Lecture 1: The architecture of learning algorithms

A few useful things to know about machine learning

Joaquin Vanschoren, Eindhoven University of Technology

Artificial Intelligence

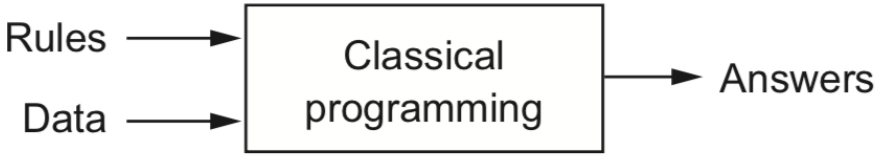
1950s: Can computers be made to 'think'?

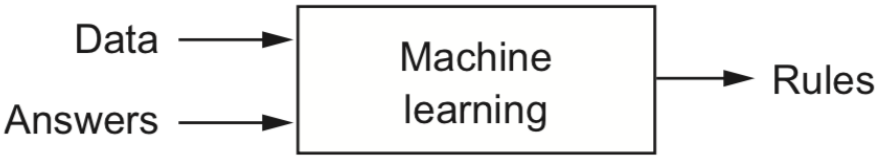
- automate intellectual tasks normally performed by humans
- encompasses learning, but also many other tasks (e.g. logic, planning,...)
- *symbolic AI*: programmed rules/algorithms for manipulating knowledge
    - Great for well-defined problems: chess, expert systems,...
    - Pervasively used today (e.g. chip design)
    - Hard for complex, fuzzy problems (e.g. images, text)

Machine Learning

Are computers capable of learning and originality? Alan Turing: Yes!

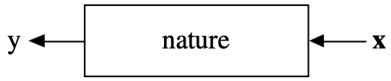
- Learn to perform a task T given experience (examples) E, always improving according to some metric M
- New programming paradigm
    - System is *trained* rather than explictly programmed
    - *Generalizes* from examples to find rules (models) to act/predict
- As more data becomes available, more ambitious problems can be tackled



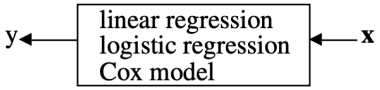


Machine learning vs Statistics

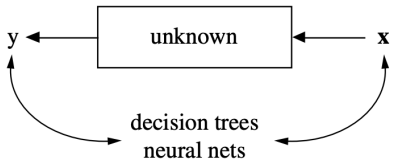
- Both aim to make predictions of natural phenomena:



- Statistics:
  - Help humans understand the world
  - Parametric: assume data is generated according to parametric model



- Machine learning:
  - Automate a task entirely (partially *replace* the human)
  - Assume that data generation process is unknown
  - Engineering-oriented, less (too little?) mathematical theory



See Breiman (2001): Statistical modelling: The two cultures

Machine Learning success stories

- Search engines (e.g. Google)
- Recommender systems (e.g. Netflix)
- Automatic translation (e.g. Google Translate)
- Speech understanding (e.g. Siri, Alexa)
- Game playing (e.g. AlphaGo)
- Self-driving cars
- Personalized medicine
- Progress in all sciences: Genetics, astronomy, chemistry, neurology, physics,..

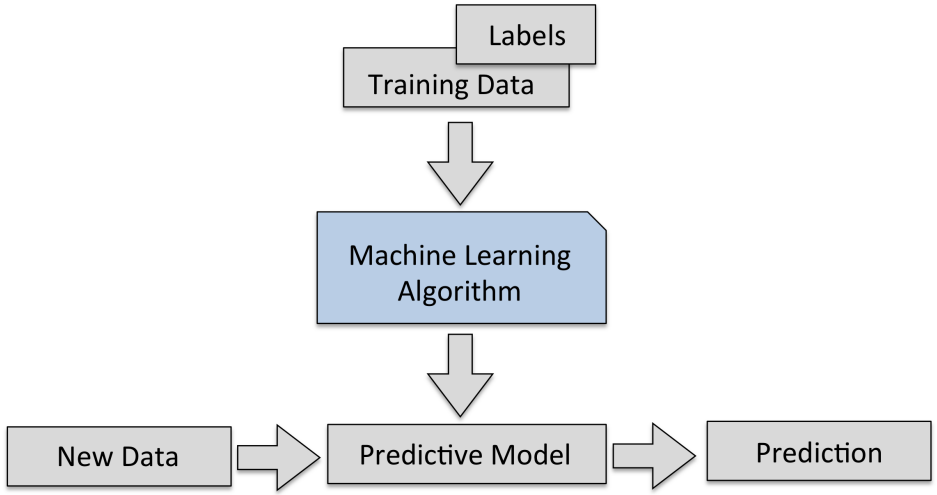
Types of machine learning

- **Supervised Learning**: learn a *model* from labeled *training data* (ground truth)
    - Given a new input *X*, predict the right output *y*
    - Given images of cats and dogs, predict whether a new image is a cat or a dog
- **Unsupervised Learning**: explore the structure of the data to extract meaningful information
    - Given inputs *X*, find which ones are special, similar, anomalous, ...
- **Semi-Supervised Learning**: learn a model from (few) labeled and (many) unlabeled examples
    - Unlabeled examples add information about which new examples are likely to occur
- **Reinforcement Learning**: develop an agent that improves its performance based on interactions with the environment

Note: Practical ML systems can combine many types in one system.

Supervised Machine Learning

- Learn a model from labeled training data, then make predictions
- Supervised: we know the correct/desired outcome (label)
- Subtypes: *classification* (predict a class) and *regression* (predict a numeric value)
- Most supervised algorithms that we will see can do both



Classification

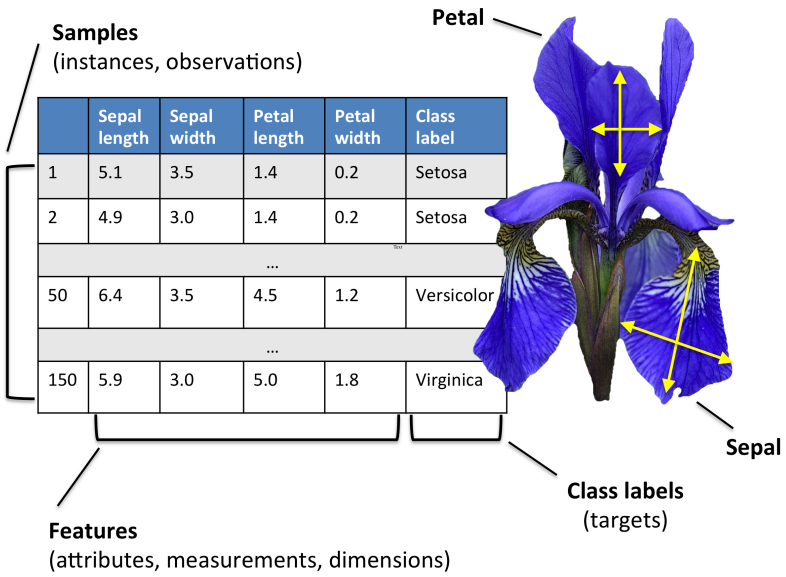
- Predict a *class label* (category), discrete and unordered
    - Can be *binary* (e.g. spam/not spam) or *multi-class* (e.g. letter recognition)
    - Many classifiers can return a *confidence* per class
- The predictions of the model yield a *decision boundary* separating the classes

Example: Flower classification

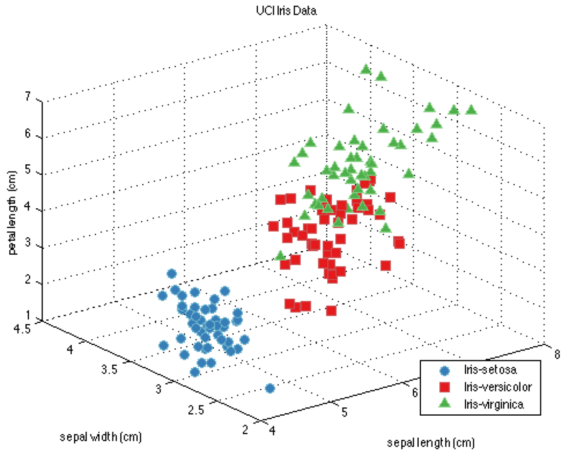
Classify types of Iris flowers (setosa, versicolor, or virginica). How would you do it?

Representation: input features and labels

- We could take pictures and use them (pixel values) as inputs (-> Deep Learning)
- We can manually define a number of input features (variables), e.g. length and width of leaves
- Every `example' is a point in a (possibly high-dimensional) space



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica



Regression

- Predict a continuous value, e.g. temperature
    - Target variable is numeric
    - Some algorithms can return a *confidence interval*
- Find the relationship between predictors and the target.
    - E.g. relationship between hours studied and final grade

Unsupervised Machine Learning

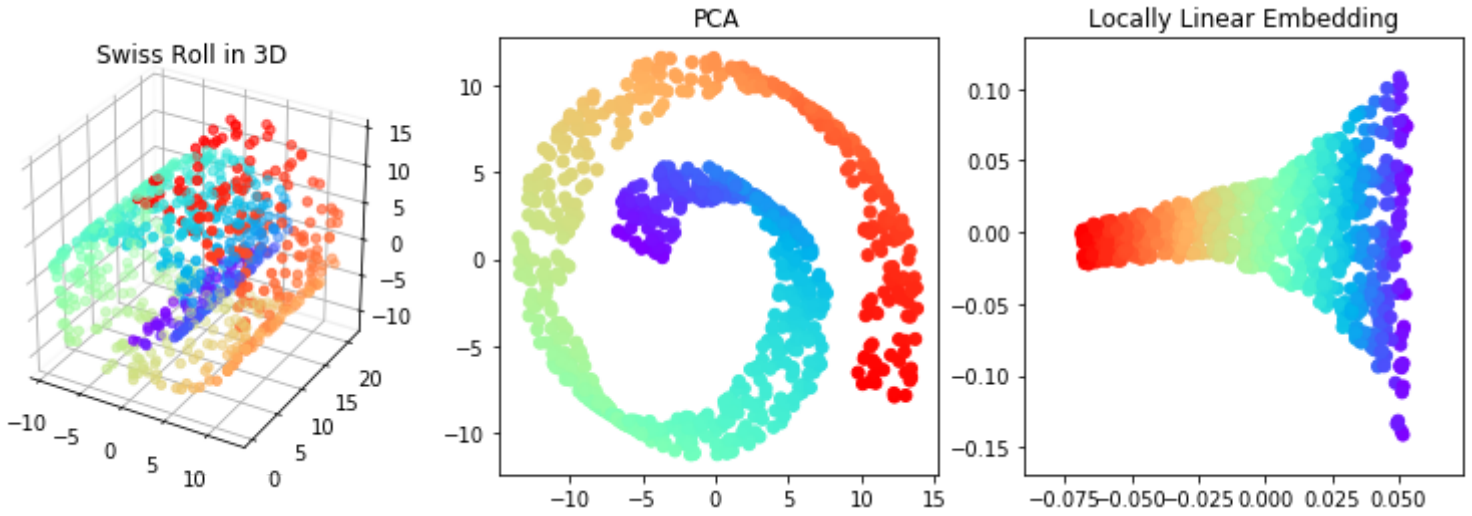
- Unlabeled data, or data with unknown structure
- Explore the structure of the data to extract information
- Many types, we'll just discuss two.

Clustering

- Organize information into meaningful subgroups (clusters)
- Objects in cluster share certain degree of similarity (and dissimilarity to other clusters)
- Example: distinguish different types of customers

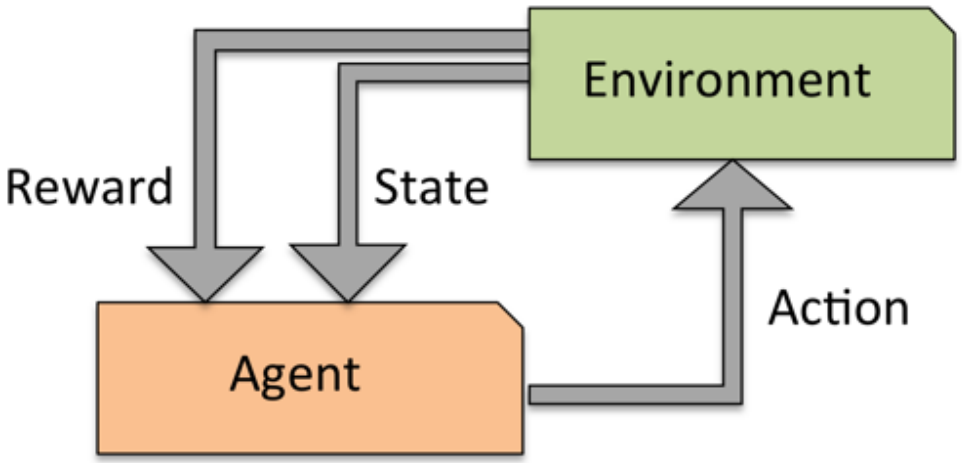
Dimensionality reduction

- Data can be very high-dimensional and difficult to understand, learn from, store,...
- Dimensionality reduction can compress the data into fewer dimensions, while retaining most of the information
- Contrary to feature selection, the new features lose their (original) meaning
- The new representation can be a lot easier to model (and visualize)



Reinforcement learning

- Develop an agent that improves its performance based on interactions with the environment
    - Example: games like Chess, Go,...
- Search a (large) space of actions and states
- *Reward function* defines how well a (series of) actions works
- Learn a series of actions (policy) that maximizes reward through exploration



Learning = Representation + evaluation + optimization

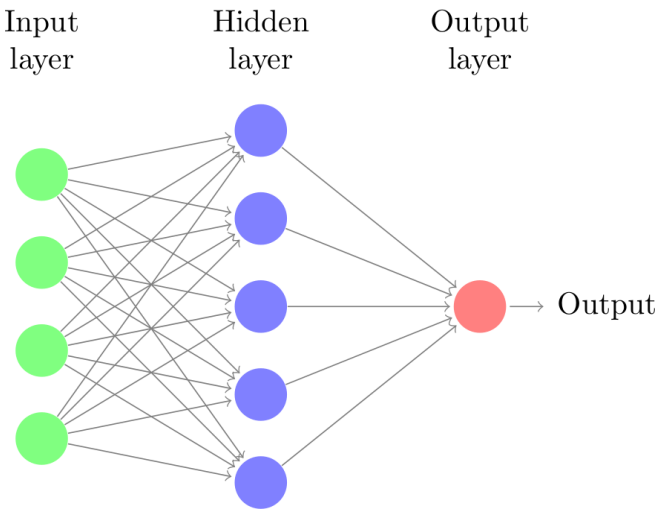
All machine learning algorithms consist of 3 components:

- **Representation**: A model must be represented in a formal language that the computer can handle
    - Defines the 'concepts' it can learn, the *hypothesis space*
    - E.g. a decision tree, neural network, set of annotated data points
- **Evaluation**: An *internal* way to choose one hypothesis over the other
    - Objective function, scoring function, loss function
    - E.g. Difference between correct output and predictions
- **Optimization**: An *efficient* way to search the hypothesis space
    - Start from simple hypothesis, extend (relax) if it doesn't fit the data
    - Defines speed of learning, number of optima,...
    - E.g. Gradient descent

A powerful/flexible model is only useful if it can also be optimized efficiently

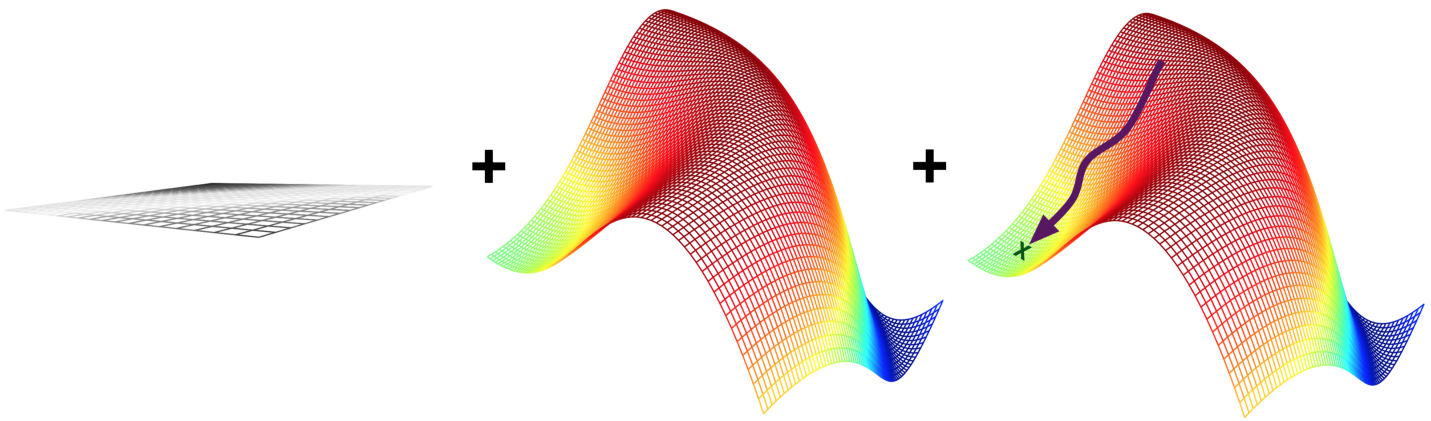
Example: neural networks

- Representation: (layered) neural network
  - Each connection has a *weight* (a.k.a. model parameters)
  - Each node receives the weighted input values and emits a new value
- The *hypothesis space* consists of the set of all weights
- The architecture, type of neurons, etc. are fixed
  - We call these *hyper-parameters* (set by user, fixed during training)
  - They can also be learned (in an outer loop)



Example: neural networks

- Representation: For illustration, consider the space of 2 model parameters
- Evaluation: A *loss function* computes, for each set of parameters, how good the predictions are
    - *Estimated* on a set of training data with the 'correct' predictions
    - We can't see the full surface, only evaluate specific sets of parameters
- Optimization: Find the optimal set of parameters
    - Usually a type of *search* in the hypothesis space
    - Given a few initial evaluations, predict which parameters may be better



Generalization, Overfitting and Underfitting

- We *hope* that the model can *generalize* from the training data: make accurate predictions on unseen data.
- We can never be sure, only hope that we make the right assumptions.
    - We typically assume that new data will be similar to previous data
    - *Inductive bias*: assumptions that we put into the algorithm (everything except the training data itself)

Example: Dating

Nr	Day of Week	Type of Date	Weather	TV Tonight	Date?
1	Weekday	Dinner	Warm	Bad	No
2	Weekend	Club	Warm	Bad	Yes
3	Weekend	Club	Warm	Bad	Yes
4	Weekend	Club	Cold	Good	No
Now	Weekend	Club	Cold	Bad	?

- Can you find a simple rule that works? Is one better than others?
- What can we assume about the future? Nothing?
- What if there is noise / errors?
- What if there are factor you don't know about?

Overfitting and Underfitting

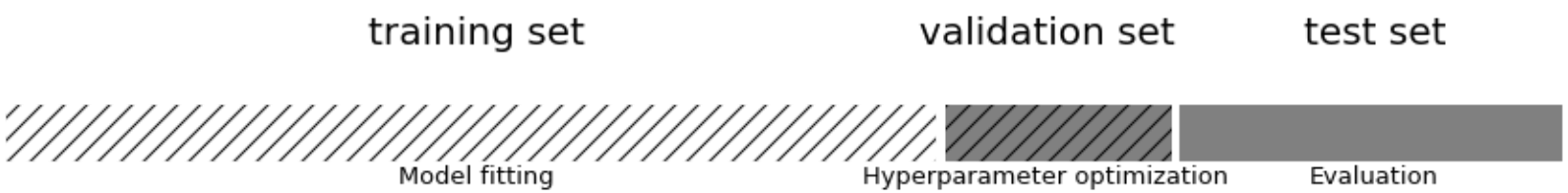
- It's easy to build a complex model that is 100% accurate on the training data, but very bad on new data
- Overfitting: building a model that is *too complex for the amount of data* that we have
    - You model peculiarities in your training data (noise, biases,...)
    - Solve by making model simpler (regularization), or getting more data
    - **Most algorithms have hyperparameters that allow regularization**
- Underfitting: building a model that is *too simple given the complexity of the data*
    - Use a more complex model
- There are techniques for detecting overfitting (e.g. bias-variance analysis). More about that later
- You can build *ensembles* of many models to overcome both underfitting and overfitting

- There is often a sweet spot that you need to find by optimizing the choice of algorithms and hyperparameters, or using more data.
- Example: regression using polynomial functions

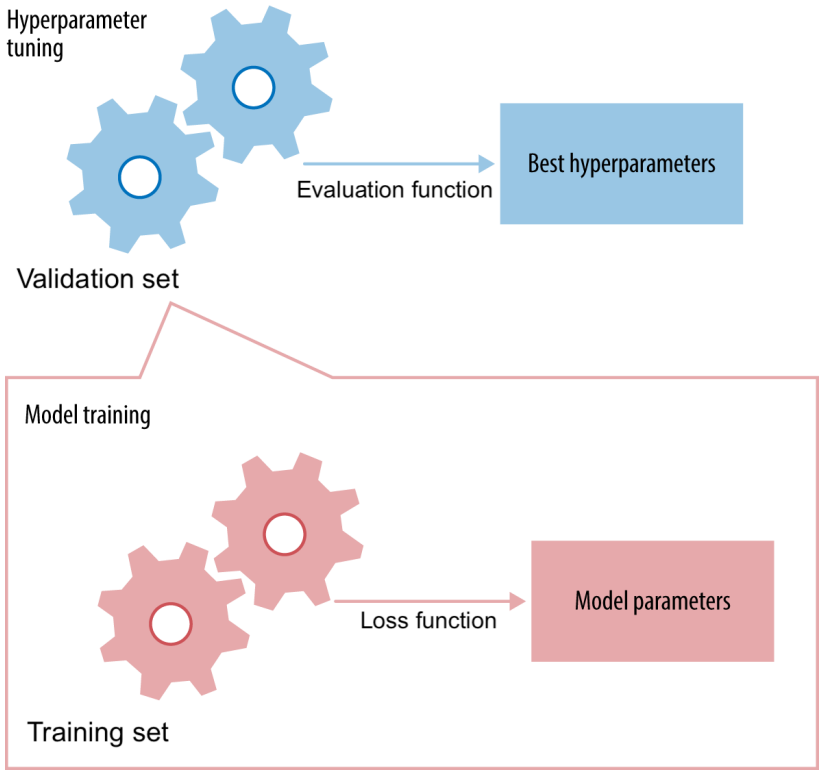
Model selection

- Next to the (internal) loss function, we need an (external) evaluation function
    - Feedback signal: are we actually learning the right thing?
    - Are we under/overfitting?
    - More freely chosen to fit the application. Loss functions have constraints (e.g. differentiable)
    - Needed to choose between algorithms (or different hyper-parameter settings)

- Data needs to be split into *training* and *test* sets
    - Optimize model parameters on the training set, evaluate on independent test set
    - To optimize hyperparameters as well, set aside part of training set as a *validation* set



Overview

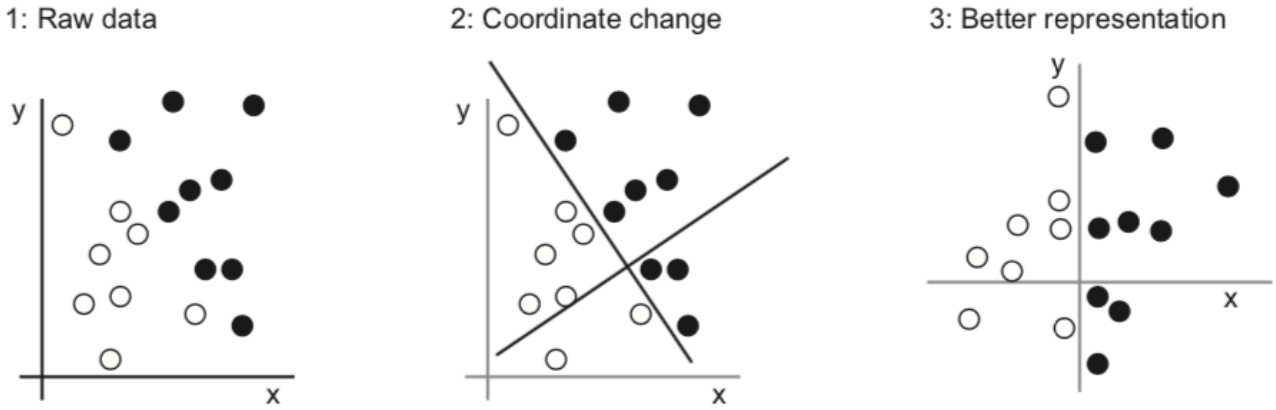

Hyperparameter tuning
Evaluation function
Best hyperparameters
Validation set
Model training
Loss function
Model parameters
Training set

Only generalization counts!

- Never evaluate your final models on the training data, except for:
    - Tracking whether the optimizer converges (learning curves)
    - Detecting under/overfitting:
        - Low training and test score: underfitting
        - High training score, low test score: overfitting
- Always keep a completely independent test set
- Avoid data leakage:
    - Never optimize hyperparameter settings on the test data
    - Never choose preprocessing techniques based on the test data
- On small datasets, use multiple train-test splits to avoid bias
    - E.g. Use cross-validation (see later)

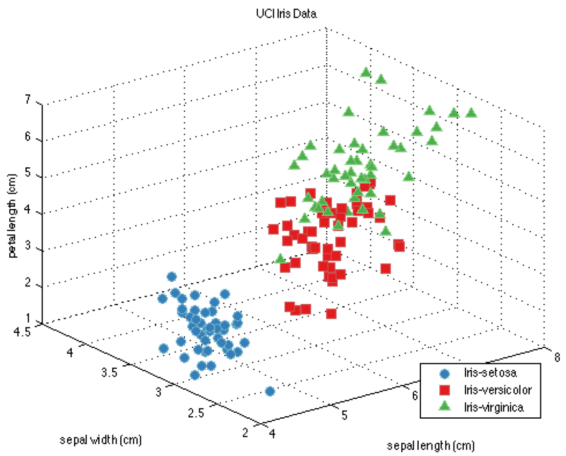
Data (problem) representation

- Algorithm needs to correctly transform the inputs to the right outputs
- A lot depends on how we present the data to the algorithm
    - Transform the data to a more useful representation (a.k.a. *encoding* or *embedding*)
    - Can be done end-to-end (e.g. deep learning) or by first 'preprocessing' the data



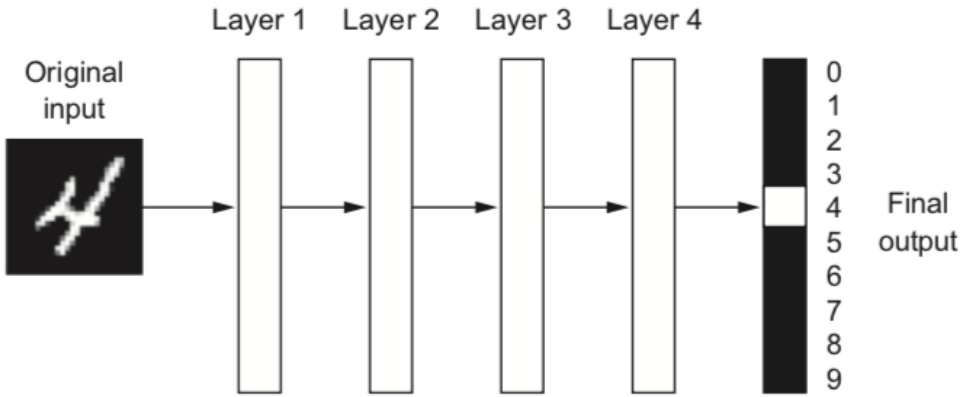
Feature engineering

- Most machine learning techniques require humans to build a good representation of the data
    - Sometimes data is naturally structured (e.g. medical tests)
- Nothing beats domain knowledge (when available) to get a good representation
    - E.g. Iris data: leaf length/width separate the classes well
- Feature engineering is often necessary to get the best results
    - Feature selection, dimensionality reduction, scaling, ...
    - *Applied machine learning is basically feature engineering (Andrew Ng)*



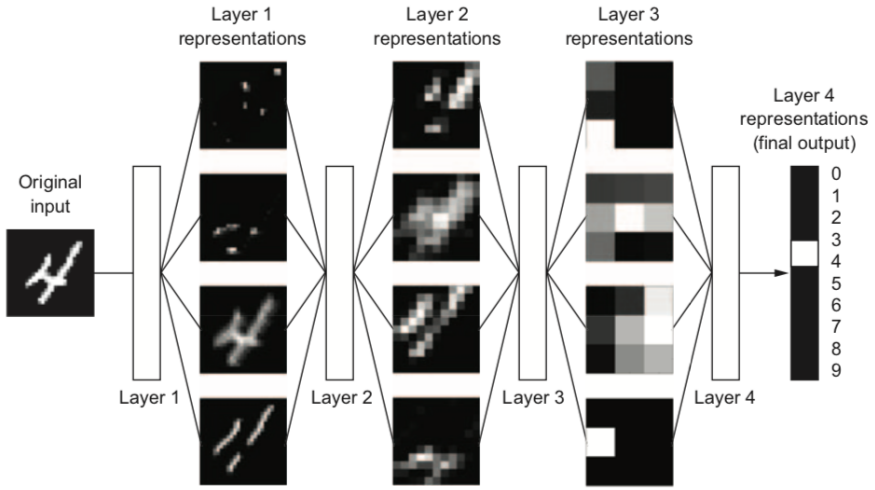
Learning data transformations end-to-end

- For unstructured data (e.g. images, text), it's hard to extract good features
- Deep learning: learn your own representation (embedding) of the data
    - Through multiple layers of representation (e.g. layers of neurons)
    - Each layer transforms the data a bit, based on what reduces the error



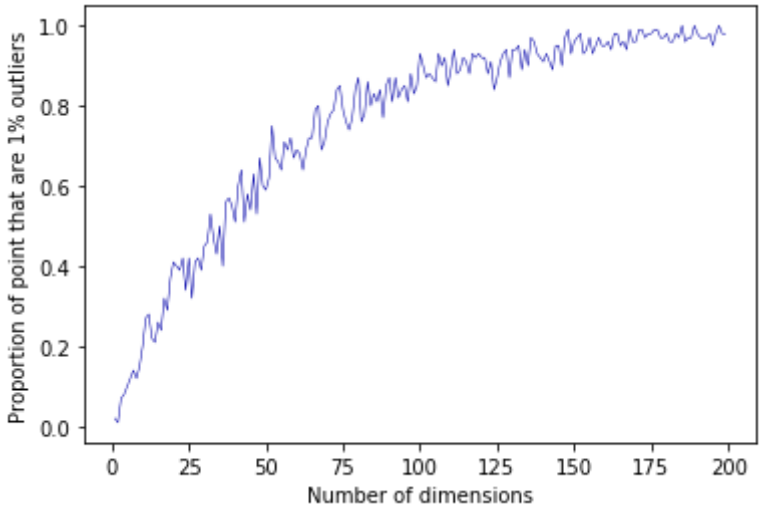
Example: digit classification

- Input pixels go in, each layer transforms them to an increasingly informative representation for the given task
- Often less intuitive for humans



Curse of dimensionality

- Intuition fails in high dimensions:
  - Randomly sample points in an n-dimensional space (e.g. a unit hypercube)
  - The more dimensions you have, the more sparse the space becomes
  - Distances between any two points will become almost identical
  - Almost all points become outliers at the edge of the space



Practical consequences

- For every dimension (feature) you add, you need exponentially more data to avoid sparseness
- Affects any algorithm that is based on distances (e.g. kNN, SVM, kernel-based methods, tree-based methods,...)
- Blessing of non-uniformity: on many applications, the data lives in a very small subspace
- You can drastically improve performance by selecting features or using lower-dimensional data representations

"More data can beat a cleverer algorithm" (but you need both)

- More data reduces the chance of overfitting
- Less sparse data reduces the curse of dimensionality
- *Non-parametric* models: number of model parameters grows with the amount of data
    - Tree-based techniques, k-Nearest neighbors, SVM,...
    - They can learn any model given sufficient data (but can get stuck in local minima)
- *Parametric* (fixed size) models: fixed number of model parameters
    - Linear models, Neural networks,...
    - Can be given a huge number of parameters to benefit from more data
    - Deep learning models can have millions of weights, learn almost any function.
- The bottleneck is moving from data to compute/scalability

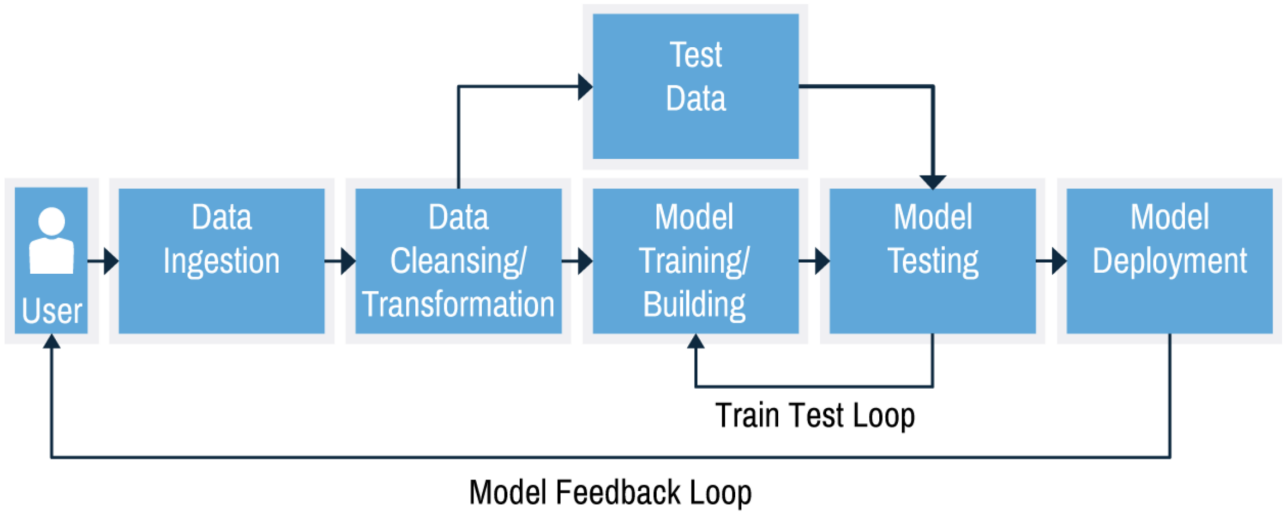
Building machine learning systems

A typical machine learning system has multiple components:

- Preprocessing: Raw data is rarely ideal for learning (Lecture 4)
    - Feature scaling: bring values in same range
    - Encoding: make categorical features numeric
    - Discretization: make numeric features categorical
    - Label imbalance correction (e.g. downsampling)
    - Feature selection: remove uninteresting/correlated features
    - Dimensionality reduction can also make data easier to learn
    - Using pre-learned embeddings (e.g. word-to-vector, image-to-vector)

- Learning and evaluation (Lecture 3)
    - Every algorithm has its own biases
    - No single algorithm is always best
    - *Model selection* compares and selects the best models
        - Different algorithms, different hyperparameter settings
    - Split data in training, validation, and test sets
- Prediction
    - Final optimized model can be used for prediction
    - Expected performance is performance measured on *independent* test set

- Together they form a *workflow* of *pipeline*
- There exist machine learning methods to automatically build and tune these pipelines (Lecture 7)
- You need to optimize pipelines continuously
    - *Concept drift*: the phenomenon you are modelling can change over time
    - *Feedback*: your model's predictions may change future data



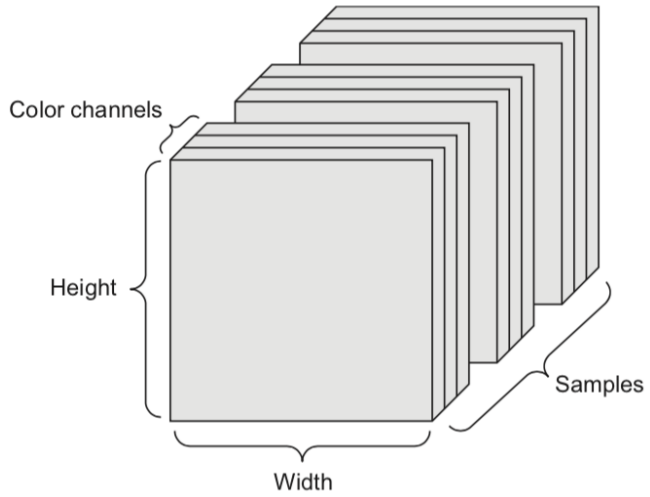
The Mathematics behind Machine Learning

- We don't want you to run machine learning algorithms blindly, you need to understand what they do.
- To understand machine learning algorithms, it often helps to describe them mathematically.
- To avoid confusion, let's specify a precise notation

Basic notation

- A *scalar* is a simple numeric value, denoted by italic letter: $x = 3.24$
- A *vector* is a 1D ordered array of n scalars, denoted by bold letter: $\mathbf{x} = [3.24, 1.2]$
    - A vector can represent a *point* in an n-dimensional space, given a *basis*.
    - 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: $\mathbf{X} = \begin{bmatrix} 3.24 & 1.2 \\ 2.24 & 0.2 \end{bmatrix}$
    - It can represent a set of points in an n-dimensional space, given a *basis*.
    - $\mathbf{X}_i$ denotes the ith *row* of the matrix
    - $\mathbf{X}_{i,j}$ denotes the *element* in the ith row, jth column, thus $\mathbf{X}_{0,1} = 2.24$
- The *standard basis* for a Euclidean space is the set of unit vectors
    - Data can also be represented in a non-standard basis (e.g. polynomials) if useful

- 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 weight 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:
$$\sum_{i=0}^{n} = x_0 + x_1 + \ldots + x_p \qquad \prod_{i=0}^{n} = x_0 \cdot x_1 \cdot \ldots \cdot x_p$$
- 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
$\mathbf{wx} = \mathbf{w} \cdot \mathbf{x} = \sum_{i=0}^{p} w_i \cdot x_i = w_0 \cdot x_0 + w_1 \cdot x_1 + \ldots + w_p \cdot x_p$
- 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
- $\operatorname{argmax}_{c \in C} f(x)$ returns the element c that maximizes f(c)

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. $f = 2x_0 + 3x_1^2 - \sin(x_2) \rightarrow \nabla f = [2, 6x_1, -cos(x_2)]$

Linear models

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

- Can be very powerful for or datasets with many features.
- If you have more features than training data points, any target y can be perfectly modeled (on the training set) as a linear function.
- Even non-linear data (or non-linearly seperable data) can be modelled with linear models with a bit of preprocessing.
    - Basis for 'Generalized Linear Models' (e.g. kernelized SVMs, see lecture 2)

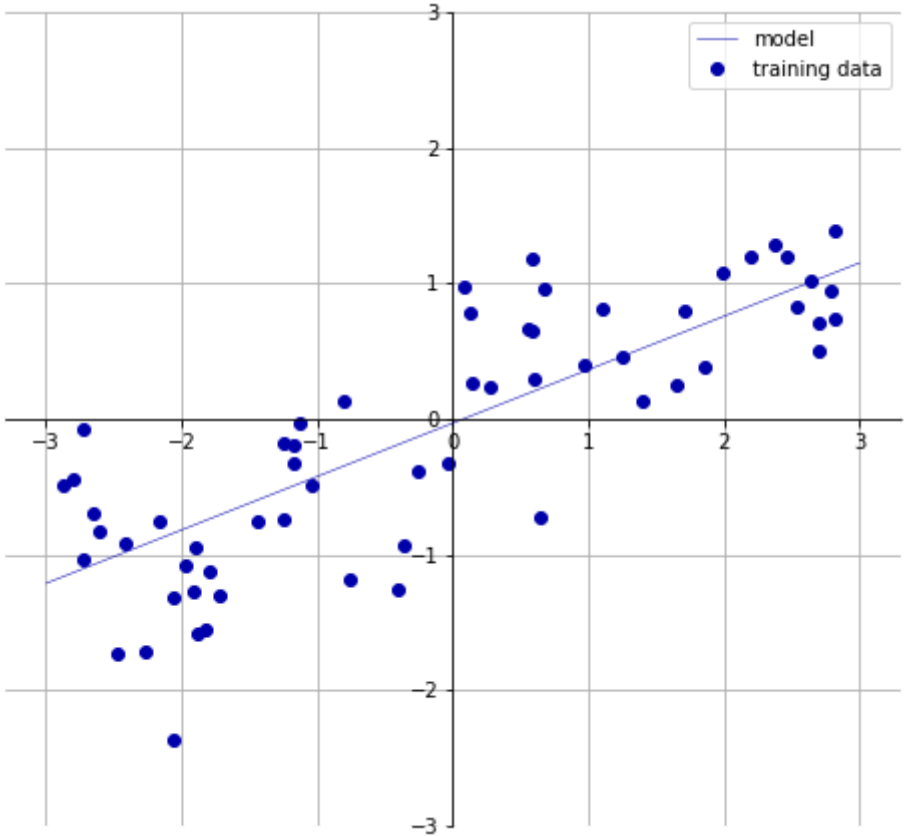
Linear models for regression

Prediction formula for input features x. w_i and b are the *model parameters* that need to be learned.

$$\hat{y} = \mathbf{w}\mathbf{x} + b = \sum_{i=0}^{p} w_i \cdot x_i + b = w_0 \cdot x_0 + w_1 \cdot x_1 + \ldots + w_p \cdot x_p + b$$

There are many different algorithms, differing in how w and b are learned from the training data.

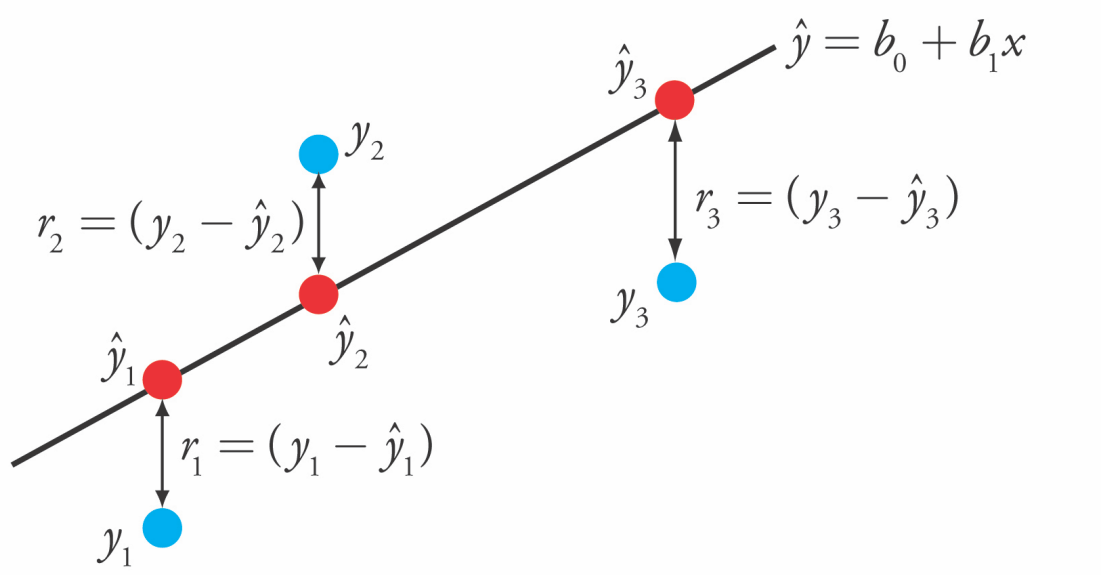
```
w[0]: 0.393906  b: -0.031804
```



Linear Regression aka Ordinary Least Squares

- Finds the parameters w and b that minimize the *mean squared error* between predictions (red) and the true regression targets (blue), y, on the training set.
    - MSE: Sum of the squared differences (residuals) between the predictions $\hat{y}_i$ and the true values y_i.

$$\mathcal{L}_{MSE} = \sum_{n=0}^{N}(y_n - \hat{y}_n)^2 = \sum_{n=0}^{N}(y_n - (\mathbf{wx_n} + b))^2$$



Solving ordinary least squares

- Convex optimization problem with unique closed-form solution (if you have more data points than model parameters w)
- It has no hyperparameters, thus model complexity cannot be controlled.
- It **very easily overfits**. What does that look like?
    - model parameters w become very large (steep incline/decline)
    - a small change in the input x results in a very different output y

Linear regression can be found in `sklearn.linear_model`. We'll evaluate it on the Boston Housing dataset.

```
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): 30.934563673645666

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

Ridge regression

- Same formula as linear regression
- Adds a penalty term to the least squares sum:

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

- Requires that the coefficients (w) are close to zero.
    - Each feature should have as little effect on the outcome as possible
- Regularization: explicitly restrict a model to avoid overfitting.
- Type of L2 regularization: prefers many small weights
    - L1 regularization prefers sparsity: many weights to be 0, others large

`Ridge` can also be found in `sklearn.linear_model`.

```
ridge = Ridge().fit(X_train, y_train)
```

```
Training set score: 0.89
Test set score: 0.75
```

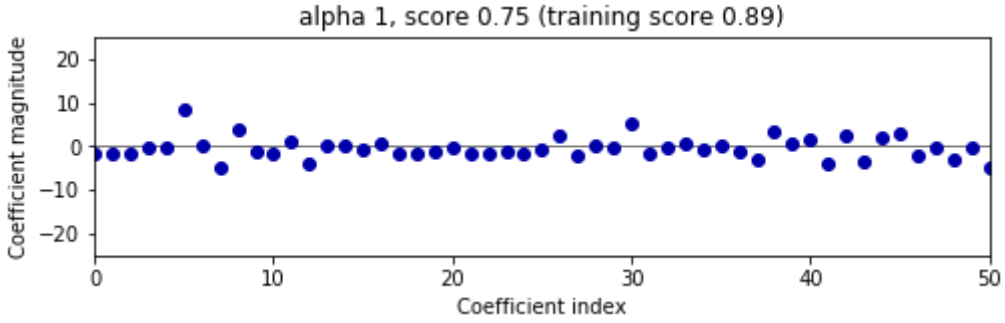
Test set score is higher and training set score lower: less overfitting!

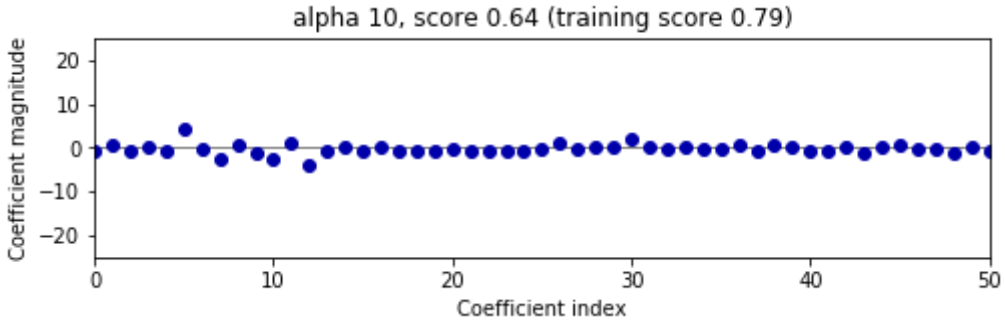
The strength of the regularization can be controlled with the `alpha` parameter. Default is 1.0.

- Increasing alpha forces coefficients to move more toward zero (more regularization)
- Decreasing alpha allows the coefficients to be less restricted (less regularization)

We can plot the weight values for differents levels of regularization. Move the slider to increase/decrease regularization. Increasing regularization decreases the values of the coefficients, but never to 0.

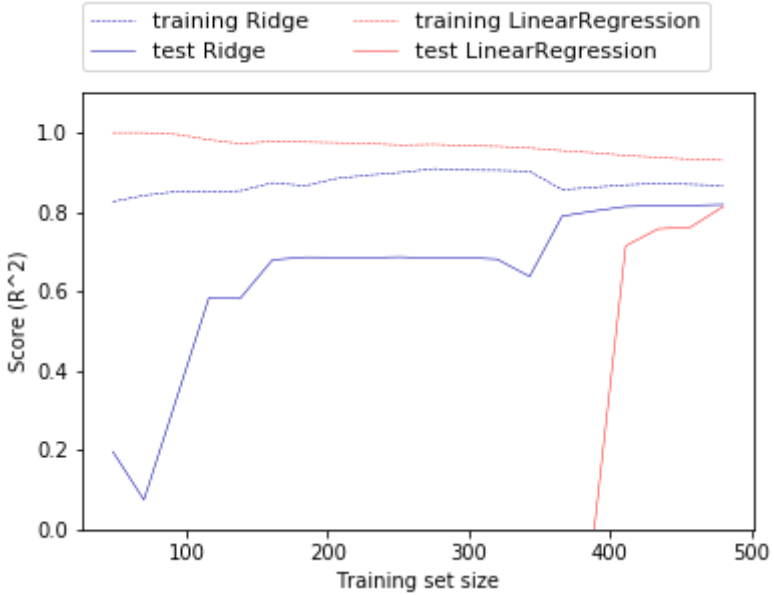

alpha 0.1, score 0.77 (training score 0.93)
Coefficient magnitude
Coefficient index
20
10
0
−10
−20
0
10
20
30
40
50


alpha 1, score 0.75 (training score 0.89)
Coefficient magnitude
Coefficient index
20
10
0
−10
−20
0
10
20
30
40
50


alpha 10, score 0.64 (training score 0.79)
Coefficient magnitude
Coefficient index
20
10
0
−10
−20
0
10
20
30
40
50

Other ways to reduce overfitting:

- Add more training data: with enough training data, regularization becomes less important
    - Ridge and linear regression will have the same performance
- Use less features, remove unimportant ones or find a lower-dimensional embedding (e.g. PCA)
    - Less degrees of freedom
- Scaling the data may also help



Lasso (Least Absolute Shrinkage and Selection Operator)

- Another form of regularization
- Adds a penalty term to the least squares sum:

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

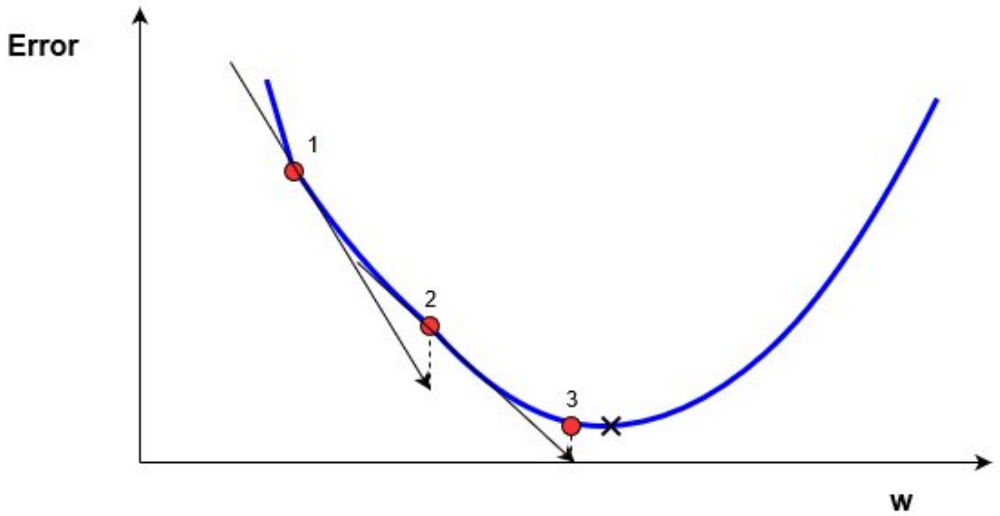
- Prefers coefficients to be exactly zero (L1 regularization).
- Some features are entirely ignored by the model: automatic feature selection.
- Same parameter `alpha` to control the strength of regularization.
- Convex, but no longer strictly convex (and NOT differentiable). Weights can be optimized using (for instance) *coordinate descent*
- New parameter `max_iter`: the maximum number of coordinate descent iterations
    - Should be higher for small values of `alpha`

Gradient Descent

- Start with a random set of p weights values $\mathbf{w}^0$
- Compute the derivative of the objective function $\mathcal{L}$ (e.g. $\mathcal{L}_{Ridge}$) and use it to find the slope (in p dimensions)
- Update all weights slightly (step size γ) in the direction of the downhill slope. For step s:

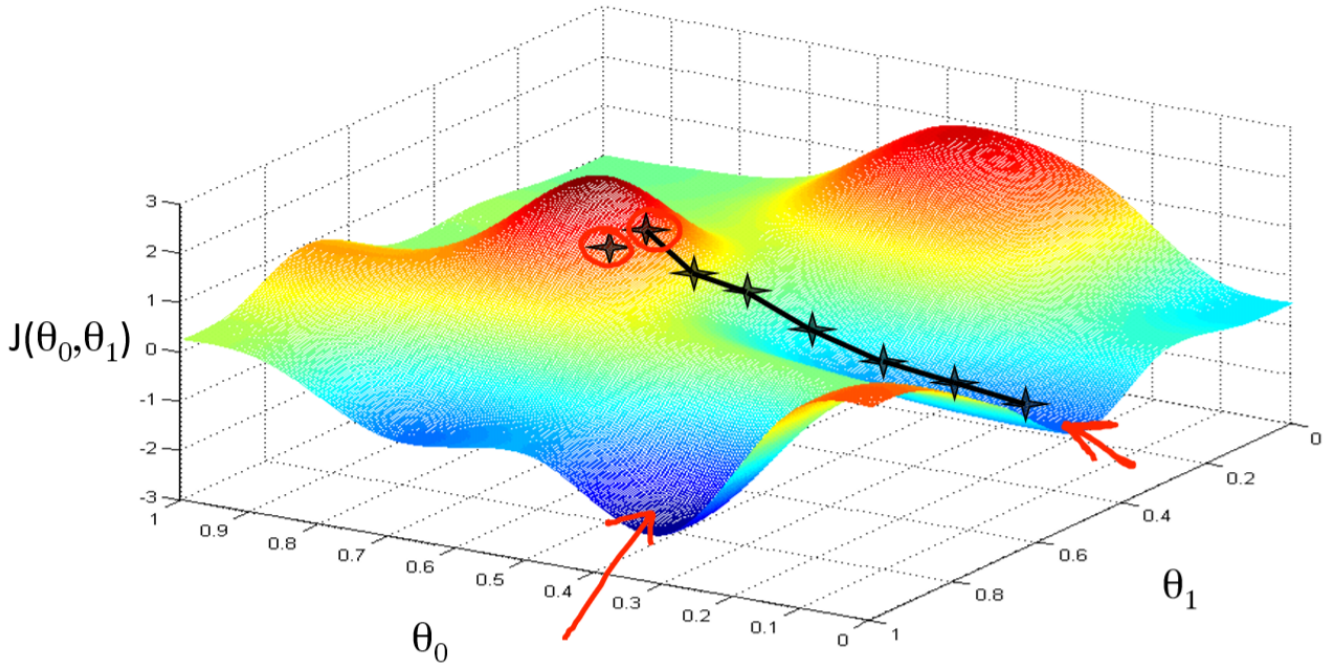
$$\mathbf{w}^{s+1} = \mathbf{w}^s - \gamma \nabla \mathcal{L}(\mathbf{w}^s)$$

- Repeat for `max_iter` iterations
- Visualization in 1 dimension (for 1 weight):



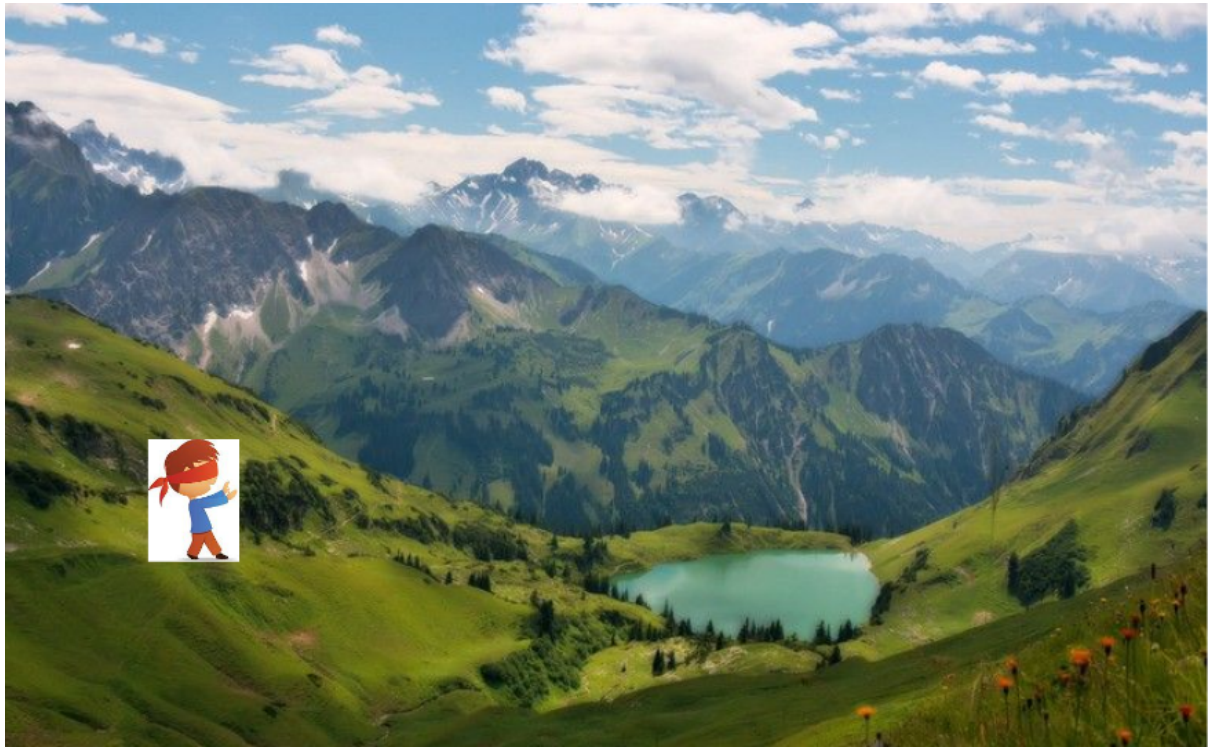
Gradient Descent

In two dimensions:



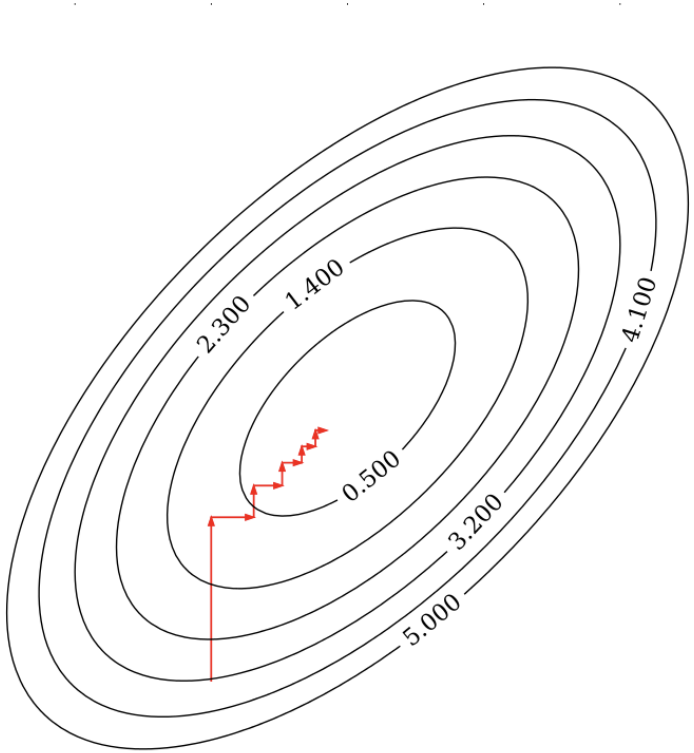
Gradient Descent

- Intuition: walking downhill using only the slope you "feel" nearby
- In many dimensions (e.g. many model parameters) you nearly always find a good local minimum

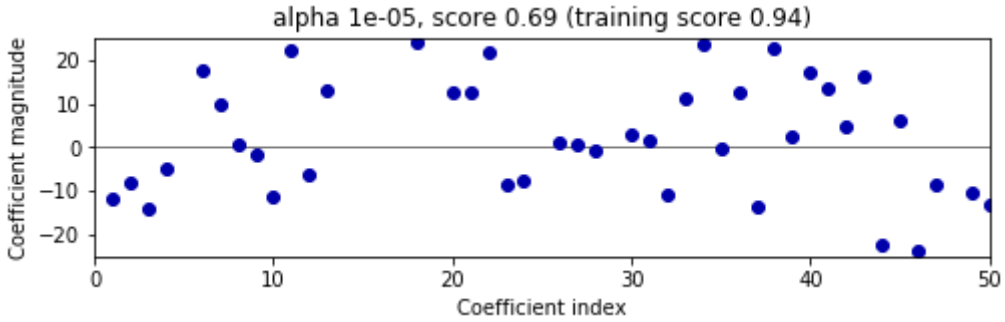


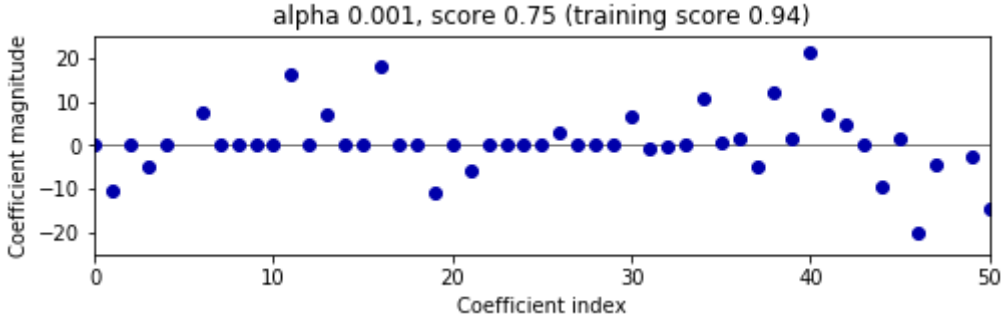
Coordinate descent

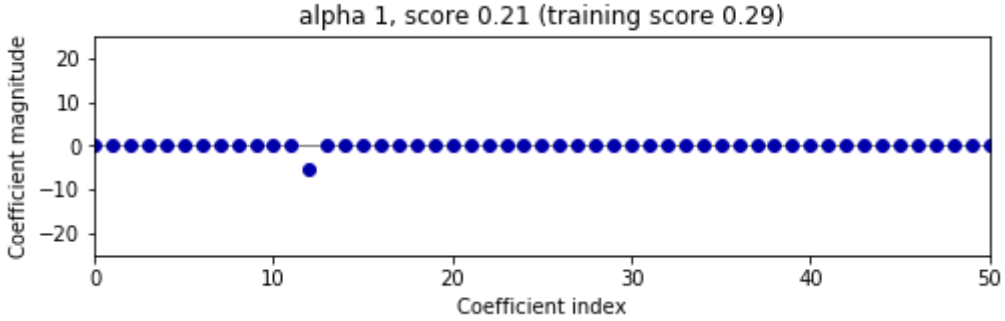
- Variation of gradient descent, also applicable for non-differentiable loss functions
- Faster iterations, may converge more slowly
- In every iteration, optimizes a single coordinate w_i, using a coordinate selection rule (e.g. round robin)



- If $\mathcal{L}$ is differentiable (e.g. L2), the update rule is based on the partial derivative: $w_i^{s+1} = w_i^s - \gamma \frac{\partial \mathcal{L}}{\partial w_i}$
- If $\mathcal{L}$ is not differentiable but convex (e.g. L1), the *subgradient* (https://www.cs.cmu.edu/~ggordon/10725-F12/slides/06-sg-method.pdf) can be computed.
- For L1, the resulting update rule (https://xavierbourretsicotte.github.io/lasso_derivation.html) includes the *soft thresholding operator* S: $w_i^{s+1} \cong S(f(w_i^s), \alpha)$
    - S sets w_i's to 0 when they are sufficiently small ('sufficiently' is defined by α)

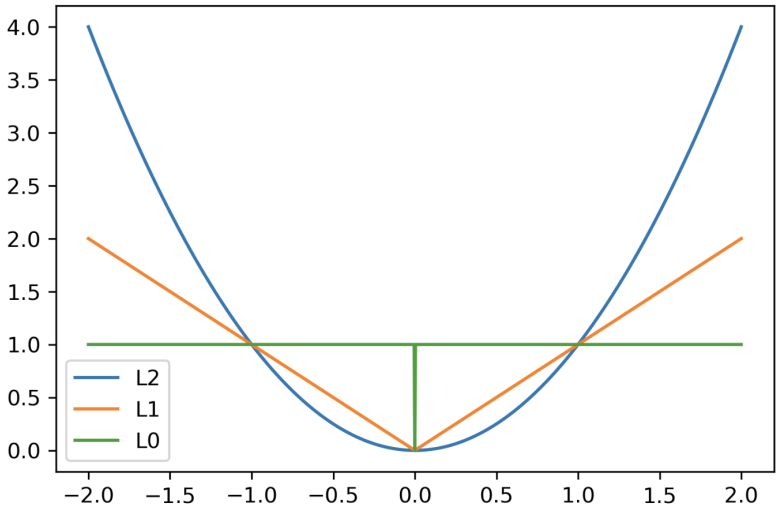

alpha 1e-05, score 0.69 (training score 0.94)
Coefficient magnitude
Coefficient index
20
10
0
−10
−20
0
10
20
30
40
50


alpha 0.001, score 0.75 (training score 0.94)
Coefficient magnitude
Coefficient index
20
10
0
−10
−20
0
10
20
30
40
50


alpha 1, score 0.21 (training score 0.29)
Coefficient magnitude
Coefficient index
20
10
0
−10
−20
0
10
20
30
40
50

Interpreting L1 and L2 loss

- L1 and L2 in function of the weights



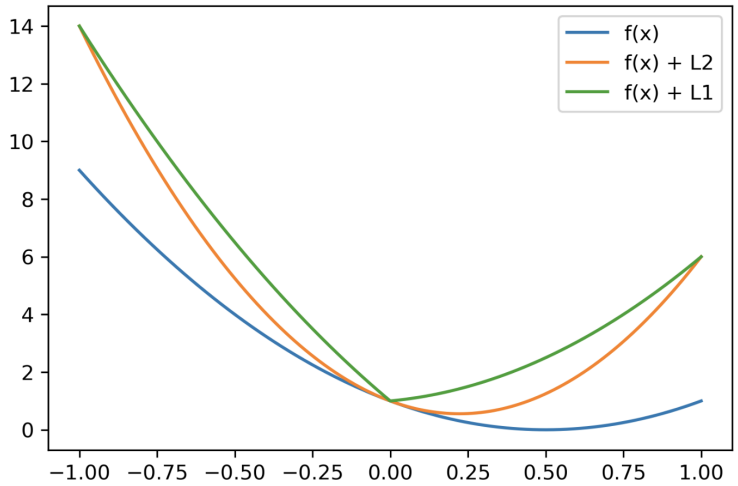
$$\ell_2(w) = \sqrt{\sum_i w_i^2}$$

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

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

Least Squares Loss + L1 or L2

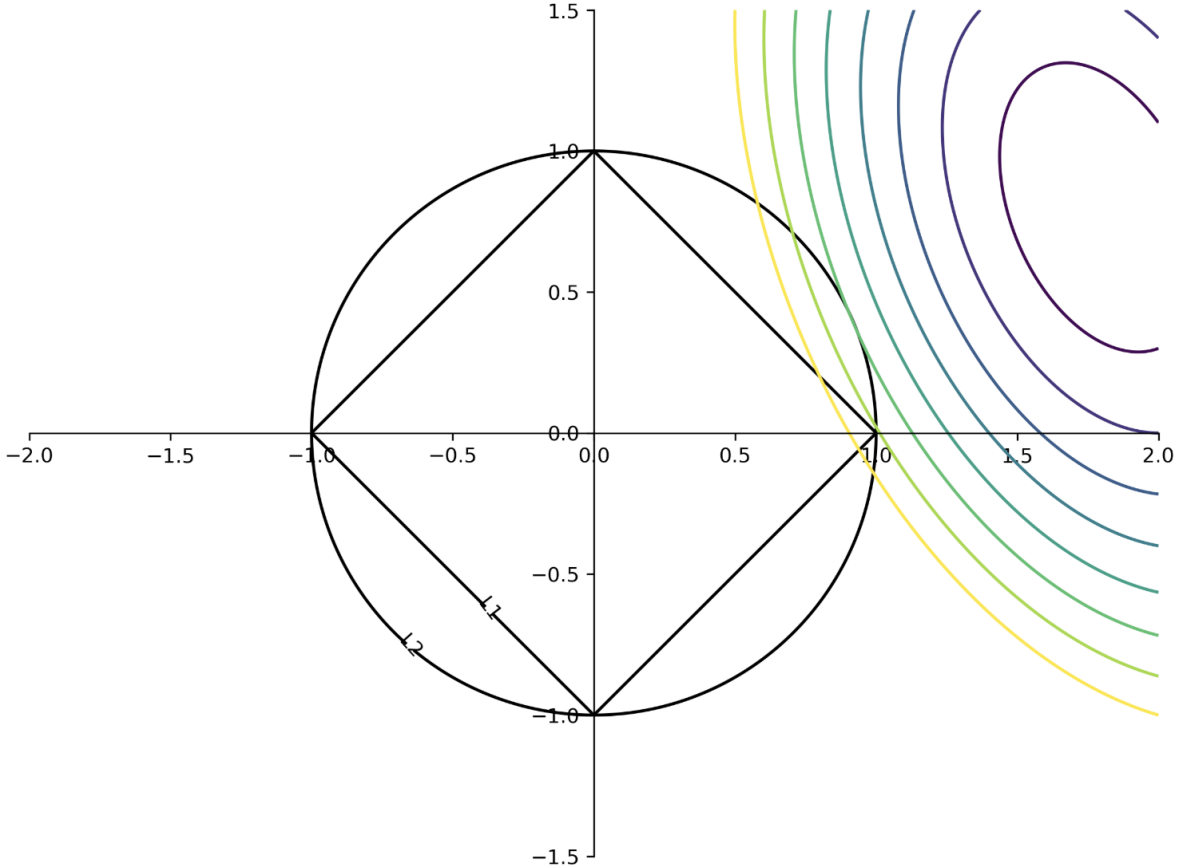
- Imagine f(x) is the least squares loss



$$f(x) = (2x-1)^2$$
$$f(x) + L2 = (2x-1)^2 + \alpha x^2$$
$$f(x) + L1 = (2x-1)^2 + \alpha|x|$$

In 2D (for 2 model weights w_1 and w_2)

- L1 loss ($\Sigma|w|$) for every $\{w_1, w_2\}$ falls on the 'diamond'
- L2 (Σw^2) for every $\{w_1, w_2\}$ falls on the unit circle
- The least squared loss is a convex function in this space
- For L1, the loss is minimized (best model) if w_1 or w_2 is 0 (not so for L2)


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
L1
L2

Linear models for Classification

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

$$\hat{y} = w_0 * x_0 + w_1 * x_1 + \ldots + w_p * x_p + b > 0$$

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

There are many algorithms for learning linear classification models, differing in:

- Loss function: evaluate how well the linear model fits the training data
- Regularization techniques

Most common techniques:

- Logistic regression:
    - `sklearn.linear_model.LogisticRegression`
- Linear Support Vector Machine:
    - `sklearn.svm.LinearSVC`

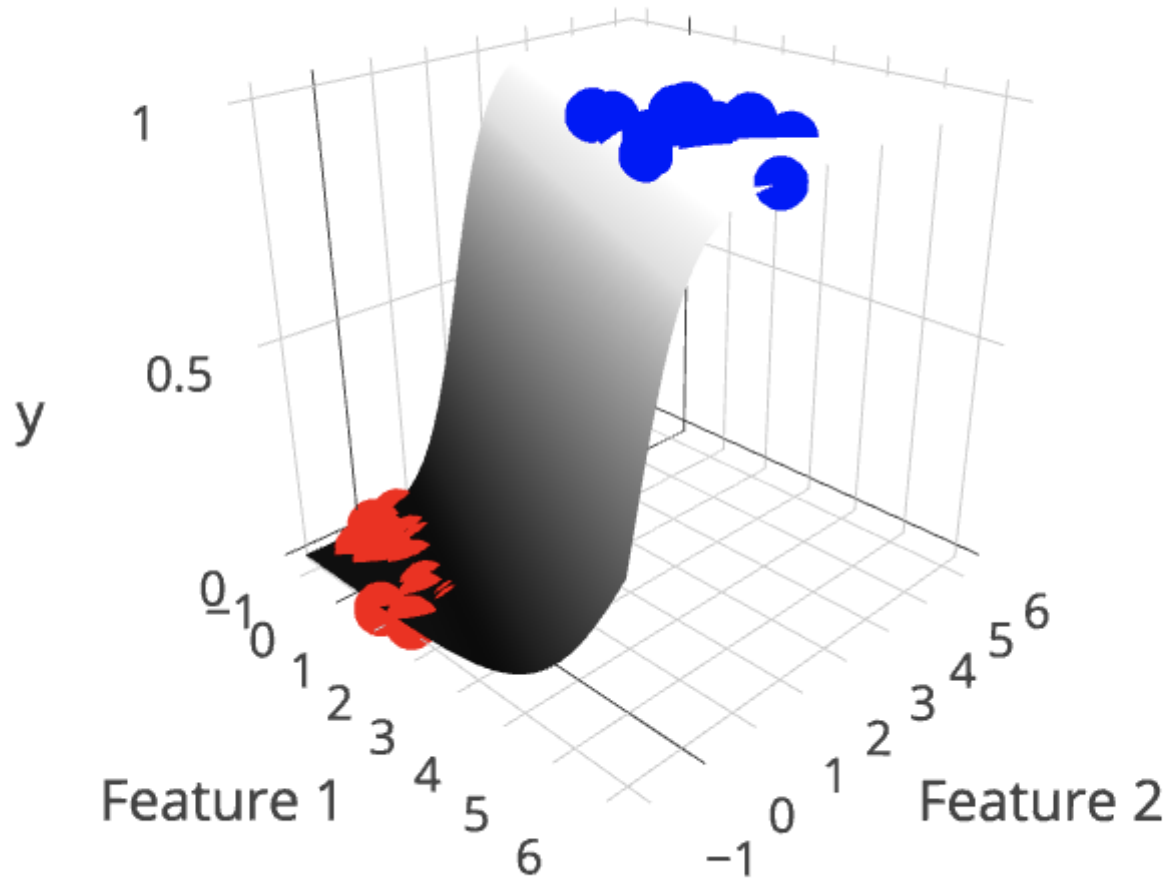
Logistic regression

- Transforms the classification problem into a regression problem
- Maps positive examples to value 1, others to value 0
- Fits a *logistic* (or *sigmoid*) function to predict whether a given sample belongs to class 1.
    - y value can be seen as a probability that the example is positive

$$z = f(x) = w_0 * x_0 + w_1 * x_1 + \ldots + w_p * x_p$$

$$\hat{y} = Pr[1|x_1, \ldots, x_k] = g(z) = \frac{1}{1 + e^{-z}}$$

On 2-dimensional data:



- The logistic function is chosen because it maps values (-Inf,Inf) to a probability [0,1]
- We add a new dimension for the dependent variable y and fit the logistic function g(z) so that it separates the samples as good as possible. The positive (blue) points are mapped to 1 and the negative (red) points to 0.
- After fitting, the logistic function provides the probability that a new point is positive. If we need a binary prediction, we can threshold at 0.5.
- There are different ways to find the optimal parameters w that fit the training data best

Loss function: cross-entropy

- Since logistic regression returns a probability, we want to use that in the loss function rather than choosing an an arbitrary threshold (e.g. positive in y > 0.5).
- We can measure the difference between the actual probabilities p_i and the predicted probabilities q_i is the cross-entropy $H(p, q)$:

$$H(p, q) = -\sum_i p_i log(q_i)$$

- Note: This is also called *maximum likelihood* estimation because instead of minimizing cross-entropy $H(p, q)$, you can maximize *log-likelihood* $-H(p, q)$
- In binary classification, $i = 0, 1$ and $p_1 = y$, $p_0 = 1 - y$, $q_1 = \hat{y}$, $q_0 = 1 - \hat{y}$
- This yields *binary cross-entropy*:

$$H(p, q) = -ylog(\hat{y}) - (1 - y)log(1 - \hat{y})$$

Cross-entropy loss

- Loss function: the average of all cross-entropies in the sample (of N data points):

$$\mathcal{L}_{log}(\mathbf{w}) = \sum_{n=1}^{N} H(p_n, q_n) = \sum_{n=1}^{N} \Big[- y_n log(\hat{y_n}) - (1 - y_n) log(1 - \hat{y_n}) \Big]$$

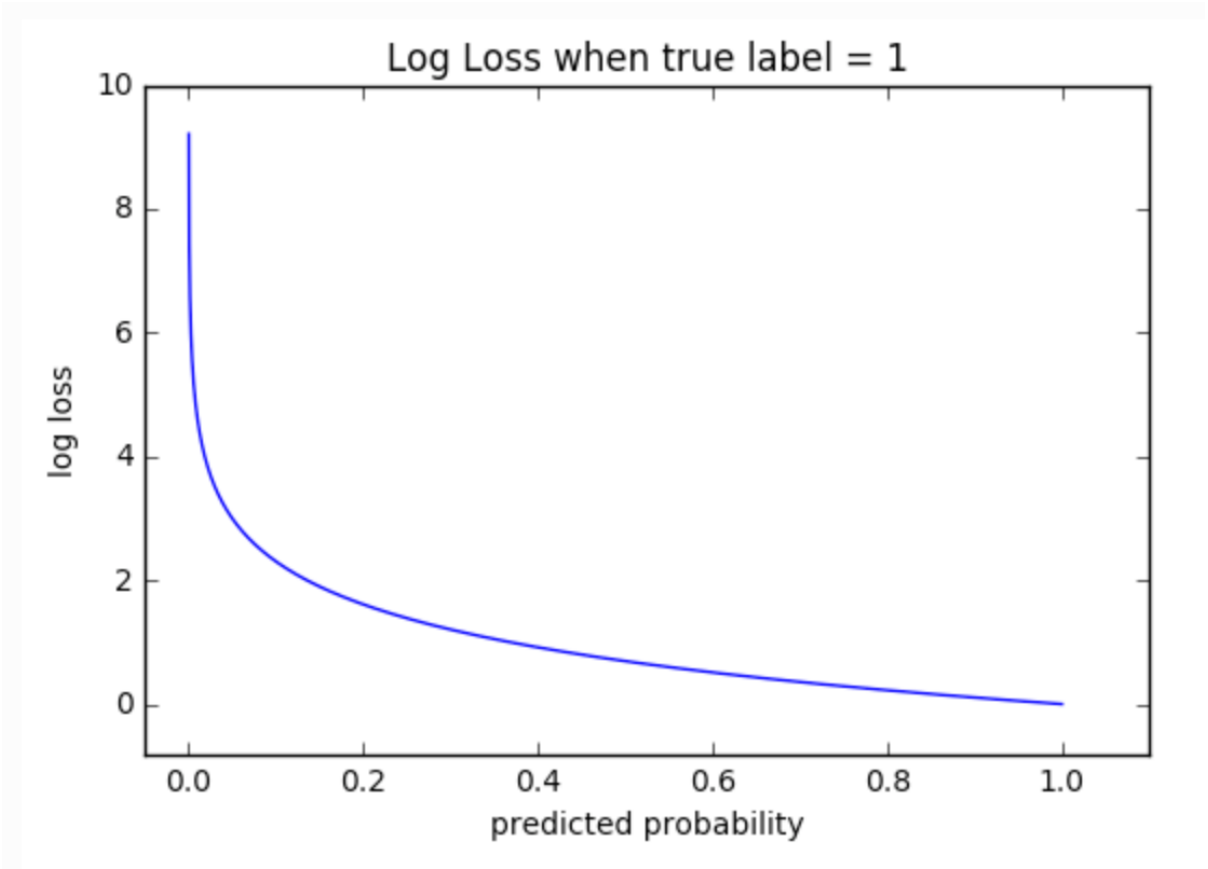
  with

$$\hat{y_n} = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}$$

- This is called *logistic loss*, *log loss* or *cross-entropy loss*
- We can (and should always) add a regularization term, either L1 or L2, e.g. for L2:

$$\mathcal{L}_{log}{}'(\mathbf{w}) = \mathcal{L}_{log}(\mathbf{w}) + \alpha \sum_{i} w_i^2$$

  - Note: sklearn uses C instead of α, and it is the inverse (smaller values, more regularization)

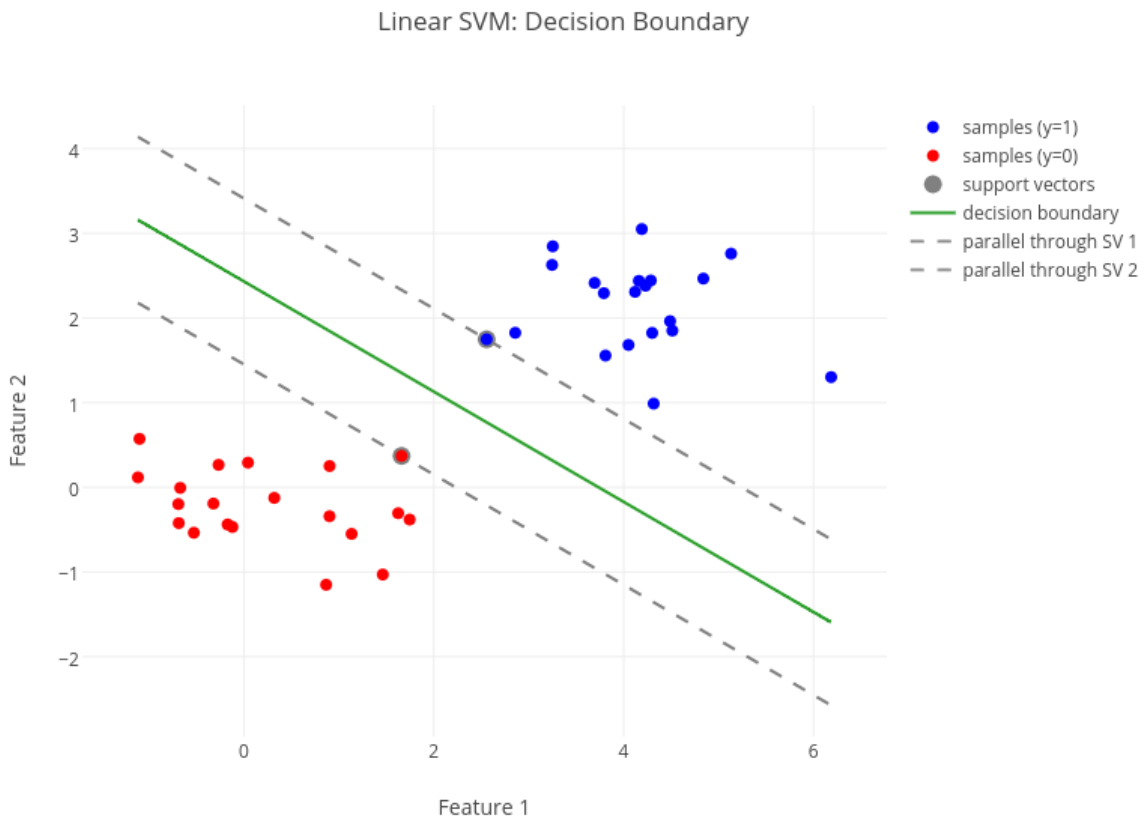

Log Loss when true label = 1
10
8
6
4
2
0
log loss
0.0
0.2
0.4
0.6
0.8
1.0
predicted probability

Optimization methods (solvers)

- There are different ways to optimize cross-entropy loss.
- Gradient descent
    - The logistic function is differentiable, so we can use (stochastic) gradient descent
    - Stochastic Average Gradient descent (SAG): only updates gradient in one direction at each step
- Coordinate descent (default, called `liblinear` in sklearn)
    - Faster, may converge more slowly, may more easily get stuck in local minima
- Newton-Rhapson (or Newton Conjugate Gradient):
    - Finds optima by computing second derivatives (more expensive)
    - Works well if solution space is (near) convex
    - Also known as *iterative re-weighted least squares*
- Quasi-Newton methods
    - Approximate, faster to compute
    - E.g. Limited-memory Broyden–Fletcher–Goldfarb–Shanno (`lbfgs`)

Linear Support Vector Machine (intuition)

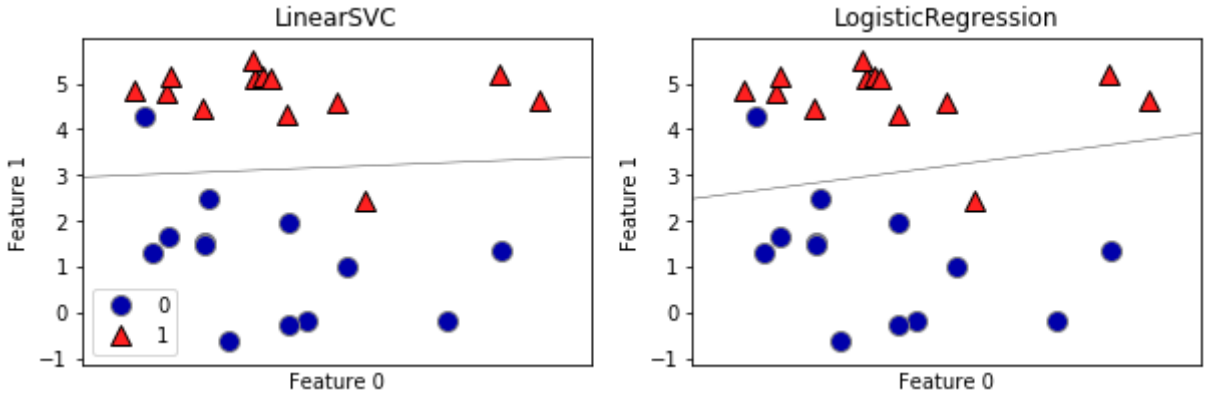
Find hyperplanes (dashed lines) maximizing the *margin* between the classes



Optimization and prediction

- Find a small number of data points to define the decision boundary (support vectors)
    - Each support vector has a weights (some are more important than others)
    - Hence, this is a non-parametric model
- Prediction is identical to (weighted) kNN:
    - Points closest to a red support vector are classified red, others blue
- The objective function penalizes every point predicted to be on the `wrong` side of its hyperplane
    - This is called *hinge loss*
- This results in a convex optimization problem solved using the *Langrange Multipliers* method
    - Can also be solved using gradient descent
    - We'll get back to this later

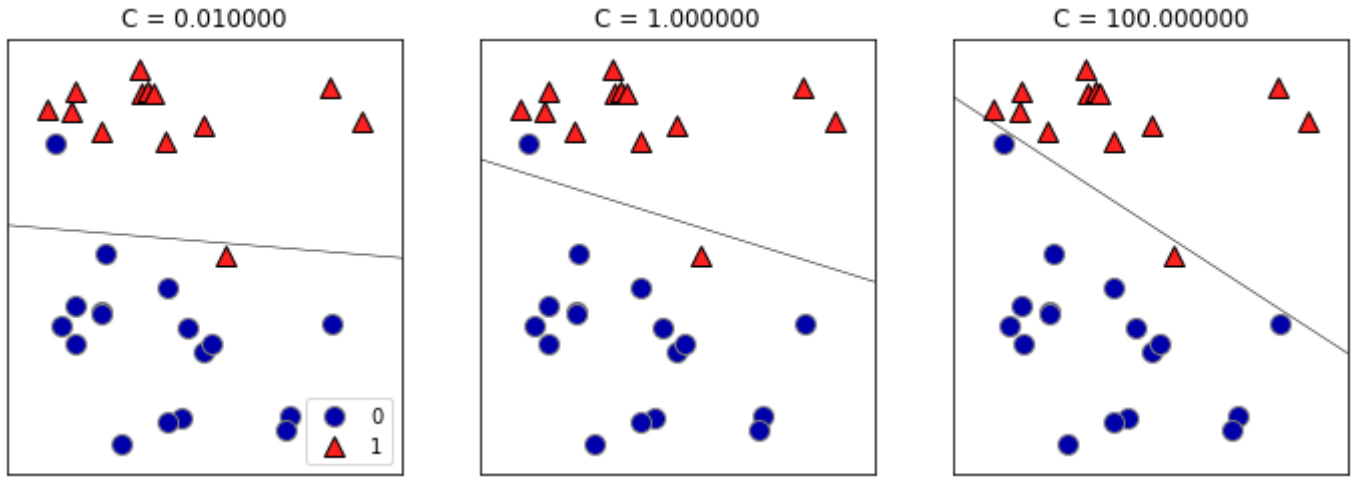
Comparison

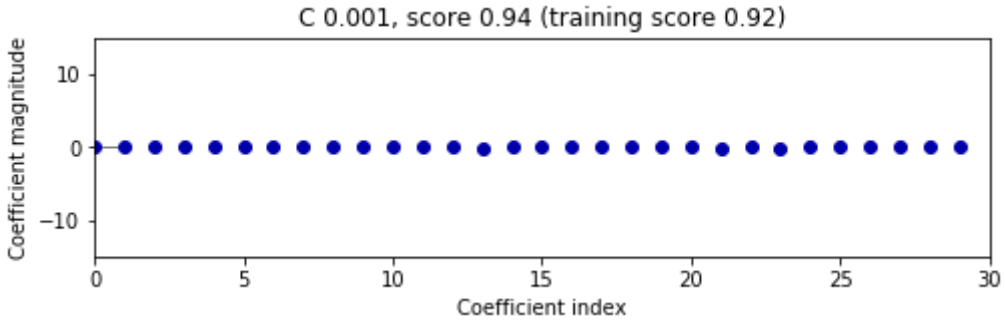

LinearSVC
LogisticRegression
Feature 1
Feature 0
0
1
5
4
3
2
1
0
−1

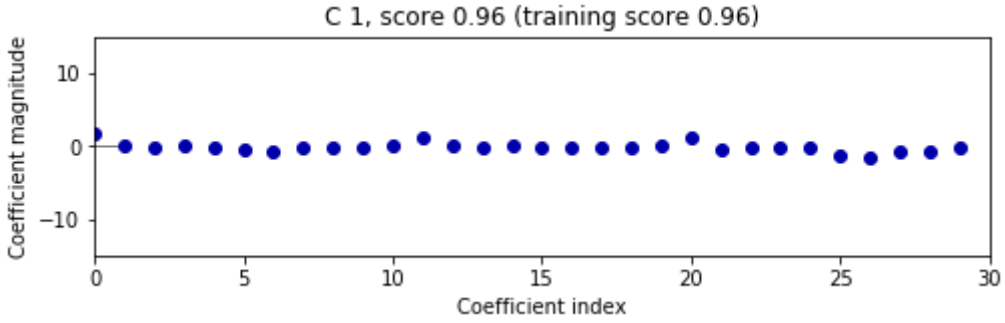
Both methods can be regularized:

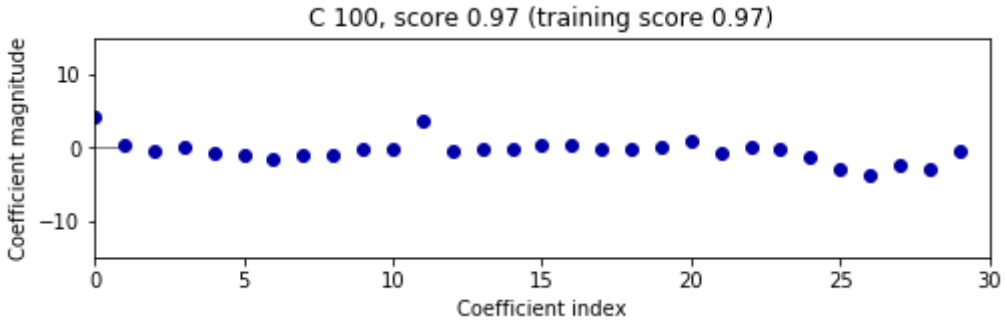
- L2 regularization by default, L1 also possible
- C parameter: inverse of strength of regularization
    - higher C: less regularization
    - penalty for misclassifying points while keeping w_i close to 0

SVM: High *C* values (less regularization): fewer misclassifications but smaller margins.




C 0.001, score 0.94 (training score 0.92)
Coefficient magnitude
10
0
−10
0
5
10
15
20
25
30
Coefficient index


C 1, score 0.96 (training score 0.96)
Coefficient magnitude
10
0
−10
0
5
10
15
20
25
30
Coefficient index

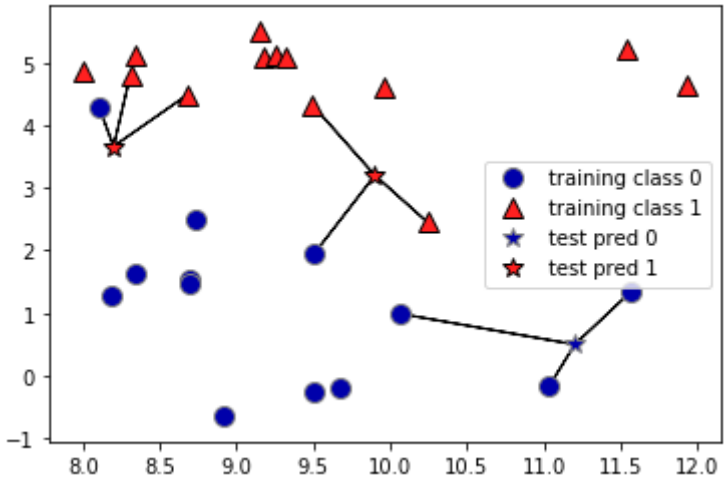

C 100, score 0.97 (training score 0.97)
Coefficient magnitude
10
0
−10
0
5
10
15
20
25
30
Coefficient index

k-Nearest Neighbor

- Building the model consists only of storing the training dataset.
- To make a prediction, the algorithm finds the *k* closest data points in the training dataset
    - Classification: predict the most frequent class of the k neighbors
    - Regression: predict the average of the values of the k neighbors
    - Both can be weighted by the distance to each neighbor
- Main hyper-parameters:
    - Number of neighbors (k). Acts as a regularizer.
    - Choice of distance function (e.g. Euclidean)
    - Weighting scheme (uniform, distance,...)
- Model:
    - Representation: Store training examples (e.g. in KD-tree)
    - Typical loss functions:
        - Classification: Accuracy (Zero-One Loss)
        - Regression: Root mean squared error
    - Optimization: None (no model parameters to tune)

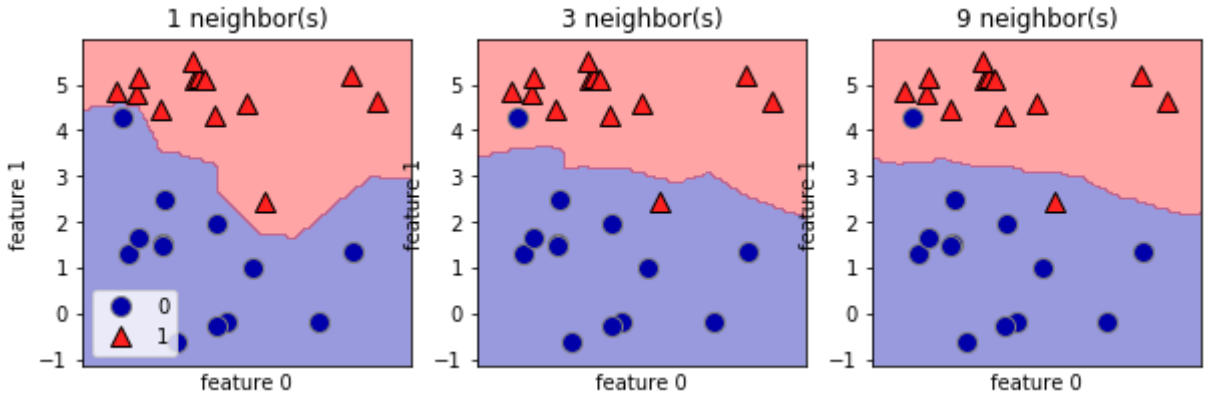
k-Nearest Neighbor Classification

k=1: look at nearest neighbor only: likely to overfit
k>1: do a vote and return the majority (or a confidence value for each class)



Analysis

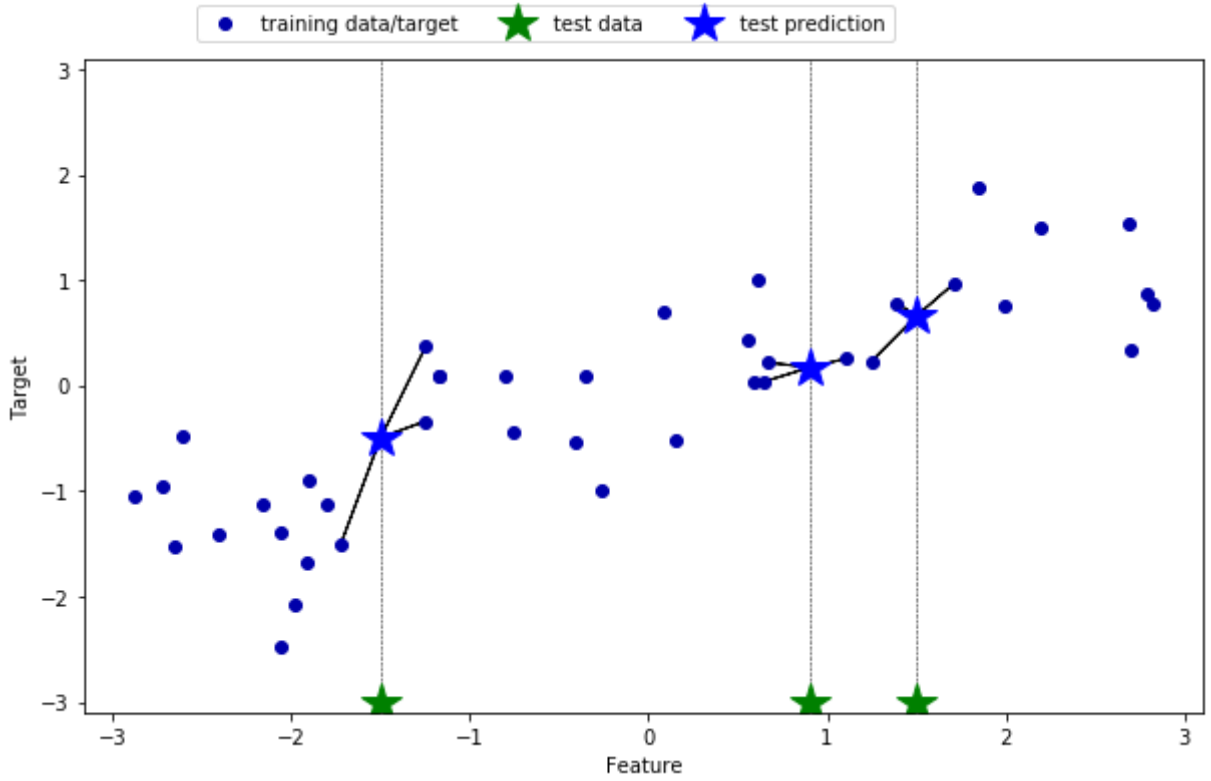
We can plot the prediction for each possible input to see the *decision boundary*



Using few neighbors corresponds to high model complexity (left), and using many neighbors corresponds to low model complexity and smoother decision boundary (right).

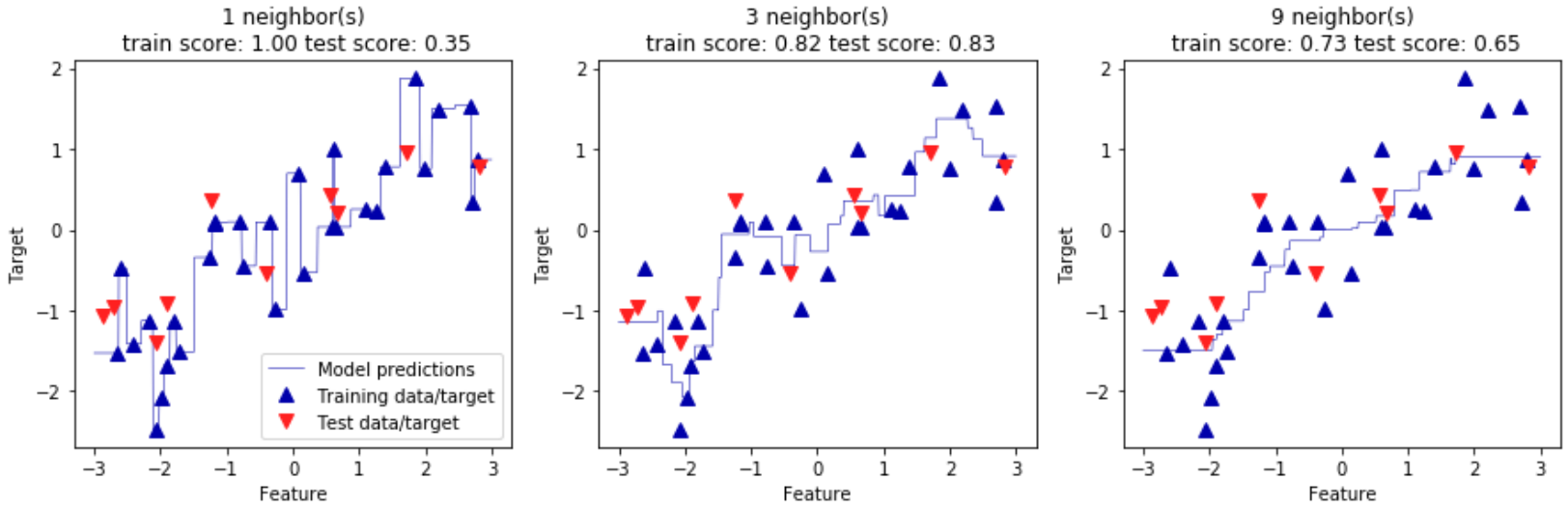
k-Neighbors Regression

k=1: return the target value of the nearest neighbor (overfits easily)
k>1: return the *mean* of the target values of the *k* nearest neighbors



Analysis

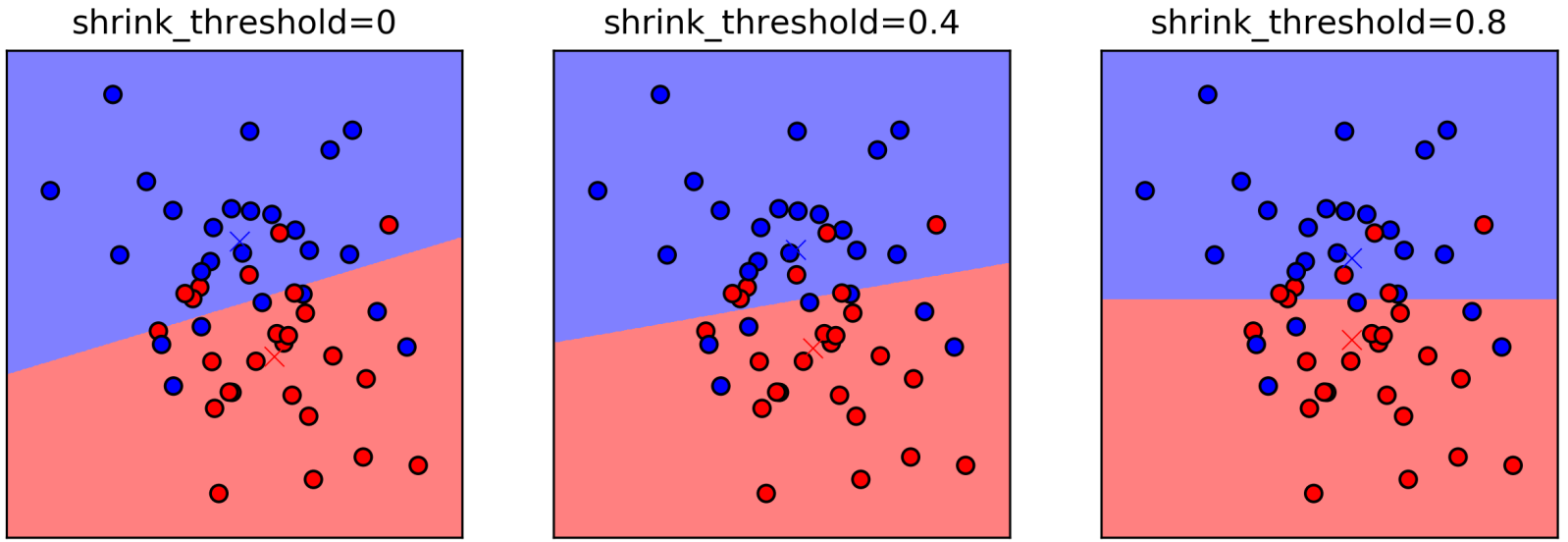
We can again output the predictions for each possible input, for different values of k.



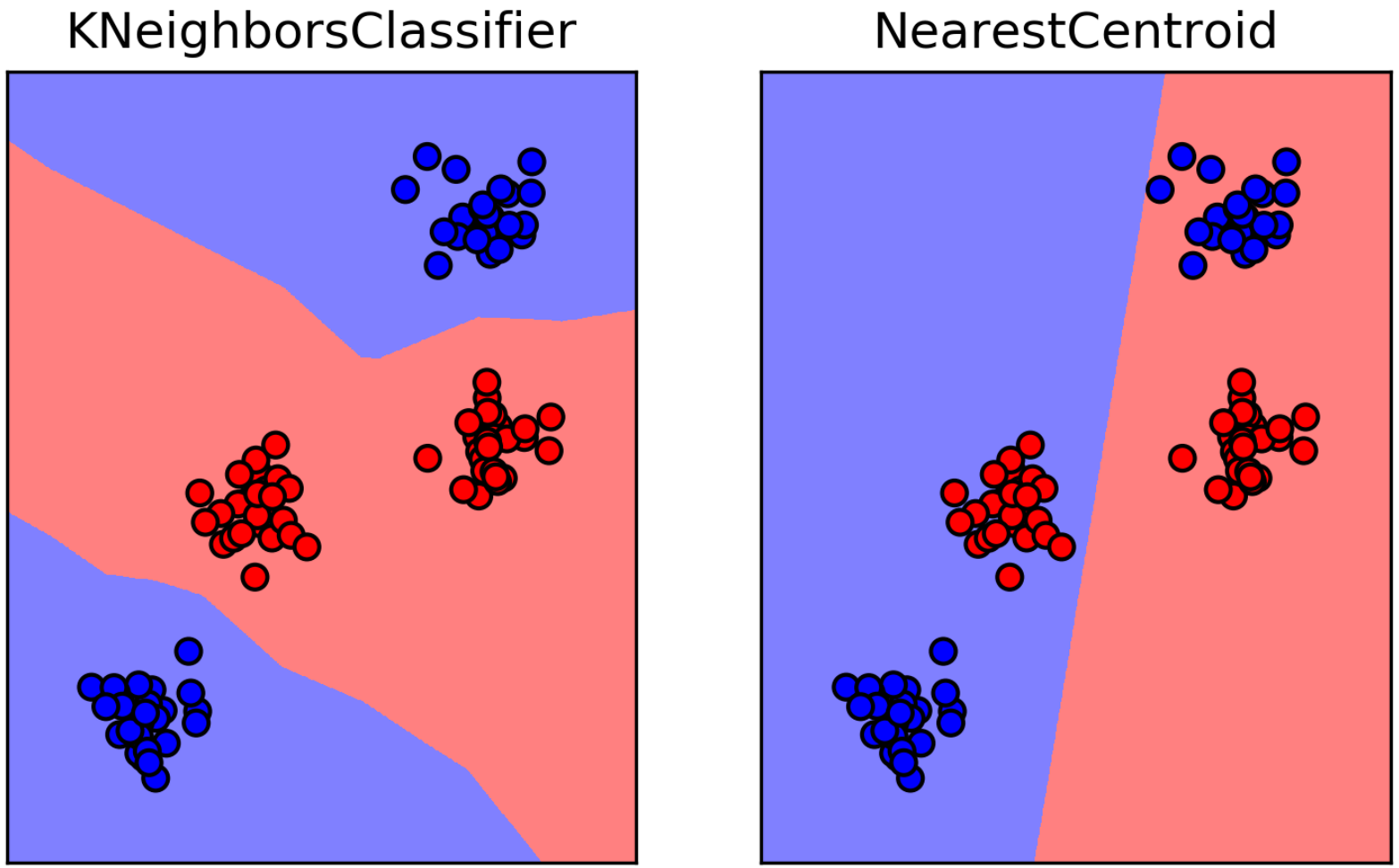
We see that again, a small k leads to an overly complex (overfitting) model, while a larger k yields a smoother fit.

Nearest Shrunken Centroid

- Nearest Centroid: Represents each class by the centroid of its members.
- Regularization is possible with the `shrink_threshold` parameter
  - Shrinks (scales) each feature value by within-class variance of that feature
  - Soft thresholding: if feature value falls below threshold, it is set to 0
  - Effectively removes (noisy) features



Note: Nearest Centroid suffers when the data is not 'convex'



Scalability

With n = nr examples and p = nr features

- Nearest shrunken threshold
    - Fit: $O(n * p)$
    - Memory: $O(nrclasses * p)$
    - Predict: $O(nrclasses * p)$
- Nearest neighbors (naive)
    - Fit: 0
    - Memory: $O(n * p)$
    - Predict: $O(n * p)$
- Nearest neighbors (with KD trees)
    - Fit: $O(p * nlogn)$
    - Memory: $O(n * p)$
    - Predict: $O(k * logn)$

kNN: Strengths, weaknesses and parameters

- Easy to understand, works well in many settings
- Training is very fast, predicting is slow for large datasets
- Bad at high-dimensional and sparse data (curse of dimensionality)