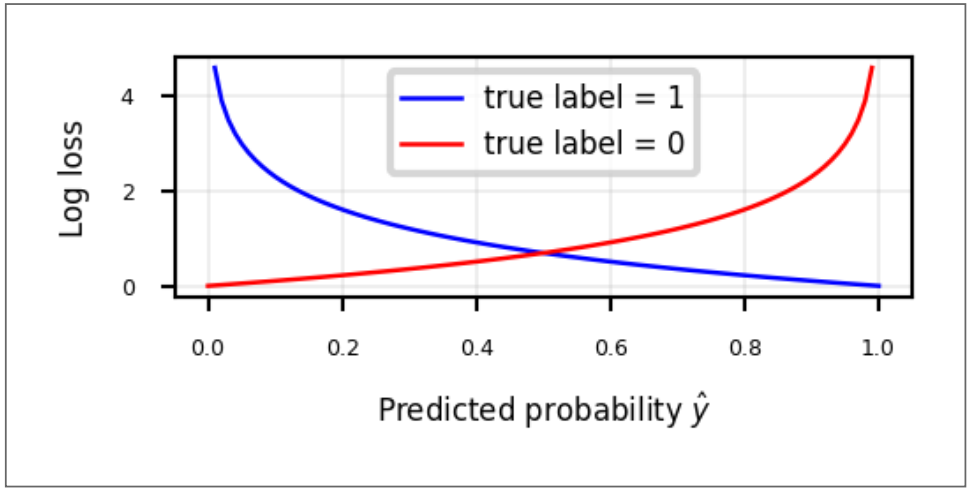


Loss function: Cross-entropy

- Models that return class probabilities can use *cross-entropy loss*

$$\mathcal{L}_{log}(\mathbf{w}) = \sum_{n=1}^{N} H(p_n, q_n) = -\sum_{n=1}^{N}\sum_{c=1}^{C} p_{n,c} log(q_{n,c})$$

  - Also known as log loss, logistic loss, or maximum likelihood
  - Based on true probabilities p (0 or 1) and predicted probabilities q over N instances and C classes
    - Binary case (C=2): $\mathcal{L}_{log}(\mathbf{w}) = -\sum_{n=1}^{N} \left[y_n log(\hat{y}_n) + (1-y_n) log(1-\hat{y}_n) \right]$
  - Penalty (or surprise) grows exponentially as difference between p and q increases
  - Often used together with L2 (or L1) loss: $\mathcal{L}_{log}{}'(\mathbf{w}) = \mathcal{L}_{log}(\mathbf{w}) + \alpha \sum_i w_i^2$



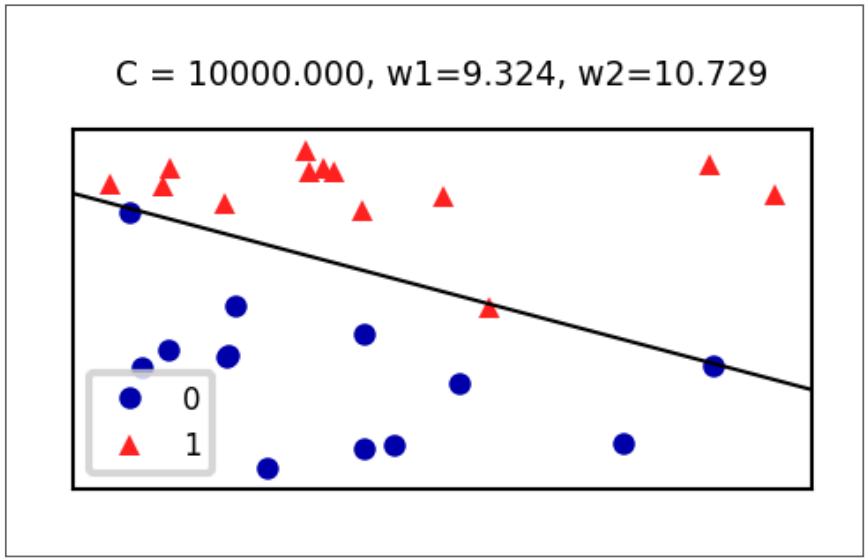
Optimization methods (solvers) for cross-entropy loss

- Gradient descent (only supports L2 regularization)
    - Log loss is differentiable, so we can use (stochastic) gradient descent
    - Variants thereof, e.g. Stochastic Average Gradient (SAG, SAGA)
- Coordinate descent (supports both L1 and L2 regularization)
    - Faster iteration, but may converge more slowly, has issues with saddlepoints
    - Called `liblinear` in sklearn. Can't run in parallel.
- Newton-Rhapson or Newton Conjugate Gradient (only L2):
    - Uses the Hessian $H = \left[\frac{\partial^2 \mathcal{L}}{\partial x_i \partial x_j}\right]$: $\mathbf{w}^{s+1} = \mathbf{w}^s - \eta H^{-1}(\mathbf{w}^s)\nabla\mathcal{L}(\mathbf{w}^s)$
    - Slow for large datasets. Works well if solution space is (near) convex
- Quasi-Newton methods (only L2)
    - Approximate, faster to compute
    - E.g. Limited-memory Broyden–Fletcher–Goldfarb–Shanno (`lbfgs`)
        - Default in sklearn for Logistic Regression
- **Some hints on choosing solvers**
    - Data scaling helps convergence, minimizes differences between solvers

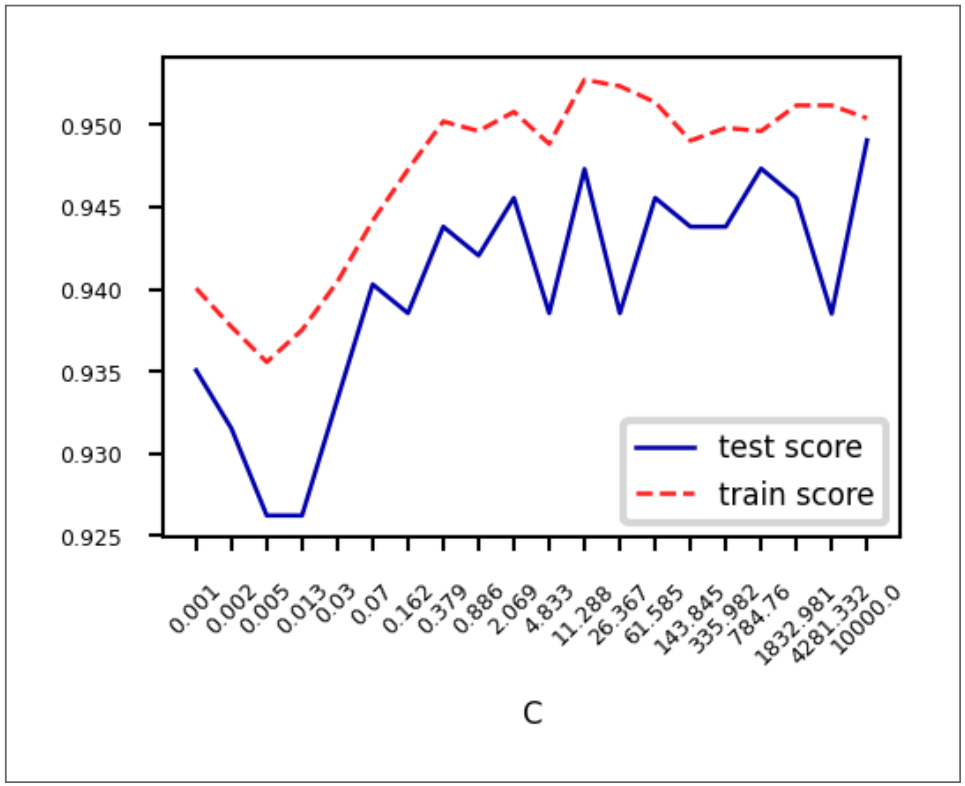
In practice

- Logistic regression can also be found in `sklearn.linear_model`.
    - `C` hyperparameter is the *inverse* regularization strength: $C = \alpha^{-1}$
    - `penalty`: type of regularization: L1, L2 (default), Elastic-Net, or None
    - `solver`: newton-cg, lbfgs (default), liblinear, sag, saga
- Increasing C: less regularization, tries to overfit individual points

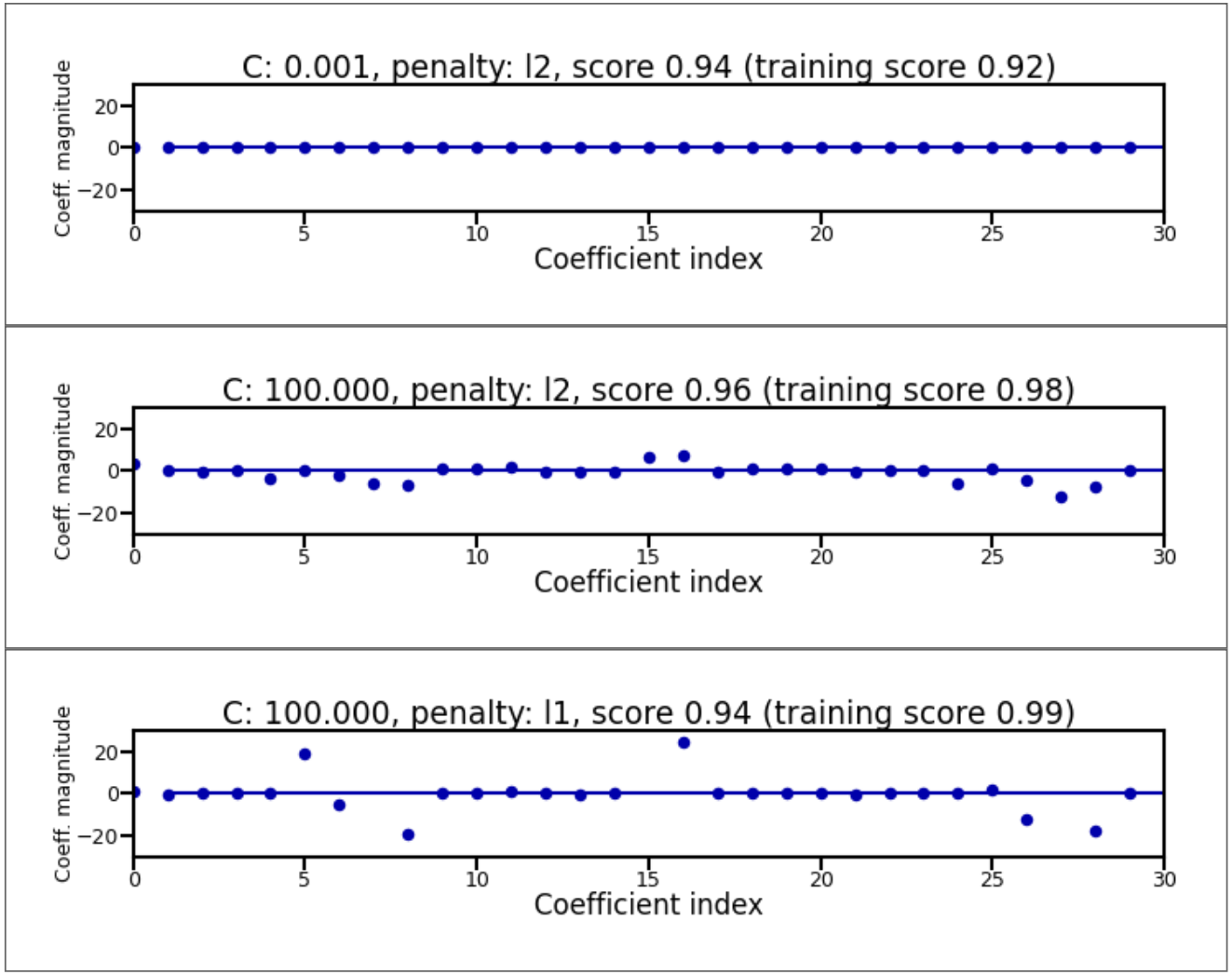
```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(C=1).fit(X_train, y_train)
```



- Analyze behavior on the breast cancer dataset
    - Underfitting if C is too small, some overfitting if C is too large
    - We use cross-validation because the dataset is small



- Again, choose between L1 or L2 regularization (or elastic-net)
- Small C overfits, L1 leads to sparse models



Ridge Classification

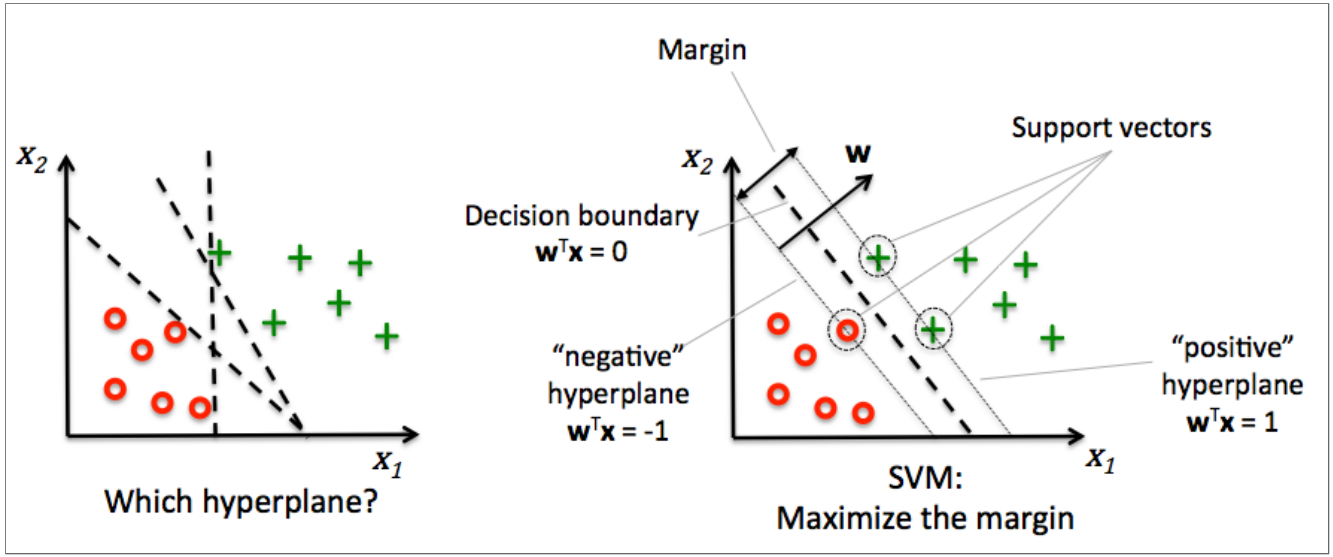
- Instead of log loss, we can also use ridge loss:

$$\mathcal{L}_{Ridge} = \sum_{n=1}^{N}(y_n - (\mathbf{wx_n} + w_0))^2 + \alpha \sum_{i=0}^{p} w_i^2$$

- In this case, target values {negative, positive} are converted to {-1,1}
- Can be solved similarly to Ridge regression:
    - Closed form solution (a.k.a. Cholesky)
    - Gradient descent and variants
        - E.g. Conjugate Gradient (CG) or Stochastic Average Gradient (SAG,SAGA)
    - Use Cholesky for smaller datasets, Gradient descent for larger ones

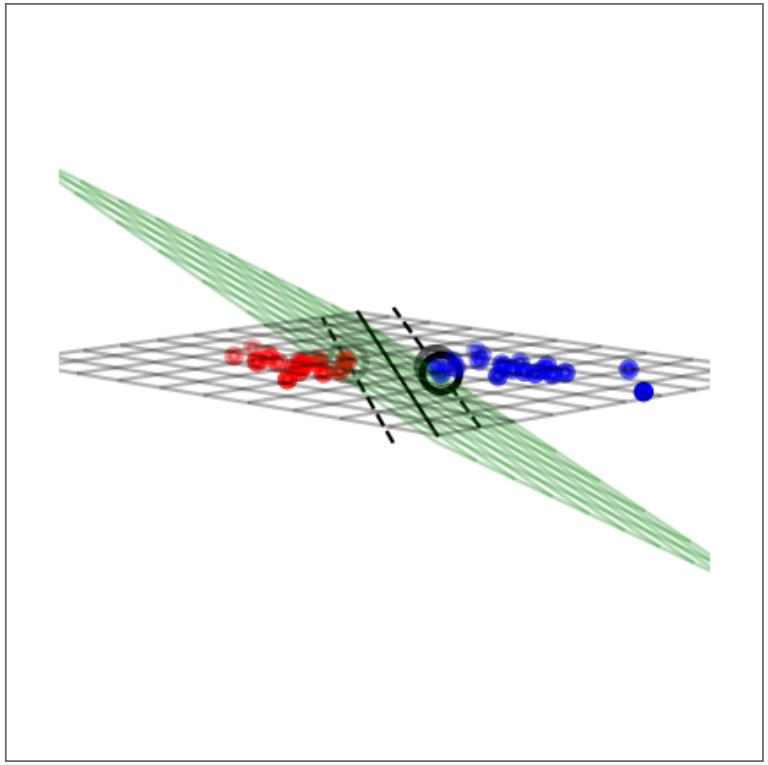
Support vector machines

- Decision boundaries close to training points may generalize badly
    - Very similar (nearby) test point are classified as the other class
- Choose a boundary that is as far away from training points as possible
- The **support vectors** are the training samples closest to the hyperplane
- The **margin** is the distance between the separating hyperplane and the *support vectors*
- Hence, our objective is to *maximize the margin*



Solving SVMs with Lagrange Multipliers

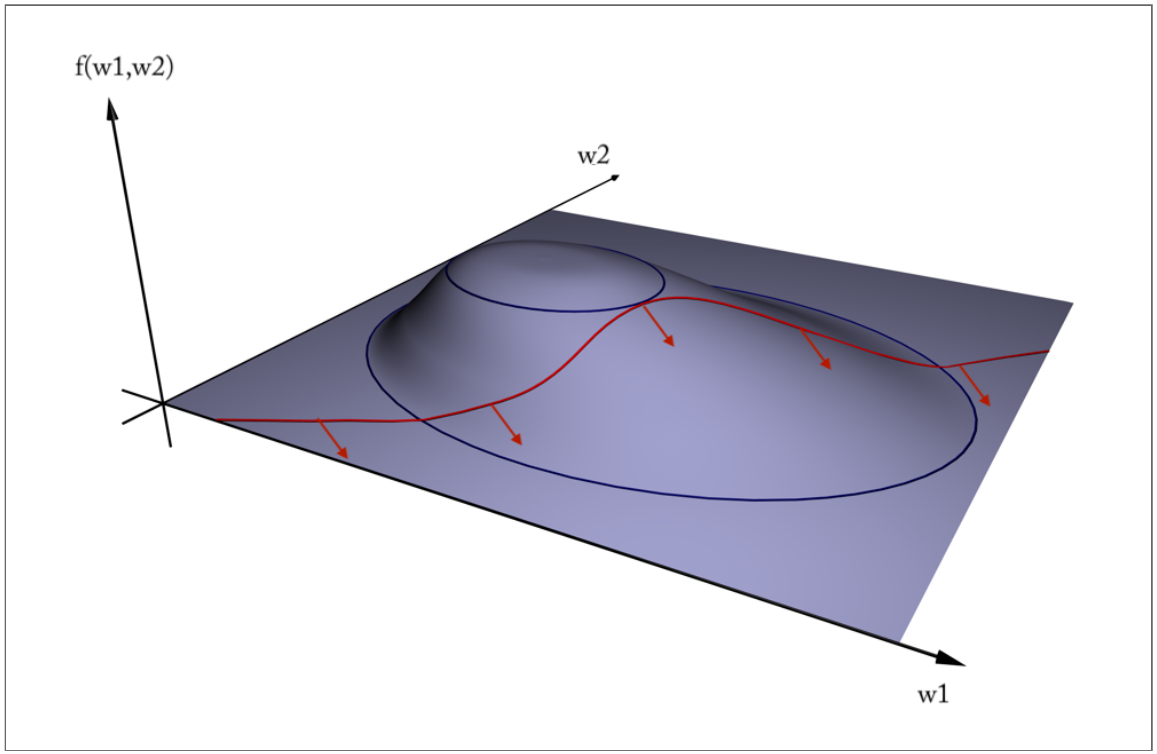
- Imagine a hyperplane (green) $y = \sum_1^p \mathbf{w}_i * \mathbf{x}_i + w_0$ that has slope $\mathbf{w}$, value '+1' for the positive (red) support vectors, and '-1' for the negative (blue) ones
    - Margin between the boundary and support vectors is $\frac{y - w_0}{||\mathbf{w}||}$, with $||\mathbf{w}|| = \sum_i^p w_i^2$
    - We want to find the weights that maximize $\frac{1}{||\mathbf{w}||}$. We can also do that by maximizing $\frac{1}{||\mathbf{w}||^2}$



Geometric interpretation

- We want to maximize $f = \frac{1}{||w||^2}$ (blue contours)
- The hyperplane (red) must be > 1 for all positive examples:

  $g(\mathbf{w}) = \mathbf{w}\mathbf{x_i} + w_0 > 1 \ \ \forall i, y(i) = 1$
- Find the weights $\mathbf{w}$ that satify g but maximize f



Solution

- A quadratic loss function with linear constraints can be solved with *Lagrangian multipliers*
- This works by assigning a weight a_i (called a dual coefficient) to every data point x_i
    - They reflect how much individual points influence the weights $\mathbf{w}$
    - The points with non-zero a_i are the *support vectors*
- Next, solve the following **Primal** objective:
    - $y_i = \pm 1$ is the correct class for example x_i

$$\mathcal{L}_{Primal} = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{n} a_i y_i(\mathbf{w}\mathbf{x_i} + w_0) + \sum_{i=1}^{n} a_i$$

so that

$$\mathbf{w} = \sum_{i=1}^{n} a_i y_i \mathbf{x_i}$$

$$a_i \geq 0 \quad \text{and} \quad \sum_{i=1}^{l} a_i y_i = 0$$

- It has a **Dual** formulation as well (See 'Elements of Statistical Learning' for the derivation):

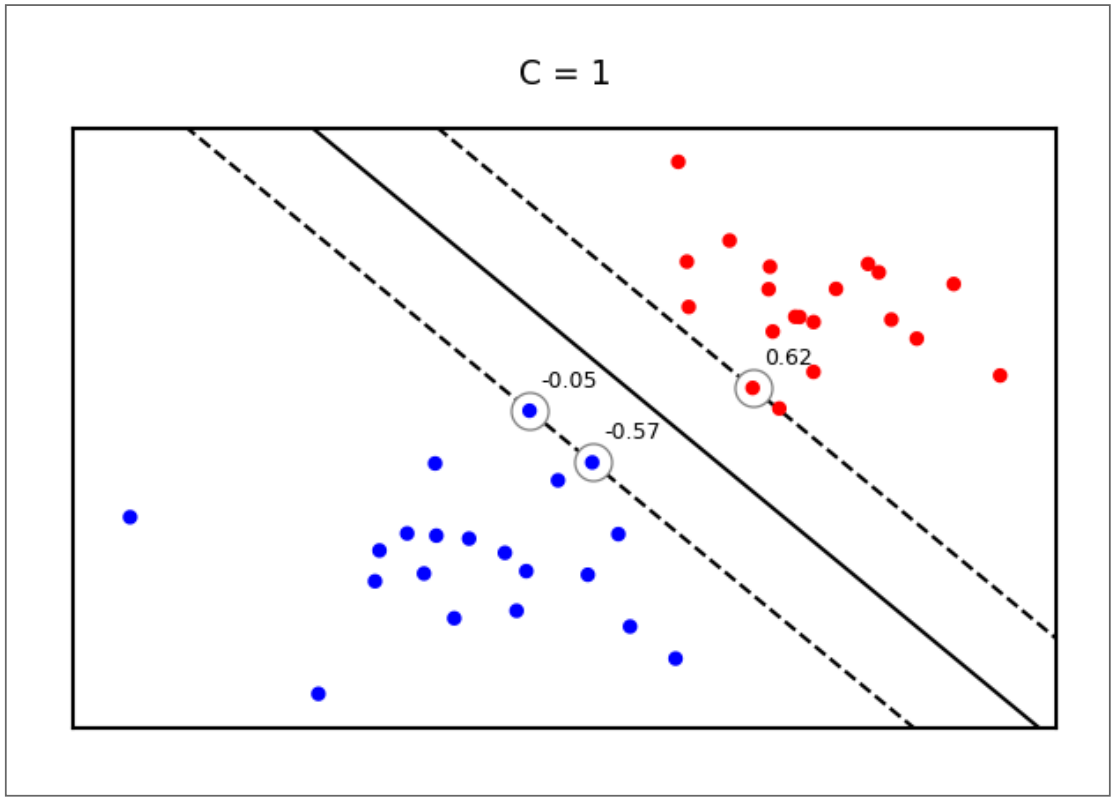
$$\mathcal{L}_{Dual} = \sum_{i=1}^{l} a_i - \frac{1}{2} \sum_{i,j=1}^{l} a_i a_j y_i y_j (\mathbf{x_i x_j})$$

so that

$$a_i \geq 0 \quad \text{and} \quad \sum_{i=1}^{l} a_i y_i = 0$$

- Computes the dual coefficients directly. A number l of these are non-zero (sparseness).
    - Dot product $\mathbf{x_i x_j}$ can be interpreted as the closeness between points $\mathbf{x_i}$ and $\mathbf{x_j}$
    - $\mathcal{L}_{Dual}$ increases if nearby support vectors $\mathbf{x_i}$ with high weights a_i have different class y_i
    - $\mathcal{L}_{Dual}$ also increases with the number of support vectors l and their weights a_i
- Can be solved with quadratic programming, e.g. Sequential Minimal Optimization (SMO)

Example result. The circled samples are support vectors, together with their coefficients.



Making predictions

- a_i will be *0* if the training point lies on the right side of the decision boundary and outside the margin
- The training samples for which a_i is not 0 are the *support vectors*
- Hence, the SVM model is completely defined by the support vectors and their dual coefficients (weights)
- Knowing the dual coefficients a_i, we can find the weights w for the maximal margin separating hyperplane:

$$\mathbf{w} = \sum_{i=1}^{l} a_i y_i \mathbf{x_i}$$

- Hence, we can classify a new sample $\mathbf{u}$ by looking at the sign of $\mathbf{wu} + w_0$

SVMs and kNN

- Remember, we will classify a new point $\mathbf{u}$ by looking at the sign of:

$$f(x) = \mathbf{w}\mathbf{u} + w_0 = \sum_{i=1}^{l} a_i y_i \mathbf{x_i}\mathbf{u} + w_0$$

- *Weighted k-nearest neighbor* is a generalization of the k-nearest neighbor classifier. It classifies points by evaluating:

$$f(x) = \sum_{i=1}^{k} a_i y_i dist(x_i, u)^{-1}$$

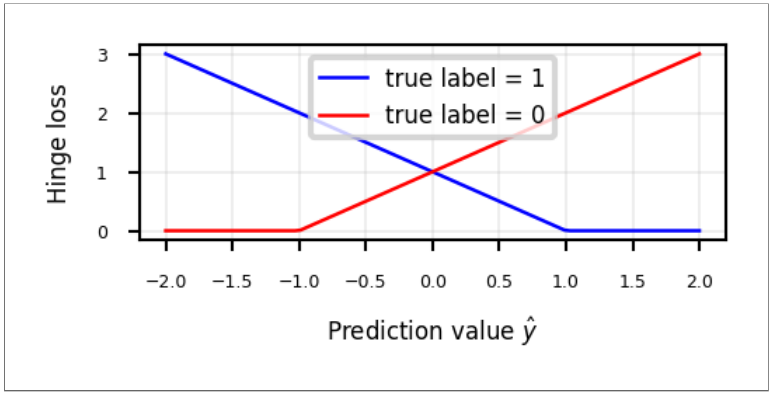
- Hence: SVM's predict much the same way as k-NN, only:
    - They only consider the truly important points (the support vectors): *much* faster
        - The number of neighbors is the number of support vectors
    - The distance function is an *inner product of the inputs* (or another kernel)
- Given $\mathbf{u}$, we predict by looking at the classes of the support vectors, weighted by their distance.

Regularized (soft margin) SVMs

- If the data is not linearly separable, (hard) margin maximization becomes meaningless
- Relax the contraint by allowing an error ξ_i: $y_i(\mathbf{wx_i} + w_0) \geq 1 - \xi_i$
- Or (since $\xi_i \geq 0$): $\xi_i = max(0, 1 - y_i \cdot (\mathbf{wx_i} + w_0))$
- The sum over all points is called *hinge loss*: $\sum_i^n \xi_i$
- Attenuating the error component with a hyperparameter C, we get the objective

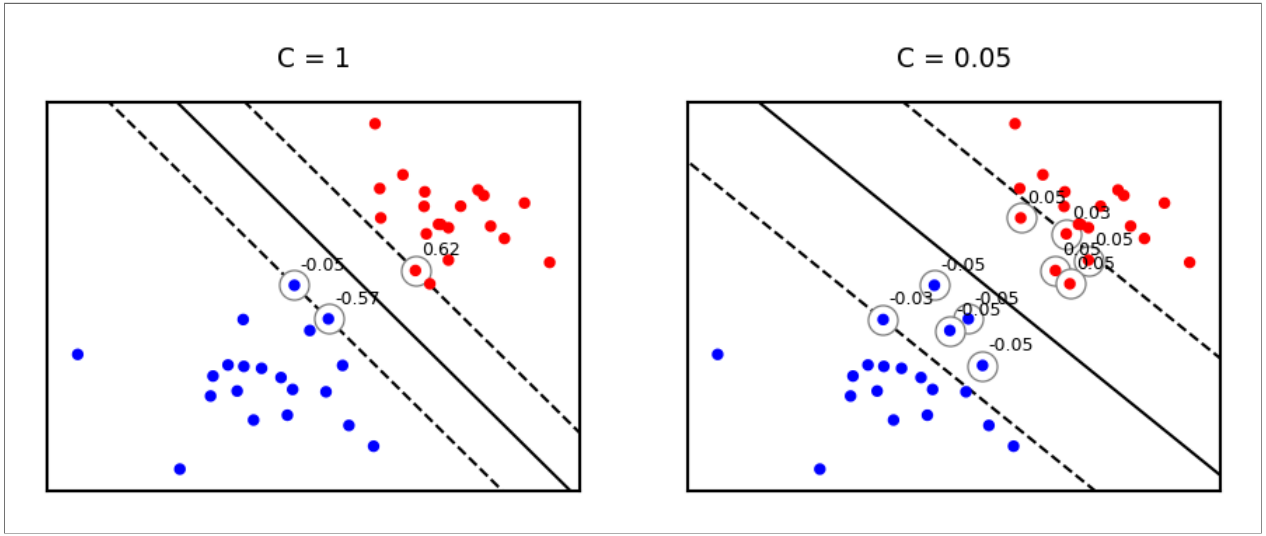
$$\mathcal{L}(\mathbf{w}) = ||\mathbf{w}||^2 + C \sum_i^n \xi_i$$

- Can still be solved with quadratic programming

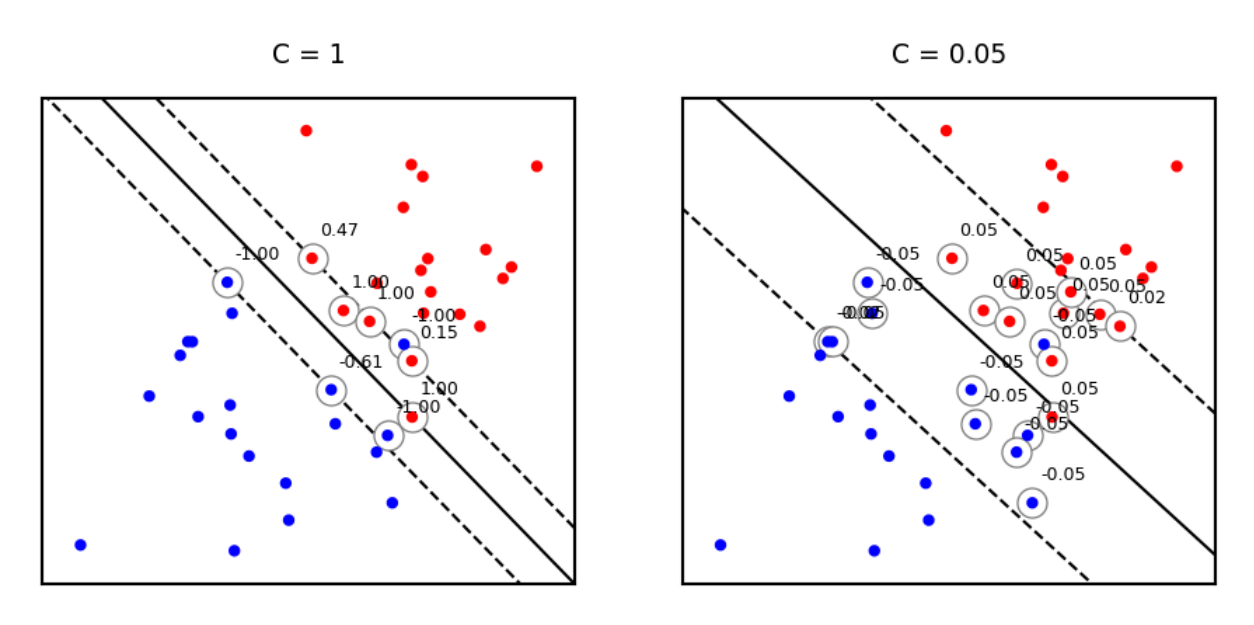


Effect of regularization on margin and support vectors

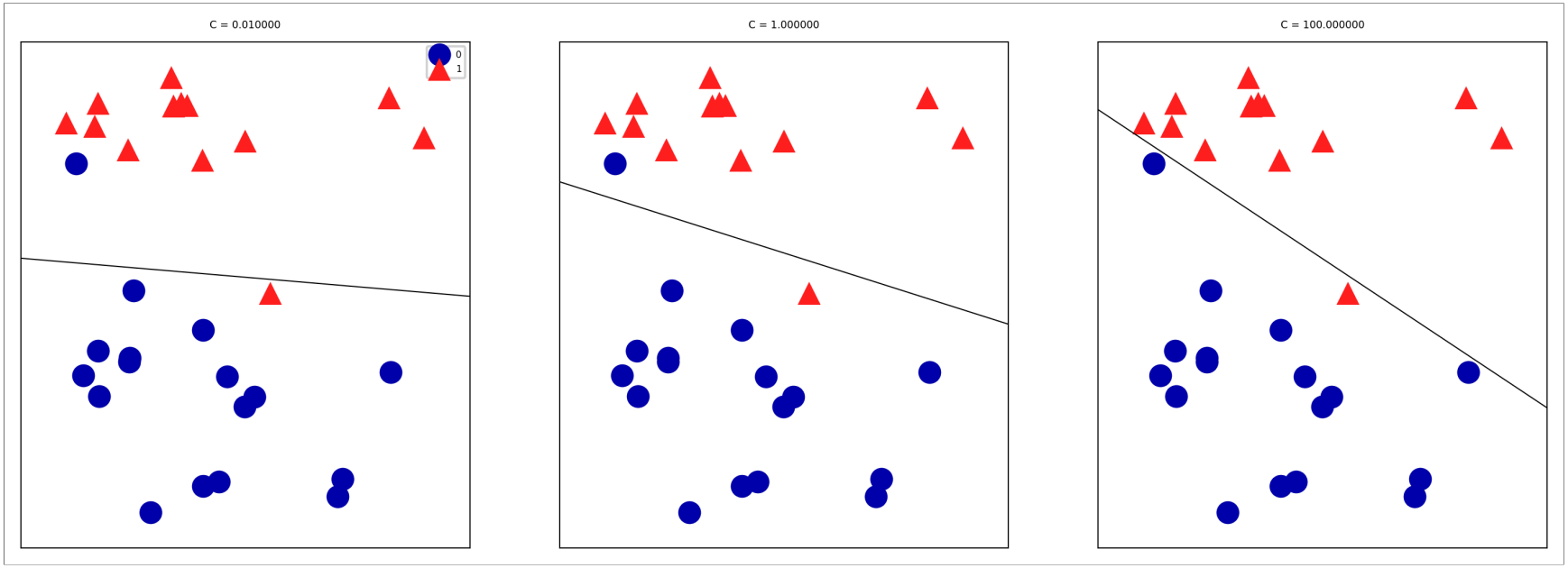
- SVM's Hinge loss acts like L1 regularization, yields sparse models
- C is the *inverse* regularization strength (inverse of α in Lasso)
    - Larger C: fewer support vectors, smaller margin, more overfitting
    - Smaller C: more support vectors, wider margin, less overfitting
- Needs to be tuned carefully to the data



Same for non-linearly separable data



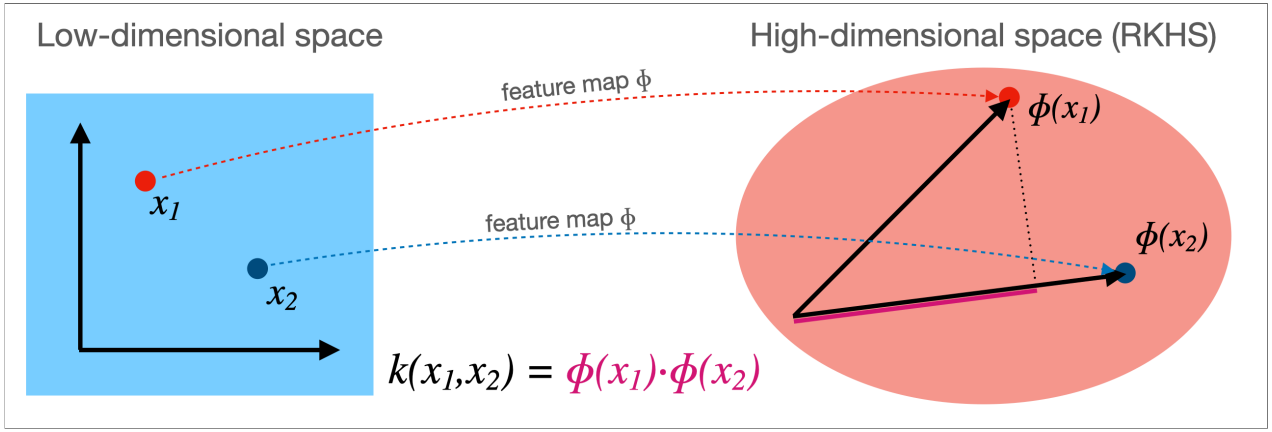
Large C values can lead to overfitting (e.g. fitting noise), small values can lead to underfitting



Kernelization

- Sometimes we can separate the data better by first transforming it to a higher dimensional space $\Phi(x)$
    - This transformation Φ is called a feature map (but can be expensive)
- For certain Φ, we know the function k that computes the dot product in $\Phi(x)$: $k(\mathbf{x_i}, \mathbf{x_j}) = \Phi(\mathbf{x_i}) \cdot \Phi(\mathbf{x_j})$
- This *kernel* function $k(\mathbf{x_i}, \mathbf{x_j})$ computes the dot product without having to construct (reproduce) $\Phi(x)$
- Kernel trick: if your loss function has a dot product, you can simply replace it with a kernel!
- For SVMs (in dual form), replacing $(\mathbf{x_i}\mathbf{x_j}) \rightarrow k(\mathbf{x_i}, \mathbf{x_j})$ yields a *kernelized SVM*:

$$\mathcal{L}_{Dual}(a_i, k) = \sum_{i=1}^{l} a_i - \frac{1}{2} \sum_{i,j=1}^{l} a_i a_j y_i y_j k(\mathbf{x_i}, \mathbf{x_j})$$



Polynomial kernel

- The **polynomial kernel** (for degree $d \in \mathbb{N}$) reproduces the polynomial feature map

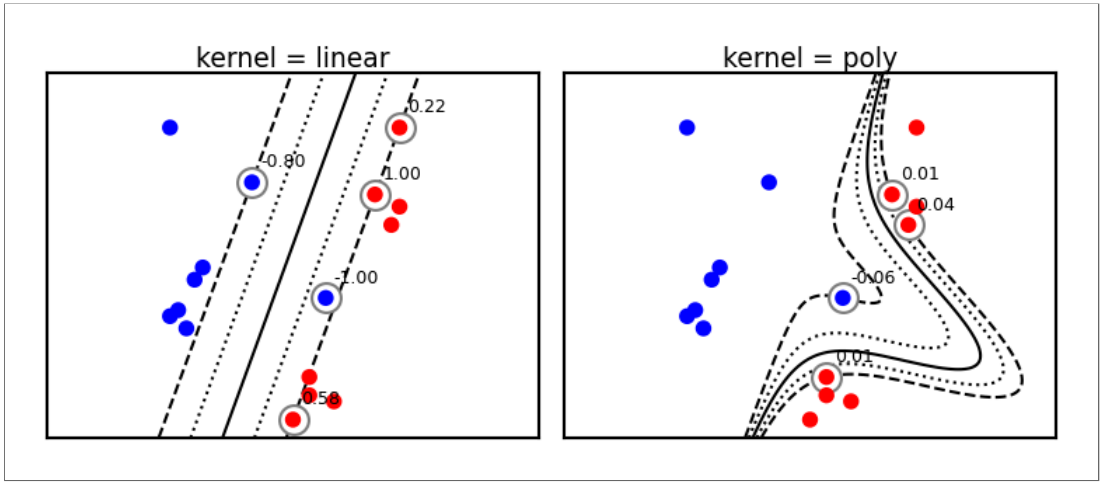
$$[1, x_1, \ldots, x_p] \overset{\phi}{\rightarrow} [1, x_1, \ldots, x_p, x_1^2, \ldots, x_p^2, \ldots, x_p^d, x_1 x_2, \ldots, x_{p-1} x_p]$$

- It can be easily computed from the original dot product:

$$k_{poly}(\mathbf{x_1}, \mathbf{x_2}) = (\gamma(\mathbf{x_1} \cdot \mathbf{x_2}) + c_0)^d$$

- It has two more hyperparameters, but you can usually leave them at default
  - γ is a scaling hyperparameter (default $\frac{1}{p}$)
  - c_0 is a hyperparameter (default 1) to trade off influence of higher-order terms

- By simply replacing the dot product with a kernel we can learn non-linear SVMs!
- It is technically still linear in $\Phi(x)$, but in our original space the boundary becomes a polynomial curve
- Prediction still happens as before, but the influence of each support vector drops of polynomially (with degree d)



Radial Basis Function (RBF) kernel

- The **RBF or Gaussian kernel** (of width $\gamma > 0$) is related to the Taylor series expansion of e^x

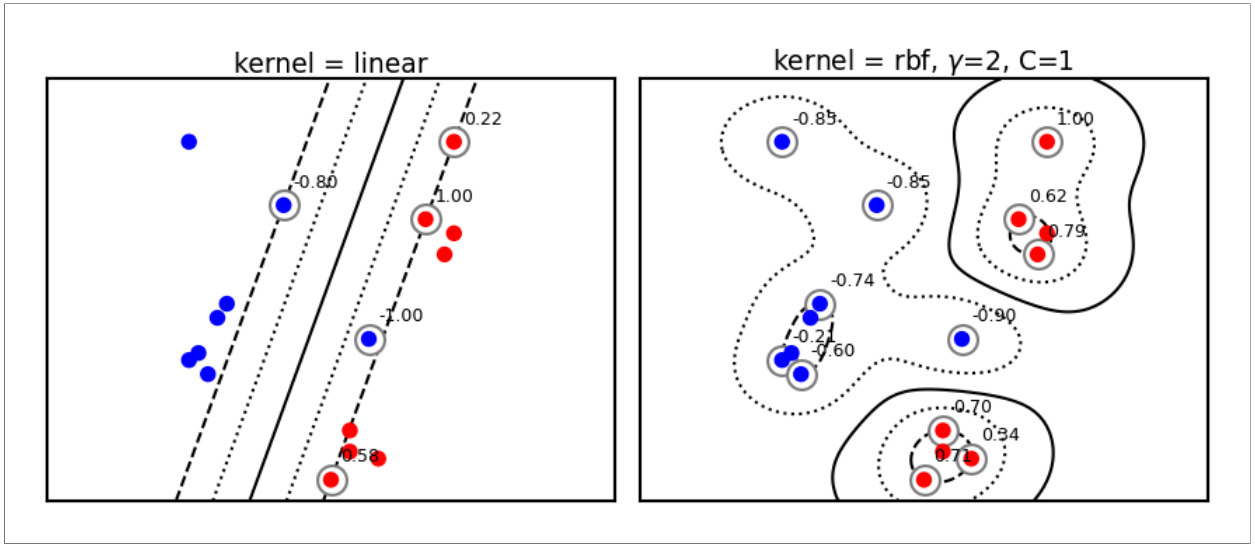
$$\Phi(x) = e^{-x^2/2\gamma^2}\left[1, \sqrt{\frac{1}{1!\gamma^2}}x, \sqrt{\frac{1}{2!\gamma^4}}x^2, \sqrt{\frac{1}{3!\gamma^6}}x^3, \dots\right]^T$$

- It is a function of how closely together two data points are:

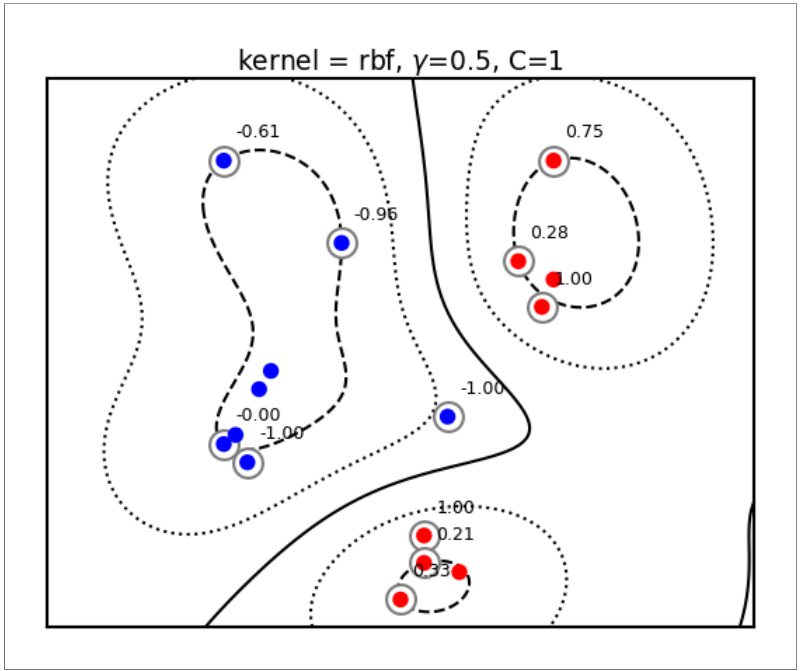
$$k_{RBF}(\mathbf{x_1}, \mathbf{x_2}) = exp(-\gamma||\mathbf{x_1} - \mathbf{x_2}||^2)$$

- The influence of a point $\mathbf{x_2}$ on point $\mathbf{x_1}$ drops off exponentially with its distance to $\mathbf{x_1}$

- The influence of each support vector now drops of *exponentially*
    - Hence, predictions are only affected by very nearby support vectors
- RBF kernels are therefore called *local* kernels



- The kernel width (γ) defines how sharply the local influence decays
    - Acts as a regularizer: low γ causes *underfitting* and high γ causes *overfitting*
- SVM's C parameter (inverse regularizer) is still at play and thus interacts with γ



Kernelization sidenotes (optional)

- You can invent many more feature maps and corresponding kernels (eg. for text, graphs,...)
    - However, learning deep learning embeddings from lots of data often works better
- You can also kernelize Ridge regression, Logistic regression, Perceptrons, Support Vector Regression,...
    - The *Representer theorem* will give you the corresponding loss function
- For more detail see the Kernelization lecture under extra materials.

SVMs IN SCIKIT-LEARN

- `svm.LinearSVC` : faster for large datasets
  - Allows choosing between the primal or dual. Primal recommended when $n >> p$
  - Returns `coef_` ($\mathbf{w}$) and `intercept_` (w_0)
- `svm.SVC` allows different kernels to be used
  - Also returns `support_vectors_` (the support vectors) and the `dual_coef_` a_i
  - Scales at least quadratically with the number of samples n
- `svm.LinearSVR` and `svm.SVR` are variants for regression

```
clf = svm.SVC(kernel='linear') # or 'RBF' or 'Poly'
clf.fit(X, Y)
print("Support vectors:", clf.support_vectors_[:])
print("Coefficients:", clf.dual_coef_[:])
```

```
Support vectors:
[[-1.021  0.241]
 [-0.467 -0.531]
 [ 0.951  0.58 ]]
Coefficients:
[[-0.048 -0.569  0.617]]
```

Solving SVMs with Gradient Descent

- SVMs can, alternatively, be solved using gradient decent
    - Good for large datasets, but does not yield support vectors or kernelization
- Hinge loss is not differentiable but convex, and has a subgradient:

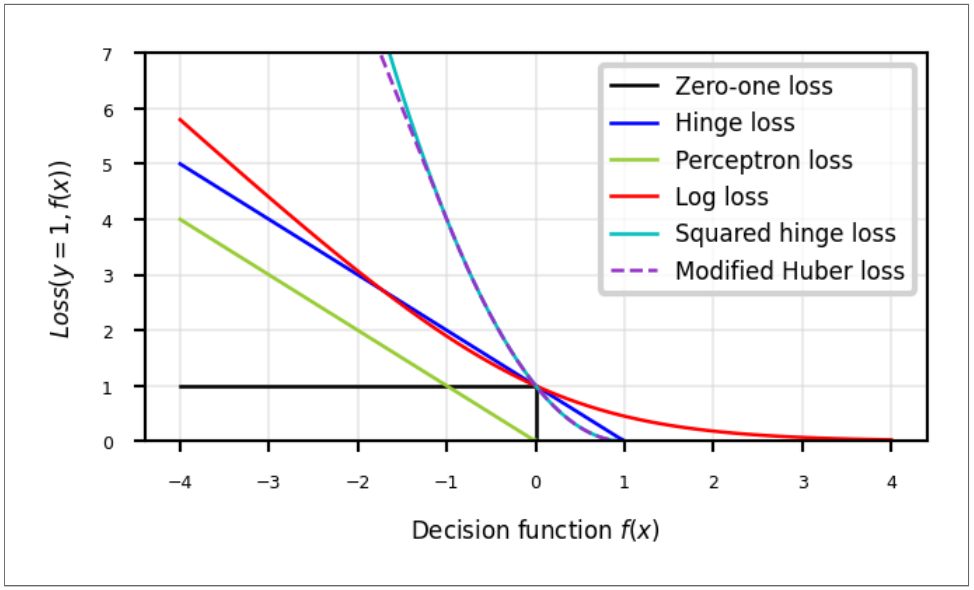
$$\mathcal{L}_{\mathcal{H}inge}(\mathbf{w}) = max(0, 1 - y_i(\mathbf{wx_i} + w_0))$$

$$\frac{\partial \mathcal{L}_{\mathcal{H}inge}}{\partial w_i} = \begin{cases} -y_i x_i & y_i(\mathbf{wx_i} + w_0) < 1 \\ 0 & \text{otherwise} \end{cases}$$

- Can be solved with (stochastic) gradient descent

Generalized SVMs

- There are many smoothed versions of hinge loss:
    - Squared hinge loss:
        - Also known as *Ridge classification*
        - Least Squares SVM: allows kernelization (using a linear equation solver)
    - Modified Huber loss: squared hinge, but linear after -1. Robust against outliers
    - Log loss: equivalent to logistic regression
- In sklearn, `SGDClassifier` can be used with any of these. Good for large datasets.

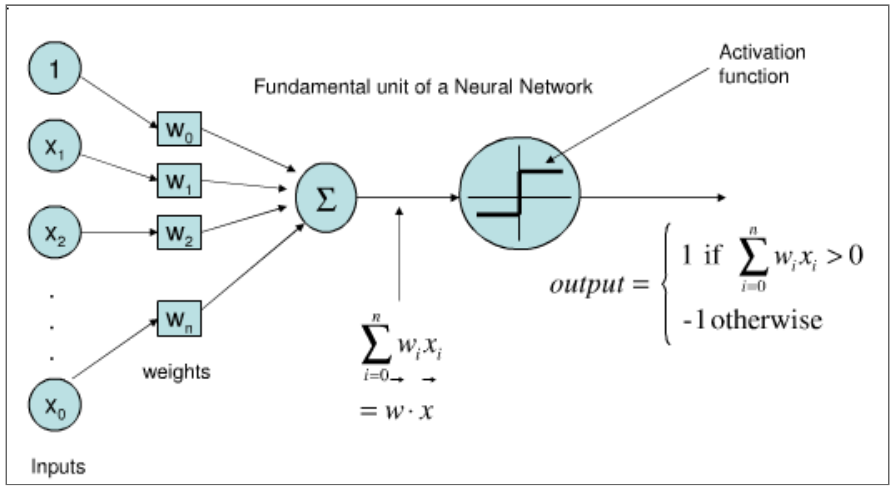


Perceptron

- Represents a single neuron (node) with inputs x_i, a bias w_0, and output y
- Each connection has a (synaptic) weight w_i. The node outputs $\hat{y} = \sum_i^n x_i w_i + w_0$
- The *activation function* predicts 1 if $\mathbf{xw} + w_0 > 0$, -1 otherwise
- Weights can be learned with (stochastic) gradient descent and Hinge(0) loss
    - Updated *only* on misclassification, corrects output by ± 1

$$\mathcal{L}_{Perceptron} = max(0, -y_i(\mathbf{wx_i} + w_0))$$

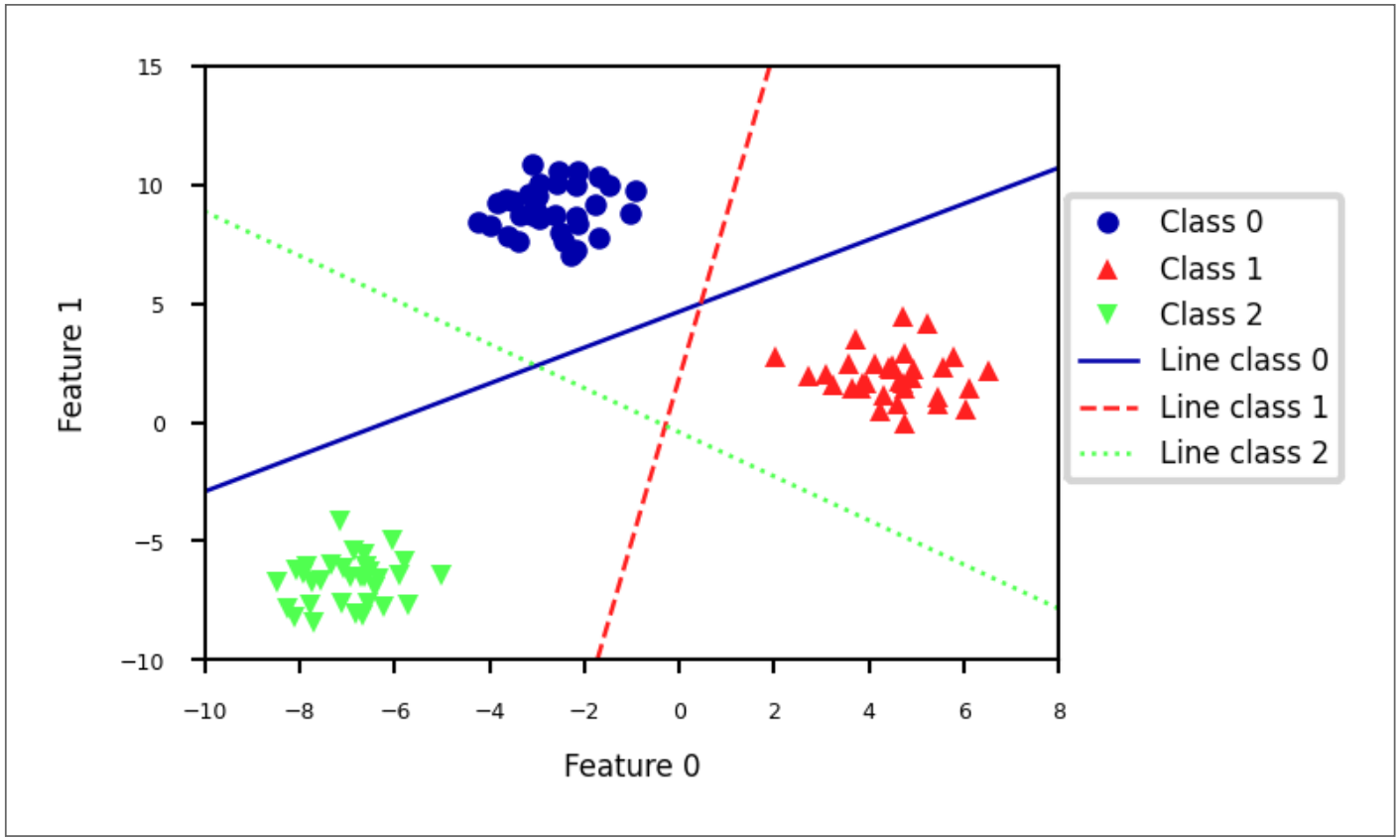
$$\frac{\partial \mathcal{L}_{Perceptron}}{\partial w_i} = \begin{cases} -y_i x_i & y_i(\mathbf{wx_i} + w_0) < 0 \\ 0 & \text{otherwise} \end{cases}$$



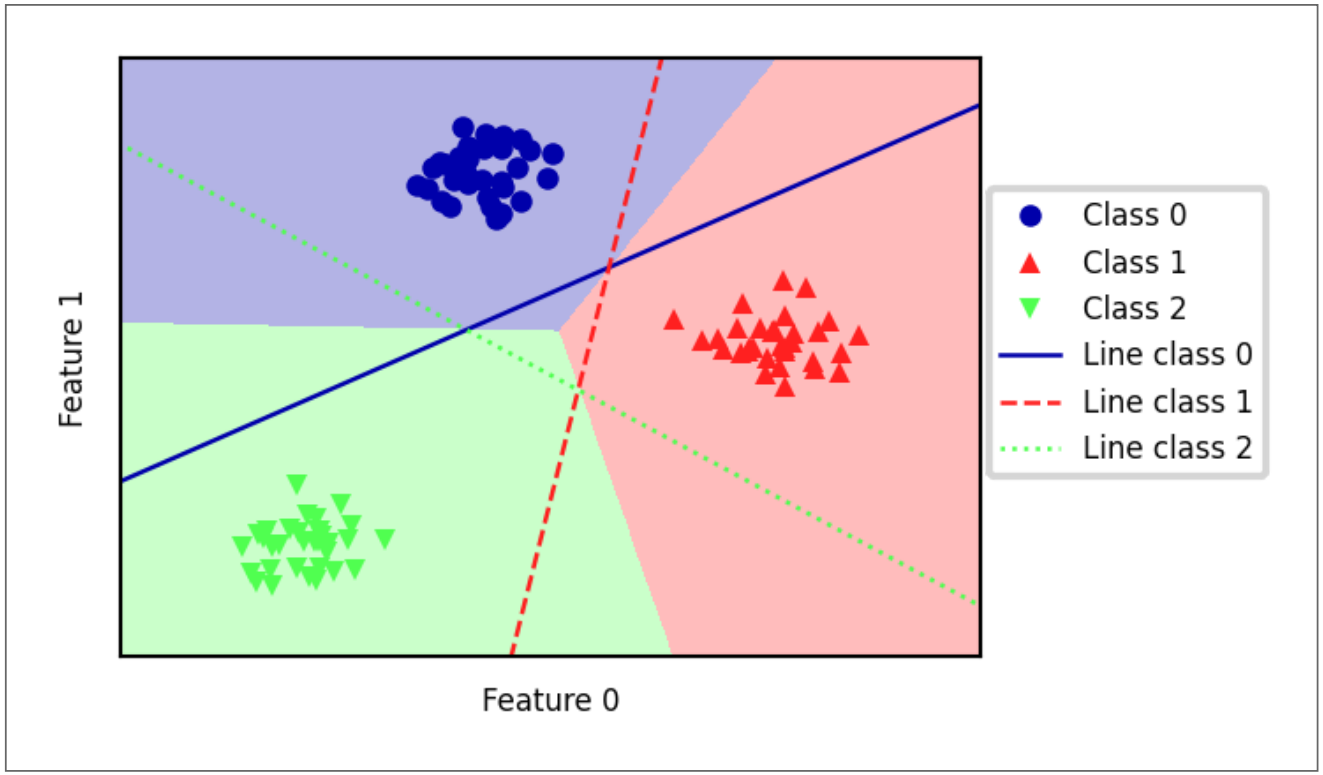
Linear Models for multiclass classification

one-vs-rest (aka one-vs-all)

- Learn a binary model for each class vs. all other classes
- Create as many binary models as there are classes



- Every binary classifiers makes a prediction, the one with the highest score (>0) wins



one-vs-one

- An alternative is to learn a binary model for every *combination* of two classes
    - For C classes, this results in $\frac{C(C-1)}{2}$ binary models
    - Each point is classified according to a majority vote amongst all models
    - Can also be a 'soft vote': sum up the probabilities (or decision values) for all models. The class with the highest sum wins.
- Requires more models than one-vs-rest, but training each one is faster
    - Only the examples of 2 classes are included in the training data
- Recommended for algorithms than learn well on small datasets
    - Especially SVMs and Gaussian Processes

Linear models overview

Name	Representation	Loss function	Optimization	Regularization
Least squares	Linear function (R)	SSE	CFS or SGD	None
Ridge	Linear function (R)	SSE + L2	CFS or SGD	L2 strength (α)
Lasso	Linear function (R)	SSE + L1	Coordinate descent	L1 strength (α)
Elastic-Net	Linear function (R)	SSE + L1 + L2	Coordinate descent	α, L1 ratio (ρ)
SGDRegressor	Linear function (R)	SSE, Huber, ϵ-ins,... + L1/L2	SGD	L1/L2, α
Logistic regression	Linear function (C)	Log + L1/L2	SGD, coordinate descent,...	L1/L2, α
Ridge classification	Linear function (C)	SSE + L2	CFS or SGD	L2 strength (α)
Linear SVM	Support Vectors	Hinge(1)	Quadratic programming or SGD	Cost (C)

Name	Representation	Loss function	Optimization	Regularization
Least Squares SVM	Support Vectors	Squared Hinge	Linear equations or SGD	Cost (C)
Perceptron	Linear function (C)	Hinge(0)	SGD	None
SGDClassifier	Linear function (C)	Log, (Sq.) Hinge, Mod. Huber,...	SGD	L1/L2, α

- SSE: Sum of Squared Errors
- CFS: Closed-form solution
- SGD: (Stochastic) Gradient Descent and variants
- (R)egression, (C)lassification

Summary

- Linear models
    - Good for very large datasets (scalable)
    - Good for very high-dimensional data (not for low-dimensional data)
- Can be used to fit non-linear or low-dim patterns as well (see later)
    - Preprocessing: e.g. Polynomial or Poisson transformations
    - Generalized linear models (kernelization)
- Regularization is important. Tune the regularization strength (α)
    - Ridge (L2): Good fit, sometimes sensitive to outliers
    - Lasso (L1): Sparse models: fewer features, more interpretable, faster
    - Elastic-Net: Trade-off between both, e.g. for correlated features
- Most can be solved by different optimizers (solvers)
    - Closed form solutions or quadratic/linear solvers for smaller datasets
    - Gradient descent variants (SGD,CD,SAG,CG,...) for larger ones
- Multi-class classification can be done using a one-vs-all approach

Lecture 4: Model Selection

Can I trust you?

Joaquin Vanschoren

Evaluation

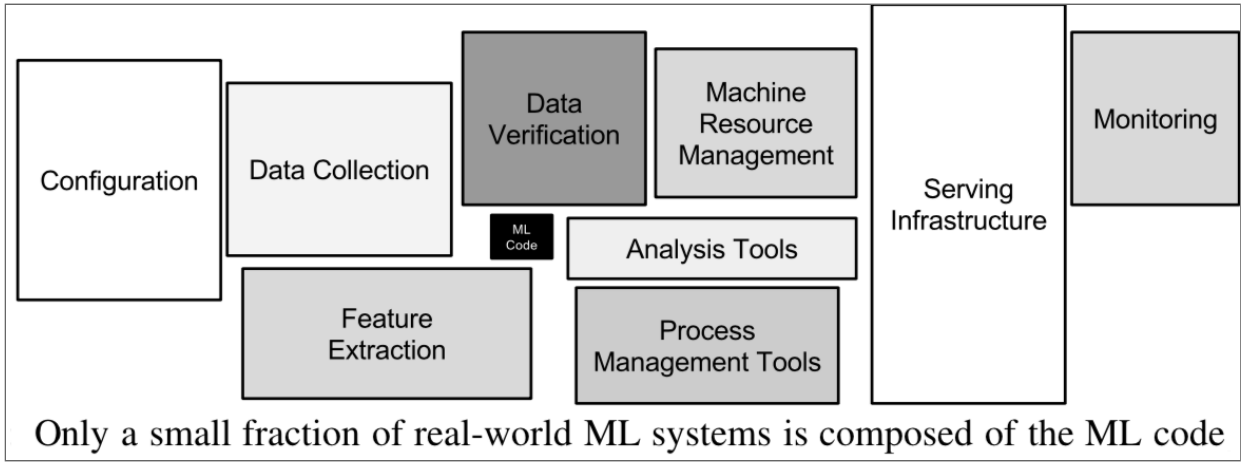
- To know whether we can *trust* our method or system, we need to evaluate it.
- Model selection: choose between different models in a data-driven way.
    - If you cannot measure it, you cannot improve it.
- Convince others that your work is meaningful
    - Peers, leadership, clients, yourself(!)
- When possible, try to *interpret* what your model has learned
    - The signal your model found may just be an artifact of your biased data
    - See 'Why Should I Trust You?' by Marco Ribeiro et al.

(a) Husky classified as wolf

(b) Explanation

Designing Machine Learning systems

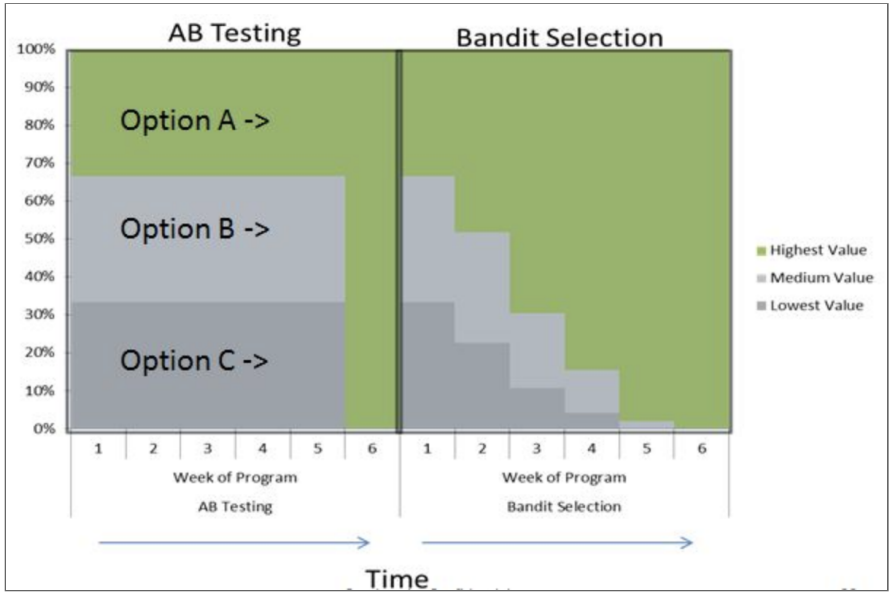
- Just running your favourite algorithm is usually not a great way to start
- Consider the problem: How to measure success? Are there costs involved?
    - Do you want to understand phenomena or do black box modelling?
- Analyze your model's mistakes. Don't just finetune endlessly.
    - Build early prototypes. Should you collect more, or additional data?
    - Should the task be reformulated?
- Overly complex machine learning systems are hard to maintain
    - See 'Machine Learning: The High Interest Credit Card of Technical Debt'



Only a small fraction of real-world ML systems is composed of the ML code

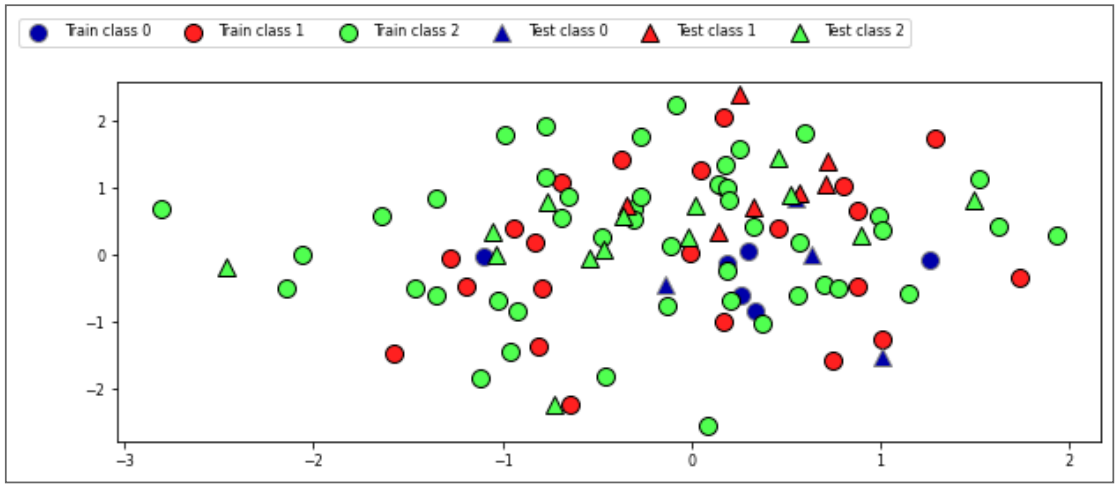
Real world evaluations

- Evaluate predictions, but also how outcomes improve *because of them*
- Beware of feedback loops: predictions can influence future input data
    - Medical recommendations, spam filtering, trading algorithms,...
- Evaluate algorithms *in the wild*.
    - A/B testing: split users in groups, test different models in parallel
    - Bandit testing: gradually direct more users to the winning system



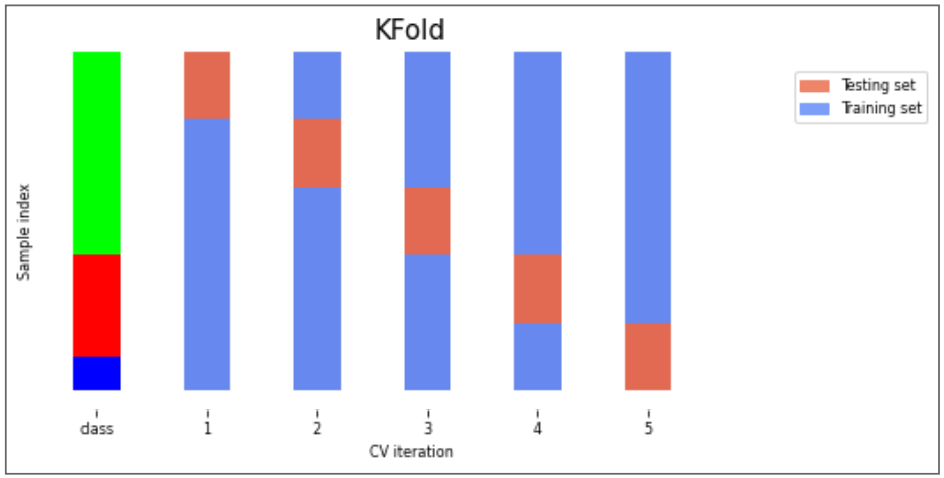
Performance estimation techniques

- Always evaluate models *as if they are predicting future data*
- We do not have access to future data, so we pretend that some data is hidden
- Simplest way: the *holdout* (simple train-test split)
    - *Randomly* split data (and corresponding labels) into training and test set (e.g. 75%-25%)
    - Train (fit) a model on the training data, score on the test data



K-fold Cross-validation

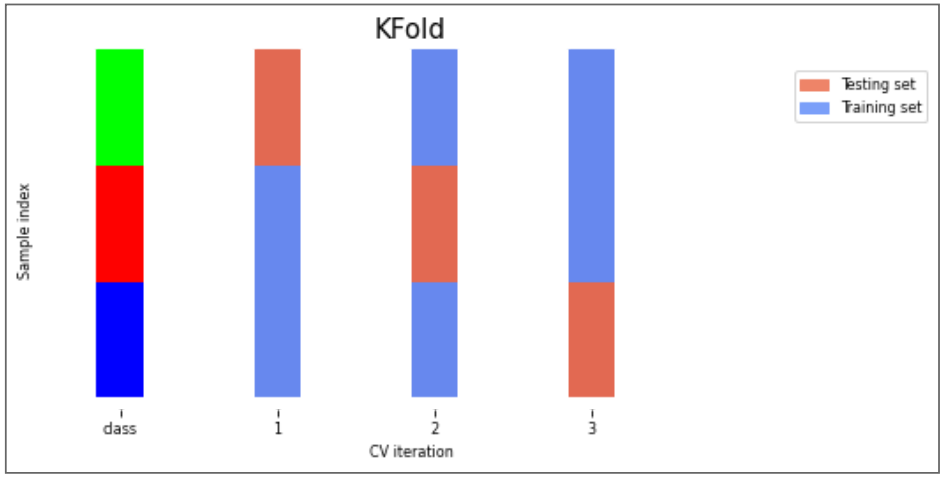
- Each random split can yield very different models (and scores)
    - e.g. all easy (of hard) examples could end up in the test set
- Split data into *k* equal-sized parts, called *folds*
    - Create *k* splits, each time using a different fold as the test set
- Compute *k* evaluation scores, aggregate afterwards (e.g. take the mean)
- Examine the score variance to see how *sensitive* (unstable) models are
- Large *k* gives better estimates (more training data), but is expensive



Can you explain this result?

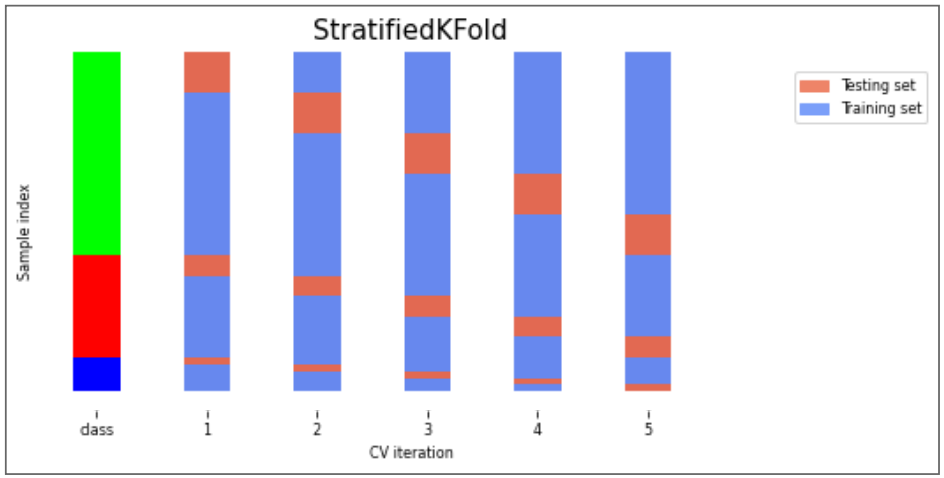
```
kfold = KFold(n_splits=3)
cross_val_score(logistic_regression, iris.data, iris.target, cv=kfold)
```

```
Cross-validation scores KFold(n_splits=3):
[0. 0. 0.]
```



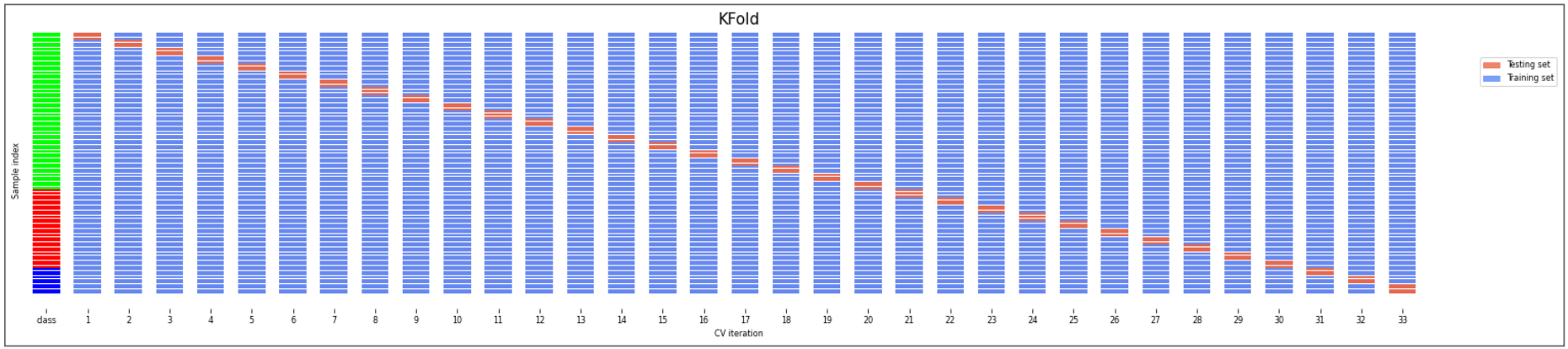
Stratified K-Fold cross-validation

- If the data is unbalanced, some classes have only few samples
- Likely that some classes are not present in the test set
- Stratification: *proportions* between classes are conserved in each fold
    - Order examples per class
    - Separate the samples of each class in *k* sets (strata)
    - Combine corresponding strata into folds



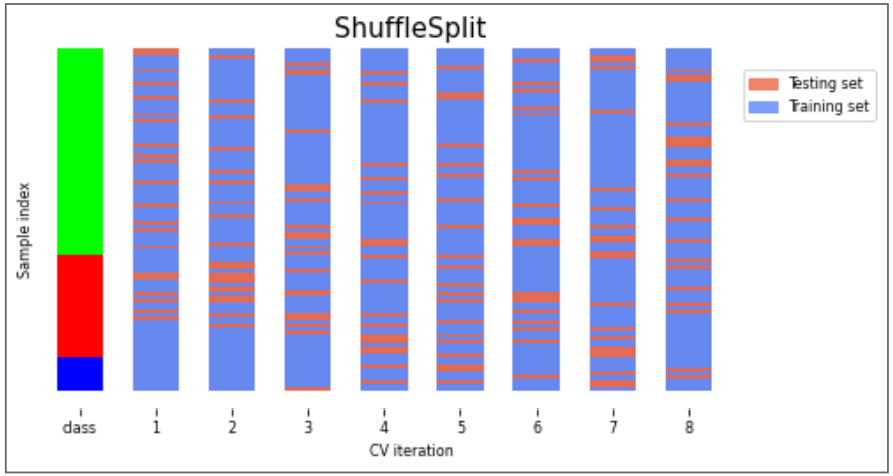
Leave-One-Out cross-validation

- *k* fold cross-validation with *k* equal to the number of samples
- Completely unbiased (in terms of data splits), but computationally expensive
- Actually generalizes *less* well towards unseen data
    - The training sets are correlated (overlap heavily)
    - Overfits on the data used for (the entire) evaluation
    - A different sample of the data can yield different results
- Recommended only for small datasets



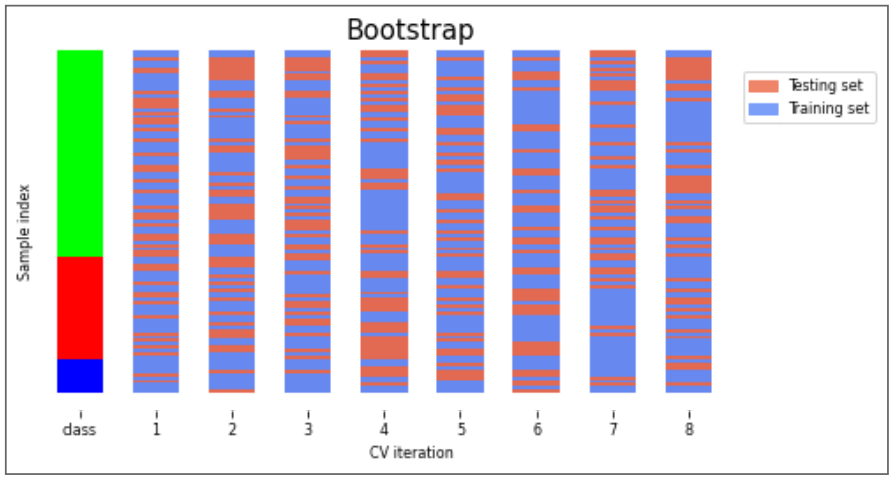
Shuffle-Split cross-validation

- Shuffles the data, samples (`train_size`) points randomly as the training set
- Can also use a smaller (`test_size`), handy with very large datasets
- Never use if the data is ordered (e.g. time series)



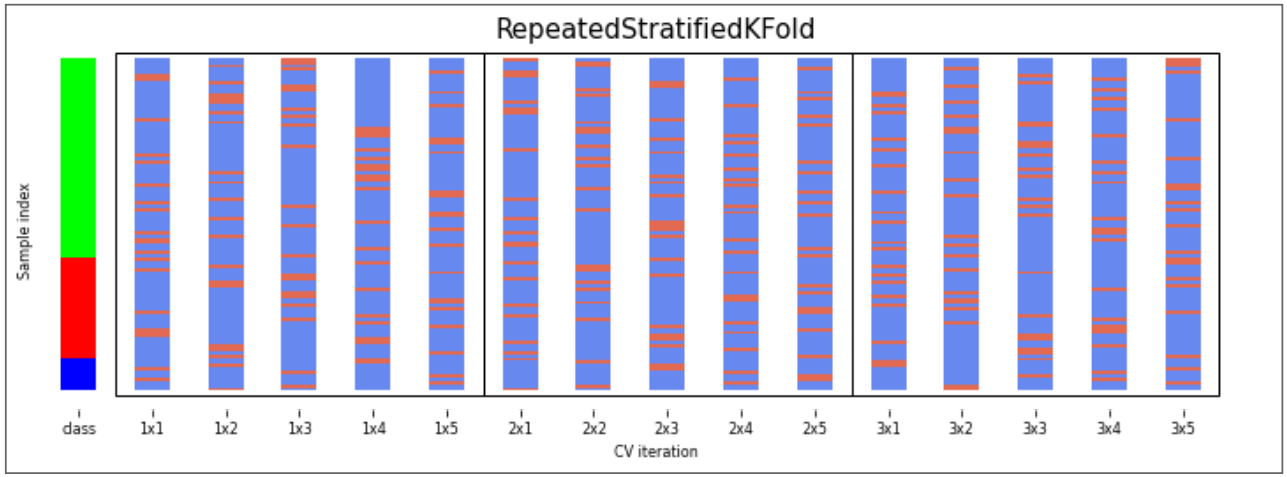
The Bootstrap

- Sample *n* (dataset size) data points, with replacement, as training set (the bootstrap)
    - On average, bootstraps include 66% of all data points (some are duplicates)
- Use the unsampled (out-of-bootstrap) samples as the test set
- Repeat k times to obtain k scores
- Similar to Shuffle-Split with `train_size=0.66`, `test_size=0.34` but without duplicates



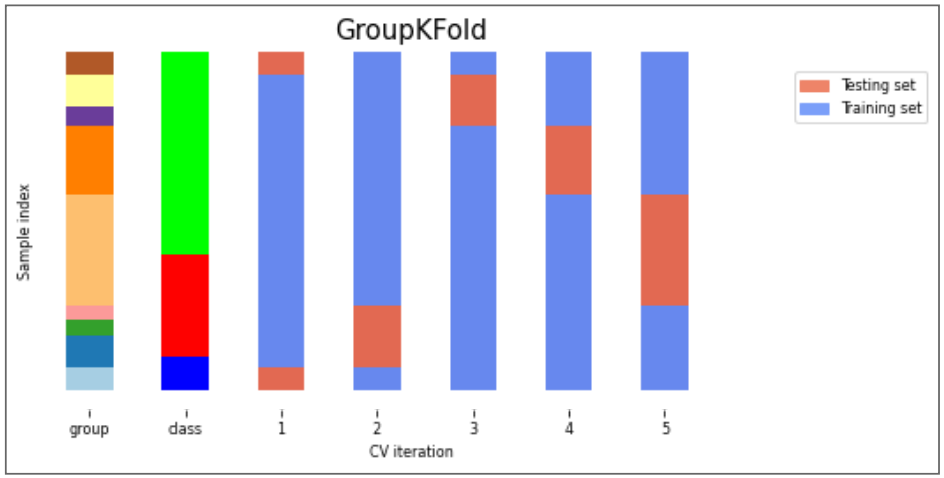
Repeated cross-validation

- Cross-validation is still biased in that the initial split can be made in many ways
- Repeated, or n-times-k-fold cross-validation:
    - Shuffle data randomly, do k-fold cross-validation
    - Repeat n times, yields n times k scores
- Unbiased, very robust, but n times more expensive



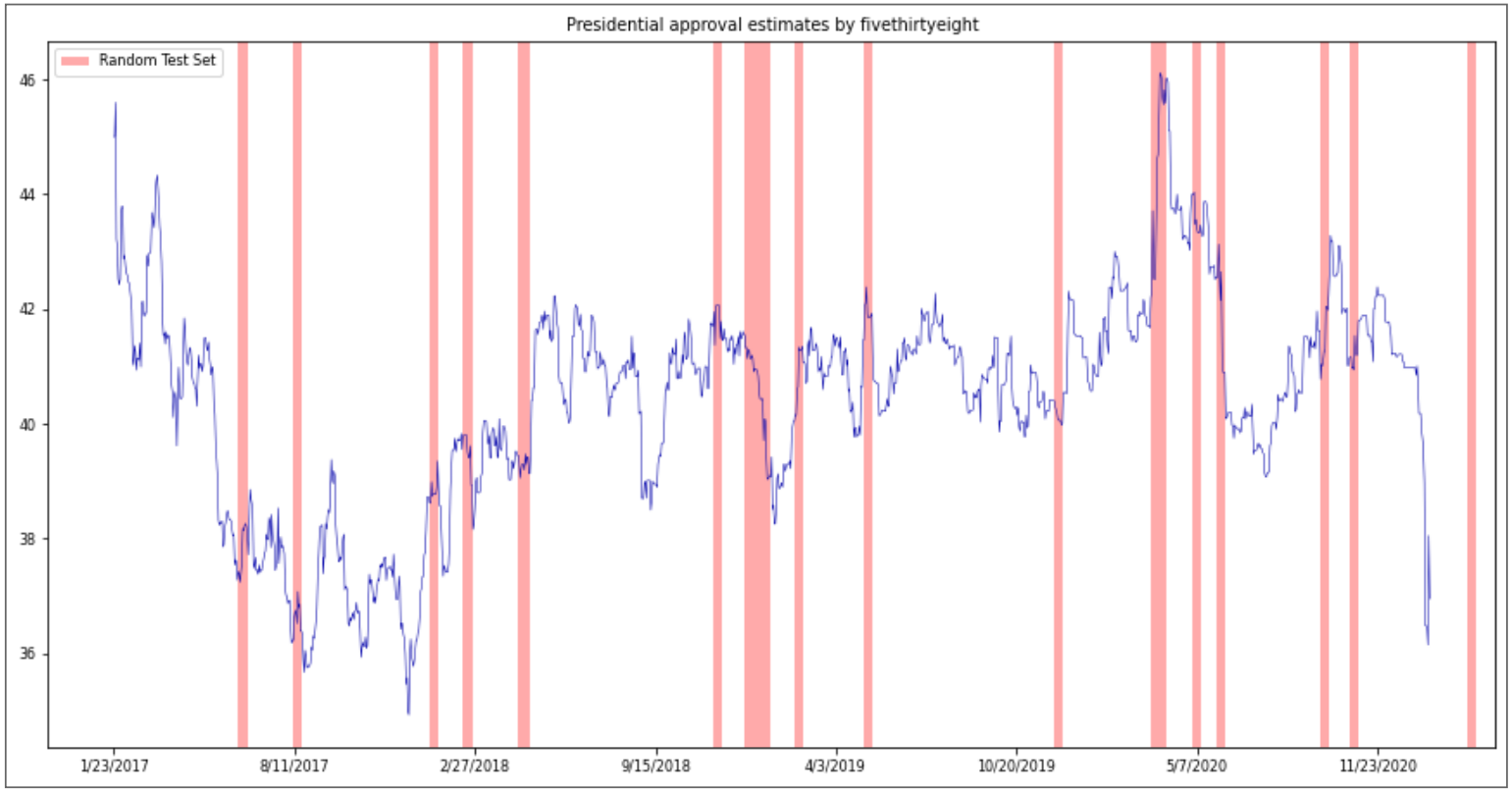
Cross-validation with groups

- Sometimes the data contains inherent groups:
    - Multiple samples from same patient, images from same person,...
- Data from the same person may end up in the training *and* test set
- We want to measure how well the model generalizes to *other* people
- Make sure that data from one person are in *either* the train or test set
    - This is called *grouping* or *blocking*
    - Leave-one-subject-out cross-validation: test set for each subject/group



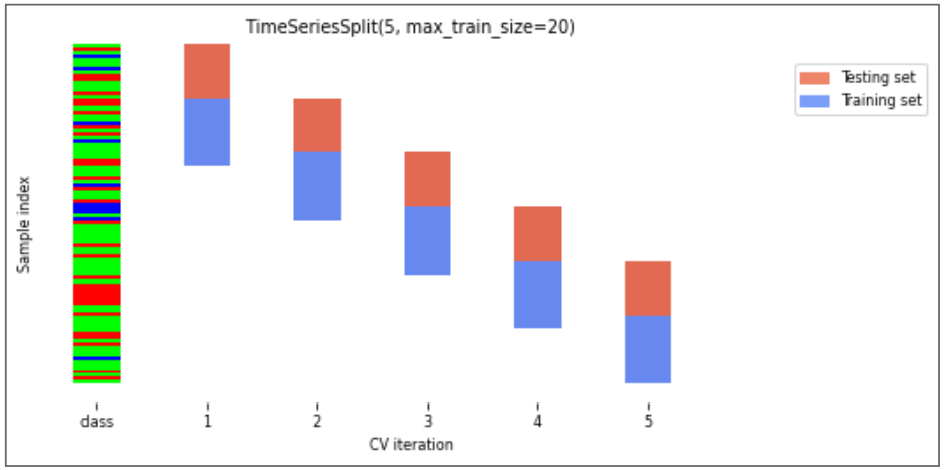
Time series

When the data is ordered, random test sets are not a good idea



Test-then-train (prequential evaluation)

- Every new sample is evaluated only once, then added to the training set
    - Can also be done in batches (of *n* samples at a time)
- `TimeSeriesSplit`
    - In the kth split, the first k folds are the train set and the (k+1)th fold as the test set
    - Often, a maximum training set size (or window) is used
        - more robust against concept drift (change in data over time)



Choosing a performance estimation procedure

No strict rules, only guidelines:

- Always use stratification for classification (sklearn does this by default)
- Use holdout for very large datasets (e.g. >1.000.000 examples)
    - Or when learners don't always converge (e.g. deep learning)
- Choose *k* depending on dataset size and resources
    - Use leave-one-out for very small datasets (e.g. <100 examples)
    - Use cross-validation otherwise
        - Most popular (and theoretically sound): 10-fold CV
        - Literature suggests 5x2-fold CV is better
- Use grouping or leave-one-subject-out for grouped data
- Use train-then-test for time series

Evaluation Metrics for Classification

Evaluation vs Optimization

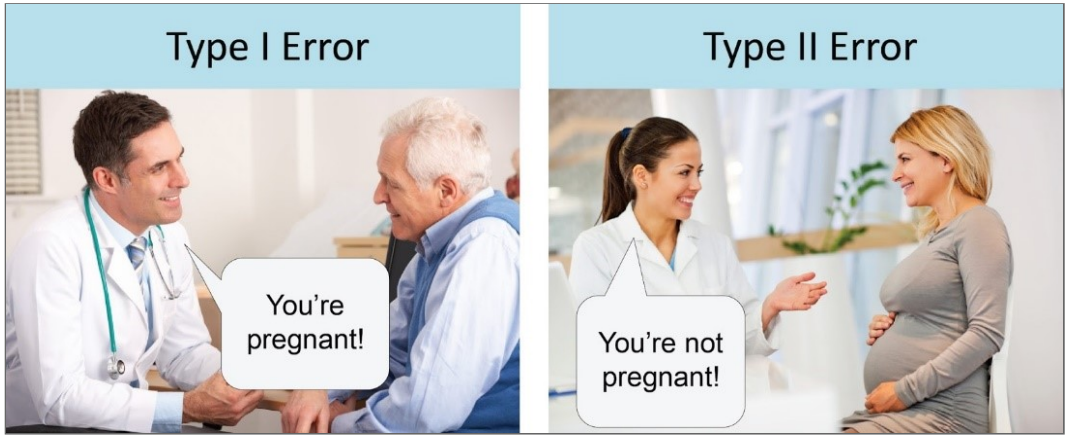
- Each algorithm optimizes a given objective function (on the training data)
    - E.g. remember L2 loss in Ridge regression

$$\mathcal{L}_{Ridge} = \sum_{n=1}^{N}(y_n - (\mathbf{w}\mathbf{x_n} + w_0))^2 + \alpha \sum_{i=0}^{p} w_i^2$$

- The choice of function is limited by what can be efficiently optimized
- However, we *evaluate* the resulting model with a score that makes sense **in the real world**
    - Percentage of correct predictions (on a test set)
    - The actual cost of mistakes (e.g. in money, time, lives,...)
- We also tune the algorithm's hyperparameters to maximize that score

Binary classification

- We have a positive and a negative class
- 2 different kind of errors:
    - False Positive (type I error): model predicts positive while true label is negative
    - False Negative (type II error): model predicts negative while true label is positive
- They are not always equally important
    - Which side do you want to err on for a medical test?



Confusion matrices

- We can represent all predictions (correct and incorrect) in a confusion matrix
    - n by n array (n is the number of classes)
    - Rows correspond to true classes, columns to predicted classes
    - Count how often samples belonging to a class C are classified as C or any other class.
    - For binary classification, we label these true negative (TN), true positive (TP), false negative (FN), false positive (FP)

	Predicted Neg	Predicted Pos
Actual Neg	TN	FP
Actual Pos	FN	TP

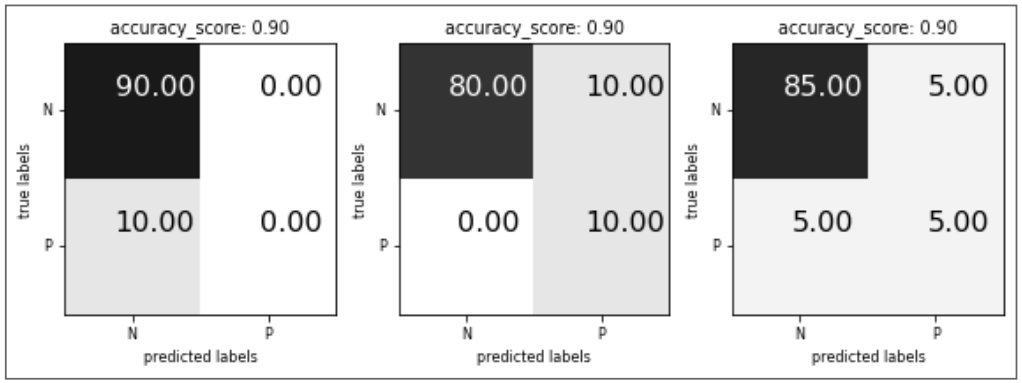
```
confusion_matrix(y_test, y_pred):
 [[48  5]
 [ 5 85]]
```

Predictive accuracy

- Accuracy can be computed based on the confusion matrix
- Not useful if the dataset is very imbalanced
    - E.g. credit card fraud: is 99.99% accuracy good enough?

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{1}$$

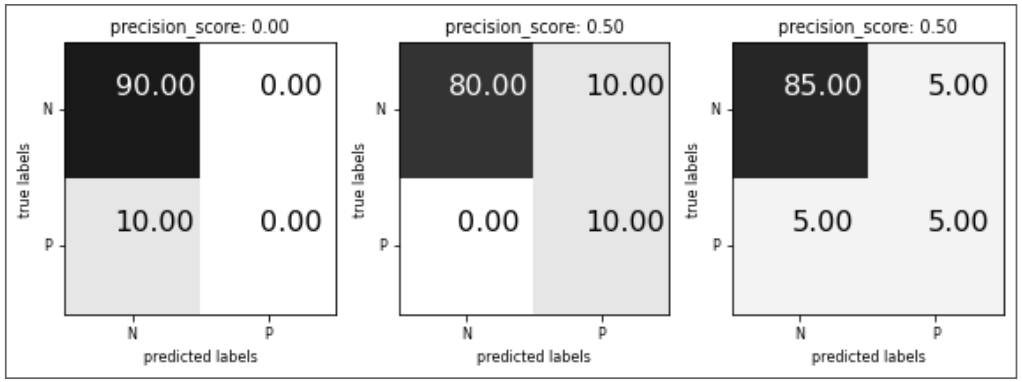
- 3 models: very different predictions, same accuracy:



Precision

- Use when the goal is to limit FPs
    - Clinical trails: you only want to test drugs that really work
    - Search engines: you want to avoid bad search results

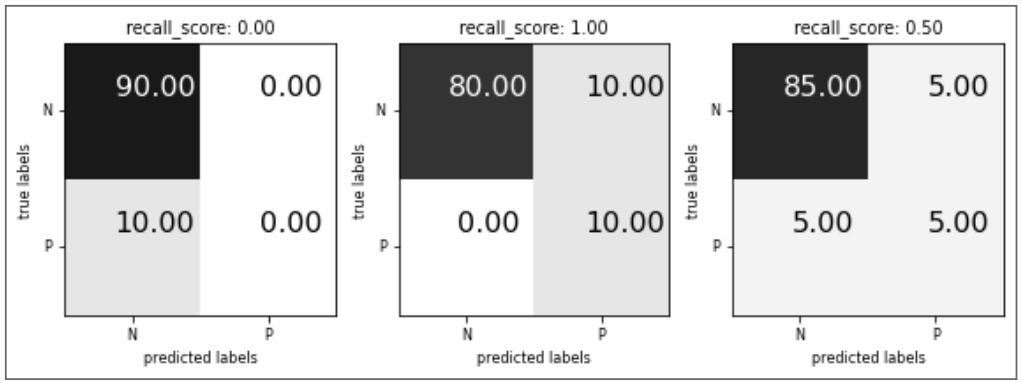
$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2}$$



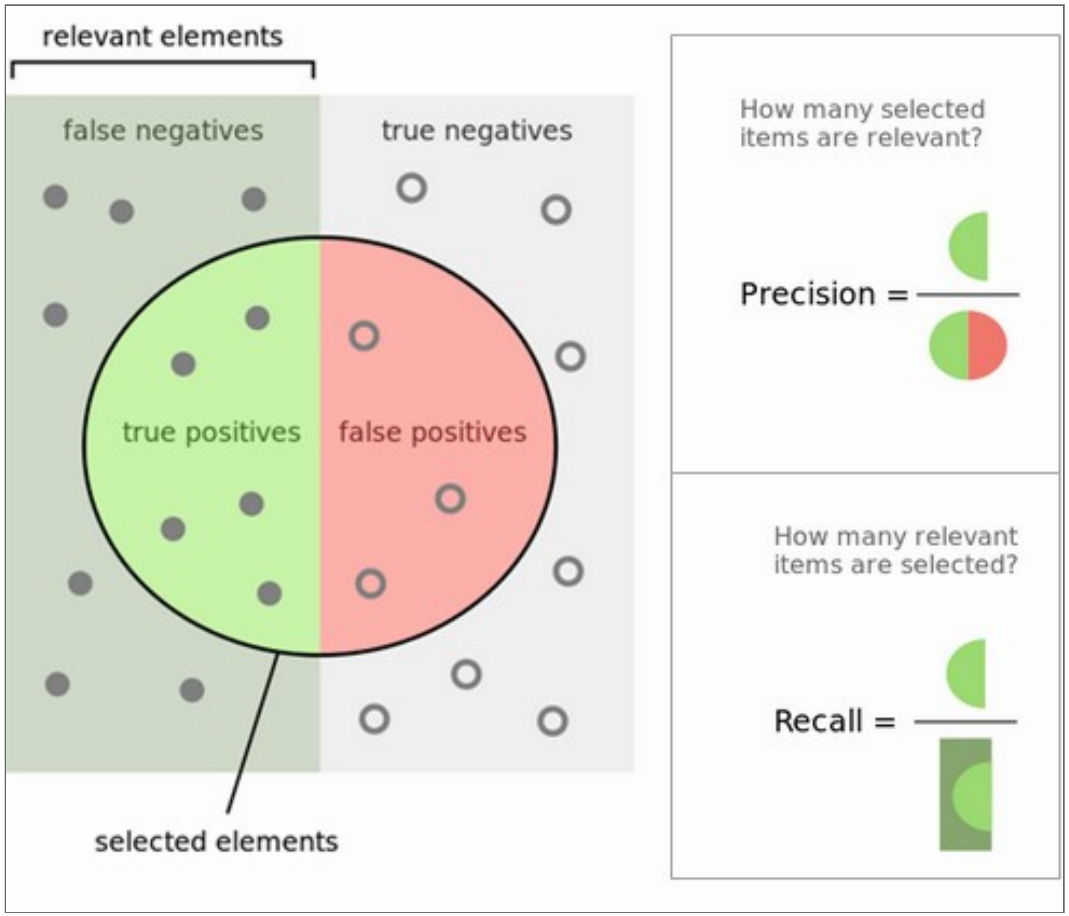
Recall

- Use when the goal is to limit FNs
    - Cancer diagnosis: you don't want to miss a serious disease
    - Search engines: You don't want to omit important hits
- Also know as sensitivity, hit rate, true positive rate (TPR)

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3}$$



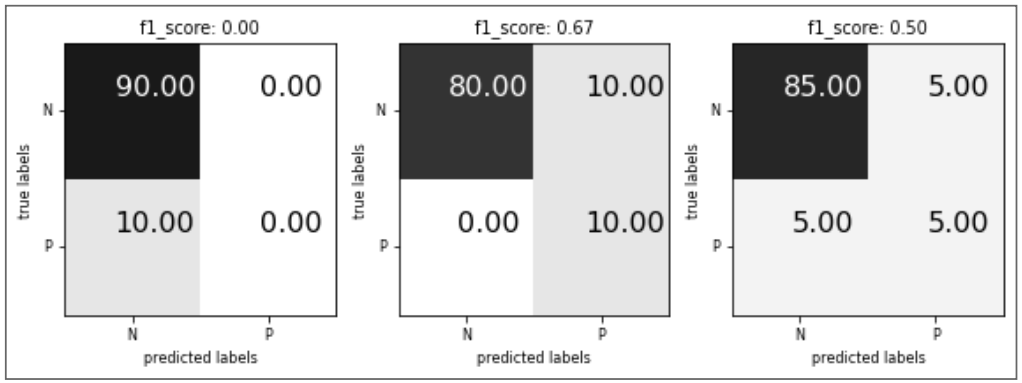
Comparison


relevant elements
false negatives
true negatives
true positives
false positives
selected elements
How many selected items are relevant?
Precision =
How many relevant items are selected?
Recall =

F1-SCORE

- Trades off precision and recall:

$$\mathrm{F1} = 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}} \tag{4}$$



Classification measure Zoo

		True condition				
	Total population	Condition positive	Condition negative	Prevalence = $\frac{\Sigma \text{ Condition positive}}{\Sigma \text{ Total population}}$	Accuracy (ACC) = $\frac{\Sigma \text{ True positive} + \Sigma \text{ True negative}}{\Sigma \text{ Total population}}$	
Predicted condition	Predicted condition positive	**True positive**, Power	**False positive**, Type I error	Positive predictive value (PPV), Precision = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ Predicted condition positive}}$	False discovery rate (FDR) = $\frac{\Sigma \text{ False positive}}{\Sigma \text{ Predicted condition positive}}$	
	Predicted condition negative	**False negative**, Type II error	**True negative**	False omission rate (FOR) = $\frac{\Sigma \text{ False negative}}{\Sigma \text{ Predicted condition negative}}$	Negative predictive value (NPV) = $\frac{\Sigma \text{ True negative}}{\Sigma \text{ Predicted condition negative}}$	
		True positive rate (TPR), Recall, Sensitivity, probability of detection = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ Condition positive}}$	False positive rate (FPR), Fall-out, probability of false alarm = $\frac{\Sigma \text{ False positive}}{\Sigma \text{ Condition negative}}$	Positive likelihood ratio (LR+) = $\frac{\text{TPR}}{\text{FPR}}$	Diagnostic odds ratio (DOR) = $\frac{\text{LR+}}{\text{LR−}}$	F_1 score = $\frac{2}{\frac{1}{\text{Recall}} + \frac{1}{\text{Precision}}}$
		False negative rate (FNR), Miss rate = $\frac{\Sigma \text{ False negative}}{\Sigma \text{ Condition positive}}$	Specificity (SPC), Selectivity, True negative rate (TNR) = $\frac{\Sigma \text{ True negative}}{\Sigma \text{ Condition negative}}$	Negative likelihood ratio (LR−) = $\frac{\text{FNR}}{\text{TNR}}$		

https://en.wikipedia.org/wiki/Precision_and_recall

Multi-class classification

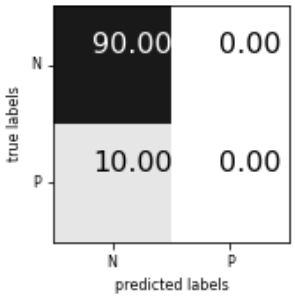
- Train models *per class* : one class viewed as positive, other(s) als negative, then average
  - micro-averaging: count total TP, FP, TN, FN (every sample equally important)
    - micro-precision, micro-recall, micro-F1, accuracy are all the same

$$\text{Precision:} \frac{\sum_{c=1}^{C} \text{TP}_c}{\sum_{c=1}^{C} \text{TP}_c + \sum_{c=1}^{C} \text{FP}_c} \xrightarrow{c=2} \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

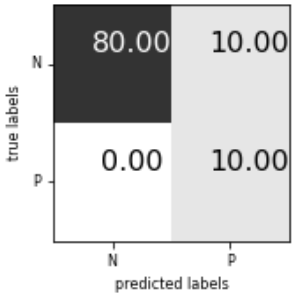
  - macro-averaging: average of scores $R(y_c, \hat{y}_c)$ obtained on each class
    - Preferable for imbalanced classes (if all classes are equally important)
    - macro-averaged recall is also called *balanced accuracy*

$$\frac{1}{C} \sum_{c=1}^{C} R(y_c, \hat{y}_c)$$

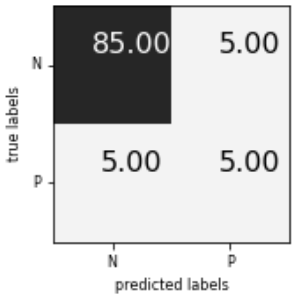
  - weighted averaging (w_c: ratio of examples of class c, aka support): $\sum_{c=1}^{C} w_c R(y_c, \hat{y}_c)$



	precision	recall	f1-score	support
0	0.90	1.00	0.95	90
1	0.00	0.00	0.00	10
accuracy			0.90	100
macro avg	0.45	0.50	0.47	100
weighted avg	0.81	0.90	0.85	100



	precision	recall	f1-score	support
0	1.00	0.89	0.94	90
1	0.50	1.00	0.67	10
accuracy			0.90	100
macro avg	0.75	0.94	0.80	100
weighted avg	0.95	0.90	0.91	100



	precision	recall	f1-score	support
0	0.94	0.94	0.94	90
1	0.50	0.50	0.50	10
accuracy			0.90	100
macro avg	0.72	0.72	0.72	100
weighted avg	0.90	0.90	0.90	100

Other useful classification metrics

- Cohen's Kappa
  - Measures 'agreement' between different models (aka inter-rater agreement)
  - To evaluate a single model, compare it against a model that does random guessing
    - Similar to accuracy, but taking into account the possibility of predicting the right class by chance
  - Can be weighted: different misclassifications given different weights
  - 1: perfect prediction, 0: random prediction, negative: worse than random
  - With p_0 = accuracy, and p_e = accuracy of random classifier:

$$\kappa = \frac{p_o - p_e}{1 - p_e}$$

- Matthews correlation coefficient
  - Corrects for imbalanced data, alternative for balanced accuracy or AUROC
  - 1: perfect prediction, 0: random prediction, -1: inverse prediction

$$MCC = \frac{tp \times tn - fp \times fn}{\sqrt{(tp + fp)(tp + fn)(tn + fp)(tn + fn)}}$$

Probabilistic evaluation

- Classifiers can often provide uncertainty estimates of predictions.
- Remember that linear models actually return a numeric value.
    - When $\hat{y} < 0$, predict class -1, otherwise predict class +1

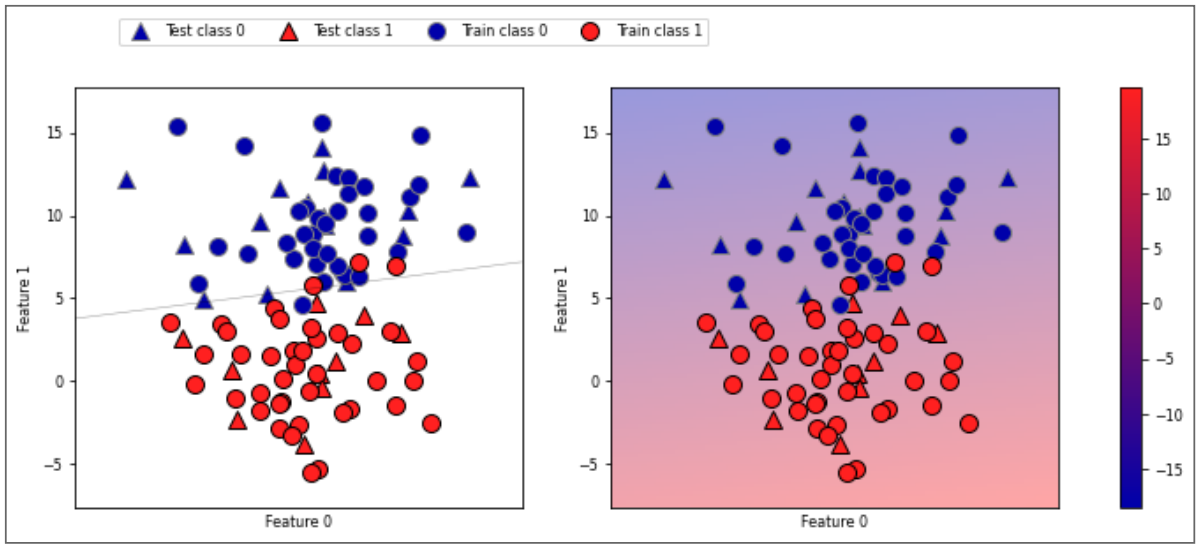
$$\hat{y} = w_0 * x_0 + w_1 * x_1 + \ldots + w_p * x_p + b$$

- In practice, you are often interested in how certain a classifier is about each class prediction (e.g. cancer treatments).
- Most learning methods can return at least one measure of *confidence* in their predicions.
    - Decision function: floating point value for each sample (higher: more confident)
    - Probability: estimated probability for each class

The decision function

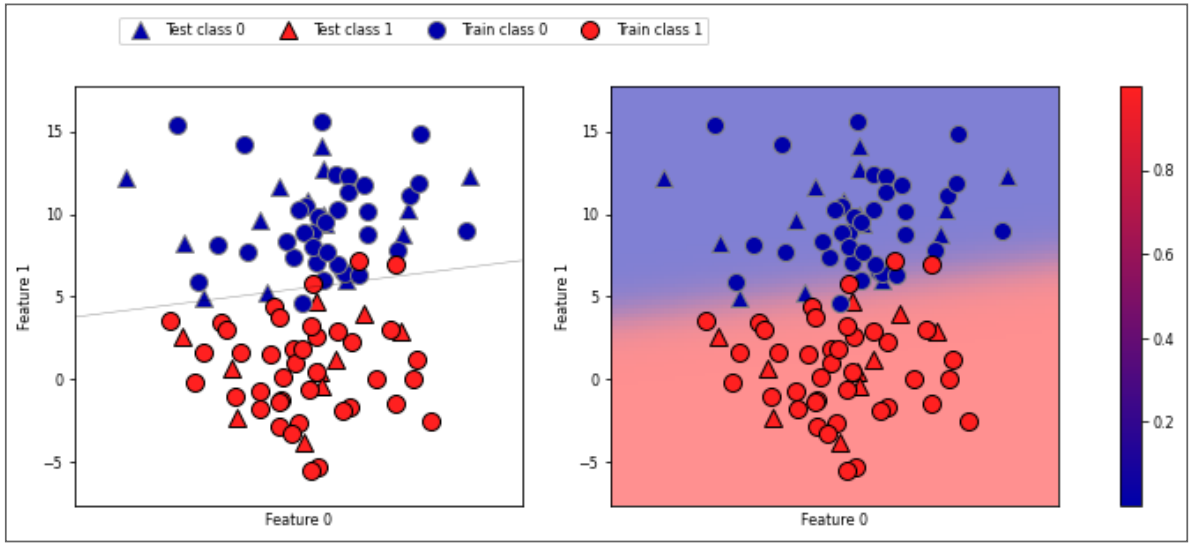
In the binary classification case, the return value of the decision function encodes how strongly the model believes a data point belongs to the “positive” class.

- Positive values indicate preference for the positive class.
- The range can be arbitrary, and can be affected by hyperparameters. Hard to interpret.



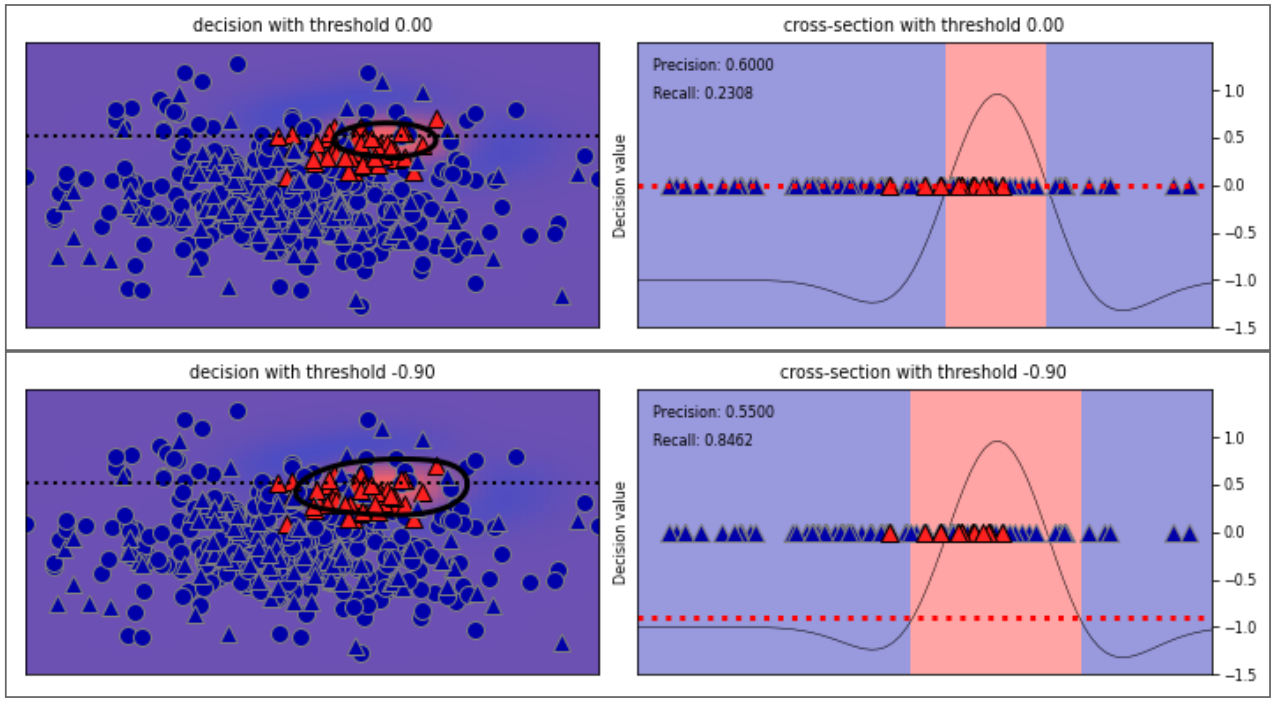
Predicting probabilities

Some models can also return a *probability* for each class with every prediction. These sum up to 1. We can visualize them again. Note that the gradient looks different now.



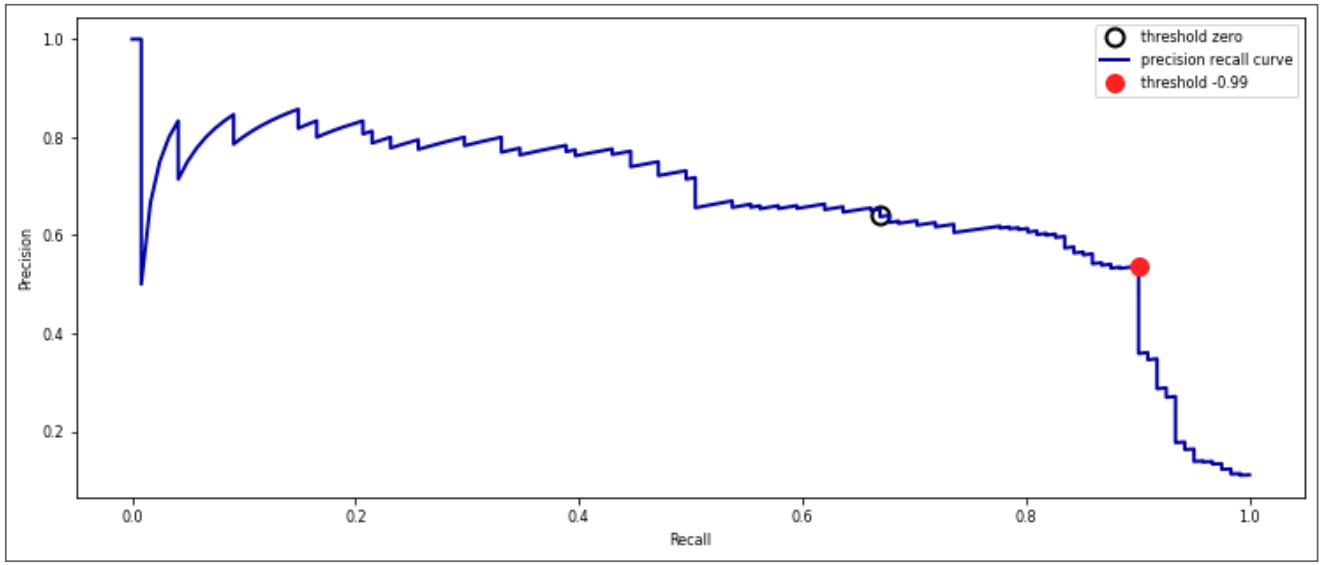
Threshold calibration

- By default, we threshold at 0 for `decision_function` and 0.5 for `predict_proba`
- Depending on the application, you may want to threshold differently
    - Lower threshold yields fewer FN (better recall), more FP (worse precision), and vice-versa



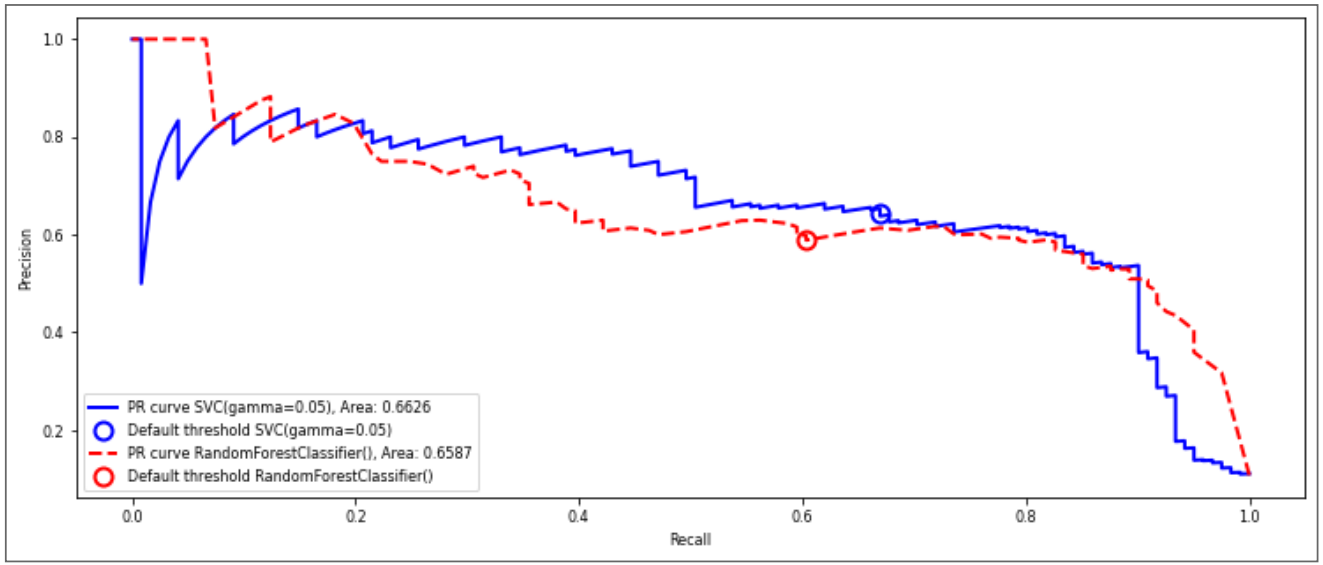
Precision-Recall curve

- The best trade-off between precision and recall depends on your application
    - You can have arbitrary high recall, but you often want reasonable precision, too.
- Plotting precision against recall *for all possible thresholds* yields a **precision-recall curve**
    - Change the treshold until you find a sweet spot in the precision-recall trade-off
    - Often jagged at high thresholds, when there are few positive examples left



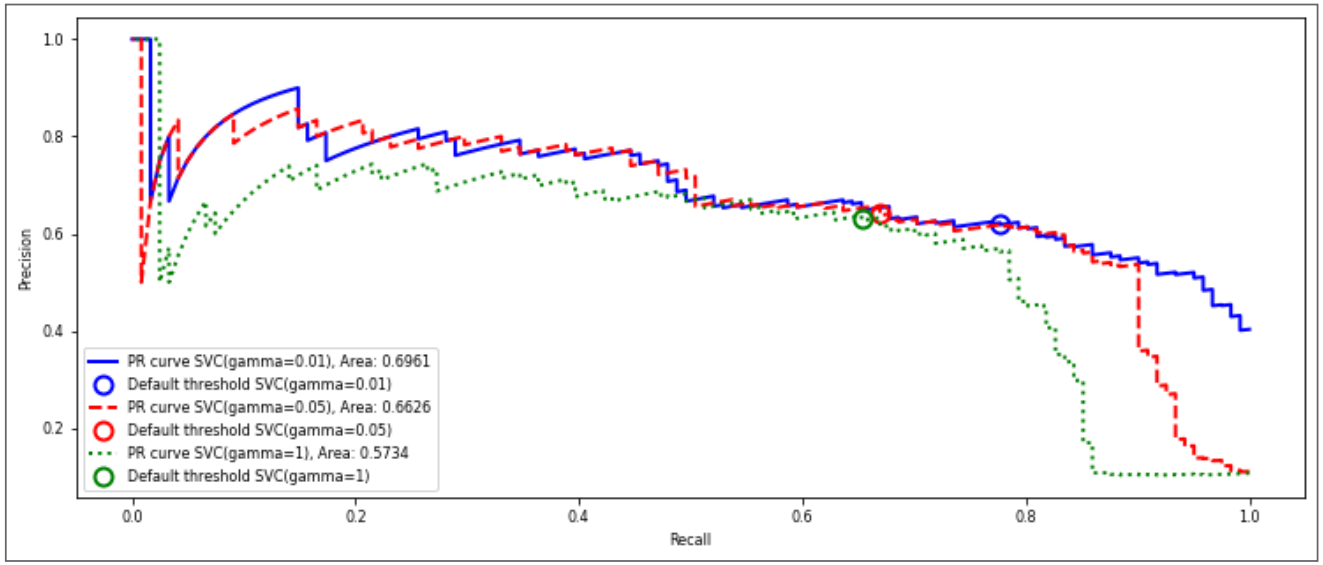
Model selection

- Some models can achieve trade-offs that others can't
- Your application may require very high recall (or very high precision)
    - Choose the model that offers the best trade-off, given your application
- The area under the PR curve (AUPRC) gives the *best overall* model



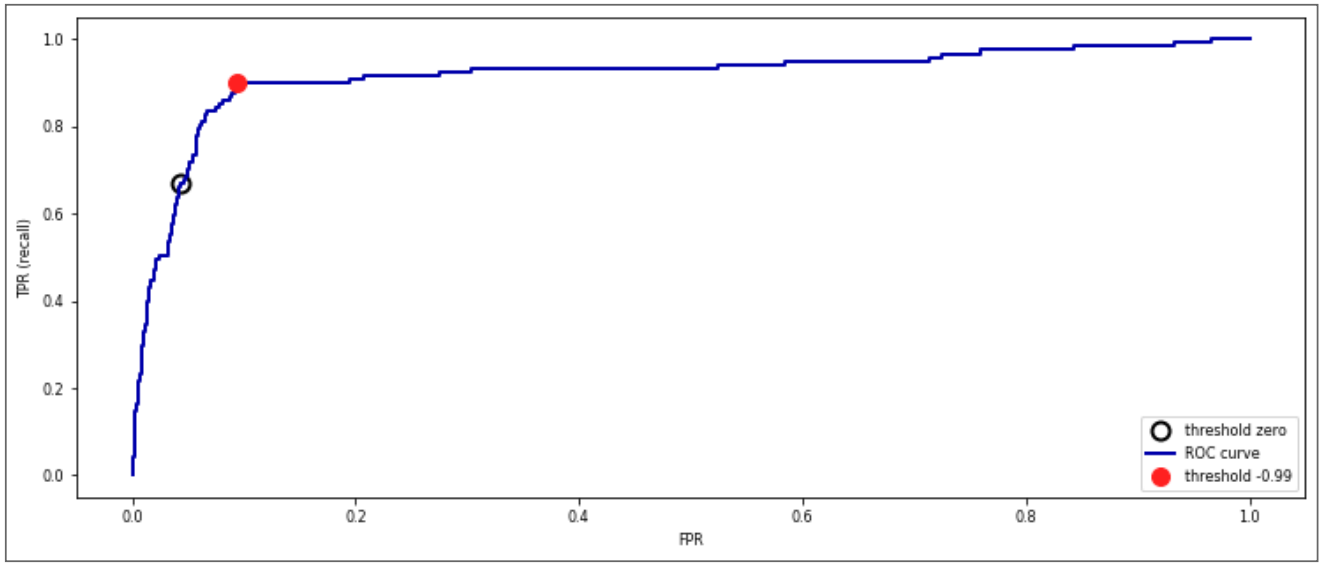
Hyperparameter effects

Of course, hyperparameters affect predictions and hence also the shape of the curve



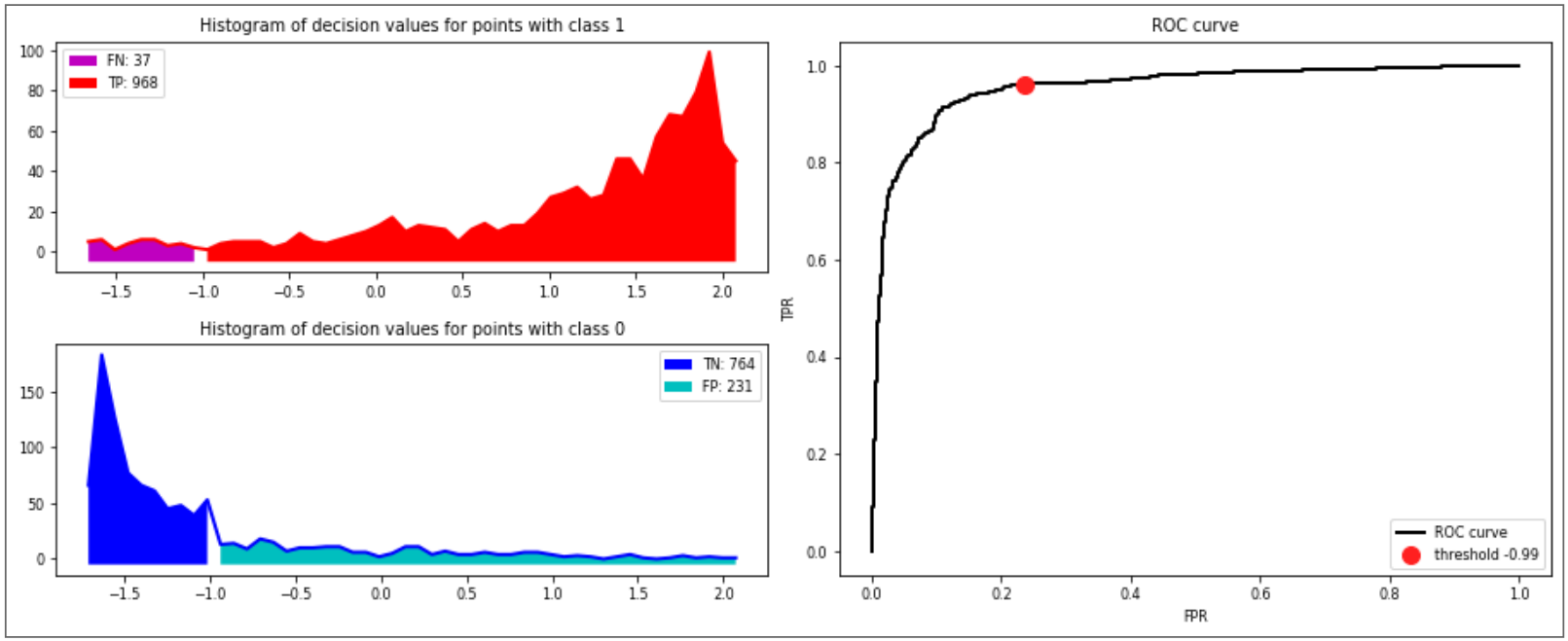
Receiver Operating Characteristics (ROC)

- Trade off *true positive rate* $TPR = \frac{TP}{TP+FN}$ with *false positive rate* $FPR = \frac{FP}{FP+TN}$
- Plotting TPR against FPR *for all possible thresholds* yields a *Receiver Operating Characteristics curve*
    - Change the treshold until you find a sweet spot in the TPR-FPR trade-off
    - Lower thresholds yield higher TPR (recall), higher FPR, and vice versa



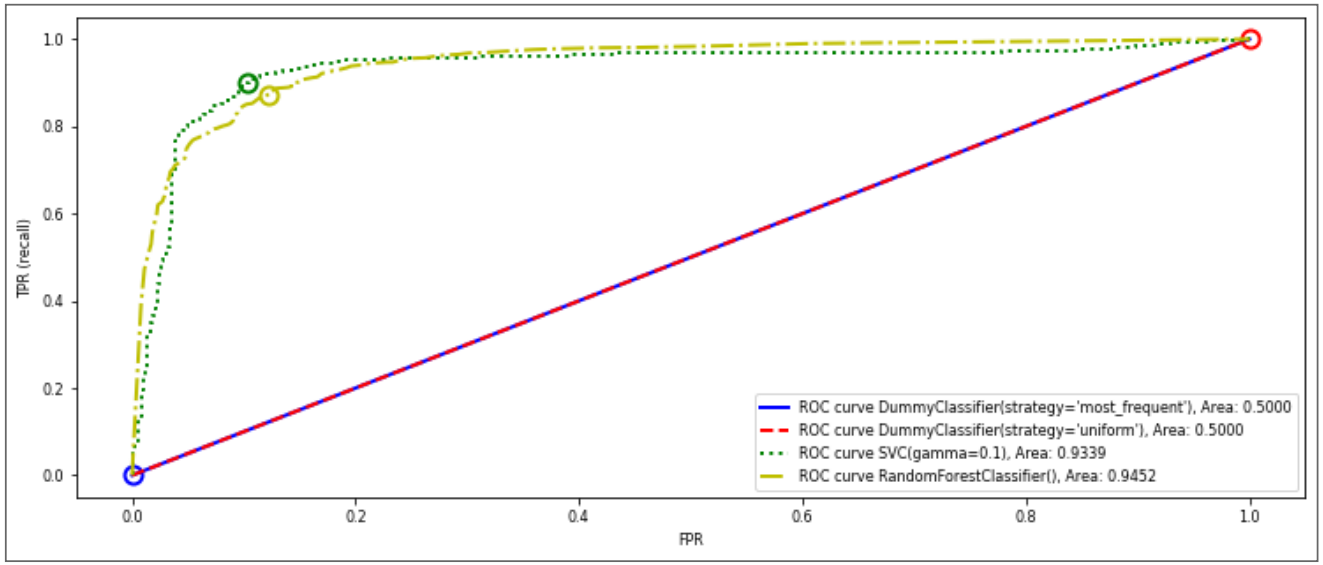
Visualization

- Histograms show the amount of points with a certain decision value (for each class)
- $TPR = \frac{TP}{TP+FN}$ can be seen from the positive predictions (top histogram)
- $FPR = \frac{FP}{FP+TN}$ can be seen from the negative predictions (bottom histogram)



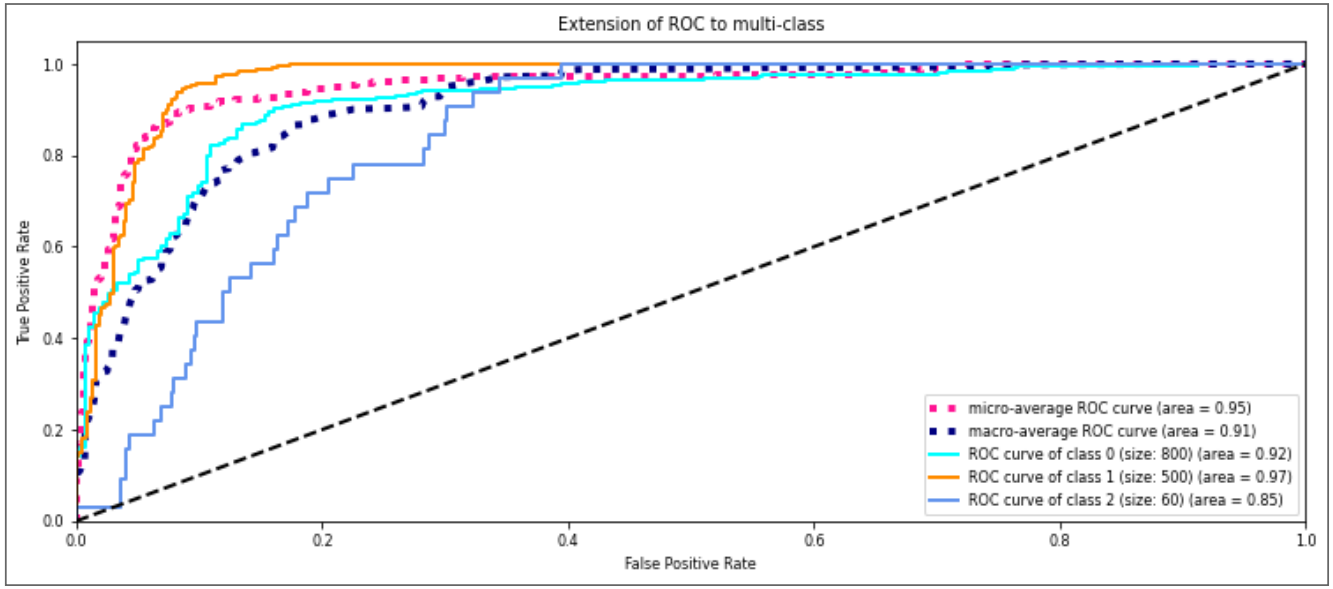
Model selection

- Again, some models can achieve trade-offs that others can't
- Your application may require minizing FPR (low FP), or maximizing TPR (low FN)
- The area under the ROC curve (AUROC or AUC) gives the *best overall* model
    - Frequently used for evaluating models on imbalanced data
    - Random guessing (TPR=FPR) or predicting majority class (TPR=FPR=1): 0.5 AUC



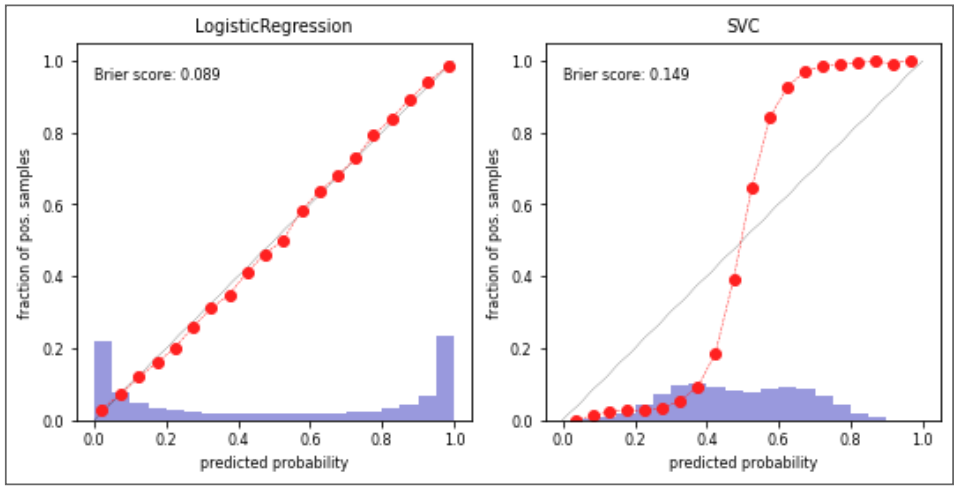
Multi-class AUROC (or AUPRC)

- We again need to choose between micro- or macro averaging TPR and FPR.
  - Micro-average if every sample is equally important (irrespective of class)
  - Macro-average if every class is equally important, especially for imbalanced data



Model calibration

- For some models, the *predicted* uncertainty does not reflect the *actual* uncertainty
    - If a model is 90% sure that samples are positive, is it also 90% accurate on these?
- A model is called *calibrated* if the reported uncertainty actually matches how correct it is
    - Overfitted models also tend to be over-confident
    - LogisticRegression models are well calibrated since they learn probabilities
    - SVMs are not well calibrated. *Biased* towards points close to the decision boundary.



Brier score

- You may want to select models based on how accurate the class confidences are.
- The **Brier score loss**: squared loss between predicted probability $\hat{p}$ and actual outcome y
    - Lower is better

$$\mathcal{L}_{Brier} = \frac{1}{n}\sum_{i=1}^{n}(\hat{p}_i - y_i)^2$$

```
Logistic Regression Brier score loss: 0.0322
SVM Brier score loss: 0.0795
```

Model calibration techniques

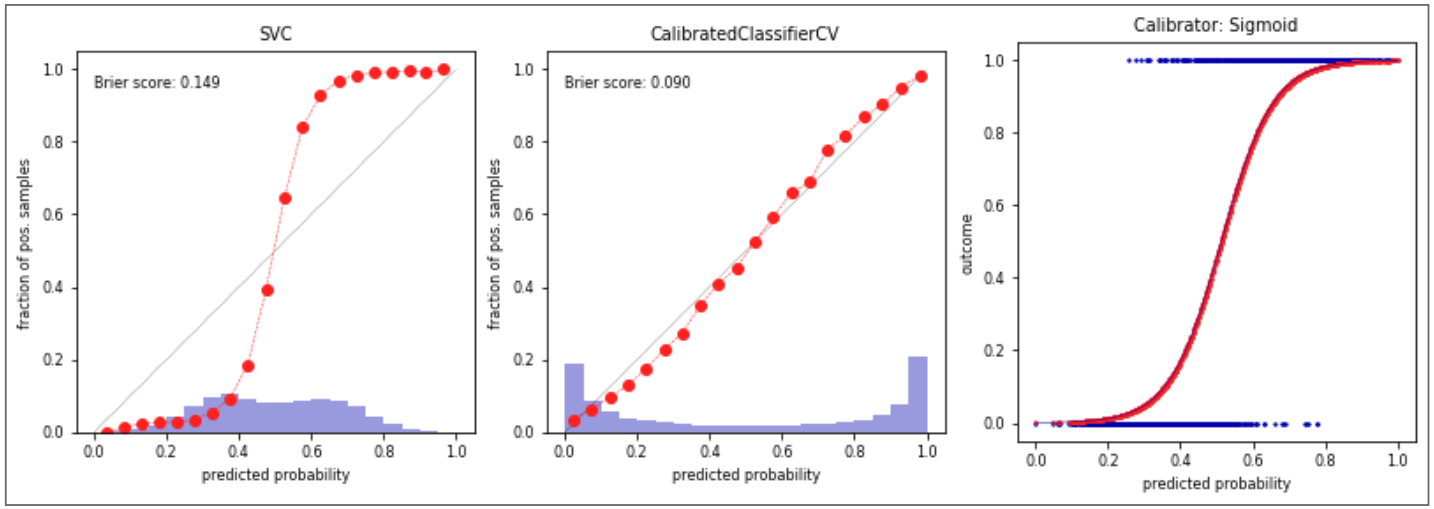
- We can post-process trained models to make them more calibrated.
- Fit a regression model (a calibrator) to map the model's outcomes $f(x)$ to a calibrated probability in [0,1]
    - $f(x)$ returns the decision values or probability estimates
    - f_{calib} is fitted on the training data to map these to the correct outcome
        - Often an internal cross-validation with few folds is used
    - Multi-class models require one calibrator per class

$$f_{calib}(f(x)) \approx p(y)$$

Platt Scaling

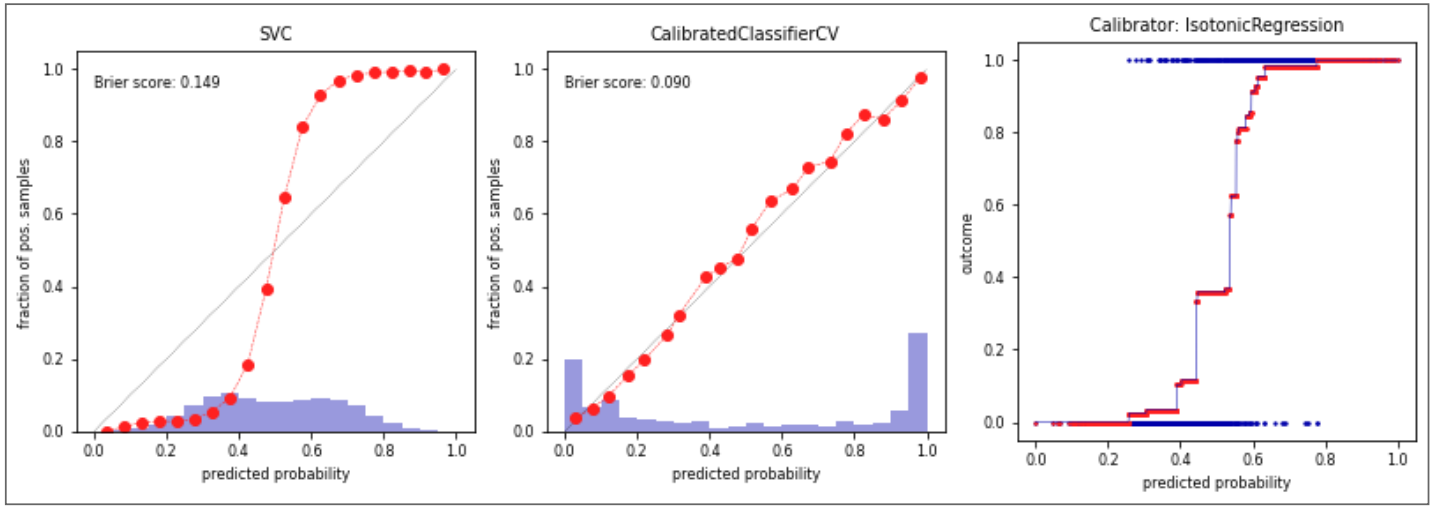
- Calibrator is a logistic (sigmoid) function:
  - Learn the weight w_1 and bias w_0 from data

$$f_{platt} = \frac{1}{1 + \exp(-w_1 f(x) - w_0)}$$



Isotonic regression

- Maps input x_i to an output $\hat{y}_i$ so that $\hat{y}_i$ increases monotonically with x_i and minimizes loss $\sum_i^n (y_i - \hat{y}_i)$
    - Predictions are made by interpolating the predicted $\hat{y}_i$
- Fit to minimize the loss between the uncalibrated predictions $f(x)$ and the actual labels
- Corrects any monotonic distortion, but tends to overfit on small samples



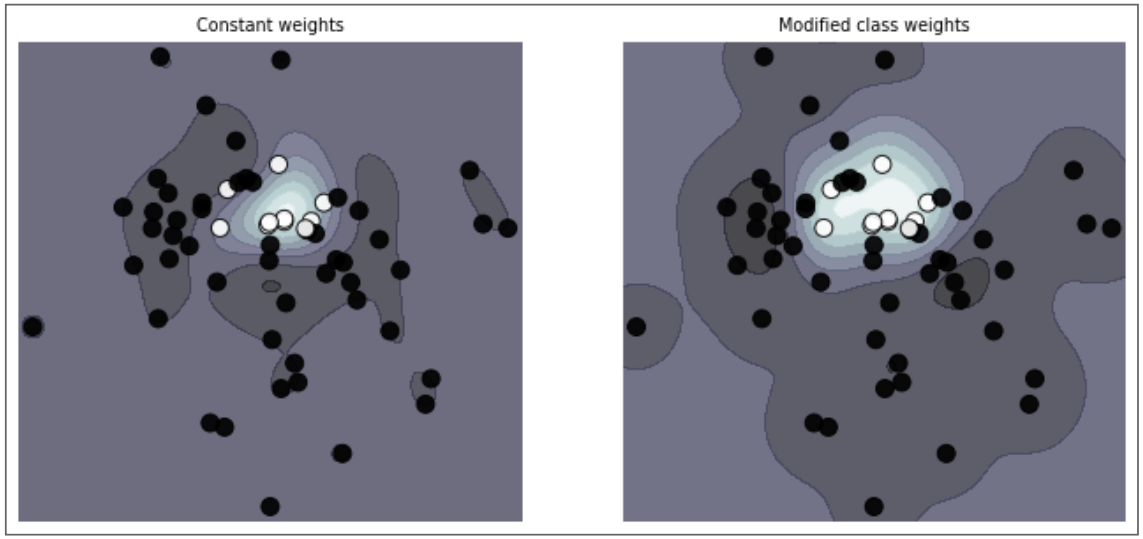
Cost-sensitive classification (dealing with imbalance)

- In the real worlds, different kinds of misclassification can have different costs
    - Misclassifying certain classes can be more costly than others
    - Misclassifying certain samples can be more costly than others
- Cost-sensitive resampling: resample (or reweight) the data to represent real-world expectations
    - oversample minority classes (or undersample majority) to 'correct' imbalance
    - increase weight of misclassified samples (e.g. in boosting)
    - decrease weight of misclassified (noisy) samples (e.g. in model compression)

Class weighting

- If some classes are more important than others, we can give them more weight
    - E.g. for imbalanced data, we can give more weight to minority classes
- Most classification models can include it in their loss function and optimize for it
    - E.g. Logistic regression: add a class weight w_c in the log loss function

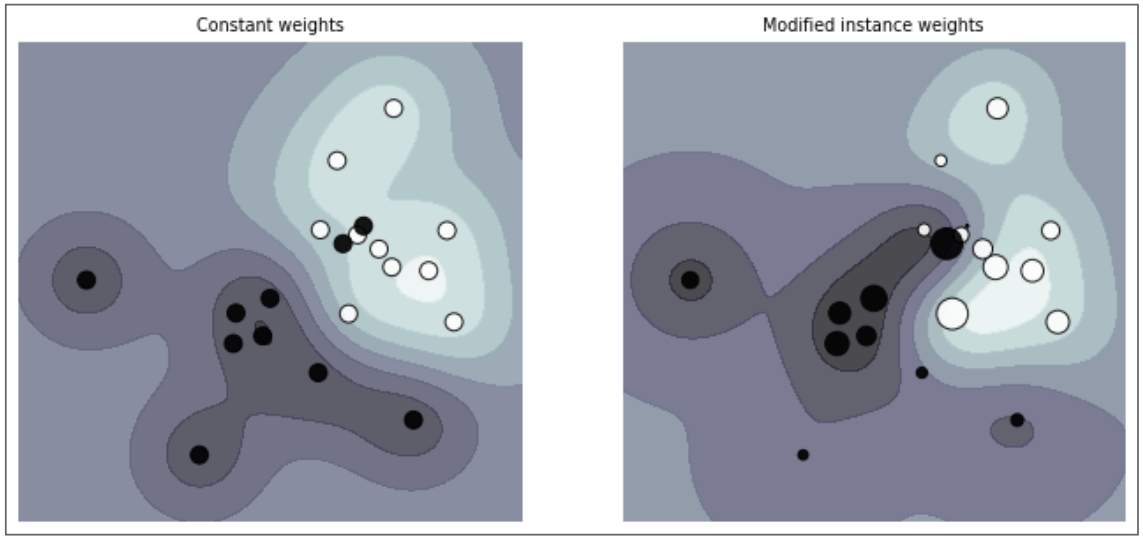
$$\mathcal{L}_{log}(\mathbf{w}) = -\sum_{c=1}^{C} w_c \sum_{n=1}^{N} p_{n,c} log(q_{n,c})$$



Instance weighting

- If some *training instances* are important to get right, we can give them more weight
    - E.g. when some examples are from groups underrepresented in the data
- These are passed during training (fit), and included in the loss function
    - E.g. Logistic regression: add a instance weight w_n in the log loss function

$$\mathcal{L}_{log}(\mathbf{w}) = -\sum_{c=1}^{C}\sum_{n=1}^{N} w_n p_{n,c} log(q_{n,c})$$



Cost-sensitive algorithms

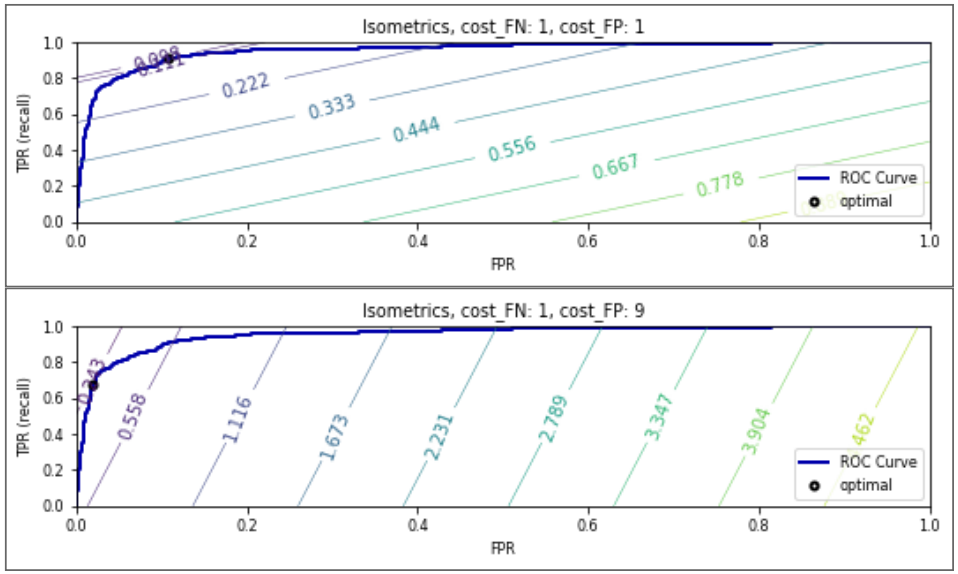
- Cost-sensitive algorithms
  - If misclassification cost of some classes is higher, we can give them higher weights
  - Some support *cost matrix* C: costs $c_{i,j}$ for every possible type of error
- Cost-sensitive ensembles: convert cost-insensitive classifiers into cost-sensitive ones
  - MetaCost: Build a model (ensemble) to learn the class probabilities $P(j|x)$
    - Relabel training data to minimize expected cost: $\operatorname{argmin}_i \sum_j P_j(x) c_{i,j}$
    - Accuracy may decrease but cost decreases as well.
  - AdaCost: Boosting with reweighting instances to reduce costs

Tuning the decision threshold

- If every FP or FN has a certain cost, we can compute the total cost for a given model:

$$\text{total cost} = \text{FPR} * cost_{FP} * ratio_{pos} + (1 - \text{TPR}) * cost_{FN} * (1 - ratio_{pos})$$

- This yields different *isometrics* (lines of equal cost) in ROC space
- Optimal threshold is the point on the ROC curve where cost is minimal (line search)



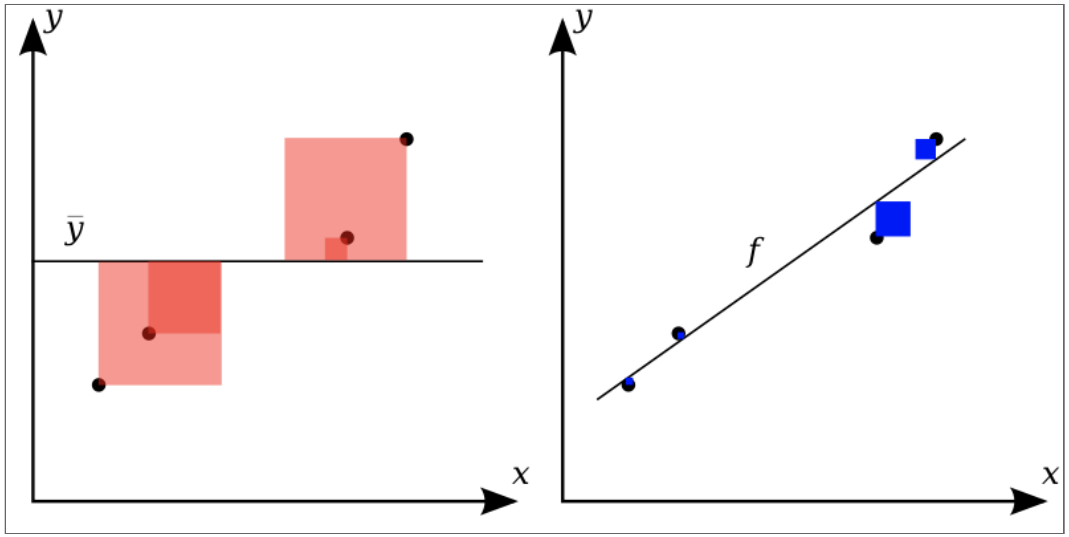
Regression metrics

Most commonly used are

- mean squared error: $\frac{\sum_i (y_{pred_i} - y_{actual_i})^2}{n}$
    - root mean squared error (RMSE) often used as well
- mean absolute error: $\frac{\sum_i |y_{pred_i} - y_{actual_i}|}{n}$
    - Less sensitive to outliers and large errors

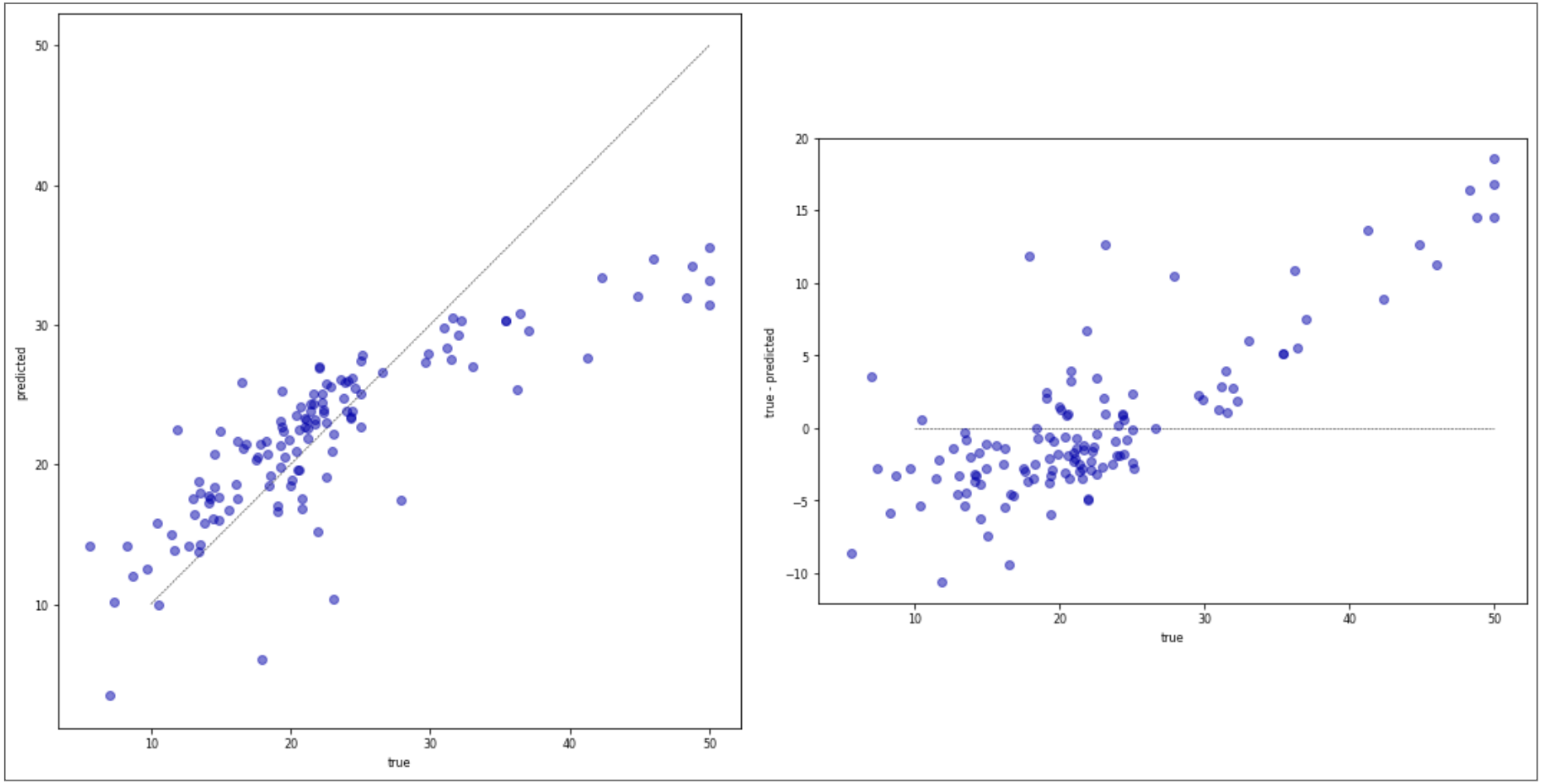
R squared

- $R^2 = 1 - \frac{\sum_i (y_{pred_i} - y_{actual_i})^2}{\sum_i (y_{mean} - y_{actual_i})^2}$
  - Ratio of variation explained by the model / total variation
  - Between 0 and 1, but *negative* if the model is worse than just predicting the mean
  - Easier to interpret (higher is better).



Visualizing regression errors

- Prediction plot (left): predicted vs actual target values
- Residual plot (right): residuals vs actual target values
    - Over- and underpredictions can be given different costs



Bias-Variance decomposition

- Evaluate the same algorithm multiple times on different random samples of the data
- Two types of errors can be observed:
    - Bias error: systematic error, independent of the training sample
        - These points are predicted (equally) wrong every time
    - Variance error: error due to variability of the model w.r.t. the training sample
        - These points are sometimes predicted accurately, sometimes inaccurately

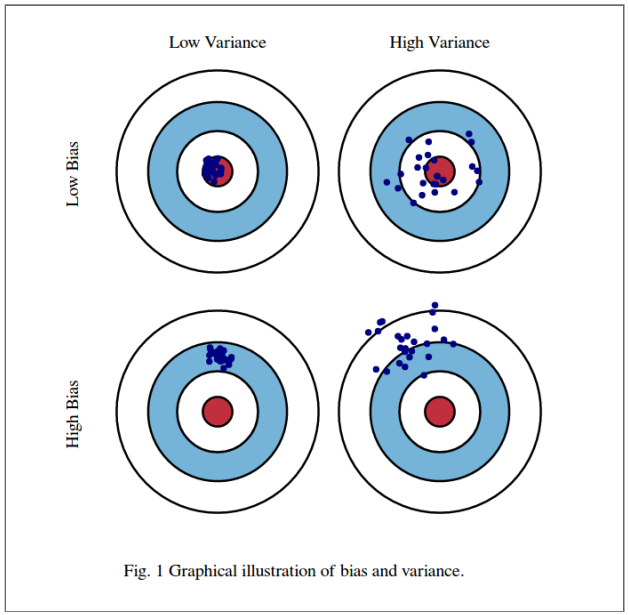


Fig. 1 Graphical illustration of bias and variance.

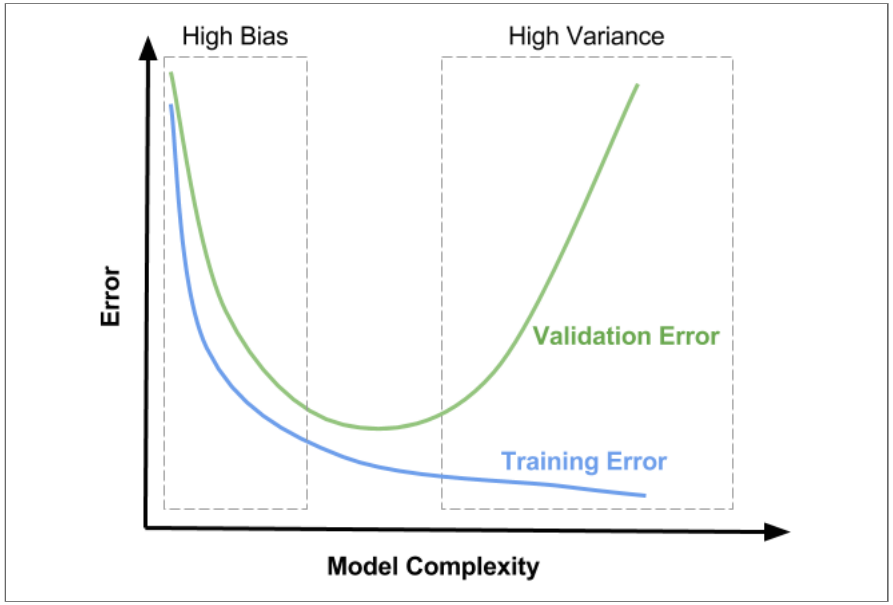
Computing bias and variance error

- Take 100 or more bootstraps (or shuffle-splits)
- Regression: for each data point x:
    - $bias(x)^2 = (x_{true} - mean(x_{predicted}))^2$
    - $variance(x) = var(x_{predicted})$
- Classification: for each data point x:
    - $bias(x)$ = misclassification ratio
    - $variance(x)$

      $= (1 - (P(class_1)^2 + P(class_2)^2))/2$
        - $P(class_i)$ is ratio of class i predictions
- Total bias: $\sum_x bias(x)^2 * w_x$ w_x: the percentage of times x occurs in the test sets
- Total variance: $\sum_x variance(x) * w_x$

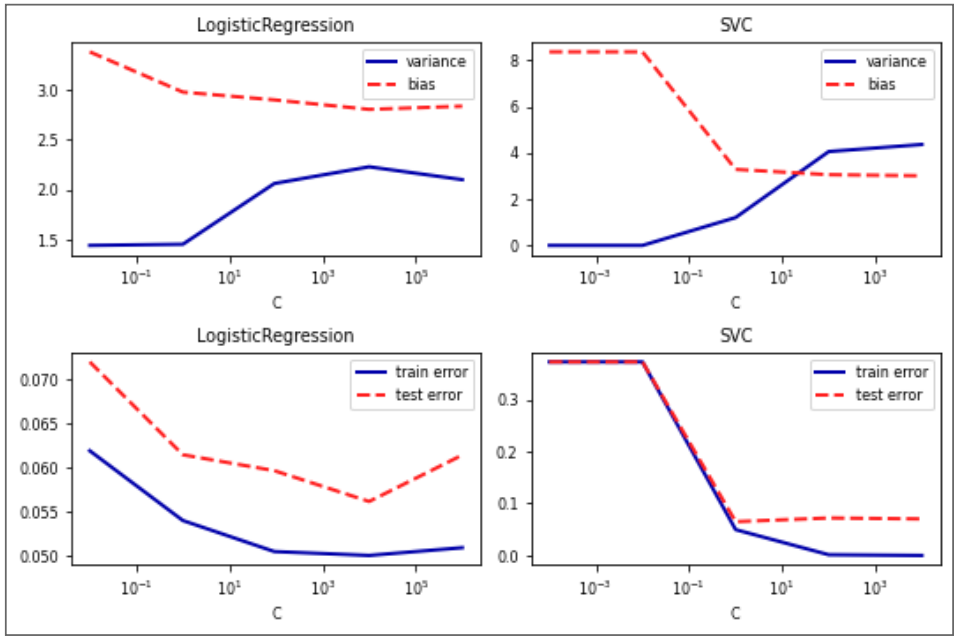
Bias and variance, underfitting and overfitting

- High variance means that you are likely overfitting
    - Use more regularization or use a simpler model
- High bias means that you are likely underfitting
    - Do less regularization or use a more flexible/complex model
- Ensembling techniques (see later) reduce bias or variance directly
    - Bagging (e.g. RandomForests) reduces variance, Boosting reduces bias

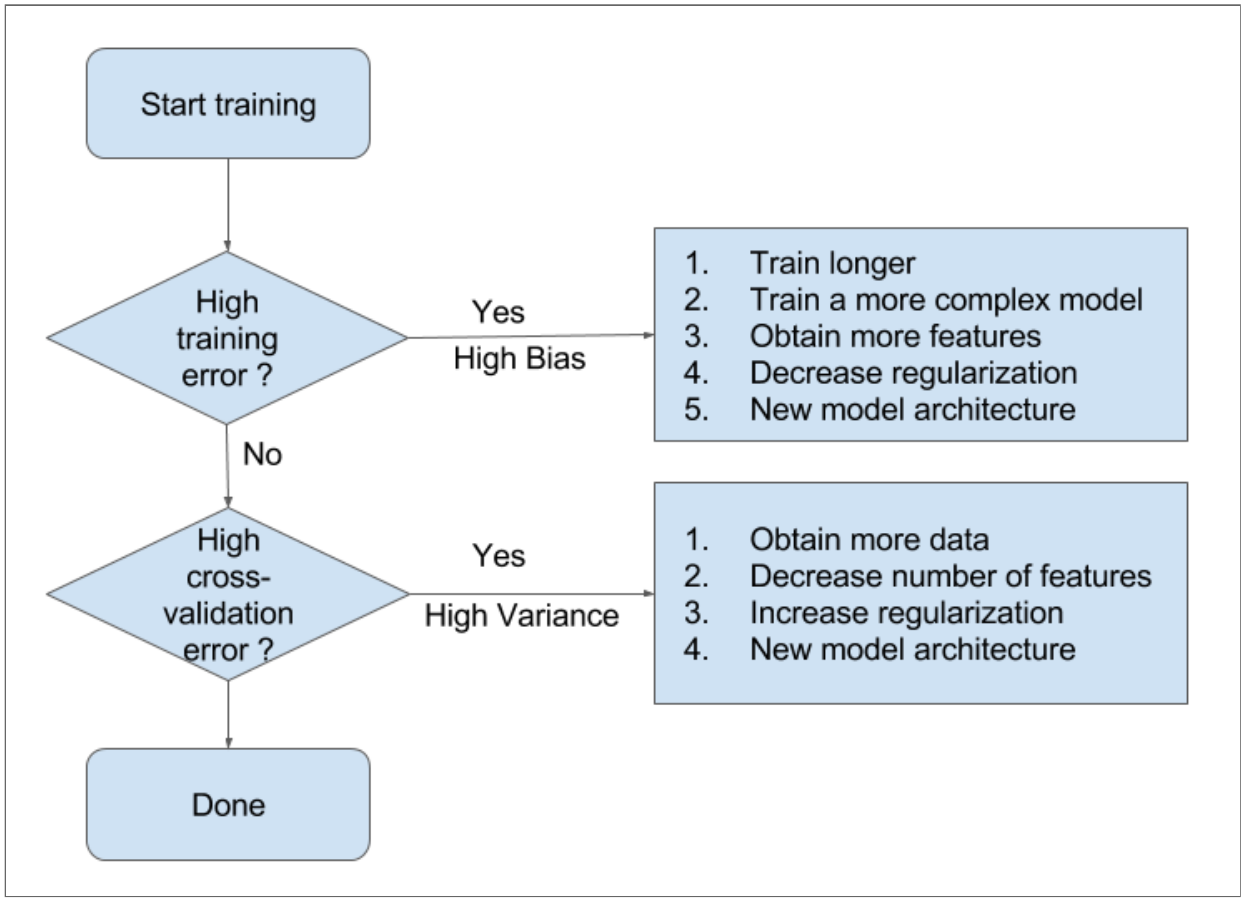


Understanding under- and overfitting

- Regularization reduces variance error (increases stability of predictions)
    - But too much increases bias error (inability to learn 'harder' points)
- High regularization (left side): Underfitting, high bias error, low variance error
    - High training error and high test error
- Low regularization (right side): Overfitting, low bias error, high variance error
    - Low training error and higher test error

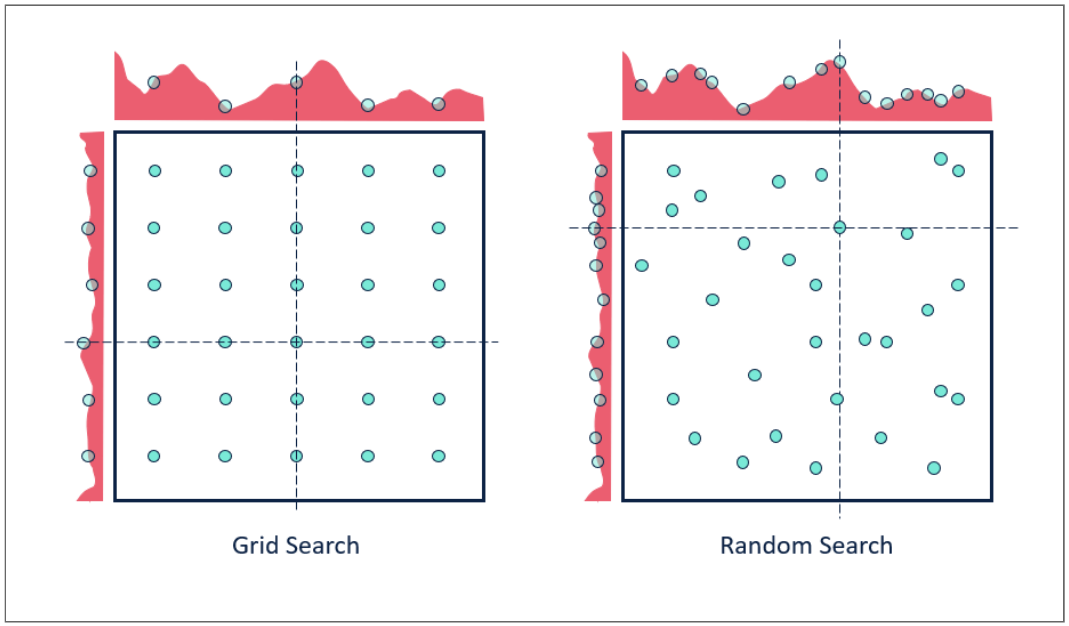


Summary Flowchart (by Andrew Ng)

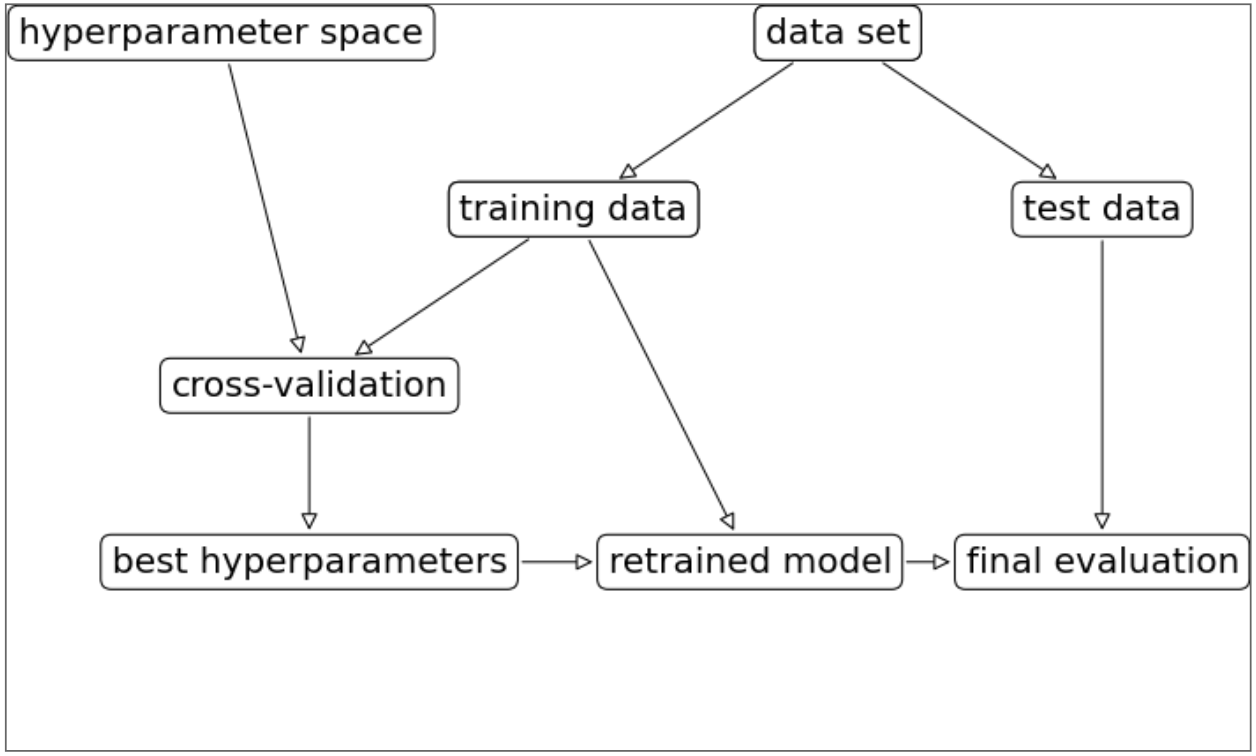


Hyperparameter tuning

- There exists a huge range of techniques to tune hyperparameters. The simplest:
    - Grid search: Choose a range of values for every hyperparameter, try every combination
        - Doesn't scale to many hyperparameters (combinatorial explosion)
    - Random search: Choose random values for all hyperparameters, iterate n times
        - Better, especially when some hyperparameters are less important
- Many more advanced techniques exist, see lecture on Automated Machine Learning



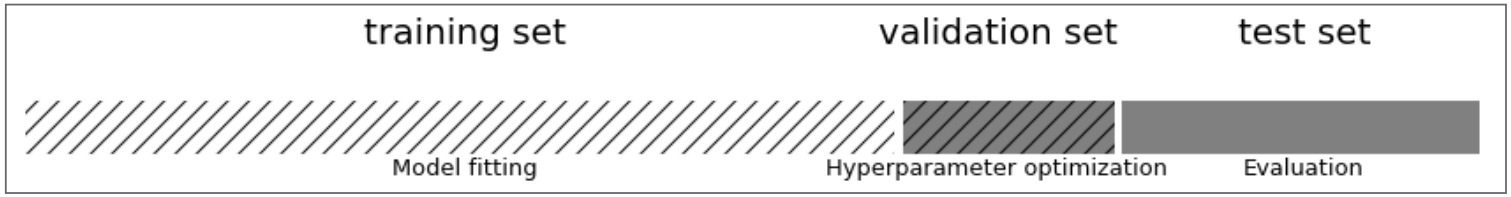
- First, split the data in training and test sets (outer split)
- Split up the training data again (inner cross-validation)
    - Generate hyperparameter configurations (e.g. random/grid search)
    - Evaluate all configurations on all inner splits, select the best one (on average)
- Retrain best configurations on full training set, evaluate on held-out test data



Nested cross-validation

- Simplest approach: single outer split and single inner split (shown below)
- Risk of over-tuning hyperparameters on specific train-test split
    - Only recommended for very large datasets
- Nested cross-validation:
    - Outer loop: split full dataset in k_1 training and test splits
    - Inner loop: split training data into k_2 train and validation sets
- This yields k_1 scores for k_1 possibly different hyperparameter settings
    - Average score is the expected performance of the tuned model
- To use the model in practice, retune on the **entire** dataset

```
hps = {'C': expon(scale=100), 'gamma': expon(scale=.1)}
scores = cross_val_score(RandomizedSearchCV(SVC(), hps, cv=3), X, y, cv=5)
```



Summary

- Split the data into training and test sets according to the application
    - Holdout only for large datasets, cross-validation for smaller ones
    - For classification, always use stratification
    - Grouped or ordered data requires special splitting
- Choose a metric that fits your application
    - E.g. precision to avoid false positives, recall to avoid false negatives
- Calibrate the decision threshold to fit your application
    - ROC curves or Precision-Recall curves can help to find a good tradeoff
- If possible, include the actual or relative costs of misclassifications
    - Class weighting, instance weighting, ROC isometrics can help
    - Be careful with imbalanced or unrepresentative datasets
- When using the predicted probabilities in applications, calibrate the models
- Always tune the most important hyperparameters
    - Manual tuning: Use insight and train-test scores for guidance
    - Hyperparameter optimization: be careful not to over-tune

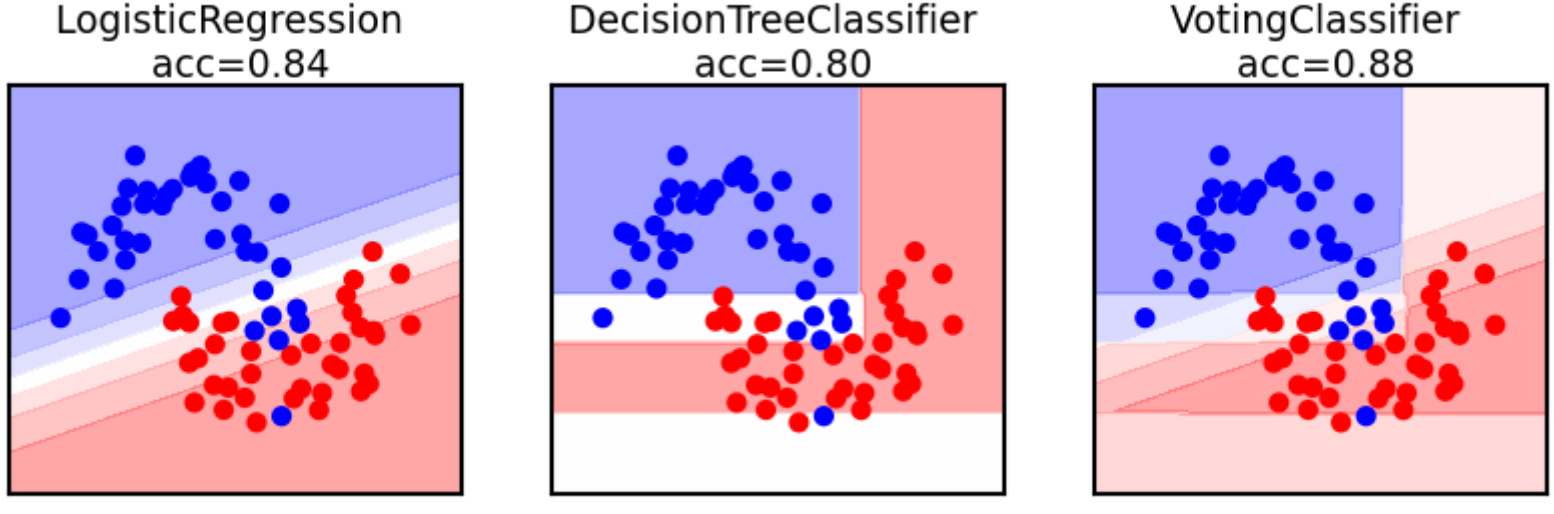
Lecture 5. Ensemble Learning

Crowd intelligence

Joaquin Vanschoren

Ensemble learning

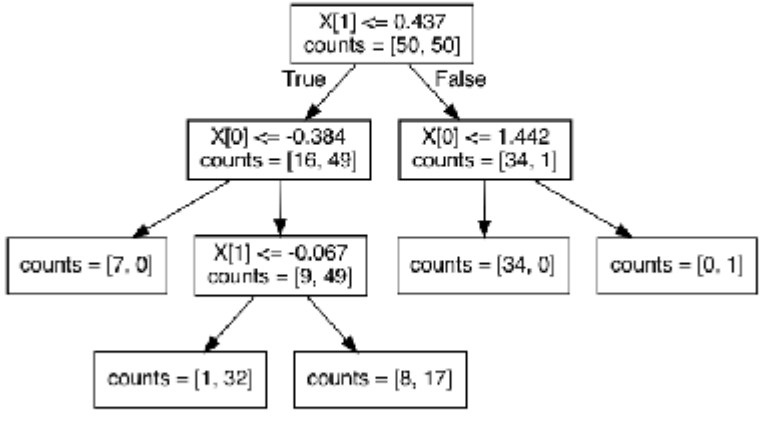
- If different models make different mistakes, can we simply average the predictions?
- Voting Classifier: gives every model a *vote* on the class label
    - Hard vote: majority class wins (class order breaks ties)
    - Soft vote: sum class probabilities $p_{m,c}$ over M models: $\underset{c}{\operatorname{argmax}} \sum_{m=1}^{M} w_c p_{m,c}$
    - Classes can get different weights w_c (default: $w_c = 1$)

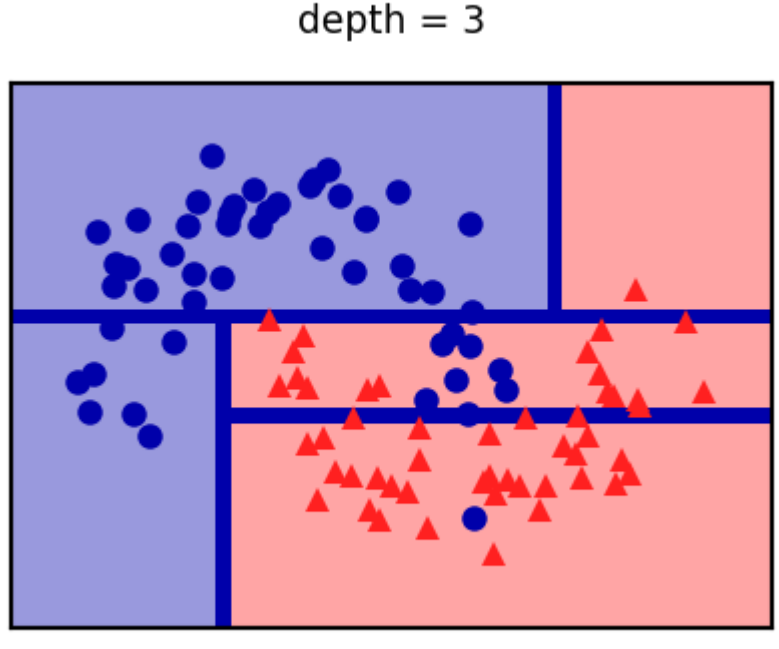


- Why does this work?
    - Different models may be good at different 'parts' of data (even if they underfit)
    - Individual mistakes can be 'averaged out' (especially if models overfit)
- Which models should be combined?
- Bias-variance analysis teaches us that we have two options:
    - If model underfits (high bias, low variance): combine with other low-variance models
        - Need to be different: 'experts' on different parts of the data
        - Bias reduction. Can be done with ***Boosting***
    - If model overfits (low bias, high variance): combine with other low-bias models
        - Need to be different: individual mistakes must be different
        - Variance reduction. Can be done with ***Bagging***
- Models must be uncorrelated but good enough (otherwise the ensemble is worse)
- We can also *learn* how to combine the predictions of different models: ***Stacking***

Decision trees (recap)

- Representation: Tree that splits data points into leaves based on tests
- Evaluation (loss): Heuristic for purity of leaves (Gini index, entropy,...)
- Optimization: Recursive, heuristic greedy search (Hunt's algorithm)
    - Consider all splits (thresholds) between adjacent data points, for every feature
    - Choose the one that yields the purest leafs, repeat

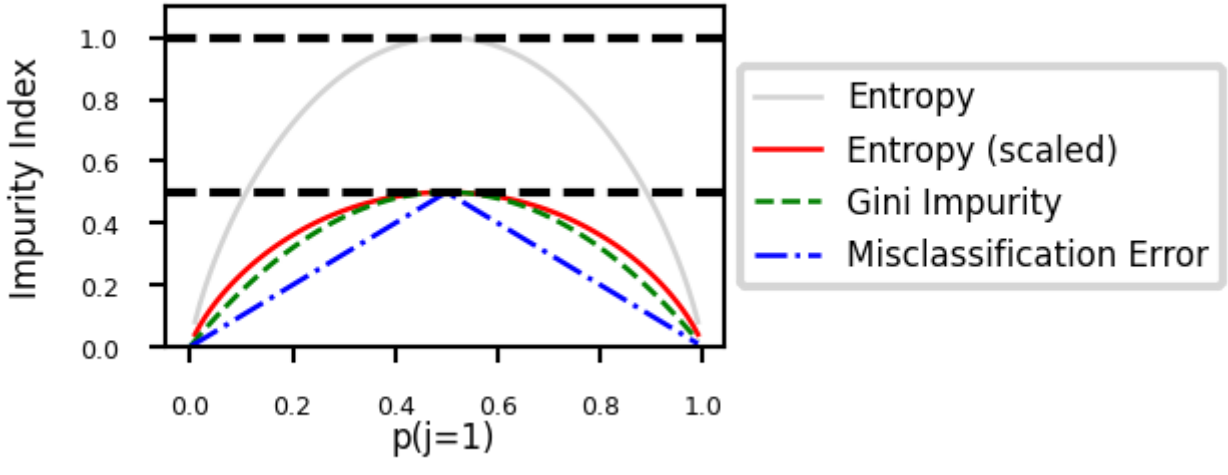




Evaluation (loss function for classification)

- Every leaf predicts a class probability $\hat{p}_c$ = the relative frequency of class c
- Leaf impurity measures (splitting criteria) for L leafs, leaf l has data X_l:
  - Gini-Index: $Gini(X_l) = \sum_{c \neq c'} \hat{p}_c \hat{p}_{c'}$
  - Entropy (more expensive): $E(X_l) = -\sum_{c \neq c'} \hat{p}_c \log_2 \hat{p}_c$
  - Best split maximizes *information gain* (idem for Gini index)

$$Gain(X, X_i) = E(X) - \sum_{l=1}^{L} \frac{|X_{i=l}|}{|X_i|} E(X_{i=l})$$

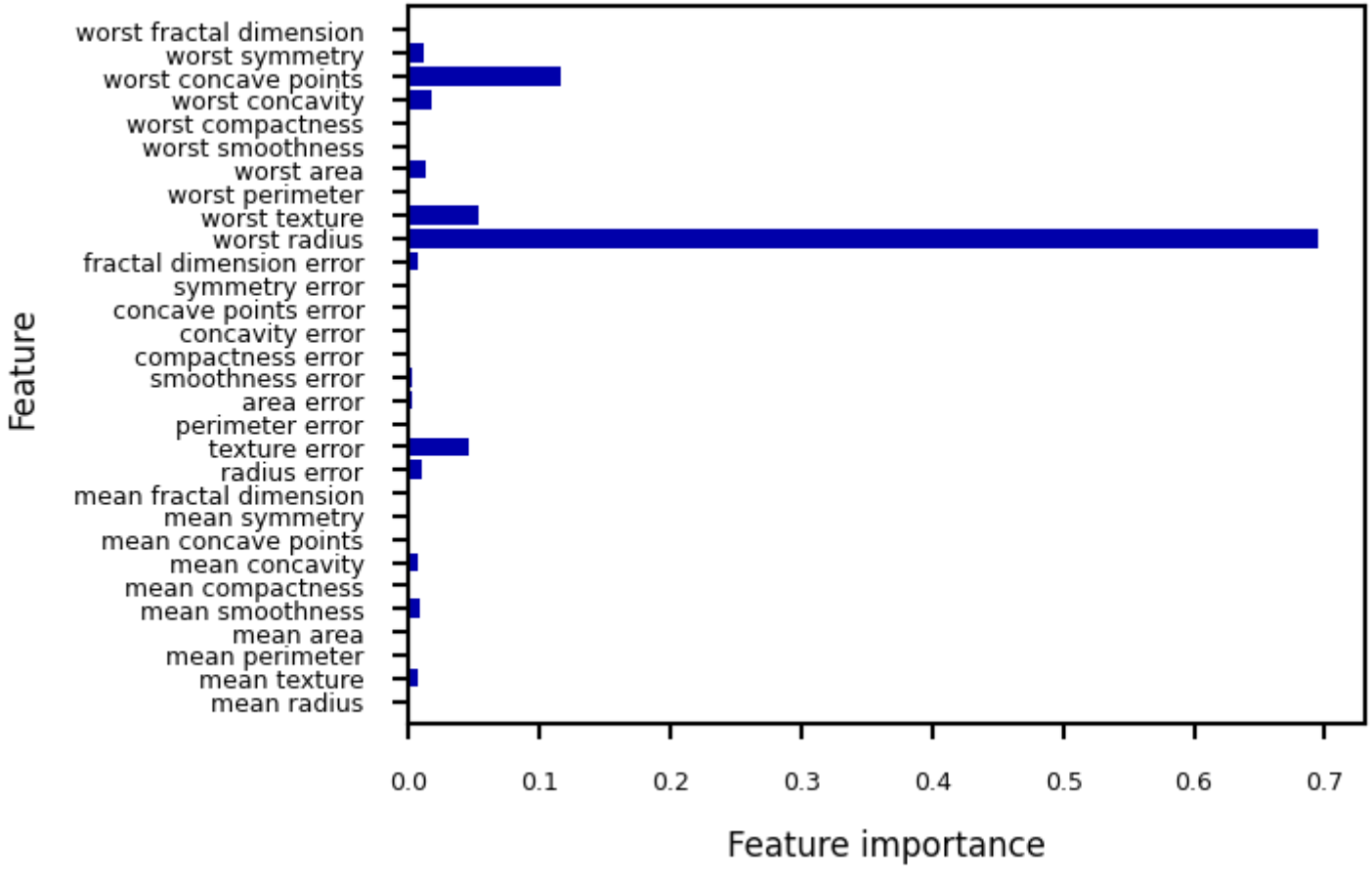


Regression trees

- Every leaf predicts the *mean* target value μ of all points in that leaf
- Choose the split that minimizes squared error of the leaves: $\sum_{x_i \in L} (y_i - \mu)^2$
- Yields non-smooth step-wise predictions, cannot extrapolate

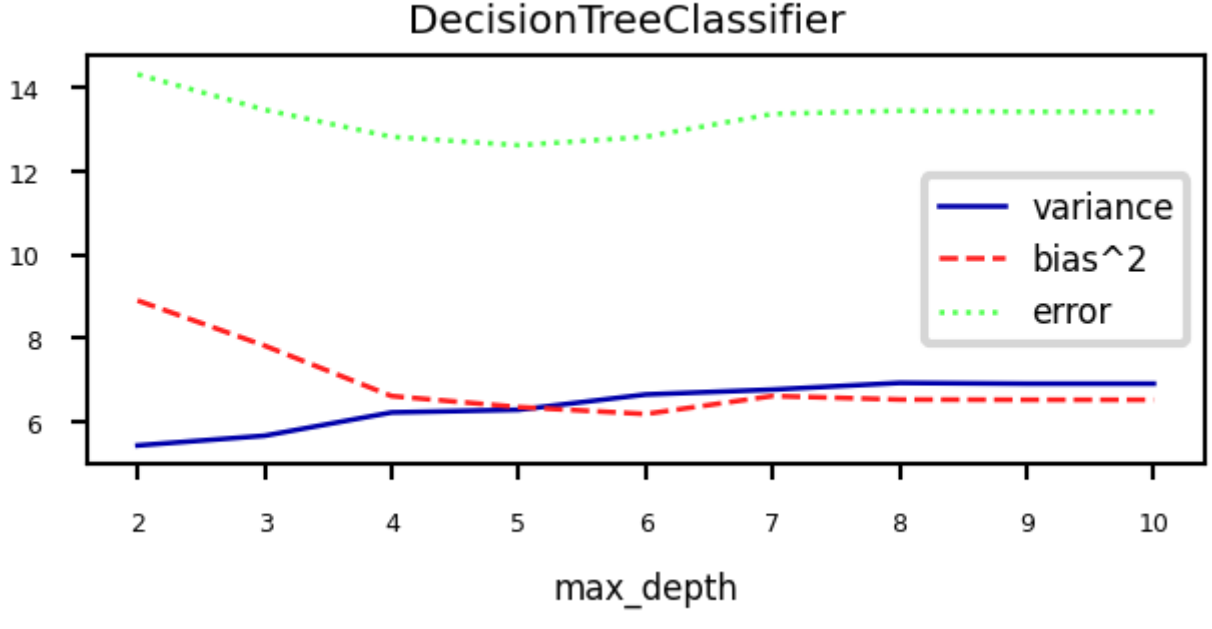
Impurity/Entropy-based feature importance

- We can measure the importance of features (to the model) based on
    - Which features we split on
    - How high up in the tree we split on them (first splits ar emore important)



Under- and overfitting

- We can easily control the (maximum) depth of the trees as a hyperparameter
- Bias-variance analysis:
    - Shallow trees have high bias but very low variance (underfitting)
    - Deep trees have high variance but low bias (overfitting)
- Because we can easily control their complexity, they are ideal for ensembling
    - Deep trees: keep low bias, reduce variance with ***Bagging***
    - Shallow trees: keep low variance, reduce bias with ***Boosting***

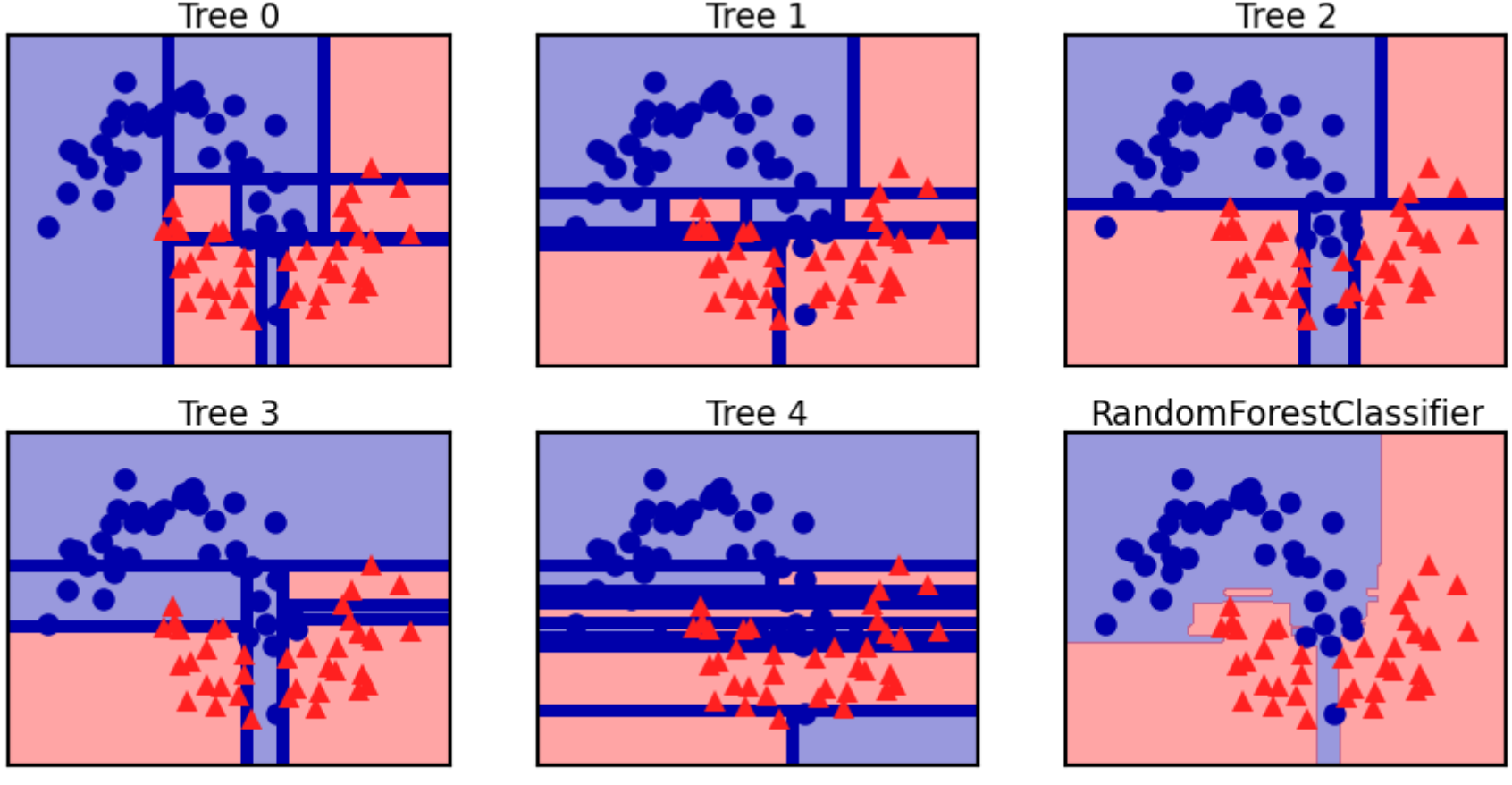


Bagging (Bootstrap Aggregating)

- Obtain different models by training the *same* model on *different training samples*
    - Reduce overfitting by averaging out individual predictions (variance reduction)
- In practice: take I bootstrap samples of your data, train a model on each bootstrap
    - Higher I: more models, more smoothing (but slower training and prediction)
- Base models should be unstable: different training samples yield different models
    - E.g. very deep decision trees, or even randomized decision trees
    - Deep Neural Networks can also benefit from bagging (deep ensembles)
- Prediction by averaging predictions of base models
    - Soft voting for classification (possibly weighted)
    - Mean value for regression
- Can produce uncertainty estimates as well
    - By combining class probabilities of individual models (or variances for regression)

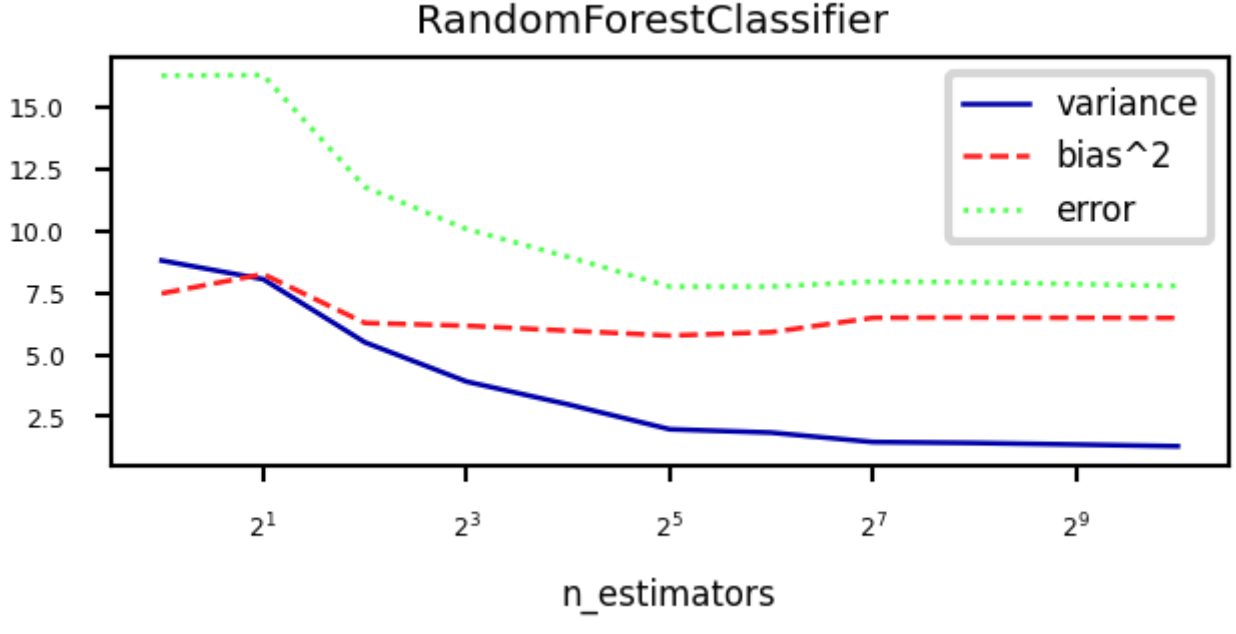
Random Forests

- Uses *randomized trees* to make models even less correlated (more unstable)
    - At every split, only consider `max_features` features, randomly selected
- Extremely randomized trees: considers 1 random threshold for random set of features (faster)



Effect on bias and variance

- Increasing the number of models (trees) decreases variance (less overfitting)
- Bias is mostly unaffected, but will increase if the forest becomes too large (oversmoothing)

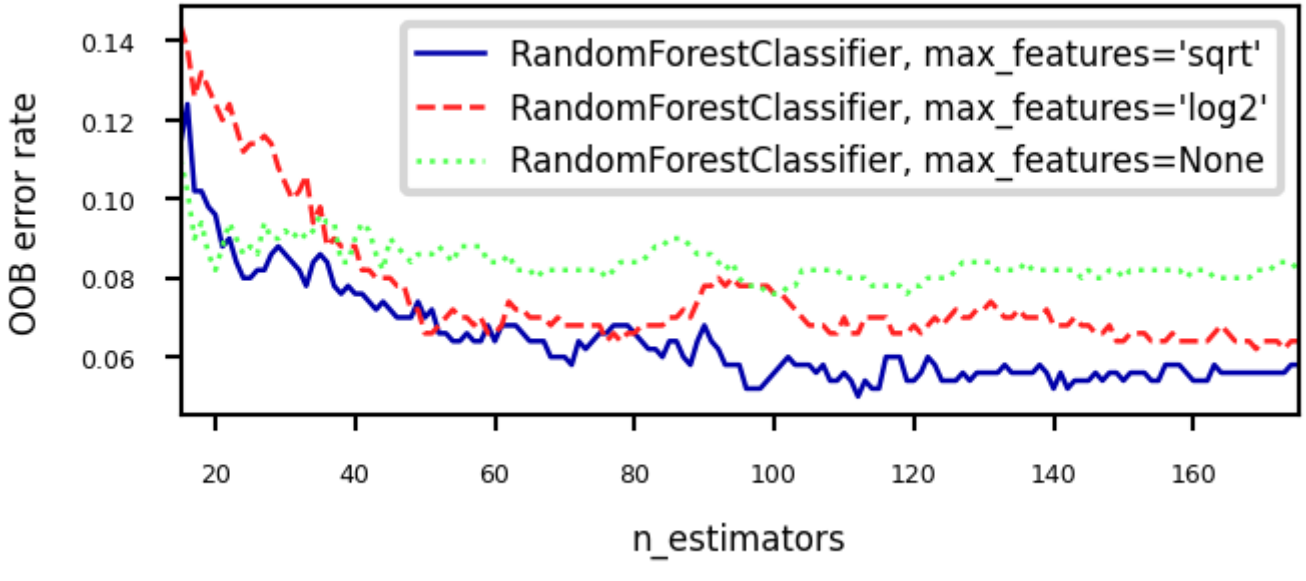


In practice

- Different implementations can be used. E.g. in scikit-learn:
    - `BaggingClassifier`: Choose your own base model and sampling procedure
    - `RandomForestClassifier`: Default implementation, many options
    - `ExtraTreesClassifier`: Uses extremely randomized trees
- Most important parameters:
    - `n_estimators` (>100, higher is better, but diminishing returns)
        - Will start to underfit (bias error component increases slightly)
    - `max_features`
        - Defaults: $sqrt(p)$ for classification, $log2(p)$ for regression
        - Set smaller to reduce space/time requirements
    - parameters of trees, e.g. `max_depth`, `min_samples_split`,...
        - Prepruning useful to reduce model size, but don't overdo it
- Easy to parallelize (set `n_jobs` to -1)
- Fix `random_state` (bootstrap samples) for reproducibility

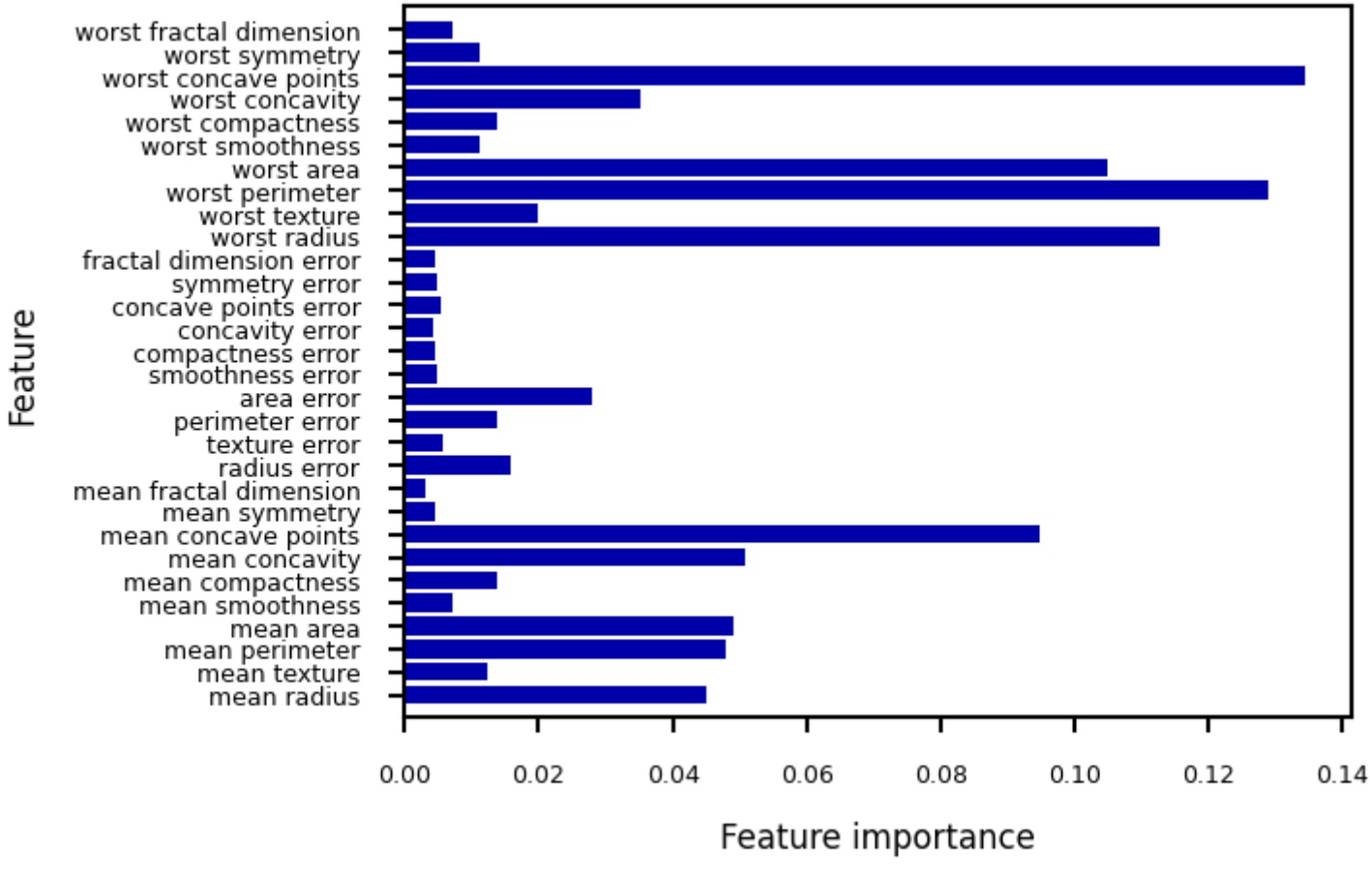
Out-of-bag error

- RandomForests don't need cross-validation: you can use the out-of-bag (OOB) error
- For each tree grown, about 33% of samples are out-of-bag (OOB)
    - Remember which are OOB samples for every model, do voting over these
- OOB error estimates are great to speed up model selection
    - As good as CV estimates, althought slightly pessimistic
- In scikit-learn: `oob_error = 1 - clf.oob_score_`



Feature importance

- RandomForests provide more reliable feature importances, based on many alternative hypotheses (trees)



Other tips

- Model calibration
    - RandomForests are poorly calibrated.
    - Calibrate afterwards (e.g. isotonic regression) if you aim to use probabilities
- Warm starting
    - Given an ensemble trained for I iterations, you can simply add more models later
    - You *warm start* from the existing model instead of re-starting from scratch
    - Can be useful to train models on new, closely related data
        - Not ideal if the data batches change over time (concept drift)
        - Boosting is more robust against this (see later)

Strength and weaknesses

- RandomForest are among most widely used algorithms:
    - Don't require a lot of tuning
    - Typically very accurate
    - Handles heterogeneous features well (trees)
    - Implictly selects most relevant features
- Downsides:
    - less interpretable, slower to train (but parallellizable)
    - don't work well on high dimensional sparse data (e.g. text)

Adaptive Boosting (AdaBoost)

- Obtain different models by *reweighting* the training data every iteration
    - Reduce underfitting by focusing on the 'hard' training examples
- Increase weights of instances misclassified by the ensemble, and vice versa
- Base models should be simple so that different instance weights lead to different models
    - Underfitting models: decision stumps (or very shallow trees)
    - Each is an 'expert' on some parts of the data
- Additive model: Predictions at iteration I are sum of base model predictions
    - In Adaboost, also the models each get a unique weight w_i

$$f_I(\mathbf{x}) = \sum_{i=1}^{I} w_i g_i(\mathbf{x})$$

- Adaboost minimizes exponential loss. For instance-weighted error ε:

$$\mathcal{L}_{Exp} = \sum_{n=1}^{N} e^{\varepsilon(f_I(\mathbf{x}))}$$

- By deriving $\frac{\partial \mathcal{L}}{\partial w_i}$ you can find that optimal $w_i = \frac{1}{2}\log(\frac{1-\varepsilon}{\varepsilon})$

AdaBoost algorithm

- Initialize sample weights: $s_{n,0} = \frac{1}{N}$
- Build a model (e.g. decision stumps) using these sample weights
- Give the *model* a weight w_i related to its weighted error rate ε

$$w_i = \lambda \log(\frac{1-\varepsilon}{\varepsilon})$$

    - Good trees get more weight than bad trees
    - Logit function maps error ε from [0,1] to weight in [-Inf,Inf] (use small minimum error)
    - Learning rate λ (shrinkage) decreases impact of individual classifiers
        - Small updates are often better but requires more iterations
- Update the sample weights
    - Increase weight of incorrectly predicted samples: $s_{n,i+1} = s_{n,i} e^{w_i}$
    - Decrease weight of correctly predicted samples: $s_{n,i+1} = s_{n,i} e^{-w_i}$
    - Normalize weights to add up to 1
- Repeat for I iterations

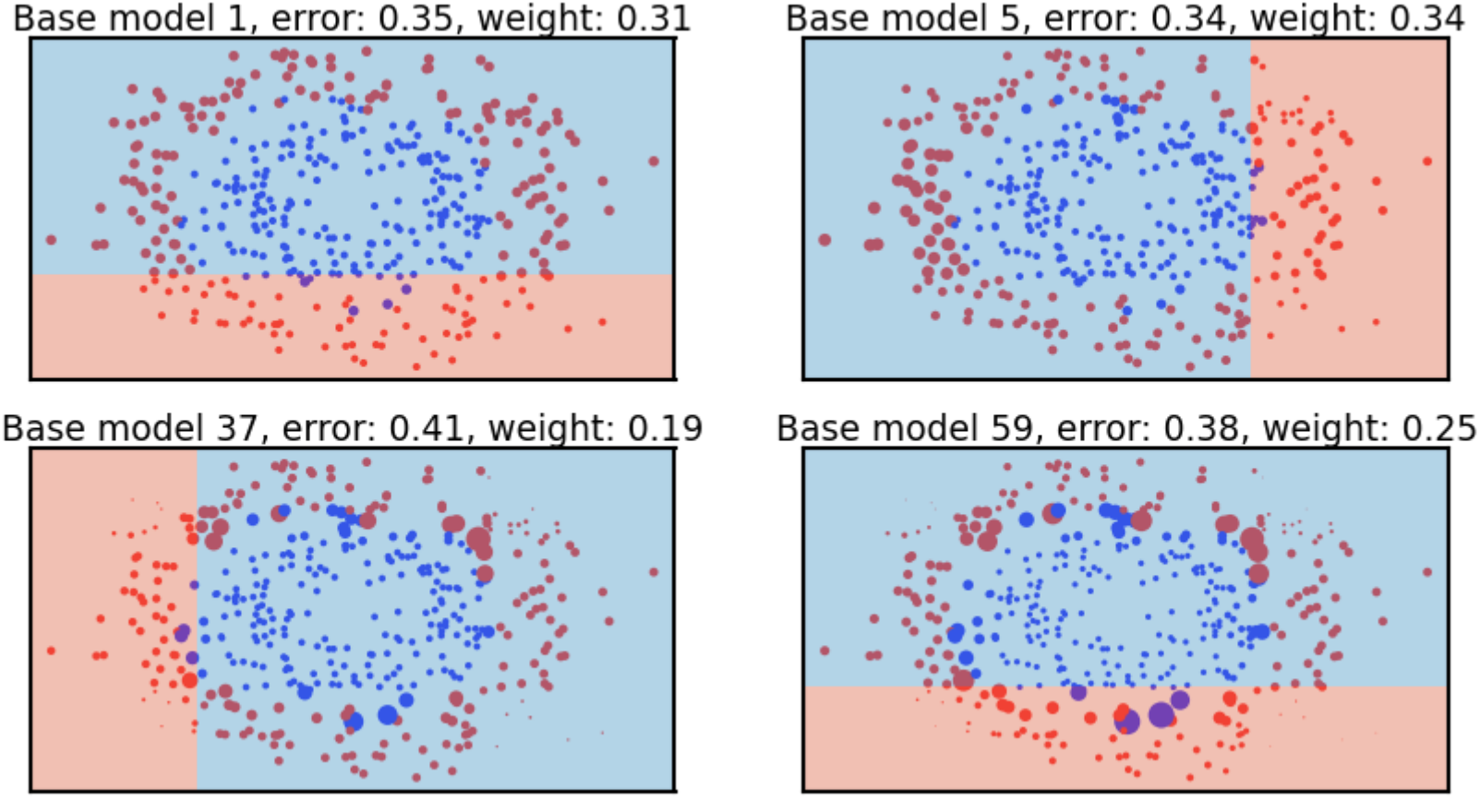
AdaBoost variants

- Discrete Adaboost: error rate ε is simply the error rate (1-Accuracy)
- Real Adaboost: ε is based on predicted class probabilities $\hat{p}_c$ (better)
- AdaBoost for regression: ε is either linear ($|y_i - \hat{y}_i|$), squared ($(y_i - \hat{y}_i)^2$), or exponential loss
- GentleBoost: adds a bound on model weights w_i
- LogitBoost: Minimizes logistic loss instead of exponential loss

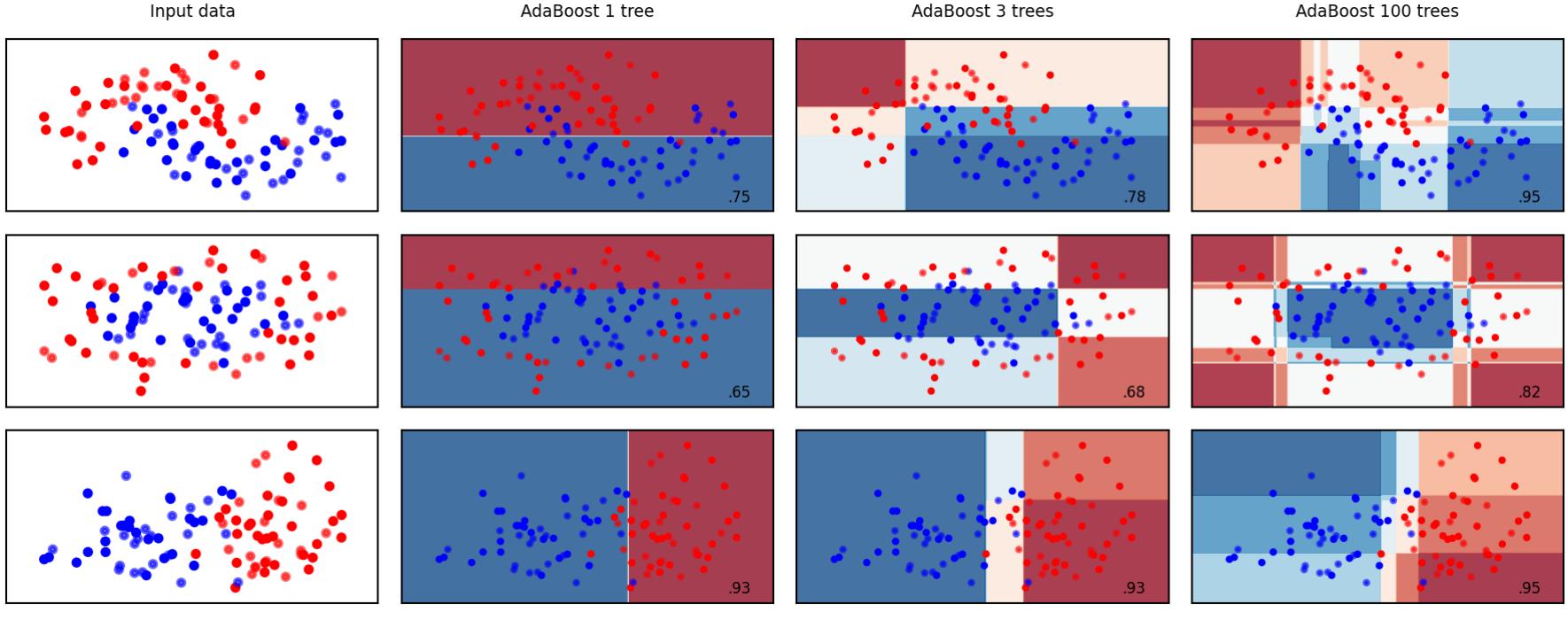
$$\mathcal{L}_{Logistic} = \sum_{n=1}^{N} log(1 + e^{\varepsilon(f_I(\mathbf{x}))})$$

Adaboost in action

- Size of the samples represents sample weight
- Background shows the latest tree's predictions

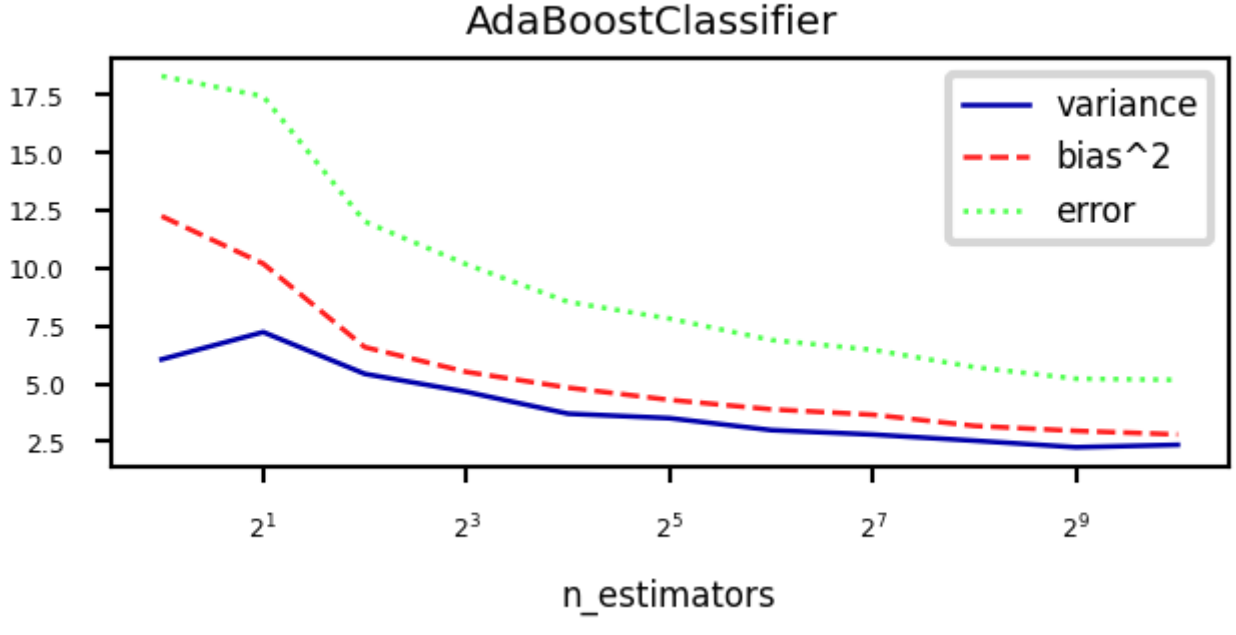


Examples


Input data
AdaBoost 1 tree
AdaBoost 3 trees
AdaBoost 100 trees
.75
.78
.95
.65
.68
.82
.93
.93
.95

Bias-Variance analysis

- AdaBoost reduces bias (and a little variance)
    - Boosting is a *bias reduction* technique
- Boosting too much will eventually increase variance



Gradient Boosting

- Ensemble of models, each fixing the remaining mistakes of the previous ones
    - Each iteration, the task is to predict the *residual error* of the ensemble
- Additive model: Predictions at iteration I are sum of base model predictions
    - Learning rate (or *shrinkage*) η: small updates work better (reduces variance)

$$f_I(\mathbf{x}) = g_0(\mathbf{x}) + \sum_{i=1}^{I} \eta \cdot g_i(\mathbf{x}) = f_{I-1}(\mathbf{x}) + \eta \cdot g_I(\mathbf{x})$$

- The *pseudo-residuals* r_i are computed according to differentiable loss function
    - E.g. least squares loss for regression and log loss for classification
    - Gradient descent: *predictions* get updated step by step until convergence

$$g_i(\mathbf{x}) \approx r_i = -\frac{\partial \mathcal{L}(y_i, f_{i-1}(x_i))}{\partial f_{i-1}(x_i)}$$

- Base models g_i should be low variance, but flexible enough to predict residuals accurately
    - E.g. decision trees of depth 2-5

Gradient Boosting Trees (Regression)

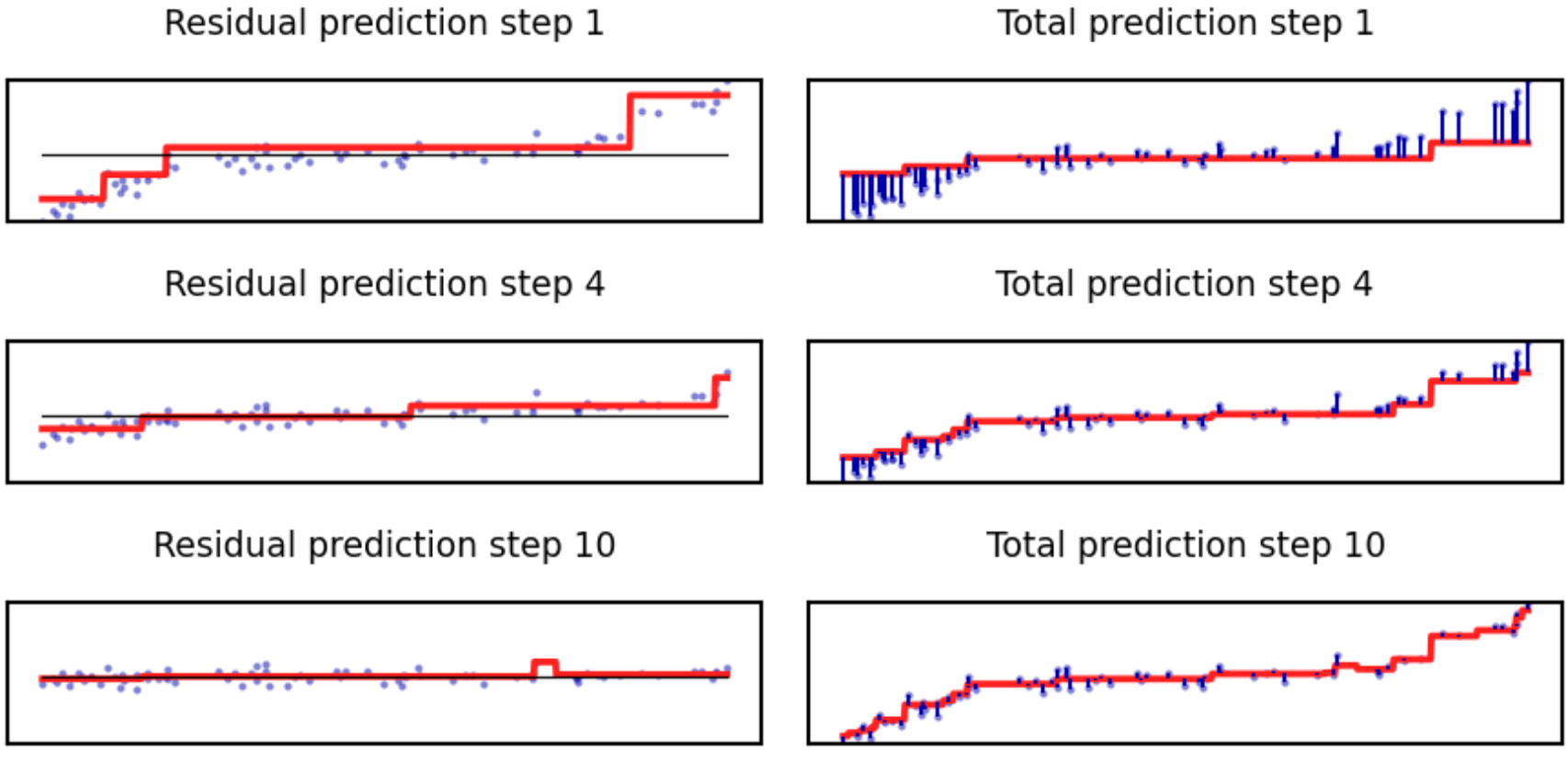
- Base models are regression trees, loss function is square loss: $\mathcal{L} = \frac{1}{2}(y_i - \hat{y}_i)^2$
- The pseudo-residuals are simply the prediction errors for every sample:

$$r_i = -\frac{\partial \mathcal{L}}{\partial \hat{y}} = -2 * \frac{1}{2}(y_i - \hat{y}_i) * (-1) = y_i - \hat{y}_i$$

- Initial model g_0 simply predicts the mean of y
- For iteration $m = 1..M$:
    - For all samples i=1..n, compute pseudo-residuals $r_i = y_i - \hat{y}_i$
    - Fit a new regression tree model $g_m(\mathbf{x})$ to r_i
        - In $g_m(\mathbf{x})$, each leaf predicts the mean of all its values
    - Update ensemble predictions $\hat{y} = g_0(\mathbf{x}) + \sum_{m=1}^{M} \eta \cdot g_m(\mathbf{x})$
- Early stopping (optional): stop when performance on validation set does not improve for nr iterations

Gradient Boosting Regression in action

- Residuals quickly drop to (near) zero

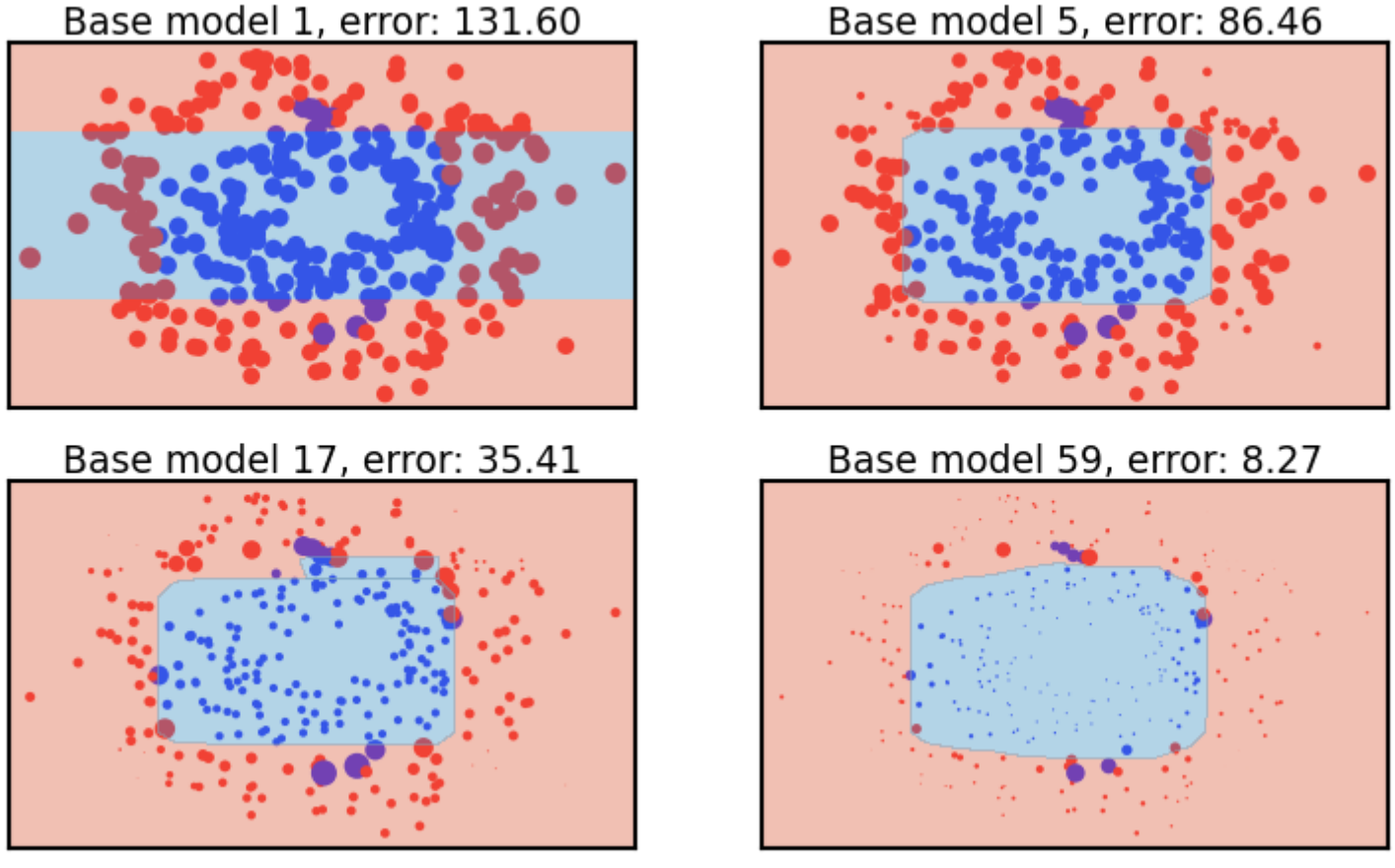


GradientBoosting Algorithm (Classification)

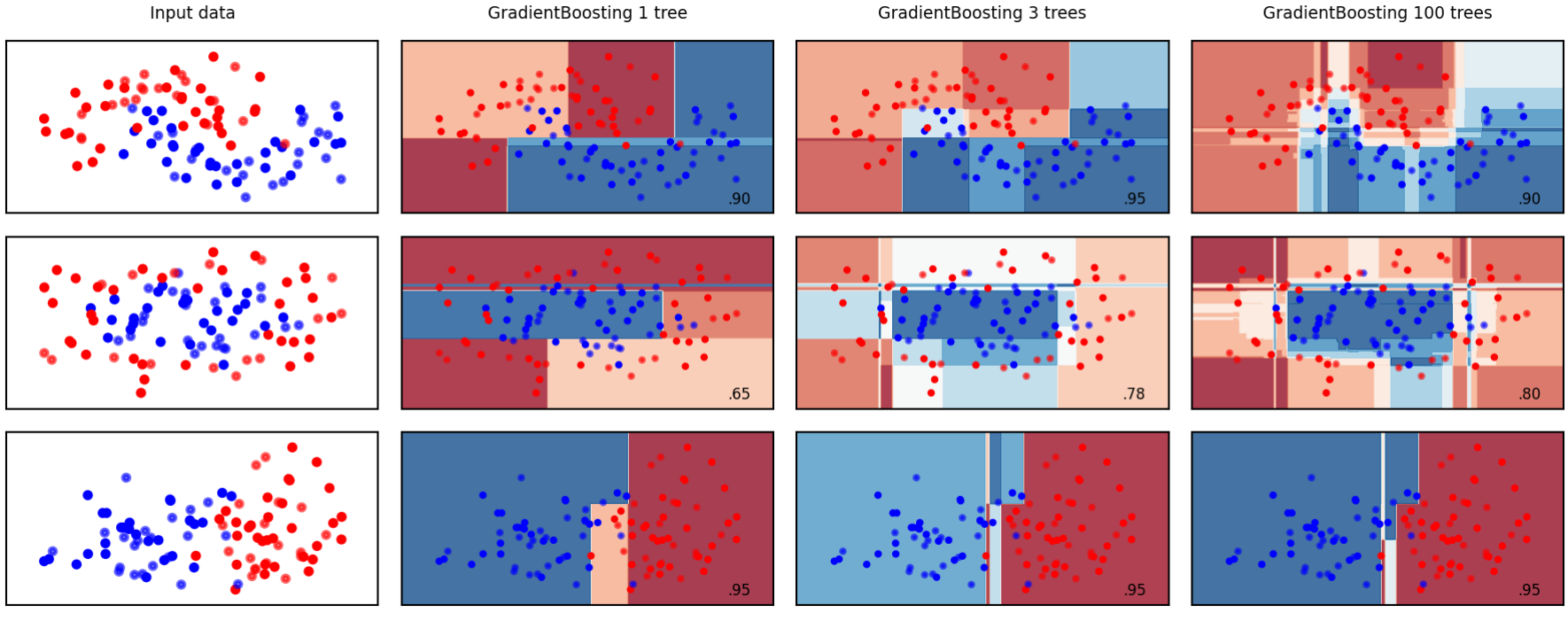
- Base models are *regression* trees, predict probability of positive class p
    - For multi-class problems, train one tree per class
- Use (binary) log loss, with true class $y_i \in 0, 1$:
  $\mathcal{L}_{log} = -\sum_{i=1}^{N} \left[y_i log(p_i) + (1 - y_i) log(1 - p_i) \right]$
- The pseudo-residuals are simply the difference between true class and predicted p:
$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \frac{\partial \mathcal{L}}{\partial log(p_i)} = y_i - p_i$$
- Initial model g_0 predicts $p = log(\frac{\#positives}{\#negatives})$
- For iteration $m = 1..M$:
    - For all samples i=1..n, compute pseudo-residuals $r_i = y_i - p_i$
    - Fit a new regression tree model $g_m(\mathbf{x})$ to r_i
        - In $g_m(\mathbf{x})$, each leaf predicts $\frac{\sum_i r_i}{\sum_i p_i(1-p_i)}$
    - Update ensemble predictions $\hat{y} = g_0(\mathbf{x}) + \sum_{m=1}^{M} \eta \cdot g_m(\mathbf{x})$
- Early stopping (optional): stop when performance on validation set does not improve for nr iterations

Gradient Boosting Classification in action

- Size of the samples represents the residual weights: most quickly drop to (near) zero

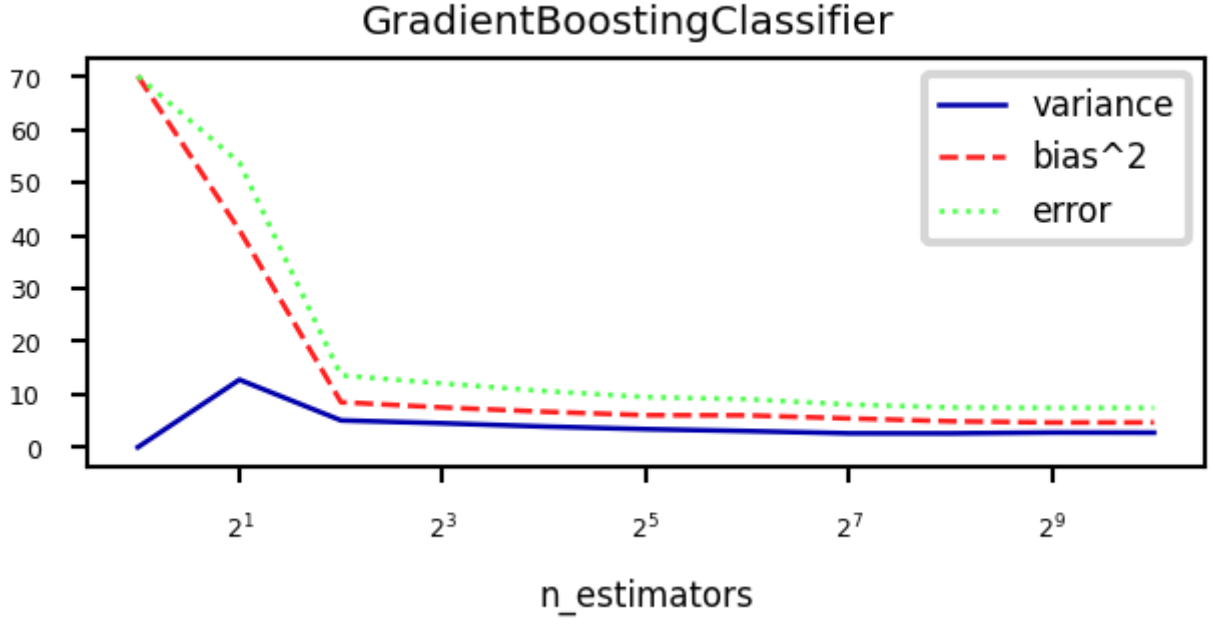


Examples


Input data
GradientBoosting 1 tree
GradientBoosting 3 trees
GradientBoosting 100 trees
.90
.95
.90
.65
.78
.80
.95
.95
.95

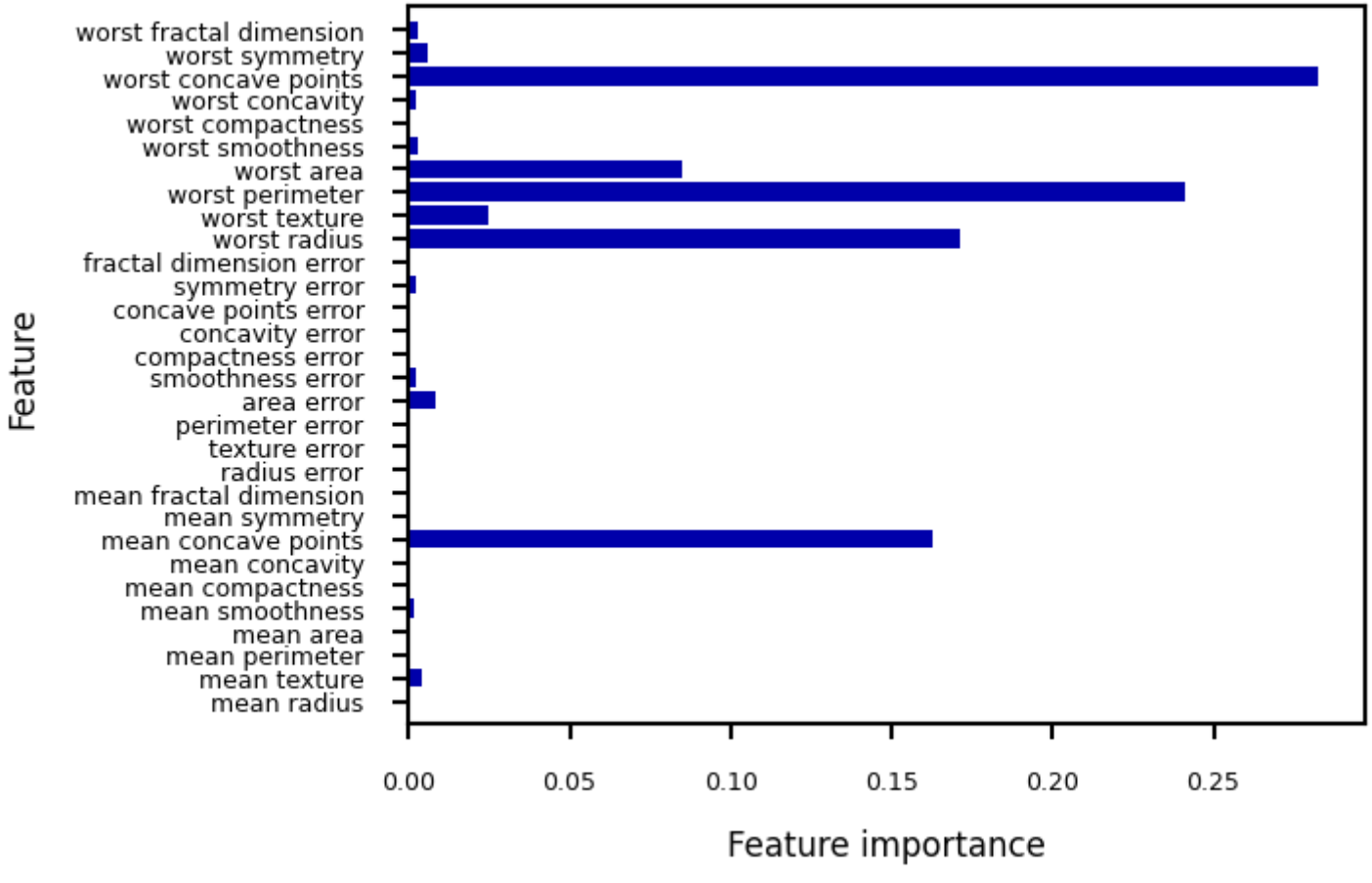
Bias-variance analysis

- Gradient Boosting is very effective at reducing bias error
- Boosting too much will eventually increase variance



Feature importance

- Gradient Boosting also provide feature importances, based on many trees
- Compared to RandomForests, the trees are smaller, hence more features have zero importance



Gradient Boosting: strengths and weaknesses

- Among the most powerful and widely used models
- Work well on heterogeneous features and different scales
- Typically better than random forests, but requires more tuning, longer training
- Does not work well on high-dimensional sparse data

Main hyperparameters:

- `n_estimators` : Higher is better, but will start to overfit
- `learning_rate` : Lower rates mean more trees are needed to get more complex models
    - Set `n_estimators` as high as possible, then tune `learning_rate`
    - Or, choose a `learning_rate` and use early stopping to avoid overfitting
- `max_depth` : typically kept low (<5), reduce when overfitting
- `max_features` : can also be tuned, similar to random forests
- `n_iter_no_change` : early stopping: algorithm stops if improvement is less than a certain tolerance `tol` for more than `n_iter_no_change` iterations.

Extreme Gradient Boosting (XGBoost)

- Faster version of gradient boosting: allows more iterations on larger datasets
- Normal regression trees: split to minimize squared loss of leaf predictions
    - XGBoost trees only fit residuals: split so that residuals in leaf are more *similar*
- Don't evaluate every split point, only q *quantiles* per feature (binning)
    - q is hyperparameter (`sketch_eps` , default 0.03)
- For large datasets, XGBoost uses *approximate quantiles*
    - Can be parallelized (multicore) by chunking the data and combining histograms of data
    - For classification, the quantiles are weighted by $p(1-p)$
- Gradient descent sped up by using the second derivative of the loss function
- Strong regularization by pre-pruning the trees
- Column and row are randomly subsampled when computing splits
- Support for out-of-core computation (data compression in RAM, sharding,...)

XGBoost in practice

- Not part of scikit-learn, but `HistGradientBoostingClassifier` is similar
    - binning, multicore,...
- The `xgboost` python package is sklearn-compatible
    - Install separately, `conda install -c conda-forge xgboost`
    - Allows learning curve plotting and warm-starting
- Further reading:
    - XGBoost Documentation
    - Paper
    - Video

LightGBM

Another fast boosting technique

- Uses *gradient-based sampling*
    - use all instances with large gradients/residuals (e.g. 10% largest)
    - randomly sample instances with small gradients, ignore the rest
    - intuition: samples with small gradients are already well-trained.
    - requires adapted information gain criterion
- Does smarter encoding of categorical features

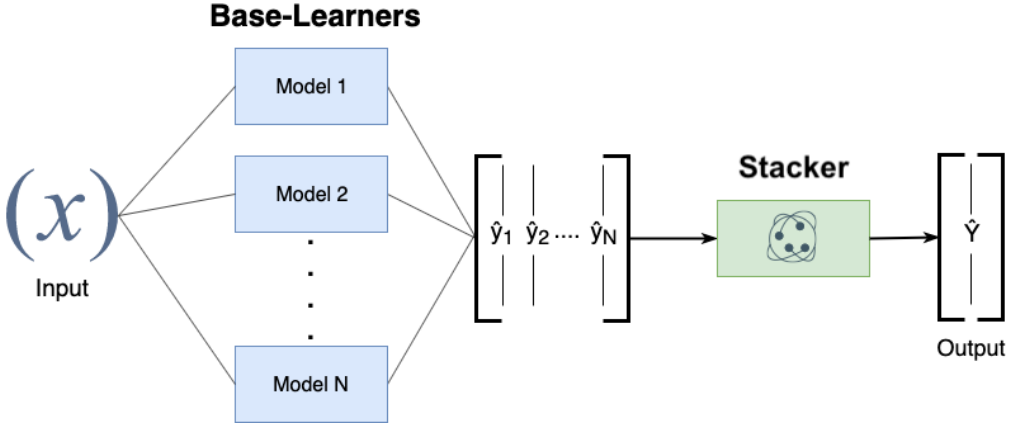
CatBoost

Another fast boosting technique

- Optimized for categorical variables
    - Uses bagged and smoothed version of target encoding
- Uses symmetric trees: same split for all nodes on a given level aka
    - Can be much faster
- Allows monotonicity constraints for numeric features
    - Model must be be a non-decreasing function of these features
- Lots of tooling (e.g. GPU training)

Stacking

- Choose M different base-models, generate predictions
- Stacker (meta-model) learns mapping between predictions and correct label
    - Can also be repeated: multi-level stacking
    - Popular stackers: linear models (fast) and gradient boosting (accurate)
- Cascade stacking: adds base-model predictions as extra features
- Models need to be sufficiently different, be experts at different parts of the data
- Can be *very* accurate, but also very slow to predict



Other ensembling techniques

- Hyper-ensembles: same basic model but with different hyperparameter settings
    - Can combine overfitted and underfitted models
- Deep ensembles: ensembles of deep learning models
- Bayes optimal classifier: ensemble of all possible models (largely theoretic)
- Bayesian model averaging: weighted average of probabilistic models, weighted by their posterior probabilities
- Cross-validation selection: does internal cross-validation to select best of M models
- Any combination of different ensembling techniques

Algorithm overview

Name	Representation	Loss function	Optimization	Regularization
Classification trees	Decision tree	Entropy / Gini index	Hunt's algorithm	Tree depth,...
Regression trees	Decision tree	Square loss	Hunt's algorithm	Tree depth,...
RandomForest	Ensemble of randomized trees	Entropy / Gini / Square	(Bagging)	Number/depth of trees,...
AdaBoost	Ensemble of stumps	Exponential loss	Greedy search	Number/depth of trees,...
GradientBoostingRegression	Ensemble of regression trees	Square loss	Gradient descent	Number/depth of trees,...
GradientBoostingClassification	Ensemble of regression trees	Log loss	Gradient descent	Number/depth of trees,...
XGBoost, LightGBM, CatBoost	Ensemble of XGBoost trees	Square/log loss	2nd order gradients	Number/depth of trees,...
Stacking	Ensemble of heterogeneous models	/	/	Number of models,...

Summary

- Ensembles of voting classifiers improve performance
    - Which models to choose? Consider bias-variance tradeoffs!
- Bagging / RandomForest is a variance-reduction technique
    - Build many high-variance (overfitting) models on random data samples
        - The more different the models, the better
    - Aggregation (soft voting) over many models reduces variance
        - Diminishing returns, over-smoothing may increase bias error
    - Parallellizes easily, doesn't require much tuning
- Boosting is a bias-reduction technique
    - Build low-variance models that correct each other's mistakes
        - By reweighting misclassified samples: AdaBoost
        - By predicting the residual error: Gradient Boosting
    - Additive models: predictions are sum of base-model predictions
        - Can drive the error to zero, but risk overfitting
    - Doesn't parallelize easily. Slower to train, much faster to predict.
        - XGBoost,LightGBM,... are fast and offer some parallellization
- Stacking: learn how to combine base-model predictions
    - Base-models still have to be sufficiently different

Lecture 6. Data preprocessing

Real-world machine learning pipelines

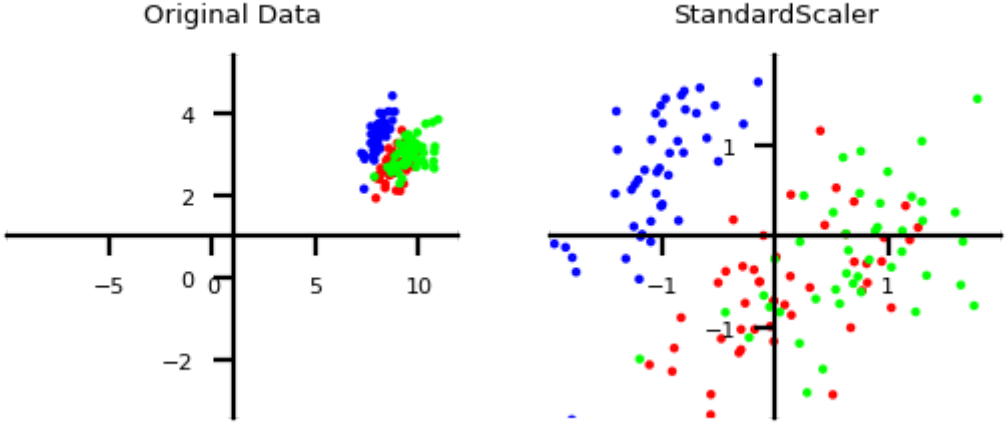
Joaquin Vanschoren

Data transformations

- Machine learning models make a lot of assumptions about the data
- In reality, these assumptions are often violated
- We build *pipelines* that *transform* the data before feeding it to the learners
    - Scaling (or other numeric transformations)
    - Encoding (convert categorical features into numerical ones)
    - Automatic feature selection
    - Feature engineering (e.g. binning, polynomial features,...)
    - Handling missing data
    - Handling imbalanced data
    - Dimensionality reduction (e.g. PCA)
    - Learned embeddings (e.g. for text)
- Seek the best combinations of transformations and learning methods
    - Often done empirically, using cross-validation
    - Make sure that there is no data leakage during this process!

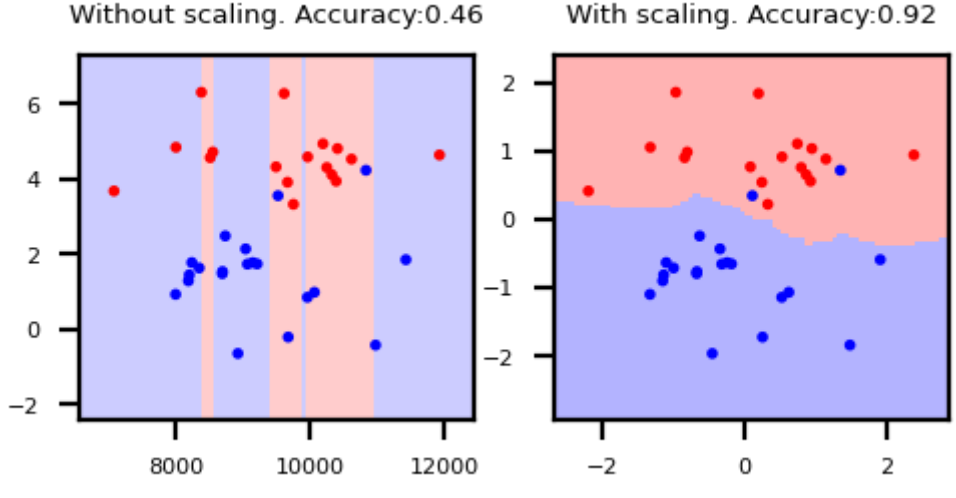
Scaling

- Use when different numeric features have different scales (different range of values)
    - Features with much higher values may overpower the others
- Goal: bring them all within the same range
- Different methods exist



Why do we need scaling?

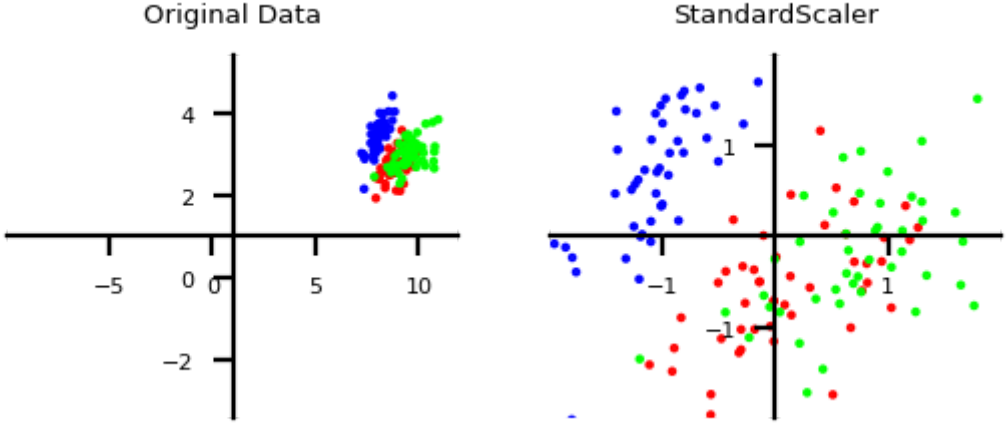
- KNN: Distances depend mainly on feature with larger values
- SVMs: (kernelized) dot products are also based on distances
- Linear model: Feature scale affects regularization
    - Weights have similar scales, more interpretable



Standard scaling (standardization)

- Generally most useful, assumes data is more or less normally distributed
- Per feature, subtract the mean value μ, scale by standard deviation σ
- New feature has $\mu = 0$ and $\sigma = 1$, values can still be arbitrarily large

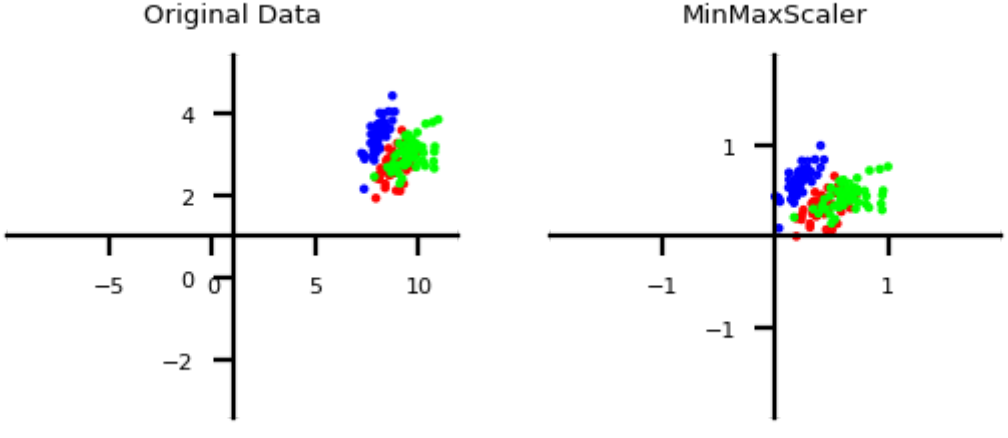
$$\mathbf{x}_{new} = \frac{\mathbf{x} - \mu}{\sigma}$$



Min-max scaling

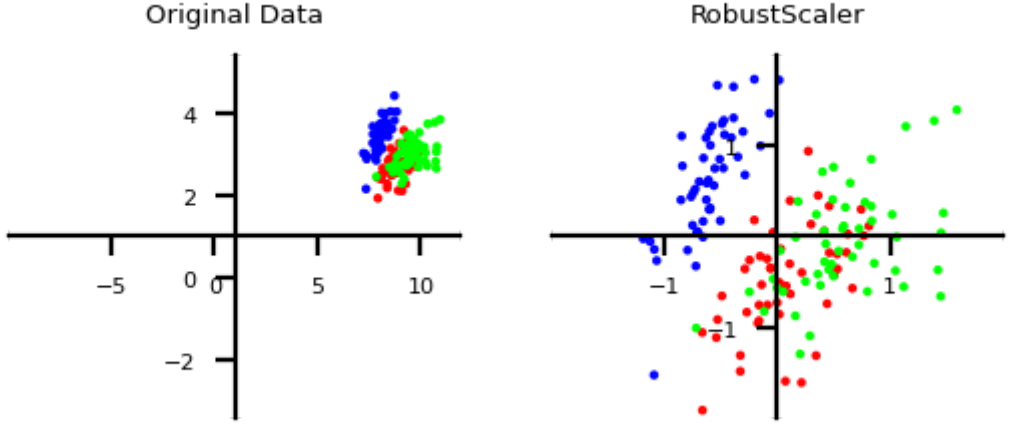
- Scales all features between a given min and max value (e.g. 0 and 1)
- Makes sense if min/max values have meaning in your data
- Sensitive to outliers

$$\mathbf{x}_{new} = \frac{\mathbf{x} - x_{min}}{x_{max} - x_{min}} \cdot (max - min) + min$$



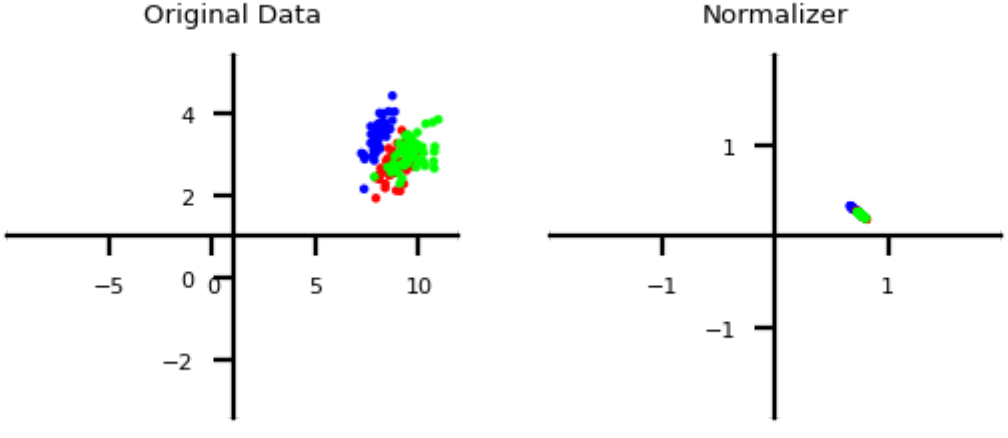
Robust scaling

- Subtracts the median, scales between quantiles q_{25} and q_{75}
- New feature has median 0, $q_{25} = -1$ and $q_{75} = 1$
- Similar to standard scaler, but ignores outliers



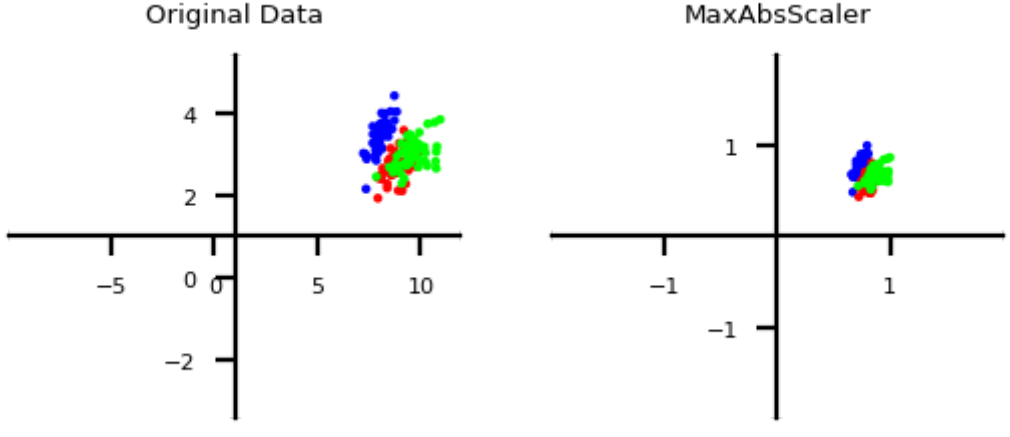
Normalization

- Makes sure that feature values of each point (each row) sum up to 1 (L1 norm)
    - Useful for count data (e.g. word counts in documents)
- Can also be used with L2 norm (sum of squares is 1)
    - Useful when computing distances in high dimensions
    - Normalized Euclidean distance is equivalent to cosine similarity



Maximum Absolute scaler

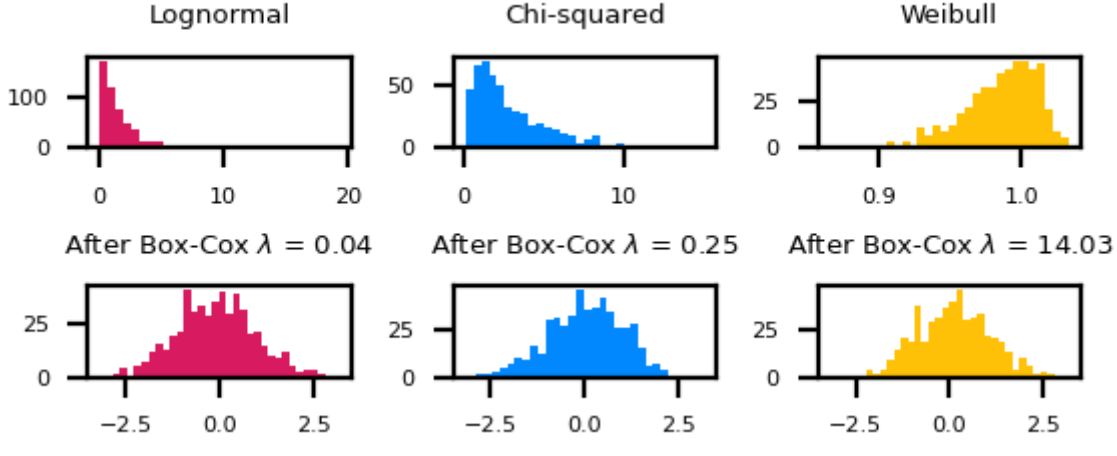
- For sparse data (many features, but few are non-zero)
    - Maintain sparseness (efficient storage)
- Scales all values so that maximum absolute value is 1
- Similar to Min-Max scaling without changing 0 values



Power transformations

- Some features follow certain distributions
    - E.g. number of twitter followers is log-normal distributed
- Box-Cox transformations transform these to normal distributions (λ is fitted)
    - Only works for positive values, use Yeo-Johnson otherwise

$$bc_\lambda(x) = \begin{cases} log(x) & \lambda = 0 \\ \frac{x^\lambda - 1}{\lambda} & \lambda \neq 0 \end{cases}$$



Categorical feature encoding

- Many algorithms can only handle numeric features, so we need to encode the categorical ones

	boro	salary	vegan
0	Manhattan	103	0
1	Queens	89	0
2	Manhattan	142	0
3	Brooklyn	54	1
4	Brooklyn	63	1
5	Bronx	219	0

Ordinal encoding

- Simply assigns an integer value to each category in the order they are encountered
- Only really useful if there exist a natural order in categories
    - Model will consider one category to be 'higher' or 'closer' to another

	boro	boro_ordinal	salary
0	Manhattan	2	103
1	Queens	3	89
2	Manhattan	2	142
3	Brooklyn	1	54
4	Brooklyn	1	63
5	Bronx	0	219

One-hot encoding (dummy encoding)

- Simply adds a new 0/1 feature for every category, having 1 (hot) if the sample has that category
- Can explode if a feature has lots of values, causing issues with high dimensionality
- What if test set contains a new category not seen in training data?
    - Either ignore it (just use all 0's in row), or handle manually (e.g. resample)

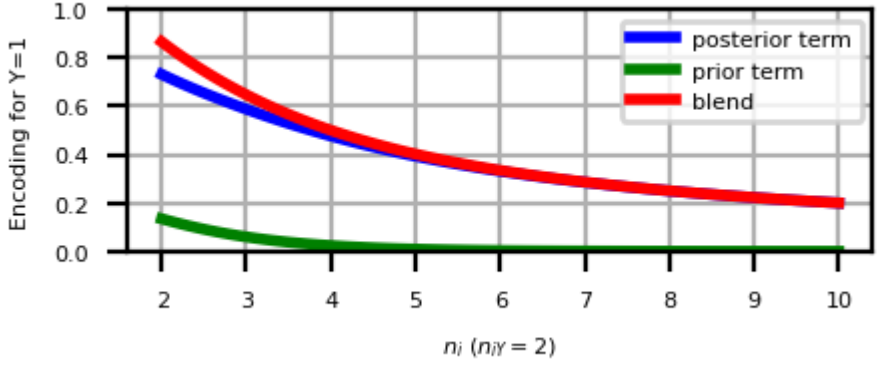
	boro	boro_Bronx	boro_Brooklyn	boro_Manhattan	boro_Queens	salary
0	Manhattan	0	0	1	0	103
1	Queens	0	0	0	1	89
2	Manhattan	0	0	1	0	142
3	Brooklyn	0	1	0	0	54
4	Brooklyn	0	1	0	0	63
5	Bronx	1	0	0	0	219

Target encoding

- Value close to 1 if category correlates with class 1, close to 0 if correlates with class 0
- Preferred when you have lots of category values. It only creates one new feature per class
- Blends posterior probability of the target $\frac{n_{iY}}{n_i}$ and prior probability $\frac{n_Y}{n}$.
  - n_{iY}: nr of samples with category i and class Y=1, n_i: nr of samples with category i
  - Blending: gradually decrease as you get more examples of category i and class Y=0

$$Enc(i) = \frac{1}{1+e^{-(n_i-1)}}\frac{n_{iY}}{n_i} + (1 - \frac{1}{1+e^{-(n_i-1)}})\frac{n_Y}{n}$$

  - Same for regression, using $\frac{n_{iY}}{n_i}$: average target value with category i, $\frac{n_Y}{n}$: overall mean



Example

- For Brooklyn, $n_{iY} = 2, n_i = 2, n_Y = 2, n = 6$
- Would be closer to 1 if there were more examples, all with label 1

$$Enc(Brooklyn) = \frac{1}{1 + e^{-1}} \frac{2}{2} + (1 - \frac{1}{1 + e^{-1}}) \frac{2}{6} = 0,82$$

- Note: the implementation used here sets $Enc(i) = \frac{n_Y}{n}$ when $n_{iY} = 1$

	boro	boro_encoded	salary	vegan
0	Manhattan	0.089647	103	0
1	Queens	0.333333	89	0
2	Manhattan	0.089647	142	0
3	Brooklyn	0.820706	54	1
4	Brooklyn	0.820706	63	1
5	Bronx	0.333333	219	0

In practice (scikit-learn)

- Ordinal encoding and one-hot encoding are implemented in scikit-learn
    - dtype defines that the output should be an integer

```
ordinal_encoder = OrdinalEncoder(dtype=int)
one_hot_encoder = OneHotEncoder(dtype=int)
```

- Target encoding is available in `category_encoders`
    - scikit-learn compatible
    - Also includes other, very specific encoders

```
target_encoder = TargetEncoder(return_df=True)
```

- All encoders (and scalers) follow the `fit-transform` paradigm
    - `fit` prepares the encoder, `transform` actually encodes the features
    - We'll discuss this next

```
encoder.fit(X, y)
X_encoded = encoder.transform(X,y)
```

Applying data transformations

- Data transformations should always follow a fit-predict paradigm
    - Fit the transformer on the training data only
        - E.g. for a standard scaler: record the mean and standard deviation
    - Transform (e.g. scale) the training data, then train the learning model
    - Transform (e.g. scale) the test data, then evaluate the model
- Only scale the input features (X), not the targets (y)
- If you fit and transform the whole dataset before splitting, you get data leakage
    - You have looked at the test data before training the model
    - Model evaluations will be misleading
- If you fit and transform the training and test data separately, you distort the data
    - E.g. training and test points are scaled differently

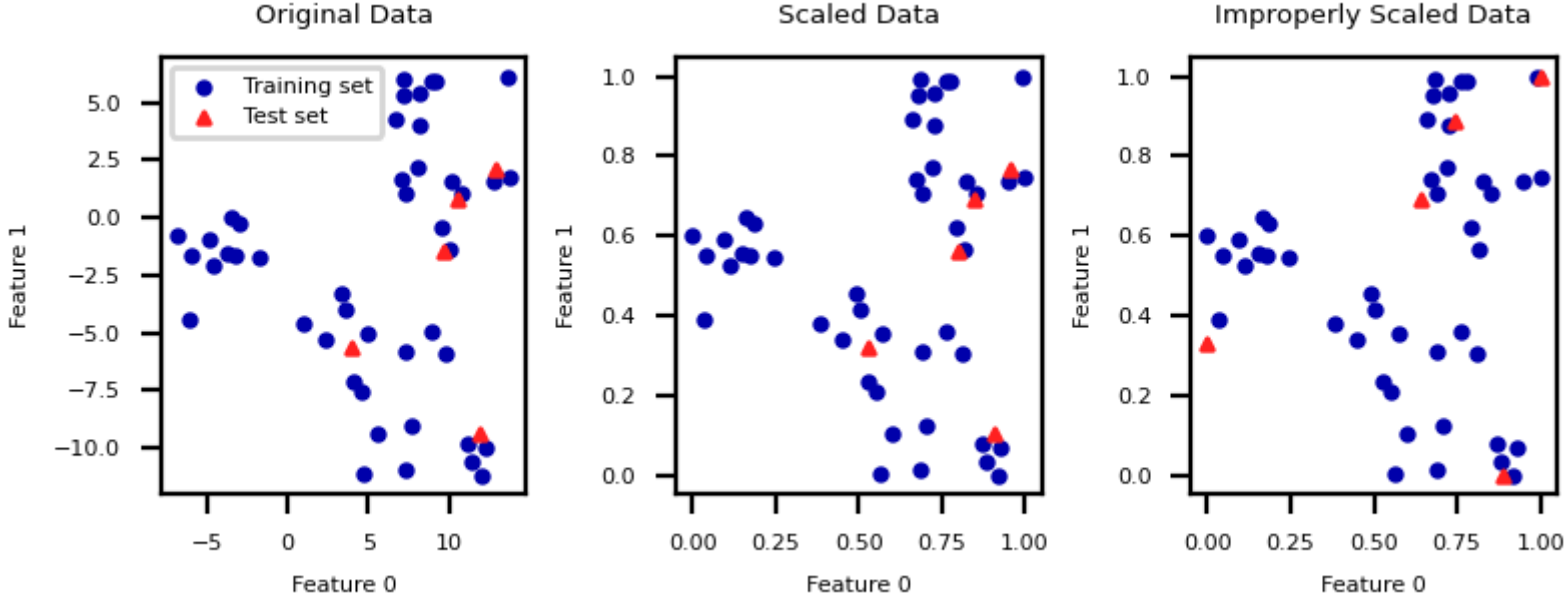
In practice (scikit-learn)

```
# choose scaling method and fit on training data
scaler = StandardScaler()
scaler.fit(X_train)

# transform training and test data
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
# calling fit and transform in sequence
X_train_scaled = scaler.fit(X_train).transform(X_train)
# same result, but more efficient computation
X_train_scaled = scaler.fit_transform(X_train)
```

Test set distortion

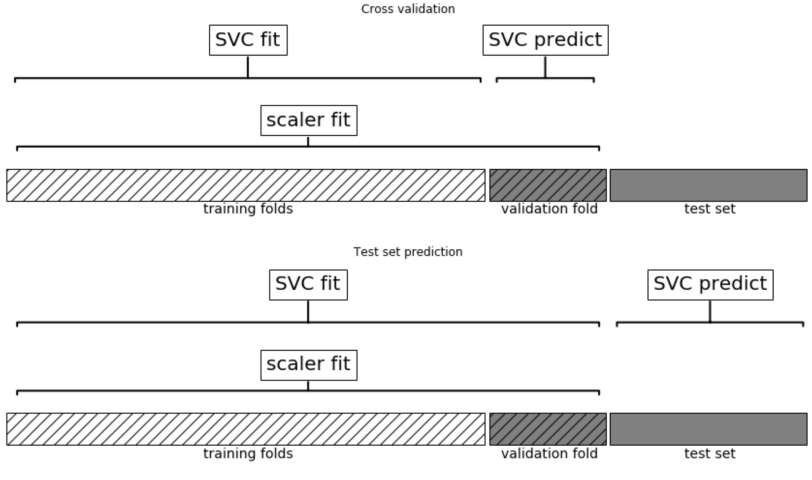
- Properly scaled: `fit` on training set, `transform` on training and test set
- Improperly scaled: `fit` and `transform` on the training and test data separately
    - Test data points nowhere near same training data points



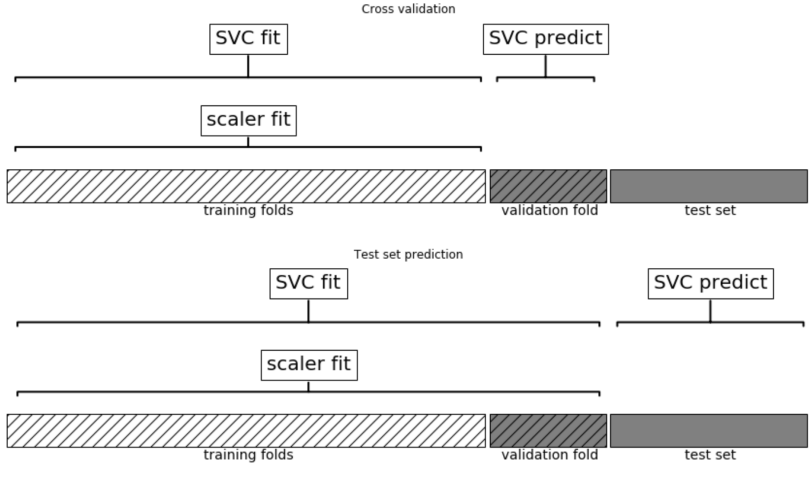
Data leakage

- Cross-validation: training set is split into training and validation sets for model selection
- Incorrect: Scaler is fit on whole training set before doing cross-validation
    - Data leaks from validation folds into training folds, selected model may be optimistic
- Right: Scaler is fit on training folds only

Information Leak



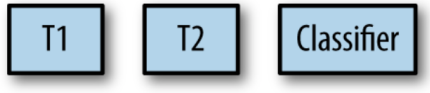
No Information leakage



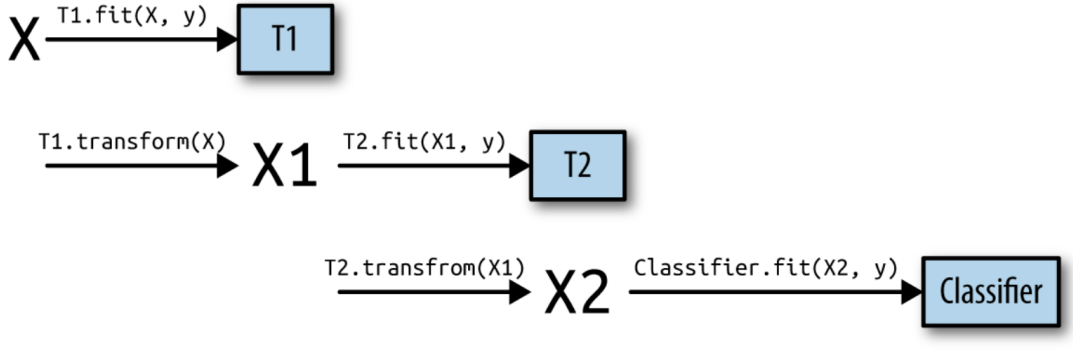
Pipelines

- A pipeline is a combination of data transformation and learning algorithms
- It has a `fit`, `predict`, and `score` method, just like any other learning algorithm
    - Ensures that data transformations are applied correctly

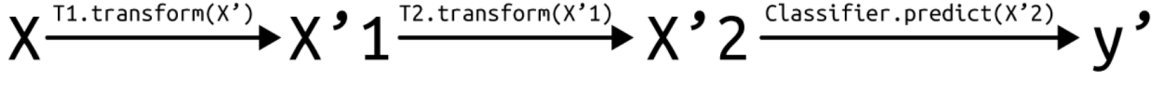
```
pipe = make_pipeline(T1(), T2(), Classifier())
```




```
pipe.fit(X, y)
```




```
pipe.predict(X')
```



In practice (scikit-learn)

- A `pipeline` combines multiple processing *steps* in a single estimator
- All but the last step should be data transformer (have a `transform` method)

```
# Make pipeline, step names will be 'minmaxscaler' and 'linearsvc'
pipe = make_pipeline(MinMaxScaler(), LinearSVC())
# Build pipeline with named steps
pipe = Pipeline([("scaler", MinMaxScaler()), ("svm", LinearSVC())])

# Correct fit and score
score = pipe.fit(X_train, y_train).score(X_test, y_test)
# Retrieve trained model by name
svm = pipe.named_steps["svm"]
```

```
# Correct cross-validation
scores = cross_val_score(pipe, X, y)
```

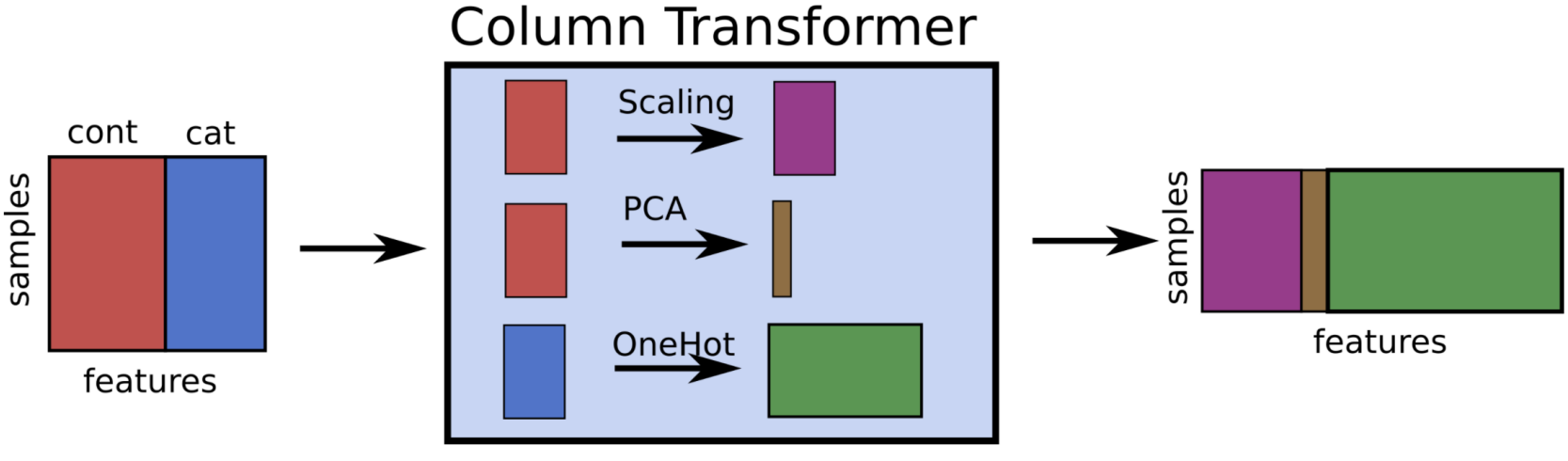
- If you want to apply different preprocessors to different columns, use `ColumnTransformer`
- If you want to merge pipelines, you can use `FeatureUnion` to concatenate columns

```
# 2 sub-pipelines, one for numeric features, other for categorical
ones
numeric_pipe = make_pipeline(SimpleImputer(),StandardScaler())
categorical_pipe = make_pipeline(SimpleImputer(),OneHotEncoder())

# Using categorical pipe for features A,B,C, numeric pipe otherwise
preprocessor = make_column_transformer((categorical_pipe,
                                        ["A","B","C"]),
                                        remainder=numeric_pipe)

# Combine with learning algorithm in another pipeline
pipe = make_pipeline(preprocess, LinearSVC())
```

```
# Feature union of PCA features and selected features
union = FeatureUnion([("pca", PCA()), ("selected", SelectKBest())])
pipe = make_pipeline(union, LinearSVC())
```

- `ColumnTransformer` concatenates features in order

```
pipe = make_column_transformer((StandardScaler(),numeric_features),
                               (PCA(),numeric_features),

(OneHotEncoder(),categorical_features))
```



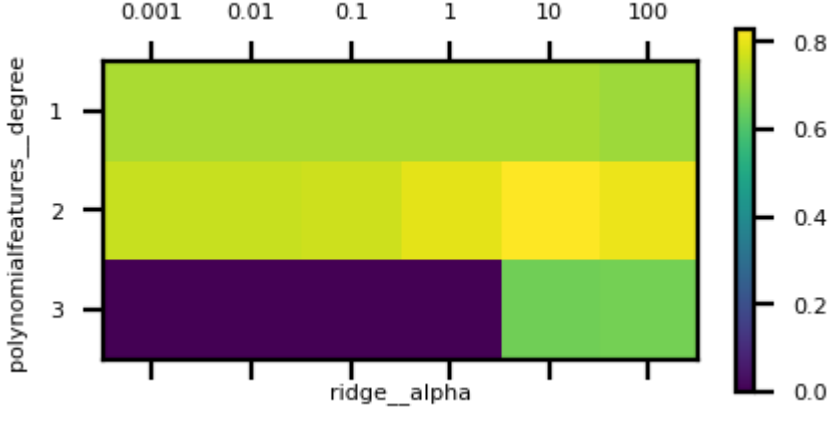
Pipeline selection

- We can safely use pipelines in model selection (e.g. grid search)
- Use `'__'` to refer to the hyperparameters of a step, e.g. `svm__C`

```
# Correct grid search (can have hyperparameters of any step)
param_grid = {'svm__C': [0.001, 0.01],
              'svm__gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
grid = GridSearchCV(pipe, param_grid=param_grid).fit(X,y)
# Best estimator is now the best pipeline
best_pipe = grid.best_estimator_

# Tune pipeline and evaluate on held-out test set
grid = GridSearchCV(pipe,
param_grid=param_grid).fit(X_train,y_train)
grid.score(X_test,y_test)
```

Example: Tune multiple steps at once

```
pipe = make_pipeline(StandardScaler(),PolynomialFeatures(), Ridge())
param_grid = {'polynomialfeatures__degree': [1, 2, 3],
              'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
grid = GridSearchCV(pipe, param_grid=param_grid).fit(X_train,
y_train)
```



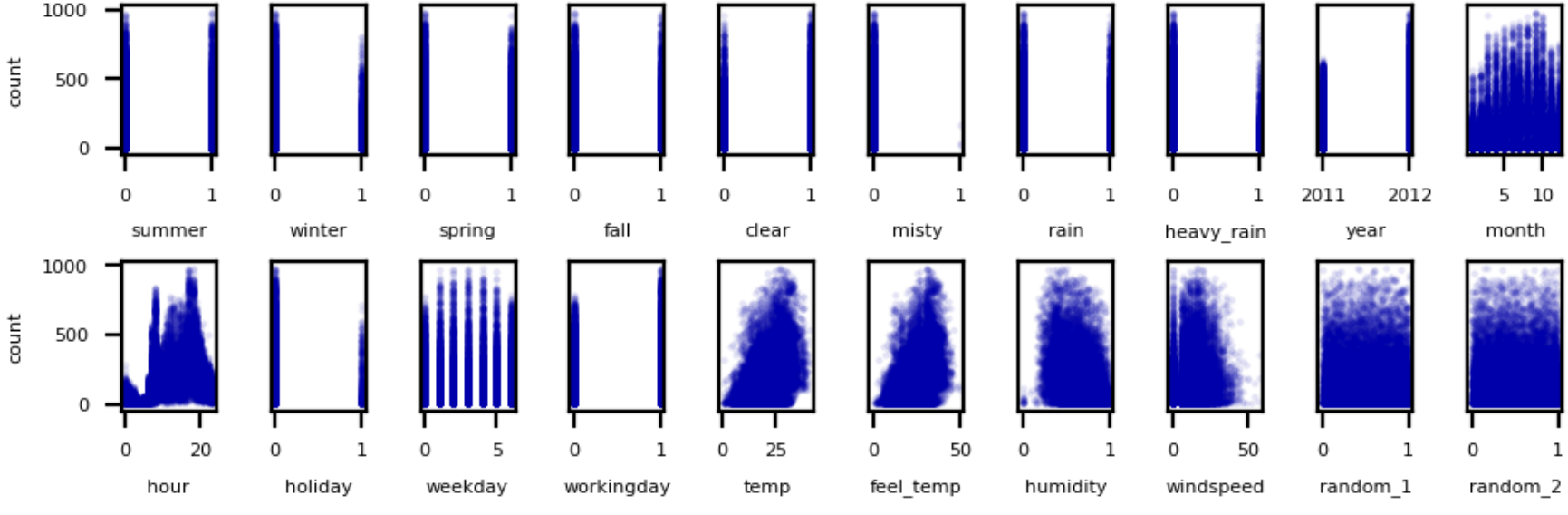
Automatic Feature Selection

It can be a good idea to reduce the number of features to only the most useful ones

- Simpler models that generalize better (less overfitting)
    - Curse of dimensionality (e.g. kNN)
    - Even models such as RandomForest can benefit from this
    - Sometimes it is one of the main methods to improve models (e.g. gene expression data)
- Faster prediction and training
    - Training time can be quadratic (or cubic) in number of features
- Easier data collection, smaller models (less storage)
- More interpretable models: fewer features to look at

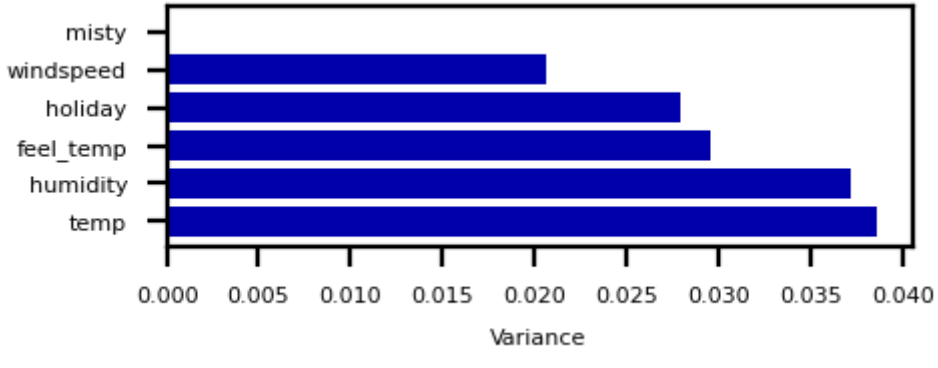
Example: bike sharing

- The Bike Sharing Demand dataset shows the amount of bikes rented in Washington DC
- Some features are clearly more informative than others (e.g. temp, hour)
- Some are correlated (e.g. temp and feel_temp)
- We add two random features at the end



Unsupervised feature selection

- Variance-based
    - Remove (near) constant features
        - Choose a small variance threshold
    - Scale features before computing variance!
    - Infrequent values may still be important
- Covariance-based
    - Remove correlated features
    - The small differences may actually be important
        - You don't know because you don't consider the target

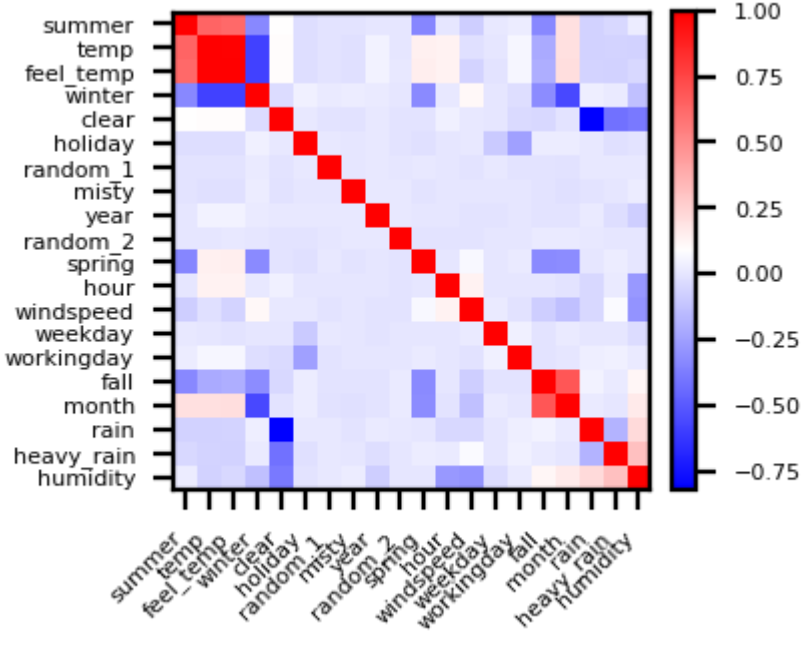


Covariance based feature selection

- Remove features X_i (= $\mathbf{X_{:,i}}$) that are highly correlated (have high correlation coefficient ρ)

$$\rho(X_1, X_2) = \frac{\text{cov}(X_1, X_2)}{\sigma(X_1)\sigma(X_2)} = \frac{\frac{1}{N-1}\sum_i (X_{i,1} - \overline{X_1})(X_{i,2} - \overline{X_2})}{\sigma(X_1)\sigma(X_2)}$$

- Should we remove `feel_temp`? Or `temp`? Maybe one correlates more with the target?

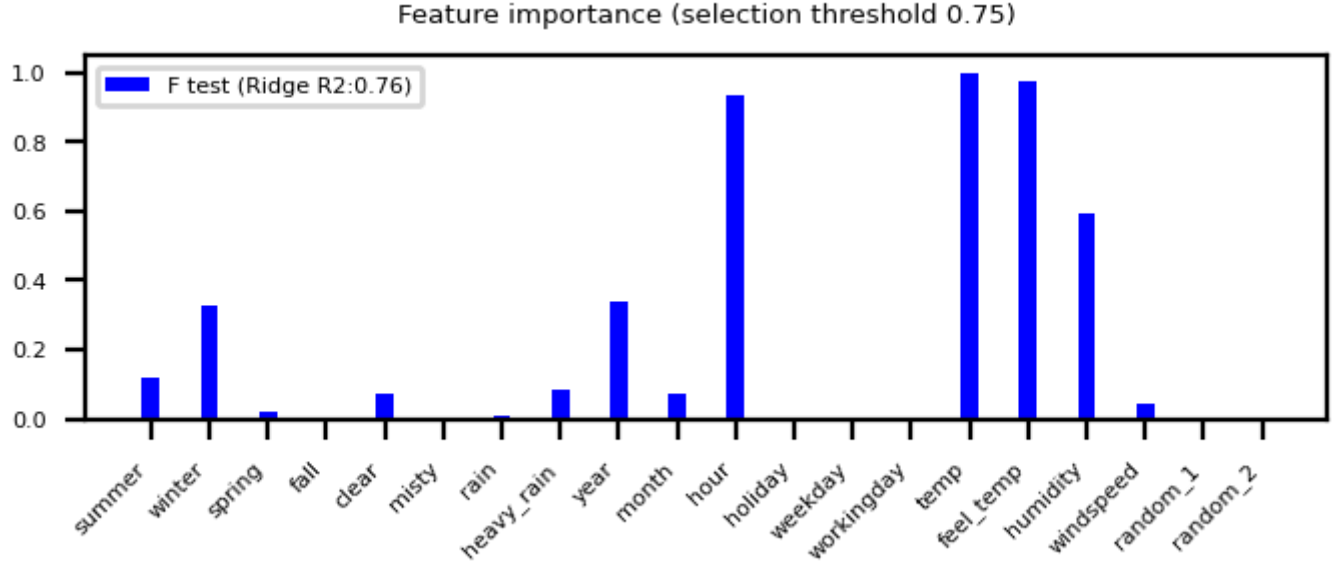


Supervised feature selection: overview

- Univariate: F-test and Mutual Information
- Model-based: Random Forests, Linear models, kNN
- Wrapping techniques (black-box search)
- Permutation importance

Univariate statistics (F-test)

- Consider each feature individually (univariate), independent of the model that you aim to apply
- Use a statistical test: is there a *linear* **statistically significant relationship** with the target?
- Use F-statistic (or corresponding p value) to rank all features, then select features using a threshold
    - Best k, best k %, probability of removing useful features (FPR),...
- Cannot detect correlations (e.g. temp and feel_temp) or interactions (e.g. binary features)



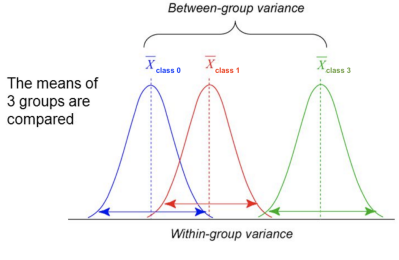
F-statistic

- For regression: does feature X_i correlate (positively or negatively) with the target y?

$$\text{F-statistic} = \frac{\rho(X_i, y)^2}{1 - \rho(X_i, y)^2} \cdot (N - 1)$$

- For classification: uses ANOVA: does X_i explain the between-class variance?
    - Alternatively, use the χ^2 test (only for categorical features)

$$\text{F-statistic} = \frac{\text{within-class variance}}{\text{between-class variance}} = \frac{\overline{var(X_i)}}{\overline{var(X_i)}}$$

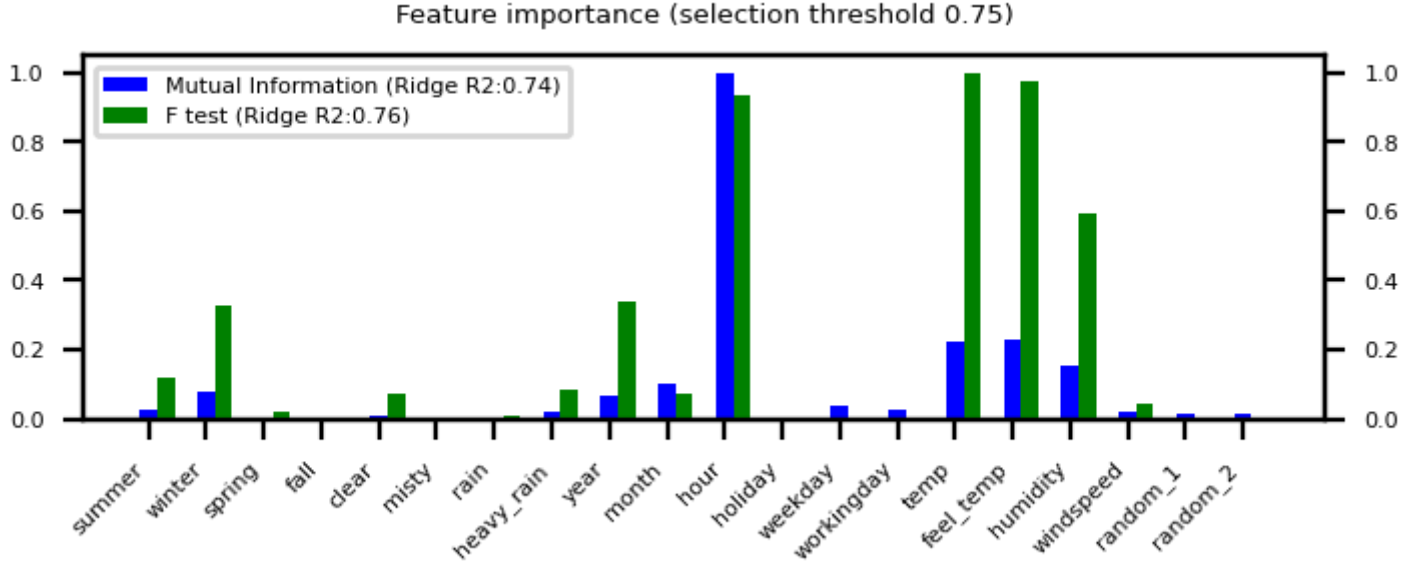


Mutual information

- Measures how much information X_i gives about the target Y. In terms of entropy H:

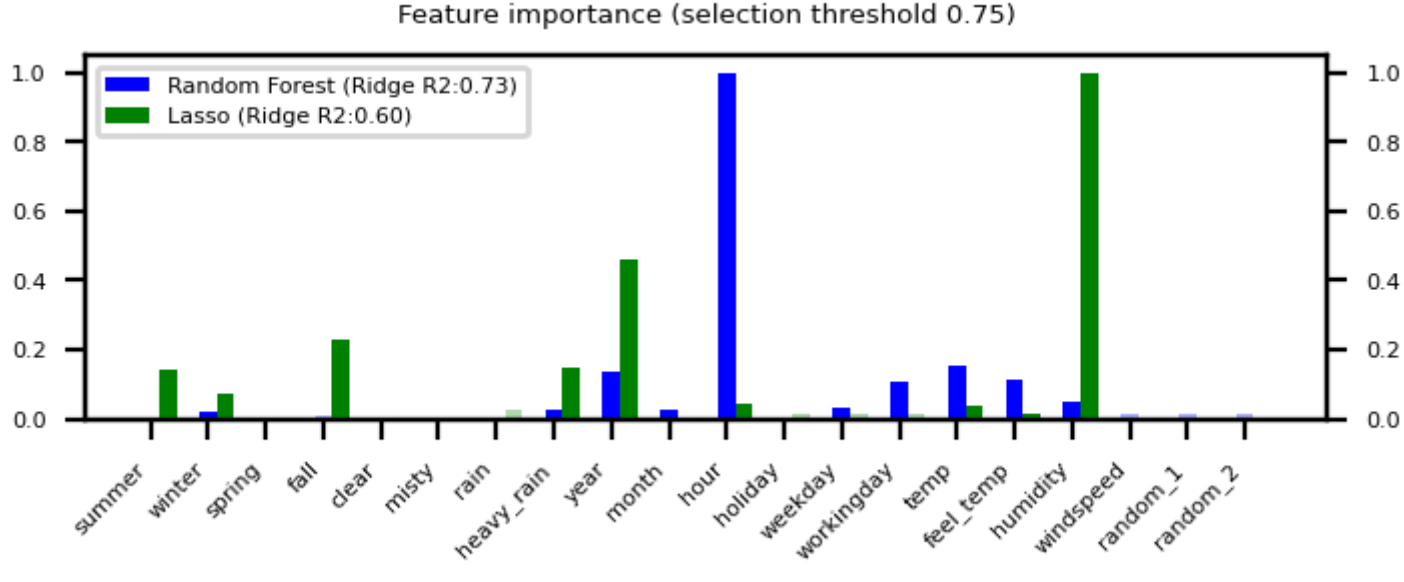
$$MI(X, Y) = H(X) + H(Y) - H(X, Y)$$

- Idea: estimate H(X) as the average distance between a data point and its k Nearest Neighbors
    - You need to choose k and say which features are categorical
- Captures complex dependencies (e.g. hour, month), but requires more samples to be accurate



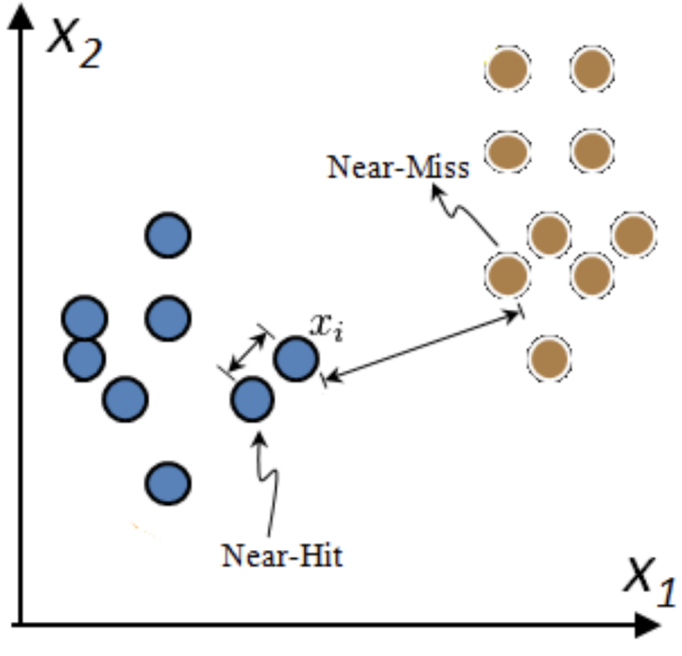
Model-based Feature Selection

- Use a tuned(!) supervised model to judge the importance of each feature
    - Linear models (Ridge, Lasso, LinearSVM,...): features with highest weights (coefficients)
    - Tree–based models: features used in first nodes (high information gain)
- Selection model can be different from the one you use for final modelling
- Captures interactions: features are more/less informative in combination (e.g. winter, temp)
- RandomForests: learns complex interactions (e.g. hour), but biased to high cardinality features



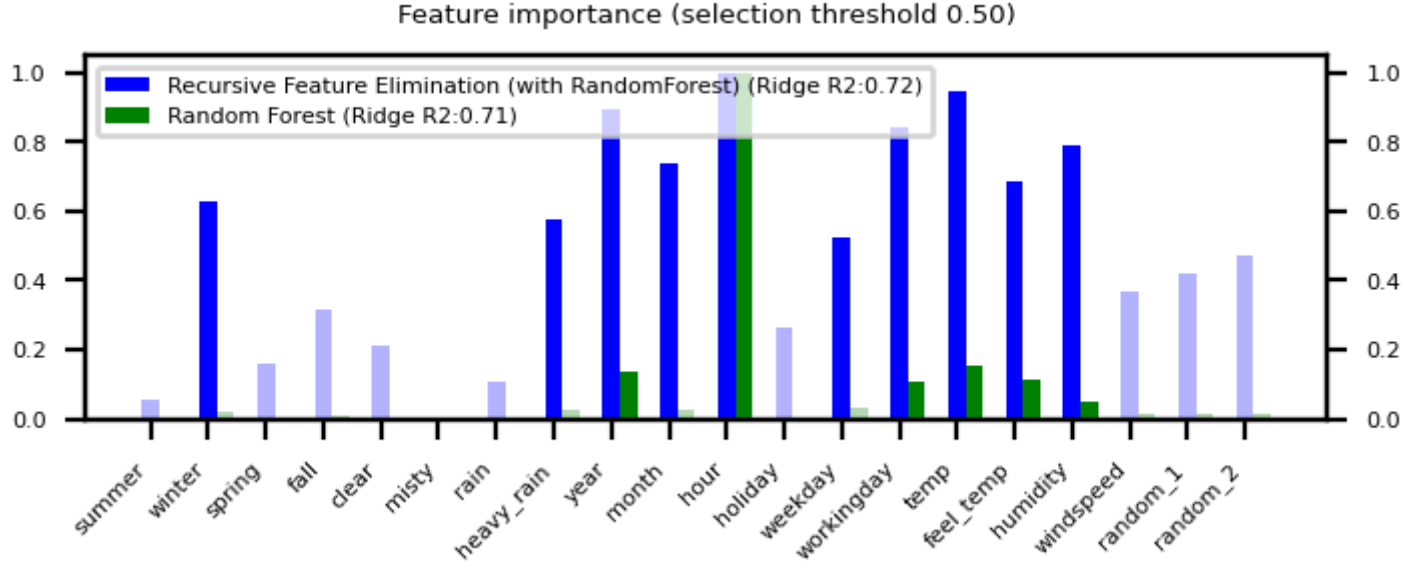
Relief: Model-based selection with kNN

- For I iterations, choose a random point $\mathbf{x_i}$ and find k nearest neighbors $\mathbf{x_k}$
- Increase feature weights if $\mathbf{x_i}$ and $\mathbf{x_k}$ have different class (near miss), else decrease
    - $\mathbf{w_i} = \mathbf{w_{i-1}} + (\mathbf{x_i} - \text{nearMiss}_i)^2 - (\mathbf{x_i} - \text{nearHit}_i)^2$
- Many variants: ReliefF (uses L1 norm, faster), RReliefF (for regression), ...



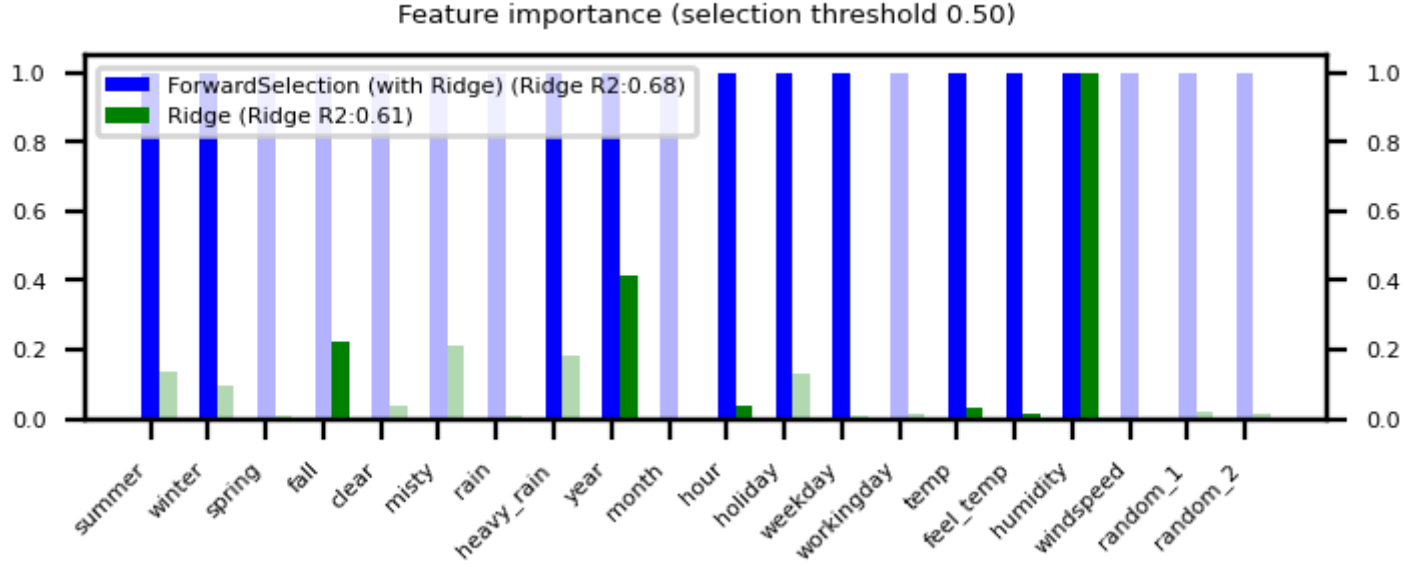
Iterative Model-based Feature Selection

- Dropping many features at once is not ideal: feature importance may change in subset
- Recursive Feature Elimination (RFE)
    - Remove s least important feature(s), recompute remaining importances, repeat
- Can be rather slow



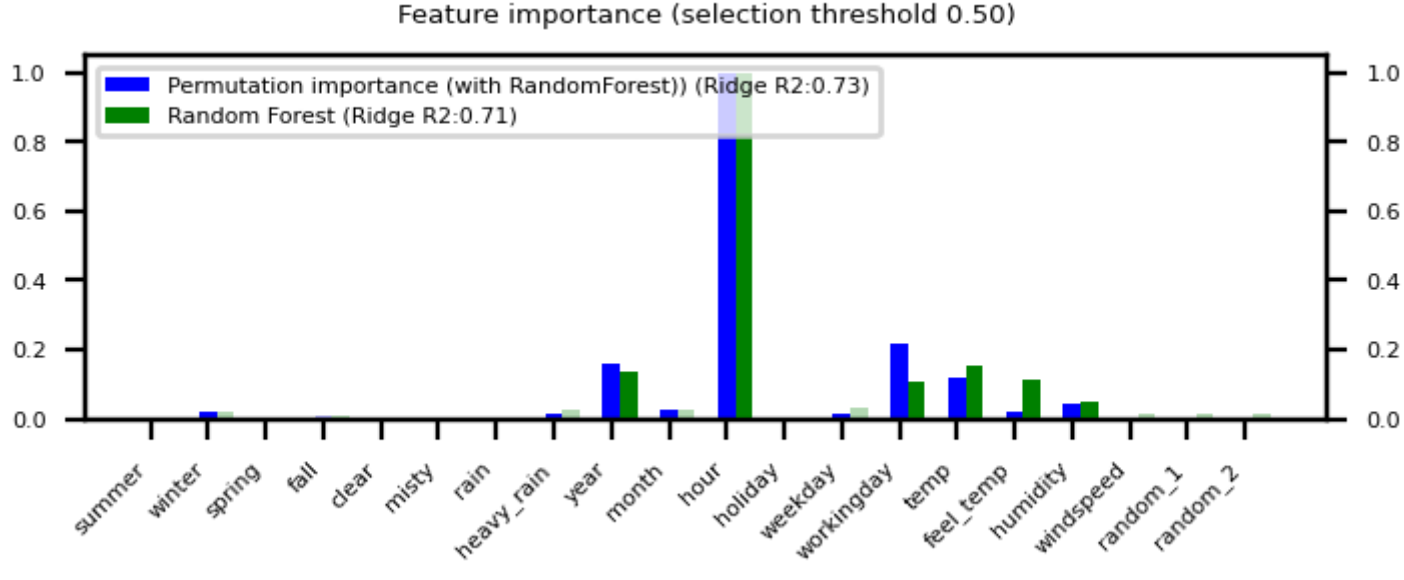
Sequential feature selection (Wrapping)

- Evaluate your model with different sets of features, find best subset based on performance
- Greedy black-box search (can end up in local minima)
    - Backward selection: remove least important feature, recompute importances, repeat
    - Forward selection: set aside most important feature, recompute importances, repeat
    - Floating: add best new feature, remove worst one, repeat (forward or backward)
- Stochastic search: use random mutations in candidate subset (e.g. simulated annealing)



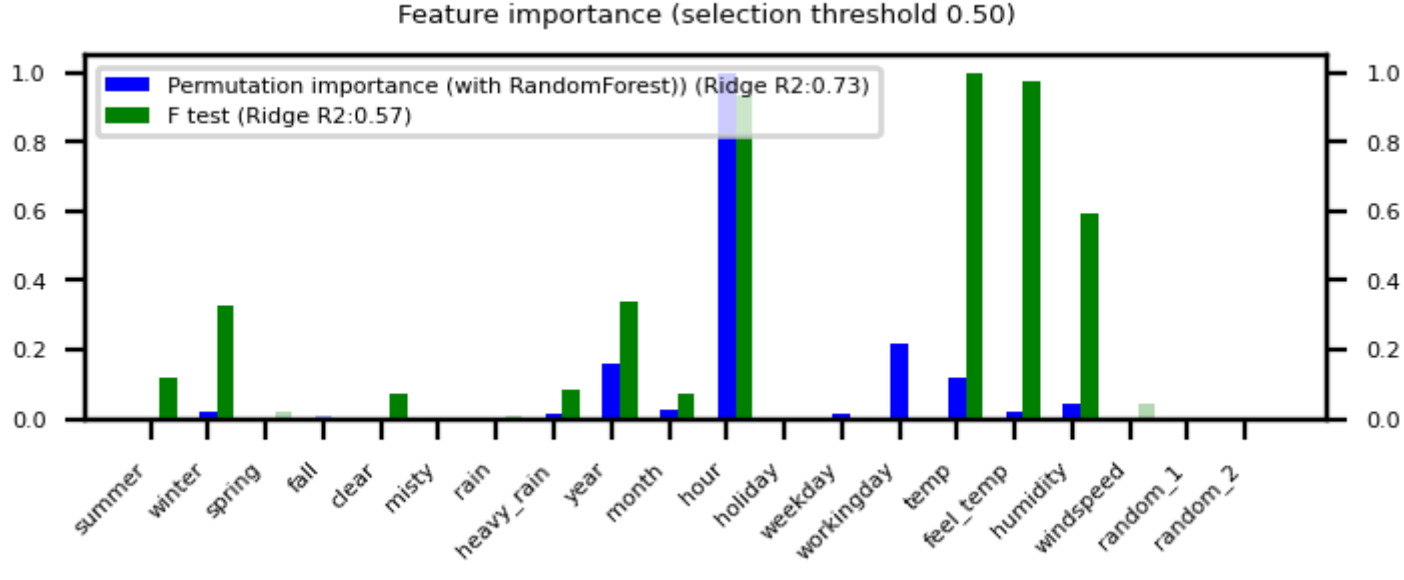
Permutation feature importance

- Defined as the decrease in model performance when a single feature value is randomly shuffled
    - This breaks the relationship between the feature and the target
- Model agnostic, metric agnostic, and can be calculated many times with different permutations
- Can be applied to unseen data (not possible with model-based techniques)
- Less biased towards high-cardinality features (compared with RandomForests)



Comparison

- Feature importances (scaled) and cross-validated R^2 score of pipeline
    - Pipeline contains features selection + Ridge
- Selection threshold value ranges from 25% to 100% of all features
- Best method ultimately depends on the problem and dataset at hand



In practice (scikit-learn)

- Unsupervised: `VarianceTreshold`

```
selector = VarianceThreshold(threshold=0.01)
X_selected = selector.fit_transform(X)
variances = selector.variances_
```

- Univariate:
    - For regression: `f_regression`, `mutual_info_regression`
    - For classification: `f_classification`, `chi2`, `mutual_info_classication`
    - Selecting: `SelectKBest`, `SelectPercentile`, `SelectFpr`,...

```
selector = SelectPercentile(score_func=f_regression, percentile=50)
X_selected = selector.fit_transform(X,y)
selected_features = selector.get_support()
f_values, p_values = f_regression(X,y)
mi_values = mutual_info_regression(X,y,discrete_features=[])
```

- Model-based:
  - `SelectFromModel`: requires a model and a selection threshold
  - `RFE`, `RFECV` (recursive feature elimination): requires model and final nr features

```
selector = SelectFromModel(RandomForestRegressor(),
threshold='mean')
rfe_selector = RFE(RidgeCV(), n_features_to_select=20)
X_selected = selector.fit_transform(X)
rf_importances = Randomforest().fit(X, y).feature_importances_
```

- Sequential feature selection (from `mlxtend`, sklearn-compatible)

```
selector = SequentialFeatureSelector(RidgeCV(), k_features=20,
forward=True,
                                     floating=True)
X_selected = selector.fit_transform(X)
```

- Permutation Importance (in `sklearn.inspection`), no fit-transform interface

```
importances =
permutation_importance(RandomForestRegressor().fit(X,y),
                                     X, y,
n_repeats=10).importances_mean
feature_ids = (-importances).argsort()[:n]
```

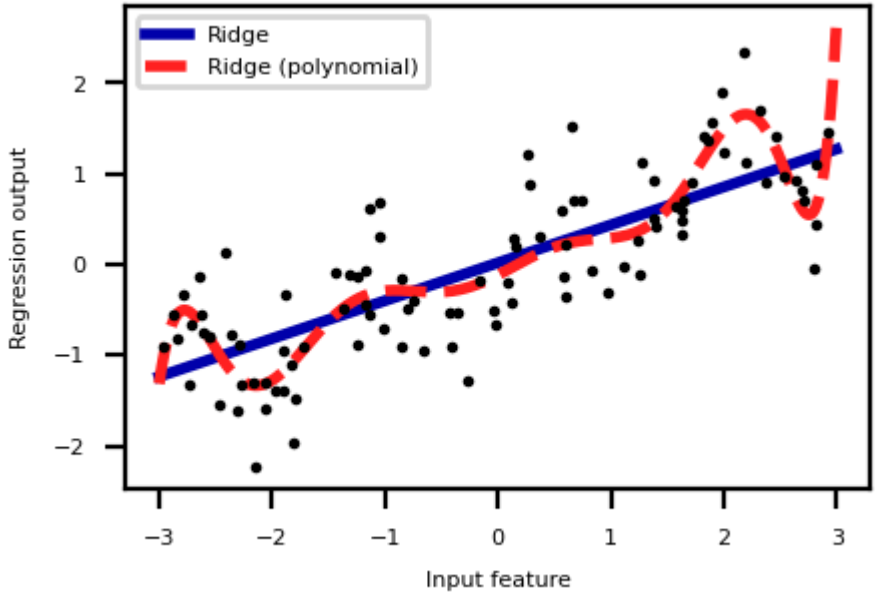
Feature Engineering

- Create new features based on existing ones
    - Polynomial features
    - Interaction features
    - Binning
- Mainly useful for simple models (e.g. linear models)
    - Other models can learn interations themselves
    - But may be slower, less robust than linear models

Polynomials

- Add all polynomials up to degree d and all products
    - Equivalent to polynomial basis expansions

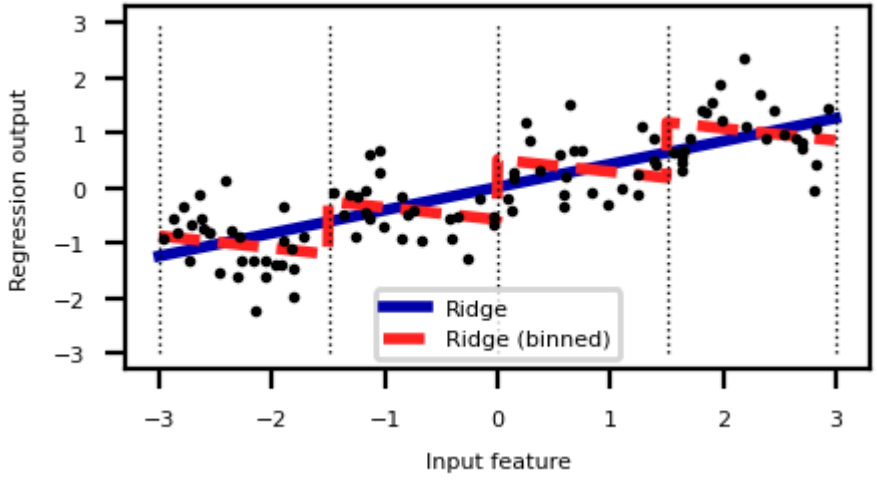
$$[1, x_1, \ldots, x_p] \rightarrow [1, x_1, \ldots, x_p, x_1^2, \ldots, x_p^2, \ldots, x_p^d, x_1 x_2, \ldots, x_{p-1} x_p]$$



Binning

- Partition numeric feature values into n intervals (bins)
- Create n new one-hot features, 1 if original value falls in corresponding bin
- Models different intervals differently (e.g. different age groups)

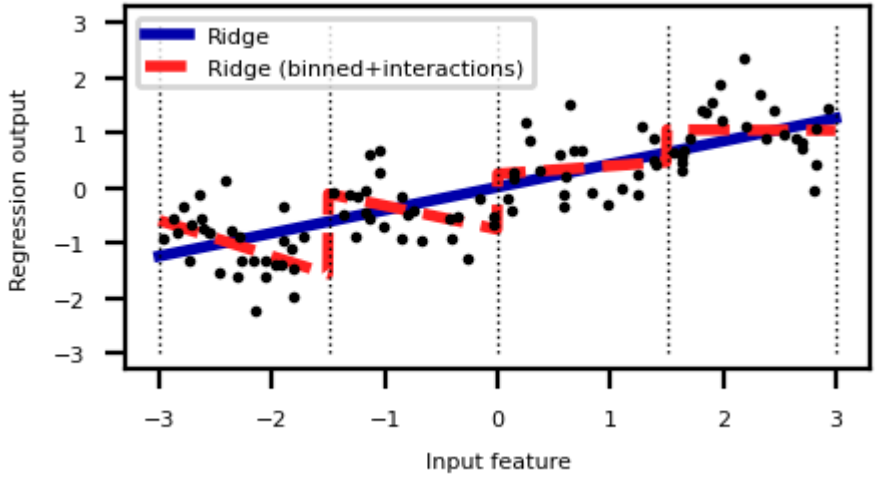
	orig	[-3.0,-1.5]	[-1.5,0.0]	[0.0,1.5]	[1.5,3.0]
0	-0.752759	0.000000	1.000000	0.000000	0.000000
1	2.704286	0.000000	0.000000	0.000000	1.000000
2	1.391964	0.000000	0.000000	1.000000	0.000000



Binning + interaction features

- Add *interaction features* (or *product features*)
  - Product of the bin encoding and the original feature value
  - Learn different weights per bin

	orig	b0	b1	b2	b3	X*b0	X*b1	X*b2	X*b3
0	-0.752759	0.000000	1.000000	0.000000	0.000000	-0.000000	-0.752759	-0.000000	-0.000000
1	2.704286	0.000000	0.000000	0.000000	1.000000	0.000000	0.000000	0.000000	2.704286
2	1.391964	0.000000	0.000000	1.000000	0.000000	0.000000	0.000000	1.391964	0.000000



Categorical feature interactions

- One-hot-encode categorical feature
- Multiply every one-hot-encoded column with every numeric feature
- Allows to built different submodels for different categories

	gender	age	pageviews	time
0	M	14	70	269
1	F	16	12	1522
2	M	12	42	235
3	F	25	64	63
4	F	22	93	21

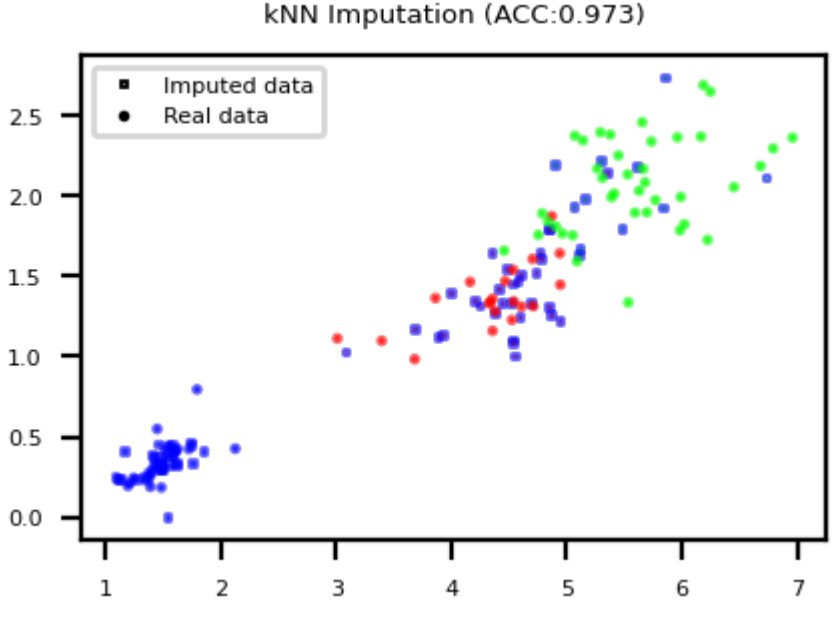
	age_M	pageviews_M	time_M	gender_M_M	age_F	pageviews_F	time_F	gender_F_F
0	14	70	269	1	0	0	0	0
1	0	0	0	0	16	12	1522	1
2	12	42	235	1	0	0	0	0
3	0	0	0	0	25	64	63	1
4	0	0	0	0	22	93	21	1

Missing value imputation

- Data can be missing in different ways:
    - Missing Completely at Random (MCAR): purely random points are missing
    - Missing at Random (MAR): something affects missingness, but no relation with the value
        - E.g. faulty sensors, some people don't fill out forms correctly
    - Missing Not At Random (MNAR): systematic missingness linked to the value
        - Has to be modelled or resolved (e.g. sensor decay, sick people leaving study)
- Missingness can be encoded in different ways:'?', '-1', 'unknown', 'NA',...
- Also labels can be missing (remove example or use semi-supervised learning)

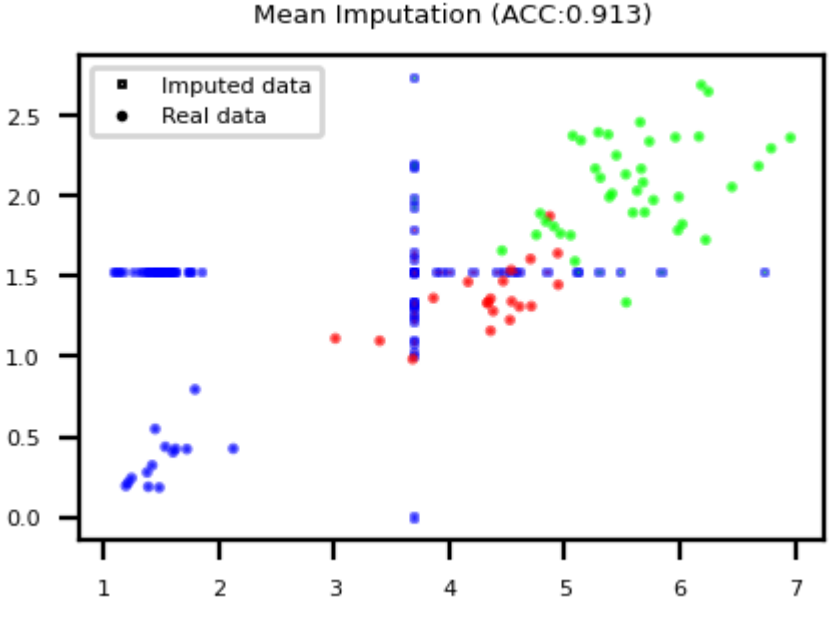
Overview

- Mean/constant imputation
- kNN-based imputation
- Iterative (model-based) imputation
- Matrix Factorization techniques



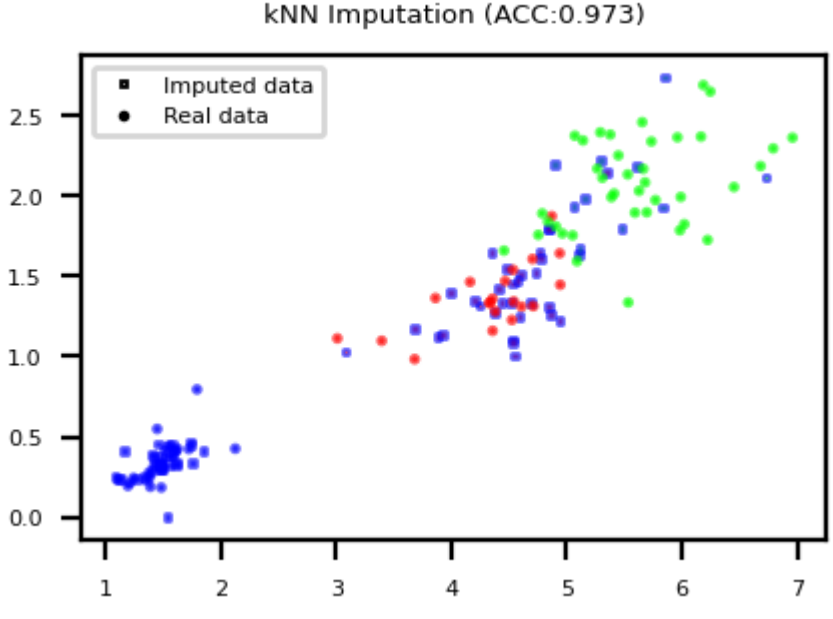
Mean imputation

- Replace all missing values of a feature by the same value
    - Numerical features: mean or median
    - Categorical features: most frequent category
    - Constant value, e.g. 0 or 'missing' for text features
- Optional: add an indicator column for missingness
- Example: Iris dataset (randomly removed values in 3rd and 4th column)



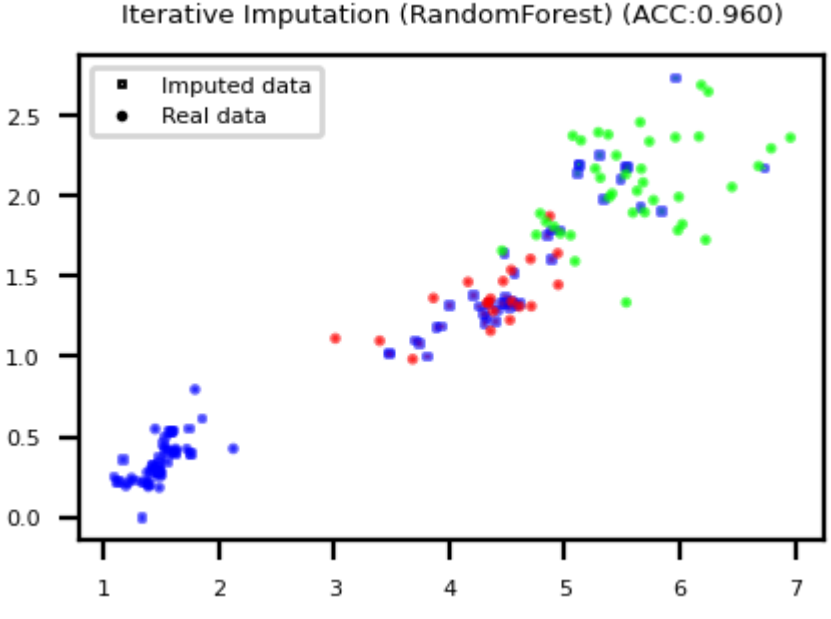
kNN imputation

- Use special version of kNN to predict value of missing points
- Uses only non-missing data when computing distances



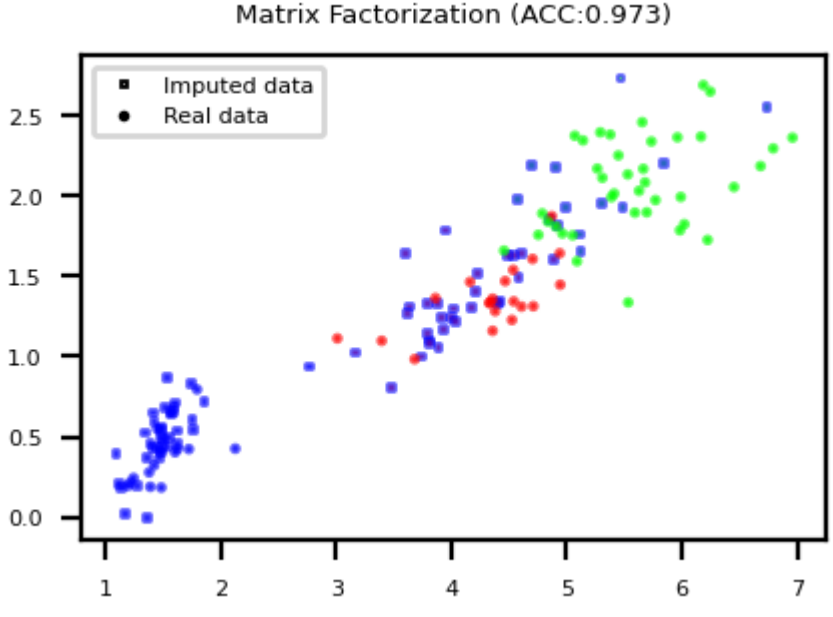
Iterative (model-based) Imputation

- Better known as Multiple Imputation by Chained Equations (MICE)
- Iterative approach
    - Do first imputation (e.g. mean imputation)
    - Train model (e.g. RandomForest) to predict missing values of a given feature
    - Train new model on imputed data to predict missing values of the next feature
        - Repeat m times in round-robin fashion, leave one feature out at a time



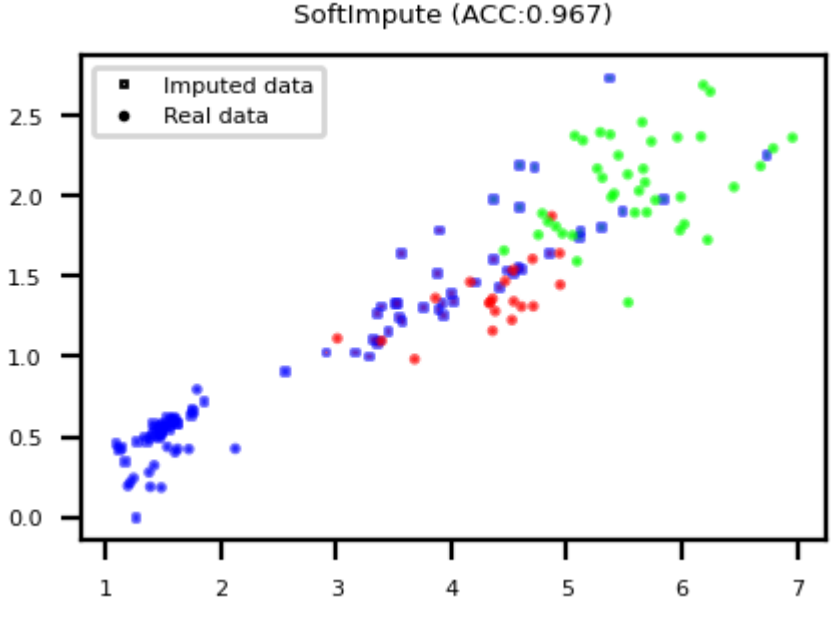
Matrix Factorization

- Basic idea: low-rank approximation
    - Replace missing values by 0
    - Factorize $\mathbf{X}$ with rank r: $\mathbf{X}^{n \times p} = \mathbf{U}^{n \times r}\mathbf{V}^{r \times p}$
        - With n data points and p features
        - Solved using gradient descent
    - Recompute $\mathbf{X}$: now complete



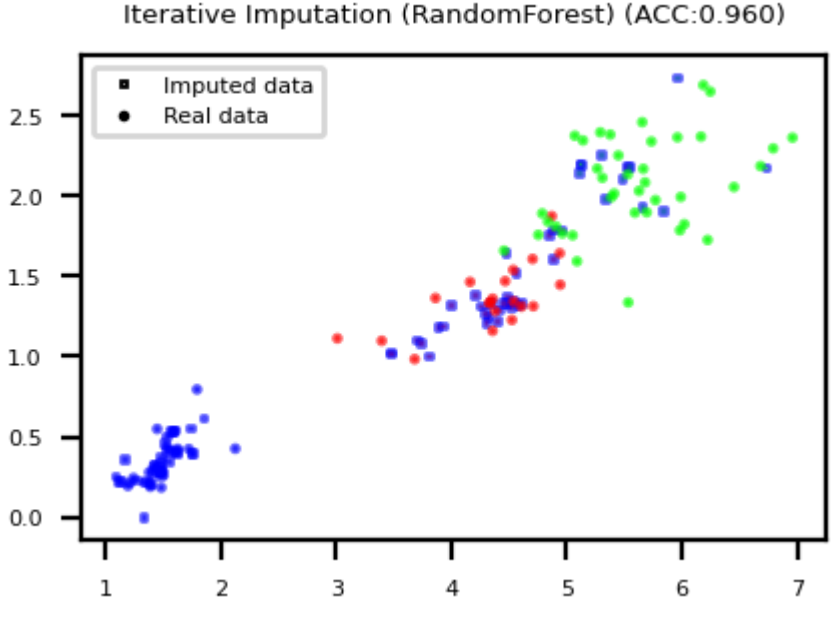
Soft-thresholded Singular Value Decomposition (SVD)

- Same basic idea, but smoother
    - Replace missing values by 0, compute SVD: $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathbf{T}}$
        - Solved with gradient descent
    - Reduce eigenvalues by shrinkage factor: $\lambda_i = s \cdot \lambda_i$
    - Recompute $\mathbf{X}$: now complete
    - Repeat for m iterations



Comparison

- Best method depends on the problem and dataset at hand. Use cross-validation.
- Iterative Imputation (MICE) generally works well for missing (completely) at random data
    - Can be slow if the prediction model is slow
- Low-rank approximation techniques scale well to large datasets



In practice (scikit-learn)

- Simple replacement: `SimpleImputer`
  - Strategies: `mean` (numeric), `median`, `most_frequent` (categorical)
  - Choose whether to add indicator columns, and how missing values are encoded

```
imp = SimpleImputer(strategy='mean', missing_values=np.nan,
add_indicator=False)
X_complete = imp.fit_transform(X_train)
```

- kNN Imputation: `KNNImputer`

```
imp = KNNImputer(n_neighbors=5)
X_complete = imp.fit_transform(X_train)
```

- Multiple Imputation (MICE): `IterativeImputer`
  - Choose estimator (default: `BayesianRidge`) and number of iterations (default 10)

```
imp = IterativeImputer(estimator=RandomForestClassifier(),
max_iter=10)
X_complete = imp.fit_transform(X_train)
```

In practice (fancyimpute)

- Cannot be used in CV pipelines (has `fit_transform` but no `transform`)
- Soft-Thresholded SVD: `SoftImpute`
    - Choose max number of gradient descent iterations
    - Choose shrinkage value for eigenvectors (default: $\frac{1}{N}$)

```
imp = SoftImpute(max_iter=10, shrinkage_value=None)
X_complete = imp.fit_transform(X)
```

- Low-rank imputation: `MatrixFactorization`
    - Choose rank of the low-rank approximation
    - Gradient descent hyperparameters: learning rate, epochs,...
    - Several variants exist

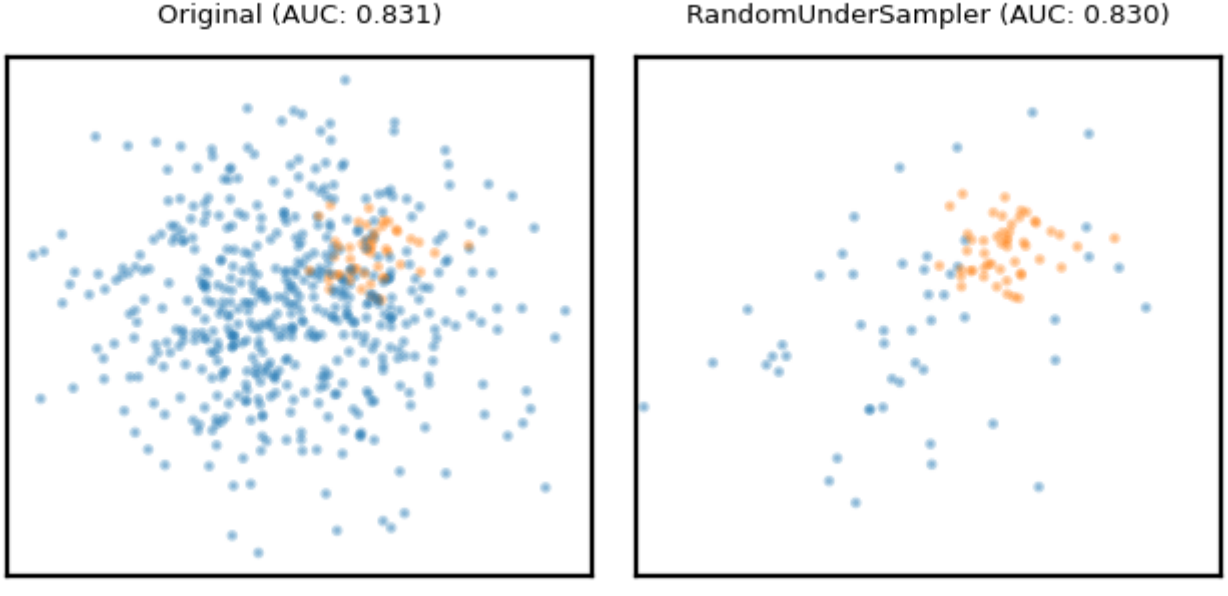
```
imp = MatrixFactorization(rank=10, learning_rate=0.001,
epochs=10000)
X_complete = imp.fit_transform(X)
```

Handling imbalanced data

- Problem:
    - You have a majority class with many times the number of examples as the minority class
    - Or: classes are balanced, but associated costs are not (e.g. FN are worse than FP)
- We already covered some ways to resolve this:
    - Add class weights to the loss function: give the minority class more weight
        - In practice: set `class_weight='balanced'`
    - Change the prediction threshold to minimize false negatives or false positives
- There are also things we can do by preprocessing the data
    - Resample the data to correct the imbalance
        - Random or model-based
    - Generate synthetic samples for the minority class
    - Build ensembles over different resampled datasets
    - Combinations of these

Random Undersampling

- Copy the points from the minority class
- Randomly sample from the majority class (with or without replacement) until balanced
    - Optionally, sample until a certain imbalance ratio (e.g. 1/5) is reached
    - Multi-class: repeat with every other class
- Preferred for large datasets, often yields smaller/faster models with similar performance

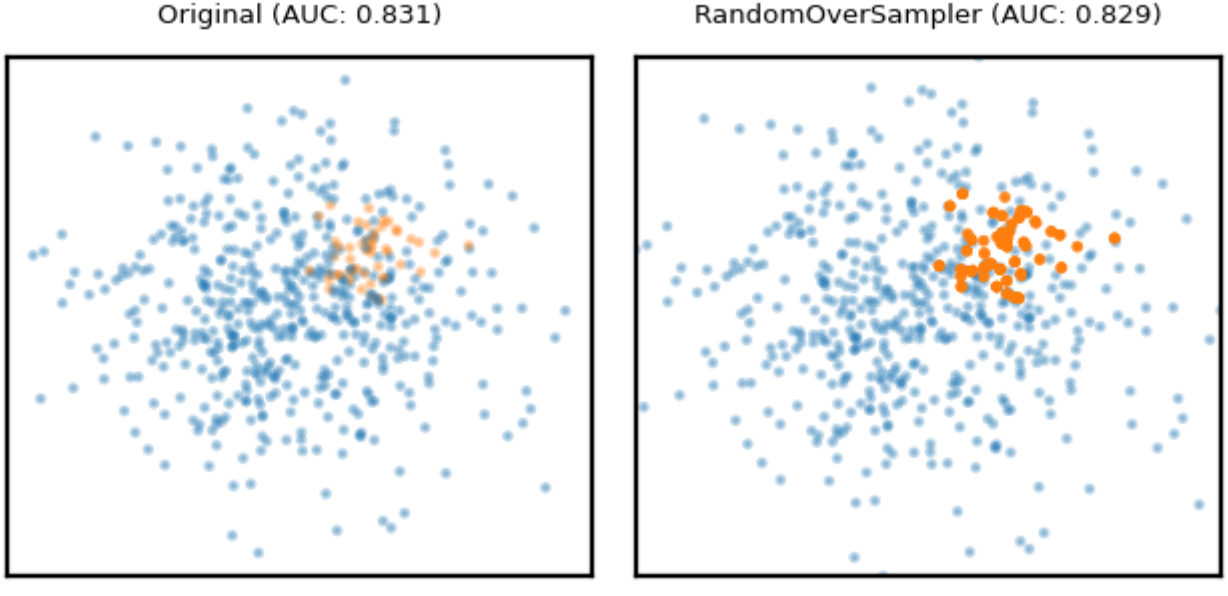


Model-based Undersampling

- Edited Nearest Neighbors
    - Remove all majority samples that are misclassified by kNN (mode) or that have a neighbor from the other class (all).
    - Remove their influence on the minority samples
- Condensed Nearest Neighbors
    - Remove all majority samples that are *not* misclassified by kNN
    - Focus on only the hard samples

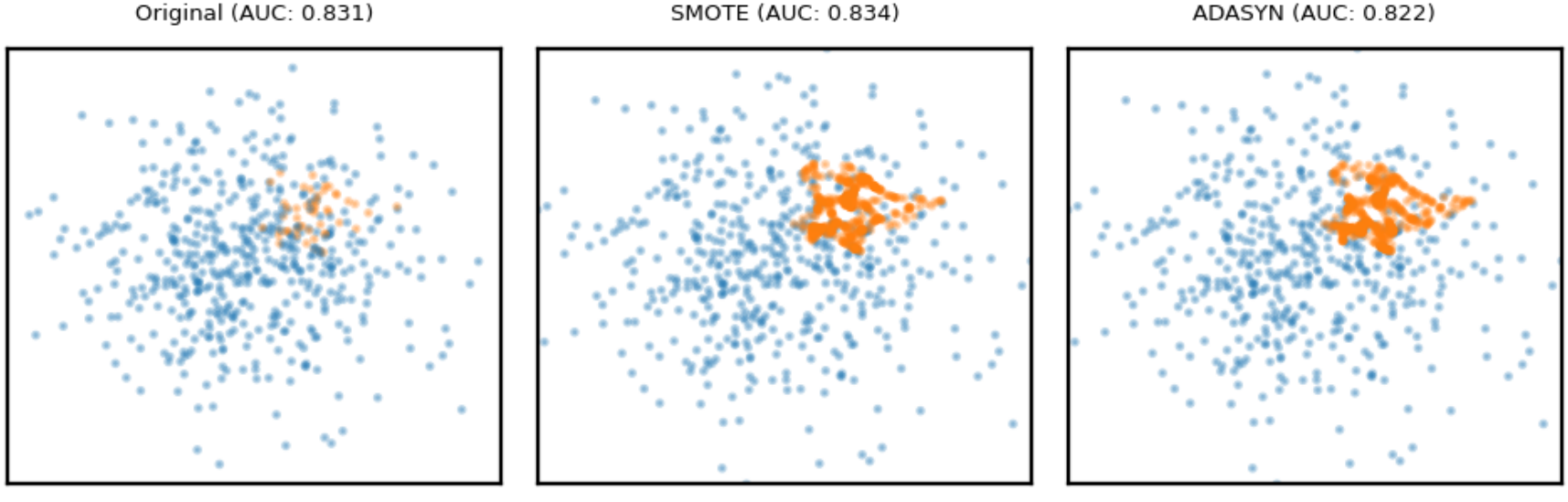
Random Oversampling

- Copy the points from the majority class
- Randomly sample from the minority class, with replacement, until balanced
    - Optionally, sample until a certain imbalance ratio (e.g. 1/5) is reached
- Makes models more expensive to train, doens't always improve performance
- Similar to giving minority class(es) a higher weight (and more expensive)



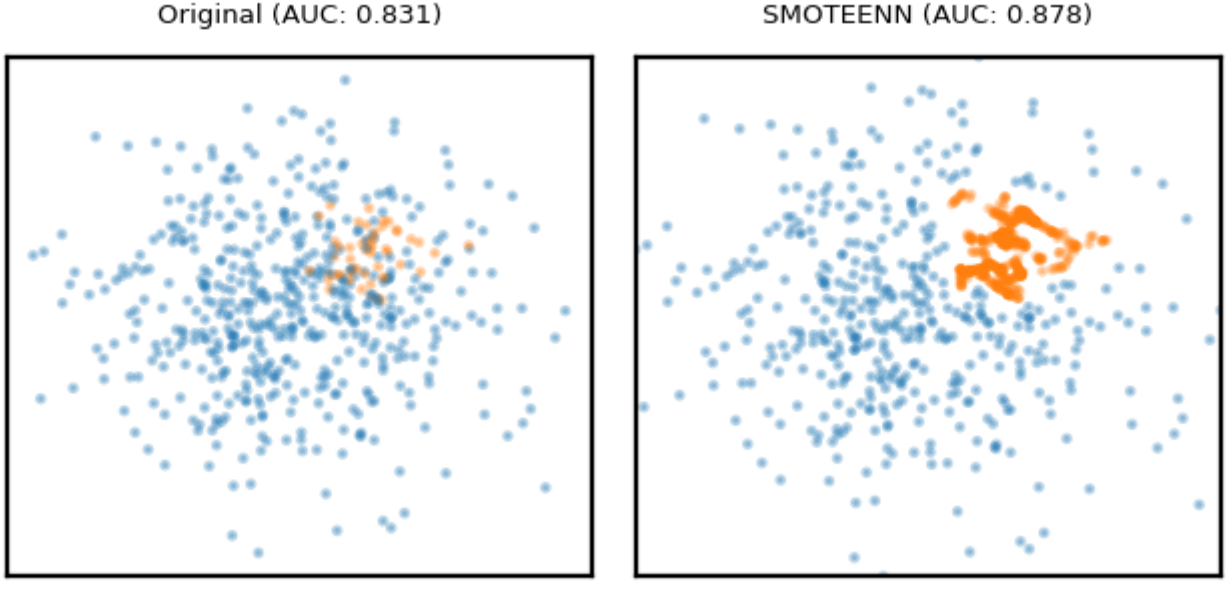
Synthetic Minority Oversampling Technique (SMOTE)

- Repeatedly choose a random minority point and a neighboring minority point
    - Pick a new, artificial point on the line between them (uniformly)
- May bias the data. Be careful never to create artificial points in the test set.
- ADASYN (Adaptive Synthetic)
    - Similar, but starts from 'hard' minority points (misclassified by kNN)



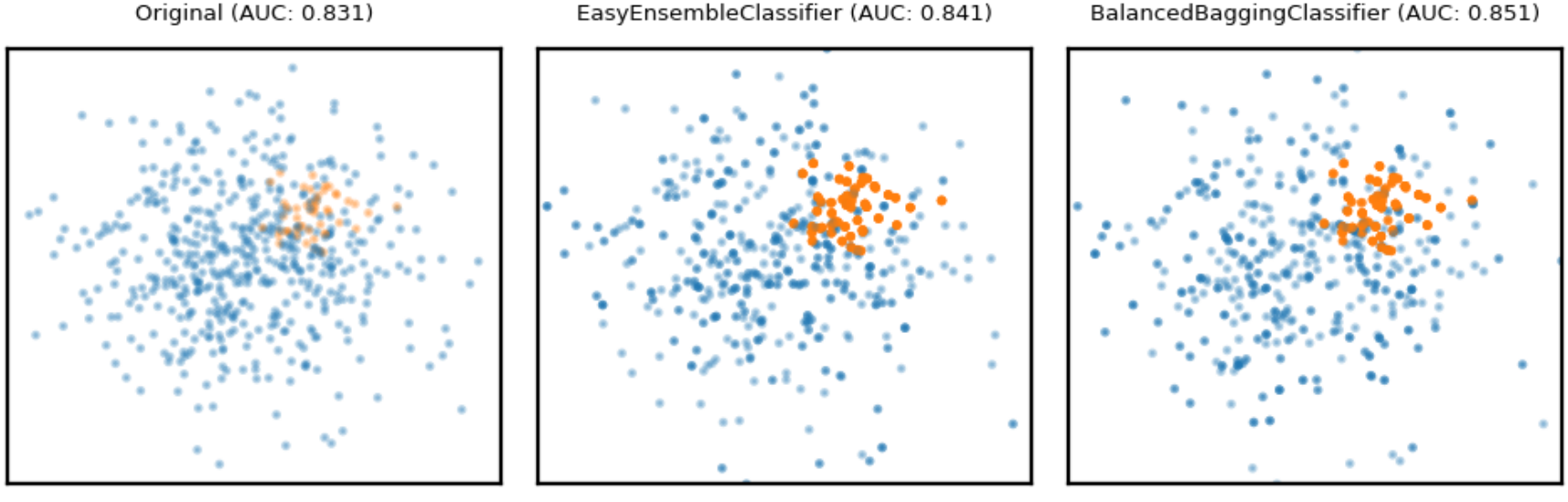
Combined techniques

- Combines over- and under-sampling
- E.g. oversampling with SMOTE, undersampling with Edited Nearest Neighbors (ENN)
    - SMOTE can generate 'noisy' point, close to majority class points
    - ENN will remove up these majority points to 'clean up' the space



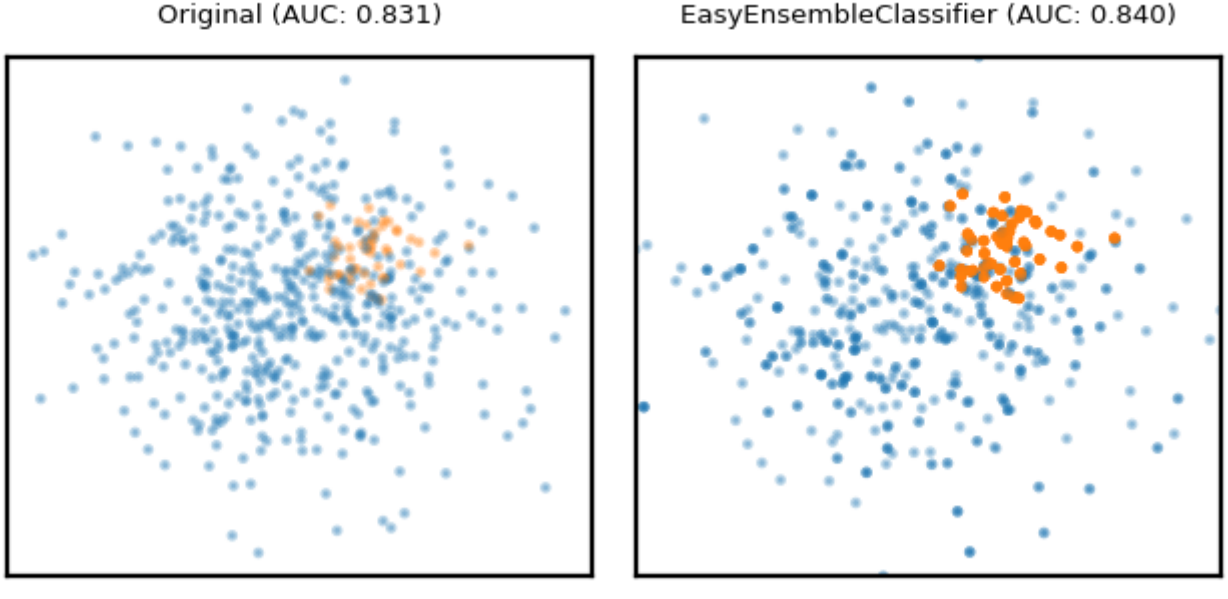
Ensemble Resampling

- Bagged ensemble of balanced base learners. Acts as a learner, not a preprocessor
- BalancedBagging: take bootstraps, randomly undersample each, train models (e.g. trees)
    - Benefits of random undersampling without throwing out so much data
- Easy Ensemble: take multiple random undersamplings directly, train models
    - Traditionally uses AdaBoost as base learner, but can be replaced



Comparison

- The best method depends on the data (amount of data, imbalance,...)
    - For a very large dataset, random undersampling may be fine
- You still need to choose the appropriate learning algorithms
- Don't forget about class weighting and prediction thresholding
    - Some combinations are useful, e.g. SMOTE + class weighting + thresholding



In practice (imblearn)

- Follows fit-sample paradigm (equivalent of fit-transform, but also affects y)
- Undersampling: RandomUnderSampler, EditedNearestNeighbours,...
- (Synthetic) Oversampling: RandomOverSampler, SMOTE, ADASYN,...
- Combinations: SMOTEENN,...

```
X_resampled, y_resampled = SMOTE(k_neighbors=5).fit_sample(X, y)
```

- Can be used in imblearn pipelines (not sklearn pipelines)
    - imblearn pipelines are compatible with GridSearchCV,...
    - Sampling is only done in `fit` (not in `predict`)

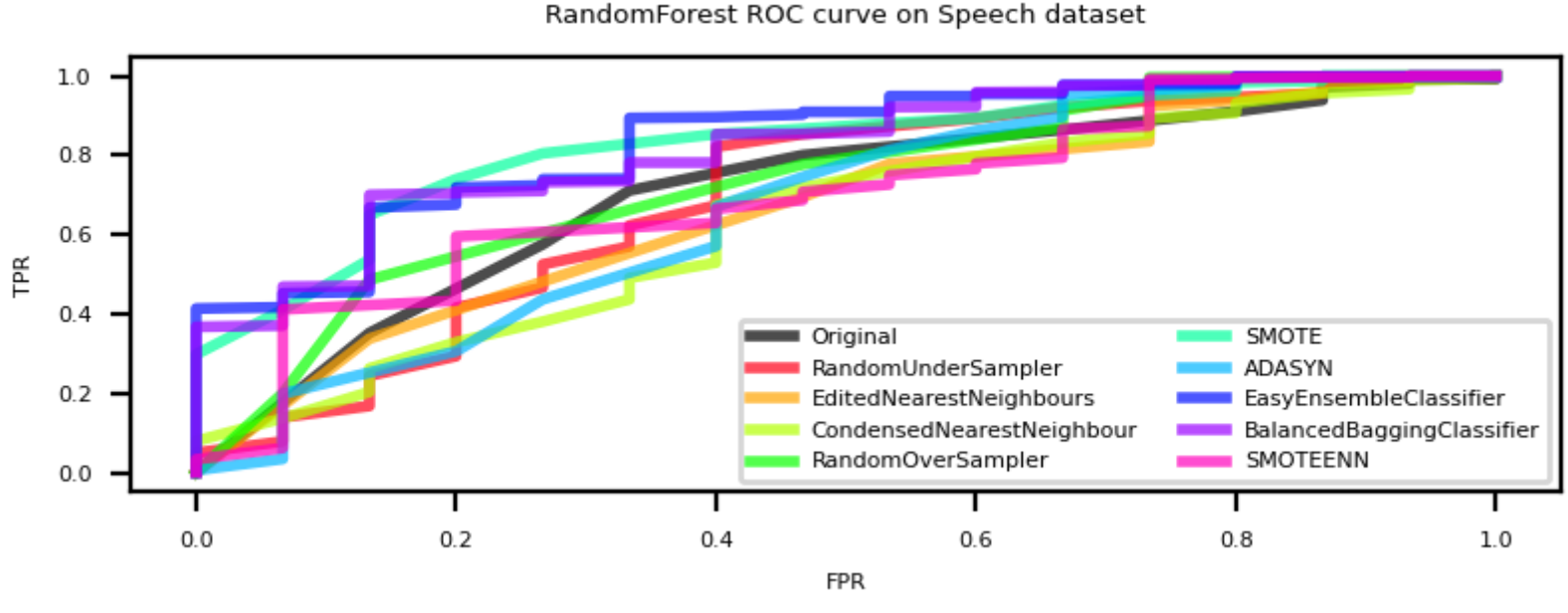
```
smote_pipe = make_pipeline(SMOTE(), LogisticRegression())
scores = cross_validate(smote_pipe, X_train, y_train)
param_grid = {"k_neighbors": [3,5,7]}
grid = GridSearchCV(smote_pipe, param_grid=param_grid, X, y)
```

- The ensembling techniques should be used as wrappers

```
clf = EasyEnsembleClassifier(base_estimator=SVC()).fit(X_train,
y_train)
```

Real-world data

- The effect of sampling procedures can be unpredictable
- Best method can depend on the data and FP/FN trade-offs
- SMOTE and ensembling techniques often work well



Summary

- Data preprocessing is a crucial part of machine learning
    - Scaling is important for many distance-based methods (e.g. kNN, SVM, Neural Nets)
    - Categorical encoding is necessary for numeric methods (or implementations)
    - Selecting features can speed up models and reduce overfitting
    - Feature engineering is often useful for linear models
    - It is often better to impute missing data than to remove data
    - Imbalanced datasets require extra care to build useful models
- Pipelines allow us to encapsulate multiple steps in a convenient way
    - Avoids data leakage, crucial for proper evaluation
- Choose the right preprocessing steps and models in your pipeline
    - Cross-validation helps, but the search space is huge
    - Smarter techniques exist to automate this process (AutoML)