Lecture 2: Linear models

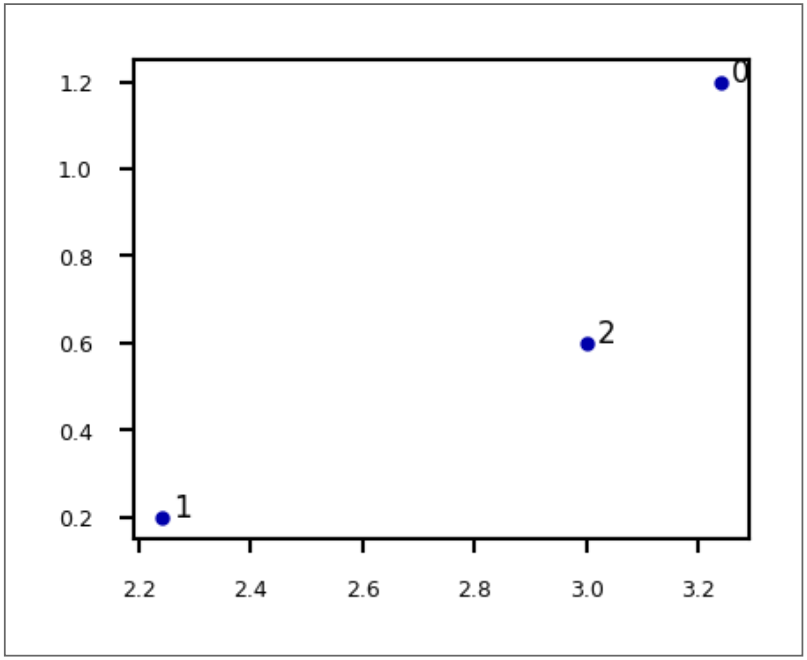
Basics of modeling, optimization, and regularization

Joaquin Vanschoren

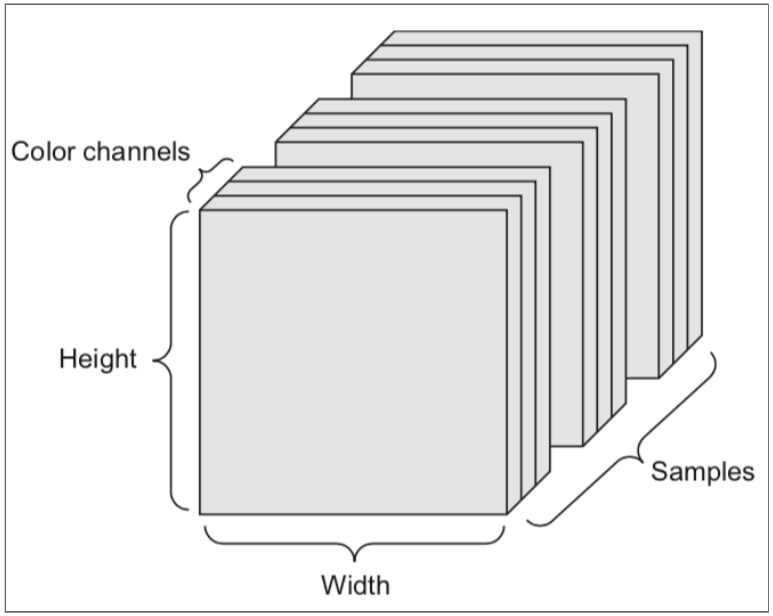
Notation and Definitions

- 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:

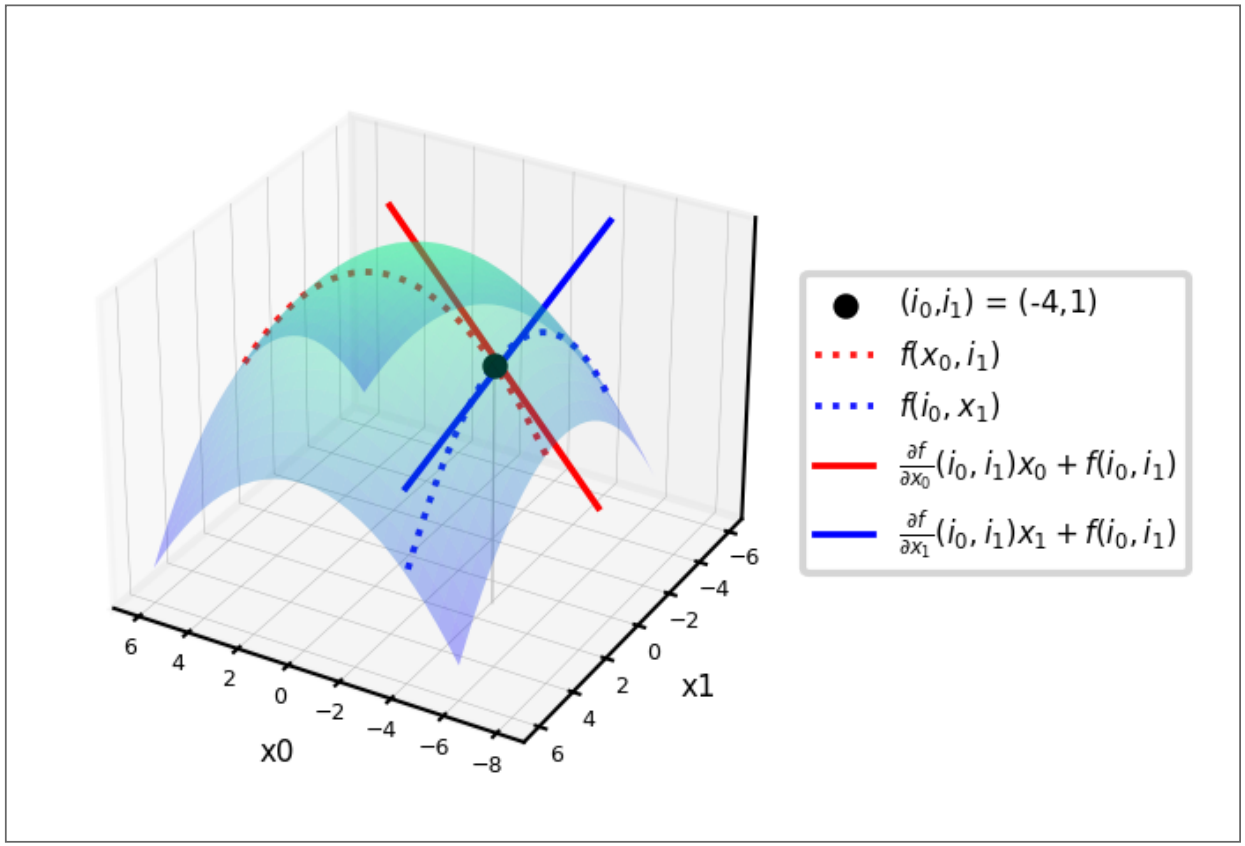
$$\sum_{i=0}^{p} = x_0 + x_1 + \ldots + x_p \quad \prod_{i=0}^{p} = x_0 \cdot x_1 \cdot \ldots \cdot x_p$$

- 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

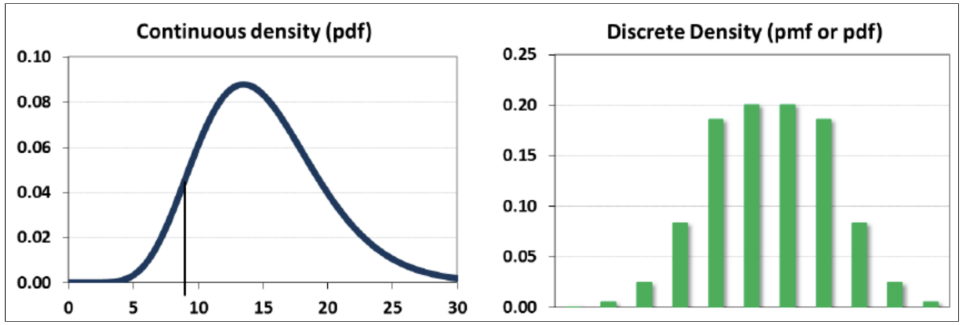
- A *derivative* f' of a function f describes how fast f grows or decreases
- The process of finding a derivative is called differentiation
    - Derivatives for basic functions are known
    - For non-basic functions we use the chain rule: $F(x) = f(g(x)) \rightarrow F'(x) = f'(g(x))g'(x)$
- A function 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 $\mathcal{L}$:

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

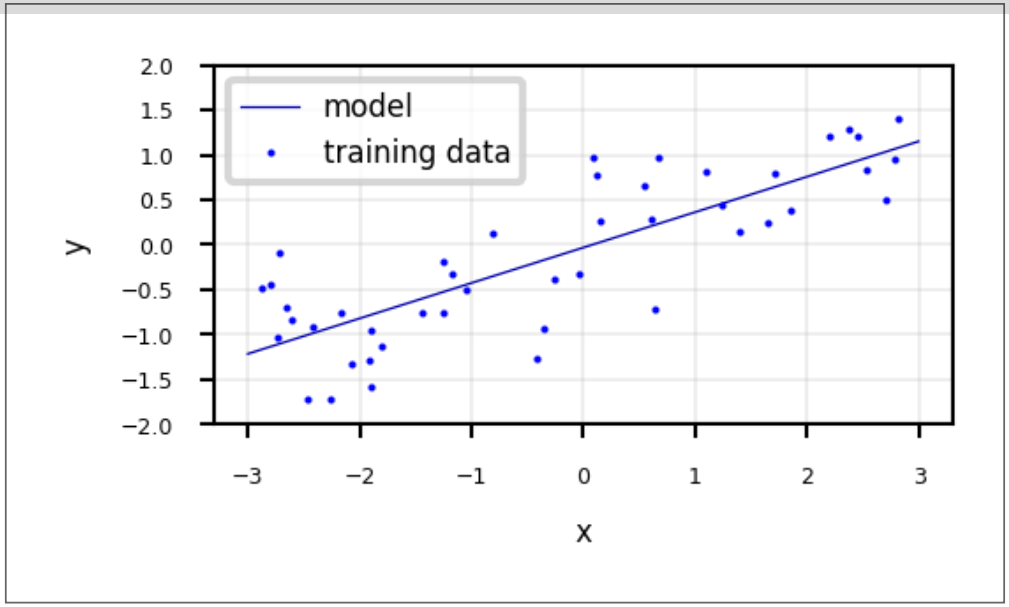
- Many algorithms with different $\mathcal{L}$: Least squares, Ridge, Lasso, Logistic Regression, Linear SVMs,...
- Can be very powerful (and fast), especially for large datasets with many features.
- Can be generalized to learn non-linear patterns: *Generalized Linear Models*
    - Features can be augmentented 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 ... w_p usually called *weights* or *coefficients* , w_0 the *bias* or *intercept*
  - Assumes that errors are $N(0, \sigma)$

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

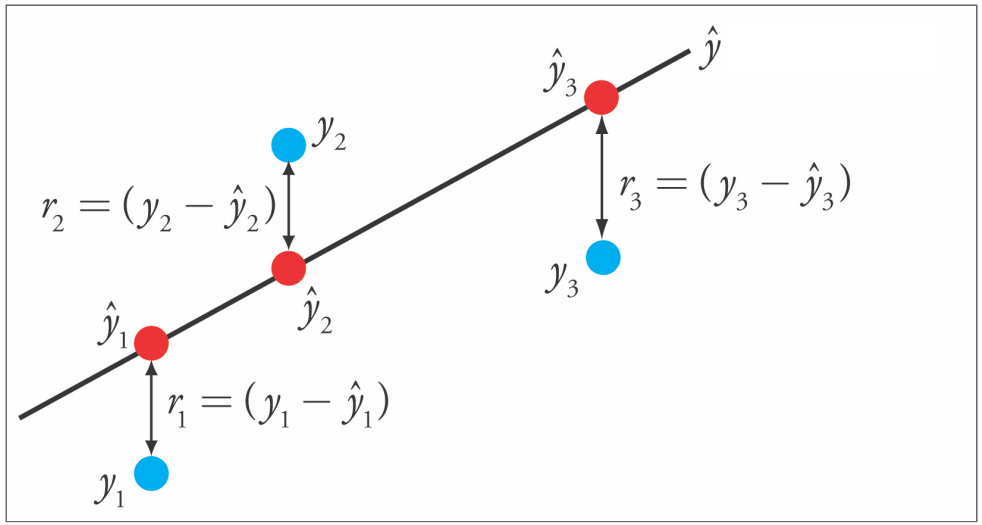
```
w_1: 0.393906  w_0: -0.031804
```



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:

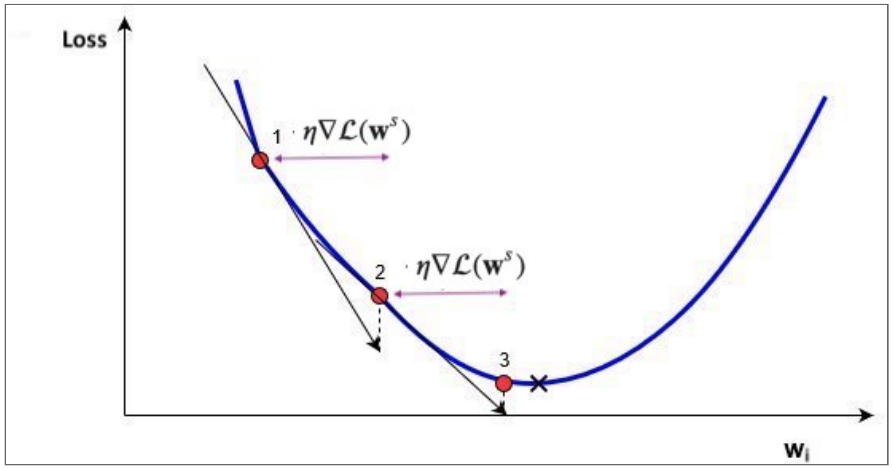
$$w^* = (X^T X)^{-1} X^T Y$$

  - 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

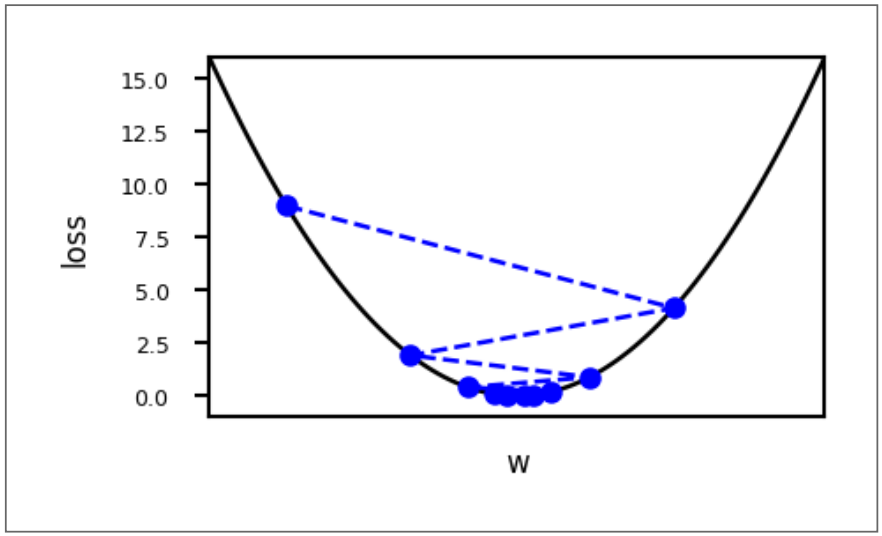
Gradient Descent

- 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):

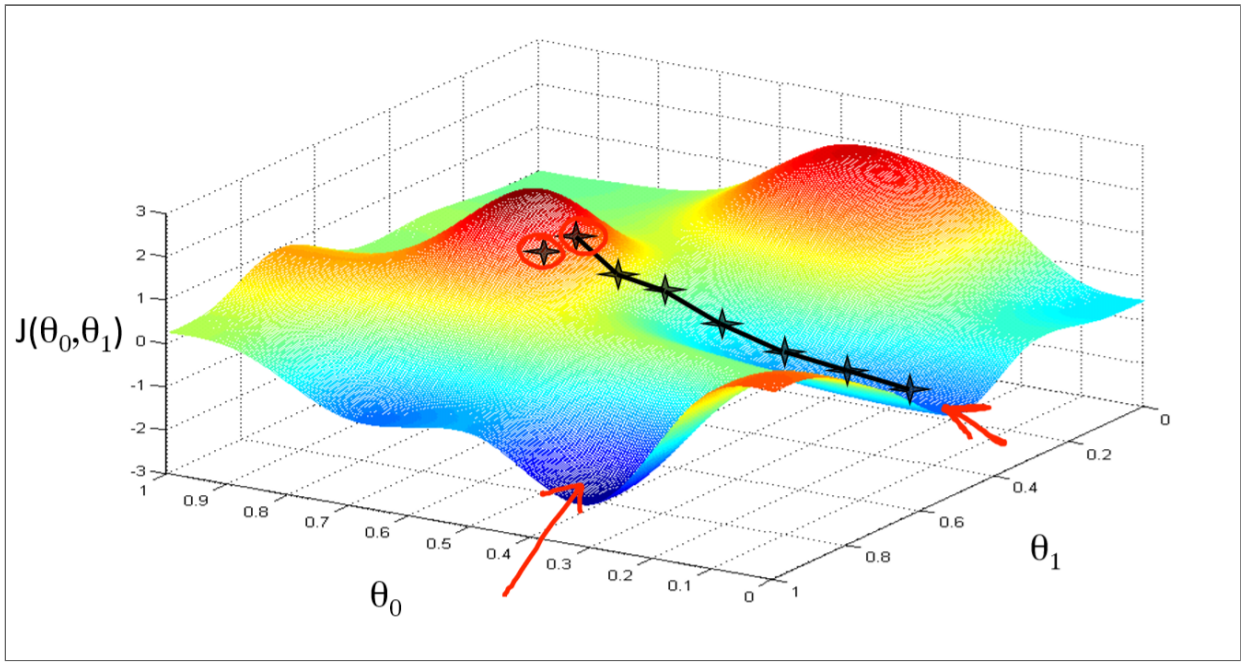
$$\mathbf{w}^{s+1} = \mathbf{w}^s - \eta\nabla\mathcal{L}(\mathbf{w}^s)$$



- 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

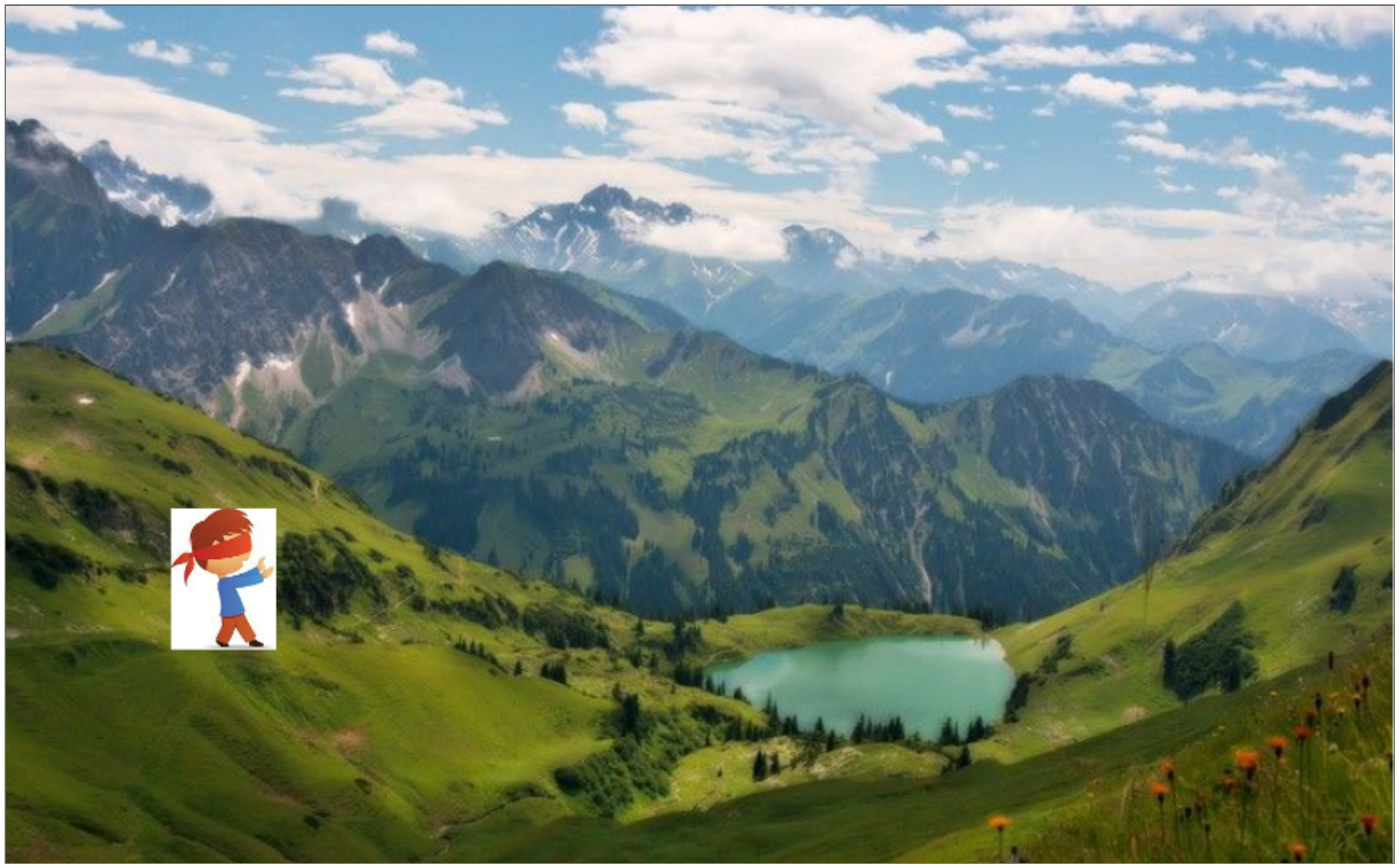


In two dimensions:



- 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

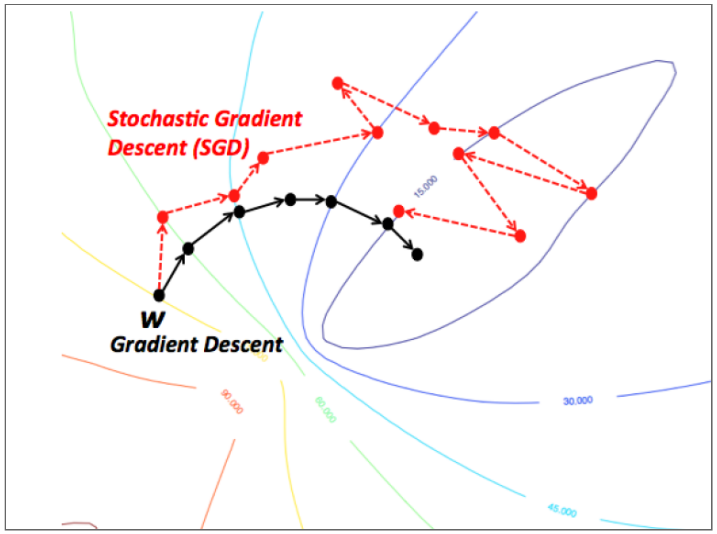
- Intuition: walking downhill using only the slope you "feel" nearby



(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): 30.93456367364078
```

```
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

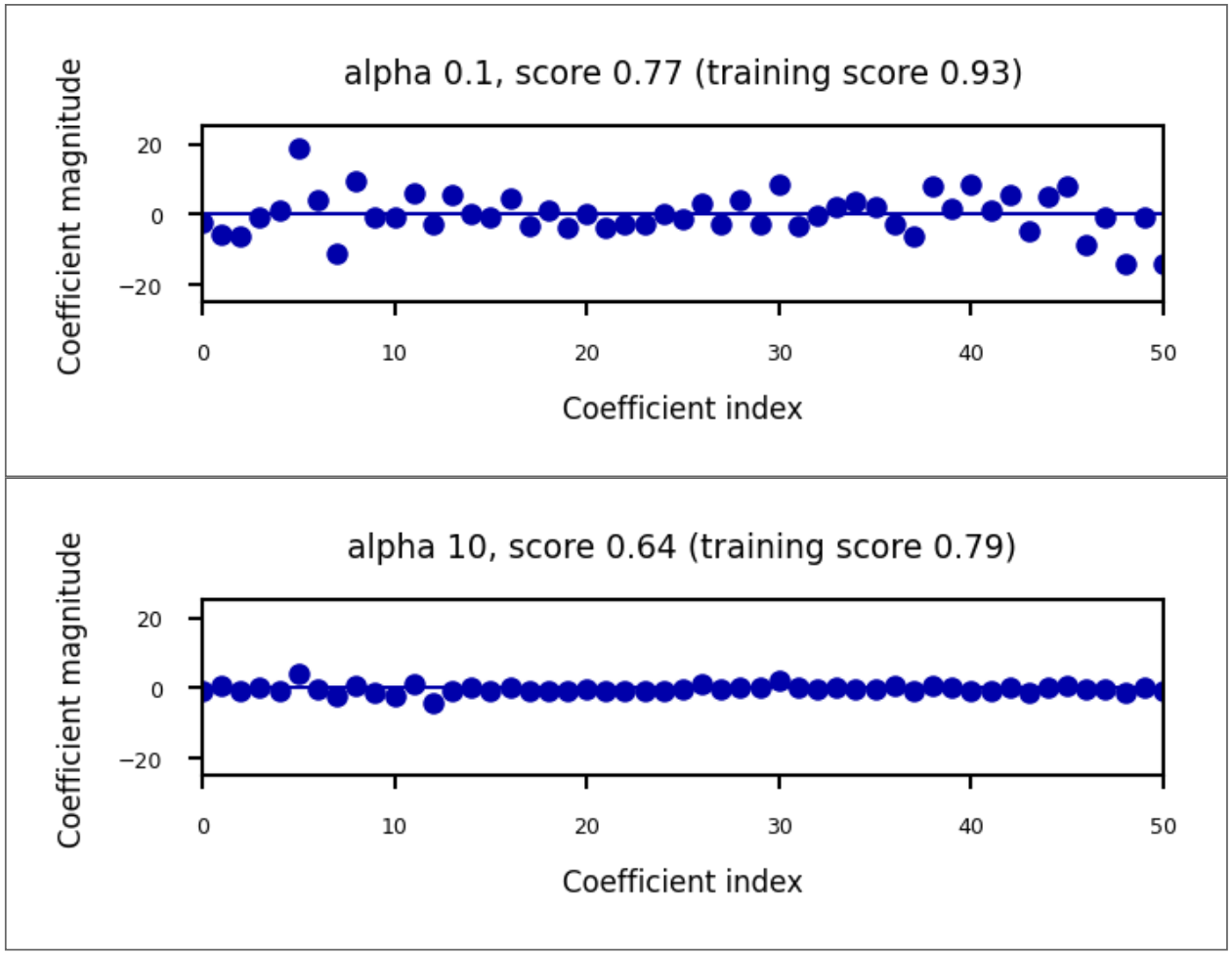
In practice

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

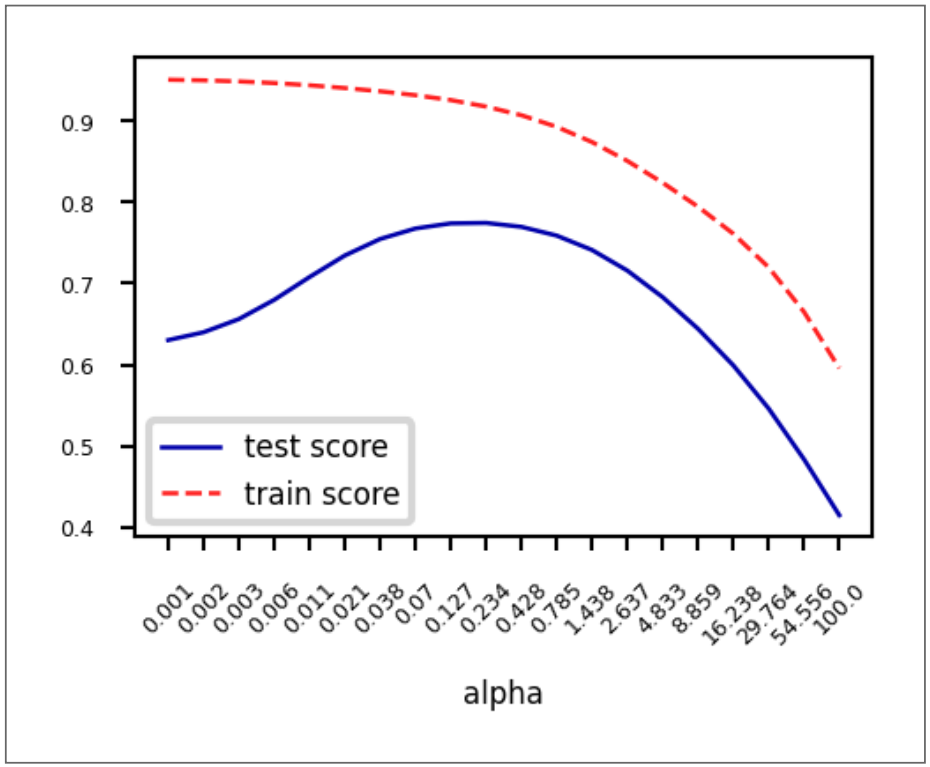
```
Weights (coefficients): [-1.414 -1.557 -1.465 -0.127 -0.079  8.332  0.255 -4.941  3.899 -1.059
 -1.584  1.051 -4.012  0.334  0.004 -0.849  0.745 -1.431 -1.63  -1.405
 -0.045 -1.746 -1.467 -1.332 -1.692 -0.506  2.622 -2.092  0.195 -0.275
  5.113 -1.671 -0.098  0.634 -0.61   0.04  -1.277 -2.913  3.395  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.

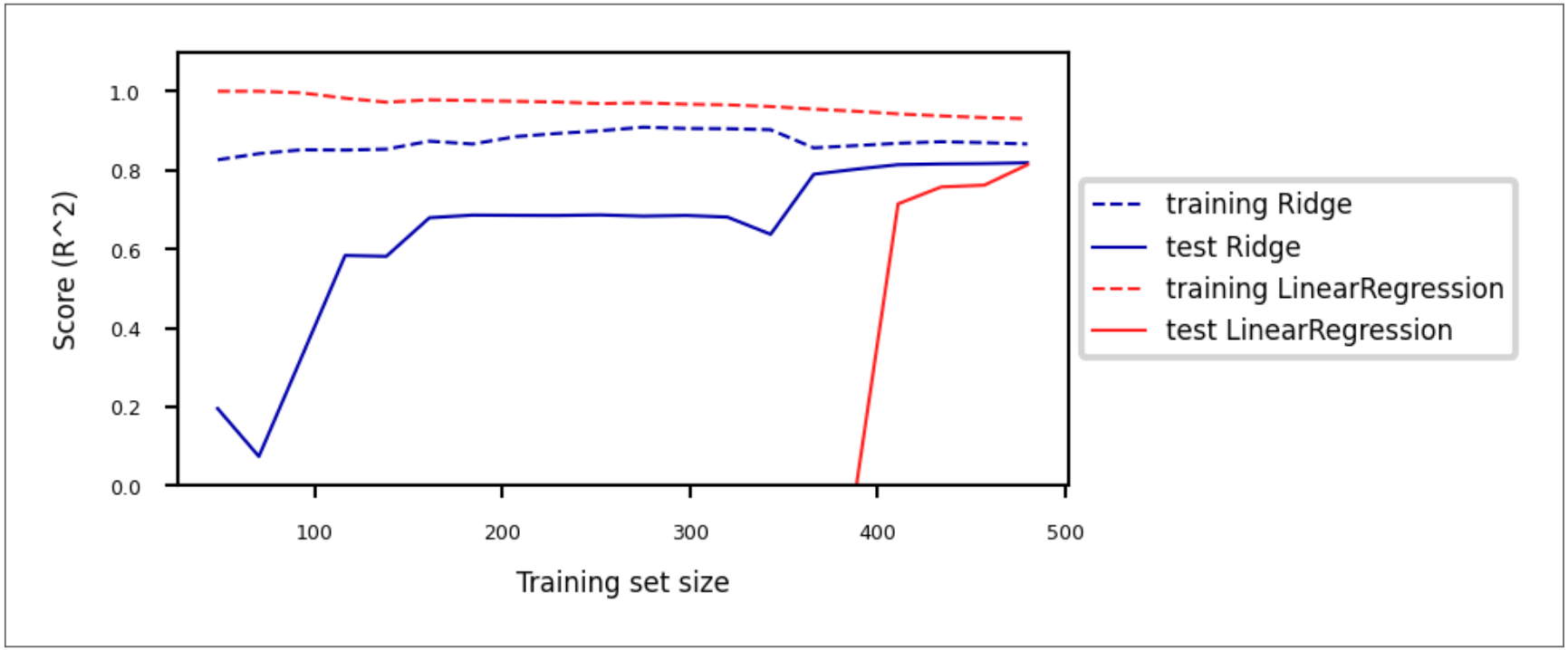


- When we plot the train and test scores for every α value, we see a sweet spot around $\alpha = 0.2$
    - Models with smaller α are overfitting
    - Models with larger α are underfitting



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)

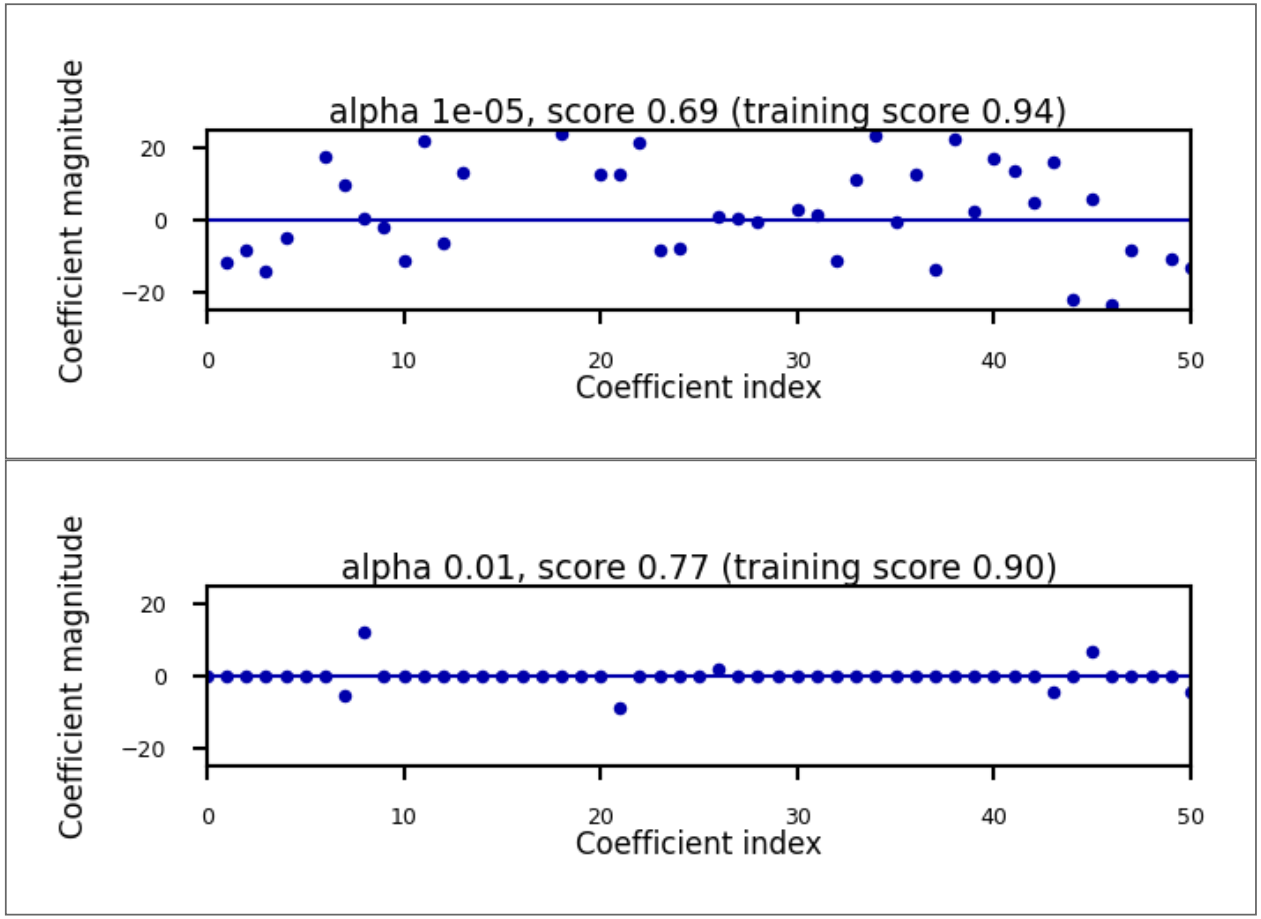
- Adds a different penalty term to the least squares sum:

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

- 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*

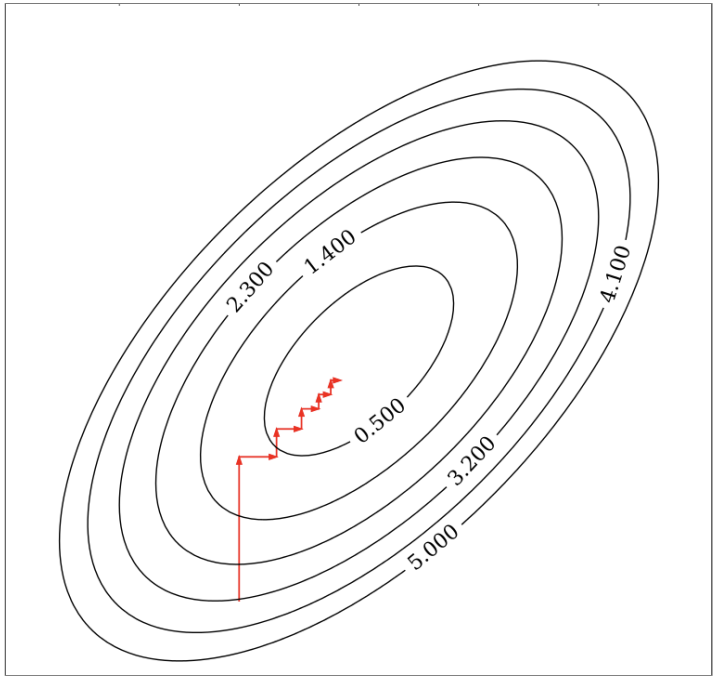
Analyze what happens to the weights:

- 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

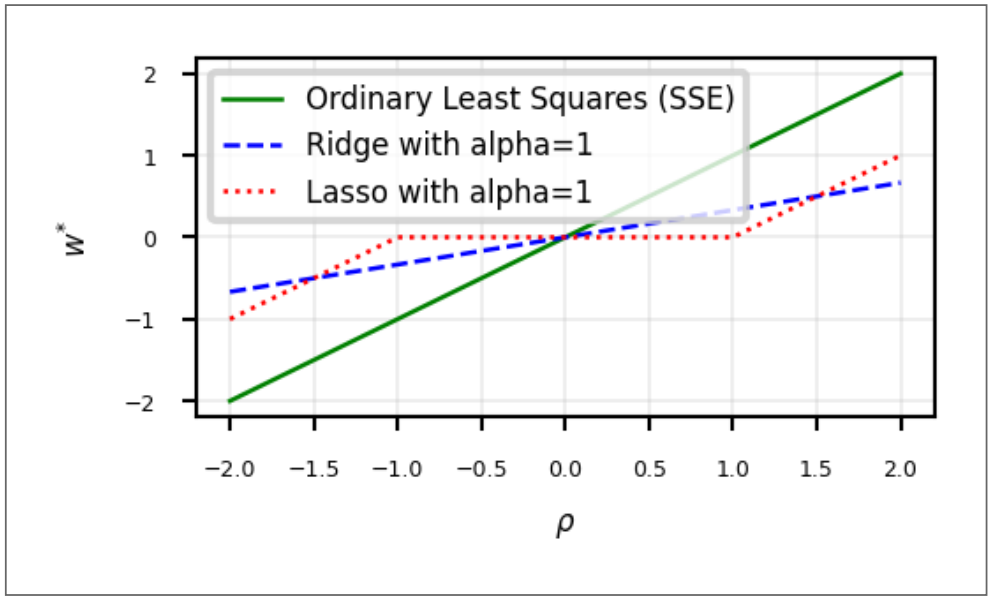
$$\begin{aligned} \partial_{w_i} \mathcal{L}_{Lasso}(w_i) &= \partial_{w_i} \mathcal{L}_{SSE}(w_i) + \partial_{w_i} \alpha |w_i| \\ 0 &= (w_i - \rho_i) + \alpha \cdot \partial_{w_i} |w_i| \\ w_i &= \rho_i - \alpha \cdot \partial_{w_i} |w_i| \end{aligned}$$

  - 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

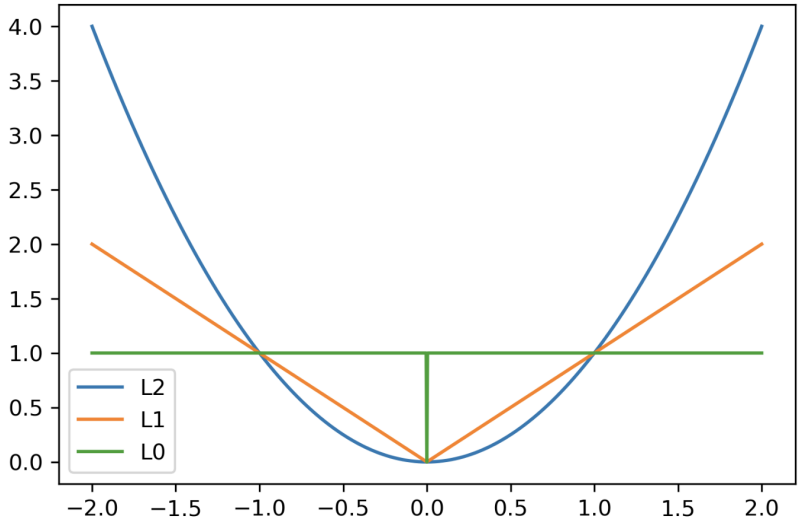
$$w_i^* = S(\rho_i, \alpha) = \begin{cases} \rho_i + \alpha, & \rho_i < -\alpha \\ 0, & -\alpha < \rho_i < \alpha \\ \rho_i - \alpha, & \rho_i > \alpha \end{cases}$$

  - 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



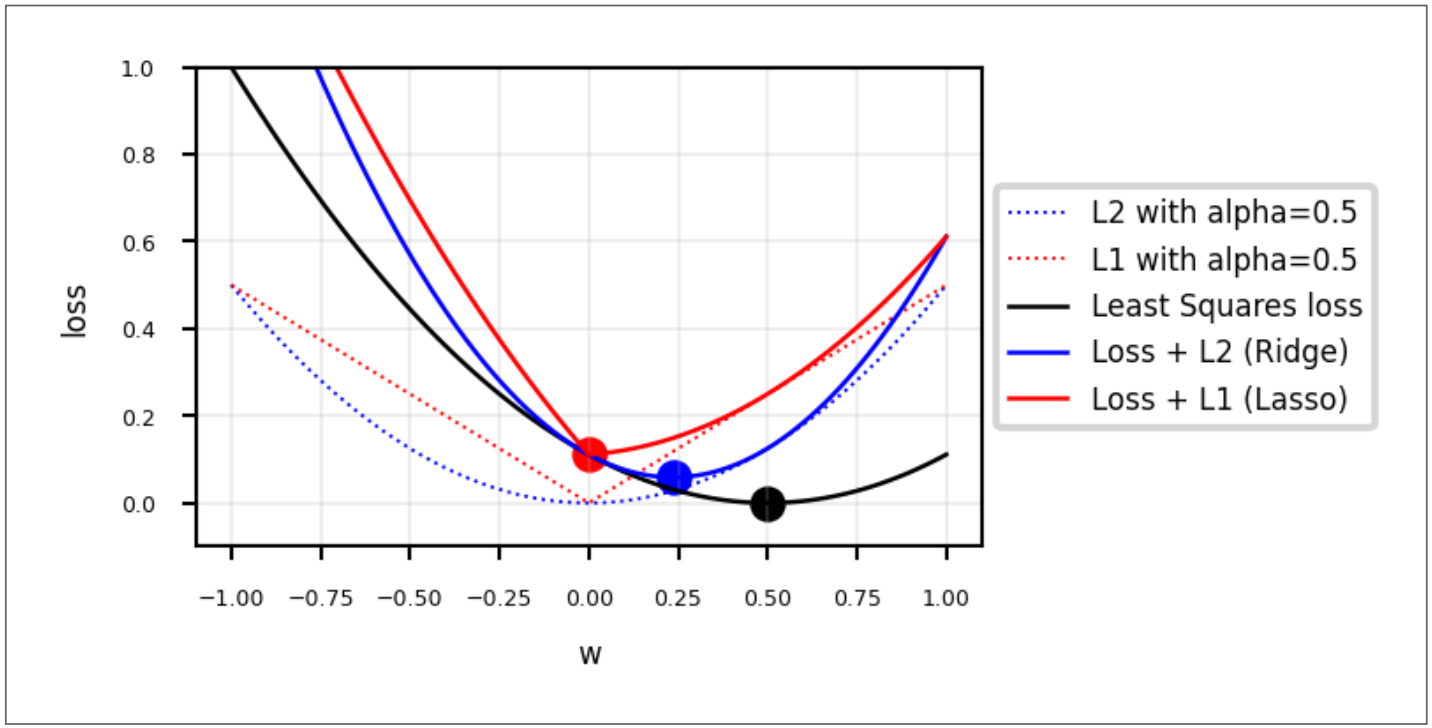
$$\ell_2(w) = \sqrt{\sum_i w_i^2}$$

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

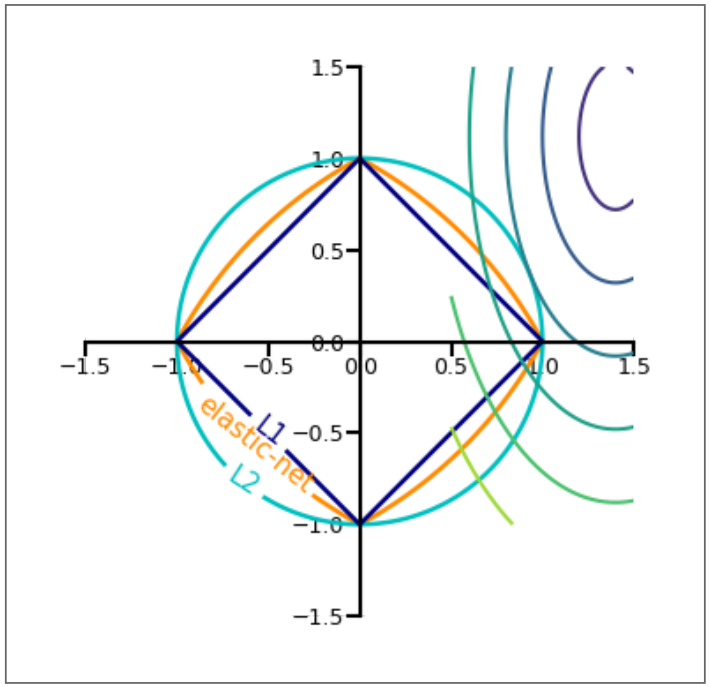
$$\ell_0(w) = \sum_i 1_{w_i != 0}$$

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

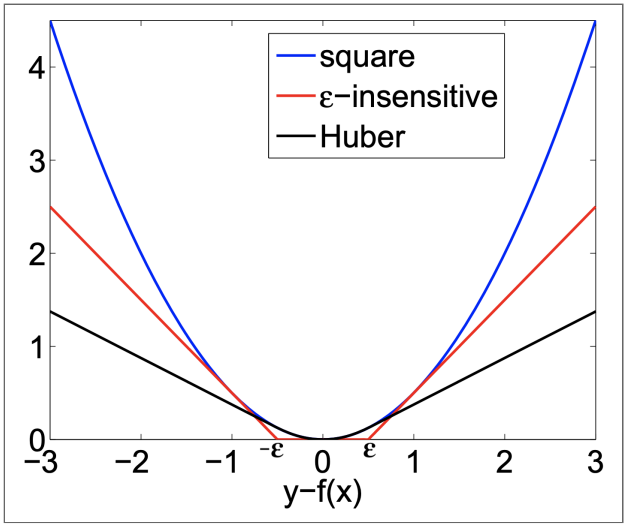
- Adds both L1 and L2 regularization:

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

- 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

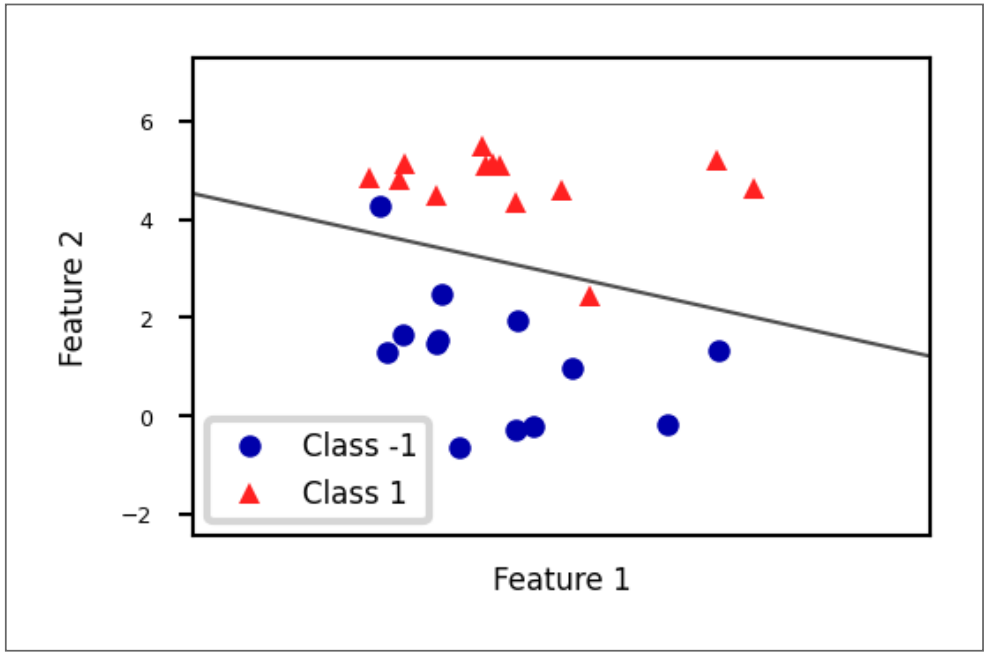


Linear models for Classification

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



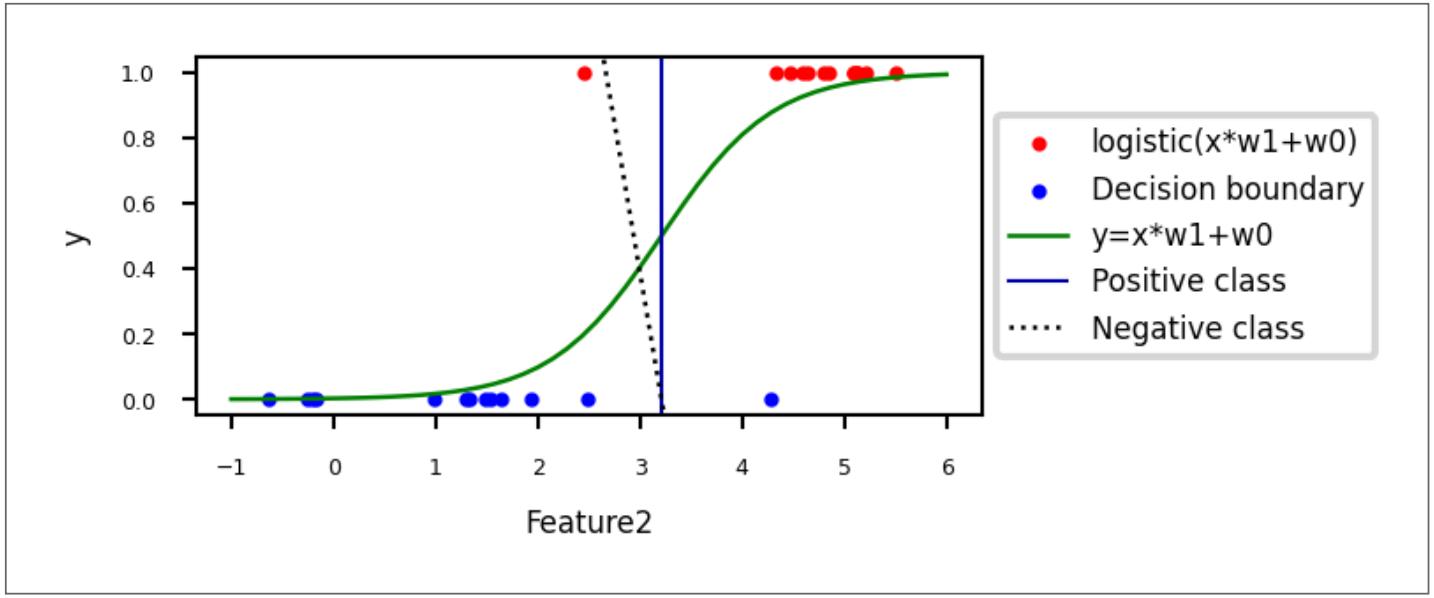
- There are many algorithms for linear classification, differing in loss function, regularization techniques, and optimization method
- Most common techniques:
    - Convert target classes {neg,pos} to {0,1} and treat as a regression task
        - Logistic regression (Log loss)
        - Ridge Classification (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

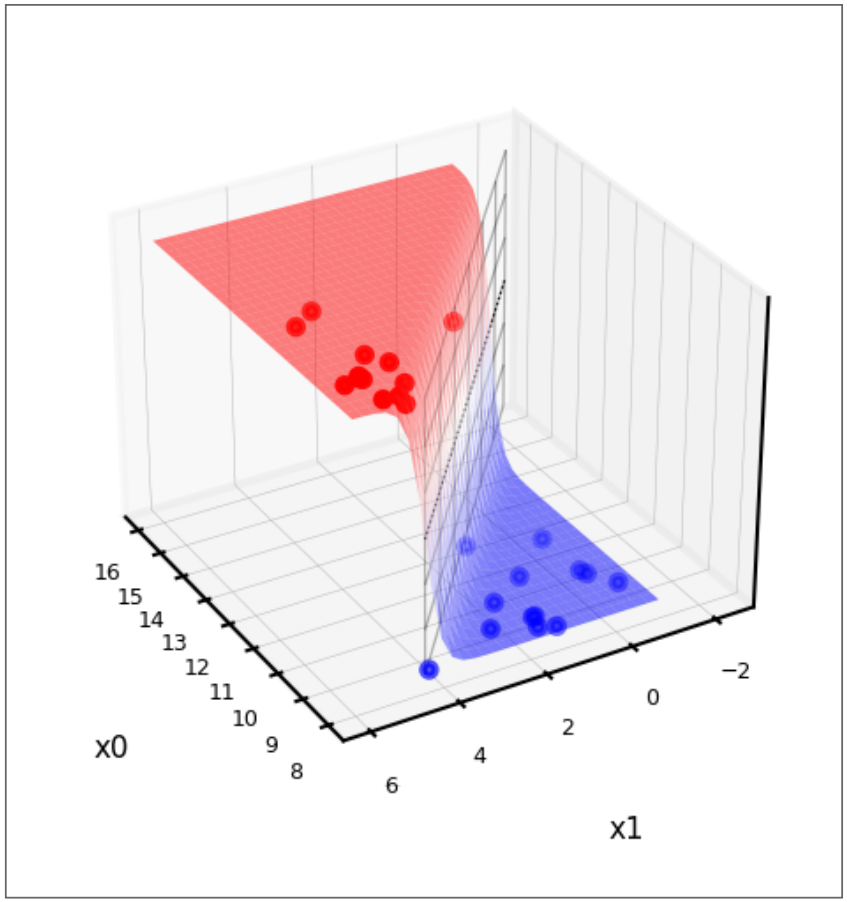
- Aims to predict the *probability* that a point belongs to the positive class
- Converts target values {negative (blue), positive (red)} to {0,1}
- Fits a *logistic* (or *sigmoid* or *S* curve) function through these points
    - Maps (-Inf,Inf) to a probability [0,1]

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

- 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)

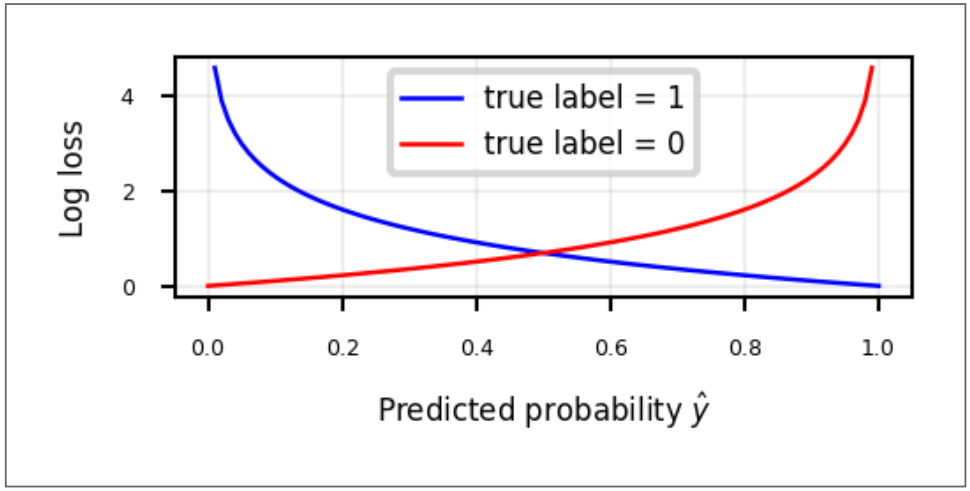


Loss function: Cross-entropy

- 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: $\mathcal{L}_{log}{}'(\mathbf{w}) = \mathcal{L}_{log}(\mathbf{w}) + \alpha \sum_i w_i^2$



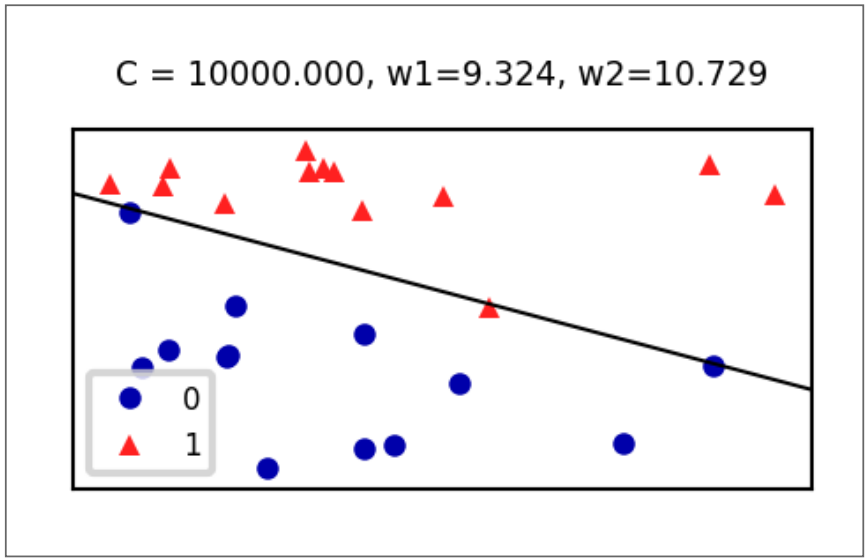
Optimization methods (solvers) for cross-entropy loss

- 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

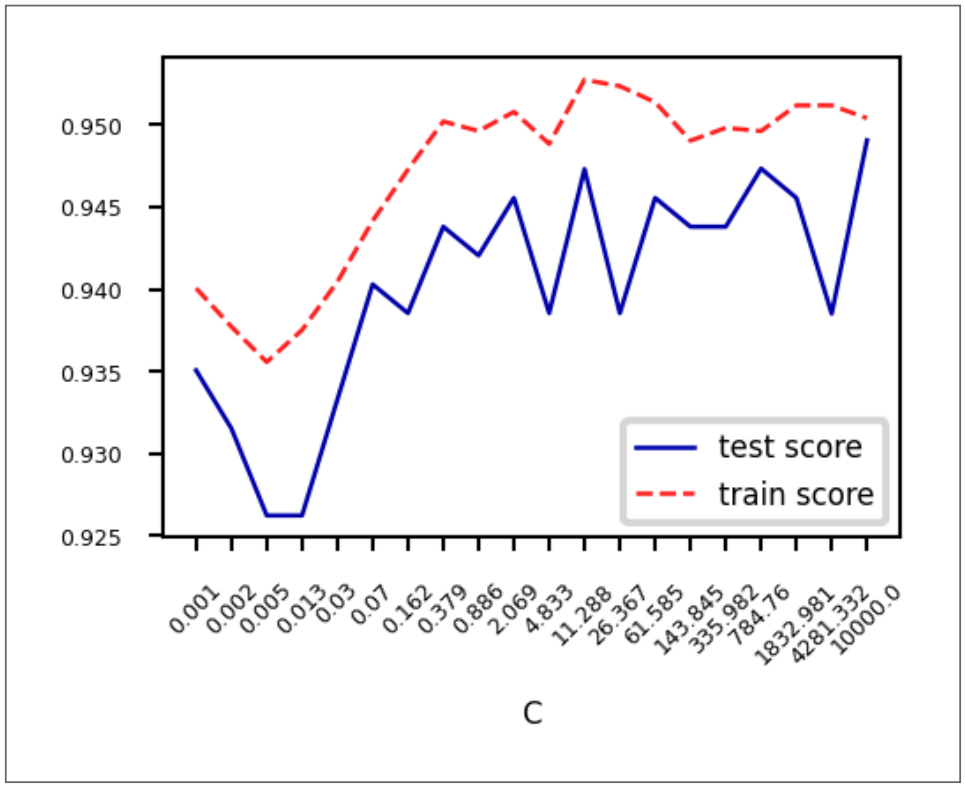
In practice

- 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

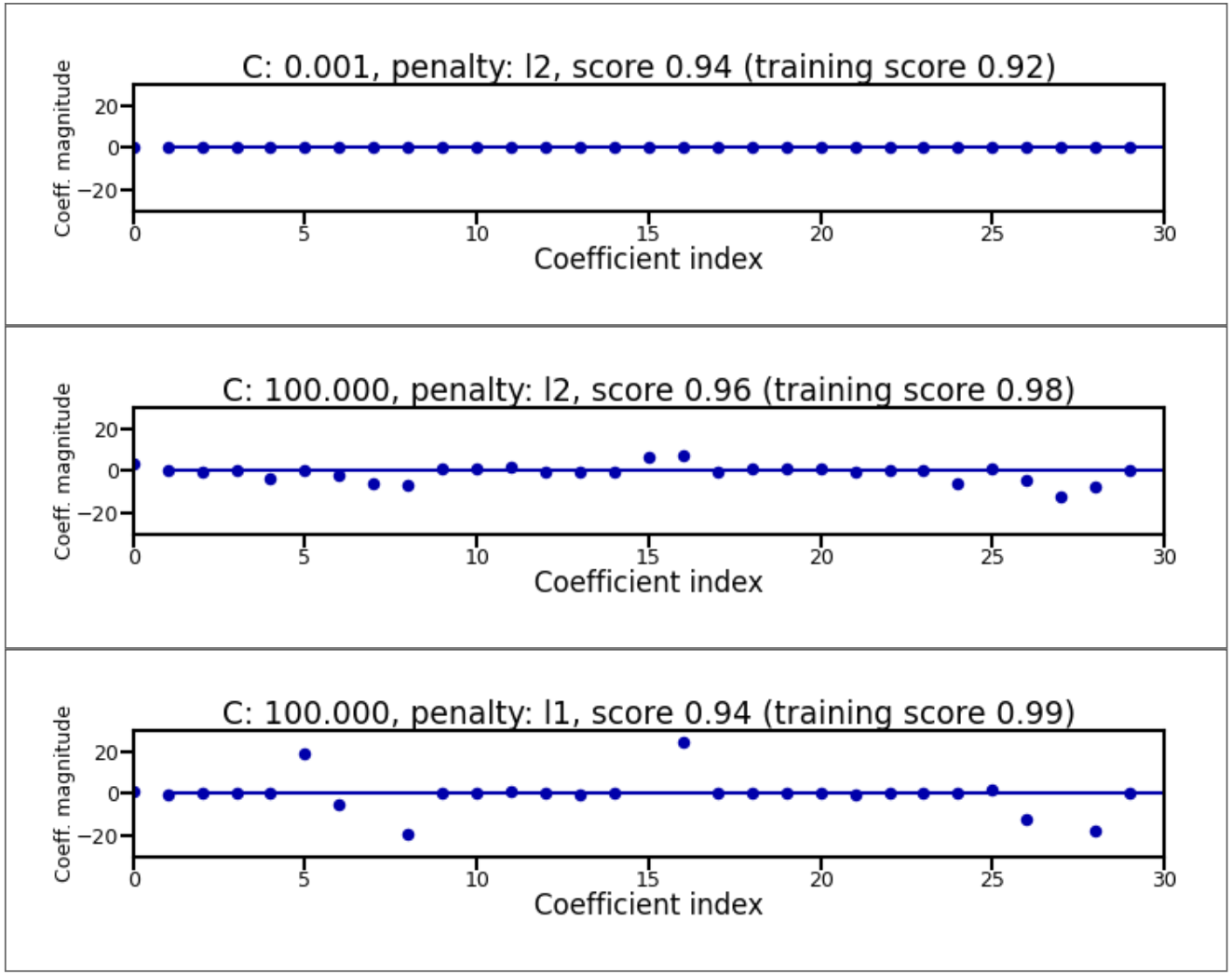
```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(C=1).fit(X_train, y_train)
```



- 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

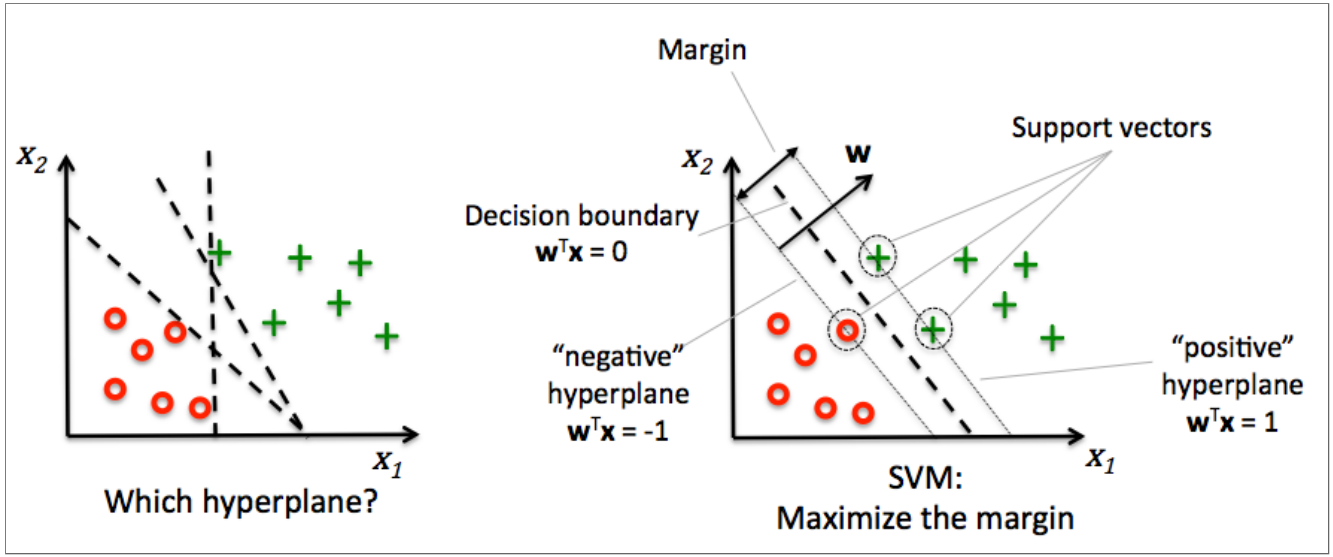
- Instead of log loss, we can also use ridge loss:

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

- 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

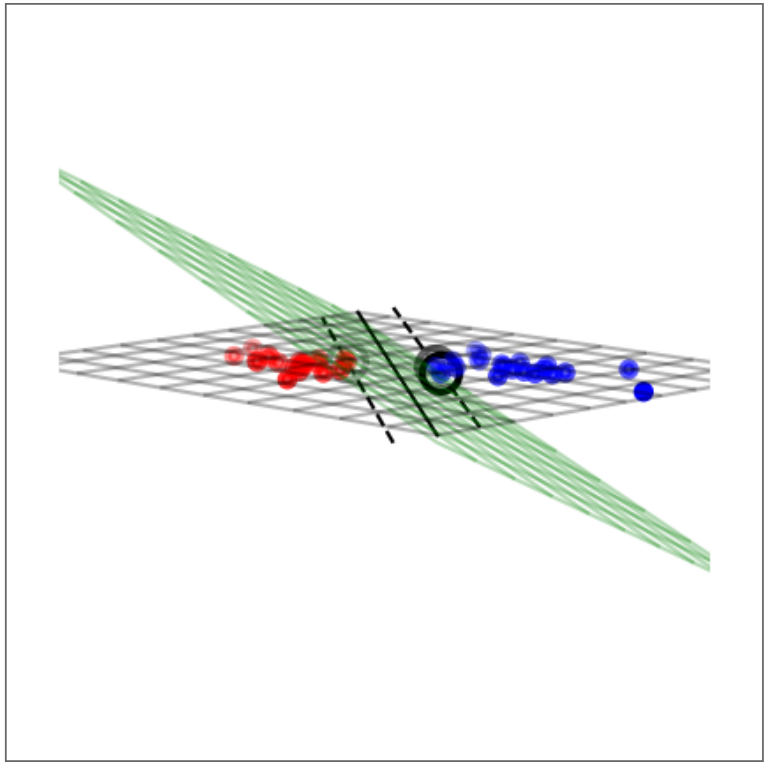
Support vector machines

- 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

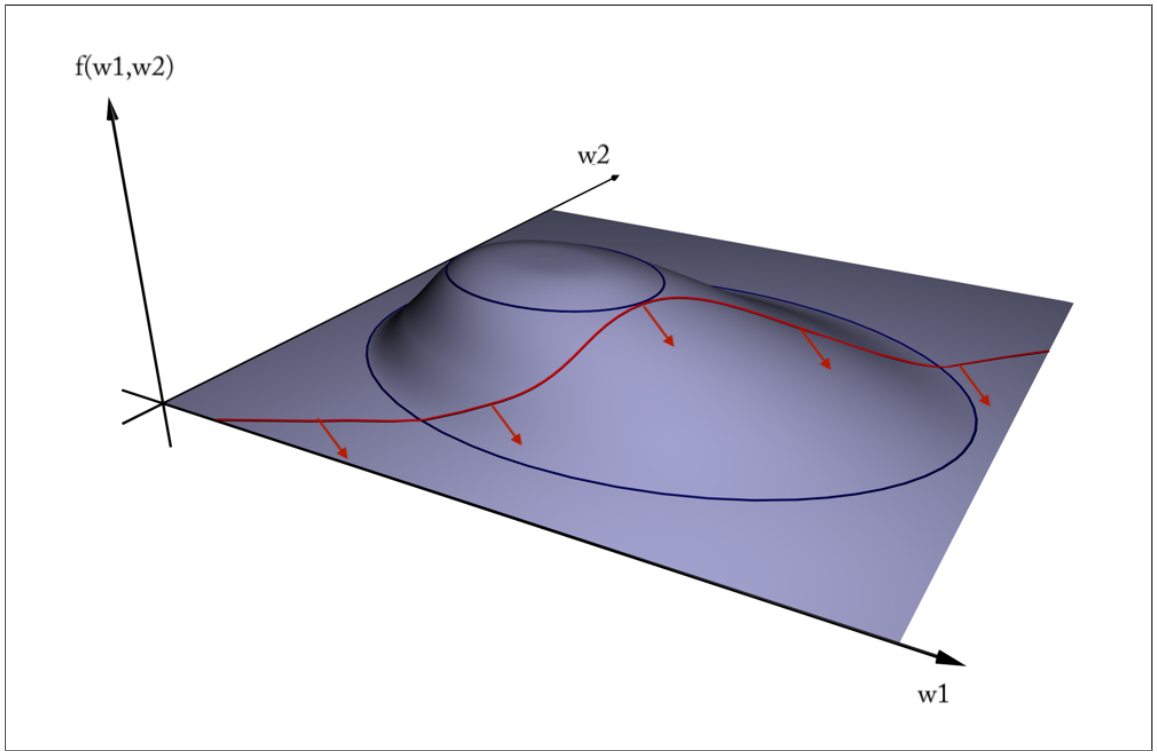
- Imagine a hyperplane (green) $y = \sum_1^p \mathbf{w}_i * \mathbf{x}_i + w_0$ that has slope $\mathbf{w}$, value '+1' for the positive (red) support vectors, and '-1' for the negative (blue) ones
    - Margin between the boundary and support vectors is $\frac{y - w_0}{||\mathbf{w}||}$, with $||\mathbf{w}|| = \sum_i^p w_i^2$
    - 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):

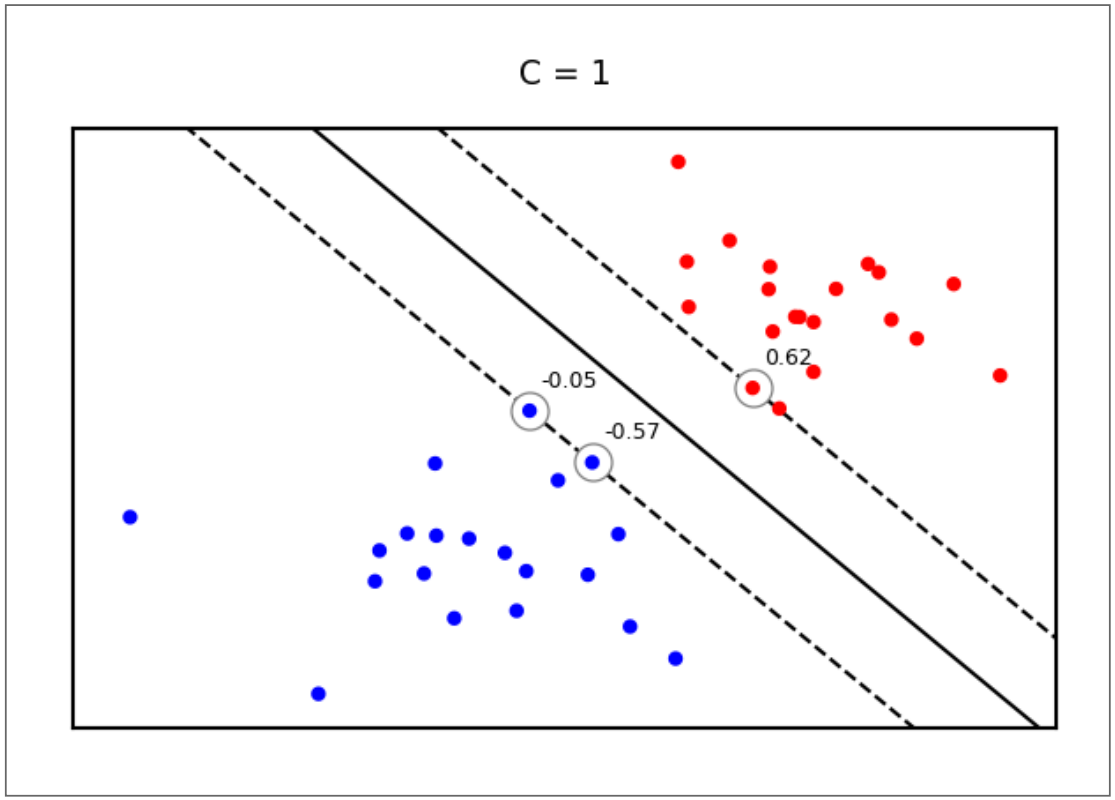
$$\mathcal{L}_{Dual} = \sum_{i=1}^{l} a_i - \frac{1}{2} \sum_{i,j=1}^{l} a_i a_j y_i y_j (\mathbf{x_i x_j})$$

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:

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

- *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*

Regularized (soft margin) SVMs

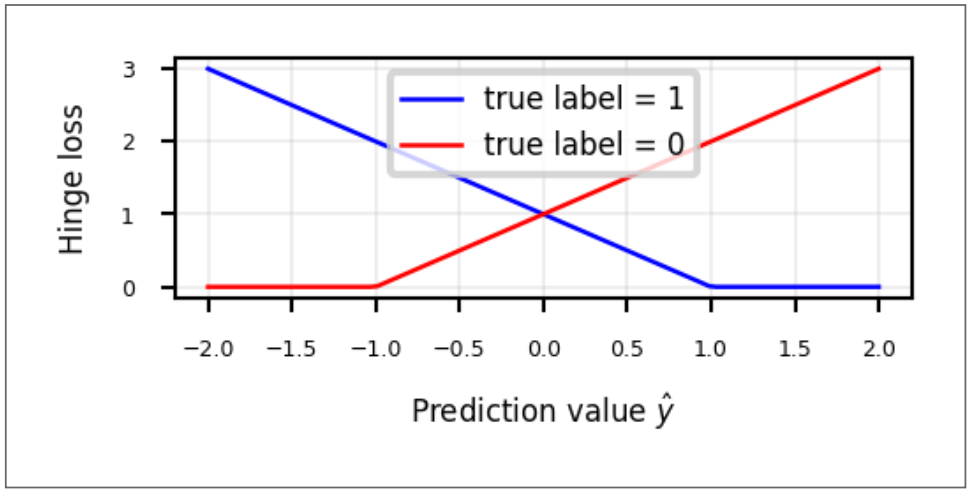
- If the data is not linearly separable, (hard) margin maximization becomes meaningless
- Relax the contraint by allowing an error ξ_i: $y_i(\mathbf{wx_i} + w_0) \geq 1 - \xi_i$
- Or (since $\xi_i \geq 0$):

$$\xi_i = max(0, 1 - y_i \cdot (\mathbf{wx_i} + w_0))$$

- 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

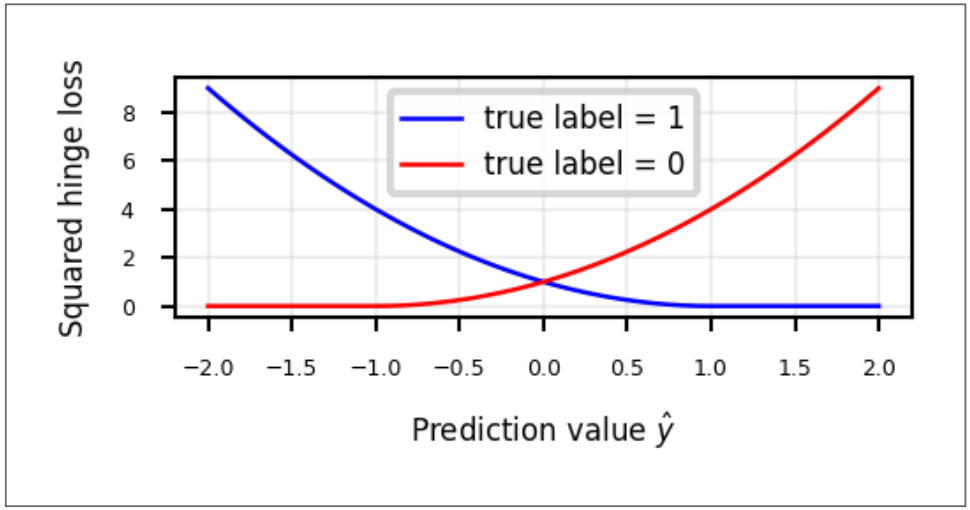


Least Squares SVMs

- We can also use the *squares* of all the errors, or squared hinge loss: $\sum_i^n \xi_i^2$
- This yields the Least Squares SVM objective

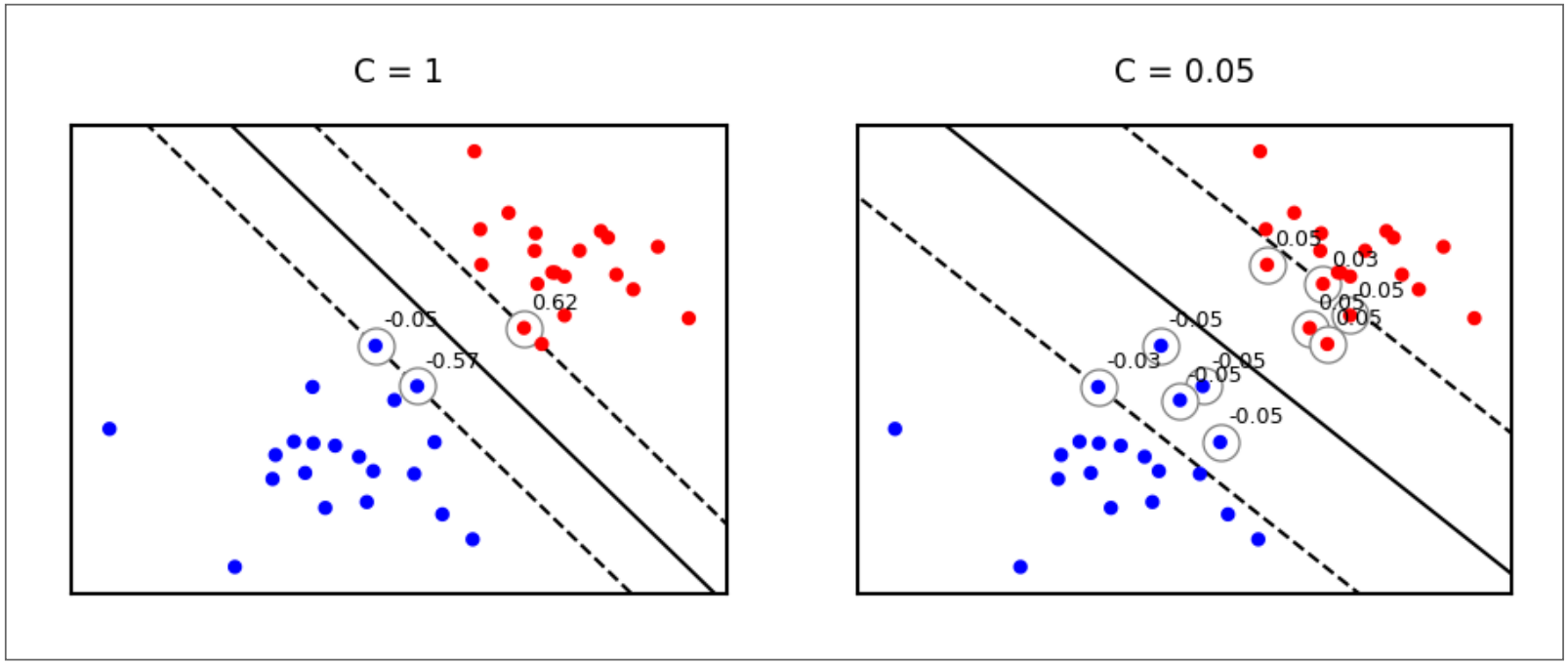
$$\mathcal{L}(\mathbf{w}) = ||\mathbf{w}||^2 + C\sum_i^n \xi_i^2$$

- Can be solved with Lagrangian Multipliers and a set of linear equations
    - Still yields support vectors and still allows kernelization
    - Support vectors are not sparse, but pruning techniques exist

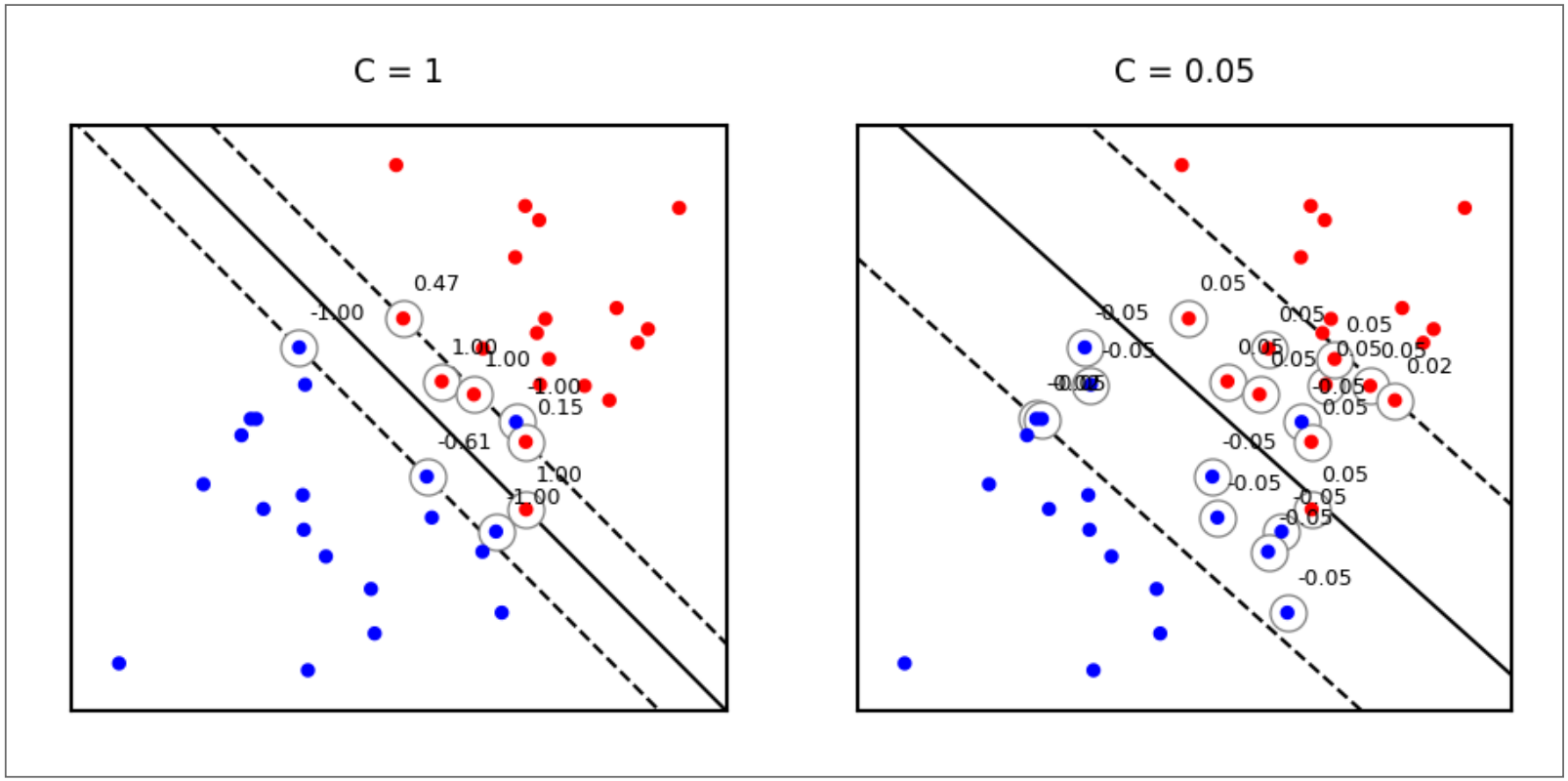


Effect of regularization on margin and support vectors

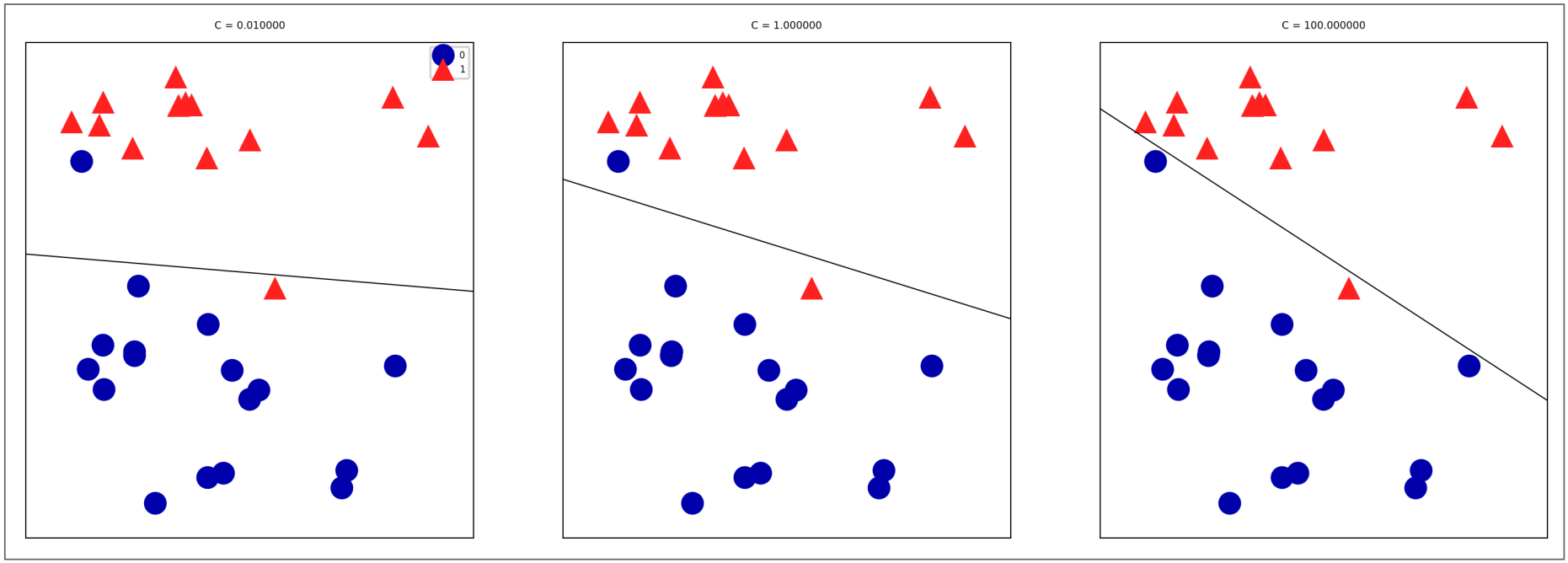
- SVM's Hinge loss acts like L1 regularization, yields sparse models
- C is the *inverse* regularization strength (inverse of α in Lasso)
    - Larger C: fewer support vectors, smaller margin, more overfitting
    - Smaller C: more support vectors, wider margin, less overfitting
- Needs to be tuned carefully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



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` with `kernel=linear`: allows *kernelization* (see later)
  - 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')
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]]
```

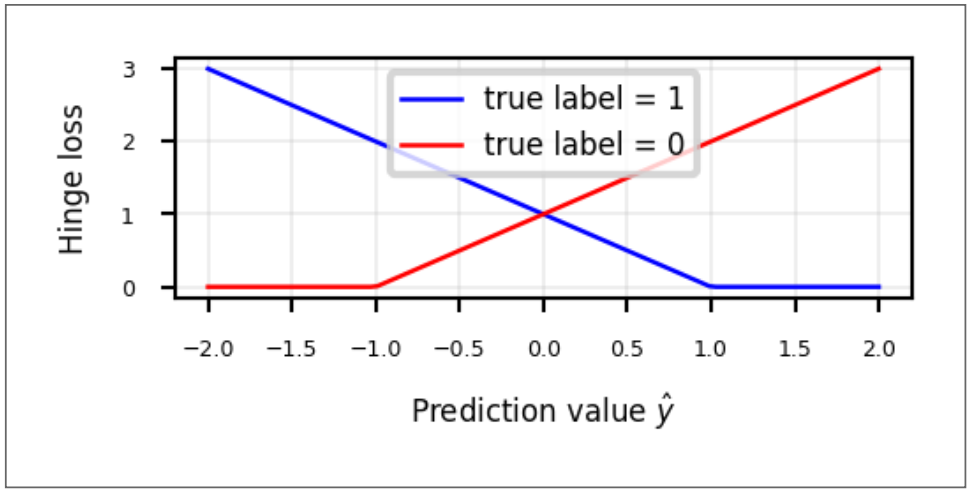
Solving SVMs with Gradient Descent

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

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

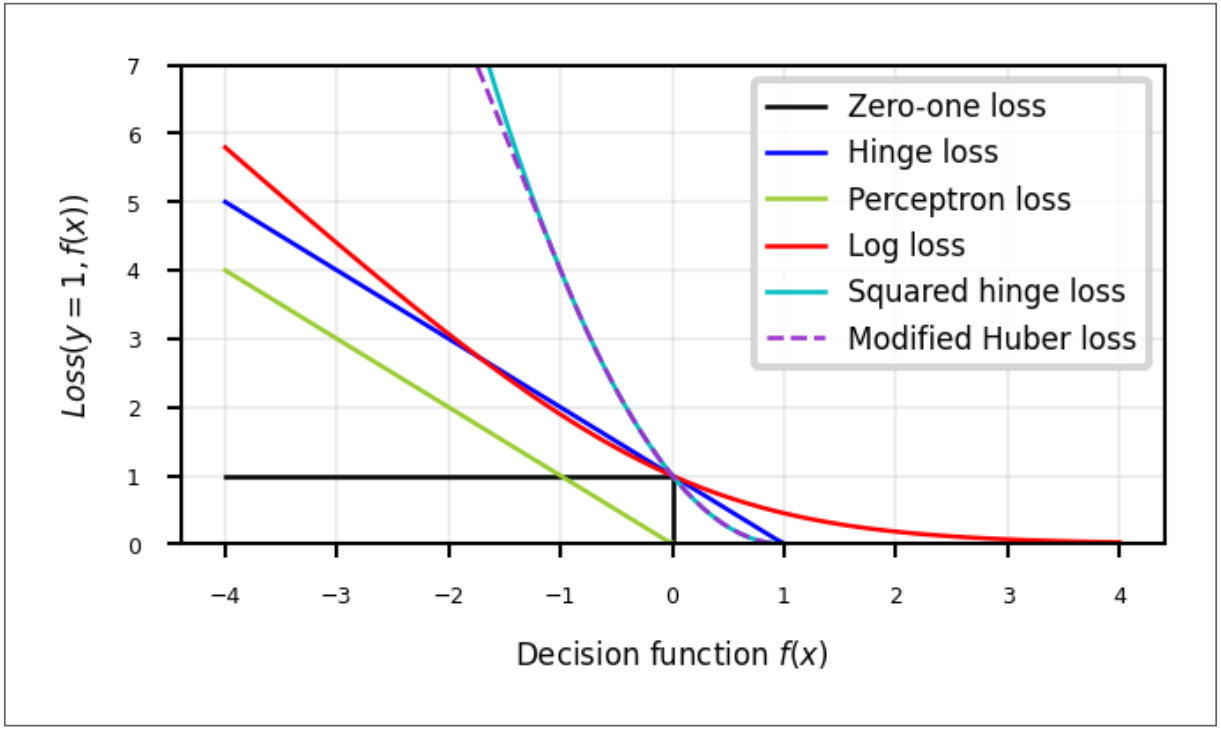
$$\frac{\partial \mathcal{L}_{Hinge}}{\partial w_i} = \begin{cases} -y_i x_i & y_i(\mathbf{wx_i} + w_0) < 1 \\ 0 & \text{otherwise} \end{cases}$$

- Can be solved with (stochastic) gradient descent



Generalized SVMs

- Because the derivative of hinge loss is undefined at y=1, smoothed versions are often used:
    - Squared hinge loss: yields *least squares SVM*
        - Equivalent to Ridge classification (with different solver)
    - Modified Huber loss: squared hinge, but linear after -1. Robust against outliers
- Log loss can also be used (equivalent to logistic regression)
- In sklearn, `SGDClassifier` can be used with any of these. Good for large datasets.

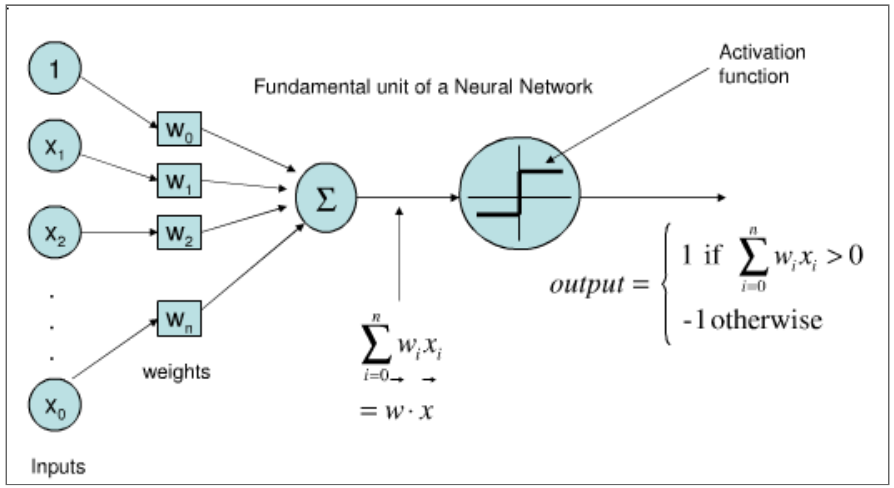


Perceptron

- 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

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

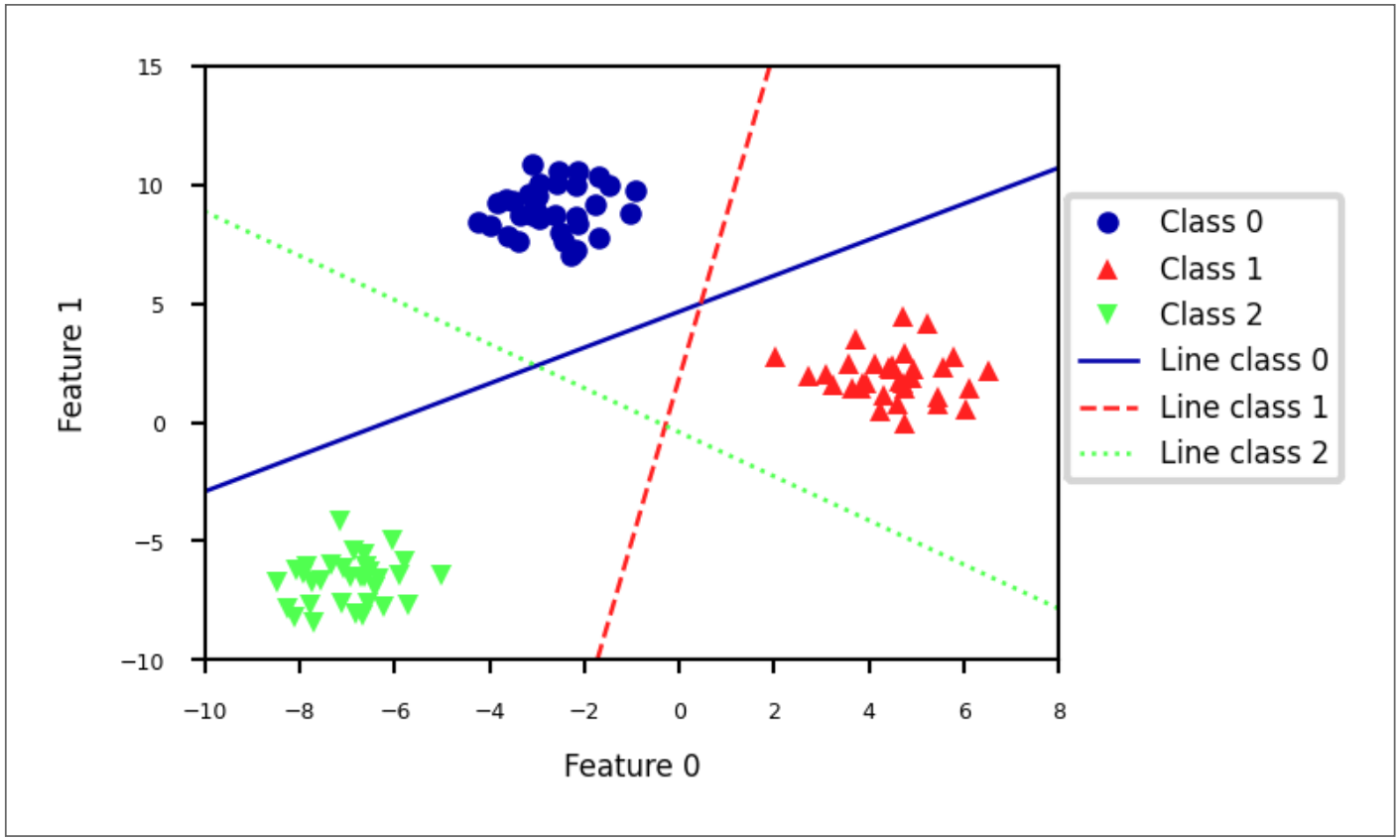
$$\frac{\partial \mathcal{L}_{Perceptron}}{\partial w_i} = \begin{cases} -y_i x_i & y_i(\mathbf{wx_i} + w_0) < 0 \\ 0 & \text{otherwise} \end{cases}$$



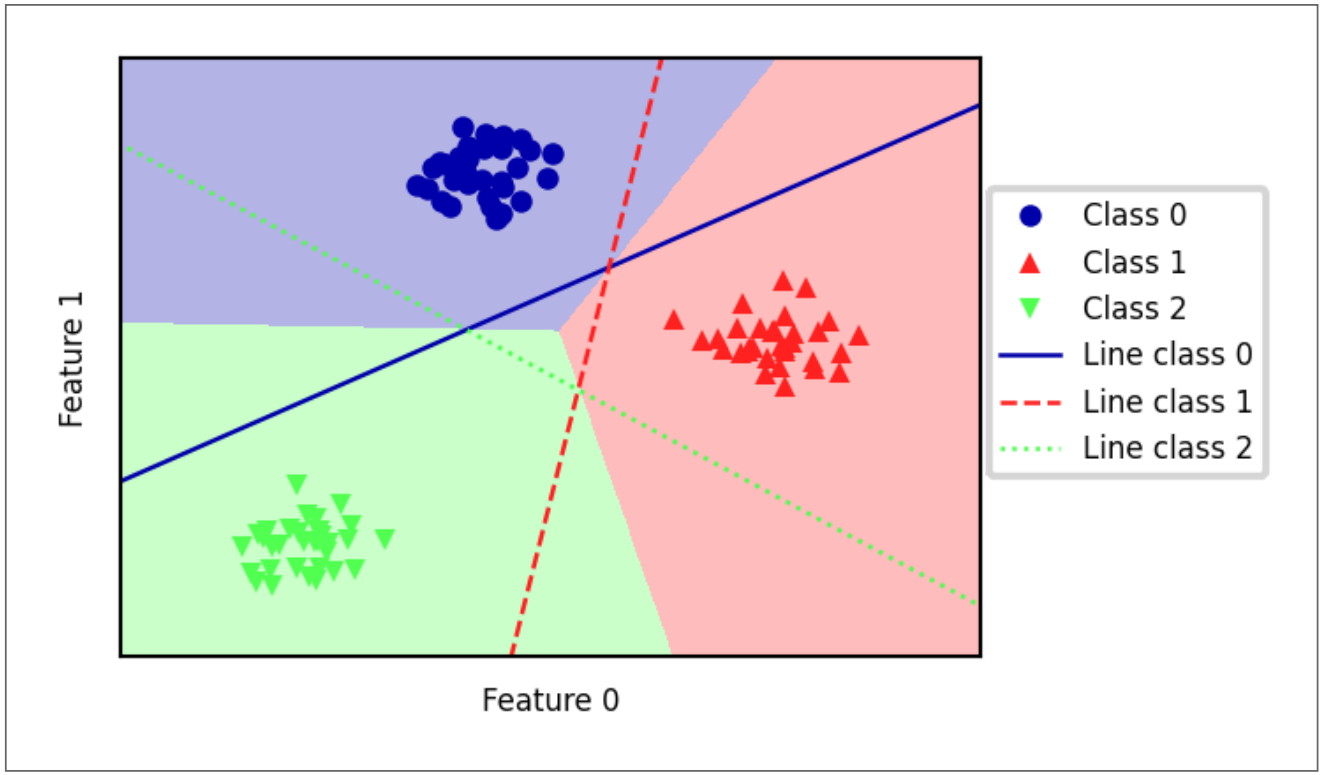
Linear Models for multiclass classification

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